

Comp.

1616

S.K.

FACULTY OF ENGINEERING - UNIVERSITY OF ALEXANDRIA

Operating on optimal..



GN:3607

BibID:10011669

005.43.A O.Comp.

# OPERATING ON OPTIMAL BINARY CODE TREES

*Thesis submitted to the*

DEPT. OF COMPUTER SCIENCE & AUTOMATIC CONTROL  
*in partial fulfillment of the requirements for  
a Ph.D. degree in Computer Science*



Submitted by:

Shymaa M. Arafat

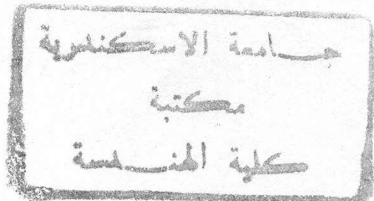
3607

Supervised by:

Prof. Dr. Ahmed A. Belal

Prof. Dr. Mohamed S. Selim

005.43  
ARA



May - 2001

*operating systems*

We certify that we have read this thesis and that, in our opinion, it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Exam Committee:

1- *Prof. Dr. Mohamed Zaki Abd el Megeed*

Faculty of Engineering  
El Azhar University

*M Zak*

2- *Prof. Dr. Ahmed A. Belal*

Department of Computer Science & Automatic  
Control  
Alexandria University

*A. Belal*

3- *Prof. Dr. Hesham H. Ali*

University of Nebraska at Omaha

For the Faculty Council

Prof. Dr. Hassan Farag

Vice Dean for Graduate Studies and Research  
Faculty of Engineering, Alexandria University

## Supervisors

### Professor Dr. Ahmed A. Belal

Professor at:

University of Alexandria,  
Faculty of Engineering,  
Dept. of Computer Science & Automatic Control.

### Professor Dr. Mohamed S. Selim

Professor at:

University of Alexandria,  
Faculty of Engineering,  
Dept. of Computer Science & Automatic Control.

## C.V.

Name : Shymaa Mohammed Arafat  
Date of Birth : 25/7/1972  
Country of Birth : Malizia  
Country of Citizenship : Egypt

### Home Address:

242 Al Fath St.,  
Janaklees,  
Alexandria,  
Egypt.

### Current Position:

Assistant lecturer at:

Ain Shams University,  
Faculty of Information & Computer Sciences  
Dept. of Computer Science.

### Academic Record:

- M.Sc. from the Dept. of Computer science & Automatic control,  
Faculty of Engineering, University of Alexandria. (1997)
- B.Sc. from the Dept. of Computer science & Automatic control,  
Faculty of Engineering, University of Alexandria. (1994)

## Abstract

Optimal alphabetic binary trees were designed to handle information of different access frequencies where a symmetric order is imposed on the resulting tree. Optimal alphabetic trees (OATs) can be build faster than optimal binary search trees, yet they can be used for binary search. OATs can also be used for data compression like Huffman codes; the loss in compression ratio is compensated for by the symmetry of the encoding and decoding process (Huffman codes need an additional hashing mechanism in the decoding).

As for the multi dimensional case, reasonably fast algorithms exist to build a nearly optimal tree. However, in the general case, the optimal tree can only be achieved through dynamic programming. For the importance and wide range of multi dimensional applications, we start the thesis by proposing a new concept "the expense of a cut" followed by an algorithm for finding the 2-d OAT, experimental results show it is reasonably faster than dynamic programming. Then we get back to the one dimensional case.

The merits of optimal alphabetic trees are mostly seen by researchers. The fast implementation to build OATs is rather complicated compared to methods for building other data structures. In addition, the fact that any editing in the weight sequence may result in the tree losing its optimality, limits the use of OATs in applications of dynamic nature.

This thesis aims to add more flexibility to the processing of OATs, opening the door for a wider use of it. We develop linear time (and space) algorithms to insert, delete, or change the weight of a node in an OAT *without losing its optimality*, to split an OAT into two OATs, and two merge two OATs into one optimal tree. In addition, we use the linear time merging algorithm to build the tree recursively in a divide and conquer manner; the resulting algorithm is much simpler, and experimental results show it is also faster, than the existing implementation.

Finally, we introduce similar algorithms to edit the weight sequence of a Huffman tree, since they can be viewed as OATs lacking the proper symmetric order. Experimental results show the efficiency of the proposed algorithms compared to known methods, except for the merging case.

Table of Contents

Abstract	ix
1 Introduction	1
1.1 Optimal Alphabetic Trees	1
1.2 Scope of the Thesis	1
1.3 Organization of the Thesis	2
2 Binary Trees	3
2.1 Basic Terminology	3
2.2 Binary Search Trees	4
2.3 Optimal Binary Search Trees	6
2.4 The Problem of Finding the Optimal BST	8
2.5 Binary Code Trees	10
2.6 Alphabetic Trees	11
2.7 Multi Dimensional Case	13
3 Optimal Alphabetic Trees Research Work	14
3.1 The Hu-Tucker Algorithm	14
3.1.1 Algorithm	14
3.1.2 Example	15
3.1.3 Implementation	16
3.2 The Garsia-Wachs Algorithm	17
3.3 Related Results	18
3.4 Applications where OATs are Used	20
4 Multi Dimensional Optimal Alphabetic Trees	22
4.1 Introduction	22
4.1.1 Two dimensional alphabetic trees	22
4.1.2 Two dimensional optimal alphabetic trees	24
4.2 Dynamic Programming	25
4.3 Related Results	25
4.4 Approximate Algorithm	27
4.5 The Expense of a Cut	29
4.6 Proposed Method	31
4.7 Performance Evaluation	35
4.7.1 Complexity analysis	35
4.7.2 Experimental results	36
4.8 Conclusion	37
5 Insertion in Optimal Alphabetic Trees	38
5.1 Preface	38
5.1.1 Problem definition	38
5.1.2 Algorithm outline	39
5.1.3 Terminology	39
5.2 Preliminary Results	41

5.3 Implementation Issues.....	47
5.4 Insertion at the Boundary.....	49
5.5 Constrained Insertion in the Middle.....	61
5.6 Unconstrained Insertion in the Middle.....	68
5.7 Performance Evaluation.....	75
5.8 Applications.....	77
5.8.1 Deletion.....	77
5.8.2 Changing the weight of a node.....	78
5.8.3 Splitting optimal alphabetic trees.....	79
5.8.4 Other Merits.....	79
5.9 Conclusion.....	80
 <b>6 Merging &amp; Constructing Optimal Alphabetic Trees</b>	<b>81</b>
6.1 Merging Optimal Alphabetic Trees.....	81
6.1.1 Problem definition.....	81
6.1.2 Algorithm outline.....	81
6.1.3 Nodes in the merging working sequence.....	82
6.1.4 Example.....	85
6.1.5 Performance analysis.....	87
6.2 Successive Merging.....	88
6.2.1 Problem statement.....	88
6.2.2 Solution.....	89
6.2.3 Example.....	90
6.3 Building the Optimal Alphabetic Tree.....	91
6.3.1 Problem definition.....	91
6.3.2 Algorithm.....	92
6.3.3 Example.....	93
6.4 Performance Analysis.....	94
6.4.1 Complexity analysis.....	94
6.4.2 Experimental results.....	94
6.5 Conclusion.....	95
 <b>7 Rebuilding Huffman Trees in Linear Time</b>	<b>96</b>
7.1 Preface.....	96
7.2 Building the Huffman Tree.....	97
7.3 Primitive Operations with Sorted Sequences-Method one.....	98
7.3.1 Insertion.....	98
7.3.2 Deletion.....	99
7.3.3 Changing the weight of a node.....	99
7.3.4 Splitting the tree into two subtrees.....	99
7.3.5 Merging.....	99
7.3.6 Other operations.....	99
7.4 Primitive Operations by Processing the <i>MP</i> list-Method two.....	99
7.4.1 Algorithm outline.....	100
7.4.2 Terminology.....	100
7.4.3 The method on Huffman trees compared to OATs.....	101
7.5 Insertion.....	102
7.6 Deletion.....	104
7.7 Merging Huffman Trees.....	108

7.8 Conclusion.....	112
<b>8 Conclusions &amp; Future Work</b>	<b>113</b>
8.1 Summary & Conclusions.....	113
8.2 future Work.....	113
<b>References</b>	<b>115</b>



# Chapter 1: Introduction

With the advent of technology and information exchange all around the world, the ability to store and quickly retrieve large amounts of information has become a persistent need. A large information system is essential to any corporation; digital libraries and dictionaries are common in any institution. Consequently, handling information has become a major computer application and a wide area for continuous research. Trees is a favorite data structure for information storing and retrieving since they are characterized by their fast access time compared to linear structures. Optimal alphabetic trees (OATs) are the subject of this thesis.

## 1.1 Optimal Alphabetic Trees (OATs)

When we are speaking about probabilistic retrieval systems, where different elements have different access frequencies and fast retrieval should be guaranteed to high frequency items, optimal alphabetic trees becomes a suitable structure for such systems. Also, they can be used in generating alphabetic codes for compression and security purposes; an essential demand for information exchange and an interesting area for research. A quite different set of applications for OATs are minimizing the inter connection delay in VLSI design and the address look up time in destination based routers.

As many information systems have multidimensional nature, or their access depends on multiple keys that are correlated, multidimensional alphabetic trees have their importance and applications.

## 1.2 Scope of the Thesis

In this thesis, we first discuss optimal alphabetic trees in the multi dimensional case and introduce a new concept, and algorithm, that limits the number of trees to be checked for optimality. Then we follow by introducing new results for the one dimensional case.

This starts with a novel linear time algorithm for inserting or deleting a node in an optimal alphabetic tree and keep it optimal after insertion (or deletion). Also, same idea is used to introduce linear time algorithms for merging and splitting optimal alphabetic trees. Then, we propose a new method to build the optimal

alphabetic tree in  $\theta(n \log n)$  for  $n$  nodes sequence by successive merging; a fast technique that is much simpler than the existing method.

Finally we apply the same techniques on Huffman binary code trees, as they can be viewed as optimal alphabetic trees that lack the alphabetic constraint. We show different methods and different linear time operations that can be applied on Huffman trees.

### 1.3 Organization of the Thesis

The thesis is organized as follows

Chapter 2 gives an overview on the subject of binary trees in general and introduces the needed terminology and definitions to understand the subject of alphabetic trees

Chapter 3 is a literature survey for all the work done in the subject of optimal alphabetic trees, optimality algorithms, results, applications and parallelization efforts.

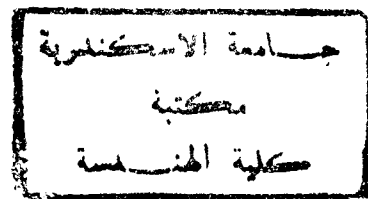
Chapter 4 discusses multidimensional alphabetic trees and optimality problem; existing algorithms are presented and a new one is introduced.

Chapter 5 introduces an algorithm for insertion and deletion form optimal alphabetic trees in linear time.

Chapter 6 shows how the same idea can be used to merge two alphabetic trees in linear time, and uses the merging algorithm to build the tree recursively.

Chapter 7 applies the same ideas on Huffman trees and compare them for other strategies that linearly manipulates a Huffman tree.

Chapter 8 concludes the thesis and gives suggestions for future work.



## Chapter 2: Binary Trees

The hierarchical nature of the tree data structure makes it more suitable for information storing and retrieval than linear structures due to its fast (logarithmic) access time. This chapter provides a quick overview on the subject of binary trees as it is an essential foundation to the study of alphabetic trees. First, it presents the basic terminology and needed definitions in section 2.1. Then section 2.2 discusses binary search trees in some detail; the definition, the basic properties and the operations defined on them with some analysis of it. The following two sections define the optimal binary search tree and the problem of finding it. Then, we put some light on binary coded trees and define alphabetic trees to be discussed deeply in the proceeding chapters. Finally, we discuss the possible extension of binary search trees (and alphabetic trees as well) to higher dimensions.

### 2.1 Basic Terminology

A **tree** is a collection of elements called nodes one of which is distinguished as a root, along with a relation ("parenthood") that places a hierarchical structure on the nodes [1]. A node in a tree can have zero or more children; a node with no children is called a *leaf*. A tree with no nodes at all is called the *null tree*.

The *height* of a node in a tree is the length of the longest path from that node to a leaf. The height of a tree is the height of the root. The *depth* (or the *level*) of a node is the length of the unique path from the root to that node.

#### Binary Trees

A **binary tree** is a tree in which every node has either no children, a left child, a right child or both a left and a right child. A binary tree is said to be *complete* (or *full*) if for some integer  $k$ , every node of depth less than  $k$  has both a left and a right child and every node of depth  $k$  is a leaf. A complete binary tree of height  $k$  has exactly  $2^{k+1}-1$  nodes [2].

#### Methods for traversing a binary tree

There are several orders in which the nodes of a tree can be visited (traversed). The three most important orders are defined recursively as follows.

##### **Preorder Traversal:**

Visit the root, then visit the left subtree in preorder, then visit the right subtree in preorder.

##### **Inorder Traversal:**

Visit the left subtree in an inroder traversal, then the root, then the right subtree in inorder.

### ***Postorder Traversal:***

Visit the left subtree in postorder, then the right subtree in postorder, then the root.

## **2.2 Binary Search Trees**

The fact that the binary search algorithm gives the minimal expected number of comparisons for an equally likely list of elements ( $\log(n+1)$  comparisons for  $n$  elements), clarifies the use of binary trees for retrieving objects associated with keys having linear order ( $<$ ).

A *binary search tree (BST)* is a binary tree that satisfies the following property:

All elements stored in the left subtree of any node  $x$  are less than the element stored at  $x$ , and all the elements stored in the right subtree of  $x$  are greater than the element stored at  $x$ .

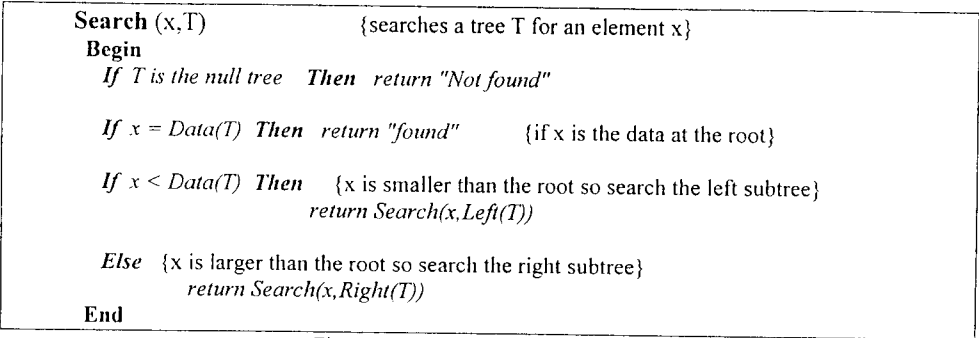
An inorder traversal of a BST gives the elements in the nodes sorted in ascending order which is sometimes called proper symmetric order of the nodes.

### **Operations defined on binary search trees**

Binary search trees are often used in information systems to store and retrieve data. The operations of adding, deleting, updating, or retrieving an element in a BST are basically done through a search operation. Deletion, updating and retrieval consist of a successful search followed by processing the found element; while the insertion process is an unsuccessful search followed by inserting the element in its proper position (in the right or the left of the last visited leaf according to its value).

### **Searching a BST**

The search process starts at the root and uses comparisons at each node to work on only one of its subtrees according to the binary search tree property. Thus, a successful search for a node  $x$  takes  $l_x+1$  comparisons where  $l_x$  is the level of node  $x$ ; an unsuccessful search takes  $l_k+1$  where  $l_k$  is the level of the last visited leaf which equals  $l_x[3]$ . Fig. 2.1 gives a pseudo code for the search.



**Two way comparison**

The three-way comparison implicitly assumed in the search (less than, equal, or greater than) is implemented via two if-statements in the pseudo code in Fig.2.1. This is because programming environments ordinarily support two-way comparison and branch. In 1991, Andresson [4] introduced a small change to the search tree and the algorithm that uses two-way comparisons at each visited node; the left branch is taken if the search argument is less than the visited node and the right branch is taken in case of greater than or equal. The search starts at the root and works down the tree till a comparison is done with a leaf node, then *a pointer is added* that points back to the last node at which a right branch was taken. Finally, an equality comparison is done with the key at that node; if not equal then the node is not in the tree (unsuccessful search).

In this method the search is carried out to a leaf node even if the search argument is in a node at a higher level. However, the amount of work done at each node is less, and thus the average time to complete a search is likely to be less. An important remark is that in this way, successful and unsuccessful search will have exactly the same steps and complexity.

**2.3 Optimal Binary Search Trees**

For a given finite set of elements there is a large number of binary trees that satisfies the binary search property; when n is large there are  $\frac{C_n^{2n}}{n+1} \approx \frac{4^n}{\sqrt{\pi} n^{3/2}}$  binary trees [3]. The measure of optimality of a BST is the average computation time for successful and unsuccessful search.

**Frequency of access**

So far it was assumed that each key is equally likely as a search argument. In a more general situation, each key will have a different access frequency (probability of being a search argument) and the expected time for a successful search should be calculated using these probabilities. A trivial and obvious example is the letters of the alphabet in storing a phone directory where the access frequency vary dramatically between letters (take 'a' and 'z' as an example). Table 2.1 presents the most common 31 words in English with their relative frequencies [5], knuth showed in his book [3] how considering these frequencies in finding the optimal BST decreases the average number of comparisons for a successful search from 4.393 to 3.437. A lot of practical examples from real database systems can be found in [6].

Word	a	and	are	as	at	be	but	by	for	From	had	have
Freq	5074	7638	1222	1853	1053	1535	1379	1392	1869	1039	1062	1344

Word	he	her	his	I	in	is	it	not	of	on	or	that
Freq	1727	1093	1732	2292	4312	2509	2255	1496	9767	1155	1101	3017

Word	the	this	to	was	which	with	you
Freq	15568	1021	5739	1761	1291	1849	1336

Table 2.1: the most common 31 words in English with their relative frequencies

Similarly, the frequency of access should be considered in the unsuccessful case too; the unsuccessful search is represented through external nodes.

### External and Internal nodes

For each leaf in the tree a right and left child are added to represent the range of values between two adjacent keys (for the case of unsuccessful search); these nodes are called *external nodes* while the nodes representing the keys are called *internal nodes*. External nodes appear in figures as square nodes, and internal nodes appear as circular nodes. A tree with external nodes added to it is called *extended* (or *full*) *binary tree*; an extended binary tree with  $n$  internal nodes has  $n+1$  external nodes [7].

### The general binary search tree

A general BST corresponds to a full binary tree with one key stored at every internal node, and none at external nodes. The keys are distributed according to the binary search tree property that was mentioned earlier. Fig. 2.3 presents a general BST of 3 nodes where circles refer to internal nodes and squares refer to external ones.

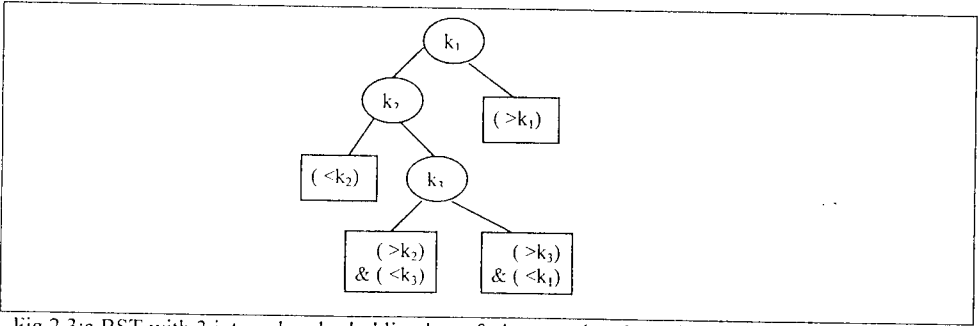


Fig 2.3:a BST with 3 internal nodes holding keys & 4 external nodes referring to ranges between them

## 2.4 The Problem of Finding the Optimal BST

Given  $2n+1$  probabilities  $p_1, p_2, \dots, p_n$  and  $q_0, q_1, \dots, q_n$ , where

$p_i$  = probability that  $k_i$  is the search argument;

$q_i$  = probability that the search argument lies between  $k_i$  and  $k_{i+1}$ .

( $q_0$  is the probability that the search argument is less than  $k_1$  and  $q_n$  is the probability that the search argument is greater than  $k_n$ )

The problem is to find a binary tree that minimizes the expected number of comparisons in the search, namely

$$\sum_{1 \leq j \leq n} p_j(l_j + 1) + \sum_{0 \leq k \leq n} q_k l_k \quad (1)$$

where  $l_j$  is the level of the  $j^{\text{th}}$  internal node in proper symmetric order and  $l_k$  is the level of the  $(k+1)^{\text{st}}$  external node, and where the root has level zero.

Since the  $p$ 's and  $q$ 's represent probabilities, they must sum to unity. However, in this definition there is no need to require that; the probabilities could be replaced by access frequencies or weights. Subsequently, equation (1) is said to represent *the cost function* of the tree and the optimal tree is the one with minimum cost for the same set of weights [3].

If the key distribution is assumed to be uniform, all nodes have the same access frequency, then all  $p$ 's and  $q$ 's are equal and the problem reduces to find the most balanced BST. However, to solve the general case several approaches have been proposed in the literature.

### Dynamic Programming

The important principle that is used in finding the optimal binary search tree is that *all subtrees of optimum tree are optimum*. Thus a computation procedure that systematically finds larger and larger optimum subtrees will end by finding the optimal tree; the general approach is known as dynamic programming [3,8].

If  $c(i, j)$  is the cost of the optimal subtree for the list of nodes (internal and external) from  $i$  to  $j$ , then it can be defined recursively as follows for  $0 \leq i \leq j \leq n$ .

$$c(i, i) = 0$$

$$c(i, j) = w(i, j) + \min_{i \leq k < j} (c(i, k-1) + c(k, j)), \quad \text{for } i < j \quad (2)$$

where  $w(i, j)$  is the sum of weights for the internal and external nodes from  $i$  to  $j$

$$w(i, j) = \sum_{k=i+1}^j p_k + \sum_{x=i}^j q_x \quad (3)$$



The values of  $k$  that achieves the minimum in equation (2) are the possible roots of the optimal tree.

The algorithm is to evaluate  $c(i, j)$  for all defined values for  $i, j$  starting from the smaller to the larger (about  $\frac{1}{2}n^2$  values). The minimization procedure is carried for each pair (there are  $j-i$  values of  $k$  in each case which leads to a total of about  $\frac{1}{6}n^3$  values). Thus it takes  $O(n^3)$  time and  $O(n^2)$  space to find the optimal binary search tree using dynamic programming.

### Removing a factor of $n$

In 1971, Knuth [9] refined the previous technique and removed a factor of  $n$  from the running time using the *monotonicity* property. The root of the subtree for the list of nodes from  $i$  to  $j$  must be larger than or equal (in order) the root for the nodes from  $i$  to  $j-1$  and less than or equal the root for nodes  $i+1$  to  $j$ . Formally speaking,

If  $R(i, j)$  denotes the set of possible roots for the subtree from nodes  $i$  to  $j$

Then for every  $r(i, j) \in R(i, j)$ , the following formula is satisfied

$$r(i, j-1) \leq r(i, j) \leq r(i+1, j)$$

This limits the search for the minimum, since only values in this range need to be examined in equation (2). It can be shown [3,9] that this reduces the total running time to  $O(n^2)$ .

### Heuristics

For large values of  $n$ , it becomes impractical to use the previous algorithm. Therefore, some ideas were proposed in the literature to produce a fairly good BST [3,10]. One idea is to insert the keys in decreasing order of frequency, however the resultant tree is not usually near to the optimum since it does not make use of the weights of the external nodes. Another approach is to choose the root such that the sum of weights for the left and right subtrees are nearly equal, this may also fail since it may result in a node with a very small weight to be the root.

In 1972, Walker and Gotlieb [11] suggested a reasonable compromise. They try to equalize the weights of the left and right subtrees but with the ability to move the root a few steps to the left or the right to find a node with relatively large weight to be the root. Then,

the algorithm continues to work in the left and right subtrees in a top-down manner. This heuristic yields fairly good trees and takes  $O(n \log n)$  time and  $O(n)$  space.

### Modifying Binary Search Trees

Several approaches have been mentioned to find the optimal BST. Another problem is after the optimal tree is found a sequence of deletions and insertions could turn it to a non optimal one which gives a disadvantage of BSTs in storing data of dynamic nature. In the general case of BST there is no solution except to rebuild the tree after a set of successive insertions and deletions. For the case of uniform access time several techniques have been suggested to keep the balance of the tree [10].

AVL trees [12,13] are a special kind of BSTs that perform insertions and deletions in a way that the difference in height between the left and right subtrees of any node in the tree is at most one. The increased time complexity for the modified deletion (and insertion) process remains in the logarithmic order.

B-trees and  $B^+$ -trees [12,13] are a special kind of BSTs that is more suitable for secondary storage. They are balanced multi-way trees, where nodes contain a range of sorted key values to minimize disk accesses.  $B^+$ -trees differ in that internal nodes contain only entries that direct the search while all data is stored in leaf nodes (each leaf leads to a disjoint interval of key values). Like AVL trees, the modified editing operations remain in the logarithmic order.

## 2.5 Binary Code Trees

A binary code tree corresponds to a full binary tree with one key stored at every external node, and none at internal nodes. It associates binary codes with keys (external nodes only); the code associated with each key is the sequence of left/right child links formed by the path from the root to the external node holding the key. The codes resulting from a binary code tree has an important feature called *the prefix property*; no code for a key is the prefix of a code for any other key, this feature allows the reverse process of decoding a string of binary codes. Binary code trees are usually used for compression purposes.

### Optimal binary code trees

An optimal binary code tree is a one that produces minimum redundancy code; it minimizes the average length of a code, weighted for expected frequency of use. An optimal

binary code tree differs from an optimal binary search tree in that all internal nodes have a weight of zero. In addition, it will not usually have the external nodes weights in the proper symmetric order; an in-order traversal of the nodes in a code tree will not necessarily scan the keys in ascending order.

### Huffman tree

One technique for finding minimum redundancy codes was discovered in 1952 by Huffman [14]. It builds the tree, widely known as Huffman tree, in a bottom up manner as follows.

The two keys with the lowest weights are replaced with a compound key which has the sum of their weights; this corresponds to forming an internal node in the tree with these two nodes as left and right children. The process is repeated till one key remains with the sum of weights; this key corresponds to the root of the tree.

Fig. 2.4 shows an example for the construction of Huffman tree and the resulting codes for a set of five keys. It shows how keys with larger weights get shorter codes and that all codes have the prefix property. It is also noticeable that nodes do not appear in the proper symmetric order in the formed tree; node (6) appears before node (4) (since it is joined with the first node (7)).

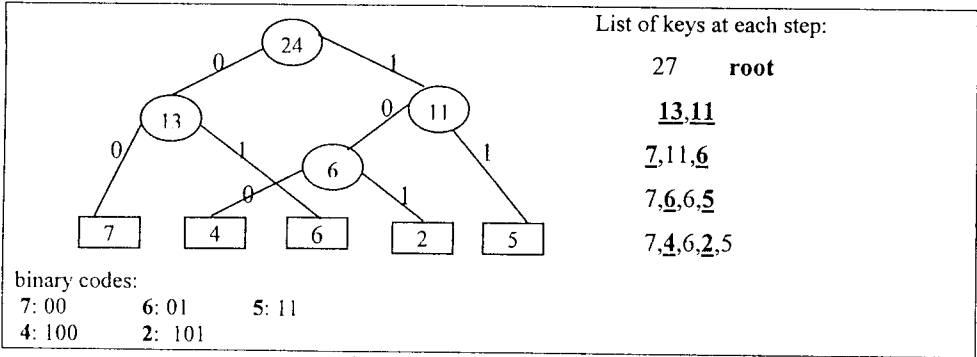


Fig 2.4. Huffman tree for 5 keys

Huffman's algorithm takes  $O(n \log n)$  time to build the tree. However, in 1987 Larmore [15] found that it can be implemented in *linear time* if the weights of leaves are already sorted.

## 2.6 Alphabetic trees

In 1959, Gilbert and Moore [8] considered an *alphabetic constraint* on binary code trees which restricts the ordering of leaves in the proper symmetric order (like BSTs). The resulting tree can be viewed as a binary search tree with zero weighted internal nodes [16,17], and thus the optimal alphabetic tree OAT is the one with minimal cost. The algorithm they introduced to find the OAT is the dynamic programming technique stated earlier, and Knuth refinement on it applies to them too. As it will be further explained, algorithms do exist to find the OAT in  $O(n \log n)$  time complexity [3].

### Optimal alphabetic tree vs. Huffman tree

Optimal alphabetic trees put an additional constraint of reserving the initial order of the weight sequence in the resulting tree. This may result in an OAT with larger cost (or average code length in other words) than the corresponding Huffman tree for the same weight sequence. Fig. 2.5 gives the OAT for the sequence of weights used in Fig. 2.4 to build Huffman tree; the cost of the Huffman tree was 54 giving an average code length of 2.25 while the cost of the OAT is 55 giving an average code length of 2.29. Note also, that the code words do not have the prefix property.

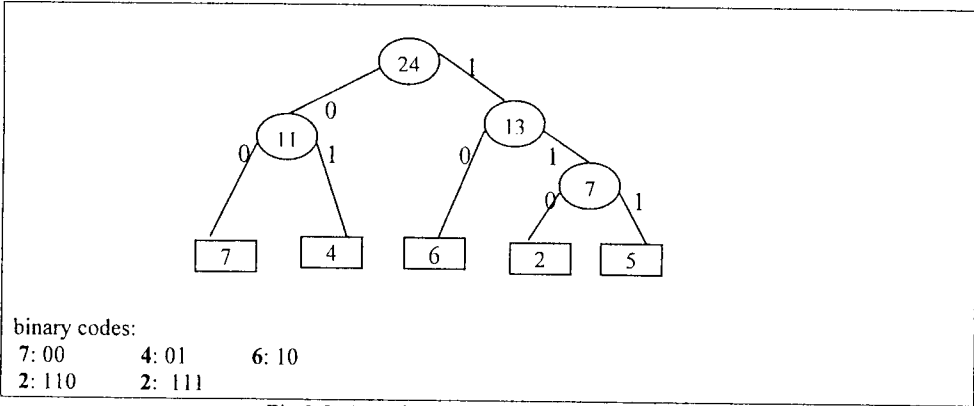


Fig 2.5: OAT for the weight sequence in fig 2.4

Another difference between the two trees is in the encoding process. Due to the lack of symmetric order, Huffman trees need an additional hashing mechanism to resolve the correspondence of the plain text letter to its code word. On the other hand, OATs are sufficient for both encoding and decoding [18].

### Optimal alphabetic tree vs. optimal binary search tree

Alphabetic trees are a subset of binary search trees by definition; the only difference is that internal nodes have zero weights. However, if the 2 way comparison introduced by Andresson [4] is used, the weights of the internal nodes makes no difference as the search always proceeds to the leaves. In 1997, Hu and Tucker [19] showed how this modification fits the alphabetic tree model and considered the potential advantage of using alphabetic trees for binary search regarding the fact that OATs (sometimes called optimal alphabetic binary search trees OABSTs) have faster construction time than OBSTs.

Optimal alphabetic trees can be used not just in encoding/decoding and binary search, but in many other applications as will be explained in the next chapter.

### 2.7 Multi Dimensional Case

When the access of data depends on multiple keys, information needs to be stored in a multi dimensional structure. Binary search trees are extendable to higher dimensions, but the problem of finding the optimal BST in higher dimensions is known to be *NP* complete. However, there are generalization attempts that assumed uniformity of access and independent key values. The k-d tree [20], is a good example where at each level of the tree the value of a different key is tested to direct the branch, keys are taken in a successive circular order till a leaf is reached.

Some work was done to solve the problem in the alphabetic tree case. An approximate algorithm was introduced in [21] for the 2-d case, also some definitions and results can be found in [6]. Chapter 4 is devoted to optimal alphabetic trees in the multi dimensional case.

## Chapter 3: Optimal Alphabetic Trees Research Work

An optimal alphabetic tree (OAT) is one with minimal cost over all possible trees for the same set of weights. This chapter presents an overview of the research done on OATs in the last few decades. It first describes the two most known algorithms for finding the optimal alphabetic tree. Then a summary of the related research results is presented. Finally, different applications for OATs and the work done on them are mentioned.

### 3.1 The Hu-Tucker Algorithm

This section describes the well known algorithm for the construction of OATs introduced by T. C. Hu and P. Tucker in 1971 [16]. Knuth has found an implementation for the algorithm in  $O(n \log n)$  time and  $O(n)$  space [3]. First we describe the algorithm then we discuss its implementation.

#### 3.1.1 Algorithm

The algorithm consists of three phases can be described as follows.

**Phase1** : Combination

Let  $q_i$  denotes the weight of the node or the node itself, when two square (external) nodes  $q_i, q_j$  combine they form a circular node (internal node) with weight  $q_i + q_j$  occupying the position of the left child in the weight sequence.

Due to the alphabetic constraint, two nodes can only combine if they form a compatible pair. Two nodes in a sequence form a compatible pair if they are adjacent in the sequence or if all nodes between them are internal nodes. Among all compatible pairs in a weight sequence, the one having the minimum weight is called the minimum compatible pair.

To break ties (resolve equalities), the Hu-Tucker algorithm uses the convention that the node on the left has a smaller weight.

A pair of nodes  $(q_j, q_k)$  is a **local minimum compatible pair (LMCP)** if

$$q_i > q_k \quad \text{for all nodes } q_i \text{ compatible with } q_j$$

$$q_j \leq q_i \quad \text{for all nodes } q_i \text{ compatible with } q_k$$

The first phase of the algorithm keeps forming *LMCPs* until a tree is formed.

However, this tree, sometimes called the *level tree*, is not alphabetic.

**Phase 2 : Assigning Levels**

Uses the tree built in phase1 to find the level of each node.

**Phase 3 : Reconstruction**

Uses a stack algorithm to construct an alphabetic binary tree based on the node levels found in phase 2. Starting from the maximum level, pairs of two adjacent nodes (in that level) are taken from left to right; each pair combines to form an internal node that appears in the higher level in place of the left node.

Phases 2,3 of the algorithm can be easily implemented in  $O(n)$  time and space [22], so the complexity of the algorithm depends on the implementation of phase1 (sec. 3.1.3).

**3.1.2 Example**

This example describes the steps of building the OAT using the Hu-Tucker algorithm. Fig 3.1 shows the resulting tree from phase 1 for the 9 nodes weight sequence shown in the external nodes. Internal nodes are marked by their order of formation.

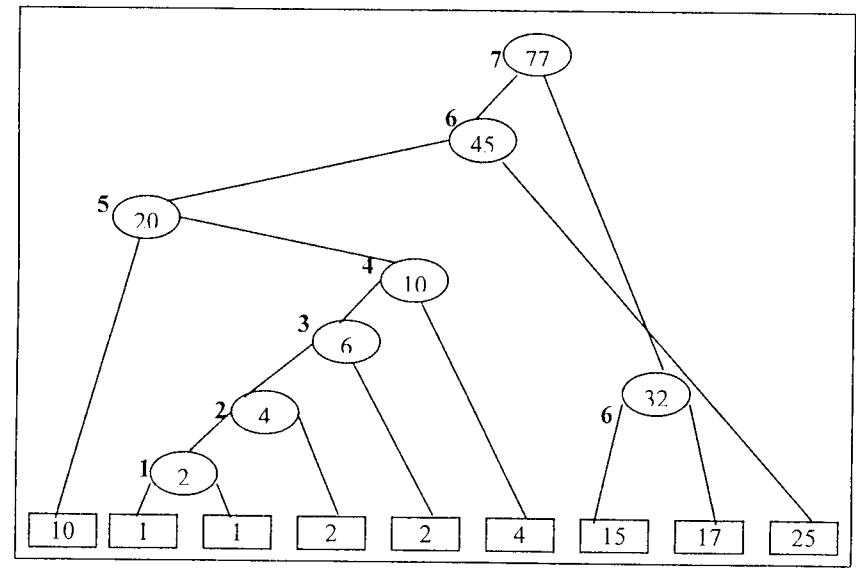


Fig. 3.1 the tree of phase1 of the Hu-Tucker algorithm

The tree in Fig. 3.1 are used to get the levels in a top-down manner, and the levels are shown in table 3.1.

Node	10	1	1	2	2	4	15	17	25
Level	3	7	7	6	5	4	2	2	2

Table 3.1: the levels of the tree in Fig. 3.1

Then, the OAT is found by applying phase3 (Fig. 3.2). Notice the difference from the tree of phase1 (Fig.3.1), where the external node (15) combined with the internal node (20) instead of the external node (17). Yet, the levels are the same.

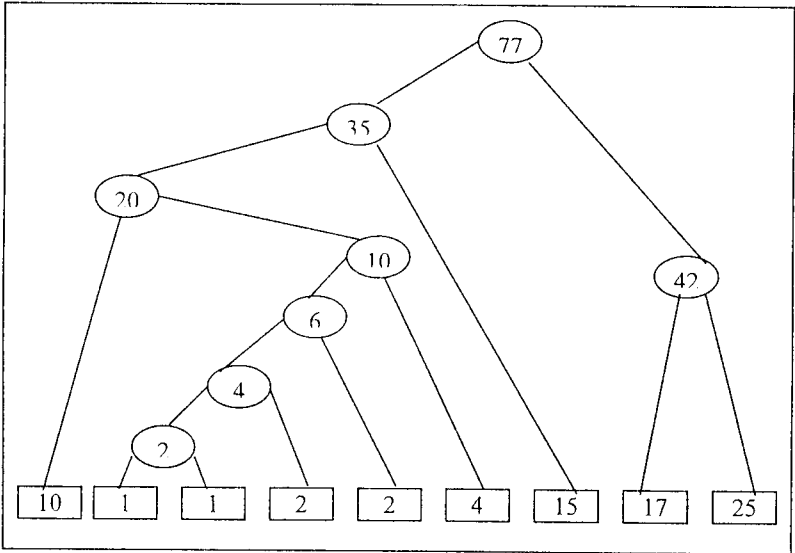


Fig. 3.2 : the OAT after applying phase3 to the weight sequence in table 3.1

### 3.1.3 Implementation

A direct implementation of phase1, the *LMCP* method as called by [18], leads to  $O(n^2)$  time. At each step, the weight sequence is scanned to find the *LMCP*. In 1972, J.M. Yohe [23] first wrote this implementation in Algol 60. Then, Knuth described in his book [3] an implementation of phase1 that, although rather complicated, achieves  $O(n \log n)$  time and  $O(n)$  space; the following is a brief description of it.

Finding the *LMCP* can be viewed as actually scanning compatible sets of nodes and finding the minimum summation in each set, the two minimum nodes, then the *LMCP* is the minimal of those minimums. Where compatible sets of nodes mean the sequence of internal nodes between each two external nodes in the current weight sequence; these compatible sets are called Huffman priority queues *HPQ* since they combine according to the rules of Huffman trees. A square node is included in both its left and right queues, when a square node is chosen in the *LMCP*, its left and right



queues are merged into one *HPQ*. Similarly, when an internal node is chosen, its deleted from the corresponding queue, and the new *LMCP* is inserted in the corresponding queue.

Thus, the problem reduced to finding a suitable data structure to implement the priority queue that can retrieve the minimum, insert, delete, and also be merged in  $O(\log n)$  time. This data structure is the *Leftist tree* (or heap), each *HPQ* is stored in a leftist tree, and the minimum summations of all the *HPQs* are stored in a master priority queue; another leftist tree with its minimum holding the current *LMCP*. More information about the leftist tree data structure and its operation can be found in [3,24]; a pseudo code of these operations and of both implementations of the Hu-Tucker algorithm can be found in [18].

### 3.2 The Garsia -Wachs Algorithm

In 1977, Garsia and Wachs introduced a similar algorithm to find the OAT [17]. The Hu-Tucker and Garsia-Wachs algorithms differ only in phase1; specifically, in the manner in which they decide which pair to combine and where the combined node is reinserted but the final tree is the same.

In Garsia-Wachs algorithm, the rightmost minimal pair combines. When combined, the resulting node moves to the right as far as it can without passing over a heavier node (a node with larger weight) or reaching the end of the weight sequence; the node is then inserted at this point. In other words, the new node is inserted to the left of the first heavier node to the right of the old nodes; if there is no such node, it is inserted at the end of the weight sequence.

A simple example that illustrates the difference is a 3 nodes weight sequence. If the three weights are equal, there are two possible OATs, each algorithm will produce a different one (Fig.3.3).

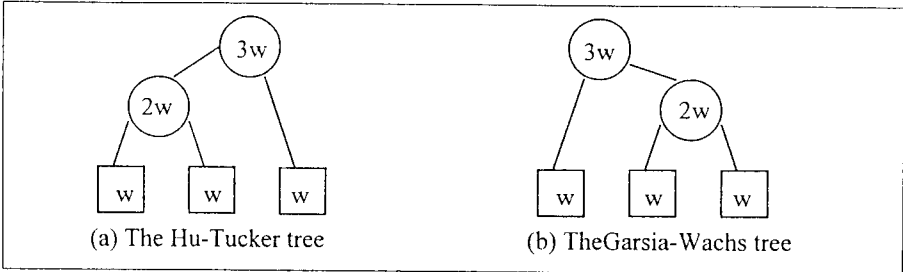


Fig. 3.3: an OAT for 3 equal weight nodes

A generalized version of the Garsia-Wachs algorithm, where any local minimal pair can combine (not necessarily the rightmost one) appears in [25,26]. The authors used it in developing their research results; as it is easier when to start without the position constraint existing in both algorithms (sec. 3.3).

### 3.3 Related Results

This section passes briefly through most of the research results related to the construction of OATs (although some of them are not relevant to the thesis).

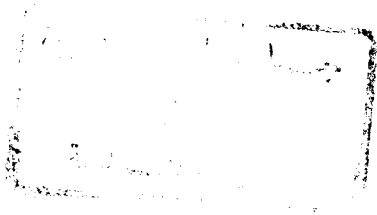
#### Correctness Proofs

Correctness proofs of the Hu-Tucker algorithm first appeared in [16,22]. However, these proofs were rather intricate, even Knuth doubted that a simpler one will ever be found [3]. Until very recently in [25], no other attempts appeared in the literature.

As for the Garsia-Wachs algorithm, it has a simpler but very technical proof based on several claims proved simultaneously by induction [17]. Another proof of the restricted version appears in [27].

Recently, in 1996, a more structural proof was found for both algorithms [25]. The proof was first conducted for the general case of the Garsia-Wachs algorithm, then a simple mutual simulation between the two algorithms is shown to prove the correctness of the Hu-Tucker algorithm.

It is also worth mentioning that testing the optimality of a given alphabetic tree was shown to be linear in several cases [28].



## Time Enhancements

The best known lower bound for the OAT problem is  $\Omega(n)$ . Linear time solutions for sorted sequences and bitonic sequences was found as early as in [22]. Then, Klawe and Mumey [29] gave an  $\Omega(n \log n)$  lower bound for certain classes of methods that include those methods that construct the *level tree*; i.e. the tree resulting from phase1 can not be constructed in less than  $\Omega(n \log n)$ . In spite, they gave linear time solutions for weights exponentially separated and weights within a constant factor.

In 1994, L.Larmore and T. Perzytycka [30] related the complexity of the general OAT problem to the complexity of priority queue operations and the complexity of sorting (recall the Hu-tucker implementation in sec 3.1.3). They reached an  $O(n \log P(k))$ -time algorithm, where  $k$  is a number at most as large as the number of local minima (number of elements in the master priority queue), and  $P(k)$  is the complexity of the priority queue operations; this gives a reduced complexity in some cases.

Then in 1996, Hu and Morgenthaler [31] achieved the same results of [29]&[30], along with some new conditions on node combining, in a simpler methodology.

## Parallelization Efforts

Earlier parallelization attempts appeared in [32,33] based on the dynamic programming solution; the resulting algorithm runs in  $O(\log^2 n)$  time using roughly  $n^6$  processors. Then, the results appeared for the simpler case of Huffman coding (no alphabetic constraint) encouraged researchers to extend them to the OAT case. An example is an  $O(\log^3 n)$  time algorithm using  $(n^2/\log n)$  processors [26]; the algorithm is based on the generalized version of the Garsia-Wachs algorithm. The same authors suggested in [34,35] approximate algorithms that finds a nearly optimal OAT.

## Height Limited OATs

The  $L$ -height limited OAT is the one with minimal cost among those with their maximum height less than or equal a given constant  $L$ . Algorithms for finding such a tree were developed in the seventies [36,37] with  $O(n^2 L)$  time complexity. Then a solution for the Huffman case (the package merge algorithm) was extended to find the  $L$ -limited OAT [38,39]. A parallel version of the same algorithm appeared in [35].

### 3.4 Applications where OATs are used

In this section we point out the applications where OATs can be used.

#### Information Retrieval

Optimal alphabetic trees have the advantage of handling skewed data distributions. Most new information systems contain data that has different access frequencies. A faster average access time is achieved when shorter paths are assigned to higher frequencies. The fact that OATs assign weights to external nodes only (unsuccessful search) was overcome in [19] using the results of [4] (sec 2.6). In addition, OATs can be constructed faster than optimal BSTs which are constructed in  $O(n^2)$  time. The use of OATs for associative queries in databases was introduced in [40].

#### Coding and Compression

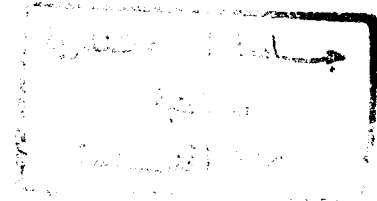
Another area where OATs can be used is data compression and decoding. Alphabetic codes provide symmetry in the encoding and decoding processes in contrast to Huffman codes, where an additional hashing mechanism is needed for the decoding. The research in alphabetic codes was revived in the 90's for the excessive information transfer in many applications. Bounds for the redundancy in alphabetic codes were found in [41,42]. An algorithm for constructing nearly optimal alphabetic code was introduced in [41]. Results for the L-restricted alphabetic codes (L-height limited OATs) appeared in [43].

A practical experiment of compressing the complete work of William Shakespeare was done in [18]. The compression ratio achieved by OATs where compared to that of a fixed-length encoding, and two well-known compression programs: the standard Unix *compress* and *gzip*; alphabetic codes were found to be superior. Other experiments were also done to compare alphabetic codes to Huffman codes (using random inputs); the results show, with confidence, that the saving in Huffman codes stabilizes around 0.09 bit per word on average [18].

## Routing Applications

OATs can be used in any application where items are assigned different weight values and it is desired to minimize an average cost function of these weights. Examples range from *address lookup in routers* that uses destination based routing where different addresses have different access frequencies [44], to *minimizing the interconnect routing delay* in VLSI design where different electrical paths have different access frequencies [45].

The importance of this topic comes from the fact that routing accounts for about 40-80% of total chip area and about 40-60% of circuit delay [46]. Hence routing delays should be minimized at all levels of design. More details on the subject can be found in [45,46,47].



## Chapter4: Multi Dimensional Optimal Alphabetic Trees

All alphabetic trees mentioned in previous chapters were of 1-dimensional nature. Alphabetic trees are well extendable to multi dimensions; multidimensional OATs can be used to store data (or codes) of multidimensional nature like in multimedia and visualization applications. However, finding multidimensional OATs is not as easy as the one dimensional case. This chapter discusses multidimensional alphabetic trees and the algorithms to find optimal ones focusing on the 2-d case for simplicity; yet all algorithms mentioned here are extendable to higher dimensions. It starts with an introductory section that gives definitions and examples of multidimensional alphabetic trees and the optimality problem, followed by discussing the dynamic programming solution for the problem. Then it proceeds by a section on special cases and some results for the multi-dimensional OAT, next a heuristic faster algorithm that finds an approximate OAT based on this results is presented. After that, we introduce a new concept that we call the Expense of a cut. we use it as a measure of goodness in choosing the roots of subtrees and follow it with an algorithm for finding the 2-d OAT in a reduced time based on this concept. Then, we show experimental results and performance evaluation of that algorithm. Finally, the chapter is ended with a conclusion on handling 2d OATs.

### 4.1 Introduction

Multidimensional trees can be used to store data that is accessed through the value of multiple keys. *Multidimensional alphabetic trees* are recommended when the frequency of access depends on the value of all the keys together (i.e. not independently) and the distribution of data elements is not uniform over the keys space [6].

#### 4.1.1 Two dimensional alphabetic trees

In the 2 dimensional case, where the weights (access frequencies) depend on the value of 2 keys X,Y, the weights are represented as an M x N 2-d array, where M is the number of bounding key values for Y, and N is the number of bounding values for key X (fig 4.1). The array elements are the corresponding weights; i.e.  $w_{ij}$  is the access frequency of elements satisfying  $Y(j-1) < Y \leq Y(j)$  and  $X(i-1) < X \leq X(i)$

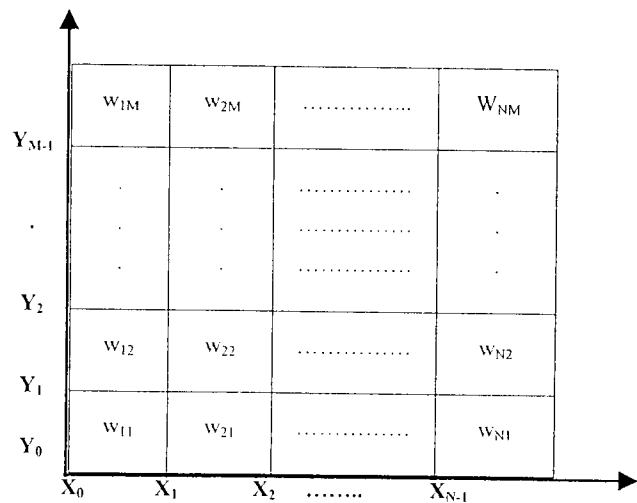


Fig. 4.1: Two dimensional array of weights

The tree is like a 1 dimensional one with  $MN$  leaves, the weight of each leaf is the weight of the corresponding cell in the array. Follows is an example of a 2-d array of weights and how the 2-d alphabetic tree for it looks like.

**Example**

Fig.4.2 shows a possible alphabetic tree for the 4x4 array of weights given in Table 4.1.

	26	62	44	83
$Y_3$	130	40	27	49
$Y_2$	209	168	16	60
$Y_1$	28	119	50	12
	$X_1$	$X_2$	$X_3$	

Table 1: a 4x4 2-d array of weights

Each leaf in the tree corresponds to a certain entry in the array. To access a certain element say with key values  $X < X_1$  &  $Y > Y_3$ , from the array we know that elements satisfying such condition have frequency of access (26). The root is examined and the search is directed to the left subtree, where  $Y_3$  is examined and the right subtree is selected. Then  $X_1$  is examined, and the left leaf is our target.

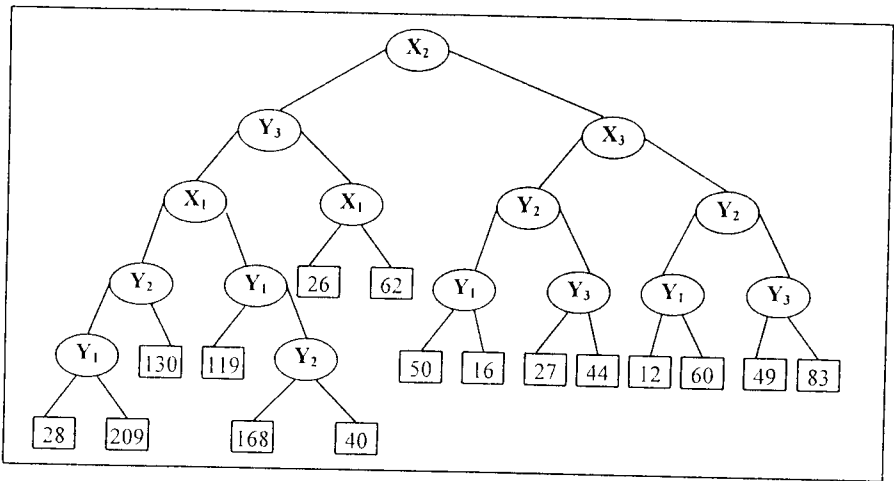


Fig.4.2: 2-d array of weights for the table 4.1

#### 4.1.2 Two dimensional optimal alphabetic trees

Like the 1-d case, a tree is considered optimal if its cost is minimal over all possible trees for the same set of weights. From the previous example, we can see that whenever a root of a subtree is examined, a part of the 2-d array is excluded from further examination. The working area gradually shrinks to reach a cell at the final step when a leaf is reached. Hence comes the isomorphism.

The problem of obtaining the OAT for the  $M \times N$  two dimensional array of weights is equivalent to the problem of minimizing the cost of cutting the array into individual cells using planar cuts along the edges, where the cost of a cut is equal to the sum of the weights of its two parts. Fig 4.3 shows the cost of cutting a general 2d array of weights like that in Fig.1 where L,R refer to the resulting left and right arrays after cutting at  $X_i$ , W refers to the sum of weights of the 2d array, and C to the cost. The root of the tree corresponds to the first cut. In what follows, we will interchangeably use the words root and cut.



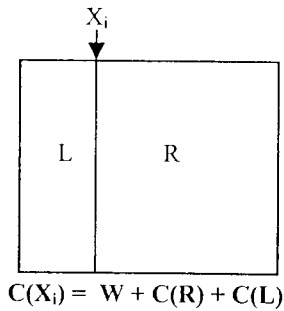


Fig. 4.3: The equivalence between finding the 2d OAT and cutting a 2d array into individual cells

## 4.2 Dynamic Programming

The  $O(N^2)$  dynamic programming technique described in chapter2 [3,9] can be extended to find the optimal alphabetic tree in higher dimensions in the same manner. Each possible subtree is examined considering all values for all keys, or In other words all possible cuts are examined at each step till we reach individual cells . The two dimensional problem can be solved in polynomial time ( $O(N^3M^2)$ ); however, this is not fast enough for many applications. Also, the computation becomes more excessive when the dimensions increase due to the enormous number of possible trees in this case; that is, the power of the polynomial function increases with the dimension ( $O(N^{2k+1})$  where  $k$  is the dimension). Thus, comes the need to find other faster algorithms, that could possibly be extensions of Hu-Tucker or similar  $O(N \log N)$  algorithms.

## 4.3 Related Results

Before we proceed to discuss methods for finding the 2-d OAT, we introduce some special cases where the problem reduces to successive repetition of the 1-d one. A logical and obvious case is first presented, then a theorem that simplifies the complexity for a wider range of 2-d problems and finally a lower bound on the cost of any 2-d optimal alphabetic tree is concluded.

### Independence of keys

When the two keys  $X, Y$  are independent then

$$\text{Probability}\{X=x, Y=y\} = \text{Probability}\{X=x\} * \text{Probability}\{Y=y\},$$

and clearly the same applies for access frequencies (weights)

This results in all rows being multiples of the original weight vector of  $X$ ; similarly all columns are multiples of the weight vector of  $Y$ . In this case, minimizing the search time for one key does not depend on the distribution of values for the other key. The optimal tree can be simply found by searching for the key values interchangeably (like in  $k$ -d trees), or by building the optimal tree for one key and attaching the other key optimal tree to each leaf. The logical meaning is to search for each key independently using its 1-d OAT. The complexity then reduces to the 1-d problem.

### Theorem

A more general case where the optimal tree can be found in the same manner is in the following theorem [6].

*Given  $W$  an  $M \times N$  2-d array of cells and a non negative weight  $w_{ij}$  assigned to each cell  $(i,j)$  then if all the rows (columns) of the 2-d array have the same optimal alphabetic tree  $T$ , then the optimal tree of the 2-d array can be obtained by replacing each terminal node of  $T$  with the optimal tree of the corresponding column (row). The resulting tree is optimal with cost*

$$C = \sum_{i=1}^M T(R_i) + \sum_{j=1}^N T(C_j) \text{ where } T(R_i), T(C_j) \text{ denote the cost of the optimal trees for row } i \text{ and column } j \text{ respectively.}$$

When the theorem applies, the complexity reduces to  $M+N$  times of the complexity of finding the 1-d OAT; i.e.  $O(MN \log N)$  (assuming  $O(N) \geq O(M)$ ). The theorem generalizes to any two weight vectors that their multiplication leads to the given 2-d array of weights. The proof of this theorem depends on the summing property of 1-d OAT that can be stated in the following theorem [6].

*If  $X, Y$  are two weight vectors having exactly the same tree structure for their optimal alphabetic trees with costs  $C_x$  and  $C_y$ , then the weight vector  $aX + bY$  will have exactly the same tree with cost  $aC_x + bC_y$ ; where  $a, b$  are positive constants.*

The approximate algorithm presented in the next section depends also on this theorem.

**Bound**

The cost of the 2-d optimal alphabetic tree ( $T_{\text{optimal}}$ ) is bounded by

$$T_{\text{optimal}} \geq \text{Bound} = \sum_{i=1}^M T(R_i) + \sum_{j=1}^N T(C_j) \tag{1}$$

This lower bound is achieved when all the rows, or all the columns, have the same optimal tree; i.e. when the previous theorem applies. Another example of such case is when one of the dimensions equal 2; i.e. the array has two rows or two columns where the optimal tree is found by cutting at this dimension.

**4.4 Approximate Algorithm**

Based on the previous theorem, a fast greedy algorithm was presented in the literature [6,21] that finds a nearly optimal 2-d alphabetic tree and is extendable to higher dimensions. The algorithm has an average complexity of  $O(MN \log N)$ , and a worst case complexity of  $O(MN^2 \log N)$ . First, we present the algorithm then we follow it by an example.

**Algorithm**

As stated before, if all rows (columns) of a 2d array of weights have the same tree, then the 2d OAT can be found by attaching the columns (rows) trees to this tree, which is also the optimal tree for the sum of all the rows weight vectors. This algorithm uses the optimal tree for the sum of all rows (columns) as a heuristic to find the root of the tree and similarly every following subtree. Follow is an informal description of the algorithm.

- 1- If one of the dimensions equal 1, then the tree is found using 1d techniques and the algorithm terminates then.
- 2- If one of the dimensions equal 2, then the first cut is at that dimension, the root is found, and the tree of each resulting weight vector is found using 1d techniques and attached to one of the leaves of that root then the algorithm terminates.
- 3- The 1d OAT is found for all the rows; if all trees have the same root then it is taken as the root of the tree. The next step is 6.
- 4- The 1d OAT is found for all the columns; if all trees have the same root then it is taken as the root of the tree. The next step is 6.

5- The general case:

- The weights are summed along all the rows, and the 1d OAT is found for the resulting weight vector.
  - Each terminal node is replaced by the 1d OAT for the corresponding column, and the cost of the resulting tree is calculated.
  - The weights are summed along all columns, and the 1d OAT is found for the resulting weight vector.
  - Each terminal node is replaced by the 1d OAT for the corresponding row, and the cost of the resulting tree is calculated.
  - The root of the tree with less cost is chosen as the root of the optimal tree
- 6- After the root is chosen, all previous steps are repeated for each resulting subtree.

Example

Table 4.2 gives a 3x4 array of weights to illustrate the implementation of the previous algorithm.

	15	40	12	60
$Y_2$	20	100	50	400
$Y_1$	5	2	10	8
	$X_1$	$X_2$	$X_3$	

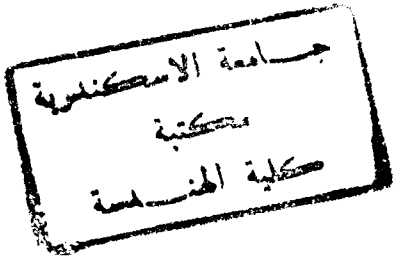


Table 4.2: a 3x4 array of weights

Following the steps of the algorithm, step 3 is reached, the optimal trees are calculated for all rows. All trees happen to have the same root which is  $X_3$ , thus  $X_3$  is chosen as the root of the resulting tree (Fig. 4.4).

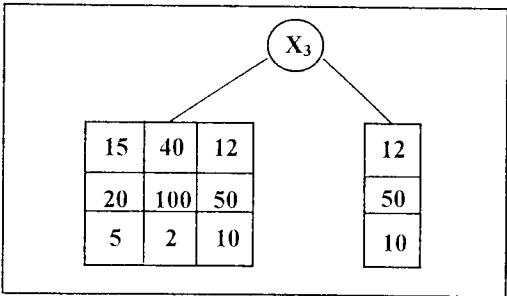


Fig.4.4:  $X_3$  is The root of the resulting tree

After the root is found, step6 is reached and the process should be repeated for the left and right subtrees. The right subtree has a dimension of 1 ( $N=1$ ) and the OAT is found at step1 using 1d techniques and attached to the right of  $X_3$ . For the left subtree, step3 is reached and all row trees are checked for having the same root which did not happen. Then at step4 all column trees are checked and found to have the same root  $Y_2$ . Then, the process is repeated again; the left subtree has dimension 2 x 3 and solved at step2 by choosing  $Y_1$  as the root, while the right subtree is 1 x 3 and solved at step1 (Fig. 4.5).

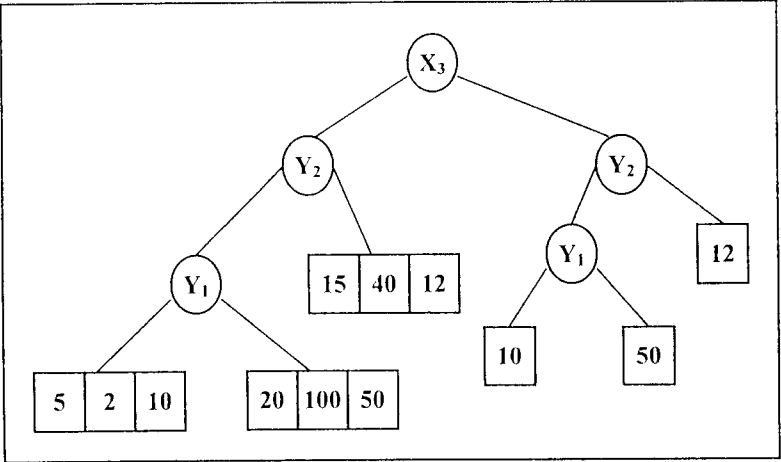


Fig. 4.5 : the resulting after applying the approximate algorithm

### 4.5 The Expense of a Cut

Trying to achieve an exactly optimal tree avoiding examining all possible cuts as in dynamic programming, we introduce a measure of goodness for each cut to limit the number of feasible solutions to the problem and thus, reduce the number of possible choices for the root of the optimal tree. In this section, we define the term “*expense*”, our measure of goodness, for each cut and show how its value is computed. Then, we give some expectation on the ratio of good cuts to all possible cuts [48].

#### Definition

The expense of a cut is defined as *the value this cut has contributed to the deviation from the optimal solution*. A cut is considered “good” if its expense is less than a limit  $L$ .

### The Limit L :

The limit L is set to be:

$$L = T_{\text{approx}} - \text{Bound} \quad (2)$$

Where  $T_{\text{approx}}$  is the cost of the OAT obtained using the previous approximate algorithm and Bound is the lower bound for the cost of an OAT defined in Eq.(1). Substituting from Eq.(1) the limit L becomes:

$$L = T_{\text{approx}} - \left( \sum_{i=1}^M T(R_i) + \sum_{j=1}^N T(C_j) \right)$$

### Computing the Expense

Suppose that a vertical cut divides an  $M \times N$  array into a left array and a right array. If each of the resulting arrays can be solved such that their respective lower bounds are achieved, then we have:

$$C(\text{left}) = \text{Cost of tree of left part} = \sum T(R'_i) + \sum T(C'_j)$$

$$C(\text{right}) = \text{Cost of tree of right part} = \sum T(R''_i) + \sum T(C''_j)$$

Where  $R', C'$  are the rows and columns of the left part, and  $R'', C''$  are the respective ones for the right part.

The resulting tree obtained by making this vertical cut will have the cost:

$$\text{Cost of tree} = C(\text{left}) + C(\text{right}) + W$$

where  $W$  is the weight of the original array

The expense (Ex) of the cut is defined as the deviation of this cost from the lower bound of the original array. Thus,

$$\text{Ex} = \sum T(R'_i) + \sum T(C'_j) + \sum T(R''_i) + \sum T(C''_j) + W - \sum T(R_i) - \sum T(C_j)$$

Giving

$$\text{Ex}(\text{vertical cut}) = \sum T(R'_i) + \sum T(R''_i) + W - \sum T(R_i) \quad (3)$$

Similarly, if the cut is horizontal, we get

$$\text{Ex}(\text{horizontal cut}) = \sum T(C'_j) + \sum T(C''_j) + W - \sum T(C_j) \quad (4)$$

As an example, for the array with two rows  $R_1, R_2$  and  $N$  columns the horizontal cut has zero expense, using Eq.(4) and noting that in this case  $T(C'_j) = T(C''_j) = 0$  for  $j=1, 2, \dots, N$  and  $\sum T(C_j) = W$ . The optimal tree in this case has the lower bound of Eq.(1) as its cost. This is expected since in the  $2 \times N$  arrays, all the columns have the same optimal tree.

## Good Cuts

Before carrying this concept any further, we need to have some idea about the number of good cuts we are going to have at each step. Although we do not have a theoretical bound on the number of good cuts, some clues can be found. For a start, analysis and experimental results shows that the approximate algorithm produces nearly optimal trees [6,21] which puts a limitation on the number of good cuts; good cuts actually represents possible trees with cost less than that of the tree produced by the approximate algorithm.

In addition, sample runs can give us some indication; through more than 1000 runs for each of 25 different values of  $M$  &  $N$ , the number of good cuts never exceeded half the possible cuts, and it is about one third in the average. Also, through all these runs good cuts were always adjacent to each other; *we never encountered a case where a bad cut is between two good cuts*. Further more, we expect the number of good cuts to decrease at each step. Fig. 4.6 shows the minimum, maximum and average number of good cuts at the first step.

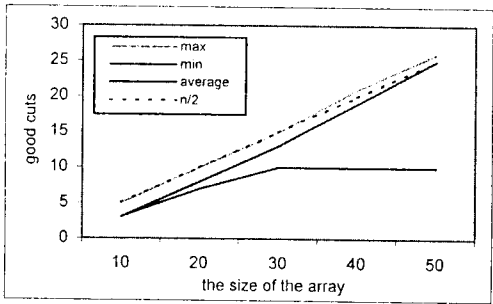


Fig.4.6: the number of good cuts at the first level

## 4.6 Proposed Method

In this section, we describe the proposed algorithm, which uses the concept of goodness, then we follow by an example to demonstrate the process [48].

### Idea

The expense  $Ex$  is computed for all possible  $(M-1) + (N-1)$  cuts is computed. Only good cuts, whose expense is less than the limit  $L$ , are retained as candidates to an optimal solution. Then for a given candidate, the two parts generated by the cut are explored for candidate cuts by looking for a pair of cuts, ones for each part, whose sum of expense is less

than the new limit (L-Ex). Finally, when the size along one of the dimensions reaches the value 4, no further exploration is needed and dynamic programming can be used to obtain the optimal tree for this array.

**Algorithm**

Given an  $M \times N$  array of weights, the steps of the algorithm are as follows.

- 1- The approximate algorithm is applied and the cost of the resulting tree is found  $T_{approx}$ .
- 2- The 1d OAT is found for all rows and columns and the lower bound is calculated Bound using Eq (1).
- 3- The limit L is set as  $T_{approx} - Bound$ .
- 4- If one of the dimensions equal 1, then the optimal tree is found using 1d techniques and the algorithm terminates then.
- 5- If one of the dimensions equals 2, then the first cut is at that dimension. The tree of each resulting weight vector is found using 1d techniques, then the algorithm terminates.
- 6- If one of the dimensions is less than or equal 4, then the optimal tree is found using dynamic programming, and the algorithm terminates.
- 7- For all possible  $(M-1) + (N-1)$  cuts
  - Evaluate the expense of the cut Ex using Eqs (3),(4).
  - If the cut is found to be good ( $Ex < L$ )
    - For each of the left and right resulting subtrees.
    - Go to step4 with the limit  $L = L - Ex$

**Example**

For the purpose of illustration, the procedure is applied to the simple 5x5 array of weights shown in Table 4.3.

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25



$Y_1$	260	62	244	183	6
$Y_2$	230	140	127	49	93
$Y_3$	249	168	116	63	241
$Y_4$	115	194	91	179	126
	28	169	50	126	269
	$X_1$	$X_2$	$X_3$	$X_4$	

Table 4.3 : a 5x5 array of weights

- The cost of the optimal tree for each row and column is obtained and this sum determines the bound of equation (1):

$$\sum_{i=1}^M T(R_i)=8040 \qquad \& \qquad \sum_{j=1}^M T(C_j)=7852$$

$$\text{Bound} = 15892.$$

- The cost of the approximate optimal tree is determined and found to be:  $T_{\text{approx}} = 16236$  and the limit is set to the value:  $L = 16236-15892 = \mathbf{344}$ .
- Using Eq.s (3), (4) we now determine all vertical and horizontal cuts with expense  $< 344$ .

$$\sum_{i=1}^M T(R_i) - W = 8040 - 3578 = 4462$$

$$Ex(X_1) = (0 + 990) + (0 + 818) + (0 + 1114) + (0 + 1180) + (0 + 1135) - 4462 = 775$$

$$Ex(X_2) = (322 + 622) + (370 + 411) + (417 + 599) + (309 + 666) + (197 + 621) - 4462 = 72$$

$$Ex(X_3) = (872 + 189) + (764 + 142) + (817 + 304) + (685 + 305) + (444 + 395) - 4462 = 455$$

$$Ex(X_4) = (1498 + 0) + (1038 + 0) + (1122 + 0) + (1158 + 0) + (746 + 0) - 4462 = 1100$$

$$\sum_{j=1}^M T(C_j) - W = 7852 - 3578 = 4274$$

$$Ex(Y_1) = (0 + 1157) + (0 + 1342) + (0 + 768) + (0 + 820) + (0 + 1458) - 4274 = 1271$$

$$Ex(Y_2) = (490 + 535) + (202 + 893) + (371 + 398) + (232 + 610) + (99 + 1003) - 4274 = 559$$

$$Ex(Y_3) = (1218 + 143) + (572 + 363) + (730 + 141) + (407 + 305) + (439 + 395) - 4274 = 439$$

$$Ex(Y_4) = (1708 + 0) + (1128 + 0) + (1155 + 0) + (877 + 0) + (905 + 0) - 4274 = 1499$$

- There is only one such cut at  $X_2$  with expense  $Ex(X_2) = 72$ . This is the only possible cut that may result in a tree with cost less than  $T_{approx} = 16236$ .
- The left part produced by the cut  $X_2$  is an array with two columns and thus the cut at  $X_1$  will have zero expense (Table 4.4).

260	62
230	140
249	168
115	194
28	169

$X_1$

Table 4.4: 5x2 left subtree after cutting at  $X_2$

- We now have to look for a cut of the 5x3 right array (Table 4.5) whose expense is less than

$$L - Ex(X_2) = 344 - 72 = 272.$$

244	183	6
127	49	93
116	63	241
91	179	126
50	126	269

$Y_3$

Table 4.5: 5x3 right subtree after cutting at  $X_2$

$$\sum_{i=1}^M T(R_i) - W = 2919 - 1963 = 956$$

$$Ex(X_3) = (0 + 189) + (0 + 142) + (0 + 304) + (0 + 305) + (0 + 395) - 956 = 379$$

$$Ex(X_4) = (427 + 0) + (176 + 0) + (179 + 0) + (270 + 0) + (176 + 0) - 956 = 272$$

$$\sum_{j=1}^M T(C_j) - W = 4277 - 1963 = 2314$$

$$Ex(Y_1) = -2314$$

$$Ex(Y_2) = (371 + 398) + (232 + 610) + (99 + 1003) - 2314 = 399$$

$$Ex(Y_3) = (730 + 141) + (407 + 305) + (439 + 395) - 2314 = 103$$

$$Ex(Y_4) = -2314$$

- The only possible cut is  $Y_3$  with expense  $Ex(Y_3) = 103$ .
- The resulting two parts will be cut respectively at  $Y_4$  with zero expense and  $X_3$  with expense  $Ex(X_3)=125$ .

- The optimal tree is now obtained, and it will have a total cost:

$$\begin{aligned} T_{\text{optimal}} &= \text{Bound} + 72 + 103 + 125 \\ &= \text{Bound} + 300 = 16192 \end{aligned}$$

## 4.7 Performance Evaluation

In this section, we first try to derive a complexity function for the algorithm and compare it with dynamic programming.

### 4.7.1 Complexity analysis

The time complexity of the proposed algorithm consists of 3 parts:

- Finding the approximate tree to calculate the limit; this has the complexity of the approximate algorithm.  $T_{\text{approx}} = O(MN \log N)$

Note that, the summations calculated for finding the bound are needed in further steps whether in the proposed algorithm or in the dynamic programming method.

- Evaluating the expense of each cut (horizontal and vertical); this involves finding the 1d

$$\text{OAT for all sub-rows and sub-columns. } T_{\text{exp}} = M \sum_{i=1}^{N-1} i \log i + N \sum_{j=1}^{M-1} j \log j$$

- Finally the cost of trying all subtrees resulting from good cuts.

Thus, the time complexity of the algorithm can be described by the following equation

$$T(N, M) = O(MN \log N) + T'(N, M)$$

where

$$T'(N, M) = M \sum_{i=1}^{N-1} i \log i + N \sum_{j=1}^{M-1} j \log j + \sum_{j=ra}^{rb} [T'(i, M) + T'(N-i, M)] + \sum_{j=ca}^{cb} [T'(N, j) + T'(N, M-j)]$$

ra, rb are the first and last horizontal good cuts respectively; while ca, cb are the corresponding vertical ones.

The reduction in further steps depends on the number of good cuts, since it leads to the number of subtrees that have to be checked. Although, sample runs showed that good cuts are at most half the possible cuts, we do not have a theoretical estimate on the number of good cuts. Thus, we can not reach a closed form time complexity for the algorithm.

However, in contrast dynamic programming can be described by the following recursive equation

$$T_d(N, M) = (M - 1) + (N - 1) + \sum_{i=1}^{N-1} [T_d(i, M) + T_d(N - i, M)] + \sum_{j=1}^{M-1} [T_d(N, j) + T_d(N, M - j)]$$

The expense method will be superior when the number of skipped subtrees (bad cuts) justifies the overhead in complexity ( $T_{\text{approx}} + T_{\text{exp}}$ ). If half of the subtrees are skipped as sample runs showed, then it certainly does.

Also, we would like to mention here that there will be further improvement in the algorithm if we can check the goodness of cuts in the same row (column) in a faster time; i.e. given the 1d OAT for  $i$  nodes, we can find the OAT after adding 1 node (to check the goodness of the next cut) in a time less than  $O((i+1) \log(i+1))$ . In the next chapter, we will introduce an algorithm that does this in  $O(i)$ .

## 4.7.2 Experimental results

The performance of the algorithm was measured using sample runs. Run time is measured for the proposed algorithm and the dynamic programming (as the only existing optimal algorithm for finding the 2d OAT) for different values for  $N, M$ . Runs were made on a Pentium I 100 MHz processor, 16 MB RAM; the program was implemented with Borland C++ programming language, and the random numbers were generated using the language built-in random number generator. Fig. 4.7 presents the curves and shows that the advantage of using the proposed method.

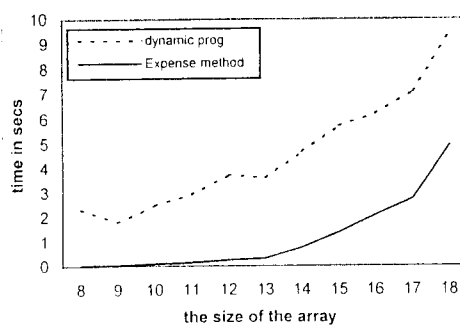


Fig.4.7: Runtime of the proposed algorithm versus dynamic programming

## 4.8 Conclusion

In this chapter, we have introduced a new concept for limiting the search for a two-dimensional OAT. A value we call *the expense of the cut* is used to measure the “goodness” of each possible cut (root). Only cuts with expense less than a given limit  $L$  are considered *good cuts*. The number of good cuts was tested experimentally at the first level of the tree, then more constraints were added for good cuts at next levels. An algorithm is suggested as a possible way to benefit from such concept where cuts are tested for goodness, then at each level only good cuts are tested as possible candidates for an optimal solution. The performance of the algorithm is tested and compared to the dynamic programming ; results shows improvement in run time.

# Chapter 5: Insertion in Optimal Alphabetic Trees

As stated earlier, optimal alphabetic trees are constructed in  $O(n \log n)$  time. Regular insertion and deletion in  $O(\log n)$  time will leave the tree non optimal. In this chapter, we introduce new algorithms for inserting (deleting) a node from an optimal alphabetic tree keeping the new tree optimal after the insertion (deletion). Section 1 is a preface that gives the idea behind the algorithm, and defines the terminology and naming conventions used throughout the algorithm. Section 2 defines properties and relations between the set of nodes we examine at each step of the algorithm. Section 3 discusses different kinds of node processing in the algorithm and how it can be implemented. Then, section 4 discusses the case of insertion at the boundary of the tree (first node or last node). Section 5 extends the algorithm to insertion in the middle under some constraints, then section 6 gives the general case of inserting a node in any arbitrary position in the weight sequence. Section 7 evaluates the performance of the algorithm through complexity analysis and sample runs. Finally, section 8 shows how the algorithm can be used to delete or change the weight of nodes, to split an optimal tree into two optimal subtrees, and many other applications as well.

## 5.1 Preface

At first, this section gives a brief reminder of the rules of choosing the *LMCPs* in Hu-Tucker algorithm as it was explained in chapter 3. Then, it presents the overall idea of the insertion algorithm, followed by defining the terms and naming conventions that we are going to use in the context.

### LMCP

The term *LMCP* refers to the *local minimal compatible pair* of nodes that is chosen at each step in phase 1 of the Hu-Tucker algorithm according to the following rules.

A pair of nodes  $(q_j, q_k)$  constitutes an *LMCP* if:

$$\begin{aligned} q_i &> q_k \quad \text{for all nodes } q_i \text{ compatible with } q_j \\ q_j &\leq q_i \quad \text{for all nodes } q_i \text{ compatible with } q_k \end{aligned}$$

where two nodes are compatible if there are no external nodes separates them.

### 5.1.1 Problem Definition

The algorithm tries to emulate the process of building the Hu-Tucker tree for the new sequence of  $n + 1$  nodes given information about the steps of building the old tree of  $n$  nodes.

Recalling that only phase1 of the Hu-Tucker algorithm takes  $O(n \log n)$  time while phases 2&3 take linear time [3], we are going to concentrate on obtaining the tree resulting from phase1 for the new tree, i.e. the sequence of new *LMCPs*, then phases 2&3 can be applied without losing linearity. Thus, the problem statement of the algorithm can be stated as follows.

*Given the sequence of LMCPs generated for an n nodes weight sequence, it is required to find, in linear time, the corresponding sequence of LMCPs after inserting a new node in the original weight sequence.*

### 5.1.2 Algorithm Outline

The old *LMCP* is chosen over all available nodes in the old tree, now we need to examine a number of more nodes to determine the new *LMCP*. The idea is to use the information already available from the old tree, according to the rules of choosing the *LMCP*, to limit this number as much as possible. We are going to prove that only a constant number of nodes needs to be examined at each step and hence comes the linearity of the algorithm.

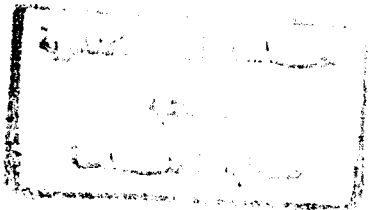
Considering the sequence of old *LMCPs*, each *LMCP* will be examined along with all nodes that became valid candidates now, while they were not in the old tree, to determine the new *LMCP*. Generally, an old node is deleted and a new node is formed. A deleted *LMCP* must be deleted from all subsequent *LMCPs*, a new *LMCP* becomes a valid candidate for next *LMCPs*.

### 5.1.3 Terminology

In what follows, we will call the sequence of *LMCPs* for the old tree, *the old tree list*. Due to the combination rules, the entries of the old tree list are sorted by nature. Similarly, we will call the sequence of *LMCPs* for the new tree, *the new tree list*, and the set of nodes we choose each new *LMCP* from, *the working sequence*.

#### Old LMCPs

A general form of an old *LMCP* is two compatible nodes that constituted the local minimum compatible pair at that time. However, during the course of the algorithm we may face special kinds of *LMCPs* that will be handled differently.



**Blocked LMCPs**

*A blocked LMCP is an old LMCP that both of its nodes are not compatible with all the nodes in the current working sequence.*

When an old LMCP is to be examined and found to have external nodes separating it from the current working sequence, we call it a blocked LMCP. Blocked LMCPs are valid LMCPs for the new tree (collary1), so blocked LMCPs are not added to the working sequence and are moved directly to the new tree list. However, although all blocked lmcp's will be in ascending order, they will not necessarily appear in their right order in the new tree list, which will make the new tree list not sorted.

**Single Nodes**

*A single node is an old LMCP with one of its nodes deleted due to a previously broken LMCP.*

When an old LMCP is examined and found not to be a valid LMCP in the new tree, it must be deleted from any further appearance in the old tree list. This will result in LMCPs with one deleted node and one valid node, in this case the valid node will be added to the working sequence. Note that an LMCP with both of its nodes deleted will be skipped.

**Nodes in the working sequence**

The set of nodes in the working sequence is the nodes in the current LMCP plus the nodes that became valid candidates for the new LMCP due to the insertion of the new node. Due to the processing of previous LMCPs, two kinds of nodes may result.

**New Nodes List**

*The new nodes list is the list of newly generated nodes that did not combine yet.*

At each step, when the working sequence is examined to determine the new LMCP, if the new LMCP is different from the old one, a new node will result. This new node will participate in further combinations and thus should be considered in the working sequence. Since, new nodes will be generated frequently, we will put them successively in a list called the *new nodes list*. Due to the rules of LMCP generation, new nodes will be generated in ascending order and are all compatible with each other. Thus the new nodes list is sorted; actually it can be best represented by the "Queue" data structure, where new nodes are inserted at the tail of the list while the head of the list refers to the smallest node. Also, since the list is sorted, only the first two nodes of the list are added to the working sequence; when



a new node combines it is virtually deleted from the list, i.e. the head is moved to the next node.

#### Leftover Nodes

*A leftover node is a node that existed in the old tree and participated in an old broken LMCP so it is free now and available for further combinations.*

When an old LMCP is added to the working sequence, we do not examine the next one till one of its nodes is chosen in the new LMCP. If only one is chosen, then the old LMCP is broken and the other node remains in the working sequence and becomes a valid candidate for next LMCPs. We will call this node a *leftover* node.

## 5.2 Preliminary Results

So far, the old LMCP, the first two nodes in the new nodes list, the leftover nodes are all members of the working sequence and considered valid candidates for the new LMCP. In this section, we will introduce lemmas and statements that define some properties and relations between those nodes to limit their number (others are defined in the context when needed). Let lemma1 be our starting point.

---

**Lemma 1:** *The LMCPs formed in the old tree before either of the nodes adjacent to the newly inserted node combines will remain valid LMCPs in the new tree.*

#### Proof:

Logically, and due to the rules of the Hu-Tucker algorithm, the inserted node can not combine with any node until one of its neighboring nodes combine, which proves the lemma. Follows a detailed analysis of all possibilities.

Let Q be the node to be inserted between nodes  $q_i, q_{i+1}$ .

Regarding the sequence of LMCPs for the new tree, there are 2 possibilities for the first LMCP that Q will appear in:

- Q combines with either  $q_i$  or  $q_{i+1}$ .
- Q combines with any node other than  $q_i, q_{i+1}$  (internal or external).

#### Case 1: Q combines with either $q_i$ (or $q_{i+1}$ ):

For each LMCP formed before  $q_i$  combined in the old tree.

The nodes forming that *LMCP* were chosen to combine in spite of the existence of  $q_i$ . This means, according to the rules of Hu-Tucker algorithm, that either  $q_i$  is larger than those nodes, or it is not compatible with one (or both) of them.

Now, in the new tree with the new node  $(Q + q_i)$  existing, the same conditions will apply (it will still be larger or not compatible) and it will not be chosen. Thus, the insertion of  $Q$  will not affect the formation of this *LMCP*, and it will remain valid in the new tree.

**Case 2: Q combines with any other node:**

Let  $X$  be the node that combines with  $Q$  in the new tree, where  $X$  could be internal or external.

Suppose that  $X$  is to right (or to the left) of  $Q$ , then  $q_{i+1}$  (or  $q_i$ ), as an external node between  $X$  and  $Q$ , must have combined before this step. Thus, all the *LMCPs* formed before  $q_{i+1}$  combined in the old tree have not been affected by the presence of the node  $Q$  in the new tree. The nodes combined were chosen from exactly the same set of nodes that were there in the old tree and the *LMCPs* will remain valid in the new tree.

Hence, The *LMCPs* generated in the old tree before either  $q_i$  or  $q_{i+1}$  combined will be valid *LMCPs* for the new tree in all cases and the lemma is proved?

---

**Collary1:** *LMCPs in the old tree having external nodes that separate them from both of the nodes adjacent to the newly inserted node at the time they was formed, will remain valid LMCPs in the new tree.*

**Proof:**

Let  $Q$  be the node to be inserted between nodes  $q_i, q_{i+1}$ .

The nodes forming those *LMCPs* were not compatible with both  $q_i, q_{i+1}$  and consequently not with  $Q$ . Thus, the choice of nodes for those *LMCPs* will not be affected by the presence of the node  $Q$  in the new tree. Only smaller *LMCPs* may exist due to the existence of  $Q$  and precede those *LMCPs* in the sequence of generation?

---

**Collary2:** *All new nodes in the new nodes list are compatible*

**Proof:**

Nodes in the new nodes list represents new *LMCPs* that are formed in the new tree.

For each node, in the new nodes list

This node represents a new *LMCP* that is formed in the new tree. The 2 nodes forming this *LMCP* are either old or new.

If one or both of them is from old nodes, nodes that existed in the old tree, then they must be compatible with the new nodes list; otherwise they would have came from a blocked *LMCP* and did not enter the working sequence at the first place (collary1).

On the other hand, a new node is an internal node that must contain at least one old node inside it (only one node is inserted), so the same argument applies and the resulting node must be compatible with all the new nodes list?

---

**Statment1:** *The leftover node is compatible with all nodes in the new nodes list*

**Proof:**

A leftover node is generated when an old *LMCP* is broken and one of its nodes, say *v*, combines to form a new node. The resulting new node is compatible with the rest of the new nodes list as just proved (collary2), then so is *v*. -----(1)

Since *v* combined with our leftover node in the old tree, then *v* must be compatible with the leftover node. -----(2)

From (1),(2):

The leftover node must be compatible with all nodes in the new nodes list?

---

**Statment2:** *Either of the nodes forming the old LMCP can not combine with a new node if a left over node exists.*

**Proof:**

Let the 2 nodes forming the old *LMCP* be called *v,w*, and let them be to the left of the new nodes list. We carry the proof for one of them say *v*, and the same argument can hold for the other

Let us call the left over node *x*, *x* could be an external or internal node,

**Case 1: *x* is an external node**

*v*      *x*      New nodes list

It was proven that *x* is compatible with all the new nodes list. Either *v* is on the same side of *x* as the new nodes (to the right) or *v* is on the other side of *x*. In the latter case, *x* stands as an external node between *v* and the new nodes list, thus they are incompatible and the proof is done.

If *v* is in the same side of *x* as the new nodes:

*x*      *v*      New nodes list

Since *x* is compatible with the new nodes, *v* must be an internal node.

In the old tree, when  $x$  was chosen with a node ( $q$ ) as an earlier  $LMCP$   $v$  was not chosen.  
 Either  $x+q < q+v$ ,  $q$  is incompatible with  $v$ , or  $v$  did not exist at the time  $x, q$  was formed

(1)  $x+q < q+v \Rightarrow x < v$

In this case,  $x$  will be favored on  $v$  in the choice for an  $LMCP$  with the head node of the new nodes list, and the proof is done.

(2)  $q$  is incompatible with  $v$


This case is not possible,  $q$  was chosen in the new tree and  $x$  was left over; i.e.  $q$  participated in an  $LMCP$  in the new tree. Thus,  $q$  is compatible with both  $v$  and the new nodes list, so  $q$  must be compatible with  $v$ .

(3)  $v$  did not exist at the time  $x, q$  was chosen

Then both of the nodes forming  $v$  existed and the same argument applies for both of them, one of them must be larger than  $q$  and so is  $v$ .

From all the above it is clear that  *$v$  can not combine with a new node if  $x$  exists*, either  $x < v$  or  $v$  is incompatible with the new nodes ( $x$  separates them).

**Case 2:  $x$  is an internal node**

$v$   New nodes list

$x$  is compatible with the new nodes, either  $v$  is compatible with all of them or not. If  $v$  is incompatible with them it can not combine and the proof is done; in fact if both  $v$  and  $w$  are incompatible then they form a blocked  $LMCP$  that will be copied to the new tree list without being added to the working sequence.

If  $v$  is compatible with  $x$ :

In the old tree, when  $x$  was chosen with a node ( $q$ ) as an  $lmcp$   $v$  was not chosen.

Either  $x+q < q+v$ , or  $v$  was incompatible with  $x$  at the time the node was formed and became compatible now in the new tree.

(1)  $x+q < q+v \Rightarrow x < v$

In this case,  $x$  will be favored on  $v$  in the choice for an  $lmcp$  with the head node of the new nodes list, and the proof is done.

(2) If  $v$  was incompatible with them, then there were external nodes (at least one) between them and they have all combined before reaching this step. Let  $y$  be the closest one to  $x$ . Two results follow from that:

(1)  $y$  was compatible with both  $x$  and  $q$  ( $x$  is internal) at the time  $x+q$  was formed in the old tree and yet was not chosen. Thus,  $y > x$  &  $y > q$  -----(1)

(2) Similarly, at the time y was combined with a node say z in the new tree, x was available and yet was not chosen. Thus,  $x > z$  -----(2)

Now also, either z is between y and x, or between y and v.

(1) z is between y and x:

This case is not possible since it means that z is compatible with x, and the summation  $z+x$  was also a valid candidate when  $z+y$  combined;

i.e  $y+z < x+z \Rightarrow y < x$ , which contradicts with (1).

(2) z is between y and v

- If z was compatible with v, then  $v+z$  was a valid candidate. Thus,

$y+z < v+z \Rightarrow y < v$ , and from (1) it follows that  $v > x$  and the new nodes will favor to combine with x than v, so the proof is done.

- If z was not compatible with v, i.e. there is at least one external node between them. A similar argument could be carried out replacing x by z, the closest external node to z replaces y and y replaces q. the analysis will always branch out to two possibilities:

Either  $v > y \Rightarrow v > x$  (from (1))

Or there is one more external node between v and x; i.e. not compatible.

Thus, we can conclude that v can never combine with a new node when x exists where it is an internal or external node?

---

**Statment3:** *If an old LMCP entry has just one node, then we do not need to look ahead for the nodes (or node) forming the next entry if a left over node exists*

**Proof:**

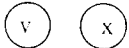
Let the node of the current LMCP be named v, and let it be to the left of the new nodes list. Let the nodes from the next one be called v' and w'. The analysis will be carried out for v' only; if w' exists the same argument holds for it too. Let us call the left over node x. What we want to prove here that v' can not combine before v or x does.

In the old tree, v combined with a node (say w) that was not broken (not formed) in the new tree. Since the deleted node was compatible with v, then we have 2 possibilities: Either v is compatible with the new nodes list. or the left over node x is an external node that stands between them.

If v' is incompatible with v then it can not combine now and the proof is done.

Note that if w' is deleted, then v' is compatible with them, or v or x is the only external nodes in its way to the new nodes.

**Case1 : v, x are internal**



-Since x is internal :  $x < v$  (statement2).  
 -Since v is internal, and v' is compatible with it, the summation w+v' was valid when v+w was formed and yet was not chosen. Thus,  $v < v'$   
 So,  $x < v < v'$   
 v' can not be favored over x,v, and thus can not be chosen, and the proof is done.

**Case 2: x is internal, v is external:**



- If v' is between v.x (or to the left of x) then it is compatible with v,x and case1 applies.
- If v' is not compatible with both x,v then it can not combine and the proof is done.
- If v' is compatible with v but not with x (i.e. to the right of v):

v' was available at the time v was chosen, and yet was not chosen

$$w + v < v + v' \Rightarrow w < v' \text{ -----(1)}$$

w is known to be compatible with x, now in the formation of node x+q in the old tree, either w or one of its components was available and not chosen. This node was compatible with q too since x is internal. Thus,

$$x + q < w + q \Rightarrow x < w \text{ -----(2)}$$

From (1) & (2) we conclude that  $x < v'$

So, x+v is favored over v+v', and with these conditions the summation x+v' is not valid (v is an external node between them). Thus, we do not need to consider node v', and the proof is done

**Case 3: x is external**



- Like case2, we know from the formation of v+w that  $w < v'$  -----(1)
  - If w was deleted as a result from the cancellation of node x+q in the new tree, then
- $$w \geq x + q \Rightarrow w \geq x \Rightarrow x < v'$$
- With the same argument like cases 1 and 2,  $v+v' > v+x$

and either  $v' < v$  or  $v'$  is not compatible with  $x$  ( $v$  is an external node that stands between them). In both cases,  $x+v'$  is not chosen and we do not need to consider  $v'$

-If  $x$  is left over from another *LMCP*, then  $x$  must have resulted from a further cancellation (from statment2, no more than one left over node exists, and  $x$  exists now so the left over node resulted from  $w$  must have combined by now)

$w$  was available at the formation of  $x+q$ , if  $w$  was compatible with  $q$  then

$$w + q > x + q \Rightarrow w > x \Rightarrow x < v'$$

and the proof is straight forward like the previous case.

Now remains the case when  $w$  is not compatible with  $q$ . Recall that both  $w$  and  $q$  was affected by the formation of the new tree. Node  $w$  was deleted and its components combined with other nodes, which means that  $w$ (or all its components in other words) was found to be compatible with the new nodes list. Node  $q$  also combined with a node other than  $x$  which means that it was found compatible with the new nodes list (or combined with a left over node that is compatible with it). This means that, in order for this situation to occur  $w$  must be compatible with  $q$ , which proves our result.

The previous analysis shows that in all cases we do not need to consider  $v'$  if  $x$  exists?

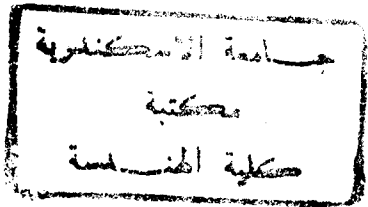
### 5.3 Implementation Issues

In this section, we discuss different kinds of node processing that happens through the algorithm. The linearity of the algorithm depends on what we are going to prove of having constant number of nodes in the working sequence at a time. However, there are some implementation details that concern node processing and needs further explaining. In this section we explore each detail and show how it can be implemented without losing the linearity of the algorithm.

#### Checking Compatibility

Although a sketch of the algorithm steps is there now. One point is still vague; that is how are we going to check whether two nodes are compatible or not. To do that, two markers will be added to each node  $i$  to point to the first external node on its left ( $left[i]$ ) and its right ( $right[i]$ ). For example, an external node  $i$ , will have its markers pointing at nodes  $i-1, i+1$  respectively; while the markers of the root node will have the values  $0, n+1$ .

The check of compatibility for 2 nodes  $i, j$ , where  $i < j$ , will be:



If (  $\text{right}[i] > \text{left}[j]$  ) Then  $i, j$  are compatible

**Result1:** *Checking whether 2 nodes are compatible or not can be done in a constant time.*

### Deleting Nodes

When an old *LMCP* is broken, it must be deleted from all subsequent *LMCP*s it appears in, in the old tree list. This can be implemented by maintaining a link from each internal node to its parent. The deletion can be done by following the links and deleting nodes along the path, till we reach either the root or a previously deleted node. From the stopping criteria, it is clear that we do not pass by a node twice, and since we have only  $n+1$  internal nodes in the old tree the deletion process will not take more than a linear time for the whole tree.

**Result2:** *Deleting broken *LMCP*s from the old tree list will not contribute more than a linear factor to the insertion algorithm.*

### An *LMCP* pair

From what stated we can conclude that an *LMCP* entry must contain a pointer to the *LMCP* that contains its parent node (to facilitate deleting broken entries). In addition it contains the two nodes forming it; each node must hold the following information

- Its weight value, and its identity (or its relative order). These values are essential for choosing nodes for an *LMCP*.
- A pointer to the *LMCP* that formed it, such a pointer will be null for an external node. This pointer is used in finding the level of each node, and also as a check for whether it is internal or external. In addition it is used to adjust the pointer to the *LMCP* parent node during the course of the algorithm.
- Left, and right borders that defines the first external nodes on its left and right respectively; such borders are used in compatibility checks.

This information is readily available during the process of finding the OAT in any known method [3,16,17,18], and can be kept in  $O(1)$  time and space for each node.

Consider as an example the working sequence  $a, b, c, \dots$  of external nodes, if the nodes  $a=1, b=2$  where the first to combine, their *LMCP* entry would look something like  $((a:1, \text{null}, *, b), (b:2, \text{null}, a, c), 3, p)$  indicating that the formed node  $ab$  with value 3 can combine with any node to its left,  $c$  is the first non-compatible node to its right and  $p$  is a



pointer to its parent node. Borders are added to each node incase the *LMCP* is broken, the borders of the new node can be easily deduced from these borders. For reasons of clarity, these additional pieces of information are not added in the context.

### Sorting the New Tree List

Due to blocked *LMCPs*, the new tree list is not sorted. Although this fact will not affect the correctness of the resulting tree, it is important to retain the sorted order for the list to make it valid for further insertion. The solution depends on the fact that blocked *LMCPs*, unsorted entries, were originally entries in the old tree list; blocked *LMCPs* maintain a sorted order between them. Thus, the problem reduces to merging two sorted sublists; blocked *LMCPs* and the rest of entries in the new tree list, which can be done in linear time. Note that the same solution applies for first entries in the old tree list before one of the nodes adjacent to the inserted node combines. since they are valid *LMCPs* too. In fact, they maintain a sorted order with blocked *LMCPs* since they all were valid *LMCPs* in the old tree.

**Result3:** *Sorting the new tree list can be done in linear time.*

### 5.4 Insertion at The Boundary

In this section, we discuss the simpler case of inserting at the boundary of an OAT; i.e. the first or last position, say the last. Given the weight sequence  $q_1, q_2, \dots, q_n$  it is desired to insert the node  $q_{n+1}$  at the right of node  $q_n$ . Fig.5.1 shows the original node sequence and its associated old tree list with nodes  $q_i, q_j$  forming the first *LMCP*.

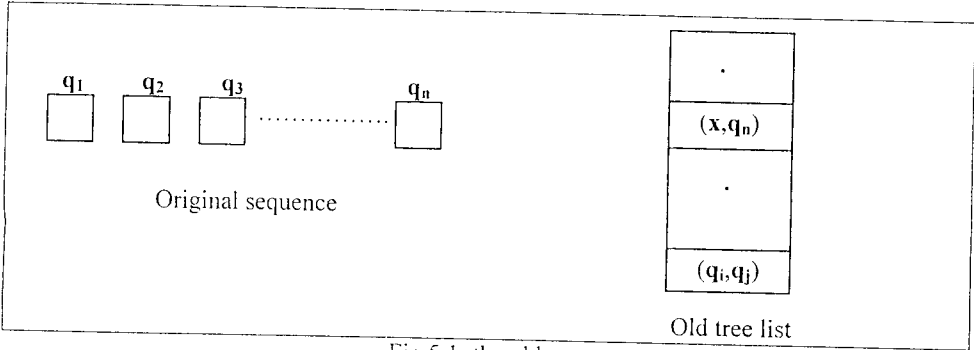


Fig.5.1: the old tree

With lemma1 in our mind, and for simplicity, we will consider the old tree list from the entry where  $q_n$  combined say with node  $x$ . In this case,  $q_n$  may choose to combine with  $q_{n+1}$  instead

of  $x$ , so the first working sequence will contain  $x, q_n, q_{n+1}$  where  $x$  could be internal or external (Fig. 5.2).

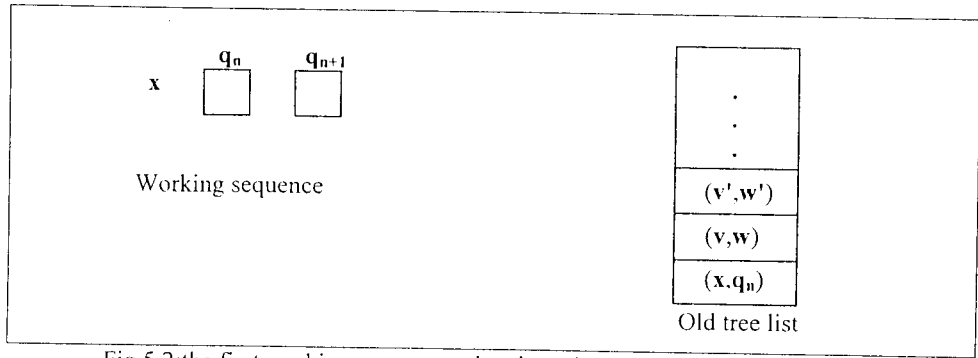


Fig.5.2:the first working sequence when inserting at the boundary of an OAT

If  $x < q_{n+1}$  then  $x, q_n$  will combine as before to form an *LMCP*. The working sequence will now contain the nodes  $v, w$  of the next entry in the old tree list and the node  $q_{n+1}$ , Fig. 5.3a.

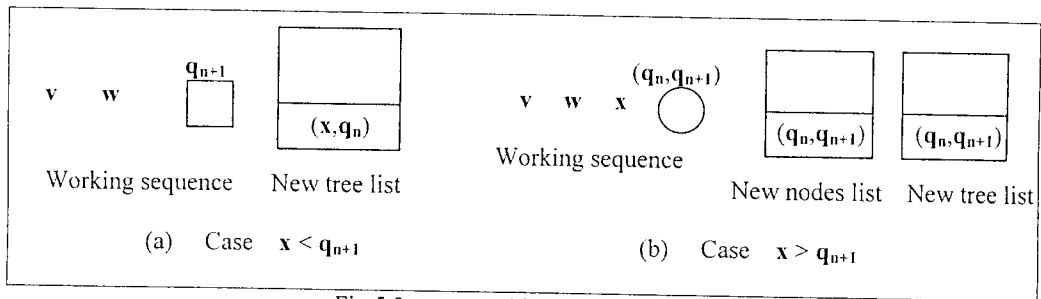


Fig.5.3: next working sequence

On the other hand, if  $x > q_{n+1}$  then node  $q_n$  will combine with node  $q_{n+1}$  to form the new *LMCP* and the old *LMCP*  $(x, q_n)$  will be deleted from all subsequent entries in the old tree list forming single nodes along its path to the root. Node  $x$  will be the first *leftover* node, and the new *LMCP*  $(q_n, q_{n+1})$  will be the first node in *the new nodes list*, Fig. 5.3b.

After that, in case of Fig. 5.3a, the working sequence will contain 3 nodes till  $q_{n+1}$  combines then we reach a case similar to Fig.5.3b. So, and without loss of generality, the following discussion is directed to the more general case of Fig. 5.3b, the results and arguments apply as well to the case of Fig.5.3a when the inserted node eventually combines.

To find the next entry in the new tree list, the working sequence must clearly contain the left-over node  $x$ , the new formed node  $(q_n, q_{n+1})$  and the next *LMCP* from the old tree list,

nodes  $v, w$ . If  $(v+w)$  was the minimum summation then it becomes the new  $LMCP$ , and the next old  $LMCP (v', w')$  will take its place in the working sequence, Fig.5.4a. If the leftover node  $x$  chooses to combine with either  $v$  or  $w$ , then the other becomes the leftover node and another new node  $(x + v \text{ or } w)$  will be added to the new nodes list and becomes a valid candidate for next  $LMCP$ s, Fig.5.4b. If  $x$  chooses to combine with the new node (statement1), then a new node is added to the new nodes list  $(x+q_n+q_{n+1})$ , this new node will replace the two in the working sequence. Note that, neither  $v$  nor  $w$  can combine with a new node since there is a leftover node  $x$  (statement2). Note also that no other old  $LMCP$  will be added till either  $v$  or  $w$  combines, unless  $v$  (or  $w$ ) is a single node then we need to look ahead for the next  $LMCP$  since there is no leftover node (statement3), Fig.5.4c.

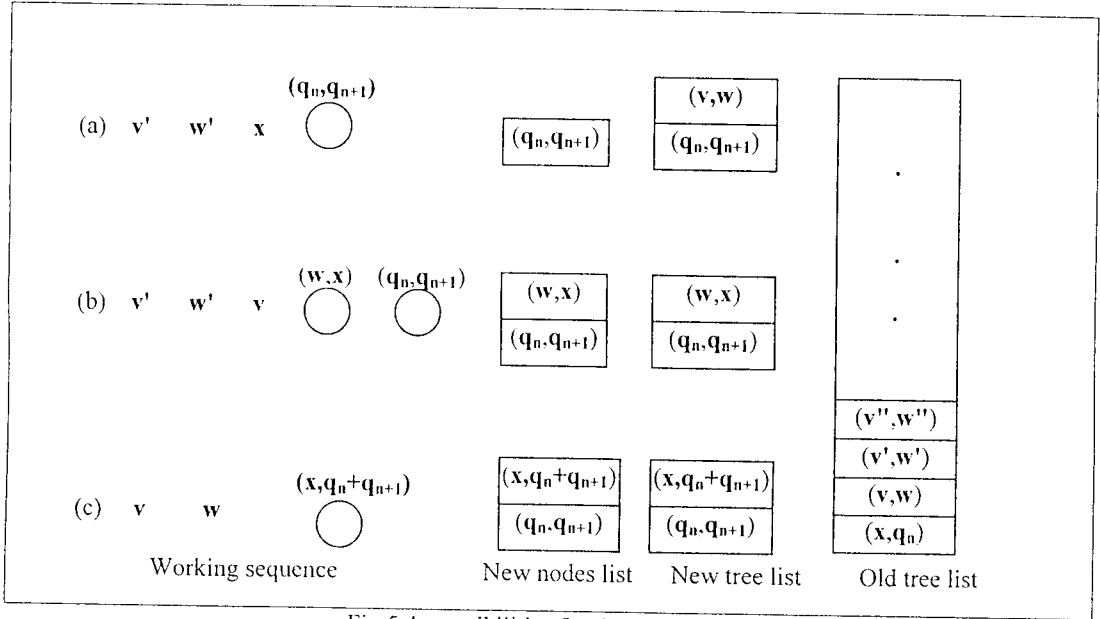


Fig.5.4: possibilities for the new  $LMCP$

The following lemma limits the number of left-over nodes

**Lemma 2:** *When inserting a node at the boundary of an OAT, at most one left-over node exists in the working sequence at any stage.*

**Proof:**

The proof of this lemma depends solely on statment2, since the nodes of an old  $LMCP$  can not combine with nodes from the new nodes list as long as a leftover node exists. Then, the

only way for one of them to be chosen alone, creating another leftover node, is to combine with the existing leftover node, deleting it from the working sequence. The number of leftover nodes can never exceed their number at the beginning of the process, which is one; new leftover nodes only take the place of old ones. Thus, there can only be one leftover node in the working sequence at a time and the proof is done.

However, and since the proof of statement2 was rather mathematical, we will demonstrate it for the case in Fig. 5.3b; we will show that neither of the nodes  $v, w$  can combine with the node  $(q_n, q_{n+1})$  leaving behind another left-over node with node  $x$ .

To show this, we go one step backwards when nodes  $x, q_n$  combined in the old tree list and consider all the different possibilities.

The square node  $q_n$  is the right-most node. Suppose that both nodes  $v, w$  are located between nodes  $x, q_n$  as in Fig. 5.5a., then since  $x$  combined with  $q_n$  in the old tree, both nodes  $v, w$  must be circular nodes also compatible with  $q_n$  and therefore each must have a value larger than the value of  $x$  otherwise they would have combined with  $q_n$  in place of  $x$ . Fig.5.6a shows the position after nodes  $q_n, q_{n+1}$  combine. With  $v, w$  larger than  $x$ , neither can combine before  $x$  does in Fig.5.6a. Node  $x$  in Figs.5.5a,5.6a can either be circular or square.

Fig.5.5b gives the second possibility with  $x$  assumed a square node, if  $x$  was circular case 5.5a applies. Here  $w$  must be circular and compatible with  $q_n$ , and must have value larger than  $x$ . Node  $v$  is prevented from combining with  $(q_n, q_{n+1})$  in Fig. 5.6b, by the external node  $x$ . Node  $v$  can either be circular or square in this case.

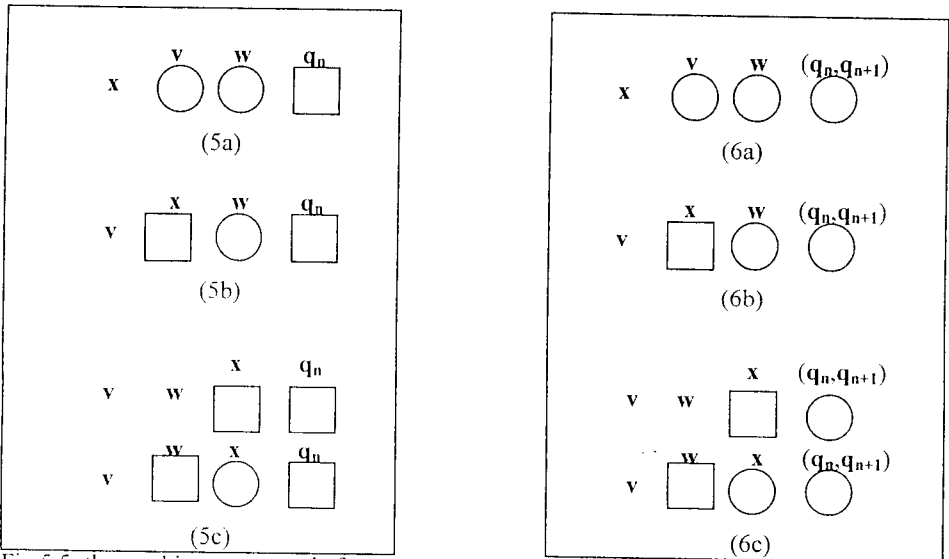


Fig.5.5: the working sequence before  $(q_n + q_{n+1})$  Fig.5.6: the working sequence after  $(q_n + q_{n+1})$

The last case is considered in Fig.5.5c with nodes  $v, w$  to the left side of  $x$ . In the first row of Fig.5.5c, node  $x$  is assumed to be a square node with  $v, w$  either circular or square. Node  $x$  will then prevent both  $v, w$  from combining with node  $(q_n, q_{n+1})$  in the first row of Fig.5.6c.

The second row in Fig.5.5c considers the case when  $x$  is a circular node and  $v$  is either circular or square. If node  $w$  is circular then case (5a) applies, so  $w$  is assumed square. If  $w$  is compatible with  $x$  in the second row of Fig.5.5c, it must also be compatible with  $q_n$  and therefore must have weight larger than that of  $x$ , and hence it cannot combine with  $(q_n, q_{n+1})$  in the second row of Fig.5.6c before  $x$  does. Node  $v$  cannot reach node  $(q_n, q_{n+1})$  because  $w$  is a square node.

Then, at the next step, the most general case is provided after node  $x$  combines with  $w$  (Fig. 5.4b). By collary2, the two nodes  $(q_n, q_{n+1})$  and  $(w, x)$  from the new nodes list are compatible to each other and since they are both circular nodes, we can safely place them next to one another in the working sequence with  $(q_n, q_{n+1})$  occupying the right-most position. An identical argument to the one given before will show that neither  $v'$  nor  $w'$  can combine with either  $(q_n, q_{n+1})$  or  $(w, x)$  leaving another left-over node with  $v$ . The argument for one new node holds for two compatible circular new nodes?

---

The algorithm continues in the same manner till the final *LMCP* (the root) is formed. With lemma 2 assuring us of no more than one left-over node in the working sequence at any stage, every *LMCP* for the new tree can thus be obtained by examining the smallest two nodes in the new nodes list, one possible left-over node and one or two nodes from the old tree list entry (as shown in the working sequence of Fig.5.3b), as one of its two nodes might have been deleted in a previous step. The next entry from the old tree list is needed if the current one is a single node. The number of nodes to examine in the working sequence is still at most five. No more nodes can affect the formation of the *LMCP* except in one case. This case is illustrated in Fig.5.7 and it is the only case in which we may need up to six nodes in the working sequence. If node  $w'$  is external, it is possible that a previously formed *LMCP*  $X$  is compatible with the working sequence and has value less than  $w'$ , and may thus affect the formation of the *LMCP* at this stage ( $X < v'$  but there is no guarantee that  $X < w'$ ). In this case we must therefore add the minimum compatible internal node (the minimal node that satisfies such conditions) to the working sequence.

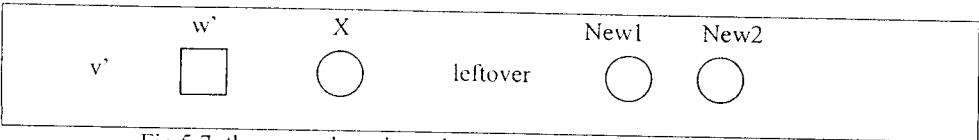


Fig.5.7: the case when six nodes are needed in the working sequence

**Locating node X:**

The node X we are looking for is a previously formed *LMCP* that is yet to appear in an entry of the old tree list. Finding this minimum internal node currently compatible with the working sequence will not contribute more than a linear time factor to the whole insertion process. This can be done, by looking ahead to upcoming entries in the old tree list. When the condition in Fig.5.7 appears (the nearest node w' is a square node) the old tree list is scanned ahead until a circular compatible node is found. If two such nodes are found in an entry we take the smaller of the two.

If the circular node found w'' was to the right of v',w' as in fig 5.8a, then it is the node X we are searching for. On the other hand, if w'' was to the left of w' as in Fig. 5.8b, then w' must be smaller than w'' since node v' chose to combine with w' although compatible with both and no node X smaller than w' can exist (see statement4).

If w'' appeared as a single node and was to the left of w' then the same conclusion applies whether it was internal or external since its companion deleted node was an internal node, Fig.5.8c.

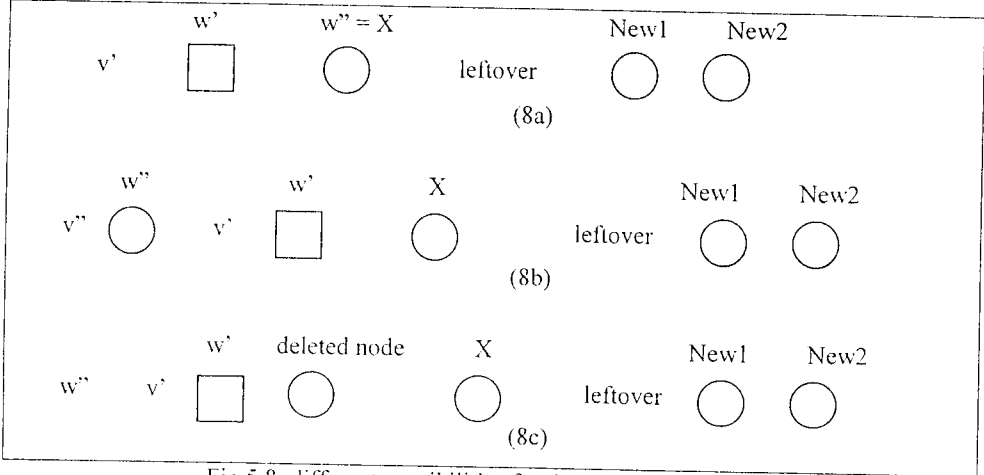


Fig.5.8: different possibilities for the position of node X

- If a node  $X$  is located it is checked against  $w'$ . If it is smaller then it is used in the working sequence and also deleted from the entries in the old tree list in which it appears. If  $w'$  is smaller then node  $X$  is left in its place in the old tree list.
- The next search for a node  $X$  will start from where the previous one stopped. The marker scans the old tree list at most once. Skipped entries are either those where the nearest node is external, or those where both nodes are not compatible with the last entry visited by the marker. No entries whose nearest node is compatible to the last entry visited by the marker is skipped. Once such an entry is found the search is stopped as no node  $X$  will exist (statement4). This is shown in Fig. 5.9 where in the process of locating node  $X$  the marker skips over entry  $(v'',w'')$  and finds the pair of circular nodes  $v''',w'''$  compatible with  $v''$ . Assuming that  $w'''$  is smaller than  $v'''$  we show that the search for  $X$  should stop as a result of the following statement.

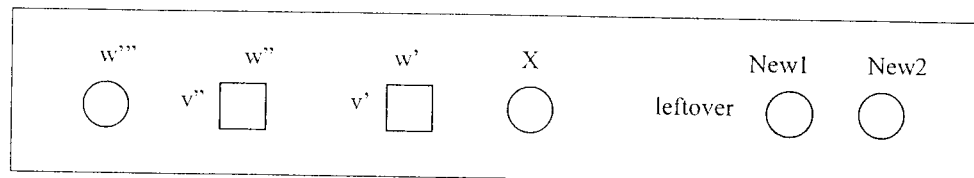


Fig.5.9:  $(v'',w'')$  is skipped and  $w'''$  is considered

---

**Statement4:** *In the working sequence of Fig.5.9, node  $X$  cannot be smaller than node  $w'$ .*

**Proof:**

Node  $w'$  is smaller than node  $w''$  since node  $v'$  chose to combine with  $w'$ . Similarly node  $w''$  is smaller than node  $w'''$ . Therefore  $w' < w'''$ .

After the combinations  $(v',w')$  and  $(v'',w'')$  node  $X$  becomes compatible with  $w'''$  and  $v'''$  and since the last two were the ones to combine it follows that  $w''' < X$ .

Thus,  $w' < X$  ?

---

- The algorithm continues in the same manner till the final *LMCP* (the root) is formed.
- We use all the lemmas and statements we introduced in section 5.2, and here, to find the nodes in the working sequence at each step. First we have the new nodes list, and since it is sorted, we only need the first two nodes in it. Then there are the 2 nodes of the old *LMCP*, and the leftover node if it exists. In case of a single node in the current *LMCP*, it is enough if a

leftover node exists (statement3) otherwise, it takes the place of the leftover node and we move to the next entry with the same number of nodes. If the node to the working sequence in the old *LMCP* pair is external, the minimum compatible internal node is added to the working sequence. Thus, the maximum number of nodes at the working sequence at a time is 6 nodes.

---

**Theorem 1:**

*When inserting a node at the boundary of an optimal alphabetic tree, every new *LMCP* can be determined by examining no more than six nodes.*

**Proof:**

Let us call the old *LMCP*  $v, w$ , the leftover node  $x$ , the minimum compatible internal node  $z$ , the 2 first nodes in the new nodes list  $N_1, N_2$ ; it was proved earlier that those six nodes are needed in the working sequence. Let us assume another node  $U$  is needed in the working sequence. We can safely assume that  $U$  is an old node, since it is clear that we do not need more than 2 node from the new nodes list. The working sequence will be as follows

$w \quad v \quad \textcircled{z} \quad x \quad U \quad \textcircled{N_1} \quad \textcircled{N_2}$

where  $z, N_1, N_2$  are internal nodes, while the rest could be internal or external. There are no restrictions on the relative positions of the nodes as long as they are all compatible.

**Case1:  $U$  is to the left of  $v$ :**

- $U$  was available in the old tree at the time  $(v, w)$  was formed and yet was not chosen. Then,  $v < U$  which means that  $v$  will be chosen over  $U$  in any summation.
- For the summation  $U+v$ , either it is not valid ( $w$  is external) or  $v+w < v+U$  (since it was chosen over it in the old tree).
- If the old *LMCP* is a single node, the deleted node must be compatible with the working sequence. So either  $v$  is deleted and  $w$  remains, or  $w$  is deleted and  $v$  is an internal node. In both cases, the only summation  $U$  can participate in is with the remaining node, say  $w$ ; if  $w$  is internal then  $w < U$  and if it is external then it blocks  $U$  from the rest of the working sequence. At the same time, from the order of *LMCPs*, the deleted node is smaller than  $U$  and larger than  $x$ , and thus  $x < U$ ; so it can not be chosen. Recall that  $x$  must exist in case of a single node otherwise we would have added the next entry (statement3).

Thus  $U$  can not be chosen in the new *LMCP*.

**Case2:  $U$  is to the right of  $v$ :**

- ♦ If  $U$  is external:



- It can not be to the right of x, since x is compatible with  $N_1, N_2$  (statement1).
  - It can not be between x,v (or v,  $N_1$  or  $N_2$  if x did not exist), otherwise (v,w) would have been considered a blocked *LMCP* and was not added to the working sequence in the first place.
  - Thus U can not be external.
  - ♦ If U is internal:
    - By definition, z is the minimum compatible internal node thus  $z < U$  and can replace it in any summation.
    - The summation  $z+U > v+w$  otherwise it would have been chosen instead.
    - Again, if the *LMCP* is a single node we still know that  $z+x < z+U$ .
- Thus, U can not be chosen in the new *LMCP*.

Hence U is not a valid candidate for the new *LMCP* whether it is external or internal, and no matter what position it is in?

### Example

The following example demonstrates most of these points. Fig.5.10 gives an 18 node weight sequence and the *LMCP* list of its OAT showing only the first six entries. It is required to find the OAT after the node Z=4 is inserted at the right end of the given weight sequence.

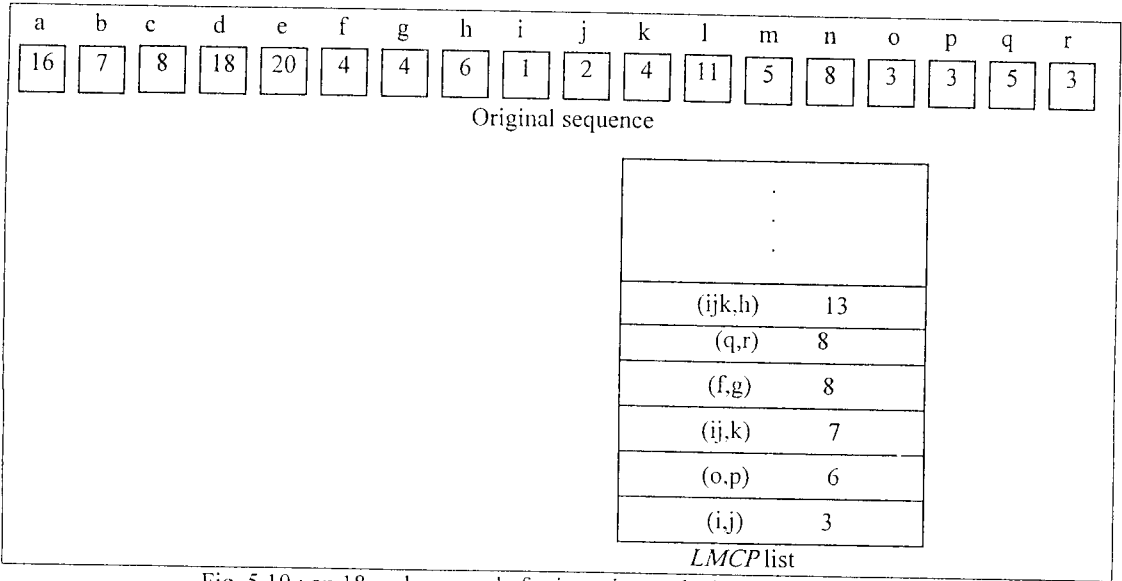


Fig. 5.10 : an 18 node example for insertion at the boundary of an OAT

Our old tree list will contain a list of *LMCPs* beginning with the entry (q,r) in Fig.5.10. This list, together with the weight sequence of nodes corresponding to it, are shown in Fig.5.11. For ease of illustration the nodes in this figure are renamed as the sequence A,B,C,...N, node a appears as node A, node q as M and node r as N.

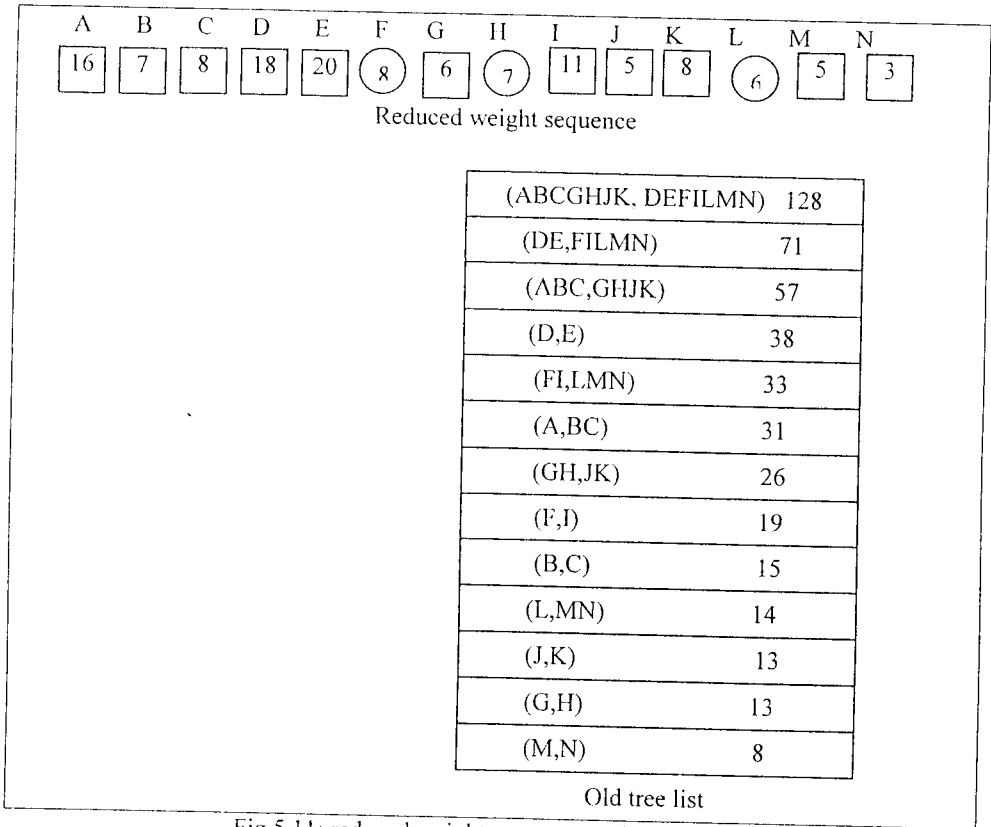


Fig.5.11: reduced weight sequence and old tree list

The new tree list, the new nodes list and the working sequence are shown in Fig.5.12 for the first four steps of the algorithm. The old tree list at the end of the four steps is also shown where an asterisk denotes an entry that was deleted, and an arrow points to the marker position.

To determine the first entry in the new tree list, the working sequence will contain the nodes M,N and the inserted node Z and nodes N,Z combine . Then nodes G,H are found to be blocked so they do not enter the working sequence. At the third step, we encounter nodes J,K where node K is external, so we activate the marker to examine the following entry; (L,\*). L is an internal compatible node, we compare it to K and since it is smaller we bring it to the

working sequence and delete it from that entry leaving (\*,\*). After the node (L,M) is formed the situation remains the same and we have to move the marker again till it reaches (GH,JK) where both nodes are circular and larger than K so we add nothing to the working sequence.

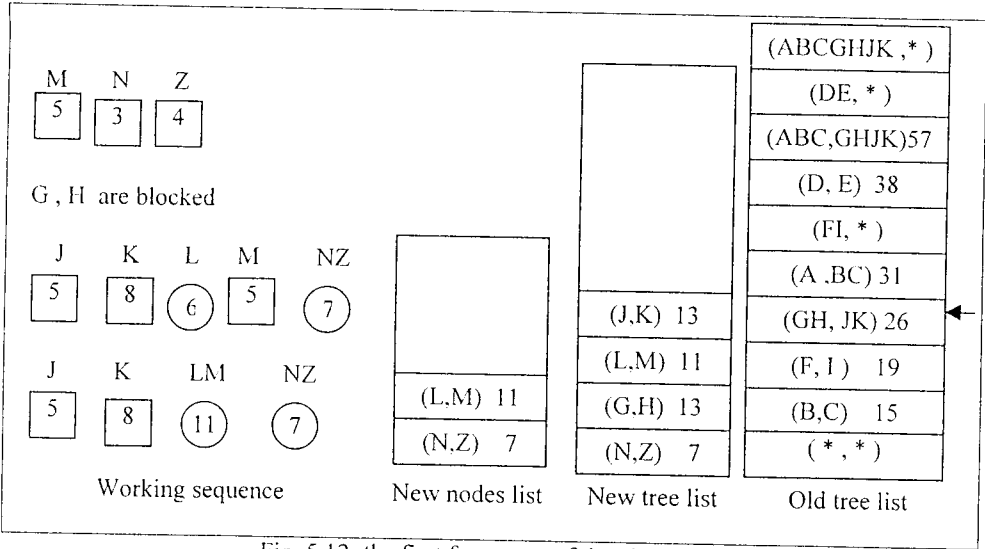


Fig. 5.12: the first four steps of the algorithm

Next we find an entry with two deleted nodes, the following entry contains two blocked nodes B,C so we move them to the new tree list. Then we find the pair (F,I) with node I external so we check the two circular nodes where the marker is standing since they are compatible. We find them both larger than I so we only add F,I to the working sequence. The remaining steps and the final lists are shown in Fig.5.13, and the right arrow shows the last position for the marker.

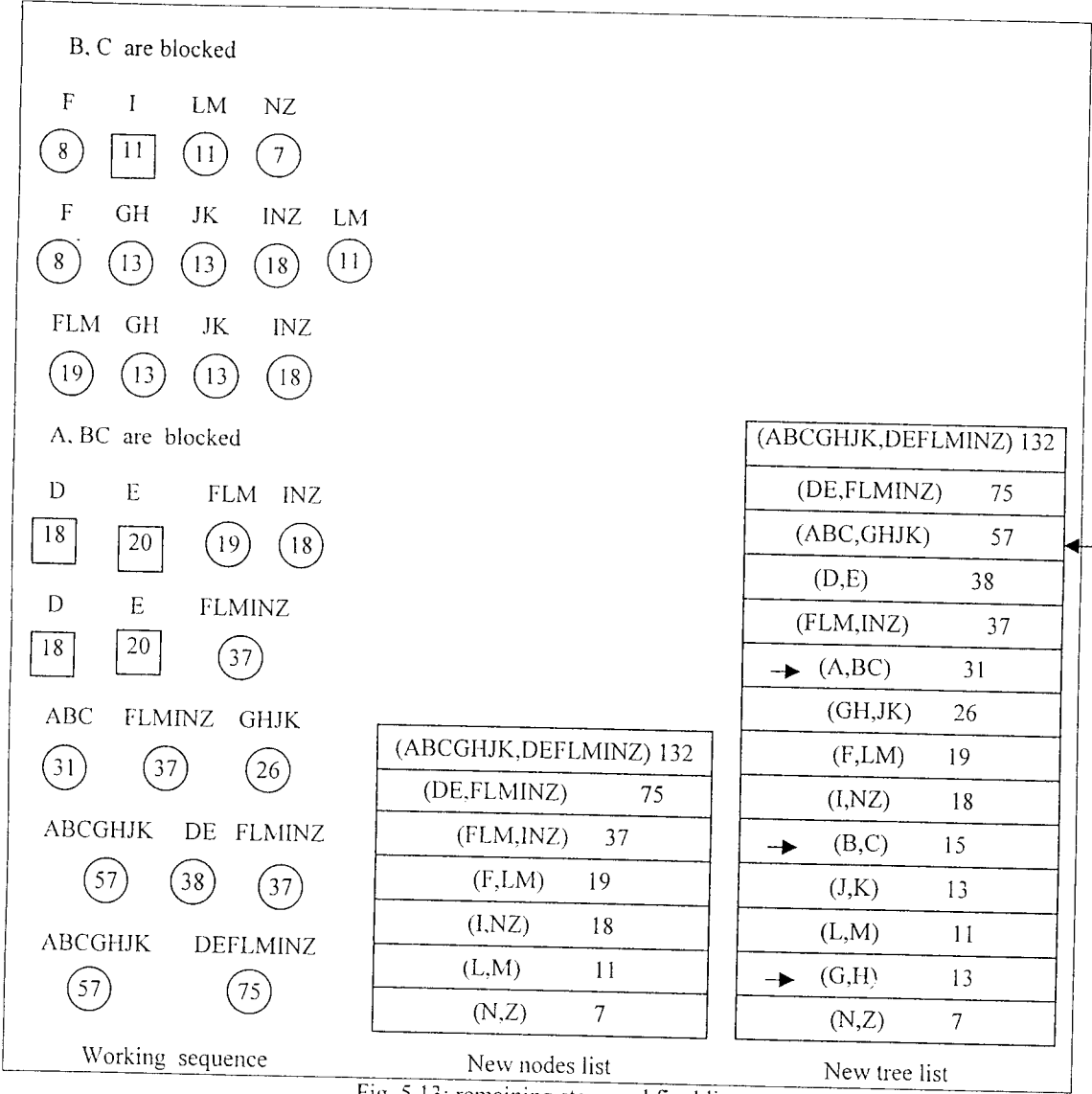


Fig.5.10 below the entry (q,r).The two lists of Fig.5.14 are now merged to get the required final *LMCP*list of the new optimal alphabetic tree.

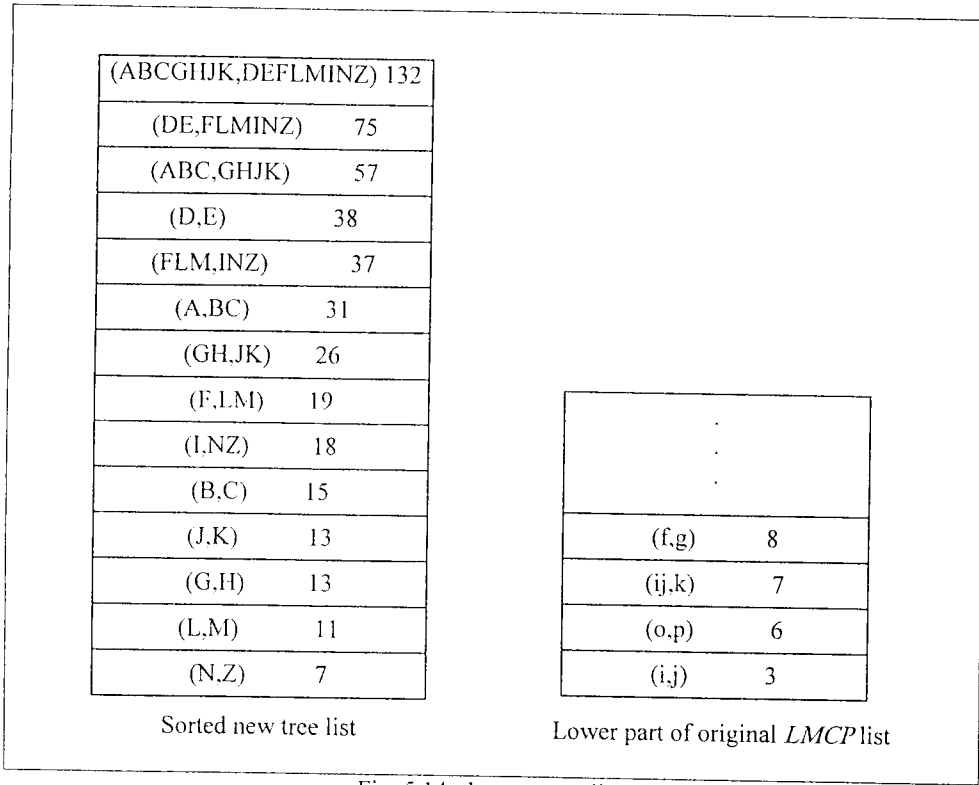


Fig. 5.14: the new tree list

5.5 Constrained Insertion in the Middle

This section is a straightforward generalization of the previous one, when we inserted at the boundary of the tree. Differences will be gradually pointed out, and considered, in the context. Assume a new node *q* is to be inserted between the two nodes *L,R*, where  $q \leq \max(L,R)$ . When either node *L* or node *R* first appear in an entry of the old tree list we proceed as before to form the entries for the new tree list and the new nodes list (lemma1). However, some points must be considered.

The first is if *L,R* appeared together as an *LMCP* in the old tree list. These two nodes are no more compatible and either *L* or *R* must combine with *q* to form the new *LMCP*, otherwise things must be worked out differently and we cannot proceed as before. *q* will

combine with the smaller of L,R if it is less than their maximum, if not the newly formed *LMCP* ( $q + \min(L,R)$ ) is not necessarily smaller than the old one ( $L+R$ ) and we can not be sure that it is the correct one. This section discusses the simpler case when *the newly inserted node q between nodes L, R is not be bigger than both L,R.*

A second point is in the general case, when L (or R) appears first. We cannot determine the new *LMCP* before examining the other node R (or L) as well. Fig.5.15 shows a simple example, where a node Z of weight 2 is inserted between nodes B = 5, C = 4 in the shown sequence of nodes. Although B appeared before C in the old tree list and combined with a node  $A > Z$ , it would be wrong to choose  $B+Z$  as the new *LMCP*, the *LMCP*(C,D) must be brought first to the working sequence where we can choose  $Z + C$  as the new *LMCP*.

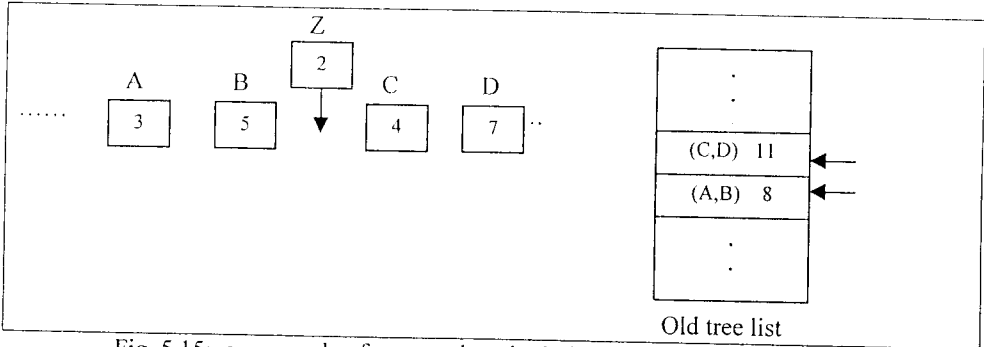


Fig. 5.15: an example of a case where both the *LMCPs* of L,R are needed

Finally, an important last consideration is about bringing old *LMCPs* to the working sequence; the same general rule still applies *but to both the left and right sides of q.* Consider the situation in Fig. 5.16, let  $(v',w')$  be the first *LMCP* that comes after  $(v,w)$  and is from the other side of  $q$ . We know that  $v < w'$  since it was chosen first, although both are compatible with  $w$ , but there is no guarantee that  $w' < w$ . Thus,  $w'$  (exactly like the minimum internal compatible node  $X$ ) might very well be chosen in the new *LMCP* and so must be brought to the working sequence.

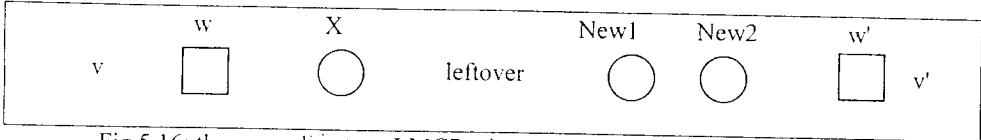


Fig.5.16: the case when two *LMCP* pairs are needed in the working sequence

### Implementation

To be able to identify nodes to the left and right of  $q$ , the old tree list is divided into two lists, the left list and the right list, containing respectively the entries lying to the left and right of  $q$ . If an entry in the old tree list is formed from one node to the left of  $q$  and one node to the right of  $q$ , then the entry is split with the left node entered in the left list and the right node in the right list.

Now, and since we have considered all the aspects of the problem, we can find out the maximum number of nodes needed in the working sequence. In the most general case, two *LMCPs* will be added to the working sequence, one from the left and one from the right of the newly inserted node  $q$ ; i.e. at most 4 nodes. Add to this, the smallest two nodes in the new nodes list, possibly one leftover node, and the minimum compatible internal node, if the nearest node of one of the *LMCPs* is external, which gives a total of 8 nodes in the working sequence. However, if both brought *LMCPs* are single nodes, and there is no leftover node, we may need up to three nodes to the left and three nodes to the right of  $q$  from the old tree list, which gives a total of 9 nodes.

### Theorem 2:

*When inserting a node  $q$  in an optimal alphabetic tree between two nodes  $L,R$  where  $q$  is not bigger than both  $L,R$ , every new *LMCP* can be determined by examining no more than nine nodes.*

#### Proof:

Let us call the old *LMCPs*  $v,w$  &  $v',w'$ , the leftover node  $x$ , the minimum compatible internal node  $z$ , the 2 first nodes in the new nodes list  $N_1, N_2$ ; it was proved earlier that those nodes are needed in the working sequence. Let us assume another node  $U$  is needed in the working sequence. We can safely assume that  $U$  is an old node, since it is clear that we do not need more than two nodes from the new nodes list. The working sequence will be as follows

$$w \quad v \quad \textcircled{z} \quad x \quad U \quad \textcircled{N_1} \quad \textcircled{N_2} \quad v' \quad w'$$

where  $z, N_1, N_2$  are internal nodes, while the rest could be internal or external. There are no restrictions on the relative positions of the nodes as long as they are all compatible.

#### Case1: $U$ is to the left of $v$ :

$$U \quad w \quad v \quad \textcircled{z} \quad x \quad \textcircled{N_1} \quad \textcircled{N_2} \quad v' \quad w'$$

- U was available in the old tree at the time  $(v, w)$  was formed and yet was not chosen. Then,  $v < U$  which means that  $v$  will be chosen over  $U$  in any summation.

- For the summation  $U+v$ , either it is not valid ( $w$  is external and  $U$  to the left of  $w$ ) or  $v+w < v+U$  (since it was chosen over it in the old tree).

- If the old *LMCP* is a single node, then like in Theorem1,  $U$  is larger than the deleted node which is larger than  $x$  hence  $x < U$  and can replace it in any summation. For the summation  $x+U$ , if the single node, say  $w$ , is external then it is not valid, and if  $w$  is internal then  $w < U$  so  $w+x < U+x$ .

Thus  $U$  can not be chosen in the new *LMCP*.

### Case2: U is to the right of $v'$ :

This case is similar to the previous one.

-  $U$  was available in the old tree at the time  $(v', w')$  was formed and yet was not chosen. Then,  $v' < U$  which means that  $v'$  will be chosen over  $U$  in any summation.

- For the summation  $U+v'$ , either it is not valid ( $w'$  is external) or  $v'+w' < v'+U$  (since it was chosen over it in the old tree).

If the right pair is a single node:



- The deleted node must be compatible with the working sequence. So either  $v'$  is deleted and  $w'$  remains, or  $w'$  is deleted and  $v'$  is an internal node. Either the remaining node, say  $w'$ , is external or  $w' < U$ , and so only the summation  $w'+U$  needs to be examined.

- At the same time, and for the same reasons in case1,  $U$  is larger than  $x$  if it exists. So,  $x$  can replace  $U$  in the summation  $U+w'$ .

If  $x$  does not exist, then  $w'$  replaces  $x$  and we get the next right entry.

Thus  $U$  can not be chosen in the new *LMCP*.

### Case3: U is between $v$ , $v'$ :



♦ If  $U$  is external:

- It can not be to the right of  $x$ , since  $x$  is compatible with  $N_1$ ,  $N_2$  (statement1).



- It can not be between  $x, v$  (or  $v, N_1$  or  $N_2$  if  $x$  did not exist), otherwise  $(v, w)$  would have been considered a blocked *LMCP* and was not added to the working sequence in the first place.

- It can not be between  $N_1, N_2$  and  $v', w'$ ; otherwise  $(v', w')$  would have been considered a blocked *LMCP*.

- Thus  $U$  can not be external.

- ♦ If  $U$  is internal:

- By definition,  $z$  is the minimum compatible internal node thus  $z < U$  and can replace it in any summation.

- The summation  $z+U > v+w$  otherwise it would have been chosen instead.

- Again, if the *LMCP* is a single node we still know that  $z+x < z+U$ .

Thus,  $U$  can not be chosen in the new *LMCP*.

Hence  $U$  is not a valid candidate for the new *LMCP* whether it is external or internal, and no matter what position it is in?

---

### Example

Consider the 16 node sequence of Fig.5.17 with the node  $Z=2$  to be inserted between nodes  $g, h$ . The lower part of the *LMCP* list contains all the *LMCP*s formed before either  $g$  or  $h$  combine. The first entry  $(f, g)$  is placed in the working sequence. Since we are going to need two *LMCP*s if both nodes of the first are at the same side of the inserted node  $q$  otherwise one *LMCP* is enough, then the point is to get the first old node from the left and from the right. So, for ease of illustration, the remaining *LMCP*s of the original tree, which are now in the old tree list, are divided into the left and the right lists. The left list will contain all nodes to the left of  $g$  while the right list will contain all nodes to the right of  $h$ .

The first entry in the right list in which node  $h$  appears is brought to the working sequence which now contains the nodes  $Z, f, g, h$ . Entries in the right list forming before node  $h$ , namely  $(j, kl)$ ,  $(o, p)$  and  $(i, m)$  are blocked *LMCP*s and will remain as valid entries in the new tree list; so no need to divide them into right and left and we just copy them to the new tree list.



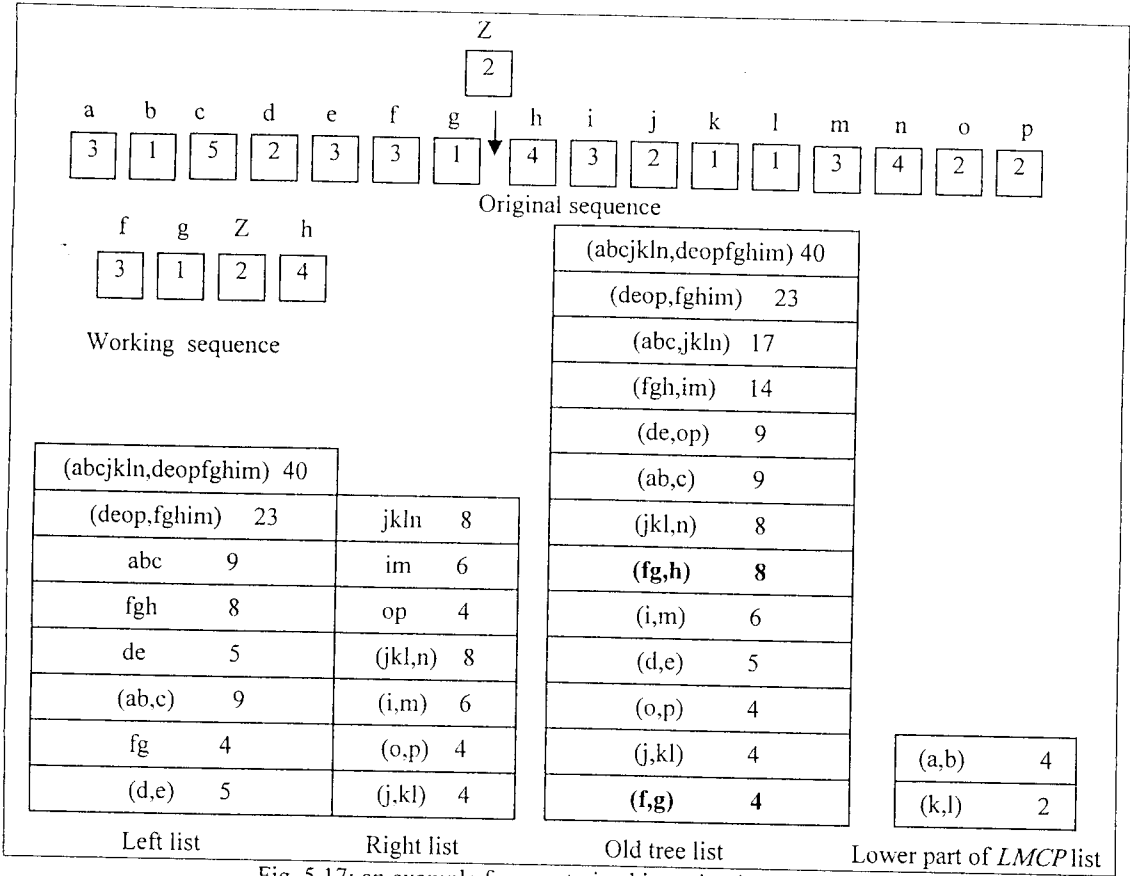


Fig. 5.17: an example for constrained insertion in the middle

The different working sequences and the final new nodes and new tree lists are shown in Fig.5.18. Steps proceed in the same manner, just note that when the pair (ab,c) was reached, and since the nearest node c is a square one, we searched for the minimum circular compatible node which was de. However, since  $de > c$  it was not brought to the working sequence.

The new tree list contains all the entries in the new nodes list together with entries from both the left and right lists and is therefore not necessarily sorted. These entries from the left and right lists can be easily identified and marked with an (L) or an (R).The list of entries marked with (L) ,the list of entries marked with (R) and the new nodes list are merged together to produce a sorted new tree list. This new tree list is next merged with the lower part of the LMCPlist to find the required list of LMCPs for the new weight sequence.

## 5.6 Unconstrained Insertion in the Middle

Here, we generalize the algorithm to include the case we avoided in the previous section. When the node  $q$  is to be inserted between 2 nodes  $l,r$  and  $q > \max(l,r)$ . If  $l$  combined with  $r$  in the old tree, then this  $LMCP$  is no longer valid and at the same time there is no guarantee that  $(q + \min(l,r))$  is a valid  $LMCP$  in the new tree. The same situation could happen if  $l,r$  combined with different nodes and end up combining with each other as internal nodes  $L,R$ . In both cases, we have to examine more nodes before we can decide the correct new  $LMCP$ . Thus, we take the 3 nodes  $q,L,R$  to the next entry in the old tree list.

The inserted node  $q$  stands as a barrier between left and right parts of the tree. The nodes on each side combine separately as in the case of insertion at the boundary, and have their own leftover node and new nodes list, where  $L$  is the first leftover node from the left, and  $R$  is the first leftover node from the right. However, forming the next new  $LMCP$  depends equally likely on nodes from either side of  $q$ . Consider the example in Fig. 5.19, where the node  $q$  is inserted between the two nodes  $L(5)$  and  $R(4)$  and three entries from the old tree list are shown. The  $LMCP(L,R)$  is broken and nodes  $L,R$  become the first leftover nodes. The next entry in the old tree list  $(6,8)$  is to the left of the inserted node, and the one following is to the right  $(8,7)$ .

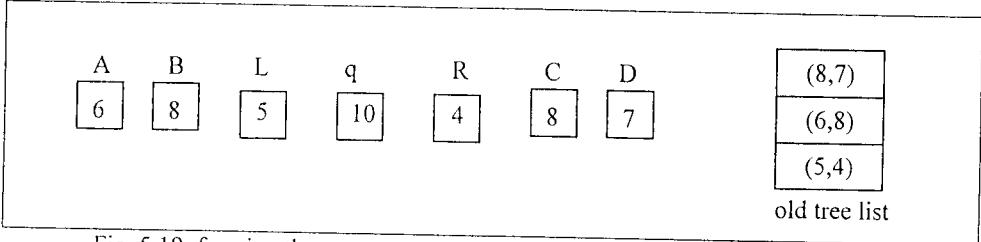


Fig. 5.19: forming the next new  $LMCP$  depends on nodes from both sides of  $q$

Suppose  $q=10$ , adding only  $(6,8)$  to the working sequence produces the new  $LMCP(B,L)=13$  which is wrong. The correct  $LMCP$  is obtained by also examining the next entry  $(8,7)$  from the old tree list giving the combination  $(R,C)=12$ .

Thus, we must examine the nodes of both sides before deciding the new  $LMCP$ . Our working sequence will contain 6 nodes from each side (as in insertion at the boundary) plus the inserted node, giving a total of 13 nodes in the working sequence.

Eventually, when  $q$  combines we return to the case of the previous section (constrained insertion in the middle). So, we merge the two new nodes lists (they are both

compatible now) and we do not need to bring more nodes to the working sequence until we have only one leftover node (see statement5). We examine pairs of *LMCPs* one from the left and one from the right, with at most 9 nodes in the working sequence at any stage.

**Statement5:** *If the inserted node is larger than both its neighboring nodes, then when the inserted node combine we do not need to add any more nodes to the working sequence till we have no more than one leftover node.*

**Proof:**

Let *q* be the inserted node, and let *L,R* be the left and right leftover nodes respectively. (Note that *q* is external)

Let the pair we were examining when *q* combined be *v,w*, and the pair we are questioning the need to it be *v'* and *w'*.

Again we will analyze for *v'* and the same goes for *w'*. We will consider the case when *v,w* are to the left and exactly the same argument can be used when they are from the right. We will also consider the case when *v* and *w* are in opposite sides.

We have now:

$$L \quad NewL \quad q \quad NewR \quad R$$

**If *q* combined with *L* or *R*:**

Then we have at most one leftover node and we should add the next *LMCP* now.  
 -Note that, *q* can not combine with *v* or *w*, since a previous result (statement1) showed us that either the leftover node in their side is smaller and should be favored, or is external and they can not combine.

**If *q* combined with a node from the new list:**

As we said earlier, we will now merge the new lists and it really does not matter from which one was the node combined with *x*. We now have 2 leftover nodes *L,R*, the new list and *v,w*. We will consider the case when *v,w* are to the left and exactly the same argument can be used when they are from the right. We will also consider the case when *v* and *w* are in opposite sides; i.e. we have 2 cases:

**(a)  $w \leq v \leq L \leq New \leq R \leq v'$ :**

-From the old tree, we know that  $L \leq v'$  and so it can replace it in all summations.

-For the summation  $L+v'$ , if  $R$  is external then it is not allowed; if  $R$  is internal then  $R \leq v'$  too and thus  $L+R \leq L+v' \& R+v'$  and we do not need to examine  $v'$ .

**(b)  $v \leq L$  New  $R \leq w \leq v'$ :**

We will analyze when  $v'$  at right, and a similar argument can be used when it is at left.

- $v \leq v'$  (from old tree) and it will be favored on all summations.

-The summation  $v+v'$  is only valid if all other nodes are internal, in this case naturally  $w+v \leq v+v'$  since it was chosen as an *LMCP* in the old tree. So, we do not need to examine  $v'$ .

Thus, we do not need to examine more *LMCPs* till we have at most one leftover node?

---

### **Theorem 3:**

*When inserting a node  $q$  in an optimal alphabetic tree between two nodes  $L, R$  every new *LMCP* can be determined by examining no more than thirteen nodes.*

**Proof:**

When the inserted node  $q$  combines no more nodes are added to the working sequence till we have only one leftover node (statement 5) then we are in the same situation as in theorem 2 where it was proven that no more than nine nodes are needed.

Thus, we are concerned on proving the theorem in the case before  $q$  combines, where we have two new nodes lists and two leftover nodes.

In addition to the inserted node  $q$ , we need at both of its sides an *LMCP* pair, two nodes from the corresponding new nodes list, a leftover node, and the minimum compatible internal node at this side; this will sum up to 13 nodes in our working sequence.

To prove that no more nodes are needed, we assume the existence of another arbitrary node  $U$  in the working sequence, and show that it can never be chosen in the new *LMCP* as there are at least two nodes that are smaller than it. We can safely assume that  $U$  is an old node, since it is clear that we do not need more than 2 node from the new nodes list.

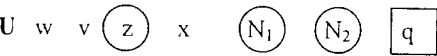
$U$  could be to the left or to the right of  $q$ , the analysis will be carried to one side (the left of  $q$ ) and exactly similar analysis to the other side (the right) will lead to the same result.

Let us call the old *LMCP*  $v,w$ , the leftover node  $x$ , the minimum compatible internal node  $z$ , the 2 first nodes in the new nodes list  $N_1,N_2$ . As explained, all those nodes are needed in the working sequence, thus it will be as follows



where  $z, N_1,N_2$  are internal nodes,  $q$  is external of course, while the rest could be internal or external. There are no restrictions on the relative positions of the nodes as long as they are all compatible. Now,  $U$  could be in any position, internal or external.

**Case1: U is to the left of v:**



- $U$  was available in the old tree at the time  $(v,w)$  was formed and yet was not chosen. Then,  $v < U$  which means that  $v$  will be chosen over  $U$  in any summation.
  - For the summation  $U+v$ , either it is not valid ( $w$  is external) or  $v+w < v+U$  (since it was chosen over it in the old tree).
  - If the old *LMCP* is a single node, the deleted node must be compatible with the working sequence. So either  $v$  is deleted and  $w$  remains, or  $w$  is deleted and  $v$  is an internal node. In both cases, the only summation  $U$  can participate in is with the remaining node, say  $w$ ; if  $w$  is internal then  $w < U$  and if it is external then it blocks  $U$  from the rest of the working sequence. At the same time, from the order of *LMCPs*, the deleted node is smaller than  $U$  (was chosen over it in the old tree) and larger than  $x$  ( $x$  was chosen in earlier pair), and thus  $x < U$ ; so it can not be chosen. Recall that  $x$  must exist in case of a single node otherwise we would have added the next entry (appendix-statement2).
- Thus  $U$  can not be chosen in the new *LMCP*.

**Case2: U is to the right of v:**



- ♦ If  $U$  is external:
- It can not be to the right of  $x$ , since  $x$  is compatible with  $N_1, N_2$  (statement1).

- It can not be between  $x, v$  (or  $v, N_1$  or  $N_2$  if  $x$  did not exist), otherwise  $(v, w)$  would have been considered a blocked *LMCP* and was not added to the working sequence in the first place.

- Thus  $U$  can not be external.

♦ If  $U$  is internal:

- By definition,  $z$  is the minimum compatible internal node thus  $z < U$  and can replace it in any summation.

- The summation  $z+U > v+w$  otherwise it would have been chosen instead.

- Again, if the *LMCP* is a single node we still know that  $z+x < z+U$ .

Thus,  $U$  can not be chosen in the new *LMCP*.

Thus  $U$  is not a valid candidate for the new *LMCP* whether it is external or internal, and no matter what position it is in. Hence only 13 nodes are needed to find the new *LMCP*.

---

### Example

Let us consider the weight sequence given in Fig.5.20, and insert a node  $Z=18$  between nodes  $E, F$ . *LMCPs* before  $E$  or  $F$  appears are copied to the new tree, then the *LMCP* containing  $E$  and  $F$  is broken with all its path deleted, and the two nodes are brought to the working sequence. The old tree list is divided to right and left lists to get a pair from each list.

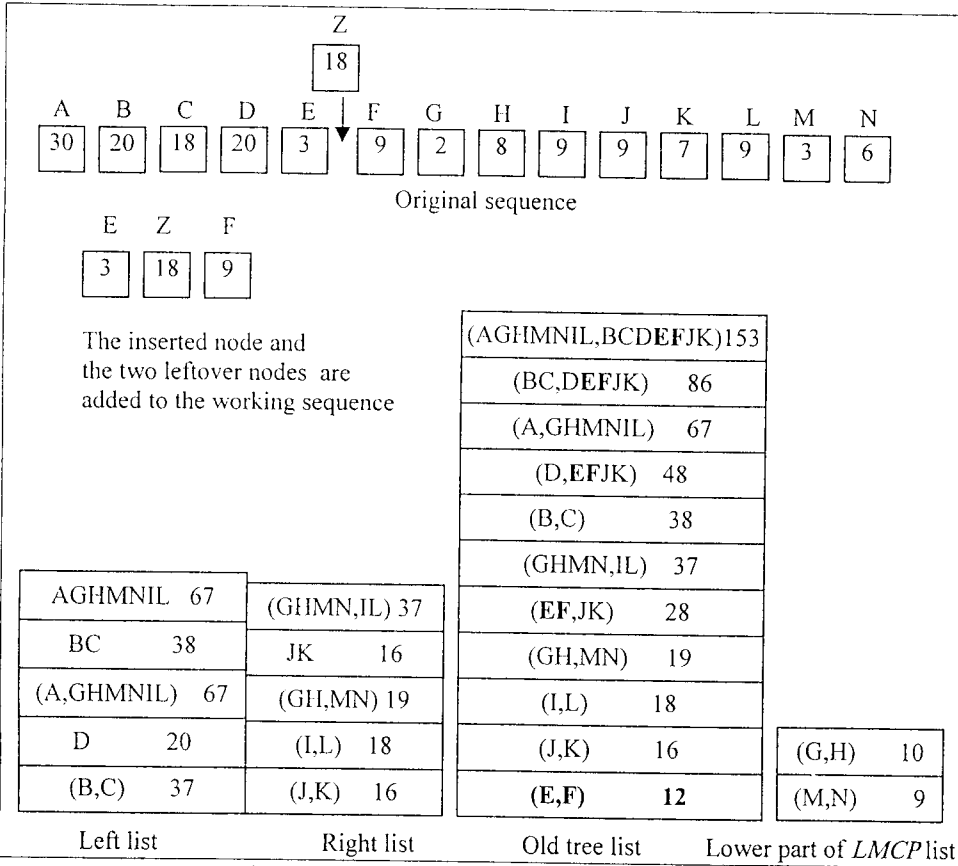


Fig.5.20: an example for inserting a larger node

Fig. 5.21 shows the working sequences, the two new nodes lists, the old tree list and the new tree list till Z combines. The first entries from right and left are blocked so they are copied to the new tree list with markers to determine their side (like the previous section). Then the following ones are added to the working sequence (I,L) and (D). (F,I) is found to be the new *LMCP* and is added to the right new nodes list since it is to the right of Z. Then the next entry in the right old tree list (GH,MN) is brought to the working sequence and (L,MN) is chosen and added to the right new nodes list. Finally, JK is added to the working sequence and Z combines with E to form the new *LMCP*.



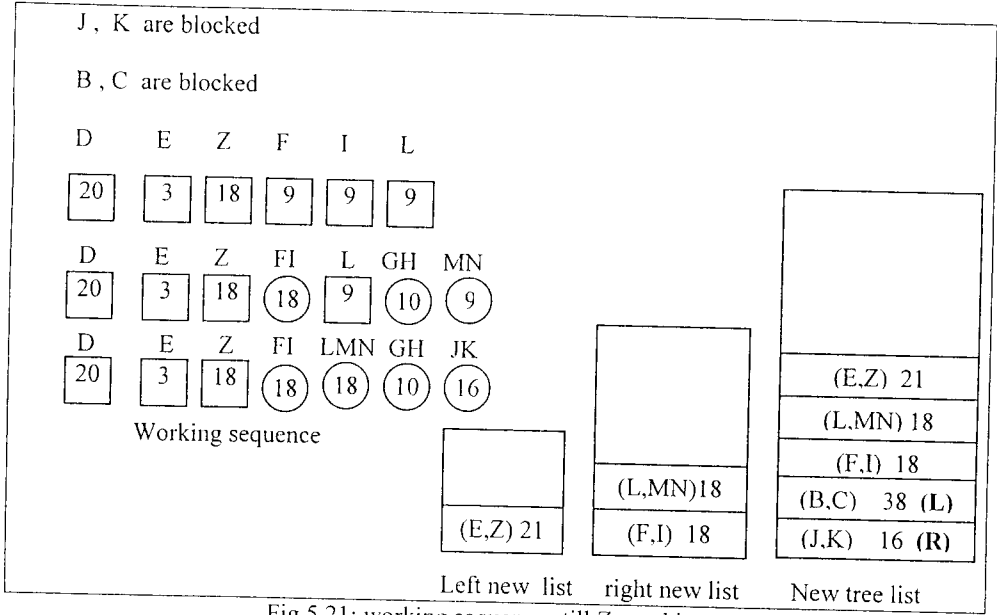


Fig.5.21: working sequences till Z combines

When Z combines with E, both new lists are merged (so easily since we know their order), and we have no leftover nodes, so we proceed taking the first left and right *LMCPs*. Fig 5.22 shows the rest of the process and the final new tree list.

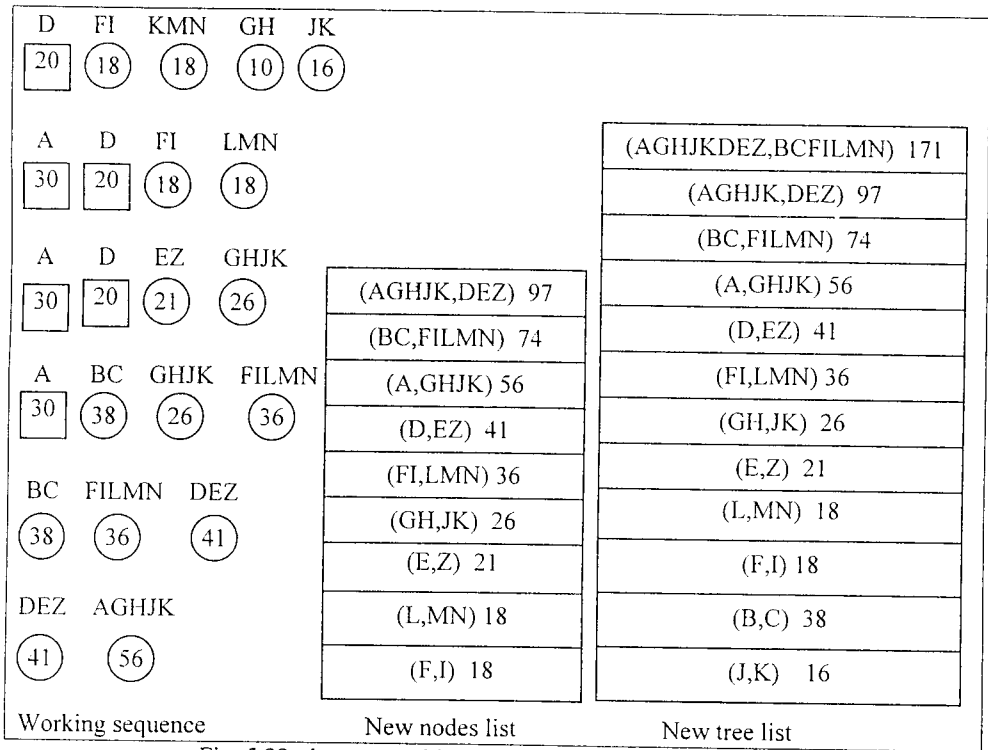
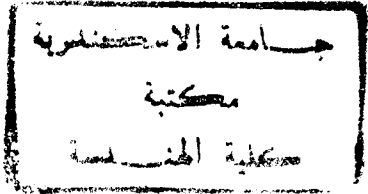


Fig. 5.22: the rest working sequences and the new tree list

### 5.7 Performance Evaluation

In this section we evaluate the performance of the algorithm. First we discuss the time complexity, then the space complexity, and we end the section with the results of sample runs.

#### Time Complexity

All the previous cases in sections 5.4 to 5.6, it was proven that only a constant number of nodes is needed to find each *LMCP* in the new tree, and since we have *n LMCP* in the new

sorted new tree list will also take linear time (result3). Finally performing phases 2,3 of H-Tucker algorithm to obtain the optimal alphabetic tree will take a linear time too.

Thus, *the insertion algorithm has a linear time complexity.*

### Space Complexity

The algorithm needs to maintain 3 lists; the old tree list, the new tree list, and the new nodes list. Each of these lists has a maximum of  $n$  *LMCPs* (or nodes for the new nodes list) in it. The working sequence has a constant number of nodes including the leftover node. Thus, the algorithm stores a linear number of nodes during the whole process.

Also, the information we need to store about each node is constant. Namely, its value, its identity, the identities of the first external nodes from its right and its left to check its compatibility with other nodes, a link to its parent *LMCP* to delete its path if broken, and finally a link to its originating *LMCP* if internal (the *LMCP* that contains its two children).

Thus, *the insertion algorithm has a linear space complexity.*

**Result 4 :** *A new node  $q$ , with any arbitrary weight, can be inserted between two nodes  $l,r$  in an optimal alphabetic tree in linear time with linear space requirements.*

### Sample Runs

Fig.5.23 shows the results of sample runs that measured the time of the insertion algorithm for different numbers of nodes ( $n$ ). Runs were made using randomly generated uniform weight sequences. The minimum, average, and maximum time were plotted for each value of  $n$ . Since we are mainly interested in showing the linearity of the worst case time, each time a node was inserted between the two nodes forming the first *LMCP* in the old tree list. This way, no *LMCPs* will be copied at the beginning and most of them will be examined to avoid the best case time of the algorithm.

Runs were made on a Pentium I 100 MHz processor, 16 MB RAM; the program was implemented with Borland C++ programming language, and the random sequences were generated using the language built-in random number generator. The curves in the figure show the linearity of all of them, as expected from the analysis.

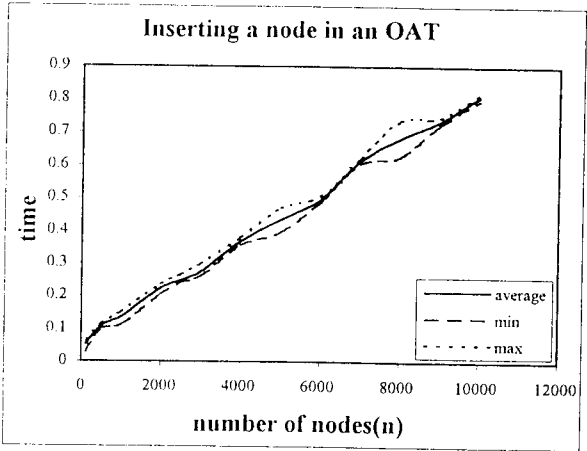


Fig. 5.23: time versus number of nodes for the insertion algorithm

### 5.8 Applications

In this section , we show how the insertion algorithm can be used to delete a node from the weight sequence or change its weight and we show also other merits of the algorithm.

#### 5.8.1 Deletion

There are two possible ways, the algorithm can be used to delete a node from an OAT.

##### Method 1

Deleting a node  $q$  between nodes  $l, r$  is accomplished by inserting the negative weight node  $(-q)$  between them. This works fine as long as the two nodes  $(q), (-q)$  are guaranteed to be combined in the final tree thus forming a node of weight zero. The required tree is obtained by removing the node of weight zero and its two children and decrementing by one the level of the node that formed an *LMCP* with the zero node.

It was shown in [31] that for the consecutive four nodes  $a, b, c, d$ , the two middle nodes will combine in the final tree if:

$$a > c \quad , \quad b \leq d \quad \text{and} \quad (b+c) < \max(a,d)$$

Therefore for the sequence of four weights  $l, q, -q, r$  the condition reduces to  $q \leq r$  while for the sequence  $l, -q, q, r$  the condition is  $l > q$ . By noticing that when two consecutive nodes are

equal it does not really matter which one of the two gets deleted, then deletion will still work fine when  $l = q$  and therefore deletion can be done provided that  $q$  is not bigger than both  $l, r$ .

## Method 2

The algorithm can be used to delete a node in linear time as follows.

- The *LMCP* that contains the node is deleted from the old tree list, and the link is followed to delete it from further appearance in the list.
- The other node in that *LMCP* is treated as new node we are trying to insert, and the same insertion rules described in the preceding sections is applied to it.

During the insertion, the old tree list is processed from its beginning, not from the deleted *LMCP*; if the inserted node (the companion of the deleted node) is internal, the list is processed from the *LMCP* following the *LMCP* that formed it.

**Result 5:** *A node  $q$  between nodes  $l, r$  in an optimal alphabetic tree can be deleted in linear-time.*

## 5.8.2 Changing The Weight of a Node

If inserting or deleting a node from an optimal alphabetic tree can be done in linear time, then it is trivial that changing the weight of a node is a linear time operation as well. The node can simply be deleted and reinserted with the new weight. Thus, we have the following result.

**Result 6:** *The weight of a node  $q$  in an optimal alphabetic tree can be changed in linear time.*

However, and since this process will double the time of the algorithm, we introduce another way, similar to method1 for deletion, that works under some constraints.

The weight of a node  $q$  between the two nodes  $l, r$  can be changed to  $q+x$  by inserting a node of weight  $x$  adjacent to it. It was shown in [31] that for the consecutive four nodes  $a, b, c, d$ , the two middle nodes will combine in the final tree if:

$$a > c, \quad b \leq d \quad \text{and} \quad (b+c) < \max(a, d)$$

Thus, for  $x$  to combine with  $q$  in the weight sequence  $l, q, x, r$ :

$$l > x, \quad q \leq r \quad \text{and} \quad (q+x) < \max(l, r)$$

If we are trying to decrement the weight of  $q$ , i.e. if  $x$  is negative, then the condition reduces to  $q$  being not larger than both  $l, r$ .

**Collary3:**

*The weight of a node  $q$  between nodes  $l, r$  can be decremented by a value  $x$  in linear time through the insertion of a negative weight  $(-x)$  adjacent to it, provided  $q$  is not bigger than both  $l, r$*

If  $x$  is positive, then the conditions can be restated as follows

$$\min(l, r) > \min(q, x) \quad \text{and} \quad (q+x) < \max(l, r)$$

Where  $x$  must be inserted such that both  $q, x$  are smaller than the node adjacent to it.

**Collary4:**

*The weight of a node  $q$  between nodes  $l, r$  can be incremented by a value  $x$  in linear time, through the insertion of a node with weight  $(x)$ , provided that the new weight is not bigger than both  $l, r$ , and the minimum of the node  $q$  and the increment value  $x$  is smaller than both  $l, r$*

### 5.8.3 Splitting Optimal Alphabetic Trees

The insertion algorithm can be used to split an optimal alphabetic tree into two optimal trees by simply inserting a node with a weight larger than the sum of weights for each part. A node with such weight will be the last to combine and thus will have the required optimal subtrees one at each side.

**Result 7:** *An optimal alphabetic tree can be split into two optimal subtrees in linear time*

### 5.8.4 Other Merits

In addition to the direct benefit of having a linear time algorithm for updating the weight sequence of an optimal alphabetic tree which make the OAT more suitable for string and retrieving information systems of dynamic nature, there are other useful merits of the algorithm.

For a start, it is known that optimal alphabetic trees can be constructed in linear time for some special weight sequences [31]. This algorithm allows us to do a constant number of linear-time insertions and deletions on those trees thus expanding the number of such trees that can be constructed in linear-time.

In the same manner, any application or algorithm that involves successive construction of 1d OATs after appending the weight sequence each time, a common procedure in 2d OAT algorithms, can use this algorithm to fasten the process. An example is the expense of a cut method introduced in the previous chapter [48], where we needed to calculate the cost of successive subsequences of all the  $M \times N$  1d sequences to find the expense of each cut. This can be done by inserting the next node at the end of the previous weight sequence each time instead of reconstructing the 1d OAT (section 4.8.1). Although this will not decrement the order complexity of the algorithm, we expect it to enhance its performance.

Finally, the idea of the algorithm can be extended and modified to merge two optimal alphabetic trees and deduce the optimal tree for the concatenated weight sequence. Furthermore, the merging algorithm can be used to build the optimal alphabetic tree in a recursive manner; an implementation method that is expected to have a better runtime than other known methods. Note also, that all these results are applicable to Huffman trees, even with less runtime, by releasing the compatibility constraint and simply considering all nodes internal. These ideas will be explained in the following chapters.

## 5.9 Conclusion

In this chapter we introduced a novel technique to facilitate the use of optimal alphabetic trees in applications of dynamic nature. We proposed an algorithm to insert a node in an optimal alphabetic tree, without losing its optimality, in linear time and space complexity. We also showed how the algorithm can be used to delete or change the weight of an existing node in the tree and to split the optimal tree into two optimal subtrees. Finally, we suggested many other different uses of the algorithm

both trees before those two boundary nodes appear remain valid *LMCPs* for the new tree (lemma1).

After that we start examining old *LMCPs* from each tree, along with other nodes in the working sequence, to determine the new *LMCP* at each step. When an old *LMCP* is broken, it is deleted from all subsequent appearance in the old tree list (result2). If the new *LMCP* is not an old one, it is added to the *new nodes list*.

When the new *LMCP* list is found, these entries that were originally valid *LMCPs* for the old trees ,although sorted amongst themselves ,will cause the new list of *LMCPs* to be unsorted. A final merging phase is required to merge 3 lists,the list of valid *LMCPs* from each old tree, and the list of newly formed *LMCPs* (result3).

### 6.1.3 Nodes in the Merging Working Sequence

Fig. 6.1 shows two symbolic sequences of *LMCPs* to merge; where the order of nodes in the pair reflects their order in the weight sequence. For simplicity, our old tree lists start from the pair where the corresponding boundary node participates as earlier pairs are just copied with no change. We trace the first few entries to determine what, and how many, nodes are needed in the working sequence.

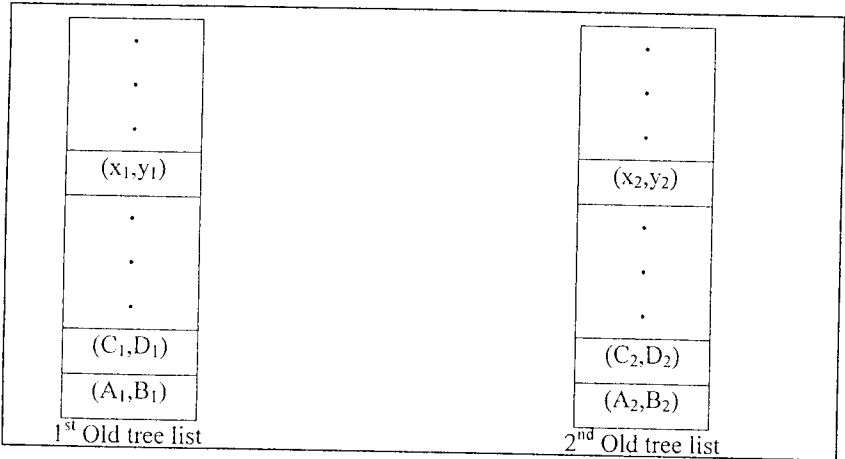


Fig.6.1: two symbolic old tree lists

B<sub>1</sub>, A<sub>2</sub> are the two boundary nodes, so no interference between the two trees till one of them combines (becomes internal). Thus, the new *LMCP* has 3 possibilities.

- (A<sub>1</sub>,B<sub>1</sub>) or (A<sub>2</sub>,B<sub>2</sub>) :



Then, we bring the next old *LMCP* from the corresponding old tree list, and we are back in the same situation to find the next new *LMCP*.

-  $(B_1, A_2)$ :

Then both *LMCPs* are broken and deleted from all subsequent appearance in the old tree lists. The new node  $(B_1 + A_2)$  is the first node in the new nodes list, and 2 leftover nodes,  $A_1$  and  $B_2$ , will result.

The process will continue in the same manner by bringing the next two *LMCPs* from both old tree lists with the two leftover nodes and the new node in the working sequence to determine the new *LMCP*. In all further steps there will be no more than one leftover node for each tree (lemma2').

---

**Lemma 2':** *There is at most one leftover node for each tree in the working sequence for merging two OATs*

**Proof:**

We conduct the proof for one tree, and it is the same for the other one.

Consider the first broken *LMCP* from one tree, we have a leftover node say  $x$  and a new node say  $N$  in our working a sequence. The next *LMCP* say  $v, w$  is brought to the working sequence. We are going to prove that neither  $v$  nor  $w$  will combine and leave its partner till  $x$  combines.

There are 3 possible relative positions between  $v, w$ .

◆  $x \quad v \quad w \quad N$ :

Since  $x$  is compatible with  $N$ , both  $v, w$  must be internal; they were available at the time  $x$  was chosen and yet was not chosen instead. Thus  $x < v, w$  and will be favored over them in any *LMCP*; i.e. neither  $v$  nor  $w$  will be chosen with any node other than  $x$ . -----(1)

◆  $v \quad x \quad w \quad N$ :

$v, w$  are compatible, so  $x$  must be internal. Also,  $x, N$  are compatible, so  $w$  is internal.  $x$  was chosen first although  $v, w$  were compatible too. Thus  $x < v, w$  and they can not combine before  $x$  does----- (2)

◆  $v \quad w \quad x \quad N$ :

-If  $x$  is internal, then  $w$  was compatible and not chosen; i.e.  $x < w$ . If  $w$  is internal too, the same applies to  $v$ ; if not then  $v$  can only combine with  $w$  causing no more leftover nodes to appear.

-If  $x$  is external then  $v, w$  can only combine with  $x$  and thus not before it.

------(3)

From (1),(2),(3):

Another leftover node can not result till the existing one combines, the same argument stands along all steps of the working sequence?

If an *LMCP* contains a single node and there is no corresponding leftover node, we consider it as a leftover node and look ahead for the next entry (statement3). Also, like the insertion algorithm, if the nearest node in old *LMCP* is external, we need to add the minimum internal compatible node (X) from that tree.

Thus, in general the working sequence will contain 2 *LMCP* pairs (one for each tree), 2 leftover nodes, 2 minimum internal compatible nodes, and finally 2 nodes from the new nodes list; a total of 10 nodes. The following example will demonstrate the process.

#### **Theorem 4:**

*When merging two optimal alphabetic trees into one optimal alphabetic tree, every new LMCP can be determined by examining no more than ten nodes.*

**Proof:**

Let us call the two old *LMCP*  $v, w$  and  $v', w'$ , the leftover nodes  $x, x'$ , the minimum compatible internal nodes  $z, z'$ , the 2 first nodes in the new nodes list  $N_1, N_2$

Let us assume another node  $U$  is needed in the working sequence. We can safely assume that  $U$  is an old node, since it is clear that we do not need more than 2 node from the new nodes list. We will assume that  $U$  belongs to the first tree, the same argument will apply if  $U$  is from the second tree. The working sequence will be as follows

$w \quad v \quad (z) \quad x \quad U \quad (N_1) \quad (N_2) \quad x' \quad (z') \quad v' \quad w'$

where  $z, z', N_1, N_2$  are internal nodes, while the rest could be internal or external. There are no restrictions on the relative positions of the nodes as long as they are all compatible.

#### **Case1: U is to the left of v:**

- $U$  was available in the old tree at the time  $(v, w)$  was formed and yet was not chosen. Then,  $v < U$  which means that  $v$  will be chosen over  $U$  in any summation.
- For the summation  $U+v$ , either it is not valid ( $w$  is external) or  $v+w < v+U$  (since it was chosen over it in the old tree).

- If the old *LMCP* is a single node, the deleted node must be compatible with the working sequence. So either  $v$  is deleted and  $w$  remains, or  $w$  is deleted and  $v$  is an internal node. In both cases, the only summation  $U$  can participate in is with the remaining node, say  $w$ ; if  $w$  is internal then  $w < U$  and if it is external then it blocks  $U$  from the rest of the working sequence. At the same time, from the order of *LMCPs*, the deleted node is smaller than  $U$  and larger than  $x$ , and thus  $x < U$ ; so it can not be chosen. Recall that  $x$  must exist in case of a single node otherwise we would have added the next entry (statement 3).

Thus  $U$  can not be chosen in the new *LMCP*.

**Case2:  $U$  is to the right of  $v$ :**

- ♦ If  $U$  is external:
  - It can not be to the right of  $x$ , since  $x$  is compatible with  $N_1, N_2$  (statement 1).
  - It can not be between  $x, v$  (or  $v, N_1$  or  $N_2$  if  $x$  did not exist), otherwise  $(v, w)$  would have been considered a blocked *LMCP* and was not added to the working sequence in the first place.

Thus  $U$  can not be external.

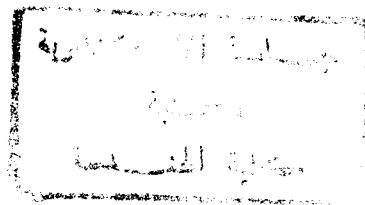
- ♦ If  $U$  is internal:
  - By definition,  $z$  is the minimum compatible internal node thus  $z < U$  and can replace it in any summation.
  - The summation  $z+U > v+w$  otherwise it would have been chosen instead.
  - Again, if the *LMCP* is a single node we still know that  $z+x < z+U$ .

Thus,  $U$  can not be chosen in the new *LMCP*.

Hence  $U$  is not a valid candidate for the new *LMCP* whether it is external or internal, and no matter what position it is in?

#### 6.1.4 Example

Fig. 6.2 shows an example of two 6 nodes OATs to be merged; the weight sequences and the old tree lists are shown.



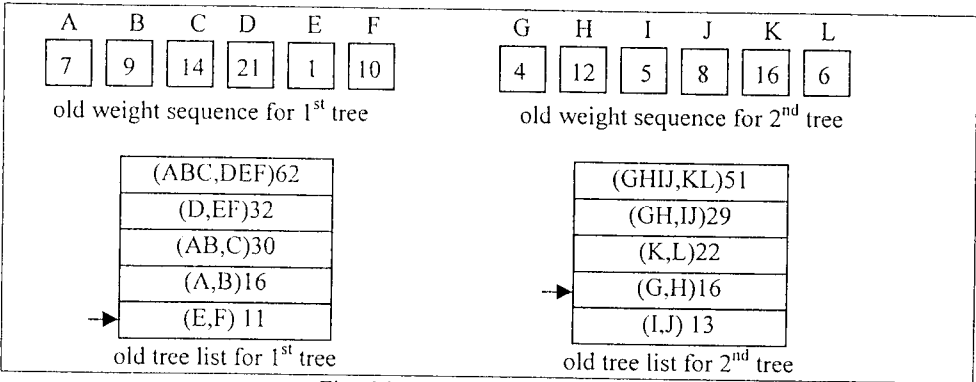


Fig. 6.2: two OATs to be merged

The old *LMCP* (I,J) is copied since it is before the node G combined. Then (E,F) and (G,H) constitute the first working sequence, and (E,F) is the chosen *LMCP*. The next one from the 1<sup>st</sup> tree is looked up (A,B) and found to be blocked, and so is (AB,C). Then (D,EF) are brought to the working sequence, where (EF,G) is the new *LMCP*, D,H become the first leftover nodes and EFG is the first node in the new nodes list. After that, ABC is a single node brought from the 1<sup>st</sup> tree, and (K,L) are brought from the 2<sup>nd</sup> one. Also, since K is external the minimum internal compatible node, I J, is brought to the working sequence. (K,L) is chosen as the new *LMCP*, and the process continues in the same manner till the root is formed at the last step; different working sequences, the new nodes list, and the new tree list are shown in Fig. 6.3. Blocked entries from the 1<sup>st</sup> tree are marked by (L), and from the right tree by (R).

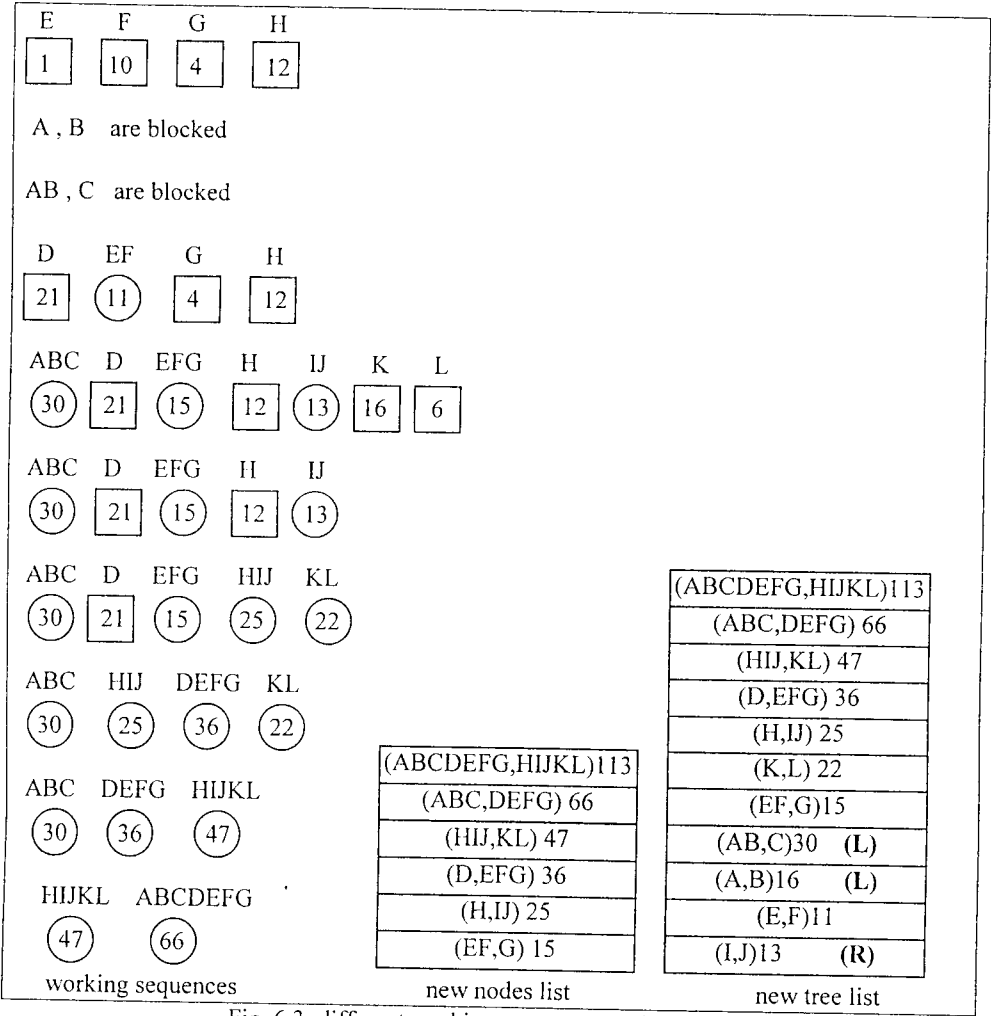


Fig. 6.3: different working sequences and final lists

### 6.1.5 Performance Analysis

It was shown through the analysis that only a constant number of nodes (10) is needed to form each LMCP in the new tree. Since there are  $n$  such LMCPs for an  $n$  node tree, merging two OATs into one tree that contains both weight sequences can be done in *linear time*, result8.

**Result 8:** Two optimal alphabetic trees can be merged into one optimal alphabetic tree in linear time

Fig.6.4 shows the results of sample runs made for the algorithms, the curves show the linearity of the algorithm.

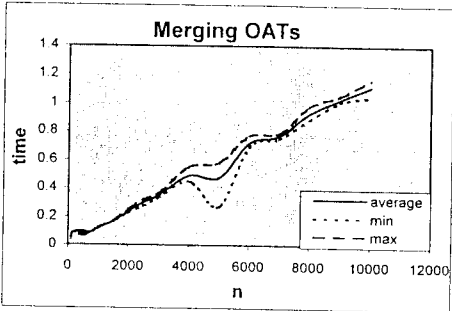


Fig 6.4.: runtime for merging two OATs into one OAT

## 6.2 Successive Merging

In this section we show how the merging algorithm can be applied successively. The discussion in this section applies to all the introduced algorithms that process *LMCP* lists (insertion, deletion, or merging).

### 6.2.1 Problem statement

In order to apply successive operations on an OAT, the new tree list produced by the proposed algorithms must store all the information needed by the algorithm exactly like the old tree list we assumed it was supplied by the original Hu-Tucker algorithm (see section 5.3). However, the proposed algorithms in the form they were prresented till now, can not get the left and right borders of each node correctly. This is because some *LMCPs* are out of order; their borders, and the borders of other *LMCPs* as well depends on their right position in the new tree list.

To illustrate the problem, consider as an example the following weight sequence of the left tree at an intermediate step of the merging algorithm; for simplicity, borders are assigned to the *LMCP* pair instead of each node.



and let the old tree list contains the following two entries.

⋮
(x,y)
(B, * )
⋮

In the old tree list C appeared in an earlier entry (leftover node), or maybe C is a new node that was composed from compatible nodes, the same applies for D,E.

(x,y) is a blocked *LMCP*, so it will be copied to the new tree list with borders ( A,B)  
(A,(x,y),B)

then B will be added to the working sequence

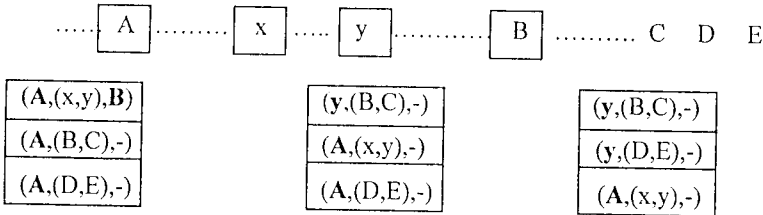
B will combine with say C forming the entry (B,C)

Then (D,E) will combine after that

The new tree list will contain the following entries(with other entries ofcourse), where (x,y) is out of order.

(x,y)
(B,C)
(D,E)

There are 3 possibilities for the sorted order, each will have different borders



Thus, there is no way of knowing the borders of these *LMCPs* before we sort the list.

### 6.2.2 Solution

The only way to adjust the borders and make the new tree list valid for successive use, is to *maintain the sorted order of the new tree list at each step*. This can be done by keeping blocked *LMCPs* in separate lists (or keeping pointers to them in their original place in the old treelists) and insert them in the right order; at each step, the selected *LMCP* is compared with the smallest blocked *LMCP* and the smaller of them is insereted.

When each *LMCP* is inserted exactly in its right order, it is much easier to adjust the borders of each node. This depends on the fact that only borders of nodes compatible with the working sequence may change, borders of blocked *LMCPs* will not change unless they became compatible when they are in the right order (like the last two possibilities for (x,y) position in the preceding example), otherwise they are copied with their old borders.

To adjust the borders of compatible nodes, we maintain two linked lists of external nodes to the left and right of the working area that did not combine yet. For example, in the insertion algorithm the left list will start with all external nodes to the left of the inserted node, and the right list will contain external nodes to the right of it. In the merging algorithm, the left list will start with all external nodes in the left tree, and the right list contains those of the right tree.

Whenever an external node combines (blocked or not blocked), it is deleted from its corresponding list (deletion is done in a constant time by maintaining a pointer from each external node to its representing node in the linked list). At any step, the borders of the working area are the heads of the two lists, then the borders of any node inside the working area can be adjusted in constant time.

**6.2.3 Example**

Here, we re solve the merging example in section 6.1.4 and show how the borders of each node can be adjusted.

We start with the two linked lists as follows

Left list: F> E> D> C> B> A

Right list: G> H> I> J> K> L

Thus, the left and right borders are F,G.

At first (I,J) is kept then when (E,F) is chosen, they are compared and (E,F) is found smaller and is inserted in the new tree list; E,F are deleted from the left list.(I,J) remains kept, (A,B) and (AB,C) are kept too since they are blocked. Then when (E,FG) is chosen, it is compared to both (I,J) and (A,B). (I,J) is the smaller so its inserted with its old borders since it remained incompatible; I,J are deleted from the right list.

The rest of the steps follow the same procedure; the chosen *LMCP* with its borders, and the corresponding left and right lists for each step of the algorithm are shown in Table 6.1.



<i>LMCP</i> pair	Left list	Right list
( <b>D</b> , (E,F),G)	D> C> B> A	G> H> I> J> K> L
( <b>H</b> , (I,J),K)	D> C> B> A	G> H> K> L
( <b>D</b> , (EF,G), <b>H</b> )	D> C> B> A	H> K> L
(-, (A,B), <b>C</b> )	D> C	H> K> L
(H, (K,L),D)	D> C	H
( <b>D</b> , (H,IJ),-)	D> C	-
(-, (AB,C), <b>D</b> )	D	-
(-, (D,EFG),-)	-	-
(-, (HIJ,KL),-)	-	-
(-, (ABC,DEFG),-)	-	-
(-, (ABCDEFGH,IJKL),-)	-	-

Table 6.1: the chosen *LMCP*, and the left and right linked lists at each step

### 6.3 Building the Optimal Alphabetic tree

In this section, we use the merging algorithm described above to build the Hu-Tucker tree in a different way that is expected to have less runtime, although it has the same order complexity. The tree is constructed by repeated merging; a procedure analog to that of the merge sort algorithm.

#### 6.3.1 Problem Definition

It is required to find the optimal alphabetic tree for a given set of weights. Here, we propose a different implementation method based on the merging algorithm of the previous section. Applying phases 2,3 of the Hu-Tucker algorithm to the *LMCP* list is a linear time process that is similar for all implementation methods. Thus, we are going to limit the discussion, whether in the analysis or in the comparison between different methods, to finding the *LMCP* list for the tree. Hence the problem can be defined as follows

*Given a weight sequence of  $n$  nodes, it is required to find the *LMCP* list for the optimal alphabetic tree of that weight sequence.*

For simplicity of recursion,  $n$  is assumed to be a power of 2; a condition that can be easily waived without loss of generality.

### 6.3.2 Algorithm

The steps of the algorithm can be summarized as follows

- The set of weights is divided into subsets of length 2, where the *LMCP* list contains a single entry, the existing pair of nodes with the node with smaller index on the left (i.e. an *LMCP* entry reserves the relative order between its two nodes).
- The  $n/2$  sublists is then grouped into  $n/4$  pairs, where each pair contains two adjacent sublists. The two *LMCP* lists of each group are merged, using the prescribed merging algorithm, giving the *LMCP* list for the subtree of the 4 nodes in the group.
- The process is repeated, at each step the existing  $n/k$  sublists each containing  $k$  nodes are merged into  $n/2k$  sublists each containing  $2k$  nodes. In the last step, two trees are merged, each of length  $n/2$ , to get the final tree.

Fig. 6.5 shows a pseudo code of the algorithm, where it is defined in a recursive manner. An optimal tree of length  $n$  is obtained by finding the optimal tree of its left and right parts, each of length  $n/2$ , recursively using the same algorithm, then merge them.

#### **Construct (1,n)**

*{ a procedure to construct a Hu-Tucker tree of the weight sequence in the array 'tree', the first parameter is the index of the beginning, and the second is of the end, the procedure returns an LMCP list }*

#### **Begin**

**If** ( $n > 2$ ) **Then**

*Construct*(1,  $n/2$ )

*Construct*( $n/2+1$ ,  $n$ )

*Merge*(1,  $n/2$ ,  $n/2+1$ ,  $n$ )

*{ a procedure for merging two sublists, the parameter represents the indices of the beginning and the end in the array tree }*

**Else**

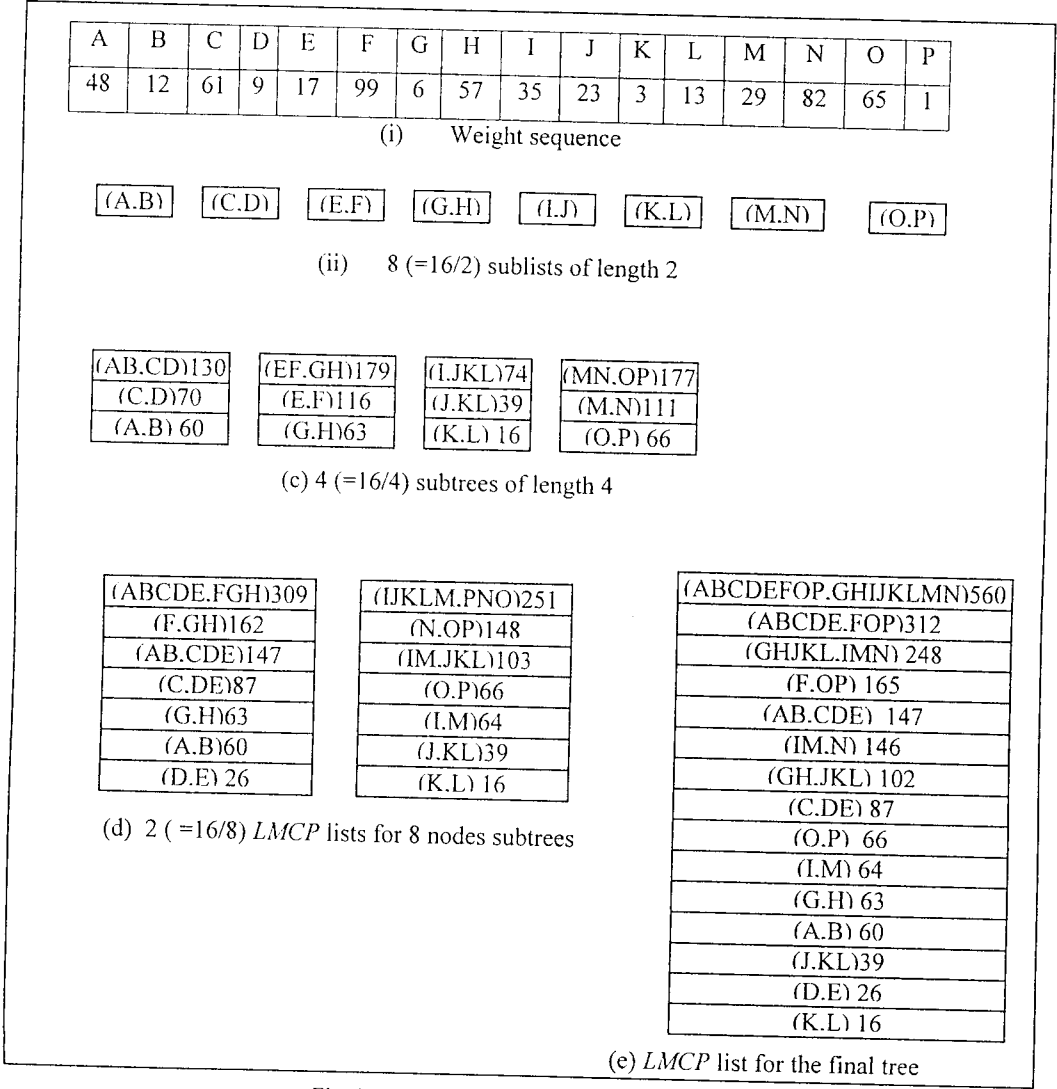
*LMCP pair* = (*tree*[1], *tree*[2])

#### **End**

Fig.6.5: pseudo code of the OAT construction algorithm

6.3.3 Example

Fig. 6.6 shows the steps of the algorithm applied on a weight sequence of 16 nodes (Fig. 6a). The corresponding LMCP lists at each step are shown in Figs 6b to 6e, where the final tree is presented.



## 6.4 Performance Analysis

In this section we evaluate the performance of the proposed method for building the optimal alphabetic tree. Performance is evaluated through complexity analysis and sample runs.

### 6.4.1 Complexity Analysis

For the space complexity, the total length of *LMCP* lists at each step of the algorithm never exceeds  $n$ . Thus the algorithm has a linear space complexity.

As for the time complexity of the algorithm  $T(n)$ , it can be described by the following recurrence relation

$$T(n) = O(n) + 2 T(n/2)$$

$$T(n) = O(n \log n)$$

The algorithm has the same order complexity as the well known method for constructing the Hu-Tucker tree. However, it is expected to have a smaller constant factor for its simplicity and since the constant of linearity for the merging algorithm is small, while the known implementation is rather complex and involves the use of many data structures (section 3.1, [18]). In specific, there are two other factors that favors our algorithm.

The first is that our algorithms has *more locality of reference*. The same set of data (memory locations) is processed by each merging tasks, then other sets are added gradually at higher levels. On the other hand, the old implementation processes different sets of data to find the *LMCP* and adjust the priority queues. Add to this, that the merging data are stored in arrays (contiguous memory locations), while the dynamic data structures there stores the data in scattered memory locations.

The second is those *copied LMCPs* in our algorithm that involves almost no processing. The rules of compatibility imposed on alphabetic trees, makes a lot of old *LMCPs* blocked and thus valid for the new tree. For example, it is a rare situation when an *LMCP* formed at the extreme left of the first tree will be broken due to the merging process.

### 6.4.2 Experimental Results

Sample runs were made to compare runtimes of the proposed algorithm with that of the known implementation. The resulting curves for the average and worst case runtimes are shown in Figs 6.7 , 6.8 respectively, where the proposed method has a considerably less runtime.

Also, sample runs showed that the number of copied *LMCPs* exceeds half the entries in the old tree list (by copied *LMCPs* we mean blocked *LMCPs* from both trees in a merging step, and *LMCPs* before the two boundary nodes appear in the old tree lists; i.e. those *LMCPs* that are not brought to the working sequence), Fig.6.9.

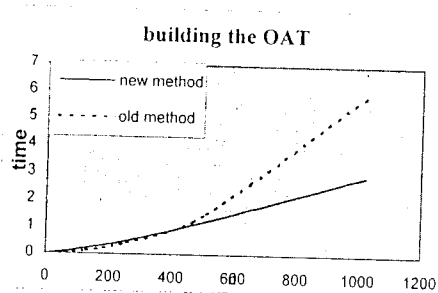


Fig.6.7: average runtime for both methods

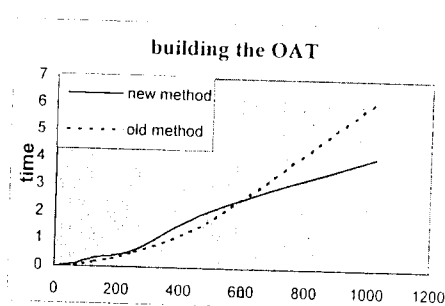


Fig.6.8: worst case runtime for both methods

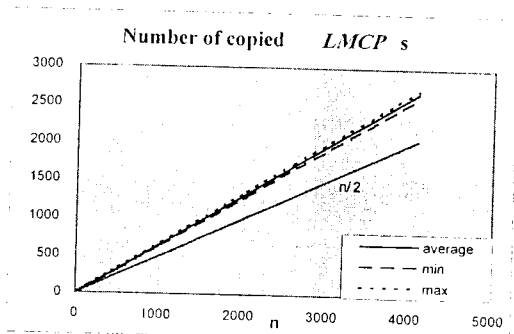


Fig. 6.9 : number of copied *LMCPs* in a merging step

All runs were made on a Pentium I 100 MHz processor, 16 MB RAM; the program was implemented with Borland C++ programming language, and the random sequences were generated using the language built-in random number generator.

### 6.5 Conclusion

In this chapter we introduced an algorithm to merge two optimal alphabetic trees into one optimal tree in linear time, such algorithm helps to expand the domain of weight sequences whose optimal alphabetic trees can be obtained in linear time. We also developed a faster technique to build the optimal alphabetic tree, using the merging algorithm, in  $O(n \log n)$  time but with a less constant than existing implementations.

## Chapter 7: Rebuilding Huffman Trees in Linear Time

Huffman trees attempt to create a minimum redundancy code that minimizes the number of bits per character. Huffman trees can be built in linear time if the weight sequence is sorted; and therefore if we can deduce the sorted order of a modified weight sequence in linear time, the modified tree can be found in linear time. In this chapter, we assert the feasibility of manipulating and editing Huffman trees in linear time by suggesting another strategy; Huffman trees can be viewed as optimal binary search trees that lack the constraint of symmetric order. So, we apply on them the techniques introduced for OATs in the preceding chapters to obtain linear time operations. Section 1 of this chapter is a preface on the work introduced in this chapter and the motivation behind it. Then section 2 gives an overview on the rules of building the Huffman tree and the linear time implementation method. Section 3 shows how this can be used to apply linear operations on Huffman trees; inserting or deleting a node from a weight sequence, changing the weight of a node, splitting a Huffman tree, and merging two Huffman trees. Then section 4 discusses how, and on what cases, the concepts and ideas introduced in the preceding chapter can be applied on Huffman trees. Section 5 discusses the insertion, section 6 the deletion, while section 7 discusses the merging, and in each case the two methods are compared through analysis and sample runs. At last, section 8 concludes the chapter.

### 7.1 Preface

Binary code trees are usually used for compression purposes (section 2.5). Huffman trees [14] are optimal binary code trees that produce a minimum redundancy code; it minimizes the average length of a code weighted for expected frequency of use. Although Huffman codes have been shown to be optimal only for block codes, as arithmetic codes are claimed to be superior to them in other cases [49], it was shown that arithmetic codes have many practical disadvantages compared to Huffman codes [50]. Thus, Huffman codes are still widely used for compression, and remain a promising area for continuous research.

Huffman trees can be built in  $O(n \log n)$  time for an  $n$  node random weight sequence. In 1990, Larmore [15] observed that the tree could be built in linear time if the weight sequence is sorted.

In this chapter, we consider several primitive operations on the weight sequence of a known Huffman tree, that still allow a linear time construction of the optimal tree for the

modified weight sequence. Examples of such operations are inserting, deleting, or changing the weight of a node, splitting an optimal tree into two optimal ones, and merging two trees into one tree. Two methods are used for the linear time construction of the new tree: the first is Larmore method based on having a sorted weight sequence, while the second is the same idea introduced in the previous two chapters, where the internal nodes already formed in the old tree are used to find the internal nodes for the new tree. *The motivation for using the second method is the fact that the changes made to the weights of a given Huffman tree may not have an effect on all its internal nodes and therefore many of the pairs combining in the given tree may still remain valid combinations for the new tree.* The simple operation of deleting a node illustrates the point; while the first method will work on the new sorted sequence of  $(n-1)$  nodes again forming all  $(n-2)$  internal nodes for the new tree, the second method may only recompute a few affected internal nodes for the new tree and just copy the unaffected ones from the old tree to the new tree.

## 7.2 Building the Huffman Tree

This section summarizes the rules for pairs of nodes to combine in the Huffman tree, and gives the linear time implementation method for a sorted sequence.

Huffman trees are built using the following bottom up technique.

Given the original weight sequence, let  $q_i$  denotes the weight of node  $i$  or the node itself. At each step a pair of nodes  $q_i, q_j$  that constitute the *minimal pair*, **MP**, combine to form another node with weight  $q_i + q_j$  that replaces them in the weight sequence. Two nodes  $q_i, q_j$  constitute an *MP* if they have the two smallest weights in the weight sequence. Nodes keep combining at each step till the last two nodes combine to form the root. Thus the Huffman tree is completely specified by its list of *MPs*.

### 7.2.1 Linear time implementation for sorted weight sequences

When the given set of nodes is sorted and in order to maintain a linear time implementation, the new nodes formed by combining pairs into *MPs* are kept in a separate list, other than the one that contains the sorted weight sequence, and since new nodes are generated in ascending order, this list will also be sorted [15]. Thus, at each step we have to choose the two smallest nodes among the nodes kept in two sorted lists, the new *MP* at each step can thus be found in  $O(1)$  time by examining at most 4 nodes. The Huffman tree can therefore be obtained in linear time when the given sequence of weights is sorted. In fact,

some  $O(n \log n)$  implementations for finding the Huffman tree apply some sorting algorithm on the weight sequence, usually the Heap Sort or the Quick Sort, then build the tree in linear time [51].

**Example**

Consider the sorted weight sequence in Table 7.1

A	B	C	D	E	F
4	7	9	13	19	25

Table 7.1: a sorted weight sequence

The steps for building the tree are demonstrated in Table 7.2, the two lists and the chosen  $MP$  are shown in each step.

Weight list	New nodes list	$MP$
{A,B,C,D,E,F}	{}	(A,B) 11
{C,D,E,F}	{AB}	(C,AB) 20
{D,E,F}	{CAB}	(D,E) 32
{F}	{CAB,DE}	(CAB,F) 45
{}	{DE,CABF}	(DE,CABF) 77

Table 7.2: the steps of building the Huffman tree for a sorted weight sequence

**7.3 Primitive Operations with sorted sequences-Method one**

Given the Huffman tree for an  $n$ -node sequence as specified by its  $MP$  list, the sorted weight sequence is readily obtained in  $O(n)$  time by simply scanning the  $MP$  list writing leaf nodes in the order of their appearance. When changes are made to the weight sequence Method one first sorts the new weight sequence then finds the new Huffman tree in linear time. Clearly if the new sorted list can be obtained in  $O(n)$  time or better the new Huffman tree can be obtained in linear time. This is trivially achieved in the following operations.

**7.3.1 Insertion**

To insert a node in a Huffman tree, the weight of the node is inserted in the sorted weight sequence in  $O(\log n)$  time.



### 7. 3.2 Deletion

The weight of the deleted node is omitted from the weight sequence keeping the sequence sorted.

### 7.3.3 Changing the weight of a node

If the change violates the sorted order, then the node must be deleted and the new weight reinserted in its proper position.

### 7.3.4 Splitting the tree into two subtrees

Splitting a tree into two trees can be done by splitting the weight sequence into two subsequences. Separating the subsequences will cost one scan to the weight sequence and will keep them sorted.

### 7. 3.5 Merging

The idea is the same, the two sorted weight sequences of the two trees are merged into one sorted sequence in linear time.

### 7. 3.6 Other operations

Any kind of operation, linear or nonlinear, that preserves the relative order of nodes in the weight sequence. Examples are

- Adding two sorted weight sequences.
- Multiplying the weight sequence by a constant, or adding a constant to it; in general  $aX + b$ , where  $a, b$  are non negative constants and  $X$  is the original weight sequence.
- Raising all weights to a constant power, like squaring the weight values for example.
- Any combinations of the above.

In all these cases, and a lot more, the resulting weight sequence remains sorted, thus the Huffman tree can be obtained in linear time.

## 7.4 Primitive Operations by processing the *MP* list-Method two

The idea is to emulate the process of building the Huffman tree for the new sequence of nodes given information about the steps of building the old tree. The problem statement is as follows : *Given the sequence of MP's generated for an  $n$ - node weight sequence, it is*

required to find, in linear time, the new sequence of *MPs* after applying some primitive operation on the original weight sequence.

#### 7.4.1 Algorithm Outline

The idea is to use the information already available from the old tree, and the rules of forming an *MP*, to limit as much as possible the number of nodes we must look at to choose a new *MP*. We are going to prove that only a constant number of nodes needs to be examined at each step and hence comes the linearity of the algorithm.

#### 7.4.2 Terminology

In what follows, we will call the sequence of *MPs* for the old tree, *the old tree list*. Due to the combination rules, the entries of the old tree list are sorted. Similarly, we will call the sequence of *MPs* for the new tree, *the new tree list*; and the set of nodes we need to examine to form each new *MP* *the working sequence*.

##### Old *MPs*

The general form of an old *MP* is two nodes that constituted the minimum pair at that time. However, during the course of the algorithm we may face a special kind of *MPs* that will be handled differently.

##### Single Nodes

A single node is an old *MP* with one of its nodes deleted.

When an old *MP* is examined and found not to be a valid *MP* for the new tree, it must be deleted from any further appearance in the old tree list. This will result in *MPs* with one deleted node and one valid node.

##### New Nodes List

The new nodes list contains all the newly generated *MPs* that are not included in the old tree list.

Due to the rules of *MP* generation, new nodes will be generated in ascending order and the new nodes list is sorted; actually it can be best represented by a "queue", where new nodes are inserted at the tail of the list while the head of the list refers to the smallest node. Also, since the list is sorted, only the first two nodes of the list need to be examined in the working sequence. When a new node combines it is virtually deleted from the list, i.e. the head is moved to the next node.

### Leftover Nodes

*A leftover node is a node that combined to form an MP for the old tree but whose partner node combined to form an MP for the new tree.*

When the two nodes forming an old MP are added to the working sequence, and only one of them combines to form a new MP, the other node remains in the working sequence and is called a *leftover node*.

Since in Huffman trees nodes combine based solely on their relative order, a leftover node is always smaller than all nodes in upcoming entries in its corresponding old tree list. Thus, another leftover node cannot result until the existing one combines, and we have at most one leftover node in the working sequence.

### 7.4.3 The method on Huffman trees compared to OATs

The reader may have noticed a resemblance between the material of this section and sections 5.1 & 5.2 since they almost describe the same concepts. In fact, the same method is expected to give less runtime than the Hu-Tucker case, as the constant of linearity will decrease since we have at most 4 nodes to examine in all cases. Also, there are no compatibility check, no blocked MPs, and thus there are no merging of blocked nodes and lower part of old tree list. The two running pointers used to get the minimum compatible internal node and the MP from the other side are no longer needed. In addition, for the case of insertion, since there are no external nodes there is no difference between inserting at the boundary or in the middle of a Huffman tree; there are no MPs that becomes non compatible due to the insertion and the inserted node is not a barrier that prevents nodes from combining. Thus, *there is always one leftover node and one new nodes list.*

### Splitting a Huffman tree

Another important difference from OATs is that *using method two in splitting a tree is not possible in Huffman trees*. This comes from the fact that Huffman trees do not reserve the initial order of nodes in the weight sequence, so there is no significance of the position the node is inserted in. A node with much larger weight will be the last to combine without affecting the combination of the rest of nodes.

7.5 Insertion

Entries from the old tree list are checked against the inserted node and, if smaller, are copied to the new tree list until an entry contains at least one node larger than the inserted node. The two smallest nodes combine as the new *MP* which is added to the new nodes list, this broken entry is deleted from all subsequent appearances in the old tree list, and the largest of them becomes the first leftover node. In all subsequent stages, in order to determine an *MP* entry in the new tree list we must examine at most two nodes from the old tree list, two nodes from the new nodes list and one left over node for a total of at most *five* nodes.

Since the leftover node is guaranteed to be smaller than both nodes in the current *MP* from the old tree list, we can exclude one more node. If we keep the relative order between the 2 nodes forming an *MP* in the old tree list, an information that was indeed available at the time of construction, only the smaller of them is needed in the working sequence. Hence only *four* nodes at most are examined at each step.

Example

Consider the weight sequence and the old tree list in Fig.7.1, it is desired to find the new tree list after inserting the node  $Z = 7$

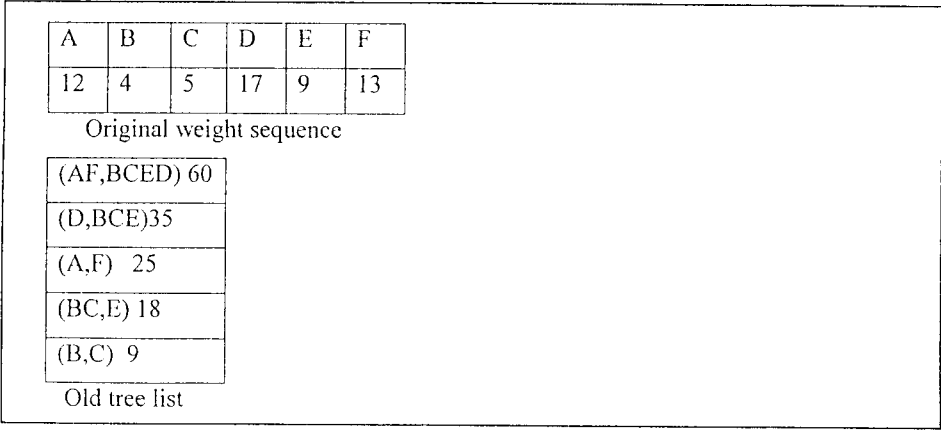


Fig.7.1: the weight sequence and old tree list

The first entry is copied to the new tree list since both of its nodes are smaller than the inserted node. The work starts at the second entry which is broken, (Z,BC) becomes the new node while E becomes a leftover node. Then (A,F) is reached; only the smallest (A) is added to the working sequence. When (A,E) is found to be the new *MP*, F becomes the leftover

node and D, the single node in the next entry, is added to the working sequence. Fig.7.2 shows the old tree list after the second entry is broken. The new nodes list, the new tree list, and the different working sequences till the final tree is reached are also shown.

A	E	BCZ			
12	9	16			
D	F	BCZ	AE		
17	13	16	21	(DAE,FBCZ) 67	(AF,*)
D	AE	FBCZ		(D,AE) 38	(D,* )
17	21	29		(F,BCZ) 29	(A,F) 25
FBCZ		DAE		(A,E) 21	(*,*)
29		38		(BC,Z) 16	(B,C) 9
Working Sequences				new nodes list	old tree list
					new tree list

Fig.7.2: remaining working sequences and final lists

### Complexity analysis

- It is clear that the insertion algorithm has a *linear time, and space, complexity* since a constant number of nodes is examined to determine each *MP*. The following notes put some light on the constant of linearity.
- The first part, the copying part, where nodes are just checked and copied to the new tree list can be done using 1 comparison to the larger node of the *MP*.
  - Two comparisons are needed to determine the new *MP* from the working sequence in every step.
  - Deleting broken entries from the old tree list will not contribute more than  $2n$  deletions to the whole process.

### The insertion algorithm versus insertion via linear construction of tree

A first draw back of Method one is an overhead of  $O(\log n)$  for inserting the node in the sorted weight sequence. Thus, this new method is expected to have less runtime. Sample runs that measured the runtime of both methods for randomly generated weight sequences as the number of nodes changes, asserts this fact. Fig.7.3 shows the average time, while Fig.7.4 shows the worst case values. The curves show that methd two (the new method) is better than method one (the old method).

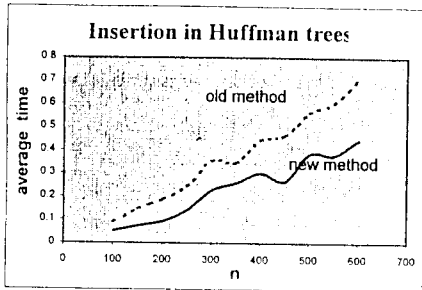


Fig.7.3: average run times

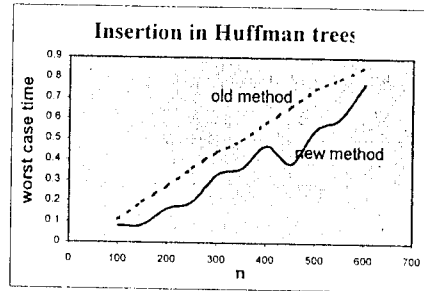


Fig.7.4: worst case run times

The second difference is that, the whole point of this technique is to make good use of the information already available from the old tree; the affected branches only are rebuilt and not the whole tree. *The runtime of this new method decreases as the inserted value gets larger*, while the performance of the other method is the same for all values.

Again, this is verified through sample runs, where a weight sequence of 500 nodes, that constitutes a degenerate tree, was fixed while the inserted value was varied to equal each of its values. The runs show that the old method has almost the same performance for all values (a horizontal line), while the runtime of the new method decreases with the increase in value, Fig.7.5.

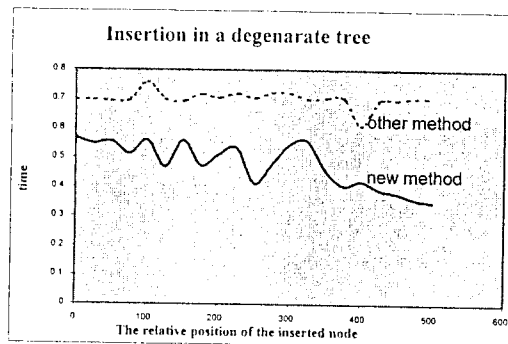
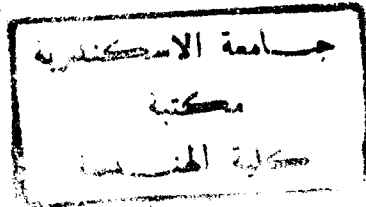


Fig.7.5: runtimes for a degenerate tree

## 7.6 Deletion

The idea of deletion is applicable here too and even much easier. The node is simply deleted from its first appearance, and all subsequent appearances, in the old tree list. Its companion node is considered a new node and the process continues as usual. A maximum of 4 nodes is needed at each step to determine the new *MP*.



Example

Let us consider the previous example and try to delete the node just inserted  $Z = 7$ . The old weight sequence and the old tree list will be those in Fig.7.6

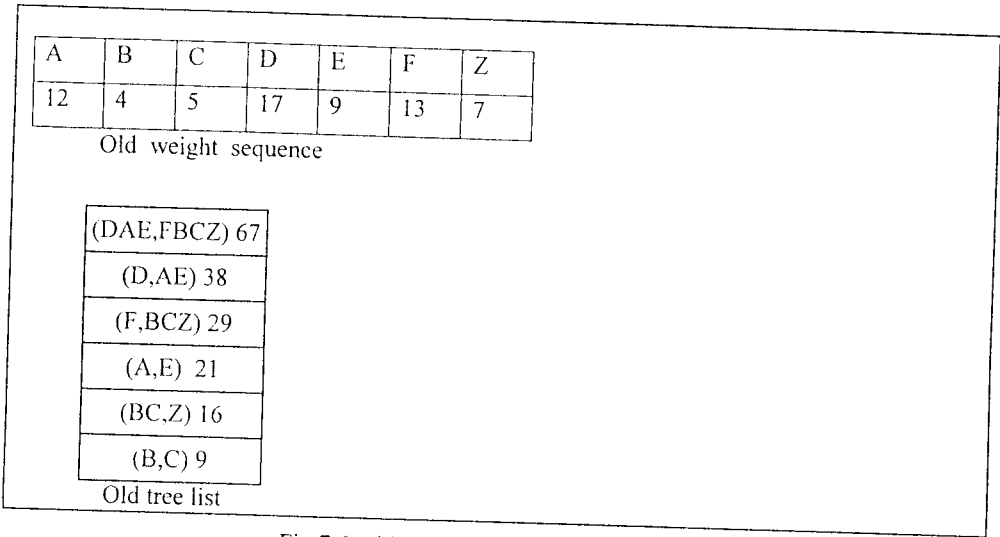


Fig.7.6: old weight sequence and old tree list

$Z$  participates in the second  $MP$ , so its broken and deleted from all subsequent appearances; the resulting old tree list is in Fig.7.7.  $BC$  now becomes a new node we are trying to insert. The following entry is checked and  $A$  is found to be larger than it; a new node  $EBC$  results and  $A$  becomes a leftover node. The final list and all working sequences appear in Fig7.7.

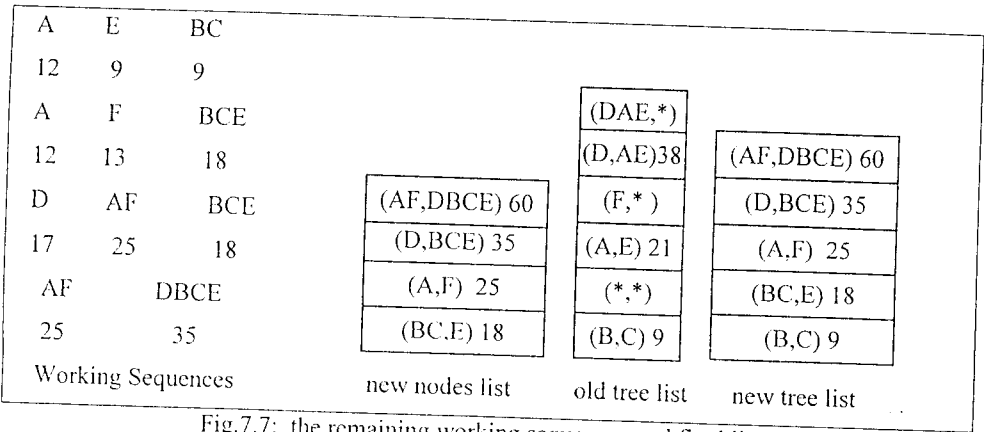


Fig.7.7: the remaining working sequences and final lists

## Complexity analysis

The procedure for deleting a node consists of two parts.

- Finding the first appearance of the deleted node in the old tree list. One may assume that a link is available from each node in the weight sequence to its position in the old tree list. However, if such link does not exist, we can scan the old tree list until we find it; a process that has a best case of 1 comparison (the smallest node in the weight sequence), a worst case of  $n$  comparisons (the largest node in a degenerate tree).
- The insertion part, which has the same complexity as the insertion algorithm of last section.

## Average and worst case

For the same reasons described in the insertion, the larger the deleted value the less the run time of this algorithm. A very good example is a degenerate tree, deleting the smallest node will break the whole tree and cause it to be completely rebuilt, while deleting the largest node just removes it and its companion node becomes the root of the new optimal tree. This algorithm uses this information and takes advantage of the existing tree, while the old algorithm (method one) treats the two cases in the same way.

The continuous curve in Fig.7.8 shows the results of sample runs that measured the time for deleting different nodes in the same weight sequence. A Fibonacci series of 500 nodes was chosen as the weight sequence as it makes a degenerate tree; the deleted node was varied from smaller nodes where the insertion starts at early entries to larger nodes where the insertion part starts later. The results show, as expected, a decreasing function; the best case is when the last value is deleted then there is no insertion at all, while the worst case is when the first entry is deleted. The dotted curve, which represents method one, is almost a horizontal line. The horizontal line has a larger value except for the worst case when the two methods almost have the same performance.



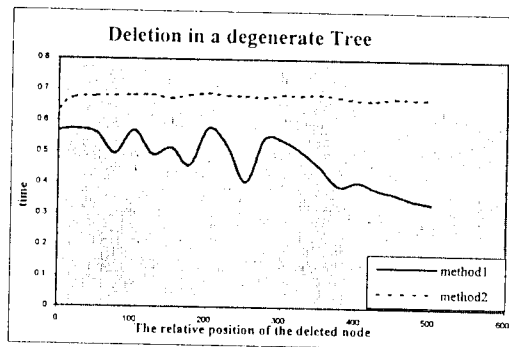


Fig.7.8: Deletion run times in a degenerate tree

Deleting the smallest node in a degenerate tree is the only case when all the *MPs* of the old tree are broken, otherwise there are always some valid entries that can be saved from the old tree. This is why, method two is superior to method one in the cases of insertion and deletion. Figs 9,10 show the results of sample runs that measured the runtime of both methods for randomly generated weight sequences as the number of nodes changes. Fig.7.9 shows the average time, while Fig.7.10 shows the worst case values, in both cases the new method has less runtime.

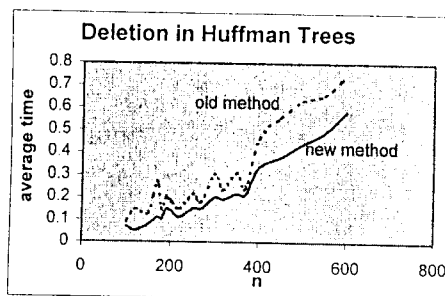


Fig.7.9: average runtimes for both methods

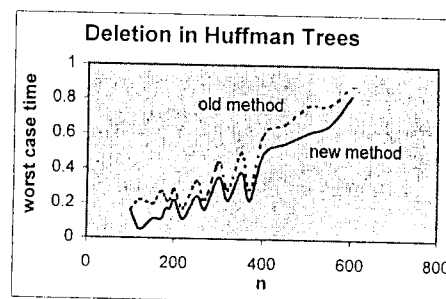


Fig.7.10 : Worst case time for both methods

### Changing the weight of a node

The only way to use method two to build the tree after a weight of a node is changed is to delete the node and then reinsert it with the desired weight. This is of course time consuming and method one is more efficient.

7.7 Merging Huffman Trees

Given 2 sequences of *MPs* generated for 2 weight sequences of length  $n_1, n_2$ , it is required to find, in linear time, the corresponding sequence of *MPs* after merging the two weight sequences into one weight sequence of length  $n_1 + n_2$ .

The idea is the same; we examine an *MP* from each tree to find the new *MP*, when one of the old *MPs* is chosen as the new *MP*, we simply get the next one from the corresponding old tree list.

Fig. 7.11 shows two symbolic sequences of *MPs* to merge, where the left node is assumed to be the smaller one. We trace the first few entries to determine which nodes are needed in the working sequence.

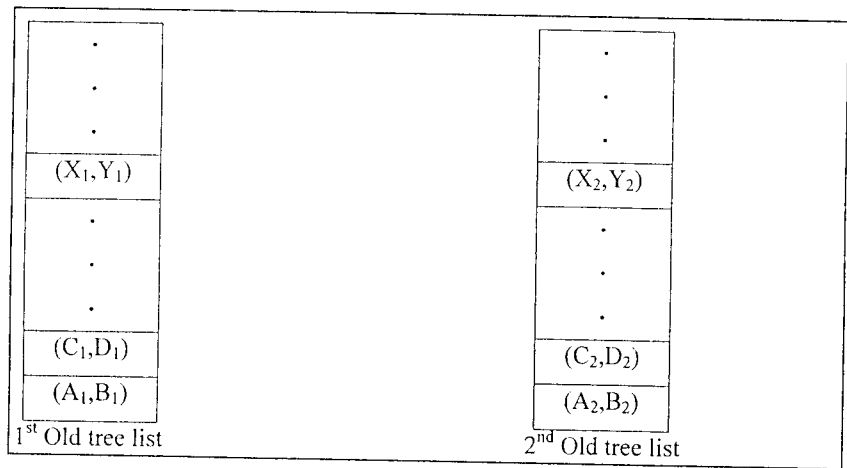


Fig.7.11: two symbolic old tree lists

We know from the rules of constructing the Huffman tree that  $A_i, B_i$  ( $i = 1, 2$ ) are smaller than all nodes in upcoming entries in their list. Thus, they are the only valid candidates for the first *MP* in the new tree. We have 3 possibilities.

- $(A_1, B_1)$  or  $(A_2, B_2)$

then, we bring the next old *MP* from the corresponding old tree list

- $(A_1, A_2)$ :

then both *MPs* are broken and deleted from all subsequent appearances in the old tree lists.

The new node  $(A_1 + A_2)$  is the first node in the new nodes list, and 2 leftover nodes,  $B_1$  and  $B_2$ , will result. The process will continue in the same manner by bringing the next two *MPs* from both old tree lists with the two leftover nodes and the new node in the working sequence to determine the new *MP*.

Since the leftover node is guaranteed to be smaller than upcoming nodes in its corresponding tree, there is only one leftover node for each old tree list. Also, since we know the relative order of the pair of nodes forming each *MP* then only the smallest of the two is needed in the working sequence. Thus, in general the working sequence for merging two Huffman trees will contain at most *six* nodes; the smallest node of each old *MP*, a left over node for each old tree list, and two nodes from the new nodes list.

Example

Fig. 7.12 shows two weight sequences, and the corresponding lists of *MPs*. It is desired to merge the two trees into one tree using the described merging algorithm. Figs 7.13& 7.14 show the working sequences for each step of the algorithm, together with the new nodes list and the new tree list.

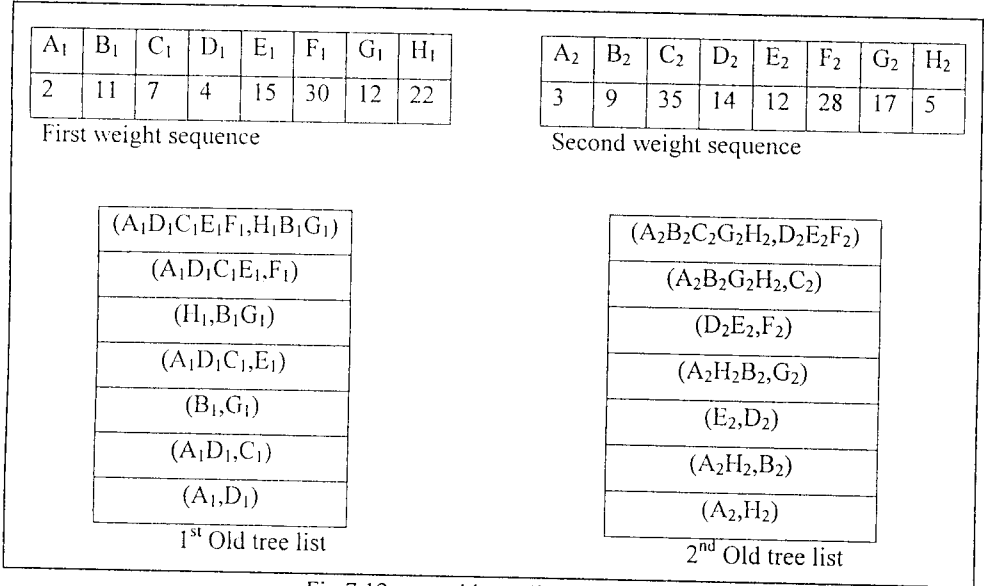


Fig.7.12: two old tree lists to be merged

The first two *MPs* are brought to the working sequence where (A<sub>1</sub>,A<sub>2</sub>) is found to be the new *MP*; the two old *MPs* are broken and deleted from all subsequent entries, and the new node is added to both the new nodes list and the new tree list. D<sub>1</sub> and H<sub>2</sub> become the first leftover nodes, and the next *MPs*, which happen to be single nodes, are added to the working sequence. Fig. 7.13 shows how the first eight *MPs* are formed, giving the working sequences, the new nodes list, and the new tree list.

A <sub>1</sub>	D <sub>1</sub>	A <sub>2</sub>	H <sub>2</sub>						
2	4	3	5						
C <sub>1</sub>	D <sub>1</sub>	A <sub>1</sub> A <sub>2</sub>	H <sub>2</sub>	B <sub>2</sub>					
7	4	5	5	9					
B <sub>1</sub>	C <sub>1</sub>	D <sub>1</sub> A <sub>1</sub> A <sub>2</sub>	H <sub>2</sub>	B <sub>2</sub>					
11	7	9	5	9					
B <sub>1</sub>	G <sub>1</sub>	D <sub>1</sub> A <sub>1</sub> A <sub>2</sub>	C <sub>1</sub> H <sub>2</sub>	B <sub>2</sub>	E <sub>2</sub>				
11	12	9	12	9	12			(G <sub>2</sub> , B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> )35	
B <sub>1</sub>	G <sub>1</sub>	C <sub>1</sub> H <sub>2</sub>	B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub>	E <sub>2</sub>	D <sub>2</sub>			(E <sub>1</sub> , D <sub>2</sub> ) 29	
11	12	12	18	12	14			(C <sub>1</sub> H <sub>2</sub> , E <sub>2</sub> ) 24	
H <sub>1</sub>	E <sub>1</sub>	C <sub>1</sub> H <sub>2</sub>	B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub>	E <sub>2</sub>	D <sub>2</sub>			(C <sub>1</sub> H <sub>2</sub> , E <sub>2</sub> ) 24	
22	15	12	18	12	14			(B <sub>2</sub> , D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> )18	
H <sub>1</sub>	E <sub>1</sub>	B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub>	C <sub>1</sub> H <sub>2</sub> E <sub>2</sub>	D <sub>2</sub>	G <sub>2</sub>			(C <sub>1</sub> , H <sub>2</sub> ) 12	
22	15	18	24	14	17			(D <sub>1</sub> , A <sub>1</sub> A <sub>2</sub> ) 9	
H <sub>1</sub>	B <sub>1</sub> G <sub>1</sub>	B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub>	C <sub>1</sub> H <sub>2</sub> E <sub>2</sub>	G <sub>2</sub>	F <sub>2</sub>			(A <sub>1</sub> , A <sub>2</sub> ) 5	
22	23	18	24	17	28				
Working sequences						new nodes list		new tree list	

The remaining steps are shown in Fig.7.14. At the ninth step the last *MP* from the second tree is brought to the working sequence; the single node  $C_2$ , while the last one from the first tree, the single node  $H_1B_1G_1$ , is brought at the tenth step. After that more nodes are added only from the new nodes list when an existing new node combines. The final list of *MPs* is also shown in the figure. Bold entries refer to old *MPs* that existed in one of the old tree lists, and do not exist in the new nodes list. It is worth mentioning that the new tree list is sorted by nature when dealing with Huffman trees; i.e., old *MPs* appear in their right order in the new tree list.

H <sub>1</sub>	B <sub>1</sub> G <sub>1</sub>	C <sub>1</sub> H <sub>2</sub> E <sub>2</sub>	E <sub>1</sub> D <sub>2</sub>	F <sub>2</sub>	C <sub>2</sub>	
22	23	24	29	28	35	(F <sub>1</sub> E <sub>1</sub> D <sub>2</sub> C <sub>2</sub> G <sub>2</sub> B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> ,H <sub>1</sub> B <sub>1</sub> G <sub>1</sub> C <sub>1</sub> H <sub>2</sub> E <sub>2</sub> F <sub>2</sub> ) 226
H <sub>1</sub> B <sub>1</sub> G <sub>1</sub>	F <sub>1</sub>	C <sub>1</sub> H <sub>2</sub> E <sub>2</sub>	E <sub>1</sub> D <sub>2</sub>	F <sub>2</sub>	C <sub>2</sub>	(F <sub>1</sub> E <sub>1</sub> D <sub>2</sub> ,C <sub>2</sub> G <sub>2</sub> B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> ) 129
45	30	24	29	28	35	(H <sub>1</sub> B <sub>1</sub> G <sub>1</sub> ,C <sub>1</sub> H <sub>2</sub> E <sub>2</sub> F <sub>2</sub> ) 97
H <sub>1</sub> B <sub>1</sub> G <sub>1</sub>	F <sub>1</sub>	E <sub>1</sub> D <sub>2</sub>	G <sub>2</sub> B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub>		C <sub>2</sub>	(C <sub>2</sub> ,G <sub>2</sub> B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> )70
45	30	29		35	35	(F <sub>1</sub> ,E <sub>1</sub> D <sub>2</sub> ) 59
H <sub>1</sub> B <sub>1</sub> G <sub>1</sub>	G <sub>2</sub> B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub>	C <sub>1</sub> H <sub>2</sub> E <sub>2</sub> F <sub>2</sub>			C <sub>2</sub>	(C <sub>1</sub> H <sub>2</sub> E <sub>2</sub> ,F <sub>2</sub> )52
45	35		52		35	<b>(H<sub>1</sub>,B<sub>1</sub>G<sub>1</sub>) 45</b>
H <sub>1</sub> B <sub>1</sub> G <sub>1</sub>	C <sub>1</sub> H <sub>2</sub> E <sub>2</sub> F <sub>2</sub>	F <sub>1</sub> E <sub>1</sub> D <sub>2</sub>				(G <sub>2</sub> ,B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> )35
45	52	59				(E <sub>1</sub> ,D <sub>2</sub> ) 29
F <sub>1</sub> E <sub>1</sub> D <sub>2</sub>	G <sub>2</sub> B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> C <sub>2</sub>					(C <sub>1</sub> H <sub>2</sub> ,E <sub>2</sub> ) 24
59	70					<b>(B<sub>1</sub>,G<sub>1</sub>) 23</b>
H <sub>1</sub> B <sub>1</sub> G <sub>1</sub> C <sub>1</sub> H <sub>2</sub> E <sub>2</sub> F <sub>2</sub>	F <sub>1</sub> E <sub>1</sub> D <sub>2</sub> G <sub>2</sub> B <sub>2</sub> D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> C <sub>2</sub>					(B <sub>2</sub> ,D <sub>1</sub> A <sub>1</sub> A <sub>2</sub> )18
97	129					(C <sub>1</sub> ,H <sub>2</sub> ) 12
Working sequences						(D <sub>1</sub> ,A <sub>1</sub> A <sub>2</sub> ) 9
						(A <sub>1</sub> ,A <sub>2</sub> ) 5
						New tree list

Fig.7.14: the rest of working sequences and final lists

Since a constant number of nodes is examined at each step this algorithm has a linear time complexity.

### Comparing methods one and two for merging

- The comparisons involved in method two just described ranges between  $2n$  and  $4n$  comparisons according to the number of valid *MPs* from each old tree, while the method one takes  $n$  comparisons to merge the sublists and  $2n$  to build the tree. Add to this the overhead of deleting broken entries in method two, something that does not exist in method one. Thus it is expected for method one to be superior in almost all cases.

- The only advantage of method two, is that it uses valid *MPs*. So, if the final tree is obtained by combining the roots of the two subtrees into one root, method two will cost  $2n$  comparisons for copying *MPs* and there will be no deleted entries while the method one will rebuild the tree in the same manner. However, this case is very rare and is not enough reason to favor method two in general.

- Figs 7.15, 7.16 show the results of sample runs that measured the runtime of both methods for randomly generated weight sequences as the number of nodes changes. Fig.7.15 shows the average time, while Fig.7.16 shows the best case values; in both cases the new method has a larger runtime although the gap is much less in the best case. In fact, getting back to the figures for insertion and deletion shows that the runtime for merging in the new method is almost double that for insertion since there are two lists to be processed, while that for the old method is almost the same since the effort is almost the same.

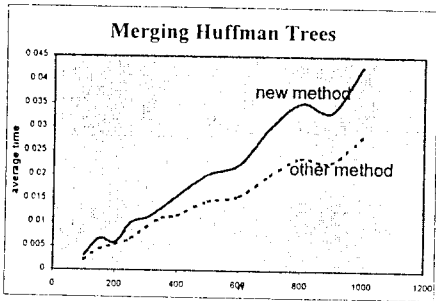


Fig.7.15: average runtime for both methods

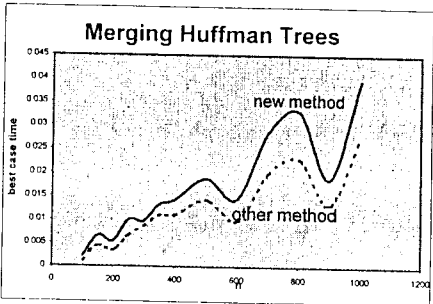


Fig.7.16: best case runtime for both methods

### Merging in Huffman trees versus merging in alphabetic trees

Unlike insertion and deletion, merging using method2 did not lead to good results, although the same idea led to good results in OATs. This is because the compatibility constraint in alphabetic trees keeps a reasonable ratio of old *LMCPs* valid for the new tree, while on Huffman trees the node forming depends solely on the weight values. So, for the same random sequence more old internal nodes tend to be broken in Huffman trees and hence more deterioration is expected in the performance.

### 7.8 Conclusion

Since Huffman trees can be built in linear time for sorted weight sequences, one should expect to be able to manipulate Huffman trees in linear time. A new method was proposed to rebuild Huffman trees after such operations as inserting a node and deleting a node and was shown to be superior to existing methods. The proposed method was also found to be inefficient in merging two huffman trees.

# Chapter 8: Conclusions & Future Work

This chapter summarizes the work done through the thesis and its conclusion and points out some ideas for future research.

## 8.1 Summary & Conclusion

Research results in this thesis are oriented in three directions

The thesis starts by proposing a new concept "the expense of a cut" followed by an algorithm for finding the 2-d OAT, experimental results show it is reasonably faster than dynamic programming, the only existing method for finding the general 2-d OAT.

Then we get back to the one dimensional problem. Linear time algorithms were developed to insert, delete, or change the weight of a node in an OAT *without losing its optimality*, to split an OAT into two OATs, and two merge two OATs into one optimal tree. Such algorithms make the OAT more dynamic and widen the range of OATs that can be found in linear time. In addition, we use the proposed merging algorithm to build the tree recursively in a divide and conquer manner; the resulting algorithm is much simpler and faster than the existing implementation.

Finally, similar algorithms were introduced for Huffman trees and compare their performance with existing methods, the proposed algorithms were found efficient in inserting or deleting a node from the weight sequence, while the merging technique was found to be inefficient.

## 8.2 Future Work

Many research points could be deduced from the results obtained in the thesis.

The same idea of processing an old tree list to get a new tree list can be studied in higher dimensions. It is logical to assume that finding the 2d OAT after inserting a row or column to an existing one can be done faster than rebuilding the tree. In addition, providing algorithms that manipulate the weight sequence of an OAT can create new ideas to faster algorithms for finding the OAT in higher dimensions.

The possibility of parallelizing the algorithms introduced in the thesis could be investigated; especially the OAT construction algorithm.

The algorithms provided in this thesis make the OAT more dynamic and easier to implement. This may open the door to new application areas for OATs, it also may enhance the performance of existing applications.



## References

- [1] A. Aho, J. Hopcroft, and J. Ullman, " *Data Structures and Algorithms* ", Addison-Wesley, Reading, MA, 1974.
- [2] A. Aho, J. Hopcroft, and J. Ullman, " *The Design and Analysis of Computer Algorithms*", Addison-Wesley, Reading, MA, 1974.
- [3] Knuth D.E., " *The Art of Computer Programming, Volume 3: Sorting and Searching*", Addison-Wesley, Reading, MA, 1973.
- [4] Andersson A., "A note on searching in a binary search tree", *Software-Practice and Experience*, 21(10) : 1125-1128, 1991.
- [5] Gaines H.F., " *Cryptanalysis*", New-York: Dover, CF, 1956.
- [6] Ahmed M. A., " *On optimal searching in multi-dimensional data with skewed access distribution*", Faculty of Engineering - Alexandria University, CS Dept., A Ph.D. thesis, 1997.
- [7] Knuth D.E., " *The Art of Computer Programming, Volume 1: Fundamental Algorithms*", Addison-Wesley, Reading, MA, 1968.
- [8] Gilbert E.N. and Moore E.F., " Variable-length binary encoding", *Bell systems technical Journal*, 38: 933-968, 1959.
- [9] Knuth D. E., "Optimum binary search trees", *Acta Informatica*, 1: 14-25, 1971.
- [10] E. Reingold and W. Hansen, " *Data Structures in Pascal*", Little, Brown and Company, Reading, MA, 1986.
- [11] Walker W.A. and Gotlieb C.C., " *Graph Theory and Computing*", Academic Press, 1972.
- [12] Decker R., " *Data structures*", Prentice-Hall, Englewood Cliffs, 1989.
- [13] Smith H., " *Data Structures Form and Function*", Harcourt Brace Jovanovich, 1987.
- [14] Huffman D.A., "A method for the construction of minimum redundancy codes", *Proceedings of the IRE*, 40(9): 1098-1101, 1952.
- [15] Larmore L.L. and Hirschberg D.S., "A fast algorithm for optimal length limited Huffman codes", *Journal ACM*, 37: 464-473, 1990.
- [16] Hu T.C. and Tucker A.C., "Optimal computer search trees and variable-length alphabetic codes", *SIAM Journal on Applied Mathematics*, 21(4): 514-532, 1971.

- [17] Garcia A.M. and Wachs M.L., "A new algorithm for minimum cost binary trees", *SIAM Journal on Computing*, 6(4): 622-642, 1977.
- [18] Davis S.T., "Hu-Tucker algorithm for building optimal alphabetic binary search trees", Rochester Institute of Technology CS Dept., A master thesis, Dec. 1998.
- [19] Hu T.C. and Tucker A.C., "Optimal alphabetic trees for binary search", June 1997.
- [20] J. Bentley, "Multi-dimensional binary search trees used for associative searching", *Communications of ACM*, 18(9): 509-517, September 1975.
- [21] Ahmed M.A., Belal A.A. and Ahmed K.M., "Optimal insertion in two-dimensional arrays", *International Journal of Information Sciences*, 99(1/2): 1-20, June 1997.
- [22] Hu T.C., "A new proof of the T-C algorithm", *SIAM Journal on Applied Mathematics*, 25(1): 83-94, 1973.
- [23] Yohe J. M., "Hu-Tucker minimum redundancy alphabetic coding methods [Z]", *Communications of the ACM*, 15(5): 360-362, 5 1972.
- [24] Tarjan. R. E., "Data Structures and Network Algorithms", SIAM, Philadelphia, PA, 1983.
- [25] Karpinski M., Larmore L.L., and Rytter W., "Correctness of constructing optimal alphabetic trees revisited", 1996.
- [26] Larmore L.L., Przytycka T.M., and Rytter W., "Parallel construction of optimal alphabetic trees", *In Proceedings of the 5<sup>th</sup> ACM Symposium on Parallel Algorithms and Architectures*, pages 214-223, 1993.
- [27] Kingston J.H., "A new proof of the Garsia-Wachs algorithm", *Journal of Algorithms*, 9: 129-136, 1988.
- [28] Ramanan P., "Testing the optimality of alphabetic trees", *Theoretical Computer Science*, 93(2): 279-301, 1992.
- [29] Klawe M.M. and Mumey B., "Upper and lower bounds on constructing alphabetic binary trees", *In Proceedings of the 4<sup>th</sup> Annual ACM-SIAM Symposium on Discrete Algorithms*, page 185-193, 1993.
- [30] Larmore L.L. and Przytycka T.M., "The optimal alphabetic tree problem revisited", *Journal of Algorithms*, 28(1): 1-20, June 1997.
- [31] Hu T.C. and Morgenthaler J.D., "Optimum alphabetic binary trees", *Combinatorics and Computer Science, 8<sup>th</sup> Franco-Japanese and 4<sup>th</sup> Franco-Chinese*

- Conference, Vol. 1120 of Lecture Notes in Computer Science, Springer-Verlag, 1996, pp.234-243.
- [32] Rytter W., "Efficient parallel computations for some dynamic programming problems", *Theory of Computer Science*, 59: 297-307, 1988.
  - [33] Attalah M.J., Kosaraju S.R., Larmore L.L., Miller G.L., and Teng. S-H., "Constructing trees in parallel", *In Proceedings of the 1<sup>st</sup> ACM Symposium on Parallel Algorithms and Architectures*, pages 499-533, 1989.
  - [34] Larmore L.L. and Przytycka T.M., "A parallel algorithm for almost optimal alphabetic trees", *Journal on Parallel and Distributed Computing*,
  - [35] Larmore L.L. and Przytycka T.M., "A parallel algorithm for optimum height limited alphabetic binary trees",
  - [36] Alon Itai, "Optimal alphabetic trees", *SIAM Journal on Computing*, 5(1): 9-18, March 1976.
  - [37] Wessner R.L., "Optimum alphabetic search trees with restricted maximal height", *Information Processing Letters*, 4: 90-94, 1976.
  - [38] Larmore L.L. and Hirschberg D.S., "Length-limited coding", *In Proceedings of the 1<sup>st</sup> Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 310-318, 1990.
  - [39] Larmore L.L. and Przytycka T.M., "A fast algorithm for optimum height limited alphabetic binary trees", *SIAM Journal on Computing*,
  - [40] Ahmed M.A., Helal A., Belal A.A. and Ahmed K.M., "Optimized tree structure for associative queries", *In Proceedings of the 3<sup>rd</sup> International Conference on Computer Science and Informatics*, March 1997.
  - [41] Yeung R.W., "Alphabetic codes revisited", *IEEE Transactions on Information Theory*, 37(3) :564-572, May 1991.
  - [42] Nakatsu N., " Bounds on the redundancy of binary alphabetical codes", *IEEE Transactions on Information Theory*, 37(4) :1225-1229, July 1991.
  - [43] Laber E., Milidiu R., and Pessoa A., "Practical Constructions of L-Restricted Alphabetic Prefix Codes", *In Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, 1998.
  - [44] Prabhakar B., Gupta P., and Boyd S., "A two-bit scheme for routing lookup", *In Proceedings of ITW*, June 1999.
  - [45] Vittal A. and Mark-Sadowska M., "Minimal delay interconnect design using alphabetic trees", *In Proceedings of the 31<sup>st</sup> Annual Conference on Design Automation*, pages 6-10, June 1994.

- [46] Vaishnav H. and Pedram M., "Alphabetic trees: theory and applications in layout-driven logic synthesis", *IEEE Transactions on Computer Aided Design Integrated circuits*, Jan 2001.
- [47] Pedram M. and Vaishnav H., "Technology decomposition using OATs", *In Proceedings of European Conference on Design Automation*, pages 573-577, Feb 1993.
- [48] Belal A.A., Ahmed M.A., Arafat S.M., "Limiting the search for 2-dimensional optimal alphabetic trees", *Fourth International Joint Conference on Information Sciences*, October 1998.
- [49] Witten I.H., Neal R.M., Cleary J.C., "Arithmetic coding for data compression", *Comm. ACM*, 30 : 520-540, 1987.
- [50] Bookstein A., Klein S.T., Raita T., "Is Huffman coding dead?", *In Proceedings of the ACM-SIGIR conference 1993*, pages 80-87
- [51] Knuth D.E., "Dynamic Huffman coding", *Journal of Algorithms*, 6: 163-180, 1985.

## ملخص الرسالة

تتسم التطبيقات الحديثة لنظم قواعد البيانات باحتوائها على كميات هائلة من البيانات يلزم تخزينها و استرجاعها بكفاءة و سرعة عالية. و لذا تزخر المراجع و الأبحاث العلمية بالعديد من هياكل البيانات المقترحة لتخزين هذه البيانات و الطرق المبتكرة لسرعة استرجاعها و التعامل معها. إلا أن معظم هذه الطرق تفترض أن معدل الاسترجاع واحد لجميع قيم البيانات. رغم أنه قد ثبت في معظم الحالات أن التعامل مع البيانات يكون غير متماثل التوزيع، من أظهر الأمثلة اختلاف معدل استرجاع الكلمات التي تبدأ بحرف معين في أي معجم لغوي.

و تعد الشجرات الهجائية المثلى من أهم هياكل البيانات التي تأخذ في الاعتبار عدم تماثل معدلات التعامل مع القيم المختلفة، كما يمكن استخدامها أيضا في تشفير البيانات حيث يخصص لكل حرف شفرة مختلفة يعتمد طولها على معدل استخدامها، بالإضافة إلى تطبيقات أخرى عديدة. إلا أنه يعيب هذه الشجرات أن إضافة أو حذف أي بيان تفقد الشجرة مثاليتها. تتضمن هذه الرسالة العديد من الخوارزمات الجديدة المقترحة للتعامل مع هذه الشجرات.

تبدأ الرسالة بعرض خوارزم كفاء للحصول على شجرة البحث المثلى ثنائية الأبعاد، فقد أصبحت بيانات العديد من التطبيقات الحديثة ذات طبيعة متعددة الأبعاد. ثم تعود الى البعد الواحد فتقدم مجموعة من الخوارزمات التي تضيف الكثير من المرونة للشجرة الهجائية المثلى؛ حيث تقوم هذه الخوارزمات بإضافة أو حذف أو تغيير أي بيان في زمن يتناسب خطيا مع عدد البيانات وذلك دون أن تفقد الشجرة مثاليتها. أيضا تقوم، خطيا، بدمج شجرتين مثليتين في شجرة واحدة مثلى و تقسيم شجرة مثلى إلى شجرتين مثليتين. ثم تشرح الرسالة كيف يمكن بناء شجرة البحث المثلى بسهولة و سرعة عن طريق تكرار الدمج (باستخدام أسلوب فرق تسد). و أخيرا تقوم الرسالة بتقديم الخوارزمات المناظرة لشجرات التشفير المثلى وهو نوع مشابه من الشجرات يفتقد لشرط الترتيب الهجائي للعناصر و يستخدم في التشفير.

تجدر الإشارة إلى أن جميع الخوارزمات المقدمة في الرسالة تم تقييم أدائه عن طريق التحليل النظري ثم تطبيقها عمليا ومقارنة زمن تنفيذها بزمن تنفيذ الطرق الأخرى المناظرة.

تقع الرسالة في ثمانية فصول محتواها كما يلي:

الفصل الأول: يشتمل على مقدمه للرسالة و فكرة عامة عن محتوى الفصول الأخرى.

الفصل الثاني: يعرض خلفية عامة عن الشجرات كهيكل بيانات؛ يبدأ الفصل بتعريف و شرح الشجرات وكيفية استخدامها في البحث عن البيانات و استرجاعها، ثم يقوم بتعريف شجرة البحث المثلى و يعرض لطرق الحصول عليها، ثم ينتهي بتقديم فكرة سريعة عن شجرات التشفير و الشجرات الهجائية.

الفصل الثالث: و يقدم شرح تفصيلي للشجرات الهجائية المثلى والطرق المختلفة للحصول عليها، ثم يعرض بإيجاز لتطبيقاتها المختلفة، و الخوارزمات و الأبحاث التي تناولتها.

**الفصل الرابع:** يتحدث عن الخوارزم المقترح للشجرات ثنائية الأبعاد، حيث يشتمل على خلفية مختصرة عن الشجرات ثنائية الأبعاد و الأبحاث التي تناولتها، و بعد ذلك يعرض مفهوم "تكلفة القطعة" و يقترح استخدامه في الحصول على شجرة البحث المثلى ثم يشرح خطوات الخوارزم المقترح مع مثال توضيحي له، و أخيرا يقوم بتقييم أدائه.

**الفصل الخامس:** يعرض الخوارزم المقترح لإضافة أو حذف أو تغيير أي بيان في الشجرة المحيائية المثلى ذات البعد الواحد؛ يعتبر هذا الفصل حجر أساس و نقطة انطلاق لبقية فصول الرسالة حيث يحتوى على تعريف جميع المصطلحات و المفاهيم المستخدمة و يقدم البراهين و الإثباتات الرياضية للنتائج و العلاقات التي تعتمد عليها مجموعة الخوارزمات المقدمة في الفصلين الخامس و السادس. بعد ذلك يشرح الخوارزم (شرحا مدعما بالأمثلة و البراهين اللازمة) ويتم إثبات خطية زمنه تدريجا لسهولة الفهم، حيث يتم شرح كيفية إضافة بيان على حواف الشجرة أولا، ثم الإضافة المشروطة في وسط الشجرة، ثم تشرح الحالة العامة للإضافة غير المشروطة. يلي ذلك شرح كيفية استخدام نفس الخوارزم لحذف أو تعديل أي بيان في الشجرة ولتقسيم الشجرة الى شجرتين مثليين. أخيرا يتم عرض فوائد و تطبيقات مختلفة لهذا الخوارزم.

**الفصل السادس:** يشتمل على عرض للخوارزم المقترح لدمج شجرتين في شجرة مثلى، ثم يشرح كيف يمكن بناء شجرة البحث المثلى بسهولة و سرعة عن طريق تكرار الدمج (باستخدام أسلوب فرق تسد) و يقارن بين أداء هذه الطريقة الجديدة و الطريقة الأخرى المعروفة حيث يتضح تفوق الطريقة المقترحة.

**الفصل السابع:** يقوم بتقديم الخوارزمات المناظرة لشجرات التشفير المثلى وهو نوع مشابه من الشجرات يفتقد لشرط الترتيب الهجائي للعناصر و يستخدم في التشفير ويقارن بين أدائها و أداء الطرق المتعارف عليها، و يوضح أماكن تفوق كل طريقة.

**الفصل الثامن:** يشتمل على ملخص لما تم في الرسالة و نتائج البحث، بالإضافة إلى اقتراحات لمجالات للبحث مستقبلا.

## التعامل مع الشجرات المثلى الثنائية للأكواد

رسالة مقدمة لقسم الآلات الحاسبة و التحكم الآلي كاستكمال جزئي لتطلبات الحصول على  
درجة الدكتوراه في علوم الحاسب



3607

من:

شيماء محمد عرفات

تحت إشراف:

أ.د. أحمد عبد الرافع بلال

أ.د. محمد صلاح الدين سليم

