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

Page

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

Page

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 BBCA 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
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

LIST OF TABLES

		Page
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

Page

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 Adel'son-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 ℓ. 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 Adel'son-Vel'skii 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 efficient 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 Gotlieb (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 tree(u) to indicate either a tree with root node u or, if u = ∅, 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 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 weight which for any key X_i is denoted α_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 β_i, $0 \leq i \leq n$, which give the frequency of unsuccessful searches and are called external weights. Considering the keys as ordered, β_i is the frequency of a key occurring between X_i and X_{i+1}, β_0 and β_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 α_i, $1 \leq i \leq n$, and β_i, $0 \leq i \leq n$.

In the above definition of an unweighted binary search tree, the β_i are zero and the α_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 $\alpha_i \geq 1$, $1 \leq i \leq n$, the external weights being zero. A weighted external binary search tree is a weighted tree where the $\beta_i \geq 0$, $0 \leq i \leq n$. 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, tree(u), with 2n+1 weights α_i and β_i we have the following definitions and terminology.

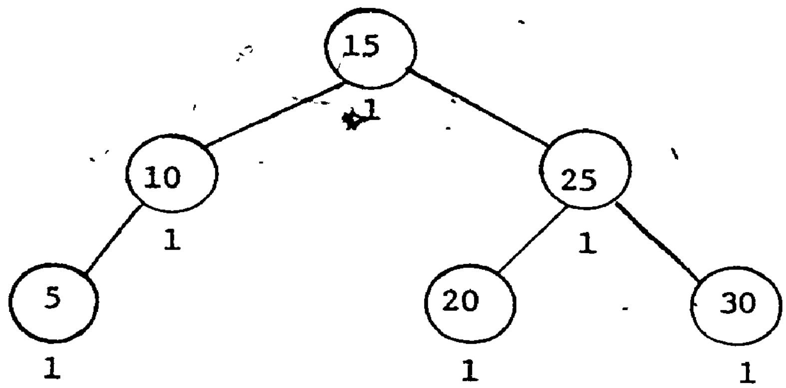
Definition

A search path from u to some node v in 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, $\ell(v)$, is the number of nodes in P(v). The height of tree(u), h(u), is the maximum level of any node in the tree. The size of tree(u), denoted size(u), is the number of nodes in tree(u). For any v in tree(u), v_ℓ and v_r denote the left and right sons of v. The subtree defined by v, tree(v), is the set of all nodes w in tree(u) such that v is an ancestor of w together with v itself. key(v) denotes the key associated with v. The weight of 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 tree(u), 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_ℓ) + WPL(u_r) + W(u). The <u>average weighted path length</u> is defined as AVGWPL (u) = $\frac{\text{WPL(u)}}{\text{W(u)}}$. The <u>value</u> of tree(u), V(u), is $\sum\ell(v)$ for all v in tree(u). The <u>minimum value</u>, denoted MINV(n), is (n+1) × r − n+2p, where r is such that n = 2^r−1+p and 0 ≤ p ≤ 2^r. The <u>maximum value</u> of tree(u), MAXV(n) is 1+2+...+n = $\frac{n(n+1)}{2}$ Letting T_n be the set of all ordered trees of n nodes (with keys 1,...,n), then the <u>mean value</u> of an n node tree, (MV(n)), 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 <u>average search length</u> 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 α_i's are unity and the β_i's are zero, the minimization of the weighted path length produces a minimum valued tree. We say that a key, X, <u>is inserted into tree(u)</u> if there exists tree(u'), such that tree(u) ⊆ tree(u'), u = u' unless u = ∅, size(u') = n+1, tree(u') is a binary search tree, and for v in tree(u') − tree(u), key(v) = X. Correspondingly, we say that a key, X, <u>is deleted from tree(u)</u> if there exists tree(u') such that tree(u') ⊆ 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) − tree(u'), key(v) = X. Finally we define the notions of "romove" and "replace" for binary trees (see Figure 2.1.4). Given tree(u), u ≠ 0, then given a node v in tree(u), v is <u>removed</u> from tree(u)

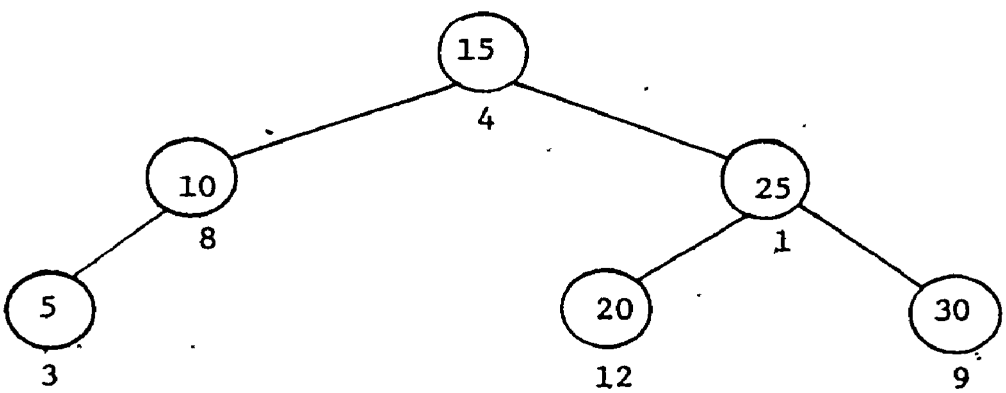
∅

key, X say, and a node v in tree(u) then key(v) is <u>replaced</u> by X when v is given X as its key.



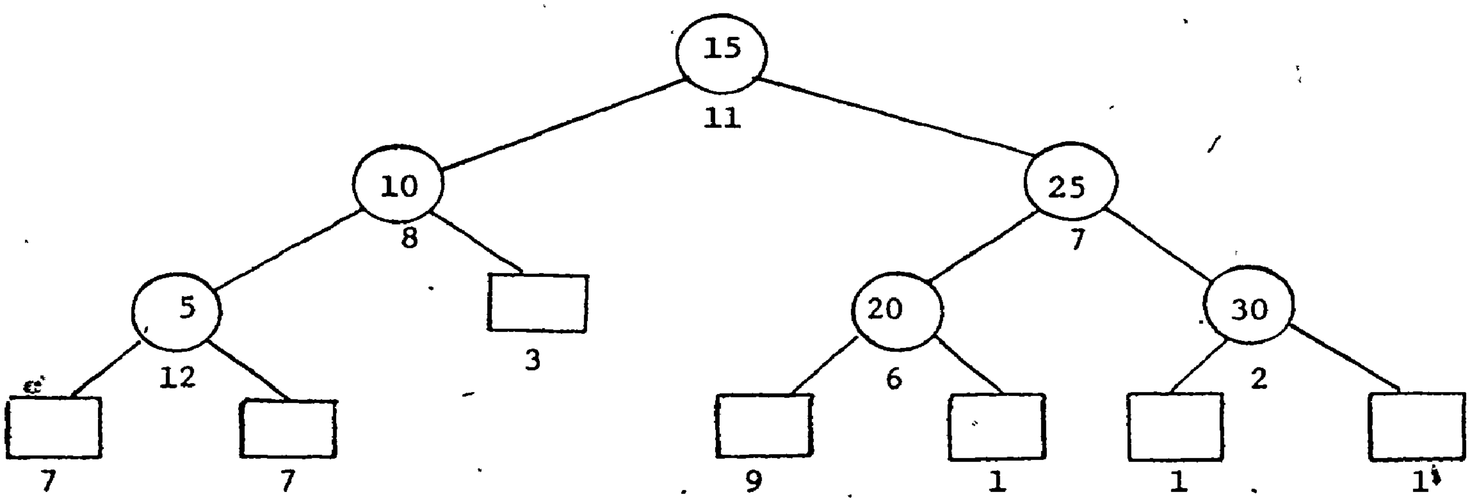
An example of an unweighted tree

Figure 2.1.1



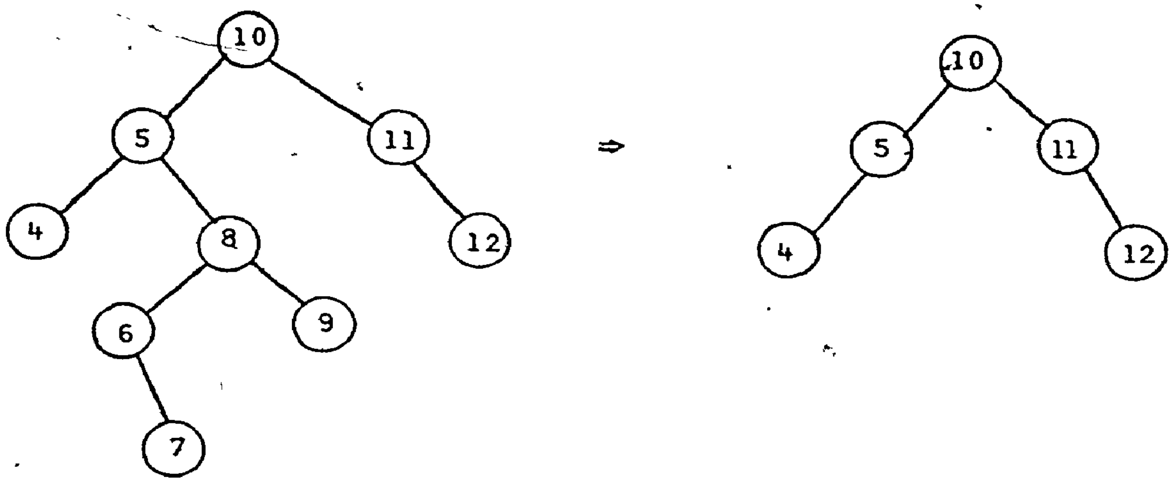
An example of a weighted tree

Figure 2.1.2

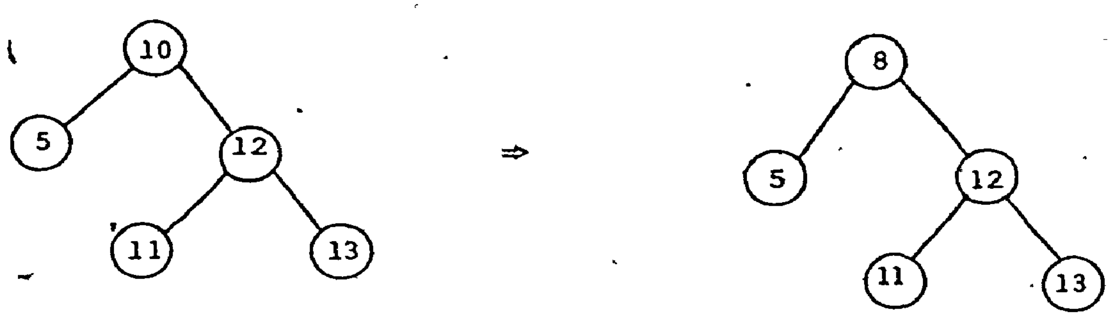


An example of a weighted external tree.

<u>Figure 2.1.3</u>



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)

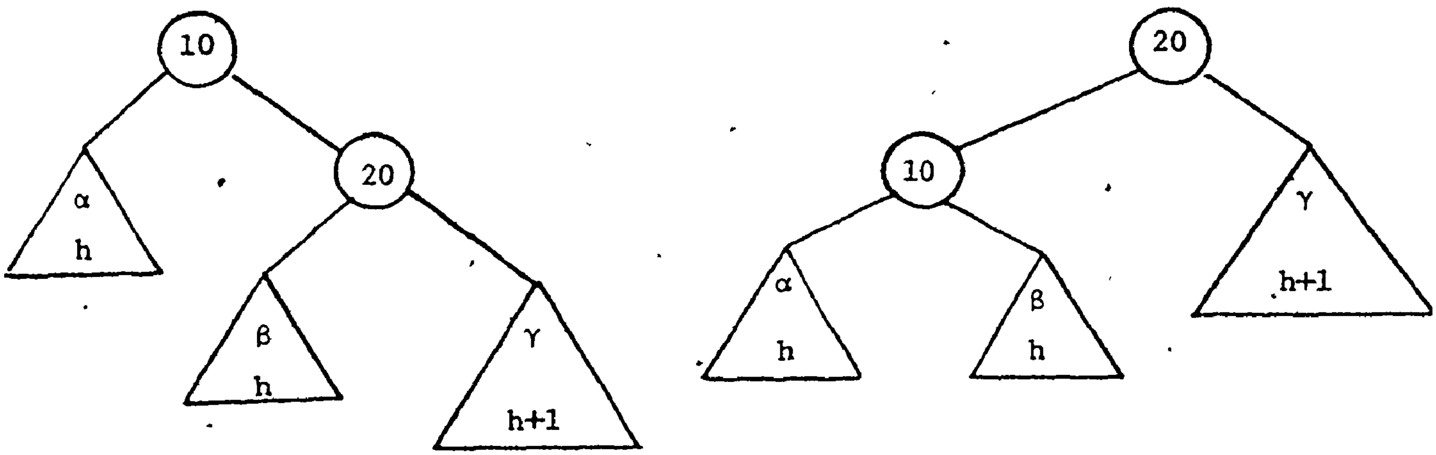
1) p↑.info

2) p↑.lpt

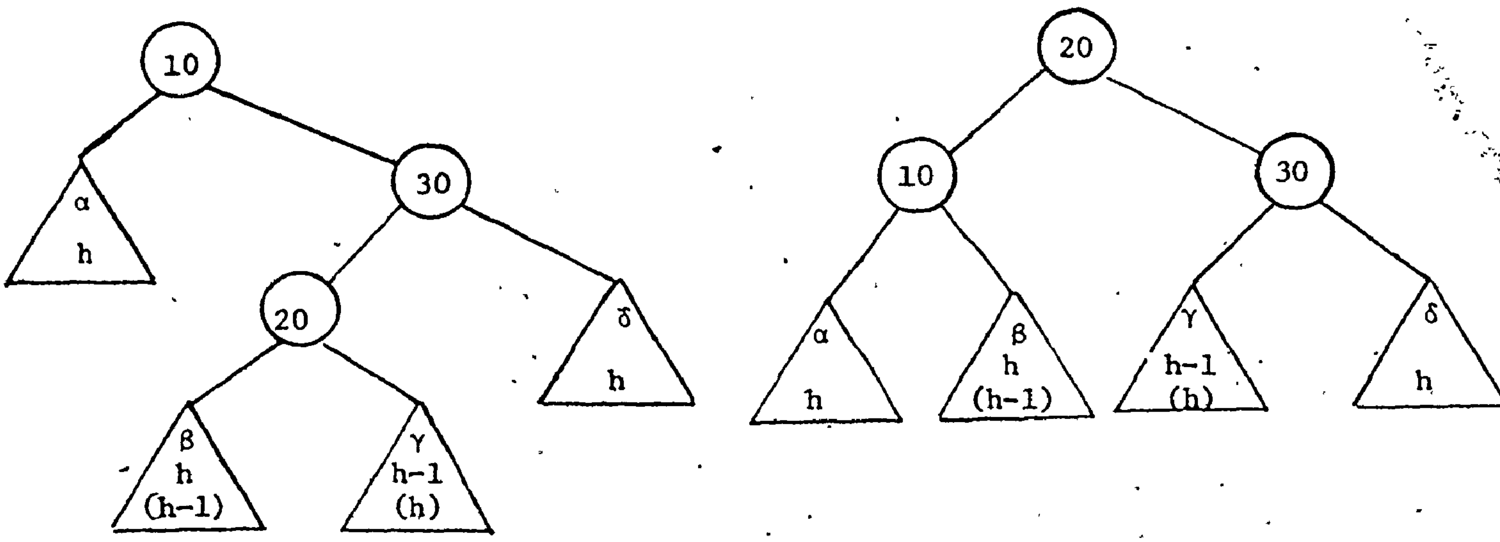
3) p↑.rpt.

(iii) Use is made of the Pascal special symbol <u>nil</u> 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 <u>nil</u> since it has no sons.

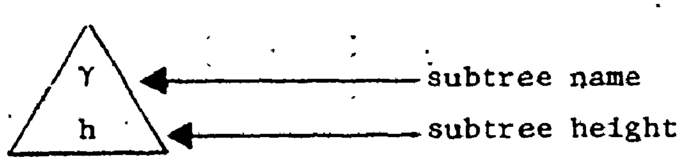
A listing of the Pascal procedures for each tree is found in Appendix 3. The utility routines which are needed by most of the construction algorithms are found in Appendix 2.



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 = Ø 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 = Ø 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.

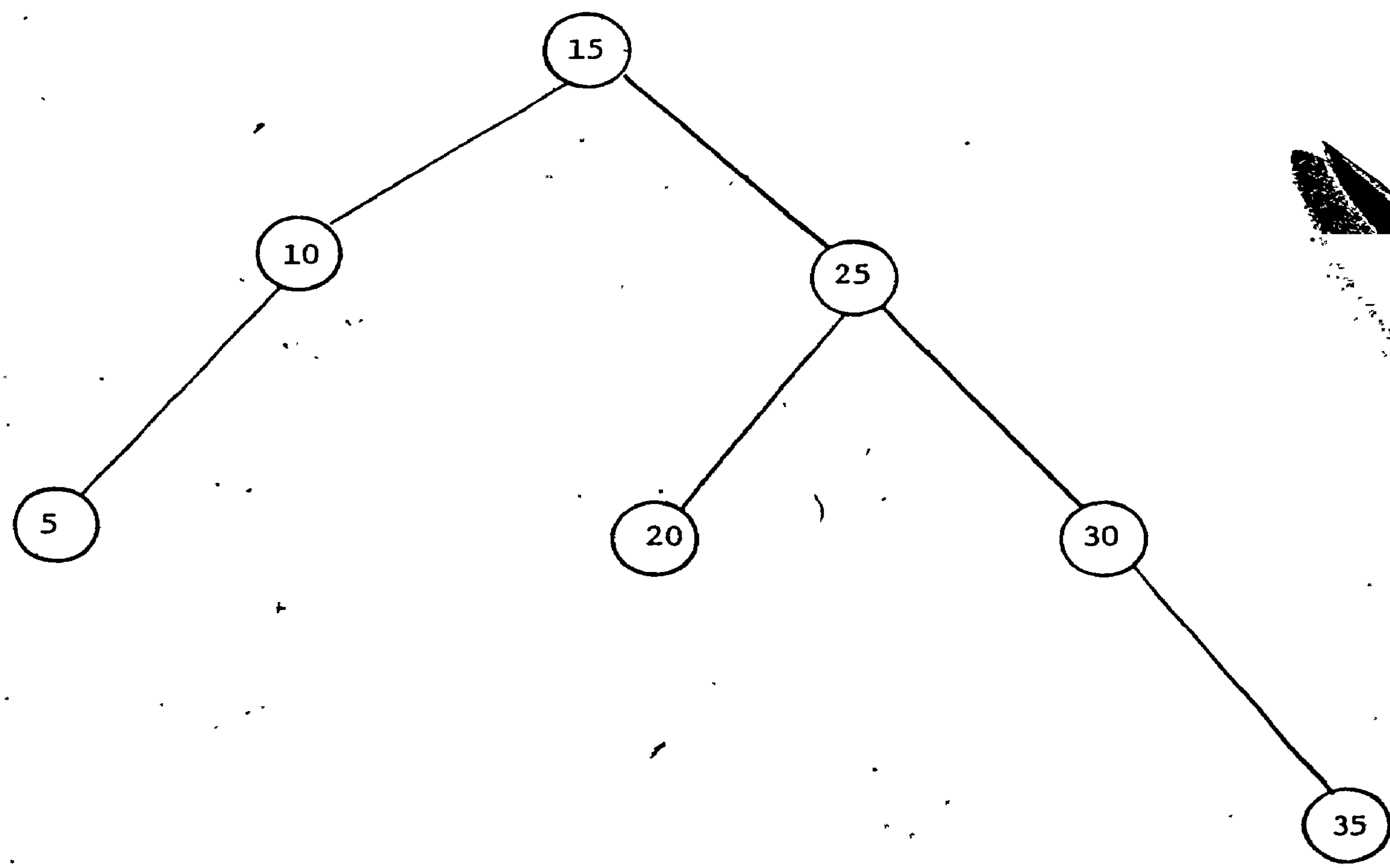
Let z be the postorder predecessor of u. Replace key(u) with key(z)+ and repeat step (ii) with u = z and X = key(z).

(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 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 <u>insertion sequence</u> 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 <u>critical node</u>, that is find the maximum i, $1 \leq i < k$, such that balfac$(u_i) \neq 0$; if there is no such node set i 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.

(i) 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 balfac$(u_k) = 1$, let z be the postorder successor of u_k. Otherwise, let z be the postorder predecessor of u_k. Replace key(u_k) with 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) balfac$(u_\ell) = 0$.

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

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

The height of tree(u_ℓ) is not reduced and the rest of the tree will be unaffected so stop (see Figure 3.3.13).

(c) 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_ℓ. Three cases result:

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

(2) balfac$(u_\ell) = -2$ and 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) balfac(u_ℓ) = 2 and balfac(v) = -1 or

(2) balfac(u_ℓ) = -2 and balfac(v) = 1.

A double rotation is performed as illustrated in Figure 3.3.16.

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

(c) balfac(u_ℓ) = 2 or -2 and 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 balfac(w) , for each node w in 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 = right, -1 = 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, $1 \leq i \leq m$, 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_ℓ, $\ell < m$, are adjusted as each is examined. Step (iii) of AVLDA now may be interpreted as:

(a) balfac(u_ℓ) = a_ℓ.

u_ℓ was left or right heavy and a node was deleted from its left or right subtree respectively.

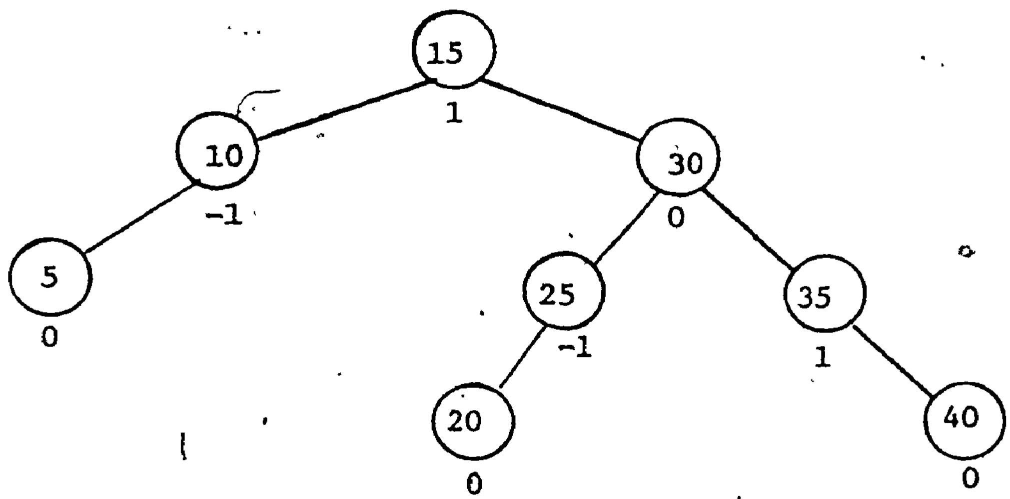
(b) balfac(u_ℓ) = 0.

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

(c) balfac(u_ℓ) = $-a_\ell$.

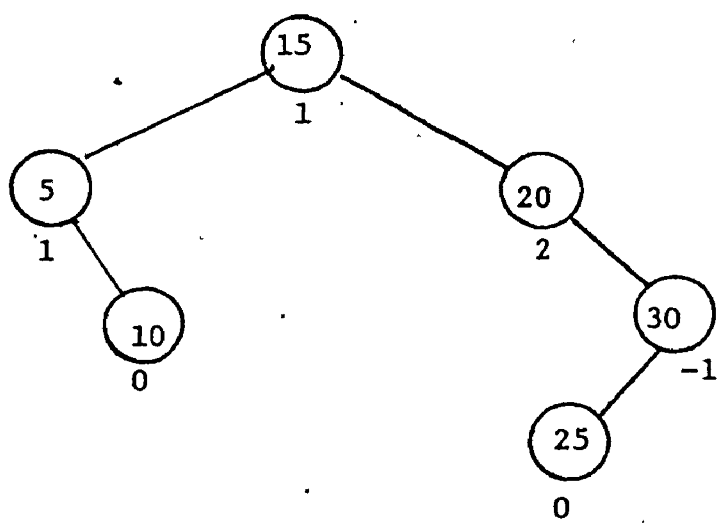
u_ℓ was left or right heavy and a node was deleted from its right or left subtree respectively. Rebalancing is necessary.

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



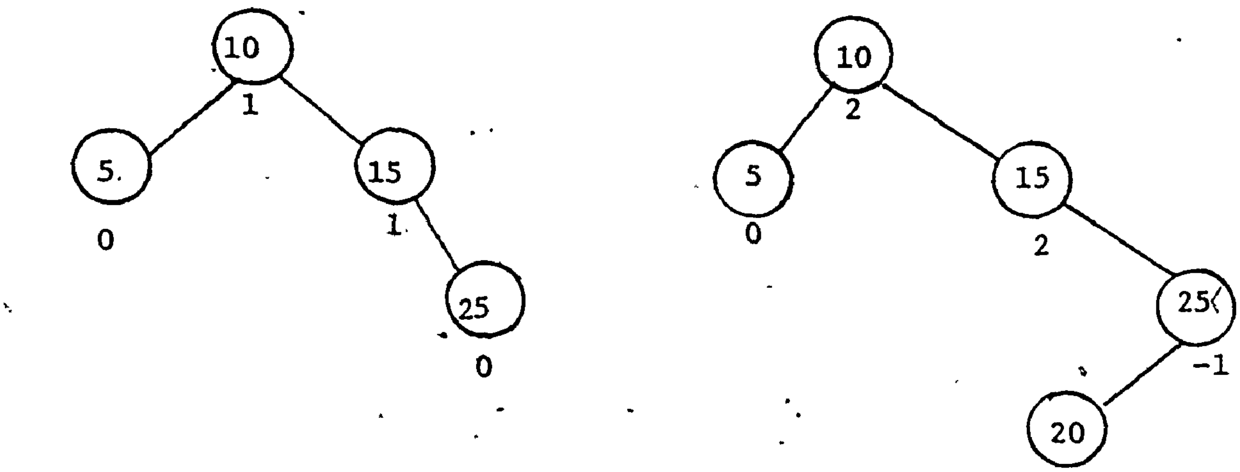
An AVL tree

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



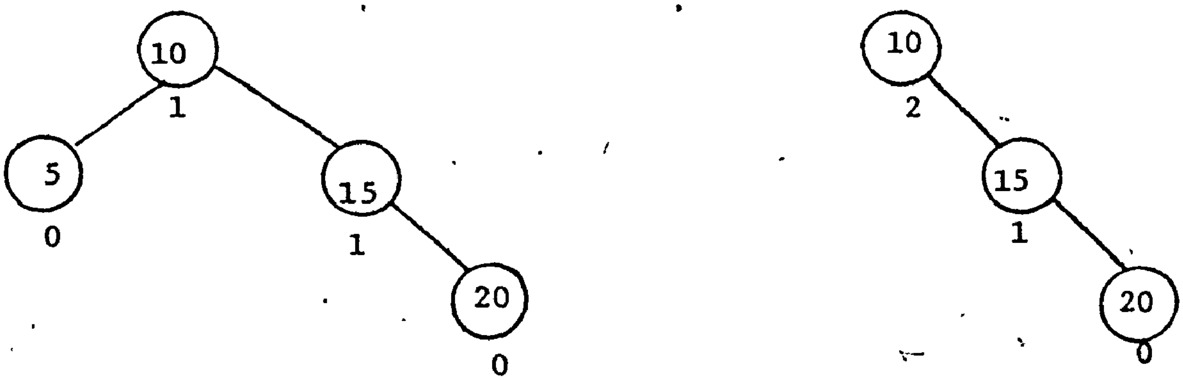
A non-AVL tree

Figure 3.3.1



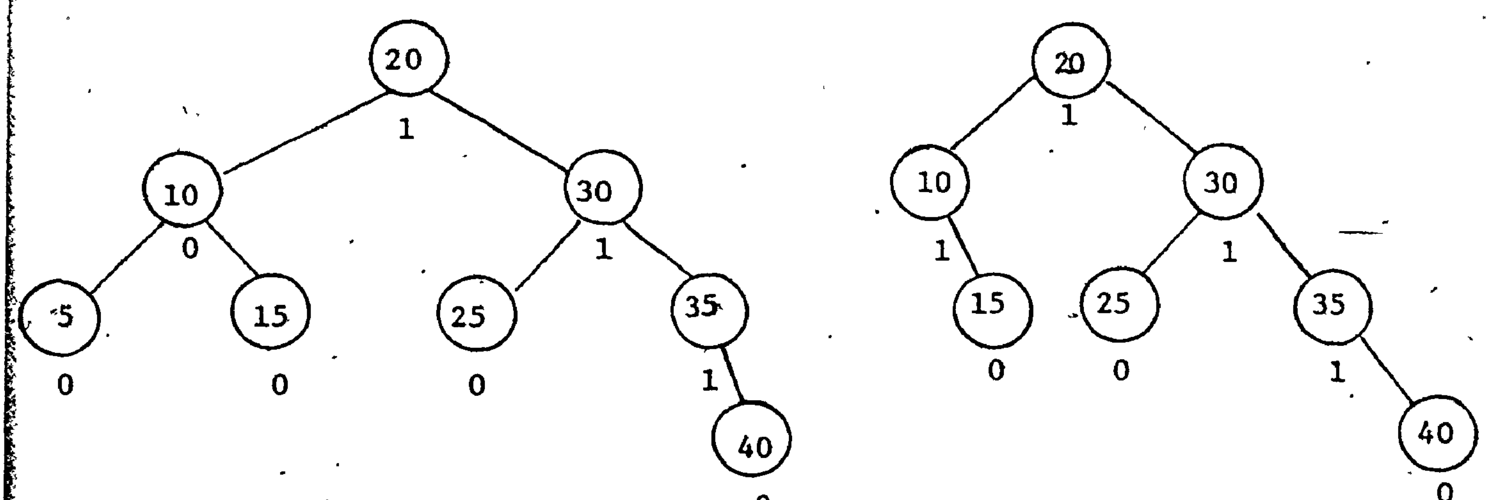
key (w) = 10

Insertion of node with key 20 causes unbalance.



key (w) = 10

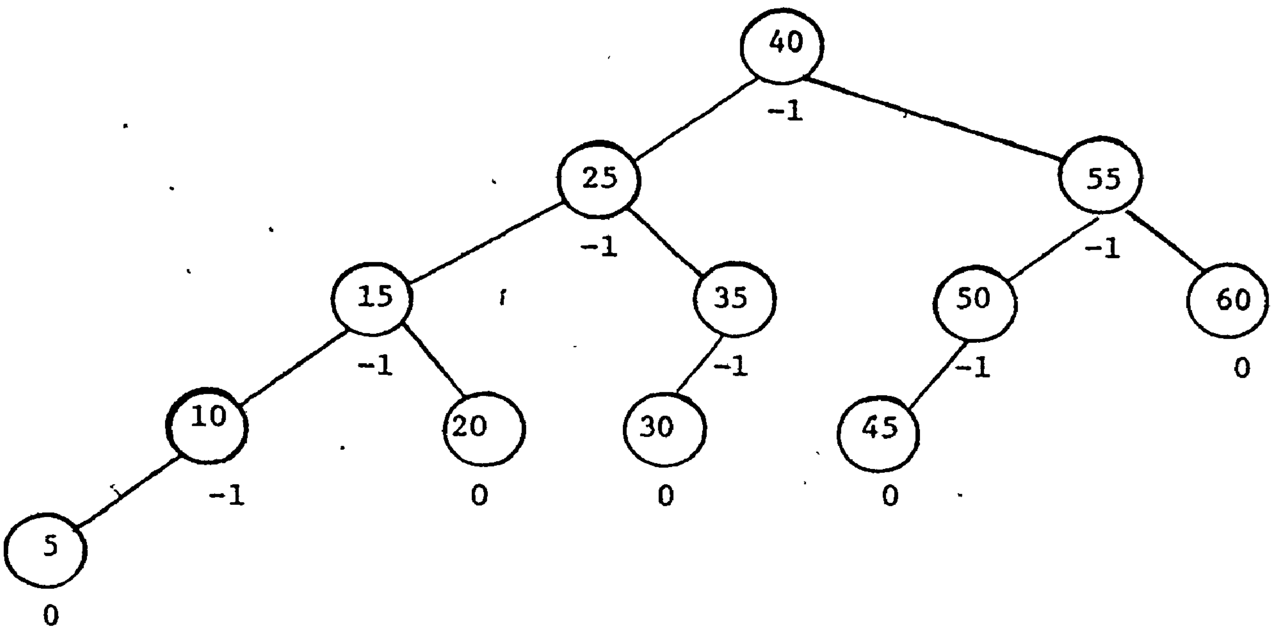
Deletion of node with key 5 causes unbalance.



key (w) = 20.

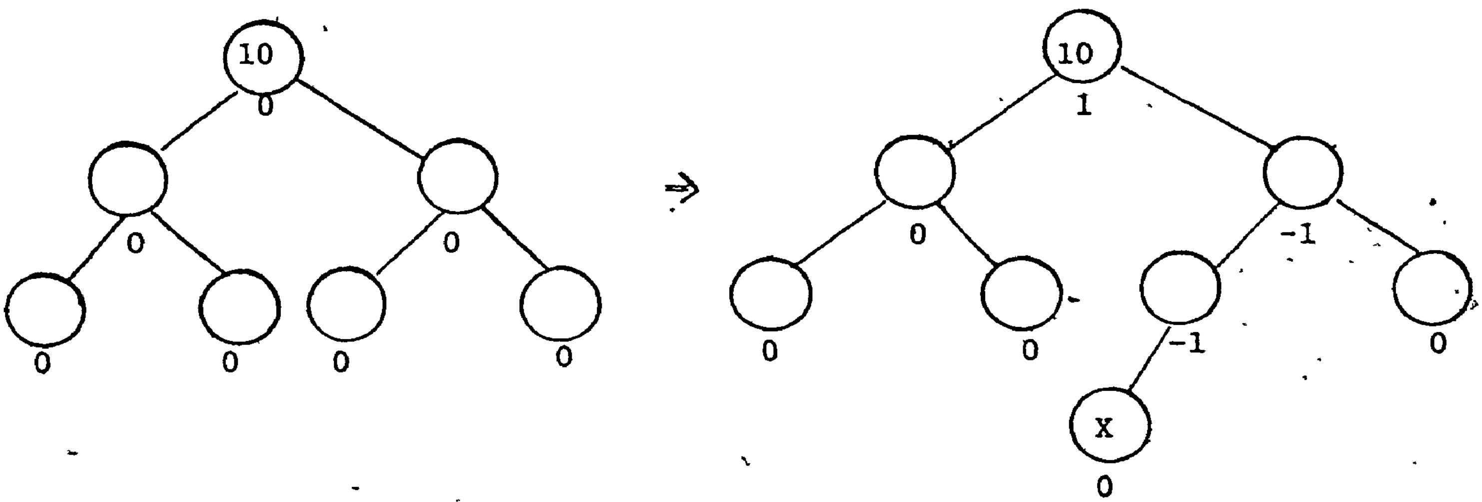
Deletion of node with key 5 does not cause unbalance.

Figure 3.3.2



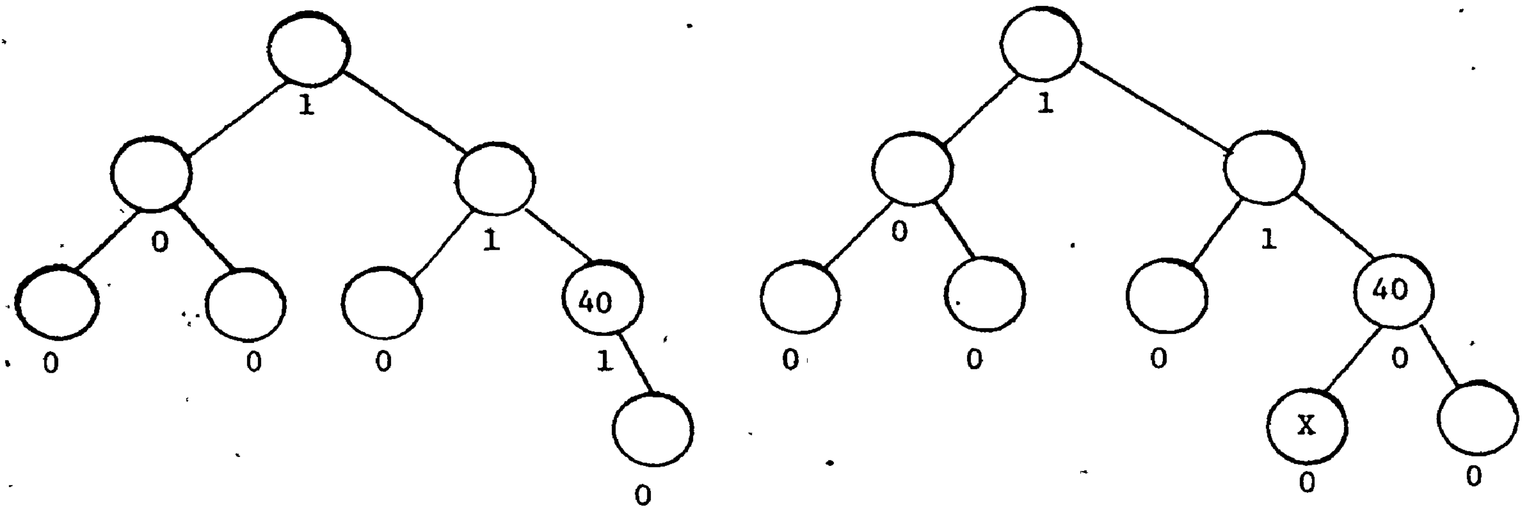
Deletion of node with key 60 causes $\lfloor \log_2 12 \rfloor -1 = 2$ rebalances.

Figure 3.3.3



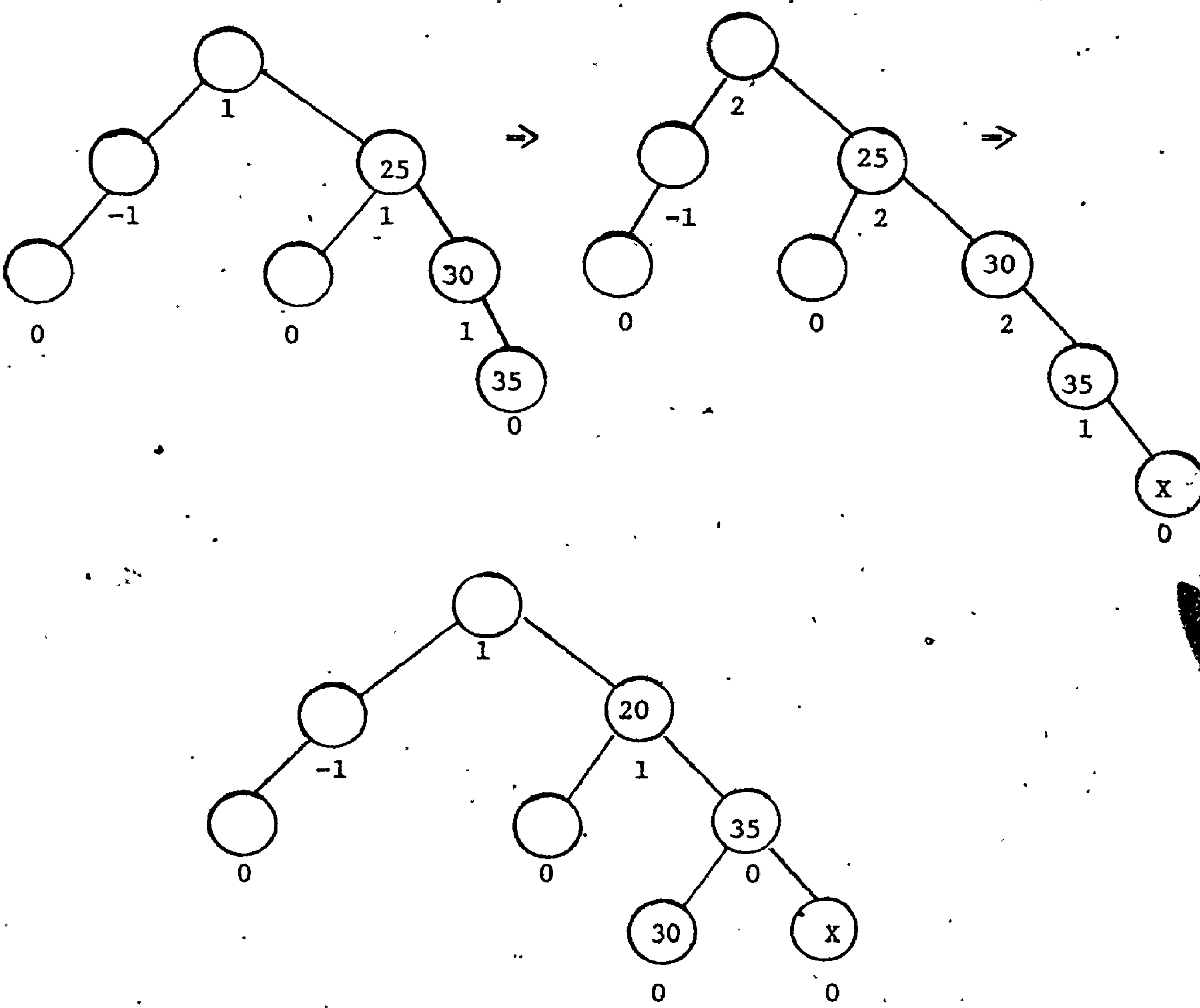
$i = 1$, key(u_i) = 10 balfac(u_i) = 0. Insertion of X causes the height of the tree to be increased.

Figure 3.3.4



$i = 3$, $key(u_i) = 40$, $balfac(u_i) = 0$. Insertion of X causes tree to become more balanced.

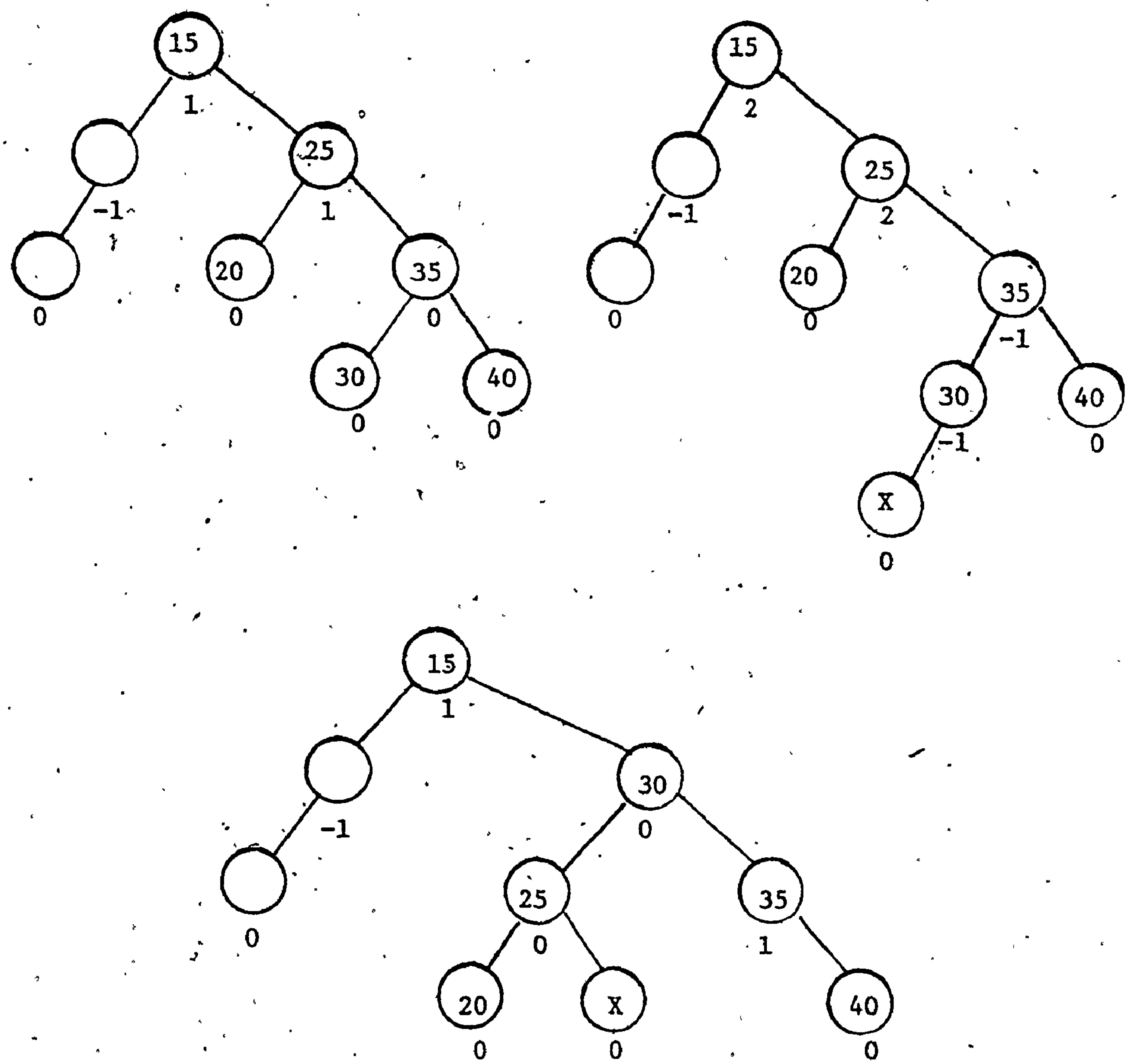
Figure 3.3.5



$i = 3$, $key(u_i) = 30$, $balfac(u_i) = 2$, $balfac(u_{i+1}) = 1$.

A single rotation restores the balance.

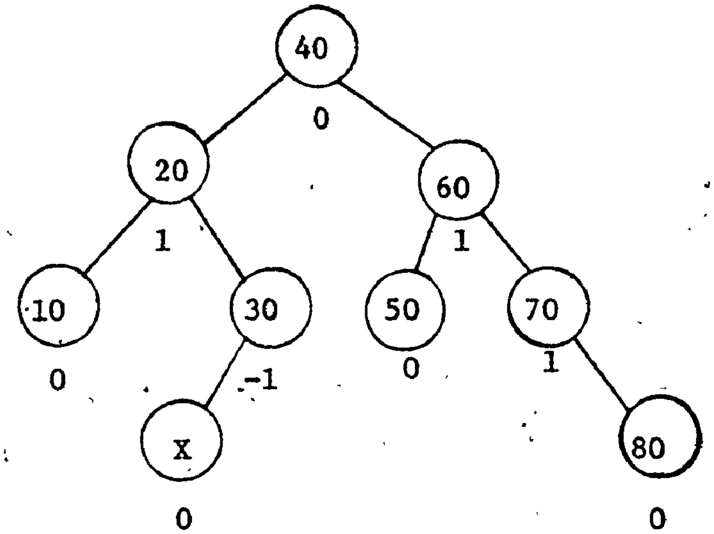
Figure 3.3.6



$i = 2$, $key(u_i) = 25$, $balfac(u_i) = 2$, $balfac(u_{i+1}) = -1$.

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

Figure 3.3.7

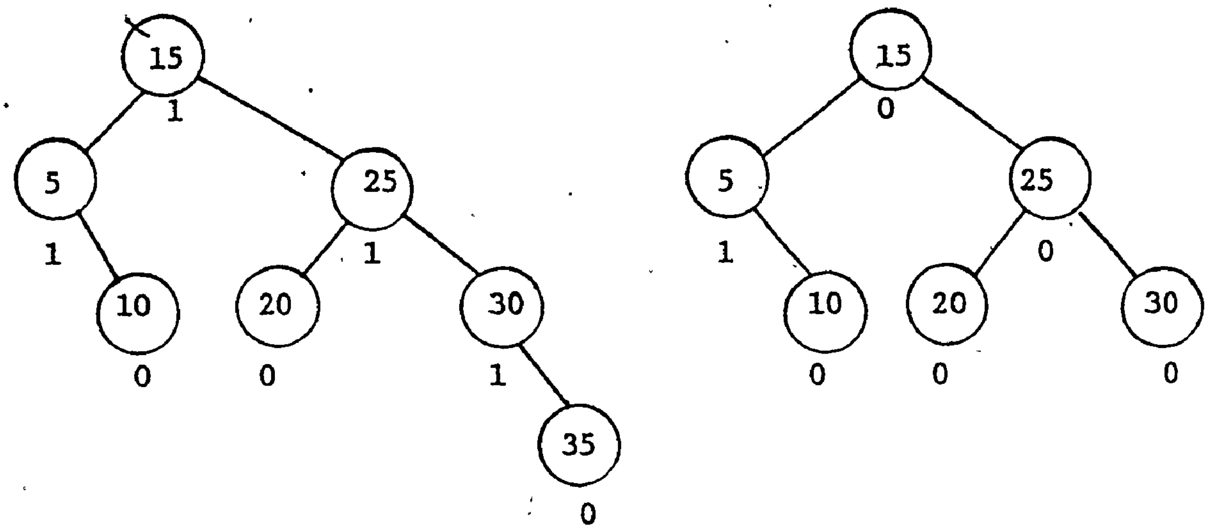


$k = 4$, $key(u_1) = 40$, $key(u_2) = 20$, $key(u_3) = 30$, $key(u_4) = X$

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

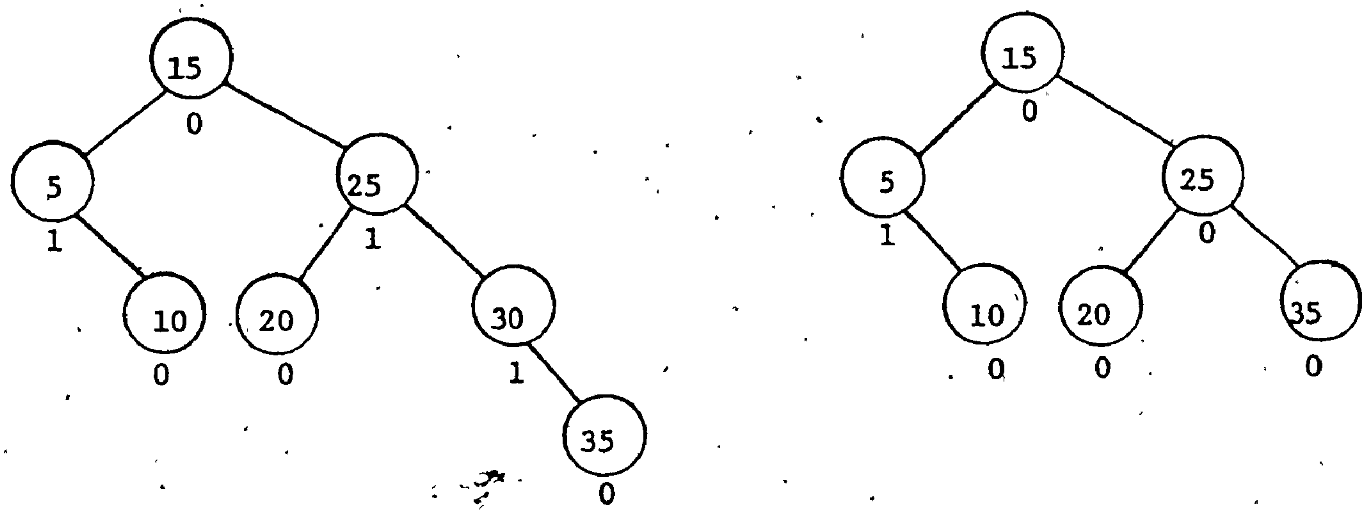
$balfac(u_4) = 0$.

Figure 3.3.8



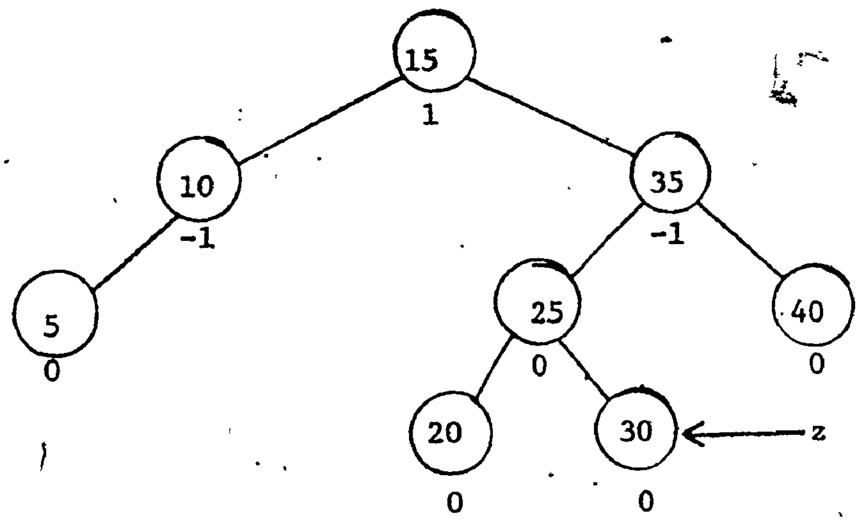
$k = 4$, $\text{key}(u_k) = 35$

Figure 3.3.9

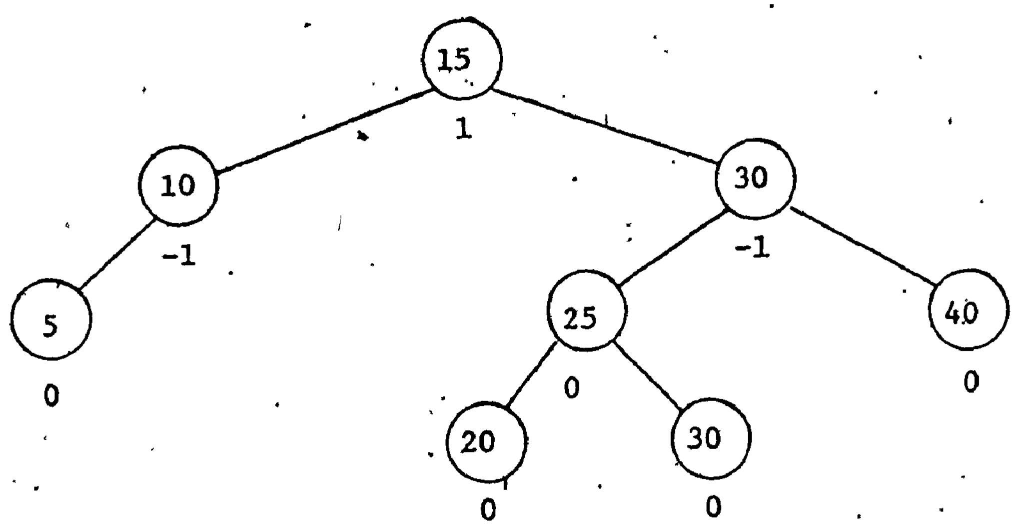


$k = 3$, $\text{key}(u_k) = 30$

Figure 3.3.10

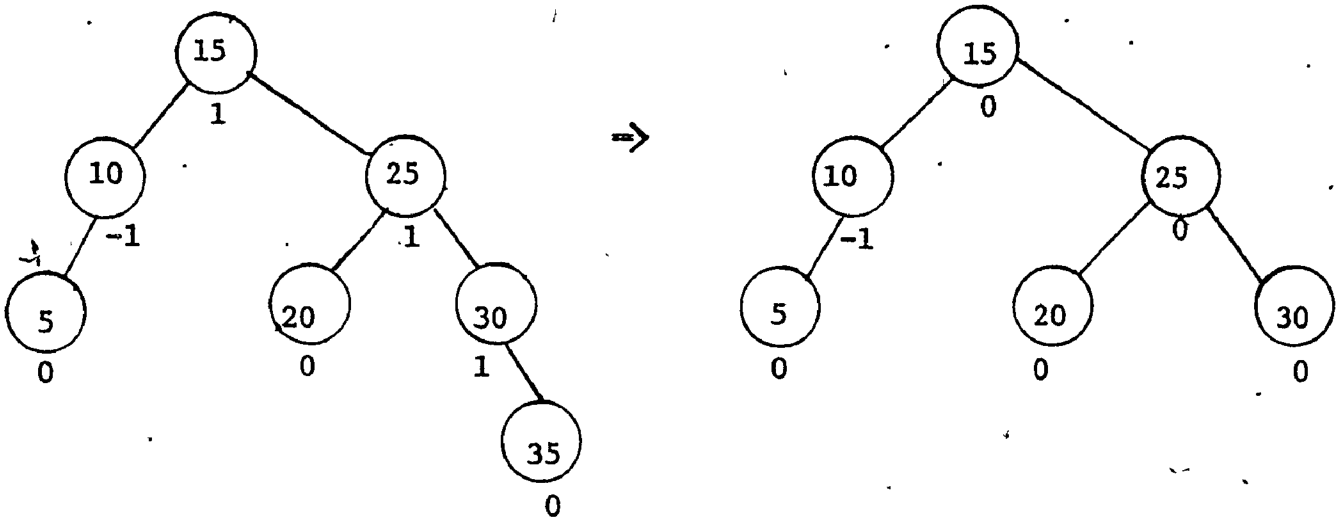


$k = 2$, $key(u_k) = 35$, $balfac(u_k) = -1$, $key(z) = 30$, z is the postorder predecessor of u_k.



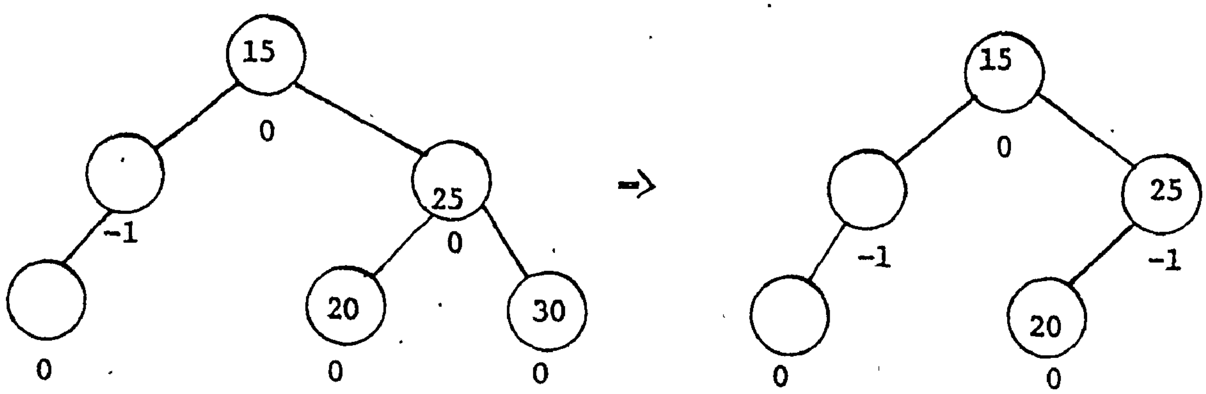
$k = 4$, $key(u_1) = 15$, $key(u_2) = 30$, $key(u_3) = 25$, $key(u_4) = 30$.

Figure 3.3.11



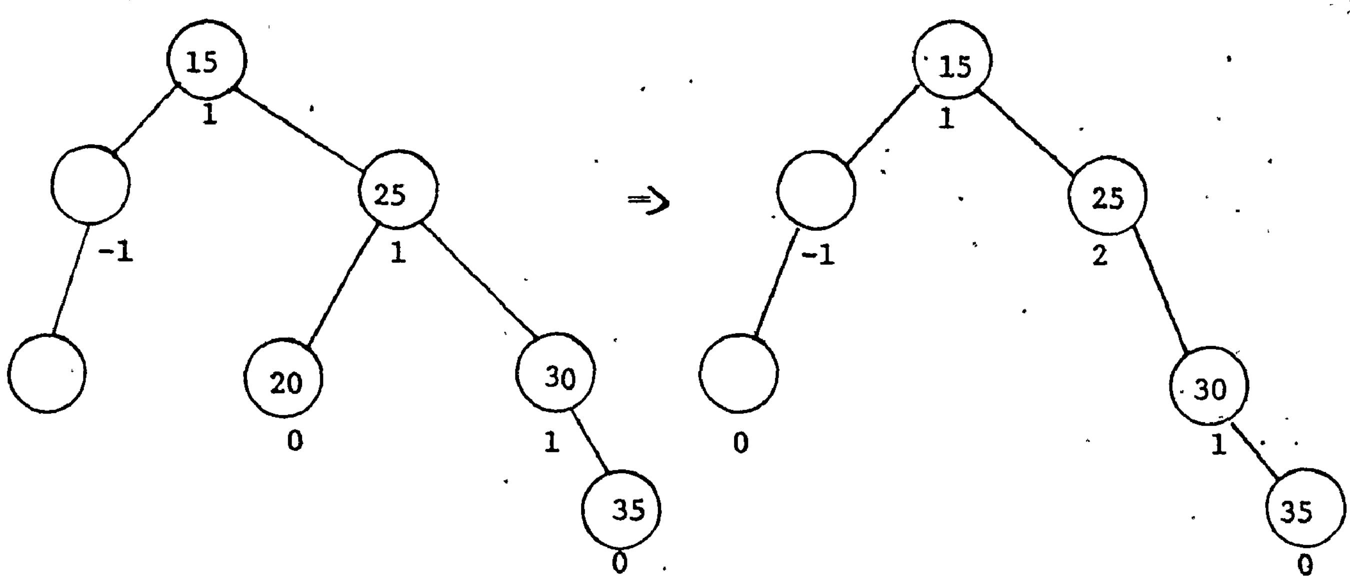
k = 4, key(u_k) = 35, ℓ = 3, key(u_ℓ) = 30, balfac(u_ℓ) = 0. The height of tree u_ℓ has been decremented.

Figure 3.3.12



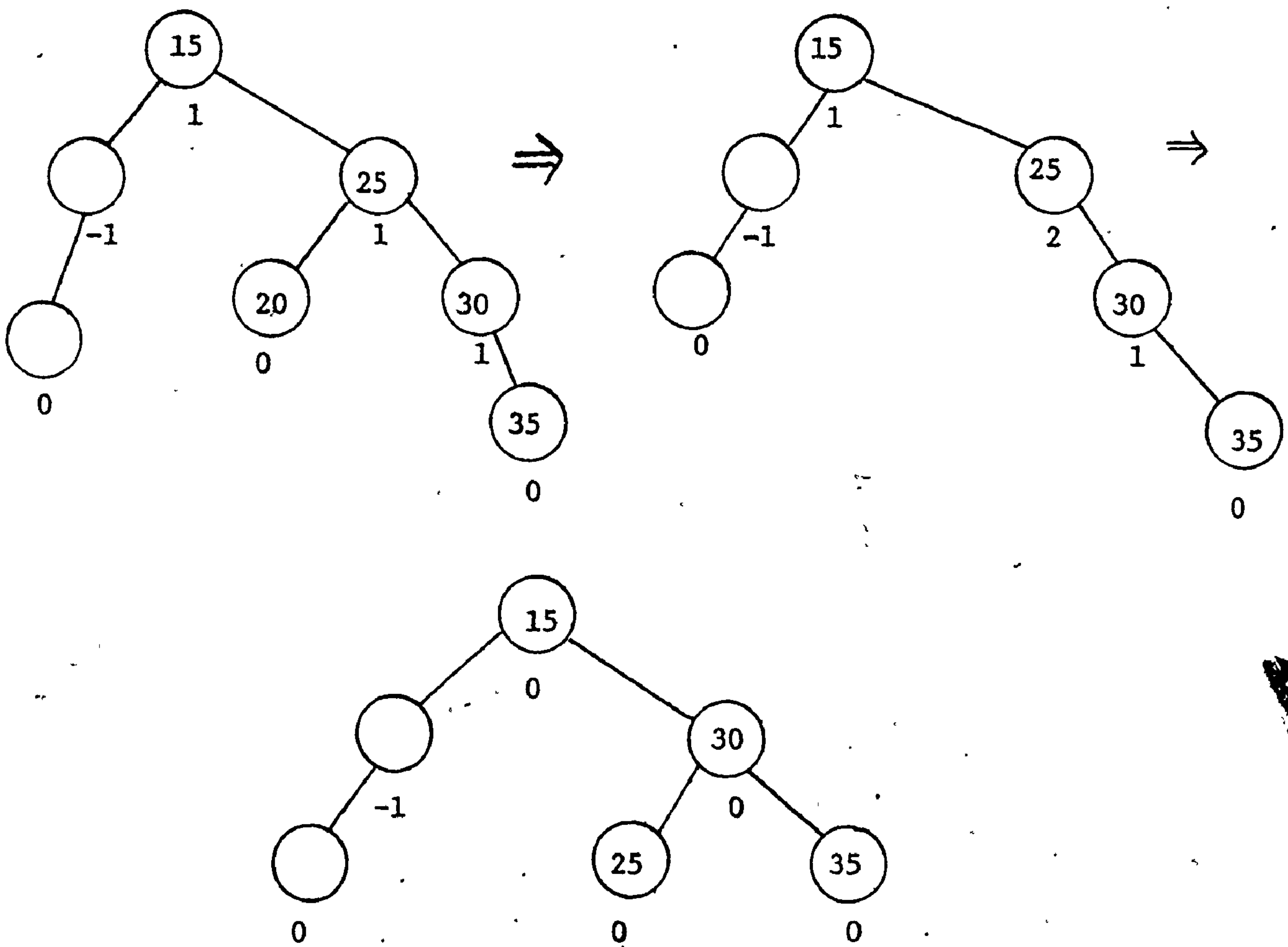
$k = 3$, $key(u_k) = 30$, $\ell = 2$, $key(u_\ell) = 25$, $balfac(u_\ell) = -1$. The height of $tree(u_\ell)$ is not reduced.

Figure 3.3.13



$k = 3$, $key(u_k) = 20$, $\ell = 2$, $key(u_\ell) = 25$, $balfac(u_\ell) = 2$. Rebalancing necessary.

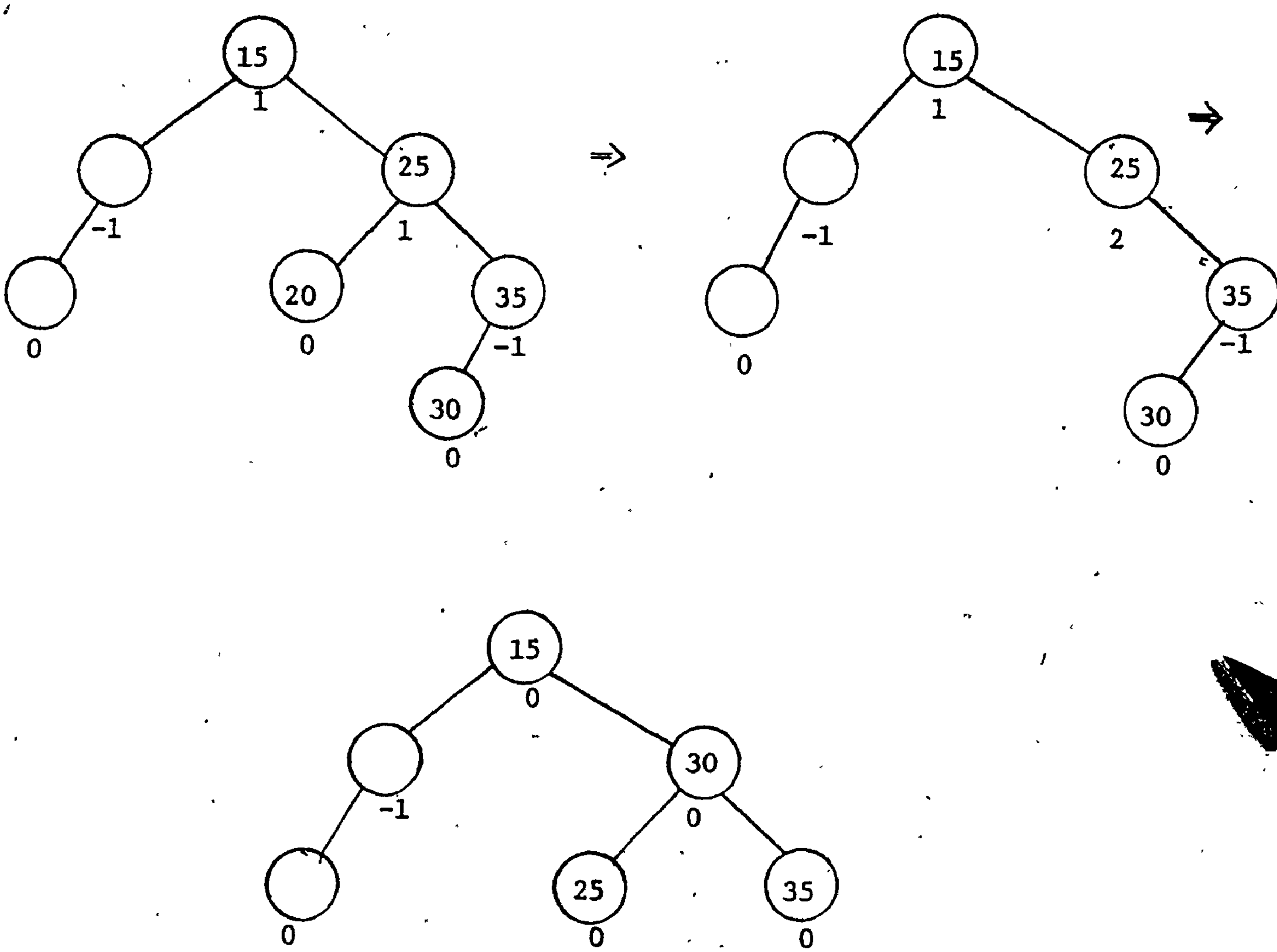
Figure 3.3.14



$k = 3$, key$(u_k) = 20$, $\ell = 2$, key$(u_\ell) = 25$, $u_{\ell+1} = \emptyset$, key$(v) = 30$

balfac$(u_\ell) = 2$, balfac$(v) = 1$. A single rotation is performed.

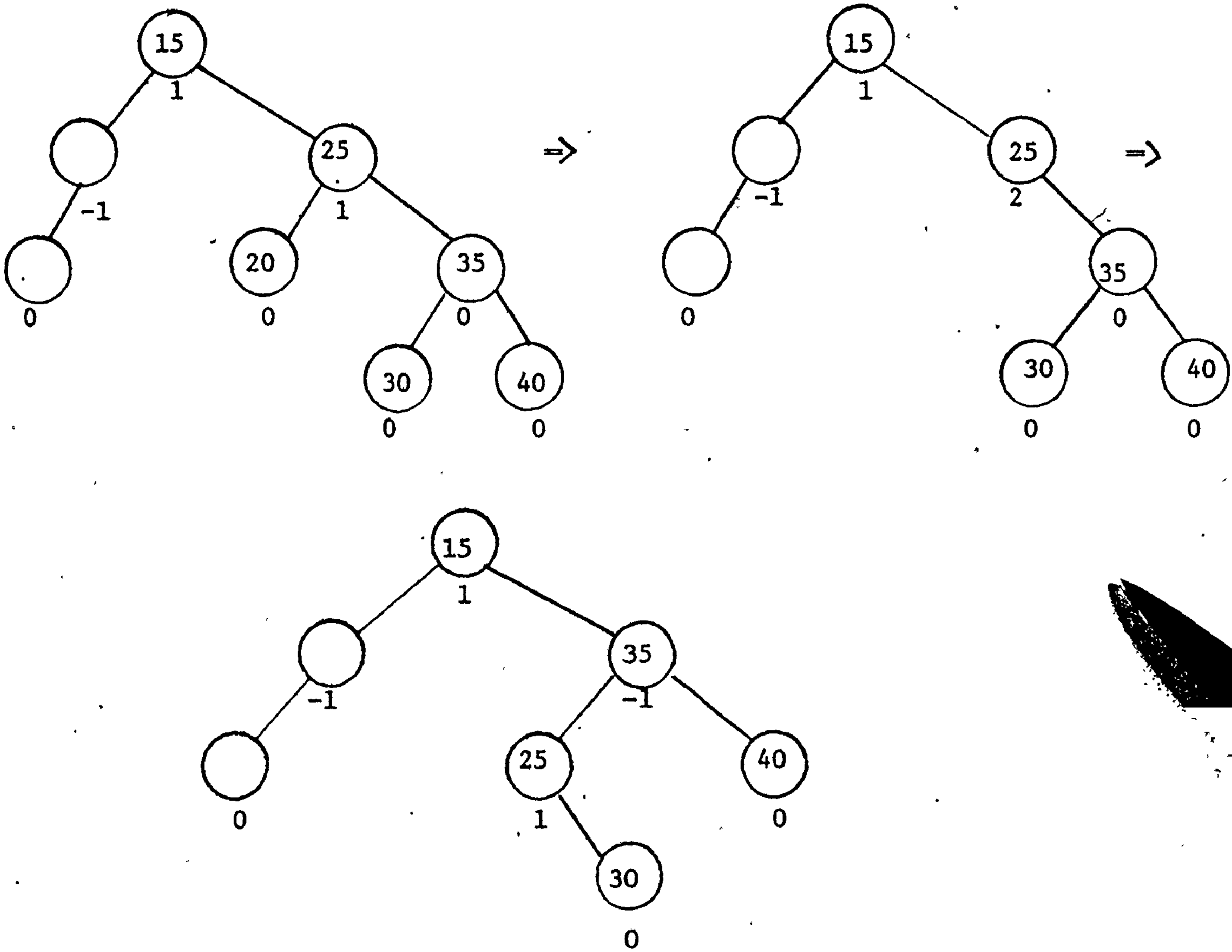
Figure. 3.3.15



$k = 3$, $key(u_k) = 20$, $\ell = 2$, $key(u_\ell) = 25$

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

Figure 3.3.16



$k = 3$, key$(u_k) = 20$, $\ell = 2$, key$(u_\ell) = 25$, $u_{\ell+1} = \emptyset$, key$(v) = 35$, balfac$(u_\ell) = 2$, balfac$(v) = 0$. A single rotation rebalances the tree; the rotated subtree has not decreased in height.

Figure 3.3.17

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 $\neq \emptyset$, then the root-balance of any node v in tree(u), RB(v), is defined as $RB(v) = \dfrac{size(v_\ell) + 1}{size(v) + 1}$. A tree, tree(u), is said to be of bounded balance α or in BB[α], $0 < \alpha \leq \frac{1}{2}$, 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[α].

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[$\frac{1}{2}$]. 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).

<u>Theorem 3.4.1</u>

For all α 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_ℓ) 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 α.

<u>Theorem 3.4.2</u>

If $0 < \alpha \leq 1 - \frac{\sqrt{2}}{2}$ 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_ℓ) then if $1-RB(v_\ell) < \frac{1-2\alpha}{1-\alpha}$, 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 α may be increased but since

(a) $1 - \frac{\sqrt{2}}{2} \simeq .2928$, (b) $BB[\alpha] - BB[\frac{1}{2}]$ is empty for $\frac{1}{3} < \alpha < \frac{1}{2}$, (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 α 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 α, $0 < \alpha \leq 1 - \frac{\sqrt{2}}{2}$ 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 $RB(v_r) < \frac{1-2\alpha}{1-\alpha}$, then a single rotation is performed, otherwise a double rotation is used (see Figure 3.4.6).

(b) X is in tree(v_ℓ). If $1 - RB(v_\ell) < \frac{1-2\alpha}{1-\alpha}$, 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[α], the parameter α, $0 < \alpha \leq 1 - \frac{\sqrt{2}}{2}$, 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, <u>the deletion sequence</u>, be $u_1, \ldots, u_k$ where $u_1 = u$ and $X = key(u_k)$.

(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 $RB(u_k) = \frac{1}{2}$ 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 tree(z) is replaced by 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) $\alpha \leq RB(u_i) \leq 1-\alpha$.

u_i 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) $RB(u_i) < \alpha$ or $RB(u_i) > 1-\alpha$.

u_i 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_i then let v be the right son of u_i. If $RB(v) < \frac{1-2\alpha}{1-\alpha}$, 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_i then let v be the left son of u_i. If $1 - RB(v) < \frac{1-2\alpha}{1-\alpha}$, 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_i, 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, $\frac{1}{4} < \alpha \leq 1 - \frac{\sqrt{2}}{2}$, then difficulties arise when $\text{size}(u_i) = 2$. 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 $\text{size}(u_i) = 2$ 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) $\text{key}(u_i) < X < \text{key}(u_{i+1})$ or (2) $\text{key}(u_{i+1}) < X < \text{key}(u_i)$. 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_i. If $\frac{1}{\text{size}(u_i) + 2} \geq \alpha$ then $\text{tree}(u_i)$ will be in $BB[\alpha]$ after the insertion and therefore the key may be simply inserted in $\text{tree}(u_i)$ 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.

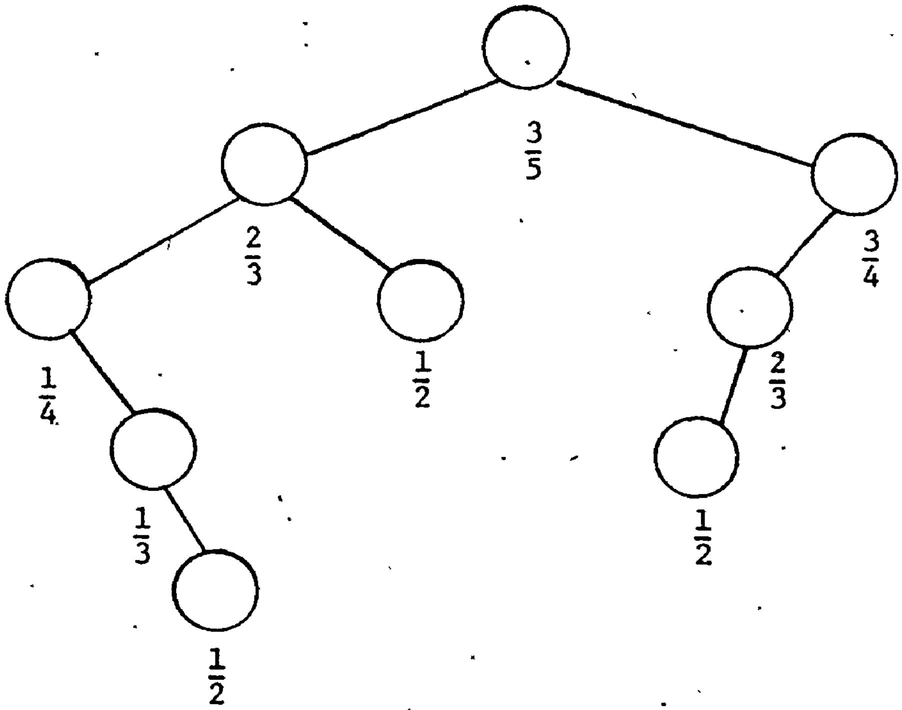
The deletion case is similar. For any u, if a node is to be deleted from tree(u_ℓ) then this may be treated as an insertion of a key into tree(u_r) and appropriate rotations invoked if necessary before any actual deletion. Here the size fields of the nodes in the deletion sequence are decremented as 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

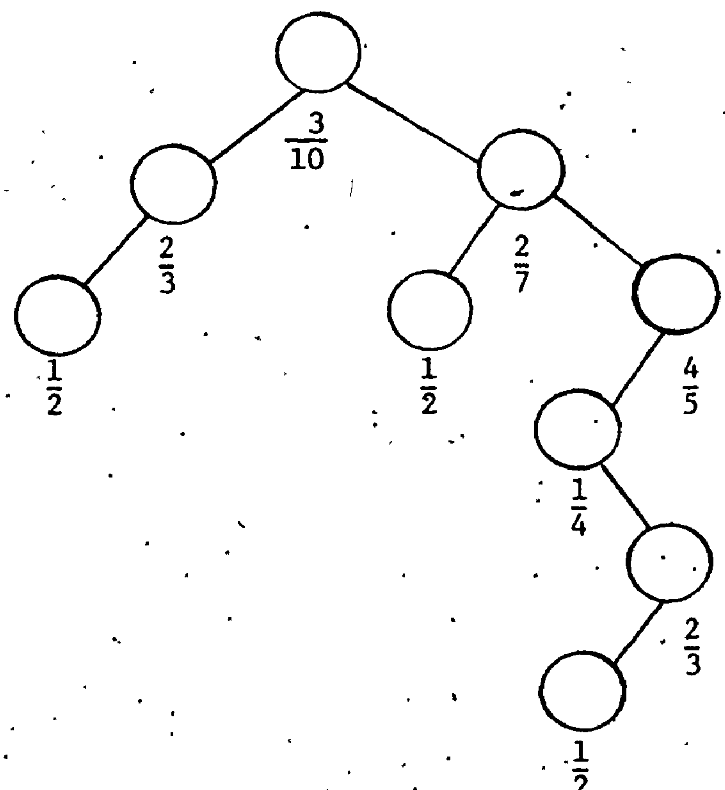
if $\alpha = \frac{a}{b}$ then $\alpha \leq RB(v) \leq 1-\alpha$ is $\frac{a}{b} \leq \frac{\text{size}(v_\ell) + 1}{\text{size}(v) + 1} \leq 1 - \frac{a}{b}$ which

becomes $a \cdot [\text{size}(v) + 1] \leq b \cdot [\text{size}(v_\ell) + 1] \leq (b-a) \cdot [\text{size}(v) + 1]$.

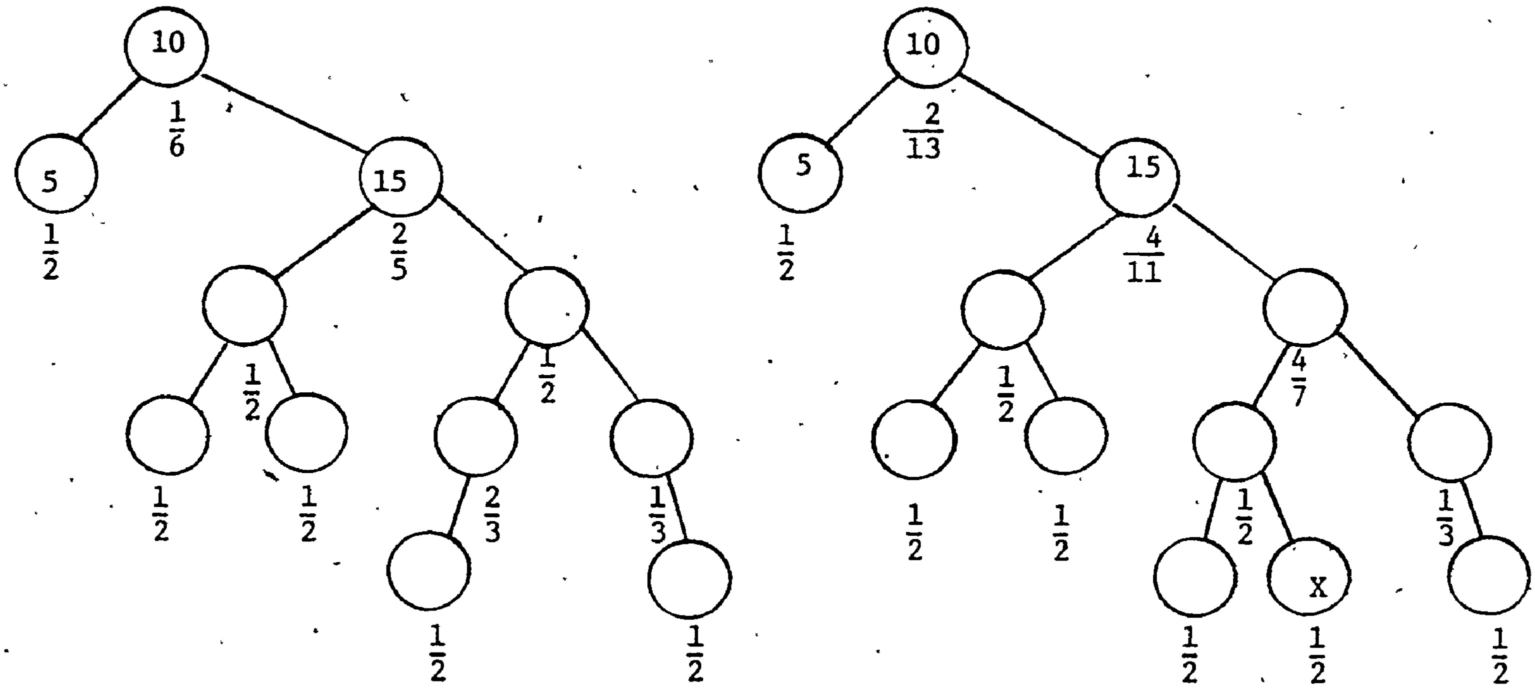
This avoids division and comparison of real quantities which may cause difficulty due to rounding error.



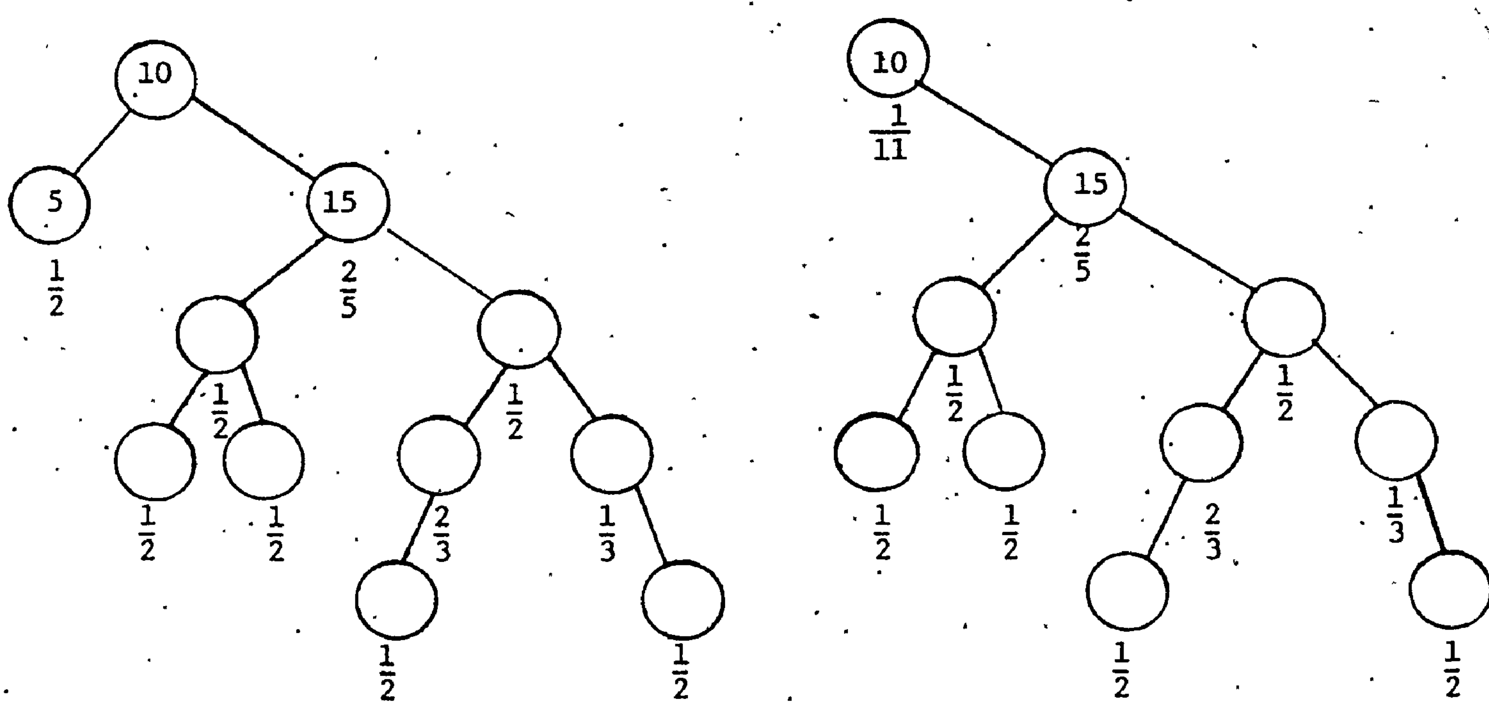
A tree in BB[$\frac{1}{4}$]



A tree in BB[$\frac{1}{7}$]

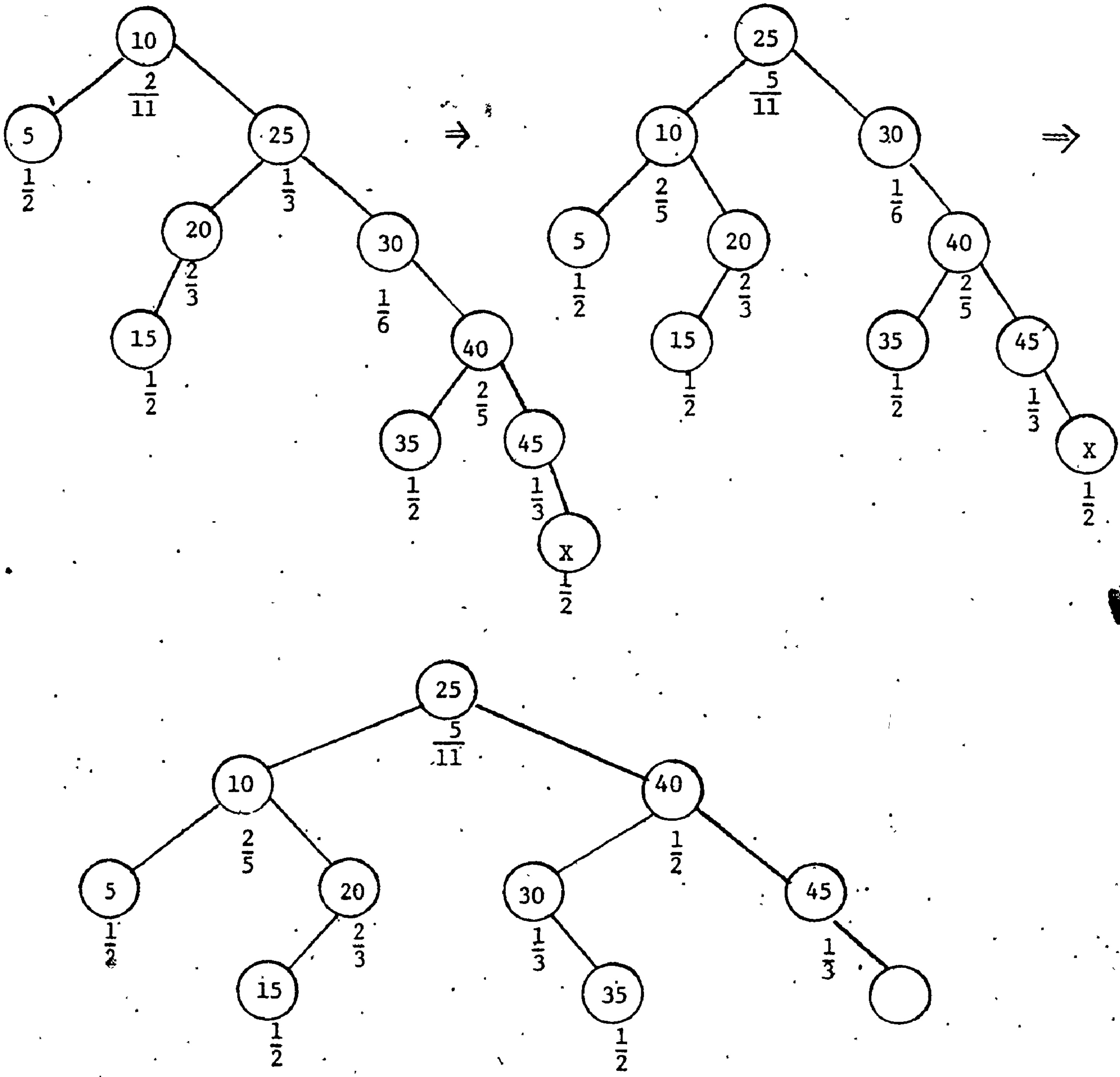


$\alpha = \frac{1}{6}$, key(v) = 10, key(v_r) = 15. Addition of node with key X in tree(v_r) causes unbalance. RB(v) = $\frac{2}{13} < \alpha$.



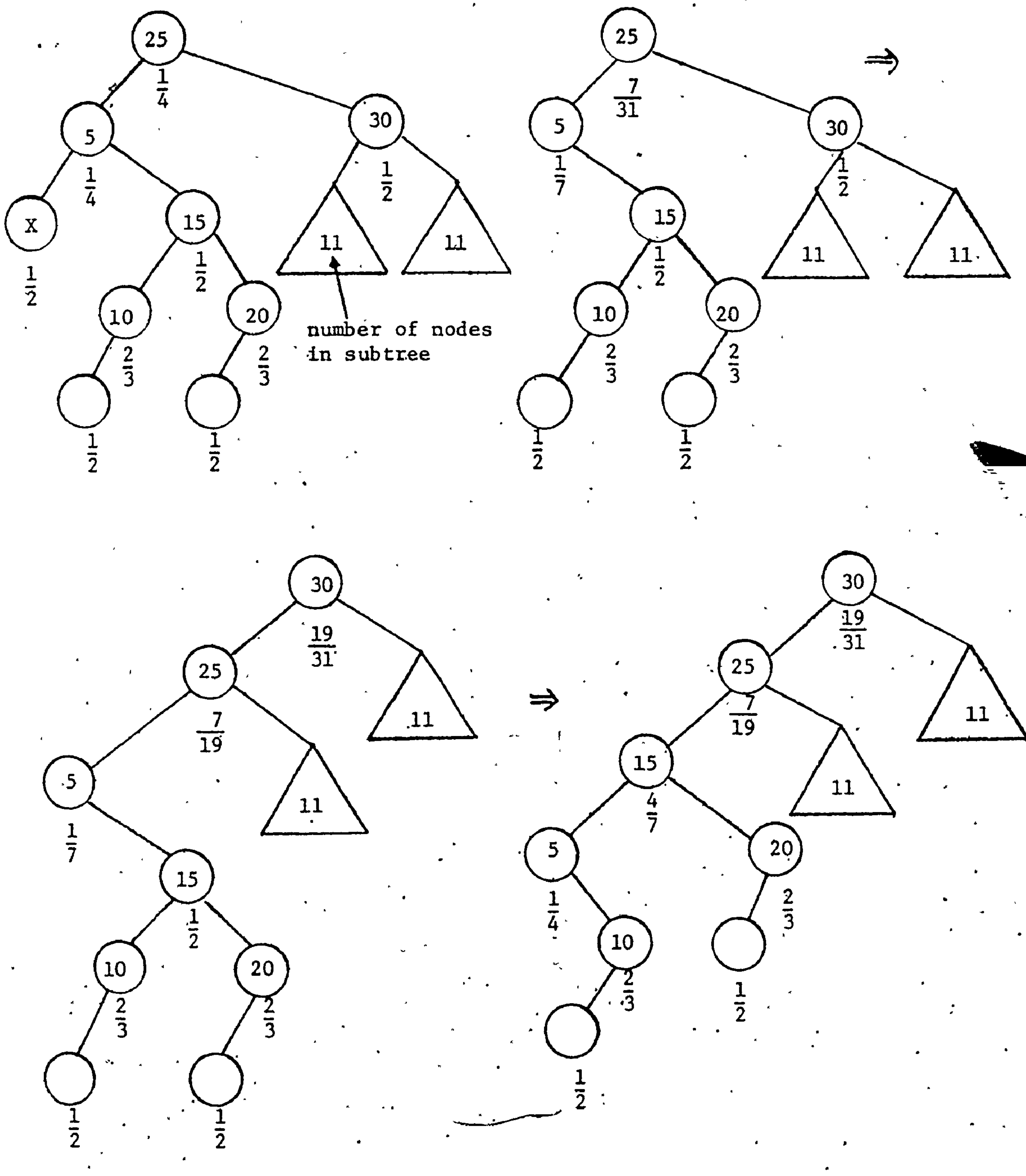
$\alpha = \frac{1}{6}$, key(v) = 10, key(v_ℓ) = 5. Deletion of node with key 5 causes unbalance. RB(v) = $\frac{1}{11} < \alpha$.

<u>Figure 3.4.2</u>

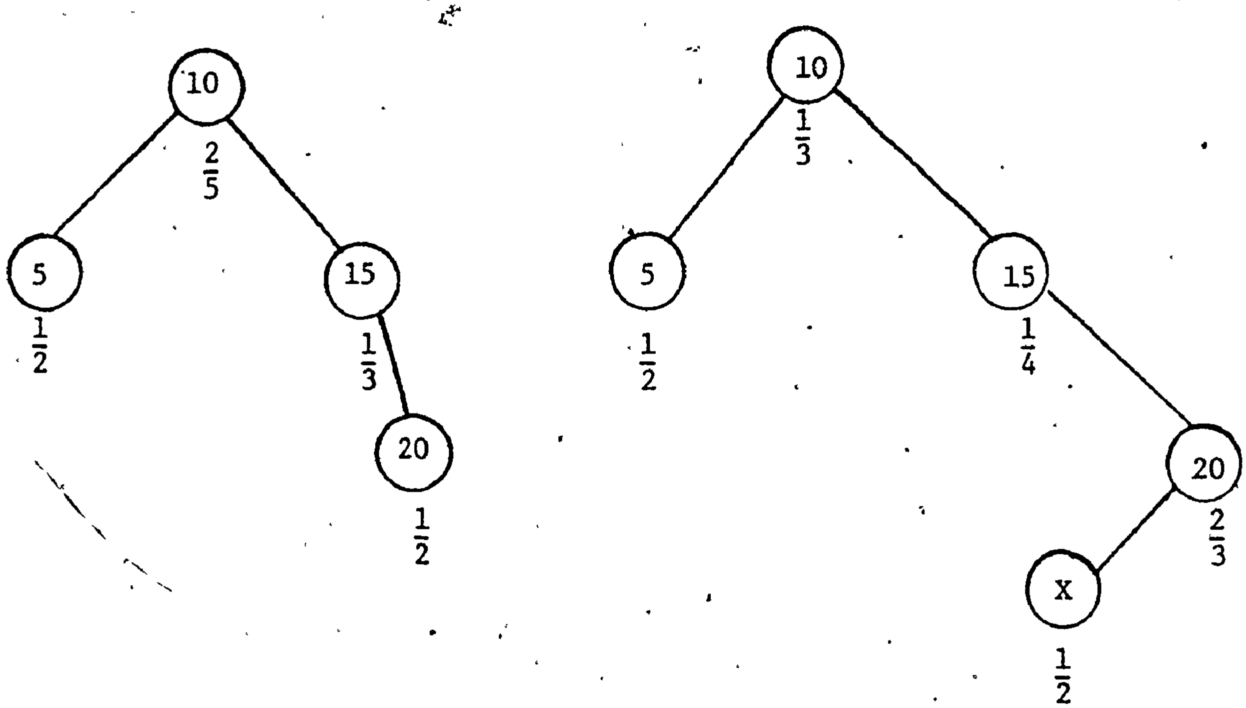


$\alpha = \frac{1}{5}$. Insertion of X requires two single rotations to restore balance. Nodes with keys 10 and 30 are unbalanced upon insertion of X.

<u>Figure 3.4.3</u>

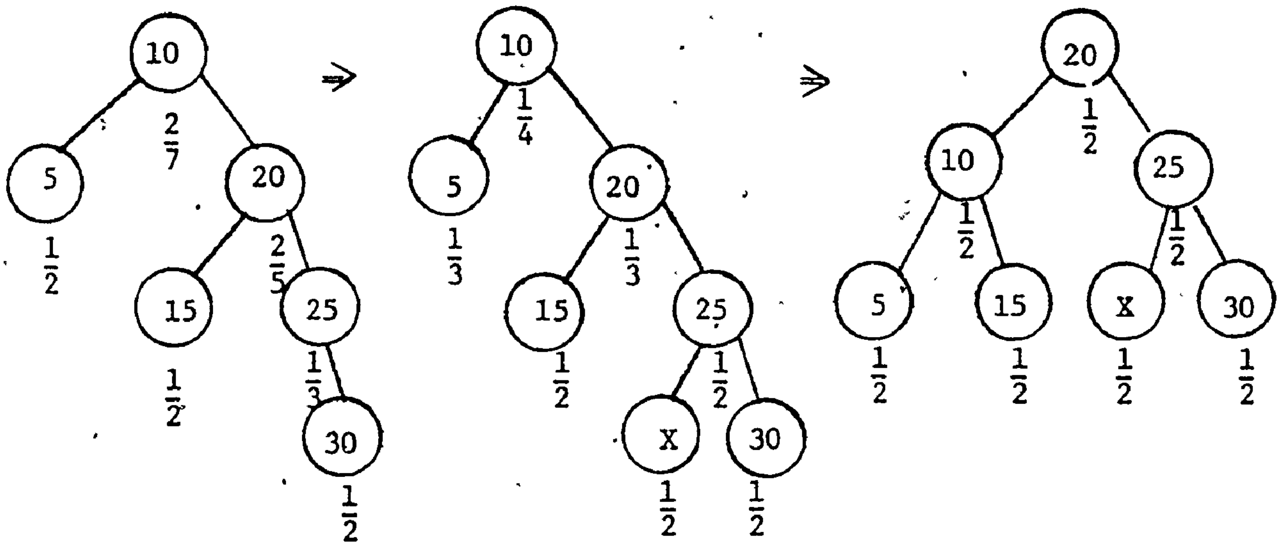


$\alpha = \frac{1}{4}$. Deletion of node with key X requires two single rotations to restore balance. Nodes with keys 25 and 5 are unbalanced.

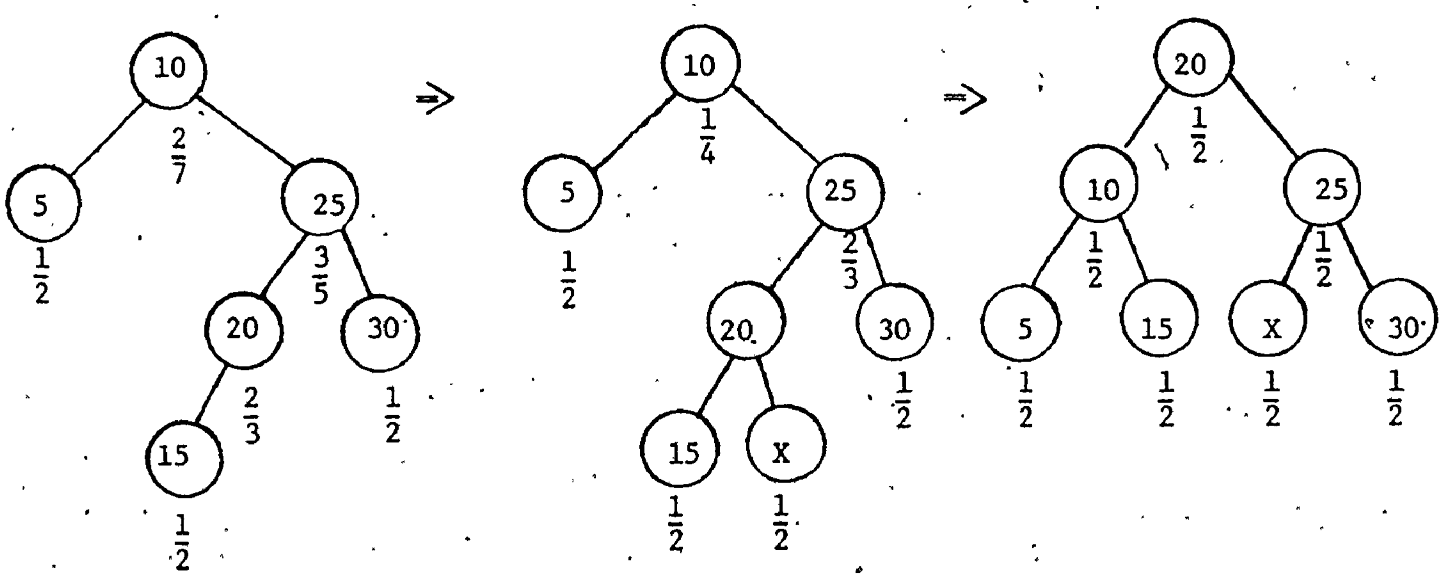


$\alpha = \frac{1}{7}$, key(v) = 10. Insertion of X does not cause unbalance of v.

Figure 3.4.5

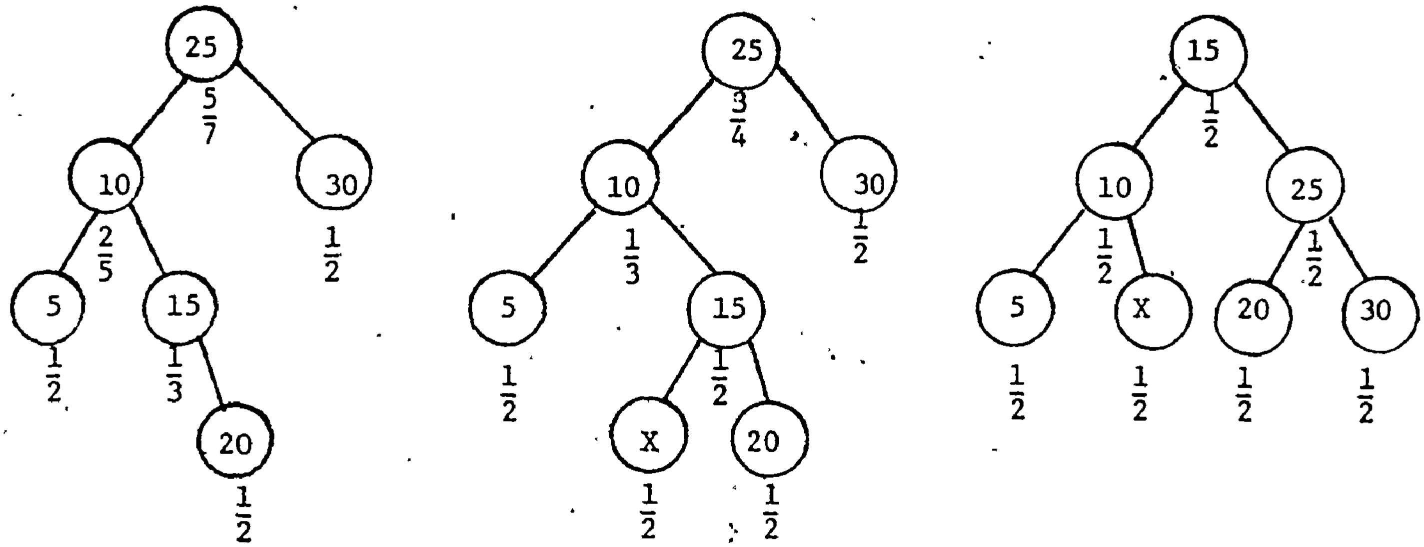


$\alpha = \frac{2}{7}$, key(v) = 10, key(v_r) = 20. Insertion of X causes unbalance at v, RR(v) < α, RB(v_r) < $\frac{1-2\alpha}{1-\alpha}$ causing a single rotation.

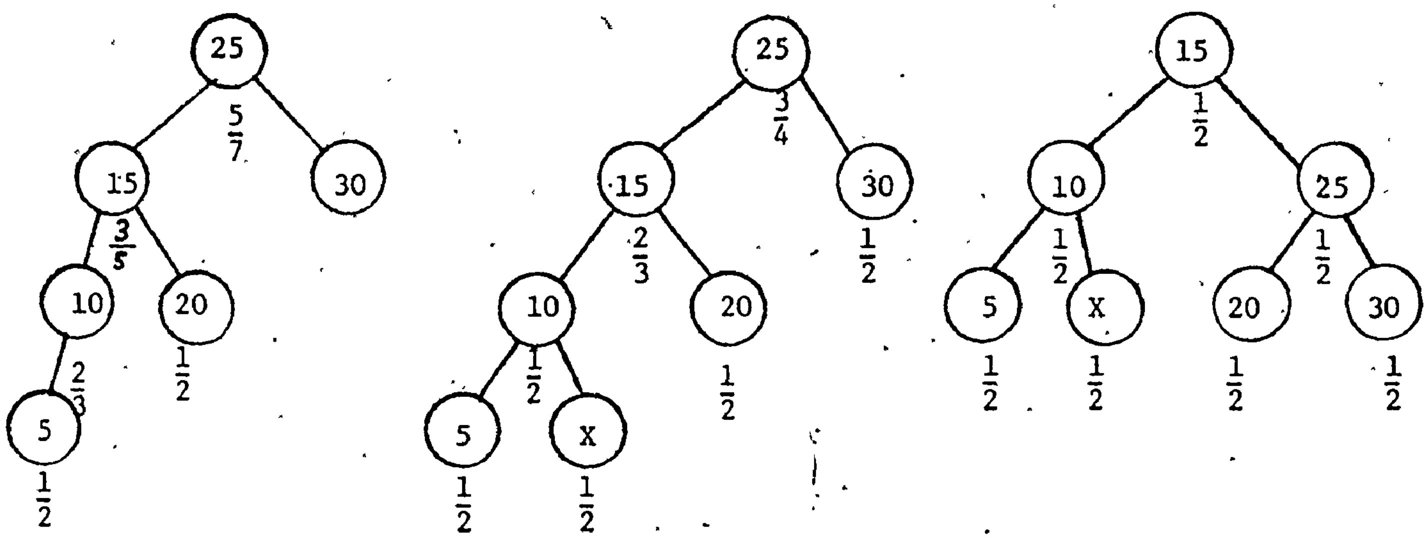


$\alpha = \frac{2}{7}$, key(v) = 10, key(v_r) = 25. Insertion of X causes unbalance at v, RB(v) < α, RB(v_r) $\not<$ $\frac{1-2\alpha}{1-\alpha}$ causing a double rotation.

Figure 3.4.6

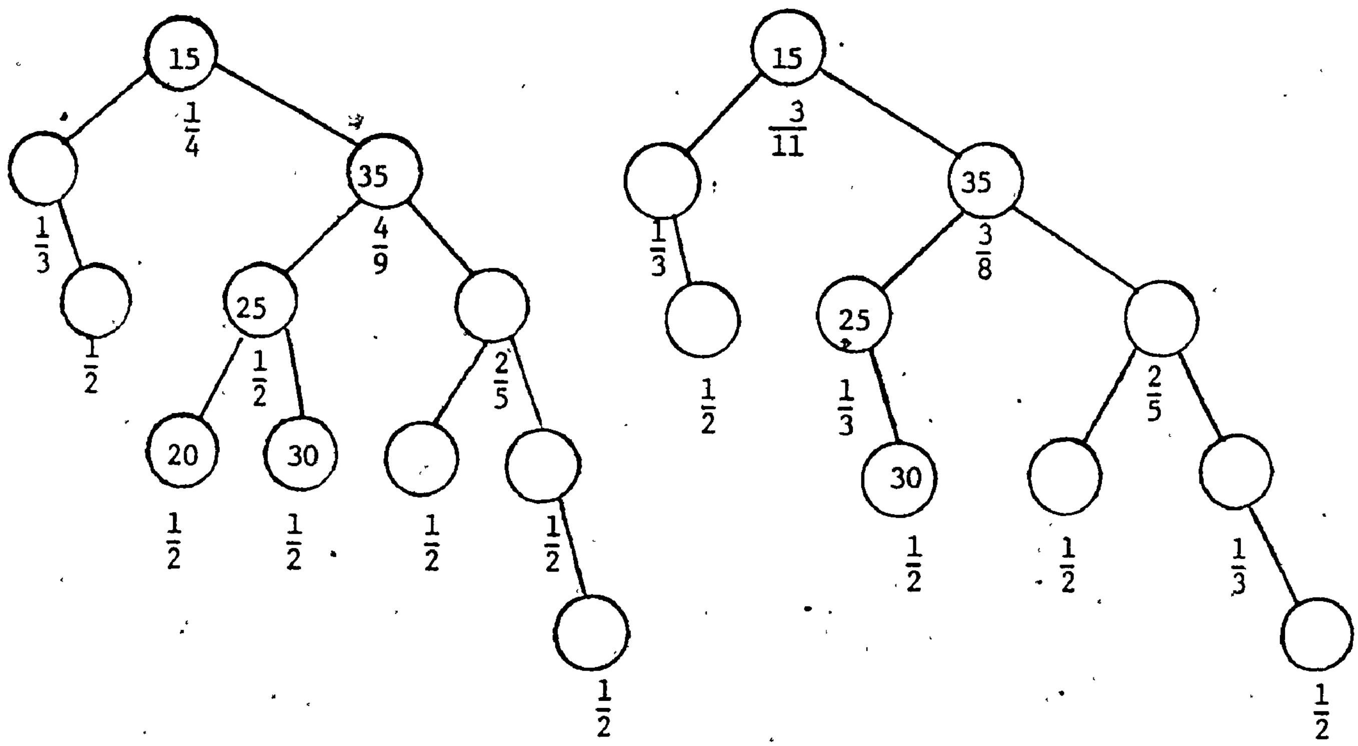


$\alpha = \frac{2}{7}$, key(v) = 25, key(v_ℓ) = 10. Insertion of X causes unbalance at v, RB(v) > $1-\alpha$, $1 - RB(v_\ell) \not< \frac{1-2\alpha}{1-\alpha}$ causing a double rotation.



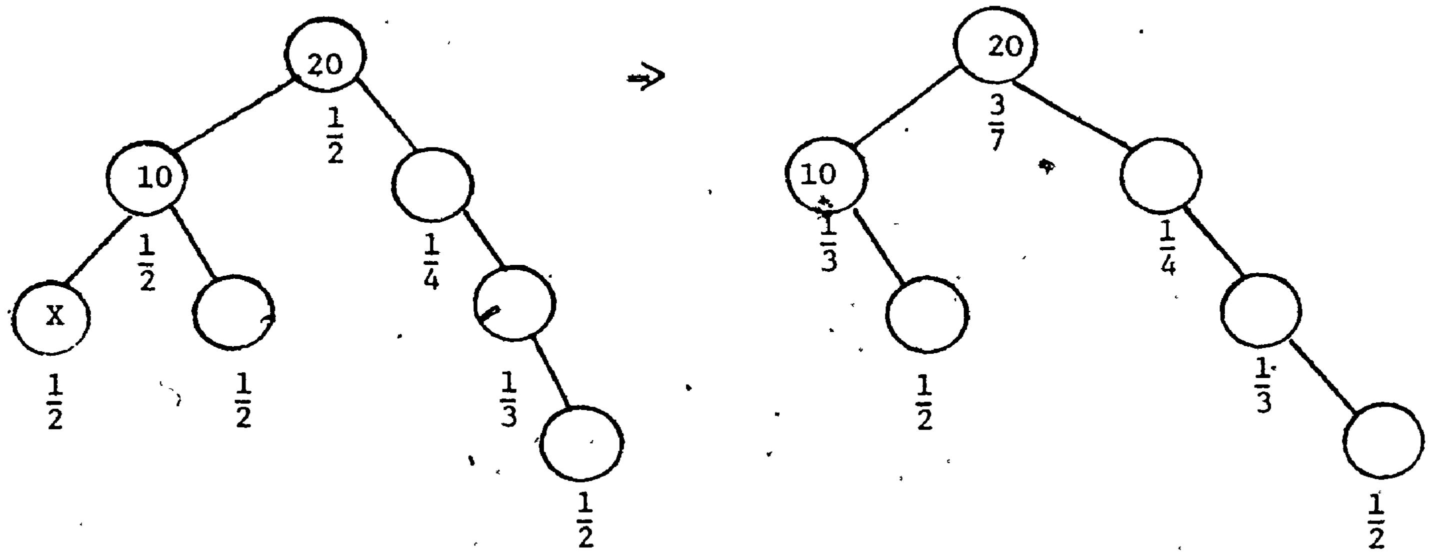
$\alpha = \frac{2}{7}$, key(v) = 25, key(v_ℓ) = 15. Insertion of X causes unbalance at v, RB(v) > $1-\alpha$, $1 - RB(v_\ell) < \frac{1-2\alpha}{1-\alpha}$ causing a single rotation.

Figure 3.4.7



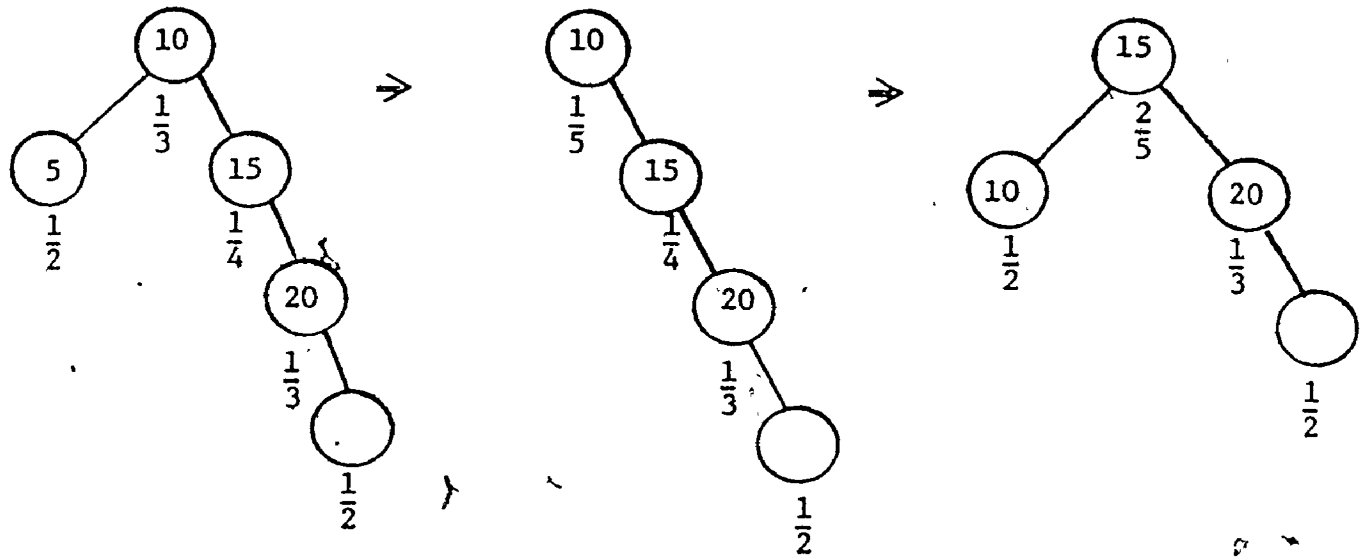
$\alpha = \frac{1}{4}$, key(u_k) = 15, key(z) = 20. Choosing z as the postorder successor improves the balance of u_k when z is deleted.

<u>Figure 3.4.8</u>



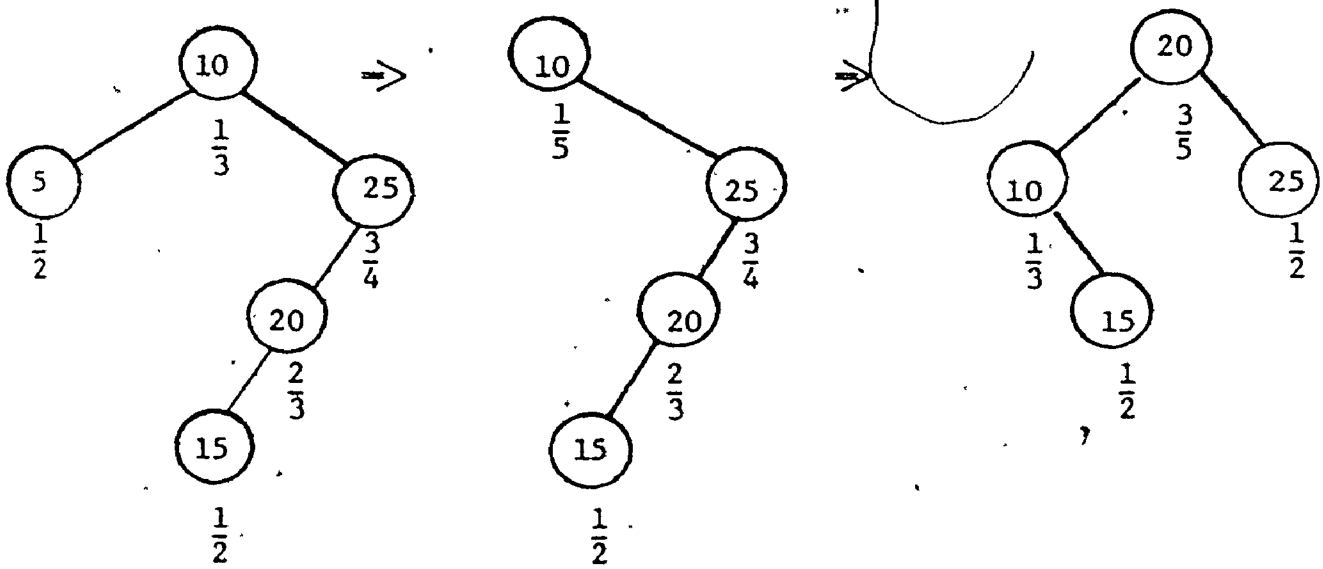
$\alpha = \frac{1}{4}$, key(u_k) = X, key(u_i) = 20. Deletion of u_k does not cause umbalance at u_i.

Figure 3.4.9



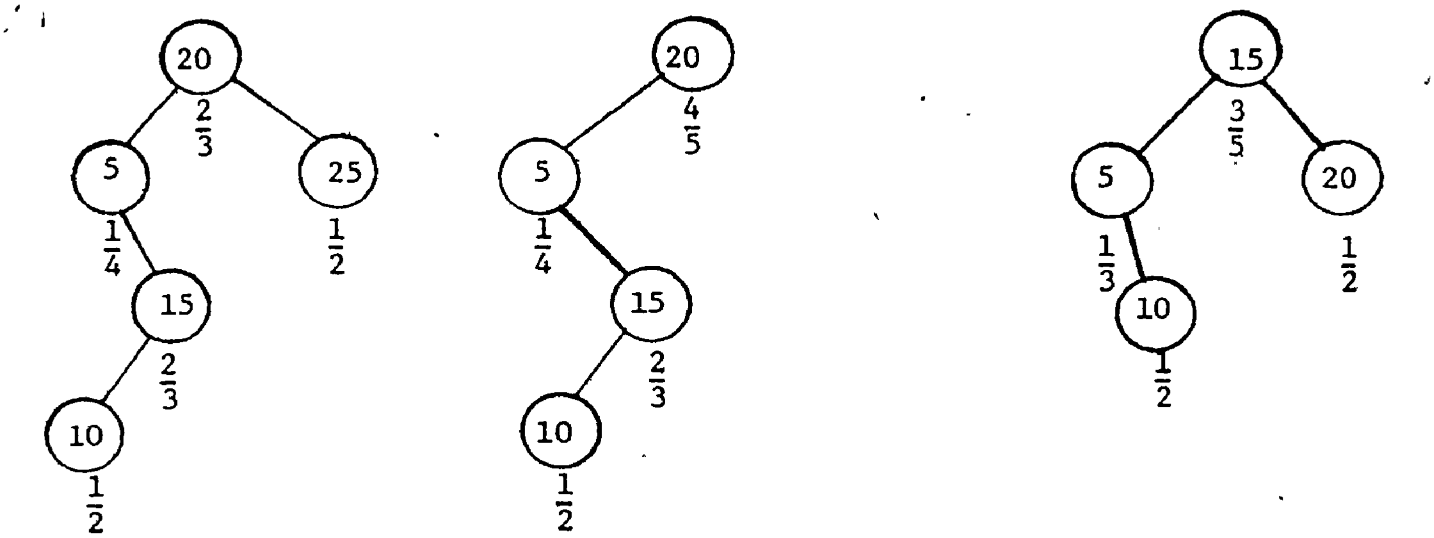
$\alpha = \frac{1}{4}$, key(u_k) = 5, key(u_i) = 10, key(v) = 15, RB$(u_i) < \alpha$, RB(v) $< \frac{1-2\alpha}{1-\alpha}$.

A single rotation restores balance.



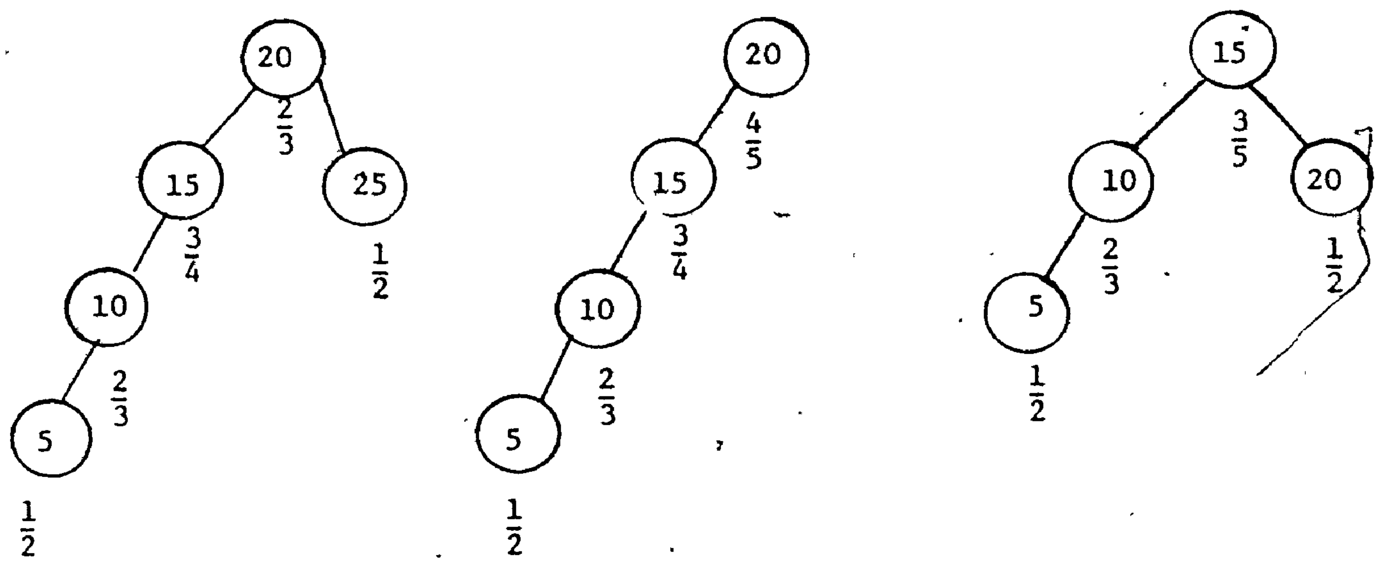
$\alpha = \frac{1}{4}$, key(u_k) = 5, key(u_i) = 10, key(v) = 25, RB$(u_i) < \alpha$, RB(v) $\not< \frac{1-2\alpha}{1-\alpha}$.

A double rotation restores balance.



$\alpha = \frac{1}{4}$, key$(u_k) = 25$, key$(u_i) = 20$, key$(v) = 5$, $RB(u_i) > 1-\alpha$,

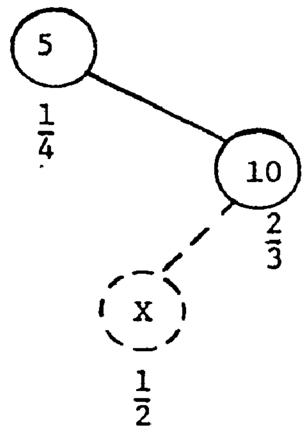
$1 - RB(v) \not< \frac{1-2\alpha}{1-\alpha}$. A double rotation restores balance.



$\alpha = \frac{1}{4}$, key$(u_k) = 25$, key$(u_i) = 20$, key$(v) = 15$, $RB(u_i) > 1-\alpha$,

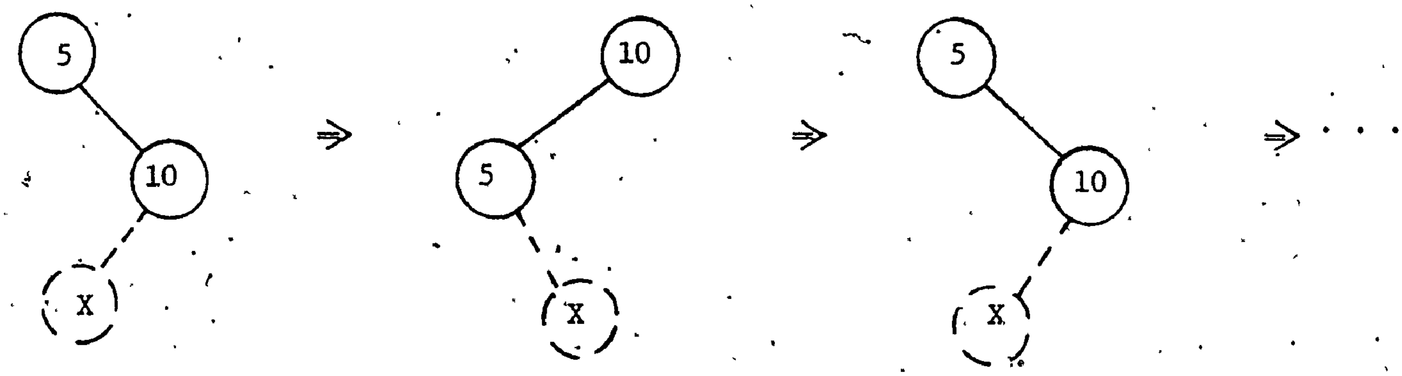
$1 - RB(v) < \frac{1-2\alpha}{1-\alpha}$. A single rotation restores balance.

Figure 3.4.11

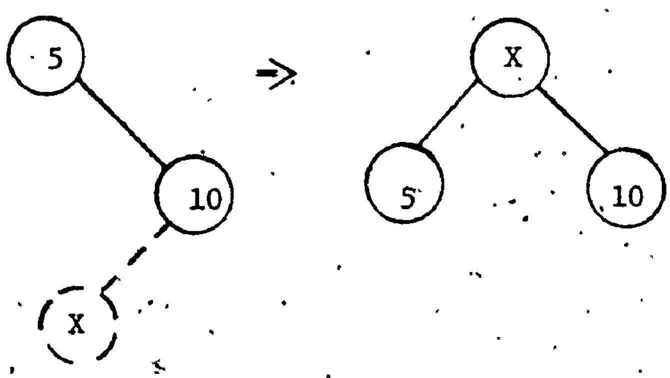


$\frac{1}{4} < \alpha \leq 1 - \frac{\sqrt{2}}{2}$, $key(u_i) = 5$, $key(u_{i+1}) = 10$, $size(u_i) = 2$, $RB(u_i) = \frac{1}{4} < \alpha$, $RB(u_{i+1}) \not< \frac{1-2\alpha}{1-\alpha}$. A double rotation cannot be performed since X is not in tree(u_i) yet.

Figure 3.4.12



$\frac{1}{4} < \alpha \leqslant 1 - \frac{\sqrt{2}}{2}$, $key(u_i) = 5, 10, \ldots$, $size(u_i) = 2$. An infinite number of single rotations result.



A special case for the implemented algorithm

Figure 3.4.13

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 techique 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 techique, 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 size(u) = 2^r-1, (ii) r-incomplete if size(u) < 2^r-1 and (iii) r-replete if 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 $\ell(v) \leq r$. For k > 0, tree(u) is a k-minimal tree iff for all v in tree(u) either (i) size(v) $\leq 2^k-1$ and tree(v) is minimal or (ii) 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 techique 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 $k' = \lfloor \log_2(1+\text{size}(u)) \rfloor$, 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

$u_1, \ldots, u_m$, $m \leq k-1$.

(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 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 $X = key(u_{k'})$, 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 $k\text{-size}(v_\ell) > k\text{-size}(v_r)$ 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 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 tree(u) is no longer a binary search tree, since key(z) occurs twice.

should be taken. For $i = 1,\ldots,k'-1$, if there is a $u_{k'-i}$ such that

(1) tree($u_{k'-i}$) is $(i+1)$-incomplete, then immediately we have that tree(u_1) is k -incomplete; thus $u_{k'}$ is removed (see Figure 3.5.9), or

(2) tree($u_{k'-i}$) is $(i+1)$-replete, then letting $u_{k'-i+1}$ and v be the sons of $u_{k'-i}$, if v is the left son let z be the postorder predecessor of key($u_{k'-i}$) otherwise let z be the postorder successor of key($u_{k'-i}$). A new deletion sequence $u'_1,\ldots,u'_{k'}$, is calculated where $u'_{k'} = z$. Replace key($u_{k'-i}$) with key(z) and then use the BIA to insert key($u_{k'-i}$) into tree($u_{k'-i+1}$). Step (ii) is now repeated with $u_1,\ldots,u_{k'}$ replaced by $u'_1,\ldots,u'_{k'}$ (see Figure 3.5.10).

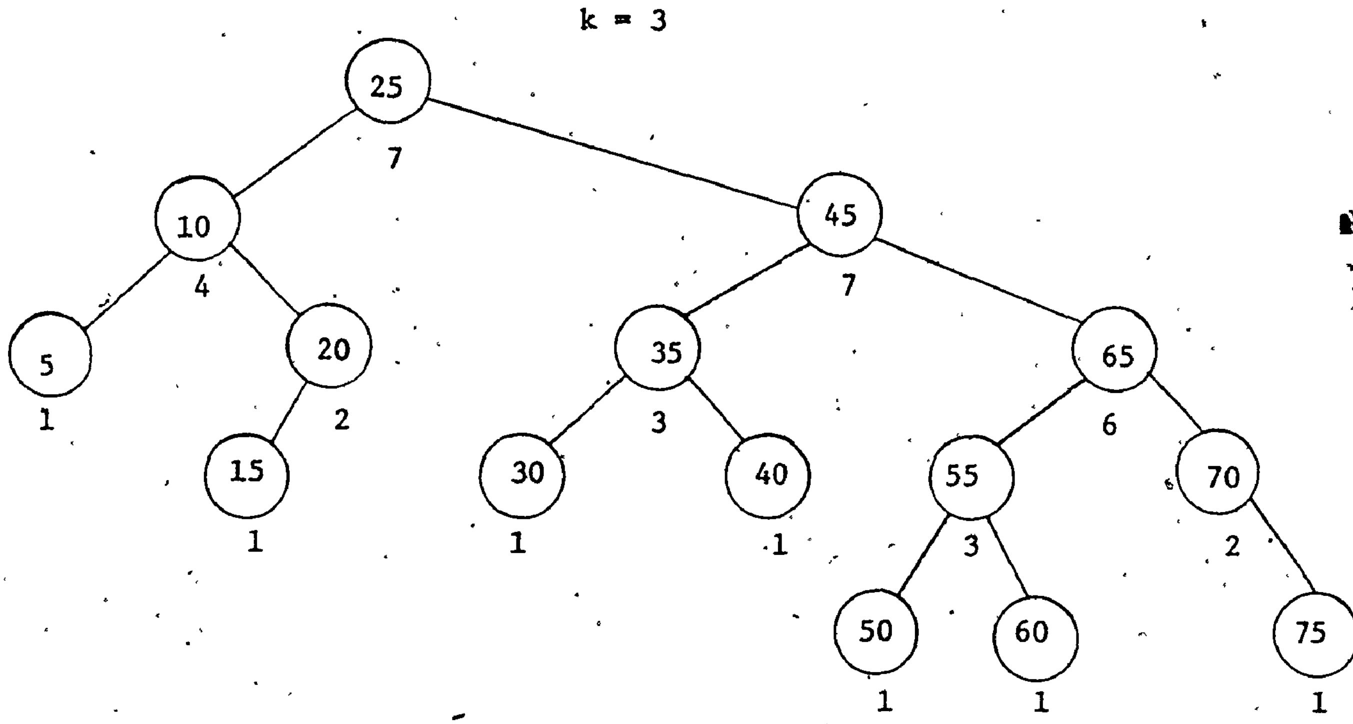
Otherwise if there is no $u_{k'-i}$ fulfilling conditions (1) or (2) then remove $u_{k'}$ (see Figure 3.5.11).

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 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 tree(u). This information is used to determine if for any v, tree(v) is ℓ-complete,

ℓ-replete or ℓ-incomplete; $1 \leq \ell \leq k$. In the insertion algorithm u_0 is determined by simply examining the size field of each node in the insertion sequence until size(v) < KVAL, 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 size$(u_i) = \left\lfloor \frac{\text{KVAL}}{2^i} \right\rfloor$, $i \geq 1$, 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.



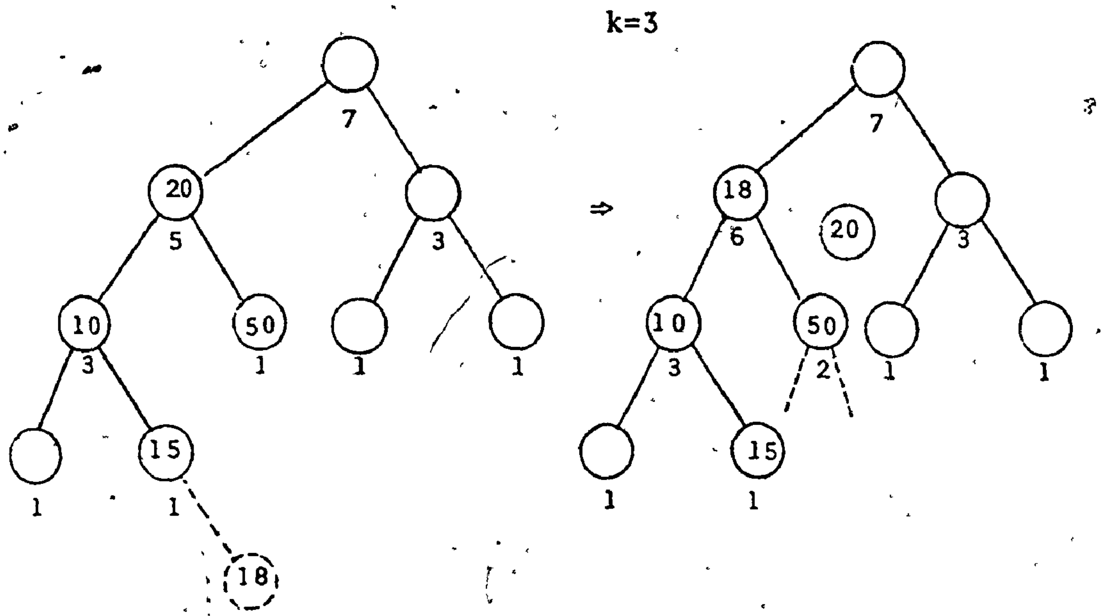
(a) Subtrees with keys 25 and 45 are 3-complete.

(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.

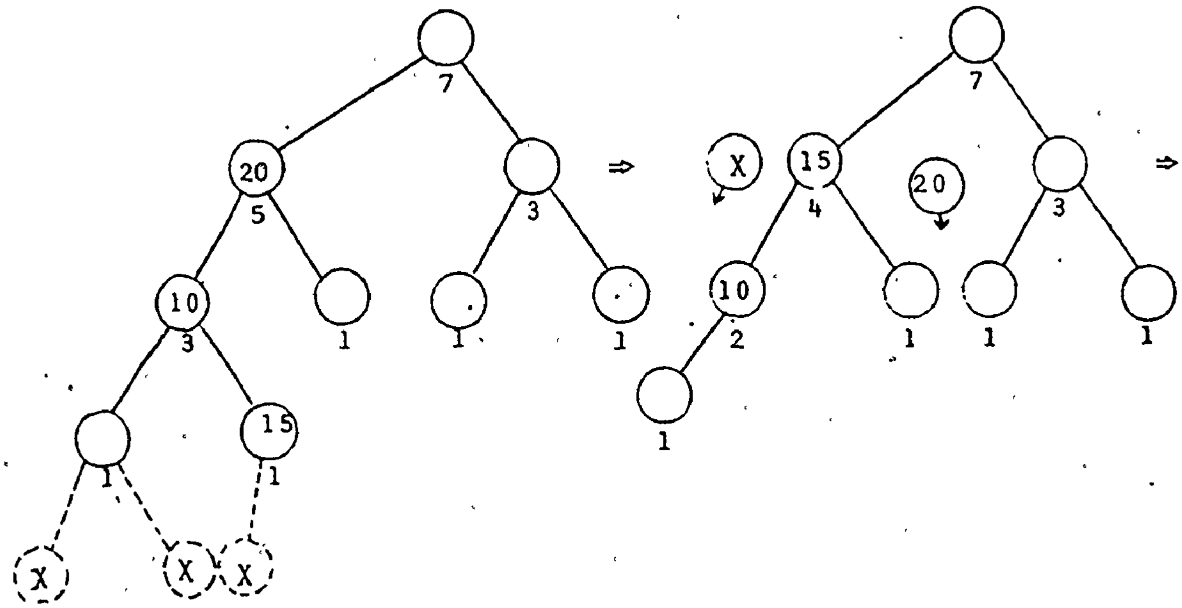
Figure 3.5.1



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).

Figure 3.5.2

k=3

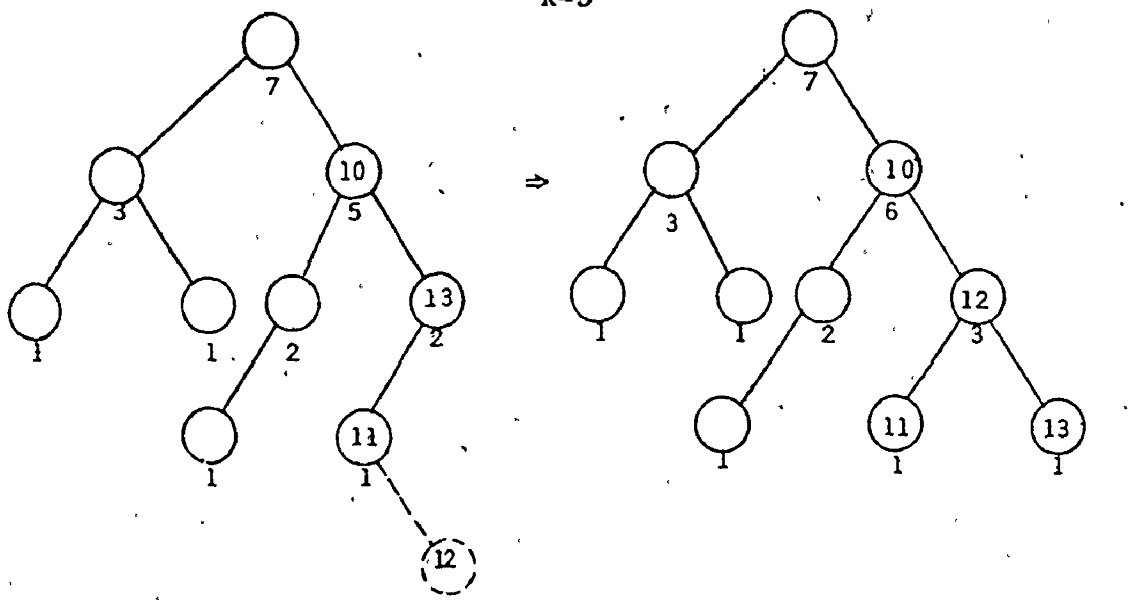


BIA for X, $i=1$, $key(u_0) = 20$, $key(u_1) = 10$, $key(z) = 15$, $X < 15$, and $tree(u_1)$ is a 2-complete tree.

After the transformation $key(u_{i-1}) = 15$, and BIA is called recursively to insert X into $tree(u_i)$ and 20 into $tree(u_{i-1})$.

Figure 3.5.3

k=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

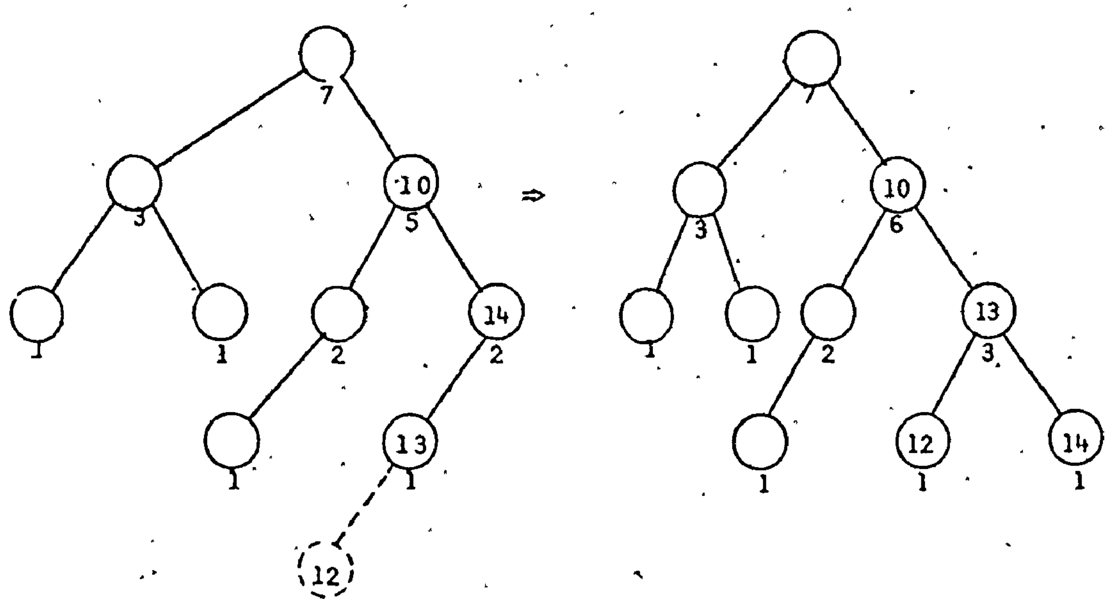
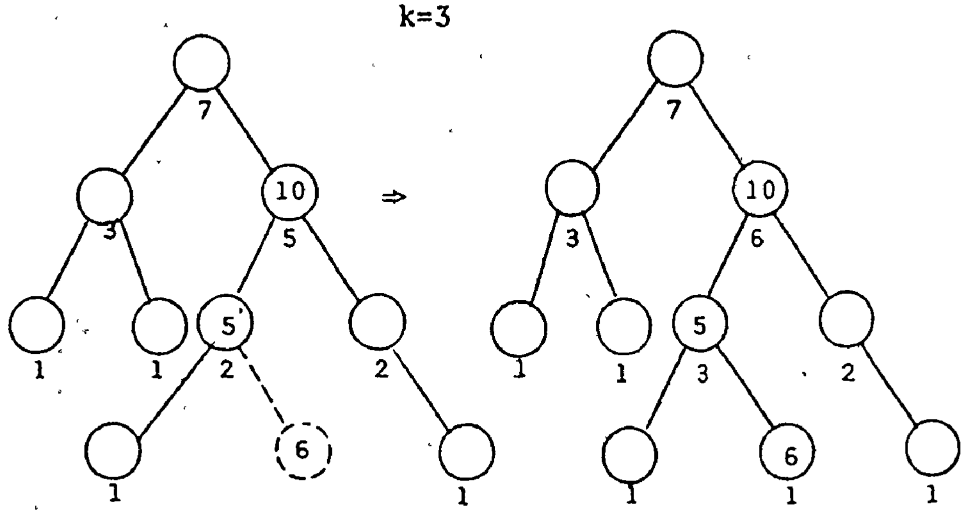


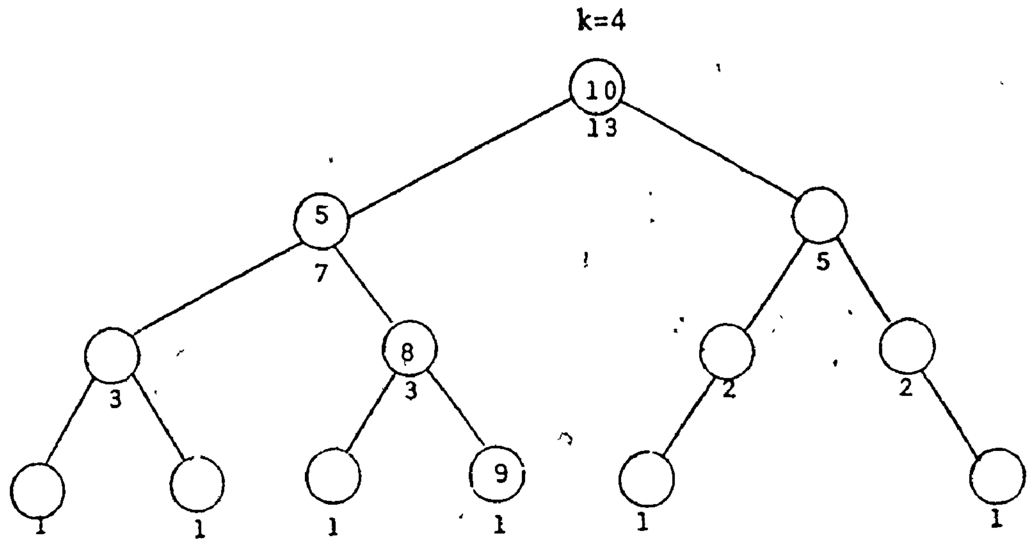
Figure 3.5.4

k=3



BIA for 6 , m=1

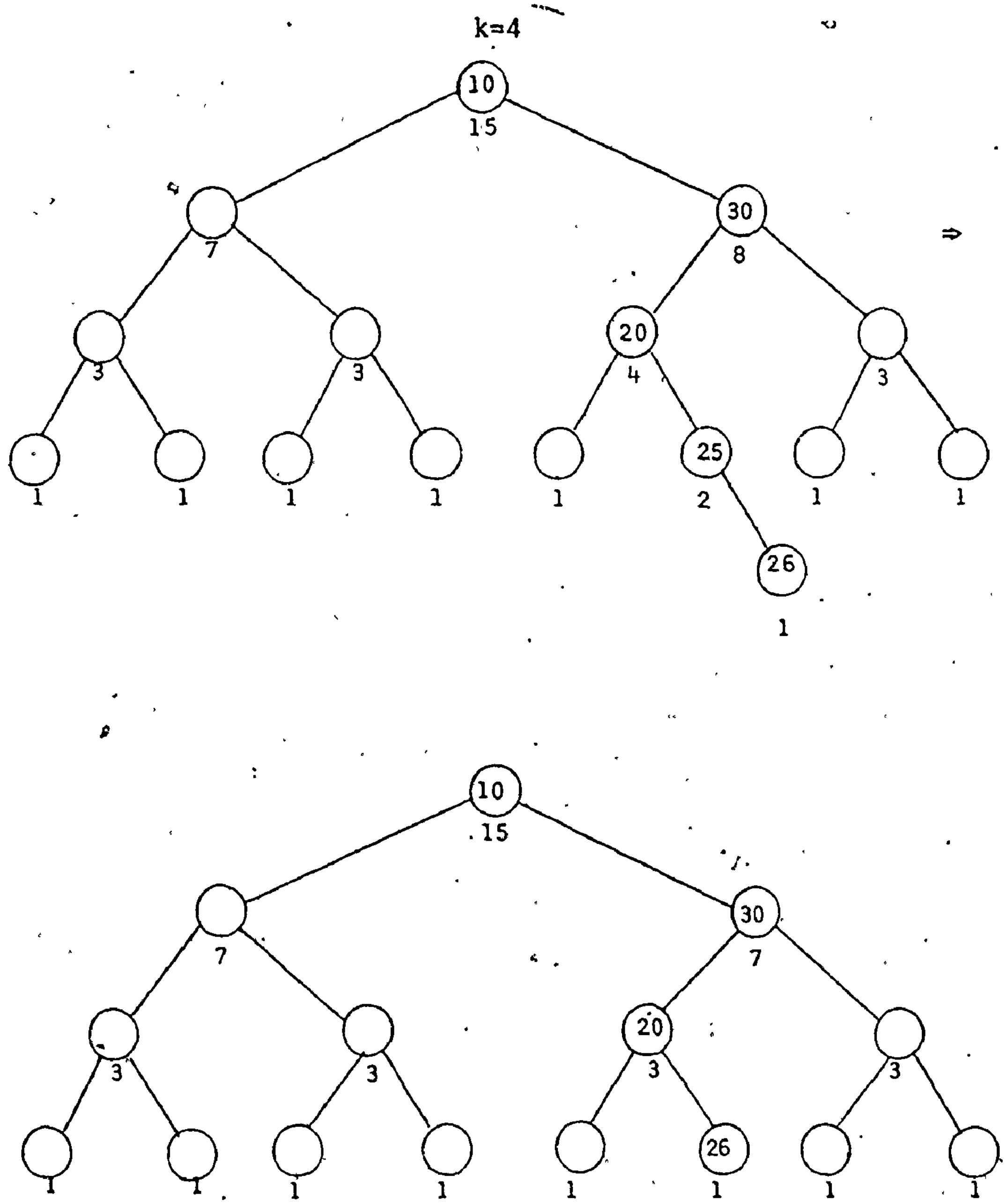
Figure 3.5.5



BDA for 9.

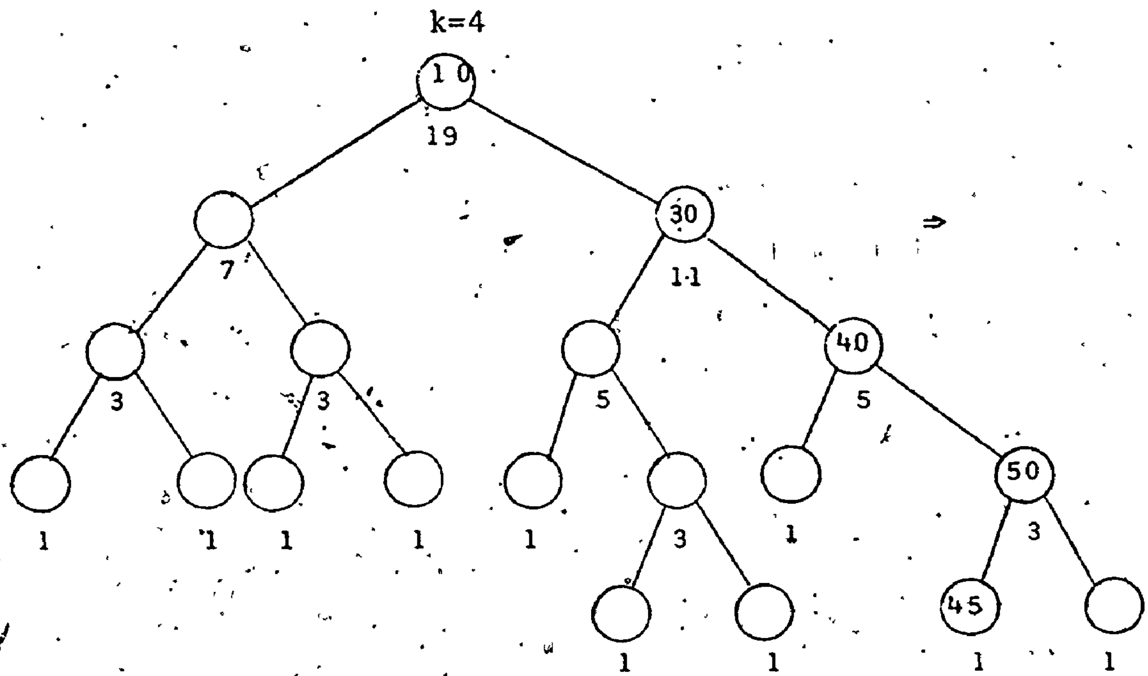
key(u_1)=10, key(u_2)=5, key(u_3)=8, key(u_4)=9, i.e., k'=4.

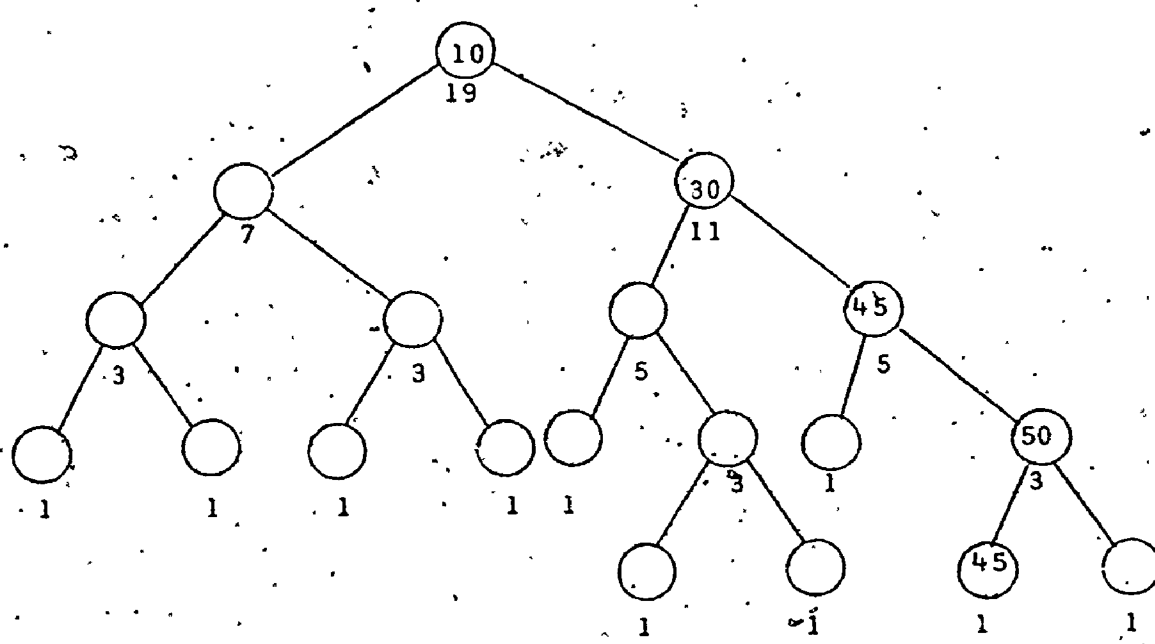
Figure 3.5.6



BDA for 25, one son case.

Figure 3.5.7

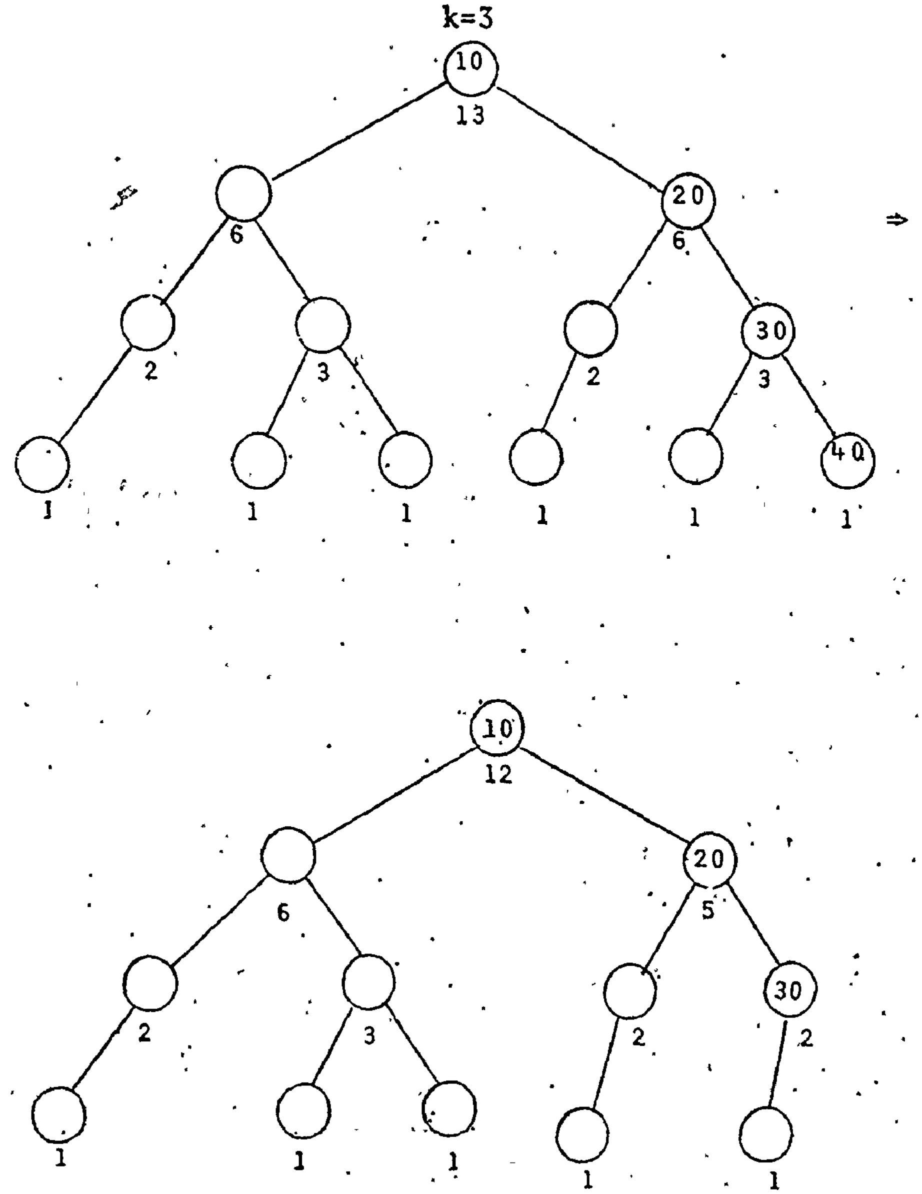




BDA for 40.

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

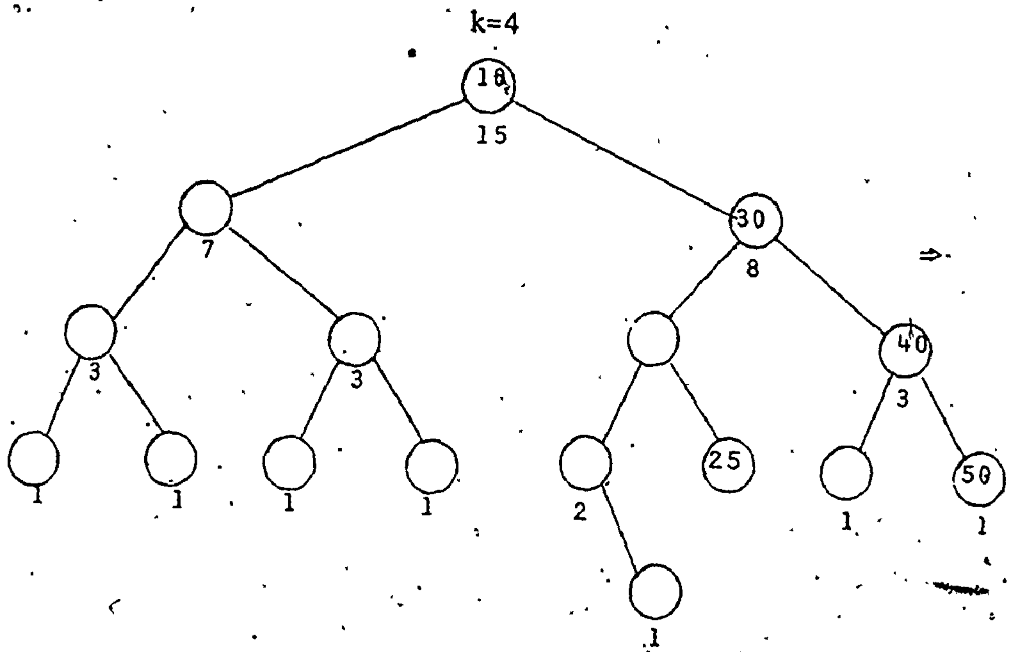
Figure 3.5.8

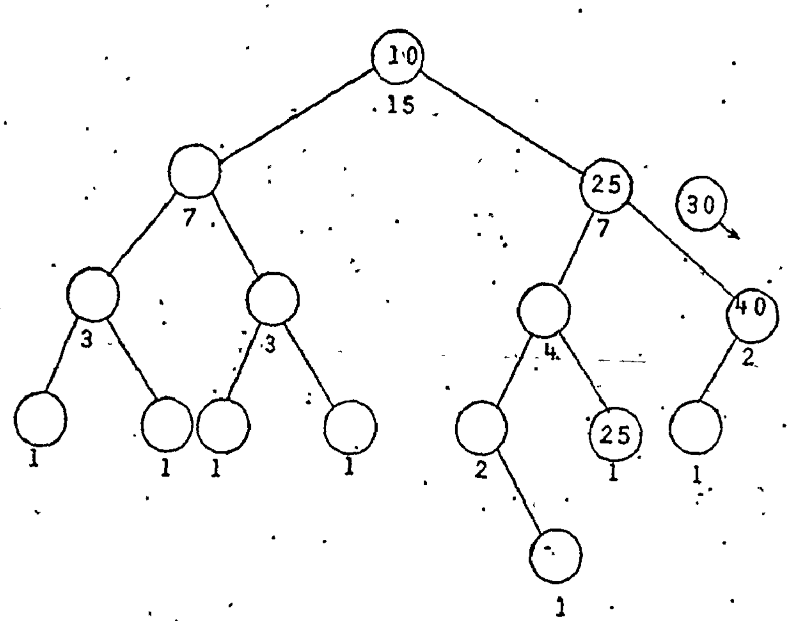


BDA for 40

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

Figure 3.5.9

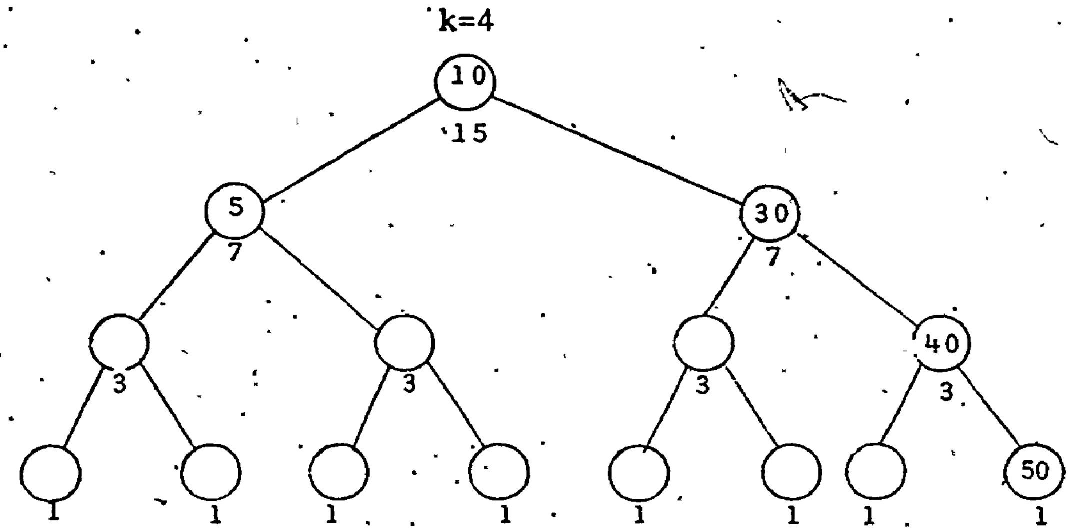




BDA for 50.

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 tree($u_{k'-i}$) is (k'-i)-complete, $1 \leq i \leq 3$ and 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 α_i, $1 \leq i \leq n$, and β_i, $0 \leq i \leq n$, the scheme is as follows. Given the root node with key X_i, then its left subtree is an optimum tree for the weights $\alpha_1,\ldots,\alpha_{i-1}$, $\beta_0,\ldots,\beta_{i-1}$; its right subtree is optimum for $\alpha_{i+1},\ldots,\alpha_n$, $\beta_i,\ldots,\beta_n$. Therefore we can build up optimum trees for all "weight intervals" $\alpha_{i+1},\ldots,\alpha_j$, $\beta_i,\ldots,\beta_j$, $i < j$, 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 β_i, $\alpha_{i+1},\ldots\ldots\ldots\ldots,\alpha_j,\beta_j$ 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.

$$\mathrm{WPL}(i,i) = W(i,i) = \beta_i, \qquad 0 \leq i \leq n, \tag{1}$$

$$W(i,j) = W(i,j-1) + \alpha_j + \beta_j, \qquad 0 \leq i < j \leq n, \tag{2}$$

$$\mathrm{WPL}(i,j) = [\min_{i < k \leq j}[\mathrm{WPL}(i,k-1) + \mathrm{WPL}(k,j)] + W(i,j)] \qquad 0 \leq i < j \leq n \tag{3}$$

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 $j-i = 1,2,\ldots,n$. Knuth (1971) proved the following theorem.

Theorem

There is always a solution to the above equations satisfying $R(i,j-1) \leq R(i,j) \leq R(i+1,j)$ for $0 \leq i < j \leq n$, 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, $1 \leq i \leq n$, and $2n + 1$ weights α_i, $1 \leq i \leq n$ and β_i, $0 \leq i \leq n$.

(i) Determine all the one-node optimum trees. For $1 \leq i \leq n$, let $R(i-1,i) = i$. Let $\ell = 2$.

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

(iii) For $j = \ell, \ell+1,\ldots,n$ let $i = j-\ell$,

$$WPL(i,j) = W(i,j) + \min(\{WPL(i,k-1) + WPL(k,j) \mid R(i,j-1) \leq k \leq R(i+1,j)\}),$$

and $R(i,j)$ equals the value of k for which the minimum occurs. Let $\ell = \ell+1$ 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_ℓ is given by (b) with $j = R(i,j)-1$ and u_r 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 α_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 $1 \leq i \leq n$, $k \geq 1$

$$t_{i,k}(T) = \begin{cases} \text{if } k = 1 \text{ then } t_i(T), \text{ otherwise} \\ \text{if } \ell(u_i) < k+1 \text{ then } T \\ \text{otherwise } t_i(t_{i,k-1}(T)). \end{cases}$$

Figure 3.7.2 illustrates $t_{i,k}(T)$ for $k = 2$. Let π_k, $k \geq 1$, be a mapping from C_n into 2^{C_n} defined for all T in C_n as follows:

$\pi_k(T) = \{T' | T' = t_{i,j}(T), 1 \leq j \leq k, 1 \leq i \leq n\}$. $\pi_k(T)$ 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, $\pi_n(T) = \pi_{n+i}(T)$ 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 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 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 α_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), π_k(tree(u)). For $j = k, k-1,\ldots,1$ do step (iii) and then stop.

(iii) Examine particular elements of π_k(tree(u)). For $i = 1,\ldots,n$ investigate $t_{i,j}$(tree(u)). If WPL($t_{i,j}$(tree(u))) < WPL(tree(u)) the weighted path length of the tree has been reduced and tree(u) is replaced by $t_{i,j}$(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 $WPL(t_{i,k}(tree(u))) < WPL(tree(u))$ without actually performing any transformations. Referring to Figure 3.7.3 one can calculate the new weighted path length of a tree, tree(u), when a node is promoted one level. Let $\overline{tree(u)}$ denote the tree after the transformation, $\overline{WPL(tree(u))}$ be its weighted path length and $\overline{W(z)}$, for any z, be the weight of $\overline{tree(z)}$. Suppose y is to be promoted and it is a right son of its father, v say, then

$$\begin{aligned} WPL(tree(u)) - \overline{WPL(tree(u))} &= [W(v) + WPL(tree(v_\ell)) + W(y) + WPL(tree(y_\ell)) + \\ &+ WPL(tree(y_r))] - [\overline{W(y)} + \overline{W(v)} + WPL(tree(v_\ell)) + \\ &+ \overline{WPL(tree(v_r))} + WPL(tree(y_r))] \\ &= [W(v) + W(y)] - [\overline{W(v)} + \overline{W(y)}] \\ &= W(y) - \overline{W(v)}. \end{aligned}$$

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

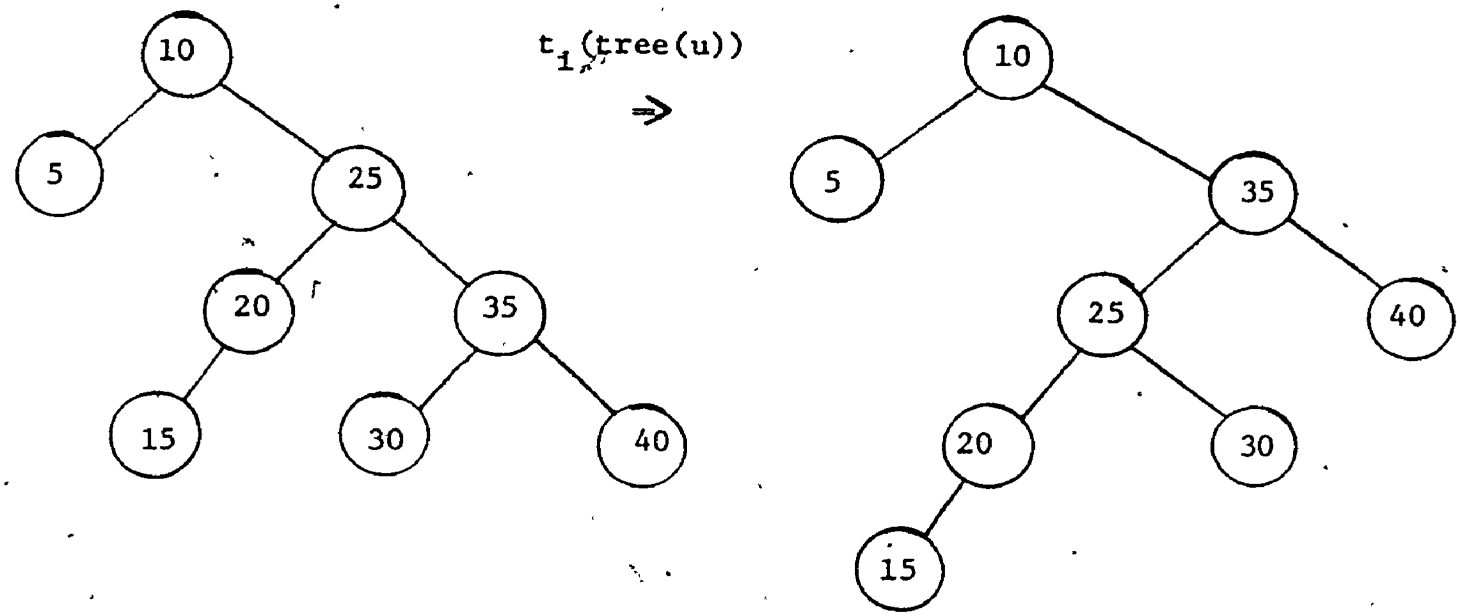
The same equation results when y is a left son of v. It is important to note that $\overline{W(v)}$ is the weight of the (left) right subtree of y after the transformation and that W(v) and consequently W(y) are the only subtree weights affected. $\overline{W(v)}$ is easily calculated. If y is a right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

$$\overline{W(y)} = \overline{W(v)} + \alpha_y + W(y_r) \text{ or}$$

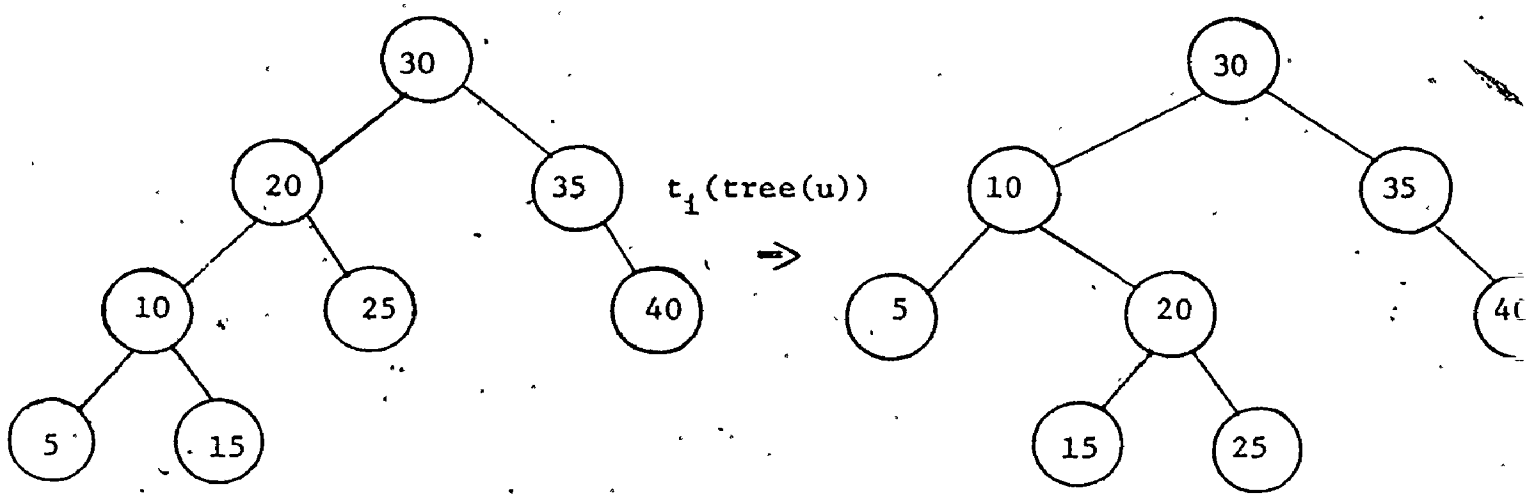
$$\overline{W(y)} = \overline{W(v)} + \alpha_y + W(y_\ell) \text{ respectively.}$$

Since the whole process hinges on the weights of the subtree of y, two variables, initialized to $W(y_\ell)$ and $W(y_r)$, are used to hold the values of $\overline{W(y_\ell)}$ 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 $\overline{WPL(tree(u))}$ 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.

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.

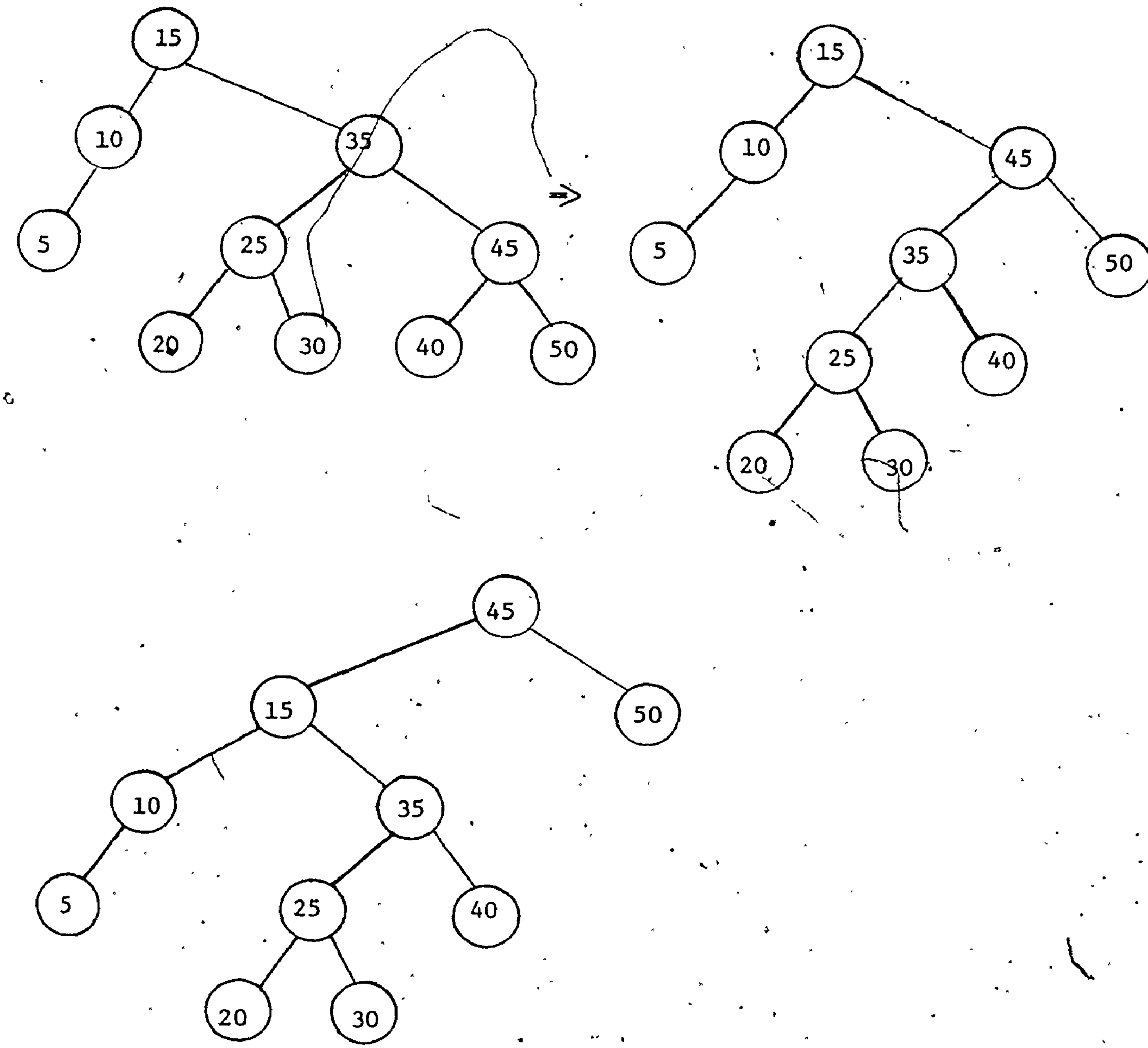


key(u) = 10, key(u_i) = 35. u_i is a right son of its father.



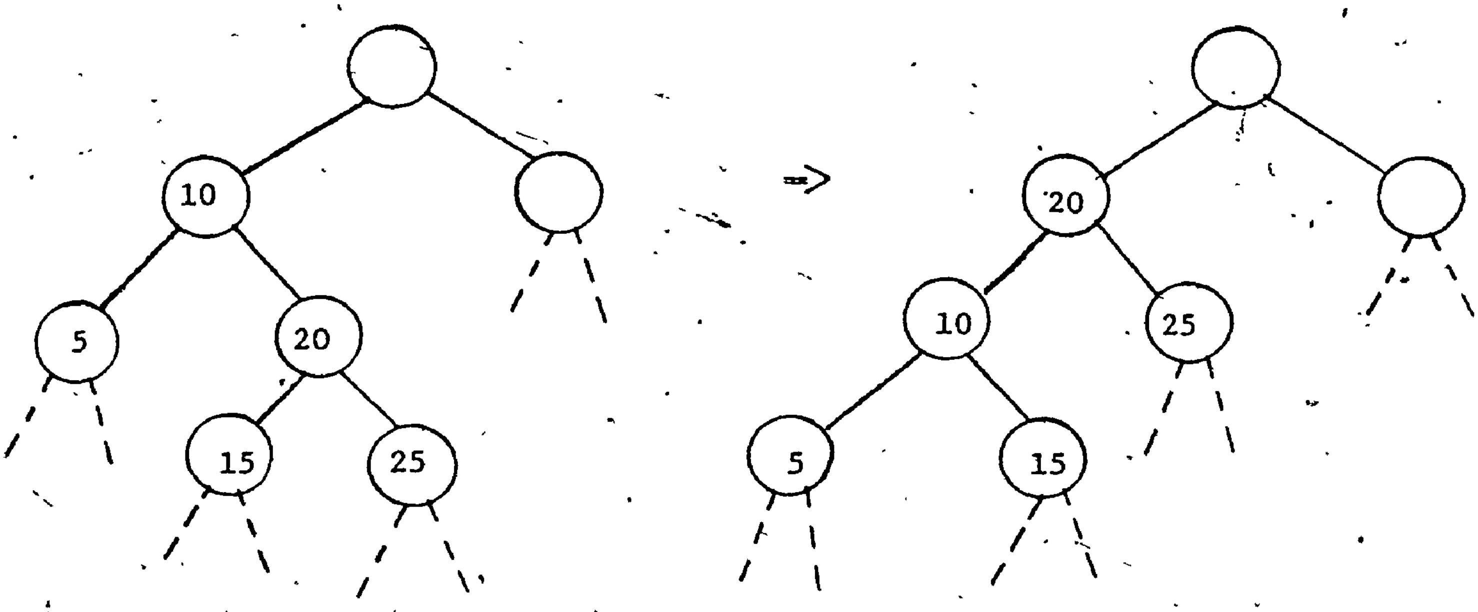
key(u) = 30, key(u_i) = 10. u_i is a left son of its father.

Figure 3.7.1

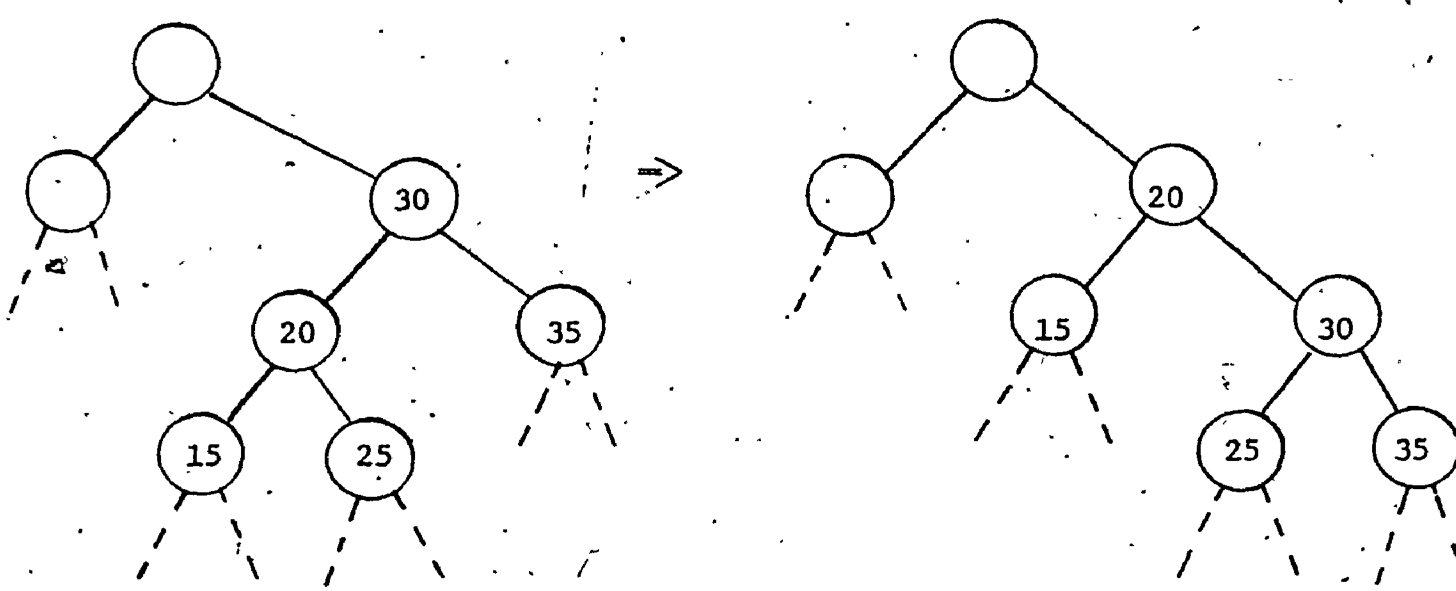


k = 2, key(u) = 15, key(u_i) = 45. $t_{i,k}$(tree(u)) results in the above.

<u>Figure 3.7.2</u>



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 (WGCA)

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 α_i and β_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 key(u_ℓ) or key(u_r) 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 $\alpha_{i-1} < \alpha_i > \alpha_{i+1}$. 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 usally corresponds to the associated α_i being a local maximum. In other words a local minimum of $WPL(X_i)$ corresponds to a local maximum for the α_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 α and β denote the sum of the α_i and β_i respectively and W(i,j) denote the weight of a tree with associated weights $\beta_i, \alpha_{i+1}, \beta_{i+1}, \ldots, \alpha_j, \beta_j, i \leq j$. 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_ℓ is a candidate for the key of the root node. The selection of F depends on α and β. They found the following choices of F to be acceptable: (1) if β is many times α, choose $F = W(0,n)$, (2) if α and β are nearly equal choose $F = 4$, (3) if β is small compared to α, the individual α_i weights determine the best value of F. After empirical investigations they conclude that unless β is many times α, $F = 4$ is an acceptable choice. Expansion of the neighborhood is possible. Suppose a set of m ordered keys has been selected, $\{X_j,\ldots,X_k\}$, $j \leq k$. Denote α^* as the maximum weight of any key in this set and X_c as the centroid. If the weight of X_j equals α^* and there is no other X_ℓ such that $X_j < X_\ell \leq X_c$ and $\alpha_\ell = \alpha^*$, then include in the set any X_{j-i-1} such that $\alpha_{j-i-1} > \alpha_{j-i}$, $i = 0,\ldots,j-p-1$ and $\alpha_{p-1} \leq \alpha_p$, or $j-p = \lfloor \log_2 n \rfloor$. 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 $\lfloor \log_2 n \rfloor$ 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 α and β. In general if β is less than a few times α, the value beyond which it does not pay to go is small, $N_0 = 15$ being a good choice. If β is many times α, 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 α_i, $1 \leq i \leq n$, and β_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 $S = T \cup \{X_c\}$ 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, max, such that α_{max} = maximum α_i where X_i is in S.

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

(v) Expand the neighborhood to the left of the centroid. If X_ℓ 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 $\alpha_{\ell-j-1} > \alpha_{\ell-j}$, $j = 0,\ldots,\ell-p-1$ and $\alpha_{p-1} \leq \alpha_p$, or $\ell-p = \lfloor \log_2 n \rfloor$. If X_ℓ is not the first member of S let L be the null set.

(vi) Determine if any key X_i in S such that $X_i \geq X_c$ has as its associated weight α_{max}. Let r be the minimum index,

$r \geq c$, such that $\alpha_r = \alpha_{max}$. 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 $\alpha_r < \alpha_{r+1}$, form the ordered set $R^* = \{X_{r+1}, X_{r+2}, \ldots, X_p\}$ where $\alpha_{r+j} < \alpha_{r+j+1}$, $j = 0, \ldots, p-r-1$ and $\alpha_p \geq \alpha_{p+1}$, or $p-r = \lfloor \log_2 n \rfloor$. If X_r is not the last member of S let R be the null set.

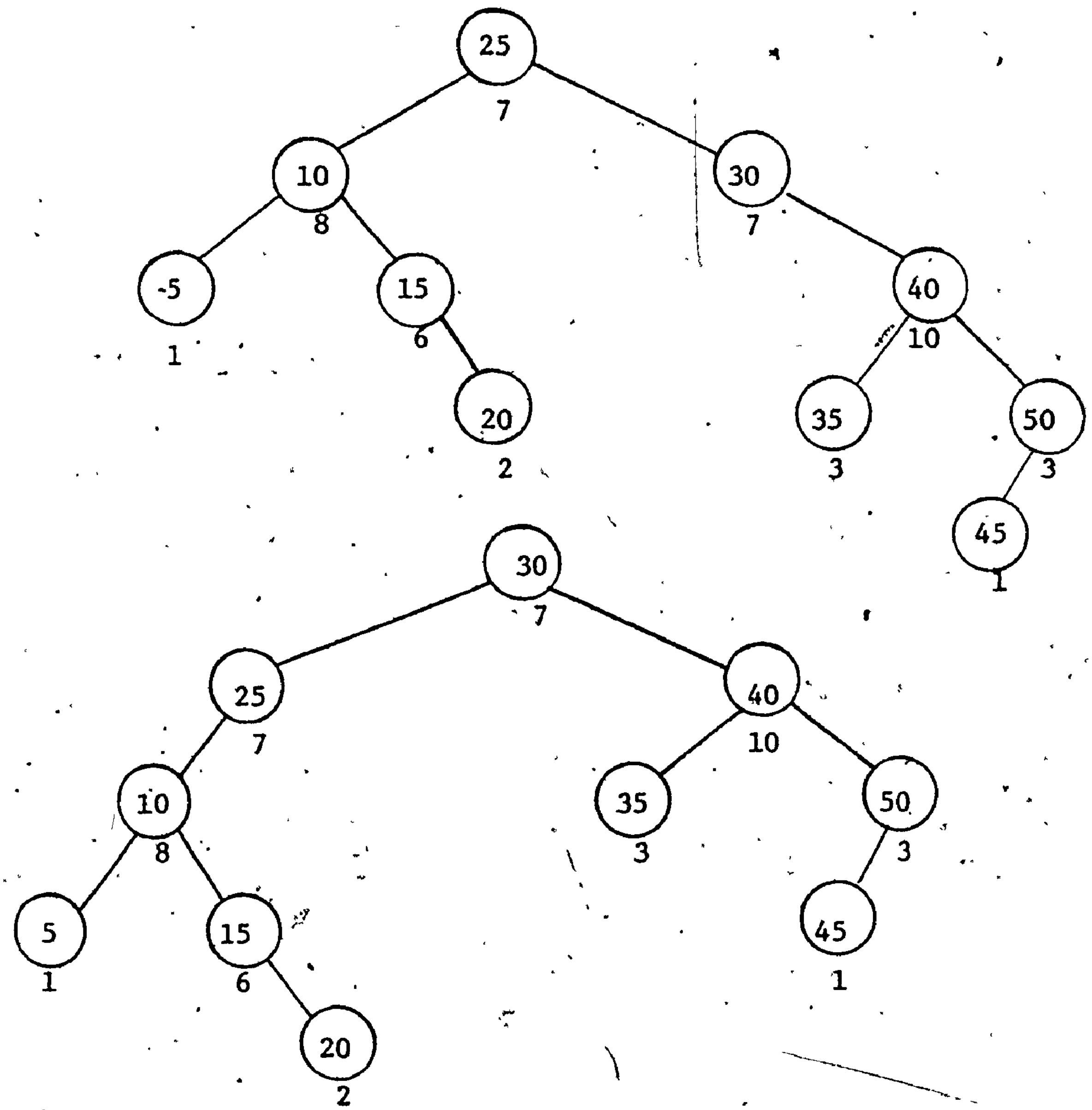
(viii) Choose the key of the root node. Find the key X_{root} of maximum weight in the set $L \cup S \cup R$ such that $|W(0,root-1) - W(root,n)|$ is minimized. Choose X_{root} as the key of the root of the tree.

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

3.8.3 IMPLEMENTATION

The implemented algorithm follows closely the steps outlined in the WGCA. 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 enternal 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 α_i and β_i vectors are arranged in the following matrices as

$$\begin{array}{cccccc} \alpha_1 & \alpha_2 , & \cdot \quad \cdot \quad \cdot \quad \cdot & , \alpha_{19} , & \alpha_{20} \\ \alpha_{21} & \alpha_{22} , & \cdot \quad \cdot \quad \cdot \quad \cdot & , \alpha_{39} , & \alpha_{40} \\ \alpha_{181}, & \alpha_{182}, & \cdot \quad \cdot \quad \cdot & , \alpha_{199}, & \alpha_{200} \end{array}$$

α_i frequencies

Set 1

4	19	3	3	1	1	28	4	4	5	1	1	1	35	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
2	1	52	3	1	4	1	1	1	6	8	13	1	4	1	1	1	1	1	26
33	1	1	8	7	1	2	1	1	2	4	10	5	10	23	1	1	9	6	2
128	350	1	2	1	87	2	1	8	3	4	1	63	2	3	14	150	1	1	1
1	2	1	1	8	1	2	1	3	4	21	1	9	1	1	2	10	4	15	1
1	24	6	2	2	1	1	1	3	1	1	3	14	4	1	2	1	1	2	2
2	4	43	2	1	4	1	1	18	1	1	7	1	45	15	9	3	1	5	4
10	8	1	5	10	1	2	10	2	1	24	29	1	16	1	1	4	1	2	1
1	1	1	1	1	4	2	2	6	4	1	14	5	5	1	53	1	1	1	4

Table 4.1.1 (cont'd.)

Set 2

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
90	3	1	4	1	4	29	1	1	1	1	2	6	2	2	1	5	5	2	40
1	3	5	4	62	7	1	1	1	3	11	4	2	1	1	2	2	6	1	1
1	1	1	24	1	1	1	6	100	15	1	2	7	1	1	1	10	3	1	2
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	4	1	1	1	1	1	1	7	1	3	39	1	1	15	11	2	1	4	6
2	1	1	1	1	2	29	3	10	24	2	1	1	1	3	1	1	1	5	1
3	6	2	2	1	23	1	9	1	1	8	1	6	1	1	30	2	1	14	90

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	14	2	12	3	1	1	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	1	1	3	2	1	1	2	1	1	4	1	7	1	7	12	2
4	10	1	2	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	1	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	33	30
2	1	1	4	2	2	1	3	1	2	1	7	1	2	4	2	6	6	77	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.)

Set 5

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	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	88	13	1	4	1	1	1	1	1	26
2	4	43	2	1	4	1	1	18	1	1	7	1	45	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

β_i frequences

<u>Class 1</u>: The β_i are all equal.

<u>case 1</u> $\beta_i = 0 \quad , 0 \leq i \leq n$.

<u>case 2</u> $\beta_i = 10 \ , 0 \leq i \leq n$.

<u>Class 2</u>: The β_i were chosen so that the sum of the β_i would be equal to or less than the sum of the α_i.

<u>case 3</u>

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	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	7	1	2	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.)

case 4

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

Class 3 The β_i were chosen so the sum of the β_i would be larger than the sum of the α_i.

case 5

2	1	52	3	1	4	1	1	1	6	8	13	1	4	1	1	1	1	1	26
128	350	1	2	1	87	2	1	8	3	4	1	63	2	3	14	140	1	1	1
2	4	43	2	1	4	1	1	18	1	1	7	1	45	15	9	3	1	5	4
10	8	1	5	10	1	2	10	2	1	24	29	1	16	1	1	4	1	2	1
90	3	1	4	1	4	29	1	1	1	1	2	6	2	2	1	5	5	2	40
1	1	1	24	1	1	1	6	100	15	1	2	7	1	1	1	10	3	1	2
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
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	14	173	1	1	2	4	4
4																			

Table 4.1.1 (cont'd.)

case 6

12	8	315	18	7	24	6	5	6	37	49	78	6	23	6	8	10	6	156	720
900	6	14	12	623	12	8	48	19	25	6	479	12	19	84	900	7	6	6	18
24	248	13	7	28	6	8	129	6	8	42	6	570	94	56	18	13	30	25	160
49	24	30	68	6	12	60	12	9	145	183	6	97	8	9	27	7	13	5	840
6	14	37	11	9	4	34	34	14	240	6	6	5	145	8	8	7	43	907	110
7	11	41	8	7	7	211	60	18	7	14	49	31	930	7	75	7	98	7	6
70	7	15	15	121	23	18	14	7	12	6	25	63	7	315	7	7	66	350	14
307	14	6	66	323	999	960	6	6	54	4	1	1	468	24	18	6	540	18	31
40	73	7	12	6	6	195	12	970	12	43	2	7	25	1	90	970	6	5	12
25	22	11	18	7	497	45	221	190	6	30	72	6	5	6	37	49	78	6	8
10																			

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'-i)-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 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 α 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 occurences 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 α increases the average search is reduced and the confidence intervals become tighter. The number of general rotations is seen to increase with α 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 $\alpha < \frac{29}{100}$. For $\alpha = \frac{29}{100}$ 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 α. 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 4.2.1 AVL tree insertion statistics

tree size	AVERAGE SEARCH AV	AVERAGE SEARCH CI	AVERAGE SINGLE ROTATION AV	AVERAGE SINGLE ROTATION CI	AVERAGE DOUBLE ROTATION AV	AVERAGE DOUBLE ROTATION CI	AVERAGE GENERAL ROTATION AV	AVERAGE GENERAL ROTATION CI
250	9.2000	0.0651	0.2660	0.0461	0.2400	0.0438	0.5060	0.0636
500	10.1680	0.0693	0.2520	0.0449	0.2340	0.0433	0.4860	0.0624
750	10.8120	0.0651	0.2360	0.0435	0.2460	0.0444	0.4820	0.0621
1000	11.2460	0.0678	0.2500	0.0447	0.2560	0.0453	0.5060	0.0636
1250	11.5140	0.0724	0.2080	0.0408	0.2320	0.0431	0.4400	0.0593
1500	11.7560	0.0743	0.2520	0.0449	0.1880	0.0388	0.4400	0.0593
1750	12.0180	0.0765	0.2580	0.0454	0.2100	0.0410	0.4680	0.0612
2000	12.2740	0.0688	0.2440	0.0442	0.2740	0.0468	0.5180	0.0644
2250	12.3640	0.0757	0.2160	0.0416	0.2120	0.0412	0.4280	0.0585
2500	12.4820	0.0741	0.2480	0.0445	0.2160	0.0416	0.4640	0.0609
2750	12.5860	0.0744	0.2360	0.0435	0.1900	0.0390	0.4260	0.0584
3000	12.7520	0.0737	0.2180	0.0418	0.2260	0.0425	0.4440	0.0596
3250	12.9240	0.0746	0.2180	0.0418	0.2260	0.0425	0.4440	0.0596
3500	13.0600	0.0721	0.2360	0.0435	0.2320	0.0431	0.4680	0.0612
3750	13.1920	0.0772	0.2480	0.0445	0.2540	0.0451	0.5020	0.0634
4000	13.2980	0.0745	0.2340	0.0433	0.2540	0.0451	0.4880	0.0625
4250	13.2920	0.0786	0.2360	0.0435	0.2300	0.0429	0.4660	0.0611
4500	13.3960	0.0765	0.2080	0.0408	0.2660	0.0461	0.4740	0.0616
4750	13.5320	0.0710	0.2460	0.0444	0.2300	0.0429	0.4760	0.0617
5000	13.5120	0.0790	0.2320	0.0431	0.2340	0.0433	0.4660	0.0611

Table 4.2.2 AVL tree deletion statistics

tree size	AVERAGE SEARCH AV	AVERAGE SEARCH CI	AVERAGE SINGLE ROTATION AV	AVERAGE SINGLE ROTATION CI	AVERAGE DOUBLE ROTATION AV	AVERAGE DOUBLE ROTATION CI	AVERAGE SPECIAL SINGLE AV	AVERAGE SPECIAL SINGLE CI	AVERAGE GENERAL ROTATION AV	AVERAGE GENERAL ROTATION CI	AVERAGE TRACEBACK AV	AVERAGE TRACEBACK CI
250	7.0680	0.1372	0.0840	0.0259	0.0780	0.0250	0.0440	0.0188	0.2060	0.0406	1.9380	0.1259
500	8.1800	0.1410	0.0580	0.0215	0.0660	0.0230	0.0620	0.0223	0.1860	0.0386	1.7500	0.1071
750	8.7320	0.1436	0.0880	0.0265	0.0820	0.0256	0.0660	0.0230	0.2360	0.0435	1.9180	0.1253
1000	9.0900	0.1357	0.0760	0.0247	0.0820	0.0256	0.0600	0.0219	0.2180	0.0418	1.9140	0.1103
1250	9.4940	0.1475	0.0680	0.0233	0.0860	0.0262	0.0540	0.0208	0.2080	0.0408	2.0180	0.1414
1500	9.7920	0.1427	0.0700	0.0237	0.0800	0.0253	0.0700	0.0237	0.2200	0.0420	2.0240	0.1432
1750	10.0220	0.1361	0.0920	0.0271	0.0820	0.0256	0.0700	0.0237	0.2440	0.0442	1.9280	0.1258
2000	10.2000	0.1434	0.0860	0.0262	0.0860	0.0262	0.0340	0.0165	0.2060	0.0406	1.8980	0.1236
2250	10.4220	0.1389	0.0540	0.0208	0.0980	0.0280	0.0560	0.0212	0.2080	0.0408	1.9600	0.1311
2500	10.4760	0.1463	0.0860	0.0262	0.1160	0.0305	0.0540	0.0208	0.2560	0.0453	2.0320	0.12
2750	10.7480	0.1447	0.0620	0.0223	0.0920	0.0271	0.0540	0.0208	0.2080	0.0408	1.8820	0.14
3000	10.8300	0.1441	0.0880	0.0265	0.0980	0.0280	0.0640	0.0226	0.2500	0.0447	1.9780	0.1305
3250	11.1580	0.1263	0.0620	0.0223	0.0940	0.0274	0.0460	0.0192	0.2020	0.0402	1.9800	0.1334
3500	11.2340	0.1249	0.0880	0.0265	0.1140	0.0302	0.0600	0.0219	0.2620	0.0458	1.9240	0.12
3750	11.3400	0.1296	0.0680	0.0233	0.0660	0.0230	0.0460	0.0192	0.1800	0.0379	1.8740	0.12
4000	11.3400	0.1347	0.1080	0.0294	0.0840	0.0259	0.0660	0.0230	0.2580	0.0454	1.9220	0.12
4250	11.2380	0.1326	0.0960	0.0277	0.0880	0.0265	0.0560	0.0212	0.2400	0.0438	1.9540	0.12
4500	11.1420	0.1508	0.0660	0.0230	0.0840	0.0259	0.0540	0.0208	0.2040	0.0404	1.8800	0.12
4750	11.2060	0.1687	0.0920	0.0271	0.0640	0.0226	0.0420	0.0183	0.1980	0.0398	1.8780	0.13
5000	8.9620	0.3666	0.0960	0.0277	0.0720	0.0240	0.0800	0.0253	0.2480	0.0445	1.9120	0.13

Table 4.2.3 Bell-tree insertion statistics AVERAGE SEARCH

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	9.7860	0.1507	9.3340	0.1350	9.1360	0.0659	9.0880	0.0494	9.0140	0.0351
500	11.1380	0.1762	10.6440	0.1107	10.2000	0.0768	10.1180	0.0528	10.0300	0.0385
750	11.9660	0.5929	11.1360	0.1129	10.8400	0.1516	10.7000	0.0556	10.6160	0.0525
1000	12.5660	0.7874	11.6200	0.3945	11.3620	0.1955	11.1560	0.0946	11.0780	0.0588
1250	12.6740	0.1789	12.0320	0.1272	11.6480	0.0856	11.4800	0.0576	11.3800	0.0486
1500	12.7560	0.1885	12.5300	0.5884	11.8280	0.0852	11.8340	0.1390	11.6120	0.0488
1750	13.3440	0.1777	12.8380	0.6773	12.0860	0.0833	12.0740	0.1604	11.8740	0.0479
2000	13.3180	0.1870	12.6120	0.1262	12.2520	0.0823	12.1260	0.0627	12.0380	0.0464
2250	13.7260	0.1845	12.9000	0.1360	12.4640	0.0813	12.3460	0.0664	12.2700	0.0494
2500	13.7540	0.2046	13.0980	0.1353	12.5700	0.0911	12.4300	0.0696	12.3960	0.0496
2750	13.9940	0.1949	13.1040	0.1328	12.7320	0.0861	12.5300	0.0639	12.5440	0.0500
3000	14.2780	0.2033	13.3540	0.1326	12.9520	0.0922	12.7880	0.0688	12.6720	0.0510
3250	14.2260	0.2041	13.4080	0.1753	12.9660	0.0899	12.7600	0.0659	12.7880	0.0524
3500	14.4900	0.2118	13.5960	0.1285	13.1540	0.0896	12.9420	0.0645	12.8940	0.0512
3750	14.7360	0.2635	13.8780	0.1332	13.2820	0.0965	13.1040	0.0631	12.9820	0.0498
4000	14.7140	0.2069	13.8040	0.1336	13.4140	0.0926	13.1980	0.0655	13.1000	0.0475
4250	14.7300	0.2049	13.8360	0.1409	13.4180	0.0947	13.2760	0.0665	13.1460	0.0497
4500	14.8700	0.1991	14.1260	0.4246	13.4940	0.0936	13.3660	0.1101	13.2700	0.0507
4750	14.9820	0.2144	14.0680	0.1431	13.5420	0.0924	13.4800	0.0671	13.2900	0.0523
5000	15.1020	0.2203	14.3360	0.4105	13.6760	0.1019	13.5100	0.1048	13.4320	0.0526

Table 4.2.4 Bell-tree insertion statistics AVERAGE SINGLE ROTATION

tree	2		3		4		5		6	
size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0.1360	0.0330	0.2260	0.0425	0.3980	0.0564	0.7140	0.0756	1.5920	0.1129
500	0.1340	0.0327	0.2680	0.0463	0.4060	0.0570	0.6740	0.0734	1.3620	0.1044
750	0.1660	0.0364	0.2240	0.0423	0.4220	0.0581	0.7280	0.0763	1.2020	0.0981
1000	0.1260	0.0317	0.2600	0.0456	0.4360	0.0591	0.6760	0.0735	1.2640	0.1006
1250	0.1440	0.0339	0.2720	0.0466	0.3800	0.0551	0.7520	0.0776	0.9720	0.0882
1500	0.1420	0.0337	0.2240	0.0423	0.4220	0.0581	0.6380	0.0714	1.1420	0.0956
1750	0.1780	0.0377	0.2080	0.0408	0.4080	0.0571	0.8260	0.0813	1.2980	0.1019
2000	0.1120	0.0299	0.2840	0.0477	0.3840	0.0554	0.7000	0.0748	1.2160	0.0986
2250	0.1140	0.0302	0.2640	0.0460	0.4260	0.0584	0.6780	0.0736	1.0600	0.0921
2500	0.1220	0.0312	0.2360	0.0435	0.4020	0.0567	0.6520	0.0722	0.9680	0.0880
2750	0.1600	0.0358	0.2220	0.0421	0.3520	0.0531	0.6940	0.0745	1.0600	0.0921
3000	0.1360	0.0330	0.2440	0.0442	0.3900	0.0559	0.6860	0.0741	1.0140	0.0901
3250	0.1480	0.0344	0.2420	0.0440	0.3400	0.0522	0.6120	0.0700	1.0180	0.0902
3500	0.1480	0.0344	0.2240	0.0423	0.3800	0.0551	0.6880	0.0742	1.1180	0.0946
3750	0.1440	0.0339	0.2560	0.0453	0.4360	0.0588	0.7100	0.0754	1.1400	0.0955
4000	0.1380	0.0332	0.2460	0.0444	0.3720	0.0546	0.7200	0.0759	1.2280	0.0991
4250	0.1520	0.0349	0.2340	0.0433	0.4240	0.0582	0.6880	0.0786	1.1820	0.0972
4500	0.1240	0.0315	0.2280	0.0427	0.3780	0.0550	0.7100	0.0754	1.0520	0.0917
4750	0.1200	0.0310	0.2480	0.0445	0.4100	0.0573	0.6420	0.0717	0.9600	0.0876
5000	0.1580	0.0356	0.2600	0.0456	0.4160	0.0577	0.6780	0.0736	1.2420	0.0997

Table 4.2.5 Bell-tree insertion statistics AVERAGE DOUBLE ROTATION

tree	2		3		4		5		6	
size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0.1480	0.0344	0.1300	0.0322	0.1440	0.0339	0.1540	0.0351	0.1520	0.0349
500	0.1560	0.0353	0.1780	0.0377	0.1680	0.0367	0.1480	0.0344	0.1760	0.0375
750	0.1460	0.0342	0.1660	0.0364	0.1560	0.0353	0.1660	0.0364	0.1500	0.0346
1000	0.1360	0.0330	0.1440	0.0339	0.1620	0.0360	0.1560	0.0353	0.1780	0.0377
1250	0.1440	0.0339	0.1980	0.0398	0.1900	0.0390	0.1400	0.0335	0.1560	0.0353
1500	0.1760	0.0375	0.1540	0.0351	0.1700	0.0369	0.1780	0.0377	0.1840	0.0384
1750	0.1580	0.0356	0.1740	0.0373	0.1720	0.0371	0.1860	0.0386	0.1720	0.0371
2000	0.1340	0.0327	0.1560	0.0353	0.1520	0.0349	0.1640	0.0362	0.1360	0.0330
2250	0.1500	0.0346	0.1420	0.0337	0.2120	0.0412	0.1480	0.0344	0.1540	0.0351
2500	0.1480	0.0344	0.1620	0.0360	0.1700	0.0369	0.1740	0.0373	0.1460	0.0342
2750	0.1080	0.0294	0.1640	0.0362	0.1760	0.0375	0.1620	0.0360	0.1800	0.0379
3000	0.1540	0.0351	0.1600	0.0358	0.1740	0.0373	0.1600	0.0358	0.1700	0.0369
3250	0.1600	0.0358	0.1600	0.0358	0.1820	0.0382	0.1480	0.0344	0.1900	0.0390
3500	0.1380	0.0332	0.1440	0.0339	0.1820	0.0382	0.1760	0.0375	0.1820	0.0382
3750	0.1180	0.0307	0.1700	0.0369	0.1480	0.0344	0.1240	0.0315	0.1520	0.0349
4000	0.1400	0.0335	0.1500	0.0346	0.1620	0.0360	0.1580	0.0356	0.1700	0.0369
4250	0.1480	0.0344	0.1560	0.0353	0.1600	0.0358	0.1840	0.0384	0.1600	0.0358
4500	0.1360	0.0330	0.1500	0.0346	0.1820	0.0382	0.1900	0.0390	0.1940	0.0394
4750	0.1840	0.0384	0.1300	0.0322	0.1760	0.0375	0.1920	0.0392	0.1860	0.0386
5000	0.1000	0.0283	0.1480	0.0344	0.1520	0.0349	0.1920	0.0392	0.1760	0.0375

Table 4.2.6 Bell-tree insertion statistics AVERAGE GENERAL ROTATION

tree size	k 2 AV	CI	3 AV	CI	4 AV	CI	5 AV	CI	6 AV	CI
250	0.2840	0.0477	0.3560	0.0534	0.5420	0.0658	0.8680	0.0833	1.7440	0.1181
500	0.2900	0.0482	0.4460	0.0597	0.5740	0.0678	0.8220	0.0811	1.5380	0.1109
750	0.3120	0.0500	0.3900	0.0559	0.5780	0.0680	0.8940	0.0846	1.3520	0.1040
1000	0.2620	0.0458	0.4040	0.0569	0.5980	0.0692	0.8320	0.0816	1.4420	0.1074
1250	0.2880	0.0480	0.4700	0.0613	0.5700	0.0675	0.8920	0.0845	1.1280	0.0950
1500	0.3180	0.0504	0.3780	0.0550	0.5920	0.0688	0.8160	0.0808	1.3260	0.1030
1750	0.3360	0.0518	0.3820	0.0553	0.5800	0.0681	0.0120	0.0900	1.4700	0.1084
2000	0.2460	0.0444	0.4400	0.0593	0.5360	0.0655	0.8640	0.0831	1.3520	0.1040
2250	0.2640	0.0460	0.4060	0.0570	0.6380	0.0714	0.8260	0.0813	1.2140	0.0985
2500	0.2700	0.0465	0.3980	0.0564	0.5720	0.0676	0.8260	0.0813	1.1140	0.0944
2750	0.2680	0.0463	0.3860	0.0556	0.5280	0.0650	0.8560	0.0828	1.2400	0.0996
3000	0.2900	0.0482	0.4040	0.0569	0.5640	0.0672	0.8460	0.0823	1.1840	0.0973
3250	0.3080	0.0496	0.4020	0.0567	0.5220	0.0646	0.7600	0.0780	1.2080	0.0983
3500	0.2860	0.0478	0.3680	0.0543	0.5620	0.0671	0.8640	0.0831	1.3000	0.1020
3750	0.2620	0.0458	0.4260	0.0584	0.5800	0.0681	0.8340	0.0817	1.2920	0.1017
4000	0.2780	0.0472	0.3960	0.0563	0.5340	0.0654	0.8780	0.0838	1.3980	0.1058
4250	0.3000	0.0490	0.3900	0.0559	0.5840	0.0684	0.8720	0.0835	1.1700	0.0967
4500	0.2600	0.0456	0.3780	0.0550	0.5600	0.0669	0.9000	0.0849	1.2460	0.0998
4750	0.3040	0.0493	0.3780	0.0550	0.5860	0.0685	0.8340	0.0817	1.1460	0.0957
5000	0.2580	0.0454	0.4080	0.0571	0.5680	0.0674	0.8700	0.0834	1.4180	0.1065

Table 4.2.7 Bell-tree insertion statistics AVERAGE SINGLE RECURSE

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0	0	0.0520	0.0204	0.0900	0.0268	0.1140	0.0302	0.0920	0.0271
500	0	0	0.0420	0.0183	0.0920	0.0271	0.0820	0.0256	0.0960	0.0277
750	0	0	0.0720	0.0240	0.0600	0.0219	0.0780	0.0250	0.1160	0.0305
1000	0	0	0.0600	0.0219	0.0760	0.0247	0.0880	0.0265	0.0900	0.0268
1250	0	0	0.0400	0.0179	0.0680	0.0233	0.1120	0.0299	0.0880	0.0265
1500	0	0	0.0520	0.0204	0.0620	0.0223	0.0860	0.0262	0.0860	0.0262
1750	0	0	0.0400	0.0179	0.0780	0.0250	0.1020	0.0286	0.1340	0.0327
2000	0	0	0.0580	0.0215	0.0640	0.0226	0.0800	0.0253	0.1040	0.0288
2250	0	0	0.0360	0.0170	0.0680	0.0233	0.1040	0.0288	0.0840	0.0259
2500	0	0	0.0580	0.0215	0.0760	0.0247	0.0800	0.0253	0.0780	0.0250
2750	0	0	0.0440	0.0188	0.0520	0.0204	0.0920	0.0271	0.1120	0.0299
3000	0	0	0.0520	0.0204	0.0700	0.0237	0.1300	0.0322	0.1040	0.0288
3250	0	0	0.0420	0.0183	0.0820	0.0256	0.0900	0.0268	0.0840	0.0259
3500	0	0	0.0360	0.0170	0.0800	0.0253	0.0960	0.0277	0.0940	0.0274
3750	0	0	0.0520	0.0204	0.0820	0.0256	0.0960	0.0280	0.1160	0.0305
4000	0	0	0.0460	0.0192	0.0700	0.0237	0.0820	0.0256	0.0840	0.0259
4250	0	0	0.0480	0.0196	0.0700	0.0237	0.0960	0.0277	0.1120	0.0299
4500	0	0	0.0640	0.0226	0.0760	0.0247	0.0960	0.0277	0.0780	0.0250
4750	0	0	0.0380	0.0174	0.0840	0.0259	0.1060	0.0291	0.0700	0.0237
5000	0	0	0.0460	0.0192	0.0760	0.0247	0.1020	0.0286	0.1100	0.0297

Table 4.2.8 Bell-tree insertion statistics AVERAGE DOUBLE RECURSE

tree size	k = 2 AV	CI	3 AV	CI	4 AV	CI	5 AV	CI	6 AV	CI
250	0	0	0.1620	0.0360	0.3020	0.0492	0.6420	0.0717	1.5400	0.1110
500	0	0	0.1460	0.0342	0.3640	0.0540	0.6180	0.0703	1.3240	0.1029
750	0	0	0.1340	0.0327	0.3260	0.0511	0.6840	0.0740	1.1280	0.0950
1000	0	0	0.1100	0.0297	0.3700	0.0544	0.6020	0.0694	1.2180	0.0987
1250	0	0	0.1820	0.0382	0.3380	0.0520	0.6760	0.0735	0.9200	0.0858
1500	0	0	0.1220	0.0312	0.3760	0.0548	0.6000	0.0693	1.1240	0.0948
1750	0	0	0.1460	0.0342	0.3600	0.0537	0.7520	0.0776	1.2580	0.1003
2000	0	0	0.1640	0.0362	0.3280	0.0512	0.6100	0.0699	1.1700	0.0967
2250	0	0	0.1520	0.0349	0.3640	0.0540	0.6200	0.0704	0.9980	0.0894
2500	0	0	0.1740	0.0373	0.2920	0.0483	0.6400	0.0716	0.9080	0.0852
2750	0	0	0.1520	0.0349	0.3080	0.0496	0.6840	0.0740	1.0140	0.0901
3000	0	0	0.1440	0.0339	0.3440	0.0525	0.5720	0.0676	0.9640	0.0878
3250	0	0	0.1440	0.0339	0.2880	0.0480	0.5800	0.0681	0.9560	0.0875
3500	0	0	0.1580	0.0356	0.3440	0.0525	0.6640	0.0729	1.1020	0.0939
3750	0	0	0.1500	0.0346	0.3420	0.0523	0.5880	0.0686	1.0740	0.0927
4000	0	0	0.1500	0.0346	0.3140	0.0501	0.6780	0.0736	1.1600	0.0963
4250	0	0	0.1160	0.0305	0.3580	0.0535	0.6180	0.0703	1.1100	0.0942
4500	0	0	0.1380	0.0332	0.3320	0.0515	0.7140	0.0756	0.9760	0.0884
4750	0	0	0.1700	0.0369	0.3620	0.0538	0.6300	0.0710	0.9280	0.0862
5000	0	0	0.1580	0.0356	0.3760	0.0548	0.6520	0.0722	1.2120	0.0985

Table 4.2.9 Bell-tree insertion statistics AVERAGE RECURSE ROTATION

tree size	k 2		3		4		5		6	
	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0	0	0.2140	0.0414	0.3920	0.0560	0.7560	0.0778	1.6320	0.1143
500	0	0	0.1880	0.0388	0.4560	0.0604	0.7000	0.0748	1.4200	0.1066
750	0	0	0.2060	0.0406	0.3860	0.0556	0.7620	0.0781	1.2440	0.0998
1000	0	0	0.1700	0.0369	0.4460	0.0597	0.6900	0.0743	1.3080	0.1023
1250	0	0	0.2220	0.0421	0.4060	0.0570	0.7880	0.0794	1.0080	0.0898
1500	0	0	0.1740	0.0373	0.4380	0.0592	0.6860	0.0741	1.2100	0.0984
1750	0	0	0.3820	0.0553	0.4380	0.0592	0.8540	0.0827	1.3920	0.1055
2000	0	0	0.2220	0.0421	0.3920	0.0560	0.6900	0.0743	1.2740	0.1010
2250	0	0	0.1880	0.0388	0.4320	0.0588	0.7240	0.0761	1.0820	0.0930
2500	0	0	0.2320	0.0431	0.3680	0.0543	0.7200	0.0759	0.9860	0.0888
2750	0	0	0.1960	0.0396	0.3600	0.0537	0.7760	0.0788	1.1260	0.0949
3000	0	0	0.1960	0.0396	0.4140	0.0575	0.7020	0.0749	1.0680	0.0924
3250	0	0	0.1860	0.0386	0.3700	0.0544	0.6700	0.0732	1.0400	0.0912
3500	0	0	0.0543	0.1940	0.4240	0.0582	0.7600	0.0780	1.1960	0.0978
3750	0	0	0.2020	0.0402	0.4240	0.0582	0.6840	0.0740	1.1900	0.0976
4000	0	0	0.1960	0.0396	0.3840	0.0554	0.7600	0.0780	1.2440	0.0998
4250	0	0	0.1640	0.0362	0.4280	0.0585	0.7140	0.0756	1.2220	0.0989
4500	0	0	0.2020	0.0402	0.4080	0.0571	0.8100	0.0805	1.0540	0.0918
4750	0	0	0.2080	0.0408	0.4460	0.0597	0.7360	0.0767	0.9980	0.0894
5000	0	0	0.2040	0.0404	0.4520	0.0601	0.7540	0.0777	1.3220	0.1028

Table 4.2.10 Bell-tree insertion statistics AVERAGE TOTAL ROTATION

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0.2840	0.0477	0.5700	0.0675	0.9340	0.0864	1.6240	0.1140	3.3760	0.1643
500	0.2900	0.0482	0.6340	0.0712	1.0300	0.0908	1.5220	0.1103	2.9580	0.1538
750	0.3120	0.0500	0.5960	0.0691	0.9640	0.0878	1.6560	0.1151	2.5960	0.1441
1000	0.2620	0.0458	0.5740	0.0678	1.0440	0.0914	1.5220	0.1103	2.7500	0.1483
1250	0.2880	0.0480	0.6920	0.0744	0.9760	0.0884	1.6800	0.1159	2.1360	0.1307
1500	0.3180	0.0504	0.5520	0.0665	1.0300	0.0908	1.5020	0.1096	2.5360	0.1424
1750	0.3360	0.0518	0.5680	0.0674	1.0180	0.0902	1.8660	0.1222	2.8620	0.1513
2000	0.2460	0.0444	0.6620	0.0728	0.9280	0.0862	1.5540	0.1115	2.6260	0.1449
2250	0.2640	0.0460	0.5940	0.0689	1.0700	0.0925	1.5500	0.1114	2.2960	0.1355
2500	0.2700	0.0465	0.6300	0.0710	0.9400	0.0867	1.5460	0.1112	2.1000	0.1296
2750	0.2680	0.0463	0.5820	0.0682	0.8880	0.0843	1.6320	0.1143	2.3660	0.1376
3000	0.2900	0.0482	0.6000	0.0693	0.9780	0.0885	1.5480	0.1113	2.2520	0.1342
3250	0.3080	0.0496	0.5880	0.0686	0.8920	0.0845	1.4300	0.1070	2.2480	0.1341
3500	0.2860	0.0478	0.5620	0.0671	0.9860	0.0888	1.6240	0.1140	2.4960	0.1413
3750	0.2620	0.0458	0.6280	0.0709	1.0040	0.0896	1.5180	0.1102	2.4820	0.1409
4000	0.2780	0.0472	0.5920	0.0688	0.9180	0.0857	1.6380	0.1145	2.6420	0.1454
4250	0.3000	0.0490	0.5540	0.0666	1.0120	0.0900	1.5860	0.1126	2.5640	0.1432
4500	0.2600	0.0456	0.5800	0.0681	0.9680	0.0880	1.7100	0.1170	2.3000	0.1356
4750	0.3040	0.0493	0.5860	0.0685	1.0320	0.0909	1.5700	0.1121	2.1440	0.1310
5000	0.2580	0.0454	0.6120	0.0700	1.0200	0.0903	1.6240	0.1140	2.7400	0.1481

Table 4.2.11 Bell-tree insertion statistics AVERAGE SHIFT

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0	0	1.000	0	2.0960	0.0264	3.3620	0.0757	5.1860	0.2166
500	0	0	1.000	0	2.1280	0.0299	3.3660	0.0778	5.0420	0.1938
750	0	0	1.000	0	2.1120	0.0282	3.4000	0.0842	4.8360	0.1788
1000	0	0	1.000	0	2.1380	0.0309	3.3720	0.0790	4.9060	0.1761
1250	0	0	1.000	0	2.1220	0.0293	3.4040	0.0795	4.6740	0.1569
1500	0	0	1.000	0	2.1360	0.0307	3.3240	0.0748	4.8780	0.1734
1750	0	0	1.000	0	1.1300	0.0301	3.4340	0.0842	4.9340	0.1905
2000	0	0	1.000	0	2.1080	0.0279	3.3360	0.0729	4.8560	0.1822
2250	0	0	1.000	0	2.1300	0.0301	3.3560	0.0745	4.7060	0.1534
2500	0	0	1.000	0	2.1120	0.0282	3.3820	0.0770	4.6280	0.1410
2750	0	0	1.000	0	2.0840	0.0248	3.4040	0.0840	4.7520	0.1591
3000	0	0	1.000	0	2.1280	0.0299	3.3500	0.0766	4.7240	0.1538
3250	0	0	1.000	0	2.1060	0.0275	3.3480	0.0735	4.7120	0.1476
3500	0	0	1.000	0	2.1180	0.0289	3.4000	0.0793	4.8520	0.1777
3750	0	0	1.000	0	2.1240	0.0295	3.3440	0.0760	4.7660	0.1626
4000	0	0	1.000	0	2.1100	0.0280	3.4540	0.0855	4.8420	0.1784
4250	0	0	1.000	0	2.1160	0.0287	3.3480	0.0794	4.8320	0.1753
4500	0	0	1.000	0	2.1100	0.0280	3.4240	0.0842	4.6940	0.1558
4750	0	0	1.000	0	2.1400	0.0312	3.3700	0.0741	4.6660	0.1515
5000	0	0	1.000	0	2.1340	0.0305	3.3920	0.0792	4.8780	0.1789

Table 4.2.12 Bell-tree insertion statistics AVERAGE RECURSE

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	1.0000	0	2.3760	0.0669	3.7900	0.1375	5.7600	0.2947	9.3580	0.7373
500	1.0000	0	2.3340	0.0642	3.9489	0.1577	5.6840	0.2938	8.7860	0.6443
750	1.0000	0	2.3400	0.0628	3.8240	0.1482	5.8460	0.3198	8.2080	0.5982
1000	1.0000	0	2.2800	0.0581	3.9540	0.1588	5.6640	0.2863	8.4320	0.6061
1250	1.0000	0	2.4040	0.0696	3.8660	0.1465	5.8680	0.3034	7.6020	0.5265
1500	1.0000	0	2.2960	0.0602	3.9500	0.1509	5.6100	0.2964	8.2120	0.5779
1750	1.0000	0	2.3320	0.0637	3.9280	0.1553	6.0400	0.3267	8.5840	0.6423
2000	1.0000	0	2.3860	0.0673	3.8280	0.1448	5.6360	0.2724	8.3000	0.6345
2250	1.0000	0	2.3400	0.0651	3.9260	0.1559	5.7000	0.2890	7.7860	0.5262
2500	1.0000	0	2.4060	0.0687	3.7720	0.1428	5.7420	0.2780	7.5220	0.4774
2750	1.0000	0	2.3480	0.0652	3.7520	0.1337	5.8640	0.3152	7.8920	0.5368
3000	1.0000	0	2.3400	0.0641	3.8860	0.1506	5.6240	0.2930	7.7560	0.5151
3250	1.0000	0	2.3300	0.0639	3.7640	0.1345	5.5980	0.2806	7.7080	0.4070
3500	1.0000	0	2.3520	0.0660	3.8860	0.1420	5.8240	0.2997	8.1500	0.5948
3750	1.0000	0	2.3520	0.0651	3.8900	0.1472	5.6160	0.2886	8.0300	0.5604
4000	1.0000	0	2.3460	0.0650	3.8080	0.1407	5.8920	0.3126	8.2460	0.6132
4250	1.0000	0	2.2800	0.0589	3.9020	0.1500	5.6800	0.3118	8.1640	0.5983
4500	1.0000	0	2.3400	0.0633	3.8500	0.1403	5.9480	0.3256	7.7240	0.5425
4750	1.0000	0	2.3780	0.0679	3.9480	0.1471	5.7360	0.2763	7.5920	0.5082
5000	1.0000	0	2.3620	0.0662	3.9620	0.1553	5.7980	0.2962	8.4120	0.6154

Table 4.2.13 Bell-tree deletion statistics AVERAGE SEARCH

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	8.2480	0.2234	7.5460	0.1733	7.3780	0.1403	7.1920	0.1380	6.9480	0.1270
500	9.3680	0.1788	8.6240	0.2086	8.3100	0.1480	8.1760	0.1370	8.1000	0.1304
750	10.1000	0.2294	9.3460	0.1819	8.9260	0.1478	8.6820	0.1377	8.7520	0.1273
1000	10.6580	0.2439	9.6940	0.1751	9.3220	0.1461	9.0160	0.1503	9.1420	0.1259
1250	10.8900	0.2404	10.1860	0.1883	9.7360	0.1548	9.6140	0.1257	9.4120	0.1325
1500	11.2660	0.2525	10.2560	0.1795	9.8060	0.1709	9.7720	0.1373	9.6700	0.1304
1750	11.6860	0.3692	10.7640	0.1679	10.2420	0.1745	10.0300	0.1356	9.8600	0.1354
2000	12.0800	0.7063	10.9840	0.4718	10.3860	0.2079	10.3220	0.1683	10.0860	0.1297
2250	11.5880	0.2742	10.8980	0.1912	10.5080	0.1564	10.4320	0.1301	10.2760	0.1324
2500	11.9900	0.2376	11.1600	0.1814	10.5780	0.1601	10.5500	0.1342	10.5280	0.1398
2750	12.1360	0.2485	11.2500	0.1821	10.7520	0.1537	10.7000	0.1362	10.5820	0.1483
3000	13.7140	1.3862	11.4580	0.1502	11.2280	0.3728	10.7580	0.1347	10.6180	0.1549
3250	12.7680	0.2192	11.9340	0.2322	11.1040	0.1401	10.9720	0.1247	10.8300	0.1237
3500	12.9480	0.2407	12.0540	0.5978	11.2100	0.1627	10.9940	0.1867	10.9300	0.1300
3750	13.0100	0.5124	11.8740	0.1720	11.5000	0.1880	11.1560	0.1302	11.0700	0.1362
4000	12.6040	0.2341	11.9360	0.4144	11.2080	0.1548	11.3180	0.1548	10.9780	0.1443
4250	13.3700	0.6109	11.9080	0.1811	11.6280	0.1976	11.1940	0.1470	11.1560	0.1345
4500	12.5720	0.2514	11.7420	0.1988	11.4860	0.1679	11.1680	0.1515	11.2260	0.1431
4750	12.3460	0.3220	11.6680	0.2114	11.1160	0.2102	11.2800	0.1643	11.2900	0.1471
5000	9.5880	0.5280	9.6820	0.4053	10.6220	0.2814	10.9860	0.2220	11.2960	0.1833

?

Table 4.2.14 Bell-tree deletion statistics AVERAGE NOSONS

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0.9960	0.0893	1.1500	0.0959	1.1940	0.0977	1.2440	0.0998	1.4140	0.1064
500	1.0040	0.0896	1.1860	0.0974	1.2300	0.0992	1.1740	0.0969	1.3120	0.1024
750	0.9660	0.0879	1.1440	0.0957	1.2520	0.1001	1.2560	0.1002	1.2760	0.1010
1000	1.2960	0.1018	1.1100	0.0942	1.2320	0.0993	1.2460	0.0998	1.4440	0.1075
1250	1.0280	0.0907	1.0660	0.0923	1.1760	0.0970	1.2340	0.0994	1.3000	0.1020
1500	1.0760	0.0928	1.0980	0.0937	1.1480	0.0958	1.2600	0.1004	1.3340	0.1033
1750	1.5300	0.1106	1.0720	0.0926	1.3060	0.1022	1.2360	0.0994	1.3260	0.1030
2000	1.4060	0.1061	1.0340	0.0910	1.2720	0.1009	1.3260	0.1030	1.3660	0.1045
2250	1.0340	0.0910	1.0980	0.0937	1.1740	0.0969	1.2580	0.1003	1.2460	0.0998
2500	1.0080	0.0898	1.0680	0.0924	1.1480	0.0958	1.2260	0.0990	1.2720	0.1009
2750	0.9480	0.0871	1.0780	0.0929	1.1620	0.0964	1.2300	0.0992	1.2860	0.1014
3000	0.9100	0.0853	1.0460	0.0915	1.1800	0.0972	1.1740	0.0969	1.2600	0.1004
3250	0.9080	0.0852	0.9480	0.0871	1.1560	0.0962	1.2440	0.0998	1.1920	0.0977
3500	0.9540	0.0874	1.0460	0.0915	1.0940	0.0936	1.2440	0.0998	1.2700	0.1008
3750	0.9780	0.0885	1.0220	0.0904	1.1580	0.0962	1.1660	0.0966	1.2460	0.0998
4000	0.9540	0.0874	1.0580	0.0920	1.0400	0.0912	1.2400	0.0996	1.3180	0.1027
4250	0.9600	0.0876	1.0300	0.0908	1.1020	0.0939	1.2260	0.0990	1.3080	0.1023
4500	0.9340	0.0864	1.0220	0.0904	1.1800	0.0972	1.1660	0.0966	1.2900	0.1016
4750	0.8920	0.0845	1.0520	0.0917	1.1180	0.0946	1.1820	0.0972	1.2980	0.1019
5000	0.8780	0.0838	1.0820	0.0930	1.0760	0.0928	1.1660	0.0966	1.1420	0.0956

Table 4.2.15 Bell-tree deletion statistics AVERAGE ONESON

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0.3000	0.4090	0.3160	0.0503	0.3100	0.0498	0.2940	0.0485	0.3080	0.0496
500	0.3180	0.0504	0.3100	0.0498	0.3300	0.0514	0.3220	0.0508	0.2780	0.0472
750	0.9660	0.0879	0.2940	0.0485	0.3080	0.0496	0.3040	0.0493	0.2780	0.0472
1000	0.3460	0.0526	0.3260	0.0511	0.3360	0.0518	0.3360	0.0518	0.2820	0.0475
1250	0.2920	0.0483	0.2920	0.0483	0.2820	0.0475	0.2780	0.0472	0.2700	0.0465
1500	0.2900	0.0482	0.2920	0.0483	0.3060	0.0495	0.2900	0.0482	0.3040	0.0493
1750	0.2780	0.0472	0.3180	0.0504	0.2940	0.0485	0.3040	0.0493	0.2820	0.0475
2000	0.3060	0.0495	0.3100	0.0498	0.3120	0.0500	0.2980	0.0488	0.3120	0.0500
2250	0.3000	0.0490	0.3280	0.0512	0.3240	0.0509	0.2660	0.0461	0.2920	0.0483
2500	0.2840	0.0477	0.3200	0.0506	0.3000	0.0498	0.3120	0.0500	0.2980	0.0488
2750	0.2940	0.0485	0.3100	0.0498	0.3020	0.0492	0.2860	0.0478	0.3020	0.0492
3000	0.3000	0.0490	0.3280	0.0512	0.3000	0.0490	0.2880	0.0480	0.1004	0.3180
3250	0.2860	0.0478	0.3240	0.0509	0.3140	0.0501	0.3180	0.0504	0.3020	0.0492
3500	0.2960	0.0487	0.2760	0.0470	0.2820	0.0475	0.2900	0.0482	0.3100	0.0498
3750	0.2740	0.0468	0.2800	0.0473	0.2680	0.0463	0.2840	0.0477	0.3060	0.0495
4000	0.2740	0.0468	0.2840	0.0477	0.3640	0.0540	0.2740	0.0468	0.3080	0.0496
4250	0.2940	0.0485	0.2760	0.0578	0.3160	0.0503	0.2640	0.0460	0.3080	0.0496
4500	0.3120	0.0500	0.2880	0.0480	0.2760	0.0470	0.2840	0.0477	0.2560	0.0453
4750	0.3220	0.0508	0.2460	0.0444	0.2780	0.0472	0.2420	0.0440	0.2520	0.0449
5000	0.3120	0.0500	0.2420	0.0440	0.2600	0.0456	0.2360	0.0435	0.2840	0.0477

Table 4.2.16 Bell-tree deletion statistics AVERAGE RECURSE

k	2		3		4		5		6	
tree size	AV	CI	AV	CI	AV	CI	AV	CI	AV	CI
250	0.2960	0.0487	0.4660	0.0611	0.5040	0.0635	0.5380	0.0656	0.7220	0.0760
500	0.3220	0.0508	0.4960	0.0630	0.5600	0.0669	0.4960	0.0630	0.5900	0.0687
750	0.2840	0.0477	0.4380	0.0592	0.5600	0.0669	0.5600	0.0669	0.5540	0.0666
1000	0.6420	0.0717	0.4360	0.0591	0.5680	0.0674	0.5820	0.0682	0.7260	0.0762
1250	0.3200	0.0506	0.3580	0.0535	0.4580	0.0605	0.5120	0.0640	0.5700	0.0675
1500	0.3660	0.0541	0.3900	0.0559	0.4540	0.0603	0.5500	0.0663	0.6380	0.0714
1750	0.8080	0.0804	0.3900	0.0559	0.6000	0.0693	0.5400	0.0657	0.6080	0.0697
2000	0.7120	0.0755	0.3440	0.0525	0.5840	0.0684	0.6240	0.0707	0.6780	0.0736
2250	0.3340	0.0517	0.4260	0.0584	0.4980	0.0631	0.5240	0.0647	0.5380	0.0656
2500	0.2920	0.0483	0.3880	0.0557	0.4480	0.0599	0.5380	0.0656	0.5700	0.0675
2750	0.2420	0.0440	0.3880	0.0557	0.4640	0.0609	0.5160	0.0642	0.5880	0.0686
3000	0.2100	0.0410	0.3740	0.0547	0.4800	0.0620	0.4620	0.0608	0.5780	0.0680
3250	0.1940	0.0394	0.2720	0.0466	0.4700	0.0613	0.5620	0.0671	0.4940	0.0629
3500	0.2500	0.0447	0.3220	0.0508	0.3760	0.0548	0.5340	0.0654	0.5800	0.0681
3750	0.2520	0.0449	0.3020	0.0492	0.4260	0.0584	0.4500	0.0600	0.5520	0.0665
4000	0.2280	0.0427	0.3420	0.0523	0.4040	0.0569	0.5140	0.0641	0.6260	0.0708
4250	0.2540	0.0451	0.3060	0.0495	0.4180	0.0578	0.4900	0.0626	0.6160	0.0702
4500	0.2460	0.0444	0.3100	0.0498	0.4560	0.0604	0.4500	0.0600	0.5460	0.0661
4750	0.2140	0.0414	0.2980	0.0488	0.3960	0.0563	0.4240	0.0582	0.5500	0.0663
5000	0.1900	0.0390	0.3240	0.0509	0.3360	0.0518	0.4020	0.0567	0.4260	0.0584

Table 4.2.17 Bell-tree deletion statistics PROBABILITY NOSONS

tree size \ k	2	3	4	5	6
250	0.4320	0.4380	0.4640	0.4760	0.4140
500	0.4480	0.4560	0.4600	0.4560	0.4540
750	0.4400	0.4780	0.4620	0.4400	0.4880
1000	0.4100	0.4580	0.4040	0.4060	0.4480
1250	0.4260	0.4520	0.4720	0.4560	0.4300
1500	0.4420	0.4360	0.4380	0.4740	0.4620
1750	0.4780	0.4540	0.4720	0.4680	0.4380
2000	0.4480	0.4520	0.4460	0.4620	0.4500
2250	0.4300	0.4340	0.4160	0.4640	0.4680
2500	0.4480	0.4400	0.4080	0.4520	0.5080
2750	0.4500	0.4240	0.4040	0.4580	0.4800
3000	0.4620	0.4160	0.4780	0.4360	0.3960
3250	0.4900	0.4740	0.4240	0.4520	0.4420
3500	0.4800	0.4520	0.4820	0.4300	0.4520
3750	0.4540	0.4180	0.5140	0.4140	0.4680
4000	0.3960	0.4160	0.4120	0.4800	0.4260
4250	0.4340	0.4480	0.4640	0.4300	0.4340
4500	0.3680	0.3940	0.4780	0.4180	0.4560
4750	0.3920	0.4260	0.4120	0.4400	0.4400
5000	0.3060	0.3540	0.3600	0.4520	0.4480

Table 4.2.18 Bell-tree deletion statistics PROBABILITY ONESON

tree size \ k	2	3	4	5	6
250	0.1480	0.0920	0.1000	0.0860	0.0900
500	0.1400	0.1140	0.1040	0.1040	0.0920
750	0.1460	0.0940	0.0820	0.1040	0.0940
1000	0.1500	0.1100	0.1240	0.1200	0.0980
1250	0.1080	0.1260	0.1080	0.0940	0.1100
1500	0.1120	0.1100	0.1080	0.0860	0.0800
1750	0.1160	0.1140	0.0880	0.0900	0.1120
2000	0.1380	0.1440	0.1020	0.0960	0.1060
2250	0.1240	0.1440	0.1180	0.1080	0.1000
2500	0.1140	0.0920	0.1160	0.1240	0.1000
2750	0.1200	0.1280	0.0900	0.0980	0.0920
3000	0.1480	0.1220	0.1120	0.1080	0.1240
3250	0.1340	0.1640	0.1200	0.1100	0.1160
3500	0.1500	0.1140	0.1060	0.0920	0.1120
3750	0.1240	0.1380	0.1100	0.1020	0.1100
4000	0.1380	0.0920	0.1200	0.1080	0.0920
4250	0.1340	0.1140	0.0960	0.0960	0.0880
4500	0.1300	0.1140	0.1000	0.0940	0.0720
4750	0.1320	0.0700	0.0900	0.0720	0.0600
5000	0.1040	0.0540	0.0960	0.0640	0.1100

Table 4.2.19 Bell-tree deletion statistics PROBABILITY TWOSONS

tree size \ k	2	3	4	5	6
250	0.4200	0.4700	0.4360	0.4380	0.4960
500	0.4120	0.4300	0.4360	0.4400	0.4540
750	0.4140	0.4280	0.4560	0.4560	0.4180
1000	0.4400	0.4320	0.4720	0.4740	0.4540
1250	0.4660	0.4220	0.4200	0.4500	0.4600
1500	0.4460	0.4540	0.4540	0.4400	0.4580
1750	0.4060	0.4320	0.4400	0.4420	0.4500
2000	0.4140	0.4040	0.4520	0.4420	0.4440
2250	0.4460	0.4220	0.4660	0.4280	0.4320
2500	0.4380	0.4680	0.4760	0.4240	0.3920
2750	0.4300	0.4480	0.5060	0.4440	0.4280
3000	0.3900	0.4620	0.4100	0.4560	0.4800
3250	0.3760	0.3620	0.4560	0.4380	0.4420
3500	0.3700	0.4340	0.4120	0.4780	0.4360
3750	0.4220	0.4440	0.3760	0.4840	0.4220
4000	0.4660	0.4920	0.4680	0.4120	0.4820
4250	0.4320	0.4380	0.4400	0.4740	0.4780
4500	0.5020	0.4920	0.4220	0.4880	0.4720
4750	0.4760	0.5040	0.4980	0.4880	0.5000
5000	0.5900	0.5920	0.5440	0.4840	0.4420

Table 4.2.20 BB-tree insertion statistics AVERAGE SEARCH

α	$\frac{1}{7}$		$\frac{1}{5}$		$\frac{1}{4}$		$\frac{29}{100}$	
tree size	AV	CI	AV	CI	AV	CI	AV	CI
250	9.8820	0.1604	9.4840	0.1239	9.3420	0.0929	9.1820	0.0717
500	10.9240	0.1688	10.4520	0.1302	10.3500	0.1077	10.1680	0.0768
750	11.6700	0.1638	11.3040	0.1302	11.0620	0.1009	10.7820	0.0792
1000	12.1560	0.1698	11.7200	0.1313	11.4120	0.1071	11.2800	0.0794
1250	12.4000	0.1673	11.9980	0.1354	11.7080	0.1050	11.5680	0.0809
1500	12.6400	0.1748	12.2540	0.1314	12.0560	0.1029	11.7800	0.0820
1750	13.0520	0.1775	12.5100	0.1401	12.3020	0.1100	12.0760	0.0798
2000	13.2600	0.1880	12.8240	0.1512	12.5160	0.1142	12.2600	0.0838
2250	13.4920	0.1778	13.0540	0.1434	12.7120	0.1085	12.4460	0.0824
2500	13.6220	0.1806	13.1800	0.1417	12.7760	0.1140	12.5920	0.0825
2750	13.7200	0.1863	13.2300	0.1443	12.9720	0.1115	12.7320	0.0875
3000	13.9300	0.1833	13.4120	0.1452	13.1100	0.1188	12.8600	0.0876
3250	13.9960	0.1864	13.4620	0.1468	13.2440	0.1216	13.0740	0.0881
3500	14.2380	0.1869	13.7180	0.1472	13.3660	0.1241	13.0040	0.0849
3750	14.2480	0.1969	13.7680	0.1525	13.4860	0.1149	13.1820	0.0905
4000	14.3660	0.1902	13.7780	0.1467	13.4140	0.1136	13.3200	0.0907
4250	14.3440	0.1878	13.9100	0.1507	13.5120	0.1156	13.2740	0.0853
4500	14.4560	0.2007	13.9900	0.1510	13.6980	0.1191	13.4780	0.0938
4750	14.6560	0.1847	14.1080	0.1488	13.7980	0.1195	13.5760	0.0904
5000	14.7420	0.1975	14.2060	0.1503	13.8700	0.1210	13.6040	0.0866

Table 4.2.21 BB-tree insertion statistics AVERAGE SINGLE ROTATION

α	$\frac{1}{7}$		$\frac{1}{5}$		$\frac{1}{4}$		$\frac{29}{100}$	
tree size	AV	CI	AV	CI	AV	CI	AV	CI
250	0.0900	0.0268	0.1300	0.0322	0.1580	0.0356	0.3280	0.0512
500	0.0700	0.0237	0.1200	0.0310	0.1660	0.0364	0.2960	0.0487
750	0.0600	0.0219	0.1400	0.0335	0.1820	0.0382	0.2660	0.0461
1000	0.0780	0.0250	0.1200	0.0310	0.1620	0.0360	0.2680	0.0463
1250	0.0860	0.0262	0.1460	0.0342	0.1780	0.0377	0.2920	0.0483
1500	0.0700	0.0237	0.1400	0.0335	0.1700	0.0369	0.2780	0.0472
1750	0.0840	0.0259	0.1200	0.0310	0.1480	0.0344	0.3060	0.0495
2000	0.0600	0.0219	0.1440	0.0339	0.1920	0.0392	0.2980	0.0488
2250	0.1080	0.0294	0.1520	0.0349	0.1640	0.0362	0.3080	0.0496
2500	0.0640	0.0226	0.1280	0.0320	0.1460	0.0342	0.2680	0.0463
2750	0.1000	0.0283	0.1240	0.0315	0.1700	0.0369	0.2820	0.0475
3000	0.0820	0.0256	0.1100	0.0297	0.1480	0.0344	0.3080	0.0496
3250	0.0760	0.0247	0.0980	0.0280	0.2140	0.0414	0.3160	0.0503
3500	0.0900	0.0268	0.1320	0.0325	0.1780	0.0377	0.2660	0.0461
3750	0.0840	0.0259	0.1160	0.0305	0.1800	0.0379	0.2640	0.0460
4000	0.1200	0.0310	0.1180	0.0307	0.1700	0.0369	0.2980	0.0488
4250	0.0800	0.0253	0.1240	0.0315	0.1480	0.0344	0.2860	0.0478
4500	0.0840	0.0259	0.1060	0.0291	0.1820	0.0382	0.2660	0.0461
4720	0.0940	0.0274	0.1220	0.0312	0.1860	0.0386	0.3160	0.0503
5000	0.0960	0.0277	0.1040	0.0288	0.1780	0.0377	0.2840	0.0477

Table 4.2.22 BB-tree insertion statistics AVERAGE DOUBLE ROTATION

α	$\frac{1}{7}$		$\frac{1}{5}$		$\frac{1}{4}$		$\frac{29}{100}$	
tree size	AV	CI	AV	CI	AV	CI	AV	CI
250	0.0140	0.0106	0.0500	0.0200	0.0720	0.0240	0.0560	0.0212
500	0.0180	0.0120	0.0300	0.0155	0.0420	0.0183	0.0640	0.0226
750	0.0100	0.0089	0.0500	0.0200	0.0800	0.0253	0.0660	0.0230
1000	0.0120	0.0098	0.0380	0.0174	0.0700	0.0237	0.0640	0.0226
1250	0.0100	0.0089	0.0420	0.0183	0.0680	0.0233	0.0800	0.0253
1500	0.0220	0.0133	0.0280	0.0150	0.0820	0.0256	0.0620	0.0223
1750	0.0100	0.0089	0.0340	0.0165	0.0660	0.0230	0.0560	0.0212
2000	0.0080	0.0080	0.0360	0.0170	0.0640	0.0226	0.0480	0.0196
2250	0.0080	0.0080	0.0260	0.0144	0.0780	0.0250	0.0520	0.0204
2500	0.0080	0.0080	0.0320	0.0160	0.0680	0.0233	0.0720	0.0240
2750	0.0180	0.0120	0.0460	0.0192	0.0720	0.0240	0.0620	0.0223
3000	0.0120	0.0098	0.0320	0.0160	0.0880	0.0265	0.0520	0.0204
3250	0.0200	0.0126	0.0240	0.0139	0.0700	0.0237	0.0820	0.0256
3500	0.0100	0.0089	0.0280	0.0150	0.0700	0.0237	0.0620	0.0223
3750	0.0080	0.0080	0.0400	0.0179	0.0760	0.0247	0.0580	0.0215
4000	0.0200	0.0126	0.0420	0.0183	0.0640	0.0226	0.0820	0.0256
4250	0.0040		0.0380	0.0174	0.0620	0.0223	0.0580	0.0215
4500	0.0100	0.0089	0.0240	0.0139	0.0700	0.0237	0.0520	0.0204
4750	0.0140	0.0106	0.0180	0.0120	0.0920	0.0271	0.0620	0.0223
5000	0.0080	0.0080	0.0260	0.0144	0.0620	0.0223	0.0840	0.0259

Table 4.2.23 BB-tree insertion statistics AVERAGE GENERAL ROTATION

α	$\frac{1}{7}$		$\frac{1}{5}$		$\frac{1}{4}$		$\frac{29}{100}$	
tree size	AV	CI	AV	CI	AV	CI	AV	CI
250	0.1040	0.0288	0.1800	0.0379	0.2300	0.0429	0.3840	0.0554
500	0.0880	0.0265	0.1500	0.0346	0.2080	0.0408	0.3600	0.0537
750	0.0700	0.0237	0.1900	0.0390	0.2620	0.0458	0.3320	0.0515
1000	0.0900	0.0268	0.1580	0.0356	0.2320	0.0431	0.3320	0.0515
1250	0.0960	0.0277	0.1880	0.0388	0.2460	0.0444	0.3720	0.0546
1500	0.0920	0.0271	0.1680	0.0367	0.2520	0.0449	0.3400	0.0522
1750	0.0940	0.0274	0.1540	0.0351	0.2140	0.0414	0.3620	0.0538
2000	0.0680	0.0233	0.1800	0.0379	0.2560	0.0453	0.3460	0.0526
2250	0.1160	0.0305	0.1780	0.0377	0.2420	0.0440	0.3600	0.0537
2500	0.0720	0.0240	0.1600	0.0358	0.2140	0.0414	0.3400	0.0522
2750	0.1180	0.0307	0.1700	0.0369	0.2420	0.0440	0.3440	0.0525
3000	0.0940	0.0274	0.1420	0.0337	0.2360	0.0435	0.3600	0.0537
3250	0.0960	0.0277	0.1220	0.0312	0.2840	0.0477	0.3980	0.0564
3500	0.1000	0.0283	0.1600	0.0358	0.2480	0.0445	0.3280	0.0512
3750	0.0920	0.0271	0.1560	0.0353	0.2560	0.0453	0.3220	0.0508
4000	0.1400	0.0335	0.1600	0.0358	0.2340	0.0433	0.3800	0.0551
4250	0.0840	0.0259	0.1620	0.0360	0.2100	0.0410	0.3440	0.0525
4500	0.0940	0.0274	0.1300	0.0322	0.2520	0.0449	0.3180	0.0504
4750	0.1080	0.0294	0.1400	0.0335	0.2780	0.0472	0.3780	0.0550
5000	0.1040	0.0288	0.1300	0.0322	0.2400	0.0438	0.3680	0.0543

Table 4.2.24 BB-tree deletion statistics AVERAGE SEARCH

α	$\frac{1}{7}$		$\frac{1}{5}$		$\frac{1}{4}$		$\frac{29}{100}$	
tree size	AV	CI	AV	CI	AV	CI	AV	CI
250	7.7240	0.1807	7.4460	0.1579	7.3020	0.1461	7.2320	0.1361
500	8.8440	0.1790	8.5920	0.1655	8.4520	0.1506	8.1660	0.1473
750	9.4300	0.1874	9.1040	0.1639	8.9020	0.1535	8.7280	0.1441
1000	9.8480	0.1996	9.4480	0.1772	9.2800	0.1613	9.2360	0.1411
1250	10.2160	0.1937	9.8500	0.1771	9.7320	0.1528	9.4240	0.1443
1500	10.5800	0.2034	10.2760	0.1764	10.0220	0.1489	9.8220	0.1513
1750	10.8600	0.1989	10.5420	0.1716	10.2800	0.1493	9.9320	0.1521
2000	11.0860	0.2009	10.7360	0.1735	10.5480	0.1718	10.2220	0.1492
2250	11.1740	0.2035	10.7740	0.1773	10.5760	0.1597	10.5100	0.1436
2500	11.3900	0.2106	10.9680	0.1811	10.7620	0.1551	10.5540	0.1359
2750	11.6660	0.2083	11.2780	0.1822	10.9780	0.1566	10.7540	0.1433
3000	11.8780	0.2181	11.4240	0.1834	11.1760	0.1667	10.9500	0.1411
3250	12.3780	0.2037	11.8120	0.1715	11.4740	0.1466	11.2400	0.1326
3500	12.0900	0.1911	11.6420	0.1642	11.3920	0.1455	11.3140	0.1218
3750	12.1400	0.0264	11.6900	0.1739	11.4580	0.1484	11.2660	0.1348
4000	12.3300	0.2040	11.8180	0.1732	11.4900	0.1528	11.3220	0.1371
4250	12.2820	0.2190	11.7820	0.1831	11.6140	0.1572	11.3080	0.1573
4500	12.1860	0.2267	11.7880	0.1995	11.5820	0.1689	11.2640	0.1547
4750	11.7220	0.2557	11.3660	0.2266	11.1380	0.2040	11.0720	0.1777
5000	9.1660	0.5248	8.8860	0.4923	8.6960	0.4550	8.8360	0.4038

Table 4.2.25 BB-tree deletion statistics AVERAGE SINGLE ROTATION

α	$\frac{1}{7}$		$\frac{1}{5}$		$\frac{1}{4}$		$\frac{29}{100}$	
tree size	AV	CI	AV	CI	AV	CI	AV	CI
250	0.0340	0.0165	0.0320	0.0160	0.0540	0.0208	0.1540	0.0351
500	0.0320	0.0160	0.0320	0.0160	0.0540	0.0208	0.1100	0.0297
750	0.0260	0.0144	0.0260	0.0144	0.0560	0.0212	0.1400	0.0335
1000	0.0220	0.0133	0.0400	0.0179	0.0700	0.0237	0.1360	0.0330
1250	0.0260	0.0144	0.0640	0.0226	0.0920	0.0271	0.1140	0.0302
1500	0.0120	0.0098	0.0260	0.0144	0.0460	0.0192	0.1280	0.0320
1750	0.0160	0.0113	0.0340	0.0165	0.0400	0.0179	0.1080	0.0294
2000	0.0300	0.0155	0.0680	0.0233	0.0740	0.0243	0.1140	0.0302
2250	0.0220	0.0133	0.0360	0.0170	0.0480	0.0196	0.1220	0.0312
2500	0.0240	0.0139	0.0400	0.0179	0.0580	0.0215	0.0940	0.0274
2750	0.0280	0.0150	0.0420	0.0183	0.0720	0.0240	0.1040	0.0288
3000	0.0300	0.0155	0.0520	0.0204	0.0520	0.0204	0.1220	0.0312
3250	0.0240	0.0139	0.0460	0.0192	0.0400	0.0179	0.1060	0.0291
3500	0.0300	0.0155	0.0540	0.0208	0.0880	0.0265	0.1340	0.0327
3750	0.0220	0.0133	0.0460	0.0192	0.0760	0.0247	0.1140	0.0302
4000	0.0360	0.0170	0.0700	0.0237	0.0640	0.0226	0.1340	0.0327
4250	0.0500	0.0200	0.0560	0.0212	0.0900	0.0268	0.1240	0.0315
4500	0.0480	0.0196	0.0640	0.0226	0.0700	0.0237	0.1400	0.0335
4750	0.0600	0.0219	0.0860	0.0262	0.0980	0.0280	0.1380	0.0332
5000	0.0740	0.0243	0.1140	0.0302	0.1180	0.0307	0.1520	0.0349

Table 4.2.26 BB-tree deletion statistics AVERAGE DOUBLE ROTATION

tree size	α = $\frac{1}{7}$ AV	CI	$\frac{1}{5}$ AV	CI	$\frac{1}{4}$ AV	CI	$\frac{29}{100}$ AV	CI
250	0.0020		0.0020		0.0180	0.0120	0.0660	0.0230
500	0	0	0.0040		0.0180	0.0120	0.0820	0.0256
750	0.0020		0	0	0.0120	0.0098	0.0640	0.0226
1000	0	0	0	0	0.0100	0.0089	0.0640	0.0226
1250	0	0	0.0040		0.0120	0.0098	0.0600	0.0219
1500	0.0020		0	0	0.0180	0.0120	0.0660	0.0230
1750	0	0	0.0040		0.0180	0.0120	0.0640	0.0226
2000	0	0	0.0040		0.0160	0.0113	0.0740	0.0243
2250	0	0	0.0040		0.0160	0.0113	0.0540	0.0208
2500	0	0	0	0	0.0060		0.0880	0.0265
2750	0	0	0	0	0.0140	0.0106	0.0480	0.0196
3000	0.0020		0.0040		0.0160	0.0113	0.0740	0.0243
3250	0.0020		0.0020		0.0140	0.0106	0.0740	0.0243
3500	0	0	0.0080	0.0080	0.0180	0.0120	0.0860	0.0262
3750	0.0020	0.0040	0.0040		0.0200	0.0126	0.0560	0.0212
4000	0	0	0.0100	0.0089	0.0200	0.0126	0.0720	0.0240
4250	0.0060		0.0080	0.0080	0.0420	0.0183	0.0780	0.0250
4500		0.0057	0.0200	0.0126	0.0440	0.0188	0.0820	0.0256
4750	0.0060		0.0080	0.0080	0.0280	0.0150	0.0940	0.0274
5000	0.0100	0.0089	0.0160	0.0113	0.0380	0.0174	0.0860	0.0262

Table 4.2.27 BB-tree deletion statistics AVERAGE GENERAL ROTATION

α	$\frac{1}{7}$		$\frac{1}{5}$		$\frac{1}{4}$		$\frac{29}{100}$	
tree size	AV	CI	AV	CI	AV	CI	AV	CI
250	0.0360	0.0170	0.0340	0.0165	0.0720	0.0240	0.2200	0.0420
500	0.0320	0.0160	0.0360	0.0170	0.0720	0.0240	0.1920	0.0392
750	0.0280	0.0150	0.0260	0.0144	0.0680	0.0233	0.2040	0.0404
1000	0.0220	0.0133	0.0400	0.0179	0.0800	0.0253	0.2000	0.0400
1250	0.0260	0.0144	0.0680	0.0233	0.1040	0.0288	0.1740	0.0373
1500	0.0140	0.0106	0.0260	0.0144	0.0640	0.0226	0.1940	0.0394
1750	0.0160	0.0113	0.0380	0.0174	0.0580	0.0215	0.1720	0.0371
2000	0.0300	0.0155	0.0720	0.0240	0.0900	0.0268	0.1880	0.0388
2250	0.0220	0.0133	0.0400	0.0179	0.0640	0.0226	0.1760	0.0375
2500	0.0240	0.0139	0.0400	0.0179	0.0640	0.0226	0.1820	0.0382
2750	0.0280	0.0150	0.0420	0.0183	0.0860	0.0262	0.1520	0.0359
3000	0.0320	0.0160	0.0560	0.0212	0.0680	0.0233	0.1960	0.0396
3250	0.0260	0.0144	0.0480	0.0196	0.0540	0.0208	0.1800	0.0379
3500	0.0300	0.0155	0.0620	0.0223	0.1060	0.0291	0.2200	0.0420
3750	0.0240	0.0139	0.0500	0.0200	0.0960	0.0277	0.1700	0.0369
4000	0.0360	0.0170	0.0800	0.0253	0.0840	0.0259	0.2060	0.0406
4250	0.0560	0.0212	0.0640	0.0226	0.1320	0.0325	0.2020	0.0402
4500	0.0520	0.0204	0.0840	0.0259	0.1140	0.0302	0.2220	0.0421
4750	0.0660	0.0230	0.0940	0.0274	0.1260	0.0317	0.2320	0.0431
5000	0.0840	0.0259	0.1300	0.0322	0.1560	0.0353	0.2380	0.0436

Table 4.2.28

Central processor timing on statistical runs

The AVL tree construction algorithm

5356

The Bell-tree construction algorithm

k	2	3	4	5	6
	7443	8483	10209	12399	15327

The BB-tree construction algorithm

α	$\frac{1}{7}$	$\frac{1}{5}$	$\frac{1}{4}$	$\frac{29}{100}$
	8056	7999	8007	8051

4.3 EMPIRICAL OBSERVATIONS ON ALGORITHMS OF CLASS (2)

4.3.1 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 THE BRUNO COFFMAN ALGORITHM

Empirical tests on this algorithm on all five sets of α data and with β 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 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 CONCLUSIONS

Table 4.3.4 gives a synopsis of the results for the above three algorithms using the five sets of α_i data and the one set of β_i data common to all three (β 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.

		α set 1	α set 2	α set 3	α set 4	α set 5
β case	1	4.2944	4.6317	4.0035	5.0362	4.1379
	2	6.6060	7.2555	6.4230	7.4164	5.9726
	3	5.9633	6.5382	5.6524	6.5494	5.3331
	4	5.8250	6.4549	5.5935	6.7508	5.4117
	5	6.2576	6.5810	6.0850	6.7320	5.8119
	6	7.0856	7.1216	7.0125	7.1119	6.9904

Table 4.3.2

Bruno Coffman statistics

α set 1

blocksize	original avg. weighted path length	final avg. weighted path length	time/sec.
25	5.1321	4.4373	3.273
50	5.5000	4.3785	2.908
75	8.0752	4.4016	3.030
100	8.1326	5.0757	2.796
125	8.2382	4.4759	2.712
150	9.1943	4.3775	2.887
175	9.0984	4.3211	3.012
200	8.3014	4.4768	2.907

α set 2

blocksize	original avg. weighted path length	final avg. weighted path length	time/sec.
25	5.3740	4.7738	3.058
50	7.1026	4.9079	2.951
75	8.3166	4.8190	2.939
100	8.5040	5.0509	3.071
125	7.4120	4.8102	2.702
150	8.6599	4.8126	2.716
175	8.7423	5.0315	2.956
200	7.7876	5.0024	2.639

Table 4.3.2 (cont'd.)

α set 3

blocksize	original avg. weighted path length	final avg. weighted path length	time/sec.
25	5.7218	4.2422	2.759
50	4.9510	4.2174	2.871
75	9.3177	4.0772	3.081
100	7.3108	4.1671	2.831
125	8.2943	4.3815	3.073
150	9.1884	4.0299	3.046
175	8.7127	4.0864	2.992
200	8.1593	4.0729	3.336

α set 4

blocksize	original avg. weighted path length	final avg. weighted path length	time/sec.
25	6.2303	5.4557	2.851
50	7.5322	5.6956	2.627
75	7.2206	5.3076	2.749
100	7.2021	5.3382	2.752
125	8.9968	5.3696	2.887
150	7.5701	5.4517	2.770
175	8.7005	5.4364	2.752
200	8.8301	5.2778	2.680

Table 4.3.2 (cont'd.)

α set 5

blocksize	original avg. weighted path length	final avg. weighted path length	time/sec.
25	5.3241	4.4918	2.936
50	5.8825	4.4775	2.669
75	7.1315	4.3974	2.776
100	8.0506	4.1899	2.853
125	9.5555	4.5832	3.328
150	8.2572	4.2681	2.748
175	8.0172	4.3381	2.907
200	8.5678	4.3194	2.672

Table 4.3.3

WALKER GOTLIEB STATISTICS

		$N_0 = 0$, $F = 1$	time (sec.)	$N_0 = 15$, $F = 3$	time (sec.)	$N_0 = 15$, $F = 4$	time (sec.)	$N_0 = 15$, $F = 5$	time (sec.)	$N_0 = 15$, $F = 6$	time (sec.)	$N_0 = 15$, $F = 10^6$	time (sec.)
<u>α set 1</u>													
β Case	1	4.7270	0.723	4.3635	0.451	4.3635	0.431	4.3708	0.440	4.4103	0.441	4.4682	0.426
	2	7.6283	0.719	6.6653	0.478	6.6809	0.430	6.6812	0.469	6.6812	0.419	6.8741	0.402
	3	7.0949	0.677	5.9964	0.432	6.0404	0.448	6.0777	0.441	6.0777	0.418	0.2380	0.435
	4	6.7102	0.677	5.8256	0.431	5.8256	0.430	5.8280	0.441	0.5870	0.420	5.9994	0.408
	5	7.8798	0.676	6.3582	0.473	6.3699	9.467	6.3793	0.454	6.3780	0.440	6.5602	0.437
	6	9.8989	0.699	7.2961	0.451	7.2746	0.472	7.3114	0.461	7.3512	0.440	7.2489	0.401
<u>α set 2</u>													
β Case	1	5.0299	0.675	4.7052	0.433	4.7003	0.460	4.6721	0.441	4.7795	0.485	4.7738	0.439
	2	8.1425	0.657	7.2749	0.470	7.2602	0.481	7.2685	0.490	7.2685	0.449	7.4535	0.438
	3	7.6646	0.679	6.5712	0.473	6.5632	0.465	6.5793	0.440	6.6759	0.489	6.7689	0.392
	4	7.2468	0.679	6.4670	0.451	6.4638	0.462	6.4630	0.477	6.4831	0.440	6.6469	0.417
	5	8.4314	0.679	6.7982	0.472	6.8593	0.460	6.7082	0.438	6.7288	0.436	7.0154	0.392
	6	9.6699	0.638	7.4320	0.498	7.4623	0.462	7.4606	0.438	7.4161	0.438	7.3423	0.445
<u>α set 3</u>													
β Case	1	4.7786	0.829	4.0512	0.482	4.0547	0.482	4.0547	0.440	4.1393	0.442	4.0655	0.409
	2	8.1525	0.769	6.4587	0.449	6.4747	0.440	6.4759	0.485	6.5035	0.441	6.4868	0.412
	3	7.1084	0.761	5.7214	0.486	5.7149	0.445	5.7560	0.451	5.7656	0.439	5.9927	0.437
	4	7.1817	0.799	5.6676	0.442	5.7259	0.480	5.7231	0.453	5.7313	0.458	5.7873	0.398
	5	9.2180	0.771	6.3468	0.482	6.3225	0.448	6.2860	0.480	6.2445	0.479	6.3284	0.443
	6	11.1363	0.768	7.2884	0.482	7.2805	0.485	7.2344	0.410	7.2344	0.458	7.2099	0.438

Table 4.3.3 (cont'd.)

		$N_0 = 0$, $F = 1$	time (sec.)	$N_0 = 15$, $F = 3$	time (sec.)	$N_0 = 15$, $F = 4$	time (sec.)	$N_0 = 15$, $F = 5$	time (sec.)	$N_0 = 15$, $F = 6$	time (sec.)	N_0 15, $F = 10^6$	time (sec.)
α set 4													
β Case	1	5.9324	0.733	5.1312	0.451	5.0572	0.481	5.0572	0.440	5.1860	0.420	5.1079	0.407
	2	8.8770	0.694	7.4975	0.480	7.4609	0.465	7.4606	0.457	7.4908	0.440	7.5341	0.446
	3	7.6391	0.699	6.6411	0.476	6.5627	0.458	6.5627	0.457	6.5892	0.468	6.6194	0.407
	4	8.4449	0.693	6.8272	0.465	6.7954	0.431	6.8179	0.463	6.7669	0.461	6.9337	0.434
	5	8.7564	0.737	6.9841	0.472	6.9473	0.458	6.9595	0.460	6.9377	0.434	7.1285	0.393
	6	10.4205	0.694	7.3783	0.454	7.3833	0.459	7.3798	0.458	7.3798	0.445	7.3373	0.413
α set 5													
β Case	1	4.8235	0.781	4.1841	0.451	4.1815	0.493	4.2028	0.435	4.2060	0.433	4.2735	0.393
	2	7.2296	0.720	6.0324	0.452	6.0523	0.450	6.1668	0.437	6.1718	0.434	6.1478	0.388
	3	6.5523	0.707	5.3779	0.452	5.3672	0.451	5.3995	0.464	5.5638	0.434	5.5450	0.429
	4	6.6474	0.721	5.4449	0.451	5.4284	0.451	5.4284	0.435	5.4522	0.465	5.7755	0.404
	5	7.2683	0.713	5.8673	0.451	5.8935	0.483	5.9134	0.469	5.9896	0.472	5.9578	0.397
	6	10.3917	0.727	7.2463	0.492	7.2023	0.435	7.1713	0.472	7.1620	0.465	7.1881	0.424

Table 4.3.4

	Knuth	Walker Gotleib		Bruno Coffman	
	avg. weighted path length	avg. weighted path length	time/sec	avg. weighted path length	time/sec
q_i set 1	4.2944	4.3635	0.431	4.3785	2.908
2	4.6317	4.7003	0.460	4.9079	2.951
3	4.0035	4.0547	0.482	4.2174	2.871
4	5.0362	5.0572	0.481	5.6956	2.627
5	4.1379	4.1815	0.493	4.4775	2.669

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

Adel'son-Vel'skii, G.M., and Landis, Y.M. (1962). An algorithm for the organization of information, Doklady Akad, Nauk USSR Moscos 16, 2, pp. 263-266.

Arora, S.R., and Dent, W.T. (1969). Randomized binary search technique, Communications of ACM 12, 2, pp. 77-80.

Bayer, R. (1972). Symmetric binary B-trees: data structure and maintenance algorithms, Acta Informatica 1, pp. 290-306.

Bayer, R., and McCreight, E. (1972). Organization and maintenance of large ordered indexes, Acta Informatica 1, pp. 173-189.

Bell, C.J. (1965). An investigation into the principles of the classification and analysis of data on an automatic digital computer, Doctoral Dissertation, Leeds University.

Booth, A.D., and Colin, A.J.T. (1960). On the efficiency of a new method of dictionary construction, Information and Control 3, pp. 327-334.

Bruno, J., and Coffman, E.G., Jr. (1971). Nearly optimum binary search trees IFIP Congress 71, TA-2, pp. 29-32.

Burge, W.H. (1958). Sorting, trees and measures of order, Information and Control 1, pp. 181-197.

Clampett, H.A., Jr. (1964). Randomized binary searching with tree structures, Communications of the ACM 7, 3, pp. 163-165.

Crane, C.A. (1972). Linear lists and priority queues as balanced binary trees, Doctoral Dissertation, Stanford University Computer Science Report 259.

Douglas, A.S. (1959). Techniques for the recording of, and reference to data in a computer, Computer Journal 1, pp. 1-9.

Foster, C.C. (1965a). Information storage and retrieval using AVL trees, Proceeding ACM 20th National Conference, pp. 192-205.

Foster, C.C. (1965b). A study of AVL trees, GER-12158, Goodyear Aerospace Corp., Akron, Ohio.

Foster, C.C. (1973). A generalization of AVL trees, Communications of the ACM 16, 8, pp. 513-517.

Fredkin, E. (1960). Trie memory, Communications of the ACM 3, pp. 490-499.

Gilbert, E.N., and Moore, E.F. (1959). Variable-length binary encoding, Bell System Technical Journal 38, pp. 933-968.

Helbich, J. (1969). Direct selection of keywords for the KWIC index, Information, Storage, Retrieval 5, pp. 123-128.

Hibbard, T.H. (1962). Some combinatorial properties of certain trees with applications to searching and sorting, Journal ACM 9, pp. 13-28.

Hu, T.C. (1972a). A comment on the double-chained tree, Communications of the ACM 15, 4, pp. 276.

Hu, T.C. (1972b). A new proof of the T-C algorithm, University of Wisconsin Mathematics Research Center, Report 1207.

Hu. T.C., and Tan, K.C. (1972a). Path length of binary search trees, SIAM Journal of Applied Math. 22, 2, pp. 225-234.

Hu, T.C., and Tan, K.C. (1972b). Least upper bound on the cost of optimum binary search trees, Acta Informatica 1, pp. 307-310.

Hu, T.C., and Tucker, A.C. (1970). Optimum binary search trees, University of Wisconsin, Mathematics Research Center, Report 1049.

Hu, T.C., and Tucker, A.C. (1971). Optimal computer search trees and variable-length aphabetic codes, SIAM Journal of Applied Math. 21, 4, pp. 514-532.

Huffman, D.A. (1952). A method for the construction of minimum-redundancy codes, Proceeding IRE, 40, pp. 1098-1101.

Karp, R.M. (1961). Minimum-redundancy coding for the discrete noiseless channel, IRE Transactions on Information Theory, IT-7, pp. 27-38.

Kennedy, S.R. (1972a). Optimal weighted doubley-chained trees, California Institute of Tech., Information Science Report 1.

Kennedy, S.R. (1972b). A note on optimal doubly-chained trees, Communications of the ACM 15, 11, pp. 997-998.

Knott, G.D. (1971). A balanced tree storage and retrieval algorithm, Proceedings Symposium on Information Storage and Retrieval, University of Maryland, pp. 175-196.

Knuth, D.E. (1968). The art of computer programming volume 1: fundamental algorithms, Addison-Wesley Publishing Co., Reading, Massachusetts.

Knuth, D.E. (1971). Optimum binary search trees, Acta Informatica 1, pp. 14-25.

Knuth, D.E. (1973). The art of computer programming, Volume 3: sorting and searching, Addison-Wesley Publishing Co., Reading, Massachusetts.

Landauer, W.I. (1963). The balanced tree and its utilization in information retrieval, IEEE Transactions on Electronic Computers EC-12, 6, pp. 863-871.

Lynch, W.C. (1965). More combinatorial properties of certain trees, Computer Journal 1, pp. 299-302.

Martin, W.A., and Ness, D.N. (1972). Optimizing binary trees grown with a sorting algorithm, Communications of the ACM 15, 2, pp. 88-93.

Muntz, R., and Uzgalis, R. (1970). Dynamic storage allocation for binary search trees in a two-level memory, Proceeding 4th Annual Princeton Conference, Princeton, New Jersey.

Nievergelt, J. (1972). Binary search trees and file organization, SIGFIDET Workshop, Denver, Colorado.

Nievergelt, J., and Pradels, J. (1972). Bounds on the weighted path length of binary trees, Information Processing Letters 1, 6, pp. 220-225.

Nievergelt, J., and Reingold, E.M. (1972). Binary search trees of bounded balance, Proceeding 4th Annual ACM Symposium on Theory of Computing, Denver, Colorado, pp. 137-142.

Nievergelt, J., and Wong, C.K. (1971). On the weighted path length of binary search trees, IBM T.J. Watson Research Center, New York, Report 3562.

Nievergelt, J., and Wong, C.K. (1973). Upper bounds for the total path length of binary trees, Journal ACM 20, 1, pp. 1-6.

Overholt, K.J. (1973). Optimal binary search methods, BIT 13, pp. 84-91.

Patt, Y.N. (1969). Variable length tree structures having minimum average search time, Communications of the ACM 12, 2, pp. 72-80.

Patt, Y.N. (1972). Minimum search tree structures for data partitioned into pages, IEEE Transactions on Computers C-21, 9, pp. 961-967.

Reingold, E.M. (1971). Notes on AVL trees, Department of Computer Science, University of Illinois.

Rissanen, J. (1973). Bounds for binary search trees, IBM Journal of Research and Development 17, 2, pp. 101-105.

Scroggs, R.E., Karlton, P.L., Fuller, S.H. and Kaehler, E.B. (1973). Observations on the performance of AVL trees, Department of Computer Science, Carnegie-Mellon University, Pittsburgh.

Stanfel, L.E. (1969). A comment on optimal tree structures, Communications of the ACM 12, 10, pp. 582.

Stanfel, L.E. (1970). Tree structures for optimal searching, Journal of the ACM 17, 3, pp. 508-517.

Stanfel, L.E. (1972). Practical aspects of doubly-chained trees for retrieval, Journal of the ACM 19, 3, pp. 425-436.

Stanfel, L.E. (1973). Optimal trees for a class of information retrieval problems, Information Storage and Retrieval 9, pp. 43-59.

Sussenguth, E.H., Jr. (1963). Use of tree structures for processing files, Communications of the ACM 6, 5, pp. 272-279.

Tan, K.C. (1972). On Foster's information storage and retrieval using AVL trees, Communications of the ACM 15, 9, pp. 843.

Walker, W.A. and Gotlieb, C.C. (1972a). A top-down algorithm for constructing nearly optimal lexicographic trees, in Read, R.C. (ed.), Graph Theory and Computing, Academic Press, New York, pp. 303-323.

Walker, W.A. and Gotlieb, C.C. (1972b). Hybrid trees: a data structure for lists of keys, in Dean, A.L. (ed.), Association for Computing Machinery, New York, pp. 189-211.

Walker, A.N., Redish, K.A. and Wood D. (1973). A simple binary tree display algorithm, Computer Science Technical Report 73/10, Mc McMaster University, Hamilton, Ontario.

Walker, A.N. and Wood, D. (1974). On bell-trees, their implementation and properties, Computer Science Technical Report 74/4, McMaster University, Hamilton, Ontario.

Weiner, P. (1971). On the heuristic design of binary search trees, Proceeding of the 5th Annual Princeton Conference, Princeton, New Jersey.

Windley, P.F. (1960). Trees, forests and rearranging, Computer Journal 3, pp. 84-88.

Wirth, N. (1971). The programming language, PASCAL, Acta Informatica 1, pp. 35-63.

Wong, C.K. and Chang, S. (1971). The generation and balancing of binary search trees, IBM T.J. Watson Research Center, New York, Report 3667.

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 A1-1 the representations are

(a) nested sets (see Figure A1-2),

(b) nested parentheses

(D(B(A)(C)) (E(H(F)(I)))) and

(c) indentation (see Figure A1-3).

All three representations give the same <u>partial</u> 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 <u>ordering problem</u>. 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 n 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 opague. 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

ENDORDER 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 ENDORDER 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.

<u>Definition</u>

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. <u>Binary Tree Display Algorithm - DISPLAY (root, indent, nodeline, width)</u>

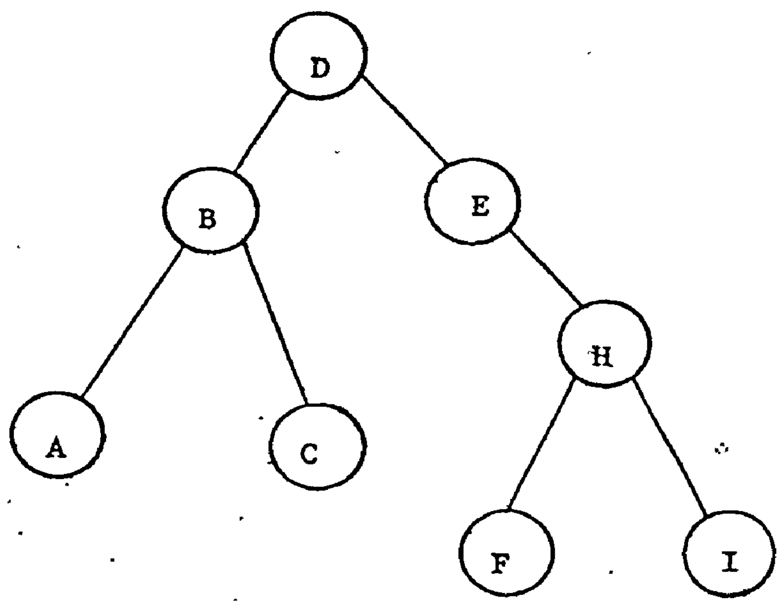
The implementation of this algorithm fulfills the following two conditions:

(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.

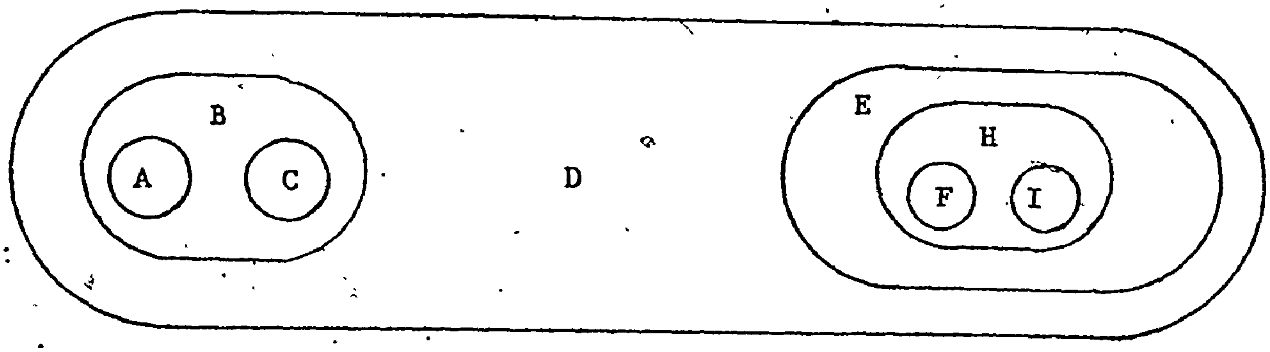
Given these conditions a reverse postorder traversal must be used. However, the traversal algorithm can easily be changed to suit one's individual taste. As the traversal proceeds down the tree a variable called level, is incremented by one at each level of the tree. The variable level is used, when the visit routine is invoked, to determine the indentation of the key to be printed (although not explicitly evaluated the indentation is equal to level x indent). This establishes an indented linear ordering of the tree much like a freehand drawing (see Figure A1-4 to A1-7). The structure is easily discernible from the display. 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 represneting 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 A1-4 to A1-7, 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 (initally *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.



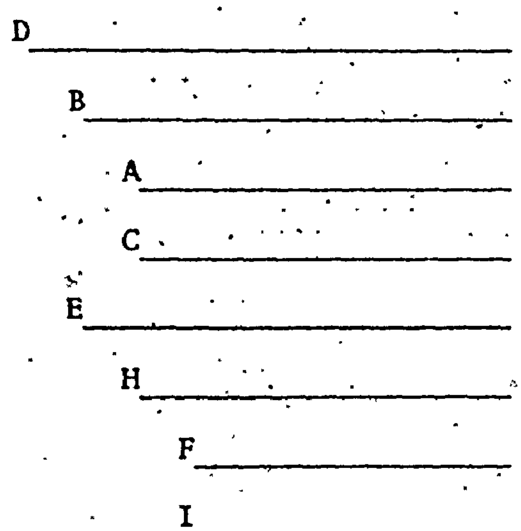
Example of binary tree

<u>Figure A1-1</u>



Nested sets representation

Figure A1-2



Indentation representation

Figure A1-3

```
THERE ARE            8    NODES IN THIS TREE
THE ROOT OF THIS TREE IS DDDDD


INDENT =             8   WIDTH =            4  NODELINE =          1
                        ------ IIIII
                        :
                        :
width                   :
                        :
               ------ HHHHH
               :        :
               :        :
               :        :
               :        :
               :        ------ FFFFF
               :
               :         indent
               :
        ------ EEEEE
        :
        :              nodeline
        :
DDDDD
        :
        :
        :
        :
        :      ------ CCCCC
        :      :
        :      :
        :      :
        ------ BBBBB
               :
               :
               :
               ------ AAAAA
```

Figure A1-4

```
THERE ARE           8   NODES IN THIS TREE
THE ROOT OF THIS TREE IS DDDDD


INDENT =            8  WIDTH =          4  NODELINE =          5

                        -- IIIII
                       :
                       :
                       :
                       :
               -- HHHHH
              :        :
              :        :
              :        :
              :        :
              :         -- FFFFF
              :
              :
              :
              :
      -- EEEEE
     :
     :
     :
     :
DDDDD
     :
     :
     :
     :
     :        -- CCCCC
     :       :
     :       :
     :       :
     :       :
      -- BBBBB
              :
              :
              :
              :
               -- AAAAA
```

Figure A1-5

```
THERE ARE            8   NODES IN THIS TREE
THE ROOT OF THIS TREE IS DDDDD


INDENT =             8  WIDTH =             4  NODELINE =             8
                        IIIII
                       :
                       :
                       :
                       :
                HHHHH -
               :       :
               :       :
               :       :
               :       :
               :        FFFFF
               :
               :
               :
               :
        EEEEE -
       :
       :
       :
       :
DDDDD -
       :
       :
       :
       :
       :
       :
       :        CCCCC
       :       :
       :       :
       :       :
       :       :
        BBBBB -
               :
               :
               :
               :
                AAAAA
```

```
THERE ARE          8   NODES IN THIS TREE
THE ROOT OF THIS TREE IS DDDDD


INDENT =           8  WIDTH =          4  NODELINE =          12

                         IIIII
                            :
                            :
                            :
                            :
                HHHHH  -----
                    :       :
                    :       :
                    :       :
                    :       :
                    :    FFFFF
                    :
                    :
                    :
                    :
        EEEEE  -----
            :
            :
            :
DDDDD  -----
            :
            :
            :
            :
            :   CCCCC
            :       :
            :       :
            :       :
        BBBBB  -----
                    :
                    :
                    :
                    :
                 AAAAA
```

Figure A1-7

THE PROCEDURE DISPLAY

```
PROCEDURE DISPLAY(ROOT:POINT;INDENT,WIDTH,NODELINE:INTEGER);
PURPOSE: TO DISPLAY A BINARY TREE IN A READABLE FORMAT

GLOBAL TYPE(S)

     DIRECTION--A SCALAR TYPE USED TO INDICATE THE DIRECTIONS WHICH
                MAY BE FOLLOWED FROM A NODE IE DIRECTION = (RIGHT,L

     POINT------VARIABLES OF THIS TYPE ARE POINTERS TO TREE NODES

LOCAL CONSTANT(S)

     PRINTLIM--THE DIMENSION OF THE BOOLEAN ARRAY PRARRAY. IT INDI(
               THE NUMBER OF LEVELS OF THE BINARY TREE WHICH CAN BE
               PRINTED ON A PAGE AND MUST BE DETERMINED BY THE USE

     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-CHARAC
               SHOULD BE PRINTED FROM A NODE ON A PARTICULAR LEVEL
               IN THE TREE. IF BRPRINT[I] = TRUE A BRANCH-CHARACTE
               MUST BE PRINTED.

     F---------BOOLEAN VARIABLE INDICATING IF:
               TRUE :SEGMENTS 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 = 10;
TYPE
          BRARY = ARRAY[0..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(' ')
END; SPACE
```

```
PROCEDURE PRNTBRANCH(LEVEL:INTEGER;BRPRINT:BRARY);
↱
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 SPACIN
             NEXT POTENTIAL BRANCH POSITION↓

            FOR M := 1 TO LEVEL DO
                IF BRPRINT[M]
                    THEN
                        BEGIN
                            WRITE(≡:≡);
                            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;BRPPINT:BRARY);
↱
PURPOSE: TO PRINT THE KEY OF THE NODE POINTED TO BY P. HOWEVER IT
         ALSO NECESSARY TO (A) PRINT CHARACTERS OF PRECEDING BRANC
         (COLONS) SO AS THEY ARE DISPLAYED AS CONTINUOUS BRANCHES
         (B) PRINT FILLER-CHACTERS (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>I
```

```
(AND BY DEFINITION NODELINE<2*INDENT)+
IF NODELINE GT INDENT
   THEN

      +IF NODE TO BE VISITED IS A SON OF THE ROOT NOD
       (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 BRANC
                CHARACTER POSITION+

               SPACE(NODELINE - 1);

               +PRINT A BRANCH-CHARACTER (BRPRINT[
                TRUE) WITH REQUIRED SPACING TO THE
                NEXT POTENTIAL BRANCH-CHARACTER
                POSITION+

               FOR I := 1 TO LEVEL - 2 DO
                  IF BRPRINT[I]
                     THEN
                        BEGIN
                           WRITE(=:=);
                           SPACE(INDENT - 1)
                        END
                     ELSE SPACE(INDENT);
               IF BRPRINT[LEVEL - 1]
                  THEN
                     BEGIN
                        WRITE(=:=);
                        SPACE(2*INDENT - NODEL
                     END
                  ELSE SPACE(2*INDENT-NODELINE+1
            END
   ELSE
      BEGIN

         +SPACE TO FIRST POTENTIAL BRANCH-CHARACTER
          POSITION+

         SPACE(NODELINE - 1);

         +PRINT A BRANCH-CHARACTER (BRPRINT[I] = TR
          WITH REQUIRED SPACING TO THE NEXT POTENTI
          BRANCH-CHARACTER POSITION+

         FOR I := 1 TO LEVEL - 1 DO
            IF BRPRINT[I]
               THEN
                  BEGIN
                     WRITE(=:=);
                     SPACE(INDENT - 1)
                  END
               ELSE SPACE(INDENT);
```

```
                                        ↑PRINT FILLER-CHARACTERS (MINUS SIGNS) BEFORE
                                         KEY IF NECESSARY↓

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

    ↑PRINT THE KEY↓

    UNPACK(P↑.INFO,CHAY,1);
    I := 1;
    WHILE CHAY[I] GE ≡A≡ 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↓

    10:IF(P↑.LPT NE NIL) OR (P↑.RPT NE NIL)
          THEN
                BEGIN
                      SPACE(1);
                      I := I + 1;
                      FOR J := I TO NODELINE - 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,BRPRINT);
                  CASE WAY OF
                       LEFT :BRPRINT[LEVEL] := FALSE;
                       RIGHT:BRPRINT[LEVEL] := TRUE
                  END;
                  VISIT(P,LEVEL,BRPRINT);
                  TRAVERSE(P↑.LPT,LEVEL + 1,LEFT);
                  IF F THEN
                  PRNTBRANCH(LEVEL,BRPRINT);
             END
END; ↱TRAVERSE↓
↱---------------------------------------------------------------------

BEGIN ↱DISPLAY↓

    ↱INITIALIZE↓

    F := FALSE;

    ↱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+-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.

U T I L I T Y   R O U T I N E S

```
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.(M.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 ELEMEN
             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[1:N] WILL STILL CONTAIN ALL THE ELEMENTS OF
             THE ORIGINAL A[1:N] AND THAT IF K = N SHUFFLE APPLIES
             A RANDOM PERMUTATION TO THE COMPLETE SEQUENCE. (M.C.PYKE
             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[I] := A[J];
            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 .. ALFALENG] OF CHAR;
      I : INTEGER;

BEGIN ↱TIME↓

   ATTENTION(24111500000000000428);
   REC.INT := MEM[423];
   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(CHAY[I]);
   WHILE (INPUT↑ NE ≡ ≡) AND (INPUT↑ NE 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
       CHAY[J] := 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
         END;
   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);
 10:GET(INPUT);

   ↱CREATE THE ALFA VARIABLE WORD↓

   PACK(CHAY,1,WORD);

6789:END; ↱READWORD↓
```

```
PROCEDURE CLASSOVERFLOW;
BEGIN ↱CLASSOVERFLOW↓
    WRITE(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
           ↱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
           END
       ELSE
           ↱TAKE THE NODE FROM THE FREE LIST↓
           BEGIN
               P := FREE;
               FREE := FREE↑.RPT
           END
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

chay ... 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.

```
(* * * * * * * * * * * * * * * * * *(*B A S I C   B I N A R Y   T R E E*)* * * * * * * * * * * * *

                                     (*GLOBAL CONSTANTS,TYPES,VARIABLES*)

LABEL       999;
CONST
            MAX = 10;
            STOPCHAR = '$';
TYPE
            POINT = ↑TREE;
            DIRECTION = (RIGHT,LEFT);
            NODE = RECORD
                              INFO:ALFA;
                              LPT:POINT;
                              RPT:POINT;
                     END;
VAR
            TREE:CLASS 2.0 OF NODE;
            FREE,HEAD,CURRENT,FATHER:POINT;
            WAY:DIRECTION;
            INWORD:ALFA;
            CHAY:ARRAY[1..MAX] OF CHAR;
            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 MAKENODE(WORD:ALFA;P:POINT);
(*
PURPOSE: TO DEFINE THE FIELDS OF A NODE AND INCREMENT NODE COUNTE

PARAMETER(S):

    WORD--KEY OF NEW NEW NODE
    P-----POINTER TO THE NEW NODE

BEGIN (*MAKENODE*)

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

END; (*MAKENODE*)
```

```
FUNCTION SEARCHITEM(WORD:ALFA):BOOLEAN;
PURPOSE: BOOLEAN FUNCTION TO DETERMINE IF THERE IS A NODE IN THE T
         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 TR

    FATHER := HEAD;
    CURRENT := HEAD↑.RPT;
    WAY := RIGHT;

    ↱SEARCH FOR THE KEY↓

    WHILE CURRENT NE NIL DO
        IF CURRENT↑.INFO NE WORD
          THEN
              BEGIN
                  FATHER := CURRENT;
                  IF WORD GT CURRENT↑.INFO
                    THEN
                        BEGIN
                            CURRENT := CURRENT↑.RPT;
                            WAY := RIGHT;
                        END
                    ELSE
                        BEGIN
                            CURRENT := CURRENT↑.LPT;
                            WAY := LEFT;
                        END
              END
          ELSE

              ↱KEY IS PRESENT↓

              BEGIN
                  SEARCHITEM := FALSE;
                  GOTO 10;
              END;

10:END; ↱SEARCHITEM↓
```

```
PROCEDURE BASICINSERT(WORD:ALFA);
↱
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:
             MAKENODE(WORD,CURRENT);

             ↱LINK NEW NODE TO ITS FATHER↓

             CASE WAY OF
                RIGHT:FATHER↑.RPT := CURRENT;
                LEFT :FATHER↑.LPT := CURRENT;
             END:
          END:
END; ↱BASICINSERT↓
↱------------------------------------------------------------------------

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 EQ 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↑.RPT EQ NIL
       THEN
          BEGIN
             TWOSUBTREE := FALSE;
             RELINK(P↑.LPT,P);
             RELEASE(P);
          END
       ELSE
          IF P↑.LPT EQ NIL
             THEN
                BEGIN
                   TWOSUBTREE := FALSE;
                   RELINK(P↑.RPT,P);
                   RELEASE(P);
                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
          ↱DNODE--POINTER TO THE NODE TO BE DELETED
           PRED---POINTER TO THE POSTORDER PREDECESSOR OF THE NODE T
                  BE DELETED↲
    DNODE,PRED:POINT;
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 TWOSUBTREE(CURRENT)
                THEN
                   BEGIN
                      ↱THE NODE TO BE DELETED HAS TWO NON-NULL
                       SUBTREES:REPLACE THE NODE WITH ITS POSTOR
                       PREDECESSOR AND THEN DELETE THIS PREDECES

                      DNODE := CURRENT;
                      PRED := CURRENT;
                      CURRENT := CURRENT↑.LPT;
                      REPEAT
                         FATHER := PRED;
                         PRED := CURRENT;
                         CURRENT := CURRENT↑.RPT;
                      UNTIL (CURRENT EQ NIL);
                      DNODE↑.INFO := PRED↑.INFO;
                      IF TWOSUBTREE(PRED)
                         THEN ;
                   END;
          END;
END; ↱BASICDELETE↲
```

```
BEGIN→DRIVER FOR BASIC BINARY TREE↓
    →PERFORM NECESSARY INITIALIZATIONS↓
    CREATEHEADER;
    →INSERT KEYS INTO THE TREE↓
    WHILE READWORD(INWORD) DO
        BASICINSERT(INWORD);
    →DELETE NODES FROM THE TREE↓
    WHILE READWORD(INWORD) DO
        BASICDELETE(INWORD);
999:END. →DRIVER FOR BASIC BINARY TREE ↓
```

```
* * * * * * * * * * * * * * →A V L  T R E E↓ * * * * * * * * * * * * * * * * * *

        →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↓

LABEL    999;
CONST
         MAX = 10;
         STOPCHAR = E:E:
         MAXDEPTH = 4_;
TYPE     POINT = ↑TREE;
         DIRECTION = (RIGHT,LEFT);
         NODE = RECORD
                   INFO:ALFA;
                   LPT:POINT;
                   RPT:POINT;
                   BALFAC:INTEGER;
               END;
VAR
         TREE:CLASS 2_ OF NODE;
         HEAD,FREE:POINT;
         WAY:DIRECTION;
         INWORD:ALFA;
         CHAY:ARRAY[1..MAX] OF CHAR;
         NODECOUNT:INTEGER;
→* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

PROCEDURE CREATEHEADER;
→
PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS

BEGIN →CREATHEADER↓

    →INITIALIZE FREE LIST POINTER AND NODE COUNTER↓

    NODECOUNT := 0;
    FREE := NIL;

    →CREATE HEADER NODE↓

    IF ACQUIRE(HEAD) THEN GOTO EXIT 999;
    HEAD↑.RPT := NIL;

END; →CREATEHEADER↓
→-----------------------------------------------------------------------

PROCEDURE MAKENODE(WORD:ALFA;P:POINT);
→
PURPOSE: TO CREATE THE FIELDS OF THE NEW NODE AND TO INCREMENT THE
         NODE COUNTER

BEGIN →MAKENODE↓

    NODECOUNT := NODECOUNT + 1;
    P↑.INFO := WORD;

    P↑.BALFAC := 0;
    P↑.RPT := NIL;
    P↑.LPT := NIL;

END; →MAKENODE↓
```

```
PROCEDURE REROOT(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 REROOT

    IF T↑.RPT EQ S
         THEN T↑.RPT := P
         ELSE T↑.LPT := P

END; REROOT
-----------------------------------------------------------------------------------

PROCEDURE SROTATE(T,S,R:POINT:BALVAL:INTEGER);

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

PARAMETERS INPUT:

    BALVAL--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 REVELANT NODES USED IN THE ROTATION

BEGIN SROTATE

    PERFORM THE APPROPRIATE ROTATION

    IF BALVAL EQ -1
         THEN
              BEGIN
                   S↑.LPT := R↑.RPT;
                   R↑.RPT := S
              END
         ELSE
              BEGIN
                   S↑.RPT := R↑.LPT;
                   R↑.LPT := S
              END;

    ADJUST THE BALANCE FACTORS OF THE NODES USED IN THE ROTATION

    S↑.BALFAC := BALVAL -R↑.BALFAC;
    R↑.BALFAC := R↑.BALFAC - BALVAL;

    RELINK THE ROTATED SUBTREE TO ITS FATHER

    REROOT(T,S,R);

END; SROTATE
```

```
PROCEDURE DROTATE(T,S,R:POINT;BALVAL:INTEGER);
PURPOSE: TO PERFORM A DOUBLE ROTATION USING THE GIVEN INPUT POINTERS.
PARAMETERS INPUT:

    BALVAL--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 P:POINT;

BEGIN →DROTATE↓

    →PERFORM THE APPROPRIATE ROTATION↓

    IF BALVAL EQ 1
       THEN
          BEGIN
             P := R↑.LPT;
             R↑.LPT := P↑.RPT;
             P↑.RPT := R;
             S↑.RPT := P↑.LPT;
             P↑.LPT := S
          END
       ELSE
          BEGIN
             P := R↑.RPT;
             R↑.RPT := P↑.LPT;
             P↑.LPT := R;
             S↑.LPT := P↑.RPT;
             P↑.RPT := S
          END;

    →ADJUST THE BALANCE FACTORS OF THE NODES USED IN THE ROTATION↓

    IF P↑.BALFAC EQ 0
       THEN
          BEGIN
             S↑.BALFAC := 0;
             R↑.BALFAC := 0
          END
       ELSE
          BEGIN
             IF P↑.BALFAC EQ BALVAL
                THEN
                   BEGIN
                      S↑.BALFAC := -BALVAL;
                      R↑.BALFAC := 0
                   END
                ELSE
                   BEGIN
                      S↑.BALFAC := 0;
                      R↑.BALFAC := BALVAL
                   END;
             P↑.BALFAC := 0
          END;

    →RELINK THE ROTATED SUBTREE TO ITS FATHER↓

    REROOT(T,S,P);

END; →DROTATE↓
```

```
FUNCTION SEARCHITEM(WORD:ALFA:VAR 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------POINTER TO THE CRITICAL NODE
    FATHCRIT--POINTER TO THE FATHER OF THE CRITICAL NODE
    FATHER----POINTER TO THE FATHER OF THE NODE TO BE CREATED

VAR
        (*CURRENT--POINTER TO THE NODE UNDER CURRENT EXAMINATION*)

    CURRENT:POINT:

BEGIN (*SEARCHITEM*)

    (*INITIALIZE POINTERS FOR SEARCH*)

    SEARCHITEM := TRUE:
    FATHER := HEAD:
    CURRENT := HEAD^.RPT:
    FATHCRIT := HEAD:
    CRIT := CURRENT:
    WAY := RIGHT:
    (*I4 := 1:*)

    (*SEARCH FOR THE KEY*)

    IF CURRENT NE NIL
        THEN
          (*BEGIN*)
            REPEAT
                (*I1 := I1 + 1:*)

                (*CHECK FOR THE CRITICAL NODE*)

                IF CURRENT^.BALFAC NE 0
                    THEN
                        BEGIN
                            (*I4 := I1;*)
                            FATHCRIT := FATHER:
                            CRIT := CURRENT:
                        END:

                (*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^.LPT;
                                        WAY := LEFT
                                    END
                                ELSE
                                    BEGIN
                                        CURRENT := CURRENT^.RPT;
                                        WAY := RIGHT.
                                    END
                        END
```

```
                    ELSE
                        BEGIN
                            SEARCHITEM := FALSE;
                            GOTO 10
                        END;
            UNTIL (CURRENT EQ NIL);
            ↑I1 := I1 + 1;
            I4 := I1 - I4;
          END
        ELSE
            I1 := 1;↓
10:END; ↑SEARCHITEM↓
↑-----------------------------------------------------------------------

PROCEDURE ADJUST(CRIT,NEWNODE:POINT;VAR SONCRIT:POINT;
                 VAR BALVAL: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
    SONCRIT-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 PAT
     THIS NODE IS USED FOR ANY POSSIBLE REBALANCE↓

    IF NEWNODE↑.INFO LT CRIT↑.INFO
        THEN
            BEGIN
                SONCRIT := CRIT↑.LPT;
                BALVAL := -1
            END
        ELSE
            BEGIN
                SONCRIT := CRIT↑.RPT;
                BALVAL := 1
            END;

    ↑ADJUST THE BALANCE FACTORS BETWEEN THE CRITICAL NODE AND THE NEW
     NODE↓

    Q := SONCRIT;
    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
```

```
PROCEDURE BALANCE(FATHCRIT,CRIT,SONCRIT: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
     SONCRIT---POINTER TO THE NODE SUCCEEDING THE CRITICAL NODE IN THE
               SEARCH PATH

BEGIN ↱BALANCE↓

     ↱IF THE CRITICAL NODE WAS BALANCED NO ROTATION IS NECESSARY;READJUS
      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 SONCRIT↑.BALFAC EQ BALVAL
                         THEN
                             BEGIN
                               ↱I2 := I2 + 1:↓
                               SROTATE(FATHCRIT,CRIT,SONCRIT,BALVAL)
                             END
                         ELSE
                             BEGIN
                               ↱I3 := I3 + 1:↓
                               DROTATE(FATHCRIT,CRIT,SONCRIT,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
VAR
        →BALVAL--INTERGER 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
         FATHCRIT-POINTER TO THE FATHER OF THE CRITICAL NODE
         FATHER---POINTER TO THE FATHER OF THE NODE TO BE CREATED
         NEWNODE--POINTER RO THE NEW 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 WAY OF
                 RIGHT:FATHER↑.RPT := NEWNODE;
                 LEFT :FATHER↑.LPT := NEWNODE;
             END;
             MAKENODE(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 NIL
                THEN
                    BEGIN
                        ADJUST(CRIT,NEWNODE,SONCRIT,BALVAL);
                        BALANCE(FATHCRIT,CRIT,SONCRIT,BALVAL);
                    END
          END
END; →AVLINSERT↓
```

```
PROCEDURE AVLDELETE(WORD:ALFA);

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 POINTER TO A NODE IN THE DELETION SEQUENCE
                 DR--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;
                        DR:INTEGER
                END;
        BACK = ARRAY[1..MAXDEPTH] OF DELPT;

VAR
        BACKP--THE DELETION STACK
         P-----POINTER TO THE NODE TO BE DELETED

        BACKP:BACK;
        I:INTEGER;
        P:POINT;
+ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +
```

```
PROCEDURE PREDORSUCC(P:POINT):

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

VAR Q,QQ:POINT;

BEGIN *PREDORSUCC*

    *DETERMINE IF THE NODE IS LEFT OR RIGHT HEAVY.*

    I := I + 2:
    IF P*.BALFAC EQ 1
       THEN

           *FIND THE SUCCESSOR REBUILDING THE STACK IN THE PROCESS*

           BEGIN
               BACKP[I - 1].DP := 1:
               Q := P*.RPT:
               BACKP[I].PT := Q:
               BACKP[I].DP := -1:
               QQ := Q*.LPT:
               WHILE QQ NE NIL DO
                     BEGIN
                         I := I + 1:
                         BACKP[I].PT := QQ:
                         BACKP[I].DP := -1:
                         QQ := QQ*.LPT
                     END
           END
       ELSE

           *FIND THE PREDECESSOR REBUILDING THE STACK IN THE PROCESS*

           BEGIN
               BACKP[I - 1].DP := -1:
               Q := P*.LPT:
               BACKP[I].PT := Q:
               BACKP[I].DP := 1:
               QQ := Q*.RPT:
               WHILE QQ NE NIL DO
                     BEGIN
                         I := I + 1:
                         BACKP[I].PT := QQ:
                         BACKP[I].DP := 1:
                         QQ := QQ*.RPT
                     END
           END
END: *PREDORSUCC*
```

```
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↓

  ↱D5 := D5 + 1;↓
  IF I NE 1 THEN
    WITH BACKP[I] DO
      BEGIN
        IF PT↑.BALFAC EQ DR
          THEN

            ↱THE CURRENT NODE IN THE DELETION STACK WAS LEFT(RIGHT)
             HEAVY AND A NODE WAS DELETED FROM ITS LEFT(RIGHT) SUBTREE
             ITS BALANCE FACTOR BECOMES ZERO.↓

            BEGIN
              PT↑.BALFAC := 0;
              I := I - 1;
              REBUILD
            END
        ELSE

            ↱IF THE CURRENT NODE IN THE DELETION STACK HAD A BALANCE
             FACTOR OF ZERO,THE TREE WILL NOT BE UNBALANCED;ADJUST
             THE BALANCE FACTOR AND EXIT↓

            IF PT↑.BALFAC EQ 0
              THEN PT↑.BALFAC := -DR
              ELSE

                ↱PERFORM THE APPROPRIATE ROTATION↓

                IF PT↑.BALFAC EQ 1
                  THEN
                    IF PT↑.RPT↑.BALFAC EQ 0
                      THEN

                        ↱THE SPECIAL CASE WHERE A SINGLE ROTATION
                         RESTORES BALANCE AND WE MAY EXIT↓

                        BEGIN
                          ↱D4 := D4 + 1;↓
                          SROTATE(BACKP[I-1].PT,PT,PT↑.RPT,1)
                        END
                    ELSE
                        BEGIN
                         IF PT↑.RPT↑.BALFAC EQ -1
                           THEN
                             BEGIN
                               ↱D3 := D3 + 1;↓
                               DROTATE(BACKP[I-1].PT,PT,PT↑.RPT,1)
                             END
                           ELSE
```

```
                    BEGIN
                        →D2 := D2 + 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
                  →D4 := D4 + 1;←
                  SROTATE(BACKP[I-1].PT,PT,PT↑.LPT,-1)
              END
            ELSE
              BEGIN
               IF PT↑.LPT↑.BALFAC EQ 1
                  THEN
                    BEGIN
                        →D3 := D3 + 1;←
                        DROTATE(BACKP[I-1].PT,PT,PT↑.LPT,-1)
                    END
                  ELSE
                    BEGIN
                        →D2 := D2 + 1;←
                        SROTATE(BACKP[I-1].PT,PT,PT↑.LPT,-1)
                    END;
                I := I - 1;
                REBUILD
              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 DR EQ 1
                            THEN PT↑.RPT := P↑.LPT
                            ELSE PT↑.LPT := P↑.LPT;
                        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 DR EQ 1
                                    THEN PT↑.RPT := P↑.RPT
                                    ELSE PT↑.LPT := P↑.RPT;
                                RELEASE(P);
                                REBUILD
                            END
                        ELSE

                            ↱THE NODE TO BE DELETED HAS TWO SONS;
                             FIND ITS POSTORDER PREDECESSOR OR
                             SUCCESSOR↓

                            BEGIN
                                PREDORSUCC(P);
                                P↑.INFO := BACKP[I].PT↑.INFO;
                              RELINK;
                            END
        END
 END; ↱RELINK↓
```

```
BEGIN →AVLDELETE↓

    →INITIALIZE THE DELETION STACK WITH THE HEADER NODE↓

    BACKP[1].PT := HEAD;
    BACKP[1].DR := 1;
    P := HEAD↑.RPT;
    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
            →D1 := D1 + 1;↓
            I := I + 1;
            BACKP[I].PT := P;
            IF P↑.INFO NE WORD
                THEN
                    IF P↑.INFO GT WORD
                        THEN
                            BEGIN
                                BACKP[I].DR := -1;
                                P := P↑.LPT;
                            END
                        ELSE
                            BEGIN
                                BACKP[I].DR := 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↑.RPT := NIL;
                                     RELEASE(P);
                                END;
                        GOTO 10
                    END
        END;

10:END; →AVLDELETE↓

→=====================================================================!

BEGIN →AVL DRIVER↓

    →PERFORM NECESSARY INITIALIZATIONS↓

    CREATEHEADER;

    →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↓
```

```
*B B  T R E E*
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
 *GLOBAL CONSTANTS,TYES,VARIABLES

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

   NUMER,DEMON,DIFF,PROD1,PROD2,PROD3--USED IN INEQUALITIES INVOLVING
                                         ALFA*

LABEL    999;
CONST
         MAX = 10;
         STOPCHAR = ≡:≡;
TYPE
         POINT = ↑TREE;
         DIRECTION = (RIGHT,LEFT);
         NODE = RECORD
                      INFO:ALFA;
                      LPT:POINT;
                      RPT:POINT;
                      SIZE:INTEGER;
         END;
VAR
         TREE:CLASS 2.1 OF NODE;
         HEAD,FREE,CURRENT,FATHER:POINT;
         WAY:DIRECTION;
         INWORD:ALFA;
         CHAY:ARRAY[1..MAX] OF CHAR;
         NODECOUNT:INTEGER;
         NUMER,DENOM,DIFF,PROD1,PROD2,PROD3:INTEGER;
* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
PROCEDURE CREATEHEADER;
↱
PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS
VAR
        ↱ALFA--THE PARAMETER TO THE ALGORITHM↓
   ALFA:REAL;
BEGIN ↱CREATEHEADER↓
    ↱DETERMINE ALFA↓
    READ(NUMER);
    READ(DENOM);
    ALFA := NUMER/DENOM;
    ↱CHECK IF ALFA IS IN RANGE↓
    IF ALFA GT (1.0 - SQRT(2.0)/2.0)
       THEN
          BEGIN
             WRITE(= ALPHA EXCEEDS 1 - SQRT(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↑.RPT := NIL;
    ↱DETERMINE QUANTITIES INVOLVING ALFA IN TERMS OF INTERGERS↓
    DIFF := DENOM - NUMER;
    PROD1 := DIFF*NUMER;
    PROD2 := DENOM*(DENOM - NUMER - NUMER);
    PROD3 := DENOM*DIFF;
END; ↱CREATEHEADER↓
↱------------------------------------------------------------------------
PROCEDURE MAKENODE(WORD:ALFA;P:POINT);
↱
PURPOSE: TO DEFINE THE FIELDS OF A RECORD REPRESENTING A NODE
PARAMETERS INPUT:
    P-----POINTER TO THE NEW NODE
    WORD--KEY OF THE NEW NODE
BEGIN ↱MAKENODE↓
    NODECOUNT := NODECOUNT + 1;
    P↑.INFO := WORD;
    P↑.SIZE := 1;
    P↑.RPT := NIL;
    P↑.LPT := NIL;
END; ↱MAKENODE↓
```

```
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↑.RPT 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↓
   ↱I2 := I2 + 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 DROTATE(PP,P,Q:POINT;BALVAL:DIRECTION);
(*PURPOSE: TO PERFORM A DOUBLE 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 (*DROTATE*)
    (*I3 := I3 + 1;*)
    PSIZE := P↑.SIZE;

    (*PERFORM THE APPROPRIATE ROTATION*)

    IF BALVAL 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 - Q↑.SIZE + S↑.LPT↑.SIZE;

             IF S↑.RPT EQ NIL
                THEN Q↑.SIZE := Q↑.SIZE - S↑.SIZE
                ELSE Q↑.SIZE := Q↑.SIZE - S↑.SIZE + S↑.RPT↑.SIZE;

             (*REBALANCE*)

             Q↑.LPT := S↑.RPT;
             S↑.RPT := Q;
             P↑.RPT := S↑.LPT;
             S↑.LPT := P;
          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;
```

```
                    IF S↑.LPT EQ 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↑.RPT;
                    S↑.RPT  := P;
                END;

    S↑.SIZE := PSIZE;

    ↱THE NODE OF THE ROTATED SUBTREE BECOMES THE NEXT NODE TO EXAMINE↲

    CURRENT := S;

    RELINK(PP,P,S);

END; ↱DROTATE↲
↱------------------------------------------------------------------------

PROCEDURE DUPLICATEKEY(WORD:ALFA);
↱
PURPOSE: TO DECREMENT THE SIZE FIELDS 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 ↱DUPLICATEKEY↲

    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; ↱DUPLICATEKEY↲
```

```
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
                    DUPLICATEKEY(WORD);
                    GOTO 10:
                END;

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

        IF WORD GT CURRENT↑.INFO
            THEN CURRENT := CURRENT↑.RPT
            ELSE CURRENT := CURRENT↑.LPT;
    UNTIL CURRENT EQ NIL;

    ↱INSERT NEW KEY INTO THE TREE↓




    IF ACQUIRE(CURRENT) THEN GOTO EXIT 999;
    MAKENODE(WORD,CURRENT):
    IF WORD LT FATHER↑.INFO
        THEN FATHER↑.LPT := CURRENT
        ELSE FATHER↑.RPT := CURRENT;

10:END: ↱JUSTINSERT↓
```

```
PROCEDURE FINDBALANCE(VAR BALFAC,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:
    BALFAC----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↓

    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;
```

```
                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;
    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 ↑LEFTINSERT↓
    LEFTSIZE := 2;

    ↑COMPUTE THE NEW BALANCE OF CURRENT AFTER INSERTION↓

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

    TREESIZE := CURRENT↑.SIZE + 2;
    BALFAC := LEFTSIZE*DENOM;
    IF(BALFAC GE TREESIZE*NUMER) 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 PROD2*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
         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 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;

    BALFAC := LEFTSIZE*DENOM;
    IF(BALFAC GE TREESIZE*NUMER) 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↑.RPT;
            END

        REBALANCE

        ELSE
            IF CURRENT↑.SIZE EQ 2
                THEN SROTATE(FATHER,CURRENT,CURRENT↑.RPT,RIGHT)
                ELSE

                    BEGIN
                        FINDBALANCE(BALFAC,TREESIZE,WORD,RIGHT);
                        IF BALFAC LT PROD2*TREESIZE
                            THEN SROTATE(FATHER,CURRENT,CURRENT↑.RPT,
                                             RIGHT)
                            ELSE DROTATE(FATHER,CURRENT,CURRENT↑.RPT,
                                             RIGHT);
                    END;
END; RIGHTINSERT
```

```
PROCEDURE BBINSERT(WORD:ALFA);

PURPOSE: TO INSERT A KEY INTO THE TREE

PARAMETERS INPUT:
    WORD--THE KEY TO BE INSERTED

VAR
    P:POINT:
BEGIN ↱BBINSERT↓

    FATHER := HEAD;
    CURRENT := HEAD↑.RPT;
    IF CURRENT EQ NIL
        THEN

            ↱ENTER FIRST KEY↓

            BEGIN
                IF ACQUIRE(CURRENT) THEN GOTO EXIT 999;
                MAKENODE(WORD,CURRENT);
                HEAD↑.RPT := 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 10;
                    END

                  ELSE
                    IF WORD GT CURRENT↑.INFO
                      THEN

                        ↱CHECK FOR CONDITION OF INFINITE LOOP↓

                        BEGIN
                          IF CURRENT↑.SIZE EQ 2
                            THEN

                              IF CURRENT↑.RPT NE NIL
                                THEN

                                  IF CURRENT↑.RPT↑.INFO GT WORD
                                    THEN
```

```
                                        BEGIN
                                         IF ACQUIRE(P) THEN GOTO EXIT 999
                                         MAKENODE(WORD,P);
                                         IF FATHER↑.RPT EQ 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 10;
                  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; ↱33INSERT↓
```

```
PROCEDURE NOSUCHNODE(WORD:ALFA);

PURPOSE: TO INCREMENT THE SIZE FIELDS OF NODES WHICH HAD BEEN
         DECREMENTED 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↑.RPT;
    WHILE CURRENT NE NIL DO
        BEGIN
            CURRENT↑.SIZE := CURRENT↑.SIZE + 1;
            IF WORD GT CURRENT↑.INFO
                THEN CURRENT := CURRENT↑.RPT
                ELSE CURRENT := CURRENT↑.LPT;
            END;
END; ←NOSUCHNODE→
------------------------------------------------------------------------

PROCEDURE DELETENODE(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 PREDOPSUCC(WAY:DIRECTION;VAR 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 SUCESSOR IS SOUGHT

PARAMETERS OUTPUT:
    WORD--THE KEY OF THE PREDECESSOR OR SUCCESSOR

VAR
    P,Q:POINT:.

BEGIN *PREDOPSUCC*

    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:
    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: *PREDOPSUCC*
```

```
PROCEDURE DELETERIGHT;
↱
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

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;
    P:POINT;

BEGIN ↱DELETERIGHT↓

    ↱COMPUTE THE BALANCE OF CURRENT AFTER THE DELETION↓

    LEFTSIZE := 1;
    IF CURRENT↑.LPT NE NIL
        THEN
            LEFTSIZE := CURRENT↑.LPT↑.SIZE + 1;
    TREESIZE := CURRENT↑.SIZE;
    BALFAC := LEFTSIZE*DENOM;
    IF (BALFAC GE TREESIZE*NUMER) 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↑.RPT;
            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↑.LPT↑.LPT NE NIL
                    THEN
                        LEFTSIZE := CURRENT↑.LPT↑.LPT↑.SIZE + 1;
                    BALFAC := (TREESIZE - LEFTSIZE)*PROD3;
                    IF BALFAC LT PROD2*TREESIZE
                        THEN
                            SROTATE(FATHER,CURRENT,CURRENT↑.LPT,LEFT)
                        ELSE
                            DROTATE(FATHER,CURRENT,CURRENT↑.LPT,LEFT);
                CURRENT↑.SIZE := CURRENT↑.SIZE - 1;
                FATHER := CURRENT;
                CURRENT := P;
            END;
END; ↱DELETERIGHT↓
```

```
PROCEDURE DELETELEFT;
↑
PURPOSE: TO DETERMINE IF DELTEION 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;
    P:POINT;

BEGIN ↑DELETELEFT↓

    ↑COMPUTE THE BALANCE OF CURRENT AFTER THE DELETION↓

    LEFTSIZE := 1;
    IF CURRENT↑.LPT NE NIL
       THEN
           LEFTSIZE := CURRENT↑.LPT↑.SIZE;
    TREESIZE := CURRENT↑.SIZE;
    BALFAC := LEFTSIZE*DENOM;
    IF (BALFAC GE TREESIZE*NUMER) 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↑.RPT↑.SIZE + 1;
               LEFTSIZE := 1;

               ↑FIND THE BALANCE OF THE RIGHT SUBTREE OF CURRENT AND
                DETERMINE THE APPROPRIATE ROTATION↓

               IF CURRENT↑.RPT↑.LPT NE NIL
                   THEN
                       LEFTSIZE := CURRENT↑.RPT↑.LPT↑.SIZE + 1;
                   BALFAC := LEFTSIZE*PROD3;
                   IF BALFAC LT PROD2*TREESIZE
                       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 BBDELETE(WORD:ALFA);

↑PURPOSE: TO DELETE THE NODE WITH GIVEN INPUT KEY

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

VAR
        ↑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↓
   NEWWORD:ALFA;
   P:POINT;

BEGIN ↑BBDELETE↓

  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↑.INFO NE WORD
        THEN
          IF WORD GT CURRENT↑.INFO
            THEN DELETERIGHT
            ELSE DELETELEFT
        ELSE

          ↑HAVE LOCATED NODE:CHECK IF NODE IS A LEAF OR SEMI-LEAF↓

          IF CURRENT↑.SIZE EQ 1
            THEN

              ↑NODE IS A LEAF;JUST DELETE↓

              BEGIN
                DELETENODE(NIL);
                GOTO 10;
              END
            ELSE
              IF CURRENT↑.SIZE EQ 2
                THEN

                  ↑NODE IS A SEMI-LEAF↓

                  BEGIN
                    IF CURRENT↑.RPT NE NIL
                      THEN P := CURRENT↑.RPT
                      ELSE P := CURRENT↑.LPT;
                    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
                                  PREDORSUCC(LEFT,NEWWORD)
                                ELSE
                                  PREDORSUCC(RIGHT,NEWWORD);
                            WORD := NEWWORD;
                           END;

  ↱NODE IS NOT IN THE TREE↓

  NOSUCHNODE(WORD);

10:END; ↱BBDELETE↓

↱===========================================================================

BEGIN ↱BB-TREE DRIVER↓

    ↱INITIALIZE↓

    CREATEHEADER;

    ↱INSET KEYS↓

    WHILE READWORD(INWORD) DO
        BBINSERT(INWORD);

    ↱DELETE KEYS↓

    WHILE READWORD(INWORD) DO
        BBDELETE(INWORD);

999:END. ↱BB-TREE DRIVER↓
```

```
(* * * * * * * * * * * * * * * * (*B E L L   T R E E*) * * * * * * * * * * * * * * * * *)

        (*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*)

LABEL     999;
CONST     MAX=10;
          STOPCHAR = '*';
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;
          WAY:DIRECTION;
          INWORD:ALFA;
          CHAY:ARRAY[1..MAX] OF CHAR;
           NODECOUNT,K,KVAL:INTEGER;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE CREATEHEADER;
(*
PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS

VAR
    I:INTEGER;

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↑.RPT := NIL;
    HEAD↑.SIZE := 0;

END; (*CREATEHEADER*)
```

```
PROCEDURE MAKENODE(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 ↱MAKENODE↲

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

END; ↱MAKENODE↲

↱---------------------------------------------------------------------

PROCEDURE NEWNODE(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 ↱NEWNODE↲

    ↱CREATE THE NEW NODE↲

    IF ACQUIRE(S) THEN GOTO EXIT 999;
    MAKENODE(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; ↱NEWNODE↲
```

```
PROCEDURE RELINK(FATHER,OLDROOT,NEWROOT:POINT);
↱
PURPOSE: TO RELINK A ROTATED SUBTREE

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

BEGIN ↱RELINK↓

    IF FATHER↑.RPT EQ OLDROOT
       THEN FATHER↑.RPT := NEWROOT
      ELSE FATHER↑.LPT := NEWROOT
END; ↱RELINK↓
↱------------------------------------------------------------------------

PROCEDURE SROTATE(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 ↱SROTATE↓

    ↱I2 := I2 + 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; ↱SROTATE↓
```

```
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---POINTERS 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↓

    RELINK(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-----POINTERS TO REVELANT NODES USED IN THE ROTATION
    WORD----KEY TO BE INSERTED

VAR
        (*S-- POINTER TO THE NEW NODE*)

    S:POINT;
BEGIN (*ROTATE*)

    (*CREATE THE NEW NODE*)

    NEXTNODE(Q,WORD,S);
    IF ACQUIRE(S) THEN GOTO EXIT 999;
    MAKENODE(WORD,S);

    (*DETERMINE THE APPROPRIATE ROTATION*)

    IF P^.LPT EQ NIL
        THEN
            CASE WAY OF
                RIGHT:SROTATE(FATHER,P,Q,S,RIGHT);
                LEFT :DROTATE(FATHER,P,Q,S,RIGHT);
            END
        ELSE
            CASE WAY OF
                RIGHT:DROTATE(FATHER,P,Q,S,LEFT);
                LEFT :SROTATE(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:POINT;WORD:ALFA;VAR NEWWORD:ALFA):BOOLEAN;

PURPOSE: BOOLEAN FUNCTION TO FIND THE POSTORDER PREDECESSOR OR SUCCESSO
         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 SUCESSOR

VAR
        *SAVE,QQ--AUXILIARY POINTER VARIABLES*

    QQ,SAVE:POINT;

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

                    *THE KEY TO BE INSERTED PRECEDES THE KEY OF THE
                     PREDECESSOR,DECREMENT THE K-SIZE VALUES OF THE
                     NODES IN THE SEARCH PATH OF THE SUBTREE
                     INVOLVED AND RETURN THE KEY OF THE PREDECESSOR

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

           *FIND THE SUCCESSOR*
```

```
          BEGIN
               WHILE Q↑.LPT NE NIL DO
                    BEGIN
                         SAVE := Q;
                         Q := Q↑.LPT
                    END;
               NEWWORD := Q↑.INFO;
               IF WORD LT NEWWORD
                    THEN
                         REPLACE := 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
          END
END; (*REPLACE*)
(*-------------------------------------------------------------------------

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-TREE

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<K
    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
                       THE NODE UNDER CURRENT EXAMINATION
          SUBSIZE-------THE SIZE THE SUBTREES OF THE NODE UNDER CURRENT
                        EXAMINATION (U SAY) MUST BE IF U IS TO BE
                        L-COMPLETE,L<K*)

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

  (*FIND THE NEXT NODE IN THE SEARCH PATH*)

  NEXTNODE(CURR,WORD,SON);
```

```
IF SON NE NIL
  THEN
    IF SON↑.SIZE EQ SUB
      THEN

        ↱THE TREE IS L-COMPLETE L<K:CHECK IF THE TREE IS 1-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(B) OF THE BIA↓

                  BEGIN
                    CHECKSIZE(SON↑.SIZE,SUBSIZE);
                    RECURSE(CURR,SON,WORD,SUBSIZE);
                    SAVE := CURR↑.INFO;
                    CURR↑.INFO := NEWWORD;
                    RECURSE(FATH,CURR,SAVE,SUB);
                  END
                ELSE

                  ↱STEP 5(A) OF THE BIA↓

                  BEGIN
                    SAVE := CURR↑.INFO;
                    CURR↑.INFO := WORD;
                    RECURSE(FATH,CURR,SAVE,SUB);
                  END;
            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 := CURR↑.SIZE + 1;
         END
  ELSE

    ↱THE SUBTREE IS L-COMPLETE FOR AN L<K:JUST ADD THE NEW NODE↓

    NEWNODE(CURR,WORD);

END; ↱RECURSE↓
```

```
FUNCTION SEARCHITEM(Q:POINT;WORD:ALFA):BOOLEAN;
↱
PURPOSE: 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:POINT;
    SUBSIZE:INTEGER;

BEGIN ↱BELLINSERT↓

    ↱INITIALIZE POINTERS↓

    FATHER := HEAD;
    CURRENT := HEAD↑.RPT;
    WAY := RIGHT;
    IF CURRENT EQ NIL
        THEN

            ↱ENTER THE FIRST NODE IN THE TREE↓

            BEGIN
                NEWNODE(HEAD,WORD);

                NODECOUNT := 1;
                ↱I1 := 1;↓
            END
        ELSE
          WHILE CURRENT↑.INFO NE WORD DO
            BEGIN
            ↱I1 := I1 + 1;↓

              ↱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
                    IF 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;
10:END; ↱BELLINSERT↓
```

```
PROCEDURE BELLDELETE(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 READJUSTMENT
           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,J:INTEGER;
    TEMP:POINT;

BEGIN (*FIXSTACK*)

    IF FLAG
       THEN

          (*THE STACK HAS EXCEEDED ITS LENGTH;READJUST 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];
                      J := K;
                      REPEAT
                          STACK[J + 1] := STACK[J];
                          J := J - 1;
                      UNTIL J EQ 0;
                      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:

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

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; (*PREDORSUCC*)

(* + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

PROCEDURE ONESON;
(*
PURPOSE: TO DELETE THE NODE,RELINK ITS ONE SUBTREE AND DECREMENT THE
         K-SIZE VALUES OF NODES IN THE DELETION STACK

VAR
        (*P--POINTER TO THE NODE TO BE DELETED
         Q-POINTER TO THE ONE SON OF THE NODE TO BE DELETED*)

    P,Q:POINT;
    JJ:INTEGER;

BEGIN (*ONESON*)

    (*D3 := D3 + 1;*)
    P := STACK[SPT];
    IF P↑.LPT EQ NIL
        THEN Q := P↑.RPT
        ELSE Q := 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 SUCESSOR IS BEST CHOSEN↲

    IF P↑.LPT↑.SIZE GT P↑.RPT↑.SIZE
        THEN
            BEGIN
                Q := P↑.LPT;
                QQ := Q↑.RPT
            END
        ELSE
            BEGIN
                Q := P↑.RPT;
                QQ := Q↑.LPT
            END;
    CHECKSTACK;
    STACK[SPT] := Q;
    PREDORSUCC(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 REPLACENODE(QQ,Q,P:POINT);
↱
PURPOSE: TO FIND THE PREDECESSOR OR SUCCESSOR OF A GIVEN INPUT NODE
         (U SAY),SWAP THE KEY OF THE PREDECESSOR (SUCCESSOR) WITH THE
         KEY OF U AND INSERT THE KEY OF U INTO THE TREE USING THE
         PROCEDURE RECURSE OF THE INSERTION ALGORITHM.(STEP 2(C)(2)
         OF THE BDA)

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
                   L<K
          WORD-----AN AUXILIARY ALFA VARIABLE↓

    WORD:ALFA;
    SUBSIZE:INTEGER:

BEGIN ↱REPLACENODE↓

    ↱FIND THE PREDECESSOR OR SUCCESSOR↓

    PREDORSUCC(QQ,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);
    ↱Q4 := Q4 + 1;↓

END; ↱REPLACENODE↓
```

```
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 EQ 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
                        STACK[3] := STACK[2]↑.RPT;
                        REPLACENODE(STACK[3]↑.LPT,STACK[3],STACK[2]);
                        OTHERSIDE := FALSE;
                    END;
        LEFT:
            IF STACK[2]↑.LPT↑.SIZE GT RSIZE
                THEN

                    "THE SUBTREE IS (K-1)-REPLETE"

                    BEGIN
                        STACK[3] := STACK[2]↑.LPT;
                        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--INTERGER VARIABLES REPRESENTING THE K-SIZES OF
                     L-COMPLETE TREES,1≤L<K
          Q,QQ---------AUXILIARY POINTERS↵

    RSIZE,SSIZE,J:INTEGER;
    QQ,Q:POINT;

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

    ↱INITIALIZE TO THE SMALLEST L-SIZE VALUES↵

    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)-REPLETE↵

                    BEGIN
                        IF OTHERSIDE(RSIZE) THEN NOSONS := FALSE;
                        GOTO 2L
                    END
                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↑.RPT
                    END;
              DELETEREDUCE(J);
              SPT := J + 1;
              STACK[SPT] := 0;
              REPLACENODE(QQ,Q,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;
END; ↱DECIDE↓
```

```
BEGIN ↱BELLDELETE↓

    ↱INITIALIZE THE DELETION STACK↓

    SPT := 1;
    STACKLENGTH := K + 1;
    FLAG := FALSE;
    STACK[1] := HEAD;
    P := HEAD↑.RPT;

    ↱SEARCH FOR THE KEY↓

    WHILE P NE NIL DO
         BEGIN
              ↱D1 := D1 + 1;↓
              CHECKSTACK;
              STACK[SPT] := P;
              IF P↑.INFO NE WORD
                   THEN
                        IF WORD GT P↑.INFO
                             THEN P := P↑.RPT
                             ELSE P := P↑.LPT
                   ELSE

                        ↱THE KEY IS PRESENT,DETERMINE HOW TO DELETE THE
                         NODE↓

                        BEGIN
                             DECIDE;
                             NODECOUNT := NODECOUNT - 1;
                             GOTO 10
                        END
         END;
10:END; ↱BELLDELETE↓
↱======================================================================

BEGIN ↱BELL-TREE DRIVER↓
    ↱INITIALIZE↓

    CREATEHEADER;

    ↱INSERT KEYS↓

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

    ↱DELETE KEYS↓

    WHILE READWORD(INWORD) DO

    BEGIN
         BELLDELETE(INWORD);
    PRINT(HEAD↑.RPT);
    END;
999:END. ↱BELL-TREE DRIVER↓
```

```
(* * * * * T H E   O P T I M A L   T R E E   O F   K N U T H * * * * * * * * * * * *)

   (*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[J
            BETAFREQ[J]
  WPL-------THE ARRAY CONTAINING THE WEIGHTED PATH LENGTH OF THE SUBTREE

LABEL    999;
CONST    MAX=10;
         STOPCHAR = '*';
         NN = 36;
TYPE     POINT = ^TREE;
         DIRECTION = (RIGHT,LEFT);
         NODE = RECORD
                              INFO:ALFA;
                              LPT:POINT;
                              RPT:POINT
              END;
VAR      TREE:CLASS 200 OF NODE;
         KROOT,FREE:POINT;
         WAY:DIRECTION;
         INWORD:ALFA;
         CHAY:ARRAY[1..MAX] OF CHAR;
         NODECOUNT:INTEGER;
         WPL,WEIGHT: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 (*INIT*)

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

    FREE := NIL;
    I := 1;
    WHILE READWORD(INWORD) DO
        BEGIN
            WD[I] := INWORD;
            I := I + 1
        END;
    IF I NE NN + 1 THEN WRITE('0 INCORRECT ',I,' KEYS READ',EOL);
    FOR I := 0 TO NN  DO
        BEGIN
            READ(J);
            BETAFREQ[I] := J;
        END;
    FOR I := 1 TO NN  DO
        BEGIN
            READ(J);
            ALFAFREQ[I] := J;
```

```
(*-----------------------------------------------------------------------*)

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-----INDICIES TO THE ROOT ARRAY;THE ROOT OF THE SUBTREE WHOSE
            KEYS ARE WD[L],I<L≤N, 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↓

            BEGIN
                IF ACQUIRE(P) THEN GOTO EXIT 999;
                MAKENODE(WD[R[I,J] + INX1],P);
                IF FATHER NE NIL
                    THEN
                        CASE WAY OF
                            RIGHT:FATHER↑.RPT := P;
                            LEFT :FATHER↑.LPT := P;
                        END
                    ELSE KROOT := P;

                ↱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 ↱OPTREEKNUTH↲

    ↱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] := BETAFREQ[I + INX1];
            FOR J := I + 1 TO N DO
                WEIGHT[I,J] := WEIGHT[I,J-1]+ALFAFREQ[J+INX1] +
                                    BETAFREQ[J+INX1]
        END;
    FOR J:= 1 TO N DO
        BEGIN
            WPL[J-1,J] := WEIGHT[J-1,J];
            R[J-1,J] := J
        END;

    ↱DETERMINE THE D NODE OPTIMUM TREES↲

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

                ↱FIND THE MIMIMUM 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; ↱OPTREEKNUTH↲
↱==================================================================

BEGIN ↱OPTREEKNUTH DRIVER↲

    ↱INITIALIZE↲

    INIT;

    ↱CREATE THE TREE↲

    OPTREEKNUTH(1,NN);

999:END. ↱OPTREEKNUTH DRIVER↲
```

```
** * * * * * * * * * * *B R U N O  C O F F M A N  T R E E* * * * * * * * * * * *

*GLOBAL CONSTANTS,TYPES,VARIABLES

  MAXDEPTH---THE MAXIMUM DIMENSION OF THE ARRAYS STACK AND WGHSAVE
  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:
             IT GIVES THE WEIGHT OF THE KEY WHICH DEFINES THE NODE
  WGHT-------AN ADDITIONAL FIELD IN ANY RECORD REPRESENTING A NODE:
             IT GIVES THE WEIGHT OF THE SUBTREE WHOSE ROOT IS THE NODE
  DEPTH------THE MAXIMUM LEVEL OF ANY NODE IN THE STARTING TREE
  ISEED------THE RANDOM NUMBER GENERATOR SEED
  KEYFREQ----THIS ARRAY CONTAINS THE WEIGHTS OF THE KEYS SUCH THAT
             KEYFREQ[I] IS THE WEIGHT OF VECTOR[I]
  VECTOR-----THIS ARRAY CONTAINS THE KEYS SUCH THAT THE WEIGHT OF
             VECTOR[I]>THE WEIGHT OF VECTOR[I+1]
  WATE-------THE WEIGHT OF THE TREE
  WPLOLD-----THE WEIGHTED PATH LENGTH OF THE STARTING TREE
  WPLNEW-----THE WEIGHTED PATH LENGTH OF THE NEARLY OPTIMAL TREE

LABEL 999:
CONST
        MAX = 10;
        STOPCHAR = E:E;
        N = 200;
        MAXDEPTH = 20;
TYPE
        POINT = *TREE;
        DIRECTION = (RIGHT,LEFT);
        KEYWORD = ARRAY[1..N] OF ALFA;
        NODE = RECORD
              INFO:ALFA;
              FREQ:INTEGER;
              WGHT:INTEGER;
              LPT:POINT;
              RPT:POINT;
          END;
VAR
        TREE:CLASS 201 OF NODE;
        FREE,HEAD:POINT;
        WAY:DIRECTION;
        INWORD:ALFA;
        CHAY:ARRAY[1..MAX] OF CHAR;
        NODECOUNT,WATE,DEPTH,WPLOLD,WPLNEW,BLOCKSIZE:INTEGER;
        ISEED:INTEGER;
        VECTOR:KEYWORD;
        KEYFREQ:ARRAY[1..N] OF INTEGER;
** * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *

PROCEDURE CREATEHEADER;
*
PURPOSE: TO PERFORM THE NECESSARY INITIALIZATIONS

BEGIN *CREATEHEADER*
    *INITIALIZE FREE LIST POINTER,NODE COUNTER,RANDOM NUMBER GENERATOR
     SEED AND THE BLOCKSIZE USED IN CREATING THE INITIAL TREE*

    FREE := NIL;
    NODECOUNT := 0;
    ISEED := 9876543;
    BLOCKSIZE := 50;

    *CREATE HEADER NODE*

    IF ACQUIRE(HEAD) THEN GOTO EXIT 999;
    HEAD*.RPT := NIL;
```

```
FUNCTION WEIGHTEDPATHLENGTH(PROOT: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:
    PROOT--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(P:POINT;I: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^.LPT,I + 1);
                PL := PL + P^.FREQ*I;
                TRAVERSE(P^.RPT,I + 1);
            END
END; (*TRAVERSE*)

(* + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + *)

BEGIN (*WEIGHTEDPATHLENGTH*)

    DEPTH := 0;
    PL := 0;

    TRAVERSE(PROOT,1);
    WEIGHTEDPATHLENGTH := PL;

END; (*WEIGHTEDPATHLENGTH*)
```

```
PROCEDURE INSERT(KEY:ALFA;KEYFQ: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
     KEYFQ--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
     PR:POINT;

BEGIN (*INSERT*)
     IF PP EQ HEAD^.RPT
        THEN

          (*THE ORDERING RELATION IS THE ALPHABETIC RELATIONSHIP
           BETWEEN THE KEYS. THIS BLOCK IS USED TO CONSTRUCT THE
           STARTING TREE.*)

          BEGIN
              REPEAT
                  BEGIN
                      PR := PP;

                       (*ADJUST THE WEIGHT OF THE SUBTREE DEFINED BY
                        THE NODE POINTED TO BY PP.*)

                       PP^.WGHT := PP^.WGHT + KEYFQ;
                       IF PP^.INFO LT KEY
                           THEN PP := PP^.RPT
                           ELSE PP := PP^.LPT;
                  END;
              UNTIL PP EQ 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^.LPT := NIL;
                           PP^.RPT := NIL;
                           PP^.INFO := KEY;
                           PP^.FREQ := KEYFQ;
```

```
                                        PP↑.WGHT := KEYFQ;
                                        IF PR↑.INFO LT KEY
                                             THEN PR↑.RPT := PP
                                             ELSE PR↑.LPT := PP
                                   END;
                    END
          ELSE
               ↑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 WEIGHTS↓
               BEGIN
                    REPEAT
                         BEGIN
                              PR := PP;
                              IF PP↑.FREQ GE KEYFQ
                                   THEN PP := PP↑.LPT
                                   ELSE PP := PP↑.RPT
                         END;
                    UNTIL PP EQ 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↑.LPT := NIL;
                                   PP↑.RPT := NIL;
                                   PP↑.INFO := KEY;
                                   PP↑.FREQ := KEYFQ;
                                   IF PR↑.FREQ GE KEYFQ
                                        THEN PR↑.LPT := PP
                                        ELSE PR↑.RPT := PP
                              END
                    END
END; ↑INSERT↓
```

```
PROCEDURE READYKEYS(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
         ↱FQ----THE WEIGHT OF A KEY
          ROOT--THE POINTER TO THE ROOT OF THE TREE USED IN THE TREESORT
                                                                      ↓
    FQ:INTEGER;
    IVEC:INTEGER;
    ROOT:POINT;

↱ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + ↓

PROCEDURE ORDERKEYS(P:POINT);
↱
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↑.RPT);
                VECTOR[IVEC] := P↑.INFO;
                KEYFREQ[IVEC] := P↑.FREQ;
                IVEC := IVEC + 1;
                RELEASE(P);
                ORDERKEYS(P↑.LPT)
            END
END; ↱ORDERKEYS↓
```

```
BEGIN ↱READYKEYS↓

    ↱CREATE A TREE,THE ORDERING RELATION BEING DEFINED AS THE USUAL
     LESS THAN RELATION ON THE WEIGHTS OF THE KEYS↓

    IF READWORD(INWORD)
        THEN

            ↱CREATE THE ROOT NODE↓

            IF ACQUIRE(ROOT)
                THEN GOTO EXIT 999
                ELSE
                    BEGIN
                        NODECOUNT := 1;
                        ROOT↑.LPT := NIL;
                        ROOT↑.RPT := NIL;
                        ROOT↑.INFO := INWORD;
                        READ(FQ);
                        WATE := FQ;
                        ROOT↑.FREQ := FQ;
                    END;
    WHILE READWORD(INWORD) DO
        BEGIN
            READ(FQ);
            WATE := WATE + FQ;
            INSERT(INWORD,FQ,ROOT);
        END;

    ↱SORT THE KEYS ON DESCENDING ORDER OF THEIR WEIGHTS↓

    IVEC := 1;
    ORDERKEYS(ROOT);
END; ↱READYKEYS↓
```

```
PROCEDURE STARTREE(BLSZ: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 WEIGHTS BY THE ARRAY ELEMENTS KEYFREQ[I].

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

    BLSZ--THE BLOCKSIZE TO BE USED

VAR
        *BOOL-----BOOLEAN ARRAY USED IN SELECTING RANDOM ELEMENTS
                  FROM EACH BLOCK OF KEYS
         INX1,INX2,INX1M1,INX2P1--VARIABLES TO DETERMINE THE CORRECT
                                  ELEMENTS OF THE ARRAY,VECTOR,WHEN
                                  SELECTING A RANDOM ELEMENT FROM
P                                 A BLOCK
         NUMBLKS--THE NUMBER OF BLOCKS*

    BOOL:ARRAY[1..N] OF BOOLEAN;
    NUMBLKS,MANY,IJ,I,J,INX1,INX2,INX1M1,INX2P1:INTEGER;
    IJJ:INTEGER;
    P:POINT;

BEGIN *STARTREE*

    *DETERMINE THE NUMBER OF BLOCKS REQUIRED*

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

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

    IJ := TRUNC(RANDOM(1.,BLSZ/1.+1.0,ISEED));
    IF ACQUIRE(P) THEN GOTO EXIT 999;
    HEAD*.RPT := P;
    NODECOUNT := 1;
    P*.LPT := NIL;
    P*.RPT := NIL;
    P*.FREQ := KEYFREQ[IJ];
    P*.INFO := VECTOR[IJ];
    P*.WGHT := KEYFREQ[IJ];

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

```
FOR J := 1 TO BLSZ 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 NUMBLKS DO
     BEGIN
          INX2 := I*BLSZ;
          INX1 := INX2 - BLSZ + 1;
          INX1M1 := INX1 - 1;
          INX2P1 := INX2 + 1;
          IF INX2 GT N
               THEN INX2 := N;
          FOR IJJ := INX1 TO INX2 DO
               BEGIN
                    IJ := TRUNC(RANDOM(INX1/1.0,INX2P1/1.0,ISEED));
                    FOR J := IJ TO INX2 DO
                         IF BOOL[J - INX1M1] EQ FALSE
                              THEN
                                   BEGIN
                                        INSERT(VECTOR[J],KEYFREQ[J],
                                                  HEAD*.RPT);
                                        GOTO 10
                                   END;
                    FOR J := INX1 TO IJ - 1 DO
                         IF BOOL[J - INX1M1] EQ FALSE
                              THEN
                                   BEGIN
                                        INSERT(VECTOR[J],KEYFREQ[J],
                                                  HEAD*.RPT);
                                        GOTO 10
                                   END;
                    10:BOOL[J - INX1M1] := TRUE;
               END;

          *REINITIALIZE THE BOOLEAN ARRAY FOR THE NEXT BLOCK+

          FOR J := 1 TO BLSZ DO
               BOOL[J] := FALSE;
     END;
END: *STARTREE+
```

```
PROCEDURE OPTREE3C(K,WPLOLD:INTEGER;VAR WPLNEW:INTEGER;VECTOR:KEYWORD);
*
PURPOSE: TO CREATE A NEARLY OPTIMAL TREE USING THE BRUNO 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 10;

VAR
         *IWS-----------THE INDEX TO THE ARRAY CONTAINING SUBTREE WEIGHTS
          Q-------------POINTER TO THE NODE BEING PROMOTED
          QFREQ---------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,IWS,QFREQ,STACKLENGTH:INTEGER;
    Q:POINT;
    STACK:ARRAY[0..MAXDEPTH] OF POINT;
    WGHSAVE:ARRAY[1..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;
    P: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;
    TEMP:POINT;

BEGIN ↑FIXSTACK↓

    IF SPT NE STACKLENGTH
        THEN

            ↑READJUST 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 JJ DO
                            STACK[J + 1] := STACK[J];
                        STACK[JJ] := TEMP
                    END
            END
END; ↑FIXSTACK↓

↑ + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + ++

BEGIN ↑DETERMINE↓

    ↑INITIALIZE↓

    DETERMINE := FALSE;
    FLAG := FALSE;
    SPT := 0;
    STACK[0] := HEAD;
    P := HEAD↑.RPT;

    ↑CREATE THE STACK↓

    WHILE P↑.INFO NE KEY DO
        BEGIN
            SPT := SPT + 1;
            STACK[SPT] := P;
            IF KEY GT P↑.INFO
                THEN P := P↑.RPT
                ELSE P := P↑.LPT;
            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 := 0;

    ↱PERFORM SKLEN ROTATIONS↓

    FOR J := SKLEN DOWNTO 1 DO

        ↱PERFORM THE ROTATION AND ADJUST THE WEIGHTS OF THE SUPTREES
         AFFECTED↓

        BEGIN
            IJ := IJ + 1;
            IF STACK[J - 1]↑.RPT EQ STACK[J]
                THEN STACK[J - 1]↑.RPT := Q
                ELSE STACK[J - 1]↑.LPT := Q;
            IF STACK[J]↑.RPT EQ Q
                THEN
                    BEGIN
                        STACK[J]↑.RPT := Q↑.LPT;
                        Q↑.LPT := STACK[J];
                        Q↑.LPT↑.WGHT := WGHSAVE[IJ]
                    END
                ELSE
                    BEGIN
                        STACK[J]↑.LPT := Q↑.RPT;
                        Q↑.RPT := STACK[J];
                        Q↑.RPT↑.WGHT := WGHSAVE[IJ]
                    END;
        END;
    Q↑.WGHT := Q↑.FREQ;
    IF Q↑.LPT NE NIL
        THEN Q↑.WGHT := Q↑.WGHT + Q↑.LPT↑.WGHT;
    IF Q↑.RPT NE NIL
        THEN Q↑.WGHT := Q↑.WGHT + Q↑.RPT↑.WGHT;

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 DETERMINE 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]:
                IWS := IWS + 1;

                ↱DETERMINE THE CORRECT DIRECTION OF THE ROTATION↓

                IF SK↑.RPT 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↑.FREQ:
                            IF SK↑.LPT NE NIL
                                THEN WL := WL + SK↑.LPT↑.WGHT;
                            WGHSAVE[IWS] := WL;
                            PLNEW := PLNEW - WI + WL:
                            WI := WL + WR + QFREQ:
                            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↑.FREQ:
                            IF SK↑.RPT NE NIL
                                THEN WR := WR + SK↑.RPT↑.WGHT;
                            WGHSAVE[IWS] := WR:
                            PLNEW := PLNEW - WI + WR:
                            WI := WL + WR + QFREQ:
                            COMPARE(PLNEW,KK - 1)
                        END
            END
        ELSE

            ↱DETERMINE IF THE WEIGHTED PATH LENGTH HAS BEEN REDUCED: IF
             SO PERFORM THE ACTUAL ROTATIONS↓

            BEGIN
                IF WPLOLD GT PLNEW
                    THEN
                        BEGIN
                            WPLOLD := PLNEW:
                            ROTATIONS;
                        END:
                EXIT 1
```

```
BEGIN ↱OPTREEBC↓

     ↱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 N DO
                    IF DETERMINE(VECTOR[J])
                         THEN

                              ↱INITIALIZE FOR THE PROMOTION OF THE NODE
                               POINTED TO BY Q↓

                              BEGIN
                                   Q := STACK[STACKLENGTH];
                                   QFREQ := Q↑.FREQ;
                                   IF Q↑.LPT NE NIL
                                        THEN WL := Q↑.LPT↑.WGHT
                                        ELSE WL :=  ;
                                   IF Q↑.RPT NE NIL
                                        THEN WR := Q↑.RPT↑.WGHT
                                        ELSE WR :=  ;
                                   WI := Q↑.WGHT;
                                   IWS := 0;

                                   ↱DETERMINE IF THE WEIGHTED PATH LENGTH WILL
                                    BE REDUCED WITH A TRANSFORMATION↓

                                   COMPARE(WPLOLD,I);
                            10:END
          END;
     WPLNEW := WPLOLD;
END; ↱OPTREEBC↓
↱=======================================================================↓
BEGIN ↱BRUNO 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);
     WPLOLD := WEIGHTEDPATHLENGTH(HEAD↑.RPT,DEPTH);

     ↱APPLY THE BRUNO COFFMAN ALGORITHM↓

     OPTREEBC(DEPTH,WPLOLD,WPLNEW,VECTOR);

999:END. ↱BRUNO COFFMAN TREE DRIVER↓
```

```
                    * W A L K E R   G O T L I E B   T R E E *
*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *

   *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 WEIGHTS 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
NO--------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 = ':';
          NUM = 200;
          NN = 31;
TYPE      POINT = ^TREE;
          DIRECTION = (RIGHT,LEFT);
          NODE = RECORD
                            INFO:ALFA;
                            LPT:POINT;
                            RPT:POINT
                END;
VAR
          TREE:CLASS 200 OF NODE;
          WGROOT,KROOT,FREE:POINT;
          NODECOUNT,NO,F:INTEGER;
          WAY:DIRECTION;
          INWORD:ALFA;
          CHAY:ARRAY[1..MAX] OF CHAR;
          WPL,WEIGHT:ARRAY[0..NN,0..NN] OF INTEGER;
          R:ARRAY[0..NN,1..NN] OF INTEGER;
          ALFAFREQ:ARRAY[1..NUM] OF INTEGER;
          BETAFREQ:ARRAY[0..NUM] 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 := NIL;
    NODECOUNT := 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(≡0INCORRECT ≡,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.,BETAFREQ[IN1],
         ALFAFREQ[IN1+1],...,BETAFREQ[IN2],ALFAFREQ[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 := BETAFREQ[IN1];
    FOR I := IN1 + 1 TO IN2 DO
        W := W + ALFAFREQ[I] + BETAFREQ[I];
    WGH := W

END; (*WGH*)
```

```
PROCEDURE OPTREEWG(NO,F:INTEGER);
(*
PURPOSE: TO STRUCTURE THE NEARLY OPTIMAL TREE OF WALKER AND GOTLIEB

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
                          CENTROID 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,MAXFREQ,LOGQUANT,L,R:INTEGER;
    SETT:ARRAY[1..NUM] OF INTEGER;

(* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * *)

PROCEDURE STEP8(VAR RROOT:INTEGER;INX1,INX2:INTEGER);
(*
PURPOSE: TO CHOOSE THE KEY FROM THE NEIGHBORHOOD ABOUT THE CENTROID
         WHICH WILL BE THE KEY OF THE ROOT OF THE SUBTREE

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

PARAMETERS OUTPUT:
    RROOT--THE INDEX OF THE KEY WHICH IS THE KEY OF THE ROOT OF THE
           SUBTREE

VAR
         (*TRW,NEWTRW--VARIABLES USED IN DETERMINING WHICH POSSIBLE
                       CANDIDATES FOR THE KEY OF THE ROOT NODE IS BEST
                       IN TERMS OF EQUALIZING THE WEIGHT OF ITS LEFT AND

                       RIGHT SUBTREES*)

    J,TRW,NEWTRW,INX: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 EQUALIZ
     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 CENX 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 :=L DOWNTO INX1 + 1 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
           LTREE-----THE WEIGHT OF THE RIGHT SUBTREE OF THE NODE IN
                     QUESTION*

    DIFF,DIF: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;
    DIFF := ABS(LTREE - RTREE);
    IF DIFF 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*

    CENX := -1;
    FOR J := 1 TO LENGTH DO
```

```
        IF SETT[J] EQ CENTROID
            THEN CENX := J;
    IF CENX LT 0
        THEN

            *THE NEIGHBORHOOD IS EMPTY;PLACE THE CENTROID IN THE
             NEIGHBORHOOD*

            BEGIN
                SETT[1] := CENTROID;
                LENGTH := 1;
                CENX := 1
            END
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 CHOOSEN AS ROOT OF TH ESUBTREE
                  BEING CONSTRUCTED*

    P:POINT;
    NUMNODE,SAVE,MAX,RROOT:INTEGER;

BEGIN *STEP1*

    *DETERMINE THE NUMBER OF KEYS IN THE SUBTREE AND IF IT CAN BE
     STRUCTURED BY KNUTH S ALGORITHM*

    NUMNODE := INX2 - INX1 + 1;
    IF NUMNODE LE NO
        THEN
            BEGIN
                IF NUMNODE NE 0
                    THEN

                        *STRUCTURE USING KNUTH S ALGORITHM*

                        BEGIN
                            SAVE := NODECOUNT;
                            OPTREEKNUTH(INX1,INX2);
                            NODECOUNT := SAVE + NODECOUNT;
                            IF FATHER^.INFO GT KROOT^.INFO
                                THEN FATHER^.LPT := KROOT
                                ELSE FATHER^.RPT := KROOT
                        END
            END
        ELSE
```

```
          *FIND THE ROOT OF THE SUBTREE WITH THE WALKER  GOTLIEB
           ALGORITHM*
          BEGIN
              LOGQUANT := 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;
              MAKENODE(WD[RROOT],P);
              IF FATHER NE NIL
                  THEN
                      IF WD[RROOT] LT FATHER^.INFO
                          THEN FATHER^.LPT := P
                          ELSE FATHER^.RPT := P
                  ELSE WGROOT := P;
              FATHER := P;

              *STRUCTURE THE LEFT AND RIGHT SUBTREES*

              STEP1(INX1,RROOT - 1,FATHER);
              STEP1(RROOT + 1,INX2,FATHER)
          END
END; *STEP 1*

* + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + + +

BEGIN *OPTREEWG*

   IF NUM LE NO
      THEN WRITE(='USE KNUTHS ALGO FOR THIS NUMBER OF ITEMS =,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 SHUFFLE
     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,D1,I23,D23--COUNTERS USED IN STATISTICS COLLECTION
     ROOTNUMTREES--THE SORT OF THE NUMBER OF TREES CREATED
     SEED----------THE SEED FOR THE RANDOM NUMBER GENERATOR
     RANDNUM-------THE RANDNUM NUMBER ARRAY

          INX1 = 4;
          NUMTREES = 500;
          MAXSIZE = 5001;
          SHUFFLEFACTOR = 3000;
          STATSIZE = 500;
          TREEINTERVAL = 10;
          ALFA = 1..10000;
          TYPEOFARRAY = ARRAY[1..MAXSIZE] OF INTEGER;
          I,II:INTEGER;
          I23,D23:INTEGER;
          RANDNUM:TYPEOFARRAY;
          ROOTNUMTREES:REAL;
          SEED,I1,I2,I3,D1: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 COMPARSIONS NEEDED TO FIND THE NODE WITH
           GIVEN INPUT KEY
                                                                         *
VAR
     P:POINT;
     SEARCHX:INTEGER;

BEGIN *SEARCH*
     P := HEAD↑.RPT;
     SEARCHX := 0;
     WHILE P NE NIL DO
          BEGIN
               SEARCHX := SEARCHX + 1;
               IF P↑.INFO NE WORD
                    THEN
                         IF WORD LT P↑.INFO
                              THEN P := P↑.LPT
                              ELSE P := P↑.RPT
                    ELSE
                         BEGIN
                              SEARCH := SEARCHX;
                              GOTO 10;
                         END;
          END;
     SEARCH := SEARCHX + 1;
10:END; *SEARCH*

*-----------------------------------------------------------------------*

PROCEDURE BBSTATISTICS;
*
PURPOSE: TO COLLECT STATISTICS ON THE BBTREE
                                                                         *
VAR
     RANDNUM:TYPEOFARRAY;
     STATSUP,STATSQP:ARRAY[1..INX1,1..STATSIZE] 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

   STATSQP                          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 := TREEINTERVAL; 10
            L := 1;

            *COLLECT INSERTION STATS*

            FOR J := 1 TO MAXSIZE DO
                BEGIN
                    IJ := IJ - 1;
                    IF IJ EQ 0
                      THEN
                        BEGIN
                          IJ := TREEINTERVAL;
                          I1 := SEARCH(RANDNUM[J]);
                          I2 := 0;
                          I3 := 0;

                          BBINSERT(RANDNUM[J]);

                          STATSUP[1,L] := STATSUP[1,L] + I1;
                          STATSUP[2,L] := STATSUP[2,L] + I2;
                          STATSUP[3,L] := STATSUP[3,L] + I3;

                          STATSUP[4,L] := STATSUP[4,L] + I1*I1;

                          L := L + 1;
                        END

                      ELSE BBINSERT(RANDNUM[J]);

                END;

            IJ := 2;
            L := STATSIZE;
            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[J]);
                        I2 := 0;
                        I3 := 0;

                        BBDELETE(RANDNUM[J]);

                        STATSOP[1,L] := STATSOP[1,L] + D1;
                        STATSOP[2,L] := STATSOP[2,L] + I2;
                        STATSOP[3,L] := STATSOP[3,L] + I3;

                        STATSOP[4,L] := STATSOP[4,L] + D1*D1;

                        L := L - 1;.
                      END
                    ELSE BBDELETE(RANDNUM[J]);
              END;
      END;
END: +BUILDANDDESTROY+

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

BEGIN +BBSTATISTICS+

   +INITIALIZE +

   FOR I := 1 TO INX1 DO
       FOR J := 1 TO STATSIZE DO
           BEGIN
               STATSUP[I,J] := 0;
               STATSOP[I,J] := 0;
           END;
   SEED := 73559;

   +INITIALIZE RANDOM NUMBER ARRAY +

   FOR I := 1 TO MAXSIZE DO
       RANDNUM[I] := I;
   ROOTNUMTREES := SQRT(NUMTREES);

   +INITIAL ALL COUNTERS+

   I2 := 0;
   I3 := 0;
   D1 := 0;

   +BUILD AND DESTROY TREES+

   CREATEHEADER;
   BUILDANDDESTROY;

   WRITE(E1BB-INSERTION THE NUMBER OF TREES OBSERVED ISE,NUMTREES:5,
       EOL);
   WRITE(EALFA IS E,NUMER:3,E / E,DENOM:3,EOL);
   WRITE(E  TREE    AVERAGE CONFIDENC    AVERAGE CONFIDENC    AVERAGE E,
        ECONFIDENC    AVERAGE CONFIDENCE,EOL);
   WRITE(E  SIZE     SEARCH  INTERVAL     SINGLE  INTERVAL     DOUBLE E,
```

```
         ≡ INTERVAL   ROTATION   INTERVAL≡,EOL);
WRITE(≡  ~≡,EOL);

FOR I := 1 TO STATSIZE DO
    BEGIN
        II  := I*TREEINTERVAL;
        AV1  := MEAN(STATSUP[1,I]);
        SD1  := SD(STATSUP[1,I],STATSUP[4,I]);
        SD1  := 2.0*SD1/ROOTNUMTREES;
        AV2  := MEAN(STATSUP[2,I]);
        SD2  := 2.0*SQRT(AV2)/ROOTNUMTREES;
        AV3  := MEAN(STATSUP[3,I]);
        SD3  := 2.0*SQRT(AV3)/ROOTNUMTREES;
        I23  := STATSUP[2,I] + STATSUP[3,I];
        AV4  := MEAN(I23);
        SD4  := 2.0*SQRT(AV4)/ROOTNUMTREES;

        WRITE(≡J≡,II:5,≡ ≡);
        WRITE(AV1:9:4,≡  ≡);
        WRITE(SD1:9:4,≡  ≡);
        WRITE(AV2:9:4,≡  ≡);
        WRITE(SD2:9:4,≡  ≡);
        WRITE(AV3:9:4,≡  ≡);
        WRITE(SD3:9:4,≡  ≡);
        WRITE(AV4:9:4,≡  ≡);
        WRITE(SD4:9:4,EOL);
    END;

WRITE(≡1PR-DELETION THE NUMBER OF TREES OBSERVED IS ≡,NUMTREES:5,
      EOL);
WRITE(≡ALFA IS ≡,NUMER:3,≡ / ≡,DENOM:3,EOL);
WRITE(≡  TREE   AVERAGE CONFIDENC   AVERAGE CONFIDENC   AVERAGE ≡,
      ≡CONFIDENC   AVERAGE CONFIDENCE≡,EOL);
WRITE(≡  SIZE   SEARCH  INTERVAL   SINGLE  INTERVAL   DOUBLE ≡,
      ≡ INTERVAL   ROTATION  INTERVAL≡,EOL);
WRITE(≡  ~≡,EOL);

FOR I := 1 TO STATSIZE DO
    BEGIN
        II  := I*TREEINTERVAL;
        AV1  := MEAN(STATSDP[1,I]);
        SD1  := SD(STATSDP[1,I],STATSDP[4,I]);
        SD1  := 2.0*SD1/ROOTNUMTREES;
        AV2  := MEAN(STATSDP[2,I]);
        SD2  := 2.0*SQRT(AV2)/ROOTNUMTREES;
        AV3  := MEAN(STATSDP[3,I]);
        SD3  := 2.0*SQRT(AV3)/ROOTNUMTREES;
        D23  := STATSDP[2,I] + STATSDP[3,I];
        AV4  := MEAN(D23);
        SD4  := 2.0*SQRT(AV4)/ROOTNUMTREES;

        WRITE(≡J≡,II:5,≡ ≡);
        WRITE(AV1:9:4,≡  ≡);
        WRITE(SD1:9:4,≡  ≡);
        WRITE(AV2:9:4,≡  ≡);
        WRITE(SD2:9:4,≡  ≡);
        WRITE(AV3:9:4,≡  ≡);
        WRITE(SD3:9:4,≡  ≡);
        WRITE(AV4:9:4,≡  ≡);
        WRITE(SD4:9:4,EOL);

    END;
END; ←B?STATISTICS←
```

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:

block → blockbody <end>

block → blockbody statement <end>

blockbody → blockbody statement <;>

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.

```
ROOT OF FOLLOWING TREE  BLOCKBODY

<END>   →BLOCK   [PROD2]
  :
  :              --- <END>   →BLOCK
  :            :           :
  :            :           :
  :            :           --- <;>    →BLOCKBODY
  :            :
  --- STATEMENT
               :
               --- LABELDEF   →BLOCKBODY
                          :
                          --- <;>    →BLOCKBODY
```