Key Comparison Optimal 2-3 Trees with Maximum Utilization

by
James R. Bitner*
and
Shou-Hsuan Huang*

TR-94
March 1979

Department of Computer Science
University of Texas
Austin, Texas 78712

* The work of these authors was supported in part by NSF Grant MCS 77-02705

Abstract: A class of 2-3 Trees is defined, called kcu-optimal trees which, out of all trees with optimal key-comparison cost, have maximum utilization. The average utilization of this class of trees is shown to be 64.7%. This is an improvement on the key-comparison optimal trees constructed by Rosenberg and Snyder (1) which have a utilization of 50%. Another measure, called the expansion is defined. Kcu-optimal trees have an average expansion of 56.7% which is near that of a random 2-3 tree. Algorithms for constructing a kcu-optimal tree from a sorted array of keys and maintaining them under insertions and deletions are given.

1. Introduction - Previous Work

Rosenberg and Snyder (1) have designed an algorithm that constructs from a given set of keys, a 2-3 tree which is optimal in terms of the expected number of key-comparison required to find a key. (We will call such trees kc-optimal, see below). However, these 2-3 trees are poor with respect to another important performance criterion, utilization. Asymptotically, their utilization is 50%, the worst possible. In this paper we give a characterization for 2-3 trees that have maximum utilization among all kc-optimal 2-3 trees with a given number of keys. The utilization for these 2-3 trees averages 64.7%, a substantial improvement over that for Rosenberg and Snyder's. We also define a new measure of space efficiency called the expansion, which measures how much extra space a 2-3 tree uses compared to a minimum space 2-3 tree. We show that the average expansion for our 2-3 trees is 56.7% which can be compared with the expansion for random 2-3 Trees which is known to be bounded (5) between 40% and 58%. Thus, these trees have relatively small expansion, yet are much better than random trees in terms of key-comparison cost. We also give algorithms for constructing such a 2-3 tree from a given set of keys and maintaining this property under deletions and insertions. The time required by the insertion algorithm is, unfortunately, linear. However the constant of proportionality appears to be quite small. A simulation was run that found an insertion into an "average" tree of height 10 required time .036n, where n is the number of nodes in the tree. "Time" here is measured in terms of the number of keys that must be moved to accomplish the insertion. Thus, this algorithm can be reasonably used if insertions are very infrequent.

We now define our terms:

Our definition of 2-3 trees is taken from Knuth (2). Note that under this

definition, the nodes having no sons (called leaves) do not contain information and do not count in determining the height of the tree. (a tree consisting of a single node with leaves as sons has height one.) We let K and N stand for respectively, the number of keys and nodes in a given 2-3 tree. A 1-node is defined as a node containing one key and a 2-node as a node containing two keys.

Another performance measure is the expected number of key comparisons. To define the measure, we let $k_i, i = 1,...K$ be the number of key comparisons required to find key i. We assume one comparison will be used at each 1-node on the path to key i. One comparison will also be used at a 2-node if we follow the left branch. If we follow the middle or right branch, two comparisons are required. The expected number of key comparisons is then $\frac{\sum_{i=1}^{K} k}{K}$.

Definition: A 2-3 tree is kc-optimal if its key comparison cost is minimal over all 2-3 trees having the same number of keys.

The following theorem characterizes kc-optimal 2-3 trees:

Theorem 1.1 (Rosenberg and Synder (1)): A 2-3 tree is kc-optimal iff no 2-node has a 2-node in either its middle or right subtrees.

Definition: The utilization of a 2-3 tree is $\frac{K}{2N}$ (the ratio of the number of keys to the number of possible keys). Note the utilization is bounded between 50% and 100%.

Definition: The expansion of a 2-3 Tree is $\frac{2N}{K} - 1$ and is bounded between 0% and 100%. Thus, a tree with an expansion of 50% takes 50% more space than is theoretically necessary. (We study the utilization because it is a more intuitive measure and the expansion because it allows a comparison with random 2-3 trees.)

Definition: A 2-3 tree is kcu-optimal iff it has maximum utilization among all kc-optimal trees having the same number of keys.

Section 2 will give a characterization theorem for kcu-optimal trees and section 3 will use this theorem to calculate the average utilization and expansion of kcu-optimal trees. Section 4 will give an efficient algorithm

for constructing a kcu-optimal tree from a sorted sequence of keys, and section 5 will give insertion and deletion algorithms that preserve the property of kcu-optimality and analyze the efficiency of the insertion algorithm.

2. A Characterization Theorem for kcu-optimality

We begin by ruling out a large class of kc-optimal 2-3 trees that can easily be shown not to have maximum utilization.

Theorem 2.1: In a kcu-optimal tree a 2-node cannot have a 1-node as its left son.

Proof: Suppose there is a 2-node having a 1-node as its left son. Apply the transform shown in Figure 2.1. This preserves the number of nodes in the tree and the fact that the tree is kc-optimal (since S_3, S_4, S_5 and S_6 must be completely binary). However, the number of 2-nodes is increased by one, increasing the utilization. Therefore the original tree did not have maximum utilization, a contradiction. □

This theorem says that a kcu-optimal tree with a 2-node as its root must have a special form:

Definition: A 2-3 tree is full iff it has the form shown in Figure 2.2.

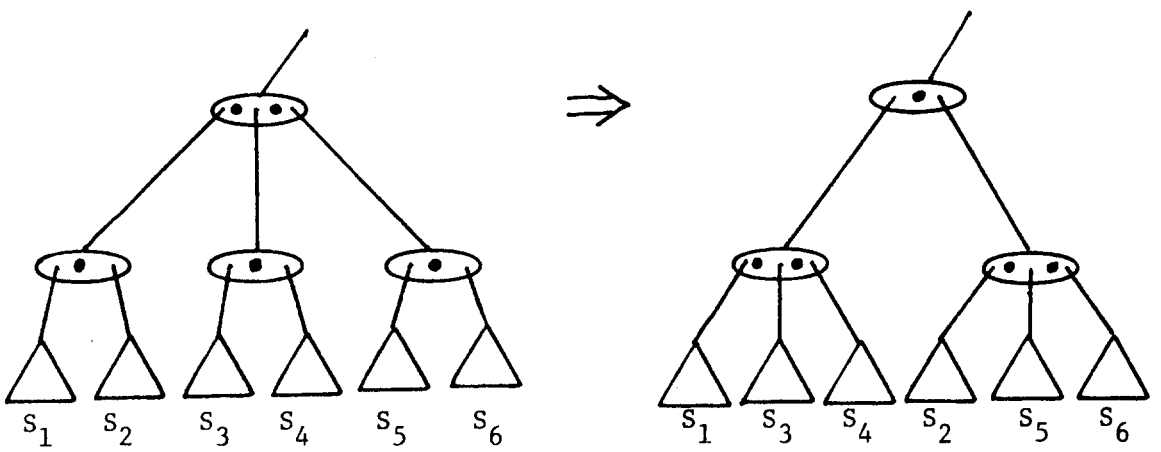

Figure 2.1

A transformation that increases the number of 2-nodes.

Lemma 2.1: If a kcu-optimal tree of height h has a 2-node as its root, it must be full. Also, it has $2^{h+1}-2$ keys.

Proof: (by induction on h).

If h=1, the tree consists of a single 2-node, and the lemma is obviously true. If h > 1, the middle and right subtrees of the root must be completely binary (since the tree is kc-optimal), and the left son of the root must be a 2-node by Theorem 2.1. Hence the left subtree is a kcu-optimal tree with a 2-node as the root, and, by induction, must be full. Therefore, the entire tree must be full. To calculate the number of nodes in the tree note that, by induction, the left subtree has $2^h - 2$ keys. Since the middle and right subtrees are completely binary, they have $2^{h-1}-1$ keys each. Adding the two keys in the root gives a total of $2^{h+1}-2$, proving the lemma. □

This lemma says that every 2-node in a kcu-optimal tree is the root of a full tree. A 2-node that has a 1-node for its father (and thus is not part of a larger full tree) is of special interest, motivating the following definition.

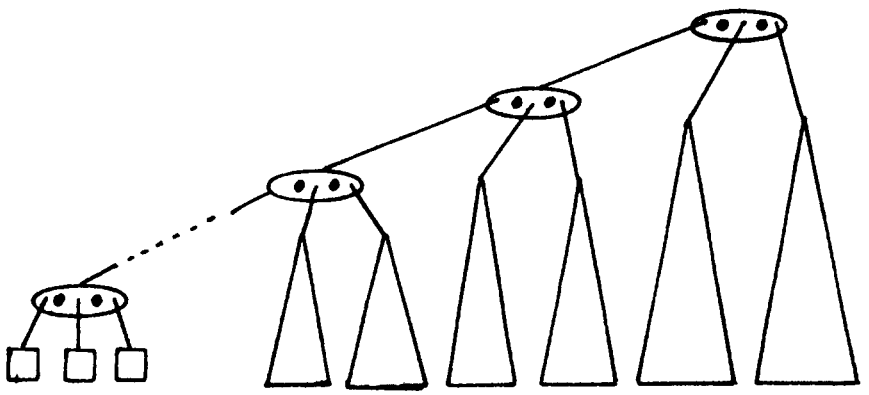

Figure 2.2

A full tree: the form for a kcu-optimal tree with a 2-node as its root. In this figure, the triangles are completely binary trees, and the squares are leaves.

Definition: A leader is a 2-node which has no 2-nodes as ancestors.

(In this definition, it will be convenient to consider leaves to behave as 2-nodes; a leaf will be a leader iff it has no 2-nodes for ancestors.)

Theorem 2.2: In a kcu-optimal tree, the levels of two leaders may differ by at most one.

Proof: By Theorem 2.1 we can assume that each 2-node has a leaf or a 2-node as its left son. Suppose there are leaders (ℓ_1 and ℓ_2) at height i and j with i-j > 1.

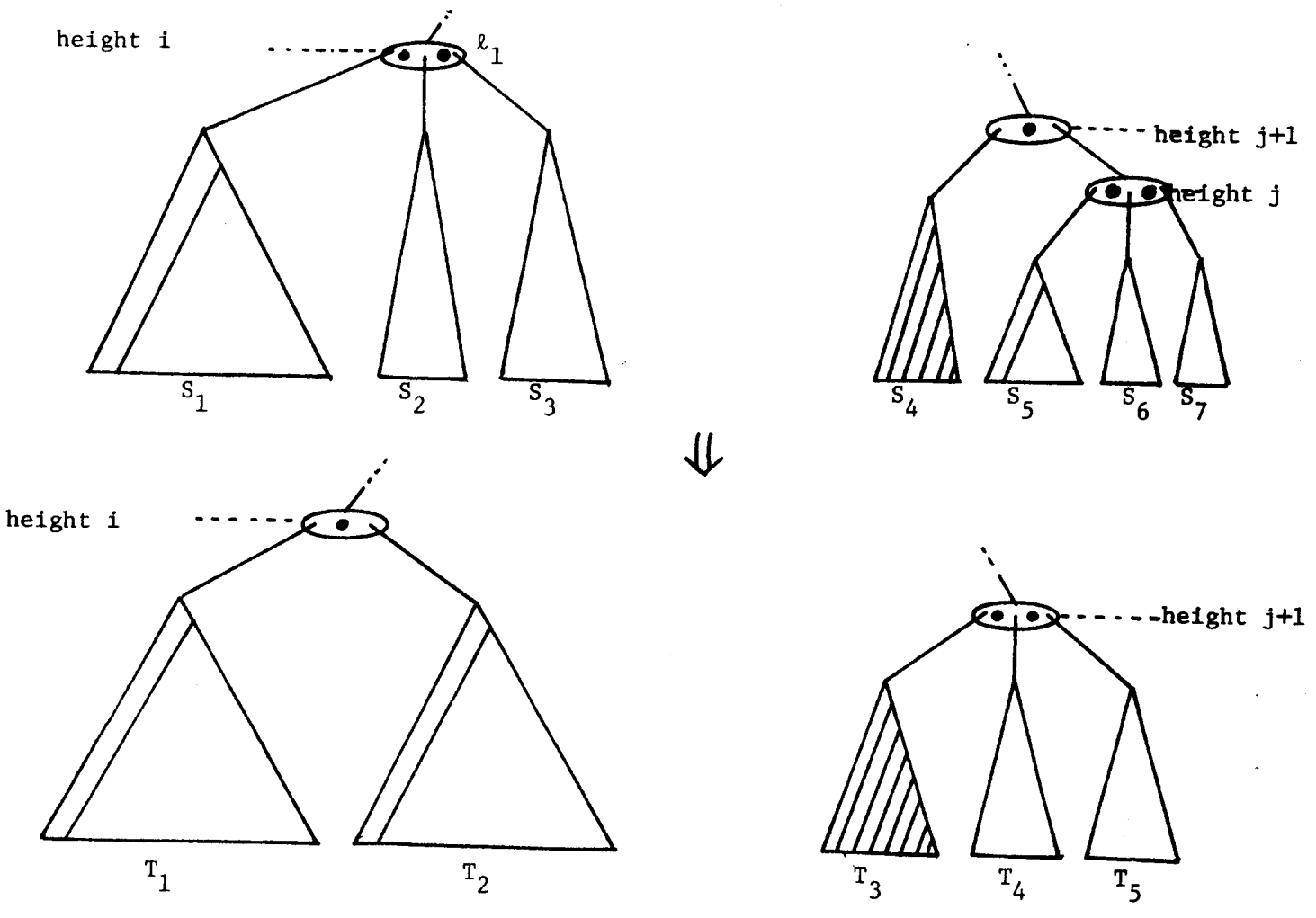

Figure 2.3

The two subtrees affected by the transformation are shown. Triangles represent completely binary trees. Triangles with a stripe (such as S_1) are full trees. Shaded trees may have arbitrary form.

Note that ℓ_1 cannot be an ancestor of ℓ_2 because ℓ_1's subtree would then be full, and ℓ_2 would not be a leader. Also note that ℓ_2's father must be a 1-node and that ℓ_1's and ℓ_2's subtrees are full. We assume without loss of generality that ℓ_2 is the right son of his father.

We now describe the transformation shown in Figure 2.3, which will preserve the number of keys and the kc-optimality but will increase the utilization. Applying this transformation to the original tree will then give a contradiction. Subtrees S_1 and S_4 are not affected and become trees T_1 and T_3 respectively. The keys in S_2 and S_3 are merged to form T_2, then one key is deleted from ℓ_1. There are $2^{j+1}-2$ keys in ℓ_2 and its subtrees. These are redistributed to form T_4 and T_5. Finally, a key is added to ℓ_2's father. (Note that the transformation will work even if ℓ_2 is a leaf; then S_5, S_6 and S_7 are empty.) In total, the number of keys remained the same, and that the tree is still kc-optimal. However, the number of 2-nodes has increased; originally there were i+j 2-nodes in the affected subtrees and now there are 2i-1. Hence the utilization of a kcu-optimal tree has been increased, a contradiction. □

Theorem 2.2 establishes a necessary condition for kcu-optimality. From now on, we only consider trees which satisfy the conditions of Theorems 2.1 and 2.2. This allows us to make the following definition:

Definition: A leader profile of a 2-3 tree is an ordered triple (h,ℓ,x) where h is the height of the tree, ℓ is the level of the leader with lowest level, and x is the number of leaders having lowest level. In addition, if the tree consists solely of 1-nodes (and hence all leaders are leaves) it will be more convenient to define the leader profile to be $(h,h-1,0)$ instead of $(h,h,2^h)$.

Because of Theorems 2.1 and 2.2, any kcu-optimal tree can be uniquely described by a leader profile (up to permutations of full subtrees). (Note that knowning x also determines the number of leaders at the higher level, and hence determines the position of all the leaders.)

These theorems give necessary conditions for kcu-optimality. In the next section, we will show that only one leader profile is possible for a given K. Hence these conditions are also sufficient. Thus, we have:

Theorem 2.3 A 2-3 Tree is kcu-optimal iff

(1) It is kc-optimal

(2) Every 2-node has a leaf or a 2-node as its left son.

(3) The heights of any two leaders differ by at most one.

Proof: The conditions are necessary by Theorems 2.1 and 2.2 and sufficient by the uniqueness of the leader profile in Theorem 3.2. □

3. Calculating the Utilization and the Expansion

Our strategy in this section is to first find a mapping from leader profiles into K and N (obviously both these mappings are one-to-one). We then find the inverse of the mapping into K and discover that this mapping is also one-to-one. Therefore knowing K uniquely determines the leader profile and uniquely determines the tree up to permutations of the full subtrees. Therefore, any tree satisfying Theorem 2.2 has a leader profile and must be kcu-optimal. Hence Theorem 2.2 provides a characterization of kcu-optimal trees.

Theorem 3.1: Let $keys(h,\ell,x)$, $nodes(h,\ell,x)$ and $twos(h,\ell,x)$ be respectively the number of keys, node and 2-nodes in a 2-3 tree with reduced leader profile (h,ℓ,x). Then

$$keys(h,\ell,x) = 2^{h+1} - 2^{\ell+1} - 1 + x$$

$$nodes(h,\ell,x) = 2^{h+1} - (h-\ell)2^{\ell+1} + (h-\ell-1)x - 1$$

$$twos(h,\ell,x) = (h-\ell-1)2^{\ell+1} - (h-\ell-2)x$$

Proof: We break our level-by-level analysis of the tree into four parts:

Part 1 (levels $0,\ldots,\ell-1$): For these levels the tree is completely binary. Therefore level i ($0 \leq i \leq \ell - 1$) will have 2^i nodes, 2^i keys and zero 2-nodes.

Part 2 (level ℓ): Here we have the x upper leaders. The remaining nodes at this level are 1-nodes. Since there are 2^ℓ nodes in total at this level (the tree is completely binary above this level), there are $2^\ell - x$ 1-nodes. Therefore, at this level there are 2^ℓ nodes, $1 \cdot (2^\ell - x) + 2 \cdot x = 2^\ell + x$ keys and x two nodes.

Part 3 (level $\ell+1$): Every 1-node at level ℓ will have two 2-nodes (lower leaders) as its sons, and every 2-node at level ℓ will have one 2-node and two 1-nodes as its sons. Thus, the inventory is $2 \cdot (2^\ell - x) + 3 \cdot x = 2^{\ell+1} + x$ nodes, $4 \cdot (2^\ell - x) + 4x = 2^{\ell+2}$ keys and $2 \cdot (2^\ell - x) + 1 \cdot x = 2^{\ell+1} - x$ two nodes.

Part 4: (levels $\ell+2,\ldots,h-1$): First, note that every 2-node at level i ($i \geq \ell+1$) has one 2-node as its son and every 1-node has zero. Therefore the number of 2-nodes at each level will remain constant at $2^{\ell+1} - x$ after level $\ell+1$.

Let n_i be the number of nodes at level i. Since each 1-node at level i-1 (for $i \geq \ell+1$) creates two nodes at level i and each 2-node creates three, we have the recurrences

$$n_i = 2\cdot(n_{i-1}-(2^{\ell+1}-x)) + 3\cdot(2^{\ell+1}-x) \text{ for } i \geq \ell+2$$

$$n_{\ell+1} = 2^{\ell+1}+x$$

whose solution is $2^{\ell+i+1} - 2^{\ell+1}+x$, the number of nodes at level i. The number of keys equals $1\cdot(2^{\ell+i+} - 2^{\ell+1}+x - (2^{\ell+1}-x))+2(2^{\ell+1}-x) = 2^{\ell+i+1}$

These results are summarized in Table 3.1. Summing over each column gives the formulas given in the theorem for $\ell < h-1$. For $\ell=h-1$, the last two rows will be equal to zero. It is easy to verify that the theorem is also true in this case. □

level	number of nodes	number of 2-nodes	number of keys
$0 \leq i \leq \ell-1$	2^i	0	2^i
ℓ	2^ℓ	x	$2^\ell+x$
$\ell+1$	$2^{\ell+1}+x$	$2^{\ell+1}-x$	$2^{\ell+2}$
$\ell+2 \leq i \leq h-1$	$2^{i+1}-2^{\ell+1}+x$	$2^{\ell+1}-x$	2^{i+1}

Table 3.1

A level-by-level analysis of the structure of a 2-3 tree with reduced leader profile (h,ℓ,x) when $\ell < h-1$. If $\ell = h-1$, the last two rows should be zero.

Definition: Let $\overline{\delta}_{ij} = \begin{cases} 1 \text{ if } i \neq j \\ 0 \text{ if } i = j \end{cases}$ i.e. the "reverse" of Kronecker's delta.

Lemma 3.1 Given any $h \geq 0$ and $0 \leq \ell \leq h-1$,

$$2^{h+1}-2^{\ell+1}-1+\overline{\delta}_{\ell,h-1} \leq 2^{h+1}-2^{\ell+1}-1+x \leq 2^{h+1}-2^{\ell}-1$$

where $\overline{\delta}_{\ell,h-1} \leq x \leq 2^\ell$

Proof: Obvious from the given inequalities □

Lemma 3.2: Given any $h \geq 0$

$$2^h-1 \leq 2^{h+1}-2^{\ell}-1+x \leq 2^{h+1}-2$$

where $0 \leq \ell \leq h-1$ and $\overline{\delta}_{\ell,h-1} \leq x \leq 2^\ell$.

Proof: To verify the lower bound:

$2^{h+1}-2^{\ell+1}-1+x \geq 2^{h+1}-2^{h}-1 = 2^{h}-1$

by Lemma 3.1. To prove the upper bound, consider any $\ell \leq h-1$. For this ℓ,

$2^{h+1}-2^{\ell+1}-1+x \leq 2^{h+1}-2^{\ell+1}-1+2^{\ell} = 2^{h+1}-2^{\ell}-1$

Since $\ell \geq 0$, the largest this can be is $2^{h+1}-2$, proving the upper bound. □

We now give a theorem that specifies the mapping from K into the leader profile. Since this mapping is one-to-one, the characterization theorem given in Section 2 will be established. Corollary 3.1 provides the intuition behind the theorem and helps to explain the derivation of the formulas in this theorem. h is easily determined. To calculate ℓ, we must find which of the h subsequences K is in and to calculate x we find the position in that subsequence.

Theorem 3.2: For a given K, let

(1) $h = \lfloor \log(K+1) \rfloor$

(2) $\ell = \begin{cases} \lfloor \log(2^{h+1}-1-K) \rfloor & \text{if } K \neq 2^{h}-1 \\ h-1 & \text{if } K = 2^{h}-1 \end{cases}$

(3) $x = 2^{\ell+1}-(2^{h+1}-1-K)$

Then (h,ℓ,x) is the only leader profile with K keys.

Proof: (Note that $K=2^{h+1}-2^{\ell+1}-1+x$) To prove equation (1), we use Lemma 3.2: $2^{h}-1 \leq K < 2^{h+1}-1$. Hence $h \leq \log(K+1) < h+1$, and $h = \lfloor \log(K+1) \rfloor$.

To prove equation (2) we require 2 cases. Note that $2^{h+1}-1-K = 2^{\ell+1}-x$.

Case 1 ($2^{h+1}-1-K < 2^{h-1}$; this implies $\ell < h-1$):

We have $2^{h-1} > 2^{\ell+1}-x > 2^{\ell+1}-2^{\ell}=2^{\ell}$. Hence $\ell<h-1$.

Therefore

$$2^{\ell}=2^{\ell+1}-2^{\ell} \leq 2^{\ell+1}-x \leq 2^{\ell+1}-\overline{\delta}_{\ell,h-1}=2^{\ell+1}-1$$

because $\overline{\delta}_{\ell,h-1} \leq x \leq 2^{\ell}$ and $\ell \neq h-1$. Therefore

$$2^{\ell} \leq 2^{\ell+1}-x=2^{h+1}-1-K < 2^{\ell+1}-1 \text{ and } \ell = \lfloor \log(2^{h+1}-1-K) \rfloor$$

Case 2 ($2^{h+1}-1-K \geq 2^{h-1}$; this implies $\ell=h-1$): We have $2^{h-1} \leq 2^{\ell+1}-x \leq 2^{\ell+1}-2^{\ell}=2^{\ell}$. Therefore $h-1 \leq \ell$. Since ℓ is restricted to be at

most h-1, ℓ=h-1. If $K=2^h-1$ equation (2) is true. If $K < 2^h-1$, we must verify the r.h.s. of equation (2) equals h-1. We have:

$2^{\ell}=2^{\ell+1}-2^{\ell} \leq 2^{\ell+1}-x=2^{h+1}-1-K < 2^h$

Therefore $\ell \leq \log(2^{h+1}-1-K) < h$, and $\lfloor \log(2^{h+1}-1-K) \rfloor = h-1=\ell$, and equation (2) is true.

Equation (3) follows directly from $K=2^{h+1}-2^{\ell+1}-1+x$. □

Corollary 3.1: For a given h the sequence of leader profiles for $K=2^h-1,\ldots,2^{h+1}-2$ begins with (h,h,0) and then consists of h subsequence The ℓ^{th} subsequence from the end consists of $(h,\ell,1),(h,\ell,2),\ldots,(h,\ell,2^{\ell})$. (See Figure 3.1)

K	Leader Profile
7	(3,2,0)
8	(3,2,1)
9	(3,2,2)
10	(3,2,3)
11	(3,2,4)
12	(3,1,1)
13	(3,1,2)
14	(3,0,1)

Figure 3.1

The sequence of leader profiles for h=3.

Lemma 3.3: If f(x) is monotone over [a,b] then

$$\min(f(a),f(b)) + \int_a^b f(x)dx \leq \sum_{i=a}^{b} f(i) \leq \max(f(a),f(b)) + \int_a^b f(x)dx$$

Hence

$$\sum_{i=a}^{b} f(i) = \left(\int_a^b f(x)dx\right) + \epsilon, \text{ where } \min(f(a),f(b)) \leq \epsilon \leq \max(f(a),f(b))$$

Lemma 3.4 Let t be an integer ≥ 1, let $r \neq 0$, and suppose $-\frac{s}{r} \notin [1,t]$ then

$$\sum_{i=1}^{t} \frac{pi+q}{ri+s} = \frac{1}{r}\left[p(t-1) + \left(q - \frac{ps}{r}\right) \ln\left(\frac{rt+s}{r+s}\right)\right] + \epsilon$$

where $\min\left(\frac{p+q}{r+s}, \frac{pt+q}{rt+s}\right) \leq \epsilon \leq \max\left(\frac{p+q}{r+s}, \frac{pt+q}{rt+s}\right)$.

Proof:
Let $f(x) = \frac{px+q}{rx+s}$. Then $f'(x) = \frac{ps-qr}{(rx+s)^2}$ and the sign of $f'(x)$ depends only on $ps-qr$, not x. Hence $f(x)$ is monotone over any region not containing $-\frac{s}{r}$. The lemma then follows directly from Lemma 3.3 and the calculation of

$$\int_{1}^{t} \frac{px+q}{rx+s}\, dx \quad \text{using the substitution } y = rx + s. \qquad \square$$

From Theorem 3.2 we can calculate the utilization of a kcu-optimal tree for any K. However, we would like to calculate an "average" utilization to indicate how well these trees perform on the average. Unfortunately, the utilization does not approach a limit as $K \to \infty$. For the leader profile $(h,h-1,0)$, the tree is completely binary, and the utilization is 50%. For $(h,h-1,2^{h-1})$, the tree is binary except for level h-1 which consists solely of 2-nodes. This tree, then, has a utilization of approximately 75%, and the utilization varies cyclically between these two extremes. We choose to average the utilization over all trees of a given height. Thus, the average is over the range from one minimum to the next.

Theorem 3.3: Let nodes (K) be the number of nodes in a kcu-optimal tree of K keys. Then for large h, the utilization $\left(= \frac{K}{2 \cdot \text{nodes}(K)}\right)$ averaged over $K=2^h-1,\ldots,2^{h+1}-2$ is

$$\frac{5}{16} + \sum_{m=1}^{\infty} \frac{1}{2m}\left[1+\left(\frac{(m-1)2^{m+2}+2}{m}\right) \ln\left(\frac{2^{m+2}-(m+2)}{2^{m+2}-(2m-1)}\right)\right] \cdot 2^{-(m+1)}$$

which can be numerically calculated as 64.7%.

Proof: We need to calculate $\left(\sum_{K=2^h-1}^{2^{h+1}-2} \frac{K}{2\cdot nodes(K)}\right)/2^h$

By Corollary 3.1 this equals $\left(\frac{keys(h,h-1,0)}{2\cdot nodes(h,h-1,0)} + \sum_{\ell=0}^{h-1} \sum_{x=1}^{2^\ell} \frac{1}{2} \cdot \frac{keys(h,\ell,x)}{nodes(h,\ell,x)}\right)/2^h$

We ignore the first term in the sum; when divided by 2^h, it vanishes as $h \to \infty$. Then, substituting m for $h-\ell-1$ and reversing the order of the first summation gives

$$\left(\sum_{m=0}^{h-1} \sum_{x=1}^{2^{h-m-1}} \frac{1}{2} \cdot \frac{keys(h,h-m-1,x)}{nodes(h,h-m-1,x)}\right)/2^h.$$

Substituting the values from Theorem 3.1 for keys(h,h-m-1,x) and nodes(h,h-m-1,x) gives

$$\left(\sum_{m=0}^{h-1} \sum_{x=1}^{2^{h-m-1}} \frac{1}{2} \cdot \frac{2^{h+1}-2^{h-m}-1+x}{2^{h+1}-(m+1)2^{h-m}+mx-1}\right)/2^h. \qquad (1)$$

Define $\alpha_{h,m}$ to be the value of the inner sum for a given h and m. We want to take the limit of $\alpha_{h,m}$ as $h \to \infty$, but the number of terms increases to infinity as $h \to \infty$. Hence, we will consider $\alpha_{h,m}$ divided by the number of terms. Normalizing the sum in this manner makes the limit exist. Define $\bar{\alpha}_{h,m} = \alpha_{h,m}/2^{h-m-1}$ and $\alpha^*_m = \lim_{h\to\infty} \bar{\alpha}_{h,m}$. (Note that $\bar{\alpha}_{h,m}$ is the average of a number of utilizations and hence $\frac{1}{2} \le \bar{\alpha}_{h,m} \le 1$. We use this fact in Lemma 3.5.)

(1) is then equal to $\sum_{m=0}^{h-1} \bar{\alpha}_{h,m} \cdot 2^{-(m+1)}$. To calculate the limit as $h \to \infty$, we use the identity

$$\lim_{h\to\infty} \sum_{m=0}^{h-1} \bar{\alpha}_{h,m} \cdot 2^{-(m+1)} = \sum_{m=0}^{\infty} \alpha^*_m 2^{-(m+1)}. \qquad (2)$$

This equality is not obvious (suppose the summand were $\delta_{h,m} \equiv 1$ if h=m and 0 otherwise instead of $\bar{\alpha}_{h,m} \cdot 2^{-(m+1)}$) and is proved in Lemma 3.5. We now calculate α^*_m for all m. If m=0, $\alpha_{h,m}$ simplifies to:

$$\frac{1}{2} \sum_{x=1}^{2^{h-1}} \frac{2^h-1+x}{2^h-1} = \frac{5\cdot 2^{2h-2} - 3\cdot 2^{h-1}}{2^{h+2}-4}$$

Dividing by 2^{h-1} and taking the limit gives $\alpha_0^* = \frac{5}{8}$.

For $m > 0$, we use Lemma 3.4 to give

$$\alpha_{h,m} = \frac{1}{2m}[2^{h-m-1} - 1 + (2^{h+1} - 2^{h-m} + 1 - \frac{2^{h+1} - (m+1)2^{h-m} - 1}{m}) \times$$

$$\ln(\frac{2^{h+1} - (m+1)2^{h-m} - 1 + m2^{h-m-1}}{2^{h+1} - (m+1)2^{h-m} - 1 + 2^{h-m-1}})] + \epsilon$$

$$= \frac{1}{2m}[2^{h-m-1} - 1 + (\frac{(m-1)2^{h+1} + 2^{h-m} + (m+1)}{m}) \ln(\frac{2^{h+1} - (m+2)2^{h-m-1} - 1}{2^{h+1} - (2m-1)2^{h-m-1} - 1})] + \epsilon$$

where $1/2 \leq \epsilon \leq 1$. (the minimum and maximum utilizations)

Dividing by 2^{h-m-1} and taking the limit as $h \to \infty$ gives

$$\alpha_m^* = \frac{1}{2m}[1 + (\frac{(m-1)2^{m+2} + 2}{m}) \ln(\frac{2^{m+2} - (m+2)}{2^{m+2} - (2m-1)})] \text{ for } m > 0.$$

Then substituting into (2) proves the theorem. □

Lemma 3.5: Let $\frac{1}{2} \leq \bar{\alpha}_{h,m} \leq 1$ for all $h,m \geq 0$ and let $\alpha_m^* = \lim_{h \to \infty} \bar{\alpha}_{h,m}$. Then

$$\lim_{h \to \infty} \sum_{m=0}^{h-1} \bar{\alpha}_{h,m} \cdot 2^{-(m+1)} = \sum_{m=0}^{\infty} \alpha_m^* \cdot 2^{-(m+1)}$$

Proof: We know that for any $\epsilon > 0$, an H_0 can be found such that

$$|\sum_{m=0}^{h-1} \bar{\alpha}_{h,m} \cdot 2^{-(m+1)} - \sum_{m=0}^{\infty} \alpha_m^* \cdot 2^{-(m+1)}| < \epsilon \text{ for all } h \geq H_0 \qquad (1)$$

For any given ϵ, let $M_0 = \frac{-\log \epsilon}{2}$ Note that

$$\sum_{m=M_0+1}^{h-1} \bar{\alpha}_{h,m} \cdot 2^{-(m+1)} < \sum_{m=M_0+1}^{\infty} 2^{-(m+1)} < \frac{\epsilon}{4}$$

Similarly, $\sum_{m=M_0+1}^{\infty} \alpha_m^* 2^{-(m+1)} < \frac{\epsilon}{4}$. Now for this M_0 choose H_0 such that

$$|\alpha_m^* 2^{-(m+1)} - \bar{\alpha}_{h,m} 2^{-(m+1)}| < \frac{\epsilon}{2M_0} \text{ for all } h \geq H_0,\ m \leq M_0.$$

Such an H_0 must exist; for every m, there exists a suitable value for H_0, and we simply pick the maximum of these values. We now have for any $h \geq H_0$,

$$\left| \sum_{m=0}^{h-1} \bar{\alpha}_{h,m} 2^{-(m+1)} - \sum_{m=0}^{\infty} \alpha^*_m 2^{-(m+1)} \right|$$

$$< \sum_{m=0}^{M_0} \left| \bar{\alpha}_{h,m} 2^{-(m+1)} - \alpha^*_m 2^{-(m+1)} \right| + \sum_{m=M_0+1}^{\infty} \bar{\alpha}_{h,m} 2^{-(m+1)} + \sum_{m=M_0+1}^{\infty} \alpha^*_m 2^{-(m+1)}$$

$$< M_0 \cdot \frac{\epsilon}{2M_0} + \frac{\epsilon}{4} + \frac{\epsilon}{4} < \epsilon$$

□

proving the lemma.

Theorem 3.4: Let nodes (K) be the number of nodes in a kcu-optimal tree of K keys. Then for large h, the expansion $(= \frac{2 \cdot \text{nodes}(K)}{K} - 1)$ averaged over $K = 2^h - 1, \ldots 2^{h+1} - 2$ is

$$\left(2 \sum_{m=0}^{\infty} \left[m - ((m-1)2^{m+2} + 2) \ln\left(\frac{2^{m+2}-1}{2^{m+2}-2}\right)\right] \cdot 2^{-(m+1)}\right) - 1$$

which can be numerically calculated as 56.7%

Proof: In a manner analagous to Theorem 3.3, we consider

$$\left(\sum_{m=0}^{h-1} \sum_{x=1}^{2^{h-m-1}} 2 \cdot \frac{2^{h+1} - (m+1)2^{h-m} + mx - 1}{2^{h+1} - 2^{h-m} - 1 + x} \right) / 2^h - 1$$

and define $\alpha_{h,m}$, $\bar{\alpha}_{h,m}$ and α^*_m. Using Lemma 3.4 gives

$$\alpha_{h,m} = 2[m(2^{h-m-1} - 1) + (2^{h+1} - (m+1)2^{h-m} - 1 - m(2^{h+1} - 2^{h-m} - 1))$$

$$\times \ln\left(\frac{2^{h-m-1} + 2^{h+1} - 2^{h-m} - 1}{1 + 2^{h+1} - 2^{h-m} - 1}\right)] + \epsilon$$

where $1 \leq \epsilon \leq 2$.

Dividing by 2^{h-m-1} and taking the limit as $h \to \infty$ gives

$$\alpha^*_m = 2\left[m - ((m-1)2^{m+2} + 2) \ln\left(\frac{2^{m+2}-1}{2^{m+2}-2}\right)\right]$$

and substituting into $\sum_{m=0}^{\infty} \alpha^*_m \cdot 2^{-(m+1)} - 1$ proves the theorem. Note that the interchange of sum and limit is still valid; an analog of Lemma 3.5 can easily

be proved because $1 \leq \bar{\alpha}_{h,m} \leq 2$.

As was stated before, this expansion of 56.7% is comparable to the expansion of a random tree which is known to be bounded (5) between 40% and 58%. Note that no such bound on the average utilization of random 2-3 trees is known, and that we cannot add one to these bounds, take their reciprocals and get bounds on the utilization because the equality "E(x) = 1/E(1/x)" does not hold for an arbitrary random variable x.

4. A Tree Construction Algorithm

In this section we describe an O(n) algorithm to construct a kcu-optimal 2-3 tree from a sorted array of keys (See Figure 4.1). The algorithm builds the tree top-down. If K, the number of keys in the tree to be built, is of the form $2^{h+1}-2$, then a full tree must be constructed. This is done in lines 5-15. If K is not of this form, (lines 16-25) the root of the tree must be 1-node, and the other keys must be partitioned into two subtrees. Many partitioning strategies are possible. Our algorithm divides the keys as evenly as possible with the left over key (if there is one) going into the left subtree. We now prove that the "even splitting" tree construction algorithm actually does construct a kcu-optimal tree.

```
1.  Function Build (lower,upper,height);
2.  Var  k1,k2:integer;
3.  Begin
4.      If height=0 then return NIL
5.  else if upper-lower+1=2^(height+1)-2 then begin
6.          allocate a node;
7.          k1:=lower+2^height-2;
8.          k2:=lower+2^height+2^(height-1)-2;
9.          node(leftkey):=keys[k1];
10.         node(rightkey):=keys[k2];
11.         node(leftson):=Build(lower,k1-1,height-1);
12.         node(middleson):=Build(k1+1,k2-1,height-1);
13.         node(rightson):=Build(k2+1,upper,height-1);
14.         return a pointer to node;
15. end
16. else begin
17.         Allocate a node;
18.         k1:= ⌈(lower + upper)/2⌉;
19.         node(leftkey):=keys[k1];
20.         node(rightkey):=empty;
21.         node(leftson):=Build(lower,k1-1,height-1);
22.         node(middleson):=Build(k1+1,upper,height-1);
23.         node(rightson):=NIL,
24.         return a pointer to node;
25. end;
26. end;
```

Figure 4.1

The "even splitting" algorithm for constructing a kcu-optimal tree from a given set of keys. We assume the n keys are stored in sorted order in position 1 through K of array "keys". Build is a function, returning a pointer to a tree. We assume the original invocation is Build(1,K,$\lfloor$log(K+1)$\rfloor$) which returns a pointer to the root of the tree.

Theorem 4.1 The "even splitting" algorithm produces a kcu-optimal tree.

Proof: (by induction on K): Proving the theorem for K=1 is trivial, so consider K > 1. If K is of the form $2^{h+1}-2$, the algorithm will construct a full tree, which is then kcu-optimal. If K is not of this form, $\lceil (K-1)/2 \rceil$ keys form the left subtree and $\lfloor (K-1)/2 \rfloor$ the right. Let $(h-1,\ell_1,x_1)$ and $(h-1,\ell_2,x_2)$ be, respectively, the leader profiles of the left and right subtrees. If K is odd, $\lceil (K-1)/2 \rceil = \lfloor (K-1)/2 \rfloor$, and the structure of the subtrees will be identical. In this case, $\ell_1 = \ell_2$. All leaders are on level ℓ_1 or ℓ_1+1 and the levels of two leaders can differ by at most one. Hence the tree is kcu-optimal. If K is even, the left subtree will have one more key than the right. It may be the case that $\ell_1=\ell_2$ in which the proof for odd K holds. If $\ell_1 \neq \ell_2$ Corollary 3.1 states that the leader profiles must be $(h-1,\ell_1,1)$ and $(h-1,\ell_2,2^{\ell_2})$ with $\ell_1=\ell_2-1$ because the number of keys in the subtrees differs by exactly one. The left subtree has leaders on level ℓ_1 and ℓ_1+1. The right subtree has leaders on only level $\ell_2 (=\ell_1+1)$, and there can be no lower leaders at level ℓ_2+1. Again, the levels of the leaders differ by at most one, and the tree is kcu-optimal. □

5. An Insertion Algorithm

We describe an algorithm to insert a key into a kcu-optimal tree while maintaining the property of kcu-optimality. A similar, but more complicated, deletion algorithm can also be defined.

The algorithm uses two subroutines, Exchange and Split. The need for Exchange is due to the fact that in the insertion algorithm, we often cannot insert the new key into the correct subtree without destroying the kcu-optimality of the tree. When this occurs, we will do an "Exchange" to get a key that can be inserted into a different subtree. The Exchange algorithm (see Figure 5.1) accomplishes this. It takes a key as input and inserts it into the subtree pointed to by p. If flag="min", it deletes the smallest key in p's subtree and returns its value in keyout. If flag="max", it returns the largest. Thus, the number of keys in p's subtree remains the same. However, we now have a more convenient key. We refer to this operation as "doing an exchange."

To do an Exchange, we first check if p's sons are leaves. If they are, the operation is particularly easy (lines 5-7). Otherwise we must call the algorithm recursively to exchange the key. Suppose flag="min." (The case for flag="max" is similar.) If key has to be inserted into the right subtree (lines 12-15), we must exchange it with the minimum key in the right subtree. The key replaces the current right key of p, and the current right key of p becomes the new key to be inserted. This key is then exchanged with the minimum key in the middle subtree (lines 16-20) and the resulting key is exchanged with the minimum

key in the left subtree (line 21). This final key is returned as the value of keyout. A similar (though less complex) operation occurs if the original key to be inserted fell in the left or middle subtree.

The Split algorithm (Figure 5.2) takes a subtree pointed to by p which must be a full subtree and a key to be inserted. It splits the subtree into two completely binary trees (pointed to by lptr and rptr) plus a key (returned in keyout) whose value is between the two trees. We first check for the trivial case (lines 3-9). If the tree is non-trivial, the middle and right subtrees, together with the p's right key, form one completely binary tree. To form the other, exchanges are done (lines 11-20) until the key to be inserted lies in the left subtree. We then call Split recursively (line 21) to split this full subtree.

We define a node to be open if it has a lower-leader in its subtrees. Otherwise it is said to be closed. (Note that if all leaders are on the same level, all are considered to be lower leaders.) If a node is open, a key can be inserted somewhere into its subtree. Otherwise, exchanges must be done to allow us to insert into some other subtrees. We assume that a bit denoting whether a node is open or closed is stored with each node and is updated as necessary as insertions are made.

We now describe the insertion algorithm. It first check if the root is a 2-node (lines 4-10), in which case the Split algorithm is called to split the entire tree. Otherwise the root is a 1-node,and we descend in the tree. If the branch the current key forces us to take

is closed (lines 17-23), we must exchange and take the other branch. It is easy to see that because we are always following an open branch, p will always be open at line 12. Eventually p's sons will be leaves (which is easily handled in line 11) or p's sons will both be 2-nodes. In the latter case, we exchange if necessary (lines 31-35) to allow us to insert into the right subtree. We then take the key to be inserted and split the right subtree (creating two binary trees) and making p a 2-node (lines 36-39).

A simulation was run to determine the average cost of an insertion with the average, as before, over trees with $K = 2^h-1,\ldots,2^{h+1}-2$ keys. A run in the simulation began with a completely binary tree of height h (in our simulation h=10). Random keys were then inserted one by one into the tree until a full tree of height h was obtained. Each key was chosen under the usual assumption that each of the K+1 gaps between, before and after the keys in the tree was equally likely. The average reorganization required (the number of keys that had to be moved) ranged from 3.09% to 4.23%, and the average of the 10 runs was 3.62%.

```
1   Procedure Exchange(key,keyout,p,flag);
2   var temp : keytype;
3   begin
4   if p's sons are leaves then begin
5           let keyout = the minimum (or maximum, depending on flag) out
6                   of key and p's keys.
7           put the remaining keys into p.
8   end
9   else
10       if flag = "min" then begin
11                  if p is a 2-node and key > p(rightkey) then begin
12                          Exchange(key,temp,p(rightson),"min");
13                          key := p(rightkey);
14                          p(rightkey) := temp;
15                  end
16                  if key > p(leftkey) then begin
17                          Exchange(key,temp,p(middleson),"min");
18                          key := p(leftkey);
19                          p(leftkey) := temp;
20                  end
21                  Exchange(key,keyout,p(leftson),"min");
22          end
23          else begin
24                  {similar to lines 11 - 21}
25          end;
26  end;
27  end;
```

Figure 5.1

```
1   Procedure Split(key,keyout,lptr,rptr,p);
2   var lptrl,rptrl : ptrtype;
3   begin
4   if p's sons are leaves then begin
5           Allocate two new nodes, pointed to by lptr and rptr.
6           Put the largest of key and p's two keys into the node
7                   pointed to by rptr.
8           Put the smallest into that pointed to by lptr.
9           Return the remaining key in keyout.
10  end
11  else begin
12          if key > p(rightkey) then begin
13                  Exchange(key,temp,p(rightson),"min");
14                  key := p(rightkey);
15                  p(rightkey) := temp;
16          end;
17          if key > p(leftkey) then begin
18                  Exchange(key,temp,p(middleson),"min");
19                  key := p(leftkey);
20                  p(leftkey) := temp;
21          end;
22          Split(key,keyout,lptrl rptrl,p(leftson));
23          Allocate a node. Point lptr to it.
24          Build the trees as shown in Figure 5.3.
25  end;
26  end;
```

Figure 5.2

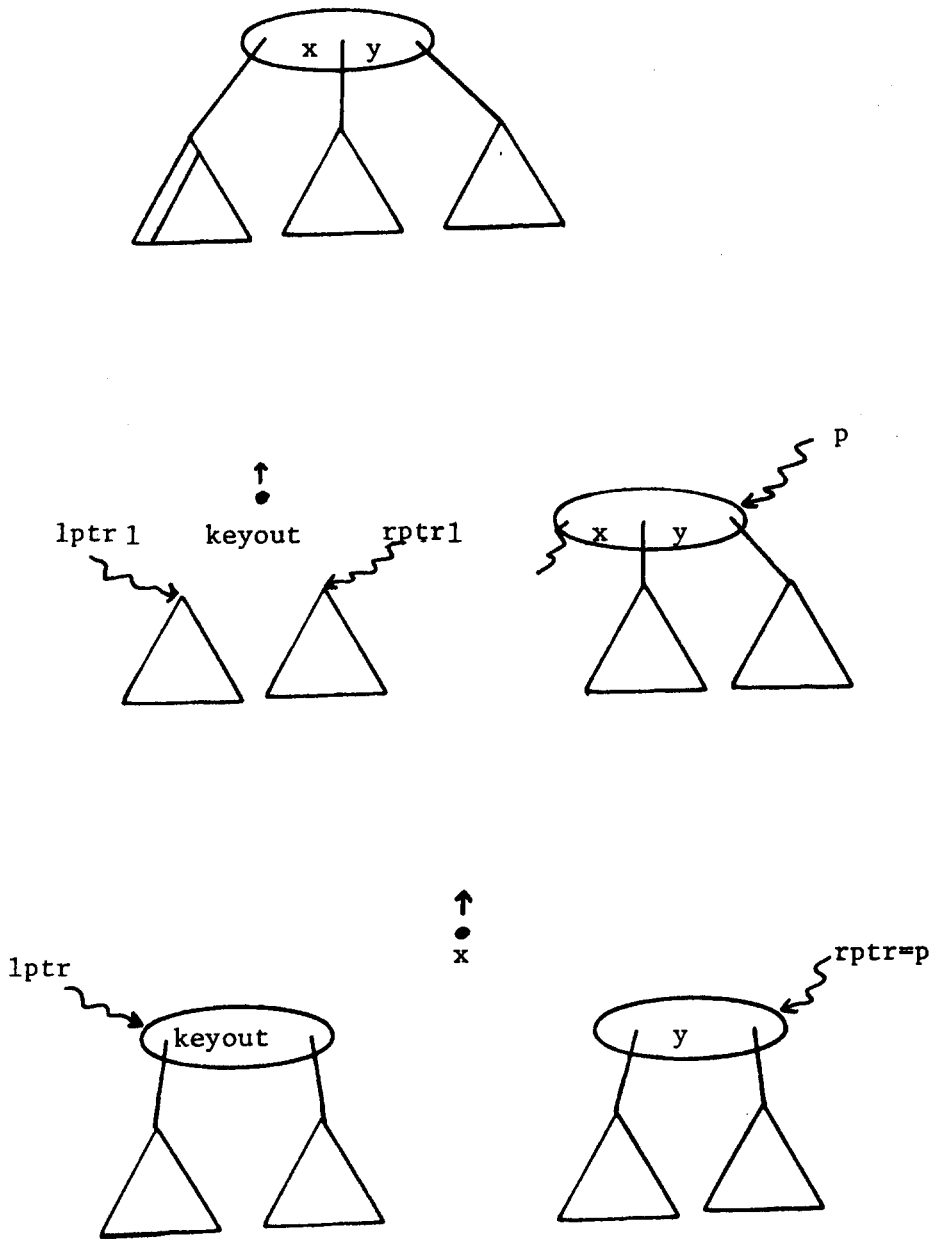

Figure 5.3

The final building of the trees in the Split algorithm

```
1   Procedure Insert(key,p);
2   var temp,keyout : keytype;
3   begin
4   if p is the root and a 2-node then begin
5           Split(key,keyout,lptr,rptr,p);
6           Allocate a new node which becomes the root of the tree.
7           node(leftkey) := keyout;
8           node(leftson) := lptr;
9           node(middleson) := rptr;
10  end
11  else if p's sons are leaves then insert key into p.
12  else if at least one of p's sons is a 1-node then begin
13  {p is an open 1-node}
14          if key < p(leftkey) then begin
15                  if p(leftson) is open
16                          then Insert(key,p(leftson))
17                          else begin
18                                  {right subtree is open}
19                                  Exchange(key,temp,p(leftson),"max");
20                                  key := p(leftkey);
21                                  p(leftkey) := temp;
22                                  Insert(key,p(middleson));
23                          end
24          end
25          else begin
26                  {similar to lines 15 - 23}
27          end
28  end
29  else begin
30          {p is open and both sons are 2-nodes}
31          if key < p(leftkey) then begin
32                  Exchange(key,temp,p(leftson) "max");
33                  key := p(rightkey);
34                  p(rightkey) := temp;
35          end,
36          Split(key,temp,lptr,rptr,p(middleson));
37          p(rightkey) := temp;
38          p(middleson) := lptr,
39          p(rightson) := rptr;
40  end;
41  end;
```

Figure 5.4

References

(1) A.L. Rosenberg and L. Snyder, Minimal comparison 2-3 Trees SIAM J. Comput. Nov. 1978.

(2) D.E. Knuth, The Art of Computer Programming, Vol. 3, Addison-Wesley, 1973.

(3) R.E. Miller, N. Pippenger, A.L. Rosenberg, L. Snyder. Optimal 2-3 Trees, SIAM J. Comput., to appear. Feb. 1979.

(4) A.L. Rosenberg and L. Snyder, Compact 2-3 Trees, IBM Tech. Rept. RC-7343.

(5) A.C. Yao, Random 3-2 Trees, Acta Informatica.