1998

Hu-Tucker alogorithm for building optimal alphabetic binary search trees

Sashka Davis

Rochester Institute of Technology
Computer Science Department

Hu-Tucker Algorithm for Building Optimal Alphabetic Binary Search Trees

by
Sashka T. Davis

A thesis, submitted to
The Faculty of the Department of Computer Science,
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Professor S. Radziszowski

Professor P. Anderson

Professor A. Kitchen

Professor E. Hemaspaandra

December 17, 1998

Title of thesis: "HU-TUCKER ALGORITHM FOR BUILDING OPTIMAL ALPHABETIC BINARY SEARCH TREES"

I, Sashka T. Davis, hereby grant permission to the Wallace Library of the Rochester Institute of Technology to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

Date: December 17, 1998 Signature of Author:

Abstract

The purpose of this thesis is to study the behavior of the Hu-Tucker algorithm for building Optimal Alphabetic Binary Search Trees (OABST), to design an efficient implementation, and to evaluate the performance of the algorithm, and the implementation.

The three phases of the algorithm are described and their time complexities evaluated. Two separate implementations for the most expensive phase, Combination, are presented achieving $O(n^2)$ and $O(n \lg n)$ time and $O(n)$ space complexity. The break even point between them is experimentally established and the complexities of the implementations are compared against their theoretical time complexities.

The electronic version of "The Complete Works of William Shakespeare" is compressed using the Hu-Tucker algorithm and other popular compression algorithms to compare the performance of the different techniques.

The experiments justified the price that has to be paid to implement the Hu-Tucker algorithm. It is shown that an efficient implementation can process extremely large data sets relatively fast and can achieve optimality close to the Optimal Binary Tree, built using the Huffman algorithm, however the OABST can be used in both encoding and decoding processes, unlike the OBT where an additional mapping mechanism is needed for the decoding phase.

Errata

1. Page 15 reads "Pair of adjacent nodes which are at..." should be "A pair of adjacent nodes which are at..."

2. Page 11 reads "Combining nodes at position 10 and 11 results in merging of two Huffman sequences [7,8,9,10] & [10,11,12]" should be "Combining nodes at position 11, 12 results in merging of two Huffman sequences [8,9,10,11] and [11,12,13]."

3. Page 30 reads "The abstraction representing the initial sequence is the array T[N], having simpler responsibility then..." should be "The abstraction representing the initial sequence is the array T[N], which has a simpler responsibility than..."

4. Page 33 reads "The engine of the algorithm during the Reconstruction sequence." should be "The engine of the algorithm during the Reconstruction phase."

5. Page 37 reads "Mark R. Brown analyzed the performance of binomial queue, leftists trees, linear lists, binary heaps, and sorted list..." should be "Mark R. Brown analyzed the performance of binomial queues, leftist trees, binary heaps, and sorted lists..."

6. Page 43 reads "$ruse.ru\_stime.tv_usec$" should be "$ruse.ru\_stime.tv\_usec$"

7. Page 44, Figure 4.1, the last values for FILH read "$1964, 2091, 2231$" should be "$1.964, 2.091, 2.231$"

8. Page 44, Figure 4.2, the values of the y axis should be multiplied by 100. It reads "$0, 0.01, 0.02...$" should be "$0, 1, 2...$"

9. Page 45, the last sentence reads "Thus the only implementation used for read time encoding of very large alphabets using OABST should use the implementation model of the FILH." should be "Thus the only practical implementation used for real time encoding and decoding of very large alphabets using OABST is the FILH implementation."

10. Page 48 reads "Figure 4.5: Time complexity of the fast implementation with leftist heaps." should be "Figure 4.5: Time complexity of the fast implementation with leftist heaps (FILH)."

11. Page 54, the last paragraph reads "The resulting version of the text was ran through *awk* filter script and the number of the words appearances in the text are counted." should be "The resulting version of the text was ran through *awk* filter script and the number of occurrences of each word were determined."

12. Page 63 reads "For references see Section 2.2.3" should be "For references see Section 2.2.4."

13. Page 63, the last paragraph reads "This study shows that the Hu-Tucker algorithm can be implemented efficiently and the implementation can build the OABST for extremely large data sets which makes the algorithm very practical for dictionaries and electronic texts encoding and decoding." should be "This study shows that the Hu-Tucker algorithm can be implemented efficiently and the implementation can build the OABST for extremely large data sets, which makes the algorithm very practical for encoding and decoding of electronic tests."

Contents

1 Optimal Topological Bifurcating Arborescences. Definitions, Algorithms, and History **3**

1.1 Binary Trees 3

1.1.1 Binary Search Trees 4

1.1.2 Alphabetic Binary Trees 4

1.1.3 Optimal Binary Trees 4

1.1.4 Prefix-free Binary Encoding 4

1.1.5 Optimal Binary Trees vs. Optimal Alphabetic Binary Search Trees 5

1.2 History Overview 8

2 Hu-Tucker Algorithm **9**

2.1 Overview 9

2.2 Detailed Description 10

2.2.1 Phase I, Combination 10

2.2.2 Phase II Level Assignment 14

2.2.3 Phase III, Recombination 15

2.2.4 Correctness 20

3 Implementation Models and Complexity Evaluation **21**

3.1 Phase I 21

3.1.1 Fast Implementation Model 21

3.1.2 Fast Implementation Time Complexity Evaluation 29
3.1.3 Implementation Model of Combination Phase With Local Minimum Compatible Pairs. . . . 30
3.2 Phase II 31
3.2.1 Implementation Model 31
3.3 Phase III 32
3.3.1 Implementation Model 32
3.4 Time and Space Complexity of the Hu-Tucker Algorithm 35
3.5 Building OABST 35
3.6 Mergable Heaps 37
3.6.1 Leftist Heaps 38

4 The Fast Implementation Builds OABST for $2,000,000$ nodes! 42
4.1 Break Even Point 43
4.2 LMCP Implementation vs. Fast Implementation Compared for Large Data Sets. 45
4.3 Time Complexity of All Phases of the Hu-Tucker Algorithm . 50
4.4 Experimental Time Complexity vs. Theoretical Time Complexity 52
4.5 Example Book Encoding 54
4.5.1 Comparison of Different Compression Algorithms . 58
4.6 Experimental Comparison of the Cost of the Huffman and the Hu-Tucker Trees 59
4.7 Conclusions 63

A Hu-Tucker Algorithm, Phase I, Combination 65
A.1 Fast Implementation 65
A.2 Implementation with Local Minimum Compatible Pairs . . . 68

B Building OABST using the Stack Algorithm 70

C Leftist Heap Operations 72

Chapter 1

Optimal Topological Bifurcating Arborescences. Definitions, Algorithms, and History

1.1 Binary Trees

A binary tree is a linked data structure in which every node has zero, one, or two descendants. Every node except the root of the tree has exactly one parent. The leaves of the tree have no descendants and also will be referred to as external, terminal, or square nodes in this thesis. Non-leaf nodes will be referred as internal, or circular nodes. A full binary tree, also called an extended binary tree, is a binary tree whose internal nodes have two descendants, left and right.

Theorem *The number of internal nodes in extended binary tree is one less than the number of external nodes.*
Proof: [Knu73a], page 399.

Assume there are N internal nodes and S external nodes in an extended binary tree. The number of edges can be represented as $2N$ and also as $N + S - 1$. $N + S - 1 = 2N \Rightarrow \mathbf{N = S - 1}$.

1.1.1 Binary Search Trees

Let every node of a binary tree be associated with a key value. A binary search tree is a tree in which every node of the tree obeys the relationship: the values of each key found in the left subtree for a given node are less than or equal to the value of the key in the node and the values of each key in the right subtree are greater than the value of the key of that node. Scanning the leaf nodes of a binary search tree from left to right results in an increasing sequence according to the key values.

1.1.2 Alphabetic Binary Trees

If a given sequence of nodes is used to build a binary tree, the tree is called alphabetic iff the leaf nodes of the tree obey the order of the initial sequence from left to right. An alphabetic key constraint does not imply a binary search mechanism. Thus the internal nodes of the alphabetic tree do not have to provide information for locating the leaf nodes of the tree.

An alphabetic *binary search* tree on the other hand implies that the initial sequence of nodes will appear with preserved alphabetic order as leaf nodes of tree but the internal nodes of the tree will also contain the required information so that a binary search procedure can be used to locate an arbitrary node from the initial sequence in the crown of the tree.

1.1.3 Optimal Binary Trees

A level l_i of a node i with weight w_i is defined to be the distance of the node from the root of the tree. See Figure 1.1. The root of the tree is at level 0. The recursive definition of a node's level is 1 plus the level of its parent. A cost of a tree T is defined to be the sum of the weighted path length of its terminal nodes, denoted as $|T| = \sum_{i=1}^{n}(w_i l_i)$. An optimal binary tree is a binary tree whose cost is minimum and it is quite easy to prove that such a tree must be full.

1.1.4 Prefix-free Binary Encoding

A binary tree can define a variable length code. Each left branch of the tree is assigned the value 0 and each right branch is assigned 1. The letters of the alphabet that we encode are the leaf nodes of the tree. The codewords

are composed by traversing the tree from the root to the desired leaf node. The resulting binary code is a prefix code where no codeword is a prefix of some other codeword; this is called the prefix-free property of the code. The decoding process, when prefix-free code is used, is unambiguous. A given code can be decoded by simply traversing the tree from the root to the leaves by following the path of the code: if we read 0 from the code sequence we take the left branch, right otherwise.

1.1.5 Optimal Binary Trees vs. Optimal Alphabetic Binary Search Trees

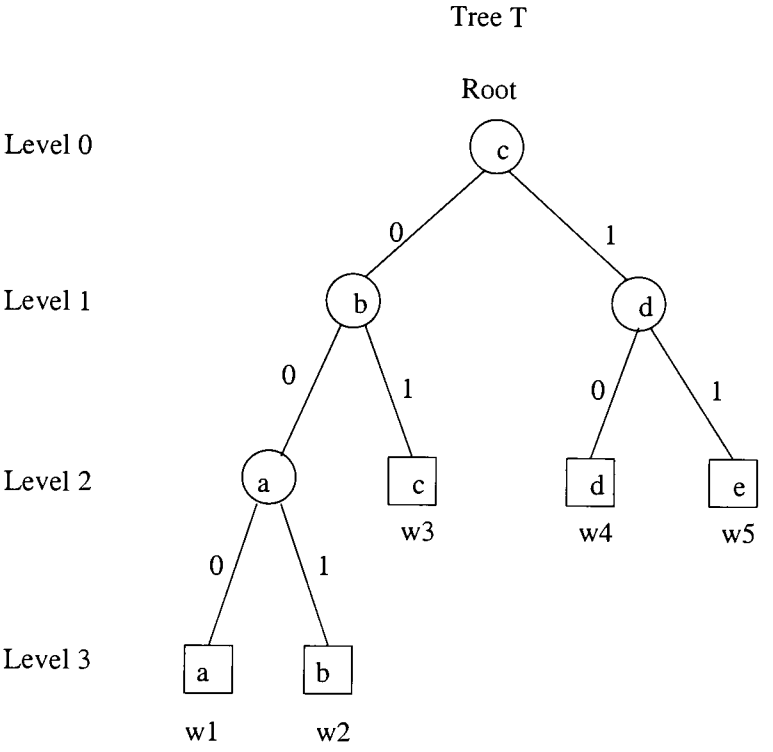
An optimal alphabetic tree built on a given initial sequence of nodes is a tree whose cost is minimum and the leaf nodes of the tree preserve the alphabetic order of the initial sequence. An optimal alphabetic binary search tree can be obtained from an existing optimal alphabetic binary tree by embedding information supporting the binary search procedure in the internal nodes of the tree. Thus, as we will see later, the cost of an optimal alphabetic binary tree and an optimal alphabetic binary search tree is the same: the level assignments of the terminal nodes is preserved, both trees obey the alphabetic order of the initial sequence, but an optimal alphabetic binary search tree has a constraint on the key values of the internal nodes according to the order of the initial sequence. A given leaf node (a letter from the alphabet) of the optimal alphabetic binary search tree can be located using a binary search procedure. The latter does not hold for optimal alphabetic tree. Algorithms for building optimal alphabetic trees can be used to build optimal alphabetic binary search trees.

Let BT denote the set of all binary trees, ABT - the set of all alphabetic binary trees, OBT the set of all optimal binary trees, and OABT - the set of all optimal alphabetic binary trees. OABST denotes a special case of the OABT. The cost of an OBT built on a given initial sequence of nodes can be lower than the cost of an OABST built on the same sequence.

$$BT \supset ABT \supset OABT; OABT \approx OABST$$
$$BT \supset OBT$$

OBT consists of larger set of trees and thus achieve a better optimality than OABST. See Section 4.6 for experimental comparison of the optimality that the Huffman and the Hu-Tucker algorithms achieve.

Building an optimal binary tree on a given initial sequence is very simple. Huffman described a greedy, bottom-up algorithm, running in $O(n \lg n)$ time that builds an optimal binary tree. The algorithm is intuitive and elegant but the resulting tree can only be used in the decoding process. An additional mapping mechanism is needed to resolve the correspondence of the plaintext letter to its codeword. On the other hand optimal alphabetic binary search trees are sufficient for both the encoding and decoding process. Decoding (same for OBTs and OABSTs) is just a traversal of the tree from the root to the terminal nodes, the choice of which branch to be taken is determined by the code sequence: 0 left branch, 1 right branch. The encoding process, when OABST exists, is also simple - it is a binary search procedure for locating the needed letter of the alphabet.



Letter	Codeword	Level
a	000	3
b	001	3
c	01	2
d	10	2
e	11	2

$$|T| = \sum_{i=1}^{5} (w_i l_i)$$

Figure 1.1: Extended alphabetic binary search tree and the corresponding prefix-free binary encoding.

1.2 History Overview

Optimal alphabetic binary trees have been studied for a long time. E. N. Gilbert and E. F. Moore first devised an algorithm for building such trees with running time $O(n^3)$ in 1959. Later, in 1962, Knuth constructed another algorithm, improving the running time to $O(n^2)$. The best time complexity, $O(n \lg n)$, for building optimal alphabetic binary trees is achieved by two algorithms: Hu-Tucker and Garsia-Wachs. T. C. Hu, jointly with A. C. Tucker. published for the first time in 1971 description of the algorithm in [HT71]. A. M. Garsia and M. L. Wachs published their algorithm in [GW77]. The two algorithms are quite similar. Both algorithms consist of three phases. During the first phase they construct an optimal tree, which is not alphabetic. The levels of the terminal nodes in the optimal tree are used to reconstruct another tree which is alphabetic, thus preserving the initial order of the terminal nodes, and also the cost of the tree constructed during the first phase. The two algorithms differ in a way they build the tree in the first phase. The reconstruction phase is the same. The Hu-Tucker algorithm was implemented by J. M. Yohe in Algol 60. Publication of the implementation appears in [Yoh72]. The running time he achieved is $O(n^2)$. The Garsia-Wachs algorithm was implemented by R. E. Tarjan in $O(n \lg n)$ time and can be found in [GW77].

B. M. Mumey introduced the idea for finding optimal alphabetic binary tree in linear time for some special classes of inputs in [Mum92]. One example is a simple case solvable in $O(n)$ time when the values of the weights in the initial sequence are all within a factor of two. Mumey showed that the region-based method, described in [Mum92], exhibits $o(n \lg n)$ time performance for a significant variety of inputs. Linear time solutions were discovered for the following special cases: when the input sequence of nodes is sorted sequence, bitonic sequence, weights exponentially separated, and weights within a constant factor, see [LP98] for references.

A parallel construction of optimal alphabetic binary trees is presented by L. L. Larmore, T. M. Przytycka, and W. Rytter in [LPR93]. The algorithm is a parallelization of Garsia-Wachs algorithm and runs in $O(\lg^3 n)$ time with $n^2 \lg n$ processors. This thesis is concerned with the general case of building optimal alphabetic binary search tree, when the input sequence of nodes does not exhibit any special properties.

Chapter 2

Hu-Tucker Algorithm

2.1 Overview

This chapter will present a detailed description of the Hu-Tucker algorithm as it appears in [Hu82] and [Knu73b].

The Hu-Tucker algorithm consists of three phases: Combination, Level Assignment, and Reconstruction. During the first phase, Combination, the initial sequence of nodes is used to build an optimal tree, which is not necessary alphabetic. At this stage the minimality of the cost of the tree is achieved.

During the second stage, Level Assignment, the tree obtained in step one is traversed and the levels of the terminal nodes are calculated, beginning at the root of the tree, which is at level 0. At the end of stage two, the tree obtained in stage one is disregarded.

Stage three, Recombination, builds an optimal alphabetic binary tree bottom up, from the initial sequence of nodes and their corresponding level assignments produced at the end of stage two, by combining adjacent nodes which have the same levels (level by level construction). The resulting Hu-Tucker tree is an optimal alphabetic binary tree.

2.2 Detailed Description

2.2.1 Phase I, Combination

The goal of this phase is to build an optimal binary tree in which the levels of the terminal nodes of the tree are realizable by the level by level construction[1] used in phase three.

The initial sequence of nodes consists of terminal nodes. Each node is identified with its position in the sequence, its weight, and the type of node, which can be internal or external. During each iteration of this phase a new working sequence is produced by combining two nodes of the previous working sequence into one node which replaces the leftmost node of the two and removing the rightmost one from the current sequence. This process continues until only one node is left in the sequence, which is the root of the tree.

Special rules govern the choice of the nodes that are combined at every iteration. Let q_i and q_k be two nodes, and their corresponding weights are w_i, and w_k, and their positions in the sequence are i and k respectively. In order for q_i and q_k to be combined the following constraints must hold, [Knu73b]:

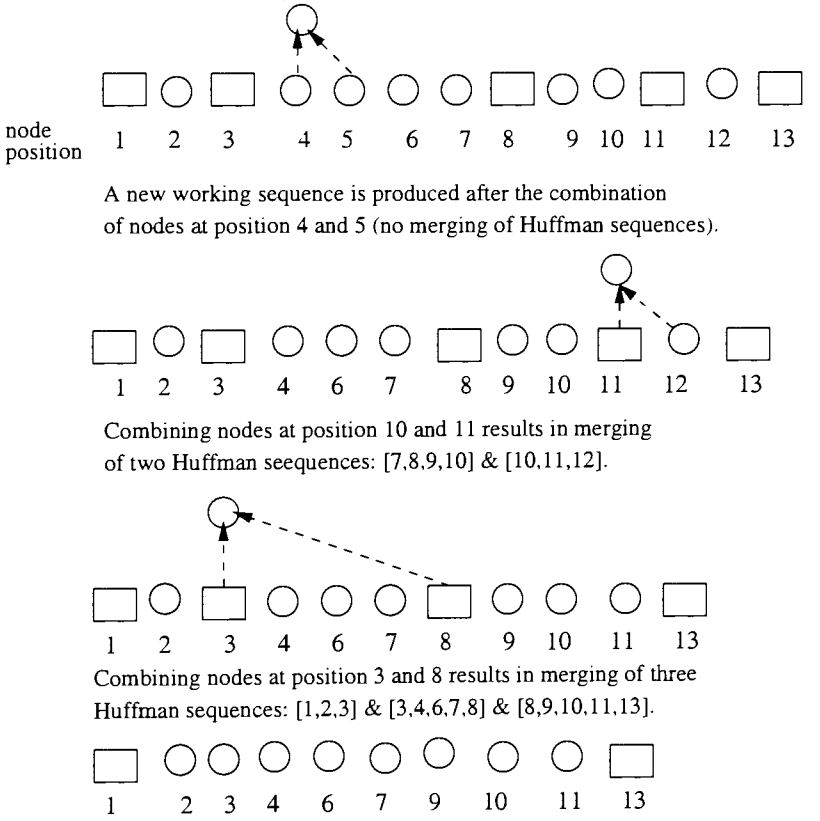
- In the current working sequence no external nodes occur between q_i and q_k.
- The combined weight $w = w_i + w_k$ is minimum among all the combined weights of all pairs of nodes between the two terminal nodes. In case of a tie, when more than one pair of nodes have minimum combined weight, the following is applied:
  - The pair with the leftmost left node is selected.
  - If they share the left node than the pair with the leftmost right node is selected.

The node produced as a result of the combination is of circular type. Its weight is equal to the sum of the weights of its children, and its position in the working sequence is the position of its left child.

[1] Sections 2.2.2 and 2.2.3 elaborate on level by level construction.

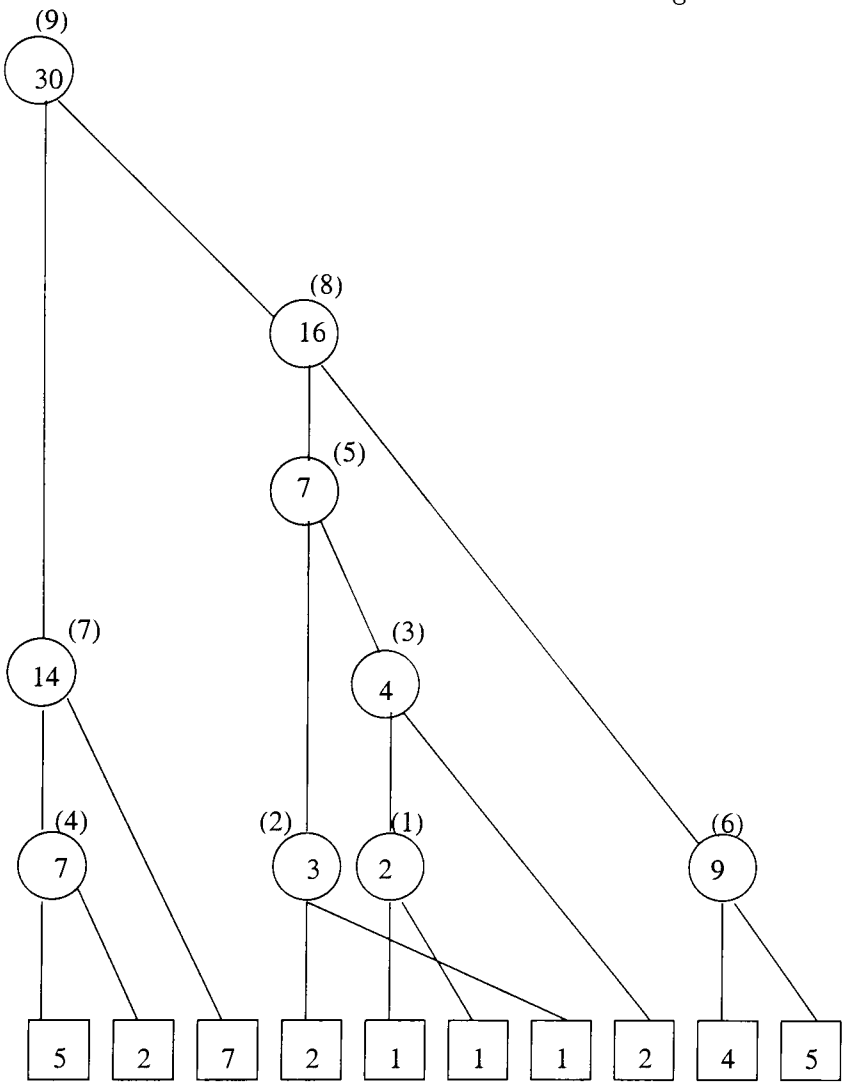
To explain the behavior of the Phase I, I define a Huffman sequence[2] to be a subsequence of a working sequence of nodes, delimited on the left and right by terminal nodes. Initially. the working sequence consists of terminal nodes only. Thus we have a sequence of $n-1$ Huffman sequences, each having two elements. During the Combination phase the number of Huffman sequences may reduce by one or two due to merging of adjacent sequences. The algorithm makes a greedy choice (ties are resolved in a deterministic manner) of combining nodes within a sequence. The combination process continues until there is only one node left in the working sequence - the root of the tree. Among all possible combinations the one which produces a node of minimum weight is performed.

Figure 2.1: Abstract representation of four working sequences and series of three combination steps.



[2]T. C. Hu and A. C. Tucker defined a Huffman set in [HT71]. The definition is the same. I call it "sequence" since the node's position is important.

Figure 2.2: Combination phase and the resulting Hu-Tucker tree.



The integers in parenthesis located above the internal nodes indicate the number of the step of the Combination phase at which the node is produced. For example: the internal node with weight 2 is produced at the first iteration. The root of the tree, internal node with weight 30, is produced at the ninth.

An alternate way to build the optimal tree in Phase I is by combining *local minimum compatible pairs* (l.c.m.p.). A pair of nodes is called to be a

compatible pair if the two nodes in it belong to the same Huffman sequence, i.e., no external nodes occur between them. A pair of nodes $\{q_i, q_j\}$ is a l.c.m.p. if the weight of the left node q_i is less than the weight of all nodes compatible with the right node q_j and the weight of the right node q_j is less than all the weights of the nodes compatible with the left node q_j. Initially, when all the nodes in the sequence are terminal nodes, a pair of nodes $\{q_{i-1}, q_i\}$ is an l.c.m.p. if

- $w_{i-2} > w_i$, and
- $w_{i-1} \leq w_{i+1}$

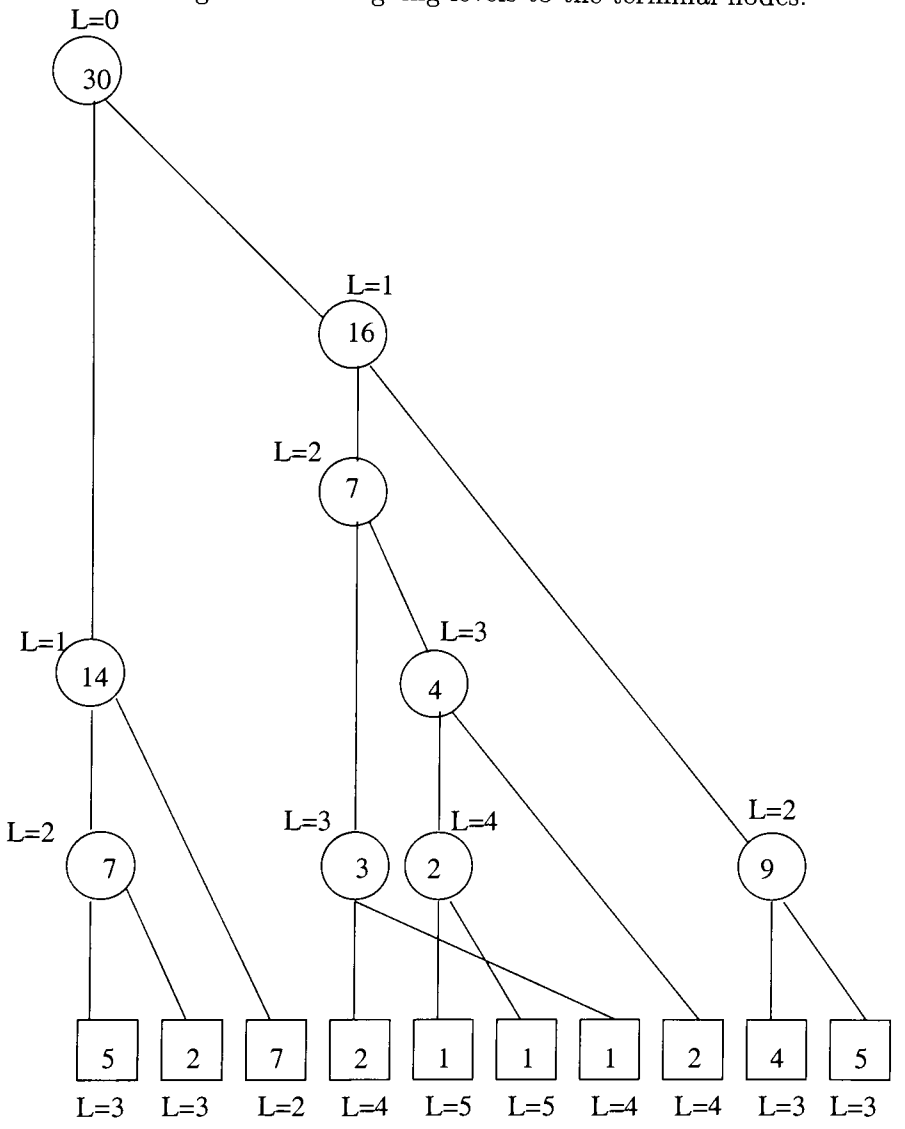
T. C. Hu proved that a l.c.m.p. in a working sequence preserves its properties after subsequent combinations of other local minimum compatible pairs, i.e., it remains as an l.c.m.p.[3] This implies that the order of combining local minimum compatible pairs does not change the tree built in this phase. The tree is unique and is not dependent on the order in which the local minimum compatible pairs are combined. Thus one can build the optimal tree by combining local minimum compatible pairs as they are discovered.

[3]Proof of this Lemma can be found in [Hu82], page 179.

2.2.2 Phase II Level Assignment

This stage determines the levels of the terminal nodes of the tree created in Phase I.

Figure 2.3: Assigning levels to the terminal nodes.



The calculated levels of the terminal nodes are realizable by level by level construction which will produce an alphabetic binary tree. The initial sequence of nodes with the assigned levels, produced in Phase II, is called a *feasible sequence.* T. C. Hu gave the following definition of a feasible sequence in [Hu71]: *A sequence of* **n** *positive integers is feasible if there exists a binary tree with* **n** *terminal nodes with path length corresponding to these* **n** *positive integers from the left to right.*

Lemma *A finite sequence of positive integers is a feasible sequence if and only if the following three conditions are satisfied:*

(i) If the largest level in the sequence is l, then there must be an even number of l's and such l's always occur in consecutive sets of even length.

(ii) If we form a reduced sequence from the original sequence by successively replacing from left to right every two consecutive l's by one occurrence of the integer $l-1$, then the reduced sequence again satisfies (i).

(iii) If the process of (ii) is repeated by considering the reduced sequence as the original sequence, (i) is still satisfied until finally a reduced sequence $1, 1$ is formed.

Proof:[HT71], page 517.

Note that the sequence of levels: $3, 3, 2, 4, 5, 5, 4, 4, 3, 3$, produced in Figure 2.3 is a feasible sequence.

2.2.3 Phase III, Recombination

This phase builds an optimal alphabetic tree from the initial sequence of terminal nodes and their levels calculated in Phase I. Pair of adjacent nodes which are at the same maximum level are combined to produce a node at level $l-1$.

Level by level construction of the tree is achieved by combining nodes q_i and q_{i+1} following the rules:

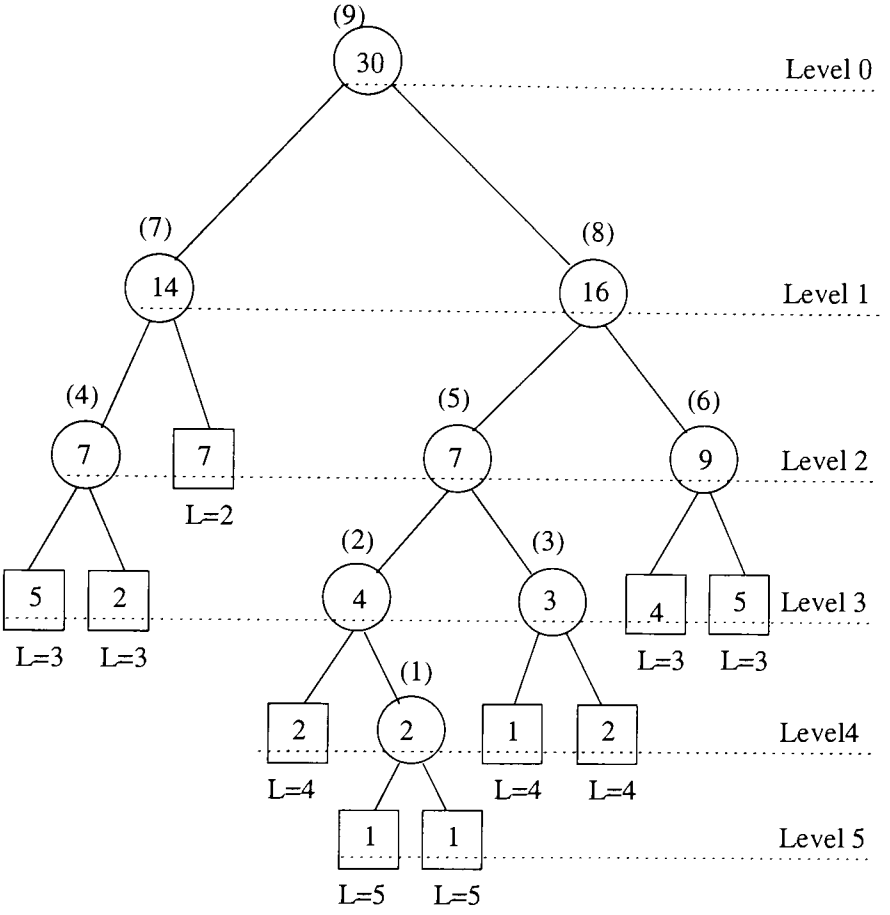
(i) q_i and q_{i+1} must be adjacent nodes in the sequence.

(ii) The levels of q_i and q_{i+1} must be the maximum among all the levels of the remaining nodes.

(iii) If q_i and q_{i+1} are at level l then q_i must be the leftmost node satisfying (i) and (ii).

Nodes at the lowest level are combined first, then the nodes on the next-to-lowest level are combined, etc. The process continues until there is only one node left and its level must be 0.

Figure 2.4: Level by level construction of alphabetic tree.



Note that the cost of the resulting tree is the same as the cost of the tree obtained at the end of the Combination phase. The tree constructed is alphabetic and optimal. The integers in the parenthesis above the circular nodes indicate the number of the iteration of the Reconstruction phase at which the node was created. Thus, circular node with weight 2 is constructed at the first step, circular node with weight 4 at the second, circular node with weight 3 at the third, etc.

The Stack algorithm, described by T. C. Hu in [Hu82] can also be used

to reconstruct the optimal alphabetic binary tree. This algorithm resembles the construction of the tree in the Phase I of the algorithm by combining local minimum compatible pairs as they are discovered. In the same fashion leftmost pairs of adjacent nodes with equal level are combined to form a node of higher level (the root of the tree has the highest level) as follows.

The algorithm uses a queue and a stack. Initially all the nodes with their levels are in the queue and the stack is empty. In subsequent steps the algorithm performs a check to see if the two elements at the top of the stack have equal levels. If they are equal then the two nodes are popped from the stack and combined to form a new node with a higher[4] level. For example, if the top of the stack has elements with levels $(l+1),(l+1)$, after the combination, the top of the stack will have an element l. If the two elements on the top of the stack are not equal then the element from the head of the queue is extracted and pushed onto the stack. This process continues until the queue is empty and the stack contains only one element with level 0.

Here is an example of the Stack algorithm applied on the sequence of nodes from Figure 2.3.

1. Queue content: $3,3,2,4,5,5,4,4,3,3$
   Stack content: empty
2. Queue content: $3,2,4,5,5,4,4,3,3$
   Stack content: 3
3. Queue content: $2,4,5,5,4,4,3,3$
   Stack content: $3,3$
4. Queue content: $2,4,5,5,4,4,3,3$
   Stack content: 2
5. Queue content: $4,5,5,4,4,3,3$
   Stack content: $2,2$
6. Queue content: $4,5,5,4,4,3,3$
   Stack content: 1
7. Queue content: $5,5,4,4,3,3$
   Stack content: $1,4$

[4]Higher level is closer to the root of the tree, which is at the highest level, and also has a lowest level value.

8. Queue content: $5, 4, 4, 3, 3$
   Stack content: $1, 4, 5$

9. Queue content: $4, 4, 3, 3$
   Stack content: $1, 4, 5. 5$

10. Queue content: $4. 4. 3, 3$
    Stack content: $1. 4. 4$

11. Queue content: $4. 4. 3. 3$
    Stack content: $1, 3$

12. Queue content: $4, 3, 3$
    Stack content: $1, 3, 4$

13. Queue content: $3. 3$
    Stack content: $1. 3, 4, 4$

14. Queue content: $3, 3$
    Stack content: $1, 3. 3$

15. Queue content: $3. 3$
    Stack content: $1, 2$

16. Queue content: 3
    Stack content: $1, 2, 3$

17. Queue content: empty
    Stack content: $1, 2, 3, 3$

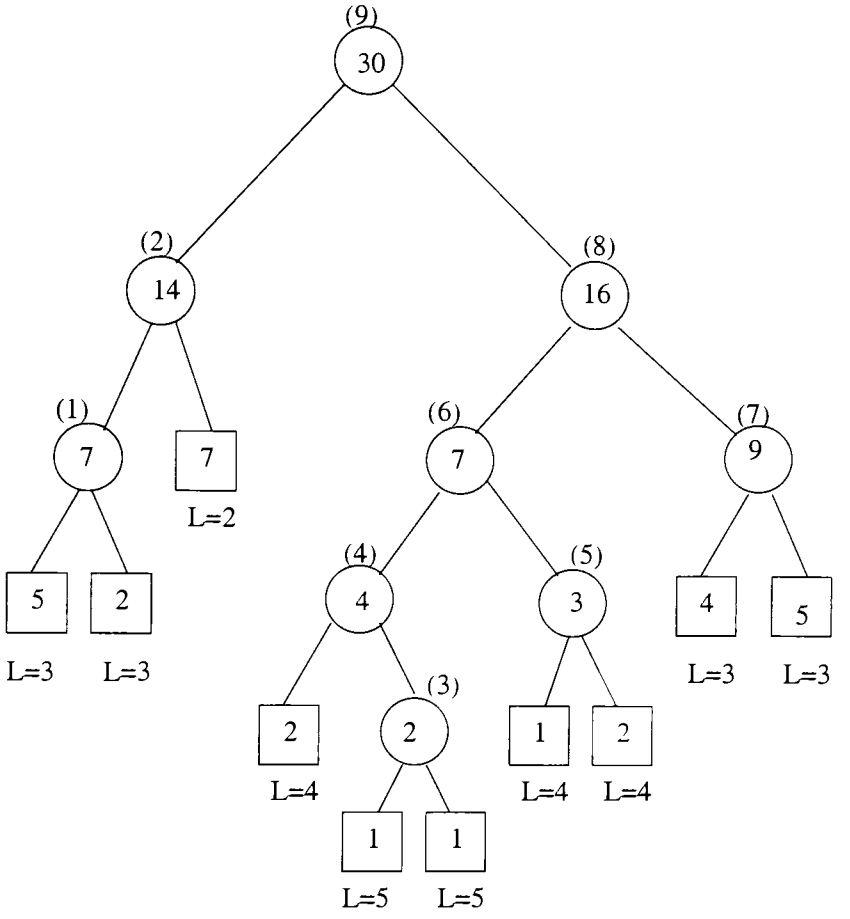
18. Queue content: empty
    Stack content: $1, 2, 2$

19. Queue content: empty
    Stack content: $1, 1$

20. Queue content: empty
    Stack content: 0

21. END.

Figure 2.5: Reconstruction phase, using the Stack Algorithm.



Note that the internal nodes are created in different order in comparison with the level by level construction. For example, the internal node with weight 14 is created at the seventh stage in the first case and during the second stage in the Stack algorithm. The result tree produced using the Stack algorithm is the same as the one constructed by combining the leftmost adjacent pairs at the lowest level first. The two algorithms achieve different time complexities. The Stack algorithm can be implemented in $O(n)$ time while the level by level reconstruction in $O(n \lg n)$ time.

2.2.4 Correctness

Proof of correctness of the Hu-Tucker algorithm was published for the first time in 1971 in [HT71]. Later, in 1973, T. C. Hu published a new proof of the algorithm in [Hu71]. In 1981 T. C. Hu published an augmented and modified version of the original proof in [Hu82]. In 1995, due to the similarities of the Hu-Tucker and Garsia-Wachs algorithms, Kaprincki, Larmore, and Rytter came with a different proof of the two algorithms, see [KLR].

All of the existing proofs are very long and involved.

Chapter 3

Implementation Models and Complexity Evaluation

The goal of this chapter is to build implementation model for the Hu-Tucker algorithm. The complexity of the algorithm depends on the complexities of the priority queue operations and complexity of sorting, see [LP98][1] for details on the OABT problem. The Hu-Tucker algorithm can solve the general OABT problem in $O(n \lg n)$ time and $O(n)$ space if proper data structures are used, where n is the number of elements in the input sequence.

3.1 Phase I

3.1.1 Fast Implementation Model

As explained in the previous chapter, Phase I of the Hu-Tucker algorithm combines compatible nodes from the initial sequence of weights until only one node is left. Two or three Huffman sequences could be merged as a result of the combination. The implementation model that I will build was suggested by Knuth in [Knu73b], page 444.

Let the initial sequence contain N terminal nodes. Two abstract data structures will be needed to describe the engine of the algorithm in Phase I: an array and a priority queue. The engine components are: two

[1] This publication is a good reference for linear time solutions for special cases of input sequences.

arrays $T[N]$ and $A[N]$; one Master Priority Queue (MPQ); and $N-1$ Huffman Sequence Priority Queues, also referred as Huffman Priority Queues (HPQ).
$T[N]$ is an array holding the initial sequence of terminal nodes. Each element of the array $T[N]$ must at least keep:

- The letter of the alphabet.
- The type of the element: terminal or internal node.
- The weight[2] of the letter.
- Each terminal node should also know the two priority queues in which it participates, i.e., each terminal node should be able to find in $O(1)$ time those elements of the MPQ, which refer to priority queues holding the two Huffman sequences which are delimited by this terminal node. See Figure 3.1.

The second array, $A[N]$, represents the current working sequence. Each element of $A[N]$ holds the address of the corresponding node of the working sequence. Initially all the nodes in the working sequence are terminal nodes. Thus the elements of $A[N]$ hold the corresponding addresses of the terminal nodes. As the Combination phase is in progress $A[N]$ will point to both terminal and internal nodes. The number of nodes in the working sequence steadily decreases. Thus some of the elements of $A[N]$ will become *null*. Initially the relationship between $A[N]$ and $T[N]$ is as follows:

$$A[1] = address\_of(T[1]);$$
$$A[2] = address\_of(T[2]);$$
$$\vdots$$
$$A[n] = address\_of(T(n));$$

For example, assume that the weight distribution of the nodes in the initial sequence is such that the first combination that ought to be performed is the pair $T[1]$ and $T[2]$. After the combination the working sequence has changed: the terminal nodes $T[1]$ and $T[2]$ no longer participate in, thus the array $A[N]$ has changed its state:

$$A[1] = address\_of(I[1]), \qquad I[1] \quad \text{new node;}$$
$$A[2] = null.$$

The rest of the elements of the array remain unchanged. See Figure 3.2.

[2]The weight is a representation of the frequency of usage of the letter in the alphabet.

The HPQs hold the elements of the Huffman sequences at any stage of the Phase I of the algorithm. Each individual priority queue represents one Huffman sequence. Initially there are $N-1$ HPQs, and each of them holds two adjacent terminal nodes from the initial sequence. Let n denotes the number of nodes in the queue. The priority queue data structure representing the HPQ must be able to perform the following operations:

- Report in $O(\lg n)$ time the two least elements of the priority queue and their positions in the working sequence.
- It must be able to find in $O(\lg n)$ time the two terminal nodes, which define the boundary of the Huffman sequence it points to.
- Delete the minimum and arbitrary element of the priority queue in $O(\lg n)$ time.
- Merge two HPQs in $O(\lg n)$ time.

The MPQ has as many elements as the current number of Huffman sequences, i.e., the number of the HPQs. The MPQ is responsible to choose a compatible pair from the current working sequence according to the rules stated in Section 2.2.1.
Each element of the MPQ must at least hold the sum of the two least elements from each individual HPQ, and their positions[3], designated as $MinSum$, i and j, on Figures 3.1 and 3.2.
Each element of the MPQ has a reference to the root of the HPQ it represents. The elements of the MPQ should not change their physical locations, since each terminal node from the initial sequence has a reference to those elements of the MPQ, that are representing the Huffman sequences to which the terminal node participate. If a terminal node is to be combined with another node, the two Huffman sequences it belongs to will be merged, thus those elements of the MPQ must be deleted, and the corresponding priority queues merged. The merge will generate a new Huffman sequence. The node representing it will be inserted to the MPQ. All the priority queue operations must be performed in at most $O(1)$ or $O(\lg n)$ time to achieve a reasonable performance.

[3]This information is needed to resolve ties. See Section 2.2.1.

Figure 3.1: The engine of the Hu-Tucker algorithm for Phase I. Initial State.

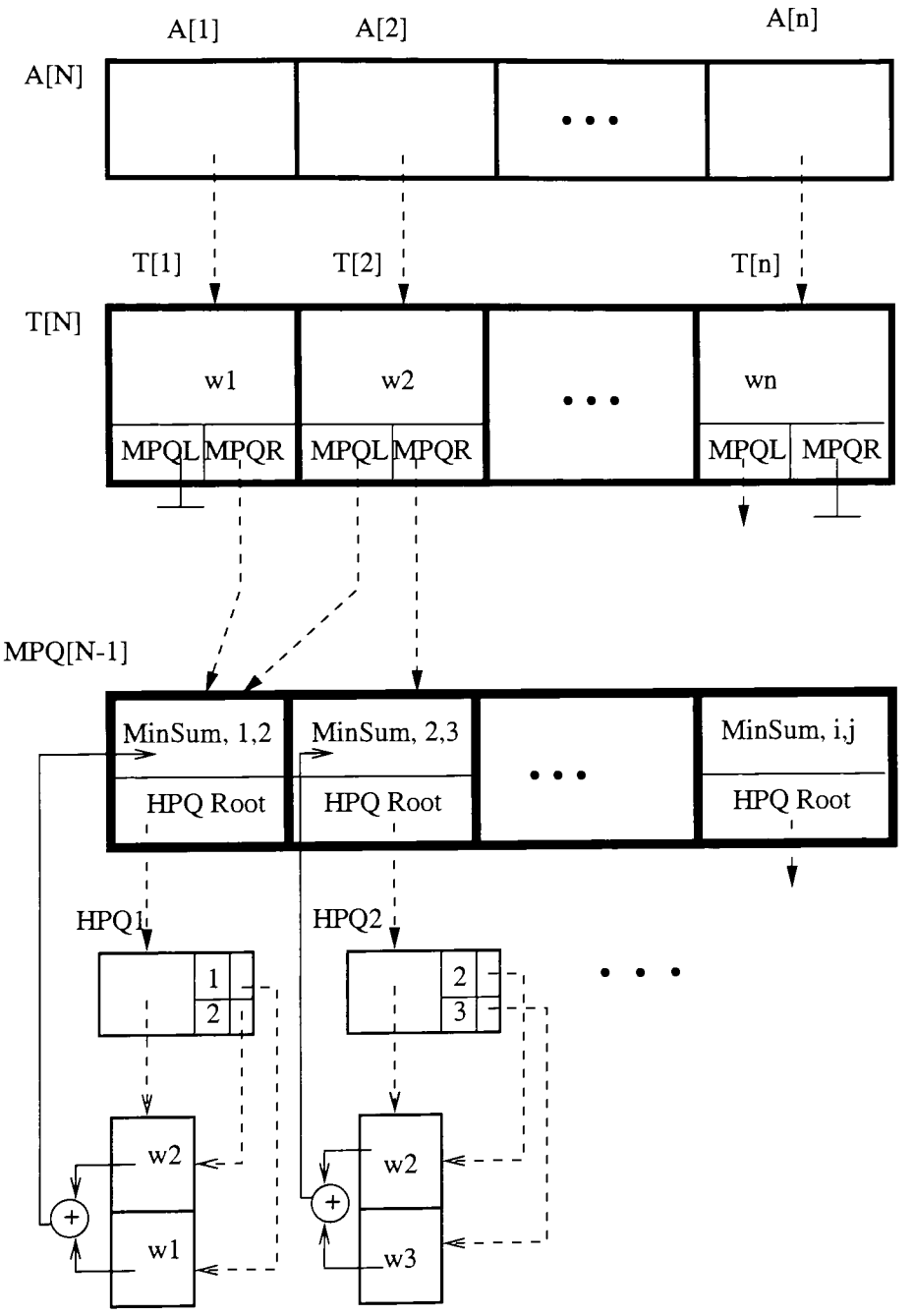
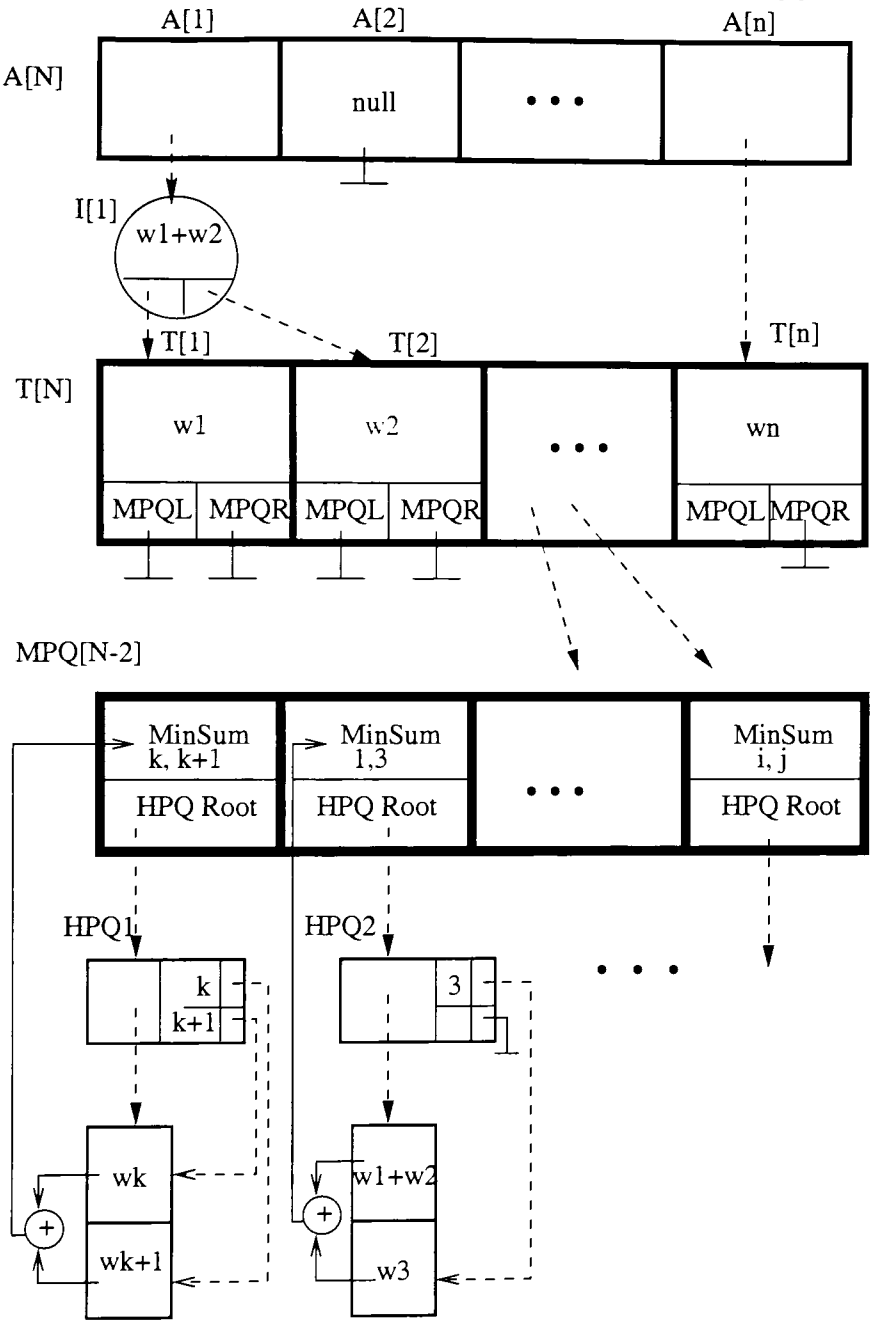


Figure 3.2: The state of the engine after combining nodes $T[1]$ and $T[2]$.



To summarize the MPQ must provide the following operations:

1. Insert an element to the queue in $O(\lg n)$ time.
2. Delete the minimum element of the queue in $O(1)$ time.
3. Delete an arbitrary element of the queue in $O(\lg n)$ time.
4. The physical locations of the priority queue elements should not change.

Figures 3.1 and 3.2 present the engine of the algorithm in Phase I. For simplicity the author has assumed that the first three weights of the initial sequence are the smallest. Thus the HPQ representing the Huffman sequence $(\mathbf{w_1}, \mathbf{w_2})$ has the smallest combined sum and as a result the MPQ's element representing it is on the top of the MPQ. Figure 3.2 illustrates the state after the first combination is performed. Note the change of the state of all components of the engine after the combination.

Figure 3.3 represents an example of Phase I of the Hu-Tucker algorithm applied on the following sequence of weights: $\mathbf{10, 2, 2, 1, 1, 4. 15, 17, 25}$, after performing three combinations. Following is a detailed description of the first three iterations of the algorithm. The MPQ has selected the pair of nodes $T[4]$ and $T[5]$ to be combined first. The internal node $I[1]$ is produced as a result of the combination. Also, three priority queues are merged into one and the duplicate terminal nodes participating are deleted. In detail: $(\mathbf{T[3], T[4]}), (\mathbf{T[4], T[5]})$, and $(\mathbf{T[5], T[6]})$ priority queues have to be merged. The elements $T[4]$ and T[5] are deleted from all three priority queues prior to the merge. After the clean up followed by the merge the new priority queue produced has the elements $(\mathbf{T[3], I[2], T[6]})$, where $I[2] = combine(T[4], T[5])$. The MPQ is updated: three elements are deleted and one is inserted. The state of the array $A[N]$ has changed as well: $A[4]$ points to $I[1]$ and $A[5]$ is *null*. At the second iteration of the algorithm the terminal nodes $T[2]$ and $T[3]$ are selected. The combination produces internal node I[2]. $A[2]$ is redirected to point to $I[2]$, and $A[3]$ is *null*. Again, two terminal nodes are combined and as a result three priority queues have to be merged into one. Clean up of the duplicate nodes is performed. Three elements of the MPQ are deleted and one is inserted. At the third step the internal node $I[1]$ is combined with the terminal node $T[6]$. Two priority queues are merged into one: $(\mathbf{T[1], I[2], I[3], T[6]})$ and $(\mathbf{T[6], T[7]})$. Terminal node $T[6]$ is deleted from both of the priority queues prior to the merge. The new internal node $I[3]$ is inserted to the new priority queue. The MPQ and the array $A[N]$ are updated.

In total $N-1 = 9-1 = 8$ combination steps are performed, and only one element is left in the working sequence which is the root of the tree. Thus only the first element of the array $A[N]$ is not *null*. The tree is built and the Combination phase has terminated. Figure 3.4 illustrates the engine of the algorithm in its final state.

Figure 3.3: The engine of the Hu-Tucker algorithm in progress.

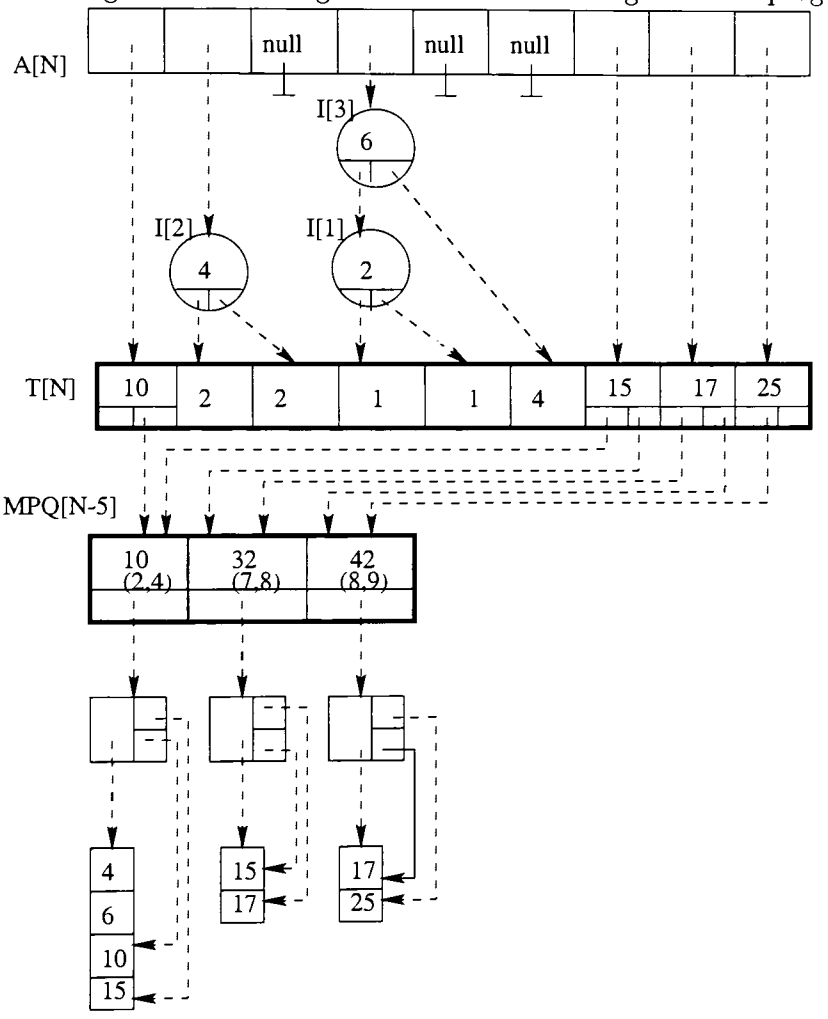
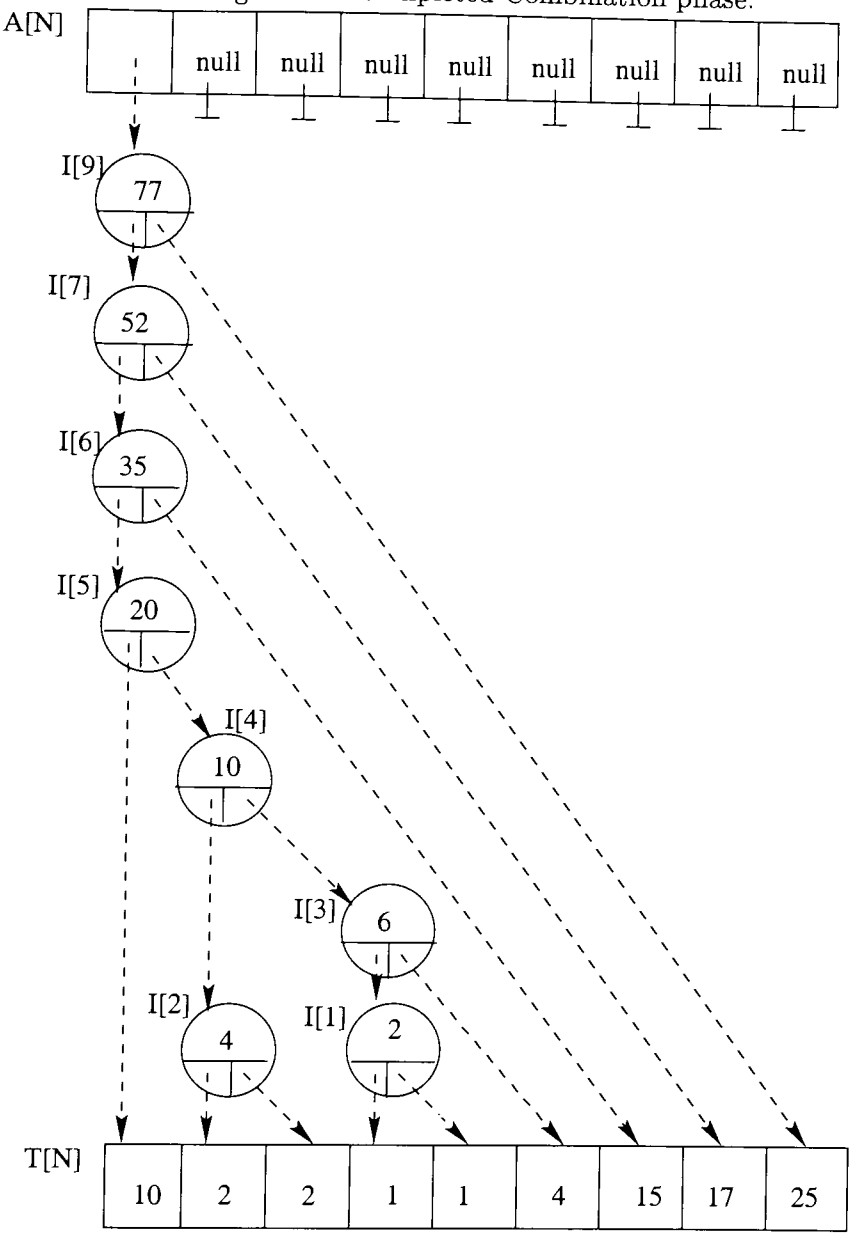


Figure 3.4: Completed Combination phase.



Hu-Tucker Algorithm Phase I, Combination

Initialization

Populate the array $T[N]$ and initialize $A[N]$

Create and initialize $(N-1)$ $HPQs$ and the MPQ

End:Initialization

Main Loop

for (k=1; $k<N$; k++)

Extract_Min(MPQ): get the two nodes of the working sequence, which are on the top of MPQ and will have to be combined.

Delete those two nodes from all $HPQs$ where they participate

Merge all those $HPQs$ into a new HPQ_NEW, if necessary

Update the MPQ: delete the merged $HPQs$ from MPQ

Combine the two nodes into a *new node*

Update the array $A[N]$

Insert the *new node* into the HPQ_NEW

Insert the HPW_NEW to MPQ

End: Main Loop

End: Phase I Completed

3.1.2 Fast Implementation Time Complexity Evaluation

The time complexity $T(n)$ of the implementation model described in Section 3.1.1 depends on the time complexity of the priority queue operations. The pseudo code of the algorithm is available in Appendix A, Section A.1. Let n denotes the number of elements in a priority queue. The following notation is used:

- $PQInsert(n)$ is the function describing the asymptotic time complexity of the insert an element operation of the priority queue.
- $PQDeleteMin(n)$ is the function describing the asymptotic time complexity of the delete minimum element operation of a priority queue.
- $PQDelete(n)$ is the function describing the asymptotic time complexity of the delete an arbitrary element operation of a priority queue.
- $PQMerge(n)$ is the function describing the asymptotic time complexity of the operation merging two priority queues.

The time complexity of the initialization in Phase I is:

$$T(n)_{Initialization} = n + nPQInsert(n)$$

The most expensive scenario of the main loop of the Combination phase of the algorithm is the case of combining two terminal nodes:

$$T(n)_{MainLoop} \leq n(PQDeleteMin(n) + 4PQDelete(n) + 3PQMerge(n) + PQInsert + c_2)$$

If the priority queue data structure chosen supports operations with the asymptotic time $O(\lg n)$ then the time complexity of Phase I is:

$T(n) = T(n)_{Initialization} + T(n)_{MainLoop}$
$T(n) = c_1 n \lg n + c_2 n \lg n + c_3 n + c_4 \leq c_5 n \lg n$
$T(n) = O(n \lg n)$;

The units of space required by the data structures in Phase I are:

$$S(n) = 6n = O(n)$$

3.1.3 Implementation Model of Combination Phase With Local Minimum Compatible Pairs.

As I mentioned earlier in Chapter 2, Section 2.2.1, an alternate way for building the optimal tree in Phase I is by combining *local minimum compatible pairs*. This section will present the algorithm for finding l.c.m.p. and building the optimal binary tree, whose leaf levels can be realized by an OABT. The implementation model is simple. We need to represent the initial sequence of terminal nodes and the constantly changing working sequence of nodes. The abstraction representing the initial sequence is the array $T[N]$, having simpler responsibility then the one described in Section 3.1.1. Each element of the array needs to keep the following information about the node: position in the sequence, type of the node which can be internal or terminal, weight, letter of the alphabet, left and right tree descendants. The node descendants of terminal nodes are *null*. The abstraction representing the working sequence of nodes at any stage of the algorithm during Phase I can be an array or doubly linked list.

The algorithm simply searches for *l.m.c.p.* in the working sequence, scanning it from left to right, finds it and combines it. This process is repeated $N - 1$ times. At the end only one node remains in the working sequence. The algorithm inspects the working sequence one Huffman Sequence at a time. The two nodes with minimum weight are located. They form a true *l.m.c.p.* if the right node of the pair is internal. If the right node of the *potential l.m.c.p.* is a terminal node, the weights of all the nodes compatible

with it have to be compared to the weight of the left node of the *potential l.m.c.p.* by inspecting the right adjacent Huffman sequence. If the *potential l.m.c.p.* is a false *l.m.c.p.* then the search for *l.m.c.p.* resumes from the next Huffman sequence. The pseudo code of the algorithm is available in Appendix A, Section A.2.

The running time of this phase depends on the time units, T_{lmcp}, needed to locate a *local minimum compatible pair*. In the worse case scenario this will require a traversal of the entire working sequence, hence the number of comparisons that have to be performed to locate an *l.m.c.p.* is $O(n)$.

$T_{lmcp} \leq n$

$T(n)_{MainLoop} = (n-1)T_{lmcp}$

$T(n)_{Combination\_with\_lmcp} \leq (n-1)n = \mathbf{O(n^2)}$

3.2 Phase II

3.2.1 Implementation Model

Phase II calculates the levels of the terminal nodes of the tree built in Phase I. This is achieved by a simple inorder tree traversal procedure.

Hu-Tucker Algorithm Phase II, Level Assignment

Procedure**AssignLevels**(*TreeRoot*, *level*)
    TreeRoot.level = level;
    if TreeRoot.LeftChild exist
    then
        AssignLevels(TreeRoot.LeftChild, *level* + 1)
    if TreeRoot.RightChild exist
    then
        AssignLevels(TreeRoot.RightChild, *level* + 1)
End: Phase II Completed

The time complexity of this phase is described by the recurrence:

$T(n) = 2T(n/2) + k.$

The solution of this simple recurrence is $T(n) = O(n)$. The space complexity is $S(n) = 0$ because this phase does not need any additional data structures.

3.3 Phase III

3.3.1 Implementation Model

The implementation model of the Recombination phase will use the Stack Algorithm described in Chapter 2. Section 2.2.3. The initial sequence of terminal nodes and their level assignments, calculated at the end of phase II. will be used to construct the optimal alphabetic binary search tree.

Hu-Tucker Algorithm Phase III, Recombination

Initialize
$Q[N] = T[N]$
S=empty
Main Loop
while (!((Q is empty) and $((size\_of(S)) == 1)$)))
  if $((size\_of(S)) \geq 2)$ and (the top 2 elements have the same levels)
  then
    $level_{top} = level\_of\_element(pop(S))$
    Combine those two elements of the queue Q, whose levels
      are at the top of the stack.
    Remove the element from the top of the stack: $pop(S)$
    Remove the element from the top of the stack: $pop(S)$
    Push an element with $level = level_{top} - 1$ on the top of the stack:
      $push(S.level_{top} - 1)$
  else
    Remove an element from the front of the queue Q
      and push it on the top of the stack S: $push(S, pop(Q))$
End: Phase III Completed

Figures 3.5, 3.6, 3.7, and 3.8 illustrate the behavior of the algorithm with the example sequence of nodes used in Figures 3.3 and 3.4.

The time complexity of the third phase is
$T(n) = c_1 n T(\mathsf{push}) + c_2 n T(\mathsf{pop}) + c_3(n-1)T(\mathsf{combine}) + c_4$

The stack and queue operations take $O(1)$ time $\Longrightarrow$
$T(n) = c_1 n O(1) + c_2 n O(1) + c_3(n-1)O(1) + c_4$
$T(n) = c_5 n O(1) \leq c_6 n$
$T(n) = O(n)$

The space complexity is:
$S(n) = 3n = O(n)$

Figure 3.5: The engine of the algorithm during the Reconstruction sequence.

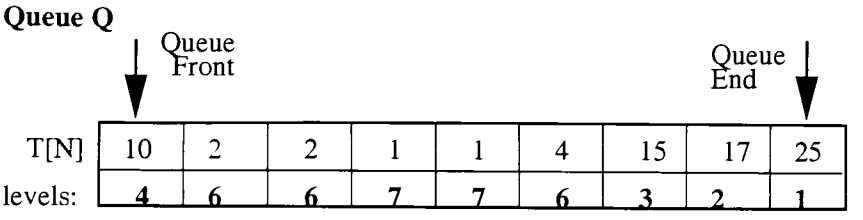


Figure 3.6: Reconstruction phase in progress. Three elements from the queue Q are pushed on the stack S.

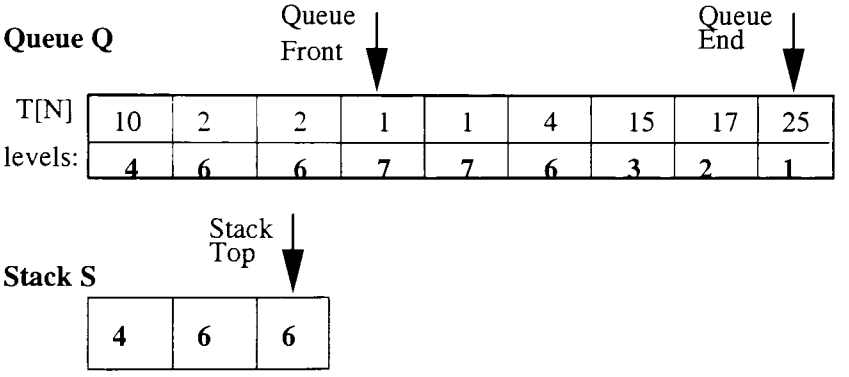


Figure 3.7: The top two elements of the stack are combined.

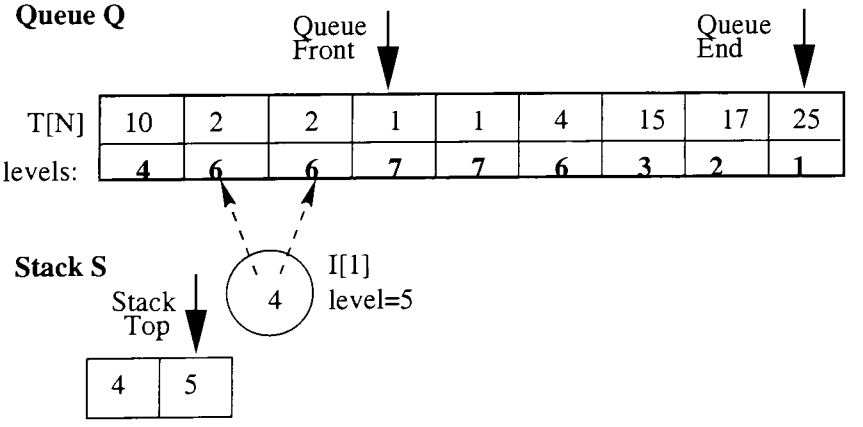
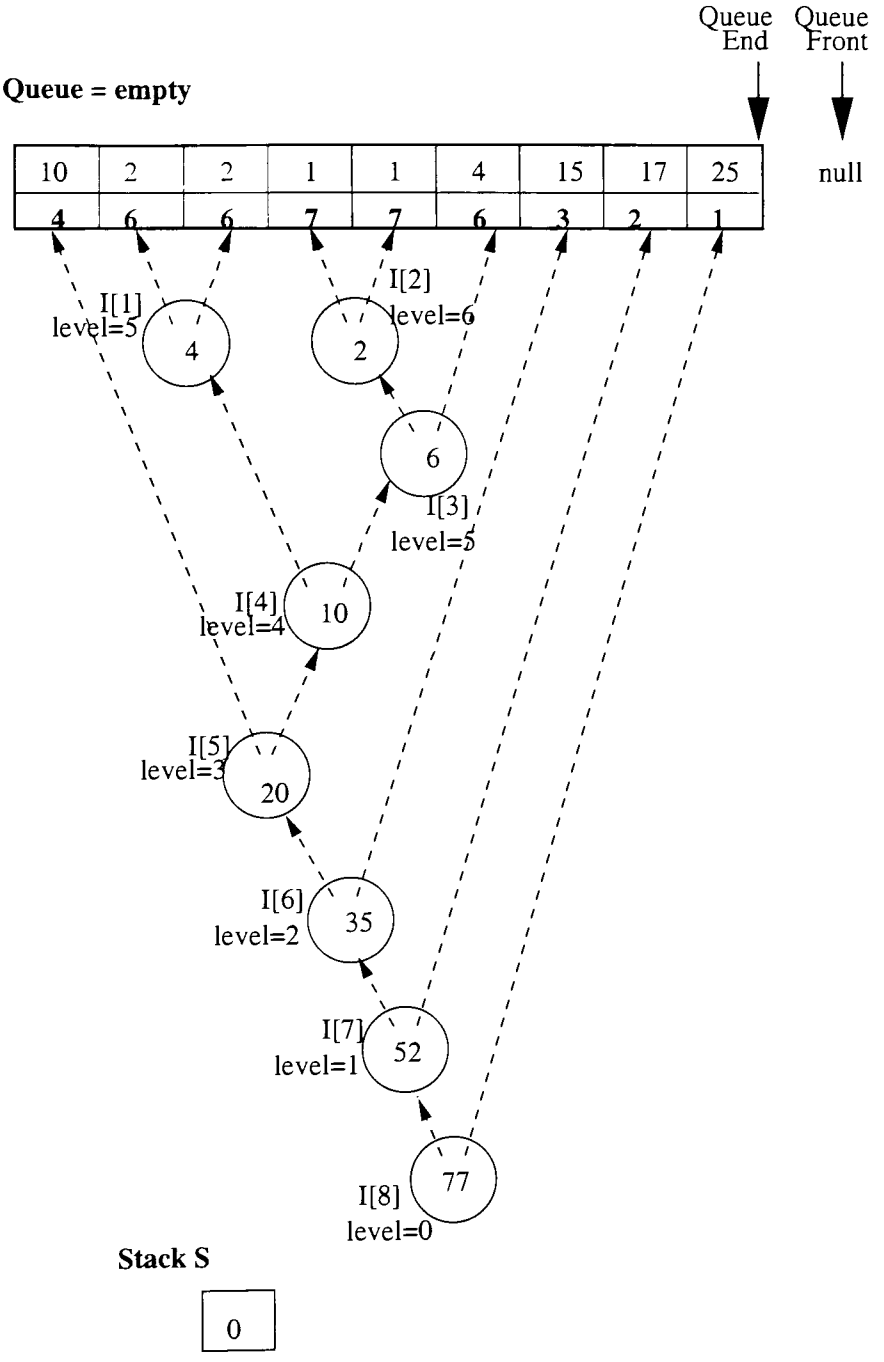


Figure 3.8: Completed Reconstruction phase.



3.4 Time and Space Complexity of the Hu-Tucker Algorithm

The time complexity of the algorithm is:

$T(n) = T_{Combination} + T_{Level\_Assignment} + T_{Reconstruction}$

When Phase I is implemented with the model described in Section 3.1.1

$T(n) = O(n \lg n) + O(n) + O(n)$

$T(n) = \mathbf{O(n \lg n)}$,

When Phase I is implemented with the model described in Section 3.1.2

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

$T(n) = \mathbf{O(n^2)}$,

The space complexity is:

$S(n) = S_{Combination} + S_{Level\_Assignment} + S_{Reconstruction}$

$S(n) = O(n) + O(n)$

$\mathbf{S(n) = O(n)}$

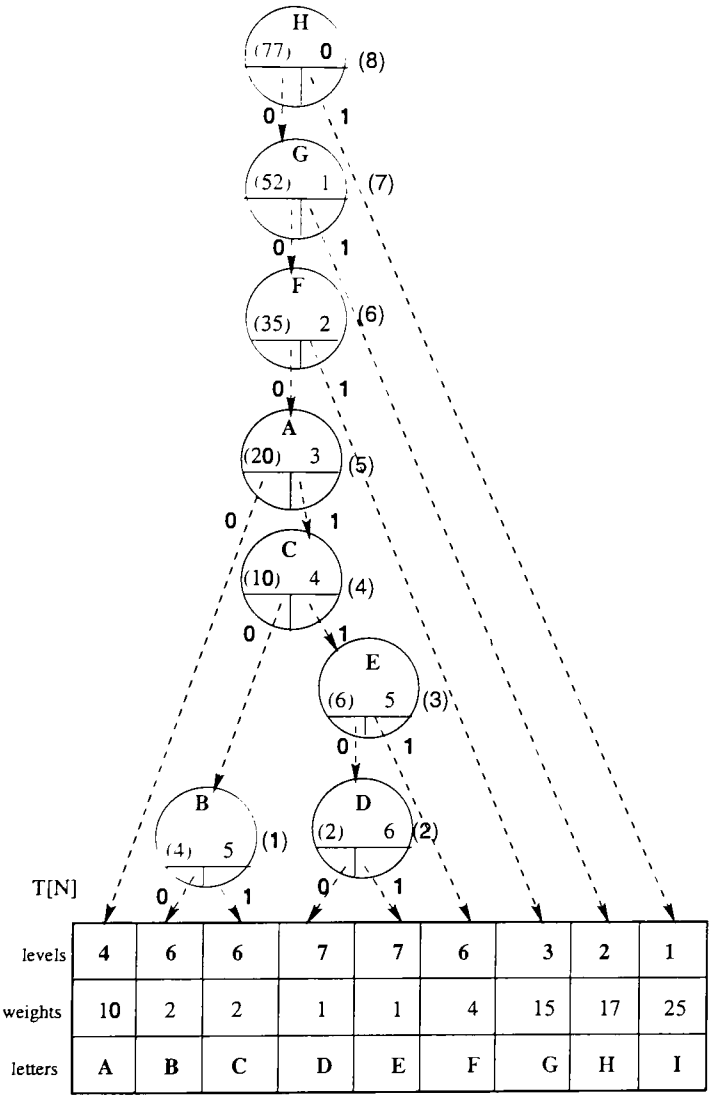
3.5 Building OABST

In this section the skeleton of the Stack Algorithm, described in Section 3.3.1 will be used to devise a bottom-up algorithm for building Optimal Alphabetic Binary Search Tree from a feasible sequence of nodes and their levels.

The internal nodes of the resulting tree have to support a binary search procedure for locating nodes of the initial sequence. Thus, an internal node must keep at least the following information: the letter of the alphabet, the weight of the node, the level of the node, and left and right node descendants. The algorithm proceeds exactly like the Stack Algorithm, the only difference occurs when the two elements from the top of the stack are popped, combined, and the newly created element is pushed onto the top of the *Stack*. The *letter* field of the newly created element is assigned the value of the *letter* field of it's left child, to support the binary search procedure. See Appendix B for pseudo code of the implementation of the algorithm. Figure 3.9 shows the resulting OABST built for the sequence of nodes used in Figure 3.5. The weights in the sequence, for the sake of the example,

represent the frequencies[4] of usage of the first nine letters on the English alphabet. The integers in parenthesis to the right of the circular nodes denote the number of the combination in the Reconstruction phase which produced the node.

Figure 3.9: The OABST built using the Stack Algorithm.



levels	4	6	6	7	7	6	3	2	1
weights	10	2	2	1	1	4	15	17	25
letters	A	B	C	D	E	F	G	H	I

[4]The real frequencies of usage of the letters of the English alphabet are different.

When encoding, every left branch of the tree is assigned a value of 0 and every right 1. The tree can be used to locate the letters from the crown (initial sequence) using binary search procedure. For any node of the tree all the nodes in the left subtree will have letter values of the predecessors in the original alphabetic sequence and the nodes in the right subtree will have letter values of the successors.
For example, let's find the encoding of the letter E. Starting at the Root of the tree:
$E < H \Rightarrow$ taking the left branch 0
$E < G \Rightarrow$ taking the left branch 0
$E < F \Rightarrow$ taking the left branch 0
$E > A \Rightarrow$ taking the right branch 1
$E > C \Rightarrow$ taking the right branch 1
$E \leq E \Rightarrow$ taking the right branch 0
$E > D \Rightarrow$ taking the right branch 1

The code word for the letter E, is 0001101

3.6 Mergable Heaps

For the implementation of the first phase of the Hu-Tucker algorithm, as described in Section 3.1.1, a priority queue data structure is needed. The binary heap invented by Williams is simple and elegant but fails to satisfy the requirement of the implementation model.

(1) The physical location of the members of the priority queue changes as elements are inserted and deleted.

(2) Merging two priority queues into one takes $O(n)$ time.

Several sophisticated data structures provide mechanisms for efficient merging. Leftist heaps, invented by Clark Allan Crane [Cra72], binomial heaps, invented by Jean Vuillemin [Vui78], relaxed heaps, described in [DGST88] all provide the basic operations: insertion of a new element (Insert), deletion of the element with the minimum priority ($DeleteMin$), report the element with minimum priority ($FindMin$), and merge two priority queues ($Merge$) in $O(\lg n)$ time.

Mark R. Brown analyzed the performance of binomial queue, leftist trees, linear lists, binary heaps, and sorted list in [Bro78]. His results showed that binomial queues outperform the rest of the priority queues for the $Merge$,

Insert, and *DeleteMin* operations. However, he points out that leftist heaps do have some advantages over binomial queues:

- Operations are easy to define and describe.
- Leftist heaps are easy to implement.
- Leftist heaps can take advantage of any tendency of insertion to follow a stack discipline.

His results did not compare the performance of the different priority queues for deletion of an arbitrary element of the queue.

3.6.1 Leftist Heaps

Donald E. Knuth was the first to suggest that the Hu-Tucker algorithm could be implemented using a leftist tree as a priority queue data structure. The following brief presentation of leftist heaps is based on the [Knu73b], and [Tar83]. A leftist heap is an extended binary tree. Every node of the leftist tree must at least have a *key* field, a *rank* field, and two pointers $left$, and *right* for the left and right subheaps or *null* when the element is a leaf node. The *rank* of a node is the shortest path from the node to a leaf node. For every node P of the leftist heap the following conditions must hold:

$P.key \leq (P.left).key$, $P.key \leq (P.right).key$;

$P.rank = 1 + min((P.left).rank, (P.right).rank)$;

$(P.left).rank \geq (P.right).rank$;

These conditions ensure that the leftist tree is a heap (the element with the minimum key is on the top of the heap), and that the shortest path to a leaf node is obtained by following the *right* link. Donald Knuth called this tree leftist, *"because it tends to "lean" so heavily to the left"*. So described leftist tree can handle *Insert*, *Merge*, and *DeleteMin* operation in $O(\lg n)$ time. To support delete of an arbitrary element of the tree (*DeleteElement*) and preserve the leftist property of the tree pointers to the parent of each node are required. *DeleteElement* operation also achieves $O(\lg n)$ time complexity.
An alternate way called *lazy deletion* for implementing deletion of an arbitrary element of the leftist tree was proposed by R. E. Tarjan and D. Cheriton in [CT76]. Their method requires an additional field which shows whether the node is deleted or not. Deletion simply marks the node as

deleted, thus requiring constant time. Whenever a $DeleteMin$ or $FindMin$ operation is required, a preorder traversal of the leftist tree is performed as needed, to discover the minimum element. All deleted elements during the traversal are physically deleted and their corresponding subtrees merged. The time complexity of $FindMin$ operation is: $T_{FindMin} = mO(\lg n)$, where m is the number of elements marked as deleted. Information on the *lazy delete* method can also be found in [Hor84] and [Tar83].

I have chosen to implement a leftist heap with parent pointers. All operations of the leftist tree are expressed using the $Merge$ operation. See Appendix C for pseudo code of the implementation.

Figure 3.10 shows a scenario of insertion of a new node in existing leftist tree. The insertion is implemented by a merge of a one-node leftist tree, $H2$, with the leftist tree, $H1$. In this particular case the insertion follows the stack discipline. Figures 3.11 and 3.12 show a merge of two leftist trees. The path of the merge is highlighted in bold.

An improvement of the performance of the merge operation can be achieved by implementing a non-recursive merge. The implementation needs to mimic the runtime stack and remember using additional data structure, the path of the merge, i.e., the points of the leftist tree where potential violation of the rank constraints may occur. After the merge is completed the implementation should follow back the merge path and correct the ranks and swap the left and right subheaps when needed.

Figure 3.10: Insertion of a node is a merge of a one-node heap with another heap.

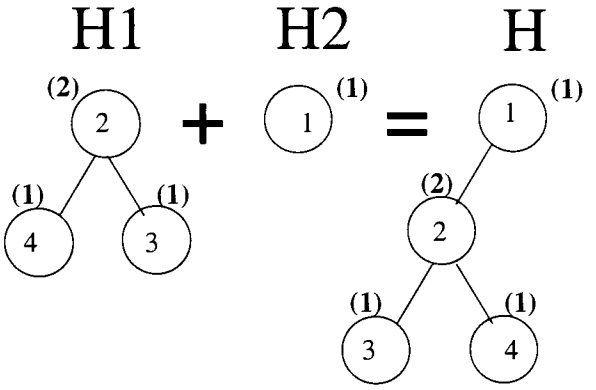


Figure 3.11: Two leftist heaps H1 and H2 prior to merging.

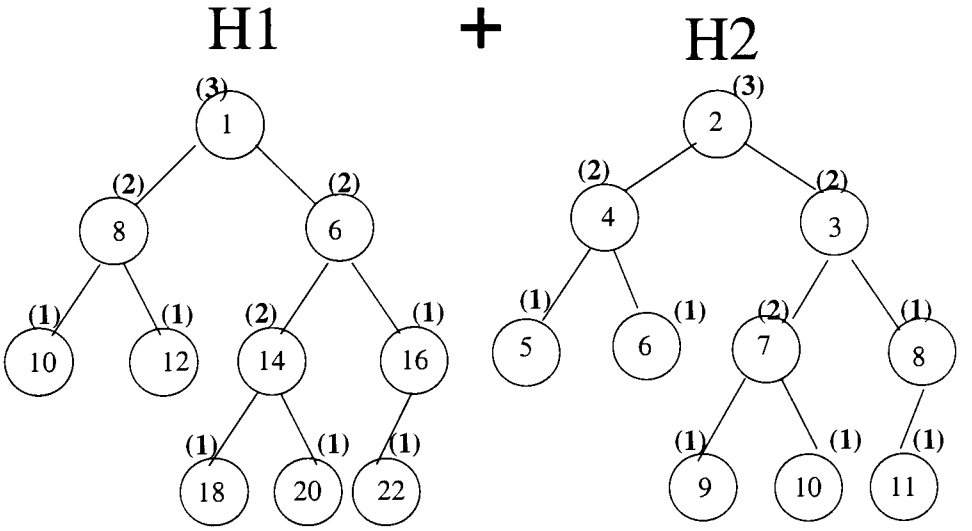
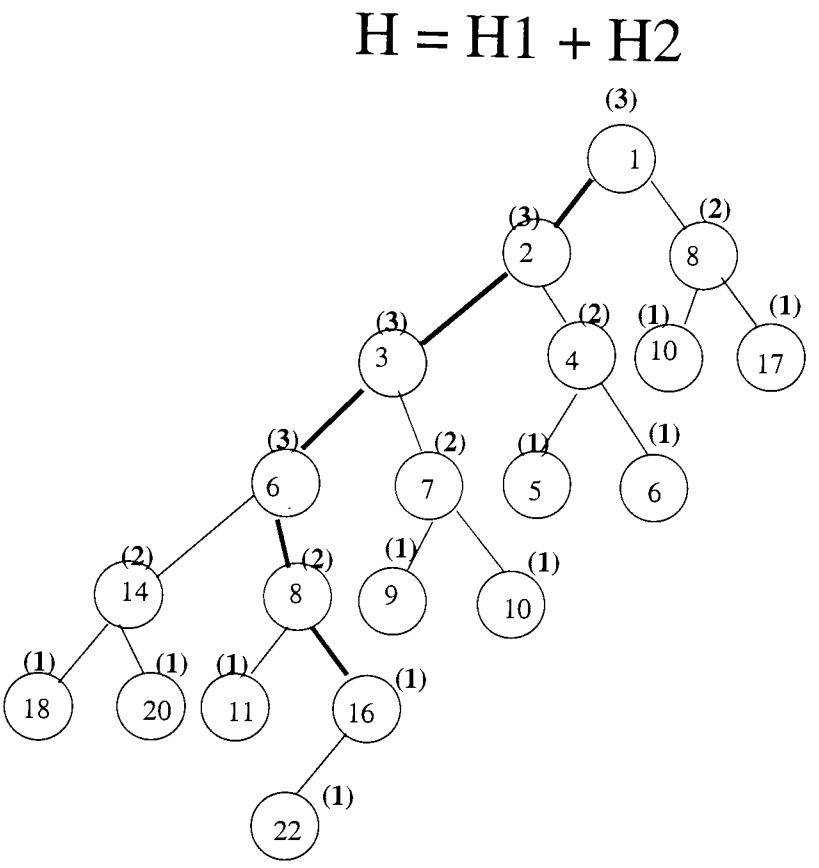


Figure 3.12: The leftist heap H produced as a result of the Merge.



Chapter 4

The Fast Implementation Builds OABST for 2,000,000 nodes!

The goal of this chapter is to evaluate the performance of the different implementations of the Hu-Tucker algorithm. The break even point between the fast implementation, described in 3.1.1 using leftist tree for priority queue, and the implementation building the tree by combining local minimum compatible pairs will be experimentally established. The time complexities of the two implementations will be compared to their corresponding theoretical time complexities. The performance of all phases of the Hu-Tucker algorithm will be measured for potential practical usage. An OABST will be build for the electronic version of "The Complete Works of William Shakespeare" and the results will be analyzed. The costs of the OABST, built using the Hu-Tucker algorithm, and the OBT, built using the Huffman algorithm, will be compared for variety of random input sequences.

The test environment for the experiments is: Sun Ultra 60, dual processor configuration with 4-way superscalar UltraSPARC-II, 300 MHz processor, MMU with 64 Instruction-TLB and 64 Data-TLB entries, 16 KB data and 16 KB instruction caches on chip, with 2 MB external secondary cache, and 512 MB RAM. The version of the instruction set is SPARC 9. The operating environment is Solaris 2.6. The source code is compiled using the GCC version 2.8.1 compiler. The *getrusage* system call is used to measure the total amount of time spent executing in user and system mode in sec-

onds. The measurements in the following sections are performed as in the following fragment of code:

struct getrusage ruse;
$t_0 = (getrusage(RUSAGE\_SELF, \&ruse), ruse.ru\_utime.tv\_sec +$
$ruse.ru\_stime.tv\_sec + 1e-6 *$
$(ruse.ru\_utime.tv\_usec + ruse.ru\_stime.tv_usec));$
callToFunction();
$t_1 = (getrusage(RUSAGE_SELF, \&ruse), ruse.ru\_utime.tv\_sec +$
$ruse.ru\_stime.tv\_sec + 1e-6 *$
$(ruse.ru\_utime.tv\_usec + ruse.ru\_stime.tv_usec));$
$User\_and\_System\_Time_{spent\_by\_(\textbf{callToFunction}())} = t_1 - t_0;$

4.1 Break Even Point

This section is concerned only with the time complexity of the Combination phase. The other two phases, Level Assignment and Reconstruction, are shared between the algorithms and the implementations, thus are excluded from the timing experiments[1]. The break even point between the two implementations, the Fast Implementation with leftist heaps (FILH) and the Combination using local minimum compatible pairs (LMCP), occurs while building OABST for a small number of input elements. To lower the error introduced by the *getrusage* system call, when very small times are measured, the time measurement is performed as follows. The function building the tree in phase I is called 100 times in a loop. The total time it takes to build 100 trees (which are identical because the same input data set is used 100 times) is measured and normalized. Ten different random number sequences are sampled, and the time measurements are averaged. The data in Figure 4.1 reflects the user and system time (seconds) as reported by *getrusage* system call. Every time measurement represents the arithmetic mean: $\frac{1}{10}(\sum_{i=1}^{10} TotalTime_{forBuilding100Trees})$

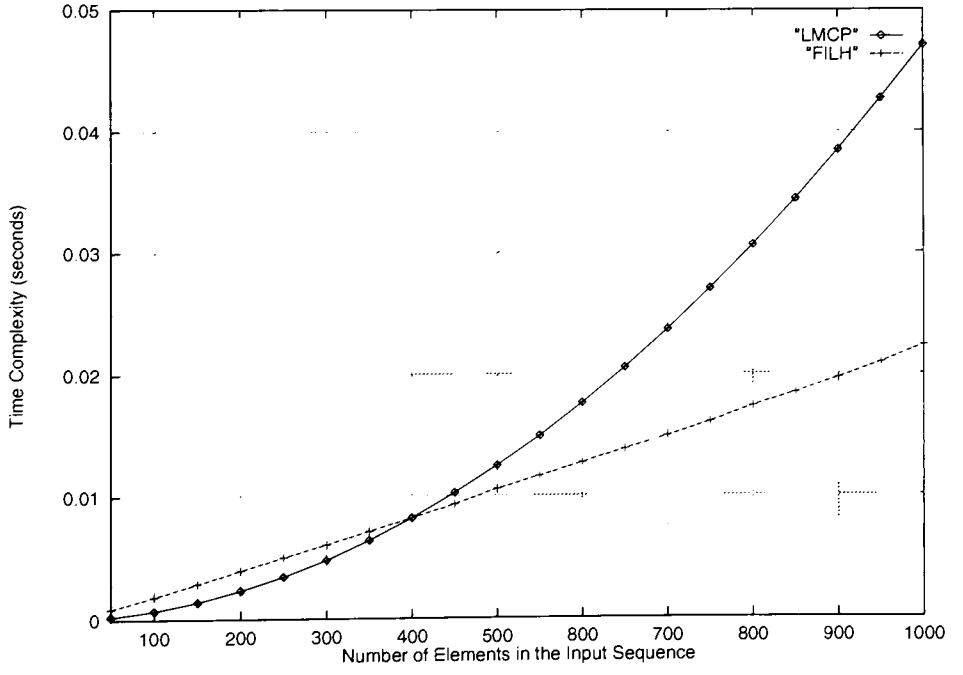
The advantage of the LMCP implementation is insignificant and is observed only for very small size input sequences (See Figures 4.1 and 4.2). The LMCP implementation marginally outperforms the FILH for the input data sizes between 2 and 400 nodes. Any practical usage of large OABST will need a fast implementation to build it.

[1]Section 4.3 discusses the time complexity of all three phases of the Hu-Tucker algorithm.

Figure 4.1: Time complexity (seconds) of the two implementations of the Combination phase for data sets between 50 and 1,000 nodes. Fast implementation with leftist heaps (FILH), implementation with combination of l.m.c.p (LMCP).

Nodes	FILH	LMCP	Nodes	FILH	LMCP
50	0.087	0.024	550	1.16	1.49
100	0.185	0.07	600	1.27	1.761
150	0.288	0.14	650	1.38	2.052
200	0.395	0.233	700	1.489	2.366
250	0.502	0.344	750	1.603	2.704
300	0.603	0.478	800	1.735	3.06
350	0.711	0.638	850	1.844	3.44
400	**0.82**	**0.82**	900	1964	3.846
450	0.93	1.024	950	2091	4.269
500	1.056	1.248	1000	2231	4.709

Figure 4.2: Break even point between the two implementations occurs for input sequence of 400 nodes.



4.2 LMCP Implementation vs. Fast Implementation Compared for Large Data Sets.

The time measurements in this section are performed, as in the previous section, only on the Combination Phase. Six different random number sequences are tested to average the running time for every size input data set. The measurements in Figure 4.4 represent the arithmetic mean:
$\frac{1}{6}(\sum_{i=1}^{6} Time\_CombinationPhase)$.
Figures 4.3, 4.5, 4.10, and 4.11 are based on the data in Figure 4.4.

The advantage of the fast implementation is obvious. It takes the FILH only 5 *seconds* to build the optimal tree in phase I for 100,000 nodes, and it takes the LMCP implementation 17 *minutes* and 39 *seconds* to build the same tree (See Figure 4.4). The time complexity of the LMCP implementation grows steadily with increment of approximately 8 seconds with every 1000 nodes increase of the input. Compare the 8 *seconds* increment of the LMCP implementation to the 0.05 *seconds* increment of the FILH for the same data sizes.

Figure 4.5 shows the growth of the running time of the fast implementation alone. The time complexity monotonically increases approximately with 0.05 seconds with each increment of the input data set of 1000 nodes. The average time complexity of the LMCP implementation grows 160 times faster than the FILH. Thus the only implementation used for real time encoding and decoding of very large alphabets using OABST should use the implementation model of the FILH.

Figure 4.3: Comparing the time complexity of the two implementations for data sets between $1,000$ and $100,000$ nodes. Fast implementation with leftist heaps (FILH), implementation with combination of l.m.c.p.(LMCP).

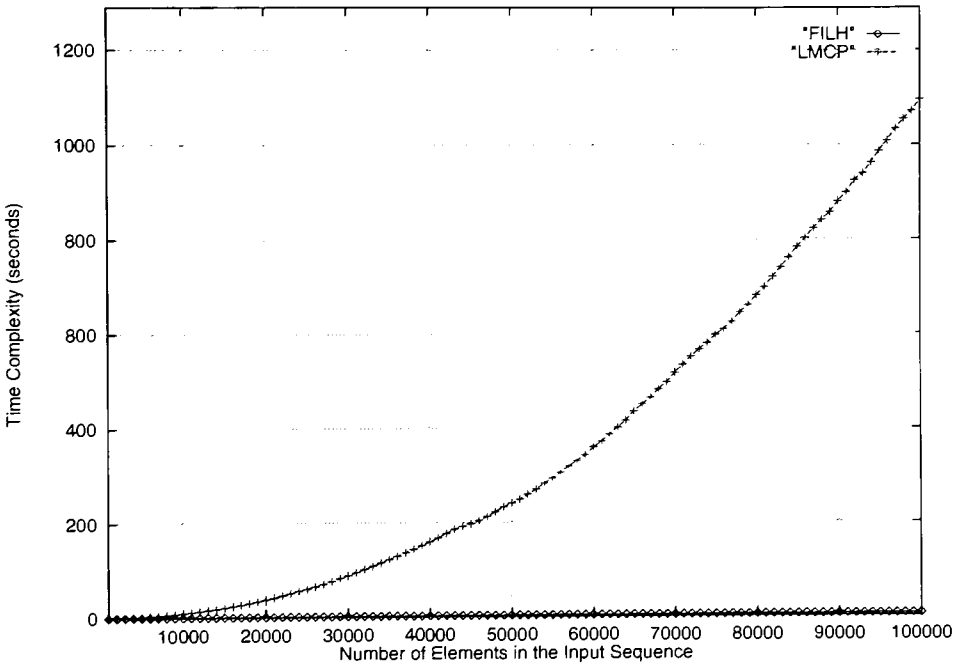


Figure 4.4: Experiments with input data sets in the range $1,000 - 100,000$ nodes. User and system times (seconds) spent by the two implementations: fast implementation with leftist heaps (FILH), implementation with combination of l.m.c.p.(LMCP).

Nodes	FILH	LMCP	Nodes	FILH	LMCP
1,000	0.03	0.09	50,000	2.39	240.7
2,000	0.06	0.39	51,000	2.43	248.48
3,000	0.09	0.86	52,000	2.48	260.08
4,000	0.13	1.52	53,000	2.54	270.16
5,000	0.16	2.37	54,000	2.6	283.15
6,000	0.20	3.39	55,000	2.64	293.97
7,000	0.24	4.61	56,000	2.74	306.22
8,000	0.28	6.03	57,000	2.75	319.17
9,000	0.32	7.61	58,000	2.82	330.47
10,000	0.36	9.40	59,000	2.87	343.66
11,000	0.41	11.38	60,000	2.95	359.75
12,000	0.46	13.5	61,000	3	371.66
13,000	0.49	15.82	62,000	3.04	386.75
14,000	0.54	18.36	63,000	3.09	402
15,000	0.59	21.05	64,000	3.17	415.63
16,000	0.64	23.93	65,000	3.22	434.42
17,000	0.69	27.04	66,000	3.27	450.5
18,000	0.74	30.22	67,000	3.31	464.26
19,000	0.78	33.69	68,000	3.39	481.79
20,000	0.83	37.35	69,000	3.41	497.9
21,000	0.88	41.43	70,000	3.48	517.32
22,000	0.93	45.79	71,000	3.54	533.99
23,000	0.98	50.61	72,000	3.63	551.42
24,000	1.03	55.51	73,000	3.65	566.86
25,000	1.10	60.04	74,000	3.71	581.11
26,000	1.15	64.56	75,000	3.77	597.1
27,000	1.18	69.87	76,000	3.81	609.7
28,000	1.22	75.68	77,000	3.86	625.3
29,000	1.28	82.38	78,000	3.94	645.2
30,000	1.33	88.36	79,000	4	660.6
31,000	1.38	94.66	80,000	4.05	681.2
32,000	1.43	101.07	81,000	4.1	698.4
33,000	1.47	107.78	82,000	4.17	720.5
34,000	1.54	114.58	83,000	4.21	740.4
35,000	1.58	121.76	84,000	4.28	761.7
36,000	1.64	128.88	85,000	4.34	783.6
37,000	1.69	136.29	86,000	4.38	802.4
38,000	1.74	143.95	87,000	4.5	823
39,000	1.79	151.73	88,000	4.51	840.7
40,000	1.85	**159.41**	89,000	4.6	856.62
41,000	1.90	167.99	90,000	4.63	879.9
42,000	1.95	177.034	91,000	4.69	898.5
43,000	1.99	186.052	92,000	4.73	924.9
44,000	2.06	191.87	95,000	4.89	986.4
45,000	2.11	197.318	96,000	4.95	1008.4
46,000	2.16	204.466	97,000	5.01	1033.1
47,000	2.22	212.924	98,000	5.04	1054.1
48,000	2.27	222.59	99,000	5.11	1070.7
49,000	2.33	232.852	100,000	5.19	1095.7

Figure 4.5: Time complexity of the fast implementation with leftist heaps.

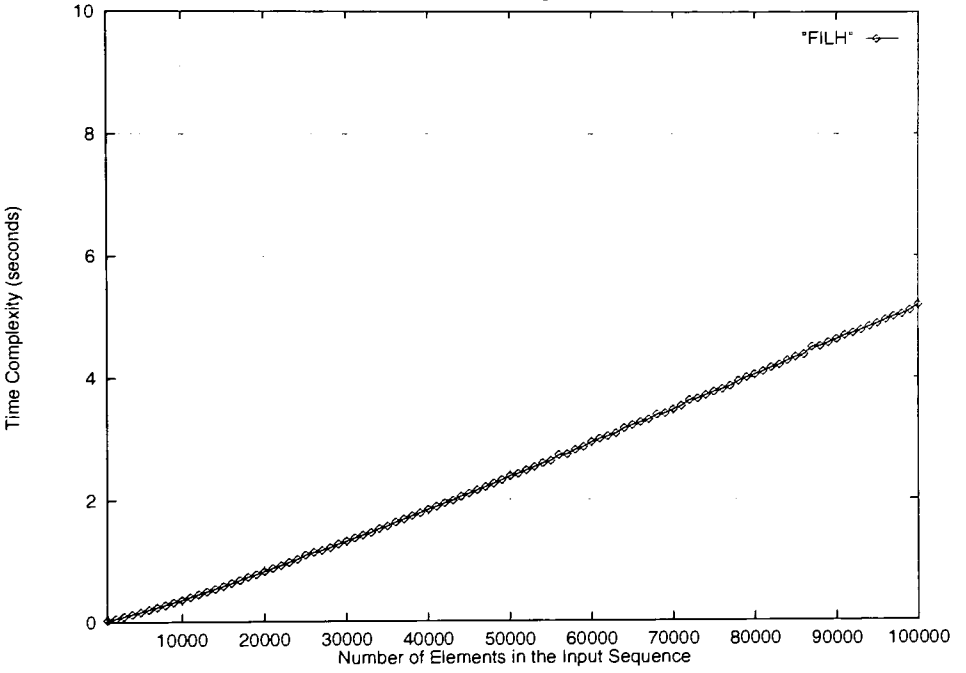
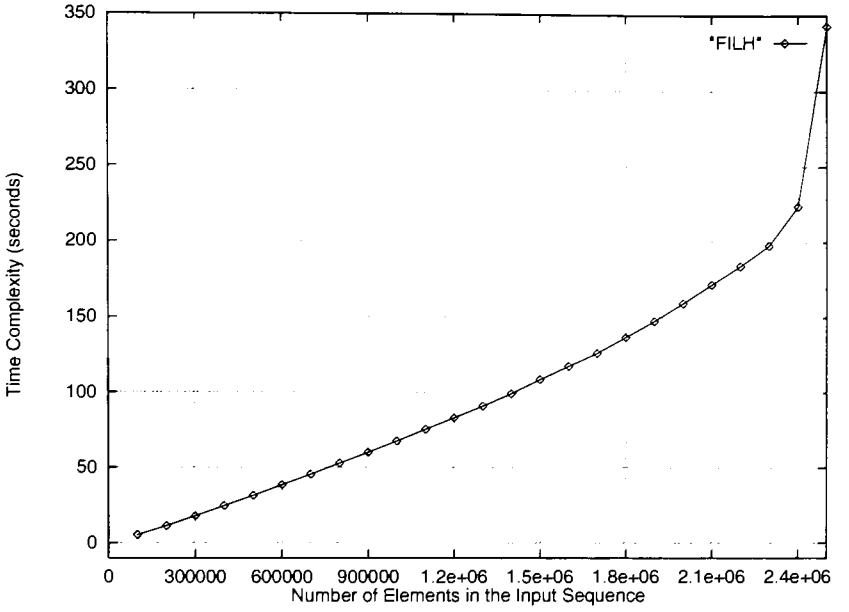


Figure 4.6: Time complexity of the fast implementation for data sets in the range $100,000 - 2,500,000$ nodes.

Nodes	FILH	Nodes	FILH
100,000	5.19	1,300,000	90.82
200,000	11.3	1,400,000	99.13
300,000	17.86	1,500,000	108.57
400,000	24.58	1,600,000	117.36
500,000	31.31	1,700,000	126.11
600,000	38.51	1,800,000	136.68
700,000	45.53	1,900,000	147.49
800,000	52.87	**2,000,000**	**159.32**
900,000	60.15	2,100,000	171.79
1,000,000	67.66	2200000	184.15
1,100,000	75.43	2300000	197.72
1,200,000	83.11	2400000	223.69
		2,500,000	342.99

The short times required by the fast implementation encouraged me to collect data for very large data sets. The fast implementation built the optimal tree for $2,000,000$ nodes for the same amount of time units the LMCP implementation needed to build the tree for $40,000$ nodes! See FigurE 4.6 and Figure 4.7. The user and system time spent in phase I, progressively increases approximately 7 seconds with the input increment of $100,000$ nodes until the input data set reaches the size $2,000,000$ nodes. Running the test cases for data sets containing more than $2,000,000$ nodes caused very extensive virtual memory usage. The experimental time complexity curve no longer follows the linear fashion of the user and system times in the range $100,000$ $2,000,000$ nodes. It was impractical to continue the experiments for data sets larger than 2,500,000 nodes due to very long execution times. However, a machine with larger RAM resources will continue to build OABST, requiring user and system times which will follow the linear growth time complexity observed for input data size up to $2,000,000$ nodes.

Figure 4.7: Time complexity of the fast implementation for very large data sets.



4.3 Time Complexity of All Phases of the Hu-Tucker Algorithm

The goal of this experiment is to evaluate the possibility of using the Hu-Tucker algorithm for building OABST for practical purposes. Thus, unlike the previous sections, this section is concerned with the measurements of the system and user time spent by all three phase of the algorithm. Previous experiments show that the winning implementation for the Combination phase is the fast implementation using the leftist heaps. Thus, in this experiment the fast implementation with leftist heaps is used for phase I, Combination. Phase II, Level Assignment, is implemented using the model described in Section 3.2.1. Phase III, Reconstruction, is implemented using the Stack algorithm, described in 3.3.1. The time measured represents:

$$Time_{Hu-Tucker} = Time_{Initialization} + Time_{Combination\_Phase(FILH)} + \\ + Time_{Level\_Assignment} + Time_{Recombination\_Phase(Stack\_Algorithm)}$$

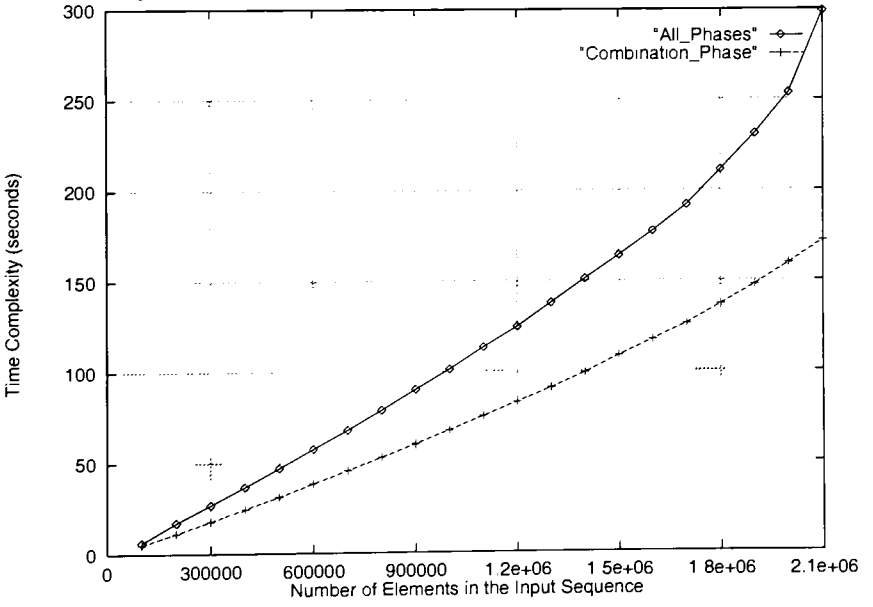
The optimal tree built at the end of the Combination phase is used to calculate the levels of the terminal nodes and the tree itself is discarded. I

chose to leave the tree built by phase I unused and not dispose the memory occupied by it because the system time it took to release the large number of dynamically allocated memory was an extremely costly operation.

Figure 4.8: Time measurements of all three phases of the Hu-Tucker algorithm (seconds) for large input sequences.

Nodes	Hu-Tucker	Nodes	Hu-Tucker
100,000	6.3	1,100,000	113.27
200,000	17.12	1,200,000	124.51
300,000	26.99	1,300,000	137.82
400,000	36.87	1,400,000	150.92
500,000	47.12	1,500,000	163.85
600,000	57.66	1,600,000	177.16
700,000	67.93	1,700,000	191.66
800,000	78.76	1,800,000	211.07
900,000	90.13	1,900,000	230.84
1,000,000	101.18	2,000,000	253.39
		2100000	298.93

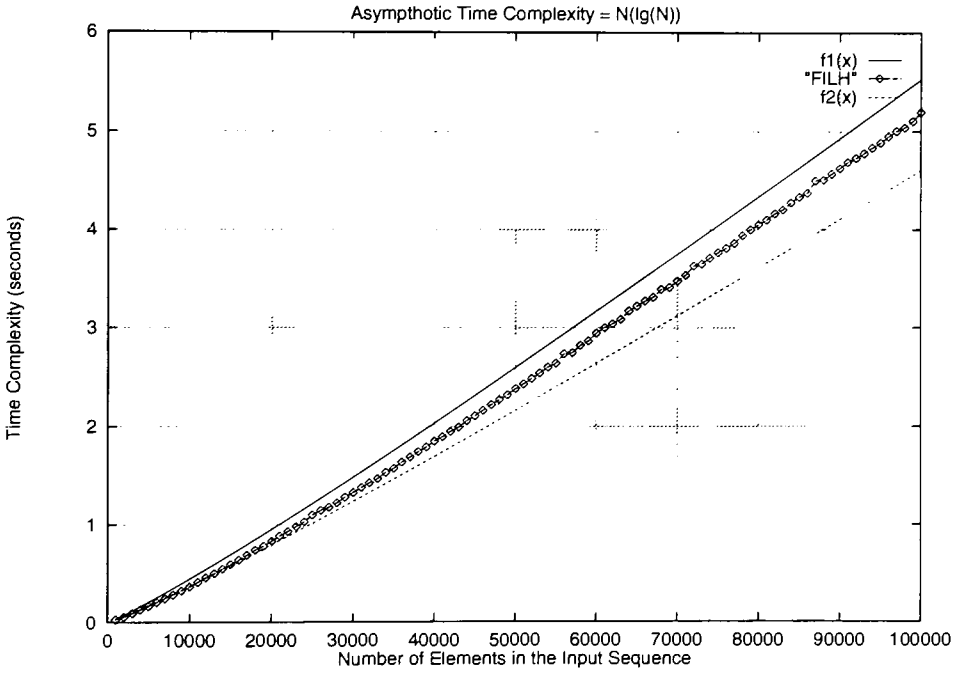
Figure 4.9: Time complexity of the Hu-Tucker algorithm.



Figures 4.8 and 4.9 show that an OABST for $2,000,000$ can be built in less than 5 *minutes*. The non linear growth of the total system and user time spent by the implementation for input sequences larger than $1,800,000$ nodes is result of the heavy virtual memory usage. Figure 4.9 also shows that the majority of the time is spent by the Combination phase. The running time of the algorithm is practical and allows for usage in real applications.

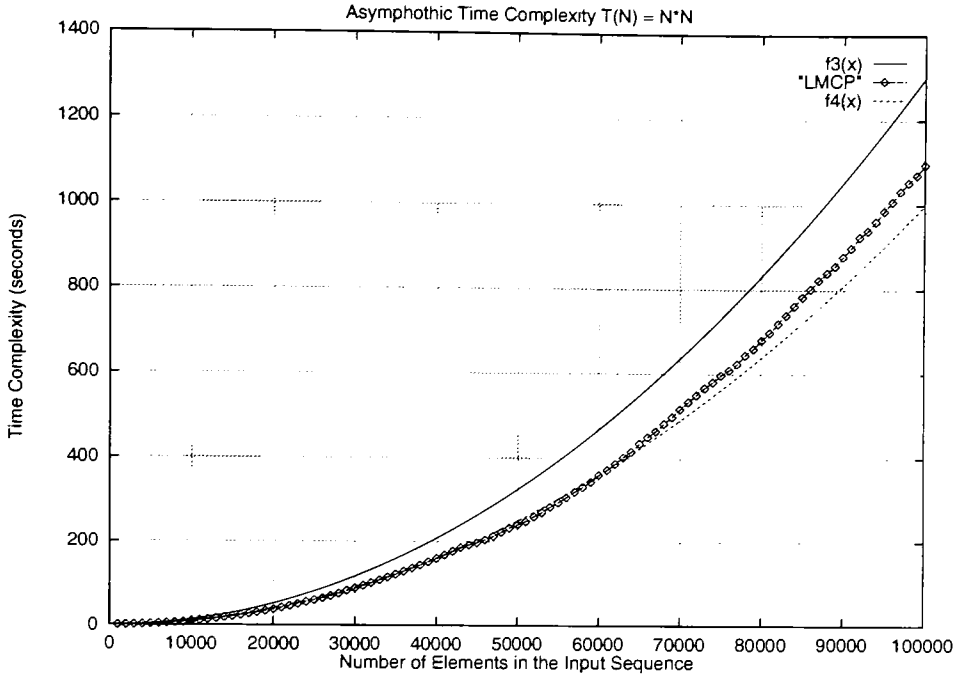
4.4 Experimental Time Complexity vs. Theoretical Time Complexity

Figure 4.10: Comparison of the theoretical and experimental time complexity of the fast implementation.



This section aims to investigate how the different complexities of the two implementations reflect their experimentally measured time complexities. Figure 4.10 shows the time complexity of the fast implementation with leftist heaps sandwiched between $f_1(n) = (4.8)10^{-6} n \lg n$ and $f_2(n) = (4)10^{-6} n \lg n$. Figure 4.11 shows the experimentally established

Figure 4.11: Comparison of the theoretical time complexity of the Combination phase with *l.m.c.p.*



time complexity of the LMCP implementation bounded by $f_3(n) = (1.3)10^{-7}n^2$ and $f_4(n) = 10^{-7}n^2$ The LMCP implementation is very simple. On every iteration one pass through the working sequence is sufficient to find a local minimum compatible pair which is combined. The structure of the implementation requires one function call per iteration. The fast implementation on the other hand is very complex. In the worst case scenario there will be three priority queue operations performed on the Master Priority queue one $PQInsert$, one $PQDeleteMin$, and one $PQDelete$, and eight operations on the HPQs: two $PQDelete$, three $PQMerge$, two $PQDeleteMin$, and one $PQInsertoperations$. The leftist tree is selected as a priority queue structure and every operation is implemented using recursive calls to the $PQMerge$, see Chapter 3, Section 3.6.1 for details. The depth recursion and the complexity of the algorithm, and the implementation, cause the leading coefficient of the largest term of the function $f_1(n)$, bounding the fast implementation to be 10 times higher than the leading coefficient of the largest term of function $f_3(n)$, bounding the the LMCP

implementation.

4.5 Example Book Encoding

The goal of this experiment is to illustrate the practical usage of the Hu-Tucker algorithm for encoding electronic texts.

"The Complete Works of William Shakespeare" by the World Library. Inc., provided by the project Gutenberg Etext of Illinois Benedictine College is selected for this study. To appropriately measure the frequencies of the words in the text the following changes to the file were necessary:

- All the letters in the file were capitalized.
- The following characters are removed:
  - >>
  - <<
  - §
  - "
- Following punctuation characters have been replaced:
  - ? $\xrightarrow{with}$ *question_mark*
  - ! $\xrightarrow{with}$ *exclamation_point*
  - $\xrightarrow{with}$ *period*
  - , $\xrightarrow{with}$ *comma*
  - : $\xrightarrow{with}$ *colon*
  - ; $\xrightarrow{with}$ *semi_colon*

The - and ' signs, when appearing at the beginning or at the end of the word, are removed[2] Thus all the words appear in capitals, and the punctuation signs appear as separate lower case strings. The resulting version of the text was ran through *awk* filter script and the number of the words appearances in the text are counted. The words list was sorted in alphabetic order. The input file fed to the FILH implementation contains the

[2]Otherwise '$THIS$, $-THIS$, and $THIS'$ are considered erroneously as different words.

total number of words, followed by the pairs: *number of word appearances, word string* sorted in alphabetic order according to the word string. In total 29338 words were encountered. They are repeated between 1 and 83067 times. The top winners are:

weight	word	weight	word
5479	THOU	9556	IS
5878	HAVE	10475	*question_mark*
5916	AS	10939	IN
6222	BUT	11078	THAT
6230	HE	12468	MY
6807	THIS	13591	YOU
6850	HIS	14545	A
6866	YOUR	18113	OF
7064	BE	19120	TO
7650	IT	20608	I
7744	ME	26636	AND
7980	WITH	27544	THE
8186	FOR	77926	*period*
8687	NOT	83067	*comma*

$$Total\_Execution\_Time = Time_{Initialization} + Time_{Combination} + Time_{Level\_Assignment} + Time_{Recombination} = \\ = 2.64 \ seconds$$

OABST Cost $= \sum_{i=1}^{29338} w_i l_i = 10468466$

$$\text{Averaged Word Encoding Length} = \frac{\sum_{i=1}^{29338} w_i l_i}{\sum_{i=1}^{29338} w_i} = \frac{10468466}{1102905} \\ = 9.4917205 \ bits \ per \ word$$

"The Complete Works of William Shakespeare":

$File\_Size = 6977448$ Bytes
$Compressed\_File = 1308558.2$ Bytes

Figure 4.12: Results of encoding "The Complete Works of William Shakespeare" Statistics of the numbers of the words at a given level.

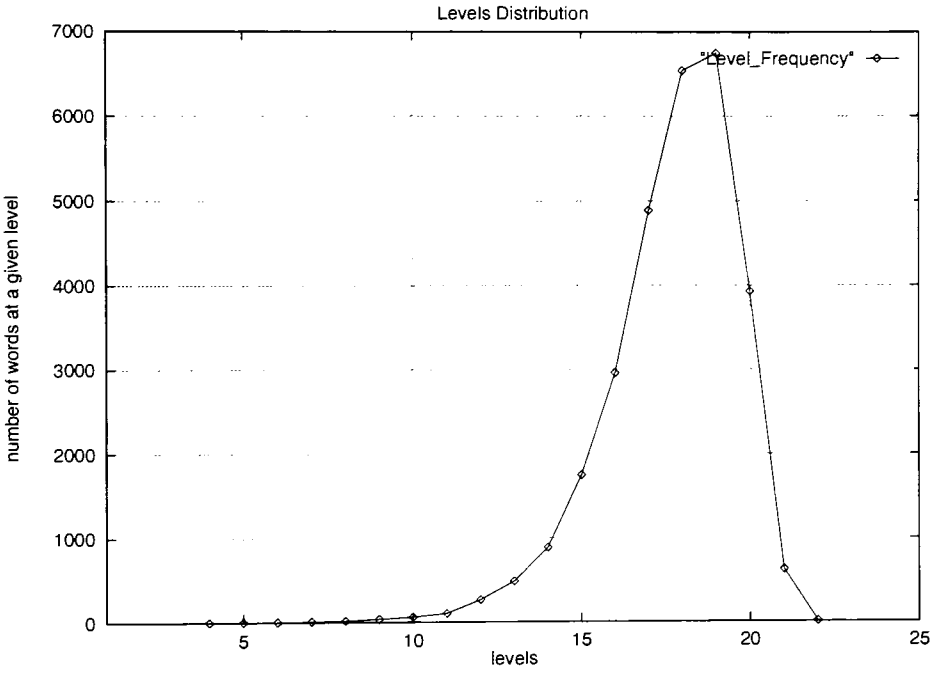
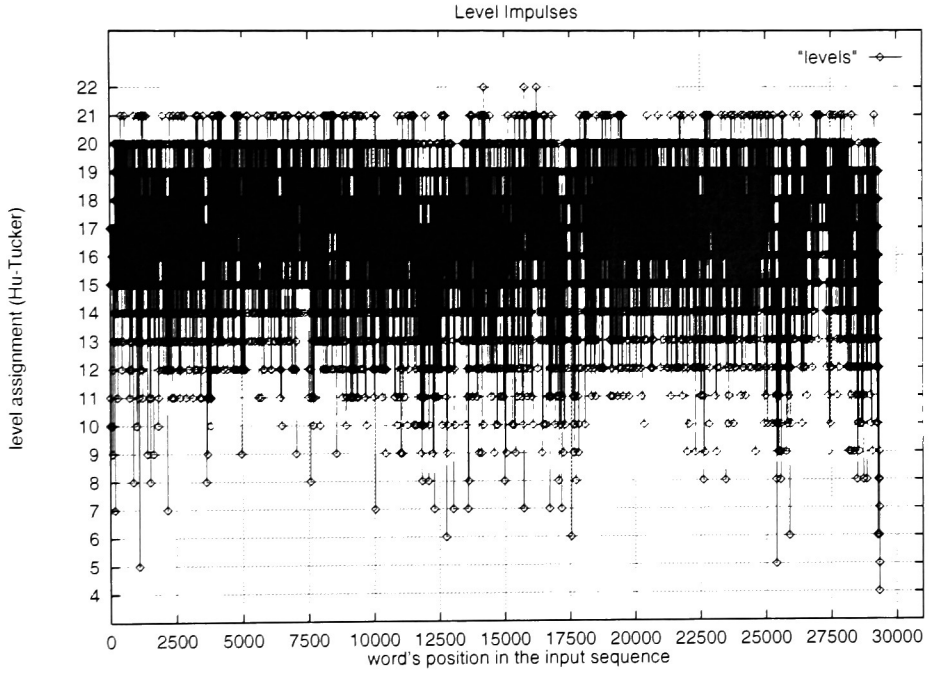


Figure 4.13: "The Complete Works of William Shakespeare" Distribution of the levels of the words in the input sequence according to their position.



4.5.1 Comparison of Different Compression Algorithms

In the following experiment the compression ratio achieved by the Hu-Tucker algorithm will be compared to the compression ratio of a fixed-length encoding, and two well-known compression programs: the standard Unix *compress* and *gzip*. The file containing the electronic version of "The Complete Works of William Shakespeare" is subject of the experiment.

Fixed length encoding calculations:
Number of different words = 29338 ⇒ 15 bits are required to encode each word. Total number of words in the file = 1102905 ⇒ Encoded File Size = 15x1102905 = 16543575 bits = 2067946.87 Bytes.

The following table shows the results. The Input file contains the "The Complete Works of William Shakespeare", Input File Size = 6977448 *Bytes*. $compression\ ratio = \frac{Input\_File\_Size[Bytes]}{Output\_File\_Size[Bytes]}$

Algorithm *command*	Operates on	Output file size [B]	Compression ratio
Hu-Tucker	words	1308558.2	5.3322
Fixed Length code	words	2067946.87	3.3741
Lempel-Ziv *compress*	characters	2224531	3.166
Lempel-Ziv *gzip*	characters	2070927	3.3692
Lempel-Ziv *gzip -9*	characters	2046942	3.4087

Both *compress* and *gzip* reduced the input file size by at least 70 percent using Lempel-Ziv coding. The *gzip* command has different options, allowing for changes of the compression ratio or the running time of the program. The default compression level is 6, forth row of the table, and the best compression level is 9, the last row of the table. The Hu-Tucker algorithm outperforms significantly the two compression programs. The reason is that the Hu-Tucker algorithm is applied to encode whole words, while *compress* and *gzip* process files on character bases.

The OABST built by the Hu-Tucker defines variable length prefix code which provides considerably better compression than fixed-length codes, see the

first two rows of the table.

The performance of the algorithm and the implementation, and the excellent compression ratio show that the Hu-Tucker can successfully be used for applications requiring optimal encoding and decoding on the fly.

One potential application using the Hu-Tucker algorithm is the File Transfer Protocol (FTP). The Hu-Tucker algorithm will provide additional layer in the application level protocol used for the data connection. The OABST built will effectively compress and decompress extremely large data files, respectively on the transmitting and receiving end. Any distributed or Client/Server application[3] where submissions of large amounts of data take place could also benefit from the Hu-Tucker algorithm to build an OABST, used for encoding and decoding as front-end and back-end during the transmission on both client and server side. It is important to realize that the knowledge of the frequencies of usage of the words in the language, or letters of the alphabet however, will have to be available to both client and server side to allow for the construction of the OABST.

4.6 Experimental Comparison of the Cost of the Huffman and the Hu-Tucker Trees

This section will compare the compression ratios of the two algorithms. Random number input sequences are generated and fed to the engine of the two algorithms. The cost of the OABST, generated by the Hu-Tucker algorithm and the OBT, generated by the Huffman is averaged by the sum of the weights in the input sequence.

For every size of input sequences 5 different random inputs are examined and the compression ratio is averaged, for the data presented in Table 4.15 and Figure 4.14. Data sets in the range of 1000 to 99000 nodes are examined with increment of 1000 nodes.

The experimental data shown in Table 4.16 and Figure 4.17 has only one data set per input size examined due to the length of the experiment.

As can be seen from the tables in Figure 4.15 and Figure 4.15, the difference in the compression ratio, expressed as *bits per word* is really negligible. For the range of input sets between 1000 and 99000 nodes the minimum and maximum differences are 0.0731 and 0.0947 respectively. For input data sets

[3]CORBA and Java RMI are good candidates.

between 100000 and 1000000 it is between 0.0901 and 0.0943. The difference in the average length of the encoding word stabilizes around 0.09 *bits per word*. It can be seen from the graphs that it does not exhibit any tendency to grow or shrink. This gives us enough confidence to expect that for any larger input data sets the difference observed will be close to 0.09 bits per word.

The Huffman algorithm has very simple implementation in comparison with the Hu-Tucker algorithm. However the tree built by the Huffman algorithm can only be used for encoding and additional hashing mechanism is needed for the decoding process, especially for alphabets containing millions of nodes.

Figure 4.14: Comparing compression ratios of the Hu-Tucker and the Huffman algorithms.

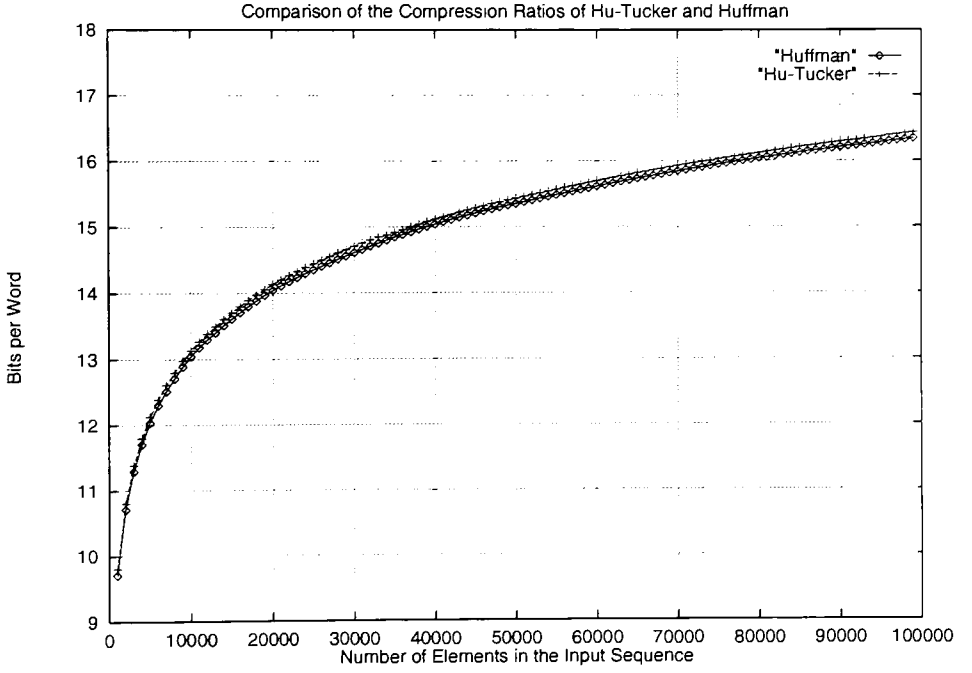


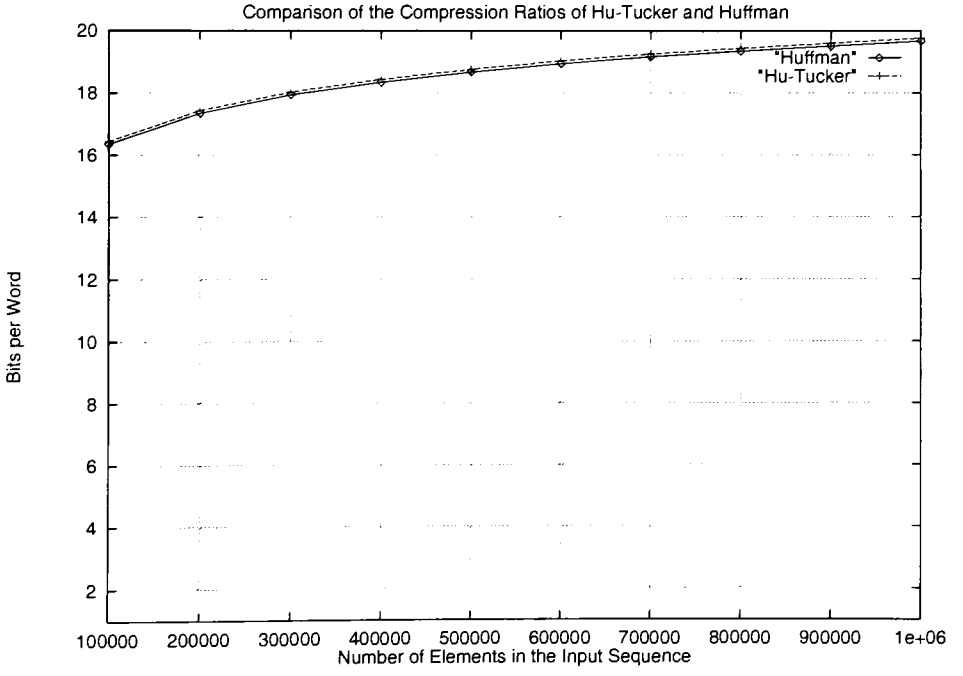
Figure 4.15: Compression ratio measured in *bits per word* for random sequences in the range of 1000 to 99000 nodes.

number of words	Hu-Tucker [*bpw*]	Huffman [*bpw*]	number of words	Hu-Tucker [*bpw*]	Huffman [*bpw*]
1,000	9.79906	9.70797	50,000	15.4328	15.3507
2,000	10.8043	10.7099	51,000	15.4593	15.3766
3,000	11.3838	11.2926	52,000	15.4871	15.4045
4,000	11.7999	11.7057	53,000	15.5146	15.4306
5,000	12.126	12.036	54,000	15.5406	15.4574
6,000	12.3878	12.2975	55,000	15.5683	15.4854
7,000	12.608	12.5146	56,000	15.5949	15.5106
8,000	12.7987	12.7068	57,000	15.6198	15.5347
9,000	12.9769	12.8861	58,000	15.6448	15.5614
10,000	13.1306	13.0427	59,000	15.6714	15.5867
11,000	13.2662	13.1773	60,000	15.6956	15.6121
12,000	13.3881	13.2972	61,000	15.7202	15.6372
13,000	13.4969	13.4022	62,000	15.7442	15.6611
14,000	13.6057	13.5111	63,000	15.7682	15.6849
15,000	13.7054	13.6109	64,000	15.7921	15.7091
16,000	13.8006	13.7083	65,000	15.8151	15.7323
17,000	13.8914	13.799	66,000	15.8376	15.7542
18,000	13.9746	13.884	67,000	15.8602	15.7758
19,000	14.0576	13.9682	68,000	15.8821	15.7997
20,000	14.1302	14.0393	69,000	15.9033	15.8206
21,000	14.2009	14.1106	70,000	15.9249	15.8416
22,000	14.2663	14.1763	71,000	15.9465	15.8636
23,000	14.3287	14.2383	72,000	15.9671	15.8847
24,000	14.3882	14.2965	73,000	15.9877	15.904
25,000	14.443	14.3494	74,000	16.0071	15.9251
26,000	14.4982	14.4056	75,000	16.0278	15.9453
27,000	14.5517	14.4573	76,000	16.0471	15.9643
28,000	14.6061	14.5115	77,000	16.0657	15.983
29,000	14.6555	14.5618	78,000	16.0856	16.0031
30,000	14.706	14.6113	79,000	16.1048	16.0218
31,000	14.7537	14.66	80,000	16.1227	16.0406
32,000	14.8031	14.7103	81,000	16.1403	16.0571
33,000	14.8465	14.7547	82,000	16.1581	16.0742
34,000	14.8741	14.7986	83,000	16.1758	16.0937
35,000	14.917	14.8439	84,000	16.1929	16.1093
36,000	14.9593	14.8841	85,000	16.2108	16.1276
37,000	14.9991	14.9257	86,000	16.2273	16.1444
38,000	15.0393	14.9654	87,000	16.243	16.1597
39,000	15.0792	15.0036	88,000	16.2601	16.176
40,000	15.1156	15.04	89,000	16.2763	16.1914
41,000	15.1511	15.0747	90,000	16.2911	16.2072
42,000	15.1857	15.1092	91,000	16.3067	16.2224
43,000	15.2202	15.1435	92,000	16.3158	16.2319
44,000	15.2539	15.1754	93,000	16.3313	16.2472
45,000	15.2851	15.2078	94,000	16.3471	16.2624
46,000	15.3169	15.2385	95,000	16.3607	16.275
47,000	15.3466	15.267	96,000	16.3869	16.2947
48,000	15.3755	15.2967	97,000	16.4011	16.3098
49,000	15.4048	15.3243	98,000	16.4157	16.3237
50,000	15.4328	15.3507	99,000	16.4288	16.3364

Figure 4.16: Comparison of compression ratios of Hu-Tucker and Huffman algorithms for very large data sets.

number of words	Hu-Tucker [bpw]	Huffman [bpw]
100000	16.4416	16.3496
200000	17.4434	17.3513
300000	18.0364	17.9458
400000	18.4434	18.3509
500000	18.7664	18.6735
600000	19.0352	18.9446
700000	19.2585	19.1684
800000	19.4436	19.351
900000	19.6114	19.5171
1000000	19.766	19.6727

Figure 4.17: Comparison of the cost of the OABST and OBT for very large data sets.



4.7 Conclusions

The OABST problem has been and still is a subject of study. After the initial publication of the two known algorithms. the Hu-Tucker and the Garsia-Wachs for building OABST, in 1972 and 1979 respectively, the theoreticians were challenged to discover simpler proofs. For references see Section 2.2.3. Both the Hu-Tucker and the Garsia-Wachs algorithms solve the general OABST problem in $O(n \lg n)$ time. The recent interest in the OABST problem has been in finding linear time solutions for the special types of input sequences, see Section 1.2 for references.

The subject of this study is to build implementation models for the Hu-Tucker algorithm, to evaluate the performance of the different implementations, and compare the compression ratios of the OABST vs. the OBT for large input sequences. Efficient model for building the the OABST was designed and implemented. The author didn't find, at the time of the writing of the paper, similar studies. Although publication of the implementation of the Hu-Tucker achieving $O(n^2)$ time complexity exist, see [Yoh72] for details, there is no data for experimental investigation of the experimental time complexity of the algorithm.

This study shows that the Hu-Tucker algorithm can be implemented efficiently and the implementation can build the OABST for extremely large data sets[4], which makes the algorithm very practical for dictionaries and electronic texts encoding and decoding. The experiments also showed that the average difference of the length of the encoding using OABST and OBT, is close to 0.09 *bits per word.* The optimality that the OABST achieves, the performance of the implementation of the Hu-Tucker algorithm, and the embedded mechanism for fast encoding and decoding are very attractive for applications.

[4]OABST for $2,000,000$ nodes was built in 253.385 seconds, less than 4.5 minutes.

Acknowledgement

My sincere gratitute goes to Professor Stanislav P. Radsizsowski for introducing me to this problem, for the invaluable suggestions, and attention to my work. I am very thankful to my family for their support.

Appendix A

Hu-Tucker Algorithm, Phase I, Combination

A.1 Fast Implementation

C language notation is used to refer to elements of arrays and structures, flow of control, and conditional constructs. Comments begin with a pound sign (#).

Initialization
for (k=1: $k \leq N$; k++)
    $A[k] = address\_of(T[k])$;

#Create $N-1$ priority queues to hold the initial Huffman sequences and
#initialize the Master Priority Queue
for (k=1; $k < N$; k++)
    $HPQ[k] = (T[k], T[k+1])$;
    Create a new node *mpq*, an element of MPQ
    Initialize *mpq*:
        $mpq.MinSum = \sum$(of the two elements in $HPQ[k]$)
        $mpq.i$ = the position of the *minimum* element of $HPQ[k]$
        $mpq.j$ = the position of the *next minimum* element of $HPQ[k]$
        $mpq.HPQRoot$ points to the Principal of the $HPQ[k]$
    $T[k].MPQR = T[k+1].MPQL = address\_of(mpq)$
    Insert the new *mpq* element to MPQ
$T[1].MPQL = T(n).MPQR = null$

End:Initialization

Main Loop

for (k=1; $k<N$; k++)

$mpq\_min = extract\_min\ (MPQ)$

#Combine the selected pair of nodes and update the array $A[N]$

Let l is the position of the left node and r is the position of the right node, thus ($l < r$): $l = mpq\_min.i$. $r = mpq\_min.j$

Create a new internal node I and a new HPQ node H

$I.position = H.position = l$

$I.weight = H.weight = mpq\_min.MinSum$

$I.type = H.type = INTERNAL\_NODE$

$I.left\_child = A[l]$

$I.right\_child = A[r]$

$A[l] = address\_of(I)$

$A[r] = null$ **if** $A[l]$ **and** $A[r]$ point to terminal nodes

then

#Case I: Combining two terminal nodes

Let $A[l]$ points to $T[l]$ and $A[r]$ points to $T[r]$

Delete $T[l]$ from $T[l].MPQL \rightarrow HPQ_{Root}$

Delete $T[l]$ from $T[l].MPQR \rightarrow HPQ_{Root}$

Delete $T[r]$ from $T[r].MPQL \rightarrow HPQ_{Root}$

Delete $T[r]$ from $T[r].MPQR \rightarrow HPQ_{Root}$

PQ = Merge ($T[l].MPQL \rightarrow HPQ_{Root}$, $T[l].MPQR \rightarrow HPQ_{Root}$. and $T[r].MPQR \rightarrow HPQ_{Root}$)

Insert the new element H to priority queue PQ

Create a new mpq element

$mpq.MinSum = \sum$(of the *minimum* and *next minimum*) elements from PQ

$mpq.i$ = the position of the minimum element of PQ

$mpq.j$ = the position of the next minimum element PQ

Update the MPQR reference of the Terminal node delimiting the $T[l].MPQR \rightarrow HPQ_{Root}$ on the left, to refer to the new mpq

Update the MPQL reference of the Terminal node delimiting the $T[r].MPQL \rightarrow HPQ_{Root}$ on the right, to refer to the new mpq

Delete the element of MPQ pointed by $T[l].MPQL$ from MPQ

Delete the element of MPQ pointed by $T[r].MPQR$ from MPQ

Delete the minimum element from MPQ

Insert the new mpq element to MPQ

continue

#End combining two terminal nodes.

if $A[mpq.l]$ **and** $A[mpq.r]$ refer to internal nodes
then
#Case II: Combining two internal nodes
Let PQ = priority queue pointed by $mpq\_min.HPQ_{Root}$
Delete *minimum* and *next minimum* elements from PQ
Insert the new element H to the priority queue PQ
Create a new *mpq* element
$mpq.MinSum = \sum$ of the *minimum* and *next minimum* elements from the PQ
$mpq.i$ = the position of the *minimum* element of PQ
$mpq.j$ = the position of the *next minimum* element PQ
Update the $MPQR/MPQL$ references of the Terminal nodes delimiting the priority queue pointed by $mpq\_min.HPQ_{Root}$ on the left/right to refer to the new *mpq*.
Delete the minimum element from the MPQ
Insert the new *mpq* element to MPQ

else
#Case III: Combining terminal and internal node.
Delete the *minimum* and *next minimum* elements from the priority queue pointed by $mpq\_min.HPQ_{Root}$
Delete the terminal node from the other HPQ priority queue to which it participates, if necessary
Merge the two priority queues, if necessary
Insert the new element H to priority queue PQ
Create a new *mpq* element
$mpq.MinSum = \sum($ of the two elements in $PQ)$
$mpq.i$ = the position of the minimum element of PQ
$mpq.j$ = the position of the next minimum element PQ
Update the $MPQR/MPQL$ references of the two terminal nodes delimiting the two priority queues on the left/right, pointed by the $MPQL$ and $MPQR$ references of the terminal node to refer to the new *mpq* node, if necessary.
Delete the minimum element from the MPQ
Insert the new *mpq* element to MPQ

End: Main Loop
End: Phase I Completed

A.2 Implementation with Local Minimum Compatible Pairs

Initialization
for (k=1; $k \leq N$; k++)
    Initialize $T[k]$
    $A[k] = address\_of(T[k])$
End:Initialization

Main Loop
for (k=1; $k < N$; k++)
    Initialize the search for **l.m.c.p** from the beginning of the
        working sequence.
    while(1)
    # This loop looks for **l.m.c.p**. When it finds it combines the nodes
    # in the pair and exits the **while** loop.
        Locate the two nodes with least weight in the current
            Huffman Sequence.
        Let the $A[i]$ and $A[j]$ point to the element with the smallest and next
            to smallest weights respectively within the current
            Huffman Sequence.
        if(both $A[i]$ and $A[j]$ point to *internal* nodes)
        then
            #This is a **l.m.c.p** $\Rightarrow$ Combine the two nodes:
            CombineNodes(A[i], A[j])
            1. Create a new node I
                $I.position = min(i, j)$
                $I.type = INTERNAL$
                $I.weight = \sum$(weights of the nodes pointed by $A[i]$ and $A[j]$)
                $I's$ left and right tree pointers = $A[i]$ and $A[j]$
            2. Update the array $A[N]$:
                if ($i < j$)
                then
                    $A[i] = address\_of(I)$
                    $A[j] = null$
                else
                    $A[j] = address\_of(I)$
                    $A[i] = null$
            end :CombineNodes()

```
            break
        else
            #One of the nodes is a terminal node
            if (the right node is a terminal node)
            then
                if(∃ node from the right adjacent Huffman Sequence with
                    smaller weight than the left node of the pair )
                then
                    Resume the search for l.m.c.p. from the current
                        Huffman Sequence
                    continue
                else
                    #l.m.c.p is found
                    CombineNodes(A[i], A[j])
                    break
            else
                #l.m.c.p. is found
                CombineNodes(A[i], A[j])
                break
    End:while
End: Main Loop
End: Phase I Completed
```

Appendix B

Building OABST using the Stack Algorithm

Initialize
$Q[N] = T[N]$
S=empty

Main Loop
while (!((Q is empty) and $((size\_of(S)) == 1)$))
  if $((size\_of(S)) \geq 2)$ and (the top 2 elements have the same levels)
  then
    $level_{top} = level\_of\_element(pop(S))$
    Combine the two elements of the Queue Q which are referred by the top two elements of the Stack
      1. Create new internal node *new_node*
      2. $new\_node.level = StackTop.level - 1$
      3. $new\_node.weight = StackTop.weight$
      4. $new\_node.letter = (StackTop - 1).letter$
      5. $new\_node.left\_child = (StackTop - 1).node$
      6. $new\_node.right\_child = StackTop.node$
    Create new Stack element *new_stack_node*
      1. $new\_stack\_node.level = new\_node.level$
      2. $new\_stack\_node.level = new\_node.level$
      3. $new\_stack\_node.reference = address\_of(new\_node)$
      4. $pop(Stack)$
      5. $pop(Stack)$

6. $push(Stack, new\_stack\_node)$

else

Remove an element from the front of the queue Q and create a new stack element new_stack_node and push it on the top of the $Stack$:

1. $new\_stack\_node.level = QueueTop.level$
2. $new\_stack\_node.letter = QueueTop.letter$
3. $new\_stack\_node.reference = address\_of(QueueTop)$
4. $push(Stack, new\_stack\_node)$

Advance the $QTop$ pointer

End: OABST is built

Appendix C

Leftist Heap Operations

heap Merge(heap H1, heap H2)

if $(H1 == null)$
then
    $return(H2)$

if $(H2 == null)$
then
    $return(H1)$

if $(H1.key > H2.key)$
then
    $swap(H1 \longleftrightarrow H2)$

if $(H1.right == null)$
then
    $H1.right = H2$
    $H2.parent = H1$
else
    $temp\_heap = Merge(H1.right, H2)$
    $H1.right = temp\_heap$
    $temp\_heap.parent = H1$

if $((H1.left).rank > (H2.right).rank)$
then
    $swap(H1.left \longleftrightarrow H2.left)$

else
$H1.rank = (H1.right).rank + 1$
$return(H1)$
End:Merge

heap Insert(heap H, element El)
element $El \rightarrow$ one element heap
$El.left = El.right = El.parent = null$
$return(Merge(H, El))$

heap DeleteMin(heap H)
$return(Merge(H.left, H.right))$
End:DeleteMin

heap DeleteElement(heap H, element El)
if $(H == El)$
then
Deleting the Root of the Leftist tree.
$return(Merge(H.left, H.right))$
heap h1 = Merge($El.left$, $El.right$)
h1.parent=el.parent;
Correct the values of $El.parent$:
Change the $rank$, and ($left$, or $right$) fields if needed.
Propagate the changes up-wards if needed.
Physically Delete the element (El)
$return(H)$
End:DeleteElement

element FindMin(heap H)
$return(firstelementofH)$
End:FindMin

Bibliography

[Bro78] Mark R. Brown. Implementation and analysis of binomial queue algorithms. *SIAM Journal of Computing*, 7(3):312–319, August 1978.

[CLR92] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Instroduction to Algorithms.* McGraw-Hill Book Company, 1992.

[Cra72] Clark Allan Crane. *Linear Lists and Priority Queues as Balanced Binary Trees.* PhD thesis, Stanford University, 1972.

[CT76] David Cheriton and Robert Endre Tarjan. Finding minimum spanning trees. *Journal of Computing*, 5(4):724–742, December 1976.

[DGST88] James R. Driscoll, Harold N. Gabow, Ruth Shrairman, and Robert Endre Tarjan. Relaxed heaps: An alternative to fibonacci heaps with applications to parallel computation. *Communications of the ACM*, 31(11):1343–1354, November 1988.

[Gon84] Gaston H. Gonnet. *Handbook of algorithms and data structures.* Addison-Wesley Publishing Co., 1984.

[GW77] Adriano M. Garsia and Michelle L. Wachs. A new algorithm for minimum cost binary trees. *SIAM Journal of Computing*, 6(4):622–642, December 1977.

[HKT79] T. C. Hu, D. J. Kleitman, and J. K. Tamaki. Binary trees optimum under various criteria. *SIAM Journal of Applied Mathematics*, 37(2):246–256, April 1979.

[Hor84] Ellis Horowitz. *Fundamentals of Data Structures in Pascal.* Computer Science Press, Rockville, MD, 1984.

[HT71] T. C. Hu and A. C. Tucker. Optimal computer search trees and variable length alphabetic codes. *SIAM Journal of Applied Mathematics*, 21(4):514–532, December 1971.

[Hu71] T. C. Hu. A new proof of the t-c algorithm. *SIAM Journal of Applied Mathematics*, 25(1):83–94, December 1971.

[Hu82] T. C. Hu. *Combinatorial Algorithms.* Addison-Wesley Publishing Co., 1982.

[KLR] Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter. Correctness of constructing optimal alphabetic trees revisited. http://cs.uni-bonn.de/info5/publications/abstract/85134-cs.abstract-en.html.

[Knu73a] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Co., 1973. Fundamental Algorithms.

[Knu73b] Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley Publishing Co., 1973. Sorting and Searching.

[Knu81] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley Publishing Co., 1981. Seminumerical Algorithms.

[Lar90] Lawrence L. Larmore. A fast algorithm for optimal length-limited Huffman codes. *Journal of the Association for Computing Machinery*, 37(3):463–473, July 1990.

[LP98] Lawrence L. Larmore and Teresa Przytycka. The optimal alphabetic tree problem revisited. *Journal of Algorithms*, pages 1–20, 1998.

[LPR93] Lawrence L. Larmore, Teresa Przytycka, and Wojciech Rytter. Parallel construction of optimal alphabetic trees. *Journal of the Association for Computing Machinery*, pages 214–223, 1993.

[Mum92] Brendan M. Mumey. Some new results on constructing optimal alphabetic binary trees. Master's thesis, University of British Columbia, 1992.

[Tar83] Robert Endre Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1983.

[Vui78] Jean Vuillemin. A data structure for manipulating priority queue. *Communications of the ACM*, 21(4):309–315, April 1978.

[Wei95] Mark Allen Weiss. *Data Structures and Algorithm Analysis*. Benjamin/Cummings Publishing Co., 1995.

[Yoh72] J. Michael Yohe. Hu-Tucker minimum redundancy alphabetic coding methods [Z]. *Communications of the ACM*, 15(5):360–362, 5 1972. Algorithm 428.