BINARY SEARCH TREES
POEMS ARE MADE BY FOOLS LIKE ME

BUT ONLY GOD CAN MAKE A TREE
POEMS ARE MADE BY FOOLS LIKE ME

BUT ONLY GOD CAN MAKE A TREE
AN INVESTIGATION AND IMPLEMENTATION
OF
SOME BINARY SEARCH TREE ALGORITHMS

By
ALDON N. WALKER

A Project
Submitted to the School of Graduate Studies
in Partial Fulfilment of the Requirements
for the Degree
Master of Science

McMaster University
November 1974
MASTER OF SCIENCE
(Computation)  McMaster University
Hamilton, Ontario

TITLE: An Investigation and Implementation of Some Binary Search Tree Algorithms

AUTHOR: Aldon N. Walker

SUPERVISOR: Professor D. Wood

NUMBER OF PAGES: viii, 149, A1-16, A2-8, A3-90, A4-2
ABSTRACT

This project documents the results of an investigation into binary search trees. Because of their favourable characteristics, binary search trees have become popular for information storage and retrieval applications in a one level store. The trees may be of two types, weighted and unweighted. Various algorithms are presented, in a machine independent context, for both types and an empirical evaluation is performed. An important software aid used for graphically displaying a binary tree is also described.
ACKNOWLEDGEMENTS

I would like to thank Professor D. Wood for his assistance and supervision of this project and his comments and suggestions regarding its form and content.

My thanks go to Professors K. Redish and R. Rink for their reading of this manuscript.

Also I would like to thank Mrs. S. Coutts for her patient and careful typing of this project.

Finally a vote of thanks to the friends I have made who made my stay at McMaster enjoyable.
# TABLE OF CONTENTS

**CHAPTER I: SURVEY AND OVERVIEW**

1.1 Introduction 1  
1.2 Survey 1  

**CHAPTER II: DEFINITIONS AND TERMINOLOGY**

2.1 Basic Definitions and Notation 5  

**CHAPTER III: BINARY SEARCH TREE CONSTRUCTION ALGORITHMS**

3.1 Introduction to and Overview of the Algorithms 13  
3.1.1 General Introduction 13  
3.1.2 General Implementation Details Algorithms 14  
3.2 The Basic Tree Algorithms 17  
3.2.1 Introduction 17  
3.2.2 The Basic Tree Insertion Algorithm (BTIA) 17  
3.2.3 The Basic Tree Deletion Algorithm (BTDA) 18  
3.2.4 Implementation 19  
3.3 The AVL Tree Algorithms 20  
3.3.1 Introduction 20  
3.3.2 The AVL Tree Insertion Algorithm (AVLIA) 21  
3.3.3 The AVL Tree Deletion Algorithm (AVLDA) 22  
3.3.4 Implementation 24  
3.4 The Bounded Balance Tree Algorithms 40  
3.4.1 Introduction 40  
3.4.2 The Bounded Balance Insertion Algorithm (BBIA) 42  
3.4.3 The Bounded Balance Deletion Algorithm (BBDA) 43  
3.4.4 Implementation 45  
3.5 The Bell-Tree Algorithms 60  
3.5.1 Introduction 60  
3.5.2 The Bell-Tree Insertion Algorithm (BIA) 61  
3.5.3 The Bell-Tree Deletion Algorithm (BDA) 63  
3.5.4 Implementation 64
3.6 Knuth Construction Algorithm (KNCA) 77
   3.6.1 Introduction 77
   3.6.2 The KNCA 78
   3.6.3 Implementation 79

3.7 The Bruno Coffman Construction Algorithm (BCCA) 80
   3.7.1 Introduction 80
   3.7.2 The BCCA 82
   3.7.3 Implementation 82

3.8 The Walker Gotlieb Construction Algorithm (WGCA) 88
   3.8.1 Introduction 88
   3.8.2 The WGCA 91
   3.8.3 Implementation 92

CHAPTER IV: EMPIRICAL RESULTS 94
4.1 Introduction 94
4.2 Empirical Observations on Algorithms of Class (1) 100
   4.2.1 AVL Trees 100
   4.2.2 Bell-Trees 101
   4.2.3 BB-Trees 103
   4.2.4 Conclusions 105

4.3 Empirical Observations on Algorithms of Class (2) 135
   4.3.1 The Optimal Tree Algorithm of Knuth 135
   4.3.2 The Bruno Coffman Algorithm 135
   4.3.3 The Walker Gotlieb Algorithm 135
   4.3.4 Conclusions 135

CHAPTER V: CONCLUDING REMARKS 144
REFERENCES 146

APPENDIX 1: A SIMPLE BINARY TREE DISPLAY ALGORITHM
APPENDIX 2: UTILITY ROUTINES
APPENDIX 3: LISTINGS OF THE BINARY TREE CONSTRUCTION ALGORITHMS
APPENDIX 4: PRACTICAL CONSIDERATIONS
| Table 4.1.1 | Test Data for Algorithms of Class (2) | 95 |
| Table 4.2.1 | AVL Tree Insertion Statistics | 107 |
| Table 4.2.2 | AVL Tree Deletion Statistics | 108 |
| Table 4.2.3 | Bell-Tree Insertion Statistics Average Search | 109 |
| Table 4.2.4 | Bell-Tree Insertion Statistics Average Single Rotation | 110 |
| Table 4.2.5 | Bell-Tree Insertion Statistics Average Double Rotation | 111 |
| Table 4.2.6 | Bell-Tree Insertion Statistics Average General Rotation | 112 |
| Table 4.2.7 | Bell-Tree Insertion Statistics Average Single Recurse | 113 |
| Table 4.2.8 | Bell-Tree Insertion Statistics Average Double Recurse | 114 |
| Table 4.2.9 | Bell-Tree Insertion Statistics Average Recurse Rotation | 115 |
| Table 4.2.10 | Bell-tree Insertion Statistics Average Total Rotation | 116 |
| Table 4.2.11 | Bell-Tree Insertion Statistics Average Shift | 117 |
| Table 4.2.12 | Bell-Tree Insertion Statistics Average Recurse | 118 |
| Table 4.2.13 | Bell-Tree Deletion Statistics Average Search | 119 |
| Table 4.2.14 | Bell-Tree Deletion Statistics Average Nosons | 120 |
| Table 4.2.15 | Bell-Tree Deletion Statistics Average Oneson | 121 |
| Table 4.2.16 | Bell-Tree Deletion Statistics Average Recurse | 122 |
| Table 4.2.17 | Bell-Tree Deletion Statistics Probability Nosons | 123 |
| Table 4.2.18 | Bell-Tree Deletion Statistics Probability Onesons | 124 |
| Table 4.2.19 | Bell-Tree Deletion Statistics Probability Twosons | 125 |
| Table 4.2.20 | BB-tree Insertion Statistics Average Search | 126 |
| Table 4.2.21 | BB-tree Insertion Statistics Average Single Rotation | 127 |
| Table 4.2.22 | BB-tree Insertion Statistics Average Double Rotation | 128 |
| Table 4.2.23 | BB-tree Insertion Statistics Average General Rotation | 129 |
Table 4.2.24  BB-tree Deletion Statistics Average Search  130
Table 4.2.25  BB-tree Deletion Statistics Average Single Rotation  131
Table 4.2.26  BB-tree Deletion Statistics Average Double Rotation  132
Table 4.2.27  BB-tree Deletion Statistics Average General Rotation  133
Table 4.2.28  Timing on Statistical Runs for Algorithms of Class (1)  134
Table 4.3.1  Average Weighted Path Length of Trees Constructed by the KNCA  137
Table 4.3.2  Bruno Coffman Statistics  138
Table 4.3.3  Walker Gotlieb Statistics  141
Table 4.3.4  Comparison of Class (2) Algorithms  143
CHAPTER I
SURVEY AND OVERVIEW

1.1 INTRODUCTION

Binary trees are an important technique for organizing large files of information because they strike a compromise between the various conflicting requirements of storage utilization, rapid retrieval time and ease of insertion and deletion of information. A great deal of effort has been given to the study of binary trees. Because of this, their properties have become better understood and new ideas have evolved for the optimization of the above requirements. Although they are not the only method of file organization (c.f. hash coding or scatter storage techniques) and in some cases not the best, for one-level storage applications they have become popular.

1.2 SURVEY

For discussion purposes binary search trees can be considered as being of two classes. The first class represents an attempt to keep the tree more or less balanced i.e. no excessively long or short search paths exist. The second class of trees have an associated set of weights which can be thought of as the frequency of a successful or unsuccessful search for particular keys. The trees in this class are constructed so as to minimize the weighted pathlength of the tree, meaning in general, that they would not necessarily be balanced. Trees of this class are referred to as optimal. Other types of trees related to the binary search tree have been reported. Fredkin (1960) proposed the TRIE structure,
Sussenguth (1963) defined the doubly-chained tree and Bayer and McCreight (1972) defined the binary B-tree. Further work and investigation of doubly-chained trees has been reported by Patt (1969, 1972), Stanfel (1969, 1970, 1972, 1973), Kennedy (1972a, 1972b) and Hu (1972a). Bayer (1972) has developed what he calls a symmetric B-tree, a modification of the B-tree.

The binary search tree and its application to searching and file maintenance was first introduced by Douglas (1959), Windley (1960), Booth and Colin (1960) and later Hibbard (1962). Windley and Hibbard derive formulas for the mean and variance of the number of comparisons necessary to insert an item into a random tree and Lynch (1965) gives a more detailed examination which encompasses the previous results. Other work concerning the mathematical properties of random trees has been done by Arora and Dent (1969).

Early work reported on balanced binary trees was given by Landauer (1963), although his trees had a structure different from the conventional binary search tree. A very elegant algorithm was formulated by Adelson-Velskii and Landis (1962) for creation and maintenance of a binary tree. This algorithm has been further commented on by Foster (1965a, 1965b, 1973), Knott (1971), Reingold (1971), Martin and Ness (1972) and Knuth (1973). Upon insertion of a key the tree is restructured to keep it "AVL balanced". The deletion algorithm has been given by Knott and Knuth and some empirical investigations appear in Scroggs, Karlton, Fuller and Kaehler (1973). Another method of keeping trees balanced was described by Bell (1965). Here an attempt to reduce the mean value of the tree (see Windley (1960)) was made by creating minimal subtrees of
a given size at each stage of the construction. An algorithm has been formulated by Walker and Wood (1974) for the insertion and deletion of nodes from this tree and some empirical investigations performed. Wong and Chang (1971) describe and analyse a method for constructing balanced binary trees. Their method structures the first $n_1 = 2^{\ell + 1} - 1$ nodes into a random tree for some positive $\ell$. This tree is then rearranged into a balanced binary tree. The process is then repeated for the next $n_2 = 2^{2(\ell + 1)} - 1$ nodes. A different class of binary search trees has been proposed by Nievergelt and Reingold (1972). They are called trees of "bounded balance" (BB-trees) and contain a parameter which can be varied so as to compromise between a short search time and infrequent restructuring. Their algorithms are similar in concept to those of Adelson-Velskii and Landis in that restructuring may take place, upon the insertion or deletion of any key, to maintain the balance of the tree.

The second class of binary search trees was first investigated by Knuth (1971). He gave an algorithm to construct an optimal binary tree and posed several questions about structuring optimal trees which became important in later studies. His algorithm requires space and time proportional to $n^2$ and is of limited practical use. For this reason some heuristic algorithms have been devised. They centre around the idea of the weight of the subtrees of nodes being almost equal. Bruno and Coffman (1971) give a very simple heuristic for constructing nearly optimal trees. Given a starting tree, it is transformed into one with a reduced weighted path length by promoting keys with heavier weights nearer to the root. Weiner (1971) also speaks of a heuristic for nearly optimal trees but no details are available. Walker and Gotlieb (1972a) gave an ef-
icient top-down heuristic algorithm for nearly optimal trees. Their method chooses as the key of the root that key of maximum weight which best minimizes the weight of the root's left and right subtrees. When the tree to be constructed has a small number of keys, Knuth's optimal algorithm is used.

Finally a new type of tree called a hybrid tree has been defined by Walker and Gutlieb (1972b). It represents a generalization of both the binary tree and the TRIE. Its search time is no worse than that of the TRIE or an optimal binary tree on the same set of keys.

Important contributions to the analysis of binary search trees as regards weighted path length has been given by Nievergelt and Wong (1970, 1973), Nievergelt and Pradels (1972), Rissanen (1973), Hu and Tan (1972a, 1972b) and Hu and Tucker (1971). A method of displaying a binary tree in a readable format has been given by Walker, Redish and Wood (1973) and is found in Appendix 1.
CHAPTER II
DEFINITIONS AND TERMINOLOGY

2.1 BASIC DEFINITIONS AND NOTATION

For the purpose of this project we define an unweighted binary search tree (henceforth a binary tree or simply a tree) as follows.

Definition

A binary tree is a finite set of nodes; if it has no nodes it is called a null tree. The following conditions obtain for each non-null binary tree T.

(i) One node of T is a distinguished node called the root of T.

(ii) Each node has at most two successors which are designated left and right sons, the node itself is the father. If a node has no sons it is a leaf, if it has one son and this one son is a leaf, it is a semi-leaf.

(iii) Each node has associated with it an item of information called a key, which is assumed to be a character string.

(iv) There is an ancestor relation on T defined by: given two nodes u and v in T, u is an ancestor of v iff either u is the father of v or u is the father of some node w and w is an ancestor of v. T must be connected, that is every node apart from the root, must have the root as one of its ancestors, and acyclic, that is no node can be its own ancestor. Hence given any two nodes it makes sense to say that one node is either to the left or to the right of the
other node.

(v) There is a linear ordering defined upon the set of possible keys by some transitive relation \(<\) (read "precedes"), which, if the set of keys is some subset of the integers, is the usual "less than" relation.

(vi) For all nodes \(u\) and \(v\) such that \(u\) is to the left of \(v\), the key of \(u\) precedes the key of \(v\).

In the following we use the notation \(\text{tree}(u)\) to indicate either a tree with root node \(u\) or, if \(u = \emptyset\), a null tree. In tree diagrams throughout this project the key of a node is displayed as the label of a node. The convention of late lower case letters for nodes and late upper case letters for keys is adopted. It should also be noted that if condition (vi) of the above definition is omitted the tree is no longer a binary search tree.

We say a set of \(n\) keys \(X_i, 1 \leq i \leq n\) is \(\text{ordered}\) if \(X_i < X_{i+1}\), \(1 \leq i < n\). Each of the \(n\) keys of a tree may have associated with it a non-negative \(\text{weight}\) which for any key \(X_i\) is denoted \(\alpha_i, 1 \leq i \leq n\). These weights can be thought of as the frequency with which a key is expected to be searched for in the tree. There may also be a set of weights denoted \(\beta_i, 0 \leq i \leq n\), which give the frequency of unsuccessful searches and are called \(\text{external weights}\). Considering the keys as ordered, \(\beta_i\) is the frequency of a key occurring between \(X_i\) and \(X_{i+1}\), \(\beta_0\) and \(\beta_n\) having obvious interpretations. Therefore the more general binary search tree will be one of \(n\) keys \(X_i, 1 \leq i \leq n\), and \(2n+1\) non-negative weights \(\alpha_i, 1 \leq i \leq n\), and \(\beta_i, 0 \leq i \leq n\).
In the above definition of an unweighted binary search tree, the $\beta_i$ are zero and the $\alpha_i$ are one. Using the six conditions given above we define two more classes of binary trees. A **weighted binary search tree** is one in which $\alpha_i > 1$, $1 \leq i \leq n$, the external weights being zero. A **weighted external binary search tree** is a weighted tree where the $\beta_i > 0$, $0 \leq i < n$. Figures 2.1.1 and 2.1.2 show examples of an unweighted and weighted binary tree. The circular nodes are denoted **internal nodes**. The weight of a key defining a node is found directly below each node. Figure 2.1.3 illustrated a weighted external tree. The square nodes are called **external nodes**, the external weights being found beneath them.

Given an $n$ node tree, $\text{tree}(u)$, with $2n+1$ weights $\alpha_i$ and $\beta_i$ we have the following definitions and terminology.

**Definition**

A **search path** from $u$ to some node $v$ in $\text{tree}(u)$, $P(v)$, is a sequence of nodes $u_1, \ldots, u_m$, $m \geq 0$, such that $u = u_1$, $v = u_m$ and $u_i$ is the father of $u_{i+1}$, $1 \leq i < m$. The **level** of a node $u$, $l(u)$, is the number of nodes in $P(v)$. The **height** of $\text{tree}(u)$, $h(u)$, is the maximum level of any node in the tree. The **size** of $\text{tree}(u)$, denoted $\text{size}(u)$, is the number of nodes in $\text{tree}(u)$. For any $v$ in $\text{tree}(u)$, $v_l$ and $v_r$ denote the left and right sons of $v$. The **subtree defined by** $v$, $\text{tree}(v)$, is the set of all nodes $w$ in $\text{tree}(u)$ such that $v$ is an ancestor of $w$ together with $v$ itself. $\text{key}(v)$ denotes the key associated with $v$. The **weight** of $\text{tree}(u)$, $W(u)$ is defined as the sum of the associated weights, i.e. $W(u) = \sum_{i=1}^{n} \alpha_i + \sum_{i=0}^{n} \beta_i$. The **weighted path length** of $\text{tree}(u)$, $\text{WPL}(u)$...
is defined as the sum of the products of the weight of a key times the level of the node defined by the key plus the sum of the products of an external weight times the corresponding level of the associated external node, i.e. \( WPL(u) = \sum_{i=1}^{n} \alpha_i \times \ell(u_i) + \sum_{i=0}^{n} \beta_i \times \ell(y_i) \) where \( y_i \) denotes an external node. Note that \( WPL(u) = WPL(u_{\emptyset}) + WPL(u_{\emptyset}) + W(u) \). The \textit{average weighted path length} is defined as \( \text{AVGWPL}(u) = \frac{WPL(u)}{W(u)} \). The value of tree(u), \( \forall \), is \( \ell(v) \) for all \( v \) in tree(u). The \textit{minimum value}, denoted \( \text{MINV}(n) \), is \( (n+1) \times r - n + 2p \), where \( r \) is such that \( n = 2^r - 1 + p \) and \( 0 \leq p \leq 2^r \). The \textit{maximum value} of tree(u), \( \text{MAXV}(n) \), is \( 1 + 2 + \ldots + n = \frac{n(n+1)}{2} \).

Letting \( T_n \) be the set of all ordered trees of \( n \) nodes (with keys 1, \ldots, n), then the \textit{mean value} of an \( n \) node tree, \( \langle \text{MV}(n) \rangle \), is given by the sum of the values of the trees in \( T_n \) divided by the number of trees in \( T_n \).

When the term \textit{average search length} is used the average weighted path length is inferred. An optimal tree is one in which the weighted path length is minimized. In the case that the \( \alpha_i \)'s are unity and the \( \beta_i \)'s are zero, the minimization of the weighted path length produces a minimum valued tree. We say that a key, \( X \), is \textit{inserted into} tree(u) if there exists tree(u'), such that tree(u) \( \subseteq \) tree(u'), \( u = u' \) unless \( u = \emptyset \), size(u') = n+1, tree(u') is a binary search tree, and for \( v \) in tree(u') \( \subseteq \) tree(u), key(v) = X. Correspondingly, we say that a key, \( X \), is \textit{deleted from} tree(u) if there exists tree(u') such that tree(u') \( \subseteq \) tree(u), \( u = u' \) unless key(u) = X, size(u') = n-1, tree(u') is a binary search tree, and for \( v \) in tree(u') \( \subseteq \) tree(u'), key(v) = X. Finally we define the notions of "remove" and "replace" for binary trees (see Figure 2.1.4). Given tree(u), \( u \neq \emptyset \), then given a node \( v \) in tree(u), \( v \) is \textit{removed from} tree(u).
key, $X$ say, and a node $v$ in $\text{tree}(u)$ then $\text{key}(v)$ is replaced by $X$
when $v$ is given $X$ as its key.
An example of an unweighted tree

Figure 2.1.1

An example of a weighted tree

Figure 2.1.2
An example of a weighted external tree.

Figure 2.1.3
Remove node whose key is 8

Replace 10 with 8, note that the replacement key has a restricted value if the resulting tree is to be a binary search tree.

Figure 2.1.4
CHAPTER III

BINARY SEARCH TREE CONSTRUCTION ALGORITHMS

3.1 INTRODUCTION TO AND OVERVIEW OF THE ALGORITHMS

3.1.1 GENERAL INTRODUCTION

For the purpose of discussion the algorithms described below can be divided into two classes:

(1) those that locally restructure the tree as keys are input to the algorithm (Sections 3.3, 3.4, 3.5) and

(2) those that globally restructure the tree and require a fixed sequence of input keys (sections 3.6, 3.7, 3.8).

The class (1) algorithms construct unweighted binary trees which are considered to be dynamic, i.e. frequent insertions and deletions are performed. We are interested in minimizing the average search path which, because the tree is unweighted, means no excessively long or short search paths to the leaves should exist. Algorithms of class (2) construct weighted or weighted external trees which are considered static, i.e. a fixed set of keys and weights are given and no deletions or insertions are performed once the tree has been created. Trees in both classes are considered to be completely kept in one level random access storage.

Transformations are performed on the trees produced by algorithms of class (1) and the trees given in Section 3.7 in order to achieve the desired structure. These transformations are illustrated in Figure 3.1.1 and henceforth will be referred to as "general transformations".
3.1.2 **GENERAL IMPLEMENTATION DETAILS**

All algorithms are implemented in Pascal, (Wirth, 1971), and the following conditions obtain:

(i) Keys are of type ALFA, a packed ARRAY of ten CHARACTERS. Keys of less than ten characters are left justified and zero filled.

(ii) Each node of the binary tree is represented as a Pascal record which is assumed to contain at least three fields of information,

1. the key of the node (represented by the variable name info of type ALFA),
2. a pointer to a node's left son (represented by the variable name lpt), and
3. a pointer to a node's right son (represented by the variable name rpt).

Each node is referenced by a pointer variable and the fields of the record are referenced through a pointer variable and their respective field names, i.e. (if p is a pointer variable)

1) p^.info
2) p^.lpt
3) p^.rpt.

(iii) Use is made of the Pascal special symbol **nil** to indicate a null pointer, so for example if p points to a node which is a leaf, then both p^.lpt and p^.rpt will be **nil** since it has no sons.
A listing of the Pascal procedures for each tree is found in Appendix 3. The utility routines which are needed by most of the construction algorithms are found in Appendix 2.
A single rotation

A double rotation

General transformations

Figure 3.1.1
3.2 THE BASIC TREE ALGORITHMS

3.2.1. INTRODUCTION

The basic binary tree algorithm is the simplest of all binary tree algorithms. It was the first to be implemented but because it results in an unfavourable valued tree, in general it is not a good algorithm except, perhaps, for short-lived trees. There is no restructuring of any kind; the resulting tree structure depends entirely on the sequence of keys input to the algorithm. See figure 3.2.1 for an example of a binary search tree.

3.2.2 THE BASIC TREE INSERTION ALGORITHM (BTIA)

We assume that we are given tree(u) and that the key to be inserted is X.

(i) If u = ∅ then enter X as the key of the root node and stop.
(ii) If X equals key(u) then X is already in the tree so stop.
(iii) If X precedes key(u) then repeat BTIA for tree(v), the left subtree of u.
(iv) If X does not precede key(u) then repeat BTIA for tree(v), the right subtree of u.
3.2.3 **THE BASIC TREE DELETION ALGORITHM (BTDA)**

We assume we are given tree(u) and that the key to be deleted is X.

(i) If \( u = \emptyset \) then X is not in the tree so stop.

(ii) If X equals key(u) then three cases arise:

(a) \( u \) has no sons.

Simply remove \( u \).

(b) \( u \) has exactly one son, \( v \) say.

Tree(u) is replaced by tree(v), that is, \( v \) is the new root node of tree(u) and \( u \) is removed.

(c) \( u \) has two sons.

Let \( z \) be the postorder predecessor of \( u \). Replace \( \text{key}(u) \) with \( \text{key}(z)^+ \) and repeat step (ii) with \( u = z \) and \( X = \text{key}(z) \).

(iii) If X precedes key(u), then repeat BTDA for tree(v), the left subtree of \( u \), otherwise repeat BTDA for tree(v), the right subtree of \( u \).

---

+ Note that at this point tree(u) is no longer a binary search tree since key(z) occurs twice.
3.2.4 **IMPLEMENTATION**

The implementation of these algorithms conforms exactly to the general implementation details of 3.1.2 except that a header node is introduced. The right pointer field of this node will always point to the root of the tree. This special node is used for convenience particularly in the deletion algorithm.

A binary search tree of 7 nodes for the input sequence

15, 25, 10, 5, 30, 20, 35

**Figure 3.2.1**
3.3 THE AVL TREE ALGORITHMS

3.3.1 INTRODUCTION

AVL trees derive their name from two Russian mathematicians Adel'son-Vel'skii and Landis, who discovered this tree search algorithm. AVL trees represent a compromise between balanced trees and arbitrary trees. The scheme attempts to avoid excessively long or short search paths. Just how well the algorithm achieves this goal will be examined later, but suffice to say that from this point of view the results are very good. To this end the algorithm dynamically restructures the tree to keep it "AVL balanced".

AVL trees possess the following property, the AVL property, which is dependent on the height of a tree, tree(u).

**Definition**

A tree, tree(u), is an **AVL tree** if for all nodes w in tree(u) the height of the left subtree of w differs from the height of the right subtree of w by at most one. Any tree fulfilling this property will be referred to as **AVL balanced**. The balance factor of any node w in tree(u), balfac(w), is defined as the height of the right subtree of w minus the height of the left subtree of w. In succeeding diagrams the balance factor is indicated by an integer directly below each subtree node. If in an AVL tree, the balance factor of any node is 1, -1 or 0, the node is said to be **right heavy**, **left heavy** or **balanced** respectively (see Figure 3.3.1).

Immediately after insertion or deletion one or more nodes may
lose the AVL property. Assume a node \( w \) has balance factor 1. On insertion tree(\( w \)) may be AVL unbalanced when a node is inserted into the right subtree of \( w \) or on deletion, when a node is deleted from the left subtree of \( w \) (see Figure 3.3.2). A symmetrical case exists if \( \text{balfac}(w) = -1 \). The given general transformations restore AVL balance while preserving the relationship of the keys. Figure 3.1.1 shows the only two cases which arise for insertion (and the two most likely for deletion) with the corresponding rotations on the tree nodes necessary to restore AVL balance. Symmetrical cases exist. Note that the height of the newly rotated subtree is the same as the original subtree before insertion of the new key. Hence for insertion only one rotation is necessary since the rest of the tree (if any) remains AVL balanced. However, in the case of deletion up to \( \lfloor \log_2 n \rfloor - 1 \) rebalances may be necessary as indicated in Figure 3.3.3. This is because deletion has caused the height of a subtree to be decremented.

3.3.2 THE AVL TREE INSERTION ALGORITHM (AVLIA)

We assume that we are given tree(\( u \)) which is AVL balanced, and that the key to be inserted is \( X \).

(i) If \( u = \emptyset \) then enter \( X \) as the key of the root node and stop.

(ii) Otherwise assume that \( X \) is to be inserted as with BTIA, then a sequence of nodes will be traced out by the BTIA. Let this insertion sequence be \( u_1, \ldots, u_k \) where \( u_1 = u \) and \( k \geq 1 \). If key(\( u_k \)) = \( X \), then stop since the key is already present in the tree.

(iii) Locate the critical node, that is find the maximum \( l \), \( 1 \leq l < k \), such that \( \text{balfac}(u_l) \neq 0 \); if there is no such node set \( l \) to 1.
(iv) Insert X as with the BTIA.

(v) Determine if rebalancing is required. Three cases arise:

(a) \( \text{balfac}(u_i) = 1 \) or \(-1\).

The tree has grown higher and no rebalancing is required so stop (see Figure 3.3.4). This case arises when \( i = 1 \) in step (iii).

(b) \( \text{balfac}(u_i) = 0 \).

The tree has become more balanced so stop (see Figure 3.3.5).

(c) \( \text{balfac}(u_i) = 2 \) or \(-2\).

The tree has become AVL unbalanced and a rotation must be performed.

(vi) Determine which rotation to perform. Two cases result:

(a) (1) \( \text{balfac}(u_i) = 2 \) and \( \text{balfac}(u_{i+1}) = 1 \) or
(2) \( \text{balfac}(u_i) = -2 \) and \( \text{balfac}(u_{i+1}) = -1 \).

A single rotation is invoked (see Figure 3.3.6).

(b) (1) \( \text{balfac}(u_i) = 2 \) and \( \text{balfac}(u_{i+1}) = -1 \) or
(2) \( \text{balfac}(u_i) = -2 \) and \( \text{balfac}(u_{i+1}) = 1 \).

A double rotation is invoked (see Figure 3.3.7).

3.3.3 THE AVL TREE DELETION ALGORITHM (AVLDA)

We assume we are given tree \((u)\) which is AVL balanced, and that the key to be deleted is \( X \).

(1) Assume that \( X \) is to be inserted with the BTIA, then a sequence of nodes will be traced out by the BTIA. Let this sequence, the deletion sequence, be \( u_1, \ldots, u_k \), where \( u_1 = u \) and \( X = \text{key}(u_k) \) (see Figure 3.3.8).
(ii) Three cases arise.

(a) $u_k$ has no sons.

Remove $u_k$ from the tree (see Figure 3.3.9).

(b) $u_k$ has one son.

Remove $u_k$ and relink the son of $u_k$ to $u_{k-1}$ (see Figure 3.3.10).

(c) $u_k$ has two sons.

If $\text{balfac}(u_k) = 1$, let $z$ be the postorder successor of $u_k$. Otherwise, let $z$ be the postorder predecessor of $u_k$. Replace $\text{key}(u_k)$ with $\text{key}(z)$. A new deletion sequence is built up where $u_1 = u$ and $u_k = z$. Repeat step (ii) (see Figure 3.3.11).

(iii) Examine the deletion sequence. Let $\ell = k-1$ (initially). If $\ell = 0$ stop, otherwise three cases arise.

(a) $\text{balfac}(u_{\ell}) = 0$.

The height of tree($u_{\ell}$) has been decremented. Repeat step (iii) with $\ell = \ell - 1$ (see Figure 3.3.12).

(b) $\text{balfac}(u_{\ell}) = 1$ or $-1$.

The height of tree($u_{\ell}$) is not reduced and the rest of the tree will be unaffected, so stop (see Figure 3.3.13).

(c) $\text{balfac}(u_{\ell}) = 2$ or $-2$.

Rebalancing is necessary (see Figure 3.3.14).

(iv) Determine which rotation to perform. Let $v$ and $u_{\ell+1}$ be the sons of $u_{\ell}$. Three cases result:

(a) (1) $\text{balfac}(u_{\ell}) = 2$ and $\text{balfac}(v) = 1$ or

(2) $\text{balfac}(u_{\ell}) = -2$ and $\text{balfac}(v) = -1$.

A single rotation is performed as illustrated in Figure 3.3.15.
Let \( \ell = \ell-1 \) and repeat step (iii).

(b) (1) \( \text{balfac}(u_x) = 2 \) and \( \text{balfac}(v) = -1 \) or
(2) \( \text{balfac}(u_x) = -2 \) and \( \text{balfac}(v) = 1 \).

A double rotation is performed as illustrated in Figure 3.3.16.

Let \( \ell = \ell-1 \) and repeat step (iii).

(c) \( \text{balfac}(u_x) = 2 \) or \(-2 \) and \( \text{balfac}(v) = 0 \).

A single rotation is performed and the algorithm terminates since the height of the rotated subtree has not decreased (see Figure 3.3.17).

3.3.4 IMPLEMENTATION

As with the basic tree algorithms a header node, whose right pointer field points to the root of the tree, is utilized. It is also useful to have \( \text{balfac}(w) \), for each node \( w \) in \( \text{tree}(u) \), available without having to compute it. To this end the balance factor is an additional field of information in any record representing a node. On insertion only pointers to the critical node, \( v \) say, its father (used for relinking a rotated subtree) and the inserted node, \( p \) say, need be saved. Only the balance factors between \( v \) and \( p \) need be adjusted. Examination of the balance factor of \( v \) and whether \( p \) is in its left or right subtree determine whether or not a rotation need be performed. At most the balance factors of three nodes need readjusting when a rotation is performed as indicated in Figure 3.1.1. On deletion a stack of pointers must be built to the deleted node. Along with these pointers it is convenient to have the direction to the next node in the search path (i.e. \( 1 = \text{right}, -1 = \text{left} \)). The stack could be \( (P_0, a_0), (P_1, a_1), \ldots, (P_m, a_m) \) where \( P_0 \) is the pointer
to the header node, $a_0 = 1$, $P_i$, $1 \leq i < m$, is the pointer to the $i$-th node in the search path, $a_i$ is the direction to move from this node to get to the next, and $P_m$ is the pointer to the node to be deleted. The balance factors of the nodes in the deletion sequence $u_{\ell}$, $1 < m$, are adjusted as each is examined. Step (iii) of AVLDA now may be interpreted as:

(a) $balfac(u_{\ell}) = a_{\ell}$.

$u_{\ell}$ was left or right heavy and a node was deleted from its left or right subtree respectively.

(b) $balfac(u_{\ell}) = 0$.

$u_{\ell}$ was balanced and a node was deleted from one of its subtrees. Set $balfac(u_{\ell}) = -a_{\ell}$ and terminate.

(c) $balfac(u_{\ell}) = -a_{\ell}$.

$u_{\ell}$ was left or right heavy and a node was deleted from its right or left subtree respectively. Rebalancing is necessary.

On deletion rebalancing is identical to insertion with exception of the special case illustrated in Figure 3.3.17.
An AVL tree

Nodes with keys 15 and 35 are right heavy. Nodes with keys 10 and 25 are left heavy. Node with key 30 is balanced.

A non-AVL tree

Figure 3.3.1
key \( w = 10 \)

Insertion of node with key 20 causes unbalance.

key \( w = 10 \)

Deletion of node with key 5 causes unbalance.

key \( w = 20 \).

Deletion of node with key 5 does not cause unbalance.

Figure 3.3.2
Deletion of node with key 60 causes \( \lfloor \log_2 12 \rfloor - 1 = 2 \) rebalances.

**Figure 3.3.3**

\( i = 1, \text{key}(u_1) = 10 \text{ balfac}(u_1) = 0 \). Insertion of \( X \) causes the height of the tree to be increased.

**Figure 3.3.4**
$i = 3$, $\text{key}(u_i) = 40$, $\text{balfac}(u_i) = 0$. Insertion of $X$ causes tree to become more balanced.
i = 3, key(u_i) = 30, balfac(u_i) = 2, balfac(u_{i+1}) = 1.

A single rotation restores the balance.
$i = 2$, $\text{key}(u_i) = 25$, $\text{balfac}(u_i) = 2$, $\text{balfac}(u_{i+1}) = -1$.

The insertion of $X$ causes a double rotation on nodes with keys 25, 30 and $X$. 

Figure 3.3.7
$k = 4$, $\text{key}(u_1) = 40$, $\text{key}(u_2) = 20$, $\text{key}(u_3) = 30$, $\text{key}(u_4) = x$

$balfac(u_1) = 0$, $balfac(u_2) = 1$, $balfac(u_3) = -1$,

$balfac(u_4) = 0$.

Figure 3.3.8
$k = 4, \text{key}(u_k) = 35$

**Figure 3.3.9**

$k = 3, \text{key}(u_k) = 30$

**Figure 3.3.10**
$k = 2$, $\text{key}(u_k) = 35$, $\text{balfac}(u_k) = -1$, $\text{key}(z) = 30$, $z$ is the postorder predecessor of $u_k$.

$k = 4$, $\text{key}(u_1) = 15$, $\text{key}(u_2) = 30$, $\text{key}(u_3) = 25$, $\text{key}(u_4) = 30$.

Figure 3.3.11
$k = 4$, $\text{key}(u_k) = 35$, $\ell = 3$, $\text{key}(u_\ell) = 30$, $\text{balfac}(u_\ell) = 0$. The height of tree $u_\ell$ has been decremented.
$k = 3$, $\text{key}(u_k) = 30$, $l = 2$, $\text{key}(u_l) = 25$, $\text{balfac}(u_l) = -1$. The height of $\text{tree}(u_l)$ is not reduced.

**Figure 3.3.13**

$k = 3$, $\text{key}(u_k) = 20$, $l = 2$, $\text{key}(u_l) = 25$, $\text{balfac}(u_l) = 2$. Rebalancing necessary.

**Figure 3.3.14**
$k = 3$, $\text{key}(u_k) = 20$, $l = 2$, $\text{key}(u_{l+1}) = 25$, $u_{l+1} = \emptyset$, $\text{key}(v) = 30$

$\text{balfac}(u_{l+1}) = 2$, $\text{balfac}(v) = 1$. A single rotation is performed.

\textbf{Figure 3.3.15}
$k = 3, \text{key}(u_k) = 20, \ell = 2, \text{key}(u_{\ell}) = 25$

$u_{k+1} = \emptyset, \text{key}(v) = 35, \text{balfac}(u_{\ell}) = 2, \text{balfac}(v) = -1$. A double rotation is performed.

Figure 3.3.16
$k = 3$, $\text{key}(u_k) = 20$, $\ell = 2$, $\text{key}(u_{k+1}) = 25$, $u_{k+1} = \emptyset$, $\text{key}(v) = 35$,

$balfac(u_k) = 2$, $balfac(v) = 0$. A single rotation rebalances the tree; the rotated subtree has not decreased in height.
3.4 THE BOUNDED BALANCE TREE ALGORITHMS

3.4.1. INTRODUCTION

Trees of bounded balance or BB trees are a class of unweighted binary search trees which are easy to maintain despite frequent insertions or deletions. In this respect they are similar to the AVL trees discussed in Section 3.3. They differ in one very important aspect and that is they contain a parameter which can be chosen arbitrarily so as to compromise between a shorter search time and infrequent restructuring.

**Definition**

Given tree(u), u ≠ ∅, then the root-balance of any node v in tree(u), RB(v), is defined as $RB(v) = \frac{size(v_l) + 1}{size(v) + 1}$. A tree, tree(u), is said to be of bounded balance $\alpha$ or in BB[$\alpha$], $0 < \alpha \leq \frac{1}{2}$, if and only if for all v in tree(u) either size(v) = 1 or size(v) > 1 implies (1) $\alpha \leq RB(v) \leq 1-\alpha$ and (2) tree(v_l) and tree(v_r) are in BB[$\alpha$].

The root balance, hereafter referred to as the balance, is an indicator of the relative number of nodes in the left and right subtrees of u; a tree is perfectly balanced if it is in BB[1/2]. In the illustrations, the balance is indicated by a rational number below each subtree node. Figure 3.4.1 shows two examples of BB trees.

It is interesting to note that there is a gap in the balance of trees as shown in Nievergelt and Reingold (1972).
Theorem 3.4.1

For all $\alpha$ in the range $\frac{1}{3} < \alpha < \frac{1}{2}$, $BB[\alpha] = BB[\frac{1}{2}]$.

Insertion or deletion may cause the tree to lose its bounded balance. Here this means $RB(v)$, for some node(s) $v$ in tree($u$), falls outside the closed interval $[\alpha, 1-\alpha]$. For example if $RB(v) = \alpha$ then the insertion of a key into tree($v_r$) or the deletion of a node in tree($v_r$) causes unbalance (see Figure 3.4.2). When $RB(v) = 1-\alpha$, similar cases exist. If $RB(v)$ falls out of range the general transformations of Section 3.1.1 are invoked which restore the balance. The transformations are the same as those in the AVL case but their use is restricted to certain values of $\alpha$.

Theorem 3.4.2

If $0 < \alpha < 1 - \frac{\sqrt{2}}{2}$ and the addition of a node $w$ to tree($u$), which is in $BB[\alpha]$, causes a node, $v$ say, to become unbalanced, then the following transformations restore balance to $v$.

(a) If $w$ is in tree($v_r$) then if $RB(v_r) < \frac{1-2\alpha}{1-\alpha}$, a single rotation restores balance, otherwise a double rotation is needed.

(b) If $w$ is in tree($v_g$) then if $1-RB(v_g) < \frac{1-2\alpha}{1-\alpha}$, a single rotation restores balance, otherwise a double rotation is needed.

The proof of this theorem can be found in Nievergelt and Reingold (1971). The case of deletion of a node is inherent in the above. For both insertion and deletion more than one rotation may be required to maintain balance (see Figures 3.4.3 and 3.4.4). Other transformations may be introduced so that $\alpha$ may be increased but since
(a) \( 1 - \frac{\sqrt{2}}{2} = .2928 \), (b) BB[a] - BB[\frac{1}{2}] is empty for \( \frac{1}{3} < a < \frac{1}{2} \), (c) the given transformations are simple and (d) Nievergelt and Reingold have shown that the average search time of trees in BB[1 - \frac{\sqrt{2}}{2}] is no worse then 15% longer than a completely balanced tree, it is reasonable to choose \( \alpha \) in the range \( 0 < \alpha \leq 1 - \frac{\sqrt{2}}{2} \).

3.4.2. THE BOUNDED BALANCE INSERTION ALGORITHM (BBIA)

We assume that we are given tree(u) which is in BB[\alpha], the parameter \( \alpha \), \( 0 < \alpha \leq 1 - \frac{\sqrt{2}}{2} \) and that the key to be inserted is \( X \).

(i) If \( u = \emptyset \) then enter \( X \) as the key of the root node and stop.

(ii) Insert \( X \) as with the BTIA.

(iii) Examine the nodes in the search path to the new node. Let \( v \) denote the node currently under examination; initially \( v = u \). If key(\( v \)) = \( X \) stop, otherwise two cases exist.

(a) \( \alpha \leq RB(v) \leq 1 - \alpha \).

\( v \) has not become unbalanced with the insertion of \( X \).

Repeat step (iii) for the next node in the search path (see Figure 3.4.5).

(b) \( RB(v) > 1 - \alpha \) or \( RB(v) < \alpha \).

\( v \) has become unbalanced with the insertion of \( X \).

A rotation must be performed.

(iv) Determine which rotation to perform. Two cases result.

(a) \( X \) is in tree(\( v_r \)). If \( \frac{RB(v_r)}{1 - \alpha} < \frac{1 - 2\alpha}{1 - \alpha} \), then a single rotation is performed, otherwise a double
rotation is used (see Figure 3.4.6).

(b) \( X \) is in tree\((v')_x \). If \( 1 - \text{RB}(v')_x < \frac{1 - 2\alpha}{1 - \delta} \), then a single rotation is performed, otherwise a double rotation is used (see Figure 3.4.7).

Let \( v \) be the root of the rotated subtree and repeat step (iii).

3.4.3. THE BOUNDED BALANCE DELETION ALGORITHM (BBDA)

We assume that we are given tree\((u)\) which is in BB\([\alpha]\), the parameter \( \alpha \), \( 0 < \alpha \leq 1 - \frac{\sqrt{2}}{2} \), and that the key to be deleted is \( X \).

(i) Assume \( X \) is to be inserted as with the BTIA, then a sequence of nodes will be traced out by the BTIA. Let this sequence, the deletion sequence, be \( u_1, \ldots, u_k \) where \( u_1 = u \) and \( X = \text{key}(u_k) \).

(ii) Examine \( u_k \). Three cases result.

(a) \( u_k \) has no sons.

Remove \( u_k \) from the tree.

(b) \( u_k \) has one son.

Remove \( u_k \) and relink the son of \( u_k \) to \( u_{k-1} \).

(c) \( u_k \) has two sons.

If \( \text{RB}(u_k) = \frac{1}{2} \) let \( z \) be the postorder successor of \( u_k \), otherwise let \( z \) be the postorder predecessor or successor of \( u_k \) depending on which will most improve the balance of \( u_k \) if \( \text{tree}(z) \) is replaced by \( \text{tree}(v) \), where \( v \) is the son of \( z \) (if any). Figure 3.4.8
illustrates the situation. Replace key(u_k) with key(z) and repeat step (ii) with u_k = z and X = key(z).

During the search for z a new deletion sequence is built up where u_1 = u and u_k = z.

Let i = 1.

(iii) Examine the deletion sequence. If i = k stop, otherwise two cases result.

(a) \( \alpha \cdot \leq \text{RB}(u_1) \leq 1 - \alpha \).

\( u_1 \) has not become unbalanced with the deletion of the node with key X (see Figure 3.4.9). Let i = i + 1 and repeat step (iii).

(b) \( \text{RB}(u_1) < \alpha \) or \( \text{RB}(u_1) > 1 - \alpha \).

\( u_1 \) has become unbalanced; perform step (iv).

(iv) Determine which rotation to perform. Two cases arise.

(a) If \( u_k \) was in the left subtree of \( u_1 \) then let \( v \) be the right son of \( u_1 \). If \( \text{RB}(v) < \frac{1-2\alpha}{1-\alpha} \), then a single rotation will restore the balance, otherwise a double rotation is used (see Figure 3.4.10).

(b) If \( u_k \) was in the right subtree of \( u_1 \) then let \( v \) be the left son of \( u_1 \). If \( 1 - \text{RB}(v) < \frac{1-2\alpha}{1-\alpha} \), then a single rotation will restore balance, otherwise a double rotation is used (see Figure 3.4.11).

Let i = i + 1 and repeat step (iii).

\[ + \] Note that tree(u) is not a binary search tree since key(z) appears twice.
3.4.4. IMPLEMENTATION

The implementation makes use of a special header node whose right pointer field points to the root of the tree. There is also an additional field contained in any record representing a node, the size field. This gives size(u) for any u. This is all the information which is needed to compute RB(u). When a key, X, is to be inserted into the tree, the size field of each node, u, in the insertion sequence is incremented and checked to see if it will be unbalanced upon the insertion of X. If so, a rotation is performed before the key is actually inserted. The rotations are determined as indicated by Theorem 3.4.2: If however,
\[ \frac{1}{4} < \alpha < 1 - \frac{\sqrt{2}}{2} \]
then difficulties arise when size(u) = 2. Referring to Figure 3.4.12, a double rotation should be performed according to the specifications of the theorem, but this is impossible since X has not yet been inserted. Therefore when size(u) = 2 and a rotation must be invoked, a single rotation is used. Still another problem exists as indicated by Figure 3.4.13. Here an infinite loop results when

1. key(u) < X < key(u+1) or 2. key(u+1) < X < key(u).

A special test is made for condition (1) and the tree is transformed as in the figure. Another special test is included for an initial examination of u. If \[ \frac{1}{size(u) + 2} > \alpha \] then tree(u) will be in BB[u] after the insertion and therefore the key may be simply inserted in tree(u) as with BTIA, remembering to increment size fields. If X is already present in the tree, the insertion sequence is scanned again and the size fields decremented but no restructuring takes place since any rotation which had occurred improved the balance.
The deletion case is similar. For any \( u \), if a node is to be deleted from \( \text{tree}(u) \) then this may be treated as an insertion of a key into \( \text{tree}(u) \) and appropriate rotations invoked if necessary before any actual deletion. Here the size fields of the nodes in the deletion sequence are decremented as each is examined and if the node to be deleted is not present, they must be incremented on a second pass but no restructuring is necessary.

The inequalities involving fractions are changed to an integer form for the different comparisons needed. For example,

if \( a = \frac{a}{b} \) then \( a \leq \text{RB}(v) \leq 1-a \) is \( a \leq \frac{\text{size}(v) + 1}{\text{size}(v) + 1} \leq 1 - \frac{a}{b} \) which becomes \( a \cdot [\text{size}(v) + 1] \leq b \cdot [\text{size}(v) + 1] \leq (b-a) \cdot [\text{size}(v) + 1] \).

This avoids division and comparison of real quantities which may cause difficulty due to rounding error.
A tree in $BB[\frac{1}{4}]$
\( \alpha = \frac{1}{6}, \text{key}(v) = 10, \text{key}(v_r) = 15. \) Addition of node with key \( X \) in \( t(v_r) \) causes unbalance. \( \text{RB}(v) = \frac{2}{13} < \alpha. \)

\( \alpha = \frac{1}{6}, \text{key}(v) = 10, \text{key}(v_g) = 5. \) Deletion of node with key 5 causes unbalance. \( \text{RB}(v) = \frac{1}{11} < \alpha. \)

**Figure 3.4.2**
\[ a = \frac{1}{5} \]. Insertion of \( X \) requires two single rotations to restore balance.

Nodes with keys 10 and 30 are unbalanced upon insertion of \( X \).

Figure 3.4.3
\[ \alpha = \frac{1}{4} \]

Deletion of node with key X requires two single rotations to restore balance. Nodes with keys 25 and 5 are unbalanced.
\( \alpha = \frac{1}{7} \), key(v) = 10. Insertion of X does not cause unbalance of v.
\( \alpha = \frac{2}{7}, \text{key}(v) = 10, \text{key}(v_r) = 20. \) Insertion of \( X \) causes unbalance at \( v \), \( \text{RR}(v) < \alpha, \text{RB}(v_r) < \frac{1-2\alpha}{1-\alpha} \) causing a single rotation.

\( \alpha = \frac{2}{7}, \text{key}(v) = 10, \text{key}(v_r) = 25. \) Insertion of \( X \) causes unbalance at \( v \), \( \text{RB}(v) < \alpha, \text{RB}(v_r) \neq \frac{1-2\alpha}{1-\alpha} \) causing a double rotation.

**Figure 3.4.6**
\[ \alpha = \frac{2}{7}, \text{key}(v) = 25, \text{key}(v_x) = 10. \] Insertion of X causes unbalance at v, 
\[ \text{RB}(v) > 1 - \alpha, 1 - \text{RB}(v_x) < \frac{1-2\alpha}{1-\alpha} \] causing a double rotation.

\[ \alpha = \frac{2}{7}, \text{key}(v) = 25, \text{key}(v_x) = 15. \] Insertion of X causes unbalance at v, 
\[ \text{RB}(v) > 1 - \alpha, 1 - \text{RB}(v_x) < \frac{1-2\alpha}{1-\alpha} \] causing a single rotation.

\textbf{Figure 3.4.7}
\[ \alpha = \frac{1}{4}, \text{key}(u_k) = 15, \text{key}(z) = 20. \] Choosing \( z \) as the postorder successor improves the balance of \( u_k \) when \( z \) is deleted.
\[ \alpha = \frac{1}{4}, \text{key}(u_k) = X, \text{key}(u_\downarrow) = 20. \] Deletion of \( u_k \) does not cause imbalance at \( u_\downarrow \).

\[ \text{Figure 3.4.9} \]
\[ \alpha = \frac{1}{4}, \; \text{key}(u_k) = 5, \; \text{key}(u_1) = 10, \; \text{key}(v) = 15, \; \text{RB}(u_1) < \alpha, \; \text{RB}(v) < \frac{1-2\alpha}{1-\alpha}. \]

A single rotation restores balance.

\[ \alpha = \frac{1}{4}, \; \text{key}(u_k) = 5, \; \text{key}(u_1) = 10, \; \text{key}(v) = 25, \; \text{RB}(u_1) < \alpha, \; \text{RB}(v) < \frac{1-2\alpha}{1-\alpha}. \]

A double rotation restores balance.
\[ \alpha = \frac{1}{4}, \text{key}(u_k) = 25, \text{key}(u_1) = 20, \text{key}(v) = 5, \text{RB}(u_1) > 1-\alpha, \]

\[ 1 - \text{RB}(v) < \frac{1-2\alpha}{1-\alpha}. \] A double rotation restores balance.

\[ \alpha = \frac{1}{4}, \text{key}(u_k) = 25, \text{key}(u_1) = 20, \text{key}(v) = 15, \text{RB}(u_1) > 1-\alpha, \]

\[ 1 - \text{RB}(v) < \frac{1-2\alpha}{1-\alpha}. \] A single rotation restores balance.

**Figure 3.4.11**
\[ \frac{1}{4} < \alpha < 1 - \frac{\sqrt{2}}{2}, \quad \text{key}(u_1) = 5, \quad \text{key}(u_{i+1}) = 10, \quad \text{size}(u_1) = 2, \]

\[ \text{RB}(u_1) = \frac{1}{4} < \alpha, \quad \text{RB}(u_{i+1}) \neq \frac{1-2\alpha}{1-\alpha}. \]

A double rotation cannot be performed since \( X \) is not in \( \text{tree}(u_1) \) yet.

\textbf{Figure 3.4.12}
\[ \frac{1}{4} < a < 1 - \frac{\sqrt{2}}{2}, \quad \text{key}(u_1) = 5, 10, \ldots, \text{size}(u_1) \neq 2. \] An infinite number of single rotations result.

A special case for the implemented algorithm
3.5 THE BELL-TREE ALGORITHMS

3.5.1. INTRODUCTION

This unweighted binary tree construction method was first discussed informally by Bell (1965). In general whenever a sequence of keys is fed to the BTIA, (Section 3.2), some subsequences will produce a maximum valued subtree, e.g. if the complete sequence of keys is in preceding order.

Bell observed this and developed a technique to remedy this situation by "forcing" what could be a maximum valued subtree to have a lower value by means of transformations. Bell did not give an explicit algorithm for this technique, however the algorithms described below are based on his work. It is necessary to introduce some new notation.

Definition

Given a tree, tree(u), then for \( r > 0 \)

\[
\text{r-size}(u) = \begin{cases} 
\text{size}(u), & \text{if size}(u) < 2^r - 1. \\
2^r - 1, & \text{if size}(u) \geq 2^r - 1.
\end{cases}
\]

In succeeding diagrams r-size(u) will be indicated below u. If tree(u) is minimal then for \( r > 0 \), tree(u) is (i) r-complete if \( \text{size}(u) = 2^r - 1 \), (ii) r-incomplete if \( \text{size}(u) < 2^r - 1 \) and (iii) r-replete if \( \text{size}(u) > 2^r - 1 \).

For \( r > 0 \), r-tree(u) is a subset of tree(u) such that for v in tree(u), v is in r-tree(u) iff \( l(v) < r \). For \( k > 0 \), tree(u) is a k-minimal tree iff for all v in tree(u) either (i) \( \text{size}(v) < 2^k - 1 \) and tree(v) is minimal or (ii) \( \text{size}(v) \geq 2^k \) and k-tree(v) is k-complete. This definition corrects an error in Bell (1965). Each of the above notions is illustrated in Figure 3.5.1.
The following tree construction algorithm builds \( k \)-minimal trees at each stage of the construction for some parameter \( k > 0 \). When \( k = 1 \) the insertion and deletion algorithms become the BTIA and BTDA of Section 3.2. The transformations used to produce and maintain \( k \)-minimal trees in both algorithms are the general transformations given in Section 3.1.1. Observing that subtrees of \( k \)-minimal trees are \((k-1)\)-minimal trees, the technique is to build \( k \)-minimal trees from \((k-1)\)-minimal trees. When a node, \( u \), becomes the root of a \( k \)-minimal tree, for a given parameter \( k \), and \( \text{size}(u) \geq 2^k \), \( u \) remains static, i.e. it is never rotated by any transformation; its two sons, \( v \) and \( y \) say, must now become roots of \( k \)-minimal trees (where \( \text{size}(v) \geq 2^k \), \( \text{size}(y) \geq 2^k \)) upon the insertion of more keys into the tree. The same idea is inherent in the deletion algorithm.

3.5.2. **The Bell-Tree Insertion Algorithm (BIA)**

Assume that \( k > 1 \), that we are given tree\( (u) \), a \( k \)-minimal tree, and the key to be inserted is \( X \).

(i) If \( u = \emptyset \) then enter \( X \) as the root node and stop.

(ii) If \( \text{size}(u) < 2^{k-1} \), then \( k' = \left\lfloor \log_2 (1+\text{size}(u)) \right\rfloor \), otherwise \( k' = k \). This ensures that initially \( p \)-minimal trees are built for \( p = 1, 2, \ldots, k \).

(iii) Assume that \( X \) is to be inserted with the BTIA, then a sequence of nodes will be traced out by the BTIA. If the key of any node equals \( X \) stop, otherwise let the first node in this sequence, which is not the root of a \( k' \)-complete tree, be \( u_0 \) and the succeeding nodes be
u_1, \ldots, u_m, \quad m \leq k-1.

(iv) If m = 0 then X is entered as a son of u_0.

(v) For i = 1, \ldots, m-1, if tree(u_i) is a (k'-i)-complete tree, then letting u_i be the first such node, do the following: if u_i is a left son of u_{i-1} then let z be the postorder predecessor of u_{i-1}, otherwise let z be the postorder successor of u_{i-1}. We deal with the predecessor case only, the successor case follows similarly. Two cases arise (see Figures 3.5.2 and 3.5.3 respectively).

(a) X > key(z). Replace key(u_{i-1}) (=Y; say) by X and then insert Y into tree(u_{i-1}) using the BIA. Note that the right son of u_{i-1}, v say, is such that tree(v) is (k'-i)-incomplete.

(b) X < key(z). Remove z from tree(u) and insert X into tree(u_i). Replace key(u_{i-1}) (=Y, say) with key(z) and insert Y into tree(u_{i-1}) using the BIA.

(vi) If there is no u_i fulfilling the conditions of step (v) then two cases arise (see Figures 3.5.4 and 3.5.5 respectively).

(a) size(u_m) = 1. If tree(u_{m-1}) is not a 2-complete tree then carry out a rotation, otherwise X is entered as a son of u_m.

(b) size(u_m) = 2. X is entered as a son of u_m.
3.5.3 THE BELL-TREE DELETION ALGORITHM (BDA)

Assume that \( k > 1 \), that we are given \( \text{tree}(u) \), a \( k \)-minimal tree, \( u \neq \emptyset \), and the key to be deleted is \( X \).

(i) Assume that \( X \) is to be inserted with the BTIA, then a sequence of nodes will be traced out by the BTIA. Let the last \( k' \) nodes in this sequence, the deletion sequence, be \( u_1, \ldots, u_{k'} \), where \( X = \text{key}(u_{k'}) \), and \( k' = k \) if \( \ell(u_{k'}) \geq k \) or \( k' = \ell(u_{k'}) \) otherwise. See Figure 3.5.6 for \( k = 4 \).

(ii) Three cases arise.

(a) \( u_{k'} \) has exactly one son, \( v \).

Remove \( u_{k'} \) and relink \( v \) to \( u_{k' - 1} \) (see Figure 3.5.7), unless \( u = u_{k'} \), in which case \( v \) becomes the new root.

(b) \( u_{k'} \) has two sons.

Let \( v_L \) and \( v_R \) be the left and right sons of \( u_{k'} \), respectively. If \( k\)-size\( (v_L) > k\)-size\( (v_R) \) then let \( z \) be the postorder predecessor of \( u_{k'} \), otherwise let \( z \) be the postorder successor of \( u_{k'} \). A new deletion sequence \( u_1', \ldots, u_{k'}' \), is calculated where \( u_{k'}' = z \). Replace \( X \) with \( \text{key}(z) \). Step (ii) is now repeated with \( u_1, \ldots, u_{k'} \) replaced by \( u_1', \ldots, u_{k'}' \) (see Figure 3.5.8).

(c) \( u_{k'} \) has no sons.

If \( u = u_{k'} \), then a null tree results, otherwise the deletion sequence is used to determine what action.

+ Note that at this point in the algorithm \( \text{tree}(u) \) is no longer a binary search tree, since \( \text{key}(z) \) occurs twice.
should be taken. For \( i = 1, \ldots, k'-1 \), if there is a \( u'_{k'-1} \) such that

(1) \( \text{tree}(u'_{k'-1}) \) is \((i+1)\)-incomplete, then immediately we have that \( \text{tree}(u'_1) \) is \( k \)-incomplete; thus \( u'_k \) is removed (see Figure 3.5.9), or

(2) \( \text{tree}(u'_{k'-1}) \) is \((i+1)\)-replete; then letting \( u'_{k'-1+i} \) and \( v \) be the sons of \( u'_{k'-1} \), if \( v \) is the left son let \( z \) be the postorder predecessor of \( \text{key}(u'_{k'-1}) \); otherwise let \( z \) be the postorder successor of \( \text{key}(u'_{k'-1}) \). A new deletion sequence \( u'_1, \ldots, u'_k \) is calculated where \( u'_k = z \). Replace \( \text{key}(u'_{k'-1}) \) with \( \text{key}(z) \) and then use the BIA to insert \( \text{key}(u'_{k'-1}) \) into \( \text{tree}(u'_{k'-1+i}) \). Step (ii) is now repeated with \( u'_1, \ldots, u'_k \), replaced by \( u'_1, \ldots, u'_k' \) (see Figure 3.5.10).

Otherwise if there is no \( u'_{k'-1} \) fulfilling conditions (1) or (2) then remove \( u_k \), (see Figure 3.5.11).

3.5.4 IMPLEMENTATION

A special header node is introduced whose right pointer field points to the root of the tree. For an input parameter \( k \), the implemented insertion algorithm also makes use of the value \( 2^k-1 \), called \( \text{KVAL} \) in the program, to determine its primary actions. An additional field of information is contained in any record representing a node, the size field. This field gives the \( k \)-size of any node \( v \) in \( \text{tree}(u) \). This information is used to determine if for any \( v \), \( \text{tree}(v) \) is \( k \)-complete,
\( l \)-replete or \( l \)-incomplete; \( 1 \leq l \leq k \). In the insertion algorithm \( u_0 \) is determined by simply examining the size field of each node in the insertion sequence until \( \text{size}(v) < K\text{VAL} \), for some \( v \). There is an initial examination of the insertion sequence to determine if the key to be inserted is already present in the tree, \( u_0 \) being located at this time as well. This initial search can be avoided if the key of each node in the insertion sequence is checked to see if it is the key being inserted and if a transformation needs to be performed at this point in the sequence. Any transformation carried out does not destroy the properties of the tree but since the key must still be located (meaning its original search path must be found) the added work and complexity may not be worth the effort. If the key is not present each node \( u_i \) succeeding \( u_0 \) in the insertion sequence is examined to see if \( \text{size}(u_i) = \left\lfloor \frac{\text{KVAL}}{2^i} \right\rfloor \), \( i > 1 \), and if so step (v) is carried out. If \( \left\lfloor \frac{\text{KVAL}}{2^i} \right\rfloor = 1 \) then this corresponds to step (vi)(a) and a simple transformation must be invoked. Step (vi)(b) is recognized when \( u_i = \emptyset \) (actually \( i = m+1 \) in BIA at this point).

For the deletion case a circular stack of length \( k+1 \) is used to contain pointers to the last \( k+1 \) nodes in the deletion sequence (stack[\( k+1 \)] points to the deleted node, stack[\( k \)] to its father, stack[\( k-1 \)] to its grandfather, ... etc). With a little thought it is seen that the stack need only be of length \( k \). The extra pointer was found convenient when deleting the last two nodes in a tree; at this point stack[1] points to the header node. Again the size field of the nodes is used to determine any action that must be taken. The size field of the node to be deleted is investigated to determine if it has
one, two or zero sons. If the node has one or two sons step (ii)(a) or (b) is carried out; otherwise using the pointers contained in the stack step (ii)(c) is performed.

\[ k = 3 \]

(a) Subtrees with keys 25 and 45 are 3-complete.
(b) Subtrees with keys 10 and 65 are 1-replete and 2-replete but 3-incomplete.
(c) Subtrees with keys 35 and 55 are 2-complete but 3-incomplete.
(d) The tree is a 3-minimal tree.

Figure 3.5.1
BIA for 18; i=1, key(u_0)=20, key(u_1)=10, key(z)=15, and X=18, and tree(u_1) is a 2-complete tree.

After the transformation key(u_0)=18, and BIA is called recursively to insert 20 into tree(u_0).
BIA for X, i=1, key(u₀) = 20, key(u₁) = 10, key(z) = 15, X < 15, and tree(u₁) is a 2-complete tree.

After the transformation key(u₁₋₁) = 15, and BIA is called recursively to insert X into tree(u₁) and 20 into tree(u₁₋₁).

Figure 3.5.3
BIA for 12.

\[ m=2 \] A double rotation on the nodes containing keys 11, 13 and 12

BIA for 12.

\[ m=2 \] A single rotation on the nodes containing keys 14, 13 and 12

Figure 3.5.4
BIA for 6, m=1

Figure 3.5.5
BDA for 9.
\[\text{key}(u_1)=10, \text{key}(u_2)=5, \text{key}(u_3)=8, \text{key}(u_4)=9, \text{i.e., } k'=4.\]
BDA for 25, one son case.

Figure 3.5.7'
BDA for 40.

$k'=3$, key($u_{k'}$) = 40, key(z) = 45. Apply step (ii) at $z = u_{k'}$.

Figure 3.5.8
\( k=3 \)

\[ \begin{array}{c}
10 \\
13 \\
20 \\
6 \\
6 \\
30 \\
40 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
1 \\
\end{array} \]

BDA for 40

\( k=3, i=2, \text{key}(u'_k)=40, \text{tree}(u'_{k-1}) \) is \((i+1)\)-incomplete (i.e. \( \beta \)-incomplete), therefore remove \( u'_k \).

\textbf{Figure 3.5.9}
BDA for 50.

$k' = 4$, key($u_{k'}$) = 50, tree($u_{k' - 2}$) is 3-replete, key($z$) = 25. Use BIA to insert 30 into tree($u_{k' - 1}$) and apply step (ii) at $z = u_{k'}$.  

Figure 3.5.10
BDA for 50

$k' = 4$, since $\text{tree}(u_{k'-1})$ is $(k'-i)$-complete, $1 \leq i \leq 3$ and $\text{tree}(u_{k'-3})$ is not 3-replete simply remove $u_{k'}$.

Figure 3.5.11
3.6 KNUTH CONSTRUCTION ALGORITHM (KNCA)

3.6.1 INTRODUCTION

This weighted external tree construction algorithm produces an optimal binary search tree; however, since it requires time and memory space proportional to \( n^2 \), it is of limited practical use. Observing that all subtrees of an optimal tree are optimal the following bottom-up tree construction method is suggested. Given \( n > 0 \), \( n \) ordered keys \( X_i \), \( 1 \leq i \leq n \), and \( 2n + 1 \) non-negative weights \( a_i \), \( 1 \leq i \leq n \), and \( b_i \), \( 0 \leq i \leq n \), the scheme is as follows. Given the root node with key \( X_1 \), then its left subtree is an optimum tree for the weights \( a_1, \ldots, a_{i-1}, b_0, \ldots, b_{i-1} \); its right subtree is optimum for \( a_{i+1}, \ldots, a_n, b_i, \ldots, b_n \). Therefore we can build up optimum trees for all "weight intervals" \( a_{i+1}, \ldots, a_j, b_i, \ldots, b_j \), \( i < j \), starting from the smallest intervals and working toward the largest.

Let \( WPL(i,j) \) and \( W(i,j) \) be the weighted path length and total weight of an optimum search tree with the weights \( b_i, a_{i+1}, \ldots, a_j, b_j \) and let \( R(i,j) \) denote the index of the key which is the root of this tree when \( i < j \). An algorithm can be formulated from the following equations:

\[
WPL(i,1) = W(i,1) = b_i, \quad 0 \leq i \leq n, \quad (1)
\]

\[
W(i,j) = W(i,j-1) + a_j + b_j, \quad 0 \leq i < j \leq n, \quad (2)
\]

\[
WPL(i,j) = \left[ \min_{k} \left( WPL(i,k-1) + WPL(k,j) \right) + W(i,j) \right], \quad 1 \leq k \leq j, \quad 0 \leq i < j \leq n \quad (3)
\]

Equation (3) is the observation that the weighted path length of a tree, with root node \( u \) say, is the sum of the weighted path lengths of the left and right subtrees of \( u \) plus the weight of the entire tree. It
forces a root to be chosen which results in minimal path length. Using equation (3) it is possible to evaluate WPL(i,j) for j-i = 1, 2, ..., n. Knuth (1971) proved the following theorem.

**Theorem**

There is always a solution to the above equations satisfying R(i,j-1) ≤ R(i,j) ≤ R(i+1,j) for 0 ≤ i < j ≤ n, when the weights are non-negative.

This limits the search for the minimum in (3) since only R(i+1,j) - R(i,j-1) + 1 values of k need to be examined instead of j-i.

### 3.6.2 THE KNCA

We are given, n > 0, n ordered keys X_i, 1 ≤ i ≤ n, and 2n + 1 weights α_i, 1 ≤ i ≤ n and β_i, 0 ≤ i ≤ n.

(i) Determine all the one-node optimum trees. For 1 ≤ i ≤ n, let R(i-1,i) = i. Let ℓ = 2.

(ii) Determine all the ℓ-node trees. If ℓ ≤ n then do step (iii), otherwise do step (iv).

(iii) For j = ℓ, ℓ+1,...,n let i = j-ℓ,

\[ WPL(i,j) = W(i,j) + \min(\{WPL(i,k-1) + WPL(k,j) \mid R(i,j-1) ≤ k ≤ R(i+1,j)\}) \]

and R(i,j) equals the value of k for which the minimum occurs.

Let ℓ = ℓ+1 and repeat step (ii).

(iv) Construct the tree, tree(u).

(a) Initially i = 0, j = n.

(b) If i = j, u = \emptyset, otherwise key(u) = X_R(i,j); u_ is given by (b) with j = R(i,j)-1 and u_ is given by (b) with i = R(i,j).
3.6.3 IMPLEMENTATION

The implementation of this algorithm utilizes two \((n+1)^2\) arrays for the values of the weight and the weighted path length, an \((n+1) \times n\) array for the values of the root indices, two arrays of size \(n\) to contain the keys and their weights and an array of size \(n+1\) to contain external weights. Only approximately one half of the storage allocated by the two dimensional arrays is actually used, however this may be readily overcome by suitable combinations and linearizations. The resulting storage would be proportional to \(n^2\). It was found, for purposes of this project, that even with this reduction certain desired trees could not be constructed for lack of space. For this reason and for clarity of the implemented algorithm the arrays remain as initially described.

Since this program is needed by another tree construction algorithm, it has superfluous parameters which are used to determine correct indices to the weight arrays.
3.7 THE BRUNO COFFMAN CONSTRUCTION ALGORITHM (BCCA)

3.7.1. INTRODUCTION

This heuristic method of finding nearly optimal weighted trees, defined by Bruno and Coffman (1971), requires relatively small amounts of storage and computation time. Given a tree, tree(u), the algorithm performs transformations on the tree so as to reduce the weighted path length. To do this, certain nodes must be promoted closer to the root. The heuristic gives an efficient method for determining when a node should be promoted and simple transformations to implement this promotion. Let us now consider the transformations in more detail.

Let $C_n$ denote the set of binary search trees on the keys $X_i$ and the weights $\alpha_i$, $1 \leq i \leq n$. Let $T$ be in $C_n$. For notational convenience let $u_i$, a node of $T$ with key $X_i$, $1 \leq i \leq n$, be denoted by its index $i$. Define a transformation $t_i$ on $C_n$, $1 \leq i \leq n$ as follows:

$$t_i(T) = \begin{cases} 
\text{if } u_i \text{ is the root then } T, \\
\text{otherwise see Figure 3.7.1.}
\end{cases}$$

If $u_i$ is not the root then after this transformation the level of $u_i$ has been decreased by one. A more general transformation $t_{i,k}$ on $C_n$, $1 \leq i \leq n$, may be defined in which the level of $u_i$ is reduced by $k \geq 1$. For $1 \leq i \leq n$, $k \geq 1$

$$t_{i,k}(T) = \begin{cases} 
\text{if } k = 1 \text{ then } t_i(T), & \text{otherwise} \\
\text{if } \varepsilon(u_i) < k+1 \text{ then } T \\
\text{otherwise } t_i(t_{i,k-1}(T)).
\end{cases}$$
Figure 3.7.2 illustrates $\tau_{i,k}(T)$ for $k = 2$. Let $\pi_k$, $k \geq 1$, be a mapping from $\mathbb{C}_n$ into $2^n$ defined for all $T$ in $\mathbb{C}_n$ as follows:

$$\pi_k(T) = \{T' | T' = \tau_{i,j}(T), 1 \leq j \leq k, 1 \leq i \leq n\}.$$  $\pi_k(T)$ is said to define a neighborhood about $T$. It is easy to see that $\pi_k(T) \subseteq \pi_{k+1}(T)$ for $k \geq 1$ and, moreover, $\pi_n(T) = \pi_{n+1}(T)$ for $i \geq 1$. There are at most $n^2$ members in $\pi_n(T)$ and consequently by increasing $k$ we do not necessarily include all the members of $\mathbb{C}_n$ in $\pi_k(T)$.

With the above information the heuristic may now be described. Given $k \geq 1$ then proceed as follows: (1) enumerate the elements of $\pi_k(T)$ in some way until a $T'$ is found such that $WPL(T') < WPL(T)$ and repeat (1) for $T'$. Since there are a finite number of elements in $\pi_k(T)$ and there is a $T'$ such that $WPL(T') \leq WPL(T)$ for all $T$ in $\mathbb{C}_n$, the process must terminate. Two questions remain to be answered, the first being the selection of the initial tree $T$. Any arbitrary tree may be chosen but in practice it is best to choose a "good" starting tree. Bruno and Coffman found the largest-first method was successful. With this the keys are partitioned into $m$ blocks $B_1, \ldots, B_m$, where for blocks $B_i$, $B_{i+1}$, $1 \leq i < m$, the weights of the keys in $B_i$ are greater than or equal to the weights of the keys in $B_{i+1}$. Now a random permutation of the keys in each block is obtained and a tree is constructed as with the BTIA using as input the randomized keys of $B_i$, $1 \leq i \leq m$. In this way keys with large weights will be closer to the root. By varying the size of the blocks different starting trees may be generated. The second question is how to enumerate $\pi_k(T)$. Again a largest-first scan has proven satisfactory. Assuming $\alpha_i \geq \alpha_{i+1}$ that is, the weight of key($u_i$) is
greater than or equal to the weight of key($u_{i+1}$), $1 \leq i < n$, then
examine $\pi_k(T)$ in the order $t_{1,k}(T), \ldots, t_{n,k}(T), t_{1,k-1}(T), \ldots, t_{n,k-1}(T), \ldots, t_{n,1}(T)$.

3.7.2. THE BCCA

Assume we are given an $n$ node tree, tree($u$), whose keys $X_i$ have
weights $\alpha_i$, $1 \leq i \leq n$, and a parameter $k$, $k \geq 1$.

(i) Define the sequence of nodes $u_1, u_2, \ldots, u_n$ such that for
$i, 1 \leq i < n$, the weight of key($u_i$) exceeds or equals
the weight of key($u_{i+1}$).

(ii) Scan the neighborhood of tree($u$), $\pi_k(\text{tree}(u))$: For
$j = k, k-1, \ldots, 1$ do step (iii) and then stop.

(iii) Examine particular elements of $\pi_k(\text{tree}(u))$. For
$i = 1, \ldots, n$ investigate $t_{i,j}(\text{tree}(u))$. If
WPL($t_{i,j}(\text{tree}(u))$) < WPL(tree($u$)) the weighted path length
of the tree has been reduced and tree($u$) is replaced by
$t_{i,j}(\text{tree}(u))$ (continue with the next value of $i$).

3.7.3. IMPLEMENTATION

The algorithm uses a special header node whose right pointer
field points to the root of the tree. Two additional fields of information are contained in any record representing a node. One contains the weight of the key which defines the node, the other contains the weight of the subtree defined by the node. Along with the starting tree and the parameter $k$, a vector of $n$ keys $X_i$ such that for $1 \leq i < n$, the weight of $X_i$ exceeds or equals the weight of $X_{i+1}$, is passed to the algorithm.
One very important aspect of the heuristic is the ease with which one can determine if \( \text{WPL}(T_{i,k}(\text{tree}(u))) < \text{WPL}(\text{tree}(u)) \) without actually performing any transformations. Referring to Figure 3.7.3 one can calculate the new weighted path length of a tree, \( \text{tree}(u) \), when a node is promoted one level. Let \( \text{tree}(u) \) denote the tree after the transformation, \( \text{WPL}(\text{tree}(u)) \) be its weighted path length and \( W(z) \), for any \( z \), be the weight of \( \text{tree}(z) \). Suppose \( y \) is to be promoted and it is a right son of its father, \( v \) say, then

\[
\text{WPL}(\text{tree}(u)) - \overline{\text{WPL}(\text{tree}(u))} = [W(v) + \overline{\text{WPL}(\text{tree}(v_L))} + W(y) + \overline{\text{WPL}(\text{tree}(v_R))}] - [\overline{W(v)} + \overline{W(y)} + \overline{\text{WPL}(\text{tree}(v_L))} + \\
+ \overline{\text{WPL}(\text{tree}(v_R))} + \overline{\text{WPL}(\text{tree}(y_L))}]
\]

\[
= [W(v) + W(y)] - [\overline{W(v)} + \overline{W(y)}]
\]

\[
= W(y) - \overline{W(v)}.
\]

Therefore \( \overline{\text{WPL}(\text{tree}(u))} = \text{WPL}(\text{tree}(u)) + \overline{W(v)} - W(y). \) (1)

The same equation results when \( y \) is a left son of \( v \). It is important to note that \( \overline{W(v)} \) is the weight of the (left) right subtree of \( y \) after the transformation and that \( W(v) \) and consequently \( W(y) \) are the only subtree weights affected. \( \overline{W(v)} \) is easily calculated. If \( y \) is a right son of \( v \) then \( \overline{W(v)} \) is the weight of the left subtree of \( y \), \( \overline{W(y_L)} \), and

\[
\overline{W(v)} = W(y_L) + \alpha_v + W(v_L), \alpha_v \text{ representing the weight of key}(v).
\]

If \( y \) is a left son of \( v \) then \( \overline{W(v)} \) is the right subtree of \( y \), \( \overline{W(y_R)} \), and

\[
\overline{W(v)} = W(y_R) + \alpha_v + W(v_R).
\]

\( \overline{W(y)} \) is now given by

\[
\overline{W(y)} = \overline{W(v)} + \alpha_y + \overline{W(y_L)} \text{ or } \overline{W(y)} = \overline{W(v)} + \alpha_y + \overline{W(y_R)} \text{ respectively.}.
\]
Since the whole process hinges on the weights of the subtree of $y$, two variables, initialized to $W(y_L)$ and $W(y_R)$, are used to hold the values of $\overline{W(y_L)}$ and $\overline{W(y_R)}$ as $y$ is promoted $k$ times. Another variable, initialized to $W(y)$, is used to contain the value of $\overline{W(y)}$. At each promotion $\overline{\text{WPL(tree}(u))}$ is calculated using equation (1). After $k$ promotions a comparison of the weighted path lengths can be made and if the weighted path length has been reduced, the transformation $t_{i,k}$ is applied. A circular stack of length $k + 1$ is used to contain pointers to $y$ and its $k$ ancestors. Also an array of length $k$ is used to contain the values $\overline{W(v)}$ as $y$ is being promoted. These values become the new weights of the left or right subtrees of $y$ as each actual promotion occurs.

$\overline{\text{WPL(tree}(u))}$ is calculated initially by simply traversing tree$(u)$ and summing the weights of the subtrees defined by each node. A wise choice of the parameter $k$ is the maximum level of any node in the starting tree, although this may not be the best choice as far as computation time.
key(u) = 10, key(u_1) = 35. u_1 is a right son of its father.

key(u) = 30, key(u_1) = 10. u_1 is a left son of its father.

**Figure 3.7.1**
$k = 2$, $\text{key}(u) = 15$, $\text{key}(u_1) = 45$. $t_{1,k} (\text{tree}(u))$ results in the above.

**Figure 3.7.2**
key(y) = 20, key(v) = 10. y is the right son of v.

key(y) = 20, key(v) = 30. y is the left son of v.

Figure 3.7.3
3.8 THE WALKER GOTLIEB CONSTRUCTION ALGORITHM (UGCA)

3.8.1 INTRODUCTION

This weighted external tree construction method was defined by Walker and Gotlieb (1972a). Like the Bruno Coffman algorithm of Section 3.7, it avoids excessive amounts of storage space and computation time and yet produces a nearly optimal tree on n ordered keys \( X_i \), \( 1 \leq i \leq n \), and \( 2n + 1 \) non-negative weights \( \alpha_i \) and \( \beta_i \). The method is a combination of two ideas suggested by Knuth (1971) to produce a tree that is very nearly optimal. The first idea was to choose as the key of the root node the input key of maximum weight and repeat this process for subsequent subtrees. This will not produce an acceptable tree in all cases since the external weights are ignored. The second idea was to choose as the key of the root node, a key which results in the weight of the left and right subtrees of the root being almost equal. There may be one or two such choices for this key as indicated in Figure 3.8.1. Walker and Gotlieb call the "largest" of the two keys (where largest refers to the ordering defined on the keys) the centroid. Using this idea a key for the root node, \( u \), say, may be selected which has a relatively small weight. However, the weight of \( \text{key}(u_r) \) or \( \text{key}(u_l) \) could be much larger and should actually be the root of the optimum tree. Trees constructed by choosing the centroid as root were much "better" than those constructed by the first method. Walker and Gotlieb approached the situation in the following manner. For a set of \( n \) ordered keys \( X_i \) and \( 2n + 1 \) associated weights, they constructed \( n \) trees by choosing as the key of the root of each tree a different \( X_i \) and forming optimal subtrees using the Knca.
Many examinations such as these revealed that the minimum weighted path length occurs when the weight of the key of the root is a local maximum, i.e. if \( X_i \) is the key of the root then \( a_{i-1} < a_i > a_{i+1} \). Denoting \( WPL(X_i) \) as the weighted path length of a tree whose root has \( X_i \) as its key, it was noted that if \( WPL(X_{i-1}) > WPL(X_i) < WPL(X_{i+1}) \) then this usually corresponds to the associated \( a_i \) being a local maximum. In other words a local minimum of \( WPL(X_i) \) corresponds to a local maximum for the \( a_i \). This relation does not apply when there are a small number of nodes in one of the subtrees since the weighted path length of a subtree may be significantly changed upon the removal or addition of even one node. The method is formulated by choosing as the key of the root node that key which has maximum weight in some neighborhood of the centroid. If this maximum is not unique the key which is closest to the centroid is selected. The process is then repeated for the subtrees of the selected root node. When the number of nodes in the subtrees becomes too small for this rule to be effective the KNCA is used to construct an optimal subtree.

Two questions remain to be answered, the first being the determination of a neighborhood about the centroid. Walker and Gotlieb supply a parameter, \( F \), to their algorithm to determine the neighborhood. Let \( \alpha \) and \( \beta \) denote the sum of the \( a_i \) and \( \beta_i \) respectively and \( W(i,j) \) denote the weight of a tree with associated weights \( \beta_i, a_{i+1}, \beta_{i+1}, \ldots, a_j, \beta_j, i \leq j \). They use the measure \( \frac{W(0,n)}{F} \) for determining the neighborhood of the centroid, i.e. if

\[
|W(0, \ell - 1) - W(\ell, n)| < \frac{W(0,n)}{F}
\]

then \( X_\ell \) is a candidate for the key of the root node. The selection of \( F \)
depends on $\alpha$ and $\beta$. They found the following choices of $F$ to be acceptable: (1) if $\beta$ is many times $\alpha$, choose $F = W(0,n)$, (2) if $\alpha$ and $\beta$ are nearly equal choose $F = 4$, (3) if $\beta$ is small compared to $\alpha$, the individual $a_i$ weights determine the best value of $F$. After empirical investigations they conclude that unless $\beta$ is many times $\alpha$, $F = 4$ is an acceptable choice.

Expansion of the neighborhood is possible. Suppose a set of $m$ ordered keys has been selected, $\{X_1, \ldots, X_k\}$, $j \leq k$. Denote $a^*$ as the maximum weight of any key in this set and $X_c$ as the centroid. If the weight of $X_j$ equals $a^*$ and there is no other $X_i$ such that $X_j < X_c < X_i$ and $a_i = a^*$, then include in the set any $X_{j-i-1}$ such that $a_{j-i-1} > a_{j-i}$, $i = 0, \ldots, j-p-1$ and $a_p \leq a_p$, or $j = \lceil \log_2 n \rceil$. A symmetrical test is performed for $X_k$. This includes any keys which have a weight larger than those detected in the initial neighborhood and are near the centroid. The bound $\lceil \log_2 n \rceil$ prevents the neighborhood from becoming too large.

The second question to be answered is that of the size of the subtrees which are structured using the KNCA. The parameter $N_0$ answers this question. The larger the value of $N_0$ the closer the tree will be to optimal since larger subtrees are created by KNCA. However, Walker and Gotlieb found that there is a value of $N_0$ beyond which the average search length of the tree decreases slowly or remains constant and that this value is determined by $\alpha$ and $\beta$. In general if $\beta$ is less than a few times $\alpha$, the value beyond which it does not pay to go is small, $N_0 = 15$ being a good choice. If $\beta$ is many times $\alpha$, $N_0$ should be increased to perhaps 25 or 30.
3.8.2 THE WGCA

We are given \( n > 0 \), \( n \) ordered keys \( X_i, 1 \leq i \leq n \), \( 2n+1 \) non-negative weights \( \alpha_i, 1 \leq i \leq n \), and \( \beta_i, 0 \leq i \leq n \), and the two parameters \( N_0 \) and \( F \).

(i) If \( n \leq N_0 \), use the KNCA to construct an optimal tree.

(ii) Determine the keys in the neighborhood of the centroid.

Let \( X_c \) be the centroid for the weights being considered.

Form the ordered set of keys \( S = T \cup \{ X_i \} \) where the members of the set \( T, X_i \), satisfy

\[
|W(0,i-1) - W(i,n)| < \frac{W(0,n)}{F}, \quad 1 \leq F \leq W(0,n).
\]

(iii) Find the maximum weight of any key in \( S \).

Find an index, \( \text{max} \), such that \( \alpha_{\text{max}} = \text{maximum } \alpha_i \) where \( X_i \) is in \( S \).

(iv) Determine if any key \( X_i \) in \( S \) such that \( X_i < X_c \) has as its associated weight \( \alpha_{\text{max}} \). Let \( \ell \) be the maximum index \( \ell < c \) such that \( \alpha_{\ell} = \alpha_{\text{max}} \). If there is no such \( \ell \), let \( L \) be the null set and go to step (vi).

(v) Expand the neighborhood to the left of the centroid.

If \( X_{\ell} \) is the first member of \( S \) and \( \alpha_{\ell-1} > \alpha_{\ell} \), form the ordered set \( L = \{ X_p, \ldots, X_{\ell-2}, X_{\ell-1} \} \) where

\[
\alpha_{\ell-j-1} > \alpha_{\ell-j}, \quad j = 0, \ldots, \ell-p-1 \quad \text{and} \quad \alpha_{p-1} \leq \alpha_p, \quad \text{or}
\]

\[
\ell-p = \lfloor \log_2 n \rfloor. \quad \text{If } X_{\ell} \text{ is not the first member of } S \text{ let } L \text{ be the null set.}
\]

(vi) Determine if any key \( X_i \) in \( S \) such that \( X_i > X_c \) has as its associated weight \( \alpha_{\text{max}} \). Let \( r \) be the minimum index,
\( r \geq c \), such that \( a_r = a_{\text{max}} \). If there is no such \( r \), let \( R \) be the null set and go to step (viii).

(vii) Expand the neighborhood to the right of the centroid.
If \( X_r \) is the last member of \( S \) and \( a_r < a_{r+1} \), form the ordered set \( R = \{ X_{r+1}, X_{r+2}, \ldots, X_p \} \) where
\[
\alpha_{r+j} < \alpha_{r+j+1}, \quad j = 0, \ldots, p-r-1 \text{ and } \alpha_p \geq \alpha_{p+1}, \quad \text{or}
\]
\[ p-r = \lceil \log_2 n \rceil \]. If \( X_r \) is not the last member of \( S \) let \( R \) be the null set.

(viii) Choose the key of the root node. Find the key \( X_{\text{root}} \) of maximum weight in the set \( L \cup S \cup R \) such that
\[
|W(0, \text{root}-1) - W(\text{root}, n)| \text{ is minimized. Choose } X_{\text{root}} \text{ as the key of the root of the tree.}
\]

(ix) Repeat the algorithm for the subtrees \( X_1, \ldots, X_{\text{root}-1}; X_{\text{root}+1}, \ldots, X_n \) where \( n \) is root-1 and n-root respectively.

3.8.3 IMPLEMENTATION

The implemented algorithm follows closely the steps outlined in the WCCA. There are two parameters input, \( N_0 \), the number of nodes structured into an optimum subtree by the KNCA, and \( F \) which is used in determining the neighborhood about the centroid from which keys are selected to be possible keys of root nodes. There are no additional fields in any record representing a node. All calculations involving weights are done using two one-dimensional arrays of size \( n \) and \( n + 1 \) respectively which contain the weights of the keys and the external weights. There are two one-dimensional arrays used to contain the keys and the set of possible root keys. Extra storage is needed for the arrays used by the
KNCA and this is determined by the value of $N_0$. The procedure STEP 1 is called recursively to find keys for the roots of subtrees until the number of keys in the subtree to be created is less than or equal to $N_0$ at which time the KNCA is invoked.

The weight of the keys are indicated directly beneath each node. The external weights are taken to be zero. Keys 25 and 30 are choices for the centroid with 30 being selected since $25 < 30$.

Figure 3.8.1
CHAPTER IV

EMPIRICAL RESULTS

4.1 INTRODUCTION

The results given below are divided into two classes as are the tree construction algorithms of Chapter III. Section 4.2 gives empirical results for the unweighted algorithms of Sections 3.3, 3.4 and 3.5; Section 4.3 gives those for the weighted tree algorithms of Sections 3.6, 3.7 and 3.8. In Section 4.2 an attempt is made not only to empirically investigate the average search path but also to investigate the average number of transformations necessary to create and maintain a tree of a particular size. To this end 500 trees of size 1 to 5000 nodes are built up and then broken down and statistics are collected on trees whose size is a multiple of 10. Permutations of the ordered sequence 1, 2, ..., 5000 are presented as keys to the algorithms each time a tree is built up or broken down. (A listing of the procedures used to collect statistics for BB-trees is given as an example at the end of Appendix 3.) Figures are given for trees whose size is a multiple of 250. The 95% confidence intervals are given for all averages shown. This is important since we are interested in "average behaviour". Two mnemonics are used in the tables of figures; "AV" refers to average, "CI" to the confidence interval. The standard measure of performance that is investigated in Section 4.3 is the average weighted path length. Trees of size 200 are constructed for this purpose. The weights used are a subset of those used by Walker and Gotlieb (1972a) and appear in Table 4.1.1.
Table 4.1.1 Test Data

The $a_i$ and $b_i$ vectors are arranged in the following matrices as

\[
\begin{array}{cccccccccccc}
  a_1 & a_2 & \cdots & a_{19} & a_{20} \\
  a_{21} & a_{22} & \cdots & a_{39} & a_{40} \\
  a_{181} & a_{182} & \cdots & a_{199} & a_{200} \\
\end{array}
\]

$a_i$ frequencies

<table>
<thead>
<tr>
<th>Set 1</th>
<th>4 19 3 3 1 1 28 4 4 5 1 1 1 35 13 27 11 32 1 50</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 1 1 27 2 31 2 3 1 2 7 59 4 1 1 7 3 2 3 22</td>
</tr>
<tr>
<td></td>
<td>2 1 52 3 1 4 1 1 1 6 8 13 1 4 1 1 1 1 1 1 26</td>
</tr>
<tr>
<td></td>
<td>33 1 1 8 7 1 2 1 1 2 4 10 5 10 23 1 1 9 6 2</td>
</tr>
<tr>
<td></td>
<td>128 350 1 2 1 87 2 1 8 3 4 1 63 2 3 14 150 1 1</td>
</tr>
<tr>
<td></td>
<td>1 2 .1 1 8 1 2 1 3 4 21 1 9 1 1 2 10 4 15 1</td>
</tr>
<tr>
<td></td>
<td>1 24 6 2 2 1 1 1 3 1 1 3 14 4 1 2 1 1 2 2</td>
</tr>
<tr>
<td></td>
<td>2 4 43 2 1 4 1 1 18 1 1 7 1 45 15 9 3 1 5 4</td>
</tr>
<tr>
<td></td>
<td>10 8 1 5 10 .1 2 10 2 1 24 29 1 16 (\text{a}) 1 1</td>
</tr>
<tr>
<td></td>
<td>1 1 1 1 4 2 2 6 4 1 14 5 5 1 53 1 1 1 4</td>
</tr>
</tbody>
</table>

\[\text{Set 1: Continued...}\]
Table 4.1.1 (cont'd.)

Set 2

<table>
<thead>
<tr>
<th>7</th>
<th>4</th>
<th>1</th>
<th>1</th>
<th>20</th>
<th>1</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>30</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>3</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>6</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>90</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>4</td>
<td>29</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>4</td>
<td>62</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>24</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>6</td>
<td>100</td>
<td>15</td>
<td>1</td>
<td>2</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>10</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>29</td>
<td>2</td>
<td>13</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>8</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>19</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>20</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>39</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>11</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>29</td>
<td>3</td>
<td>10</td>
<td>24</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>23</td>
<td>1</td>
<td>9</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>30</td>
<td>2</td>
<td>1</td>
<td>14</td>
<td>90</td>
</tr>
</tbody>
</table>

Set 3

| 8 | 5 | 306 | 1 | 12 | 1 | 13 | 1 | 1 | 10 | 1 | 2 | 2 | 2 | 3 | 2 | 2 | 1 | 2 | 1 |
|---|---|------|---|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 1 | 5 | 1 | 1 | 1 | 1 | 1 | 1 | 4 | 2 | 12 | 3 | 1 | 1 | 1 | 1 |
| 2 | 1 | 7 | 24 | 1 | 1 | 1 | 5 | 2 | 42 | 1 | 1 | 2 | 1 | 4 | 2 | 1 | 1 | 5 | 1 |
| 1 | 1 | 10 | 1 | 3 | 2 | 1 | 1 | 2 | 1 | 4 | 1 | 7 | 1 | 7 | 12 | 2 |
| 4 | 10 | 1 | 2 | 1 | 1 | 1 | 11 | 40 | 2 | 51 | 2 | 1 | 11 | 54 | 227 | 227 | 1 | 1 | 9 | 4 |
| 3 | 1 | 24 | 2 | 2 | 2 | 1 | 2 | 1 | 7 | 1 | 1 | 3 | 1 | 5 | 1 | 1 | 3 | 1 | 1 |
| 1 | 1 | 78 | 4 | 3 | 1 | 40 | 3 | 1 | 1 | 2 | 3 | 1 | 84 | 7 | 37 | 30 | 1 | 5 | 12 |
| 1 | 2 | 1 | 1 | 33 | 2 | 183 | 2 | 7 | 2 | 1 | 4 | 1 | 15 | 173 | 1 | 1 | 2 | 4 | 4 |
| 4 | 1 | 1 | 6 | 17 | 12 | 3 | 2 | 1 | 1 | 4 | 3 | 2 | 10 | 2 | 107 | 7 | 1 | 4 |
| 1 | 1 | 1 | 3 | 2 | 2 | 2 | 6 | 1 | 1 | 1 | 2 | 2 | 5 | 2 | 1 | 11 | 1 | 1 | 8 |

Set 4

| 7 | 4 | 1 | 1 | 20 | 1 | 2 | 1 | 3 | 1 | 1 | 1 | 30 | 1 | 1 | 1 | 1 | 1 | 3 | 30 |
|---|---|---|---|----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 1 | 1 | 4 | 2 | 2 | 1 | 3 | 1 | 2 | 1 | 7 | 1 | 2 | 4 | 2 | 6 | 6 | 7 | 1 |
| 3 | 1 | 3 | 2 | 1 | 1 | 29 | 2 | 13 | 3 | 1 | 1 | 2 | 1 | 2 | 8 | 1 | 10 | 11 | 9 |
| 1 | 6 | 19 | 2 | 1 | 3 | 1 | 1 | 1 | 4 | 1 | 20 | 1 | 1 | 3 | 3 | 6 | 2 | 11 | 3 |
| 2 | 1 | 1 | 1 | 1 | 2 | 29 | 3 | 10 | 24 | 2 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 55 | 1 |
| 4 | 19 | 3 | 3 | 1 | 1 | 28 | 4 | 4 | 5 | 1 | 1 | 1 | 35 | 13 | 27 | 11 | 32 | 11 | 50 |
| 2 | 1 | 1 | 27 | 2 | 31 | 2 | 3 | 1 | 2 | 7 | 59 | 4 | 1 | 1 | 7 | 3 | 2 | 33 | 22 |
| 33 | 1 | 1 | 8 | 7 | 1 | 2 | 1 | 1 | 2 | 4 | 10 | 5 | 10 | 23 | 1 | 1 | 9 | 66 | 2 |
| 1 | 2 | 1 | 1 | 8 | 1 | 2 | 1 | 3 | 4 | 21 | 1 | 9 | 1 | 1 | 2 | 10 | 4 | 15 | 1 |
| 2 | 4 | 43 | 2 | 1 | 4 | 1 | 1 | 18 | 1 | 1 | 7 | 1 | 45 | 15 | 9 | 3 | 1 | 5 | 4 |
Table 4.1.1 (cont'd.)

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 90| 3 | 1 | 4 | 1 | 4 | 29 | 1 | 1 | 1 | 1 | 2 | 6 | 2 | 2 | 1 | 5 | 5 | 2 | 40 |
| 1 | 2 | 1 | 1 | 33 | 2 | 183| 2 | 7 | 2 | 11| 4 | 1 | 15| 173| 1 | 1 | 2 | 4 | 4 |
| 1 | 1 | 1 | 24 | 1 | 1 | 6 | 100| 15 | 11| 2 | 7 | 1 | 1 | 1 | 10| 3 | 1 | 2 |
| 1 | 1 | 78| 4 | 3 | 1 | 40 | 3 | 1 | 1 | 22| 3 | 1 | 84| 7 | 37| 30 | 1 | 5 | 12|
| 2 | 1 | 52| 3 | 1 | 4 | 1 | 1 | 1 | 6 | 68| 13 | 1 | 4 | 1 | 1 | 1 | 1 | 1 | 26|
| 2 | 4 | 43| 2 | 1 | 4 | 1 | 1 | 18| 1 | 1 | 7 | 1 | 43| 15 | 9 | 3 | 1 | 5 | 4 |
| 8 | 5 | 306| 1| 12 | 1 | 13 | 1 | 1 | 10| 1 | 2 | 2 | 2 | 3 | 2 | 2 | 1 | 2 | 1 |
| 4 | 10| 1 | 2 | 1 | 1 | 11 | 40 | 2 | 51| 2 | 1 | 11 | 54| 227| 227| 1 | 1 | 9 | 4 |
| 10| 8 | 1 | 5 | 10| 1 | 2 | 10| 2 | 1 | 24| 29| 1 | 16| 1 | 1 | 4 | 1 | 2 | 1 |
| 128| 350| 1| 2 | 1 | 87 | 2 | 1 | 8 | 3 | 4 | 1 | 63 | 2 | 3 | 14 | 150| 1 | 1 | 1 |

$\beta_i$ frequencies

Class 1: The $\beta_i$ are all equal.

**case 1** $\beta_i = 0$, $0 \leq i \leq n$.

**case 2** $\beta_i = 10$, $0 \leq i \leq n$.

Class 2: The $\beta_i$ were chosen so that the sum of the $\beta_i$ would be equal to or less than the sum of the $\alpha_i$.

**case 3**

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 3 | 1 | 3 | 2 | 1 | 1 | 29 | 2 | 13 | 3 | 1 | 1 | 2 | 1 | 2 | 8 | 1 | 10 | 1 | 9 |
| 1 | 6 | 19| 2 | 1 | 3 | 1 | 1 | 1 | 4 | 1 | 20 | 1 | 1 | 3 | 3 | 6 | 2 | 1 | 3 |
| 2 | 1 | 1 | 1 | 1 | 2 | 29 | 3 | 10 | 24 | 2 | 1 | 1 | 1 | 3 | 1 | 1 | 1 | 5 | 1 |
| 2 | 1 | 1 | 4 | 2 | 2 | 1 | 3 | 1 | 2 | 1 | 7 | 1 | 2 | 4 | 2 | 6 | 6 | 7 | 1 |
| 7 | 4 | 1 | 1 | 20| 1 | 2 | 1 | 3 | 1 | 1 | 1 | 30| 1 | 1 | 1 | 1 | 1 | 3 | 30|
| 4 | 19| 3 | 3 | 1 | 1 | 28 | 4 | 4 | 5 | 1 | 1 | 1 | 35| 13 | 13 | 27 | 11 | 32| 1 | 50|
| 2 | 1 | 1 | 27| 2 | 31| 2 | 3 | 1 | 2 | 7 | 59 | 4 | 1 | 1 | 7 | 3 | 2 | 3 | 22|
| 33| 1 | 1 | 8 | 1 | 2 | 1 | 1 | 1 | 2 | 4 | 10| 5 | 10| 23| 1 | 1 | 9 | 6 | 2 |
| 1 | 2 | 1 | 1 | 8 | 1 | 2 | 1 | 3 | 4 | 21| 1 | 9 | 1 | 1 | 2 | 10| 4 | 15 | 1 |
| 2 | 4 | 43| 2 | 1 | 4 | 1 | 1 | 18| 1 | 1 | 7 | 1 | 45| 15 | 9 | 3 | 1 | 5 | 4 |

4
Table 4.1.1 (cont'd.)

<table>
<thead>
<tr>
<th>case 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 4 43 2 1 4 1 1 18 1 1 7 1 45 15 9 3 1 5 4</td>
</tr>
<tr>
<td>1 2 1 1 8 1' 2 1 1 3 4 21 1 9 1 1 2 10 4 15 1</td>
</tr>
<tr>
<td>33 1 1 8 7 1 2 1 1 2 4 10 5 10 23 1 1 9 6 2</td>
</tr>
<tr>
<td>2 1 1 27 2 31 2 3 1 2 7 59 4 1 1 7 3 2 3 22</td>
</tr>
<tr>
<td>4 19 3 3 1 1 28 4 4 5 1 1 1 35 13 27 11 32 1 50</td>
</tr>
<tr>
<td>3 1 3 2 1 1 29 2 13 3 1 1 2 1 2 8 1 10 1 9</td>
</tr>
<tr>
<td>1 6 19 2 1 3 1 1 1 4 1 20 1 1 1 3 3 6 2 1 3</td>
</tr>
<tr>
<td>2 1 1 1 1 2 29 3 10 24 2 1 1 1 3 1 1 1 5 1</td>
</tr>
<tr>
<td>2 1 1 4 2 2 1 3 1 2 1 7 1 2 4 2 6 6 7 1</td>
</tr>
<tr>
<td>7 4 1 1 20 1 2 1 3 1 1 1 3 1 1 1 1 1 3 3</td>
</tr>
</tbody>
</table>

Class 3 The $\beta_i$ were chosen so the sum of the $\beta_i$ would be larger than the sum of the $\alpha_i$.

<table>
<thead>
<tr>
<th>case 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 1 52 3 1 4 1 1 1 6 8 13 1 4 1 1 1 1 1 1 26</td>
</tr>
<tr>
<td>128 350 1 2 1 87 2 1 8 3 1 4 1 63 2 3 14 140 1 1 1</td>
</tr>
<tr>
<td>2 4 43 2 1 4 1 1 18 1 1 7 1 45 15 9 3 1 5 4</td>
</tr>
<tr>
<td>10 8 1 5 10 1 2 10 2 1 24 29 1 16 1 1 4 1 2 1</td>
</tr>
<tr>
<td>90 3 1 4 1 4 29 1 1 1 1 2 6 2 2 1 5 5 2 40</td>
</tr>
<tr>
<td>1 1 1 24 1 1 1 6 100 15 1 2 7 1 1 1 10 3 1 2</td>
</tr>
<tr>
<td>8 5 306 1 12 1 13 1 1 10 1 2 2 2 3 .2 2 1 2 1</td>
</tr>
<tr>
<td>4 10 1 2 1 1 1 11 40 2 51 2 1 11 54 227 227 1 1 9 4</td>
</tr>
<tr>
<td>1 1 78 4 3 1 40 3 1 1 2 3 1 84 7 37 30 1 5 12</td>
</tr>
<tr>
<td>1 2 1 1 33 2 183 2 7 2 1 4 1 14 173 1 1 2 4 4</td>
</tr>
</tbody>
</table>

4
Table 4.1.1 (cont'd.)

<table>
<thead>
<tr>
<th>case 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>315</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>49</td>
</tr>
<tr>
<td>78</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>156</td>
</tr>
<tr>
<td>720</td>
</tr>
<tr>
<td>900</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>623</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>48</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>479</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>84</td>
</tr>
<tr>
<td>900</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>129</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>42</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>570</td>
</tr>
<tr>
<td>94</td>
</tr>
<tr>
<td>56</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>160</td>
</tr>
<tr>
<td>49</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>68</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>145</td>
</tr>
<tr>
<td>183</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>97</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>840</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>240</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>45</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>43</td>
</tr>
<tr>
<td>907</td>
</tr>
<tr>
<td>110</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>41</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>211</td>
</tr>
<tr>
<td>60</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>49</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>930</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>75</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>98</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>70</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>121</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>63</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>315</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>66</td>
</tr>
<tr>
<td>350</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>307</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>66</td>
</tr>
<tr>
<td>323</td>
</tr>
<tr>
<td>999</td>
</tr>
<tr>
<td>960</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>54</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>468</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>540</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>73</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>195</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>970</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>43</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>90</td>
</tr>
<tr>
<td>970</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>497</td>
</tr>
<tr>
<td>45</td>
</tr>
<tr>
<td>221</td>
</tr>
<tr>
<td>190</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>.72</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>349</td>
</tr>
<tr>
<td>78</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>
4.2 EMPIRICAL OBSERVATIONS ON ALGORITHMS OF CLASS (1)

4.2.1 AVL TREES

For insertion of a key into a tree of a given size the following statistics are tabulated:

(i) the average number of comparisons necessary to reach the position where the new node will be inserted,

(ii) the average number of single rotations performed when inserting the new key,

(iii) the average number of double rotations performed when inserting the new key,

(iv) the average number of general rotations (single and double) performed when inserting the new key.

The corresponding tabulated statistics are found in Table 4.2.1.

For deletion of a node in a tree of a given size the following are collected:

(i) the average number of comparisons necessary to locate the node to be deleted; this quantity represents the average search for the tree size being considered,

(ii) the average number of single rotations necessary to maintain the AVL balance upon deletion of the node,

(iii) the average number of double rotations necessary to maintain balance,

(iv) the average number of special single rotations performed (corresponding to step (iv) (c) in the AVLDA),

(v) the average number of general rotations (single and double) necessary to maintain balance,
(vi) the average number of nodes visited during the traceback procedure (i.e. the average number of nodes investigated, using the deletion stack, until it is determined that the tree is AVL balanced.

These figures are found in Tables 4.2.2.

From the tables it is clear that the average search for deletion and insertion is logarithmic in the number of nodes in the tree. Other statistics are observed to be independent of the size of the tree which agrees with the claims of Scroggs, Karlton, Fuller and Kaehler (1973). It is of interest to note that the average traceback when deleting a node is only about 2.0, an unexpected result.

4.2.2 BELL-TREES

For insertion of a key into a tree of a given size the following statistics are tabulated:

(i) the average number of comparisons necessary to reach the position where the new node will be inserted,

(ii) the average number of single rotations performed when inserting a key,

(iii) the average number of double rotations performed,

(iv) the average number of general rotations performed (single and double rotations),

(v) the average number of times step (v) (a) of the BIA is performed (henceforth referred to as single recurse rotation),

(vi) the average number of times step (v) (b) of the BIA is
performed (henceforth referred to as double recurse rotation),

(vii) the average of the sum of the single and double recurse counts (referred to as average recurse rotation),

(viii) the average of all rotations performed upon the insertion of a key (the average of the sum of (ii), (iii), (v), (vi),

(ix) the average number of nodes examined in the sequence in step (v) of the BIA before a \((k'-1)\)-complete tree is found (henceforth referred to as average shift),

(x) the average number of times the procedure recurse is called in the implemented version of the BIA.

The figures corresponding to the above are found in Tables 4.2.3 to 4.2.12 respectively.

For the deletion case the following are collected:

(i) the average number of comparisons necessary to locate the node to be deleted; this represents the average search path for the tree size being considered,

(ii) the average number of times the procedure nosons is called,

(iii) the average number of times the procedure oneson is called,

(iv) the average number of times the procedure recurse is called,

(v) the average number of times the node to be deleted has no sons (henceforth referred to as probability nosons),

(vi) the average number of times the node to be deleted has one son (referred to as probability oneson),

(vii) the average number of times the node to be deleted has two sons (referred to as probability twosons).
These figures are found in Tables 4.2.13 to 4.2.19 respectively.

For the insertion statistics a general comment holds true: as the parameter \( k \) increases the search path decreases and the amount of rotations performed increases. This is expected since as \( k \) increases minimal subtrees of an increasing size are constructed at each stage and in order to do this more rotations are needed. It is interesting to note that all statistics, other than average search, are independent of tree size but increase with \( k \).

For deletion Table 2.4.13 measures the average search path and shows it is logarithmic in the size of the tree. As \( k \) increases the search path decreases but for values of \( k \) equal to 4, 5 and 6 the change is very small. Table 4.2.16, AVERAGE RECURSE, is an indicator of the complexity and the amount of work needed to maintain the tree. It is seen that it increases with \( k \). The probability counts are as expected, the average number of times a node to be deleted has two sons and no sons (a leaf) being approximately equal.

4.2.3 \( \text{BB-TREES} \)

For both insertion and deletion of a key into a tree of a given size the following statistics are collected:

(i) the average number of comparisons necessary to reach the position where the new node will be inserted and a node will be deleted,

(ii) the average number of single rotations performed when inserting and deleting nodes,
(iii) the average number of double rotations performed when inserting and deleting nodes,
(iv) the average number of general rotations performed (single and double rotations).

Tests were run for $a$ values of $\frac{1}{7}$, $\frac{1}{5}$, $\frac{1}{4}$ and $\frac{29}{100}$; the results for insertion are found in Tables 4.2.20,...,4.2.23, those for deletion in Tables 4.2.24,...,4.2.27. In the deletion tables some entries under the confidence interval column are blank. Because the number of occurrences is very small for the statistic being collected, the average is very low and the corresponding confidence intervals are ignored. Figures however can be found in a table of confidence intervals for the expectation of a Poisson variable.

From Table 4.2.24 it is clear that the average search path is logarithmic in the size of the table. It is observed that as $a$ increases the average search is reduced and the confidence intervals become tighter. The number of general rotations is seen to increase with $a$ and those for the insertion statistics are higher than for deletion. However, in no case did the average general rotation exceed a value of one, meaning infrequent restructuring is performed. This result does not agree with the analytical discussion given by Nievergelt and Reingold (1972). It is of interest to note that a single rotation occurs much more frequently than a double rotation. From examining Theorem 3.4.2 one would expect this result. All statistics, other than average search, appear to be independent of tree size.
4.2.4 CONCLUSIONS

Examination of the average search path for the three tree construction algorithms above (Tables 4.2.2, 4.2.13 and 4.2.24) indicates that for larger values of k the Bell-tree algorithm is the best. The AVL trees are seen to be superior to the BB-trees for values of \( a < \frac{29}{100} \). For \( a = \frac{29}{100} \) the difference is marginal. Examination of Tables 4.2.1, 4.2.2, 4.2.10, 4.2.16, 4.2.23 and 4.2.27 gives an idea as to the amount of work involved in producing the various tree structures. For insertion the BB-tree is superior. The AVL trees are very close to the BB-trees with respect to average general rotation, each in fact being less than one. The average total rotations involved in structuring a Bell-tree is seen to be greater than the BB-tree or the AVL tree for values of k greater than three. For the deletion case the BB-tree is superior with the AVL tree again very close. It is difficult to analyse the Bell-tree case but Table 4.2.16 does indicate an increasing amount of work as k increases.

Central processor timings (in seconds) are included for the running time of each tree algorithm during the statistics collection phase (see Table 4.2.28). The AVL trees are certainly superior in this respect. The overall conclusion reached is that the AVL tree construction algorithms result in the "best" tree. The search path and work involved in structuring an AVL tree is almost the same as in a BB-tree. However other considerations can be given. The construction time is least for the AVL tree algorithms, it is relatively simple to understand and implement as compared with the other two algorithms, only one rotation is needed to restore an unbalanced tree upon insertion of a key and no stack is needed, and empirically, at most only two rebalances are required for
deletion. BB-trees are seen to be inexpensive to maintain but the average search path deteriorates for small values of \( \alpha \). The amount of work involved in producing the Bell-tree and its complexity override the average search path considerations. Therefore it is seen that the AVL tree algorithm and the AVL tree structure is superior to both BB and Bell.
<table>
<thead>
<tr>
<th>Tree Size</th>
<th>Average Search</th>
<th>Average Single Rotation</th>
<th>Average Double Rotation</th>
<th>Average General Rotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>9.2000</td>
<td>0.0651</td>
<td>0.2660 0.0461</td>
<td>0.2400 0.0438</td>
</tr>
<tr>
<td>500</td>
<td>10.1680</td>
<td>0.0693</td>
<td>0.2520 0.0449</td>
<td>0.2340 0.0433</td>
</tr>
<tr>
<td>750</td>
<td>10.8120</td>
<td>0.0651</td>
<td>0.2360 0.0435</td>
<td>0.2460 0.0444</td>
</tr>
<tr>
<td>1000</td>
<td>11.2460</td>
<td>0.0678</td>
<td>0.2500 0.0447</td>
<td>0.2560 0.0453</td>
</tr>
<tr>
<td>1250</td>
<td>11.5140</td>
<td>0.0724</td>
<td>0.2080 0.0408</td>
<td>0.2320 0.0431</td>
</tr>
<tr>
<td>1500</td>
<td>11.7560</td>
<td>0.0743</td>
<td>0.2520 0.0449</td>
<td>0.1880 0.0388</td>
</tr>
<tr>
<td>1750</td>
<td>12.0180</td>
<td>0.0765</td>
<td>0.2580 0.0454</td>
<td>0.2100 0.0410</td>
</tr>
<tr>
<td>2000</td>
<td>12.2740</td>
<td>0.0688</td>
<td>0.2440 0.0442</td>
<td>0.2740 0.0468</td>
</tr>
<tr>
<td>2250</td>
<td>12.3640</td>
<td>0.0757</td>
<td>0.2160 0.0416</td>
<td>0.2120 0.0412</td>
</tr>
<tr>
<td>2500</td>
<td>12.4820</td>
<td>0.0741</td>
<td>0.2480 0.0445</td>
<td>0.2160 0.0416</td>
</tr>
<tr>
<td>2750</td>
<td>12.5860</td>
<td>0.0744</td>
<td>0.2360 0.0435</td>
<td>0.1900 0.0390</td>
</tr>
<tr>
<td>3000</td>
<td>12.7520</td>
<td>0.0737</td>
<td>0.2180 0.0418</td>
<td>0.2260 0.0425</td>
</tr>
<tr>
<td>3250</td>
<td>12.9240</td>
<td>0.0746</td>
<td>0.2180 0.0418</td>
<td>0.2260 0.0425</td>
</tr>
<tr>
<td>3500</td>
<td>13.0600</td>
<td>0.0721</td>
<td>0.2360 0.0435</td>
<td>0.2320 0.0431</td>
</tr>
<tr>
<td>3750</td>
<td>13.1920</td>
<td>0.0772</td>
<td>0.2480 0.0445</td>
<td>0.2540 0.0451</td>
</tr>
<tr>
<td>4000</td>
<td>13.2980</td>
<td>0.0745</td>
<td>0.2340 0.0433</td>
<td>0.2540 0.0451</td>
</tr>
<tr>
<td>4250</td>
<td>13.2920</td>
<td>0.0786</td>
<td>0.2360 0.0435</td>
<td>0.2300 0.0429</td>
</tr>
<tr>
<td>4500</td>
<td>13.3960</td>
<td>0.0765</td>
<td>0.2080 0.0408</td>
<td>0.2660 0.0461</td>
</tr>
<tr>
<td>4750</td>
<td>13.5320</td>
<td>0.0710</td>
<td>0.2460 0.0444</td>
<td>0.2300 0.0429</td>
</tr>
<tr>
<td>5000</td>
<td>13.5120</td>
<td>0.0790</td>
<td>0.2320 0.0431</td>
<td>0.2340 0.0433</td>
</tr>
</tbody>
</table>
### Table 4.2.2  AVL tree deletion statistics

<table>
<thead>
<tr>
<th>tree size</th>
<th>AVERAGE SEARCH</th>
<th>AVERAGE SINGLE ROTATION</th>
<th>AVERAGE DOUBLE ROTATION</th>
<th>AVERAGE SPECIAL SINGLE ROTATION</th>
<th>AVERAGE GENERAL TRACEBACK</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>7.0680</td>
<td>0.1372</td>
<td>0.0840</td>
<td>0.0259</td>
<td>0.0780</td>
</tr>
<tr>
<td>500</td>
<td>8.1800</td>
<td>0.1410</td>
<td>0.0580</td>
<td>0.0215</td>
<td>0.0660</td>
</tr>
<tr>
<td>750</td>
<td>8.7320</td>
<td>0.1436</td>
<td>0.0880</td>
<td>0.0265</td>
<td>0.0820</td>
</tr>
<tr>
<td>1000</td>
<td>9.0900</td>
<td>0.1357</td>
<td>0.0760</td>
<td>0.0247</td>
<td>0.0820</td>
</tr>
<tr>
<td>1250</td>
<td>9.4940</td>
<td>0.1475</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.0860</td>
</tr>
<tr>
<td>1500</td>
<td>9.7920</td>
<td>0.1427</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.0800</td>
</tr>
<tr>
<td>1750</td>
<td>10.0220</td>
<td>0.1361</td>
<td>0.0920</td>
<td>0.0271</td>
<td>0.0820</td>
</tr>
<tr>
<td>2000</td>
<td>10.2000</td>
<td>0.1434</td>
<td>0.0860</td>
<td>0.0262</td>
<td>0.0860</td>
</tr>
<tr>
<td>2250</td>
<td>10.4220</td>
<td>0.1389</td>
<td>0.0540</td>
<td>0.0208</td>
<td>0.0980</td>
</tr>
<tr>
<td>2500</td>
<td>10.9470</td>
<td>0.1463</td>
<td>0.0860</td>
<td>0.0262</td>
<td>0.1160</td>
</tr>
<tr>
<td>2750</td>
<td>10.7480</td>
<td>0.1447</td>
<td>0.0620</td>
<td>0.0223</td>
<td>0.0920</td>
</tr>
<tr>
<td>3000</td>
<td>10.8300</td>
<td>0.1441</td>
<td>0.0880</td>
<td>0.0265</td>
<td>0.0980</td>
</tr>
<tr>
<td>3250</td>
<td>11.1580</td>
<td>0.1263</td>
<td>0.0620</td>
<td>0.0223</td>
<td>0.0940</td>
</tr>
<tr>
<td>3500</td>
<td>11.2340</td>
<td>0.1249</td>
<td>0.0880</td>
<td>0.0265</td>
<td>0.1140</td>
</tr>
<tr>
<td>3750</td>
<td>11.3400</td>
<td>0.1296</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.0660</td>
</tr>
<tr>
<td>4000</td>
<td>11.3400</td>
<td>0.1347</td>
<td>0.1080</td>
<td>0.0294</td>
<td>0.0840</td>
</tr>
<tr>
<td>4250</td>
<td>11.2380</td>
<td>0.1326</td>
<td>0.0960</td>
<td>0.0277</td>
<td>0.0880</td>
</tr>
<tr>
<td>4500</td>
<td>11.1420</td>
<td>0.1508</td>
<td>0.0660</td>
<td>0.0230</td>
<td>0.0840</td>
</tr>
<tr>
<td>4750</td>
<td>11.2060</td>
<td>0.1687</td>
<td>0.0920</td>
<td>0.0271</td>
<td>0.0640</td>
</tr>
<tr>
<td>5000</td>
<td>8.9620</td>
<td>0.3666</td>
<td>0.0960</td>
<td>0.0277</td>
<td>0.0720</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>--------</td>
<td>-------</td>
</tr>
<tr>
<td>250</td>
<td>0.30</td>
<td>0.1507</td>
<td>0.1350</td>
<td>9.1360</td>
<td>0.0659</td>
</tr>
<tr>
<td>500</td>
<td>0.57</td>
<td>0.1762</td>
<td>0.1107</td>
<td>10.2000</td>
<td>0.0768</td>
</tr>
<tr>
<td>750</td>
<td>0.77</td>
<td>0.5929</td>
<td>0.1129</td>
<td>10.8400</td>
<td>0.1516</td>
</tr>
<tr>
<td>1000</td>
<td>0.99</td>
<td>0.7874</td>
<td>0.3945</td>
<td>11.3620</td>
<td>0.1955</td>
</tr>
<tr>
<td>1250</td>
<td>0.99</td>
<td>0.1789</td>
<td>0.1272</td>
<td>11.6480</td>
<td>0.0856</td>
</tr>
<tr>
<td>1500</td>
<td>0.99</td>
<td>0.1885</td>
<td>0.5884</td>
<td>11.8280</td>
<td>0.0852</td>
</tr>
<tr>
<td>1750</td>
<td>0.99</td>
<td>0.1777</td>
<td>0.6773</td>
<td>12.0860</td>
<td>0.0833</td>
</tr>
<tr>
<td>2000</td>
<td>0.99</td>
<td>0.1870</td>
<td>0.1262</td>
<td>12.2520</td>
<td>0.0823</td>
</tr>
<tr>
<td>2250</td>
<td>0.99</td>
<td>0.1845</td>
<td>0.1360</td>
<td>12.4640</td>
<td>0.0813</td>
</tr>
<tr>
<td>2500</td>
<td>0.99</td>
<td>0.2046</td>
<td>0.1353</td>
<td>12.5700</td>
<td>0.0911</td>
</tr>
<tr>
<td>2750</td>
<td>0.99</td>
<td>0.1949</td>
<td>0.1328</td>
<td>12.7320</td>
<td>0.0861</td>
</tr>
<tr>
<td>3000</td>
<td>0.99</td>
<td>0.2033</td>
<td>0.1326</td>
<td>12.9520</td>
<td>0.0922</td>
</tr>
<tr>
<td>3250</td>
<td>0.99</td>
<td>0.2041</td>
<td>0.1753</td>
<td>12.9660</td>
<td>0.0899</td>
</tr>
<tr>
<td>3500</td>
<td>0.99</td>
<td>0.2118</td>
<td>0.1285</td>
<td>13.1540</td>
<td>0.0896</td>
</tr>
<tr>
<td>3750</td>
<td>0.99</td>
<td>0.2635</td>
<td>0.1332</td>
<td>13.2820</td>
<td>0.0965</td>
</tr>
<tr>
<td>4000</td>
<td>0.99</td>
<td>0.2069</td>
<td>0.1336</td>
<td>13.4140</td>
<td>0.0926</td>
</tr>
<tr>
<td>4250</td>
<td>0.99</td>
<td>0.2049</td>
<td>0.1409</td>
<td>13.4180</td>
<td>0.0947</td>
</tr>
<tr>
<td>4500</td>
<td>0.99</td>
<td>0.1991</td>
<td>0.4246</td>
<td>13.4940</td>
<td>0.0936</td>
</tr>
<tr>
<td>4750</td>
<td>0.99</td>
<td>0.2144</td>
<td>0.1431</td>
<td>13.5420</td>
<td>0.0924</td>
</tr>
<tr>
<td>5000</td>
<td>0.99</td>
<td>0.2203</td>
<td>0.4105</td>
<td>13.6760</td>
<td>0.1019</td>
</tr>
<tr>
<td>tree size</td>
<td>2</td>
<td>CI</td>
<td>3</td>
<td>CI</td>
<td>4</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>250</td>
<td>0.1360</td>
<td>0.0330</td>
<td>0.2260</td>
<td>0.0425</td>
<td>0.3980</td>
</tr>
<tr>
<td>500</td>
<td>0.1340</td>
<td>0.0327</td>
<td>0.2680</td>
<td>0.0463</td>
<td>0.4060</td>
</tr>
<tr>
<td>750</td>
<td>0.1660</td>
<td>0.0364</td>
<td>0.2240</td>
<td>0.0423</td>
<td>0.4220</td>
</tr>
<tr>
<td>1000</td>
<td>0.1260</td>
<td>0.0317</td>
<td>0.2600</td>
<td>0.0456</td>
<td>0.4360</td>
</tr>
<tr>
<td>1250</td>
<td>0.1440</td>
<td>0.0339</td>
<td>0.2720</td>
<td>0.0466</td>
<td>0.3800</td>
</tr>
<tr>
<td>1500</td>
<td>0.1420</td>
<td>0.0337</td>
<td>0.2240</td>
<td>0.0423</td>
<td>0.4220</td>
</tr>
<tr>
<td>1750</td>
<td>0.1780</td>
<td>0.0377</td>
<td>0.2080</td>
<td>0.0408</td>
<td>0.4080</td>
</tr>
<tr>
<td>2000</td>
<td>0.1120</td>
<td>0.0299</td>
<td>0.2840</td>
<td>0.0477</td>
<td>0.3840</td>
</tr>
<tr>
<td>2250</td>
<td>0.1140</td>
<td>0.0302</td>
<td>0.2640</td>
<td>0.0460</td>
<td>0.4260</td>
</tr>
<tr>
<td>2500</td>
<td>0.1220</td>
<td>0.0312</td>
<td>0.2360</td>
<td>0.0435</td>
<td>0.4020</td>
</tr>
<tr>
<td>2750</td>
<td>0.1600</td>
<td>0.0358</td>
<td>0.2220</td>
<td>0.0421</td>
<td>0.3520</td>
</tr>
<tr>
<td>3000</td>
<td>0.1360</td>
<td>0.0330</td>
<td>0.2440</td>
<td>0.0442</td>
<td>0.3900</td>
</tr>
<tr>
<td>3250</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.2420</td>
<td>0.0440</td>
<td>0.3400</td>
</tr>
<tr>
<td>3500</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.2240</td>
<td>0.0423</td>
<td>0.3800</td>
</tr>
<tr>
<td>3750</td>
<td>0.1440</td>
<td>0.0339</td>
<td>0.2560</td>
<td>0.0453</td>
<td>0.4360</td>
</tr>
<tr>
<td>4000</td>
<td>0.1380</td>
<td>0.0332</td>
<td>0.2460</td>
<td>0.0444</td>
<td>0.3720</td>
</tr>
<tr>
<td>4250</td>
<td>0.1520</td>
<td>0.0349</td>
<td>0.2340</td>
<td>0.0433</td>
<td>0.4240</td>
</tr>
<tr>
<td>4500</td>
<td>0.1240</td>
<td>0.0315</td>
<td>0.2280</td>
<td>0.0427</td>
<td>0.3780</td>
</tr>
<tr>
<td>4750</td>
<td>0.1200</td>
<td>0.0310</td>
<td>0.2480</td>
<td>0.0445</td>
<td>0.4100</td>
</tr>
<tr>
<td>5000</td>
<td>0.1580</td>
<td>0.0356</td>
<td>0.2600</td>
<td>0.0456</td>
<td>0.4160</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
<td>------</td>
<td>-----</td>
</tr>
<tr>
<td>250</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.1300</td>
<td>0.0322</td>
<td>0.1440</td>
</tr>
<tr>
<td>500</td>
<td>0.1560</td>
<td>0.0353</td>
<td>0.1780</td>
<td>0.0377</td>
<td>0.1680</td>
</tr>
<tr>
<td>750</td>
<td>0.1460</td>
<td>0.0342</td>
<td>0.1660</td>
<td>0.0364</td>
<td>0.1560</td>
</tr>
<tr>
<td>1000</td>
<td>0.1360</td>
<td>0.0330</td>
<td>0.1440</td>
<td>0.0339</td>
<td>0.1620</td>
</tr>
<tr>
<td>1250</td>
<td>0.1440</td>
<td>0.0339</td>
<td>0.1980</td>
<td>0.0398</td>
<td>0.1900</td>
</tr>
<tr>
<td>1500</td>
<td>0.1760</td>
<td>0.0375</td>
<td>0.1540</td>
<td>0.0351</td>
<td>0.1700</td>
</tr>
<tr>
<td>1750</td>
<td>0.1580</td>
<td>0.0356</td>
<td>0.1740</td>
<td>0.0373</td>
<td>0.1720</td>
</tr>
<tr>
<td>2000</td>
<td>0.1340</td>
<td>0.0327</td>
<td>0.1560</td>
<td>0.0353</td>
<td>0.1520</td>
</tr>
<tr>
<td>2250</td>
<td>0.1500</td>
<td>0.0346</td>
<td>0.1420</td>
<td>0.0337</td>
<td>0.2120</td>
</tr>
<tr>
<td>2500</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.1620</td>
<td>0.0360</td>
<td>0.1700</td>
</tr>
<tr>
<td>2750</td>
<td>0.1080</td>
<td>0.0294</td>
<td>0.1640</td>
<td>0.0362</td>
<td>0.1760</td>
</tr>
<tr>
<td>3000</td>
<td>0.1540</td>
<td>0.0351</td>
<td>0.1600</td>
<td>0.0358</td>
<td>0.1740</td>
</tr>
<tr>
<td>3250</td>
<td>0.1600</td>
<td>0.0358</td>
<td>0.1600</td>
<td>0.0358</td>
<td>0.1820</td>
</tr>
<tr>
<td>3500</td>
<td>0.1380</td>
<td>0.0332</td>
<td>0.1440</td>
<td>0.0339</td>
<td>0.1820</td>
</tr>
<tr>
<td>3750</td>
<td>0.1180</td>
<td>0.0307</td>
<td>0.1700</td>
<td>0.0369</td>
<td>0.1480</td>
</tr>
<tr>
<td>4000</td>
<td>0.1400</td>
<td>0.0335</td>
<td>0.1500</td>
<td>0.0346</td>
<td>0.1620</td>
</tr>
<tr>
<td>4250</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.1560</td>
<td>0.0353</td>
<td>0.1600</td>
</tr>
<tr>
<td>4500</td>
<td>0.1360</td>
<td>0.0330</td>
<td>0.1500</td>
<td>0.0346</td>
<td>0.1820</td>
</tr>
<tr>
<td>4750</td>
<td>0.1840</td>
<td>0.0384</td>
<td>0.1300</td>
<td>0.0322</td>
<td>0.1760</td>
</tr>
<tr>
<td>5000</td>
<td>0.1000</td>
<td>0.0283</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.1520</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>-----------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>250</td>
<td>0.2840</td>
<td>0.0477</td>
<td>0.3560</td>
<td>0.0534</td>
<td>0.5420</td>
</tr>
<tr>
<td>500</td>
<td>0.2900</td>
<td>0.0482</td>
<td>0.4460</td>
<td>0.0597</td>
<td>0.5740</td>
</tr>
<tr>
<td>750</td>
<td>0.3120</td>
<td>0.0500</td>
<td>0.3900</td>
<td>0.0559</td>
<td>0.5780</td>
</tr>
<tr>
<td>1000</td>
<td>0.2620</td>
<td>0.0458</td>
<td>0.4040</td>
<td>0.0569</td>
<td>0.5980</td>
</tr>
<tr>
<td>1250</td>
<td>0.2880</td>
<td>0.0480</td>
<td>0.4700</td>
<td>0.0613</td>
<td>0.5700</td>
</tr>
<tr>
<td>1500</td>
<td>0.3180</td>
<td>0.0504</td>
<td>0.3780</td>
<td>0.0550</td>
<td>0.5920</td>
</tr>
<tr>
<td>1750</td>
<td>0.3360</td>
<td>0.0518</td>
<td>0.3820</td>
<td>0.0553</td>
<td>0.5800</td>
</tr>
<tr>
<td>2000</td>
<td>0.2460</td>
<td>0.0444</td>
<td>0.4400</td>
<td>0.0593</td>
<td>0.5360</td>
</tr>
<tr>
<td>2250</td>
<td>0.2640</td>
<td>0.0460</td>
<td>0.4060</td>
<td>0.0570</td>
<td>0.6380</td>
</tr>
<tr>
<td>2500</td>
<td>0.2700</td>
<td>0.0465</td>
<td>0.3980</td>
<td>0.0564</td>
<td>0.5720</td>
</tr>
<tr>
<td>2750</td>
<td>0.2680</td>
<td>0.0463</td>
<td>0.3860</td>
<td>0.0556</td>
<td>0.5280</td>
</tr>
<tr>
<td>3000</td>
<td>0.2900</td>
<td>0.0482</td>
<td>0.4040</td>
<td>0.0569</td>
<td>0.5640</td>
</tr>
<tr>
<td>3250</td>
<td>0.3080</td>
<td>0.0496</td>
<td>0.4020</td>
<td>0.0567</td>
<td>0.5220</td>
</tr>
<tr>
<td>3500</td>
<td>0.2860</td>
<td>0.0478</td>
<td>0.3680</td>
<td>0.0543</td>
<td>0.5620</td>
</tr>
<tr>
<td>3750</td>
<td>0.2620</td>
<td>0.0458</td>
<td>0.4260</td>
<td>0.0584</td>
<td>0.5800</td>
</tr>
<tr>
<td>4000</td>
<td>0.2780</td>
<td>0.0472</td>
<td>0.3960</td>
<td>0.0563</td>
<td>0.5340</td>
</tr>
<tr>
<td>4250</td>
<td>0.3000</td>
<td>0.0490</td>
<td>0.3900</td>
<td>0.0559</td>
<td>0.5840</td>
</tr>
<tr>
<td>4500</td>
<td>0.2600</td>
<td>0.0456</td>
<td>0.3780</td>
<td>0.0550</td>
<td>0.5600</td>
</tr>
<tr>
<td>4750</td>
<td>0.3040</td>
<td>0.0493</td>
<td>0.3780</td>
<td>0.0550</td>
<td>0.5860</td>
</tr>
<tr>
<td>5000</td>
<td>0.2580</td>
<td>0.0454</td>
<td>0.4080</td>
<td>0.0571</td>
<td>0.5680</td>
</tr>
<tr>
<td>Tree Size</td>
<td>k</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>-----------</td>
<td>----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>250</td>
<td>0</td>
<td>0</td>
<td>0.0520</td>
<td>0.0204</td>
<td>0.0900</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0.0420</td>
<td>0.0183</td>
<td>0.0920</td>
</tr>
<tr>
<td>750</td>
<td>0</td>
<td>0</td>
<td>0.0720</td>
<td>0.0240</td>
<td>0.0600</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0.0600</td>
<td>0.0219</td>
<td>0.0760</td>
</tr>
<tr>
<td>1250</td>
<td>0</td>
<td>0</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.0680</td>
</tr>
<tr>
<td>1500</td>
<td>0</td>
<td>0</td>
<td>0.0520</td>
<td>0.0204</td>
<td>0.0620</td>
</tr>
<tr>
<td>1750</td>
<td>0</td>
<td>0</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.0780</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>0.0580</td>
<td>0.0215</td>
<td>0.0640</td>
</tr>
<tr>
<td>2250</td>
<td>0</td>
<td>0</td>
<td>0.0360</td>
<td>0.0170</td>
<td>0.0680</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>0</td>
<td>0.0580</td>
<td>0.0215</td>
<td>0.0760</td>
</tr>
<tr>
<td>2750</td>
<td>0</td>
<td>0</td>
<td>0.0440</td>
<td>0.0188</td>
<td>0.0520</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
<td>0</td>
<td>0.0520</td>
<td>0.0204</td>
<td>0.0700</td>
</tr>
<tr>
<td>3250</td>
<td>0</td>
<td>0</td>
<td>0.0420</td>
<td>0.0183</td>
<td>0.0820</td>
</tr>
<tr>
<td>3500</td>
<td>0</td>
<td>0</td>
<td>0.0360</td>
<td>0.0170</td>
<td>0.0800</td>
</tr>
<tr>
<td>3750</td>
<td>0</td>
<td>0</td>
<td>0.0520</td>
<td>0.0204</td>
<td>0.0820</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>0.0460</td>
<td>0.0192</td>
<td>0.0700</td>
</tr>
<tr>
<td>4250</td>
<td>0</td>
<td>0</td>
<td>0.0480</td>
<td>0.0196</td>
<td>0.0700</td>
</tr>
<tr>
<td>4500</td>
<td>0</td>
<td>0</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.0760</td>
</tr>
<tr>
<td>4750</td>
<td>0</td>
<td>0</td>
<td>0.0380</td>
<td>0.0174</td>
<td>0.0840</td>
</tr>
<tr>
<td>5000</td>
<td>0</td>
<td>0</td>
<td>0.0460</td>
<td>0.0192</td>
<td>0.0760</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>-----------</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
<td>----</td>
</tr>
<tr>
<td>250</td>
<td>0</td>
<td>0</td>
<td>0.1620</td>
<td>0.0360</td>
<td>0.3020</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0.1460</td>
<td>0.0342</td>
<td>0.3640</td>
</tr>
<tr>
<td>750</td>
<td>0</td>
<td>0</td>
<td>0.1340</td>
<td>0.0327</td>
<td>0.3260</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0.1100</td>
<td>0.0297</td>
<td>0.3700</td>
</tr>
<tr>
<td>1250</td>
<td>0</td>
<td>0</td>
<td>0.1820</td>
<td>0.0382</td>
<td>0.3380</td>
</tr>
<tr>
<td>1500</td>
<td>0</td>
<td>0</td>
<td>0.1220</td>
<td>0.0312</td>
<td>0.3760</td>
</tr>
<tr>
<td>1750</td>
<td>0</td>
<td>0</td>
<td>0.1460</td>
<td>0.0342</td>
<td>0.3600</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>0.1640</td>
<td>0.0362</td>
<td>0.3280</td>
</tr>
<tr>
<td>2250</td>
<td>0</td>
<td>0</td>
<td>0.1520</td>
<td>0.0349</td>
<td>0.3640</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>0</td>
<td>0.1740</td>
<td>0.0373</td>
<td>0.2920</td>
</tr>
<tr>
<td>2750</td>
<td>0</td>
<td>0</td>
<td>0.1520</td>
<td>0.0349</td>
<td>0.3080</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
<td>0</td>
<td>0.1440</td>
<td>0.0339</td>
<td>0.3440</td>
</tr>
<tr>
<td>3250</td>
<td>0</td>
<td>0</td>
<td>0.1440</td>
<td>0.0339</td>
<td>0.2880</td>
</tr>
<tr>
<td>3500</td>
<td>0</td>
<td>0</td>
<td>0.1580</td>
<td>0.0356</td>
<td>0.3440</td>
</tr>
<tr>
<td>3750</td>
<td>0</td>
<td>0</td>
<td>0.1500</td>
<td>0.0346</td>
<td>0.3420</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>0.1500</td>
<td>0.0346</td>
<td>0.3140</td>
</tr>
<tr>
<td>4250</td>
<td>0</td>
<td>0</td>
<td>0.1160</td>
<td>0.0305</td>
<td>0.3580</td>
</tr>
<tr>
<td>4500</td>
<td>0</td>
<td>0</td>
<td>0.1380</td>
<td>0.0332</td>
<td>0.3320</td>
</tr>
<tr>
<td>4750</td>
<td>0</td>
<td>0</td>
<td>0.1700</td>
<td>0.0369</td>
<td>0.3620</td>
</tr>
<tr>
<td>5000</td>
<td>0</td>
<td>0</td>
<td>0.1580</td>
<td>0.0356</td>
<td>0.3760</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>----</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>250</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>750</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1250</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1500</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1750</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2250</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2750</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3250</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3500</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3750</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4250</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4500</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4750</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>tree size</td>
<td>k</td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----</td>
<td>----</td>
<td>---</td>
<td>----</td>
<td>---</td>
</tr>
<tr>
<td></td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>250</td>
<td>0.2840</td>
<td>0.0477</td>
<td>0.5700</td>
<td>0.0675</td>
<td>0.9340</td>
</tr>
<tr>
<td>500</td>
<td>0.2900</td>
<td>0.0482</td>
<td>0.6340</td>
<td>0.0712</td>
<td>1.0300</td>
</tr>
<tr>
<td>750</td>
<td>0.3120</td>
<td>0.0500</td>
<td>0.5960</td>
<td>0.0691</td>
<td>0.9640</td>
</tr>
<tr>
<td>1000</td>
<td>0.2620</td>
<td>0.0458</td>
<td>0.5740</td>
<td>0.0678</td>
<td>1.0440</td>
</tr>
<tr>
<td>1250</td>
<td>0.2880</td>
<td>0.0480</td>
<td>0.6920</td>
<td>0.0744</td>
<td>0.9760</td>
</tr>
<tr>
<td>1500</td>
<td>0.3180</td>
<td>0.0504</td>
<td>0.5520</td>
<td>0.0665</td>
<td>1.0300</td>
</tr>
<tr>
<td>1750</td>
<td>0.3360</td>
<td>0.0518</td>
<td>0.5680</td>
<td>0.0674</td>
<td>1.0180</td>
</tr>
<tr>
<td>2000</td>
<td>0.2460</td>
<td>0.0444</td>
<td>0.6620</td>
<td>0.0728</td>
<td>0.9280</td>
</tr>
<tr>
<td>2250</td>
<td>0.2640</td>
<td>0.0460</td>
<td>0.5940</td>
<td>0.0689</td>
<td>1.0700</td>
</tr>
<tr>
<td>2500</td>
<td>0.2700</td>
<td>0.0465</td>
<td>0.6300</td>
<td>0.0710</td>
<td>0.9400</td>
</tr>
<tr>
<td>2750</td>
<td>0.2680</td>
<td>0.0463</td>
<td>0.5820</td>
<td>0.0682</td>
<td>0.8880</td>
</tr>
<tr>
<td>3000</td>
<td>0.2900</td>
<td>0.0482</td>
<td>0.6000</td>
<td>0.0693</td>
<td>0.9780</td>
</tr>
<tr>
<td>3250</td>
<td>0.3080</td>
<td>0.0496</td>
<td>0.5880</td>
<td>0.0686</td>
<td>0.8920</td>
</tr>
<tr>
<td>3500</td>
<td>0.2860</td>
<td>0.0478</td>
<td>0.5620</td>
<td>0.0671</td>
<td>0.9860</td>
</tr>
<tr>
<td>3750</td>
<td>0.2620</td>
<td>0.0458</td>
<td>0.6280</td>
<td>0.0709</td>
<td>1.0040</td>
</tr>
<tr>
<td>4000</td>
<td>0.2780</td>
<td>0.0472</td>
<td>0.5920</td>
<td>0.0688</td>
<td>0.9180</td>
</tr>
<tr>
<td>4250</td>
<td>0.3000</td>
<td>0.0490</td>
<td>0.5540</td>
<td>0.0666</td>
<td>1.0120</td>
</tr>
<tr>
<td>4500</td>
<td>0.2600</td>
<td>0.0456</td>
<td>0.5800</td>
<td>0.0681</td>
<td>0.9680</td>
</tr>
<tr>
<td>4750</td>
<td>0.3040</td>
<td>0.0493</td>
<td>0.5860</td>
<td>0.0685</td>
<td>1.0320</td>
</tr>
<tr>
<td>5000</td>
<td>0.2580</td>
<td>0.0454</td>
<td>0.6120</td>
<td>0.0700</td>
<td>1.0200</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>----------</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>250</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.0960</td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1280</td>
</tr>
<tr>
<td>750</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1120</td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1380</td>
</tr>
<tr>
<td>1250</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1220</td>
</tr>
<tr>
<td>1500</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1360</td>
</tr>
<tr>
<td>1750</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>1.1300</td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1080</td>
</tr>
<tr>
<td>2250</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1300</td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1120</td>
</tr>
<tr>
<td>2750</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.0840</td>
</tr>
<tr>
<td>3000</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1280</td>
</tr>
<tr>
<td>3250</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1060</td>
</tr>
<tr>
<td>3500</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1180</td>
</tr>
<tr>
<td>3750</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1240</td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1100</td>
</tr>
<tr>
<td>4250</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1160</td>
</tr>
<tr>
<td>4500</td>
<td>0</td>
<td>0</td>
<td>1.000</td>
<td>0</td>
<td>2.1100</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>-----------</td>
<td>------</td>
<td>----</td>
<td>------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>250</td>
<td>1.00</td>
<td>0</td>
<td>2.3760</td>
<td>0.0669</td>
<td>3.7900</td>
</tr>
<tr>
<td>500</td>
<td>1.00</td>
<td>0</td>
<td>2.3340</td>
<td>0.0642</td>
<td>3.9489</td>
</tr>
<tr>
<td>750</td>
<td>1.00</td>
<td>0</td>
<td>2.3400</td>
<td>0.0628</td>
<td>3.8240</td>
</tr>
<tr>
<td>1000</td>
<td>1.00</td>
<td>0</td>
<td>2.2800</td>
<td>0.0581</td>
<td>3.9540</td>
</tr>
<tr>
<td>1250</td>
<td>1.00</td>
<td>0</td>
<td>2.4040</td>
<td>0.0696</td>
<td>3.8660</td>
</tr>
<tr>
<td>1500</td>
<td>1.00</td>
<td>0</td>
<td>2.2960</td>
<td>0.0602</td>
<td>3.9500</td>
</tr>
<tr>
<td>1750</td>
<td>1.00</td>
<td>0</td>
<td>2.3320</td>
<td>0.0637</td>
<td>3.9280</td>
</tr>
<tr>
<td>2000</td>
<td>1.00</td>
<td>0</td>
<td>2.3860</td>
<td>0.0673</td>
<td>3.8280</td>
</tr>
<tr>
<td>2250</td>
<td>1.00</td>
<td>0</td>
<td>2.3400</td>
<td>0.0651</td>
<td>3.9260</td>
</tr>
<tr>
<td>2500</td>
<td>1.00</td>
<td>0</td>
<td>2.4060</td>
<td>0.0687</td>
<td>3.7720</td>
</tr>
<tr>
<td>2750</td>
<td>1.00</td>
<td>0</td>
<td>2.3480</td>
<td>0.0652</td>
<td>3.7520</td>
</tr>
<tr>
<td>3000</td>
<td>1.00</td>
<td>0</td>
<td>2.3400</td>
<td>0.0641</td>
<td>3.8860</td>
</tr>
<tr>
<td>3250</td>
<td>1.00</td>
<td>0</td>
<td>2.3300</td>
<td>0.0639</td>
<td>3.7640</td>
</tr>
<tr>
<td>3500</td>
<td>1.00</td>
<td>0</td>
<td>2.3520</td>
<td>0.0660</td>
<td>3.8860</td>
</tr>
<tr>
<td>3750</td>
<td>1.00</td>
<td>0</td>
<td>2.3520</td>
<td>0.0651</td>
<td>3.8900</td>
</tr>
<tr>
<td>4000</td>
<td>1.00</td>
<td>0</td>
<td>2.3460</td>
<td>0.0650</td>
<td>3.8080</td>
</tr>
<tr>
<td>4250</td>
<td>1.00</td>
<td>0</td>
<td>2.2800</td>
<td>0.0589</td>
<td>3.9020</td>
</tr>
<tr>
<td>4500</td>
<td>1.00</td>
<td>0</td>
<td>2.3400</td>
<td>0.0633</td>
<td>3.8500</td>
</tr>
<tr>
<td>4750</td>
<td>1.00</td>
<td>0</td>
<td>2.3780</td>
<td>0.0679</td>
<td>3.9480</td>
</tr>
<tr>
<td>5000</td>
<td>1.00</td>
<td>0</td>
<td>2.3620</td>
<td>0.0662</td>
<td>3.9620</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>250</td>
<td>8.2480</td>
<td>0.2234</td>
<td>7.5460</td>
<td>0.1733</td>
<td>7.3780</td>
</tr>
<tr>
<td>500</td>
<td>9.3680</td>
<td>0.1788</td>
<td>8.6240</td>
<td>0.2086</td>
<td>8.3100</td>
</tr>
<tr>
<td>750</td>
<td>10.1000</td>
<td>0.2294</td>
<td>9.3460</td>
<td>0.1819</td>
<td>8.9260</td>
</tr>
<tr>
<td>1000</td>
<td>10.6580</td>
<td>0.2439</td>
<td>9.6940</td>
<td>0.1751</td>
<td>9.3220</td>
</tr>
<tr>
<td>1250</td>
<td>10.8900</td>
<td>0.2404</td>
<td>10.1860</td>
<td>0.1883</td>
<td>9.7360</td>
</tr>
<tr>
<td>1500</td>
<td>11.2660</td>
<td>0.2525</td>
<td>10.2560</td>
<td>0.1795</td>
<td>9.8060</td>
</tr>
<tr>
<td>1750</td>
<td>11.6860</td>
<td>0.3692</td>
<td>10.7640</td>
<td>0.1679</td>
<td>10.2420</td>
</tr>
<tr>
<td>2000</td>
<td>12.0800</td>
<td>0.7063</td>
<td>10.9840</td>
<td>0.4718</td>
<td>10.3860</td>
</tr>
<tr>
<td>2250</td>
<td>11.5880</td>
<td>0.2742</td>
<td>10.8980</td>
<td>0.1912</td>
<td>10.5080</td>
</tr>
<tr>
<td>2500</td>
<td>11.9900</td>
<td>0.2376</td>
<td>11.1600</td>
<td>0.1814</td>
<td>10.5780</td>
</tr>
<tr>
<td>2750</td>
<td>12.1360</td>
<td>0.2485</td>
<td>11.2500</td>
<td>0.1821</td>
<td>10.7520</td>
</tr>
<tr>
<td>3000</td>
<td>13.7140</td>
<td>1.3862</td>
<td>11.4580</td>
<td>0.1502</td>
<td>11.2280</td>
</tr>
<tr>
<td>3250</td>
<td>12.7680</td>
<td>0.2192</td>
<td>11.9340</td>
<td>0.2322</td>
<td>11.1040</td>
</tr>
<tr>
<td>3500</td>
<td>12.9480</td>
<td>0.2407</td>
<td>12.0540</td>
<td>0.5978</td>
<td>11.2100</td>
</tr>
<tr>
<td>3750</td>
<td>13.0100</td>
<td>0.5124</td>
<td>11.8740</td>
<td>0.1720</td>
<td>11.5000</td>
</tr>
<tr>
<td>4000</td>
<td>12.6040</td>
<td>0.2341</td>
<td>11.9360</td>
<td>0.4144</td>
<td>11.2080</td>
</tr>
<tr>
<td>4250</td>
<td>13.3700</td>
<td>0.6109</td>
<td>11.9080</td>
<td>0.1811</td>
<td>11.6280</td>
</tr>
<tr>
<td>4500</td>
<td>12.5720</td>
<td>0.2514</td>
<td>11.7420</td>
<td>0.1988</td>
<td>11.4860</td>
</tr>
<tr>
<td>4750</td>
<td>12.3460</td>
<td>0.3220</td>
<td>11.6680</td>
<td>0.2114</td>
<td>11.1160</td>
</tr>
<tr>
<td>5000</td>
<td>9.5880</td>
<td>0.5280</td>
<td>9.6820</td>
<td>0.4053</td>
<td>10.6220</td>
</tr>
<tr>
<td>tree size</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td></td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>250</td>
<td>0.9960</td>
<td>0.0893</td>
<td>1.1500</td>
<td>0.0959</td>
<td>1.1940</td>
</tr>
<tr>
<td>500</td>
<td>1.0040</td>
<td>0.0896</td>
<td>1.1860</td>
<td>0.0974</td>
<td>1.2300</td>
</tr>
<tr>
<td>750</td>
<td>0.9660</td>
<td>0.0879</td>
<td>1.1440</td>
<td>0.0957</td>
<td>1.2520</td>
</tr>
<tr>
<td>1000</td>
<td>1.2960</td>
<td>0.1018</td>
<td>1.1100</td>
<td>0.0942</td>
<td>1.2320</td>
</tr>
<tr>
<td>1250</td>
<td>1.0280</td>
<td>0.0907</td>
<td>1.0660</td>
<td>0.0923</td>
<td>1.1760</td>
</tr>
<tr>
<td>1500</td>
<td>1.0760</td>
<td>0.0928</td>
<td>1.0980</td>
<td>0.0937</td>
<td>1.1480</td>
</tr>
<tr>
<td>1750</td>
<td>1.5300</td>
<td>0.1106</td>
<td>1.0720</td>
<td>0.0926</td>
<td>1.3060</td>
</tr>
<tr>
<td>2000</td>
<td>1.4060</td>
<td>0.1061</td>
<td>1.0340</td>
<td>0.0910</td>
<td>1.2720</td>
</tr>
<tr>
<td>2250</td>
<td>1.0340</td>
<td>0.0910</td>
<td>1.0980</td>
<td>0.0937</td>
<td>1.1740</td>
</tr>
<tr>
<td>2500</td>
<td>1.0080</td>
<td>0.0898</td>
<td>1.0680</td>
<td>0.0924</td>
<td>1.1480</td>
</tr>
<tr>
<td>2750</td>
<td>0.9480</td>
<td>0.0871</td>
<td>1.0780</td>
<td>0.0929</td>
<td>1.1620</td>
</tr>
<tr>
<td>3000</td>
<td>0.9100</td>
<td>0.0853</td>
<td>1.0460</td>
<td>0.0915</td>
<td>1.1800</td>
</tr>
<tr>
<td>3250</td>
<td>0.9080</td>
<td>0.0852</td>
<td>0.9480</td>
<td>0.0871</td>
<td>1.1560</td>
</tr>
<tr>
<td>3500</td>
<td>0.9540</td>
<td>0.0874</td>
<td>1.0460</td>
<td>0.0915</td>
<td>1.0940</td>
</tr>
<tr>
<td>3750</td>
<td>0.9780</td>
<td>0.0885</td>
<td>1.0220</td>
<td>0.0904</td>
<td>1.1580</td>
</tr>
<tr>
<td>4000</td>
<td>0.9540</td>
<td>0.0874</td>
<td>1.0580</td>
<td>0.0920</td>
<td>1.0400</td>
</tr>
<tr>
<td>4250</td>
<td>0.9600</td>
<td>0.0876</td>
<td>1.0300</td>
<td>0.0908</td>
<td>1.1020</td>
</tr>
<tr>
<td>4500</td>
<td>0.9340</td>
<td>0.0864</td>
<td>1.0220</td>
<td>0.0904</td>
<td>1.1800</td>
</tr>
<tr>
<td>4750</td>
<td>0.8920</td>
<td>0.0845</td>
<td>1.0520</td>
<td>0.0917</td>
<td>1.1180</td>
</tr>
<tr>
<td>5000</td>
<td>0.8780</td>
<td>0.0838</td>
<td>1.0820</td>
<td>0.0930</td>
<td>1.0760</td>
</tr>
<tr>
<td>tree size</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>250</td>
<td>0.3000</td>
<td>0.4090</td>
<td>0.3160</td>
<td>0.0503</td>
<td>0.3100</td>
</tr>
<tr>
<td>500</td>
<td>0.3180</td>
<td>0.0504</td>
<td>0.3100</td>
<td>0.0498</td>
<td>0.3300</td>
</tr>
<tr>
<td>750</td>
<td>0.9660</td>
<td>0.0879</td>
<td>0.2940</td>
<td>0.0485</td>
<td>0.3080</td>
</tr>
<tr>
<td>1000</td>
<td>0.3460</td>
<td>0.0526</td>
<td>0.3260</td>
<td>0.0511</td>
<td>0.3360</td>
</tr>
<tr>
<td>1250</td>
<td>0.2920</td>
<td>0.0483</td>
<td>0.2920</td>
<td>0.0483</td>
<td>0.2820</td>
</tr>
<tr>
<td>1500</td>
<td>0.2900</td>
<td>0.0482</td>
<td>0.2920</td>
<td>0.0483</td>
<td>0.3060</td>
</tr>
<tr>
<td>1750</td>
<td>0.2780</td>
<td>0.0472</td>
<td>0.3180</td>
<td>0.0504</td>
<td>0.2940</td>
</tr>
<tr>
<td>2000</td>
<td>0.3060</td>
<td>0.0495</td>
<td>0.3100</td>
<td>0.0498</td>
<td>0.3120</td>
</tr>
<tr>
<td>2250</td>
<td>0.3000</td>
<td>0.0490</td>
<td>0.3280</td>
<td>0.0512</td>
<td>0.3240</td>
</tr>
<tr>
<td>2500</td>
<td>0.2840</td>
<td>0.0477</td>
<td>0.3200</td>
<td>0.0506</td>
<td>0.3000</td>
</tr>
<tr>
<td>2750</td>
<td>0.2940</td>
<td>0.0485</td>
<td>0.3100</td>
<td>0.0498</td>
<td>0.3020</td>
</tr>
<tr>
<td>3000</td>
<td>0.3000</td>
<td>0.0490</td>
<td>0.3280</td>
<td>0.0512</td>
<td>0.3000</td>
</tr>
<tr>
<td>3250</td>
<td>0.2860</td>
<td>0.0478</td>
<td>0.3240</td>
<td>0.0509</td>
<td>0.3140</td>
</tr>
<tr>
<td>3500</td>
<td>0.2960</td>
<td>0.0487</td>
<td>0.2760</td>
<td>0.0470</td>
<td>0.2820</td>
</tr>
<tr>
<td>3750</td>
<td>0.2740</td>
<td>0.0468</td>
<td>0.2800</td>
<td>0.0473</td>
<td>0.2680</td>
</tr>
<tr>
<td>4000</td>
<td>0.2740</td>
<td>0.0468</td>
<td>0.2840</td>
<td>0.0477</td>
<td>0.3640</td>
</tr>
<tr>
<td>4250</td>
<td>0.2940</td>
<td>0.0485</td>
<td>0.2760</td>
<td>0.0578</td>
<td>0.3160</td>
</tr>
<tr>
<td>4500</td>
<td>0.3120</td>
<td>0.0500</td>
<td>0.2880</td>
<td>0.0480</td>
<td>0.2760</td>
</tr>
<tr>
<td>4750</td>
<td>0.3220</td>
<td>0.0508</td>
<td>0.2460</td>
<td>0.0444</td>
<td>0.2780</td>
</tr>
<tr>
<td>5000</td>
<td>0.3120</td>
<td>0.0500</td>
<td>0.2420</td>
<td>0.0440</td>
<td>0.2600</td>
</tr>
<tr>
<td>tree size</td>
<td>k</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>-----------</td>
<td>---</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>250</td>
<td></td>
<td>0.2960</td>
<td>0.0487</td>
<td>0.4660</td>
<td>0.0611</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>0.3220</td>
<td>0.0508</td>
<td>0.4960</td>
<td>0.0630</td>
</tr>
<tr>
<td>750</td>
<td></td>
<td>0.2840</td>
<td>0.0477</td>
<td>0.4380</td>
<td>0.0592</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>0.6420</td>
<td>0.0717</td>
<td>0.4360</td>
<td>0.0591</td>
</tr>
<tr>
<td>1250</td>
<td></td>
<td>0.3200</td>
<td>0.0506</td>
<td>0.3580</td>
<td>0.0535</td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td>0.3660</td>
<td>0.0541</td>
<td>0.3900</td>
<td>0.0559</td>
</tr>
<tr>
<td>1750</td>
<td></td>
<td>0.8080</td>
<td>0.0804</td>
<td>0.3900</td>
<td>0.0559</td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td>0.7120</td>
<td>0.0755</td>
<td>0.3440</td>
<td>0.0525</td>
</tr>
<tr>
<td>2250</td>
<td></td>
<td>0.3340</td>
<td>0.0517</td>
<td>0.4260</td>
<td>0.0584</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>0.2920</td>
<td>0.0483</td>
<td>0.3880</td>
<td>0.0557</td>
</tr>
<tr>
<td>2750</td>
<td></td>
<td>0.2420</td>
<td>0.0440</td>
<td>0.3880</td>
<td>0.0557</td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td>0.2100</td>
<td>0.0410</td>
<td>0.3740</td>
<td>0.0547</td>
</tr>
<tr>
<td>3250</td>
<td></td>
<td>0.1940</td>
<td>0.0394</td>
<td>0.2720</td>
<td>0.0466</td>
</tr>
<tr>
<td>3500</td>
<td></td>
<td>0.2500</td>
<td>0.0447</td>
<td>0.3220</td>
<td>0.0508</td>
</tr>
<tr>
<td>3750</td>
<td></td>
<td>0.2520</td>
<td>0.0449</td>
<td>0.3020</td>
<td>0.0492</td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td>0.2280</td>
<td>0.0427</td>
<td>0.3420</td>
<td>0.0523</td>
</tr>
<tr>
<td>4250</td>
<td></td>
<td>0.2540</td>
<td>0.0451</td>
<td>0.3060</td>
<td>0.0495</td>
</tr>
<tr>
<td>4500</td>
<td></td>
<td>0.2460</td>
<td>0.0444</td>
<td>0.3100</td>
<td>0.0498</td>
</tr>
<tr>
<td>4750</td>
<td></td>
<td>0.2140</td>
<td>0.0414</td>
<td>0.2980</td>
<td>0.0488</td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td>0.1900</td>
<td>0.0390</td>
<td>0.3240</td>
<td>0.0509</td>
</tr>
<tr>
<td>Tree Size</td>
<td>k = 2</td>
<td>k = 3</td>
<td>k = 4</td>
<td>k = 5</td>
<td>k = 6</td>
</tr>
<tr>
<td>-----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>250</td>
<td>0.4320</td>
<td>0.4380</td>
<td>0.4640</td>
<td>0.4760</td>
<td>0.4140</td>
</tr>
<tr>
<td>500</td>
<td>0.4480</td>
<td>0.4560</td>
<td>0.4600</td>
<td>0.4560</td>
<td>0.4540</td>
</tr>
<tr>
<td>750</td>
<td>0.4400</td>
<td>0.4780</td>
<td>0.4620</td>
<td>0.4400</td>
<td>0.4880</td>
</tr>
<tr>
<td>1000</td>
<td>0.4100</td>
<td>0.4580</td>
<td>0.4040</td>
<td>0.4060</td>
<td>0.4480</td>
</tr>
<tr>
<td>1250</td>
<td>0.4260</td>
<td>0.4520</td>
<td>0.4720</td>
<td>0.4560</td>
<td>0.4300</td>
</tr>
<tr>
<td>1500</td>
<td>0.4420</td>
<td>0.4360</td>
<td>0.4380</td>
<td>0.4740</td>
<td>0.4620</td>
</tr>
<tr>
<td>1750</td>
<td>0.4780</td>
<td>0.4540</td>
<td>0.4720</td>
<td>0.4680</td>
<td>0.4380</td>
</tr>
<tr>
<td>2000</td>
<td>0.4480</td>
<td>0.4520</td>
<td>0.4460</td>
<td>0.4620</td>
<td>0.4500</td>
</tr>
<tr>
<td>2250</td>
<td>0.4300</td>
<td>0.4340</td>
<td>0.4160</td>
<td>0.4640</td>
<td>0.4680</td>
</tr>
<tr>
<td>2500</td>
<td>0.4480</td>
<td>0.4400</td>
<td>0.4080</td>
<td>0.4520</td>
<td>0.5080</td>
</tr>
<tr>
<td>2750</td>
<td>0.4500</td>
<td>0.4240</td>
<td>0.4040</td>
<td>0.4580</td>
<td>0.4800</td>
</tr>
<tr>
<td>3000</td>
<td>0.4620</td>
<td>0.4160</td>
<td>0.4780</td>
<td>0.4360</td>
<td>0.3960</td>
</tr>
<tr>
<td>3250</td>
<td>0.4900</td>
<td>0.4740</td>
<td>0.4240</td>
<td>0.4520</td>
<td>0.4420</td>
</tr>
<tr>
<td>3500</td>
<td>0.4800</td>
<td>0.4520</td>
<td>0.4820</td>
<td>0.4300</td>
<td>0.4520</td>
</tr>
<tr>
<td>3750</td>
<td>0.4540</td>
<td>0.4180</td>
<td>0.5140</td>
<td>0.4140</td>
<td>0.4680</td>
</tr>
<tr>
<td>4000</td>
<td>0.3960</td>
<td>0.4160</td>
<td>0.4120</td>
<td>0.4800</td>
<td>0.4260</td>
</tr>
<tr>
<td>4250</td>
<td>0.4340</td>
<td>0.4480</td>
<td>0.4640</td>
<td>0.4300</td>
<td>0.4340</td>
</tr>
<tr>
<td>4500</td>
<td>0.3680</td>
<td>0.3940</td>
<td>0.4780</td>
<td>0.4180</td>
<td>0.4560</td>
</tr>
<tr>
<td>4750</td>
<td>0.3920</td>
<td>0.4260</td>
<td>0.4120</td>
<td>0.4400</td>
<td>0.4400</td>
</tr>
<tr>
<td>5000</td>
<td>0.3060</td>
<td>0.3540</td>
<td>0.3600</td>
<td>0.4520</td>
<td>0.4480</td>
</tr>
<tr>
<td>tree size</td>
<td>k</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>----------</td>
<td>-----</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>250</td>
<td>0.1480</td>
<td>0.0920</td>
<td>0.1000</td>
<td>0.0860</td>
<td>0.0900</td>
</tr>
<tr>
<td>500</td>
<td>0.1400</td>
<td>0.1140</td>
<td>0.1040</td>
<td>0.1040</td>
<td>0.0920</td>
</tr>
<tr>
<td>750</td>
<td>0.1460</td>
<td>0.0940</td>
<td>0.0820</td>
<td>0.1040</td>
<td>0.0940</td>
</tr>
<tr>
<td>1000</td>
<td>0.1500</td>
<td>0.1100</td>
<td>0.1240</td>
<td>0.1200</td>
<td>0.0980</td>
</tr>
<tr>
<td>1250</td>
<td>0.1080</td>
<td>0.1260</td>
<td>0.1080</td>
<td>0.0940</td>
<td>0.1100</td>
</tr>
<tr>
<td>1500</td>
<td>0.1120</td>
<td>0.1100</td>
<td>0.1080</td>
<td>0.0860</td>
<td>0.0800</td>
</tr>
<tr>
<td>1750</td>
<td>0.1160</td>
<td>0.1140</td>
<td>0.0880</td>
<td>0.0900</td>
<td>0.1120</td>
</tr>
<tr>
<td>2000</td>
<td>0.1380</td>
<td>0.1440</td>
<td>0.1020</td>
<td>0.0960</td>
<td>0.1060</td>
</tr>
<tr>
<td>2250</td>
<td>0.1240</td>
<td>0.1440</td>
<td>0.1180</td>
<td>0.1080</td>
<td>0.1000</td>
</tr>
<tr>
<td>2500</td>
<td>0.1140</td>
<td>0.0920</td>
<td>0.1160</td>
<td>0.1240</td>
<td>0.1000</td>
</tr>
<tr>
<td>2750</td>
<td>0.1200</td>
<td>0.1280</td>
<td>0.0900</td>
<td>0.0980</td>
<td>0.0920</td>
</tr>
<tr>
<td>3000</td>
<td>0.1480</td>
<td>0.1220</td>
<td>0.1120</td>
<td>0.1080</td>
<td>0.1240</td>
</tr>
<tr>
<td>3250</td>
<td>0.1340</td>
<td>0.1640</td>
<td>0.1200</td>
<td>0.1100</td>
<td>0.1160</td>
</tr>
<tr>
<td>3500</td>
<td>0.1500</td>
<td>0.1140</td>
<td>0.1060</td>
<td>0.0920</td>
<td>0.1120</td>
</tr>
<tr>
<td>3750</td>
<td>0.1240</td>
<td>0.1380</td>
<td>0.1100</td>
<td>0.1020</td>
<td>0.1100</td>
</tr>
<tr>
<td>4000</td>
<td>0.1380</td>
<td>0.0920</td>
<td>0.1200</td>
<td>0.1080</td>
<td>0.0920</td>
</tr>
<tr>
<td>4250</td>
<td>0.1340</td>
<td>0.1140</td>
<td>0.0960</td>
<td>0.0960</td>
<td>0.0880</td>
</tr>
<tr>
<td>4500</td>
<td>0.1300</td>
<td>0.1140</td>
<td>0.1000</td>
<td>0.0940</td>
<td>0.0720</td>
</tr>
<tr>
<td>4750</td>
<td>0.1320</td>
<td>0.0700</td>
<td>0.0900</td>
<td>0.0720</td>
<td>0.0600</td>
</tr>
<tr>
<td>5000</td>
<td>0.1040</td>
<td>0.0540</td>
<td>0.0960</td>
<td>0.0640</td>
<td>0.1100</td>
</tr>
<tr>
<td>Tree Size</td>
<td>k</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>-----------</td>
<td>----</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>250</td>
<td></td>
<td>0.4200</td>
<td>0.4700</td>
<td>0.4360</td>
<td>0.4380</td>
</tr>
<tr>
<td>500</td>
<td></td>
<td>0.4120</td>
<td>0.4300</td>
<td>0.4360</td>
<td>0.4400</td>
</tr>
<tr>
<td>750</td>
<td></td>
<td>0.4140</td>
<td>0.4280</td>
<td>0.4560</td>
<td>0.4560</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td>0.4400</td>
<td>0.4320</td>
<td>0.4720</td>
<td>0.4740</td>
</tr>
<tr>
<td>1250</td>
<td></td>
<td>0.4660</td>
<td>0.4220</td>
<td>0.4200</td>
<td>0.4500</td>
</tr>
<tr>
<td>1500</td>
<td></td>
<td>0.4460</td>
<td>0.4540</td>
<td>0.4540</td>
<td>0.4400</td>
</tr>
<tr>
<td>1750</td>
<td></td>
<td>0.4060</td>
<td>0.4320</td>
<td>0.4400</td>
<td>0.4420</td>
</tr>
<tr>
<td>2000</td>
<td></td>
<td>0.4140</td>
<td>0.4040</td>
<td>0.4520</td>
<td>0.4420</td>
</tr>
<tr>
<td>2250</td>
<td></td>
<td>0.4460</td>
<td>0.4220</td>
<td>0.4660</td>
<td>0.4280</td>
</tr>
<tr>
<td>2500</td>
<td></td>
<td>0.4380</td>
<td>0.4680</td>
<td>0.4760</td>
<td>0.4240</td>
</tr>
<tr>
<td>2750</td>
<td></td>
<td>0.4300</td>
<td>0.4480</td>
<td>0.5060</td>
<td>0.4440</td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td>0.3900</td>
<td>0.4620</td>
<td>0.4100</td>
<td>0.4560</td>
</tr>
<tr>
<td>3250</td>
<td></td>
<td>0.3760</td>
<td>0.3620</td>
<td>0.4560</td>
<td>0.4380</td>
</tr>
<tr>
<td>3500</td>
<td></td>
<td>0.3700</td>
<td>0.4340</td>
<td>0.4120</td>
<td>0.4780</td>
</tr>
<tr>
<td>3750</td>
<td></td>
<td>0.4220</td>
<td>0.4440</td>
<td>0.3760</td>
<td>0.4840</td>
</tr>
<tr>
<td>4000</td>
<td></td>
<td>0.4660</td>
<td>0.4920</td>
<td>0.4680</td>
<td>0.4120</td>
</tr>
<tr>
<td>4250</td>
<td></td>
<td>0.4320</td>
<td>0.4380</td>
<td>0.4400</td>
<td>0.4740</td>
</tr>
<tr>
<td>4500</td>
<td></td>
<td>0.5020</td>
<td>0.4920</td>
<td>0.4220</td>
<td>0.4880</td>
</tr>
<tr>
<td>4750</td>
<td></td>
<td>0.4760</td>
<td>0.5040</td>
<td>0.4980</td>
<td>0.4880</td>
</tr>
<tr>
<td>5000</td>
<td></td>
<td>0.5900</td>
<td>0.5920</td>
<td>0.5440</td>
<td>0.4840</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>250</td>
<td>9.8820</td>
<td>0.1604</td>
<td>9.4840</td>
<td>0.1239</td>
<td>9.3420</td>
</tr>
<tr>
<td>500</td>
<td>10.9240</td>
<td>0.1688</td>
<td>10.4520</td>
<td>0.1302</td>
<td>10.3500</td>
</tr>
<tr>
<td>750</td>
<td>11.6700</td>
<td>0.1638</td>
<td>11.3040</td>
<td>0.1302</td>
<td>11.0620</td>
</tr>
<tr>
<td>1000</td>
<td>12.1560</td>
<td>0.1698</td>
<td>11.7200</td>
<td>0.1313</td>
<td>11.4120</td>
</tr>
<tr>
<td>1250</td>
<td>12.4000</td>
<td>0.1673</td>
<td>11.9980</td>
<td>0.1354</td>
<td>11.7080</td>
</tr>
<tr>
<td>1500</td>
<td>12.6400</td>
<td>0.1748</td>
<td>12.2540</td>
<td>0.1314</td>
<td>12.0560</td>
</tr>
<tr>
<td>1750</td>
<td>13.0520</td>
<td>0.1775</td>
<td>12.5100</td>
<td>0.1401</td>
<td>12.3020</td>
</tr>
<tr>
<td>2000</td>
<td>13.2600</td>
<td>0.1880</td>
<td>12.8240</td>
<td>0.1512</td>
<td>12.5160</td>
</tr>
<tr>
<td>2250</td>
<td>13.4920</td>
<td>0.1778</td>
<td>13.0540</td>
<td>0.1434</td>
<td>12.7120</td>
</tr>
<tr>
<td>2500</td>
<td>13.6220</td>
<td>0.1806</td>
<td>13.1800</td>
<td>0.1417</td>
<td>12.7760</td>
</tr>
<tr>
<td>2750</td>
<td>13.7200</td>
<td>0.1863</td>
<td>13.2300</td>
<td>0.1443</td>
<td>12.9720</td>
</tr>
<tr>
<td>3000</td>
<td>13.9300</td>
<td>0.1833</td>
<td>13.4120</td>
<td>0.1452</td>
<td>13.1100</td>
</tr>
<tr>
<td>3250</td>
<td>13.9960</td>
<td>0.1864</td>
<td>13.4620</td>
<td>0.1468</td>
<td>13.2440</td>
</tr>
<tr>
<td>3500</td>
<td>14.2380</td>
<td>0.1869</td>
<td>13.7180</td>
<td>0.1472</td>
<td>13.3660</td>
</tr>
<tr>
<td>3750</td>
<td>14.2480</td>
<td>0.1969</td>
<td>13.7680</td>
<td>0.1525</td>
<td>13.4860</td>
</tr>
<tr>
<td>4000</td>
<td>14.3660</td>
<td>0.1902</td>
<td>13.7780</td>
<td>0.1467</td>
<td>13.4140</td>
</tr>
<tr>
<td>4250</td>
<td>14.3440</td>
<td>0.1878</td>
<td>13.9100</td>
<td>0.1507</td>
<td>13.5120</td>
</tr>
<tr>
<td>4500</td>
<td>14.4560</td>
<td>0.2007</td>
<td>13.9900</td>
<td>0.1510</td>
<td>13.6980</td>
</tr>
<tr>
<td>4750</td>
<td>14.6560</td>
<td>0.1847</td>
<td>14.1080</td>
<td>0.1488</td>
<td>13.7980</td>
</tr>
<tr>
<td>5000</td>
<td>14.7420</td>
<td>0.1975</td>
<td>14.2060</td>
<td>0.1503</td>
<td>13.8700</td>
</tr>
</tbody>
</table>
Table 4.2.21  BB-tree insertion statistics  AVERAGE SINGLE ROTATION

<table>
<thead>
<tr>
<th>tree size</th>
<th>AV</th>
<th>CI</th>
<th>AV</th>
<th>CI</th>
<th>AV</th>
<th>CI</th>
<th>AV</th>
<th>CI</th>
</tr>
</thead>
<tbody>
<tr>
<td>250</td>
<td>0.0900</td>
<td>0.0268</td>
<td>0.1300</td>
<td>0.0322</td>
<td>0.1580</td>
<td>0.0356</td>
<td>0.3280</td>
<td>0.0512</td>
</tr>
<tr>
<td>500</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.1200</td>
<td>0.0310</td>
<td>0.1660</td>
<td>0.0364</td>
<td>0.2960</td>
<td>0.0487</td>
</tr>
<tr>
<td>750</td>
<td>0.0600</td>
<td>0.0219</td>
<td>0.1400</td>
<td>0.0335</td>
<td>0.1820</td>
<td>0.0382</td>
<td>0.2660</td>
<td>0.0461</td>
</tr>
<tr>
<td>1000</td>
<td>0.0780</td>
<td>0.0250</td>
<td>0.1200</td>
<td>0.0310</td>
<td>0.1620</td>
<td>0.0360</td>
<td>0.2680</td>
<td>0.0463</td>
</tr>
<tr>
<td>1250</td>
<td>0.0860</td>
<td>0.0262</td>
<td>0.1460</td>
<td>0.0342</td>
<td>0.1780</td>
<td>0.0377</td>
<td>0.2920</td>
<td>0.0483</td>
</tr>
<tr>
<td>1500</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.1400</td>
<td>0.0335</td>
<td>0.1700</td>
<td>0.0369</td>
<td>0.2780</td>
<td>0.0472</td>
</tr>
<tr>
<td>1750</td>
<td>0.0840</td>
<td>0.0259</td>
<td>0.1200</td>
<td>0.0310</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.3060</td>
<td>0.0495</td>
</tr>
<tr>
<td>2000</td>
<td>0.0600</td>
<td>0.0219</td>
<td>0.1440</td>
<td>0.0339</td>
<td>0.1920</td>
<td>0.0392</td>
<td>0.2980</td>
<td>0.0488</td>
</tr>
<tr>
<td>2250</td>
<td>0.0680</td>
<td>0.0294</td>
<td>0.1520</td>
<td>0.0349</td>
<td>0.1640</td>
<td>0.0362</td>
<td>0.3080</td>
<td>0.0496</td>
</tr>
<tr>
<td>2500</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.1280</td>
<td>0.0320</td>
<td>0.1460</td>
<td>0.0342</td>
<td>0.2680</td>
<td>0.0463</td>
</tr>
<tr>
<td>2750</td>
<td>0.1000</td>
<td>0.0283</td>
<td>0.1240</td>
<td>0.0315</td>
<td>0.1700</td>
<td>0.0369</td>
<td>0.2820</td>
<td>0.0475</td>
</tr>
<tr>
<td>3000</td>
<td>0.0820</td>
<td>0.0256</td>
<td>0.1100</td>
<td>0.0297</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.3080</td>
<td>0.0496</td>
</tr>
<tr>
<td>3250</td>
<td>0.0760</td>
<td>0.0247</td>
<td>0.0980</td>
<td>0.0280</td>
<td>0.2140</td>
<td>0.0414</td>
<td>0.3160</td>
<td>0.0503</td>
</tr>
<tr>
<td>3500</td>
<td>0.0900</td>
<td>0.0268</td>
<td>0.1320</td>
<td>0.0325</td>
<td>0.1780</td>
<td>0.0377</td>
<td>0.2660</td>
<td>0.0461</td>
</tr>
<tr>
<td>3750</td>
<td>0.0840</td>
<td>0.0259</td>
<td>0.1160</td>
<td>0.0305</td>
<td>0.1800</td>
<td>0.0379</td>
<td>0.2640</td>
<td>0.0460</td>
</tr>
<tr>
<td>4000</td>
<td>0.1200</td>
<td>0.0310</td>
<td>0.1180</td>
<td>0.0307</td>
<td>0.1700</td>
<td>0.0369</td>
<td>0.2980</td>
<td>0.0488</td>
</tr>
<tr>
<td>4250</td>
<td>0.0800</td>
<td>0.0253</td>
<td>0.1240</td>
<td>0.0315</td>
<td>0.1480</td>
<td>0.0344</td>
<td>0.2860</td>
<td>0.0478</td>
</tr>
<tr>
<td>4500</td>
<td>0.0840</td>
<td>0.0259</td>
<td>0.1060</td>
<td>0.0291</td>
<td>0.1820</td>
<td>0.0382</td>
<td>0.2660</td>
<td>0.0461</td>
</tr>
<tr>
<td>4720</td>
<td>0.0940</td>
<td>0.0274</td>
<td>0.1220</td>
<td>0.0312</td>
<td>0.1860</td>
<td>0.0386</td>
<td>0.3160</td>
<td>0.0503</td>
</tr>
<tr>
<td>5000</td>
<td>0.0960</td>
<td>0.0277</td>
<td>0.1040</td>
<td>0.0288</td>
<td>0.1780</td>
<td>0.0377</td>
<td>0.2840</td>
<td>0.0477</td>
</tr>
<tr>
<td>Tree Size</td>
<td>$\frac{1}{7}$</td>
<td>$\frac{1}{5}$</td>
<td>$\frac{1}{4}$</td>
<td>$\frac{29}{100}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
<td>---------------</td>
<td>---------------</td>
<td>-----------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>0.0140</td>
<td>0.0106</td>
<td>0.0500</td>
<td>0.0200</td>
<td>0.0240</td>
<td>0.0560</td>
<td>0.0212</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>0.0180</td>
<td>0.0120</td>
<td>0.0300</td>
<td>0.0155</td>
<td>0.0420</td>
<td>0.0183</td>
<td>0.0640</td>
<td>0.0226</td>
</tr>
<tr>
<td>750</td>
<td>0.0100</td>
<td>0.0089</td>
<td>0.0500</td>
<td>0.0200</td>
<td>0.0800</td>
<td>0.0253</td>
<td>0.0660</td>
<td>0.0230</td>
</tr>
<tr>
<td>1000</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0380</td>
<td>0.0174</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.0640</td>
<td>0.0226</td>
</tr>
<tr>
<td>1250</td>
<td>0.0100</td>
<td>0.0089</td>
<td>0.0420</td>
<td>0.0183</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.0800</td>
<td>0.0253</td>
</tr>
<tr>
<td>1500</td>
<td>0.0220</td>
<td>0.0133</td>
<td>0.0280</td>
<td>0.0150</td>
<td>0.0820</td>
<td>0.0256</td>
<td>0.0620</td>
<td>0.0223</td>
</tr>
<tr>
<td>1750</td>
<td>0.0100</td>
<td>0.0089</td>
<td>0.0340</td>
<td>0.0165</td>
<td>0.0660</td>
<td>0.0230</td>
<td>0.0560</td>
<td>0.0212</td>
</tr>
<tr>
<td>2000</td>
<td>0.0080</td>
<td>0.0080</td>
<td>0.0360</td>
<td>0.0170</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.0480</td>
<td>0.0196</td>
</tr>
<tr>
<td>2250</td>
<td>0.0080</td>
<td>0.0080</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0780</td>
<td>0.0250</td>
<td>0.0520</td>
<td>0.0204</td>
</tr>
<tr>
<td>2500</td>
<td>0.0080</td>
<td>0.0080</td>
<td>0.0320</td>
<td>0.0160</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.0720</td>
<td>0.0240</td>
</tr>
<tr>
<td>2750</td>
<td>0.0180</td>
<td>0.0120</td>
<td>0.0460</td>
<td>0.0192</td>
<td>0.0720</td>
<td>0.0240</td>
<td>0.0620</td>
<td>0.0223</td>
</tr>
<tr>
<td>3000</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0320</td>
<td>0.0160</td>
<td>0.0880</td>
<td>0.0265</td>
<td>0.0520</td>
<td>0.0204</td>
</tr>
<tr>
<td>3250</td>
<td>0.0200</td>
<td>0.0126</td>
<td>0.0240</td>
<td>0.0139</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.0820</td>
<td>0.0256</td>
</tr>
<tr>
<td>3500</td>
<td>0.0100</td>
<td>0.0089</td>
<td>0.0280</td>
<td>0.0150</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.0620</td>
<td>0.0223</td>
</tr>
<tr>
<td>3750</td>
<td>0.0080</td>
<td>0.0080</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.0760</td>
<td>0.0247</td>
<td>0.0580</td>
<td>0.0215</td>
</tr>
<tr>
<td>4000</td>
<td>0.0200</td>
<td>0.0126</td>
<td>0.0420</td>
<td>0.0183</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.0820</td>
<td>0.0256</td>
</tr>
<tr>
<td>4250</td>
<td>0.0040</td>
<td>0.0380</td>
<td>0.01740</td>
<td>0.0620</td>
<td>0.0223</td>
<td>0.0580</td>
<td>0.0215</td>
<td></td>
</tr>
<tr>
<td>4500</td>
<td>0.0100</td>
<td>0.0089</td>
<td>0.0240</td>
<td>0.0139</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.0520</td>
<td>0.0204</td>
</tr>
<tr>
<td>4750</td>
<td>0.0140</td>
<td>0.0106</td>
<td>0.0180</td>
<td>0.0120</td>
<td>0.0920</td>
<td>0.0271</td>
<td>0.0620</td>
<td>0.0223</td>
</tr>
<tr>
<td>5000</td>
<td>0.0080</td>
<td>0.0080</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0620</td>
<td>0.0223</td>
<td>0.0840</td>
<td>0.0259</td>
</tr>
<tr>
<td>tree size</td>
<td>( \frac{1}{7} )</td>
<td>CI</td>
<td>( \frac{1}{5} )</td>
<td>AV</td>
<td>CI</td>
<td>( \frac{1}{4} )</td>
<td>AV</td>
<td>CI</td>
</tr>
<tr>
<td>-----------</td>
<td>----------------</td>
<td>-----</td>
<td>----------------</td>
<td>-----</td>
<td>-----</td>
<td>----------------</td>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td>250</td>
<td>0.1040</td>
<td>0.0288</td>
<td>0.1800</td>
<td>0.0379</td>
<td>0.2300</td>
<td>0.0429</td>
<td>0.3840</td>
<td>0.0554</td>
</tr>
<tr>
<td>500</td>
<td>0.0880</td>
<td>0.0265</td>
<td>0.1500</td>
<td>0.0346</td>
<td>0.2080</td>
<td>0.0408</td>
<td>0.3600</td>
<td>0.0537</td>
</tr>
<tr>
<td>750</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.1900</td>
<td>0.0390</td>
<td>0.2620</td>
<td>0.0458</td>
<td>0.3320</td>
<td>0.0515</td>
</tr>
<tr>
<td>1000</td>
<td>0.0900</td>
<td>0.0268</td>
<td>0.1500</td>
<td>0.0356</td>
<td>0.2320</td>
<td>0.0431</td>
<td>0.3320</td>
<td>0.0515</td>
</tr>
<tr>
<td>1250</td>
<td>0.0960</td>
<td>0.0277</td>
<td>0.1880</td>
<td>0.0388</td>
<td>0.2460</td>
<td>0.0444</td>
<td>0.3720</td>
<td>0.0546</td>
</tr>
<tr>
<td>1500</td>
<td>0.0920</td>
<td>0.0271</td>
<td>0.1680</td>
<td>0.0367</td>
<td>0.2520</td>
<td>0.0449</td>
<td>0.3400</td>
<td>0.0522</td>
</tr>
<tr>
<td>1750</td>
<td>0.0940</td>
<td>0.0274</td>
<td>0.1540</td>
<td>0.0351</td>
<td>0.2140</td>
<td>0.0414</td>
<td>0.3620</td>
<td>0.0538</td>
</tr>
<tr>
<td>2000</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.1800</td>
<td>0.0379</td>
<td>0.2560</td>
<td>0.0453</td>
<td>0.3460</td>
<td>0.0526</td>
</tr>
<tr>
<td>2250</td>
<td>0.1160</td>
<td>0.0305</td>
<td>0.1780</td>
<td>0.0377</td>
<td>0.2420</td>
<td>0.0440</td>
<td>0.3600</td>
<td>0.0537</td>
</tr>
<tr>
<td>2500</td>
<td>0.0720</td>
<td>0.0240</td>
<td>0.1600</td>
<td>0.0358</td>
<td>0.2140</td>
<td>0.0414</td>
<td>0.3400</td>
<td>0.0522</td>
</tr>
<tr>
<td>2750</td>
<td>0.1180</td>
<td>0.0307</td>
<td>0.1700</td>
<td>0.0369</td>
<td>0.2420</td>
<td>0.0440</td>
<td>0.3440</td>
<td>0.0525</td>
</tr>
<tr>
<td>3000</td>
<td>0.0940</td>
<td>0.0274</td>
<td>0.1420</td>
<td>0.0337</td>
<td>0.2360</td>
<td>0.0435</td>
<td>0.3600</td>
<td>0.0537</td>
</tr>
<tr>
<td>3250</td>
<td>0.0960</td>
<td>0.0277</td>
<td>0.1220</td>
<td>0.0312</td>
<td>0.2840</td>
<td>0.0477</td>
<td>0.3980</td>
<td>0.0564</td>
</tr>
<tr>
<td>3500</td>
<td>0.1000</td>
<td>0.0283</td>
<td>0.1600</td>
<td>0.0358</td>
<td>0.2480</td>
<td>0.0445</td>
<td>0.3280</td>
<td>0.0512</td>
</tr>
<tr>
<td>3750</td>
<td>0.0920</td>
<td>0.0271</td>
<td>0.1560</td>
<td>0.0353</td>
<td>0.2560</td>
<td>0.0453</td>
<td>0.3220</td>
<td>0.0508</td>
</tr>
<tr>
<td>4000</td>
<td>0.1400</td>
<td>0.0335</td>
<td>0.1600</td>
<td>0.0358</td>
<td>0.2340</td>
<td>0.0433</td>
<td>0.3800</td>
<td>0.0551</td>
</tr>
<tr>
<td>4250</td>
<td>0.0840</td>
<td>0.0259</td>
<td>0.1620</td>
<td>0.0360</td>
<td>0.2100</td>
<td>0.0410</td>
<td>0.3440</td>
<td>0.0525</td>
</tr>
<tr>
<td>4500</td>
<td>0.0940</td>
<td>0.0274</td>
<td>0.1300</td>
<td>0.0322</td>
<td>0.2520</td>
<td>0.0449</td>
<td>0.3180</td>
<td>0.0504</td>
</tr>
<tr>
<td>4750</td>
<td>0.1080</td>
<td>0.0294</td>
<td>0.1400</td>
<td>0.0335</td>
<td>0.2780</td>
<td>0.0472</td>
<td>0.3780</td>
<td>0.0550</td>
</tr>
<tr>
<td>5000</td>
<td>0.1040</td>
<td>0.0288</td>
<td>0.1300</td>
<td>0.0322</td>
<td>0.2400</td>
<td>0.0438</td>
<td>0.3680</td>
<td>0.0543</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
</tr>
<tr>
<td>----------</td>
<td>---------</td>
<td>-----------</td>
<td>---------</td>
<td>-----------</td>
<td>---------</td>
<td>-----------</td>
<td>---------</td>
<td>-----------</td>
</tr>
<tr>
<td>250</td>
<td>7.7240</td>
<td>0.1807</td>
<td>7.4640</td>
<td>0.1579</td>
<td>7.3020</td>
<td>0.1461</td>
<td>7.2320</td>
<td>0.1361</td>
</tr>
<tr>
<td>500</td>
<td>8.8440</td>
<td>0.1790</td>
<td>8.5920</td>
<td>0.1655</td>
<td>8.4520</td>
<td>0.1506</td>
<td>8.1660</td>
<td>0.1473</td>
</tr>
<tr>
<td>750</td>
<td>9.4300</td>
<td>0.1874</td>
<td>9.1040</td>
<td>0.1639</td>
<td>8.9020</td>
<td>0.1535</td>
<td>8.7280</td>
<td>0.1441</td>
</tr>
<tr>
<td>1000</td>
<td>9.8480</td>
<td>0.1996</td>
<td>9.4480</td>
<td>0.1772</td>
<td>9.2800</td>
<td>0.1613</td>
<td>9.2360</td>
<td>0.1411</td>
</tr>
<tr>
<td>1250</td>
<td>10.4260</td>
<td>0.2137</td>
<td>9.8500</td>
<td>0.1771</td>
<td>9.7320</td>
<td>0.1528</td>
<td>9.4240</td>
<td>0.1443</td>
</tr>
<tr>
<td>1500</td>
<td>10.5800</td>
<td>0.2034</td>
<td>10.2760</td>
<td>0.1764</td>
<td>10.0220</td>
<td>0.1489</td>
<td>9.8220</td>
<td>0.1513</td>
</tr>
<tr>
<td>1750</td>
<td>10.8600</td>
<td>0.1989</td>
<td>10.5420</td>
<td>0.1716</td>
<td>10.2800</td>
<td>0.1493</td>
<td>9.9320</td>
<td>0.1521</td>
</tr>
<tr>
<td>2000</td>
<td>11.0860</td>
<td>0.2009</td>
<td>10.7360</td>
<td>0.1735</td>
<td>10.5480</td>
<td>0.1718</td>
<td>10.2220</td>
<td>0.1492</td>
</tr>
<tr>
<td>2250</td>
<td>11.1740</td>
<td>0.2035</td>
<td>10.7740</td>
<td>0.1773</td>
<td>10.5760</td>
<td>0.1597</td>
<td>10.5100</td>
<td>0.1436</td>
</tr>
<tr>
<td>2500</td>
<td>11.3900</td>
<td>0.2106</td>
<td>10.9680</td>
<td>0.1811</td>
<td>10.7620</td>
<td>0.1551</td>
<td>10.5540</td>
<td>0.1359</td>
</tr>
<tr>
<td>2750</td>
<td>11.6660</td>
<td>0.2083</td>
<td>11.2780</td>
<td>0.1822</td>
<td>10.9780</td>
<td>0.1566</td>
<td>10.7540</td>
<td>0.1433</td>
</tr>
<tr>
<td>3000</td>
<td>11.8780</td>
<td>0.2181</td>
<td>11.4240</td>
<td>0.1834</td>
<td>11.1760</td>
<td>0.1667</td>
<td>10.9500</td>
<td>0.1411</td>
</tr>
<tr>
<td>3250</td>
<td>12.3780</td>
<td>0.2037</td>
<td>11.8120</td>
<td>0.1715</td>
<td>11.4740</td>
<td>0.1466</td>
<td>11.2400</td>
<td>0.1326</td>
</tr>
<tr>
<td>3500</td>
<td>12.0900</td>
<td>0.1911</td>
<td>11.6420</td>
<td>0.1642</td>
<td>11.3920</td>
<td>0.1455</td>
<td>11.3140</td>
<td>0.1218</td>
</tr>
<tr>
<td>3750</td>
<td>12.1400</td>
<td>0.0264</td>
<td>11.6900</td>
<td>0.1739</td>
<td>11.4580</td>
<td>0.1484</td>
<td>11.2660</td>
<td>0.1348</td>
</tr>
<tr>
<td>4000</td>
<td>12.3300</td>
<td>0.2040</td>
<td>11.8180</td>
<td>0.1732</td>
<td>11.4900</td>
<td>0.1528</td>
<td>11.3220</td>
<td>0.1371</td>
</tr>
<tr>
<td>4250</td>
<td>12.2820</td>
<td>0.2190</td>
<td>11.7820</td>
<td>0.1831</td>
<td>11.6140</td>
<td>0.1572</td>
<td>11.3080</td>
<td>0.1573</td>
</tr>
<tr>
<td>4500</td>
<td>12.1860</td>
<td>0.2267</td>
<td>11.7880</td>
<td>0.1995</td>
<td>11.5820</td>
<td>0.1689</td>
<td>11.2640</td>
<td>0.1547</td>
</tr>
<tr>
<td>4750</td>
<td>11.7220</td>
<td>0.2557</td>
<td>11.3660</td>
<td>0.2266</td>
<td>11.1380</td>
<td>0.2040</td>
<td>11.0720</td>
<td>0.1777</td>
</tr>
<tr>
<td>5000</td>
<td>9.1660</td>
<td>0.5248</td>
<td>8.8860</td>
<td>0.4923</td>
<td>8.6960</td>
<td>0.4550</td>
<td>8.8360</td>
<td>0.4038</td>
</tr>
<tr>
<td>Tree size</td>
<td>( \alpha = \frac{1}{7} )</td>
<td>( \alpha = \frac{1}{5} )</td>
<td>( \alpha = \frac{1}{4} )</td>
<td>( \frac{29}{100} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-----------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>0.0340</td>
<td>0.0165</td>
<td>0.0320</td>
<td>0.0160</td>
<td>0.0540</td>
<td>0.0208</td>
<td>0.1540</td>
<td>0.0351</td>
</tr>
<tr>
<td>500</td>
<td>0.0320</td>
<td>0.0160</td>
<td>0.0320</td>
<td>0.0160</td>
<td>0.0540</td>
<td>0.0208</td>
<td>0.1100</td>
<td>0.0297</td>
</tr>
<tr>
<td>750</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0560</td>
<td>0.0212</td>
<td>0.1400</td>
<td>0.0335</td>
</tr>
<tr>
<td>1000</td>
<td>0.0220</td>
<td>0.0133</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.1360</td>
<td>0.0330</td>
</tr>
<tr>
<td>1250</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.0920</td>
<td>0.0271</td>
<td>0.1140</td>
<td>0.0302</td>
</tr>
<tr>
<td>1500</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0460</td>
<td>0.0192</td>
<td>0.1280</td>
<td>0.0320</td>
</tr>
<tr>
<td>1750</td>
<td>0.0160</td>
<td>0.0113</td>
<td>0.0340</td>
<td>0.0165</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.1080</td>
<td>0.0291</td>
</tr>
<tr>
<td>2000</td>
<td>0.0300</td>
<td>0.0155</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.0740</td>
<td>0.0243</td>
<td>0.1140</td>
<td>0.0302</td>
</tr>
<tr>
<td>2250</td>
<td>0.0220</td>
<td>0.0133</td>
<td>0.0360</td>
<td>0.0170</td>
<td>0.0480</td>
<td>0.0196</td>
<td>0.1220</td>
<td>0.0312</td>
</tr>
<tr>
<td>2500</td>
<td>0.0240</td>
<td>0.0139</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.0580</td>
<td>0.0215</td>
<td>0.0940</td>
<td>0.0274</td>
</tr>
<tr>
<td>2750</td>
<td>0.0280</td>
<td>0.0150</td>
<td>0.0420</td>
<td>0.0183</td>
<td>0.0720</td>
<td>0.0240</td>
<td>0.1040</td>
<td>0.0288</td>
</tr>
<tr>
<td>3000</td>
<td>0.0300</td>
<td>0.0155</td>
<td>0.0520</td>
<td>0.0204</td>
<td>0.0520</td>
<td>0.0204</td>
<td>0.1220</td>
<td>0.0312</td>
</tr>
<tr>
<td>3250</td>
<td>0.0240</td>
<td>0.0139</td>
<td>0.0460</td>
<td>0.0192</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.1060</td>
<td>0.0291</td>
</tr>
<tr>
<td>3500</td>
<td>0.0300</td>
<td>0.0155</td>
<td>0.0540</td>
<td>0.0208</td>
<td>0.0880</td>
<td>0.0265</td>
<td>0.1340</td>
<td>0.0327</td>
</tr>
<tr>
<td>3750</td>
<td>0.0220</td>
<td>0.0133</td>
<td>0.0460</td>
<td>0.0192</td>
<td>0.0760</td>
<td>0.0247</td>
<td>0.1140</td>
<td>0.0302</td>
</tr>
<tr>
<td>4000</td>
<td>0.0360</td>
<td>0.0170</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.1340</td>
<td>0.0327</td>
</tr>
<tr>
<td>4250</td>
<td>0.0500</td>
<td>0.0200</td>
<td>0.0560</td>
<td>0.0212</td>
<td>0.0900</td>
<td>0.0268</td>
<td>0.1240</td>
<td>0.0315</td>
</tr>
<tr>
<td>4500</td>
<td>0.0480</td>
<td>0.0196</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.0700</td>
<td>0.0237</td>
<td>0.1400</td>
<td>0.0335</td>
</tr>
<tr>
<td>4750</td>
<td>0.0600</td>
<td>0.0219</td>
<td>0.0860</td>
<td>0.0262</td>
<td>0.0980</td>
<td>0.0280</td>
<td>0.1380</td>
<td>0.0332</td>
</tr>
<tr>
<td>5000</td>
<td>0.0740</td>
<td>0.0243</td>
<td>0.1140</td>
<td>0.0302</td>
<td>0.1180</td>
<td>0.0307</td>
<td>0.1520</td>
<td>0.0349</td>
</tr>
<tr>
<td>tree size</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
<td>AV</td>
<td>CI</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>250</td>
<td>0.0020</td>
<td>0.0020</td>
<td>0.0180</td>
<td>0.0120</td>
<td>0.0660</td>
<td>0.0230</td>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>0</td>
<td>0</td>
<td>0.0180</td>
<td>0.0120</td>
<td>0.0820</td>
<td>0.0256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>750</td>
<td>0.0020</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>0</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1250</td>
<td>0</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1500</td>
<td>0.0020</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1750</td>
<td>0</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>0</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2250</td>
<td>0</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2500</td>
<td>0</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2750</td>
<td>0</td>
<td>0</td>
<td>0.0120</td>
<td>0.0098</td>
<td>0.0640</td>
<td>0.0226</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>0.0020</td>
<td>0</td>
<td>0.1400</td>
<td>0.0106</td>
<td>0.0480</td>
<td>0.0196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3250</td>
<td>0.0020</td>
<td>0</td>
<td>0.1400</td>
<td>0.0106</td>
<td>0.0480</td>
<td>0.0196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3500</td>
<td>0</td>
<td>0</td>
<td>0.0080</td>
<td>0.0080</td>
<td>0.0180</td>
<td>0.0120</td>
<td>0.0860</td>
<td>0.0262</td>
</tr>
<tr>
<td>3750</td>
<td>0.0020</td>
<td>0.0040</td>
<td>0.0040</td>
<td>0.0200</td>
<td>0.0126</td>
<td>0.0560</td>
<td>0.0212</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>0</td>
<td>0</td>
<td>0.0100</td>
<td>0.0089</td>
<td>0.0200</td>
<td>0.0126</td>
<td>0.0720</td>
<td>0.0240</td>
</tr>
<tr>
<td>4250</td>
<td>0.0060</td>
<td>0.0080</td>
<td>0.0080</td>
<td>0.0420</td>
<td>0.0183</td>
<td>0.0780</td>
<td>0.0250</td>
<td></td>
</tr>
<tr>
<td>4500</td>
<td>0.0057</td>
<td>0.0200</td>
<td>0.0126</td>
<td>0.0440</td>
<td>0.0188</td>
<td>0.0820</td>
<td>0.0256</td>
<td></td>
</tr>
<tr>
<td>4750</td>
<td>0.0060</td>
<td>0.0080</td>
<td>0.0080</td>
<td>0.0280</td>
<td>0.0150</td>
<td>0.0940</td>
<td>0.0274</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>0.0100</td>
<td>0.0089</td>
<td>0.0160</td>
<td>0.0113</td>
<td>0.0380</td>
<td>0.0174</td>
<td>0.0860</td>
<td>0.0262</td>
</tr>
<tr>
<td>Tree Size</td>
<td>$\alpha = \frac{1}{7}$</td>
<td>$\frac{1}{5}$</td>
<td>$\frac{1}{4}$</td>
<td>$\frac{29}{100}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------</td>
<td>---------</td>
<td>---------</td>
<td>----------</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>250</td>
<td>0.0360</td>
<td>0.0170</td>
<td>0.0340</td>
<td>0.0165</td>
<td>0.0720</td>
<td>0.0240</td>
<td>0.2200</td>
<td>0.0420</td>
</tr>
<tr>
<td>500</td>
<td>0.0320</td>
<td>0.0160</td>
<td>0.0360</td>
<td>0.0170</td>
<td>0.0720</td>
<td>0.0240</td>
<td>0.1920</td>
<td>0.0392</td>
</tr>
<tr>
<td>750</td>
<td>0.0280</td>
<td>0.0150</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.2040</td>
<td>0.0404</td>
</tr>
<tr>
<td>1000</td>
<td>0.0220</td>
<td>0.0133</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.0800</td>
<td>0.0253</td>
<td>0.2000</td>
<td>0.0400</td>
</tr>
<tr>
<td>1250</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.1040</td>
<td>0.0288</td>
<td>0.1740</td>
<td>0.0373</td>
</tr>
<tr>
<td>1500</td>
<td>0.0140</td>
<td>0.0106</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.1940</td>
<td>0.0394</td>
</tr>
<tr>
<td>1750</td>
<td>0.0160</td>
<td>0.0113</td>
<td>0.0380</td>
<td>0.0174</td>
<td>0.0580</td>
<td>0.0215</td>
<td>0.1720</td>
<td>0.0371</td>
</tr>
<tr>
<td>2000</td>
<td>0.0300</td>
<td>0.0155</td>
<td>0.0720</td>
<td>0.0240</td>
<td>0.0900</td>
<td>0.0268</td>
<td>0.1880</td>
<td>0.0388</td>
</tr>
<tr>
<td>2250</td>
<td>0.0220</td>
<td>0.0133</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.1760</td>
<td>0.0375</td>
</tr>
<tr>
<td>2500</td>
<td>0.0240</td>
<td>0.0139</td>
<td>0.0400</td>
<td>0.0179</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.1820</td>
<td>0.0382</td>
</tr>
<tr>
<td>2750</td>
<td>0.0280</td>
<td>0.0150</td>
<td>0.0420</td>
<td>0.0183</td>
<td>0.0860</td>
<td>0.0262</td>
<td>0.1520</td>
<td>0.0359</td>
</tr>
<tr>
<td>3000</td>
<td>0.0320</td>
<td>0.0160</td>
<td>0.0560</td>
<td>0.0212</td>
<td>0.0680</td>
<td>0.0233</td>
<td>0.1960</td>
<td>0.0396</td>
</tr>
<tr>
<td>3250</td>
<td>0.0260</td>
<td>0.0144</td>
<td>0.0480</td>
<td>0.0196</td>
<td>0.0540</td>
<td>0.0208</td>
<td>0.1800</td>
<td>0.0379</td>
</tr>
<tr>
<td>3500</td>
<td>0.0300</td>
<td>0.0155</td>
<td>0.0620</td>
<td>0.0223</td>
<td>0.1060</td>
<td>0.0291</td>
<td>0.2200</td>
<td>0.0420</td>
</tr>
<tr>
<td>3750</td>
<td>0.0240</td>
<td>0.0139</td>
<td>0.0500</td>
<td>0.0200</td>
<td>0.0960</td>
<td>0.0277</td>
<td>0.1700</td>
<td>0.0369</td>
</tr>
<tr>
<td>4000</td>
<td>0.0360</td>
<td>0.0170</td>
<td>0.0800</td>
<td>0.0253</td>
<td>0.0840</td>
<td>0.0259</td>
<td>0.2060</td>
<td>0.0406</td>
</tr>
<tr>
<td>4250</td>
<td>0.0560</td>
<td>0.0212</td>
<td>0.0640</td>
<td>0.0226</td>
<td>0.1320</td>
<td>0.0325</td>
<td>0.2020</td>
<td>0.0402</td>
</tr>
<tr>
<td>4500</td>
<td>0.0520</td>
<td>0.0204</td>
<td>0.0840</td>
<td>0.0259</td>
<td>0.1140</td>
<td>0.0302</td>
<td>0.2220</td>
<td>0.0421</td>
</tr>
<tr>
<td>4750</td>
<td>0.0660</td>
<td>0.0230</td>
<td>0.0940</td>
<td>0.0274</td>
<td>0.1260</td>
<td>0.0317</td>
<td>0.2320</td>
<td>0.0431</td>
</tr>
<tr>
<td>5000</td>
<td>0.0840</td>
<td>0.0259</td>
<td>0.1300</td>
<td>0.0322</td>
<td>0.1560</td>
<td>0.0353</td>
<td>0.2380</td>
<td>0.0436</td>
</tr>
</tbody>
</table>
**Table 4.2.28**

Central processor timing on statistical runs

The AVL tree construction algorithm

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5356</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Bell-tree construction algorithm

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7443</td>
<td>8483</td>
<td>10209</td>
<td>12399</td>
<td>15327</td>
</tr>
</tbody>
</table>

The BB-tree construction algorithm

<table>
<thead>
<tr>
<th>k</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>$\frac{1}{7}$</td>
<td>$\frac{1}{5}$</td>
<td>$\frac{1}{4}$</td>
<td>29</td>
<td>$\frac{100}{100}$</td>
</tr>
</tbody>
</table>

| a | 8056 | 7999 | 8007 | 8051 |
4.3 \textbf{EMPIRICAL OBSERVATIONS ON ALGORITHMS OF CLASS (2)}

4.3.1 \textbf{THE OPTIMAL TREE ALGORITHM OF KNUTH}

No empirical tests were made on the average weighted path length of optimal trees constructed by Knuth's algorithm because of storage considerations which has been previously pointed out in Section 3.6.3. The figures shown in Table 4.3.1 are taken from Walker and Gotlieb (1972a).

4.3.2 \textbf{THE BRUNO COFFMAN ALGORITHM}

Empirical tests on this algorithm on all five sets of \( \alpha \) data and with \( \beta \) necessarily 0 (case 1) are found in Table 4.3.2. The running time of the tree construction, the average weighted path length of the starting tree and the blocksize that was used in constructing the starting tree are also included. The results of the average weighted path length are slightly higher than those given by Bruno and Coffman.

4.3.3 \textbf{THE WALKER GOTLIEB ALGORITHM}

Trees are constructed for all tree weight combinations of Table 4.1.1 and the running time for each construction has been included. The results agree with Walker and Gotlieb (see Table 4.3.3).

4.3.4 \textbf{CONCLUSIONS}

Table 4.3.4 gives a synopsis of the results for the above three algorithms using the five sets of \( \alpha_1 \) data and the one set of \( \beta_1 \) data common to all three (\( \beta \) case 1). For the Walker and Gotlieb case, the tree constructed with parameters \( N_0 = 15 \) and \( F = 4 \) is used in the comparison since this is an acceptable choice for most cases. For the Bruno and Coffman statistics the blocksize is chosen as 50 since this was the
common size Bruno and Coffman used in their tests. The Walker and Gotlieb algorithm is seen to be superior from both average weighted path length and timing considerations.
Table 4.3.1

Average weighted path length of trees constructed by the KNCA.

<table>
<thead>
<tr>
<th>β case</th>
<th>α set 1</th>
<th>α set 2</th>
<th>α set 3</th>
<th>α set 4</th>
<th>α set 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.2944</td>
<td>4.6317</td>
<td>4.0035</td>
<td>5.0362</td>
<td>4.1379</td>
</tr>
<tr>
<td>2</td>
<td>6.6060</td>
<td>7.2555</td>
<td>6.4230</td>
<td>7.4164</td>
<td>5.9726</td>
</tr>
<tr>
<td>3</td>
<td>5.9633</td>
<td>6.5382</td>
<td>5.6524</td>
<td>6.5494</td>
<td>5.3331</td>
</tr>
<tr>
<td>4</td>
<td>5.8250</td>
<td>6.4549</td>
<td>5.5935</td>
<td>6.7508</td>
<td>5.4117</td>
</tr>
<tr>
<td>5</td>
<td>6.2576</td>
<td>6.5810</td>
<td>6.0850</td>
<td>6.7320</td>
<td>5.8119</td>
</tr>
<tr>
<td>6</td>
<td>7.0856</td>
<td>7.1216</td>
<td>7.0125</td>
<td>7.1119</td>
<td>6.9904</td>
</tr>
<tr>
<td>blocksize</td>
<td>original avg. weighted path length</td>
<td>final avg. weighted path length</td>
<td>time/sec.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------</td>
<td>-------------------------------</td>
<td>-----------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>5.1321</td>
<td>4.4373</td>
<td>3.273</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>5.5000</td>
<td>4.3785</td>
<td>2.908</td>
<td></td>
<td></td>
</tr>
<tr>
<td>75</td>
<td>8.0752</td>
<td>4.4016</td>
<td>3.030</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>8.1326</td>
<td>5.0757</td>
<td>2.796</td>
<td></td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>8.2382</td>
<td>4.4759</td>
<td>2.712</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>9.1943</td>
<td>4.3775</td>
<td>2.887</td>
<td></td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>9.0984</td>
<td>4.3211</td>
<td>3.012</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>8.3014</td>
<td>4.4768</td>
<td>2.907</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>blocksize</th>
<th>original avg. weighted path length</th>
<th>final avg. weighted path length</th>
<th>time/sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>5.3740</td>
<td>4.7738</td>
<td>3.058</td>
</tr>
<tr>
<td>50</td>
<td>7.1026</td>
<td>4.9079</td>
<td>2.951</td>
</tr>
<tr>
<td>75</td>
<td>8.3166</td>
<td>4.8190</td>
<td>2.939</td>
</tr>
<tr>
<td>100</td>
<td>8.5040</td>
<td>5.0509</td>
<td>3.071</td>
</tr>
<tr>
<td>125</td>
<td>7.4120</td>
<td>4.8102</td>
<td>2.702</td>
</tr>
<tr>
<td>150</td>
<td>8.6599</td>
<td>4.8126</td>
<td>2.716</td>
</tr>
<tr>
<td>175</td>
<td>8.7423</td>
<td>5.0315</td>
<td>2.956</td>
</tr>
<tr>
<td>200</td>
<td>7.7876</td>
<td>5.0024</td>
<td>2.639</td>
</tr>
<tr>
<td>blocksize</td>
<td>original avg. weighted path length</td>
<td>final avg. weighted path length</td>
<td>time/sec.</td>
</tr>
<tr>
<td>-----------</td>
<td>-----------------------------------</td>
<td>---------------------------------</td>
<td>-----------</td>
</tr>
<tr>
<td>25</td>
<td>5.7218</td>
<td>4.2422</td>
<td>2.759</td>
</tr>
<tr>
<td>50</td>
<td>4.9510</td>
<td>4.2174</td>
<td>2.871</td>
</tr>
<tr>
<td>75</td>
<td>9.3177</td>
<td>4.0772</td>
<td>3.081</td>
</tr>
<tr>
<td>100</td>
<td>7.3108</td>
<td>4.1671</td>
<td>2.831</td>
</tr>
<tr>
<td>125</td>
<td>8.2943</td>
<td>4.3815</td>
<td>3.073</td>
</tr>
<tr>
<td>150</td>
<td>9.1884</td>
<td>4.0299</td>
<td>3.046</td>
</tr>
<tr>
<td>175</td>
<td>8.7127</td>
<td>4.0864</td>
<td>2.992</td>
</tr>
<tr>
<td>200</td>
<td>8.1593</td>
<td>4.0729</td>
<td>3.336</td>
</tr>
</tbody>
</table>

For α set 4:

<table>
<thead>
<tr>
<th>blocksize</th>
<th>original avg. weighted path length</th>
<th>final avg. weighted path length</th>
<th>time/sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>6.2303</td>
<td>5.4557</td>
<td>2.851</td>
</tr>
<tr>
<td>50</td>
<td>7.5322</td>
<td>5.6956</td>
<td>2.627</td>
</tr>
<tr>
<td>75</td>
<td>7.2206</td>
<td>5.3076</td>
<td>2.749</td>
</tr>
<tr>
<td>100</td>
<td>7.2021</td>
<td>5.3382</td>
<td>2.752</td>
</tr>
<tr>
<td>125</td>
<td>8.9968</td>
<td>5.3696</td>
<td>2.887</td>
</tr>
<tr>
<td>150</td>
<td>7.5701</td>
<td>5.4517</td>
<td>2.770</td>
</tr>
<tr>
<td>175</td>
<td>8.7005</td>
<td>5.4364</td>
<td>2.752</td>
</tr>
<tr>
<td>200</td>
<td>8.8301</td>
<td>5.2778</td>
<td>2.680</td>
</tr>
</tbody>
</table>
### Table 4.3.2 (cont'd.)

<table>
<thead>
<tr>
<th>blocksize</th>
<th>original avg. weighted path length</th>
<th>final avg. weighted path length</th>
<th>time/sec.</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>5.3241</td>
<td>4.4918</td>
<td>2.936</td>
</tr>
<tr>
<td>50</td>
<td>5.8825</td>
<td>4.4775</td>
<td>2.669</td>
</tr>
<tr>
<td>75</td>
<td>7.1315</td>
<td>4.3974</td>
<td>2.776</td>
</tr>
<tr>
<td>100</td>
<td>8.0506</td>
<td>4.1899</td>
<td>2.853</td>
</tr>
<tr>
<td>125</td>
<td>9.5555</td>
<td>4.5832</td>
<td>3.328</td>
</tr>
<tr>
<td>150</td>
<td>8.2572</td>
<td>4.2681</td>
<td>2.748</td>
</tr>
<tr>
<td>175</td>
<td>8.0172</td>
<td>4.3361</td>
<td>2.907</td>
</tr>
<tr>
<td>200</td>
<td>8.5678</td>
<td>4.3194</td>
<td>2.672</td>
</tr>
<tr>
<td>( N_0 = 0 )</td>
<td>( N_0 = 15 )</td>
<td>( N_0 = 15 )</td>
<td>( N_0 = 15 )</td>
</tr>
<tr>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>( \alpha ) set 1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \beta ) Case</td>
<td>F = 1 (sec.)</td>
<td>F = 3 (sec.)</td>
<td>F = 4 (sec.)</td>
</tr>
<tr>
<td>1</td>
<td>4.7270</td>
<td>0.723</td>
<td>4.3635</td>
</tr>
<tr>
<td>2</td>
<td>7.6283</td>
<td>0.719</td>
<td>6.6653</td>
</tr>
<tr>
<td>3</td>
<td>7.0949</td>
<td>0.677</td>
<td>5.9964</td>
</tr>
<tr>
<td>4</td>
<td>6.7102</td>
<td>0.677</td>
<td>5.8256</td>
</tr>
<tr>
<td>5</td>
<td>7.8798</td>
<td>0.676</td>
<td>6.3582</td>
</tr>
<tr>
<td>6</td>
<td>9.8989</td>
<td>0.699</td>
<td>7.2961</td>
</tr>
<tr>
<td>( \alpha ) set 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \beta ) Case</td>
<td>F = 1 (sec.)</td>
<td>F = 3 (sec.)</td>
<td>F = 4 (sec.)</td>
</tr>
<tr>
<td>1</td>
<td>5.0299</td>
<td>0.675</td>
<td>4.7052</td>
</tr>
<tr>
<td>2</td>
<td>8.1425</td>
<td>0.657</td>
<td>7.2749</td>
</tr>
<tr>
<td>3</td>
<td>7.6646</td>
<td>0.679</td>
<td>6.5712</td>
</tr>
<tr>
<td>4</td>
<td>7.2468</td>
<td>0.679</td>
<td>6.4670</td>
</tr>
<tr>
<td>5</td>
<td>8.4314</td>
<td>0.679</td>
<td>6.7982</td>
</tr>
<tr>
<td>6</td>
<td>9.6699</td>
<td>0.638</td>
<td>7.4320</td>
</tr>
<tr>
<td>( \alpha ) set 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \beta ) Case</td>
<td>F = 1 (sec.)</td>
<td>F = 3 (sec.)</td>
<td>F = 4 (sec.)</td>
</tr>
<tr>
<td>1</td>
<td>4.7786</td>
<td>0.829</td>
<td>4.0512</td>
</tr>
<tr>
<td>2</td>
<td>8.1525</td>
<td>0.769</td>
<td>6.4587</td>
</tr>
<tr>
<td>3</td>
<td>7.1084</td>
<td>0.761</td>
<td>5.7214</td>
</tr>
<tr>
<td>4</td>
<td>7.1817</td>
<td>0.799</td>
<td>5.6676</td>
</tr>
<tr>
<td>5</td>
<td>9.2180</td>
<td>0.771</td>
<td>6.3468</td>
</tr>
<tr>
<td>6</td>
<td>11.1363</td>
<td>0.768</td>
<td>7.2884</td>
</tr>
<tr>
<td>$\alpha$ set 4</td>
<td>$\beta$ Case</td>
<td>$\alpha$ set 5</td>
<td></td>
</tr>
<tr>
<td>---------------</td>
<td>-------------</td>
<td>---------------</td>
<td></td>
</tr>
<tr>
<td>$N_0 = 0$</td>
<td>$F = 1$ (sec.)</td>
<td>$N_0 = 15$</td>
<td>$F = 3$ (sec.)</td>
</tr>
<tr>
<td>$N_0 = 15$</td>
<td>$F = 4$ (sec.)</td>
<td>$N_0 = 15$</td>
<td>$F = 5$ (sec.)</td>
</tr>
<tr>
<td>$\beta$ Case</td>
<td>$\beta$ Case</td>
<td>$\beta$ Case</td>
<td>$\beta$ Case</td>
</tr>
<tr>
<td>1 5.9324</td>
<td>1 4.8235</td>
<td>1 7.2683</td>
<td>1 10.3917</td>
</tr>
<tr>
<td>2 8.8770</td>
<td>2 7.2296</td>
<td>2 6.5523</td>
<td>2 10.3917</td>
</tr>
<tr>
<td>3 7.6391</td>
<td>3 6.5523</td>
<td>3 6.5474</td>
<td>3 10.3917</td>
</tr>
<tr>
<td>4 8.4449</td>
<td>4 6.4474</td>
<td>4 6.5474</td>
<td>4 10.3917</td>
</tr>
<tr>
<td>5 8.7564</td>
<td>5 6.4474</td>
<td>5 6.5474</td>
<td>5 10.3917</td>
</tr>
<tr>
<td>6 10.4205</td>
<td>6 10.4205</td>
<td>6 10.4205</td>
<td>6 10.4205</td>
</tr>
</tbody>
</table>
Table 4.3.4

<table>
<thead>
<tr>
<th>(a)</th>
<th>set 1</th>
<th>(\bar{a}) set 1</th>
<th>(\bar{a}) set 2</th>
<th>(\bar{a}) set 3</th>
<th>(\bar{a}) set 4</th>
<th>(\bar{a}) set 5</th>
<th>time/sec</th>
<th>(\bar{a}) set 1</th>
<th>(\bar{a}) set 2</th>
<th>(\bar{a}) set 3</th>
<th>(\bar{a}) set 4</th>
<th>(\bar{a}) set 5</th>
<th>time/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.2944</td>
<td>4.3635</td>
<td>0.431</td>
<td>4.3785</td>
<td>2.908</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4.6317</td>
<td>4.7003</td>
<td>0.460</td>
<td>4.9079</td>
<td>2.951</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4.0035</td>
<td>4.0547</td>
<td>0.482</td>
<td>4.2174</td>
<td>2.871</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5.0362</td>
<td>5.0572</td>
<td>0.481</td>
<td>5.6956</td>
<td>2.627</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>4.1379</td>
<td>4.1815</td>
<td>0.493</td>
<td>4.4775</td>
<td>2.669</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER V

CONCLUDING REMARKS

In this report various binary search tree construction algorithms have been presented and implemented in Pascal (Wirth, 1971). Two types of trees have been examined: (1) unweighted trees which are dynamically maintained by local restructuring and (2) weighted trees and weighted external trees which are static in the sense that they require a fixed number of input keys and their structure is determined by a global method involving all the keys. Through empirical observations the various tree construction algorithms are evaluated. For the unweighted case, the AVL tree is seen to be superior, followed by the BB-tree and finally the Bell-tree. Considering the Bruno-Coffman tree as a weighted external tree (external weights are zero), then in the weighted external case the Walker-Gotlieb tree algorithm is best, followed by the Bruno-Coffman algorithm and then Knuth’s optimal tree.

A type of tree which could be examined in the future is a weighted dynamic tree which is constructed using local restructuring. For example one might consider a tree such as the BB-tree and ask whether the weight of subtrees, instead of their size, may be used to determine a rotation to yield a tree whose weighted path length is reduced. Also another problem mentioned by Knuth (1971) and which is yet unsolved, is a method of readjusting a weighted tree to reflect changes in the weight values of the keys. The tree would update the weights of its keys depending on their frequency of search and perform restructuring from...
time to time to account for the changes in weight. The idea could also be extended to any weighted dynamic tree that may be devised. Such a tree could be of practical use in library applications.

A practical use of the optimal binary tree construction algorithm of Knuth is given in Appendix 4 along with a further application of the binary tree display algorithm (given in Appendix 1).
REFERENCES


APPENDIX 1

A SIMPLE BINARY TREE DISPLAY ALGORITHM

I. Visual Representation of Trees

Methods of displaying trees have been discussed by Knuth (1968). Knuth refers to three methods which as he says "have no resemblance to actual trees". They are illustrated in the following example.

Given the binary tree in Figure Al-1 the representations are

(a) nested sets (see Figure Al-2),
(b) nested parentheses

\[ D(B(A)(C)) \quad E(H(F)(I)) \]
and

(c) indentation (see Figure Al-3).

All three representations give the same partial information regarding the structure and order of the original binary tree (since information is lost concerning the order of a subtree with only one son); this is the ordering problem. Considering the subtree defined by the node with key E, it is not clear whether the node with key H is a left or right son of its father in any of the above representations. However, by introducing a notation for the representation of null trees, as can be done for each of these three representations, the ordering problem is no longer a problem!!

Representation (a) displays the binary tree structure and order clearly but is difficult to produce. Representation (b) is the familiar way of representing a list. This representation is easily produced but the structure and order though present are opaque. Representation (c)
is also familiar. In COBOL and PL/1 the concept of a structure has been introduced which is simply an ordered tree. Both languages encourage the programmer to present the definition of a particular structure by the method of indentation as given above. Similarly the contents section of Knuth (1968) is another example of its use. This representation is easy to produce and the structure and order are more readily seen than in (b).

In all representations some type of binary tree traversal must be performed to obtain the keys (Knuth, 1968), such that each node is visited only once in some defined order. There are three principal ways of traversing a binary tree which are defined as follows: if the binary tree is null do nothing, otherwise the traversal algorithms are defined recursively.

**PREORDER TRAVERSAL**

visit the root
traverse the left subtree
traverse the right subtree

**POSTORDER TRAVERSAL**

traverse the left subtree
visit the root
traverse the right subtree

**INORDER TRAVERSAL**

traverse the left subtree
traverse the right subtree
visit the root

Three other traversal algorithms may be defined which are the reverse of the above three.

**REVERSE PREORDER TRAVERSAL**

visit the root
traverse the right subtree
traverse the left subtree

**REVERSE POSTORDER TRAVERSAL**

traverse the right subtree
visit the root
traverse the left subtree

**REVERSE INORDER TRAVERSAL**

traverse the right subtree
traverse the left subtree
visit the root
Examination of representations (b) and (c) shows a preorder traversal is performed on the binary tree. As cited above, the ordering problem results when some subtree has only one son. This problem can be solved by introducing a representation for null trees as pointed out above. However, this solution would still give representations that do not resemble trees, so we choose not to do this but rather choose a different approach which makes use of a postorder or a reverse postorder traversal. If a postorder traversal or reverse postorder traversal is performed, the order of the subtree nodes will be retained since the root of a subtree will be printed between its sons. In this way, the ordering problem is avoided and the complete structure and ordering information can be displayed in a straightforward manner. This observation, which leads to a contradiction of Knuth's statement in which he says representations (a), (b) and (c) "have no resemblance to actual trees", is the basis of the following display algorithm based on representation (c). The algorithm can be used to display, as well as a binary search tree, any binary tree where there is no linear ordering defined upon the set of possible keys by some transitive relation.

**Definition**

We say a display of a binary tree is canonical iff a left (right) son is printed to the left (right) of its father.

II. **Binary Tree Display Algorithm - DISPLAY (root, indent, nodeline, width)**

The implementation of this algorithm fulfills the following two conditions:
(a) the displayed tree will be in canonical form and
(b) the tree will be displayed, rotated through 90° for convenience,
    with the root to the left of the page and its successors to its
    right. Therefore the key of the rightmost node will be printed
    first.

Given these conditions a reverse postorder traversal must be
used. However, the traversal algorithm can easily be changed to suit
one's individual taste. As the traversal proceeds down the tree a variable
called level, is incremented by one at each level of the tree. The
variable level is used, when the visit routine is invoked, to determine
the indentation of the key to be printed (although not explicitly evalua-
ted the indentation is equal to level \times indent). This establishes an
indented linear ordering of the tree much like a freehand drawing (see
Figure Al-4 to Al-7). The structure is easily discernible from the dis-
play. The parameter root is simply a pointer variable which points to
the root node of the tree to be displayed. There may be more fields in
a record representing a node, other than the key field, which also may
be displayed; however care should be taken not to exceed the number of
print positions available on a print line.

To clarify the structure, branches are also printed from each
node to its sons. As will be noticed from Figures Al-4 to Al-7, the keys
are printed a fixed number of print lines apart, in fact, "width" print
lines. Hence branches which leave a key at the "nodeline"th character
will necessarily be printed in segments of length "width" at a time.
As each print line is set up, it is necessary to know when a branch
character should or should not be printed. This information is given
by the BOOLEAN ARRAY brprint for level i. To ensure that keys are printed "width" print lines apart, a BOOLEAN variable f is used (initially false) which is set to true whenever a key has been printed (see procedure visit), thus ensuring that procedure prntbranch will be called before another key is printed. Procedure prntbranch always resets f to false before returning, thus ensuring that prntbranch will not be called until another key has been printed. Finally, it should be noted that nodeline is restricted within DISPLAY, to be in the range 1 \leq nodeline < 2 \times indent. With a little thought it will be realized that allowing nodeline to have values greater than 2 \times indent will, in general, cause confusion in the displayed tree. The procedure DISPLAY together with illustrative printouts (Figures A1-4 to A1-7) of the tree given in Figure A1-1 (except that each key is made up of 5 copies of each letter, for clarification purposes) are included in the following.
Example of binary tree

Figure Al-1
Nested sets representation

Figure Al-2

D

B

A

C

E

H

F

I

Indentation représentation

Figure Al-3
THERE ARE 8 NODES IN THIS TREE

THE ROOT OF THIS TREE IS DDDDD

INDENT = 8  WIDTH = 4  NODELINE = 1

Figure Al-4
THERE ARE 8 NODES IN THIS TREE
THE ROOT OF THIS TREE IS D0000

INDENT = 8 WIDTH = 4 NODELINE = 5

--- I III

--- HH HHH

--- F F FFF

--- E EEEE

D0000

--- C C CCC

--- 3333 B

--- A AAAA

Figure A1-5
THERE ARE 8 NODES IN THIS TREE
THE ROOT OF THIS TREE IS DDDDD

INDENT = 8 WIDTH = 4 NODELINE = 8
THERE ARE 8 NODES IN THIS TREE
THE ROOT OF THIS TREE IS DODDD

INDENT = 8 WIDTH = 4 NODELINE = 12

Figure AI-7
THE PROCEDURE DISPLAY

PROCEDURE DISPLAY(ROOT:POINT;INDENT,WİDTH,NODELINE:INTEGER);

PURPOSE: TO DISPLAY A BİNARY TREE IN A READABLE FORMAT

GLOBAL TYPE(S):
DIRECTION--A SCALAR TYPE USED TO INDICATE THE DIRECTIONS WHICH
MAY BE FOLLOWED FROM A NODE IN DIRECTION = (RIGHT,LİFT)
POINT------_VARIABLES OF THIS TYPE ARE POINTERS TO TREE NODES

LOCAL CONSTANT(S)
PRINTLIM--THE DIMENSION OF THE BOOLEAN ARRAY PRARRAY. IT INDICATES
THE NUMBER OF LEVELS OF THE BİNARY TREE WHICH CAN BE
PRINTED ON A PAGE AND MUST BE DETERMINED BY THE USER.
MAX--------_THE MAXIMUM NUMBER OF CHARACTERS WHICH ARE ALLOWED
IN A KEY

LOCAL VARIABLE(S)
BRPRINT---A BOOLEAN ARRAY USED TO INDICATE IF A BRANCH-CHARACTER
SHOULD BE PRINTED FROM A NODE ON A PARTICULAR LEVEL
IN THE TREE. IF BRPRINT[I] = TRUE A BRANCH-CHARACTER
MUST BE PRINTED.
F---------_BOOLEAN VARIABLE INDICATING IF TRUE SECTIONS OF BRANCHES SHOULD BE PRINTED IN THE
NEXT PRINT LINE
FALSE: THE KEY OF THE NEXT NODE SHOULD BE PRINTED IN
NEXT PRINT LINE

CONST
PRINTLIM = 33;
MAX = 30;

TYPE
BRARY = ARRAY[1..PRINTLIM] OF BOOLEAN;

VAR
BRPRINT: BRARY;
F: BOOLEAN;

PROCEDURE SPACE(I:INTEGER);
PURPOSE: TO PRINT THE NUMBER OF SPACES INDICATED BY ITS PARAMETER

VAR J:INTEGER;
BEGIN SPACE:
FOR J := 1 TO I DO WRITE(' E E')
END; SPACE


PROCEDURE PRNTBRANCH(LEVEL:INTEGER; BRPRINT:ARRAY);  

PURPOSE: TO PRINT THE CHARACTERS (COLONS) OF THE SEGMENTS OF THE 
BRANCHES BETWEEN A NODE JUST VISITED AND THE NEXT NODE TO 
VISITED (THIS IS A DISTANCE OF WIDTH PRINT LINES)

VAR M,N:INTEGER; BEGIN PRNTBRANCH + 
FOR N = 1 TO WIDTH DO 
BEGIN 
  *CARRIAGE CONTROL AND INITIAL SPACING TO FIRST BRANCH- 
  CHARACTER POSITION* 
  SPACE(NODELINE); 
  *PRINT A BRANCH CHARACTER (COLON) AT THIS POSITION IF A 
  BRANCH EXISTS (BRPRINT[M] = TRUE) WITH REQUIRED SPACING 
  NEXT POTENTIAL BRANCH POSITION* 
  FOR M = 1 TO LEVEL DO 
  IF BRPRINT[M] THEN 
  BEGIN 
    WRITE(EOL); 
    SPACE(INDENT - 1) 
  END ELSE SPACE(INDENT); 
  WRITE(EOL) 
  END; 
  *RESET FLAG INDICATING NEXT PRINT LINE WILL CONTAIN A KEY* 
  F = FALSE 
END; *PRNTBRANCH* 

PROCEDURE VISIT(P:POINT; LEVEL:INTEGER; BRPRINT:ARRAY);  

PURPOSE: TO PRINT THE KEY OF THE NODE POINTED TO BY P, HOWEVER IT 
ALSO NECESSARY TO (A) PRINT CHARACTERS OF PRECEDING 
(COLON) SO AS THEY ARE DISPLAYED AS CONTINUOUS BRANCHES 
(B) PRINT FILLER-CHARACTERS (MINUS SIGNS) PRECEDING THE KEY 
LINK IT TO ITS FATHERS BRANCH (C) PRINT FILLER-CHARACTERS 
FOLLOWING THE KEY TO LINK IT TO ITS SONS BRANCH 

VAR I,J:INTEGER; BEGIN VISIT + 
  *CARRIAGE CONTROL* 
  SPACE(1); 
  *IF THE NODE IS NOT THE ROOT NODE, PRINT AND FILLER-CHARACTERS 
  IN THE SAME PRINT LINE AS THE PRESENT KEY* 
  IF P NE ROOT THEN 
  *TWO CASES ARISE: EITHER NODELINE<INDENT OR NODELINE=1
(AND BY DEFINITION NODELINE<2*INDENT)↓

IF NODELINE GT INDENT
THEN

*IF NODE TO BE VISITED IS A SON OF THE ROOT NODE
(LEVEL = 1), NO BRANCH-CHARACTERS WILL BE PRINT
HENCE, SPACE TO THE FIRST PRINT POSITION OF THE

IF LEVEL EQ 1
THEN SPACE(INDENT)
ELSE
BEGIN

*SPACE TO THE FIRST POTENTIAL BRANCH CHARACTER POSITION↑
SPACE(NODELINE - 1);

*PRINT A BRANCH-CHARACTER (BRPRINT[I] TRUE) WITH REQUIRED SPACING TO THE
NEXT POTENTIAL BRANCH-CHARACTER POSITION↑
FOR I = 1 TO LEVEL - 2 DO
IF BRPRINT[I] THEN
BEGIN
WRITE(#13);
SPACE(INDENT + 1)
END
ELSE SPACE(INDENT);
IF BRPRINT[LEVEL - 1] THEN
BEGIN
WRITE(#13);
SPACE(2*INDENT - NODEL
END
ELSE SPACE(2*INDENT-NODELINE+1

ELSE
BEGIN

*SPACE TO FIRST POTENTIAL BRANCH-CHARACTER POSITION↑
SPACE(NODELINE - 1):

*PRINT A BRANCH-CHARACTER (BRPRINT[I] = TRUE WITH REQUIRED SPACING TO THE NEXT POTENTI
BRANCH-CHARACTER POSITION↑
FOR I = 1 TO LEVEL - 1 DO
IF BRPRINT[I] THEN
BEGIN
WRITE(#13);
SPACE(INDENT - 1)
END
ELSE SPACE(INDENT);

END
PRINT FILLER CHARACTERS (MINUS SIGNS) BEFORE KEY IF NECESSARY

SPACE(1):
IF INDENT - NOEDLINE GT 1
THEN
BEGIN
FOR I := 1 TO INDENT-NOEDLINE-1 DO WRITE(=-);
END;
SPACE(1)
END;

PRINT THE KEY
UNPACK(P+.INFO, CHAY, 1);
I := 1;
WHILE CHAY[I] NE EAE DO
BEGIN
WRITE(CHAY[I]);
I := I + 1;
IF I GT MAX THEN GOTO 10;
END;

PRINT FILLER CHARACTERS (MINUS SIGNS) AFTER KEY UNLESS IT IS A LEAF
IF(P+.LPT NE NIL) OR (P+.RPT NE NIL)
THEN
BEGIN
SPACE(1);
I := I + 1;
FOR J := I TO NOEDLINE - 1 DO WRITE(=-);
END;
WRITE(EOL);
SET FLAG INDICATING NEXT WIDTH PRINT LINES WILL CONTAIN BRANCH CHARACTERS
F := TRUE
END: VISIT

PROCEDURE TRAVERSE(P+POINT; LEVEL; INTEGER; WAY; DIRECTION);
PURPOSE TO PERFORM A REVERSE POSTORDER TRAVERSAL OF THE BINARY TREE AND INITIATE THE PRINTING OF KEYS AND BRANCHES
BEGIN TRAVERSE
WHEN P IS NIL THE TRAVERSAL CAN PROCEED NO FURTHER
IF P NE NIL
THEN
BEGIN
CASE WAY OF
RIGHT+BRPRINT(LEVEL) := FALSE;
LEFT+BRPRINT(LEVEL) := TRUE
END;
TRAVERSE(P+.RPT, LEVEL + 1, RIGHT);
IF F THEN
PRNTBRANCH(LEVEL, BPRINT);
CASE WAY OF
    \hspace{1cm} LEFT BPRINT[LEVEL] := FALSE;
    \hspace{1cm} RIGHT BPRINT[LEVEL] := TRUE
END;
VISIT(P, LEVEL, BPRINT);
TRAVERSE(P* LPT, LEVEL + 1, LEFT);
IF F THEN
PRNTBRANCH(LEVEL, BPRINT);
END; TRAVERSE

BEGIN DISPLAY
   INITIALIZE
   F := FALSE;
   \hspace{1cm} RESTRICT NODELINE TO BE LESS THAN 2*INDENT
   IF NODELINE GE 2*INDENT THEN NODELINE := 2*INDENT - 1;
   TRAVERSE(ROOT, 0, RIGHT)
END; DISPLAY
APPENDIX 2

UTILITY ROUTINES

The Standard Pascal Procedures New, Pack, Unpack and Get

New(p) is a dynamic allocation procedure which allocates a new variable and assigns the pointer to the variable to the pointer variable p. Assuming that b is a CHARACTER array variable, z, is an ALFA variable and i is an integer expression, then pack(b,i,z) packs the n characters b[i]...b[i+n-1] into the ALFA variable z and unpack(z,b,i) unpacks the ALFA value z into the variables b[i]...b[i+n-1]. Get is a file positioning procedure, get(f) advances the file pointer of file f to the next file component when the file is in the input mode.

The Routines Random and Shuffle

These two routines are used in the statistics collection procedures for the unweighted trees. Their listings follow.

The Routines Attention and Time

These two routines allow access to the central processor clock.

The Routine Readword

This boolean function is used to read the keys of the nodes. It reads characters and converts them into a variable of type ALFA, left justified zero-filled, using the standard Pascal procedure pack. Blank is used as a delimiter and any string containing more than ten characters is truncated. If a string has fewer than ten characters, its remaining character positions are zero-filled. The routine returns a value of
false when an end of file is detected or the special stop-character, :, is read.

The List Management Routines Acquire, Release and Classoverflow

These three procedures are used to manage the allocation of nodes and to collect nodes which have been deleted from the tree and would otherwise be lost and not available for future use. Procedure release links together any nodes deleted from the tree using the right-pointer field of the record representing the node. The variable free is used as a pointer to the first node in this list called the "free list". Function acquire allocates nodes (i.e. returns a pointer to a node). If the free list is not empty (free ≠ nil), then acquire returns the pointer to the last node entered into the free list and free is made to point to the succeeding node in the free list. If free = nil, the free list is empty and acquire tries to allocate a node by using the standard Pascal procedure new. If new returns the pointer nil, the maximum number of nodes which were declared are all allocated and in use. Here acquire calls classoverflow to print a message and the function returns as false. This signals, in the implemented algorithms, an emergency exit which terminates the particular algorithm.
UTILITY ROUTINES

FUNCTION RANDOM(A,B:REAL;VAR Y:INTEGER):REAL;

PURPOSE: RANDOM generates a pseudo-random number in the open interval
(A,B) where A < B.

DESCRIPTION: The procedure assumes that integer arithmetic up to
3125*67108863 = 209715196875 is available. The actual
parameter corresponding to Y must be an integer identifier
and at the first call of the procedure its value must be
an odd integer within the limits 1 to 67108863 inclusive.
If a correct sequence is to be generated, the value of this
integer must not be changed between successive calls of
of the function. (H.C. Pike, I.D. Hill, Algorithm 266
COMM ACM 8 (Oct. 1965); P605)

BEGIN RANDOM

Y := 3125*Y;
Y := Y - (Y DIV 67108864)*67108864;
RANDOM := Y/67108864.0*(B - A) + A.

END; RANDOM

-----------------------------------------------

PROCEDURE SHUFFLE(VAR A:TYPEOFARRAY;VAR SEED:INTEGER;N,K:INTEGER);

PURPOSE: SHUFFLE applies a random permutation to the sequence A[i]
I = 1,...,N in such a way that after K calls of the procedure
random the elements A[i] for i = N - K + 1,...,N - K + 2,...,N
are a random permutation of the original N elements A[i] where
I = 1,2,...,N taken K at a time.

DESCRIPTION: The procedure RANDOM is supposed to supply a random element
from a large population of real numbers uniformly
distributed over the open interval (0,1). The array A
is declared to be the same type as the variable SAVE. Note
that at exit A[i..N] will still contain all the elements of
the original A[i..N] and that if K = N SHUFFLE applies
a random permutation to the complete sequence. (H.C. Pike
Remark on Algorithm 235 COMM ACM 7 (1965) P445)

VAR I,J:INTEGER;
SAVE:INTEGER;

BEGIN SHUFFLE

K := N + 1 - K;
FOR I = N DOWNTO K DO
BEGIN
    J := TRUNC((I*RANDOM(0.0,1.0,SEED)) + 1);
    SAVE := A[I];
    A[J] := SAVE
END

END; SHUFFLE
PROCEDURE ATTENTION(C:INTEGER);
BEGIN ATTENTION
    MEM[1] := C;
    WHILE MEM[1] ≠ 0 DO;
END ATTENTION

PROCEDURE TIME(VAR M:INTEGER);
VAR
    REC : RECORD CASE B:BOOLEAN OF
        TRUE: (ALF:ALFA);
        FALSE: (INT:INTEGER);
    END;
    T : ARRAY[1..ALFLEN] OF CHAR;
    I : INTEGER;
BEGIN TIME
    ATTENTION(241115003000000006428);
    REC.INT := MEM[428];
    UNPACK(REC.ALF,T,1);
    FOR I := 1 TO 4 DO T[I] := CHR(0);
    PACK(T,1,REC.ALF);
    M := REC.INT MOD 4096 + (REC.INT DIV 4096) * 1000;
END TIME

FUNCTION READWORD(VAR WORD:ALFA):BOOLEAN;
PURPOSE: TO READ THE CHARACTERS WHICH FORM THE KEYS OF THE NODES AND CONVERT THEM INTO AN ALFA VARIABLE

OUTPUT PARAMETERS:
    WORD--THE KEY OF THE NODE.
NESTED PROCEDURES--BIGALFA,EOLFILL
VAR LENGTH:INTEGER;
PROCEDURE BIGALFA;

PURPOSE: TO READ THE REMAINING CHARACTERS OF STRINGS LONGER THAN 10 CHARACTERS AND PRINT A MESSAGE AS TO TRUNCATION OF THE STRING.

VAR I: INTEGER;
BEGIN BIGALFA
   WRITE("0ITEM TRUNCATES TO 10 CHARACTERS");

   WRITE THE KEY
   FOR I := 1 TO MAX DO
      WRITE(CHAI[I]);
   WHILE (INPUT ≠ EOL) DO
      BEGIN
         WRITE(INPUT);
         GET(INPUT)
      END;
   WRITE(EOL)
END; BIGALFA

PROCEDURE EOLFILL(LENGTH: INTEGER);
PURPOSE: TO EOL-FILL THE REMAINING CHARACTERS OF A STRING OF LESS THAN 10 CHARACTERS

VAR I, J: INTEGER;
BEGIN EOLFILL
   I := LENGTH + 1;
   FOR J := I TO MAX DO
      CHAI[I] := EOL
END; EOLFILL
BEGIN »READWORD«

»SPAN BLANKS AND END OF LINE«

WHILE (INPUT* EQ » ») OR (INPUT* EQ EOL) DO
BEGIN

»CHECK FOR END OF FILE«

IF EOF(INPUT)
THEN
BEGIN
READWORD := FALSE;
GOTO 6789
END;
GET(INPUT);
END;

»CHECK FOR SPECIAL STOP-CHARACTER«

IF INPUT* EQ STOPCHAR
THEN
BEGIN
READWORD := FALSE;
GET(INPUT);
GOTO 6789
READWORD := TRUE;
LENGTH := 0;

»READ THE CHARACTERS OF THE KEY«

WHILE (INPUT* NE » ») AND (INPUT* NE EOL) DO
BEGIN
IF LENGTH EQ MAX:
THEN

»THE KEY EXCEEDS 10 CHARACTERS, TRUNCATE IT«

BEGIN
BIGALFA;
GOTO 10
END
ELSE
BEGIN
LENGTH := LENGTH + 1;
CHAY[LENGTH] := INPUT*;
GET(INPUT)
END
END;

»ZERO-FILL THE CHARACTER ARRAY«

EOLFILL(LENGTH);
GET(INPUT);

»CREATE THE ALFA VARIABLE WORD«

PACK(CHAY,1,WORD);

6789: END; »READWORD«
PROCEDURE CLASSOVERFLOW;
BEGIN *CLASSOVERFLOW*
    WRITE(EOL,EOL,EOL,EOL,CLASSOVERFLOW,EOL);
END; *CLASSOVERFLOW*

-----------------------------------------------------

FUNCTION ACQUIRE(VAR P:POINT):BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO RETURN A POINTER TO A NODE. ACQUIRE FIRST TRIES TO FIND A NODE IN THE FREE LIST AND IF THIS FAILS IT TRIES TO ALLOCATE A NEW NODE USING THE STANDARD PASCAL PROCEDURE NEW. IF ALL NODES HAVE BEEN ALLOCATED AND ARE IN USE THE OUTPUT PARAMETER IS RETURNED AS NIL AND THE FUNCTION IS TRUE.

OUTPUT PARAMETERS:

P--POINTER TO THE NEW NODE

BEGIN *ACQUIRE*

    ACQUIRE := FALSE;
    IF FREE EQ NIL THEN
        BEGIN *TRY TO ALLOCATE A NEW NODE; IF THIS FAILS PRINT A MESSAGE, AND RETURN THE FUNCTION VALUE TRUE*
            BEGIN
                NEW(P);
                IF P EQ NIL THEN
                    BEGIN
                        CLASSOVERFLOW;
                        ACQUIRE := TRUE
                    END
                ELSE
                    BEGIN *TAKE THE NODE FROM THE FREE LIST*
                        BEGIN
                            P := FREE;
                            FREE := FREE+.RPT
                        END
            END
        END
    ELSE
END; *ACQUIRE*
PROCEDURE RELEASE(P:POINT);

PURPOSE: To place a node (pointed to by the given input parameter) deleted from the tree on the free list. The right pointer fields of the nodes are used to form the chain.

INPUT PARAMETERS:
  P -- pointer to the deleted node

BEGIN RELEASE:
  IF FREE EQ NIL THEN
    «There is only one node in the free list; define FREE to point to it»
    BEGIN
      FREE := P;
      FREE+.RPT := NIL
    END
  ELSE
    «Place the node on the free list»
    BEGIN
      P+.RPT := FREE;
      FREE := P
    END
  END; "RELEASE"
APPENDIX 3

LISTING OF THE BINARY TREE CONSTRUCTION ALGORITHMS

Global constants, types and variables common to most algorithms are described in the following. The basic record structure of the nodes is given in Section 3.1.2 and any additional fields added to the record are described with each algorithm.

Common Global Constants

max ... the maximum number of characters packed into an ALFA variable
stopchar ... a special character used in the function readword to indicate end of data

Common Global Types

direction ... indicates which way to go from one node in the tree to get to one of its sons, i.e. left or right.
point ... variables of this type are pointers to tree nodes

Common Global Variables

delay ... the CHARACTER array used by the standard Pascal procedure pack, to hold the array of characters which are packed into an ALFA variable
current ... pointer to the current node under examination
father ... pointer to the father of the node currently under examination
free ... the free list pointer
head ... pointer to the special header node defined for certain trees

inword ... the ALFA variable representing the key

nodecount ... variable giving the number of nodes in a particular tree

way ... a variable of type direction which assumes values left or right

In the listing presented below any statements in a comment were used in collecting statistics.
GLOBAL CONSTANTS, TYPES, VARIABLES

LABEL

CONST
MAX = 1;
STOPCHR = "E";

TYPE
POINT = TREE;
DIRECTION = (RIGHT, LEFT);
NODE = RECORD
INFO: ALFA;
LPT: POINT;
RPT: POINT;
END;

VAR
TREE: CLASS 2, C OF NODE;
FREE, HEAD, CURRENT, FATHER: POINT;
WAY: DIRECTION;
INCHORD: ALFA;
CHAY ARRAY[1..MAX] OF CHAP;
NODECOUNT: INTEGER;

PROCEDURE CREATEHEADER:
PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS
BEGIN CREATEHEADER

* INITIALIZE FREE LIST POINTER AND NODE COUNTER *
FREE := NIL;
NODECOUNT := 0;

* CREATE HEADER NODE *
IF Acquire(HEAD) THEN GOTO: EXIT 999;
HEAD+.RPT := NIL;
HEAD+.LPT := NIL;

END CREATEHEADER

PROCEDURE MAKE_NODE(WORD: ALFA; P: POINT):
PURPOSE: TO DEFINE THE FIELDS OF A NODE AND INCREMENT NODE COUNT
PARAMETER(S):
WORD - KEY OF NEW NEW NODE
P - POINTER TO THE NEW NODE
BEGIN MAKE_NODE

P+.INFO := WORD;
P+.LPT := NIL;
P+.RPT := NIL;
NODECOUNT := NODECOUNT + 1;

END MAKE_NODE
FUNCTION SEARCHITEM (WORD: ALFA): BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO DETERMINE IF THERE IS A NODE IN THE TREE WITH GIVEN INPUT KEY. ON EXIT IF THE FUNCTION IS TRUE, THE KEY IS NOT PRESENT.

PARAMETER(S):
    WORD—the key being searched

BEGIN  SEARCHITEM
    SEARCHITEM := TRUE;
    // INITIALIZE POINTERS TO THE HEADER NODE AND THE ROOT OF THE TREE
    FATHER := HEAD;
    CURRENT := HEAD».RPT;
    WAY := RIGHT;
    // SEARCH FOR THE KEY
    WHILE CURRENT.NE NIL DO
        IF CURRENT».INFO NE WORD
            THEN
                FATHER := CURRENT;
                IF WORD GT CURRENT».INFO
                    THEN
                        BEGIN
                            CURRENT := CURRENT».RPT;
                            WAY := RIGHT;
                        END
                    ELSE
                        BEGIN
                            CURRENT := CURRENT».LRT;
                            WAY := LEFT;
                        END
                ELSE
                    KEY IS PRESENT
                    BEGIN
                        SEARCHITEM := FALSE;
                        GOTO 1;
                    END
            END
    END

10: END; SEARCHITEM

END;
PROCEDURE BASICINSERT(WORD:ALPHA);

PURPOSE: TO CREATE A NEW NODE WITH GIVEN INPUT KEY
PARAMETER(S):
  WORD--THE KEY OF THE NEW NODE TO BE CREATED
BEGIN • BASICINSERT
  • DETERMINE IF KEY IS ALREADY PRESENT
  IF SEARCHITEM(WORD) THEN
    • KEY IS NOT PRESENT: CREATE A NEW NODE HAVING THIS KEY
    BEGIN
      IF ACQUIRE(CURRENT) THEN GOTO EXIT 999;
      MAKE NODE(WORD,CURRENT);
      • LINK NEW NODE TO ITS FATHER
      CASE WAY OF
      RIGHT+FATHER\+RPT \= CURRENT:
        LEFT+FATHER\+LPT \= CURRENT;
    END;
  END; • BASICINSERT
END;

PROCEDURE RELINK(P,Q:POINT):

PURPOSE: TO LINK A SUBTREE WHOSE ROOT IS POINTED TO BY A GIVEN
POINTER TO THE TREE.

PARAMETER(S):
  P--POINTER TO THE ROOT OF THE SUBTREE TO BE RELINKED
  Q--POINTER TO THE DELETED NODE
BEGIN • RELINK
  IF FATHER\+RPT \= Q THEN FATHER\+RPT \= P
  ELSE FATHER\+LPT \= P;
END • RELINK
FUNCTION TWOSUBTREE(P:POINT):BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO DETERMINE IF A NODE SPECIFIED BY AN
INPUT POINTER HAS TWO NON-NULL SUBTREES AND IF NOT TO
DELETE THIS NODE AND RELINK ITS SUBTREE (IF ANY). ON EXIT
TWOSUBTREE = TRUE IF THE NODE HAS TWO NON-NULL SUBTREES.

PARAMETER(S):
    P -- POINTER TO THE NODE TO BE DELETED

BEGIN TWOSUBTREE;
    TWOSUBTREE := TRUE;
    IF P^RROT = NIL
        THEN BEGIN
            TWOSUBTREE := FALSE;
            RELINK(P^LROT,P);
            RELEASE(P);
        END
        ELSE IF P^LROT = NIL
            THEN BEGIN
                TWOSUBTREE := FALSE;
                RELINK(P^RROT,P);
                RELEASE(P);
            END
    END
END TWOSUBTREE;
PROCEDURE BASICDELETE(WORD: ALFA):
PURPOSE: TO DELETE FROM THE TREE A NODE HAVING GIVEN INPUT KEY
PARAMETER(S):
WORD--THE KEY OF THE NODE TO BE DELETED
VAR
ONODE--POINTER TO THE NODE TO BE DELETED
PRED--POINTER TO THE POSTORDER PREDECESSOR OF THE NODE TO BE DELETED

BEGIN BASICDELETE

• DETERMINE IF THE KEY IS PRESENT IN THE TREE
IF SEARCHITEM(WORD)
THEN
• THE KEY IS PRESENT
BEGIN
• DECREMENT NODE/COUNT
NODECOUNT := NODECOUNT - 1;
• CHECK IF NODE TO BE DELETED HAS TWO
NON-NULL SUBTREES
IF TWOSUBTREES(CURRENT)
THEN
BEGIN
• THE NODE TO BE DELETED HAS TWO NON-NULL
SUBTREES; REPLACE THE NODE WITH ITS POSTORDEREFFICESSOR AND THEN DELETE THIS PREDECESSOR

ONODE := CURRENT;
PRED := CURRENT;
CURRENT := CURRENT*LPT;
REPEAT
FATHER := PRED;
PRED := CURRENT;
CURRENT := CURRENT*PRT;
UNTIL (CURRENT = NIL);
ONODE*INFO := PRED*INFO;
IF TWO_SUBTREES(PRED)
THEN THE
END;

END:

END: BASICDELETE
BEGIN DRIVER FOR BASIC BINARY TREE
  • PERFORM NECESSARY INITIALIZATIONS
  CREATE HEADER:
  • INSERT KEYS INTO THE TREE
  WHILE READWORD(INWORD) TO
    BASICINSERT(INWORD):
  • DELETE NODES FROM THE TREE
  WHILE READWORD(INWORD) TO
    BASICDELETE(INWORD):
  9999 END. DRIVER FOR BASIC BINARY TREE
AVL TREE

GLOBAL CONSTANTS, TYPES, VARIABLES:

BALFAC -- AN ADDITIONAL FIELD CONTAINED IN ANY RECORD REPRESENTING A

NODE AND USED TO CONTAIN THE BALANCE FACTOR OF THE NODE.

MAXDEPTH -- THE DIMENSION OF THE DELETION STACK.

END

PROCEDURE CREATEHEADER;
PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS
BEGIN: CREATEHEADER;
  INITIALIZE FREE LIST POINTERS AND NODE COUNT;
  NODECOUNT = 1;
  FREE = NIL;
  CREATE HEADER NODE;
  IF ACOUNT (HEAD) THEN GOTO EXIT 999;
  HEAD = PRT = NIL;
END: CREATEHEADER;

PROCEDURE TAKENODE (NO = NODE, ALFA = ALPHA, PRT = POINT);
PURPOSE: TO CREATE THE FIELDS OF THE NEW NODE AND TO INCREMENT THE

NODE COUNT.
BEGIN: TAKENODE;
  NODECOUNT = NODECOUNT + 1;
  INFO = NO;
  BALFAC = ALFA;
  PRT = NIL;
  LRT = NIL;
END: TAKENODE;

MAX = 1;
STOPCCHAR = EOL;
MAXDEPTH = 4;
PROCEDURE REPORT(T,S,P;POINT):  
PURPOSE: TO RELINK A ROTATED SUBTREE TO ITS FATHER  
PARAMETERS INPUT:  
P--POINTER TO THE ROOT OF THE ROTATED SUBTREE  
S--POINTER TO THE OLD ROOT OF THE SUBTREE  
T--POINTER TO THE FATHER OF S  
BEGIN REPORT  
IF T\star.ROOT \neq S  
THEN T\star.ROT := P  
ELSE T\star.LPT := P  
END; REPORT  

PROCEDURE SROTATE(T,S,R;POINTER;RVAL;INTEGER):  
PURPOSE: TO PERFORM A SINGLE ROTATION USING THE GIVEN INPUT POINTERS.  
PARAMETERS INPUT:  
RVAL--INDICATES THE DIRECTION OF THE ROTATION (+1 TO THE LEFT  
-1 TO THE RIGHT)  
T--POINTER TO THE FATHER OF THE SUBTREE TO BE ROTATED  
S,R--POINTER TO PARENT NODES USED IN THE ROTATION  
BEGIN SROTATE  
PERFORM THE APPROPRIATE ROTATION.  
IF RVAL = -1  
THEN BEGIN  
S\star.LPT := R\star.PPT;  
R\star.ROT := S  
END  
ELSE BEGIN  
S\star.ROT := R\star.LPT;  
R\star.LPT := S  
END;  
ADJUST THE BALANCE FACTORS OF THE NODES USED IN THE ROTATION  
S\star.BALFACTOR := RVAL;  
R\star.BALFACTOR := -RVAL;  
RELINK THE ROTATED SUBTREE TO ITS FATHER  
REPORT(T,S,P);  
END; SROTATE.
PROCEDURE DROTA TE(T, S, R; POINT: RALVAL: INTEGER);

PURPOSE: TO PERFORM A DOUBLE ROTATION USING THE GIVEN INPUT POINTERS.

PARAMETERS INPUT:

RALVAL—INDICATES THE DIRECTION OF THE ROTATION (+1 TO THE LEFT
-1 TO THE RIGHT)
T—POINTER TO THE FATHER OF THE SUBTREE TO BE ROTATED
S, R—POINTERS TO THE RELEVANT NODES USED IN THE ROTATION

VAR POINT;

BEGIN DROTA TE;

PERFORM THE APPROPRIATE ROTATION:

IF RALVAL EQ 1 THEN

BEGIN

P*: LPT := P*: LPT;

P*: RPT := P*: RPT;

S*: RPT := S*: RPT;

P*: LPT := P*: LPT;

P*: LPT := S

END;

ELSE

BEGIN

S*: RALFAC := S;

R*: RALFAC := R;

END;

END:

ADJUST THE BALANCE FACTORS OF THE NODES USED IN THE ROTATION:

IF P*: RALFAC EQ 0 THEN

BEGIN

S*: RALFAC := 0;

R*: RALFAC := 0;

END;

ELSE

BEGIN

IF P*: RALFAC EQ RALVAL THEN

BEGIN

S*: RALFAC := -RALVAL;

R*: RALFAC := 0;

END;

ELSE

BEGIN

S*: RALFAC := 0;

P*: RALFAC := RALVAL;

END;

END;

RETURN THE ROTATED SUBTREE TO ITS FATHER:

REPO D(T, S, P);

END: DROTA TE;
FUNCTION SEARCHITEM(INORDIALFA:V, FATHCRIT, CRIT, FATHER:POINT):BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO DETERMINE IF THE GIVEN INPUT KEY IS PRESENT
IN THE TREE. ON EXIT IF THE FUNCTION IS TRUE, THE KEY IS
NOT PRESENT.

PARAMETERS INPUT:

WORD--THE KEY BEING SEARCHED

PARAMETERS OUTPUT:

CRIT--POINT TO THE CRITICAL NODE
FATHCRIT--POINT TO THE FATHER OF THE CRITICAL NODE
FATHER--POINT TO THE FATHER OF THE NODE TO BE CREATED

VAR

CURRENT--POINT TO THE NODE UNDER CURRENT EXAMINATION
CURRENTP:POINT:

BEGIN SEARCHITEM

*INITIALIZE POINTERS FOR SEARCH

SEARCHITEM := TRUE;
FATHER := HEAD;
CURRENT := HEAD.PRT;
FATHCRIT := HEAD;
CRIT := CURRENT;
WAY := RIGHT;

*SEARCH FOR THE KEY

IF CURRENT NE NIL
THEN

BEGIN

*CHECK FOR THE CRITICAL NODE

IF CURRENT.RALFA.NE. &
THEN

BEGIN

*BEGIN

11 := 1 + 1;

*DETERMINE THE NEXT NODE IN THE SEARCH SEQUENCE

IF CURRENT.INFO NE WORD
THEN

BEGIN

FATHER := CURRENT;

IF CURRENT.INFO GT WORD
THEN

BEGIN

CURRENT := CURRENT.PRT;
WAY := LEFT;

END

ELSE

BEGIN

CURRENT := CURRENT.RPT;
WAY := RIGHT;

END

END

END

END:
ELSE
    BEGIN
        SEARCHITEM := FALSE;
        GOTO 10
    END:
UNTIL (CURRENT EO NIL);
    *I1 := I1 + 1;
    I4 := I1 - I4;
END
ELSE
    I1 := 1;
END

10: END: SEARCHITEM.

---

PROCEDURE ADJUST(CRIT, NEWNODE, POINT, VAR SONSPOINT):
    VAR SONYVAL: INTEGER;

PURPOSE: TO DETERMINE IF THE CRITICAL NODE IS LEFT OR RIGHT HEAVY AND TO ADJUST THE BALANCE FACTORS BETWEEN THE CRITICAL NODE AND THE NEW NODE.

PARAMETERS INPUT:
    CRIT----POINTER TO THE CRITICAL NODE
    NEWNODE----POINTER TO THE NEW NODE.

PARAMETERS OUTPUT:
    BALVAL----INDICATES IF THE NEW NODE IS IN THE LEFT (-1) OR RIGHT (1) SUBTREE OF THE CRITICAL NODE
    SONSPOINT----POINTER TO THE NODE SUCCEEDING THE CRITICAL NODE IN THE SEARCH PATH

VAR, Q, POINT:

BEGIN ADJUST
    "DETERMINE THE NODE SUCCEEDING THE CRITICAL NODE IN THE SEARCH PATH. THIS NAME IS USED FOR ANY POSSIBLE REBALANCE.
    IF NEWNODE + .INFO LT CRIT + .INFO
        THEN
            BEGIN
                SONSPOINT := CRIT + .LPT;
                BALVAL := -1
            END
        ELSE
            BEGIN
                SONSPOINT := CRIT + .RPT;
                BALVAL := 1
            END
        
    "ADJUST THE BALANCE FACTORS BETWEEN THE CRITICAL NODE AND THE NEW NODE;
    Q := SONSPOINT;
    WHILE Q NE NEWNODE DO
        BEGIN
            IF NEWNODE + .INFO LT Q + .INFO
                THEN
                    BEGIN
                        Q + .BALFAC := -1;
                        Q := Q + .LPT
                    END
                ELSE
                    BEGIN
                        Q + .BALFAC := 1;
                        Q := Q + .RPT
                    END
            END
        END
    END

---
PROEDURE BALANCE(FATHCRIT, CRIT, SONECRIT, POINT, BALVAL; INTEGER);

PURPOSE: TO DETERMINE IF THE TREE HAS BECOME AVL UNBALANCED AND IF SO TO PERFORM A TRANSFORMATION TO RESTORE BALANCE

PARAMETERS INPUT:

BALVAL----INDICATES IF THE NEW NODE IS IN THE LEFT(-1) OR RIGHT(1) SUBTREE OF THE CRITICAL NODE
CRIT----POINTER TO THE CRITICAL NODE
FATHCRIT----POINTER TO THE FATHER OF THE CRITICAL NODE
SONECRIT----POINTER TO THE NODE SUCCEEDING THE CRITICAL NODE IN THE SEARCH PATH

BEGIN BALANCE

IF THE CRITICAL NODE HAS BALANCED NO ROTATION IS NECESSARY; READJUST THE BALANCE FACTOR OF THE CRITICAL NODE AND EXIT

IF CRIT*.BALFAC EQ 0 THEN CRIT*.BALFAC := BALVAL
ELSE IF CRIT*.BALFAC EQ -BALVAL THEN CRIT*.BALFAC := 0 ELSE

PERFORM THE APPROPRIATE ROTATION

IF SONECRIT*.BALFAC EQ BALVAL THEN BEGIN
    I2 := I2 + 1;
    SRotate(FATHCRIT, CRIT, SONECRIT, BALVAL)
    END
ELSE BEGIN
    I3 := I3 + 1;
    DRotate(FATHCRIT, CRIT, SONECRIT, BALVAL)
    END

END BALANCE.
PROCEDURE AVLINSERT(WORD,ALFA):
PURPOSE: TO CREATE A NEW NODE WITH GIVEN INPUT KEY
PARAMETERS INPUT:
  WORD--THE KEY OF THE NEW NODE TO BE CREATED
  VAP

  BALVAL--INTEGER VARIABLE WITH VALUE 1 OR -1. 1 INDICATES THE NEW NODE IS ADDED TO THE RIGHT SUBTREE OF THE CRITICAL NODE; -1 INDICATES THE LEFT SUBTREE
  CRIT--POINTER TO THE CRITICAL NODE
  FATHER--POINTER TO THE FATHER OF THE CRITICAL NODE
  NEWNODE--POINTER TO THE FATHER OF THE NODE TO BE CREATED
  FATHCRIT,CRIT,FATHER,SONCRIT,NEWNODE,POINT;
  BALVAL: INTEGER:

BEGIN AVLINSERT:
  DETERMINE IF THE KEY IS ALREADY PRESENT IN THE TREE
  IF SEARCHITEM(WORD,FATHCRIT,CRIT,FATHER)
  THEN
    CREATE THE NEW NODE AND LINK IT TO ITS FATHER
    BEGIN
      IF ACQUIRE(NEWNODE) THEN GOTO EXIT 999;
      CASE VALUE OF
        RIGHT:FATHER:RPT:=NEWNODE;
        LEFT:FATHER:LPT:=NEWNODE;
      END:
      MAKE_NODE(WORD,NEWNODE);
      ADJUST THE BALANCE FACTORS OF THE NODES IN THE SEARCH PATH BETWEEN THE CRITICAL NODE AND THE NEW NODE AND REBALANCE THE TREE IF NECESSARY. IF THERE IS ONLY ONE NODE IN THE TREE, NO NEED TO ADJUST OR BALANCE
      IF CRIT:NE:NULL
      THEN
        BEGIN
          ADJUST(CRIT,NEWNODE,SONCRIT,BALVAL);
          BALANCE(FATHCRIT,CRIT,SONCRIT,BALVAL);
        END
      END
    END
END: AVLINSERT
PROCEDURE AVLDELETE(WORD; ALPA);
PURPOSE: TO DELETE THE NODE WHOSE KEY IS SPECIFIED BY THE GIVEN INPUT KEY
NESTED PROCEDURES: PREDORSUCC, REBUILD, RELINK
PARAMETERS INPUT:
WORD - KEY OF THE NODE TO BE DELETED

TYPE
DELPT - A LOCAL TYPE DEFINING A RECORD USED AS A DELETION STACK ELEMENT. THE RECORD HAS TWO FIELDS
PT - A POINT TO A NODE IN THE DELETION SEQUENCE
DIR - THE DIRECTION TO PROCEED FROM THE NODE POINTED TO BY PT TO GET TO THE NEXT NODE IN THE DELETION SEQUENCE (+1=RIGHT, -1=LEFT),
BACK - A LOCAL TYPE: DEFINING THE DELETION STACK

DELPT = RECORD
PT: POINT;
DIR: INTEGER;
END;
BACK = ARRAY[1..AXDEPTH] OF DELPT;

VAP
BACKP - THE DELETION STACK POINTER TO THE NODE TO BE DELETED

BACKP = BACK;
I: INTEGER;
PT: POINT;

PROCEDURE PRECORSUCC(P:POINT):

PURPOSE: TO FIND THE POSTORDER PREDECESSOR OR SUCCESSOR OF THE NODE
POINTED TO BY P. IF P IS A LEFT HEAVY THEN THE SUCCESSOR IS
FOUND, ELSE THE PREDECESSOR IS FOUND. THE DELETION STACK IS
REBUILT DURING THIS PROCESS.

VAR R, R1, R2, P, L, L1, L2, LPT,
BEGIN R := PRED
//DETERMINING IF THE NODE IS LEFT OR RIGHT HEAVY.
I := I + 2
IF P.RALFAC = 1 THEN

//FIND THE SUCCESSOR REBUILDING THE STACK IN THE PROCESS.
BEGIN
RACKP[I] := -1, N := 1
N := P.RALFAC
RACKP[I], PT := N
RACKP[I], NP := -1
N := N.RALFAC
WHILE N := NTL DO
BEGIN
I := I + 1
RACKP[I], PT := N
RACKP[I], NP := -1
N := N.RALFAC
END
ELSE END

//FIND THE PREDECESSOR REBUILDING THE STACK IN THE PROCESS.
BEGIN
RACKP[I] := -1, N := -1
N := P.RALFAC
RACKP[I], PT := N
RACKP[I], NP := 1
N := N.RALFAC
WHILE N := NTL DO
BEGIN
I := I + 1
RACKP[I], PT := N
RACKP[I], NP := 1
N := N.RALFAC
END
END: PRECORSUCC

END.
PROCEDURE REBUILD:

PURPOSE: TO EXAMINE THE DELETION STACK, ADJUST THE BALANCE FACTORS OF NODES IN THE DELETION STACK AND INITIATE ANY ROTATIONS NECESSARY TO MAINTAIN BALANCE.

BEGIN REBUILD

IF DS = 05 + 1 THEN
    WITH BACKP[I] DO
        BEGIN
            IF PT+.BALFAC = 0 THEN
                BEGIN
                    *THE CURRENT CODE I: THE DELETION STACK WAS LEFT(RIGHT)
                    HEAVY AND A CODE WAS ADDED FROM ITS LEFT(RIGHT) SUBTREE
                    ITS BALANCE FACTOR BECOMES ZERO.*
                BEGIN
                    PT+.BALFAC := 0;
                    I := I - 1;
                    REBUILD
                END
            ELSE
                IF THE CURRENT CODE IN THE DELETION STACK HAD A BALANCE FACTOR OF 0, THE TREE WILL NOT BE UNBALANCED; ADJUST THE BALANCE FACTOR AND EXIT.
                IF PT+.BALFAC = 0 THEN PT+.BALFAC := -0;
            ELSE
                *PERFORM THE APPROPRIATE ROTATION*
                IF PT+.BALFAC = 1 THEN IF PT+.PT+.BALFAC = 0 THEN
                    *THE SPECIAL CASE WHERE A SINGLE ROTATION RESTORES BALANCE AND WE MAY EXIT*
                    BEGIN
                        I := 04 + 1;
                        STAT := (BACKP[I-1].PT,PT,PT+.RPT,1)
                    END
                ELSE BEGIN
                    IF PT+.PT+.BALFAC = 0 THEN
                        BEGIN
                            +DS := 03 + 1;
                            STAT := (BACKP[I-1].PT,PT,PT+.RPT,1)
                        END
                    ELSE
                        ...
                    END
                ELSE...
            END
        END
    END
END
BEGIN
  @O1 := O2 + 1;
  SRotate(BACKP[I-1], PT, PT, PT+.RPT, 1);
END;
I := I - 1;
REBUILD
END
ELSE
  PERFORM THE APPROPRIATE ROTATION
  IF PT+.LPT+.BALFAC EQ 0
    THEN
      THE SPECIAL CASE WHERE A SINGLE ROTATION
      RESTORES BALANCE AND WE MAY EXIT
      BEGIN
        @O4 := O4 + 1;
        SRotate(BACKP[I-1], PT, PT, PT+.LPT, -1)
      END
    ELSE
      BEGIN
        IF PT+.LPT+.BALFAC EQ 1
          THEN
            BEGIN
              @O3 := O3 + 1;
              SRotate(BACKP[I-1], PT, PT, PT+.LPT, -1)
            END
          ELSE
            BEGIN
              @O2 := @O2 + 1;
              SRotate(BACKP[I-1], PT, PT, PT+.LPT, -1)
            END
        END
      END
  END
END; /* REBUILD */
PROCEDURE RELINK;

PURPOSE: TO DELETE THE NODE POINTED TO BY P AND RELINK ITS SUBTREE
(IF ANY). IF THE NODE HAS TWO SONS ITS POSTORDER PREDECESSOR
OR SUCCESSOR, Z SAY, IS FOUND. KEY(Z) REPLACES THE KEY OF THE
NODE TO BE DELETED AND Z NOW BECOMES THE NODE TO BE DELETED.

BEGIN RELINK

P := BACKP[I].PT;
I := I - 1;
WITH BACKP[I] DO
  BEGIN IF P+.RPT EQ NIL THEN
      "THE NODE TO BE DELETED HAS NO RIGHT SON;
      DELETE THE NODE, RELINK ITS ONE SUBTREE AND
      DETERMINE IF THE TREE REQUIRES BALANCING.
      BEGIN IF P+.LPT EQ Z THEN P+.RPT := P+.LPT;
         ELSE P+.LPT := P+.RPT;
         RELEASE(P);
         REBUILD;
         END
      ELSE IF P+.LPT EQ NIL THEN
         "THE NODE TO BE DELETED HAS NO LEFT
         SON; DELETE THE NODE, RELINK ITS ONE
         SUBTREE AND DETERMINE IF THE TREE
         REQUIRES BALANCING.
         BEGIN IF P+.RPT EQ Z THEN P+.LPT := P+.RPT;
            ELSE P+.LPT := P+.RPT;
            RELEASE(P);
            REBUILD;
            END
      ELSE IF P+.LPT EQ Z THEN "THE NODE TO BE DELETED HAS TWO SONS;
         FIND ITS POSTORDER PREDECESSOR OR
         SUCCESSOR;
         BEGIN PRECORSUC(P);
            P+.INFO := BACKP[I].PT+.INFO;
            RELINK;
         END
      END
   END;

END RELINK

BEGIN *AVLDELETE*

*INITIALIZE THE DELETION STACK WITH THE HEADER NODE*

BACKP[1].PT := HEA:
BACKP[1].DP := 1:
P := HEAD+.OPT;
I := 1;

*DETERMINE THE SEARCH SEQUENCE TO THE NODE TO BE DELETED SAVING THE POINTERS AND DIRECTIONS IN THE DELETION STACK*

WHILE P NE NIL DO
BEGIN
   DO1 = DO1 + 1:
   I := I + 1;
   BACKP[I].PT := P;
   IF P+.INFO NE WORD
   THEN
      IF P+.INFO GT WORD
      THEN
         BACKP[I].OR := -1;
         P := P+.LPT;
      ELSE
         BACKP[I].OR := 1;
         P := P+.RPT;
      END
   ELSE
      *THE NODE IS PRESENT, DELETE THE NODE AND DETERMINE IF REBALANCING IS NECESSARY*

      BEGIN
         NODECOUNT := NODECOUNT - 1;
         IF NODECOUNT NE 0
         THEN RELINK
         ELSE
            BEGIN
               HEAD+.ROT := NIL;
               RELEASE(P);
            END
         GOTO 10;
      END
   END

10: END; *AVLDELETE*

BEGIN *AVL DRIVER*

*PERFORM NECESSARY INITIALIZATIONS*

CREATE HEADER;

*INSERT KEYS INTO THE TREE*

WHILE READWORD(INWORD) DO
   AVLINSERT(INWORD);

*DELETE KEYS FROM THE TREE*

WHILE READWORD(INWORD) DO
   AVLDELETE(INWORD);

999: END; *AVL DRIVER*
GLOBAL CONSTANTS, TYPES, VARIABLES

SIZE -- AN ADDITIONAL FIELD CONTAINED IN ANY RECORD REPRESENTING A
    NODE AND GIVING THE SIZE OF THE SUBTREE DEFINED BY THAT NODE

NUMER, DENOM, DIFF, PROP1, PROP2, PROP3 -- USED IN INEQUALITIES INVOLVING
    ALFA

LABEL
    399:
    MAX = 10;
    STOPCHAR = E;

TYPE
    POINT = *TREE;
    DIRECTION = (RIGHT, LEFT);
    NODE = RECORD
        INFO: ALFA;
        LEFT: POINT;
        RIGHT: POINT;
        SIZE: INTEGER;
    END;

VAR
    TREE: CLASS 2.1 OF NODE;
    HEAD, FREE, CURRENT, FATHER: POINT;
    MAXDIRECTION;
    INFO: ALFA;
    CHARTAB[1..MAX] OF CHAR;
    NODECOUNT: INTEGER;
    NUMER, DENOM, DIFF, PROP1, PROP2, PROP3: INTEGER;
PROCEDURE CREATEHEADER;
\*PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS
\*VAP
\*ALFA--THE PARAMETER TO THE ALGORITHM
\*ALFAPEAL;
BEGIN CREATEHEADER
\*DETERMINE ALFA
READ(NUMER);
READ(DENOM);
ALFA := NUMER/DENOM;
\*CHECK IF ALFA IS IN RANGE
IF ALFA GT (1.) - SORT(2.)/2.0 THEN
BEGIN
WRITE(= ALFA EXCEEDS 1 - SORT(2)/2. STOP,EOL);
GOTO EXIT 999;
END:
\*INITIALIZE NODE COUNT, FREE LIST POINTER AND HEADER NODE
NODECOUNT := 0;
FREE := NIL;
IF ACQUIRE(HEAD) THEN GOTO EXIT 999;
HEAD++,ROT := NIL;
\*DETERMINE QUANTITIES INVOLVING ALFA IN TERMS OF INTEGERS
DIFF := DENOM - NUMER;
PROD1 := DIFF*NUMER;
PROD2 := DENOM*(DENOM - NUMER - NUMER);
PROD3 := DENOM+DIFF;
END; CREATEHEADER

------------------------------------------------------------------------

PROCEDURE MAKENODE(WORD, ALFA, PI, POINT);
\*PURPOSE: TO DEFINE THE FIELDS OF A RECORD REPRESENTING A NODE
\*PARAMETERS: INPUT:
\*PI--POINTER TO THE NEW NODE
\*POINT--KEY OF THE NEW NODE
\*WORD--KEY OF THE NEW NODE
BEGIN MEKENODE
NODECOUNT := NODECOUNT + 1;
PI.INFO := WORD;
PI.SIZE := 1;
PI.ROT := NIL;
PI.LPT := NIL;
END; MEKENODE
PROCEDURE RELINK (FATHER, OLDROOT, NEWROOT, POINT);  
PURPOSE: TO RELINK A ROTATED SUBTREE TO THE TREE

PARAMETERS INPUT:
FATHER --- POINTER TO THE FATHER OF THE ROOT OF THE SUBTREE TO BE ROTATED
NEWROOT --- POINTER TO THE NEW ROOT OF THE ROTATED SUBTREE
OLDROOT --- POINTER TO THE OLD ROOT OF THE ROTATED SUBTREE

BEGIN RELINK
   IF FATHER.*PPT EQ OLDROOT
      THEN FATHER.*RPT := NEWROOT;
   ELSE FATHER.*LPT := NEWROOT;
   END; RELINK

PROCEDURE SRotate (PP, P, Q, POINT; BALVAL, DIRECTION);
PURPOSE: TO PERFORM A SINGLE ROTATION

PARAMETERS INPUT:
BALVAL --- THE DIRECTION OF THE ROTATION
P, Q --- POINTERS TO THE RELEVANT NODES IN THE ROTATION
PP --- FATHER OF THE ROOT OF THE SUBTREE TO BE ROTATED

VAR
   *PSIZE --- THE SIZE OF THE ROTATED SUBTREE
   S --- AUXILIARY POINTER VARIABLE

   PSIZE : INTEGER;
   S : POINT;

BEGIN SRotate
   ★P12 := A2 + 1;★
   ★PERFORM THE APPROPRIATE ROTATION★

   IF BALVAL EQ RIGHT
      THEN BEGIN ★
         S := Q.*LPT;
         P.*RPT := S;
         Q.*LPT := P;
      END;
      ELSE BEGIN ★
         S := Q.*RPT;
         P.*LPT := S;
         Q.*RPT := P;
      END;
      PSIZE := P.*SIZE;
   ★ADJUST SIZE VALUES★
   IF S EQ NIL
      THEN P.*SIZE := P.*SIZE - Q.*SIZE
      ELSE P.*SIZE := P.*SIZE - Q.*SIZE + S.*SIZE;
   Q.*SIZE := PSIZE;
   ★THE NODE OF THE ROTATED SUBTREE BECOMES THE NEXT NODE TO EXAMINE★
   CURRENT := Q;
   RELINK (PP, P, Q);
END; SRotate★
PROCEDURE DOROTATE(PP, P, Q, POINT; RVAL, VAL; DIRECTION);

PURPOSE: TO PERFORM A DOUBLE ROTATION

PARAMETERS INPUT:
  RVAL -- THE DIRECTION OF THE LOCATION
  P, Q -- POINTERS TO THE RELEVANT NODES IN THE LOCATION
  PP -- FATHER OF THE ROOT OF THE SUBTREE TO BE ROTATED

VAR

+PSIZE -- THE SIZE OF THE ROTATED SUBTREE
S -- AUXILIARY POINTER VARIABLE

PSIZE: INTEGER;
S: POINT;

BEGIN DOROTATE;
  I3 := I3 + 1;
  PSIZE := P+.SIZE;
  PERFORM THE APPROPRIATE ROTATION;

IF RVAL EQ RIGHT THEN BEGIN

  S := Q+.LPT;
  *ADJUST SIZE VALUES"
  IF S+.LPT EQ NIL THEN P+.SIZE := P+.SIZE + Q+.SIZE
  ELSE P+.SIZE := P+.SIZE + S+.LPT+.SIZE;
  IF S+.RPT EQ NIL THEN Q+.SIZE := S+.SIZE
  ELSE Q+.SIZE := Q+.SIZE + S+.SIZE + S+.RPT+.SIZE;
  *REBALANCE"
  S+.LPT := S+.RPT;
  S+.RPT := Q;
  PP+.RPT := S+.LPT;
  S+.LPT := PP;
END

ELSE BEGIN

  S := Q+.RPT;
  *ADJUST SIZE VALUES"
  IF S+.RPT EQ NIL THEN P+.SIZE := P+.SIZE - Q+.SIZE
  ELSE P+.SIZE := P+.SIZE - Q+.SIZE + S+.RPT+.SIZE;

END


IF S->LPT = NIL
 THEN Q->SIZE := Q->SIZE - S->SIZE
 ELSE Q->SIZE := Q->SIZE - S->SIZE + S->LPT->SIZE;

REBALANCE
Q->RPT := S->LPT;
S->LPT := Q;
P->LPT := S->OPT;
S->RPT := P;
END;
S->SIZE := P->SIZE;

THE NODE OF THE ROTATED SUBTREE BECOMES THE NEXT NODE TO EXAMINE;
CURRENT := S;
RELINK(P,P,P,S);
END: "DOPOTATE"

---

PROCEDURE DOPOTATEKEY(WORD:ALFA);
PURPOSE: TO DECREASE THE SIZE FIELD OF THE NODES IN THE SEARCH
PATH TO THE NEW KEY WHICH IS ALREADY PRESENT IN THE TREE
PARAMETERS: INPUT:
WORD--THE NEW KEY
BEGIN "DOPOTATEKEY"
CURRENT := HEAD->RPT;
UNTIL KEY IS LOCATED, REDUCE SIZE FIELDS IN THE SEARCH PATH:
WHILE CURRENT->INFO NE WORD DO
BEGIN
CURRENT->SIZE := CURRENT->SIZE - 1;
IF WORD LT CURRENT->INFO
 THEN CURRENT := CURRENT->LPT
 ELSE CURRENT := CURRENT->RPT;
END;
END: "DOPOTATEKEY"
PROCEDURE JUSTINSERT(WORD: ALFA):

PURPOSE: TO INSERT A KEY INTO A TREE AS IN THE BASIC TREE CONSTRUCTION
ALGORITHM

PARAMETERS INPUT:
WORD—THE KEY TO BE INSERTED

BEGIN  JUSTINSERT

REPEAT

*CHECK FOR KEY ALREADY PRESENT IN THE TREE*

IF CURRENT+.INFO EQ WORD
THEN
BEGIN
    DUPLICATE KEY(WORD);
    GOTO 10;
END;

CURRENT+.SIZE := CURRENT+.SIZE + 1;
FATHER := CURRENT;

IF WORD GT CURRENT+.INFO
THEN CURRENT := CURRENT+.PRT
ELSE CURRENT := CURRENT+.LPT;
UNTIL CURRENT EQ NIL;

*INSERT NEW KEY INTO THE TREE*

IF ACQUIRE(CURRENT) THEN GOTO EXIT 399;
MAKEONE(WORD, CURRENT);
IF WORD LT FATHER+.INFO
THEN FATHER+.LPT := CURRENT
ELSE FATHER+.RPT := CURRENT;

10: END  JUSTINSERT
PROCEDURE FINDBALANCE(VAP BALVAL, TREESIZE; INTEGER; WORD: ALFA; 
BALVAL: DIRECTION);

PURPOSE: TO COMPUTE THE BALANCE OF THE (LEFT) RIGHT SUBTREE OF THE 
NODE POINTED TO BY CURRENT AFTER THE INSERTION OF THE KEY

PARAMETERS INPUT:
BALVAL---INDICATES WHICH SUBTREE (LEFT OR RIGHT) FOR WHICH THE 
BALANCE IS SOUGHT 
WORD------KEY TO BE INSERTED

PARAMETERS OUTPUT:
BALVAL----THE BALANCE OF THE (LEFT) RIGHT SON OF CURRENT AFTER 
THE INSERTION OF THE KEY 
TREESIZE--THE SIZE OF THE (LEFT) RIGHT SUBTREE OF CURRENT AFTER 
INSERTION

VAR
  LEFTSIZE--THE SIZE OF THE LEFT SUBTREE OF THE LEFT OR RIGHT 
  SUBTREE OF CURRENT

  LEFTSIZE: INTEGER;

BEGIN FINDBALANCE
BEGIN IF BALVAL EQ RIGHT 
THEN
BEGIN 
  COMPUTE BALANCE OF RIGHT SUBTREE OF CURRENT AFTER 
  INSERTION
  TREESIZE := CURRENT+.RPT+.SIZE + 2;
  IF WORD LT CURRENT+.RPT+.INFO 
  THEN 
    IF CURRENT+.RPT+.LPT NE NIL 
    THEN 
      LEFTSIZE := CURRENT+.RPT+.LPT+.SIZE + 
    ELSE 
      LEFTSIZE := 2
  ELSE 
  BEGIN 
    IF WORD EQ CURRENT+.RPT+.INFO 
    THEN 
      TREESIZE := TREESIZE - 1;
    IF CURRENT+.RPT+.LPT NE NIL 
    THEN 
      LEFTSIZE := CURRENT+.RPT+.LPT+.SIZE + 
    ELSE 
      LEFTSIZE := 1;
  END; 
END;
BALFAC := LEFTSIZE*PROD3;
END
ELSE
BEGIN
*COMPUTE BALANCE OF LEFT SUBTREE OF CURRENT AFTER INSERTION*
TREESIZE := CURRENT++.LPT++.SIZE + 2;
IF WORD LT CURRENT++.LPT++.INFO
THEN IF CURRENT++.LPT++.LPT NE NIL
THEN LEFTSIZE := CURRENT++.LPT++.LPT++.SIZE + 2
ELSE LEFTSIZE := 2
ELSE BEGIN
IF WORD EQ CURRENT++.LPT++.INFO
THEN TREESIZE := TREESIZE - 1;
IF CURRENT++.LPT++.LPT NE NIL
THEN LEFTSIZE := CURRENT++.LPT++.LPT++.SIZE + 1
ELSE LEFTSIZE := 1;
END;
BALFAC := (TREESIZE - LEFTSIZE)*PROD3;
END;
*FINDBALANCE*
PROCEDURE LEFTINSERT(WORD, ALFA);

PURPOSE: TO DETERMINE IF THE INSERTION OF A KEY INTO THE LEFT SUBTREE OF A NODE POINTED TO BY CURRENT WILL CAUSE CURRENT TO BE UNBALANCED; IF SO A ROTATION IS PERFORMED TO RESTORE BALANCE.

PARAMETERS INPUT:
WORD—THE KEY TO BE INSERTED

VAR
BALFAC—-THE BALANCE OF THE NODE UNDER CONSIDERATION AFTER
THE INSERTION OF THE NEW KEY
LEFTSIZE—-THE SIZE OF THE LEFT SUBTREE OF THE CURRENT NODE
UNDER EXAMINATION AFTER THE INSERTION
TREESIZE—-THE SIZE OF THE SUBTREE OF THE CURRENT NODE UNDER
EXAMINATION AFTER THE INSERTION

BALFAC, LEFTSIZE, TREESIZE: INTEGER;

BEGIN
LEFTSIZE := 2;

*COMPUTE THE NEW BALANCE OF CURRENT AFTER INSERTION*

IF CURRENT+.LOT NE NIL THEN
LEFTSIZE := CURRENT+.LOT+.SIZE + 2;

TREESIZE := CURRENT+.SIZE + 2;
BALFAC := LEFTSIZE - TREESIZE;

IF (BALFAC GE TREESIZE+NUMP) AND (BALFAC LE TREESIZE+DIFF) THEN

*NO REBALANCE IS NECESSARY; INCREMENT THE SUBTREE SIZE AND
FIND THE NEXT NODE IN THE SEARCH PATH*

BEGIN
CURRENT+.SIZE := CURRENT+.SIZE + 1;
FATHER := CURRENT;
CURRENT := CURRENT+.LPT;
END

*REBALANCE*

ELSE IF CURRENT+.SIZE EQ 2 THEN SROTATE(FATHER, CURRENT, CURRENT+.LPT, LEFT)
ELSE
BEGIN
FINDBALANCE(BALFAC, TREESIZE, WORD, LEFT);
IF BALFAC LT PRED2*TREESIZE THEN SROTATE(FATHER, CURRENT, CURRENT+.LPT, LEFT)
ELSE DROTATE(FATHER, CURRENT, CURRENT+.LPT, LEFT);
END;

END: LEFTINSERT
PROCEDURE RIGHTINSERT(WORD:ALFA);*

PURPOSE: TO DETERMINE IF THE INSERTION OF A KEY INTO THE RIGHT SUBTREE OF A NODE POINTED TO BY CURRENT WILL CAUSE CURRENT TO BE UNBALANCED; IF SO A ROTATION IS PERFORMED TO RESTORE BALANCE.

PARAMETERS INPUT:
WORD--THE KEY TO BE INSERTED

VAR
BALFA--THE BALANCE OF THE NODE UNDER CONSIDERATION AFTER THE INSERTION OF THE NEW KEY
LEFTSIZE--THE SIZE OF THE LEFT SUBTREE OF THE CURRENT NODE UNDER EXAMINATION AFTER THE INSERTION
TREESIZE--THE SIZE OF THE SUBTREE OF THE CURRENT NODE UNDER EXAMINATION AFTER THE INSERTION
BALFA,LEFTSIZE,TREESIZE:INTEGER;

BEGIN /*RIGHTINSERT*/
  LEFTSIZE := 1;
  /*COMPUTE THE NEW BALANCE OF CURRENT AFTER INSERTION*/
  IF CURRENT*LPT NE NIL
     THEN LEFTSIZE := CURRENT*LPT*SIZE + 1;
  TREESIZE := CURRENT*SIZE + 2;
  BALFA := LEFTSIZE*DENOM;
  IF (BALFA GE TREESIZE*NUMER) AND (BALFA LE TREESIZE*DIFF)
     THEN /*NO REBALANCE IS NECESSARY! INCREMENT THE SUBTREE SIZE AND FIND THE NEXT NODE IN THE SEARCH PATH*/
     BEGIN
       CURRENT*SIZE := CURRENT*SIZE + 1;
       FATHER := CURRENT;
       CURRENT := CURRENT*RPT;
     END
  ELSE /*REBALANCE*/
    ELSE IF CURRENT*SIZE EQ 2
      THEN SROTATE(FATHER,CURRENT,CURRENT*RPT,RIGHT)
      ELSE
        BEGIN
          FINDBALANCE(BALFA,TREESIZE,WORD,RIGHT);
          IF BALFA LT PROD2*TREESIZE
            THEN SROTATE(FATHER,CURRENT,CURRENT*RPT,RIGHT)
            ELSE DROTATE(FATHER,CURRENT,CURRENT*RPT,RIGHT);
        END
  END /*RIGHTINSERT*/
PROCEDURE BINSERT(WORD:ALFA);  
PURPOSE: TO INSERT A KEY INTO THE TREE  
PARAMETERS INPUT:  
WORD--THE KEY TO BE INSERTED  
VAR  
   PIPOINTER;  
BEGIN  
   FATHER := HEAD;  
   CURRENT := HEAD\$OPT;  
   IF CURRENT EQ NIL  
    THEN  
      "ENTER FIRST KEY"  
      BEGIN  
      IF ACQUIRE(CURRENT) THEN GOTO EXIT 999;  
      MAKENODE(WORD,CURRENT);  
      HEAD\$OPT := CURRENT;  
      END  
    ELSE  
     BEGIN  
     REPEAT  
     IF CURRENT\$INFO NE WORD  
     THEN  
     IF NUMER(CURRENT\$SIZE + 2) LE DENOM  
     THEN  
      "THE SIZE OF THE TREE IS SMALL ENOUGH SO THAT THE  
      KEY MAY BE SIMPLY INSERTED"  
      BEGIN  
      JUSTINSERT(WORD);  
      GOTO 1;  
      END  
     ELSE  
     IF WORD GT CURRENT\$INFO  
     THEN  
      "CHECK FOR CONDITION OF INFINITE LOOP"  
      BEGIN  
      IF CURRENT\$SIZE EQ 2  
      THEN  
      IF CURRENT\$OPT NE NIL  
      THEN  
      IF CURRENT\$OPT\$INFO GT WORD  
      THEN  
      END  
     ELSE  
     IF WORD LT CURRENT\$INFO  
     THEN  
      "CHECK FOR CONDITION OF INFINITE LOOP"  
      BEGIN  
      IF CURRENT\$SIZE EQ 2  
      THEN  
      IF CURRENT\$OPT NE NIL  
      THEN  
      IF CURRENT\$OPT\$INFO LT WORD  
      THEN  
      END  
    END  
  END  
END

BEGIN
IF ACQUIRE(p) THEN GOTO EXIT 999
MAKENODE(WORD,p);
IF FATHER*.RPT = CURRENT
THEN FATHER*.RPT = p
ELSE FATHER*.LPT = p;
P*.LPT = CURRENT;
P*.RPT = CURRENT*.RPT;
CURRENT*.RPT = NIL;
P*.SIZE = 3;
CURRENT+.SIZE = 1;
GOTO 10;
END;

*KEY TO BE INSERTED INTO RIGHT SUBTREE*
RIGHTINSERT(WORD);
END
ELSE

*KEY TO BE INSERTED INTO THE LEFT SUBTREE*
LEFTINSERT(WORD);
ELSE

*KEY ALREADY PRESENT IN THE TREE*
BEGIN
DUPLICATEKEY(WORD);
GOTO 11;
END
UNTIL CURRENT EQ 'NIL';

*HAVE ENCOUNTERED A NIL SUBTREE, JUST ADD THE NEW NODE*
IF ACQUIRE(CURRENT) THEN GOTO EXIT 999:
MAKENODE(WORD,CURRENT);
IF WORD LT FATHER*.INFO
THEN FATHER*.LPT = CURRENT
ELSE FATHER*.RPT = CURRENT;
END

10:END

*LEFTINSERT*
PROCEDURE NOSUCHNODE(WORD: ALFA):

PURPOSE: TO INCREASE THE SIZE FIELDS OF NODES WHICH HAD BEEN
DECREASED PRIOR TO THE DISCOVERY THAT THE KEY WAS NOT
PRESENT.

PARAMETERS INPUT:
WORD -- THE KEY OF THE NODE TO BE DELETED

BEGIN NOSUCHNODE
    CURRENT := HEAD.PRT;
    WHILE CURRENT NE NIL DO
        BEGIN
            CURRENT.PSIZE := CURRENT.PSIZE + 1;
            IF WORD GT CURRENT.PINFO
            THEN CURRENT := CURRENT.PRT
            ELSE CURRENT := CURRENT.PLT;
        END;
    END:
END NOSUCHNODE

PROCEDURE DELTENODE(P:POINT):

PURPOSE: TO DELETE A NODE AND RELINK ITS SUBTREE (IF ANY)

PARAMETERS INPUT:
P -- POINTER TO THE SUBTREE OF THE NODE TO BE DELETED (IF ANY)

BEGIN DELETENODE
    NODECOUNT := NODECOUNT - 1;
    RELINK(FATHER,CURRENT,P);
    RELEASE(CURRENT);

END DELETENODE
PROCEDURE PREDOPSUCCE WAY, DIRECTION, VAP, WORD, ALFA):

PURPOSE: TO FIND THE POSTORDER PREDECESSOR OR SUCCESSOR OF THE NODE TO BE DELETED AND TO REPLACE ITS KEY WITH THAT OF THE PREDECESSOR OR SUCCESSOR

PARAMETERS INPUT:
Way--Indicates whether the predecessor or successor is sought

PARAMETERS OUTPUT:
Word--The key of the predecessor or successor

VAR:
P, Q, POINT;
BEGIN *PREDOPSUCCE*

FATHER = CURRENT;
CASE WAY OF

RIGHT:

*FIND THE SUCCESSOR*
BEGIN
P := CURRENT+.RPT;
Q := P+.LPT;
WHILE Q NE NIL DO
BEGIN
P := Q;
Q := Q+.LPT;
END;
CURRENT := CURRENT+.RPT;
END:

LEFT:

*FIND THE PREDECESSOR*
BEGIN
P := CURRENT+.LPT;
Q := P+.RPT;
WHILE Q NE NIL DO
BEGIN
P := Q;
Q := Q+.RPT;
END;
CURRENT := CURRENT+.LPT;
END:

*REPLACE THE KEY OF THE NODE TO BE DELETED WITH THE KEY OF THE PREDECESSOR OR SUCCESSOR*
FATHER+.INFO := P+.INFO;
FATHER+.SIZE := FATHER+.SIZE - 1;
WORD := P+.INFO;
END: *PREDOPSUCCE*
PROCEDURE DELERIGHT:

PURPOSE: TO DETERMINE IF DELETION OF A NODE IN THE RIGHT SUBTREE OF THE
NODE UNDER CURRENT EXAMINATION WILL CAUSE UNBALANCE AND IF SO
TO PERFORM A ROTATION TO RESTORE BALANCE.

VARIABLES:
- BALFAC: THE BALANCE OF THE NODE UNDER CURRENT EXAMINATION
- LEFTSIZE: THE SIZE OF THE LEFT SUBTREE OF THE NODE UNDER
  CURRENT EXAMINATION
- P: AUXILIARY POINTERS
- LEFTSIZE, TREESIZE, BALFAC IN: PTR
- P: POINT

BEGIN DELERIGHT

*COMPUTE THE BALANCE OF CURRENT AFTER THE DELETION*

LEFTSIZE := 1;
IF CURRENT.N.LPT NE NIL
  THEN
    LEFTSIZE := CURRENT.N.LPT_SIZE + 1;
    TREESIZE := CURRENT.N.TREE_SIZE;
    BALFAC := LEFTSIZE *2 - TREESIZE;
    IF (BALFAC GE TREESIZE *2 * NUMBER) AND (BALFAC LE TREESIZE *2 DIFF)
      THEN
        \NO REBALANCE IS NECESSARY; DECREMENT THE CURRENT SUBTREE
        SIZE AND FIND THE NEXT NODE IN THE SEARCH PATH*
        BEGIN
          CURRENT.N.TREE_SIZE := CURRENT.N.TREE_SIZE - 1;
          FATHER := CURRENT;
          CURRENT := CURRENT.N.LPT;
        END
      ELSE
        \REBALANCE*
        BEGIN
          P := CURRENT;
          TREESIZE := LEFTSIZE;
          LEFTSIZE := 1;
          \FIND THE BALANCE OF THE LEFT SUBTREE OF CURRENT AND
          DETERMINE THE APPROPRIATE ROTATION*
          IF CURRENT.N.LPT.N.LPT.NE NIL
            THEN
              LEFTSIZE := CURRENT.N.LPT.N.LPT_SIZE + 1;
              BALFAC := (TREESIZE - LEFTSIZE) *2;
              IF BALFAC LT 0003
                THEN
                  SROTATE (FATHER, CURRENT, CURRENT.N.LPT, LEFT)
                ELSE
                  DROTATE (FATHER, CURRENT, CURRENT.N.LPT, LEFT)
              CURRENT.N.TREE_SIZE := CURRENT.N.TREE_SIZE - 1;
              FATHER := CURRENT;
              CURRENT := P;
            END
          END
END: DELERIGHT
PROCEDURE DELETELEFT;

PURPOSE: TO DETERMINE IF DELETION OF A NODE IN THE LEFT SUBTREE OF THE
NODE UNDER CURRENT EXAMINATION WILL CAUSE UNBALANCE AND IF SO
TO PERFORM A ROTATION TO RESTORE BALANCE.

VAR
BALFAC---THE BALANCE OF THE NODE UNDER CURRENT EXAMINATION
LEFTSIZE---THE SIZE OF THE LEFT SUBTREE OF THE NODE UNDER
CURRENT EXAMINATION
P---------AUXILIARY POINTER

LEFTSIZE, TREESIZE, BALFAC, INTEGER;

BEGIN *DELETELEFT*

*COMPUTE THE BALANCE OF CURRENT AFTER THE DELETION*

LEFTSIZE := 1;
IF CURRENT+.LPT NE NIL THEN
LEFISIZE := CURRENT+.LPT+.SIZE;
TREESIZE := CURRENT+.SIZE + 1;
BALFAC := LEFTSIZE-DEVOY;
IF (BALFAC GE TREESIZE*HUP+P) AND (BALFAC LE TREESIZE*DIFF)
OR (TREESIZE EQ 1) THEN

*NO REBALANCE IS NECESSARY; DECREMENT THE CURRENT SUBTREE
SIZE AND FIND THE NEXT NODE IN THE SEARCH PATH*

BEGIN
CURRENT+.SIZE := CURRENT+.SIZE - 1;
FATHER := CURRENT;
CURRENT := CURRENT+.LPT;
END
ELSE

*REBALANCE*

BEGIN
P := CURRENT;
TREESIZE := CURRENT+.PPT+.SIZE + 1;
LEFTSIZE := 1;

*FIND THE BALANCE OF THE RIGHT SUBTREE OF CURRENT AND
DETERMINE THE APPROPRIATE ROTATION*

IF CURRENT+.PPT+.LPT NE NIL THEN

LEFTSIZE := CURRENT+.RPT+.LPT+.SIZE + 1;
BALFAC := LEFTSIZE-DEVOY;
IF BALFAC GE TREESIZE*PPOD1
THEN
SROTATE(FATHER, CURRENT, CURRENT+.RPT, RIGHT)
ELSE
DROTATE(FATHER, CURRENT, CURRENT+.RPT, RIGHT);
CURRENT+.SIZE := CURRENT+.SIZE - 1;
FATHER := CURRENT;
CURRENT := P;
END:

END: *DELETELEFT*
PROCEDURE BDELETE(WORD:ALFA):  
PURPOSE: TO DELETE THE NODE WITH GIVEN INPUT KEY  
PARAMETERS INPUT:  
WORD--THE KEY OF THE NODE TO BE DELETED  
VAR  
NEWWORD:ALFA;  
*NEWWORD--THE KEY OF THE PREDECESSOR OR SUCCESSOR OF THE  
NODE TO BE DELETED  
P--IF THE NODE TO BE DELETED IS A SEMI-LEAF, P POINTS  
TO ITS SON*  
*POINT:  
BEGIN BDELETE*  
FATHER := HEAD;  
CURRENT := HEAD*RPT;  
*CHECK IF (INITIALLY) THE TREE IS EMPTY OR THE NODE IS NOT PRESENT*  
WHILE CURRENT NE NIL DO  
IF CURRENT*RPT NE CURRENT*INFO THEN  
IF WORD GT CURRENT*INFO THEN DELETEFIGHT  
ELSE DELETENET  
ELSE  
*HAVE LOCATED NODE* CHECK IF NODE IS A LEAF OR SEMI-LEAF*  
IF CURRENT*RPT*SIZE EN 1 THEN  
*NODE IS A LEAF* JUST DELETE*  
BEGIN  
DELETENODE(NIL);  
GOTO 10;  
END  
ELSE  
IF CURRENT*RPT*SIZE EN 2 THEN  
*NODE IS A SEMI-LEAF*  
BEGIN  
IF CURRENT*RPT NE NIL THEN P := CURRENT*RPT  
ELSE P := CURRENT*LRT;  
DELETENODE(P);  
GOTO 10;  
END  
ELSE  
*NODE IS NOT A LEAF OR A SEMI-LEAF* REPLACE THE  
NODE WITH ITS POSTORDER SUCCESSOR OR PREDECESSOR  
DEPENDING WHICH MOST IMPROVES THE BALANCE*
IF CURRENT+.LPT EQ NIL
THEN
  *NODE DOES NOT HAVE A LEFT SON*
  BEGIN
    DELETENODE(CURRENT+.RPT);
    GOTO 10;
  END
ELSE
  IF CURRENT+.RPT EQ NIL
  THEN
    *NODE DOES NOT HAVE A RIGHT SON*
    BEGIN
      DELETENODE(CURRENT+.LPT);
      GOTO 10;
    END
  ELSE
      BEGIN
        *DETERMINE WHETHER PREDECESSOR OR SUCCESSOR IS BEST CHOSEN*
        IF CURRENT+.LPT+.SIZE GT CURRENT+.RPT+.SIZE
        THEN
          PREDSUCCEED(LEFT, NEWWORD);
        ELSE
          SUCCEED(RIGHT, NEWWORD);
          WORD := NEWWORD;
        END;
    END;
  END;
  *NODE IS NOT IN THE TREE*
  NOSUCHNODE(WORD);
10: END; *BADELETE*

BEGIN *99-TREE DRIVER*
  *INITIALIZE*
  CREATETREE;
  INSERT KEYS;
  WHILE READWORD(INWORD) DO
    INSERT(INWORD);
  DELETE KEYS;
  WHILE READWORD(INWORD) DO
    BADELETE(INWORD);
999: END. *99-TREE DRIVER*
**RELL TREE**

*GLOBAL CONSTANTS, TYPES, VARIABLES*

- **K** — THE PARAMETER OF THE ALGORITHM
- **KVAL** — THE VALUE $2^k - 1$
- **SIZE** — AN ADDITIONAL FIELD IN ANY RECORD USED TO CONTAIN THE K-SIZE OF A NODE

```plaintext
LABEL 999;
CONST MAX = 10;
STOPCHAR = 'N';
TYPE POINT = *TREE;
DIRECTION = (RIGHT, LEFT);
NODE = RECORD
  INFO: *ALFA;
  SIZE: INTEGER;
  LPT: POINT;
  RPT: POINT;
END;

VAR TREE: CLASS 250 OF NODE;
FREE, HEAD: POINT;
INWORD: ALFA;
CHAY: ARRAY [1..MAX] OF CHAR;
NODECOUNT, K, KVAL: INTEGER;
```

**PROCEDURE CREATEHEADER;**

**PURPOSE:** TO PERFORM THE NECESSARY INITIALIZATIONS

**VAR**

- I: INTEGER;

```plaintext
BEGIN CREATEHEADER

*INITIALIZE FREE LIST pointer, NODE COUNT AND THE PARAMETER K*

FREE := NIL;
NODECOUNT := 0;
READ(K);
KVAL := 1;
FOR I := 1 TO K DO
  KVAL := KVAL * 2;
KVAL := KVAL - 1;

* CREATE THE HEADER NODE*

IF ACQUIRE(HEAD) THEN GOTO EXIT 999;
HEAD.PRT := NIL;
HEAD.SIZE := 0;
END CREATEHEADER
```
PROCEDURE MAKE_NODE(WORD:ALFA; P:POINT);

PURPOSE: TO CREATE THE FIELDS OF A NODE

PARAMETERS INPUT:
P----POINTER TO THE NODE
WORD--KEY OF THE NODE

BEGIN MAKE_NODE

P^.INFO := WORD;
P^.SIZE := 1;
P^.LPT := NIL;
P^.RPT := NIL;

END; MAKE_NODE

------------------------------------------------------------------------

PROCEDURE NEW_NODE(FATHER:POINT; WORD:ALFA);

PURPOSE: TO CREATE A NEW NODE

PARAMETERS INPUT:
FATHER--FATHER OF THE NODE TO BE CREATED
WORD----THE NEW KEY

VAR

S----POINTER TO THE NEW NODE
S:POINT;

BEGIN NEW_NODE

CREATE THE NEW NODE

IF ACQUIRE(S) THEN GOTO EXIT 999;
MAKE_NODE(WORD, S);
CASE WAY OF
RIGHT: FATHER^.RPT := S;
LEFT: FATHER^.LPT := S;
END;

INCREMENT THE K-SIZE VALUE OF THE FATHER

IF FATHER NE HEAD THEN

IF FATHER^.SIZE NE KVAL

THEN FATHER^.SIZE := FATHER^.SIZE + 1;

END; NEW_NODE

------------------------------------------------------------------------
PROCEDURE RELINK(FATHER, OLDROOT, NEWRoot, POINT);

PURPOSE: TO RELINK A ROTATED SUBTREE

PARAMETERS INPUT:
FATHER—-THE POINTER TO THE FATHER OF THE ROTATED SUBTREE
OLDROOT—-THE POINTER TO THE OLD ROOT OF THE SUBTREE
NEWRoot—-THE POINTER TO THE NEW ROOT OF THE SUBTREE

BEGIN RELINK

IF FATHER+. PPT EQ OLDROOT
  THEN FATHER+. PPT := NEWRoot
  ELSE FATHER+. PPT := NEWRoot
END; RELINK

PROCEDURE SPOTATE(FATHER, P, Q, S, POINT; BAlVAL, DIRECTION);

PURPOSE: TO PERFORM A SINGLE ROTATION

PARAMETERS INPUT:
BALVAL—-INDICATES THE DIRECTION OF THE ROTATION, LEFT OR RIGHT
FATHER—-POINTER TO THE FATHER OF THE SUBTREE TO BE ROTATED
P, Q, S—-POINTERS TO RELEVANT NODES IN THE ROTATION

BEGIN SPOTATE

*12 := 12 + 1;*
*PERFORM THE APPROPRIATE ROTATION*
IF BALVAL EQ LEFT
  THEN BEGIN
    P+. LPT := NIL;
    Q+. RPT := P;
    Q+. LPT := S
  END;
ELSE BEGIN
  P+. RPT := NIL;
  Q+. LPT := P;
  Q+. RPT := S
  END;
*ADJUST THE K-SIZE VALUES*
Q+. SIZE := 3;
S+. SIZE := 1;
P+. SIZE := 1;
*RELINK THE ROTATED SUBTREE*
RELINK(FATHER, P, Q);
END; SPOTATE
PROCEDURE DROTATE(FATHER,P,Q,S,POINT,BALVAL,DIRECTION);

PURPOSE: TO PERFORM A DOUBLE ROTATION

PARAMETERS INPUT:
BALVAL--INDICATES THE DIRECTION OF THE ROTATION, LEFT OR RIGHT
FATHER--POINTER TO THE FATHER OF THE SUBTREE TO BE ROTATED,
P,Q,S--POINTER TO RELEVANT NODES IN THE ROTATION

BEGIN #DROTATE#

#I3 := I3 + 1;#
#PERFORM THE APPROPRIATE ROTATION#

IF BALVAL EQ RIGHT
THEN
BEGIN
   S+.RPT := Q;
   P+.RPT := NIL;
   S+.LPT := P
END
ELSE
BEGIN
   S+.LPT := Q;
   P+.LPT := NIL;
   S+.RPT := P
END;
#ADJUST THE K-SIZE VALUES#
S+.SIZE := 3;
P+.SIZE := 1;
Q+.SIZE := 1;
#RELINK THE ROTATED SUBTREE#
PELINK(FATHER,P,S);
END: #DROTATE#


PROCEDURE NEXTNODE(P:POINT;WORD:ALFA;VAR Q:POINT);

PURPOSE: TO FIND THE NEXT NODE IN THE SEARCH PATH SUCCEEDING A GIVEN
INPUT NODE

PARAMETERS INPUT:
P--POINTER TO LAST KNOWN NODE IN THE SEARCH PATH
WORD--KEY TO BE INSERTED

PARAMETERS OUTPUT:
Q--POINTER TO THE NEXT NODE IN THE SEARCH PATH

BEGIN #NEXTNODE#

IF WORD GT P+.INFO
THEN
BEGIN
   Q := P+.RPT;
   WAY := RIGHT;
END
ELSE
BEGIN
   Q := P+.LPT;
   WAY := LEFT
END:
END: #NEXTNODE#
PROCEDURE ROTATE(FATHER,P,Q;POINT;WORD;ALFA);

PURPOSE: TO INITIATE ONE OF THE GENERAL TREE TRANSFORMATIONS

PARAMETERS INPUT:
FATHER--POINTER TO THE FATHER OF THE ROOT OF THE SUBTREE TO BE
   ROTATED
P,Q----POINTER TO RELEVANT NODES USED IN THE ROTATION
WORD----KEY TO BE INSERTED

VAR
S-- POINTER TO THE NEW NODE

BEGIN #ROTATE#
   CREATE THE NEW NODE
   NEXTNODE(?,WORD,S);
   IF ACQUIRE(S) THEN GOTO EXIT.999;
   MAKENODE(WORD,S);
   #DETERMINE THE APPROPRIATE ROTATION#
   IF P?.L = NULL
      THEN
      CASE WAY OF
         RIGHT: ROTATE(FATHER,P,Q,S,RIGHT);
         LEFT: ROTATE(FATHER,P,Q,S,LEFT);
      END
   ELSE
      CASE WAY OF
         RIGHT: ROTATE(FATHER,P,Q,S,LEFT);
         LEFT: ROTATE(FATHER,P,Q,S,LEFT);
      END
   END; #ROTATE#

PROCEDURE CHECKSIZE(PSIZE:INTEGER;VAR SUB:INTEGER);

PURPOSE: TO DETERMINE THE MAXIMUM K-SIZE A SUBTREE MAY HAVE IF ITS ROOT IS THE SON OF AN L-COMPLETE TREE, L<K

PARAMETERS INPUT:
PSIZE--THE K-SIZE OF THE SUBTREE

PARAMETERS OUTPUT:
SUB--THE MAXIMUM K-SIZE THE SUBTREE MAY ATTAIN

BEGIN #CHECKSIZE#
   SUB = KVAL;
   WHILE PSIZE LT SUB DO
      BEGIN
         SUB = SUB DIV 2
      END
   END; #CHECKSIZE#
FUNCTION REPLACE(Q:POINTER;WORD:ALFA;VAR NEWWORD:ALFA):BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO FIND THE POSTORDER PREDECESSOR OR SUCCESSOR OF A NODE AND DETERMINE WHETHER IT NEEDS TO BE DELETED IN ORDER TO MAINTAIN A BELL-TREE UPON INSERTION OF THE NEW KEY. IF THIS IS THE CASE IT RETURNS A VALUE OF TRUE.

PARAMETERS INPUT:
Q——POINTER TO THE SON OF THE NODE FOR WHICH THE PREDECESSOR IS SOUGHT
WORD—KEY OF THE NEW NODE TO BE INSERTED

PARAMETERS OUTPUT:
NEWWORD—THE KEY OF THE PREDECESSOR OR SUCCESSOR

VAR
SAVE,QQ——AUXILIARY POINTER VARIABLES

Q,SAVE:POINTER;

BEGIN REPLACE
REPLACE := TRUE;
QQ := Q;
IF WAY EQ LEFT THEN

FIND THE PREDECESSOR

BEGIN
WHILE Q->RPT NE NIL DO
BEGIN
SAVE := Q;
Q := Q->RPT;
END;
NEWWORD := Q->INFO;
IF WORD GT NEWWORD THEN
REPLACE := FALSE
ELSE

BEGIN
RELEASE(SAVE->RPT);
SAVE->RPT := NIL;
WHILE QQ NE NIL DO
BEGIN
QQ->SIZE := QQ->SIZE - 1;
QQ := QQ->RPT
END
END
ELSE

FIND THE SUCCESSOR
BEGIN
WHILE Q*, LPT NE NIL DO
    BEGIN
        SAVE i = Q;
        Q = Q*, LPT
    END:
NEWWORD i = Q*, INFO;
IF WORD LT NEWWORD
THEN
    REPLACE i = FALSE
ELSE

THE KEY TO BE INSERTED SUCCEEDS THE KEY OF THE
SUCCESSOR, DELETE THE SUCCESSOR, DECREMENT THE
K-SIZE VALUES OF THE NODES IN THE SEARCH PATH
OF THE SUBTREE INVOLVED AND RETURN THE
KEY OF THE SUCCESSOR.

BEGIN
    RELEASE(SAVE*, LPT);
SAVE*, LPT = NIL;
WHILE QQ NE NIL DO
    BEGIN
        QQ*, SIZE = QQ*, SIZE - 1;
        QQ = QQ*, LPT
    END:
END

******************************************

PROCEDURE RECURSE (FATH, CURR* POINT; WORD; ALFA; SUB; INTEGER);

PURPOSE: TO DETERMINE WHAT CHANGES MUST BE MADE IN THE TREE STRUCTURE
IN ORDER TO INSERT A NEW KEY AND MAINTAIN THE BELL-SPÉEE

PARAMETERS INPUT:
CURR*--POINTER TO THE ROOT OF THE SUBTREE IN WHICH THE KEY IS TO
BE INSERTED
FATHER--POINTER TO THE FATHER OF THE ROOT OF THE SUBTREE
SUB--THE SIZE THE SUBTREES OF THE NODE UNDER CURRENT EXAMINATION
      (U SAY) MUST BE IF U IS TO BE L-COMPLETE, L<\in
WORD--THE KEY TO BE INSERTED

VAR
NEWWORD, SAVE--ALFA VARIABLES USED TO CONTAIN KEY VALUES
SON--THE NEXT NODE IN THE SEARCH PATH FOLLOWING
SUBSIZE--THE SIZE THE SUBTREES OF THE NODE UNDER CURRENT
         EXAMINATION (U SAY) MUST BE IF U IS TO BE
         L-COMPLETE, L<\in

SON* POINT;
NEWWORD, SAVE* ALFA;
SUBSIZE* INTEGER;
BEGIN RECURSE

*FINDE THE NEXT NODE IN THE SEARCH PATH*
NEXTNODE (CURR, WORD, SON);
IF SON ME NIL
   THEN
       IF SON+, SIZE EQ SUB
           THEN
               THE TREE IS L-COMPLETE L<K; CHECK IF THE TREE IS I-COMPLETE
               AND IF SO PERFORM A ROTATION.
               IF SUB EQ 1
                   THEN
                       ROTATE(FATH,CURR,SON,WORD)
                   ELSE
                       PERFORM STEP 5. OF THE BIA;
               BEGIN
                   IF REPLACE(SON,WORD,NEWWORD)
                       THEN
                           STEP 5(3) OF THE BIA:
                           BEGIN
                               CHECKSIZE(SON+, SIZE, SUBSIZE);
                               RECURSE(CURR,SON,WORD, SUBSIZE);
                               SAVE 1 = CURR+, INFO;
                               CURR+, INFO 1 = NEWWORD;
                               RECURSE(FATH,CURR,SAVE, SUB);
                           END
                       END
                       ELSE
                           STEP 5(4) OF THE BIA:
                           BEGIN
                               SAVE 1 = CURR+, INFO;
                               CURR+, INFO 1 = WORD;
                               RECURSE(FATH,CURR,SAVE, SUB);
                           END
                   ELSE
                       THE CURRENT SUBTREE IS NOT L-COMPLETE L<K; DETERMINE IF THE
                       NEXT SUBTREE IS COMPLETE:
                       BEGIN
                           CHECKSIZE(SON+, SIZE, SUBSIZE);
                           RECURSE(CURR,SON,WORD, SUBSIZE);
                           CURR+,SIZE 1 = CURR+, SIZE + 1;
                       END
                       ELSE
                           THE SUBTREE IS L-COMPLETE FOR AN L<K; JUST ADD THE NEW NODE:
                           NEWNODE(CURR+,WORD);
                   END
               END
           END
       ELSE
           THE TREE IS L-COMPLETE L<K; CHECK IF THE TREE IS I-COMPLETE
           AND IF SO PERFORM A ROTATION.
       END
FUNCTION SEARCHITEM(Q:POINT;WORD:ALFA):BOOLEAN;
PUPPOSE: BOOLEAN FUNCTION USED TO DETERMINE IF THE GIVEN INPUT KEY IS
PRESENT IN THE TREE. IF THE NODE IS NOT PRESENT, THE FUNCTION
IS RETURNED AS TRUE.
PARAMETERS INPUT:
Q----POINTER TO THE SUBTREE ROOT WHERE THE SEARCH BEGINS
WORD--THE KEY SOUGHT
BEGIN "SEARCHITEM"
    SEARCHITEM := TRUE;
    REPEAT
        I1 := I1 + 1;
        IF Q+.INFO NE WORD THEN
            IF Q+.INFO GT WORD THEN Q := Q+.LPT
                ELSE Q := Q+.RPT
            ELSE BEGIN
                SEARCHITEM := FALSE;
                GOTO 10
            END;
        UNTIL (Q EQ NIL);
10:END; "SEARCHITEM"
PROCEDURE BELLINSERT(WORD: ALFA);

PURPOSE: TO CREATE A NEW NODE WITH GIVEN INPUT KEY

PARAMETERS input:
  WORD--THE KEY OF THE NEW NODE

VAR
  CURRENT--POINTER TO THE CURRENT NODE UNDER EXAMINATION
  FATHER--POINTER TO THE FATHER OF THE CURRENT NODE UNDER
          EXAMINATION
  SONCURR--POINTER TO THE NEXT NODE IN THE SEARCH PATH
  SUBSIZE--THE SIZE OF THE SUBTREES OF THE NODE UNDER
          CONSIDERATION (U SAY) MUST BE IF U IS TO BE L-COMPLET
          L<K

  FATHER, CURRENT, SONCURR: POINTER;
  SUBSIZE: INTEGER;

BEGIN • BELLINSERT •

• INITIALIZE POINTERS •

  FATHER := HEAD;
  CURRENT := HEAD * PPT;
  WAY := RIGHT;
  IF CURRENT EQ NIL
  THEN

  • ENTER THE FIRST NODE IN THE TREE •

  BEGIN
    NEWNODE (HEAD, WORD);
    NODECOUNT := 1;
    *II := 1;
  END

  ELSE
  WHILE CURRENT * INFO NE WORD DO
  BEGIN
    *II := *II + 1;
  END

  • FIND THE FIRST NODE WHICH IS THE ROOT OF A K-INCOMPLETE
    TREE •

  IF CURRENT * SIZE EQ KVAL
  THEN

    BEGIN
      NEXTNODE (CURRENT, WORD, SONCURR);
      FATHER := CURRENT;
      CURRENT := SONCURR;
    END

  ELSE

    • DETERMINE IF THE KEY IS ALREADY PRESENT IN THE TREE •

    BEGIN
      SEARCHITEM (CURRENT, WORD)
      THEN

        • ENTER THE NEW KEY INTO THE TREE •

        BEGIN
          CHECKSIZE (CURRENT, SIZE, SUBSIZE);
          RECURSE (FATHER, CURRENT, WORD, SUBSIZE);
          NODECOUNT := NODECOUNT + 1;
        END
        GOTO 10
      END
      END
    END
  END

10: END • BELLINSERT •
PROCEDURE BELLEDELETE(WORD: ALFA);  

PURPOSE: TO DELETE THE NODE WITH GIVEN INPUT KEY 

PARAMETERS INPUT:
WORD: THE KEY OF THE NODE TO BE DELETED 

NESTED PROCEDURES: CHECKSTACK, FIXSTACK, PREDORSUCC, ONESON, TWOSONS, 
DELETEREDUCE, REPLACENODE, OTHERSIDE, NOSONS, DECIDE 

VAR: 
P: POINTER TO THE NODE TO BE DELETED 
FLAG: BOOLEAN VARIABLE, INDICATING THE DELETION STACK 
NEEDS REASSIGNMENT 
STACK: THE DELETION STACK 
SPT: THE STACK INDEX 
STACKLENGTH: THE LENGTH OF THE STACK FOR A PARTICULAR VALUE 
OF K+ 

STACK: ARRAY[1..40] OF POINT; 
P: POINT; 

STACKLENGTH, SPT: INTEGER; 
FLAG: BOOLEAN; 

+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + 

PROCEDURE CHECKSTACK; 

PURPOSE: TO DETERMINE IF THE LENGTH OF THE DELETION STACK HAS BEEN 
EXCEEDED, IF SO ITS INDEX IS SET TO 1 AND A FLAG IS SET 
TO INDICATE THIS. 

BEGIN *CHECKSTACK*

SPT := SPT + 1; 
IF SPT GT STACKLENGTH 
THEN 
BEGIN 
FLAG := TRUE; 
SPT := 1 
END 

END: *CHECKSTACK*
PROCEDURE FIXSTACK:

PURPOSE: TO ADJUST THE DELETION STACK SO THAT THE MAXIMUM INDEX REFERENCES THE NODE TO BE DELETED

VAR: JJ, JJ INTEGER;
     TEMP POINT;
BEGIN FIXSTACK

   IF FLAG
      THEN

      *THE STACK HAS EXCEEDED ITS LENGTH; ADJUST IT SO THAT THE DELETION SEQUENCE IS IN CORRECT ORDER, THE NODE TO BE DELETED HAVING MAXIMUM INDEX*

      BEGIN
         FOR JJ = SPT TO K DO
            BEGIN
               TEMP := STACK[STACKLENGTH];
               JJ := K;
               REPEAT
                  STACK(J + 1) := STACK(J);
                  JJ := JJ - 1;
               UNTIL JJ = 1;
               STACK(1) := TEMP
            END;
         SPT := STACKLENGTH;
         FLAG := FALSE
      END;
   END; FIXSTACK
PROCEDURE PREDORSUCC(QQ; Q:POINT);

* PURPOSE: TO FIND THE POSTORDER PREDECESSOR OR SUCCESSOR OF A NODE AND
         REBUILD THE DELETION STACK

PARAMETERS INPUT:

QQ---POINTER TO THE SON OF THE NODE FOR WHICH THE PREDECESSOR OR
SUCCESSOR IS SUGHT
QQ---THE SON OF THE NODE POINTED TO BY QQ

BEGIN PREDORSUCC
   IF QQ EQ Q:RPT
      THEN
         * FIND THE PREDECESSOR *
         WHILE QQ NE NIL DO
            BEGIN
            CHECKSTACK;
            STACK[SPT] := QQ;
            QQ := QQ:RPT
            END
         ELSE
            * FIND THE SUCCESSOR *
            WHILE QQ NE NIL DO
               BEGIN
               CHECKSTACK;
               STACK[SPT] := QQ;
               QQ := QQ:LPT
               END
         END
   PROCEDURE ONESON;

* PURPOSE: TO DELETE THE NODE, RELINK ITS ONE SON, AND DECREMENT THE
          K-SIZE VALUES OF NODES IN THE DELETION STACK

VAR

P: POINT;
Q: POINT TO THE ONE SON OF THE NODE TO BE DELETED
Q: POINTER TO THE NODE TO BE DELETED

BEGIN ONESON
   D3 := D3 + 1;
   P := STACK[SPT];
   IF P:LPT EQ NIL
      THEN Q := P:RPT
      ELSE D := P:LPT;
   "DELETE THE NODE AND RELINK ITS ONE SON"
   RELEASE(P);
   IF STACK[SPT - 1]:LPT EQ P
      THEN STACK[SPT - 1]:LPT := Q
      ELSE STACK[SPT - 1]:RPT := Q;
   "DECREMENT THE K-SIZE OF THE NODES IN THE DELETION STACK"
   FOR JJ := SPT - 1 DOWNTO 2 DO
      IF STACK[JJ]:SIZE NE KVAL
         THEN STACK[JJ]:SIZE := STACK[JJ]:SIZE - 1
      ELSE GOTO 10;
10: END; ONESON
PROCEDURE TWOSONS;

PURPOSE: TO REPLACE THE KEY OF THE NODE TO BE DELETED WITH THE KEY OF ITS POSTORDER PREDECESSOR OR SUCCESSOR (STEP 2(B) OF THE BDA)

VAR
  *P--POINTER TO THE NODE TO BE DELETED*
  P,Q,QQ: POINT;
BEGIN *TWOSONS*
  P := STACK[SPT];
  *DETERMINE IF PREDECESSOR OR SUCCESSOR IS BEST CHOSEN*
  IF P.*LPT*.SIZE GT P.*RPT*.SIZE
  THEN BEGIN
    Q := P.*LPT; 
    QQ := Q.*LPT
  END
  ELSE BEGIN
    Q := P.*RPT; 
    QQ := Q.*LPT
  END;
  CHECKSTACK;
  STACK[SPT] := Q;
  PREDORSUCQ(QQ,Q); 
  P.*INFO := STACK[SPT]*.INFO;
END; *TWOSONS*

PROCEDURE DELETEREDUCE(J:INTEGER);

PURPOSE: TO DELETE THE LAST NODE IN THE DELETION STACK AND REDUCE THE K-SIZE VALUES OF THE OTHER NODES IN THE STACK

PARAMETERS INPUT:
  J--THE STACK INDEX INDICATING THE NODE IN THE DELETION STACK TO TERMINATE DECREMENTATION OF K-SIZE VALUES

VAR
  JJ:INTEGER;
BEGIN *DELETEREDUCE*
  *DELETE THE NODE*
  RELEASE(STACK[SPT]):
  IF STACK[SPT - 1].RPT EQ STACK[SPT]
  THEN STACK[SPT - 1].RPT := NIL
  ELSE STACK[SPT - 1].LPT := NIL;
  *REDUCE THE K-SIZE VALUES*
  FOR JJ := SPT - 1 DOWNTO J DO
    STACK[JJ]*.SIZE := STACK[JJ]*.SIZE - 1
END; *DELETEREDUCE*
PROCEDURE REPLECNODE(QO,Q,P,PPOINT);


PARAMETERS INPUT:

P---POINTER TO THE NODE FOR WHICH THE PREDECESSOR (SUCCESSOR) IS SOUGHT.
Q---POINTER TO THE SON OF THE NODE POINTED TO BY P
QQ---POINTER TO THE SON OF THE NODE POINTED TO BY Q

VAR

• SUBSIZE--THE SIZE OF THE SUBTREES OF THE NODE UNDER CONSIDERATION (U SAY) MUST BE IF U IS TO BE L-COMPLET E

• WORD-----AN AUXILIARY ALFA VARIABLE

• WORD\*ALFA;
• SUBSIZE\*INTEGER:

BEGIN *REPLECNODE*

• FIND THE PREDECESSOR OR SUCCESSOR

PREORSUCC(QO,Q);

• SWAP THE KEYS

WORD \* = P\*\*INFO;
P\*\*INFO \* = STACK\{SPT\}*\*INFO;

• INSERT THE KEY INTO THE TREE

CHECKSIZE(P\*\*SIZE, SUBSIZE);

RECURSE(NIL, P\*, WORD, SUBSIZE);

O4 \* = O4 \* + 1;

END; *REPLECNODE*
FUNCTION OTHERSIDE(RSIZE:INTEGER):BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO DETERMINE IF THE SUBTREE OF THE INITIAL K-INCOMPLETE SUBTREE NOT CONTAINING THE NODE TO BE DELETED IS (K-1)-REPLETE. IF SO, THE FUNCTION RETURNS FALSE.

PARAMETERS INPUT:
RSIZE--THE VALUE 2**(K-1) - 1

VAR WAY:DIRECTION;
BEGIN OTHERSIDE+
  "DETERMINE IF THE SUBTREE WE WISH TO EXAMINE IS A LEFT OR RIGHT SUBTREE"
  IF STACK[2]*LPT*,INFO EO STACK[3]*.INFO THEN WAY := RIGHT ELSE WAY := LEFT;

  "DELETE THE NODE AND DECREMENT THE K-SIZE VALUES OF THE NODE IN THE DELETION STACK"
  DELETEREDUCE(2);
  SPT * := 3;
  OTHERSIDE := TRUE;
  CASE WAY OF
  "RIGHT"
  IF STACK[2]*.RPT*,SIZE GT RSIZE THEN
    "THE SUBTREE IS (K-1)-REPLETE"
    BEGIN
      REPLACENODE(STACK[3]*.LPT,STACK[3],STACK[2]);
      OTHERSIDE := FALSE;
    END;
  END;

  "LEFT"
  IF STACK[2]*.LPT*,SIZE GT RSIZE THEN
    "THE SUBTREE IS (K-1)-REPLETE"
    BEGIN
      REPLACENODE(STACK[3]*.RPT,STACK[3],STACK[2]);
      OTHERSIDE := FALSE;
    END;
  END;
END; OTHERSIDE
FUNCTION NOSONS(P:POINT):BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO DELETE A NODE HAVING NO SONS AND MAINTAIN
THE BELL-TREE STRUCTURE. IF THE NODE MAY BE SIMPLY DELETED
WITHOUT RESTRUCTURING THE FUNCTION RETURNS FALSE.

PARAMETERS INPUT:

P--POINTER TO THE NODE TO BE DELETED

VAR

RSIZE,SIZE--INTEGER VARIABLES REPRESENTING THE K-SIZES OF
L-COMPLETE TREES,1=<k
Q,QQ---------AUXILIARY POINTERS
RSIZE,SSIZE,J:INTEGER;
QQ,Q:POINT;

BEGIN NOSONS
D2 := D2 + 1;
NOSONS := TRUE;

END: INITIALIZE TO THE SMALLEST L-SIZE VALUES

BEGIN RSIZE := 1;
SSIZE := 3;
J := SPT - 1;

EXAMINE THE DELETION STACK TO DETERMINE IF A SUBTREE IS
L-COMPLETE FOR SOME L=<k

WHILE STACK(J).SIZE EQ SSIZE DO
BEGIN
J := J - 1;
IF J EQ 1 THEN
ALL THE SUBTREES ARE L-COMPLETE, L=<k; DETERMINE IF
THE OTHER SUBTREE OF THE INITIAL K-INCOMPLETE TREE
IS (K-1)-COMPLETE
BEGIN
IF OTHERSIDE(RSIZE) THEN NOSONS := FALSE;
GOTO 2L
ELSE
ADJUST THE K-SIZE VALUES FOR THE NEXT SUBTREE
BEGIN
RSIZE := SSIZE;
SSIZE := SSIZE + SSIZE + 1
END
END: IF STACK(J).SIZE LT SSIZE THEN
A SUBTREE IS L-INCOMPLETE, L=<k; SIMPLY DELETE THE NODE
BEGIN
DELETEREDUCE(2);
NOSONS := FALSE
END
ELSE

* A SUBTREE IS L-REPLETE, L<K; PERFORM A TRANSFORMATION ON
THE TREE *

BEGIN

IF STACK(J)*.LPT EQ STACK(J + 1)
THEN
BEGIN
Q := STACK(J)*.PPT;
QQ := Q*.LPT
END
ELSE
BEGIN
Q := STACK(J)*.LPT;
QQ := Q*.PPT
END:
DELETEPREDUCE(J):
SPT := J + 1;
STACK[SPT] := 0;
REPLACENODE(QQ, 0, STACK(J))
END:
20 END; NOSONS

PROCEDURE DECIDE:

PURPOSE: TO DETERMINE IF A NODE TO BE DELETED HAS ZERO, ONE OR TWO SONS
AND TO INITIATE THE APPROPRIATE ACTION

BEGIN DECIDE:

* THE NODE TO BE DELETED IS THE LAST ELEMENT IN THE DELETION STACK *

P := STACK[SPT];
FIXSTACK;
IF P*.SIZE EQ 1
THEN

* THE NODE HAS NO SONS *

BEGIN
*IF FLAG1 THEN ZERO := TRUE;
FLAG1 := FALSE;
IF NOSONS(P) THEN DECIDE;
END
ELSE
IF P*.SIZE EQ 2
THEN

* THE NODE HAS ONE SON *

BEGIN
*IF FLAG1 THEN ONE := TRUE;
FLAG1 := FALSE;
ONESON;
END
ELSE

* THE NODE HAS TWO SONS *

BEGIN
*FLAG1 := FALSE;
TWOSONS;
DECIDE;
END;
DECIDE;
END;
BEGIN *BELLODELETE*

*INITIALIZE THE DELETION STACK*

SPT i = 1;
STACKLENGTH i = \* + 1;
FLAG i = FALSE;
STACK[i] i = HEAD;
P i = HEAD.RPT;

*SEARCH FOR THE KEY*

WHILE P ≠ NIL DO
BEGIN
  01 i = 01 + 1;
  CHECKSTACK;
  STACK(SPT) i = P;
  IF P\+.INFO NE WORD THEN
    IF WORD GT P\+.INFO THEN P i = P\+.RPT
    ELSE P i = P\+.LPT
  ELSE
    *THE KEY IS PRESENT, DETERMINE HOW TO DELETE THE NODE*
    BEGIN
      NODECOUNT i = NODECOUNT - 1;
      GOTO 10
    END
  END;
END;

10:  END; *BELLODELETE*

BEGIN *BELL-TREE DRIVER*

*INITIALIZE*

CREATEHEADER;

*INSERT KEYS*

WHILE READWORD(INWORD) DO
  BELLINSERT(INWORD);
  PRINT(HEAD.RPT);

*DELETE KEYS*

WHILE READWORD(INWORD) DO
BEGIN
  BELLODELETE(INWORD);
  PRINT(HEAD.RPT);
END;

999: END; *BELL-TREE DRIVER*
**THE OPTIMAL TREE OF KNUTH**

**GLOBAL CONSTANTS, TYPES, VARIABLES**

ALFAFREQ -- THE WEIGHTS OF THE KEYS; ALFAFREQ[I] GIVING THE WEIGHT OF KEY WD[I]

BETAFREQ -- THE EXTERNAL WEIGHTS; BETAFREQ[I] GIVING THE EXTERNAL WEIGHT BETWEEN KEYS WD[I] AND WD[I+1]

KROOT ----- THE ROOT OF THE TREE

NN ------- THE NUMBER OF KEYS TO STRUCTURED INTO AN OPTIMAL TREE

R ------- THE ARRAY CONTAINING THE SUBTREE ROOTS

WD ------- THE ARRAY CONTAINING THE KEYS

WEIGHT ------- THE ARRAY CONTAINING SUBTREE WEIGHTS; WEIGHT[I,J] GIVING THE SUM OF THE WEIGHTS BETAFREQ[I], ALFAFREQ[I+1], ... , ALFAFREQ[I]

WPL ------- THE ARRAY CONTAINING THE WEIGHTED PATH LENGTH OF THE SUBTREE

LABEL 999;

CONST

MAX = 10;

STOPCHAR = ^E;

NN = 36;

TYPE

POINT = *TREE;

DIRECTION = (RIGHT, LEFT);

NODE = RECORD

INFOALFA;

LPIPOINT;

RPIPOINT

END;

VAR

TREECLASS 200 OF NODE;

KROOT, FREEPOINT;

MAIN, DIRECTION;

INWORD, ALFA;

CHAYARRAY[1..MAX] OF CHAR;

NODECOUNT, INTEGER;

WPLWEIGHT, ARRAY[0..NN, 0..NN] OF INTEGER;

R, ARRAY[0..NN, 1..NN] OF INTEGER;

ALFAFREQ, ARRAY[1..NN] OF INTEGER;

BETAFREQ, ARRAY[0..NN] OF INTEGER;

WD, ARRAY[1..NN] OF ALFA;

PROCEDURE INIT;

PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS

VAR I, J, INTEGER;

BEGIN

* INITIALIZE FREE LIST, POINTER, READ THE KEYS, THE EXTERNAL WEIGHTS,
  AND THE WEIGHTS OF THE KEYS *

FREE 1 = NIL;

I 1 = 1;

WHILE READWORD(INWORD) DO BEGIN

WD[I] = INWORD;

I 1 = I + 1

END;

IF I NE NN + 1 THEN WRITE(^INCORRECT ^I, ^E KEYS READ, ^EOL);

FOR I 1 = 0 TO NN DO BEGIN

READ(J);

BETAFREQ[I] = J;

END;

FOR I 1 = 1 TO NN DO BEGIN

READ(J);

ALFAFREQ[I] = J;

END;
PROCEDURE MAKENODE(WORD: ALFA; P: POINT);

PURPOSE: TO CREATE THE FIELDS OF ANY RECORD REPRESENTING A NODE

PARAMETERS INPUT:
  P --- THE POINTER TO THE NODE
  WORD --- THE KEY OF THE NODE

BEGIN MAKENODE

  NODECOUNT := NODECOUNT + 1;
  P^.INFO := WORD;
  P^.LPT := NIL;
  P^.RPT := NIL

END MAKENODE

PROCEDURE OPTREEKNUTH(INX1, INX2: INTEGER);

PURPOSE: TO CREATE THE OPTIMAL TREE OF KNUTH

PARAMETERS INPUT:
  INX1 --- THE MINIMUM INDEX OF THE ARRAY CONTAINING THE WEIGHTS OF THE KEYS
  INX2 --- THE MAXIMUM INDEX OF THE ARRAY CONTAINING THE WEIGHTS OF THE KEYS

VAR
  D --- THE SIZE OF THE SUBTREE CURRENTLY BEING CONSIDERED
       TO DETERMINE ITS MINIMUM WEIGHTED PATH LENGTH AND THE KEY WHICH WILL BE ITS ROOT
  MIN --- VARIABLE USED IN CALCULATING A TREE OF MINIMUM WEIGHTED PATH LENGTH FOR A SUBTREE OF SIZE D
  N --- THE NUMBER OF KEYS TO BE STRUCTURED INTO AN OPTIMAL TREE

D, I, J, MIN, M, KK, K, N: INTEGER;
PROCEDURE FORMOPT(I,J:INTEGER;FATHER:POINT);

PURPOSE: TO FORM THE OPTIMAL TREE USING THE ARRAY OF SUBTREE ROOTS
       GIVEN BY THE R ARRAY

PARAMETERS INPUT:
   I,J---INDICES TO THE ROOT ARRAY; THE ROOT OF THE SUBTREE WHOSE
       KEYS ARE MD[I], I<LEN, IS TO BE LOCATED
   FATHER---THE POINTER TO THE FATHER OF THE ROOT OF THE NEW SUBTREE

VAR P:POINT;
BEGIN FORMOPT
   IF I NE J
      THEN
       \ CREATE THE NEW NODE AND LINK IT TO THE TREE \n       BEGIN
       IF ACQUIRE(P) THEN GOTO EXIT 999;
       MAKENODE(MD[I,J] + INX1,P);
       IF FATHER NE NIL
           THEN
               CASE WAY OF
                   | RIGHT\FATHER+.RPT := P;
                   | LEFT \FATHER+.LPT := P;
               END
           ELSE KROOT := P;
           END
       \ FORM THE LEFT AND RIGHT SUBTREES 
       WAY := LEFT;
       FORMOPT(I,R[I,J] - 1,P);
       WAY := RIGHT;
       FORMOPT(R[I,J],J,P)
       END
   END; FORMOPT
BEGIN *OPTREEKUTH*

*INITIALIZE THE TOTAL NUMBER OF KEYS*

NODECOUNT := 0;
INX1 := INX1 - 1;
N := INX2 - INX1;

*INITIALIZE AND DETERMINE THE ONE NODE OPTIMUM TREES*

FOR I := 0 TO N DO
BEGIN
    WPL[I,I] := 0;
    WEIGHT[I,I] := BETAFCUR[1 + INX1];
    FOR J := I + 1 TO N DO
                        BETAFCUR[J+INX1]
    END;

FOR J := 1 TO N DO
BEGIN
    WPL[J-1,J] := WEIGHT[I-1,J];
    R[I-1,J] := J
END;

*DETERMINE THE D NODE OPTIMUM TREES*

FOR D := 2 TO N DO
FOR J := 0 TO N DO
BEGIN
I := J - D;

*FIND THE MINIMUM WEIGHTED PATH LENGTH*
K := R[I,J-1];
MIN := WPL[I,K-1] + WPL[K,J];
FOR KK := K+1 TO R[I+1,J] DO
BEGIN
    M := WPL[I,KK-1] + WPL[KK,J];
    IF M LT MIN
THEN
BEGIN
    MIN := M;
    K := KK
END
END;

WPL[I,J] := WEIGHT[I,J] + MIN;
R[I,J] := K
END;

*FORM THE OPTIMUM TREE USING THE R ARRAY*
FORMOPT(0,N,NIL);
END; *OPTREEKUTH*

BEGIN *OPTREEKUTH DRIVER*

*INITIALIZE*
INIT;

*CREATE THE TREE*
OPTREEKUTH(1,NN);
999:END. *OPTREEKUTH DRIVER*
GLOBAL CONSTANTS, TYPES, VARIABLES

MAXDEPTH --- THE MAXIMUM DEPTH OF THE ARRAYS STACK AND WHERE SAVE
N ---------- THE NUMBER OF NODES TO BE STRUCTURED
KEYWORD ---- THE TYPE DEFINING THE VECTOR OF KEYS USED BY THE ALGORITHM
BLOCKSIZE --- THE BLOCKSIZE USED IN CONSTRUCTING THE STARTING TREE
FREQ ------- AN ADDITIONAL FIELD IN ANY RECORD REPRESENTING A NODE
WHT ------- AN ADDITIONAL FIELD IN ANY RECORD REPRESENTING A NODE
DEPTH ------ THE MAXIMUM LEVEL OF ANY NODE IN THE STARTING TREE
ISEED ------ THE RANDOM NUMBER GENERATOR SEED
KEYFREQ ------ THIS ARRAY CONTAINS THE FREQUENCIES OF THE KEYS SUCH THAT KEYFREQ[I] IS THE WEIGHT OF VECTORD[I]
VECTOR ------ THIS ARRAY CONTAINS THE KEYS SUCH THAT THE WEIGHT OF VECTORD[I] = WEIGHT OF VECTOR[I+1]
WLT ------ THE WEIGHT OF THE TREE
WLPNEW ------ THE WEIGHT OF THE OPTIMAL TREE
LABEL 996:
CONST
MAX = 10
STOPCHAR = EIE
N = 2UC
MAXDEPTH = 3

TYPE
POINT = *TRE5;
DIRECTION = (RIGHT, LEFT);
KEYWORD = ARRAY[1...N] OF ALFA;
NONE = RECORD
  INFOALFA:
  FREQ INTEGER;
  WHT INTEGER;
END;

VAR
FREQ: INTEGER 2..1 OF 'IOT';
FREE: HEAD: POINT;
WAY: DIRECTION;
INFOALFA:
KEYFREQ: ARRAY[1...N] OF CHAR;
NODECOUNT: INTEGER;
DEPTH, FLP: PCI;
WLPNEW, BLOCKSIZE INTEGER;
VECTORD: VECTOR;
KEYFREQ: ARRAY[1...N] OF INTEGER;

PROCEDURE CREATEBHEADER;
PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS
BEGIN CREATEBHEADER
  *INITIALIZE FREE LIST POINTERS, NODE COUNTER, RANDOM NUMBER GENERATOR
  SEED AND THE BLOCKSIZE USED IN CREATING THE INITIAL TREE
  FREE := NIL;
  NODECOUNT := 0;
  ISEQI := 98576543;
  BLOCKSIZE := 5;
  *CREATE HEADER NODE
  IF ACQUIRE(HEAD) THEN GOTO EXIT 996;
  HEAD.RPT := NIL;
FUNCTION WEIGHTEDPATHLENGTH(POINT:POINT; VAR DEPTH:INTEGER):INTEGER;

PURPOSE: INTEGER FUNCTION TO CALCULATE THE WEIGHTED PATH LENGTH OF A
TREE AND THE MAXIMUM LEVEL OF ANY NODE IN THE TREE. THE
FUNCTION NAME RETURNS THE WEIGHTED PATH LENGTH.

PARAMETERS INPUT:
    ROOT--POINTER TO THE ROOT OF THE TREE

PARAMETERS OUTPUT:
    DEPTH--THE MAXIMUM LEVEL OF ANY NODE IN THE TREE

NESTED PROCEDURE TRAVERSE

VAR PL:INTEGER;

PROCEDURE TRAVERSE(POINT:POINT; INTEGER):

PURPOSE: TO PERFORM A POSTORDER TRAVERSAL OF THE TREE

PARAMETERS INPUT:
    I--MAXIMUM LEVEL OF ANY NODE ENCOUNTERED SO FAR IN THE TRAVERSAL
    P--THE ROOT OF THE CURRENT SUBTREE

BEGIN TRAVERSE
    IF P NE NIL THEN
        BEGIN
            IF I GT DEPTH
                THEN DEPTH := I;
            TRAVERSE(P, DOT, I + 1);
            PL := PL + P POINT
            TRAVERSE(P, DOT, I + 1);
        END:

END TRAVERSE

BEGIN WEIGHTEDPATHLENGTH
    DEPTH := 0;
    PL := 0;
    TRAVERSE(ROOT, 1);
    WEIGHTEDPATHLENGTH := PL;

END WEIGHTEDPATHLENGTH
PROCEDURE INSERT(KEY; ALFA; KEYFO; INTEGER; PP; POINT);

PURPOSE TO CREATE A BASIC BINARY TREE WHERE THE ORDERING RELATION DEPENDS ON THE PARAMETER PP.

PARAMETERS INPUT:

KEY --- THE KEY OF THE NODE TO BE CREATED
KEYFO --- THE WEIGHT OF THE KEY
PP --- POINTER TO THE ROOT OF THE TREE. THIS DETERMINES WHICH ORDERING RELATION IS USED IN THE CONSTRUCTION OF THE TREE.

VAR
PP; POINT;
BEGIN
*INSERT*
   IF PP = HEAD+ .NOI
     THEN
       *THE ORDERING RELATION IS THE ALPHABETIC RELATIONSHIP BETWEEN THE KEYS. THIS BLOCK IS USED TO CONSTRUCT THE STARTING TREE.+
       BEGIN
         REPEAT
           BEGIN
             PP := PP+;
             *ADJUST THE HEIGHT OF THE SUBTREE DEFINED BY THE NODE POINTED TO BY PP.+
             PP+ .NOI := PP+ .NOI + KEYFO;
             IF PP+ .INFO LT KEY
               THEN PP := PP+ .IPT
               ELSE PP := PP+ .RPT;
           END;
         UNTIL PP = NIL;
         *CREATE THE NEW NODE+
         IF ACQUIRE(PP)
           THEN GOTO EXIT 999
         ELSE
         *DEFINE THE FIELDS OF THE NEW NODE AND LINK IT TO ITS FATHER+
           BEGIN
             NODECOUNT := NODECOUNT + 1;
             PP+ .IPT := NIL;
             PP+ .RPT := NIL;
             PP+ .INFO := KEYFO;
             PP+ .NOI := KEYFO;
           END;
       END;
     END;
   END;
END;
PP+1, WGH T := KE YF Q;  
IF PR+1, INFO, LT KE Y  
THEN PR+1, RPT := PP  
ELSE PR+1, LPT := PP  
END:

END

*THE ORDERING RELATION IS THE USUAL LESS THAN RELATION  
BETWEEN THE WEIGHTS OF THE KEYS; THIS BLOCK IS USED TO  
SORT THE KEYS ON THEIR WEIG HT S*

BEGIN  
REPEAT  
BEGIN
      PP := PP;  
      IF PP+1, FREQ GE KE YF Q  
      THEN PP := PP+1, LPT  
      ELSE PP := PP+1, RPT  
      END;  
UNT IL PP EQ NIL;  
*CREATE THE NEW NODE*  
IF AC Q UIRE (PP)  
THEN GOTO EXIT 399  
ELSE

*DEFINE THE FIELDS OF THE NEW NODE AND LINK  
IT TO ITS FATHER*  
BEGIN
      NODECOUNT := NODECOUNT + 1;  
      PP+1, LPT := NIL;  
      PP+1, RPT := NIL;  
      PP+1, INFO := KE YF Q;  
      PP+1, FREQ := KE YF Q;  
      IF OR+1, FREQ GE KE YF Q  
      THEN PP+1, LPT := PP  
      ELSE PP+1, RPT := PP  
      END

END

END: *INSERT*
PROCEDURE READKEYS(VAR WATE:INTEGER);

PURPOSE: TO READ THE KEYS AND THEIR ASSOCIATED WEIGHTS AND TO RETURN
THE KEYS SORTED IN DESCENDING ORDER ON THEIR WEIGHTS. A
TREESORT IS USED AS THE SORTING METHOD.

PARAMETERS OUTPUT:
WATE--THE SUM OF THE WEIGHTS

VAR
   F0:THE WEIGHT OF A KEY
   ROOT:THE POINTER TO THE ROOT OF THE TREE USED IN THE TREESORT
   P:INTEGER;
   II:INTEGER;
   LPT:POINTER;

PROCEDURE ORDERKEYS(P:POINTER);

PURPOSE: TO TRAVERSE A BINARY TREE IN REVERSE POSTORDER FASHION
RETURNING THE KEY FIELD OF THE I TH NODE VISITED IN THE ARRAY
ELEMENT VECTOR[I] AND ITS WEIGHT IN ARRAY ELEMENT KEYFREQ[I]

PARAMETERS INPUT
   P:POINTER TO THE ROOT OF THE TREE

BEGIN ORDERKEYS
   WHEN P IS NIL THE TRAVERSAL CAN PROCEED NO FURTHER
   IF P NE NIL THEN
      PERFORM A REVERSE POSTORDER TRAVERSAL OF THE TREE. AFTER
      VISITING A NODE RETURN ITS POINTER TO THE FREE LIST
      BEGIN
         ORDERKEYS(P^.PPT);
         VECTOR[I^.INFO] := P^.INFO;
         KEYFREQ[I^.IVEC] := P^.FREQ;
         IVEC := IVEC + 1;
         RELEASE(P);
      END
      ORDERKEYS(P^.LPT)
   END: ORDERKEYS

BEGIN *READYKEYS*

*CREATE A TREE, THE OPERATING RELATION BEING DEFINED AS THE USUAL
LESS THAN RELATION ON THE HEIGHTS OF THE KEYS*

IF READWORD(INWORD)
   THEN
      *CREATE THE ROOT NODE*

      IF ACQUIRE(ROOT)
      THEN GOTO EXIT 999
      ELSE
         BEGIN
            NODECOUNT = 1;
            ROOT.+LFT = NIL;
            ROOT.+RFT = NIL;
            ROOT.+IFO = INWORD;
            READ(FO);
            WAIT = FO;
            ROOT.+FPO = FO;
         END;
      END;
      WHILE READWORD(INWORD) DO
         BEGIN
            READ(FO);
            WAIT = WAIT + FO;
            INSERT(INWORD,FO,ROOT);
         END;
      END;

*SORT THE KEYS ON DESCENDING ORDER OF THEIR HEIGHTS*

IVEC = 1;
OPDSEPKYES(ROOT);
END; *READYKEYS*
PROCEDURE STARTTREE(ALSZ, INTEGER):

PURPOSE: TO CREATE THE STARTING TREE FOR THE BRUNO COFFMAN TREE
CONSTRUCTION ALGORITHM. THE KEYS SORTED ON DESCENDING ORDER
ON THEIR WEIGHTS, ARE GIVEN BY THE ARRAY ELEMENTS VECTOR(I),
THEIR RESPECTIVE WEIGHS BY THE ARRAY ELEMENTS KEYRE(I).

DESCRIPTION: THIS PROCEDURE DIVIDES THE ARRAY, VECTOR, INTO SECTIONS OF
ALSZ IN LENGTH AND SELECTS ELEMENTS AT RANDOM FROM EACH
BLOCK. THE ORDER OF THE ELEMENTS OF VECTOR IS NOT
DISTURBED SO THAT ALSZ CAN BE CHANGED AND NEW RANDOM
PERMUTATIONS FROM "LOCKS OF A DIFFERENT SIZE CAN BE
OBTAINED.

PARAMETERS INPUT

ALSZ--THE BLOCKSIZE TO BE USED

VAR

BOOL Boolean array, used in selecting random elements
FROM EACH BLOCK OF KEYS
INX1, INX2, INX1+1, INX2P1--VARIABLES TO DETERMINE THE CORRECT
ELEMENTS OF THE ARRAY, VECTOR, WHEN
SELECTING A RANDOM ELEMENT FROM
A BLOCK

NUMBLKS--THE NUMBER OF BLOCKS

BOOL ARRAY[1..N] OF BOOLEAN:
NUMBLKS, MANY, IJ, I, J, INX1, INX2, INX1+1, INX2P1, INTEGER;
IJJ: INTEGER;
P: POINT;

BEGIN STARTREE

DETERMINE THE NUMBER OF BLOCKS REQUIRED

NUMBLKS := N DIV ALSZ;
MANY := NUMBLKS*ALSZ;
IF MANY LT N
THEN
MEMBLKS := NUMBLKS + 1;

CHOOSE A KEY AT RANDOM FROM THE FIRST BLOCK AND MAKE IT THE KEY
OF THE ROOT NODE

IJ := TRUNC(RANDU(1,ALSZ/1.0,. . . ,ISEED));
IF ACQUIRE(P) THEN (GOTO EXIT 993);
HEAD+POT := P;
NODECOUNT := 1;
P+LPI := NIL;
P+FIR := KEYE[I][IJ];
P+INF := VECTOR[I][IJ];
P+HIG := KEYE[I][IJ];

INITIALIZE THE BOOLEAN ARRAY USED IN OBTAINING A RANDOM
PERMUTATION OF THE KEYS IN A BLOCK


FOR J := 1 TO HLS7 DO
  BOOL[J] := FALSE;
  BOOL[IJ] := TRUE;

*SELECT KEYS AT RANDOM FROM A BLOCK AND INSERT THEM INTO THE TREE UNTIL THE BLOCK IS EXHAUSTED. REPEAT THIS FOR EACH BLOCK.*

FOR I := 1 TO NUMALGS DO
  BEGIN
    INX2 := I*LSZ;
    INX1 := INX2 - LSZ + 1;
    INX1M1 := INX1 - 1;
    INX2P1 := INX2 + 1;
    IF INX2 GT N THEN INX2 := N;
    FOR IJ := INX1 TO INX2 DO
      BEGIN
        IJ := UNIC(RANDO((INX1/1.0, INX2P1/1.0, ISEED)));
        FOR J := IJ TO INX2 DO
          IF BOOL[J - INX1M1] EQ FALSE THEN
            BEGIN
              INSERT(VECTOR[IJ], KEYFREQ[J], HEAD*RPT);
              GOTO 16;
            END;
        FOR J := INX1 TO IJ - 1 DO
          IF BOOL[J - INX1M1] EQ FALSE THEN
            BEGIN
              INSERT(VECTOR[IJ], KEYFREQ[J], HEAD*RPT);
              GOTO 16;
            END;
      END;
    BOOL[IJ - INX1M1] := TRUE;
  END;

*REINITIALIZE THE BOOLEAN ARRAY FOR THE NEXT BLOCK.*

FOR J := 1 TO HLS7 DO
  BOOL[J] := FALSE;

END: *STARTOFF*
PROCEDURE OPTREE3C(K, WPLOLD:INTEGER;VAR WPLNEW:INTEGER;VECTOR;KEYWORD);

PURPOSE: TO CREATE A NEARLY OPTIMAL TREE USING THE BRURO COFFMAN METHOD

PARAMETERS INPUT:
K----------THE PARAMETER OF THE ALGORITHM
VECTOR--AN ARRAY CONTAINING THE KEYS SORTED IN DESCENDING ON THEIR
WEIGHTS
WPLOLD--THE WEIGHTED PATH LENGTH OF THE STARTING TREE

PARAMETERS OUTPUT:
WPLNEW--THE WEIGHTED PATH LENGTH OF THE NEARLY OPTIMAL TREE

NESTED PROCEDURES: DETERMINE, ROTATIONS, COMPARE

LABEL 1:

VAR
INS---------THE INDEX TO THE ARRAY CONTAINING SUBTREE WEIGHTS
Q-----------POINTER TO THE NODE BEING PROMOTED
OPEQ--------THE WEIGHT OF THE KEY OF THE NODE TO BE PROMOTED
SKLEN-------THIS VARIABLE INDICATES THE NUMBER OF ROTATIONS
NECESSARY TO PROMOTE THE NODE CURRENTLY BEING
EXAMINED
STACK-------THE ARRAY OF POINTERS REPRESENTING THE SEARCH
PATH TO THE NODE TO BE PROMOTED
STACKLENGTH--THE MAXIMUM INDEX OF THE ARRAY STACK
WGHSAVE------THE ARRAY WHICH CONTAINS THE NEW WEIGHTS OF THE
LEFT AND RIGHT SUBTREES OF THE NODE
BEING PROMOTED
WI---------THE WEIGHT OF THE SUBTREE WHOSE ROOT IS THE
NODE BEING PROMOTED
WL---------THE WEIGHT OF THE LEFT SUBTREE WHOSE ROOT IS
THE NODE BEING PROMOTED
WR---------THE WEIGHT OF THE RIGHT SUBTREE WHOSE ROOT IS THE
NODE BEING PROMOTED

WI, WL, WR, SKLEN, I, J, INS, OPEQ, STACKLENGTH: INTEGER;
Q;POINT;
STACKARRAY(I..MAXDEPTH) OF POINT;
WGHSAVE:ARRAY(I..MAXDEPTH) OF INTEGER;

FUNCTION DETERMINE(KEY:ALFA):BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO DETERMINE IF THE CURRENT TRANSFORMATION
CAN BE PERFORMED; IF IT CAN THE FUNCTION RETURNS TRUE

PARAMETERS INPUT:
KEY--THE KEY OF THE NODE TO BE PROMOTED

NESTED PROCEDURE: FIXSTACK

VAR

+FLAG--BOOLEAN VARIABLE INDICATING (TRUE) THE STACK OF POINTERS
TO THE NODE TO BE PROMOTED IS FULL
SPT--INDEX TO THE STACK+

FLAG:BOOLEAN;
SPT:INTEGER;
Q:POINT;
PROCEDURE FIXSTACK:

PURPOSE: To adjust the stack of pointers to the node to be promoted so that the maximum index references the node to be promoted.

VAR
INC, JJ, J: INTEGER;
TENN: POINT;
BEGIN FIXSTACK

IF SPT NE STACKLENGTH THEN

REJUST THE STACK SO THAT THE NODES ARE IN CORRECT SEQUENCE WITH THE MAXIMUM INDEX POINTING TO THE NODE TO BE PROMOTED.

BEGIN
INC := STACKLENGTH - 1;
FOR JJ := SPT TO INC DO
BEGIN
TEMP := STACK[STACKLENGTH];
FOR J := INC DOWNTO 1 DO
STACK[J + 1] := STACK(J);
STACK[J] := TEMP
END
END

END: FIXSTACK

BEGIN DETERMINE

INITIALIZE

DETERMINE := FALSE;
FLAG := FALSE;
SPT := 1;
STACK[1] := HEAD;
P := HEAD.PTR;

CREATE THE STACK

WHILE P.INFO NE KEY DO
BEGIN
SPT := SPT + 1;
STACK[SPT] := P;
IF KEY GT P.INFO
THEN P := P.ESP;
ELSE P := P.ELP;
IF SPT EQ STACKLENGTH
THEN BEGIN
SPT := -1;
FLAG := TRUE
END
END

SPT := SPT + 1;
STACK[SPT] := P;
IF (FLAG) OR (SPT EQ STACKLENGTH) THEN

THE ROTATION CAN BE PERFORMED.

BEGIN
FIXSTACK;
DETERMINE := TRUE
END

END: DETERMINE
PROCEDURE ROTATIONS:

PURPOSE: TO ACTUALLY PERFORM THE TRANSFORMATION ON THE TREE

VAR

J, IJ: INTEGER;

BEGIN rotations

IJ = I;

*PERFORM SKLEIN ROTATION*

FOR J = SKLEIN DOWNTO 1 DO

*PERFORM THE ROTATION AND ADJUST THE WEIGHTS OF THE SUBTREES AFFECTED*

BEGIN

IJ = IJ + 1;

IF STACK(J-1) * RPT = 2, STACK(J) THEN

STACK(J) * RPT := 0;

ELSE STACK(J-1) * LPT := 1;

IF STACK(J) * RPT = 0 THEN

BEGIN

STACK(J) * RPT := 0;

STACK(J) * LPT := STACK(J);

STACK(J) * RPT * WGT := WHSAVE(IJ);

END

ELSE BEGIN

STACK(J) * LPT := STACK(J);

STACK(J) * RPT := STACK(J);

STACK(J) * RPT * WGT := WHSAVE(IJ);

END;

END OF WGT := 0 * FREE;

IF 0 * LPT NE NIL THEN 0 * WGT := 0 * LWT + 0 * LPT * WGT;

IF 0 * RPT NE NIL THEN 0 * WGT := 0 * WGT + 0 * RPT * WGT;

END: rotations
PROCEDURE COMPARE(PLNEW, KK: INTEGER);

PURPOSE: TO DETERMINE IF THE WEIGHTED PATH LENGTH OF THE CURRENT TREE
       WILL BE REDUCED IF A TRANSFORMATION IS PERFORMED (WITHOUT
       ACTUALLY PERFORMING ANY ROTATIONS)

PARAMETERS INPUT:
       KK------INDEX TO THE ARRAY OF POINTERS WHICH GIVES THE CORRECT NODE
               IN THE SEARCH PATH TO THE NODE BEING PROMOTED AND WHICH
               IS USED TO DETECT IN WHICH WAY (LEFT OR RIGHT) A ROTATION
               SHOULD BE PERFORMED
       PLNEW----THE WEIGHTED PATH LENGTH OF THE TREE

VAR
       SK:POINT;

BEGIN COMPARE
       WHEN KK IS 0 THE TRANSFORMATION IS COMPLETE
       IF KK NE 0 THEN
          BEGIN
             SK := STACK[KK];
             INS := INS + 1;
             DETERMINE THE CORRECT DIRECTION OF THE ROTATION
             IF SK*,ROT EQ STACK[KK + 1]
                THEN
                   THE NODE TO BE PROMOTED IS A RIGHT SON OF ITS
                   FATHER. CALCULATE THE WEIGHT OF ITS LEFT
                   SUBTREE, THE NEW WEIGHTED PATH LENGTH AND THE
                   NEW WEIGHT OF THE SUBTREE AFTER THE PROMOTION
                   BEGIN
                      WL := WL + SK*,WGT;
                      IF SK*,LPT NE NULL
                         THEN WL := 2L + SK*,LPT*.WGT;
                      WGHSAVE[INS] := WL;
                      PLNEW := PLNEW - WI - WL;
                      WI := WL + WR + 2FREQ;
                      COMPARE(PLNEW, KK - 1)
                   END
           ELSE
                   THE NODE TO BE PROMOTED IS A LEFT SON OF ITS
                   FATHER. CALCULATE THE WEIGHT OF ITS RIGHT
                   SUBTREE, THE NEW WEIGHTED PATH LENGTH AND THE
                   NEW WEIGHT OF THE SUBTREE AFTER THE PROMOTION
                   BEGIN
                      WR := WR + SK*,WGT;
                      IF SK*,ROT NE NULL
                         THEN WR := 2R + SK*,ROT*.WGT;
                      WGHSAVE[INS] := WR;
                      PLNEW := PLNEW - WI - WR;
                      WI := WL + WR + 2FREQ;
                      COMPARE(PLNEW, KK - 1)
                   END
           END
       END

       DETERMINE IF THE WEIGHTED PATH LENGTH HAS BEEN REDUCED: IF
       SO PERFORM THE ACTUAL ROTATIONS
       BEGIN
          IF WPLOLD GT PLNEW
             THEN
                WPLOLD := PLNEW;
                ROTATIONS:
                WI := 1
             END
       END
BEGIN ATREEPC

*BEGIN THE EXAMINATION OF THE NEIGHBORHOOD OF THE CURRENT TREE*

FOR I := K DOWNTO 1 DO
BEGIN
STACKLENGTH := I + 1:
SKLEN := I:
FOR J := 1 TO I DO:
IF DETERMNETS(Vектор(j)) THEN

/*INITIALIZE FOR THE PROMOTION OF THE NODE POINTED TO BY J*/
BEGIN
  J := STACK(STACKLENGTH):
  OFFPC := J + FPED:
  IF OFFPC.LF := NIL
  THEN WL := OFFPC.RF + WHT
  ELSE WL := J:
  IF OFFPC.RF := NIL
  THEN WR := OFFPC.RF + WHT
  ELSE WR := J:
  WL := OFFPC.RF + WHT:
  IRS := J:

/*DETERMINE IF THE WEIGHTED PATH LENGTH WILL BE REDUCED WITH A TRANSFORMATION*/
COMPARE(WPlOLOD, I):
END:

WPlNEW := WPlOLOD:
END: ATREEPC

BEGIN ABRUNO COFFMAN TREE DRIVER*

*PERFORM NECESSARY INITIALIZATIONS*
CREATEHEADER;
READ IN THE KEYS AND THEIR WEIGHTS AND READY THEM FOR THE STARTING TREE;
READYKEYS(WATE):
CREATE THE STARTING TREE:
STARTREE(BLOCKSIZE):
WPlOLOD := WEIGHTEDPATHLENGTH(HEAD*,RPT,DEPTY):
/*APPLY THE BRUNO COFFMAN ALGORITHM*/
OPTREEPC(DEPTY,WPlOLOD,WPlNEW,VECTOR):

999: END. ABRUNO COFFMAN TREE DRIVER*
GLOBAL CONSTANTS, TYPES, VARIABLES

NN --- THE NUMBER OF NODES STRUCTURED BY THE OPTIMAL TREE OF KNUTH
NUM --- THE NUMBER TO BE STRUCTURED BY THE ALGORITHM

ALFAFREQ --- THE HEIGHTS OF THE KEYS; ALFAFREQ[I] GIVING THE WEIGHT OF
KEY[I]
BETAFREQ --- THE EXTERNAL WEIGHTS; BETAFREQ[I] GIVING THE EXTERNAL WEIGHT
BETWEEN KEYS WD[I] AND WD[I+1]
F -------- VARIABLE USED IN DETERMINING THE NEIGHBORHOOD ABOUT THE
CENTROID
KROOT --- THE ROOT OF A SUBTREE CONSTRUCTED BY KNUTH'S ALGORITHM
N0 -------- THE NUMBER OF NODES TO BE STRUCTURED BY THE OPTIMAL TREE OF
KNUTH
WEIGHT ---- THE ARRAY CONTAINING SUBTREE WEIGHTS USED BY KNUTH'S ALGORITHM;
WEIGHT[I, J] GIVING THE SUM OF THE WEIGHTS BETAFREQ[I],
ALFAFREQ[I+1], ..., ALFAFREQ[J], BETAFREQ[J]
WD -------- THE ARRAY CONTAINING THE KEYS
WGROOT --- THE ROOT OF THE TREE
WPL -------- THE ARRAY CONTAINING THE WEIGHTED PATH LENGTH OF THE SUBTREES
USED IN KNUTH'S ALGORITHM

LABEL 999;

CONST
    MAX = 10;
    STOPCHAR = 'i';
    NUM = 200;
    NN = 31;

TYPE
    POINT = *Tree;
    DIRECTION = (Right, Left);
    NODE = RECORD
        Info: Alfa;
        LPtr: Point;
        RPtr: Point;
    END;

VAR
    Tree: Array 200 of Node;
    WGRoot, KRoot, Free: Point;
    NodeCount, Nd, F: Integer;
    Way: Direction;
    InWord: Alfa;
    Chay: Array[1..MAX] of Char;
    WPl, Weight: Array[0..NN+6..NN] of Integer;
    R: Array[0..NN, 1..NN] of Integer;
    AlfaFreq: Array[1..NN] of Integer;
    BetaFreq: Array[0..NN] of Integer;
    WD: Array[1..NUM] of Alfa;
PROCEDURE INIT;

PURPOSE: TO PERFORM NECESSARY INITIALIZATIONS AND TO READ THE KEYS, THE EXTERNAL WEIGHTS, AND THE WEIGHTS OF THE KEYS

VAR I, J: INTEGER;
BEGIN INIT

*INITIALIZE FREE LIST POINTER, NODE COUNTER AND PARAMETERS TO THE ALGORITHM*
FREE I := NIL;
NODECOUNT I := 0;
READ(NO, F);
I := 1;

*READ THE KEYS*
WHILE READWORD(INWORD) DO
BEGIN
WD[I] := INWORD;
I := I + 1
END;
IF I NE NUM + 1 THEN WRITE(ENOINCORRECT I, I, KEYS READ, EOL);

*READ THE EXTERNAL WEIGHTS*
FOR I := 0 TO NUM DO
BEGIN
READ(J);
BETAFREQ[I] := J;
END;

*READ THE WEIGHTS OF THE KEYS*
FOR I := 1 TO NUM DO
BEGIN
READ(J);
ALFAFREQ[I] := J;
END;
END INIT
PROCEDURE MAKENODE(WORD:ALFA;P:POINT);
PURPOSE: TO CREATE THE FIELDS OF ANY RECORD REPRESENTING A NODE
PARAMETERS INPUT:
    P———POINTER TO THE NODE
    WORD——KEY OF THE NODE
BEGIN »MAKENODE«
    NODECOUNT := NODECOUNT + 1;
P+.INFO := WORD;
P+.LPT := NIL;
P+.RPT := NIL
END; »MAKENODE«

FUNCTION WGH(IN1,IN2:INTEGER):INTEGER;
PURPOSE: INTEGER FUNCTION WHICH RETURNS THE WEIGHT OF THE SUBTREE WITH
WEIGHTS GIVEN BY THE INPUT PARAMETERS, i.e., BETA FREQ[IN1], ALFA FREQ[IN1+1], . . . , BETA FREQ[IN2], ALFA FREQ[IN2]
PARAMETERS INPUT:
    IN1——THE MINIMUM INDEX TO THE EXTERNAL WEIGHTS
    IN2——THE MAXIMUM INDEX TO THE WEIGHT OF THE KEYS
VAR I,W:INTEGER;
BEGIN »WGH«
    W := BETA FREQ[IN1];
    FOR I := IN1 + 1 TO IN2 DO
        W := W + ALFA FREQ[I] * BETA FREQ[I];
    WGH := W
END; »WGH«
PROCEDURE OPTREEWG(NO,F:INTEGER);

PURPOSE: TO STRUCTURE THE NEARLY OPTIMAL TREE OF WALKER AND GOLTLEB

PARAMETERS INPUT:
F---VARIABLE USED IN DETERMINING THE NEIGHBORHOOD ABOUT THE CENTROID
NO---THE SIZE OF SUBTREES STRUCTURED BY KNUTH'S ALGORITHM

NESTED PROCEDURES: STEP8,STEP7,STEP6,STEP5,STEP4,STEP3,STEP2,STEP1

VAR

CENX----------THIS VARIABLE REPRESENTS THE INDEX OF THE
CENTER IN THE ARRAY SETT
L----------THE INDEX OF THE KEY OF WEIGHT, MAXFREQ, TO THE
LEFT OF THE CENTROID
LEN----------THE TOTAL NUMBER OF KEYS IN THE NEIGHBORHOOD
OF THE CENTROID
LENGTH-------THE NUMBER OF KEYS IN THE INITIAL NEIGHBORHOOD
BEFORE ANY EXPANSION IS PERFORMED
LOGQUANT-----THE FLOOR OF THE LOG BASE TWO OF THE NUMBER
OF KEYS TO BE STRUCTURED INTO A SUBTREE
MAXFREQ-----THE MAXIMUM WEIGHT OF ANY KEY IN THE
NEIGHBORHOOD ABOUT THE CENTROID
R----------THE INDEX OF THE KEY OF WEIGHT, MAXFREQ, TO THE
RIGHT OF THE CENTROID
SETT---------THIS ARRAY IS USED TO CONTAIN THE INDICES OF
KEYS WHICH ARE IN THE NEIGHBORHOOD OF THE
CENTROID
WEIGHTFACTOR--THIS QUANTITY DETERMINES THE NEIGHBORHOOD ABOUT
THE CENTROID

WEIGHTFACTOR,REAL;
CENX,LENGTH,LEN,MXFREQ,LOGQUANT,L,R:INTEGER;
SETT:ARRAY[1..NUM] OF INTEGER;

*** +++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++++ ++++++
BEGIN *STEP 8*

RROOT := SETT[1];
INX := 1;

*FIND THE KEY OF MINIMUM INDEX AND MAXIMUM WEIGHT*

FOR J := 2 TO LEN DO
  IF ALFAFREQ[SETT[J]] GT ALFAFREQ[RROOT] THEN
    BEGIN
      INX := J;
      RROOT := SETT[J]
    END;
TRW := ABS(WGH(INX1 - 1, RROOT - 1) - WGH(RROOT, INX2));

*DETERMINE WHICH KEY AS THE KEY OF THE ROOT NODE WOULD BEST EQUALIZE THE WEIGHT OF ITS LEFT AND RIGHT SUBTREES AND CHOOSE IT AS THE KEY OF THE ROOT*

FOR J := INX + 1 TO LEN DO
  IF ALFAFREQ[SETT[J]] EQ ALFAFREQ[RROOT] THEN
    BEGIN
      NEWTRW := ABS(WGH(INX1 - 1, SETT[J] - 1) - WGH(SETT[J], INX2));
      IF NEWTRW LT TRW THEN
        BEGIN
          RROOT := SETT[J];
          TRW := NEWTRW
        END
    END;
END; *STEP 8*

* + + + + + + + + + + + + + + + + + + + + + + + + + + *

PROCEDURE STEP7(INX2:INTEGER);
* PURPOSE: TO EXPAND THE INITIAL NEIGHBORHOOD TO THE RIGHT
* PARAMETERS INPUT:
  INX2 ---- THE MAXIMUM INDEX TO THE KEY ARRAY USED IN CONSTRUCTING THE CURRENT SUBTREE
VAR PP:INTEGER;
BEGIN *STEP 7*
  *CHECK IF THE KEY OF MAXIMUM WEIGHT IS THE LAST MEMBER OF THE NEIGHBORHOOD; IF SO TRY TO EXPAND THE NEIGHBORHOOD*
  IF R EQ SETT[LENGTH] THEN
    FOR PP := R TO INX2 - 1 DO
      IF(ALFAFREQ[PP] GE ALFAFREQ[PP + 1]) OR (PP - R EQ LOGQUANT) THEN GOTO 10
    ELSE
      BEGIN
        LEN := LEN + 1;
        SETT[LEN] := PP + 1
      END;

10: END; *STEP 7*
FUNCTION STEP6(MAX: INTEGER): BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO FIND THE INDEX OF THE KEY OF WEIGHT, MAXFREQ, TO THE RIGHT OF THE CENTROID. IF THE FUNCTION RETURNS TRUE, THERE IS SUCH AN INDEX AND THE NEIGHBORHOOD MAY BE ABLE TO BE EXPANDED TO THE RIGHT.

PARAMETERS_INPUT:
MAX—THE INDEX OF THE KEY OF MAXIMUM WEIGHT IN THE NEIGHBORHOOD OF THE CENTROID

VAR J: INTEGER;
BEGIN +STEP 6+

STEP6 := TRUE;
R := -1;
FOR J := LENGTH DOWNTO GENX DO
  IF ALFAFREQ[SETT(J)] EQ MAXFREQ
  THEN R := SETT(J);
  IF R LT 0 THEN STEP6 := FALSE;
END; +STEP 6+

PROCEDURE STEP5(INX1: INTEGER);

PURPOSE: TO EXPAND THE INITIAL NEIGHBORHOOD TO THE LEFT

PARAMETERS INPUT:
INX1—THE MINIMUM INDEX TO THE KEY ARRAY USED IN CONSTRUCTING THE CURRENT SUBTREE

VAR PP: INTEGER;
BEGIN +STEP 5+

+CHECK_IF_THE_KEY_OF_MAXIMUM_WEIGHT_IS_THE_FIRST_MEMBER_OF_THE_NEIGHBORHOOD;_IF_SO_TRY_TO_EXPAND_THE_NEIGHBORHOOD+

IF L EQ SETT(1)
  THEN
    FOR PP := INX1 + 1 DOWNTO 0 DO
      IF (ALFAFREQ[PP - 1] LE ALFAFREQ[PP]) OR (L - PP EQ LOGQUANT)
      THEN GOTO 10
      ELSE
        BEGIN
          LEN := LEN + 1;
          SETT[LEN] := PP - 1
        END;
 10 END; +STEP 5+
FUNCTION STEP4(MAX: INTEGER): BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO FIND THE INDEX OF THE KEY OF WEIGHT, MAXFREQ, TO THE LEFT OF THE CENTROID. IF THE FUNCTION RETURNS TRUE, THERE IS SUCH AN INDEX AND THE NEIGHBORHOOD MAY BE ABLE TO BE EXPANDED TO THE LEFT.

PARAMETERS INPUT:
MAX--THE INDEX OF THE KEY OF MAXIMUM WEIGHT IN THE NEIGHBORHOOD OF THE CENTROID

VAR J: INTEGER;
BEGIN *STEP 4*
LEN := LENGTH;
MAXFREQ := ALFAFREQ[MAX];
STEP4 := TRUE;
L := -1;
FOR J := 1 TO CENX DO
  IF ALFAFREQ[SETT[J]] EQ MAXFREQ THEN L := SETT[J];
IF L LT 0 THEN STEP4 := FALSE;
END; *STEP 4*

PROCEDURE STEP3(VAR MAX: INTEGER);

PURPOSE: TO FIND THE INDEX OF THE KEY OF MAXIMUM WEIGHT IN THE NEIGHBORHOOD OF THE CENTROID

PARAMETERS OUTPUT:
MAX--THE DESIRED INDEX

VAR J: INTEGER;
BEGIN *STEP 3*
MAX := SETT[1];
FOR J := 2 TO LENGTH DO
  IF ALFAFREQ[SETT[J]] GT ALFAFREQ[MAX] THEN MAX := SETT[J];
END; *STEP 3*
PROCEDURE STEP2(INX1, INX2: INTEGER);

PURPOSE: TO LOCATE THE CENTROID AND DETERMINE THE KEYS IN THE INITIAL
NEIGHBORHOOD

PARAMETERS INPUT:
INX1—THE MINIMUM INDEX TO THE KEY ARRAY USED IN CONSTRUCTING
THE CURRENT SUBTREE
INX2—THE MAXIMUM INDEX TO THE KEY ARRAY USED IN CONSTRUCTING
THE CURRENT SUBTREE

VAR
CENTROID—THE INDEX OF THE KEY WHICH IS THE CENTROID
DIF, DIFF—QUANTITIES USED TO DETERMINE THE CENTROID AND IF A
KEY WILL BE IN THE NEIGHBORHOOD OF THE CENTROID
LTREE—THE WEIGHT OF THE LEFT SUBTREE OF THE NODE IN
QUESTION
RTREE—THE WEIGHT OF THE RIGHT SUBTREE OF THE NODE IN
QUESTION

DIF, DIFF: REAL;
J, CENTROID, LTREE, RTREE: INTEGER;

BEGIN *STEP 2*

* INITIALIZE *

LENGTH := 0;
CENTROID := INX1;
LTREE := BETAFreq(INX1 - 1);
RTREE := WGH(INX1, INX2);
WEIGHTFACTOR := (LTREE + RTREE + ALFAFreq(INX1))/F;
DIF := ABS(LTREE - RTREE);
IF DIF LT WEIGHTFACTOR
THEN
BEGIN
  SETT[1] := INX1;
  LENGTH := 1;
END;

* FOR EACH KEY DETERMINE IF IT WILL BE IN THE NEIGHBORHOOD
  OF THE CENTROID *

FOR J := INX1 + 1 TO INX2 DO
BEGIN
  LTREE := LTREE + ALFAFreq(J - 1) + BETAFreq(J - 1);
  RTREE := RTREE - ALFAFreq(J) - BETAFreq(J - 1);
  DIF := ABS(LTREE - RTREE);
  IF DIF LT WEIGHTFACTOR
  THEN
     BEGIN
       LENGTH := LENGTH + 1;
       SETT[LENGTH] := J
     END;

  * CHECK FOR THE CENTROID *
  IF DIF LE DIFF
  THEN
     BEGIN
       CENTROID := J;
       DIFF := DIF
     END
  END;

* DETERMINE THE INDEX OF THE ARRAY SETT WHICH IS THE CENTROID *

CENT := -1;
FOR J := 1 TO LENGTH DO
IF SETI(J) EQ CENTROID
  THEN CENX := J;
IF CENX LT 0
  THEN
    *THE NEIGHBORHOOD IS EMPTY; PLACE THE CENTROID IN THE NEIGHBORHOOD*
    BEGIN
      SETI(J) := CENTROID;
      LENGTH := 1;
      CENX := 1;
    END; *STEP 2*

PROCEDURE STEP1(INX1, INX2; INTEGER; FATHER; POINT);
PURPOSE: THIS PROCEDURE CONTROLS THE NECESSARY STEPS TO CONSTRUCT THE NEARLY OPTIMAL TREE

PARAMETERS INPUT:
  FATHER---THE POINTER TO THE ROOT OF THE SUBTREE TO BE CREATED
  INX1----THE MINIMUM INDEX TO THE KEY ARRAY USED IN CONSTRUCTING THE CURRENT SUBTREE
  INX2----THE MAXIMUM INDEX TO THE KEY ARRAY USED IN CONSTRUCTING THE CURRENT SUBTREE

VAR
  *NUMNODE---THE NUMBER OF NODES THAT WILL BE IN THE SUBTREE TO BE CREATED
  MAX------THE INDEX OF THE KEY OF MAXIMUM WEIGHT IN THE NEIGHBORHOOD OF THE CENTROID
  RROOT-----THE INDEX OF THE KEY CHOSEN AS ROOT OF THE SUBTREE BEING CONSTRUCTED*
  P; POINT;
  NUMNODE, SAVE, MAX, RROOT; INTEGER;
BEGIN *STEP 1*
  *DETERMINE THE NUMBER OF KEYS IN THE SUBTREE AND IF IT CAN BE STRUCTURED BY KNUUTH'S ALGORITHM*
  NUMNODE := INX2 - INX1 + 1;
  IF NUMNODE GE 0
    THEN
      BEGIN
        IF NUMNODE NE 0
          THEN
            *STRUCTURE USING KNUUTH'S ALGORITHM*
            BEGIN
              SAVE := NODECOUNT;
              NDFREEKNUTH(INX1, INX2);
              NODECOUNT := SAVE + NODECOUNT;
              IF FATHER + INFO GT KROOT + INFO
                THEN FATHER + LPT := KROOT
              ELSE FATHER + RPT := KROOT
            END
          ELSE
            ELSE
          END
        END
      END
    ELSE
      ELSE
    END
END
• FIND THE ROOT OF THE SUBTREE WITH THE WALKER GOTO
ALGORITHM •

BEGIN
LOGQUANT i = TRUNC(LN(NUMNODE)/LN(2));
STEP2(INX1, INX2);
STEP3(MAX);
IF STEP4(MAX) THEN STEP5(INX1);
IF STEP6(MAX) THEN STEP7(INX2);
STEP8(RROOT, INX1, INX2);
IF ACQUIRE(P) THEN GOTO EXIT 999;
MAKEVOE(WD[RROOT], P);
IF FATHER NE NIL
THEN
   IF WDI[RROOT] LT FATHER*.INFO
       THEN FATHER*.LPT I = P
       ELSE FATHER*.RPT I = P
   ELSE WGROOT I = P;
   FATHER I = P;
• STRUCTURE THE LEFT AND RIGHT SUBTREES •
STEP1(INX1, RROOT - 1, FATHER);
STEP1(RROOT + 1, INX2, FATHER)
END; • STEP 1 •

BEGIN •OPTREEWG •

IF NUM LE NO
   THEN WRITE(• USE KNUTHS ALGO FOR THIS NUMBER OF ITEMS 3, EOL)
   ELSE STEP1(1, NUM, NIL);
END; •OPTREEWG •

BEGIN •OPTREEWG DRIVER •

• PERFORM INITIALIZATIONS •
INIT;
• CREATE THE TREE •
OPTREEWG(NO, F);
999; END; • OPTREEWG DRIVER •
**B** B T R E E S T A T I S T I C S R O U T I N E S

**CONSTANTS DEFINED GLOBALLY**

INX1, STATSIZE -- THE DIMENSIONS OF THE STATISTICAL ARRAYS
MAXSIZE -- THE NUMBER OF NODES IN THE TREE TO BE CREATED
NUMTREES -- THE NUMBER OF TREES BUILT UP AND BROKEN DOWN
SHUFFLEFACTOR -- THE PARAMETER TO THE PROCEDURE SHUFFLE WHICH DETERMINES HOW MANY ELEMENTS OF AN ARRAY ARE SHUFFLED;
TREEINTERVAL -- STATISTICS ARE COLLECTED FOR TREES WHOSE SIZE IS A MULTIPLE OF TREEINTERVAL

**TYPES DEFINED GLOBALLY**

ALFA -- A REDEFINITION OF THE PREDEFINED TYPE AS A SUBRANGE OF INTEGER VALUES;
TYPEOFARRAY -- THE TYPE DEFINING THE RANDOM NUMBER ARRAY

**VARIABLES DEFINED GLOBALLY**

I1, I2, I3, I01, I23, I23 -- COUNTERS USED IN STATISTICS COLLECTION
ROOTNUMTREES -- THE SORT OF THE NUMBER OF TREES CREATED
SEED -- THE SEED FOR THE RANDOM NUMBER GENERATOR
RANDNUM -- THE RANDOM NUMBER ARRAY

INX1 = 4;
NUMTREES = 500;
MAXSIZE = 50.1;
SHUFFLEFACTOR = 5.0;
STATSIZE = 5.0;
TREEINTERVAL = 1.0;
ALFA = 1.0; 10.0;
TYPEOFARRAY = ARRAY[1..MAXSIZE] OF INTEGER;
I1, I2, I3, I01: INTEGER;
I23, I23: INTEGER;
NUMTREES: REAL;
SEED, I1, I2, I3, I01: INTEGER;

**FUNCTION MEAN (SUM: INTEGER) : REAL**

**PURPOSE:** FUNCTION TO FIND THE MEAN OF ITS ARGUMENT

BEGIN
  MEAN := SUM / NUMTREES;
END;
FUNCTION SD(SUM, SUMSQ: INTEGER): REAL;

PURPOSE: FUNCTION TO FIND THE STANDARD DEVIATION FROM ITS ARGUMENTS

PARAMETERS INPUT:
SUM: SUM OF THE OBSERVATIONS
SUMSQ: SUM OF THE SQUARES OF THE OBSERVATIONS

BEGIN
SD := SQRT((NUMTREES*SUMSQ - SUM*SUM) / (NUMTREES*(NUMTREES - 1)));
END;

FUNCTION SEARCH(WORD, ALFA: INTEGER);

PURPOSE: TO COUNT THE NUMBER OF COMPARISONS NEEDED TO FIND THE NODE WITH GIVEN INPUT KEY

VAR
P: POINT;
SEARCHX: INTEGER;

BEGIN
SEARCH := 0;
P := HEAD.PRT;
SEARCHX := ?;
WHILE P NE NIL DO
BEGIN
SEARCH := SEARCH + 1;
IF P.INFO NE WORD THEN
IF WORD LT P.INFO
THEN P := P.LPT
ELSE P := P.RPT
ELSE
BEGIN
SEARCH := SEARCH;
GOTO 10;
END;
END;
SEARCH := SEARCH + 1;
10: END: "SEARCH"

PROCEDURE BSTATISTICS;

PURPOSE: TO COLLECT STATISTICS ON THE BTREE

VAR
RANDNUM: TYPE OF ARRAY;
STATSUP, STATSDP: ARRAY[1..10, 1...STATS7E] OF INTEGER;
I, J, II: INTEGER;
AV1, AV2, AV3, AV4, SD1, SD2, SD3, SD4: REAL;

STATSUP VARIABLE
1: SEARCH COUNTS I1
2: SINGLE ROTATION COUNTS I2
3: DOUBLE ROTATION COUNTS I3
4: SUM OF SQUARES OF 1

STATSDP VARIABLE
1: SEARCH COUNTS D1
2: SINGLE ROTATION COUNTS I2
3: DOUBLE ROTATION COUNTS I3
4: SUM OF SQUARES OF 1
PROCEDURE BUILDANDDESTROY;

PURPOSE: TO BUILD UP AND BREAK DOWN TREES AND COLLECT STATISTICS

VAR
  I,J,L,IJ:INTEGER;
BEGIN
  BUILDANDDESTROY;
  PERMUTE THE ELEMENTS IN THE RANDOM NUMBER ARRAY
  SHUFFLE(RANDNUM,SEED,MAXSIZE,SHUFFLEFACTOR):
  FOR I := 1 TO NUMTREES DO
  BEGIN
    IJ := TREEI NTERVAL; 10
    L := 1;
  END;
  COLLECT INSERTION STATS
  FOR J := 1 TO MAXSIZE DO
  BEGIN
    IJ := IJ - 1;
    IF IJ EQ 0
    THEN
      BEGIN
        IJ := TREEI NTERVAL;
        T1 := SEARCH(RANDNUM(IJ));
        I2 := L;
        I3 := L;
        RINSERT(RANDNUM(IJ));
        STATSUP[1,L] := STATSUP[1,L] + I1;
        STATSUP[4,L] := STATSUP[4,L] + I1*1;
        L := L + 1;
      END;
    ELSE
      BEGIN
        RINSERT(RANDNUM(IJ));
      END;
    END;
  IJ := 2;
  L := MAXSIZE;
  SHUFFLE(RANDNUM,SEED,MAXSIZE,SHUFFLEFACTOR);
  COLLECT DELETION STATS
  FOR J := 1 TO MAXSIZE DO
  BEGIN
    IJ := IJ - 1;
    IF IJ EQ 0
THEN
BEGIN
  IJ := TREEINTERVAL;
  D1 := SEARCH(RANDNUM[IJ]);
  I2 := C;
  I3 := G;
  RDELETE(RANDNUM[IJ]);
  STATSOP[1,L] := STATSOP[1,L] + D1;
  L := L - 1;
END;
ELSE RDELETE(RANDNUM[IJ]);
END;
END;
END: BUILDANDDESTROY

+B9STATISTICS+

+INITILIZE+
FOR I := 1 TO INX1 DO
  FOR J := 1 TO STATSIZE DO
    BEGIN
      STATSOP[I,J] := 0;
      STATSOP[I,J] := 0;
    END;
SEED := 73559;

+INITIALIZE RAOON NUMBER ARRAY+
FOR I := 1 TO MAXSIZE DO
  RANDNUM[I] := 1;
ROOTNUMTREES := STAT(NUMTREES);

+INITIAL ALL COUNTERS+
I2 := 0;
I3 := 0;
D1 := 0;

+BUILD AND DESTROY TREES+
CREATEHEADER;
BUILDANDDESTROY;
WRITE("E-10-INSERTION THE NUMBER OF TREES OBSERVED IS ,NUMTREES, ,EOL");
WRITE("ALPHA IS ,ALPHA, BETA ,BETA, ,EOL");
WRITE("AVERAGE CONFIDENCE INTERVAL",EOL);
WRITE("SIZE SEARCH INTERVAL SINGLE INTERVAL DOUBLE E");
WRITE(3, E, EOL);

FOR I := 1 TO STATSIZE DO

BEGIN

  II := I * TREE*INTERVAL;
  AV1 := MEAN(STATSUP(1, I));
  SD1 := SD(STATSUP(1, I), STATSUP(4, I));
  AV2 := MEAN(STATSUP(2, I));
  SD2 := 2. * SD/ROOTNUMTREES;
  AV3 := MEAN(STATSUP(3, I));
  SD3 := 2. * SD/ROOTNUMTREES;
  I23 := STATSUP(2, I) + STATSUP(3, I);
  AV4 := MEAN(I23);
  SD4 := 2. * SD/ROOTNUMTREES;

  WRITE(E, E, III, 5, E, EOL):
  WRITE(AV1, 9, 14, E, EOL):
  WRITE(SD1, 9, 14, E, EOL):
  WRITE(AV2, 9, 14, E, EOL):
  WRITE(SD2, 9, 14, E, EOL):
  WRITE(AV3, 9, 14, E, EOL):
  WRITE(SD3, 9, 14, E, EOL):
  WRITE(AV4, 9, 14, E, EOL):
  WRITE(SD4, 9, 14, E, EOL):

END;

WRITE(E, E, EOL):

FOR I := 1 TO STATSIZE DO

BEGIN

  II := I * TREE*INTERVAL;
  AV1 := MEAN(STATSUP(1, I));
  SD1 := SD(STATSUP(1, I), STATSUP(4, I));
  AV2 := MEAN(STATSUP(2, I));
  SD2 := 2. * SD/ROOTNUMTREES;
  AV3 := MEAN(STATSUP(3, I));
  SD3 := 2. * SD/ROOTNUMTREES;
  I23 := STATSUP(2, I) + STATSUP(3, I);
  AV4 := MEAN(I23);
  SD4 := 2. * SD/ROOTNUMTREES;

  WRITE(E, E, III, 5, E, EOL):
  WRITE(AV1, 9, 14, E, EOL):
  WRITE(SD1, 9, 14, E, EOL):
  WRITE(AV2, 9, 14, E, EOL):
  WRITE(SD2, 9, 14, E, EOL):
  WRITE(AV3, 9, 14, E, EOL):
  WRITE(SD3, 9, 14, E, EOL):
  WRITE(AV4, 9, 14, E, EOL):
  WRITE(SD4, 9, 14, E, EOL):

END;

END: "BSTATISTICS"
APPENDIX 4

PRACTICAL CONSIDERATIONS

Knuth's optimal tree algorithm was used to construct two search trees (their respective sizes were less than 40), one for the keywords and the other for the predefined identifiers of the programming language Pascal (Wirth, 1971). These optimal trees were used by C.A. Bryce (McMaster University) in a Pascal cross-reference program and resulted in an average time saving of approximately 10% as compared with unoptimized trees.

The binary tree display algorithm was used by C.B. Johns, a fellow graduate student at McMaster University, to print a reverse syntax graph which is needed for precedence parsing (note reverse of syntax graph). A reverse syntax graph is a particular way of representing the right-hand sides of rules of a grammar. It may be treated as a set of binary trees. The reverse syntax graph is used for matching a potential right-hand side of a rule in the parser and can be considered as an unordered tree, which means that the display algorithm can be used to display it. Because of the data structure used for the reverse syntax graph, an ordering is imposed upon it. This is as follows: left corresponds to alternative, right corresponds to successor. For example consider the following rules:

\[\text{block} \rightarrow \text{blockbody} \langle \text{end} \rangle\]
\[\text{block} \rightarrow \text{blockbody statement} \langle \text{end} \rangle\]
\[\text{blockbody} \rightarrow \text{blockbody statement} \langle ; \rangle\]
blockbody → blockbody labeldef

blockbody → blockbody < ; > .

This will give us one tree. The reverse syntax graph is a set of such trees. The root of this tree is blockbody; a printout of the tree is shown below.