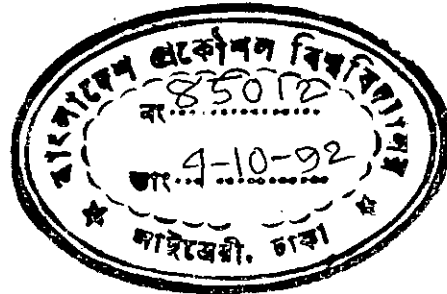


**PERFORMANCE ANALYSIS OF
SEARCHING ALGORITHMS**

**BY
CHOWDHURY MOFIZUR RAHMAN**

001.6423
1992
MOF

**A THESIS SUBMITTED TO THE DEPARTMENT OF
COMPUTER SCIENCE AND ENGINEERING, BUET, IN PARTIAL
FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE IN ENGINEERING**



**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY
JULY, 1992**



#85012#

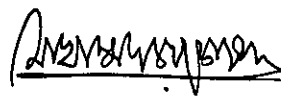
001.6423
1992
MOF

PERFORMANCE ANALYSIS
OF
SEARCHING ALGORITHMS

A thesis submitted by

CHOWDHURY MOFIZUR RAHMAN
Roll No. 891820P, Registration No. 82111,
for the partial fulfillment of the degree of
M.Sc. Engg. in Computer Science & Engineering.
Examination held on: July 28, 1992.

Approved as to style and contents by:


28/7/1992

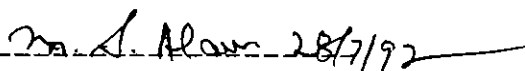
DR. MOHAMMAD KAYKOBAD
Assistant Professor,
Department of Computer Science & Engineering
B.U.E.T., Dhaka-1000, Bangladesh.

Chairman
and
Supervisor



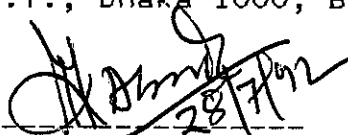
DR. SYED MAHBUBUR RAHMAN
Head,
Department of Computer Science & Engineering
B.U.E.T., Dhaka-1000, Bangladesh.

Member


28/7/92

DR. MD. SHAMSUL ALAM
Associate Professor,
Department of Computer Science & Engineering
B.U.E.T., Dhaka-1000, Bangladesh.

Member


28/7/92

DR. KAZI MOHIUDDIN AHMED
Associate Professor,
Department of Electrical & Electronic Engineering
B.U.E.T., Dhaka-1000, Bangladesh.

Member
(External)

CERTIFICATE OF RESEARCH

Certified that the work presented in this Thesis is the result of the investigation carried out by the candidate under the supervision of Dr. Mohammad Kaykobad at the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka, Bangladesh.

28.07.92

Date

Cyral

Signature of the
Candidate

DECLARATION

I do hereby declare that neither this thesis nor any part thereof has been submitted or is being concurrently submitted in candidature for any degree at any other university.

28.07.92

Date

C. M. S.

Signature of the
Candidate

ACKNOWLEDGEMENTS

The author would like to express his indebtedness and gratitude to his supervisor Dr. Mohammad Kaykobad, Assistant Professor of Computer Science and Engineering Department, BUET, for his endless patience, friendly supervision and invaluable assistance in making a difficult task a pleasant one.

The author wishes to express his thanks and regards to the Head of the Department of Computer Science and Engineering, BUET, for his support during the work. Sincerest thanks to friends and colleagues for their constant support and criticism of the research work.

ABSTRACT

In this work various searching algorithms have been studied both theoretically and experimentally. There are a large number of searching algorithms: some of them are suitable for static tables, some other are suitable for dynamic tables with frequent insertions and deletions, and yet some other employ transformation of keys to locate an item in a table. For the majority of algorithms presented in this thesis different variations of each category have been compared in terms of computational cost. Therefore, there is a scope to select the best one among a number of alternatives for a particular application. Attempts have been made to give detailed mathematical analysis of each algorithm, and in cases where exact mathematical analysis is lacking, empirical results have been obtained through experiment. Furthermore, mathematical analysis of each algorithm has been supported by experimental results.

This thesis begins with the algorithms dealing with static tables. The Binary search algorithm along with different variations thereof have been dealt with both analytically and experimentally. In order to make Binary search algorithm faster, another algorithm called Fibonacci search which uses the Fibonacci tree as its decision tree and which avoids the division operation in its implementation have also been introduced along with other search algorithms for static table.

The behaviour of a dynamic search tree assuming the input keys are random has been extensively studied both theoretically and experimentally. There has been an endeavour to analyze the behaviour of a random search tree after random deletions made on it. Hibbard's theorem on the behaviour of random search trees obtained from

deleting a key randomly has been proved to be incorrect. The detailed procedure to build up an optimal search tree given the successful and unsuccessful frequencies of each key has been studied extensively and verified experimentally.

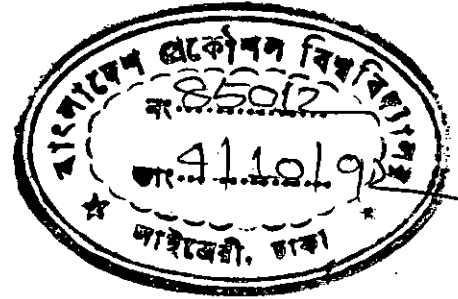
In order to have a guarantee against bad worst case performance when the search keys are nonrandom, two restricted tree structures namely Balanced tree and 2-3-4 tree have been introduced. Unfortunately the average behaviour of these trees are still unknown. The empirical results about the average behaviour of these trees has been found out through simulation experiments.

Beyond the searching algorithms which work by comparison between keys, a new dimension in this arena has been added recently by a new class of algorithms called Hashing. This class of algorithms have been extensively studied and verified through both mathematical analysis and experiments.

CONTENTS

INTRODUCTION.	1
CHAPTER 1. SEARCHING BY COMPARISON OF KEYS	9
1.1 Introduction	9
1.2 Binary Search	10
1.3 Analysis of Binary Search Algorithm	11
1.4 Variation of Binary Search Algorithm	15
1.5 Fibonacci Search	20
CHAPTER 2. SEARCH TREES	25
2.1 Introduction	25
2.2 Dynamic Trees	27
2.3 Dynamic Tree Analysis	30
2.4 Deletion from Dynamic Tree	34
2.4.1 Analysis of Deletion	36
2.5 Static Trees	40
2.6 Heuristics on Optimality	46
CHAPTER 3. BALANCED TREE	49
3.1 Introduction	49
3.2 Definition of a Height Balanced Tree	50
3.2.1 Height of a Balanced Tree	50
3.3 Balanced Tree Search, Insertion and Deletion	54
3.4 Balanced Tree Search and Insertion without using stack	60
3.5 Some interesting empirical results about Balanced Trees	66
3.6 2-3-4 Trees	68
3.7 RED BLACK Trees	74

CHAPTER 4. HASHING TECHNIQUES	95
4.1 Introduction	95
4.2 Collision Resolution	96
4.3 Chaining	97
4.4 Linear Probing	106
4.5 Double Hashing	116
4.6 Brent's Algorithm	118
4.7 Ordered Hash table	122
4.8 Improvement with additional memory	126
4.9 Hashing in external storage	129
4.10 The Separator method	131
4.11 Dynamic Hashing and Extendible Hashing	133
4.12 Choosing a Hash function	138
CHAPTER 5. RESULTS AND CONCLUSIONS	143
5.1 Introduction	143
5.2 Searching by comparison of keys	143
5.3 Search Trees	154
5.4 Balanced Trees	177
5.5 Hashing Techniques	183
5.6 Conclusions	206
5.7 Suggestion for further study	208
BIBLIOGRAPHY.	210



INTRODUCTION

GENERAL.

A fundamental operation intrinsic to a great many computational tasks is searching: retrieving some particular piece or pieces of information from a large amount of previously stored information. Applications of searching are widespread and involve a variety of different operations. For example, a bank needs to keep track of all its customers' account balances and to search through them to check various types of transactions. An airline reservation system has similar demands, in some ways, but most of the data is rather short-lived. Searching is the most time consuming part of many programs and the substitution of a good search method for a bad one often leads to a substantial increase in speed. Statistical data shows that about two thirds of computation time is spent for searching. Since searching is such a common task in computing, a knowledge of search algorithms goes a long way toward making a good programmer.

Searching may be classified in several ways. We might divide them into static vs. dynamic searching, where "static" means that the contents of the search table are essentially unchanging (so that it is important to minimize the search time without regard for the time required to set up the table), and "dynamic" means that the table is subject to frequent insertions (and perhaps also deletions). Another possible scheme is to classify search methods according to whether they are based on comparisons between the keys or on digital properties of the keys. A third possibility is to divide search methods into internal vs. external searching. Searches in which the entire table is constantly in main memory are called internal searches, whereas those in which most

of the table is kept in external storage are called external searches. Finally we might classify searching into those methods which use the actual keys and those which work with transformed keys.

Let us now bring together the terminology commonly used in searching. Throughout this thesis we will assume that a table or a file is a group of elements, each of which is called a record. Associated with each record is a key, which is used to differentiate among different records. The association between a record and its key may be simple or complex. In the simplest form, the key is contained within the record at a specific offset from the start of the record. Such a key is called an internal key or an embedded key. In other cases there is a separate table of keys that includes pointer to the records. Such keys are called external. A search algorithm is an algorithm that accepts an argument K and tries to find a record whose key is K . The algorithm may return the entire record or, more commonly, it may return a pointer to that record. The task of comparing the argument key with a table location is called a probe. It is possible that the search for a particular argument in a table is unsuccessful; that is, there is no record in the table with that argument as its key. Very often, if a search is unsuccessful it may be desirable to add a new record with the argument as its key. An algorithm that does this is called a search and insertion algorithm.

Note that we have said nothing about the manner in which the table or file is organized. It may be an array of records, a linked list, a tree or even a graph. Because different search techniques may be suitable for different table organizations, a table is often designed with a specific search technique in mind. The table may be contained completely in memory, completely in auxiliary storage or it may be divided between the two. Clearly different search techniques are necessary under these different assumptions.

HISTORICAL PERSPECTIVE.

Before making any further comment on search algorithms, it may be helpful to put things in historical perspective. It is the Binary search algorithm mentioned by John Mauchly what was perhaps the first published discussion of nonnumerical programming methods. The method became well known, but nobody seems to have worked out the details of what should be done in general situations. H. Bottenbruch [JACM 9(1962), 214] was apparently the first to publish a binary search algorithm which works for all N . He presented an interesting variation which avoids a separate test for equality until the very end. K. E. Iverson [A Programming Language (Wiley, 1962), 141] gave the detailed procedure of Algorithm Binary search, but without considering the possibility of an unsuccessful search. D. E. Knuth [CACM 6(1963), 556-558] presented Binary search algorithm as an example used with an automated flowcharting system. The Uniform Binary search algorithm was suggested by A. K. Chandra of Stanford university in 1971. Fibonacci search was invented by David E. Ferguson [CACM 3(1960), 684], but his flowchart and analysis was incorrect.

The first published descriptions of tree insertion were by P. F. Windley [Comp. J. 3(1960), 84-88], A. D. Booth and A. J. T. Colin [Information and Control 3(1960), 327-334], and Thomas N. Hibbard [JACM 9(1962), 13-28]. All three of these authors seem to have developed the method independently of one another and all three authors gave some different proofs of the average number of comparisons. The three authors also went on to treat different aspects of the algorithm: Windley gave a detailed discussion of tree insertion sorting; Booth and Colin discussed the effect of preconditioning by making the first $2^n - 1$ elements form a perfectly balanced tree; Hibbard introduced the idea of deletion and showed the connection between the analysis of tree insertion and

analysis of quicksort.

The idea of optimum binary search tree was first developed for the special case $p_1 = \dots = p_n = 0$, in the context of alphabetic binary encoding. A very interesting paper by E. N. Gilbert and E. F. Moore [Bell system Tech. J. 38(1959), 933-968] discussed this problem and its relation to other coding problems. Gilbert and Moore observed, among other things, that an optimum tree could be constructed in $O(n^3)$ steps. K.E. Iverson [A Programming Language (Wiley, 1962), 142-144] independently considered the other case, when all the q 's are zero. He suggested that an optimum tree would be obtained if the root is chosen so as to equalize the left and right subtree probabilities as much as possible; unfortunately it was seen that this idea does not work. D. E. Knuth [20] subsequently considered the case of general p and q weights and proved that the algorithm could be reduced to $O(n^2)$ steps; he also presented an example from a compiler application, where the keys in the tree are reserved words in an ALGOL like language. T. C. Hu had been studying his own algorithm for the $p=0$ case for several years; a rigorous proof of the validity of that algorithm was difficult to find because of the complexity of the problem, but eventually obtained a proof jointly with A. C. Tucker in 1969 [SIAM J. Applied Math. 21(1971), 514-532].

C. C. Foster [11] has studied the generalized balanced trees which arise when we allow the height difference of subtrees to be greater than one, but at most four (say). Another interesting alternative to balanced trees, called 2-3 trees was introduced by John Hopcroft in 1970 (unpublished). The idea is to have either 2-way or 3-way branching at each node, and to stipulate that all external nodes appear on the same level. Hopcroft has observed that deletion, concatenation and splitting can all be done with 2-3 trees, in a reasonably straight forward manner analogous to the corresponding operations with

balanced trees. R. Bayer [Proc. ACM - SIGFIDET workshop (1971), 219-235] has suggested an interesting binary tree representation for 2-3 trees. The concept of 2-3-4 trees is attributed to Guibas and Sedgewick's 1978 paper [14] which shows how to fit many classical balanced tree algorithms into the red-black framework and gives several other implementations.

Hash coding was first published in the open literature by Arnold I. Dumey, Computers and Automation 5, 12(December, 1956), 6-9. He was the first to mention the idea of dividing by a prime number and using the remainder as the hash address. Dumey's interesting article mentioned chaining but not open addressing. Robert Morris [26] wrote a very influential survey of the subject in which he introduced the idea of random probing. Morris' paper touched off a flurry of activity which culminated in Double hashing algorithm and its refinements. A comprehensive discussion of hash functions has been introduced by Knott, G. D., [17] and a complete analysis of ordered hash tables has been carried out by Amble, O. and D. E. Knuth [3]. Guibas, L. J. and E. Szemerédi [13] have analyzed the Double hashing method in their 1978 paper. A different reordering scheme, attributable to Brent [6], can be used to improve the average search time for successful search when Double hashing is used. Brent's method reduces the average number of comparisons for successful retrievals but has no effect on the number of comparisons for unsuccessful searches. Nishihara, S. and K. Ikeda in their paper [27] focussed the idea of reducing the retrieval time by using predictor and their method is particularly applicable to linear probing. The advantage of their method is that it can be adapted quite easily when only a few extra bits are available in each table position.

One technique for reducing access time in external hash tables at the expense of increasing insertion time is attributable to Larson [23]. His algorithm ensures the ability

to access any record in the file with only a single external memory access. The extendible hashing algorithm comes from Fagin, Nievergelt, Pippenger and Strong's 1979 paper [10]. This paper is a must for anyone wishing further information on external searching algorithms. The paper also contains a detailed analysis and a discussion of practical ramifications.

THESIS ORGANIZATION AND OBJECTIVE.

This thesis comprises five chapters. Chapter one discusses the improvements which can be made over sequential methods of searching, based on comparison between keys, using alphabetic or numeric order to govern the decisions. Therefore, in this chapter we shall concentrate on methods which are appropriate for searching a static table whose keys are in order making random accesses to the table entries.

The methods of chapter one are appropriate mainly for fixed size tables, since the sequential allocation of records makes insertions and deletions rather expensive. If the table is dynamically changing, we might spend more time maintaining it using the methods of chapter one than searching it. As a result chapter two evolves to facilitate the searching in a dynamic table. In this chapter we will discuss the search, insertion and deletion algorithms using dynamic data structure and at the very end of this chapter we will enlighten an algorithm to build an optimal search tree of minimum cost.

The algorithms presented in chapter two work very well for a wide variety of applications, but they do have the problems of bad worst case performance depending on the nature of the keys. The algorithms presented in chapter three is an endeavour to provide insurance against a bad worst case performance at relatively little cost. Two different tree structures have been used to achieve the aforesaid goal.

Chapter four discusses an important class of methods called hashing techniques.

based on arithmetic transformation of the actual keys. This contrasts sharply with the techniques presented in previous chapters which were based on comparisons but in this chapter we discuss techniques based on directly transforming the keys into an address at which it will be stored.

Chapter five is intended to present the experimental results based on the algorithms discussed so far in the previous chapters. This chapter compares the various searching strategies based on the experimental results and verifies the theoretical aspects of each algorithm.

This thesis is an attempt to walk through the realm of searching. But the field of searching is so enormous in its entirety that only some selective searching algorithms having wide range of applicability have been chosen for the topics of this work. There exists a large number of searching algorithms without exact theoretical analysis. In such cases we have tried to investigate the behaviour of such algorithms in terms of time and space complexity and developed simulation programs in order to have some empirical results. For example, deletion in a dynamic binary search tree does not have sufficient theoretical analysis for its performance. We have, in this case, tried to explain the behaviour of deletion algorithm in a simplified way. In particular we have shown that Hibbard's theorem related to deletion of an element from a randomly generated tree is incorrect. We have pointed out in chapter two what is wrong with his theorem and why his theorem does not reflect the true practical situation. Again the average behaviour of balanced tree algorithms discussed in chapter three is unknown. Here also we have investigated, through simulation experiments, the average behaviour of balanced tree algorithms. We have also investigated the average performance of 2-3-4 trees and have enlightend some empirical behaviour.

Nearly all of the algorithms in this thesis have been verified extensively to see if they actually follow the theoretical results if there is any. Many of the algorithms have similar characteristics such that there are many alternatives to use one of them in a particular application. This thesis, among many other things, will help to choose the best of them for optimum performance of a particular application. Exact mathematical analysis for the majority of the algorithms have been presented along with the constraints and limitations thereof so that the reader can easily aware himself of the suitability of a particular algorithm in a particular application. All of the simulation experiments have been carried out on IBM PC compatible machines having 80286 processor of 16 MHz speed.

Almost every branches of searching have been travelled through by this thesis, but emphasis has been given primarily on internal searching; however we mention some techniques of external searching when they relate closely to the methods we study.

CHAPTER 1

SEARCHING BY COMPARISON OF KEYS

1.1 Introduction.

In this chapter we shall discuss search methods which are based on a linear ordering of the keys. When a large number of records must be searched sequential scanning is out of the question and an ordering relation simplifies the job enormously. Of course, if we only need to search a few times, it is faster to do a sequential search than to do a complete sort of the records; but if we need to make repeated searches in the same records, we are better off having these in order. Therefore in this chapter we shall concentrate on methods which are appropriate for searching a table whose keys are in order,

$$K_1 < K_2 < K_3 \dots \dots \dots < K_N,$$

making random accesses to the table entries. After comparing a key K to K_i in such a table we either have

- * $K < K_i$ [R_i, R_{i+1}, \dots, R_N are eliminated from consideration];
- or * $K = K_i$ [the search is done];
- or * $K > K_i$ [R_1, R_2, \dots, R_i are eliminated from consideration].

In each of these cases, substantial progress has been made, unless i is near one end of the table; this is why the ordering leads to an efficient algorithm. The sequential search method is essentially limited to a two way decision ($K = K_i$ vs. $K \neq K_i$), but if we free ourselves from this restriction of sequential access it becomes possible to make effective use of an order relation.

1.2 Binary Search.

If the set of records is large, then the total search time can be significantly reduced by using a search procedure based on the application of divide and conquer paradigm: divide the set of records into two parts, determine which of the two parts the key sought belongs to, then concentrate on that part. A reasonable way to divide the set of records into parts is to keep the records sorted, then use indices into the sorted array to delimit the part of the array being worked on. To find if a given key K is in the table, first compare it with the element at the middle position of the table. If K is smaller, then it must be in the first half of the table; if K is greater, then it must be in the second half of the table. Then we can apply this method recursively. The binary search algorithm makes use of two pointers, left and right, which indicates the current lower and upper limits of the search, as follows:

ALGORITHM 1.1 (Binary Search):

Given a table of records R_1, R_2, \dots, R_N whose keys are in increasing order $K_1 < K_2 < \dots < K_N$, this algorithm searches for a given argument K .

```
left = 1; right = N; [[ Initialize ]]  
  
while ( right ≥ left ) do  
  
    middle = [( left + right ) / 2 ] [[ Get midpoint ]]  
  
    if (  $K < K_{middle}$  ) right = middle - 1 [[ Adjust right ]]  
  
    else if (  $K > K_{middle}$  ) left = middle + 1 [[ Adjust left ]]  
  
    else return [[ the algorithm terminates successfully ]]  
  
endif
```

repeat

exit [[At this point the algorithm terminates unsuccessfully]]

End Algorithm 1.1

Tree Representation of Binary Search:

In order to really understand what is happening in algorithm 1.1, it is best to think of it as a binary decision tree as shown in Fig 1.1 for the case $N = 16$. When $N = 16$, the first comparison made by the algorithm is $K : K_8$; this is represented by the root node (8) in the figure. Then if $K < K_8$, the algorithm follows the left subtree, comparing K to K_4 ; Similarly if $K > K_8$ the right subtree is used. An unsuccessful search will lead to one of the external square nodes numbered 0 through 16; for example we reach [5] if and only if $K_5 < K < K_6$.

1.3 Analysis of Binary Search Algorithm.

It is evident from the decision tree of Fig. 1.1 that the number of records is at least halved at each step and consequently this method of searching never uses more than $\log_2^N + 1$ comparisons for either successful or unsuccessful search. If $2^{k-1} \leq N < 2^k$, a successful search requires (min 1, max K) comparisons. If $N = 2^k - 1$, an unsuccessful search requires K comparisons; and if $2^{k-1} \leq N < 2^k - 1$, an unsuccessful requires either K - 1 or K comparisons. An upper bound on the number of comparisons satisfies the recurrence $C_N = C_{N/2} + 1$ with $C_1 = 1$, which implies the stated result.

The tree representation also shows us how to compute the average number of comparisons in a simple way. Let S_N be the average number of comparison in a successful search, assuming that each of the N keys is an equally likely argument; and let U_N be the average number of comparisons in an unsuccessful search, assuming that each of the

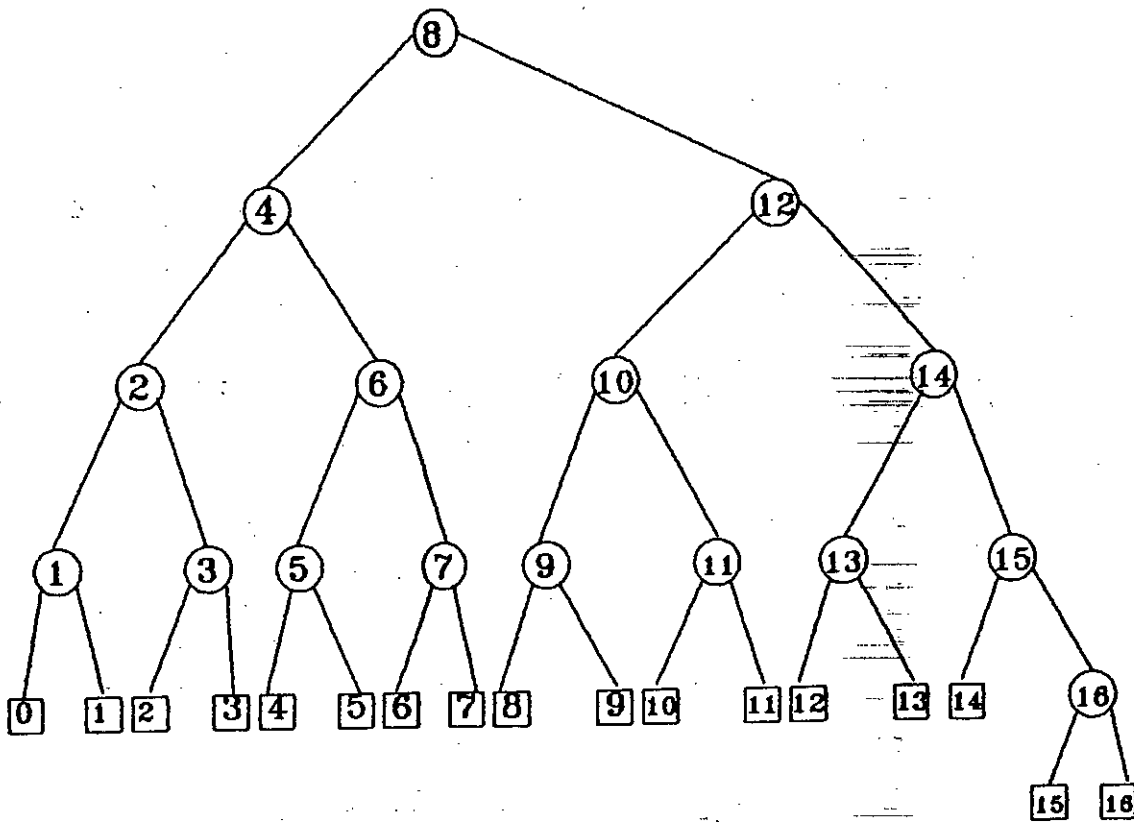


Fig. 1.1 A binary tree corresponding to binary search with $N = 16$

$N + 1$ intervals between keys is equally likely. Then we have

$$U_N = \frac{\text{Sum of levels of the } (N + 1) \text{ external nodes}}{N + 1}$$

$$S_N = \frac{\text{Sum of } (1 + \text{level of each of the internal nodes})}{N}$$

Let us define the external and internal path length of a search tree. External path length $E(T)$ is the sum of levels of all the external nodes whereas internal path length $I(T)$ is the sum of levels of all the internal nodes assuming that the level of the root is zero.

$$\text{Hence } U_N = \frac{E(T)}{N + 1} \text{ and } S_N = 1 + \frac{I(T)}{N}$$

From the above relations we see that the best way to search by comparison is one for which corresponding search tree has minimum path length, over all binary trees with N internal nodes. A binary tree has minimum path length if and only if all its external nodes occur in at most two adjacent levels. Let us verify this statement by considering the problem of discovering such a tree with N nodes having minimum path length. Clearly only one node (root) can be zero distance from the root; at most two nodes can be at a distance one from the root, at most four can be two away, etc. So the internal path length is always at least as big as the sum of the first N terms of the series

$$0, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3, 4, 4, 4, 4, \dots$$

Therefore the tree all external nodes of which occur in at most two adjacent levels has minimum path length among all possible trees. In the binary search procedure all the external nodes appear at two adjacent levels of the tree making external path length of the tree minimum.

If l_1, l_2, \dots, l_{N+1} be the levels of the $(N + 1)$ external nodes in a binary tree then it can be shown by induction that

$$\sum_{i=1}^{N+1} 2^{-l_i} = 1 \quad (1.1)$$

Now we are ready to compute the minimum external path length of a binary search tree with N internal nodes and $(N + 1)$ external nodes. Let there be K external nodes on level l and $N + 1 - K$ on level $l + 1$, $l \leq K \leq N + 1$ (that is, all the external nodes may be on level l). From equation (1.1) we can write

$$K 2^{-l} + (N + 1 - K) 2^{-l-1} = 1 \text{ and hence } \dots$$

$$K = 2^{l+1} - (N + 1) \quad (1.2)$$

Since $K \geq 1$, $2^{l+1} > N + 1$ and since $K \leq N + 1$, we have $2^l \leq N + 1$; that is

$$l = \lfloor \log_2(N + 1) \rfloor \quad (1.3)$$

Combining (1.2) and (1.3) gives $K = 2^{\lfloor \log_2(N+1) \rfloor + 1} - N - 1$ and the minimum path length is thus

$$lK + (l+1)(N+1-K) = (N+1) \lfloor \log_2(N+1) \rfloor + 2(N+1) - 2^{\lfloor \log_2(N+1) \rfloor + 1}$$

Let $\theta = \log_2(N + 1) - \lfloor \log_2(N + 1) \rfloor$, $0 \leq \theta < 1$, the minimum external path length becomes

$$(N + 1) \log_2(N + 1) + (N + 1)(2 - \theta - 2^{1-\theta}) \quad (1.4)$$

The function $f(\theta) = 2 - \theta - 2^{1-\theta}$ is small in the interval $0 < \theta < 1$; more precisely $0 \leq f(\theta) \leq 0.0861$ in this interval. In the light of the above computations we conclude that

$$U_N = \frac{E(T)}{N+1} = \log_2(N+1) + 2 - \theta - 2^{1-\theta} \text{ and}$$

$$S_N = 1 + \frac{I(T)}{N} = (1 + \frac{1}{N})U_N - 1$$

$$= (1 + \frac{1}{N}) [\log_2(N + 1) + 2 - \theta - 2^{1-\theta}] - 1$$

where $\theta = \log_2(N + 1) - \lfloor \log_2(N + 1) \rfloor$ To summarize: algorithm 1.1 never makes more than $\lfloor \log_2 N \rfloor + 1$ comparisons and it makes about $\log_2 N - 1$ comparisons in an average successful search.

It is important to note that the time required to insert new records is high for binary search: the array must be kept sorted, so some records must be moved to make room for any new record. A random insertion requires that $N/2$ records be moved, on the average. Thus it is best suited for situations in which the table can be built ahead of time, perhaps using shellsort or quicksort algorithms, and then used for a large number of searches.

1.4 Variation of Binary Search algorithm.

Instead of using three pointers left, right and middle in the search we can use only two, the current position i and the rate of change of δ ; after each unequal comparison, we could then set $i = i \pm \delta$ and $\delta = \frac{\delta}{2}$. Using this approach the following variation of Binary Search results:

Algorithm 1.2 (Uniform Binary Search)

Given a table of records R_1, R_2, \dots, R_N whose keys are in increasing order $K_1 < K_2 < \dots < K_N$, this algorithm searches for a given argument K . If N is even, the algorithm will sometimes refer to a dummy key K_0 which should be set $-\infty$. We assume that $N > 1$.

$$i = \lfloor N/2 \rfloor; \quad m = \lfloor N/2 \rfloor \quad \text{[[Initialize]]}$$


```

while ( $m \neq 0$ ) do
    if ( $K < K_i$ ) then  $i = i - \lfloor m/2 \rfloor$ ;  $m = \lfloor m/2 \rfloor$            [[ Decrease i ]]
    else if ( $K > K_i$ ) then  $i = i + \lfloor m/2 \rfloor$ ;  $m = \lfloor m/2 \rfloor$        [[ Increase i ]]

    else return                [[ the algorithm terminates successfully ]]

endif

repeat
    exit                [[ At this point the algorithm terminates unsuccessfully ]]

```

End Algorithm 1.2

Fig. 1.2 shows the corresponding binary tree for the search when $N = 10$. In an unsuccessful search, the algorithm may make a redundant comparison just before termination; these nodes have been labeled in the figure. This search has been termed uniform because the difference between the number of a node on level l and the number of its ancestor on level $l-1$ has a constant value δ for all nodes on level l . It is to be observed that the lengths of two intervals at the same level differ by at most unity; this makes it possible to choose an appropriate middle element, without keeping track of the exact lengths. The principal advantage of algorithm 1.2 is that we need not maintain the value of m at all; we need only refer to short table of the various δ to use at each level of the tree. Thus the running time decreases considerably as compared to binary search. In a successful search, this algorithm corresponds to a binary tree with the same internal path length as the tree of algorithm Binary Search, so the average number of comparisons for successful search is the same as before. In an unsuccessful search algorithm 1.2 always makes exactly $\lfloor \log_2 N \rfloor + 1$ comparisons. Another modification of Binary Search is one

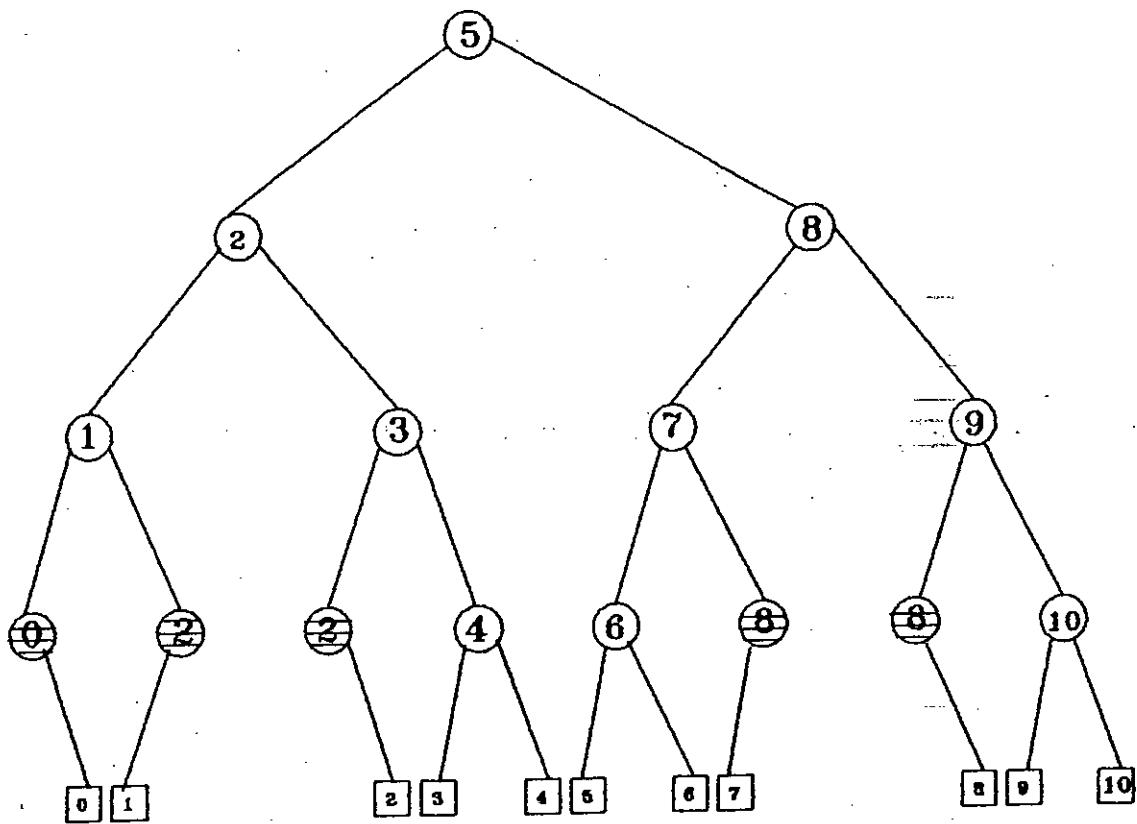


Fig. 1.2 Uniform search tree corresponding to $N = 10$

in which it is uniform after the first step and it is still faster than the Uniform Binary search. The first step is to compare K with K_i , where $i = 2^k$, $K = \lfloor \log_2 N \rfloor$. If $K < K_i$, we use a uniform search with the δ 's equal to $2^{k-1}, 2^{k-2}, \dots, 1, 0$. On the other hand if $K > K_i$ and $N > 2^k$, we set i to $i' = N + 1 - 2^l$, where $l = \lfloor \log_2(N - 2^k) \rfloor + 1$, and pretend that the first comparison was actually $K > K'_i$, using a uniform search with the δ 's equal to $2^{l-1}, 2^{l-2}, \dots, 1, 0$. The binary tree for this variation is shown in figure 1.3. Like previous algorithms this method never makes more than $\lfloor \log_2 N \rfloor + 1$ comparisons, but it occasionally goes through several redundant steps in succession.

An interesting variation of algorithm binary search is that we can avoid a separate test for equality until the very end of the algorithm. This algorithm is as follows:

Algorithm 1.3

```

    l = 1;          u = N          [[Initialize]]
    while (u ≠ l) do
        i = [(l + u)/2]
    if (K < Ki)    then u = i - 1 endif    [[ Adjust u ]]
    else if (K ≥ Ki) then l = i          [[ Adjust l ]]
    endif
    repeat

```

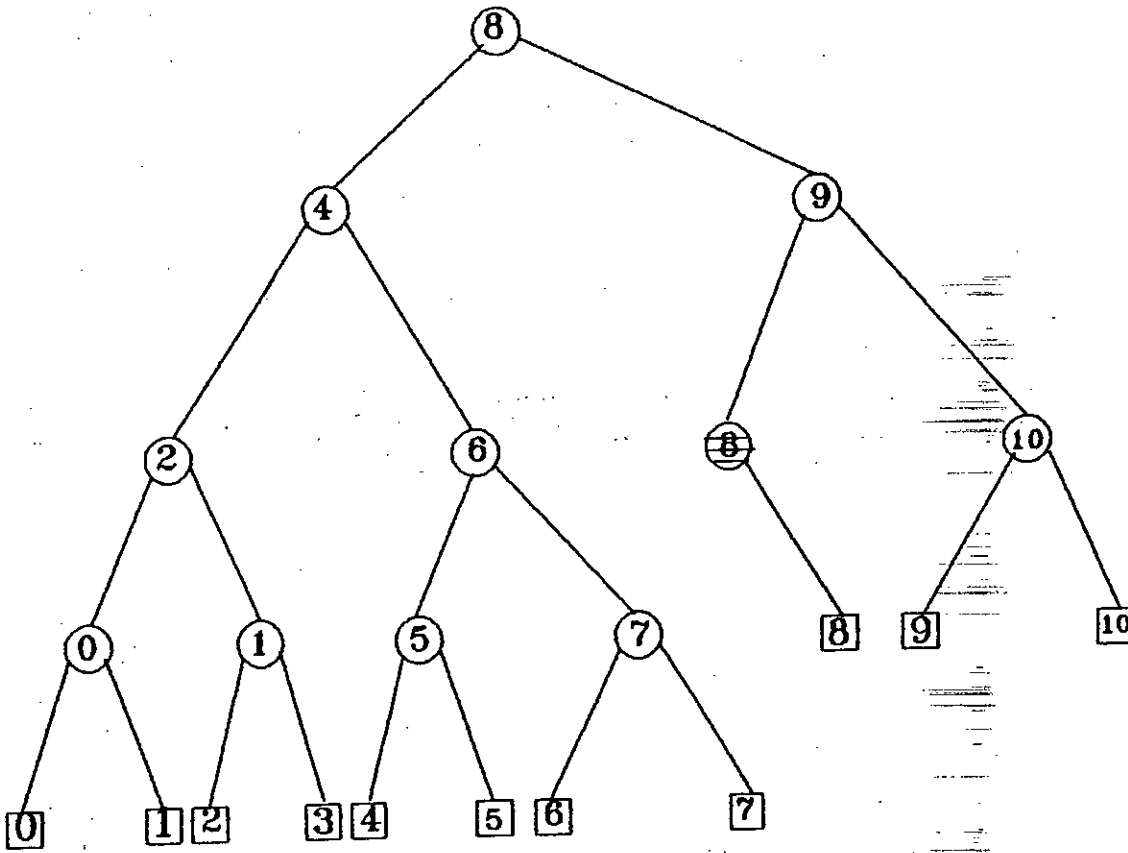


Fig. 1.3 Tree corresponding to variation of Uniform search

```

if ( $K = K_l$ ) then return; [[ The search is successful ]]
else exit [[ At this point the algorithm terminates unsuccessfully ]]
endif

```

End Algorithm 1.3.

In the above algorithm using $i = \lfloor (l + u)/2 \rfloor$ inside the while loop we have set $l = i$ whenever $K \geq K_i$; then $u - l$ decreases at every step. Eventually when $l = u$, we have $K_l \leq K < K_{l+1}$ and we can test whether or not the search was successful by making one more comparison. Such a trick will make Binary search a little bit faster for large N . We would need $N > 2^{36}$ in order to compensate for the extra iteration necessary.

1.5 Fibonacci Search.

Fibonacci numbers provide us with an alternative to binary search and it involves only addition and subtraction, not division by two. So it may be preferable on some computers. If we start to explain the method simply from programming point of view, it seems to work by magic. But the mystery disappears as soon as the corresponding search tree is displayed.

In general the Fibonacci tree of order K has $F_{k+1} - 1$ internal nodes and F_{k+1} external nodes and it is constructed as follows:

If $K = 0$ or $K = 1$, the tree is simply $[0]$. If $K \geq 2$, the root is (F_k) ; the left subtree is the Fibonacci tree of order $K - 1$ and the right subtree is the Fibonacci tree of order $K - 2$ with all numbers increased by F_k . Using this rule we can construct the tree of figure 1.4 of order 6. From the tree structure it is evident that the numbers on the two

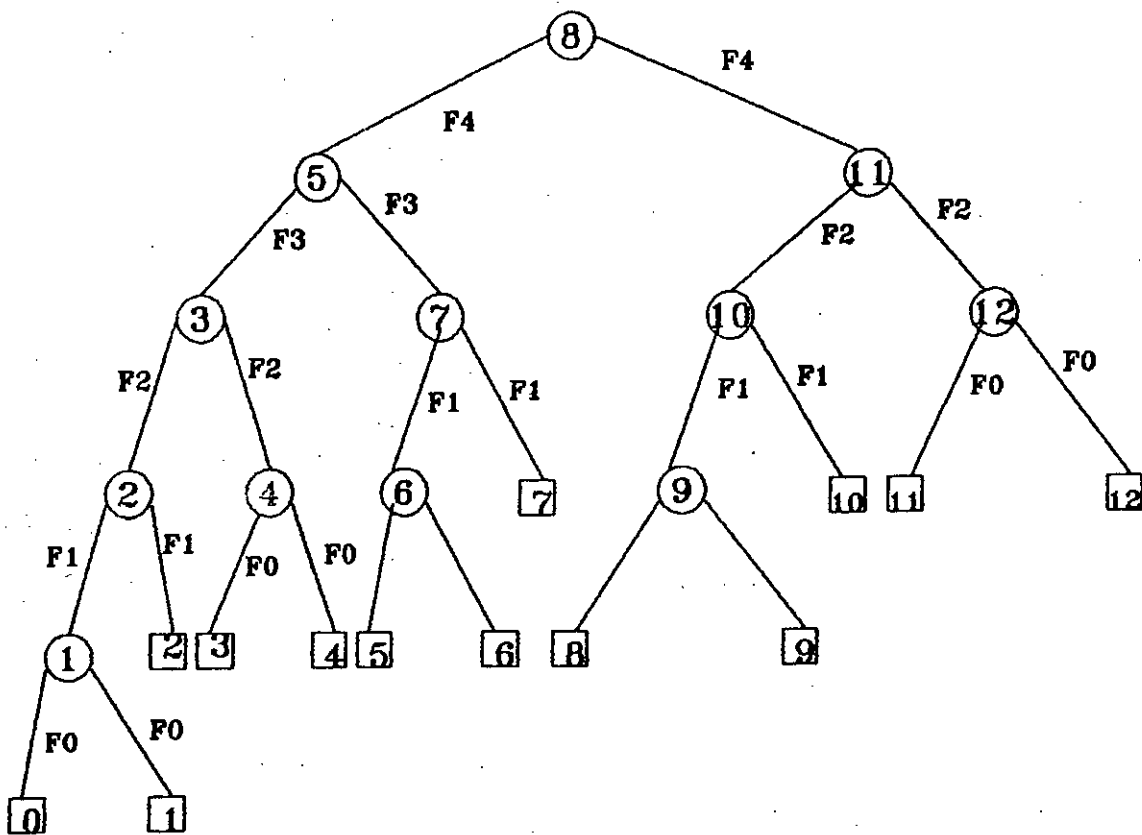


Fig. 1.4 A Fibonacci tree of order 6

nodes of each internal node differ from the father's number by the same amount and the amount is a Fibonacci number. Thus $3 = 5 - F_3$ and $7 = F_3 + 5$ in fig. 1.4. When the difference is F_i , the corresponding Fibonacci difference for the next branch on the left is F_{i-1} while on the right it skips down to F_{i-2} . These observations can be combined with an appropriate mechanism for recognizing the external nodes and initialization we can arrive at the following algorithm.

Algorithm 1.4 (Fibonacci Search)

Given a table of records R_1, R_2, \dots, R_N whose keys are in increasing order $K_1 < K_2 < \dots < K_N$, this algorithm searches for a given argument K . For convenience in description, this algorithm assumes that $N + 1$ is a perfect Fibonacci number, F_{k+1} . It is not difficult to make the method work for arbitrary N , if a suitable initialization is provided which will be focused after the description of the algorithm.

$$i = F_k, \quad p = F_{k-1}, \quad q = F_{k-2}$$

[[Throughout the Algorithm, p and q will be consecutive Fibonacci numbers]]

while ($K \neq K_i$) do

```

if ( $K < K_i$ ) then
if ( $q = 0$ ) then exit; endif [[ The algorithm terminates unsuccessfully ]]
 $i = i - q$ ;   ( $p, q$ )  $\leftarrow$  ( $q, p - q$ ) [[ Decrease  $i$  ]]
else if ( $K > K_i$ ) then
if ( $p = 1$ ) then exit; endif [[ The algorithm terminates unsuccessfully ]]
 $i = i + q$ ;  $p = p - q$ ;  $q = q - p$  [[ Increase  $i$  ]]
endif
endif

repeat

```

End Algorithm 1.4

If $N + 1$ is not a perfect Fibonacci number then we have to find the least $M \geq 0$ such that $N + M$ has the form $F_{k+1} - 1$, then to start with $i = F_k - M$ and to insert "if $i \leq 0$, $i = i + q$; $p = p - q$; $q = q - p$ " at the very beginning of the first if structure inside the while loop in the above algorithm. Another idea is to check the result of the very first comparison. If it comes true that $K > K_{F_1}$, then we can set $i = i - M$ and proceed normally thereafter.

The number of comparisons required in binary search is approximately $\log_2 N$ whereas in Fibonacci search it is approximately $(\phi/\sqrt{5}) \log_\phi N$. Fig. 1.4 shows that a left branch is taken somewhat more often than a right branch - which we might have guessed, since each probe divides the remaining interval into two parts, with the left part about ϕ times as large as the right.

The external nodes appear on levels $\lfloor K/2 \rfloor$ through $K - 1$ in the Fibonacci tree of order K . The difference between these levels is greater than unity except when $K = 0$.

1. 2. 3. 4. Thus for this values of K , the Fibonacci tree offers an optimal search path in the sense that fewest comparisons are made on the average.

CHAPTER 2

SEARCH TREES

2.1 Introduction.

As we saw in chapter one, the order in which the elements are examined by binary search is governed by an implicit binary tree on the table elements. In this chapter we will discuss the benefits of making such a binary tree structure explicit instead of implicit. These benefits are two fold. For tables in which the elements do not change through insertions and deletions (static tables), an explicit tree structure can be used to take advantage of a known distribution of the frequency access of the elements. For dynamic tables that change through the insertions and deletions, an explicit tree structure gives us the flexibility to search the table in logarithmic time and to make insertion and deletion also in logarithmic time.

A binary search tree is a binary tree in which the inorder traversal of the node gives the elements stored therein in the natural order. In other words, every node P in the tree has the property that elements in its left subtree are before $KEY(P)$ in the natural order and those in its right subtree after $KEY(P)$ in the natural order. Figure 2.1 shows a binary search tree for the set of names { A, E, I, O, U }.

If we now search for K, starting at the root or apex of the tree, we find it is less than O, so we move to the left; it is greater than E, so we move to right; it is greater than I, so we move to the right again and arrive at an external node. The search was unsuccessful. In a similar way we can search for any of the existing keys in the tree and in that case the search will be successful terminating at an internal node.

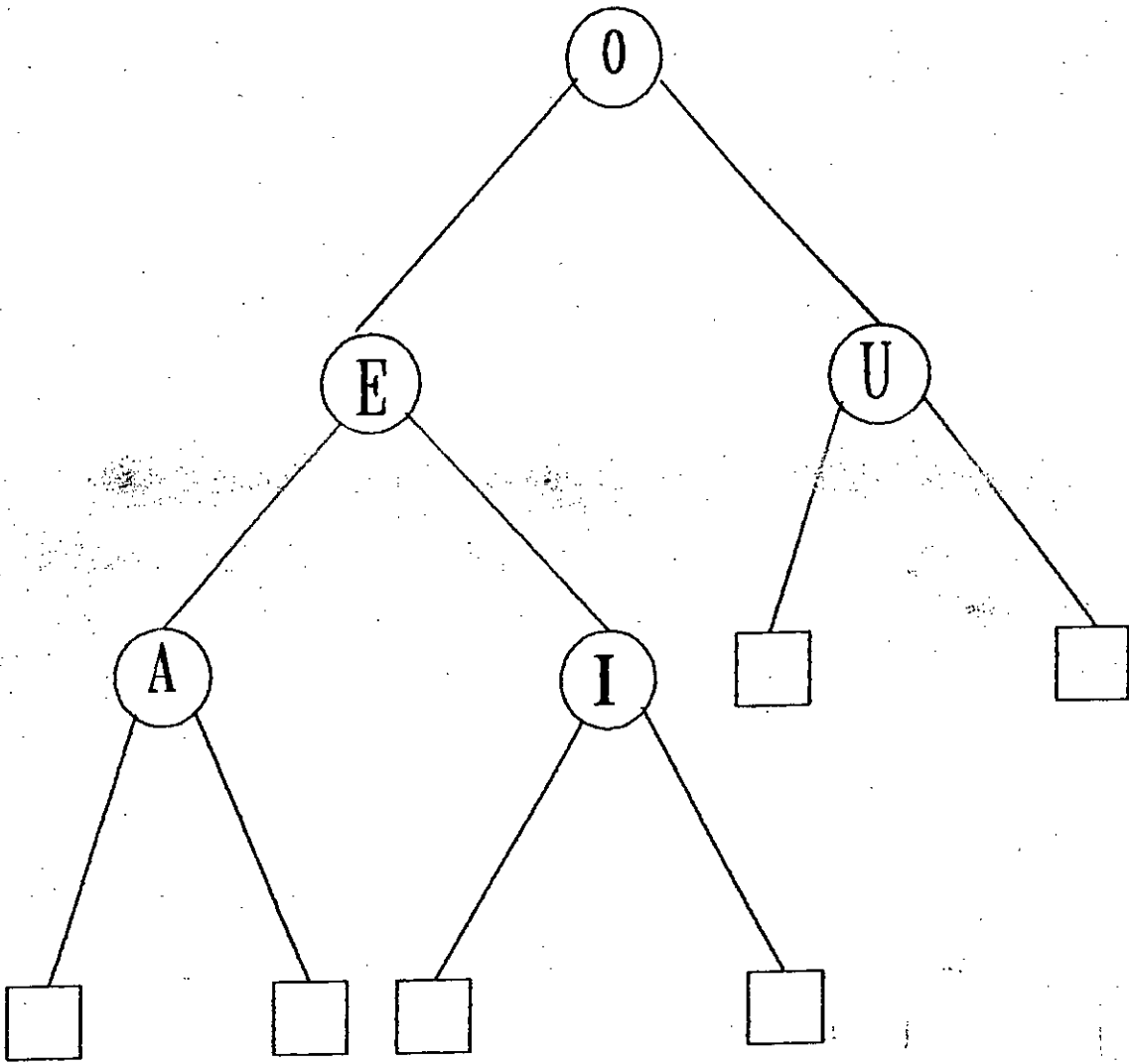


Fig. 2.1 A binary search tree

2.2 Dynamic trees.

For any given value of N , the tree corresponding to binary search achieves the theoretical minimum number of comparisons that are necessary to search a table by means of key comparisons. But the methods of chapter one are appropriate mainly for fixed size tables, since the sequential allocation of records makes insertions and deletions rather expensive. If the table is dynamically changing, we might spend more time maintaining it than we save in binary-searching it.

The use of an explicit binary tree structure makes it possible to insert and delete records quickly, as well as to search the table efficiently. As a result we essentially have a method which is useful both for searching and for sorting. This gain in flexibility is achieved by adding two link fields to each record of the table. Techniques for searching a growing table are often called symbol table algorithms, because assemblers and compilers and other system routines generally use such methods to keep track of the user defined symbols. The search and insertion techniques to be described in this section are quite efficient for use as symbol table algorithms, especially in applications where it is desirable to print out a list of the symbols in alphabetic order.

Inserting a new element Z into an existing binary search tree T is not difficult if we do not care what the effect is on the shape of the tree. If the elements in the tree are $x_1 < x_2 < x_3 < \dots < x_n$ and $x_i < z < x_{i+1}$, $0 \leq i \leq n$ (with x_0 and x_{n+1} considered as $-\infty$ and ∞ , respectively), then the i th external node can simply be replaced with the new element z . For example, adding the letter Y to the tree of figure 2.1 yields the tree of figure 2.2. Thus given a binary search tree and a new element Z to be inserted, there is a unique external node at which to insert the element because the element falls into a unique gap between some x_i and x_{i+1} .

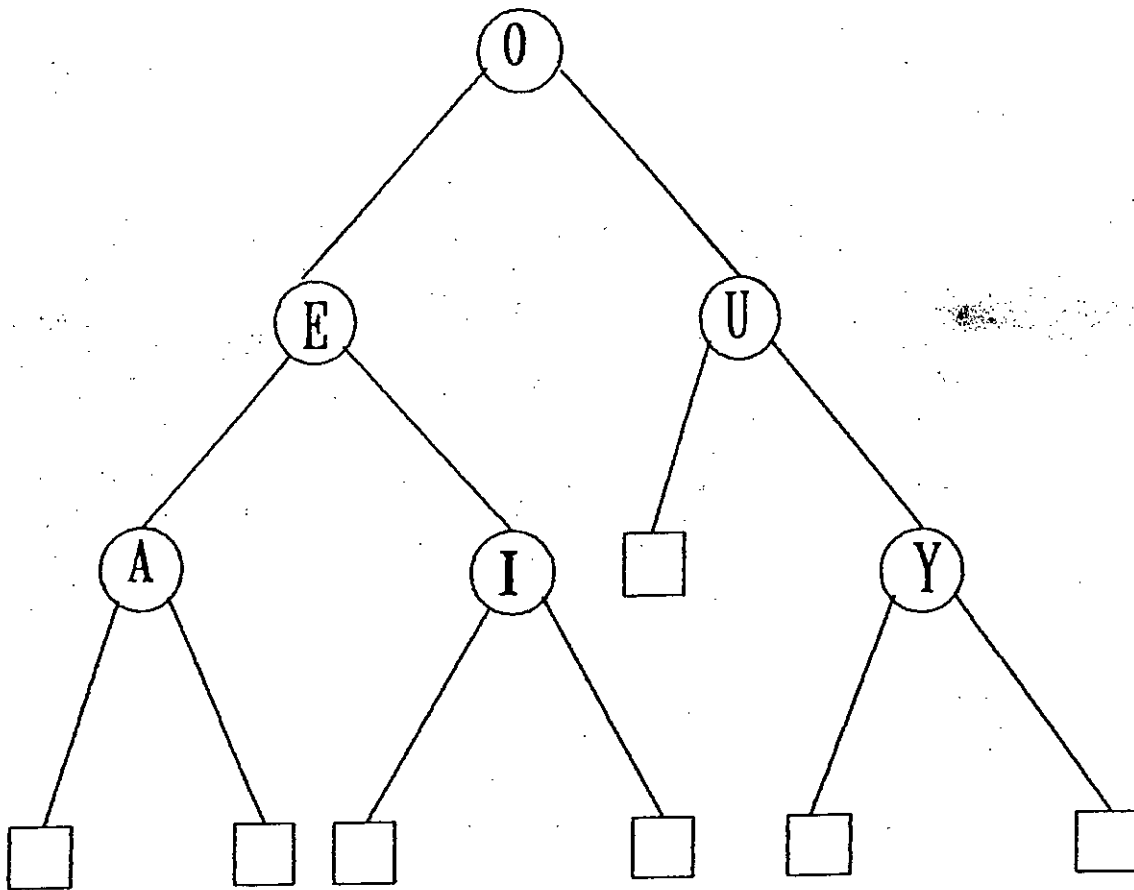


Fig. 2.2 The tree of fig. 2.1 with the letter **Y** added at its proper place

All of the keys in the left subtree of the root in fig. 2.2 are alphabetically less than the root and all keys in the right subtree are alphabetically greater. A similar statement holds for left and right subtrees of every node. It follows that the keys appear in strict alphabetic sequence from left to right if we traverse the tree in symmetric order, since symmetric order is based on traversing the left subtree of each node just before that node, then traversing the right subtree. The following algorithm spells out the searching and insertion process in detail.

Algorithm 2.1 Tree Search and Insertion

Given a table of records which form a binary search tree as described above, this algorithm searches for a given argument K. If K is not in the table, a new node containing K is inserted into the tree in the appropriate place.

The nodes of the tree assumed to contain at least the following fields :

KEY(P) = key stored in NODE(P)

LLINK(P) = Pointer to the left subtree of NODE(P)

RLINK(P) = Pointer to the right subtree of NODE(P).

The variable ROOT points to the root of the tree. For convenience we assume that the tree is not empty

LINK P, Q

P ← ROOT

```

loop
case
:  $K < KEY(P)$ : if  $LLINK(P) = \Lambda$  exit endif
                 $P \leftarrow LLINK(P)$ 
:  $K > KEY(P)$ : if  $RLINK(P) = \Lambda$  exit endif
                 $P \leftarrow RLINK(P)$ 
: else          :return
endcase
repeat

 $Q \leftarrow AVAIL; KEY(Q) \leftarrow K; LLINK(Q) \leftarrow RLINK(Q) \leftarrow \Lambda$ 

if  $K < KEY(P)$  then  $LLINK(P) \leftarrow Q$ 
else
 $RLINK(P) \leftarrow Q$ 
endif

```

End Algorithm 2.1

2.3 Dynamic Tree Analysis.

What happens if we use the above algorithm to construct search trees? In the worst case, of course, the tree can degenerate into a linear list; this happens, for example, if the order of insertion is A, E, I, O, U, which specifies essentially a sequential search. Are things really that bad on the average? If we have a random insertion order what will be the average search time in the tree constructed? To answer, we recall that the

external path length is the measure of the average search time. We want to compute the expected external path length in a tree constructed by algorithm 2.1 for random insertion order without further information, we may as well assume that each of the $n!$ permutations of the n elements is equally likely as the insertion order. Let E_n be the expected external path length in a tree constructed from n elements taken at random order. To develop a recurrence relation for E_n we observe that if the n elements are in random order, then the probability that any particular one is first is $1/n$ and the remaining elements are again in random order. Furthermore, if the first one happens to be x_i , the i th element of the n elements in the natural order, then those elements less than x_i i.e. $x_1 < x_2 < x_3 < \dots < x_{i-1}$ are in a random order as are those larger than x_i i.e. $x_{i+1}, x_{i+2}, \dots, x_n$. Thus if x_i happens to be the first element inserted into the tree as the root, it will have, by the nature of insertion process, a random tree made up of $x_1 < x_2 < x_3 < \dots < x_{i-1}$ as its left subtree and a random tree made up of $x_{i+1}, x_{i+2}, \dots, x_n$ as its right subtree.

This gives us

$$E_0 = 0$$

$$E_n = \sum_{i=1}^n (n+1 + E_{i-1} + E_{n-i}) \quad \text{Pr}(i \text{ will be the root})$$

Since the probability that i will be the root is equal for all i , it is $1/n$ and the above equation becomes

$$E_n = \sum_{i=1}^n \frac{1}{n} (n+1 + E_{i-1} + E_{n-i})$$

which by using some elementary algebra can be written as

$$E_n = n+1 + \frac{2}{n} \sum_{i=0}^{n-1} E_i$$

To solve this recurrence relation we will make a short detour to solve

$$t_n = an + b + \frac{2}{n} \sum_{i=0}^{n-1} t_i, \quad n \geq n_0 \quad (2.1)$$

for t_n in terms of $n, a, b, n_0, t_0, t_1, \dots, t_{n_0-1}$. To eliminate the summation from (2.1) we first multiply each side by n to obtain

$$nt_n = an^2 + bn + 2 \sum_{i=0}^{n-1} t_i, \quad n \geq n_0 \quad (2.2)$$

Replacing n by $n-1$, we get

$$(n-1)t_{n-1} = a(n-1)^2 + b(n-1) + 2 \sum_{i=0}^{n-2} t_i, \quad n \geq n_0 + 1 \quad (2.3)$$

Subtracting (2.3) from (2.2) gives

$$nt_n - (n-1)t_{n-1} = 2t_{n-1} + 2an + b - a, \quad n \geq n_0 + 1$$

$$\text{or } nt_n - (n+1)t_{n-1} = 2an + b - a, \quad n \geq n_0 + 1$$

Dividing this by $n(n+1)$ we have

$$\frac{t_n}{n+1} - \frac{t_{n-1}}{n} = \frac{3a-b}{n+1} + \frac{b-a}{n}, \quad n \geq n_0 + 1$$

Replacing n by i and summing gives

$$\sum_{n_0+1}^n \left(\frac{t_i}{i+1} - \frac{t_{i-1}}{i} \right) = \sum_{i=n_0+1}^n \left(\frac{3a-b}{i+1} + \frac{b-a}{i} \right) \quad (2.3)$$

The left hand side is the telescoping sum

$$\begin{aligned} \frac{t_n}{n+1} - \frac{t_{n-1}}{n} + \frac{t_{n-1}}{n} - \frac{t_{n-2}}{n} + \dots - \frac{t_{n_0+1}}{n_0+2} + \frac{t_{n_0+1}}{n_0+2} - \frac{t_{n_0}}{n_0+1} \\ = \frac{t_n}{n+1} - \frac{t_{n_0}}{n_0+1} \end{aligned}$$

and the right hand side gives

$$\begin{aligned}
 & \sum_{i=n_0+1}^n \left(\frac{3a-b}{i+1} + \frac{b-a}{i} \right) \\
 &= (3a-b) \left(\frac{1}{n_0+2} + \frac{1}{n_0+3} + \dots + \frac{1}{n+1} \right) \\
 & \quad + (b-a) \left(\frac{1}{n_0+1} + \frac{1}{n_0+2} + \dots + \frac{1}{n} \right) \\
 &= 2a(H_n - H_{n_0}) - (3a-b) \left(\frac{1}{n_0+1} - \frac{1}{n} \right)
 \end{aligned}$$

where $H_n = \sum_{i=1}^n \frac{1}{i}$ is called the harmonic function. Thus (2.3) yields

$$\begin{aligned}
 t_n &= 2anH_n + n \left(\frac{t_{n_0} - 3a + b}{n_0 + 1} - 2aH_{n_0} \right) + 2aH_n \\
 & \quad + \frac{t_{n_0} - 3a + b}{n_0 + 1} + 3a + b - 2aH_{n_0}
 \end{aligned} \tag{2.4}$$

$$\text{where } t_{n_0} = an_0 + b + \frac{2}{n_0} \sum_{i=0}^{n_0-1} t_i$$

Since $H_n = \ln n + O(1)$, (2.4) tells us that

$$\begin{aligned}
 t_n &= 2an \ln n + O(n) \\
 &= (an \ln 2)n \lg n + O(n)
 \end{aligned} \tag{2.5}$$

For E_n this yields

$$\begin{aligned}
 E_n &= (2 \ln 2)n \lg n + O(n) \\
 &\approx 1.38n \lg n
 \end{aligned} \tag{2.6}$$

Equation (2.6) tells us that in binary search trees built at random, the average search time will be about $1.38 \lg n$ or about 38 percent longer than in an optimal tree. The simplicity of algorithm T would make it acceptable in spite of the increase over

the minimum search time, except for an important fact. Our analysis assumed that the insertion order was random and this is almost never true in practice, since there are often sequences of elements arriving in their natural order. Thus, despite equation (2.6), algorithm 2.1 must be considered unreliable except in truly random circumstances.

2.4 Deletion from a Dynamic Tree.

So far we have considered only insertions. What about deletions? Deletions are somewhat more complex than insertions, because insertions cause changes only in the external nodes, but a deletion affects the internal nodes as well. There is no problem if the element to be deleted has two nil sons; we just replace the pointer to it by nil. Also, if the element to be deleted has only one nil son, we replace the pointer to it with a pointer to its single son. But when both LLINK and RLINK are non null pointers, we have to do something special. Since we cannot point two ways at once, such an element has an inorder predecessor which has null right son and it has an inorder successor which has a null left son. Thus we can replace the element to be deleted by either its predecessor or its successor, deleting that node from its original place. This operation preserves the essential left-to-right order of the table entries. The following algorithm gives a detailed description of the general way to do this.

Algorithm 2.2 Tree Deletion

Let Q be a variable which points to a node of a binary search tree. This algorithm deletes that node, leaving a binary search tree.

LINK T, R, S

$T \leftarrow Q$

if $RLINK(T) = \Lambda$ then $Q \leftarrow LLINK(T); AVAIL \leftarrow T$

return [[Is RLINK null ?]]

endif

if $LLINK(T) = \Lambda$ then $Q \leftarrow RLINK(T); AVAIL \leftarrow T$

return [[Is LLINK null ?]]

endif

[[Find Successor]]

$R \leftarrow RLINK(T)$

if $LLINK(R) = \Lambda$ then $LLINK(R) \leftarrow LLINK(T); Q \leftarrow R$

$AVAIL \leftarrow T$

return

endif

[[Find null LLINK]]

$S \leftarrow LLINK(R)$

while ($LLINK(S) \neq \Lambda$) do

$R \leftarrow S$

$S \leftarrow LLINK(R)$

repeat

$LLINK(S) \leftarrow LLINK(T); LLINK(R) \leftarrow RLINK(S)$

$RLINK(S) \leftarrow RLINK(T)$

$Q \leftarrow S; AVAIL \leftarrow T$

End Algorithm 2.2

2.4.1 Analysis of deletion.

Since Algorithm 2.2 is quite unsymmetrical between left and right, it stands to reason that a sequence of deletions will make the tree get out of balance, so that the efficiency estimates we have made will be invalid. T. N. Hibbard [1962] has proved that after a random element is deleted from a random tree by algorithm 2.2, the resulting tree is still random. If this statement is true tree behaviour will remain same after random deletions, i.e, the average number of probes will remain same in both successful and unsuccessful searches as it would be if the tree were built afresh by inserting random keys. The tree behaviour will also remain same if some random keys are inserted after the tree has got experience of some random deletions. But in practice tree behaviour deteriorates after deletion of some random nodes and this deterioration becomes severe when insertions are made in the same tree after random deletions. The analysis of this behaviour is still unknown, but the aforesaid picture indicates that if Hibbard's statement is true we cannot expect such anomalous behaviour from a dynamic tree whatever operations are carried on upon this tree. It can be inferred from the aforesaid behaviour that there is something missing in Hibbard's theorem which is stated in Knuth [19, pp. 429] as follows.

Theorem H (T.N. Hibbard, 1962). After a random element is deleted from a random tree by Algorithm 2.2, the resulting tree is still random.

Let us first follow the arguments stated in the above-mentioned reference. "This statement of the theorem is, of course, very vague. We can summarize the situation more

precisely as follows: Let T be a tree of n elements, and let $P(T)$ be the probability that T occurs if its keys are inserted in random order by Algorithm 2.1. Some trees are more probable than others. Let $Q(T)$ be the probability that T will occur if $n+1$ elements are inserted in random order by the same algorithm and then one of these elements is chosen at random and deleted by Algorithm 2.2. In calculating $P(T)$, we assume that the $n!$ permutations of the keys are equally likely; in calculating $Q(T)$, we assume that the $(n+1) \cdot (n+1)!$ permutations of keys and selections of the key to delete are equally likely. The theorem states that $P(T)=Q(T)$ for all T ." Knuth further remarks that "Although Theorem H is rigorously true, in the precise form we have stated it, it cannot be applied, as we might expect, to a sequence of deletions followed by insertions. ..."

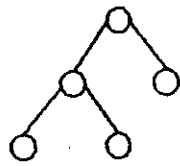
Let us now consider his theorem more rigorously. According to his theorem, after a random element is deleted from a random tree the resulting tree remains random. He proved that deletion of a random element from a random permutation results in a random permutation. In particular, using the hypothesis of Theorem H that all $(n+1)!$ permutations and $(n+1)$ deletions from each permutation are equally probable, he showed that each of the $n!$ permutations can be generated in exactly $(n+1)^2$ ways. Therefore, each of the permutations has probability $\frac{(n+1)^2}{(n+1)(n+1)!} = \frac{1}{n!}$, which is equal to the probability if the $n!$ permutations were generated by taking elements randomly from n elements. He concluded that alike permutations, the probability of a tree generated by deleting a random node from a tree of $(n+1)$ nodes will be equal to that of a tree originated from the insertion of n random keys one by one. The conflicting point is that Hibbard is carrying out all his arguments and assumptions based on the permutations and deletions to trees. In spite of the equiprobability of permutations, trees generated are not necessarily equiprobable, as remarked by them and noted earlier. The point is

that if we pick up permutations from a tree from which to delete an element randomly, probabilities of permutations do not remain equal, because probability of each permutation generating the same tree is exactly equal to the probability of occurrence of that tree, which varies from tree to tree. That is why their assumption of equiprobability of permutations and deletions do not remain valid. We can also see this from the following example.

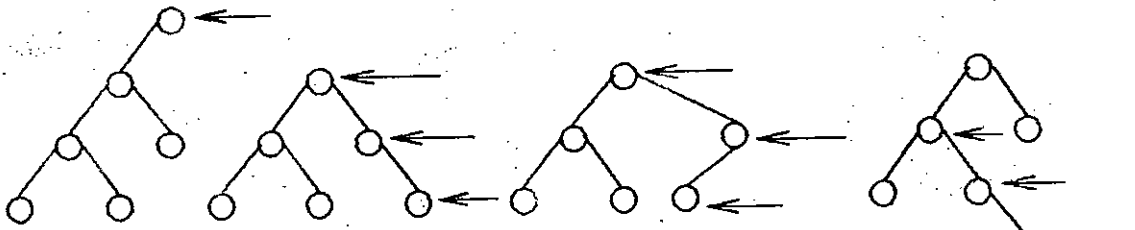
Let us consider the tree T of Fig. 2.3. This five-noded tree can be generated from 8 different permutations, and a total of 120 permutations are possible from 5 distinct keys. Therefore, the probability of tree T is $\frac{8}{120} = \frac{1}{15}$. Let us now consider the 6-noded trees A through G which are the only possible trees capable of generating T through deletion of a node, each of which has been shown in the corresponding figure with an arrow. The frequency of each tree has been shown in brackets. Thus by deleting a random node from 6-noded trees T can be generated in $(8*1+20*3+20*3+15*3+15*3+15*2+15*2) = 278$ ways, whereas there are altogether $6 * 6!$ trees (5-noded) possible by the deletion of a random node from all 6-noded trees. Therefore, tree generated in this way has the probability $\frac{278}{6*6!}$.

This example clearly shows that the assertion of Theorem H that $P(T)=Q(T)$ is incorrect. It may be noted here that the statement $P(T)=Q(T)$ is valid for trees with upto 4 nodes, which may well have misled them. In order to compute the number of permutations which can generate a particular tree (which we have used in the above example) we have derived the following recursive formula.

Let T_1 and T_2 be the left and right subtrees of tree T . By T , T_1 and T_2 we denote both corresponding trees as well as number of internal nodes. Let $N(T)$ be the number



Tree T

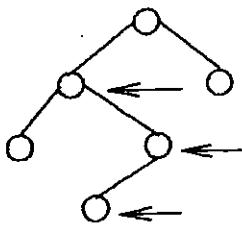


A (8)

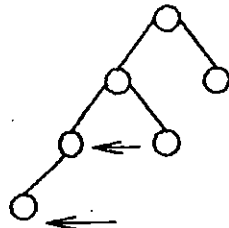
B (20)

C (20)

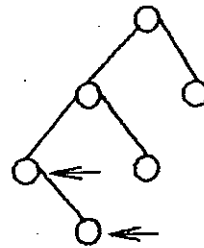
D (15)



E (15)



F (15)



G (15)

Fig. 2.3 All possible 6-noded trees generating tree T

of permutations generating T . Then

$$N(T) = N(T_1) * N(T_2) * \binom{T_1 + T_2}{T_1}$$

Subtrees T_1 and T_2 can be obtained by $N(T_1)$ and $N(T_2)$ permutations respectively. Again $T_1 + T_2$ positions can be filled up by T_1 elements in $\binom{T_1 + T_2}{T_1}$ ways. For each T_1 and T_2 their keys can appear in $\binom{T_1 + T_2}{T_1}$ ways. So product of $N(T_1)$, $N(T_2)$ and $\binom{T_1 + T_2}{T_1}$ gives the result.

2.5 Static trees.

The application of binary search trees to static tables is concerned entirely with arranging the tree so as to minimize search time. If we want to minimize the worst case search time, we can simply use the tree corresponding to binary search and we do not need an explicit tree at all because we have assumed that the table has been constructed once and that its contents will change either never or so infrequently that it will be possible to reconstruct the entire table to make a change.

The more difficult problem is to minimize the average search time, given some distribution of how the search will end. If the table consists of elements $x_1 < x_2 < x_3 < \dots < x_n$, then the search can end successfully at any of the x_i (internal nodes) and unsuccessfully in any of the $n + 1$ gaps between the x_i and at the endpoints. Let us assume that we have values $p_1, p_2, p_3, \dots, p_n$ and $q_0, q_1, q_2, \dots, q_n$ where p_i is the relative frequency with which a search will end successfully at x_i and q_i is the relative frequency with which the search for Z will end unsuccessfully at y_i , i.e., with $x_i < Z < x_{i+1}$ (defining $x_0 = -\infty$ and $x_{n+1} = \infty$).

The problem is to choose among the many possible binary trees with n internal nodes for a particular set of values p_i and q_i . We will measure the desirability of a tree by the cost of an average search; the cost will be the number of probes. In chapter one

we introduced such a measure, the internal path length or the related external path length. That measure is not sufficient for our purpose because it does not take the varying frequencies into account. However we can generalize the path length as follows : the required path length of a binary tree T with internal nodes $x_1, x_2, x_3, \dots, x_n$, external nodes y_0, y_1, \dots, y_n and p_i and q_i defined above is

$$\sum_{i=1}^n p_i [1 + \text{level}(x_i)] + \sum_{i=0}^n q_i \text{level}(y_i) \quad (2.7)$$

As in the cases of external and internal path lengths, it is convenient to define weighted path length recursively.

$$W(\text{null}) = 0$$

$$W(T) = W(T_l) + W(T_r) + \sum p_i + \sum q_i \quad (2.8)$$

Where the summations $\sum p_i$ and $\sum q_i$ are over all p_i and q_i in T .

Our problem is to determine the binary search tree that will have an optimal (minimal) weighted path length, given the frequencies p_i and q_i . Since the number of possible trees is exponentially large as a function of n we cannot do the obvious way of examining all possibilities, computing the weighted path length of each and choosing the smallest. In fact, the large number of possibilities makes it seem doubtful that there is any reasonable way to make the determination. However a simple but crucial observation about the nature of the weighted path length of a tree will show us the way to proceed.

The observation is that subtrees of an optimal tree must themselves be optimal. More precisely, if T is an optimal binary search tree on weights $q_0, p_1, q_1, \dots, p_n, q_n$ and it has weight p_i at the root, then the left subtree must be optimal over the weights $q_0, p_1, q_1, \dots, p_{i-1}, q_{i-1}$ and the right subtree must be optimal over weights

$q_i, p_{i+1}, q_{i+1}, \dots, p_n, q_n$. To see why this optimality principle must hold, suppose that some tree over $q_0, p_1, q_1, \dots, p_{i-1}, q_{i-1}$ had lower weighted path length than the one that is the left subtree of T . Then by (2.8) we could get a tree T' with lower weighted path length than T by replacing the left subtree of T by the one of lower weighted path length we have supposed to exist. This contradicts the assumed optimality of T . We can argue similarly about the right subtree of T and in fact any subtree of T . This optimality principle is the basis of a technique called dynamic programming, which we will use to compute optimal binary search trees.

The optimality principle together with (2.8) allows us to write the following recursive description of optimal binary search trees: Let $C_{i,j}, 0 \leq i \leq n$, be the cost of an optimal tree over the frequencies $q_i, p_{i+1}, \dots, p_j, q_j$. Then

$$C_{ii} = 0$$

$$\text{and } C_{ij} = \min_{i < k \leq j} (C_{i,k-1} + C_{kj}) + \sum_{t=i}^j q_t + \sum_{t=i+1}^j p_t$$

by (2.8), since the optimality principle guarantees that if x_k is the root of the optimal tree, then $C_{i,k-1}$ and C_{kj} are the costs of the left and right subtrees respectively.

Defining $W_{ii} = q_i$

$$W_{ij} = W_{i,j-1} + p_j + q_j, \quad i < j \quad (2.9)$$

We get $C_{ii} = 0$

$$C_{ij} = W_{ij} + \min_{i < k \leq j} (C_{i,k-1} + C_{kj}) \quad (2.10)$$

Equation (2.9) and (2.10) form the basis of our computation of the optimal search trees, in evaluating (2.10) to get C_{0n} , the cost of the optimal tree over $q_0, p_1, \dots, p_n, q_n$, we

need only keep track of the choice of k that achieves the minimum in (2.10). We thus define

$$R_{ij} = \text{a value of } k \text{ that minimizes } C_{i,k-1} + C_{kj} \text{ in (2.10)} \quad (2.11)$$

R_{ij} is the root of an optimal tree over $q_i, p_{i+1}, \dots, p_j, q_j$.

We are left with the problem of organizing the computation from (2.9), (2.10) and (2.11). Of course we could simply make (2.9) and (2.10) into recursive procedures as they stand, but that would lead to an exponential time algorithm because many computations would be repeated over and over again. The obvious way to avoid this difficulty is to insure that each C_{ij} is computed only once. We do this by observing that the value of C_{ij} in (2.10) depends only on values below and / or to the left of C_{ij} in the matrix from all combinations of i and j . We thus compute the matrix C (and in parallel, W and R) starting from the main diagonal and moving up one diagonal at a time. First $C_{ii} = 0$, $0 \leq i \leq n$ by (2.10). Then we compute $C_{i,i+1}$, $0 \leq i \leq n-1$, then $C_{i,i+2}$, $0 \leq i \leq n-2$ and so on. Algorithm 2.3 embodies this idea.

Algorithm 2.3 Optimal tree

[[Initialize the main diagonal]]

for $i = 0$ to n do

$R_{ii} \leftarrow i$

$W_{ii} \leftarrow q_i$

$C_{ii} \leftarrow 0$

Repeat

[[Visit each of the n upper diagonals]]

```

for  $l = 1$  to  $n$  do

  [[ Visit each entry in the  $l$ th diagonal ]]

  for  $i = 0$  to  $n - l$  do

     $j \leftarrow i + l$ 

    [[ The elements on the  $l$ th diagonal have  $j - i \equiv l$  ]]

    [[ Compute  $(i,j)$  entries;  $R_{ij}$  is a value of  $k$ ,  $i < k \leq j$ 
    minimizing  $C_{i,k-1} + C_{k,j}$ ]]

     $R_{ij} \leftarrow i + 1$ 

    for  $k = i + 2$  to  $j$  do

      if  $C_{i,k-1} + C_{kj} < C_{i,R_{ij}-1} + C_{R_{ij}j}$  then  $R_{ij} \leftarrow k$ 

    endif

    repeat

       $W_{ij} \leftarrow W_{i,j-1} + p_j + q_j$ 

       $C_{ij} \leftarrow C_{i,R_{ij}-1} + C_{R_{ij}j} + W_{ij}$ 

    repeat

    repeat

  End Algorithm 2.3

```

The running time of algorithm 2.3 will be roughly proportional to the number of comparisons " $C_{i,k-1} + C_{kj} < C_{i,R_{i,j-1}} + C_{R_{i,j}j}$ " made in the innermost loop which is executed

$$\begin{aligned} \sum_{l=1}^n \sum_{i=0}^{n-l} \sum_{k=i+2}^{i+l} &= \sum_{l=1}^n \sum_{i=0}^{n-l} (l-1) \\ &= \sum_{l=1}^n (n-l+1)(l-1) \\ &= \frac{1}{6}n^3 + O(n^2) \end{aligned} \quad (2.12)$$

times. Algorithm 2.12 thus runs in time proportional to n^3 not very acceptable in constructing search trees of several thousand elements. A factor of n can actually be removed from the running time if we make use of a monotonicity property. Let $r(i,j)$ denote an element of $R(i,j)$; we need not compute the entire set $R(i,j)$, a single representative is sufficient. Once we have found $r(i,j-1)$ and $r(i+1,j)$ we may always assume that

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

when the weights are nonnegative. This limits the search for the minimum, since only $r(i+1,j) - r(i,j+1) + 1$ values of K need to be examined in algorithm 2.3 instead of $j-i$. The total amount of work when $j-i=d$ is now bounded by the telescoping series

$$\begin{aligned} \sum_{\substack{d \leq j \leq n \\ i=j-d}} (r(i+1, j) - r(i, j-1) + 1) \\ = r(n-d+1, n) - r(0, d-1) + n-d+1 < 2n \end{aligned}$$

hence the total running time is reduced to $O(n^2)$. Based on this observation we can replace the statement " $R_{ij} \leftarrow i+1$ " by " $R_{ij} \leftarrow R_{i,j-1}$ " and the innermost loop "for $k=i+2$ to j " by "for $k=R_{i,j-1}$ to $R_{i+1,j}$ " in the above algorithm for the construction of optimum binary search.

2.6 Heuristics on Optimality.

Even the improved version of the optimal binary search algorithm may not be efficient enough in certain circumstances. If n is several thousand, it may be quite expensive to construct the optimal tree; furthermore, the frequencies p_i and q_i are rarely known with any accuracy and it would be foolish to invest much computation time to get an optimal tree from inaccurate frequencies. In such cases a tree that approximates the optimal tree may be satisfactory and will certainly be less expensive to construct. We now examine heuristics for the construction of "near optimal" binary search trees.

Given frequencies p_i and q_i , two heuristics immediately suggest themselves; we discuss them in turn. The monotonic rule constructs a binary search tree by choosing the root to be x_i , where p_i is the largest p value and proceeding recursively on the left and right subtrees. This may cause poor performance because we have totally disregarded unsuccessful search frequencies. This suggests that perhaps the monotonic rule may work very well in the special case of only successful searches occurring. Unfortunately that is not the case: the monotonic rule produces poor trees in general, even when all the $q_i = 0$: on the average a tree constructed according to the monotonic rule is no better than a tree at random.

The second heuristic is the balancing rule: Choose the root so as to equalize as much as possible the sum of the frequencies in the left and right subtrees, breaking ties arbitrarily. The cost of the tree resulting from the balancing rule is always extremely close to the cost of the optimal tree. The balancing rule can be implemented in time proportional to n , the number of elements in the table. We want to choose the root to equalize as much as possible the total frequencies of the left and right subtrees. In other words we need to find an i such that $|(q_0 + p_1 + \dots + p_{i-1} + q_{i-1}) - (q_i +$

$p_{i+1} + \dots + p_n + q_n$) is minimized and we must repeat the computation recursively for $q_0, p_1, \dots, p_{i-1}, q_{i-1}$ and $q_i, p_{i+1}, \dots, p_n, q_n$ to find the left and right subtrees, respectively. The computation is organized as follows. We first compute the

$$W_{0i} = q_0 + p_1 + \dots + p_i + q_i \quad 0 \leq i \leq n$$

by the recurrence relation

$$W_{00} = q_0$$

$$W_{0,i+1} = W_{0i} + p_{i+1} + q_{i+1}$$

The computation of the W_{0i} thus requires only time proportional to n . Given the W_{0i} , we can immediately get any needed W_{ij} with two subtractions, since

$$\begin{aligned} W_{ij} &= q_i + p_{i+1} + \dots + p_j + q_j \\ &= W_{0j} - W_{0,i-1} - p_i \end{aligned}$$

To find where $|W_{0,i-1} - W_{in}|$ is minimized we need to find where $W_{0,i-1} - W_{in}$ changes sign; the i we want will be on either side of the sign change. More exactly, if $W_{0,k-1} - W_{k,n} \leq 0 < W_{0,k} - W_{k+1,n}$ then we want either $i = k$ or $i = k + 1$ depending on whether or not $|W_{0,k-1} - W_{k,n}|$ is less than $|W_{0,k} - W_{k+1,n}|$. We can find K by a binary search type of process. Initially we know that $1 \leq k \leq n$; In general if $l \leq k \leq h$, check the sign of $W_{0,m-1} - W_{m,n}$ where $m = \lfloor (l+h)/2 \rfloor$. If it is positive, set $h \leftarrow m$ or if negative set $l \leftarrow m$ and continue. If it is zero we are done. Finding i in this way will require work proportional to $\lg n$ and the total amount of work will be given by the recurrence relation :

$$T(n) \leq \max_{1 \leq i \leq n} [T(i-1) + T(n-i) + c \lg n]$$

where $T(i-1)$ and $T(n-i)$ are the work to find left and right subtrees respectively, once the root is found. The $c \lg n$ term is the time required to find the root i . $T(0)$ is some constant. The solution to this recurrence relation gives $T(n)$ proportional to $n \lg n$. We can reduce the computation time needed by searching for i in a slightly different way. We find the spot where $W_{0,k-1} - W_{k,n}$ changes sign by checking $k = 1, k = 2, k = n-1, k = 4, k = n-3, k = 8, k = n-7, \dots$. In other words, we check from the left and right simultaneously, doubling the interval at each step. In this way we spend time proportional to $\min[\lceil \lg i \rceil, \lceil \lg(n-i) \rceil]$ to find an interval containing i . This interval has length proportional to $\min[i, n-i]$ and i can be located by binary search in time proportional to $\min[1 + \min[\lceil \lg i \rceil, \lceil \lg(n-i) \rceil]]$. The recurrence relation for $T(n)$ becomes

$$T(n) \leq \max_{1 \leq i \leq n} \{T(i-1) + T(n-i) + d(1 + \min[\lceil \lg i \rceil, \lceil \lg(n-i) \rceil])\}$$

and the solution of this gives $T(n)$ proportional to n , as desired.

CHAPTER 3

BALANCED TREE

3.1 Introduction.

The binary tree algorithms work very well for a wide variety of applications, but they do have the problems of bad worst case performance. Files already in order, files in reverse order or files with alternating large and small keys cause the binary-tree search algorithm to perform very badly. The tree insertion algorithm will produce a good search tree when the input data is random, but there is still the annoying possibility that a degenerate tree will occur. Perhaps we could devise an algorithm which keeps the tree optimum at all times; but unfortunately that seems to be very difficult. Another idea is to keep track of the total path length and to completely reorganize the tree whenever its path length exceeds a certain limit, but this approach might require a large number of reorganizations as the tree is being built.

A very pretty solution to the problem of maintaining a good search tree can be achieved by keeping the tree perfectly balanced at all times. Unfortunately when the tree is thus constrained, it is more costly to insert or delete an element than to rearrange elements in the sequentially allocated arrays required by linear search. Instead our goal is to allow more flexibility in the shape of the tree so that insertions and deletions will not be so expensive yet search times will remain logarithmic. The method for achieving all this involves what we shall call "balanced trees". The height of search trees of n elements will be $O(\log_2 N)$, so that search times are logarithmic and insertions and deletions will require only local changes along a single path from the root to a leaf, requiring only

time proportional to the height of the tree - that is, $O(\log_2 N)$. Furthermore, there is no advantage to balanced trees unless N is reasonably large; thus if we have an efficient method that takes $64 \log_2 N$ units of time and an inefficient method that takes $2N$ units of time, we should use the inefficient method unless N is greater than 256. On the other hand N should not be too large, either; balanced trees are appropriate chiefly for internal storage of data. Since internal memories seem to be getting large and large as time goes by, balanced trees are becoming more and more important.

3.2 Definition of a height balanced tree.

The height of a tree is defined to be its maximum level, the length of the longest path from the root to an external node. A binary tree is called balanced if the absolute difference of heights of left and right subtrees does not exceed one. Fig. 3.1 Shows a balanced tree with 17 internal nodes and height 5; the balance factor within each node is shown as +, 0, or - according as the right subtree height minus the left subtree height is +1, 0, or -1.

3.2.1 Height of a balanced tree :

The definition of balance represents a compromise between optimum binary trees (with all external nodes required to be on two adjacent levels and arbitrary trees (unrestricted)). It is, therefore, natural to ask how far from optimum a balanced tree can be.

Once we have shown that a height balanced tree of n nodes has height $O(\log_2 N)$, then the worst case search times is $O(\log_2 N)$ and since the insertion / deletion time will also be proportional to the height, we will be done with the satisfaction that we can safely use height-balanced trees as a storage structure for dynamic tables.

What is the height of the tallest height-balanced trees containing N internal nodes

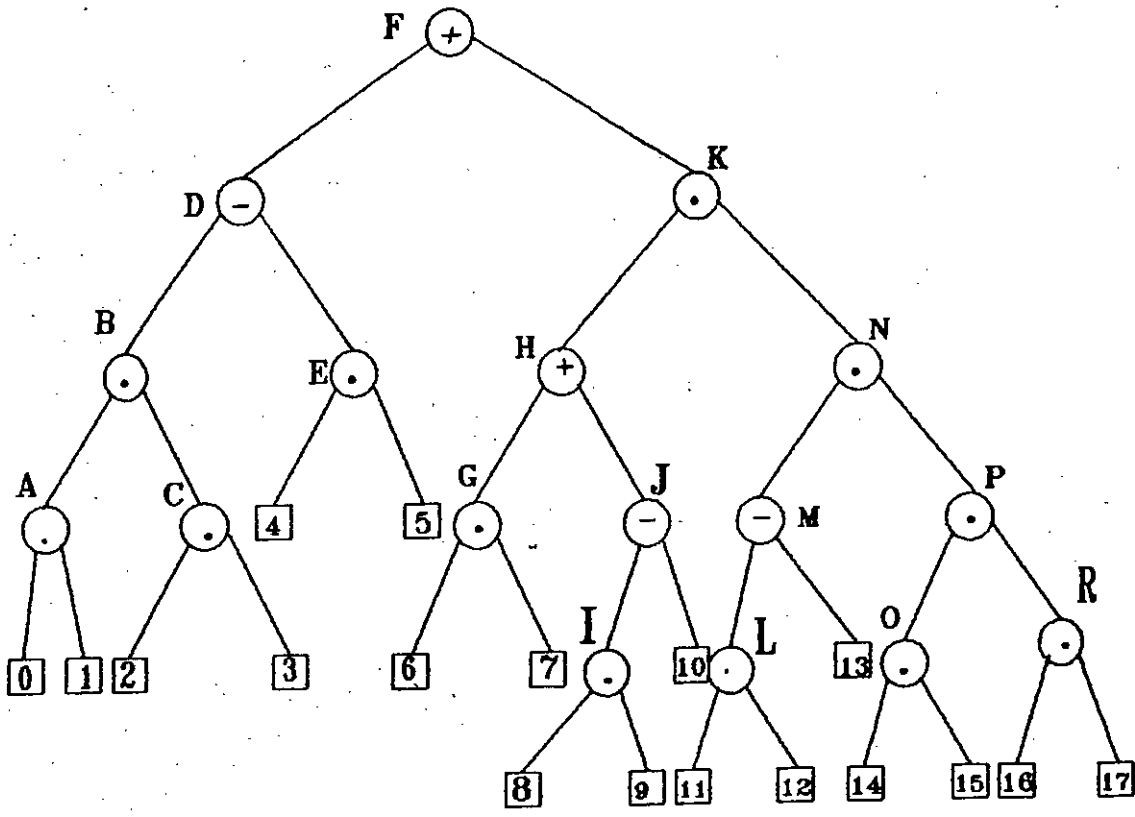


Fig. 3.1 A balanced binary tree

and $N+1$ external nodes ? To answer this question we will turn it around and ask what is the least number of internal nodes necessary to achieve height h in a height-balanced tree.

Let T_h be a height balanced tree of height h with n_h internal nodes, the fewest possible. Obviously,

$$n_0 = 0 \text{ and } n_1 = 1 \quad (3.1)$$

Now let us consider T_h , $h \geq 2$. Since T_h is height balanced and has height h , it must have a tree of height $h-1$ as its left or right subtree and a tree of height $h-1$ or $h-2$ as its other subtree. For any k , a tree of height k has a subtree of height $k-1$ and thus $n_k > n_{k-1}$; this tells us that T_h has one subtree of height $h-1$ and the other of height $h-2$, for if T_h had two subtrees of height $h-1$, we would replace one of them by T_{h-2} and, since $n_{h-1} > n_{h-2}$, this would contradict the assumption that T_h had as few nodes as possible for height balanced tree of height h . Similarly the two subtrees of T_h must be height - balanced and have the fewest nodes possible, for otherwise, we could replace one or both subtrees with same height subtrees of fewer nodes, again contradicting the assumption that T_h has as few nodes as possible. Thus T_h has T_{h-1} as one subtree and T_{h-2} as the other.

$$n_h = n_{h-1} + n_{h-2} + 1 \quad (3.2)$$

To find n_h in terms of h we will deviate a little from the current topic. let us consider the recurrence relations

$$a_0 = r, a_1 = s, a_{n+2} = a_{n+1} + a_n, \quad n \geq 0 \quad (3.3)$$

$$b_0 = 0, b_1 = 1, b_{n+2} = b_{n+1} + b_n + c, \quad n \geq 0 \quad (3.4)$$

From the first relation [eq. (3.3)] it is easy to show that $a_n = rF_{n-1} + sF_n$ where F_n is the Fibonacci number of order n .

Now (3.2) and (3.4) is similar in the sense that $c = 1$ gives (3.3) from (3.4). Let us try to solve relation (3.4). From (3.4)

$$(b_{n+2} + c) = (b_{n+1} + c) + (b_n + c)$$

which can be written as

$$b'_{n+2} = b'_{n+1} + b'_n, \quad b'_0 = c \quad \text{and} \quad b'_1 = c + 1$$

From the solution of relation (3.3) it is evident that

$$b'_n = cF_{n-1} + (c+1)F_n$$

i.e, $b_n = cF_{n-1} + (c+1)F_n - c$

Replacing c with 1 we get

$$\begin{aligned} n_h &= F_{h-1} + 2F_h - 1 \\ &= F_{h+2} - 1 \\ &= \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2} - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^{h+2} - 1 \end{aligned} \quad (3.5)$$

Since $\left| \frac{1-\sqrt{5}}{2} \right| < 1$, the term $\left(\frac{1-\sqrt{5}}{2} \right)^{h+2} / \sqrt{5}$ is always quite small, so that

$$n_{h+1} = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2} + O(1)$$

Since the tree of height h with the fewest nodes has n_h nodes, it follows that any tree with fewer than n_h nodes has height less than h , then

$$n + 1 \geq n_{h+1} = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^{h+2} + O(1)$$

implying that

$$h \leq \frac{1}{\log_2 \frac{1+\sqrt{5}}{2}} \log_2(n+1) + O(1) \\ \approx 1.44 \log_2(n+1)$$

Thus in the worst case the number of probes into a height balanced tree of n internal nodes will never be more than 45 percent higher than the optimum.

3.3 Balanced tree Search, Insertion and Deletion.

To make an insertion or deletion we will use the approach of normal dynamic tree insertion and deletion respectively when the tree can change in an unconstrained manner; then we will follow it with a rebalancing pass that verifies or restores the height balanced state of the tree. In order to verify / restore the tree we need to be able to test whether the element inserted or deleted has changed the relationship between the heights of the subtrees of a node so as to violate the height constraints. For this purpose, we will store a condition code in each node of a height balanced tree as described earlier. Storing condition codes requires an extra two-bit field per node in the tree.

Roughly speaking, the rebalancing pass consists of retracing path upward from the newly inserted node (or from the site of the deletion) to the root. Here we will store the path node by node, on a stack as we go down the tree from the root to the site of the modification.

As the path is followed upward, we check for instances of the taller subtree growing taller (on an insertion) or the shorter subtree becoming shorter (on a deletion). When we find such an occurrence, we apply a local transformation to the tree at that point. In the case of an insertion it will turn out that applying the transformation at the first such occurrence will completely rebalance the tree. In the case of a deletion the

transformation may need to be applied at many points along the way up to the root. Since the rebalancing after an insertion is a little bit simpler than after a deletion we consider it first. It may happen that the new item has been added to the bottom of the taller of two subtrees of some node. Without loss of generality suppose the right subtree was taller before the insertion as shown in fig. 3.2. The way to repair the newly created imbalance depends on where within the taller subtree T the insertion was made. Suppose it was in the right subtree of T ; we then have a situation that can be repaired as shown in fig. 3.3.

The transformation shown in fig. 3.3 is called a rotation and it is considered to be applied to the element A . Obviously, if the left subtree of fig. 3.2 had been taller and the insertion made it even taller, we would have to rotate in the other direction, using the mirror image of the transformation shown in fig. 3.3. If the insertion had been to the left subtree of T in fig. 3.2 i.e. to T_2 in fig. 3.3 then the repair is made as shown in fig. 3.4. This transformation called a double rotation, is considered to be applied at A . The new element can be at the bottom of either T_2 or T_3 . Again a mirror-image transformation would be needed in the comparable case where the left subtree in fig. 3.2 was the taller.

The transformations of fig. 3.3 and fig. 3.4 have two critical properties: the in-order of the elements of the tree remains the same after the transformation as it was before, and the overall height of the tree is the same after the transformation as it was before the insertion.

Now we are in a position to state the insertion algorithm which is as follows: We will use dynamic tree insertion algorithm to insert the new element to its proper place, setting its condition code to \cdot , and storing the path followed down the tree, node

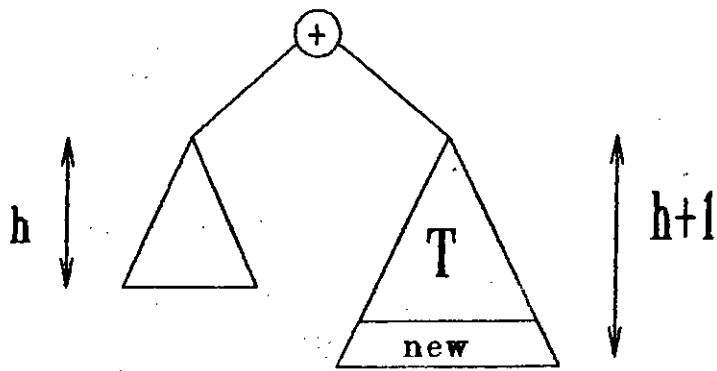


Fig. 3.2

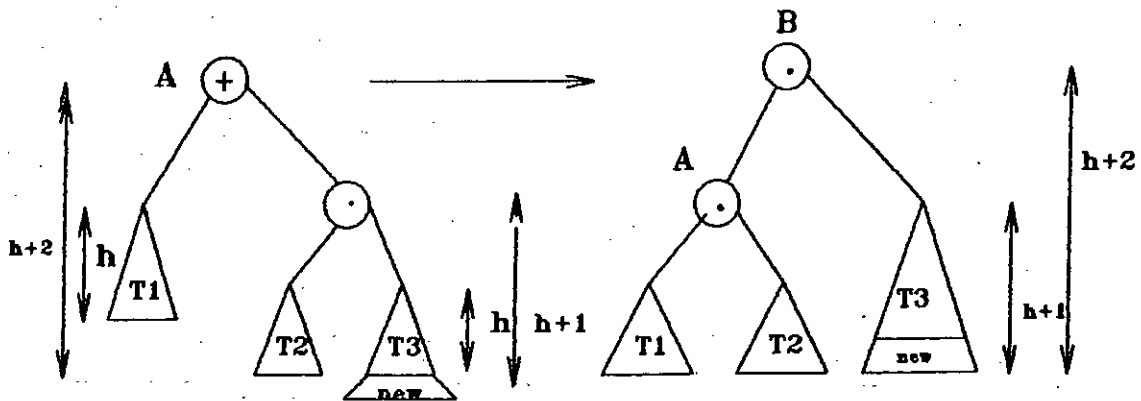


Fig. 3.3 Single Rotation in a Balanced tree

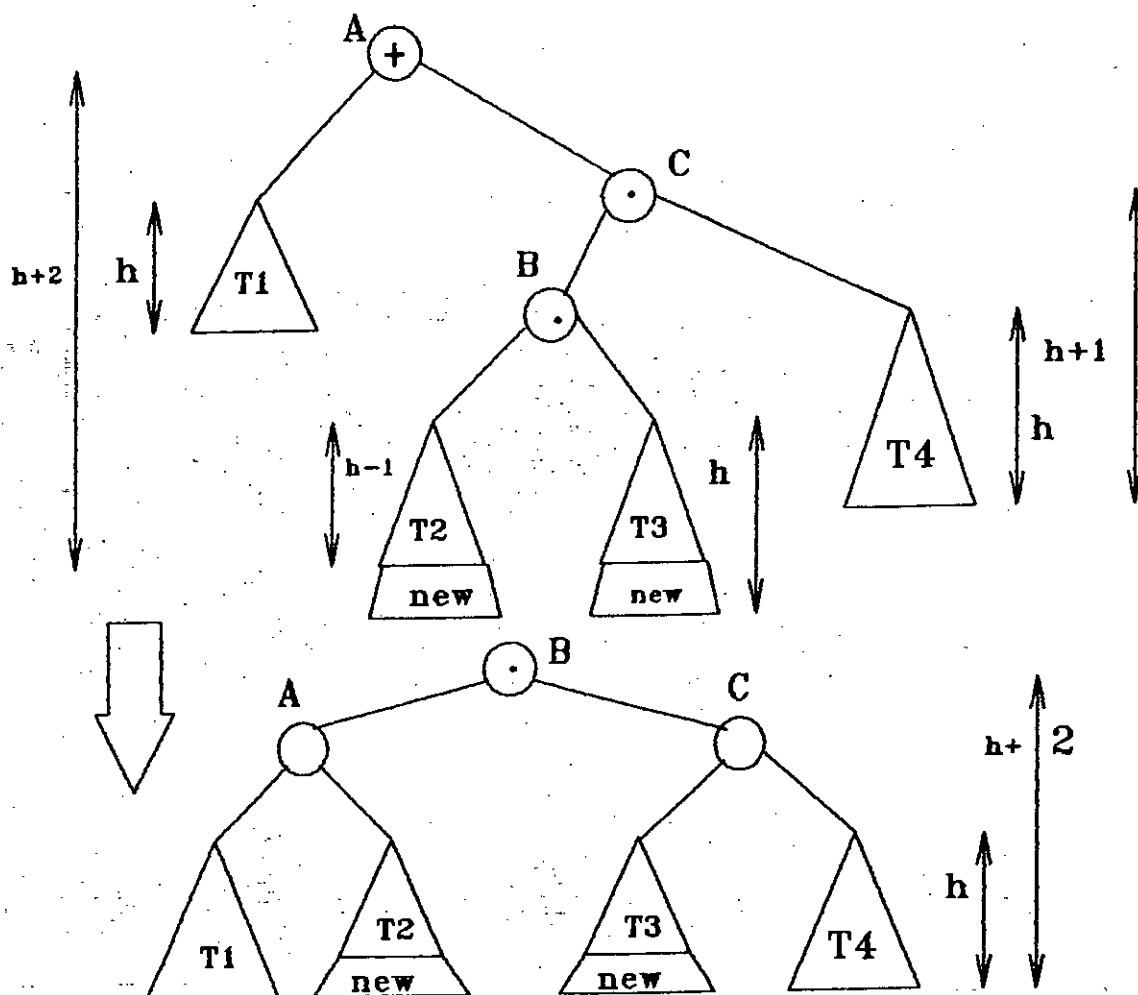


Fig. 3.4 Double rotation in a balanced tree

by node, in a stack. Then, we will retrace that path backward, up the tree, popping nodes off the stack and correcting the height condition codes until either the root is reached and its height condition code corrected, or we reach a point at which a rotation or double rotation is necessary to rebalance the tree. More specifically, we follow this path backward, node by node, taking actions as defined by the following rules, where current is the current node on the path, son is the node before current on the path and grandson is the node before son on the path. Initially, son is the new element just inserted, current is its father and grandson is nil :

(1) If current has height condition \cdot , change it to $+$ if son = RIGHT (current) and to $-$ if son = LEFT (current). In this case the subtree rooted at son grew taller by one unit, causing the subtree rooted at current to grow taller by one unit, so we continue up the path, unless current is the root, in which case we are done. To continue up the path we set grandson \leftarrow son, son \leftarrow current, and current to the top stack entry, which is removed from the stack.

(2) If current has height condition $-$ and son = RIGHT (current) or current has height condition $+$ and son = LEFT (current), change the height condition of current to \cdot , and the procedure terminates. In this case the shorter of the two subtree of current has grown one unit taller, making the tree better balanced.

(3) If current has height condition $-$ and son = LEFT (current) or current has height condition $+$ and son = RIGHT (current), then the taller of the two subtrees of current has become one unit taller, unbalancing the tree at current. A transformation is

performed according to the following four case :

	Grandson = RIGHT (son)	Grandson = LEFT (son)
Son = Right(current)	Single rotation at current using fig. 3.3	Double rotation around current using fig 3.4
Son = LEFT(current)	Double rotation around current using the mirror image of fig. 3.4	Single Rotation around current using the mirror image of fig. 3.3

The deletion process is more complicated than insertion because it will not always be sufficient to apply a transformation only at the lowest point of imbalance; transformations may need to be applied at many levels between the site of the deletion and the root. To delete a node from a height-balanced tree, we proceed as for unconstrained trees : if the node is on leaf, just delete it; if it has one nonnil son, replace it with its son; if it has two nonnil sons, replace it by its inorder predecessor which will have a null right son or replace it by its inorder successor which will have a null left son.

As in the insertion algorithm, we store on a stack the path followed down the tree to the site of the node to be deleted, then we retrace the path backward up the tree, popping nodes off the stack, correcting height condition codes and making transformations as needed. As we go back up the path actions are taken as defined by the following rules, where current is the current node on the path and son is the node before current. Initially current is the father of the node deleted and son is the node deleted :

(1) If current has height condition code .. then shortening either subtree does not affect the height of the tree rooted at current. The condition code of current is changed to + if son = LEFT (current) and to - if son = RIGHT (current). The procedure then terminates.

(2) If current has height condition + and son = RIGHT (current) or current has height condition - and son = LEFT (current), then condition code of current is changed to ..

The subtree rooted at current has become shorter by one unit, so we continue up the path, unless current is the root, in which case we are done. To continue up the path we set $\text{son} \leftarrow \text{current}$ and current to the stack entry, which is removed from the stack.

(3) If current has height condition + and $\text{son} = \text{LEFT}(\text{current})$, then the height constraint is violated at current. There are three subcases, depending on the height condition code at $\text{RIGHT}(\text{current})$, the brother of son. The subcases are as given in fig. 3.5, 3.6, and 3.7.

In fig. 3.7 the height condition codes of A and C are both .., if that of B was .. If B was +, then A is - and C is .. If B was -, then A is . and C is +.

For both configuration in fig. 3.6 and 3.7, the height of the subtree is one less than it was before deletion. Thus if current is the root we can terminate the procedure; otherwise we have to continue up the path towards the root.

(4) If current has height condition - and $\text{son} = \text{RIGHT}(\text{current})$, then the height constraint is violated at current. There are three subcases, depending on the height condition code $\text{LEFT}(\text{current})$, the brother of son. The subcases are the mirror images of those given in fig. 3.5, 3.6 and 3.7.

The deletion algorithm will clearly require only time proportional to the height of the tree; the deletion can be accomplished in $O(\log_2 n)$ time, as can insertion. An insertion, however, will need at most one rotation / double rotation to rebalance the tree, while a deletion from a height balanced tree of height h can require as many as $\lfloor h/2 \rfloor$ rotations / double rotations but no more.

3.4 Balanced Tree search and insertion without using stack.

The backward scan up the tree from the site of the insertion in a height balanced tree can be eliminated at the expense of retracing part of the path down the tree. This

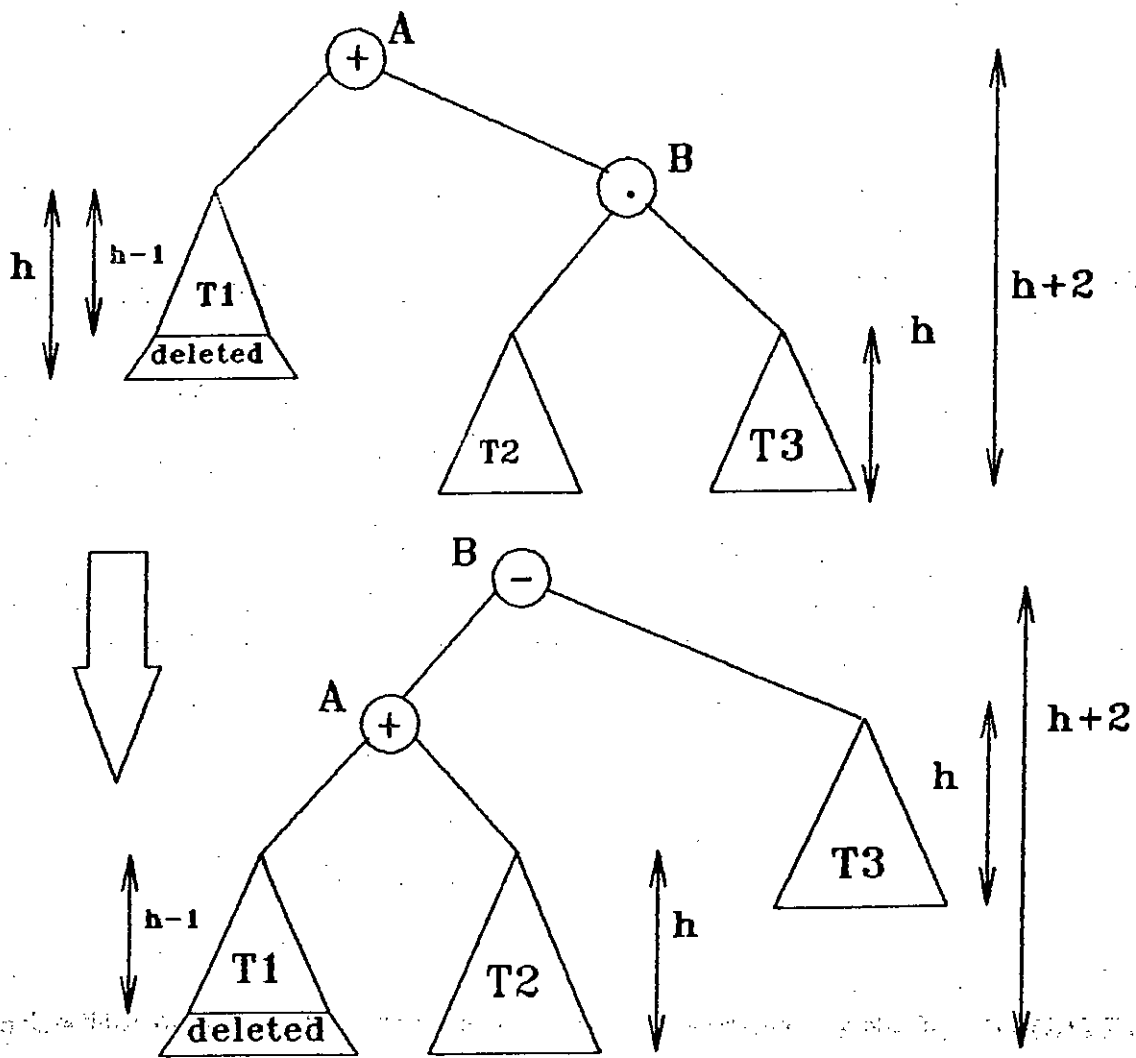


Fig. 3.5 Single rotation in times of deletion when condition code is ‘.’

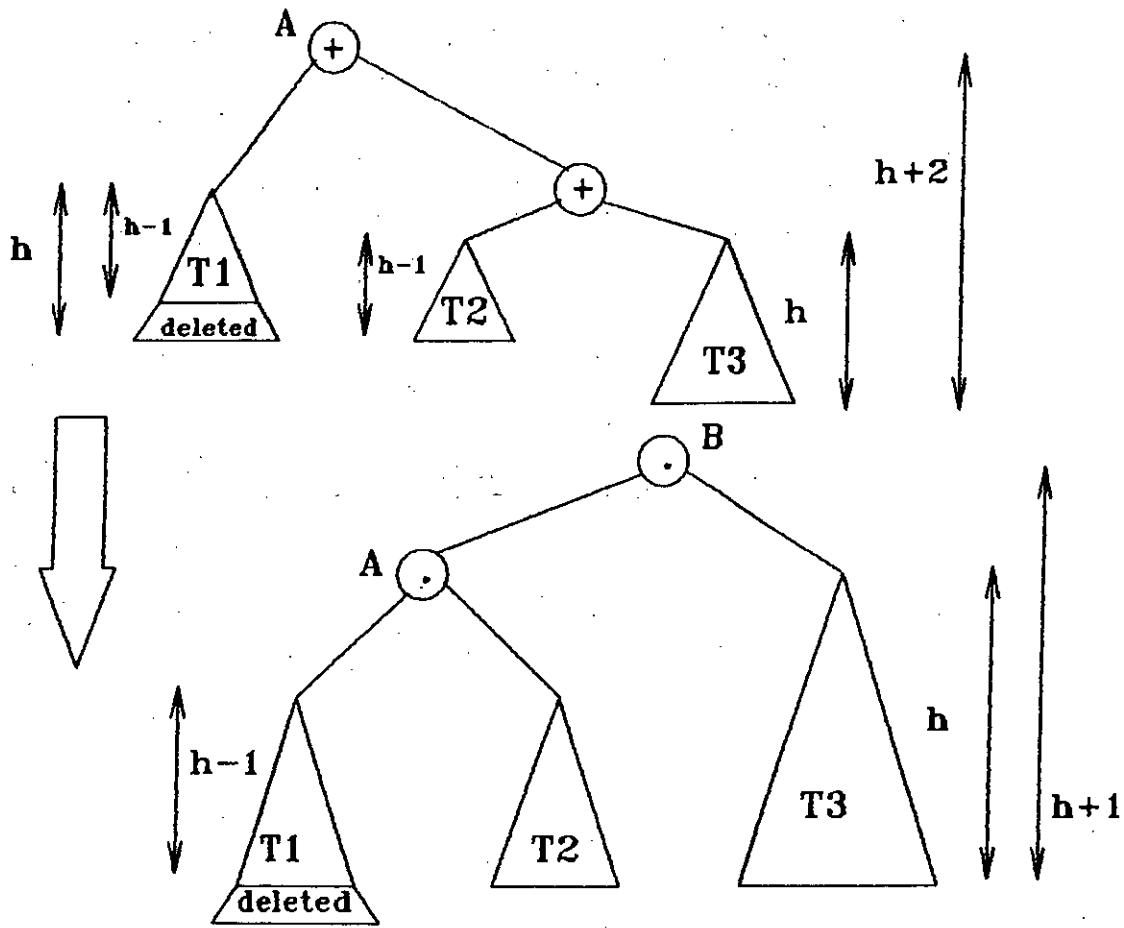


Fig. 3.6 Single rotation in times of deletion when condition code is '+'

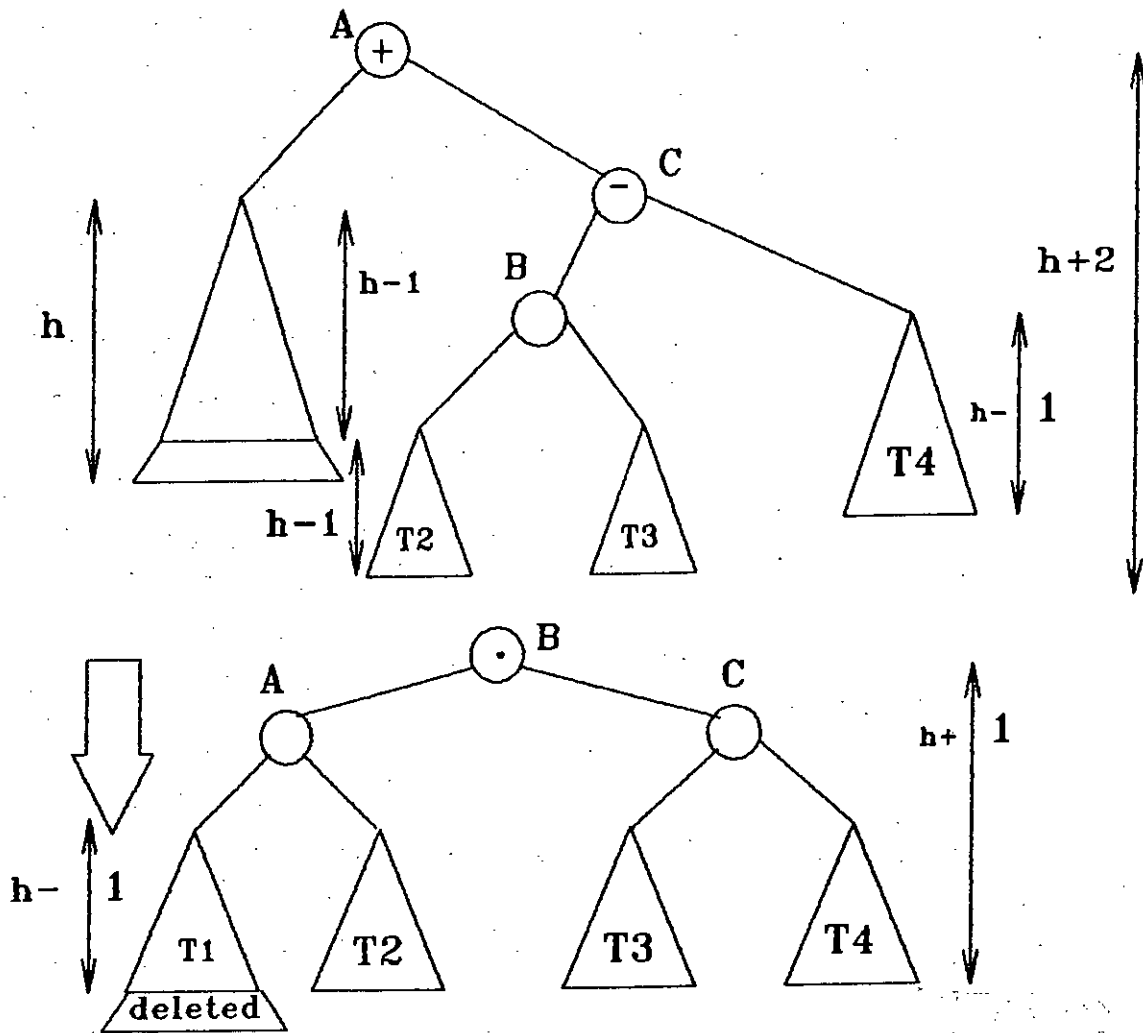


Fig. 3.7 Double rotation in times of deletion from a Balanced tree

can be done roughly as follows. As we go down in the tree to the site of the insertion, keep track of the node S that is the latest one along the path to have height condition + or -. When the insertion is done, each of the elements between S and the newly inserted element has height condition code . and each must be changed to + or -. It is at S that a rotation or double rotation may be needed. Let us work out the details of the algorithm.

Algorithm 3.1 Balanced tree search and insertion

Given a table of records which form a balanced binary tree as described earlier, this algorithm searches for a given argument k. If k is not in the table, a new node containing k is inserted into the tree in the appropriate place and the tree is rebalanced if necessary. The nodes of the tree are assumed to contain KEY, LLINK, and RLINK fields. We also have a new field B(P) = balance factor of NODE(P), the height of the right subtree minus the height of the left subtree; this field always contains either +1, 0 or -1. A special header node also appears at the top of the tree, in location HEAD; the value of RLINK(HEAD) is a pointer to the root of the tree and LLINK(HEAD) is used to keep track of the overall height of the tree. We assume that the tree is nonempty, i.e., that RLINK(HEAD) \neq A. For convenience in description, the algorithm uses the notation LINK(a,P) as a synonym for LLINK(P) if a = -1 and for RLINK(P) if a = +1.

This algorithm is rather long, but it divides into three simple parts : Steps A1 - A4 do the search, steps A5 - A7 insert a new node and steps A8 - A10 rebalance the tree if necessary.

A1. [[Initialize]] Set T \leftarrow HEAD, S \leftarrow P \leftarrow RLINK(HEAD) (The pointer variable P will move down the tree; S will point to the place where rebalancing may be necessary

and T always points to the father of S.)

A2. [[Compare]] If $K < \text{KEY}(P)$, goto A3; if $K > \text{KEY}(P)$, goto A4 and if $K = \text{KEY}(P)$, the search terminates successfully.

A3. [[Move left]] Set $Q \leftarrow \text{LLINK}(P)$. If $Q = \Lambda$, set $Q \leftarrow \text{AVAIL}$ and $\text{LLINK}(P) \leftarrow Q$ and goto step A5. Otherwise if $B(Q) \neq 0$, set $T \leftarrow P$ and $S \leftarrow Q$. Finally set $P \leftarrow Q$ and return to step A2.

A4. [[Move right]] Set $Q \leftarrow \text{RLINK}(P)$. If $Q = \Lambda$, Set $Q \leftarrow \text{AVAIL}$ and $\text{RLINK}(P) \leftarrow Q$ and goto step A5. Otherwise if $B(Q) \neq 0$, set $T \leftarrow P$ and $S \leftarrow Q$. Finally set $P \leftarrow Q$ and return to step A2.

A5. [[Insert]] (We have just linked a new node, $\text{NODE}(Q)$, into the tree and its fields need to be initialized) set $\text{KEY}(Q) \leftarrow K$, $\text{LLINK}(Q) \leftarrow \text{RLINK}(Q) \leftarrow \Lambda$, $B(Q) \leftarrow 0$.

A6. [[Adjust balance factor]] (Now the balance factors on nodes between S and Q need to be changed from zero to ± 1 .) If $K < \text{KEY}(S)$, Set $R \leftarrow P \leftarrow \text{LLINK}(S)$, otherwise set $R \leftarrow P \leftarrow \text{RLINK}(S)$. Then repeatedly do the following operations zero or more times until $P = Q$. If $K < \text{KEY}(P)$ set $B(P) \leftarrow -1$ and $P \leftarrow \text{LLINK}(P)$; If $K > \text{KEY}(P)$, set $B(P) \leftarrow +1$ and $P \leftarrow \text{RLINK}(P)$ (If $K = \text{KEY}(P)$, then $P = Q$ and we may go on to the next step.)

A7. [[Balancing act.]] If $K < \text{KEY}(S)$ set $a = -1$, otherwise set $a = +1$. Several cases now arise :

i) If $B(S) = 0$ (The tree has growth higher), set $B(S) \leftarrow a$, $\text{LLINK}(\text{HEAD}) \leftarrow \text{LLINK}(\text{HEAD}) + 1$ and terminate the algorithm.

ii) If $B(S) = -a$ (The tree has gotten more balanced), set $B(S) = 0$ and terminate the algorithm.

iii) If $B(S) = a$ (The tree has gotten out of balance). goto step A8 if $B(R) = a$, to A9 if $B(R) = -a$.

A8. [[Single rotation]] Set $P \leftarrow R$, $LINK(a,s) \leftarrow LINK(-a,R)$, $LINK(-a,R) \leftarrow S$, $B(S) \leftarrow B(R) \leftarrow 0$, goto A10.

A9. [[Double rotation]] set $P \leftarrow LINK(-a,R)$, $LINK(-a,R) \leftarrow LINK(a,P)$, $LINK(a,P) \leftarrow R$, $LINK(a,S) \leftarrow LINK(-a,P)$, $LINK(-a,P) \leftarrow S$. Now Set

$$(B(S), B(R)) \leftarrow \begin{cases} (-a, 0) & \text{if } B(P) = a \\ (0, 0) & \text{if } B(P) = 0 \\ (0, a) & \text{if } B(P) = -a \end{cases}$$

and then set $B(P) \leftarrow 0$.

A10. [[Finishing touch]] (We have completed the rebalancing transformation, with P pointing to new root and T pointing to the father of the old root.) If $S = RLINK(T)$ then set $RLINK(T) \leftarrow P$, otherwise set $LLINK(T) \leftarrow P$.

3.5 Some interesting empirical results about balanced trees.

In the first place we can ask about the number B_{nh} of balanced binary trees with n internal nodes and height h . It is not difficult to compute the generating function $B_h(z) = \sum_{n \geq 0} B_{nh} z^n$ for small h , from the relations

$$B_0(z) = 1, B_1(z) = z, B_{h+1}(z) = zB_h(z)(B_h(z) + 2B_{h-1}(z)) \quad (3.6)$$

Equation (3.6) derives itself from the fact that the coefficient of z^n in $zB_i(z)B_j(z)$ is the number of n node binary trees with left subtree of height i and with right subtree of height j .

Thus

$$B_2(z) = 2z^2 + z^3$$

$$B_3(z) = 4z^4 + 6z^5 + 4z^6 + z^7$$

$$B_4(z) = 16z^7 + 32z^8 + 44z^9 + \dots + 8z^{14} + z^{15}$$

The total number of balanced trees with height h is $B_h = B_h(1)$, which satisfies the recurrence

$$B_0 = B_1 = 1, B_{h+1} = B_h^2 + 2B_h B_{h-1} \quad (3.7)$$

so that $B_2 = 3, B_3 = 3.5, B_4 = 3^2 \cdot 5 \cdot 7, B_5 = 3^2 \cdot 5^2 \cdot 7 \cdot 23$ and in general

$$B_h = A_0^{F_h} \cdot A_1^{F_{h-1}} \cdots A_{h-1}^{F_1} \cdot A_h^{F_0} \quad (3.8)$$

Where $A_0 = 1, A_1 = 3, A_2 = 5, A_3 = 7, A_4 = 23, A_5 = 347, \dots, A_h = A_{h-1} B_{h-2} + 2$.

The sequences B_h and A_h grow very rapidly, in fact they are doubly exponential. If we consider each of the B_h trees to be equally likely, we can roughly compute the average number of nodes in a tree of height h as $(0.70118)2^h \dots \dots \dots (3.9)$

This indicates that the height of a balanced tree with n nodes usually is much closer to $\log_2 n$ than to $\log_3 n$. Unfortunately these results do not bear the true picture of the above mentioned algorithm, since this algorithm makes some trees much more probable than others. For example consider the case $N = 7$, where 17 balanced trees are possible. There are $7! = 5040$ possible orderings in which seven keys can be inserted and the perfectly balanced complete tree is obtained 2160 times. The fact that perfect balanced tree is obtained with such high probability together with (3.9), which corresponds to the case of equal probability - makes it extremely plausible that the average search time for a balanced tree is about $\log_2 N + C$ comparisons for some small constant C . Empirical tests support this conjecture. The average number of comparisons needed to insert the N th item seems to be approximately $\log_2 N + .25$ for large N .

An approximate model of the behaviour of algorithm 3.1 can be established which is not rigorously accurate, but it is close enough to the truth to give some insight. Let us assume that P is the probability that the balance factor of a given node in a large

tree built by algorithm 3.1 is 0; then the balance factor is + 1 with probability $\frac{1}{2}(1 - P)$, and it is - 1 with the same probability $\frac{1}{2}(1 - P)$.

Let us assume further (without justification) that the balance factors of all nodes are independent. Then the probability that step A6 set exactly $k - 1$ balance factors nonzero is $P^{k-1}(1 - p)$; so the average value of $K - 1$ is $P/(1 - P)$. The probability that we need to rotate part of the tree is $1/2$. Inserting a new node should increase the number of balanced nodes by P , on the average; this number is actually increased by 1 in step A5, by $-P/(1 - P)$ in step A6, by $1/2$ in step A7 and by $\frac{1}{2} \cdot 2$ in step A8 or A9, so we should have

$$P = 1 - \frac{P}{1 - p} + \frac{1}{2} + 1$$

Solving P we have $P \approx 0.649$ and $\frac{P}{1 - P} \approx 1.851$. This agrees with the fact that about 68% of all nodes were found to be balanced in random trees built by algorithm 3.1.

Some other ways of organizing trees, so as to guarantee logarithmic access time, have been proposed. We can promote the height balanced trees which arise when we allow the height difference of subtrees to be greater than one; but at most four (say).

Another alternative to balanced trees called 2-3 trees incorporates the idea that they have either 2-way or 3-way branching at each node and to stipulate that all external nodes appear on the same level. An interesting tree representation for 2-3 trees has also been suggested by using one extra bit per node. The next section of this chapter travels through these variations of balanced trees.

3.6 2-3-4 Trees.

In order to construct the top - down 2-3-4 trees some flexibility in the data structures are needed. To get this flexibility, let us assume that the nodes in the trees can hold more than one key. Specifically, we will allow 3-nodes and 4-nodes, which can hold

two and three keys respectively. A 3-node has three links coming out of it, one for all records with keys smaller than both its keys, one for all records with keys in between its two keys and one for all records with keys larger than both its keys. Similarly, a 4-node has four links coming out of it, one for each of the intervals defined by its three keys. We will see below some efficient ways to define and implement the basic operations on these extended nodes; for now, let us assume we can manipulate them conveniently and see how they can be put together to form trees.

For example Figure 3.8 shows a 2-3-4 tree. It is easy to see how to search in such a tree. For example, to search for 15 in the tree in figure 3.8, we would follow the middle link from the root, since 15 is between 12 and 40, then terminate the unsuccessful search at the left link from the node containing 20, 30 and 35.

To Insert a new node in a 2-3-4 tree, we would like to do an unsuccessful search and then hook the the node on. It is easy to see what to do if the node at which the search terminates is a 2-node: we can just turn it into a 3-node. For example, 50 could be added to the tree in figure 3.8 by adding it to the node containing 45. Similarly, a 3-node can easily be turned into a 4-node. But what should we do if we need to insert new node into a 4-node? For example, how will this be done if we insert 15 into the tree in Figure 3.8? One possibility would be to hook it on as a new leftmost child of the 4-node containing 20, 30, and 35 but a better solution is shown in Figure 3.9: first we will split the 4-node into two 2-nodes and pass one of its keys up to its parent. First the 4-node containing 20, 30, and 35 is split into two 2-nodes (one containing 20, the other containing 35) and the "middle key" 30 is passed up to the 3-node containing 12 and 40, turning it into a 4-node. Then there is room for 15 in the 2-node containing 20.

But what if we need to split a 4-node whose parent is also a 4-node? One method

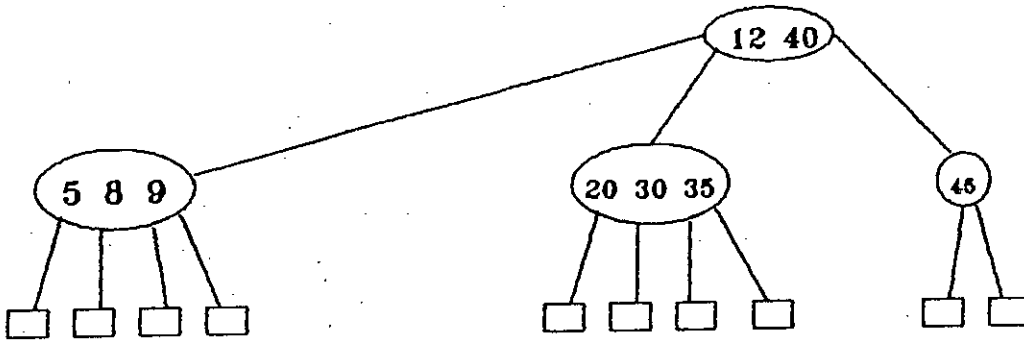


Fig 3.8 A 2-3-4 Tree

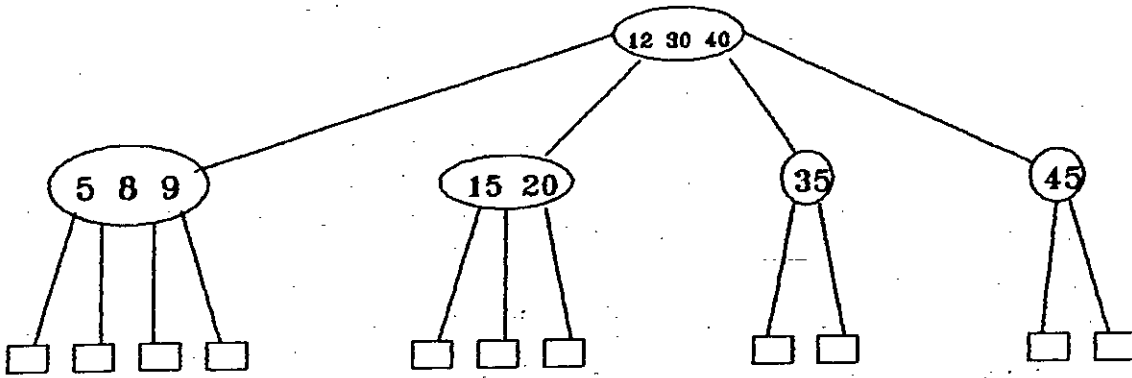


Fig 3.9 Insertion of 15 into a 2-3-4 tree

would be to split the parent also, but we could keep having to do this all the way back up the tree. An easier way is to make sure that the parent of any node won't be a 4-node by splitting any 4-node we see on the way down the tree.

All what we need to do is to insert new nodes into 2-3-4 trees by doing a search and splitting 4-nodes on the way down the tree. Specifically, every time we encounter a 2-node connected to a 4-node, we should transform it into a 3-node connected to two 2-nodes and every time we encounter a 3-node connected to a 4-node, we should transform it into a 4-node connected to two 2-nodes which are shown in the figure 3.10.

The crucial point is that these transformations are purely "local" : no part of the tree need be examined or modified other than that shown in Figure 3.10. Each of the transformations passes up one of the keys from a 4-node to its parent in the tree and restructures links accordingly. Note that we needn't worry explicitly about the parent being a 4-node, since our transformations ensure us that as we pass through each node in the tree, we come out on a node that is not a 4-node. In particular, when we come out the bottom of the tree, we are not on a 4-node, and we can insert the new node directly by transforming either a 2-node to a 3-node or a 3-node to a 4-node. Actually, it is convenient to treat the insertion as a split of an imaginary 4-node at the bottom which passes up the new key to be inserted. Whenever the root of the tree becomes a 4-node, we'll split it into three 2-nodes, as we did for our first node split in the example as shown in the Figure 3.11. This (and only this) makes the tree grow one level "higher".

The algorithm sketched above gives a way to do searches and insertions in 2-3-4 trees: since the 4-nodes are split up on the way from the top down: the trees are called top down 2-3-4 trees. What's interesting is that, even though we haven't been worrying about balancing at all, the resulting trees are perfectly balanced !

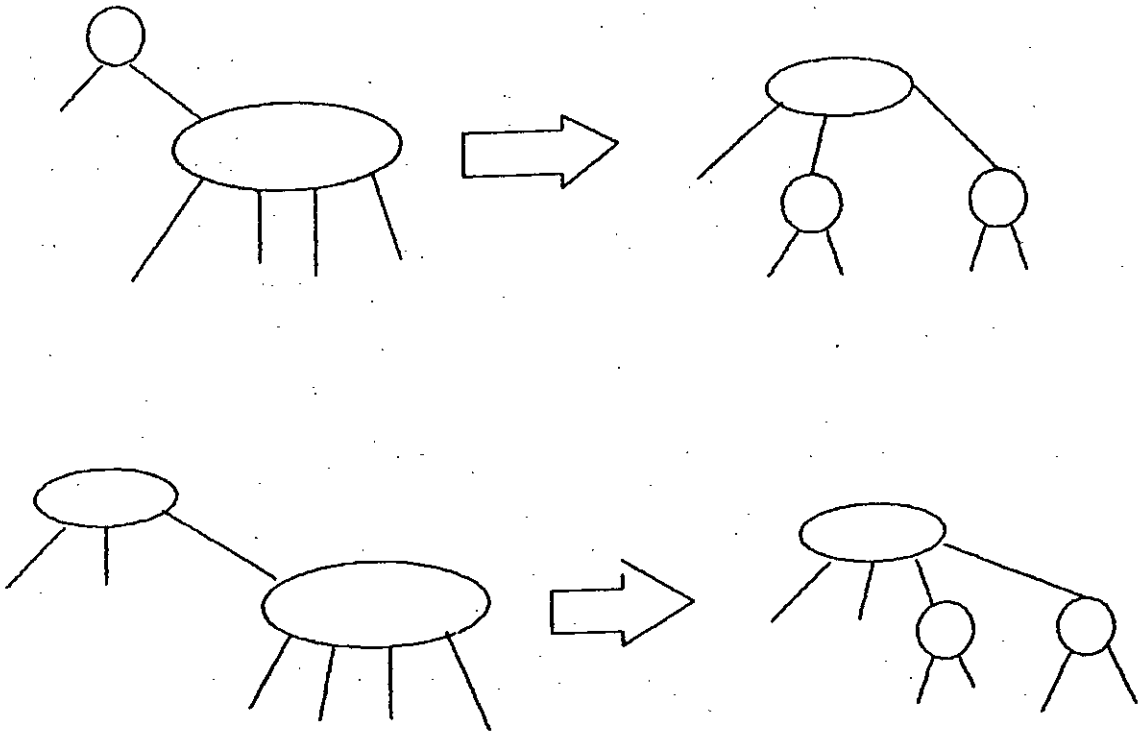


Fig. 3.10 Splitting 4 nodes

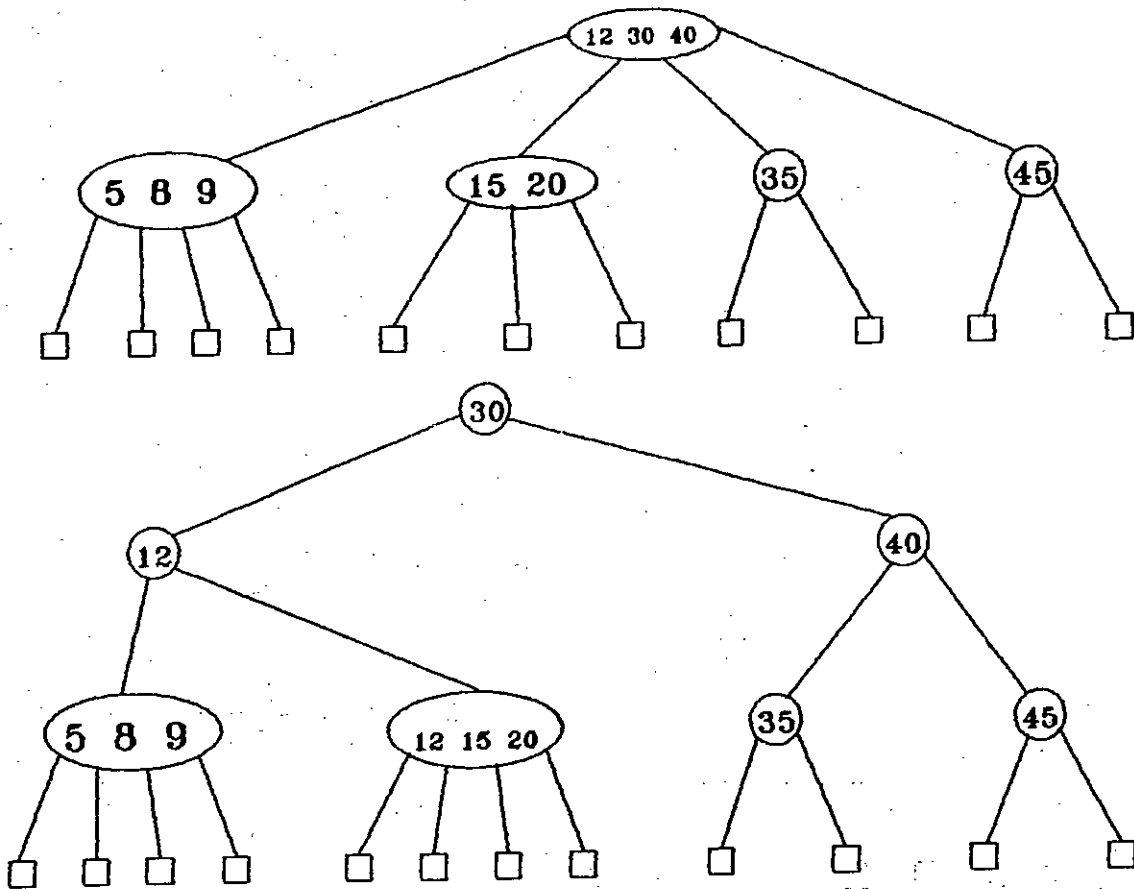


Fig. 3.11 Splitting of root node in a 2-3-4 tree

The distance from the root to every external node is the same: the transformation that we perform have no effect on the distance from any node to the root, except when we split the root, and in this case the distance from all nodes to the root is increased by one. If all the nodes are 2-nodes then searches in N-node trees never visit more than $\log_2 N + 1$ nodes, since the tree is like a full binary tree; if there are 3-nodes and 4-nodes, the height can only be lower.

Insertions into N-node 2-3-4 trees require fewer than $\log_2 N + 1$ node split in the worst case and seem to require less than one node split on the average. The worst thing that can happen is that all the nodes on the path to the insertion point are 4-nodes, all which would be split. But in a tree built from a random permutation of N elements, not only is this worst case unlikely to occur, but also few splits seem to be required on the average, because there are not many 4-nodes. Analytical results on the average case performance of 2-3-4 trees have so far eluded the experts, but empirical studies consistently show that very few splits are done.

3.7 RED BLACK Trees.

The description given in section 3.6 is sufficient to define an algorithm for searching using 2-3-4 trees which has guaranteed good worst case performance. However, we are only halfway towards an actual implementation. While it would be possible to write algorithms which actually perform transformations on distinct data types representing 2-, 3-, and 4-nodes, most of the things that need to be done are very inconvenient in this direct representation. Furthermore, the overhead incurred in manipulating the more complex node structures is likely to make the algorithms slower than standard binary-tree search. The primary purpose of balancing is to provide "insurance" against a bad worst case, but it would be unfortunate to have to pay the overhead cost for

that insurance on every run of the algorithm. Fortunately, as we'll see below there is a relatively simple representation of 2-, 3-, and 4-nodes that allows the transformations to be done in a uniform way with very little overhead beyond the cost incurred by standard binary tree-search.

Remarkably, it is possible to represent 2-3-4 trees as standard binary trees (2-nodes only) by using only one extra bit per node. The idea is to represent 3-nodes and 4-nodes as small binary trees bound together by "red" links; these contrast with the "black" links that bind the 2-3-4 tree together. The representation is simple: as shown in Figure 3.12, 4-nodes are represented as three 2-nodes connected by red links and 3-nodes are represented as two 2-nodes connected by a red link (red links are drawn as thick lines). (Either orientation is legal for a 3-node.)

Figure 3.14 shows one way to represent the tree of Figure 3.13. If we eliminate the red links and collapse together the nodes they connect, the result is 2-3-4 tree in Figure 3.13. The extra bit per node is used to store the color of the link pointing to that node: we'll refer to 2-3-4 trees represented in this way as red black trees.

These trees have many structural properties that follow directly from the way in which they are defined. For example, there are never two red links in a row along any path from the root to an external node, and all such paths have an equal number of black links. Note that it is possible that one path (alternating black-red) be twice as long as another (all black), but that all path lengths are still proportional to $\log_2 N$.

One very nice property of red-black trees is that the tree search procedure for standard binary tree search works without modification (except for the matter of duplicate keys). We will implement the link colors by adding a boolean field `red` to each node which is true if the link pointing to the node is red; false if it is black; the tree search pro-

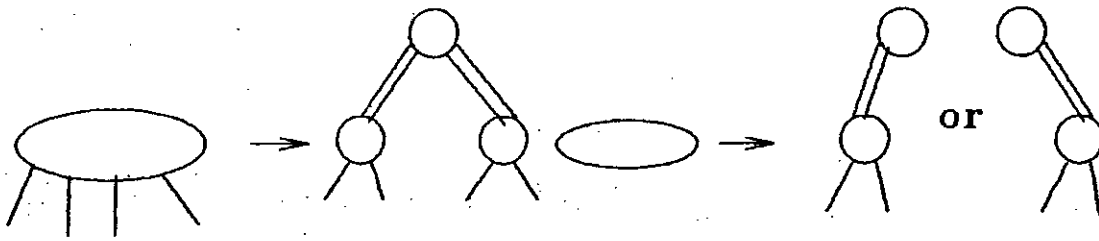


Fig. 3.12 Red-Black Representation of 3-nodes and 4-nodes

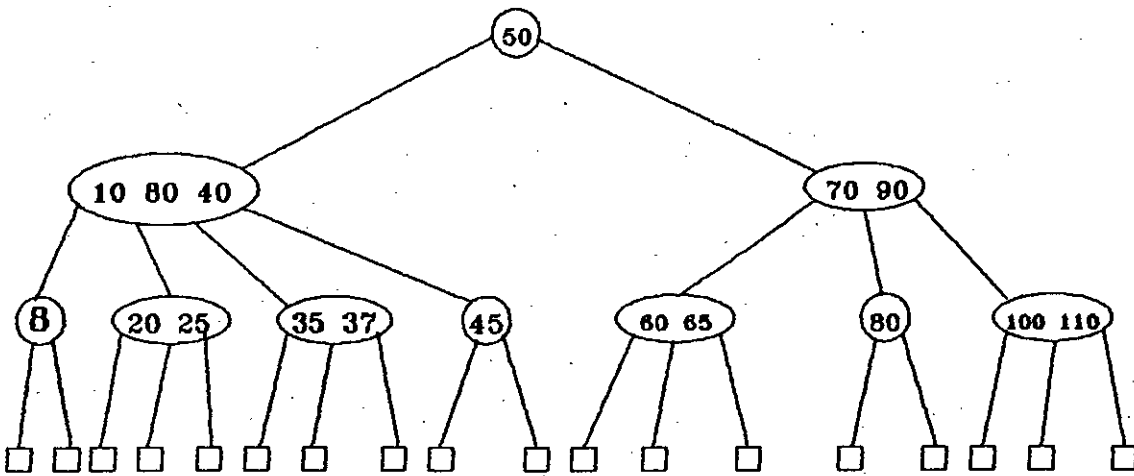


Fig. 3.13 A 2-3-4 Tree

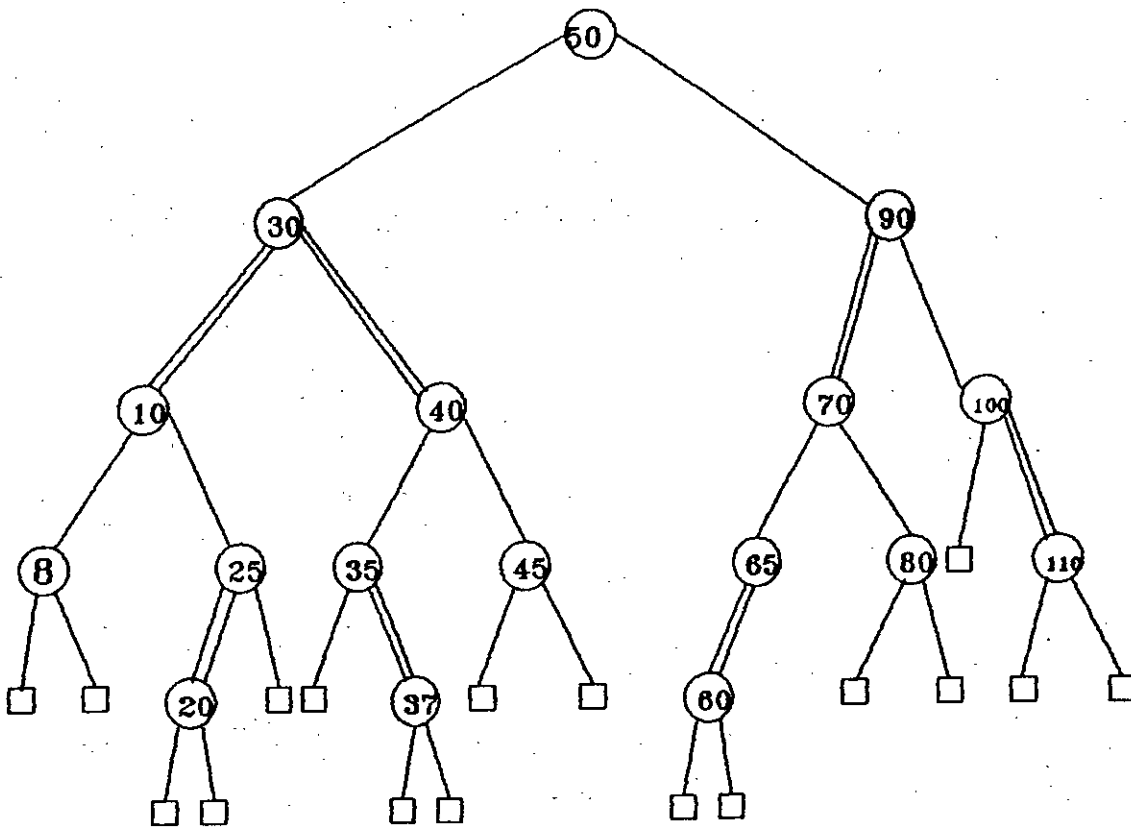


Fig. 3.14 A Red-Black tree

cedure simply never examines that field. Thus, no "overhead" is added by the balancing mechanism to the time taken by the fundamental searching procedure. Since each key is inserted just once, but may be searched for many times in a typical application, the end result is that we get improved search times (because the trees are balanced) at relatively little cost (because no work for balancing is done during the searches). Below is the description of searching algorithm for Red - Black trees which in turn calls the necessary procedures as described later on.

Algorithm 3.2 (Red Black Tree Search and Insertion)

```
type link = ↑ node
node = record key : integer; llink, rlink : link;
red : boolean end;
var head, z, x : link;
searchkey : integer

new (z); z ↑ .llink = z; z ↑ .rlink = z; z ↑ .red = false

new (head); head ↑ key = 0; head ↑ .rlink = z

searchkey = rand()

x = search (head, searchkey)
```

```

if ( $x \neq z$ ) print : The search is successful
else rbtreeinsert (searchkey, head) [[ The search is unsuccessful ]]
endif

```

End Algorithm 3.2

Moreover, the overhead for insertion is very small: we have to do something different only when we see 4-nodes, and there aren't many 4-nodes in the tree because we're always breaking them up. The inner loop needs only one extra test (if a node has two red children, it's a part of a 4-node), as shown in the following implementation of the insert procedure :

```

rbtreeinsert (searchkey : integer; head : link)

    gg, g, p, x : link
    x = head; p = head; g = head;
    while ( $x \neq z$ ) do
        gg = g; g = p; p = x

        if (searchkey <  $x \uparrow$  key) then  $x = x \uparrow$  llink
        else  $x = x \uparrow$  rlink
        endif

        if ( $x \uparrow$  llink  $\uparrow$  red) and ( $x \uparrow$  rlink  $\uparrow$  red) then
            x = split (searchkey, gg, g, p, x);
        endif
    endif

```



```

repeat
new(x); x ↑ key = searchkey; x ↑ llink = z; x ↑ rlink = z;

if (searchkey < p ↑ key) then p ↑ llink = x
else p ↑ rlink = x
endif

x = split ( searchkey, gg, g, p, x)

```

End Procedure rbtreeninsert

In this program, x moves down the tree and gg, g, p are kept pointing to x 's great grandfather, grandparent, and parent in the tree. To see why all these links are needed, let us consider the addition of 120 to the tree of Fig. 3.14. When the the external node at the right of the 3-node containing 100 and 110 is reached, gg is 90, g is 100 and p is 110. Now 120 must be added to make a 4-node containing 100, 110 and 120, resulting in the tree shown in Fig. 3.15.

We need a pointer to 90 (gg) because 90's right link must be changed to point to 110, not 100. To see exactly how this comes about, we need to look at the operation of the split procedure. Let us consider the red-black representation for the two transformations we must perform: if we have a 2 node connected to a 4-node, then we should convert them into a 3-node connected to two 2-nodes; if we have a 3-node connected to a 4-node, we should convert them into a 4-node connected to two 2-nodes. When a new node is added at the bottom, it is considered to be the middle node of an imaginary 4-node (that is, think of z as being red, though this is never explicitly tested).

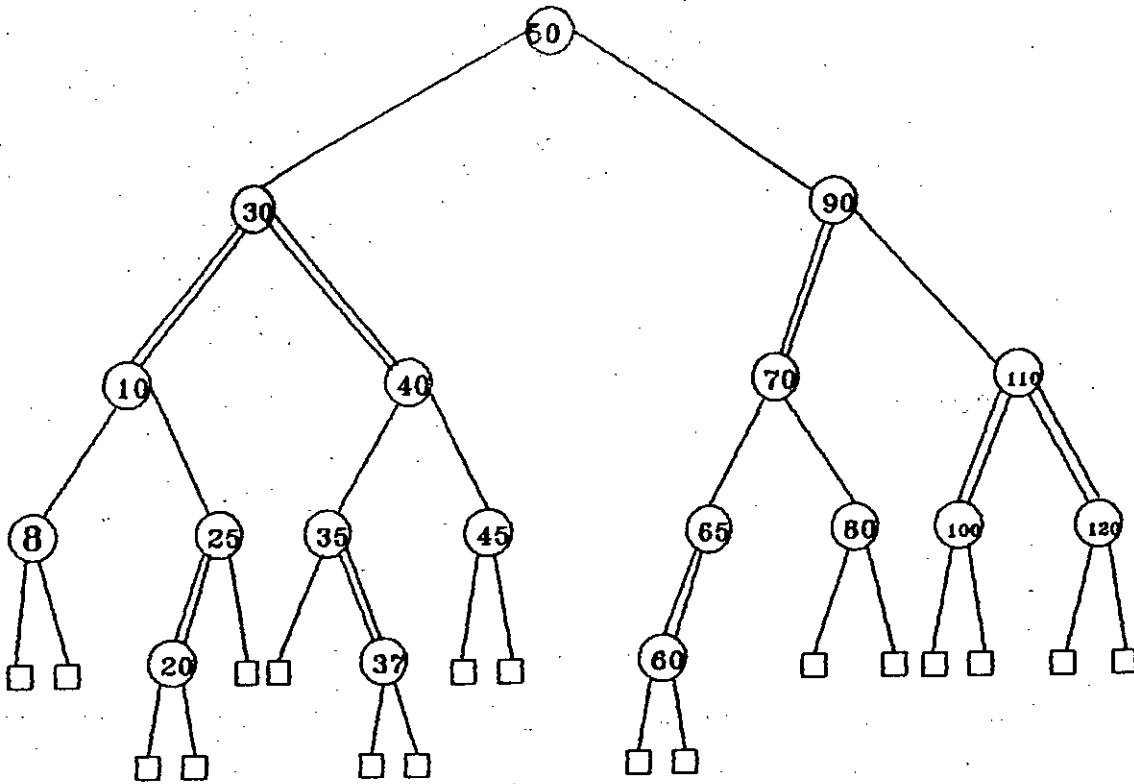


Fig. 3.15 Insertion of 120 into a Red-Black tree

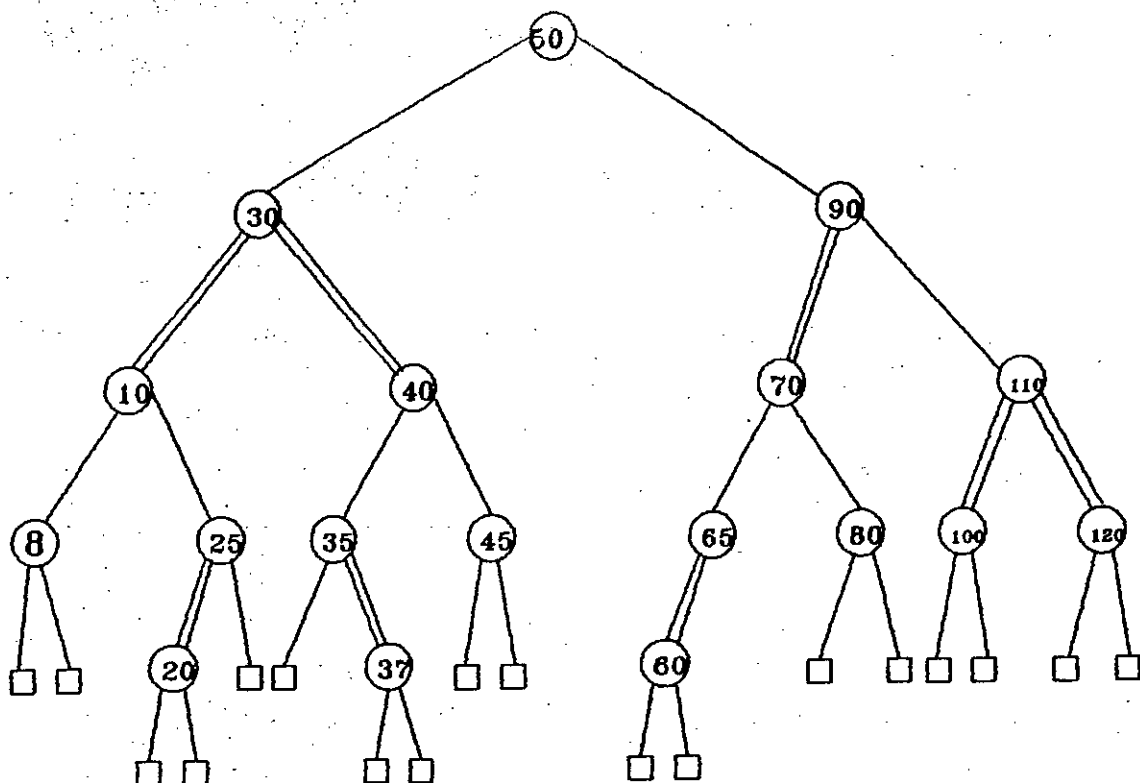


Fig. 3.15 Insertion of 120 into a Red-Black tree

The transformation required when we encounter a 2-node connected to a 4-node is easy, and the same transformation works if we have a 3-node connected to a 4-node in the "right" way, as shown in Figure 3.16. Thus, split begins by marking x to be red and the children of x to be black.

This leaves the two other situations that can arise if we encounter a 3-node connected to a 4-node, as shown in Figure 3.17. (Actually, there are four situations, since the mirror images of these two can also occur for 3-nodes of the other orientation.) In these cases, splitting the 4-node has left two red links in a row, an illegal situation which must be corrected. This is easily tested for in the code: we just marked x red, so if x 's parent is also red, we must take further action. The situation is not too bad because we do have three nodes connected by red links: all we need do is transform the tree so that the red links point down from the same node.

There is a simple operation which achieves the desired effect. Let us begin with the easier of the two, the first (top) case from Figure 3.17, where the red links are oriented the same way. The problem is that the 3-node was oriented the wrong way: accordingly, we restructure the tree to switch the orientation of the 3-node and thus reduce this case to be the same as the second case from Figure 3.16, where the color flip of x and its children was sufficient. Restructuring the tree to reorient a 3-node involves changing three links, as shown in Figure 3.18; note that Figure 3.18 is the same as Figure 3.15, but with the 3-node containing 70 and 90 rotated. The left link of 90 was changed to point to 80, the right link of 70 was changed to point to 90, and the right link of 50 was changed to point to 70. Also note carefully that the colors of the two nodes are switched.

This single rotation operation is defined on any binary search tree (if we disregard

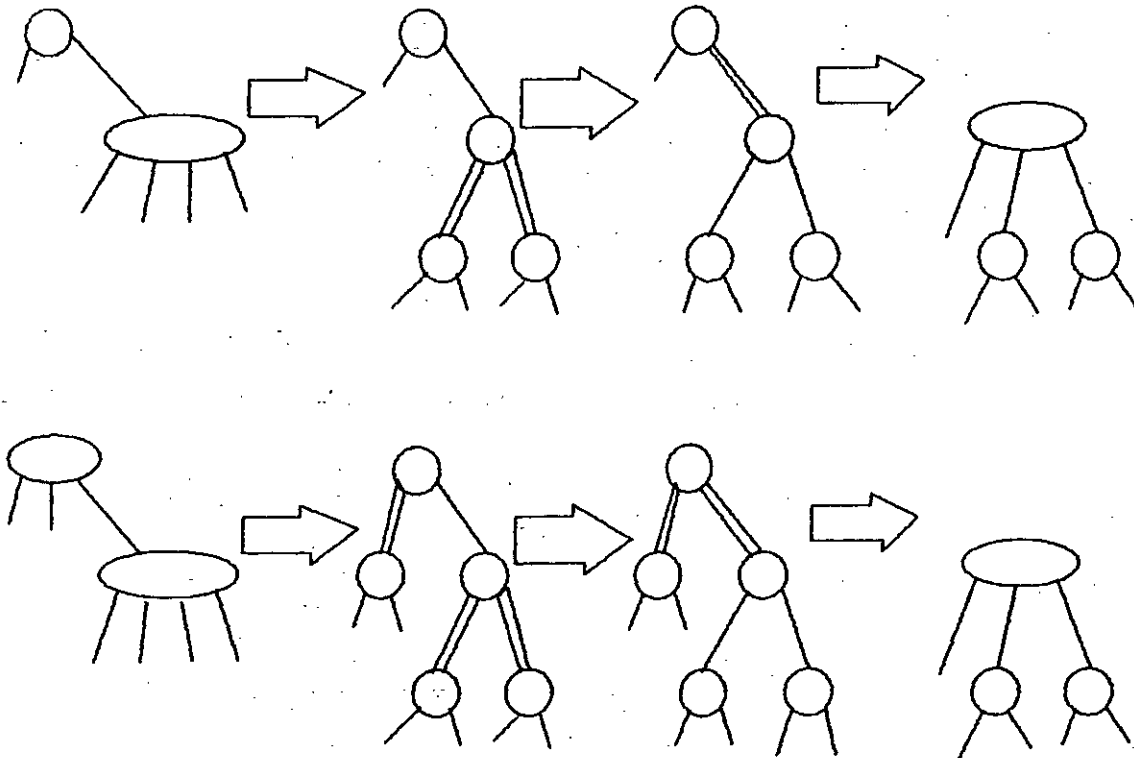


Fig. 3.16 Splitting 4-nodes with a color flip

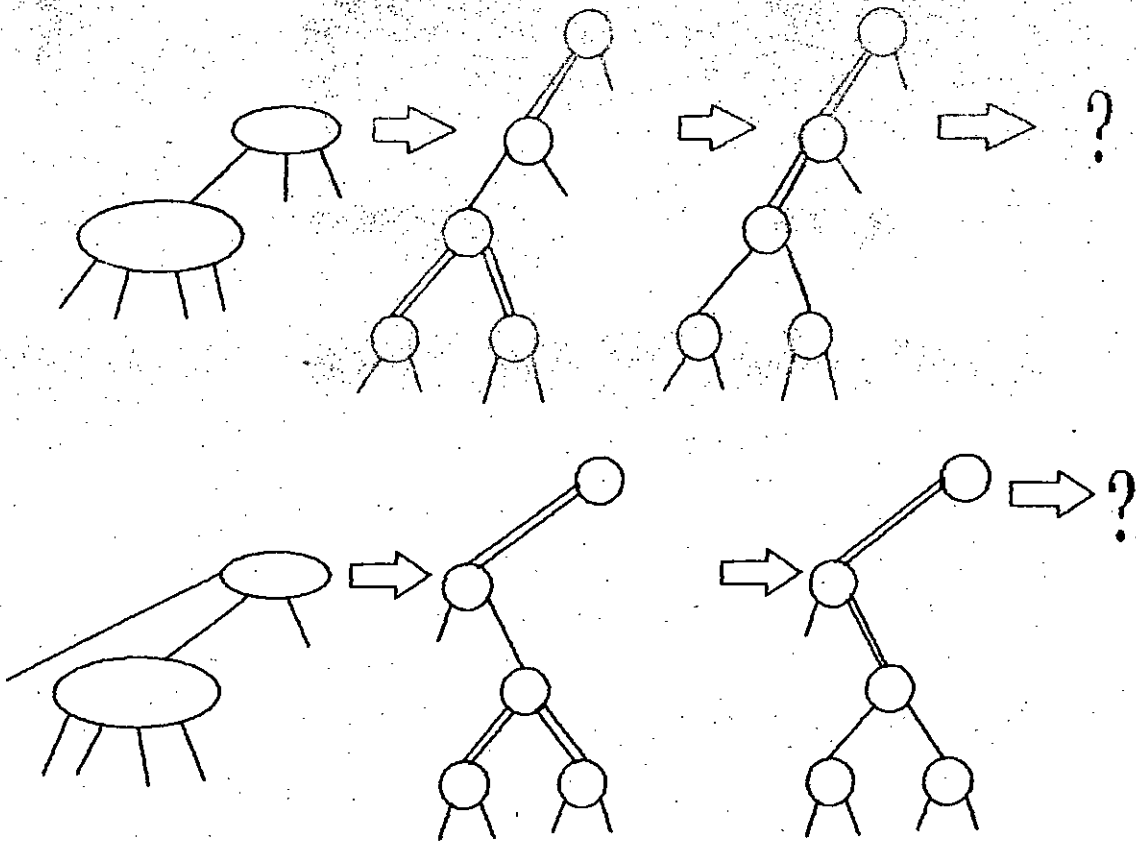


Fig. 3.17 Splitting 4-nodes with color flip: Rotation needed

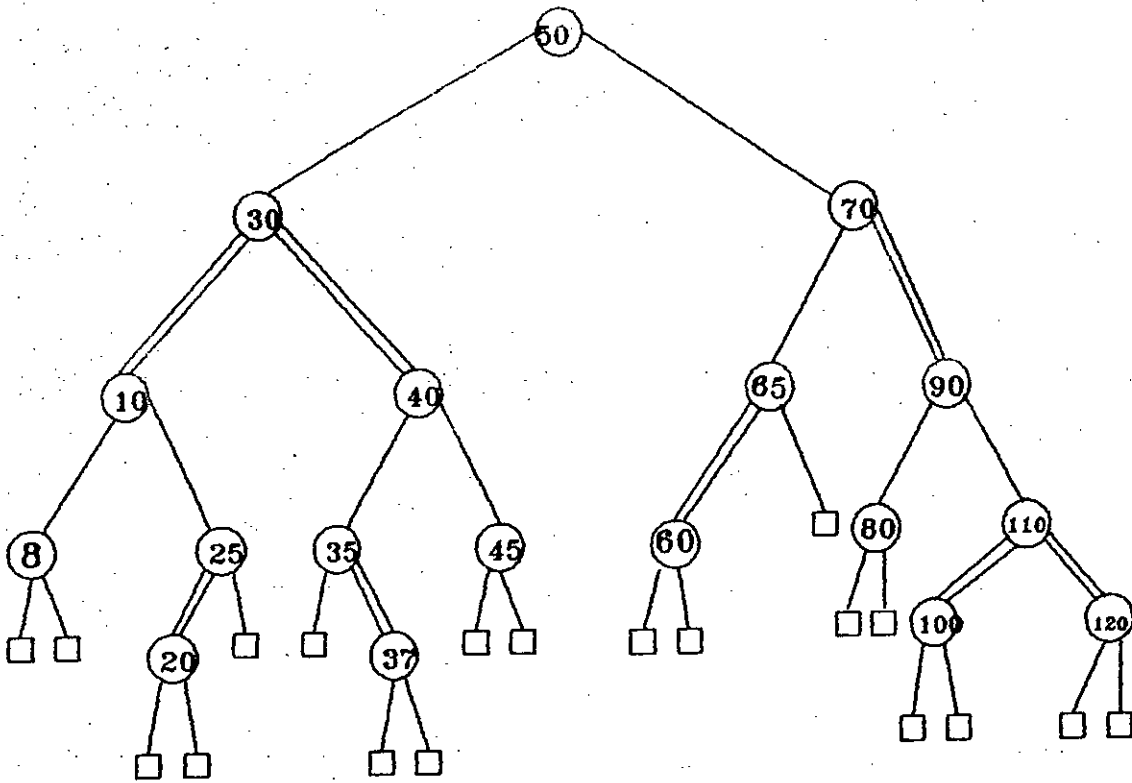


Fig. 3.18 Rotating a 3-node in fig. 3.15

operations involving the colors) and is the basis for several balanced-tree algorithms, because it preserves the essential characters of the search tree and is local modification involving only tree link changes. It is important that doing a single rotation does not necessarily improve the balance of the tree. In Figure 3.18, the rotation brings all the nodes to the left of 70 one step closer to the root, but all the nodes to the right of 90 are lowered one step. In this case the rotation makes the tree less, not more balanced. Top-down 2-3-4 trees may be viewed simply as a convenient way to identify single rotations which are likely to improve the balance.

Doing a single rotation involves modifying the structure of the tree, something that should be done with caution. When considering the rotation algorithm, the code is more complicated than might seem necessary because there are a number of similar cases with left-right symmetries. For example, suppose that the links *y*, *c*, and *gc* point to 50, 90, and 70 respectively in Figure 3.15. Then the transformation to Figure 3.18 is effected by the link changes $c \uparrow .llink = gc \uparrow .rrlink; gc \uparrow .rrlink = c; y \uparrow .rlink = gc$.

There are three other analogous cases: the 3-node could be oriented the other way or it could be on the left side of *y* (oriented other way). A convenient way to handle these four different cases is to use the search key *u* to rediscover the relevant child (*c*) and grand-child (*gc*) of the node *y*, (we know that we will only be reorienting a 3-node if the search took us to its bottom node.) This leads to some what simpler code than alternative of remembering during the search not only the two links corresponding to *c* and *gc* but also whether they are right or left links. We have the following function for reorienting a 3-node along the search path for *v* whose parent is *y*:

```
rotate ( v : integer; y : link ) : link
```


$c, gc : \text{link}$

if ($v < y \uparrow \text{key}$) then $c = y \uparrow \text{llink}$

else $c = y \uparrow \text{rlink}$

endif

if ($v < c \uparrow \text{key}$ then)

$gc = c \uparrow \text{llink}; c \uparrow \text{llink} = gc \uparrow \text{rlink}; gc \uparrow \text{rlink} = c$

else

$gc = c \uparrow \text{rlink}; c \uparrow \text{rlink} = gc \uparrow \text{llink}; gc \uparrow \text{llink} = c$

endif

if ($v < y \uparrow \text{key}$) then $y \uparrow \text{llink} = gc$

else

$y \uparrow \text{rlink} = gc$

endif

return (gc)

End Procedure rotate

If y points to the root, c is the right link of y and gc is the left link of c , this makes exactly the link transformations needed to produce the tree in Figure 3.18 from 3.15. We may check the other cases. This function returns the link to the top of the 3-node, but does not do the color switch itself.

Thus, to handle the third case for split (see Figure 3.17), we can make g red, then set x to rotate (v, gg), then make x black. This reorients the 3-node consisting of the

two nodes pointed to by g and P and reduces this case to be the same as the second case, when the 3-node was oriented the right way.

Finally, to handle the case when the two red links are oriented in different directions (see Figure 3.17), we simply set P to rotate (v.g). This orients the "illegal" 3-node consisting of the two nodes pointed to by P and x . These nodes are the same color, so no color change is necessary, and we are immediately reduced to the third case. Combining this and the rotation for the third case is called a double rotation for obvious reasons.

Figure 3.19 shows the split occurring in our example when 40 is added. First, there is a color flip to split up the 4-node containing 45, 50, and 70. Next, a double rotation is needed: the first part around the edge between 50 and 90, and the second part around the edge between 30 and 50. After these modifications, 40 can be inserted on the left of 45, as shown in the tree in Figure 3.20.

This completes the description of the operations to be performed by split. It must switch the colors of x and its children, do the bottom part of a double rotation if necessary and then do the single rotation if necessary, as follows :

```
split ( v : integer; gg, g, p, x : link ) : link
```

```
x | red = true; x | llink | red = false; x | rlink | red = false;
```

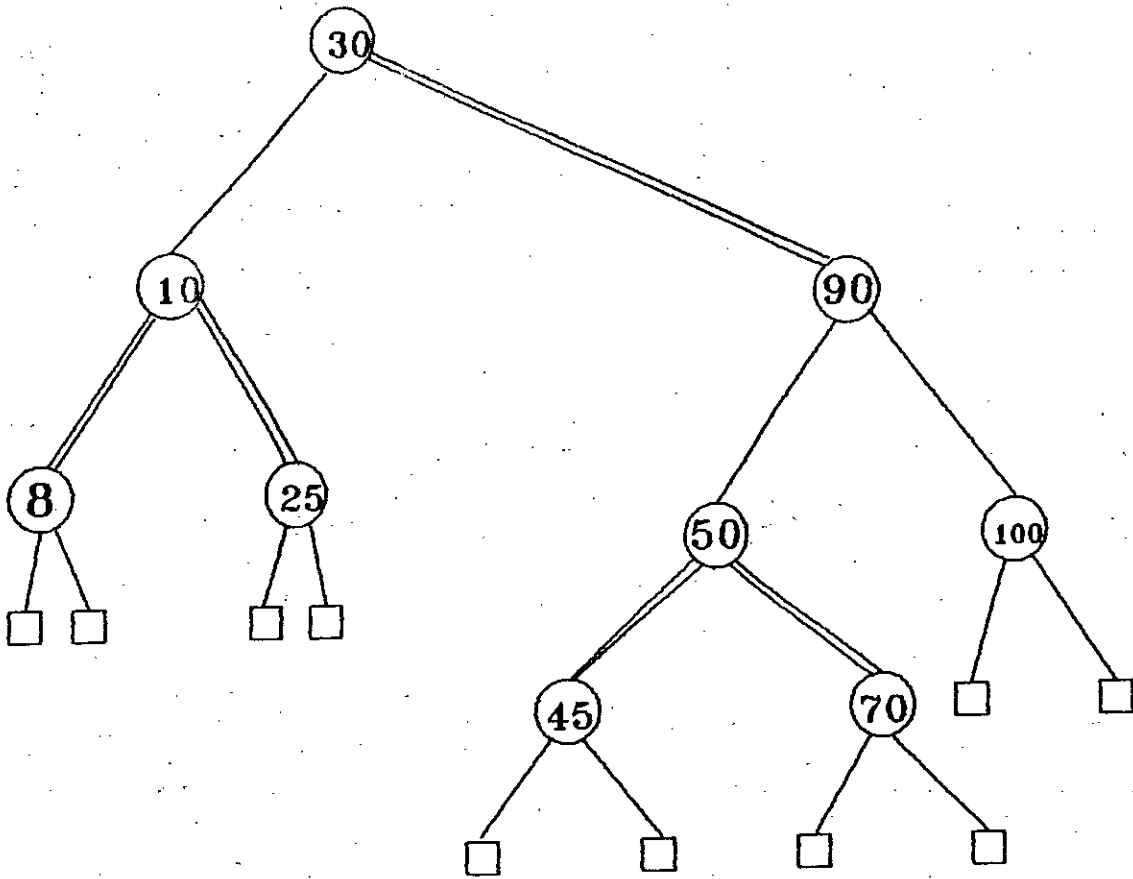


Fig. 3.19 Splitting a node in a Red-Black tree

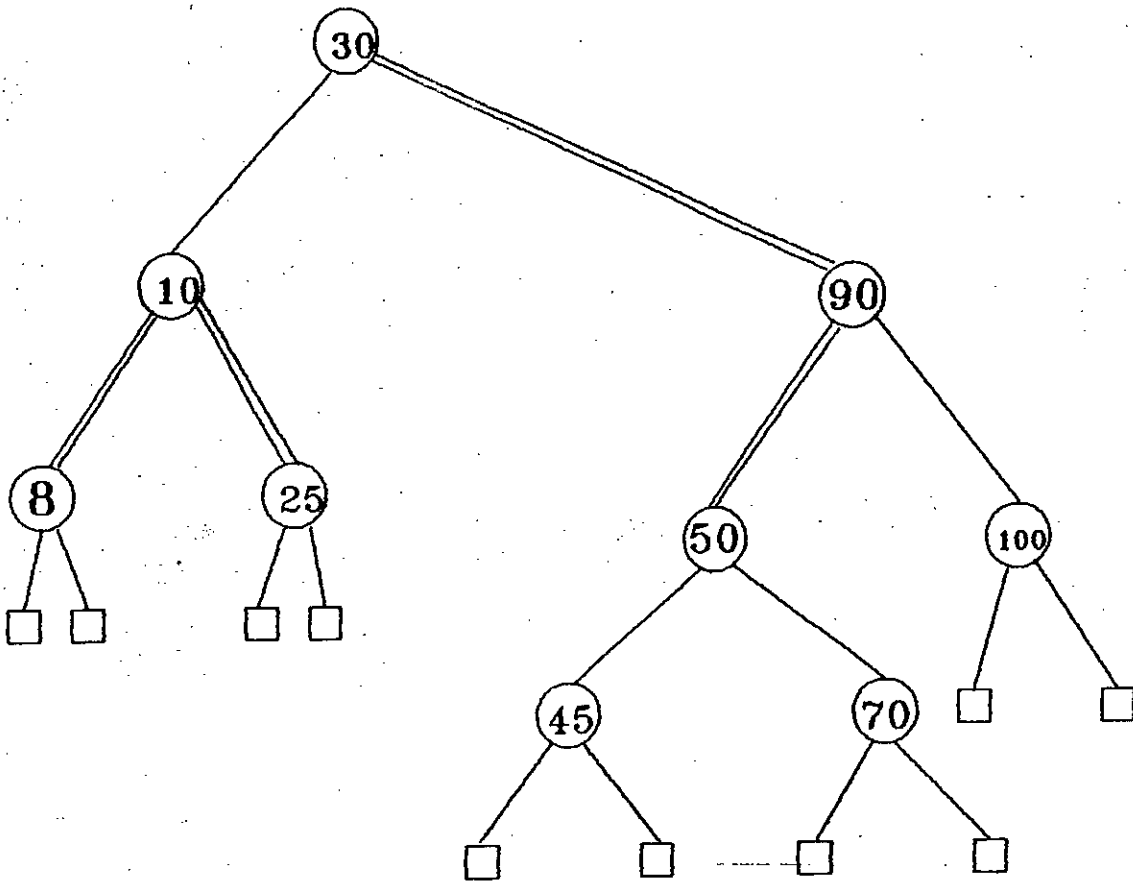


Fig. 3.19 continued

Fig. 3.19 continued

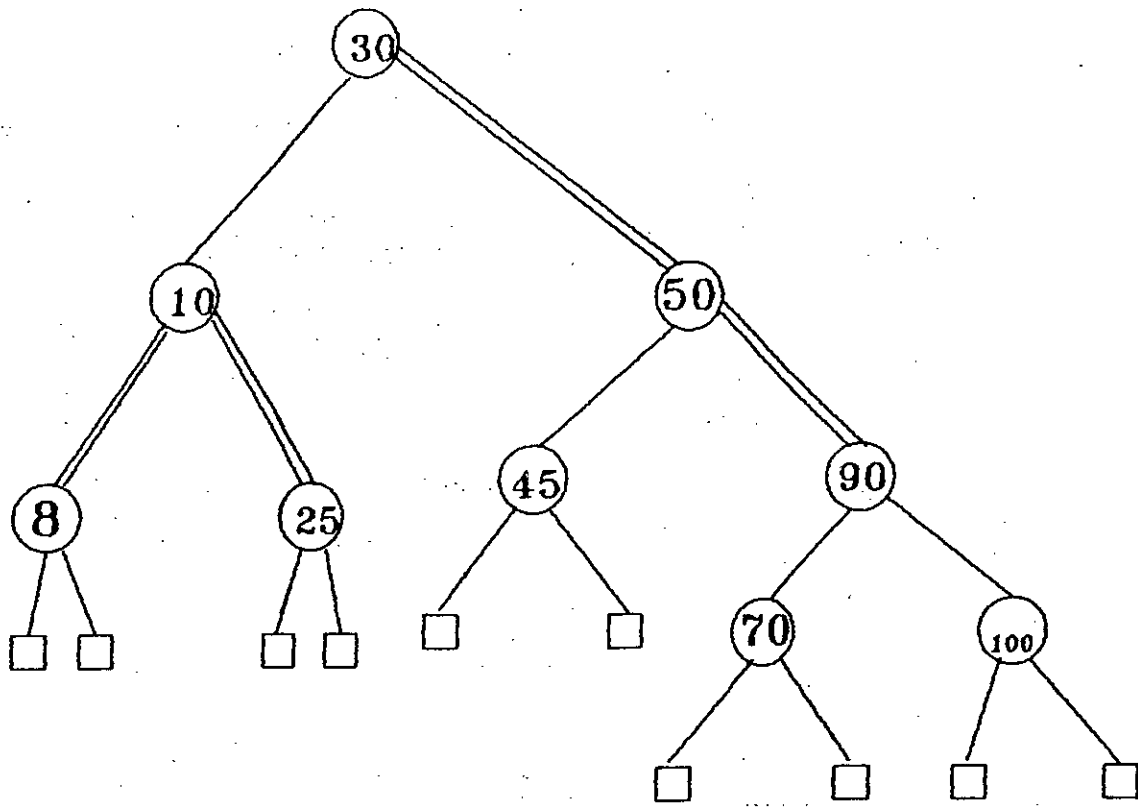


Fig. 3.19 continued

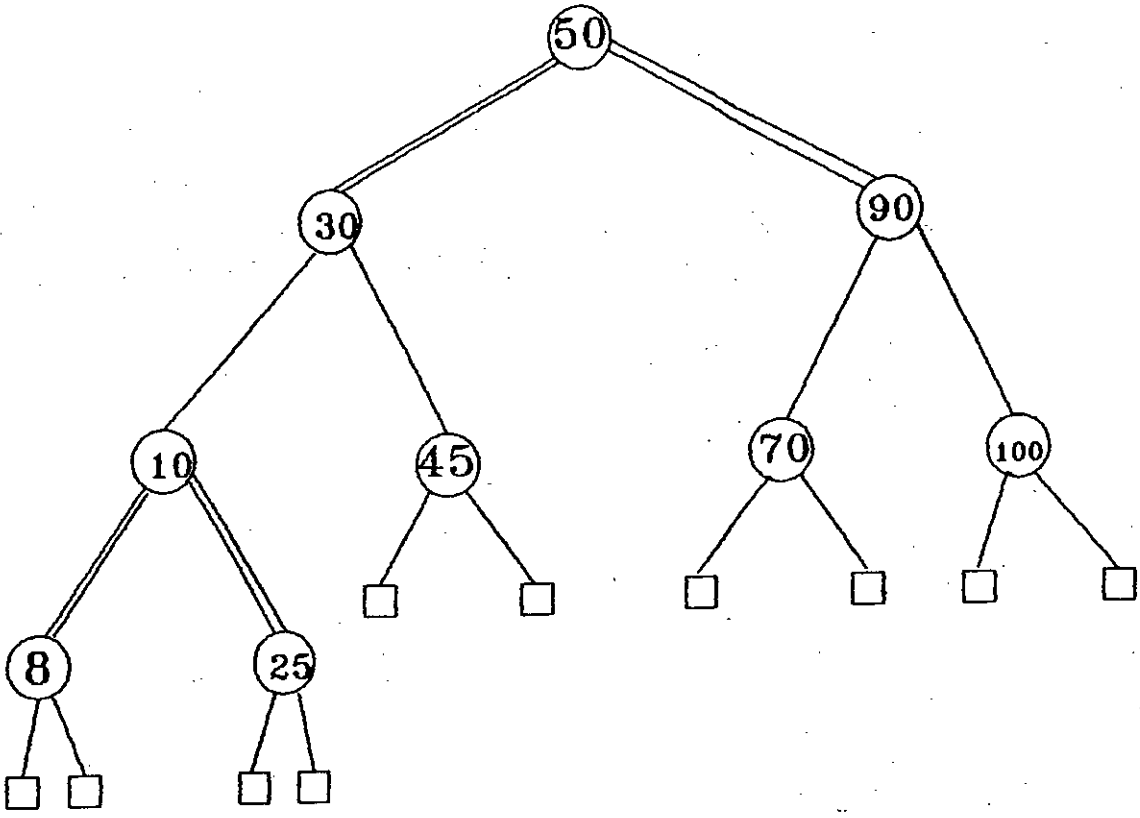


Fig. 3.19 continued

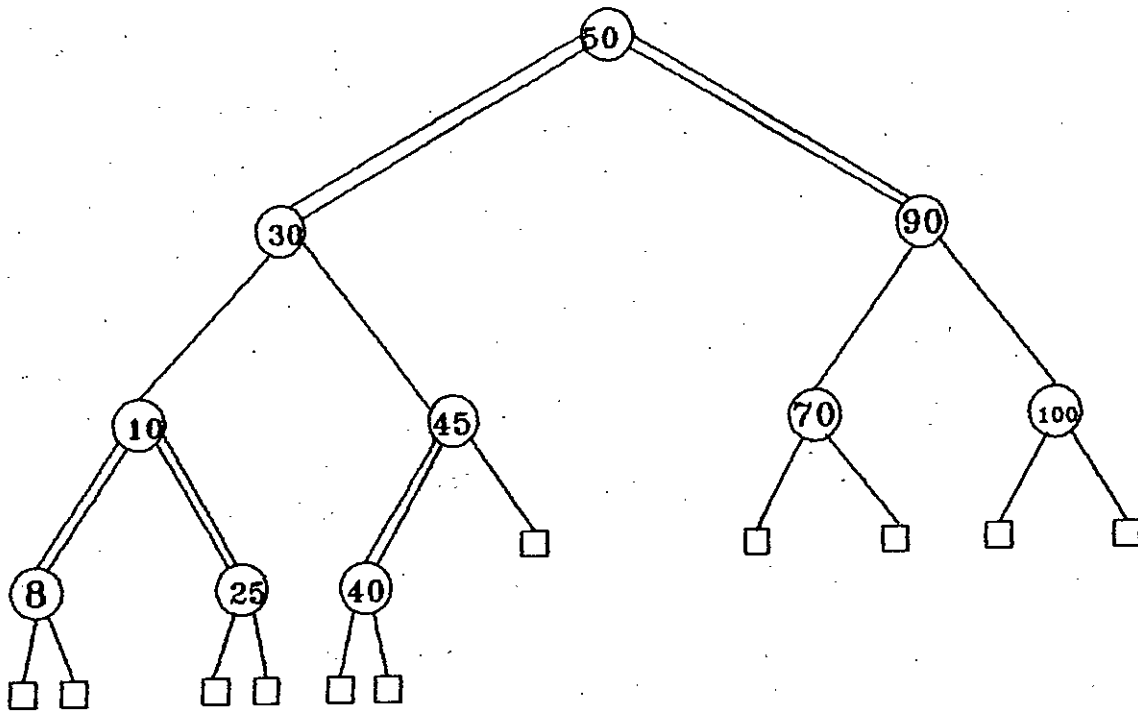


Fig. 3.20 The resulting tree of Fig. 3.19 after 40 is inserted

```

if (p ↑ red ) then
  g ↑ red = true
  if (v < g ↑ key ) ≠ (v < p ↑ key) then
    p = rotate(v, g)
  endif
  x = rotate(v, gg)
  x ↑ red = false
endif

return (x)

```

End Procedure split

This procedure fixes the color after a rotation and also restarts x high enough in the tree to ensure that the search does not get lost due to all the link changes. The long argument list is included for clarity ; this procedure should more properly be declared local to rbtreeinsert, with access to its variables.

If the root is a 4-node then the split procedure makes the root red : this corresponds to transforming if, along with the dummy node above it, into a 3-node. Of course, there is no reason to do this, so a statement is included at the end of split to keep the root black.

Assembling the code fragments above gives a very efficient, relatively simple algorithm for insertion using a binary tree structure that is guaranteed to take a logarithmic number of steps for all searches and insertions. This is one of the few searching algorithms with that property, and its use is justified whenever bad worst case performance simply cannot be tolerated.

CHAPTER 4

HASHING TECHNIQUES

4.1 Introduction.

In this chapter we will examine a special class of table organization in which we attempt to store elements in locations that are easily computed from the value or representation of the elements. This contrasts markedly with the techniques presented in previous chapters; in those chapters we based a search on comparisons and the locations in which an element was stored depended on its position in an ordered arrangement of the elements. In this chapter we discuss techniques based on directly transforming the elements into an address at which it will be stored.

More generally, we will suppose that, we have an array of m table locations $T[0], T[1], \dots, T[m-1]$, say, and given an element z to be inserted we transform it to a location $h(z)$, $0 \leq h(z) < m$; h is called the hash function. We then examine $T[h(z)]$ to see if it is empty. Most of the time it will be, so we set $T[h(z)] \leftarrow z$, and we are done. If $T[h(z)]$ is not empty, a collision has occurred, and we must resolve it somehow. Taken together, the hash function and the collision resolution method are referred to as hashing or scatter storage schemes.

Under the proper conditions, hashing is unsurpassed in its efficiency as a table organization, since the average time for a search or an insertion is generally constant, independent of the size of the table. However, some important caveats are in order. First, hashing requires a strong belief in the law of averages, since in the worst case collision occurs every time, and hashing degenerates into linear search. Second, while it

is easy to make insertions into a hash table, the full size of the table must be specified a priori, because it is closely connected to the hash function used; this makes it extremely expensive to change dynamically. If we choose too small a size the performance will be poor and the table may overflow, but if we choose too large a size much memory will be wasted. Third deletions from the table are not easily accommodated. Finally, the order of the elements in the table is unrelated to any natural order that may exist among the elements, and so an unsuccessful search results only in the knowledge that the element sought is not in the table, with no information about how it relates to the elements in the table.

4.2 Collision Resolution.

Typically, the number of possible elements is so enormous compared to the relatively small number of table locations that no hash function, not even the most carefully designed one, can prevent collisions from occurring in practice. In fact, the likelihood of collisions under even the most ideal circumstances suggests that the collision resolution scheme is more critical to overall performance than the hash functions, provided at least minimum care is taken to avoid primary clustering.

When a collision occurs, and the location $T[h(z)]$ is already filled at the time we try to insert z , we must have some method for specifying another location in the table where z can be placed. A collision resolution scheme is a method of specifying a list of table location $\alpha_0 = h(z), \alpha_1, \alpha_2, \dots, \alpha_{m-1}$ for an element z . To insert z , the locations are inspected in that order until an empty one is found. Our two choices are to store pointers describing the sequence explicitly or to specify the sequence implicitly by a fixed relationship between z , α_i , and i . Techniques for collision resolution based on these two possibilities are explored in the following sections.

4.3 Chaining.

In this scheme a sequence of pointers is built going from the hash location $h(z)$ to the location in which z is ultimately stored. In separate chaining each table location $T[i]$ is a list header, pointing to a linked list of those elements z with $h(z) = i$. A link field should be included in each record, and there will also be M list heads. If the list is unordered we insert z just after the list header $T[h(z)]$, before the first element on that list. A search for Z in this case is done by applying algorithm 4.1 to the list $T[h(z)]$.

Algorithm 4.1 Unordered list search.

```

     $q \leftarrow NULL; \quad p \leftarrow T[h(z)]$ 

    while  $(p \neq NULL \text{ and } z \neq KEY(p))$  do

         $q \leftarrow p; \quad p \leftarrow LINK(p)$ 

        repeat

            if  $(p \neq NULL)$  then return [[ Found:  $p$  points to the element ]]

            else

                 $s = \text{getnode}(); \quad s(KEY) = z;$ 

                 $LINK(s) = NULL; \quad$  [[ Not found:  $z$  is not in the list ]]

            if  $(q = NULL)$  then  $T[h(z)] = s$ 

            else

                 $LINK(q) = s$ 

            endif

        endif

    endif
```

End Algorithm 4.1

The time for unsuccessful searches can be reduced by keeping each of the lists ordered by key. Then only half of the list need to be traversed on the average to determine that an item is missing. If the list is ordered insertion of Z is done by applying algorithm 4.2 to the list $T[h(z)]$ and a search for Z is done by applying algorithm 4.3 to the list $T[h(z)]$.

Algorithm 4.2 Ordered list insertion

$pred \leftarrow T[h(z)]; \quad p \leftarrow pred$

while $((p \neq NULL) \text{ and } (KEY(p) < z))$ do

$pred \leftarrow p$

$p \leftarrow LINK(p)$

repeat

$KEY(new) \leftarrow z$

$LINK(new) \leftarrow LINK(pred)$

$LINK(pred) \leftarrow new$

End Algorithm 4.2

Algorithm 4.3 Ordered list search

$p \leftarrow T[h(z)]$

while $(z > KEY(p))$ do $p \leftarrow LINK(p)$ repeat.

```

if ( $z = KEY(p)$ ) return [[ Found:  $p$  points to the element ]]
else call Algorithm 4.2 [[ Not found:  $z$  is not in the list ]]
endif

```

End Algorithm 4.3

Chaining is quite fast, because the lists are short. If 365 people are gathered together in one room, there will probably be many pairs having the same birth day, but the average number of people with any given birthday will be only one. In general if there are N keys and M lists the average list size is N/M ; thus hashing decreases the amount of work needed for sequential searching by roughly a factor of M . One technique that can be used to reduce the number of probes in separate chaining is to maintain the records that hash into the same value as a binary search tree emanating from the hash bucket rather than as a linked list. However, this requires two pointers to be kept with each record. Since chains are usually small, the added space and programming complexity do not seem to be warranted.

For the sake of speed we would like to make M rather large. But when M is large, many of the lists will be empty and much of the space for the M list heads will be wasted. This suggests another approach, when the records are small: we can overlap the record storage with the list heads, making room for a total of M records and M links instead of for N records and $M + N$ links. In this case, the overhead of the M list headers needed by separate chaining can be eliminated by using coalesced chaining, in which each table location $T[i]$ is used to store a record, containing within it a field $LINK[i]$. When $T[h(z)]$ is found to contain another element on an attempted insertion of Z , we follow the $LINK$ fields until we reach one that is $NULL$; then we take an empty location $T[free]$, set the last $NULL$ $LINK$ field to point to it and store Z in $T[free]$.

The search for empty locations originally begins at $T[m-1]$ and goes backward in the table toward $T[0]$. Each time an empty location is needed, we continue backward from where we stopped on the previous occasion; to stop this search, we introduce a dummy element that is always empty. The table will overflow when all the locations are full. The details of such a search and insertion process for coalesced chaining is as follows :

Algorithm 4.4 (chained scatter table search and insertion)

This algorithm searches for an M -node table, looking for a given key K . If K is not in the table, and the table is not full, K is inserted. The nodes of the table are denoted by $T[i]$, for $0 \leq i \leq M$, and they are of two distinguishable types, empty and occupied. An occupied node contains a key field $KEY[i]$, a link field $LINK[i]$ and possibly other fields.

The algorithm makes use of a hash function $h(k)$. An auxiliary variable R is also used, to help find empty spaces; when the table is empty, we have $R = M + 1$ and as insertions are made it will always be true that $T[i]$ is occupied for j in the range $R \leq j \leq M$. By convention, $T[0]$ will always be empty.

$i \leftarrow h(k) + 1; \quad \text{Now } 1 \leq i \leq M$

if ($T[i] = \text{empty}$) then

 [[Insert new key]]

 Mark $T[i]$ occupied; $KEY[i] = K$

$LINK[i] \leftarrow 0$

return

endif

```

while (LINK[i] ≠ 0) do
if (K = KEY[i]) return; endif [[ The algorithm terminates successfully ]]

i ← LINK[i]

repeat

[[ Find empty nodes ]]

Decrease R one or more times until T[R] = empty

if (R = 0) turn on overflow flag and return

else

LINK[i] ← R; i ← R [[ Insert new key ]]

Mark T[i] occupied

KEY[i] ← K

LINK[i] ← 0

endif

```

End Algorithm 4.4

When collisions are resolved by separate chaining with unordered lists, the average number of probes in a successful search in a table of M locations containing N elements can be shown to be

$$S(\alpha) = 1 + \frac{n-1}{2m} \approx 1 + \frac{\alpha}{2}$$

and in an unsuccessful search

$$U(\alpha) = \left(1 - \frac{1}{M}\right)^N + \frac{N}{M} \approx e^{-\alpha} + \alpha$$

When $\alpha = \frac{N}{M}$ is called the load factor. It is customary to express the behaviour of the collision resolution in terms of α rather than N and M because the behaviour of the algorithms is typically governed more by the fullness of the table in relative, rather than absolute terms. If the lists are kept in order, then the average number of probes for an unsuccessful search is decreased to

$$U(\alpha) = 1 + \frac{1}{2} \frac{N}{M} - \frac{M}{N+1} \left[1 - \left(1 - \frac{1}{M}\right)^{N+1} \right] + \left(1 - \frac{1}{M}\right)^N$$

$$\approx 1 + \frac{1}{2} \alpha - \frac{1}{\alpha} (1 - e^{-\alpha}) + e^{-\alpha}$$

Let us investigate the average number of probes in a chained scatter table when the lists are kept separate as in algorithm 4.1. The probabilistic model we shall use for this purpose assumes that each of the M^N possible "hash sequence" $a_1 a_2 \dots a_N$, $0 \leq a_j < M$, is equally likely, where a_j denotes the initial hash address of the j th key inserted into the table. The average number of probes in a successful search is assumed to be the average number of probes needed to find the k th key, averaged over $1 \leq k \leq N$ with each key equally likely and averaged over all hash sequences with each sequence equally likely. Similarly, the average number of probes needed when the N th key is inserted, considering all sequences to be equally likely is the average number of probes in an unsuccessful search starting with $N-1$ elements in the table. Let P_{NK} be the probability that a given list has length K . There are $\binom{N}{K}$ ways to choose a set of K elements having a particular value, and $(M-1)^{N-K}$ ways to assign values to other a 's. Therefore,

$$P_{NK} = \binom{N}{K} (M-1)^{N-K} / M^N$$

The generating function for this probability distribution is

$$P_N(z) = \sum_{k \geq 0} P_{Nk} z^k = \left(1 + \frac{z-1}{M}\right)^N$$

An unsuccessful search in a list of length K requires $K + \delta_{k0}$ probes. Therefore average number of probes required in unsuccessful search is

$$C'_N = \sum (K + \delta_{k0}) P_{NK} = P'_N(1) + P_N(0)$$

But from the above written generating function $P'_N(1) = \frac{N}{M}$ and $P_N(0) = (1 - \frac{1}{M})^N$.

Inserting these values we have

$$U(\alpha) = \frac{N}{M} + (1 - \frac{1}{M})^N \approx \alpha + e^{-\alpha}$$

For successful search let us consider the total number of probes to find all the keys. A list of length K contributes $\binom{K+1}{2}$ to the total; hence

$$\begin{aligned} S(\alpha) &= M \sum \binom{K+1}{2} P_{NK} / N \\ &= \left(\frac{M}{N}\right) \left(\frac{1}{2} P''_N(1) + P'_N(1)\right) \\ &= \left(\frac{M}{N}\right) \left(\frac{1}{2} \frac{N(N-1)}{M^2} + \frac{N}{M}\right) \\ &= 1 + \frac{N-1}{2M} \approx 1 + \frac{\alpha}{2} \end{aligned}$$

On the other hand, when the lists are kept ordered average number of probes required in an unsuccessful search becomes

$$\begin{aligned} &\sum \left(1 + \frac{K}{2} - (K+1)^{-1} + \delta_{k0}\right) P_{NK} \\ &= 1 + \frac{1}{2} \sum K P_{NK} - \sum \frac{P_{NK}}{K+1} + \sum \delta_{k0} P_{NK} \\ &= 1 + \frac{1}{2} P'_N(1) + P_N(0) - \sum \frac{P_{NK}}{K+1} \\ &= 1 + \frac{N}{2M} + (1 - \frac{1}{M})^N - \frac{M}{N+1} \left(1 - (1 - \frac{1}{M})^{N+1}\right) \\ &\approx 1 + \frac{\alpha}{2} - (1 - e^{-\alpha}) / \alpha + e^{-\alpha} \end{aligned}$$

But since the search keys are purely random, ordered lists do not affect the behaviour in case of successful searches, that is, it remains at the same level as that for unordered lists.

In coalesced chaining the average number of probes required in an unsuccessful search is

$$U(\alpha) = 1 + \frac{1}{4} \left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M} \right) \\ \approx 1 + \frac{1}{4} (e^{2\alpha} - 1 - 2\alpha)$$

Thus when the table is half full, the average number of probes made in an unsuccessful search is about $\frac{1}{4}(e + 2) \approx 1.18$ and even when the table gets full, the average number of probes made just before inserting the final item will be only about $\frac{1}{4}(e^2 + 1) \approx 2.10$. These statistics prove that the lists stay short even though the algorithm occasionally allows them to coalesce, when the hash function is random.

The average number of probes in a successful search may be computed by summing the quantity $C + A$ over the first N unsuccessful searches and dividing by N , if we assume that each key is equally likely, where

C = number of table entries probed while searching

$A = 1$ if the initial probe found an occupied node.

In a successful search we always have $A = 1$. Thus we obtain

$$s(\alpha) = \frac{1}{N} \sum_{0 \leq k < N} \left(U_k + \frac{K}{M} \right) \\ = 1 + \frac{1}{8} \frac{M}{N} \left(\left(1 + \frac{2}{M}\right)^N - 1 - \frac{2N}{M} \right) + \frac{1}{4} \frac{N-1}{M} \\ \approx 1 + \frac{1}{8\alpha} (e^{2\alpha} - 1 - 2\alpha) + \frac{1}{4} \alpha$$

as the average number of probes in a random successful search. Even a full table will require only about 1.80 probes, on the average, to find an item. At the first glance it

may appear that number of table entries probed while looking for an empty space is inefficient, since it has to search sequentially for an empty position. But actually the total number of table probes for an empty position as a table is being built will never exceed the number of items in the table; so we make an average of at most one of these probes per insertion. This figure is actually αe^α in a random unsuccessful search.

Another major advantage of coalesced chaining method is that they permit efficient deletion without penalizing the efficiency of subsequent retrievals. An item being deleted can be removed from its list, its position in the table is freed and free location pointer is reset to the following position. This may slow down the second subsequent insertion somewhat by forcing the search for free location through a long series of occupied positions, but that is not very significant. If the free table positions are kept in a linked list, this penalty also disappears.

A generalization of the standard coalesced hashing method, which we call general coalesced hashing, adds extra position to the hash table that can be used for list nodes in the case of collisions, but not for initial hash locations. Thus the table would consist of t entries (numbered 0 to $t-1$), but keys would hash only one of $m < t$ values (0 to $m-1$). The extra $t-m$ positions are called the cellar, and are available for storing items whose hash positions are full.

Using a cellar results in less conflict between list of items with different hash values, and therefore, reduces the length of the lists. However, a cellar that is too large could increase list lengths relative to what they might be if the cellar positions were permitted as hash locations. Higher values of m/t produce lower successful and unsuccessful search times for lower load factors. Value of t/m between 15 to 20% is optimum both for successful and unsuccessful searches.

If the number of records grows beyond the number of table positions, it is impossible to insert them without allocating a larger table and recomputing the hash values of keys of all records already in the table using a new hash function.

The above statements are true for most of the hashing schemes. But in general coalesced hashing, the old table can be copied into the first half of the new table and the remaining position of the new table used to enlarge the cellar so that items do not have to be rehashed.

A brief examination of the above formulas indicates that separate chaining is superior to coalesced chaining which in turn is superior to other methods of collision resolutions as we will see soon in our next discussion. In separate chaining we also have the advantage of ignoring the table of overflowing its allocated storage. This means that we can even have $N > M$, giving $\alpha > 1$; the formulas for separate chaining are also valid in this case. A further advantage of separate chaining is that it allows very easy deletion of elements, something difficult or impossible with other collision resolution schemes. The disadvantage of chaining compared to other schemes is that it requires additional storage overhead for the link fields; this makes the other schemes more desirable in some circumstances.

4.4 Linear probing.

The simplest alternative to chaining that does not require the storage of LINK fields is to resolve collisions by probing sequentially, one location at a time, starting from the hash address, until an empty location is found. This is called open addressing with linear probing or simply linear probing.

The idea is to formulate some rule by which every key k determines a "probe sequence", namely a sequence of table positions which are to be inspected whenever k

is inserted or looked up. If we encounter an open position while searching for k , using the probe sequence determined by k , we can conclude that k is not in the table, since the same sequence of probes will be made every time k is processed. The simplest open addressing scheme, known as linear probing, uses the cyclic probe sequence $h(k), h(k) - 1, \dots, 0, M - 1, M - 2, \dots, h(k) + 1$ as in the following algorithm.

Algorithm 4.6 (Open scatter table search and insertion) :

This algorithm searches for an M -node table, looking for a given key k . If k is not in the table and the table is not full, k is inserted.

The nodes of the table are denoted by $TABLE[i]$, for $0 \leq i < M$, and they are of two distinguishable types, empty and occupied. An occupied node contains a key, called $KEY[i]$ and possibly other fields. An auxiliary variable N is used to keep track of how many nodes are occupied; this variable is considered to be part of the table and it is increased by 1 whenever a new key is inserted.

This algorithm makes use of a hash function $h(k)$, and it uses the probing sequence $h(k), h(k) - 1, \dots, 0, M - 1, M - 2, \dots, h(k) + 1$ to address the table.

```

i ← h(k) [[ Now 0 ≤ i < M ]]

while (KEY[i] ≠ k and TABLE[i] is nonempty )do
    i ← i - 1

if (i < 0) then i ← i + M endif

```

```

repeat
    if ( $key[i] = k$ ) then return; endif [[ The algorithm terminates successfully ]]
    if ( $N = M - 1$ ) then return; [[ The algorithm terminates with overflow ]]
    else
         $N = N + 1$ ; Mark  $TABLE[i]$  occupied
         $KEY[i] \leftarrow k$ 
    endif

```

End Algorithm 4.6

Experience with linear probing shows that the algorithm works fine until the table begins to get full; but eventually the process slows down, with long drawn-out searches becoming increasingly frequent. The reason for this behaviour can be understood by considering the hypothetical scatter table in figure 4.1 with $M = 19$, $N = 9$. Shaded squares represent occupied positions. The next key k to be inserted in the table will go into one of the ten empty spaces, but these are not equally likely; in fact, k will be inserted into positions 11 if $11 \leq h(k) \leq 15$ while it will fall into position 8 only if $h(k) = 8$. Therefore position 11 is five times as likely as position 8; long lists tend to grow even longer.

This phenomenon is not enough by itself to account for the relatively poor behaviour of linear probing, since a similar thing occurs in coalesced chaining. The real problem occurs when a cell like 4 or 16 is filled in the given figure; then two separate lists are combined, while the lists in coalesced chaining never grow by more than one step at a time. Consequently the performance of linear probing degrades rapidly when N approaches M .

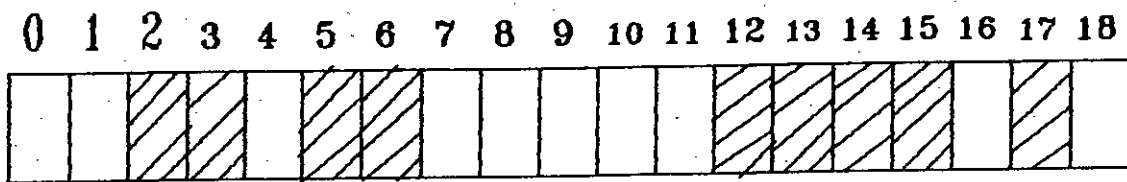


Fig. 4.1 Pile up phenomenon in Linear open addressing

Now we shall prove that the average number of probes needed by linear probing is approximately

$$C'_N \approx \frac{1}{2} \left(1 + \left(\frac{1}{1-\alpha} \right)^2 \right) \text{ (Unsuccessful search)}$$

$$C_N \approx \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right) \text{ (Successful search)}$$

Where $\alpha = N/M$ is the load factor of the table.

The probabilistic model we shall use for this purpose assumes that each of the M^N possible "hash sequence"s $a_1 a_2 \dots a_N$ $0 \leq a_j < M$, is equally likely, where a_j denotes the initial hash address of the j th key inserted into the table. The average number of probes in a successful search is assumed to be the average number of probes needed to find the k th key, averaged over $1 \leq k \leq N$ with each key equally likely and averaged over all hash sequences with each sequence equally likely. Similarly, the average number of probes needed when the N th key is inserted, considering all sequences to be equally likely is the average number of probes in an unsuccessful search starting with $N-1$ elements in the table. When open addressing is used

$$C_N = \frac{1}{N} \sum_{0 \leq k < N} C'_k \quad (4.1)$$

so that we can deduce one quantity from the other.

Let $f(M, N)$ be the number of hash sequences such that position 0 of the table will be empty after the keys have been inserted by algorithm 4.6. The circular symmetry of linear probing implies that position 0 is empty just as often as any other position, so it is empty with probability $1 - N/M$; in other words

$$f(M, N) = \left(1 - \frac{N}{M} \right) M^N \quad (4.2)$$

Now let $g(M, N, K)$ be the number of hash sequences such that the algorithm leaves position 0 empty, position 1 through K occupied and position $K + 1$ empty. We have

$$g(M, N, K) = \binom{N}{K} f(K + 1, K) f(M - K - 1, N - K) \quad (4.3)$$

because all such hash sequences are composed of two subsequences, one (containing K elements $a_i \leq K$) that leaves position 0 empty and position 1 through K occupied and one (containing $N - K$ elements $a_i \geq K + 1$) that leaves position $K + 1$ empty; there are $f(K + 1, K)$ subsequences of the former type and $f(M - K - 1, N - K)$ of the latter and there are $\binom{N}{K}$ ways to interperse two such subsequences. Finally let P_k be the probability that exactly $K + 1$ probes will be needed when the $(N + 1)$ st key is inserted; it follows that

$$P_k = M^{-N} (g(M, N, K) + g(M, N, K + 1) + \dots + g(M, N, N)) \quad (4.4)$$

Let

$$\begin{aligned} S(n, x, y) &= \sum_{k \geq 0} \binom{n}{k} (x + k)^{k+1} (y - k)^{n-k-1} (y - n) \\ &= \sum_{k \geq 0} \binom{n}{k} x (x + k)^k (y - k)^{n-k-1} (y - n) \\ &\quad + n \sum_{k \geq 0} \binom{n-1}{k-1} (x + k)^k (y - k)^{n-k-1} (y - n) \end{aligned} \quad (4.5)$$

Replacing k by $n - k$ in the first sum of equation (4.5) and finally applying Abel's formula

$$(x + y)^r = \sum_k \binom{r}{k} x (x - kz)^{k-1} (y + kz)^{r-k}$$

in the first sum gives

$$S(n, x, y) = x(x + y)^n + ns(n - 1, x + 1, y - 1) \quad (4.6)$$

Now from equation (4.3), putting the value of $f(M,N)$ from equation (4.2) we get

$$\begin{aligned}
 g(M, N, K) &= \binom{N}{K} f(K+1, K) f(M-K-1, N-K) \\
 &= \binom{N}{K} \left(1 - \frac{K}{K+1}\right) (K+1)^K \left(1 - \frac{N-K}{M-K-1}\right) (M-K-1)^{N-K} \\
 &= \binom{N}{K} (K+1)^{K-1} (M-K-1)^{N-K-1} (M-N-1) \quad (4.7)
 \end{aligned}$$

Now

$$C'_N = \sum_{0 \leq K \leq N} (K+1)P_k \quad (4.8)$$

Let us calculate the value of $M^N \sum (K+1)P_k$. This can be evaluated as follows:

$$\begin{aligned}
 &M^N \sum (K+1)P_k \\
 &= M^N \sum (K+1)M^{-N} (g(M, N, K) + g(M, N, K+1) + \dots + g(M, N, N)) \\
 &= \sum_{K \geq 0} \binom{K+2}{2} g(M, N, K) \\
 &= \frac{1}{2} \left(\sum_{K \geq 0} (K+1)g(M, N, K) + \sum_{K \geq 0} (K+1)^2 g(M, N, K) \right) \\
 &= \frac{1}{2} \left(M^N \sum P_k + \sum_{K \geq 0} \binom{N}{K} (K+1)^{K+1} (M-K-1)^{N-K-1} (M-N-1) \right) \\
 &= \frac{1}{2} (M^N + s(N, 1, M-1)) \\
 &= \frac{1}{2} (M^N + M^N + N(2M^{N-1} + (N-1)(3M^{N-3} + \dots))) \\
 &= \frac{1}{2} (M^N + M^N Q_1(M, N)) \quad (4.9)
 \end{aligned}$$

where

$$\begin{aligned}
 Q_r(M, N) &= \binom{r}{0} + \binom{r+1}{1} \frac{N}{M} + \binom{r+2}{2} \frac{N(N-1)}{M^2} + \dots \\
 &= \sum_{K \geq 0} \binom{r+k}{k} \frac{N}{M} \frac{N-1}{M} \dots \frac{N-K+1}{M} \quad (4.10)
 \end{aligned}$$

Therefore from (4.8) the average number of probes needed for unsuccessful search is

$$C'_N = \frac{1}{2} (1 + Q_1(M, N)) \quad (4.11)$$

Now from (4.10) it can be shown that

$$Q_1(M, N) = (N + 1)Q_0(M, N) - NQ_0(M, N - 1) \quad (4.12)$$

Average number of probes required for successful search thus follows from equation (4.1)

$$\begin{aligned} C_N &= \frac{1}{N} \sum_{0 \leq K < N} C'_K \\ &= \frac{1}{N} \sum_{0 \leq K < N} \frac{1}{2} (1 + Q_1(M, K)) \\ &= \frac{1}{2N} \sum_{0 \leq K < N} 1 + Q_1(M, K) \\ &= \frac{1}{2N} \sum_{0 \leq K < N} 1 + (K + 1)Q_0(M, K) - KQ_0(M, K - 1) \\ &= \frac{1}{2N} (N + Q_0(M, 0) - 0 + 2Q_0(M, 1) - Q_0(M, 0) + 3Q_0(M, 2) \\ &\quad - 2Q_0(M, 1) + \dots + NQ_0(M, N - 1) - (N - 1)Q_0(M, N - 2)) \\ &= \frac{1}{2} (1 + Q_0(M, N - 1)) \end{aligned} \quad (4.13)$$

The rather strange looking function $Q_r(M, N)$ is not really hard to deal with. We have

$$N^K - \binom{K}{2} N^{K-1} \leq N(N - 1) \dots (N - K + 1) \leq N^K$$

Hence if $\frac{N}{M} = \alpha$,

$$\begin{aligned} \sum_{K \geq 0} \binom{r+k}{k} \left(N^k - \binom{k}{2} N^{k-1} \right) / M^k &\leq Q_r(M, N) \\ &\leq \sum_{k \geq 0} \binom{r+k}{k} \frac{N^k}{M^k} \end{aligned}$$

$$\sum_{k \geq 0} \binom{r+k}{k} \alpha^k - \frac{\alpha}{M} \sum_{k \geq 0} k \geq 0 \binom{r+k}{k} \binom{k}{2} \alpha^{k-2} \leq Q_r(M, \alpha M)$$

$$\leq \sum_{k \geq 0} \binom{r+k}{k} \alpha^k$$

i.e.

$$\frac{1}{(1-\alpha)^{r+1}} - \frac{1}{M} \binom{r+2}{2} \frac{\alpha}{(1-\alpha)^{r+3}} \leq Q_r(M, \alpha M)$$

$$\leq \frac{1}{(1-\alpha)^{r+1}} \quad (4.14)$$

This relation gives us a good approximation of $Q_r(M, N)$ when M is large and α is not too close to unity and the lower bound is a better approximation than the upper bound.

Thus

$$Q_1(M, N-1) \approx \frac{1}{1-\alpha} - \frac{1}{M} \frac{\alpha}{(1-\alpha)^3}$$

or

$$Q_1(M, N-1) \approx \frac{1}{(1-\alpha)^2}$$

If we take the second approximation then obviously

$$C'_N \approx \frac{1}{2} \left(1 + \left(\frac{1}{1-\alpha} \right)^2 \right)$$

as said earlier.

Similarly

$$C_N \approx \frac{1}{2} \left(1 + \frac{1}{1-\alpha} \right)$$

So algorithm 4.6 is almost as fast as algorithm 4.4, when the table is 75 percent full. On the other hand when α approaches unity, its performance degrades very rapidly. When $N = M - 1$ we have

$$Q_1(M, M-1) = M - (M - M + 1 - 1) Q_0(M, M-1) = M$$

using the relation $Q_1(M, N) = M - (M - N - 1)Q_0(M, N)$. Therefore when $N = M - 1$ i.e, the table is full then

$$C'_N = \frac{1}{2}(1 + Q_1(M, M - 1)) = \frac{1}{2}(1 + M) \quad (4.15)$$

Now

$$\begin{aligned} Q_0(M, M - 1) &= 1 + \frac{M(M - 1)}{M^2} + \frac{M(M - 1)(M - 2)}{M^3} + \dots \\ &= Q(M) \end{aligned}$$

The asymptotic value of this series is

$$Q(M) = \sqrt{\frac{\pi M}{2}} - \frac{1}{3} + \frac{1}{12} \sqrt{\frac{\pi}{2M}} - \frac{4}{135M} + \frac{1}{288} \sqrt{\frac{\pi}{2M^3}} + O(M^{-2})$$

Using this value we get that the average number of probes required for successful search when the table is completely full is

$$C_N \approx \sqrt{\frac{\pi M}{8}} \quad (4.16)$$

The pile up phenomenon which makes linear probing costly on a nearly full table is aggravated by the use of division hashing if consecutive key values are likely to occur, since these keys will have consecutive hash codes. Multiplicative hashing will break up these clusters satisfactorily. Another way to protect against consecutive hash code is to set $i \leftarrow i - c$ instead of $i \leftarrow i - 1$ in algorithm 4.6.

Any positive value of c will do, so long as it is relatively prime to M , since the probe sequence will still examine every position of the table in this case. It does not alter the pile up phenomenon since groups of c -apart records will still be formed; but the appearance of consecutive keys $K, K+1, K+2, \dots$ will now actually be a help instead of hindrance.

4.5 Double hashing.

Part of the problem with linear probing is the phenomenon of secondary clustering : the tendency of two elements that have collided to follow the same sequence of locations in the resolution of the collision of location. Clearly such a tendency will aggravate the unavoidable fact that long lists are more likely to grow than short lists. This suggests that the sequence of locations followed in resolving a collision of z should be a function of the element z . This can be accomplished very easily by only a minor change to algorithm 4.6 : instead of decrementing i by 1, we decrement it by an amount Δ , $1 \leq \Delta < M$ where Δ is a function of z . In order to ensure that every location in the table will be probed on collisions, we must have Δ and M relatively prime. Since we want Δ to have pseudorandom behaviour, we can use another hash function $\delta(z)$, $1 \leq \delta(z) < M$ as our value for Δ . This means that we will now have to compute two functions instead of one but the resulting improvement in behaviour will be more than compensate for the extra calculation. As a practical matter, it is easiest to guarantee that $\delta(z)$ and M are relatively prime for all z by insisting that M be a prime number.

Algorithm 4.7 (Open addressing with double hashing)

This algorithm uses two hash functions $h_1(k)$ and $h_2(k)$. As usual $h_1(k)$ produces a value between 0 and $M-1$, inclusive, but $h_2(k)$ must produce a value between 1 and $M-1$ that is relatively prime to M .

$i \leftarrow h_1(k)$ [[Now $0 \leq i < M$]]

while ($KEY[i] \neq k$ and $TABLE[i]$ is nonempty) do

```

c ← h2(k)    [[ Second Hash ]]

i ← i - c

if (i < 0) then i ← i + M endif

repeat

if (key[i] = k) then return; endif [[ Terminates successfully ]]

if (N = M - 1) then return: [[ Terminates with overflow ]]

else

    N = N + 1; Mark TABLE[i] occupied

    KEY[i] ← k

endif

```

End Algorithm 4.7

Several possibilities have been suggested for computing $h_2(k)$. If M is prime and $h_1(k) = K \bmod M$, we might let $h_2(k) = 1 + (K \bmod (M-1))$; but since $M-1$ is even, it would be better to let $h_2(k) = 1 + (K \bmod (M-2))$. This suggests choosing M so that M and $M-2$ are twin primes like 1021 and 1019. If $M = 2^m$ and we are using multiplicative hashing, $h_2(k)$ can be computed simply by shifting ~~left~~ m more bits and ORing in a 1, and this is obviously faster than the division method.

In each of the techniques suggested above, $h_1(k)$ and $h_2(k)$ are independent in the sense that different keys will have same value for both h_1 and h_2 with probability $O(1/M^2)$ instead of $O(1/M)$. Empirical tests show that the behaviour of algorithm 4.7 with independent hash functions is essentially indistinguishable from number of probes

which would be required if the keys were inserted at random into the table; there is practically no piling up or clustering as in algorithm 4.6.

A complete analysis of the average behaviour of double hashing has not yet been made, but both empirical results and some fragmentary theoretical results indicate that it behaves approximately like uniform hashing, an idealization of double hashing that we can analyze. In our model we assume that the keys go into random locations of the table, so that each of the $\binom{M}{N}$ possible configurations of N occupied cells and $M-N$ empty cells is equally likely. This model ignores any effect of primary or secondary clustering: the occupancy of each cell in the table is essentially independent of the others. For this model the probability that exactly r probes are needed to insert the $(N+1)$ st item is the number of configurations in which $r-1$ given cells are occupied, and another one is empty, divided by $\binom{M}{N}$, namely

$$P_r = \binom{M-r}{N-r+1} / \binom{M}{N}$$

therefore the average number of probes for uniform hashing is

$$\begin{aligned} C'_N &= \sum_{1 \leq r \leq M} r P_r = M+1 - \sum_{1 \leq r \leq M} (M+1-r) P_r \\ &= M+1 - \sum_{1 \leq r \leq M} (M+1-r) \binom{M-r}{M-N-1} / \binom{M}{N} \\ &= M+1 - \sum_{1 \leq r \leq M} (M-N) \binom{M+1-r}{M-N} / \binom{M}{N} \\ &= M+1 - (M-N) \binom{M+1}{M-N+1} / \binom{M}{N} \\ &= M+1 - (M-N) \frac{M+1}{M-N+1} = \frac{M+1}{M-N+1} \text{ for } 1 \leq N < M. \quad (4.17) \end{aligned}$$

Setting $\alpha = \frac{N}{M}$, this exact formula for C'_N is approximately equal to $\frac{1}{1-\alpha}$. The corre-

sponding average number of probes for a successful search is

$$\begin{aligned}
 C_N &= \frac{1}{N} \sum_{0 \leq K < N} C_K \\
 &= \frac{M+1}{N} \left(\frac{1}{M+1} + \frac{1}{M} + \dots + \frac{1}{M-N+2} \right) \\
 &= \frac{M+1}{N} (H_{M+1} - H_{M-N+1}) \\
 &\approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}
 \end{aligned} \tag{4.18}$$

As remarked above extensive tests show that algorithm 4.7 with two independent hash functions behaves essentially like uniform hashing for all practical purposes.

4.6 Brent's Algorithm.

This algorithm modifies 4.7 so that the average successful search time remains bounded as the table gets full. This method is based on the fact that successful searches are much more common than insertions, in many applications; therefore it is logical to do more work when inserting an item, moving records in order to reduce the expected retrieval time. Standard compilers use its symbol table algorithm a large number of times when compiling a program. But on the average make an insertion into the table per 10 to 15 successful searches. Sometimes a table is actually created only once and it is used thereafter purely for retrieval. The idea of Brent's algorithm is to change the insertion process in algorithm 4.7 as follows:

Suppose an unsuccessful search has probed locations $p_0, p_1, p_2, \dots, p_{t-1}, p_t$ where $p_j = (h_1(k) - j h_2(k)) \bmod M$ and $TABLE[p_t]$ is empty. If $t \leq 1$, we insert K in position p_t as usual; but if $t \geq 2$, we compute $C_0 = h_2(K_0)$, where $K_0 = KEY[p_0]$, and see if $TABLE[(p_0 - c_0) \bmod M]$ is empty. If it is, we set it to $TABLE[p_0]$ and then insert K in position p_0 . This increases the retrieval time for K_0 by one step, but it decreases the retrieval time for K by $t \geq 2$ steps, so it results in a net improvement. Similarly

if $TABLE[(p_0 - c_0) \bmod M]$ is occupied and $t \geq 3$, we try $TABLE[(p_0 - 2c_0) \bmod M]$; if that is full too, we compute $C_1 = h_2(KEY[p_1])$ and try $TABLE[(p_1 - c_1) \bmod M]$; etc. In general, let $C_j = h_2(KEY[p_j])$ and $P_{j,k} = (P_j - kC_j) \bmod M$; if we have found $TABLE[P_{j,k}]$ occupied for all indices j, k such that $j + k < r$, and if $t \geq r + 1$, we look at $TABLE[P_{0,r}], TABLE[P_{1,r-1}], \dots, TABLE[P_{r-1,1}]$. If the first empty space occurs at position $P_{j,r-j}$, we set $TABLE[P_{j,r-j}] \leftarrow TABLE[P_j]$ and insert K in position P_j .

Algorithm 4.8 (Brent's variation of double hashing)

$t \leftarrow 0; \quad i \leftarrow h_1(K)$

While($KEY[i] \neq K$ and $TABLE[i]$ is nonempty) do

$c \leftarrow h_2[K]; \quad t \leftarrow t + 1; \quad i \leftarrow i - c$

if ($i < 0$) then $i \leftarrow i + M$ endif

repeat

if ($KEY[i] = K$) then return: [[The algorithm terminates successfully]]

```

if( $N = M - 1$ ) then return: [[ Table overflows.]]
else
 $N = N + 1$ ;  $t = t + 1$ ;  $r = t - 1$ 
for  $j = 0$  to  $r - 1$  do
 $P = (h_1(K) - jh_2(K)) \bmod M$ ;  $c = h_2(\text{KEY}(P))$ 
for  $K = 0$  to  $j + K < r$  do
 $P_1 = P - Kc$ 
if  $\text{TABLE}[P_1]$  is empty then
 $\text{TABLE}[P_1] = \text{TABLE}[P]$ 
 $\text{TABLE}[P] = K$ 
return
repeat
repeat
for  $j = 0$  to  $r - 1$  do
 $P = h_1(K) - jh_2(K) - (r - j) * h_2(\text{KEY}[(h_1(K) - jh_2(K)) \bmod M])$ 
if  $\text{TABLE}[P]$  is empty then
 $\text{TABLE}[P] = \text{TABLE}[(h_1(K) - jh_2(K)) \bmod M] = K$ 
return
endif
repeat
endif

```

End Algorithm 4.8 .

Brent's algorithm reduces the average number of probes per successful search with a maximum value of 2.49; but number of probes in an unsuccessful search is not reduced by Brent's variation, it remains at the same level as uniform hashing. The average number of times h needs to be computed per insertion is $\alpha^2 + \alpha^5 + \frac{1}{3}\alpha^6 + \dots$ according to Brent's analysis, eventually approaching the order of \sqrt{M} , and the number of additional table positions probed while deciding how to make the insertions is about $\alpha^2 + \alpha^4 + \frac{4}{3}\alpha^5 + \alpha^8 + \dots$

An extension of Brent's method that yields even greater improvements in retrieval times at the expense of correspondingly greater insertion times involves recursive insertion of items displaced in the table. That is, in determining the minimum number of rehashes required to displace an item on a rehash path, all the items on that item's subsequent rehash path are considered for displacement as well, and so on. However the recursion cannot be allowed to proceed to its natural conclusion, since insertion times would then become so large as to become impractical, even though insertion is infrequent. A minimum recursion depth must be defined to yield average retrievals very close to optimal with reasonable efficiency. Statistics shows that as the table becomes full, approximately 2.5 probes per retrieval are required on the average, regardless of the table size. This compares very favorably with ordinary double hashing, in which retrieval from a full table requires $O(\log n)$ probes.

4.7 Ordered hash tables.

In many, if not most, cases there is an ordering of the elements that may be useful in speeding up searches in hash tables just as it is for linear lists. We will now investigate how such an ordering can be utilized in conjunction with a hashing scheme. The idea

will be applicable to chaining, linear probing or double hashing, but we will consider it only in the context of double hashing, because chaining is so fast that it needs no improvement, while linear probing is so inefficient we would probably never choose it over double hashing if economy were a factor.

If we had been extremely lucky in algorithm 4.7 and the keys arrived in decreasing order to be inserted then each of the lists built up through collisions would be decreasing order by element. Assuming that an empty location had a value less than that of any element in the table, we could do a search by algorithm 4.9. This algorithm stops a search as soon as it reaches an element less than the search object z .

Of course, we cannot count on the elements being inserted into the table in decreasing order, making algorithm 4.9 useless we can somewhat keep the hash table ordered no matter in what order the elements are inserted.

When an insertion is made, and there is no collision or when the element being inserted is less than the elements it collides with, algorithm 4.7 works fine and the hash table remains ordered. When an insertion leads to a collision with a smaller element, the algorithm must react as though the smaller element were not in the table. In such a collision, then, the idea is to have the larger element being inserted bump the smaller resident element it collided with temporarily out of the table; the larger element takes the location formerly occupied by the smaller. To reinsert the displaced element into the table, we simply apply the insertion algorithm to it; if that leads to a collision with a smaller element, the smaller element is bumped from its location and then reinserted. Each element thus bumped is smaller than the previous one, so the process must end.

Algorithm 4.9 (Ordered double hashing)

$$i \leftarrow h_1(k) \text{ [[Now } 0 \leq i < M \text{]]$$

```

while (TABLE[i] > k)do
    c ← h2(k)    [[ Second Hash ]]
    i ← i - c
    if (i < 0) then i ← i + M endif
    repeat
if (TABLE[i] = k) then return; endif [[ Terminates successfully ]]

if (N = M - 1) then return; [[ Terminates with overflow ]]
else
while TABLE [i] is not empty do
    if (TABLE[i] < k) then TABLE[i] ↔ k; c ← h2(k) endif
    i ← i - c
    if (i < 0) then i ← i + M endif
    repeat
endif

N = N + 1; TABLE[i] ← k

```

End Algorithm 4.9

We can give an approximate analysis of the number of probes needed on the average to search an ordered hash table; as in double hashing this analysis is based on the uniform hashing assumption: the sequence of locations $a_0 = h(z), a_1, a_2, \dots$ used to insert an element z into the table has the property that each a_i is equally likely to be

0, 1, 2, ..., $M - 1$ independently of the other a_i 's. In a table with load factor α the probability of at least k probes in an unsuccessful search is α^{k-1}/k , computed as follows:

α^{k-1} = Probability that first $k - 1$ locations probed will be full.

$\frac{1}{k}$ = Probability that of the $k - 1$ elements thus probed the search object is smaller than all of them.

It is equally likely for the search object to be larger than all of the $k-1$ elements probed, larger than only $k-2$ of them,, larger than only one of them or smaller than all of them. Of these k equally likely possibilities, only in the last case will more than $k-1$ probes be needed. The probability of at least k probes is thus the product $\frac{1}{k}\alpha^{k-1}$

The expected number of probes for an unsuccessful search in an ordered hash table can now be computed as follows:

$$\begin{aligned}
 U(\alpha) &= \sum_{k=1}^{\infty} k \text{ pr (exactly } k \text{ probes)} \\
 &= \sum_{k=1}^{\infty} \left(\sum_{i=k}^{\infty} \text{pr (exactly } i \text{ probes)} \right) \\
 &= \sum_{k=1}^{\infty} \text{pr (at least } k \text{ probes)} \\
 &= \sum_{k=1}^{\infty} \frac{\alpha^{k-1}}{k} \\
 &= \frac{1}{\alpha} \sum_{k=1}^{\infty} \frac{\alpha^k}{k}
 \end{aligned}$$

This final summation is the Taylor series expansion for $\ln \frac{1}{1-\alpha}$ so that

$$U(\alpha) = \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

For $S(\alpha)$, we argue that it is exactly same as for double hashing. Since the ultimate contents of the table are as if the elements had been inserted in decreasing order by

double hashing, we may assume that they were so inserted. In this case the expected number of probes for a successful search is $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$ from our previous discussion

Comparing this to double hashing, we see that successful searches are no different, but unsuccessful searches require many fewer probes on the average as the table fills up. Specially ordered hash tables are to be recommended over conventional hash tables when unsuccessful searches are common and $h_2(k)$ can be computed without much expense.

4.8 Improvement with additional memory.

Thus far we have assumed that no additional memory is available in each table element. If additional memory is available, we can maintain some information in each entry to reduce the number of probes required to find a record or to determine that the desired record is absent.

Before looking at specific techniques, we should make one observation. The most obvious use to which additional memory can be put is to expand the size of the hash table. This reduces the load factor and immediately improves efficiency. Therefore in evaluating any efficiency improvements caused by adding more information to each table entry, one must consider whether the improvement outweighs utilizing the memory to expand the table.

On the other hand, the benefit of expanding each table entry by one or two bytes may indeed be worthwhile. Each table item (including space for key and record) may require 10, 50, 100 or even 1000 bytes, so that utilizing the space to expand the table may not buy as much as utilizing the space for small increments in each table element. (In reality, long records would not be kept within a hash table, since empty table entries waste too much space. Instead each table entry would contain a key and a pointer to

the record. This could still require 30 or 40 bytes if the key were large and 10 to 15 bytes for typical key sizes).

The first improvement that we consider reduces the time required for an unsuccessful search, but not that for a retrieval. It involves keeping with each table entry a one bit field whose value is initialized to zero and is set to 1 whenever a key to be inserted hashes or rehashes to that position but the position is found occupied. When hashing or rehashing a key during a search and finding the bit still set to 0, we immediately know that the key is not in the table, since if it were, it would either be found in that position or the bit would have been reset to 1 when it or some other key had been inserted. This method is called the pass-bit method; since the additional bit indicated whether a table element has been passed over while inserting an item.

The next method can be used with linear rehashing. In this case we can define a function $\text{prb}(j, \text{key})$ that directly computes the position of the j th rehash of key, which is the position of the j th probe in searching for key. $\text{Prb}(0, \text{key})$ is defined as $h(\text{key})$. For linear rehashing [$\text{rh}(i) = (i + c) \bmod \text{tablesize}$, whether c is constant], $\text{prb}(j, \text{key})$ is defined as $[h(\text{key}) + j * c] \bmod \text{tablesize}$. Note that no such routine can be defined for double hashing; therefore the method is not applicable to that technique.

The method uses an additional integer field, called a predictor, in each table position. Let $\text{prd}(i)$ be the predictor field in table position i . Initially all predictor fields are 0. Under linear rehashing, the predictor field is reset as follows. Suppose that key k_1 is being inserted and that j is the smallest integer such that $\text{prb}(j, k_1)$ is a probe position whose predictor field $\text{prd}[\text{prb}(j, k_1)]$ is 0. Then after k_1 is rehashed several more times and is inserted in position $\text{prb}(p, k_1)$, $\text{prd}[\text{prb}(j, k_1)]$ is reset from 0 to $p-j$. Then during a search, when position $\text{prb}(j, k_1)$ is found not to contain k_1 , the next position examined

is $\text{prb}[j + \text{prd}(\text{prb}(j, k1)), k1]$ or $\text{prb}(p, k1)$ rather than $\text{prb}(j + 1, k1)$. This eliminates $p-j-1$ probes.

An advantage of this approach is that it can be adapted quite easily when only a few extra bits are available in each table position. Since the predictor field contains the number of additional rehashes needed, in most cases this number is low and can fit in the available space. If only b bits are available for the prd field, and the predictor field cannot fit, the field value can be set to $2^b - 1$ (the largest integer representable by b bits). Then we would skip at least $2^b - 2$ probes after reaching such a position.

Unfortunately the predictor method cannot be applied at all under double hashing. The reason for this is that even secondary clustering is eliminated, so that there is no guarantee that $\text{prb}(n+x, K1)$ equals $\text{prb}(n+x, K2)$ even if $h(K1)$ equals $h(K2)$ and $\text{prb}(n, K1)$ equals $\text{prb}(n, K2)$.

An extension of the predictor method is the multiple predictor method. Under this technique, n_p predictor fields are maintained in each table position. A predictor hash routine $\text{ph}(\text{key})$, whose value is between 0 and n_p-1 , determines which predictor is used for a particular key. The j th predictor in table position i is referenced as $\text{prd}(i, j)$. When a key probes an occupied slot i that equals $\text{prb}(j, \text{key})$ such that $\text{ph}(k(i))$ equals $\text{ph}(\text{key})$, the next position probed is $\text{prb}[j + \text{prd}(i, \text{ph}(\text{key}), \text{key})]$. Similarly if $\text{ph}(k(i))$ equals $\text{ph}(\text{key})$ and $\text{prd}(i, \text{ph}(\text{key}))$ is 0, we know that key is not in the table. If key is inserted at $\text{prb}(i + x, \text{key})$, $\text{prd}(i, \text{ph}(\text{key}))$ is set to x .

The advantage of multiple predictor method is similar to the advantages of double hashing; it eliminates the effects of secondary clustering by dividing the list of elements that hash or rehash into a particular location into n_p separate and shorter lists. The predictor method also reduces the average number of probes for unsuccessful searches.

4.9 Hashing in external storage.

If a hash table is maintained in external storage on a disk or some other direct access device, time rather than space is the critical factor. Most systems have sufficient external storage to allow the luxury of unused allocated space for growth but cannot afford the time needed to perform an I/O operation for every element on a linked list. In such a situation the table in external storage is divided into a number of blocks called buckets. Each bucket consists of a useful physical segment of external storage such as a page or a disk track or track fraction. The buckets are usually contiguous and can be accessed by bucket offsets from 0 to $\text{tablesize} - 1$ that serves as hash values, much like indexes of an array in internal storage.

Alternatively, one or more contiguous storage blocks can be used as a hash table containing pointers to buckets distributed noncontiguously. In that situation the hash table is most likely read into memory as soon as the file is opened and remains in memory until the file is closed. When a record is requested, its key is hashed and the hash table is used to locate the external storage address of the appropriate bucket. Such a hash table is often called an index.

Each external memory bucket contains room for a moderate number of records (in practical situation, from 10 to 100). An entire bucket is read into memory at once and sequentially searched for the appropriate record. (of course a binary search or some other appropriate search mechanism based on the internal organization of the records within the bucket can be used, but the number of records in a bucket is usually small enough that no significant advantage is gained).

We should note that when dealing with external storage, the computational efficiency of a hash function is not as important as its success at avoiding hash clashes. It

is more efficient to spend micro seconds computing a complex hash function at internal CPU speeds than milliseconds or longer accessing additional buckets at I/O speeds when a bucket overflows. We also note that external storage space is inexpensive. Thus the number of contiguous initial buckets or the size of the hash table should be chosen such that it is unlikely that any of the buckets become full, even though this entails allocating unused space. Then when a new record must be inserted, there usually is room in the appropriate bucket, and an additional expensive I/O is not required.

If a bucket is full, and a record must be inserted, any of the rehash or chaining techniques discussed previously can be used. Of course, additional I/O operation are required when searching for records that are not in the buckets directly corresponding to the hash value. The size of the hash table is crucial. A hash table that is too large implies that most buckets will be empty, and a great deal of space is wasted. A hash table that is too small implies that buckets will be full, and large number of I/O operations will be required to access many records. If a file is very volatile, growing and shrinking rapidly and unpredictably, this simple hashing technique is inefficient in either space or time. We will see how to deal with this situation shortly.

When dealing with external storage such as a disk, the number of buckets that have to be read from external storage is not the only determinant of access efficiency. Another important factor is dispersal of the buckets accessed that is, how far apart the buckets accessed are from each other. In general a major factor in the time it takes to read a block from a disk is the seek time. This is the time it takes for the disk head to move to the location of the desired data on the disk. If two buckets accessed one after other are far apart, more time is required then if they are close together. Given this fact, it would seem that linear rehashing is the most effective technique because

although it may require accessing more buckets, the bucket it accesses are contiguous.

If separate chaining is used, it is desirable to reserve an overflow area in each cylinder of the file so that full buckets in that cylinder can link to the overflow records in the same cylinder, thus minimizing seek time and essentially eliminating the dispersal penalty. It should be noted that the overflow area need not be organized into buckets, and should be organized as individual records with links. In general few records overflow and there is only a small chance that sufficiently many will overflow from a single bucket to fill an additional complete bucket. Thus by keeping individual overflow records, more buckets will overflow into the same cylinder. Since space is reserved within the file for overflow records, the load factor does not represent a true picture of storage utilization for this version of separate chaining. The number of accesses in separate chaining is therefore higher for a given amount of external storage.

Although double hashing requires fewer accesses than linear rehashing, it disperses the buckets that must be accessed to a degree that may overwhelm this advantage. However, in systems where dispersal is not a factor, double hashing is preferred. This is true of modern large multi-user systems in which many user may be requesting access to a disk simultaneously, and the requests are scheduled by the operating system based on the way the data is arranged in the disk. In such situations, waiting time for disk accesses is required in any case, so that dispersal is not a significant factor. The major drawback in using hashing for external file storage is that sequential access is not possible, since a good hash function disperses the keys without regard to order.

4.10 The Separator method.

One technique for reducing access time in external hash table at the expense of increasing insertion is the separator method. The method uses rehashing (either linear rehashing

or double hashing) to resolve collisions but also uses an additional hash routine, S , called the signature function. Given a key key , let $h(key, i)$ be the i th rehash of key and let $S(key, i)$ be the i th signature of key . If a record with key key is stored in bucket number $h(key, j)$, the current signature of the record and the key, $sig(key)$ is defined as $S(key, j)$. That is if a record is placed in a bucket corresponding to its key's j th rehash, its current signature is its key's j th signature.

A separator table, sep , is maintained in internal memory. If b is a bucket number $sep(b)$ contains a signature value greater than the current signature of every record in bucket b . To access the record with key key , repeatedly hash key until obtaining a value j such that $sep(h(key, j)) > S(key, j)$. At that point, if the record is in the file it must be in bucket $(h(key, j))$. This ensures the ability to access any record in the file with only a single external memory access.

If m is the number of bits allowed in each item of the separator table, the signature function, S , is restricted to producing values between 0 and $2^m - 2$. Initially, before any overflows have occurred in bucket b , the value of $sep(b)$ is set to $2^m - 1$, so that any record whose key hashes to b can be inserted directly into bucket (b) regardless of its signature. Now, suppose that bucket (b) is full and a new record to be inserted hashes into b . Then the records in b with the largest current signature (lcs) must be removed from bucket b to make room for the new record. The new record is then inserted into bucket (b) , and the old records with current signature lcs that were removed from bucket b are rehashed and relocated into new buckets. $sep(b)$ is then reset to lcs , since the current signature of all records in bucket (b) are less than lcs . Note that more than one record may have to be removed from a bucket if they have equal maximal current signature values.

Records that overflow from a bucket during an insertion may cause cascading overflows in other buckets when attempting to relocate them. This means that an insertion may cause an indefinite number of additional external storage reads and writes. In practice, a limit is placed on the number of such cascading overflows beyond which the insertion fails. If the insertion fails, it is necessary to restore the file to the state it was in before inserting the new record that caused the original overflow. This is usually done by delaying writing modified buckets to external storage, keeping the modified versions in internal memory until it is determined that the insertion can be completed successfully. If the insertion is aborted because the cascade limit is reached no writes are done, leaving the file in its original state. The number of modified pages per insertion rises rapidly as the load factor is increased, so the technique is impractical with a load factor greater than 95 percent. Larger signature values and larger bucket sizes permit the method to be used with larger load factors.

4.11 Dynamic Hashing and Extendible Hashing.

One of the most serious drawbacks of hashing for external storage is that it is insufficiently flexible. Unlike internal data structures, files and databases are semipermanent data structures that are not usually created and destroyed within the lifetime of a single program. Further the contents of an external storage structure tend to grow and shrink unpredictably. All the hash table structuring methods that we have examined have a sharp space/time trade-off. Either the table uses a large amount of space for efficient access, resulting in much wasted space when the structure shrinks, or it uses a small amount of space and accommodates growth very poorly by sharply increasing the access time for overflow elements. We would like to develop a scheme that does not utilize too much extra space when a file is small but permits efficient access when it grows

larger. Two such schemes are called dynamic hashing, attributable to Larson [31], and extendible hashing, attributable to Fagin, Nievergelt, Pippenger, and Strong [28].

The basic concept under both methods is the same. Initially, m buckets and a hash table (or index) of size m are allocated. Assume that m equals 2^b and assume a hash routine h that produces hash values that are $w > b$ bits in length. Let $hb(\text{key})$ be the integer between 0 and m represented by the first b bits of $h(\text{key})$. Then, initially, hb is used as hash routine, and records are inserted into the m buckets as in ordinary external storage hashing.

When a bucket overflows, the bucket is split in two and its records are assigned to the two new buckets based on the $(b+1)$ st bit of $h(\text{key})$. If the bit is zero, the record is assigned to the first (or left) new bucket; if the bit is 1, the record is assigned to the second (or right) bucket. The records in each of the two new buckets now all have the same first $b+1$ bits in their hash keys, $h(\text{key})$. Similarly, when a bucket representing i bits overflows, the bucket is split and the $(i+1)$ st key bit of $h(\text{key})$ for each record in the bucket is used to place the record in the left or right new bucket. Both new buckets then represent $i+1$ bits of the hash key. We call the bucket whose keys have 0 in their $(i+1)$ st bit the 0-bucket and the other bucket the 1-bucket.

Dynamic and extendible hashing differs as to how the index is modified when a bucket splits. Under dynamic hashing, each of the m original index entries represents the root of a binary tree of whose leaves contains a pointer to a bucket. Initially each tree consists of only one node (a leaf node) that points to one of the m initially allocated buckets. When a bucket splits, two new leaf nodes are created to point to the two new buckets. The former leaf that had pointed to the bucket being split is transformed into a nonleaf node whose left son is the leaf pointing to the 0-bucket and whose right son

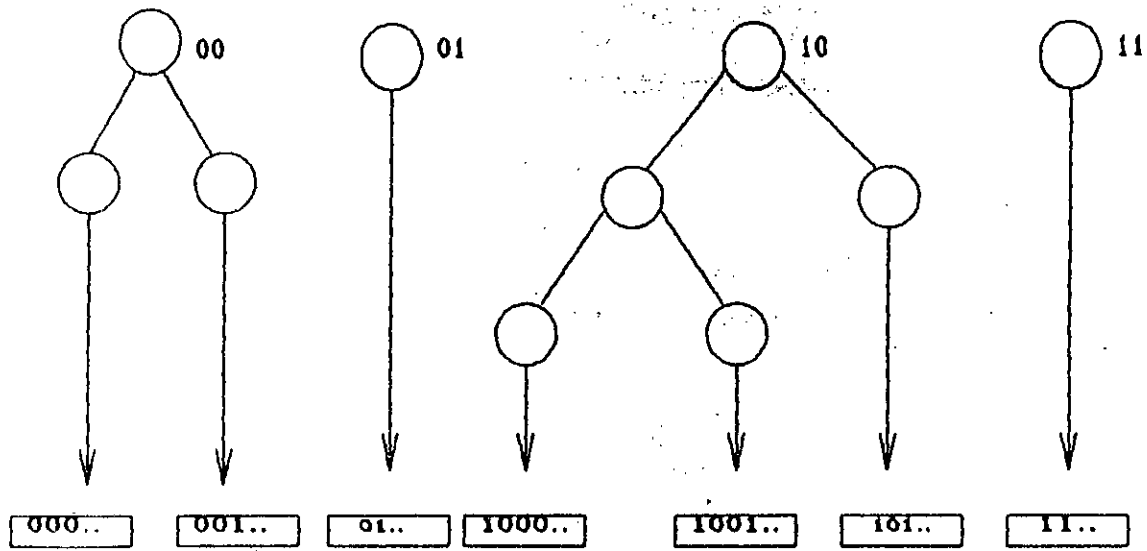


Fig. 4.2 Dynamic Hashing Scheme

Depth = 4

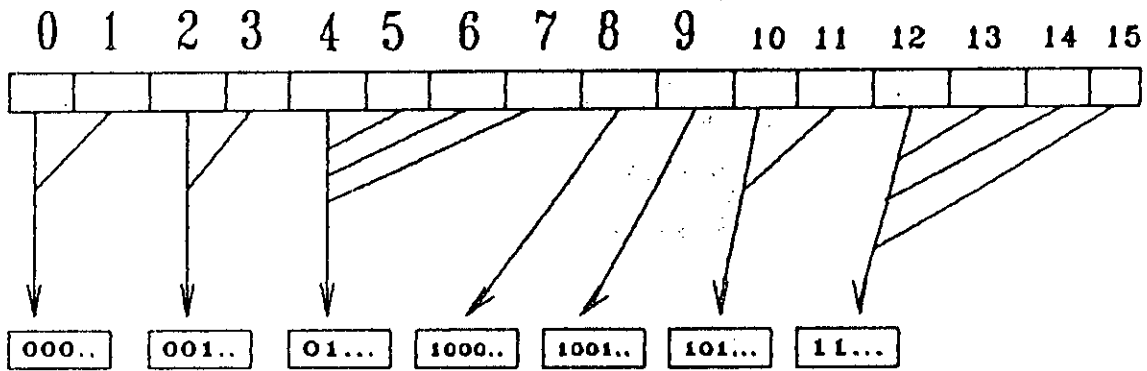


Fig. 4.3 Extendible Hashing Scheme

is the leaf pointing to the 1-bucket. Dynamic hashing with $b=2$ ($m=4$) is illustrated in Fig. 4.2.

To locate a record under dynamic hashing, compute $h(\text{key})$ and use the first b bits to locate a root node in the original index. Then use each successive bit of $h(\text{key})$ to move down the tree, going left if the bit is zero and right if the bit is 1, until a leaf is reached. Then use the pointer in the leaf to locate the bucket that contains the desired record, if it exists.

In extendible hashing, each bucket contains an indication of the number of bits of $h(\text{key})$ that determine which records are in that bucket. This number is called the bucket depth. Initially, this number is b for all bucket entries; it is increased by 1 each time a bucket splits. Associated with the index is the index depth, d , which is the maximum of all the bucket depths. The size of the index is always 2^d (initially, 2^b).

Suppose that a bucket of depth i is to be split. Let a_1, a_2, \dots, a_i be the first i bits of $h(\text{key})$ for the records in the bucket being split. There are two cases to consider: $i < d$ and $i = d$. If $i < d$ all index positions with bit values $a_1, a_2, \dots, a_i 0 0 \dots 0$ (up to a bit size d) through $a_1, a_2, \dots, a_i 0 1 1 \dots 1$ of the index (that is all positions starting with $a_1 \dots a_i 0$) are reset to point to the 0-bucket, and the index positions with bit values $a_1, a_2, \dots, a_i 1 0 \dots 0$ through $a_1, a_2, \dots, a_i 1 1 \dots 1$ (that is, all positions starting with $a_1 \dots a_i 1$) are reset to point to the 1-bucket. If $i = d$ the index is doubled in size from 2^d to 2^{d+1} ; the old contents of index positions $x_1 \dots x_d$ are copied into the new positions $x_1 \dots x_d 0$ and $x_1 \dots x_d 1$; the contents of index positions $a_1 \dots a_d 0$ is set to point to the new 0-bucket, and the contents of index position $a_1 \dots a_d 1$ to point to the new 1-bucket. Extendible hashing is illustrated in Fig. 4.3.

To locate a record under extendible hashing, compute $h(\text{key})$ and use the first d

bits (where d is the index depth) to obtain a position in the index. The contents of this position point to the bucket containing the desired record, if it exists.

In comparing dynamic and extendible hashing, we note that extendible hashing is more time efficient, since a tree path need not be traversed as in dynamic hashing. However, if the entire index is kept in memory, the time spent in traversing the tree path does not involve any I/Os. Traversal time is therefore likely to be negligible compared with the time for accessing the bucket. The maximum number of tree nodes required in dynamic hashing is $2n-1$, assuming n buckets, whereas there may be as many as 2^{n-1} index entries required under extendible hashing. However, usually fewer than twice as many extendible hashing index entries as dynamic hashing hashing tree nodes are required and the tree nodes require two pointers compared with one for each extendible hashing index entry. Thus the two methods are comparable in average internal space utilization.

It is also possible to compress very large extendible hashing indexes by keeping only one copy of each bucket pointer and maintaining from/to indicators. Another point to note is that extendible hashing performs the same way regardless of the value of m , the initial number of index entries, whereas dynamic hashing requires longer tree paths if m is smaller.

4.12 Choosing a Hash Function.

Let us now turn to the question of how to choose a good hash function. Clearly, the function should produce as few hash clashes as possible; that is, it should spread the keys uniformly over the possible array indices. Of course, unless the keys are known in advance, it cannot be determined whether a particular hash function disperses them properly. However, although it is rare to know the keys before selecting a hash function,

it is fairly common to know some properties of the keys that affect their dispersal.

In general, a hash function should depend on every single bit of the key, so that two keys that differ in only one bit or one group of bits (regardless of whether the group is at the beginning, end, or middle of the key or strewn throughout the key) hash into different locations. Thus a hash function that simply extracts a portion of a key is not suitable. Similarly, if two keys are simply digit or character permutations of each other (such as 139 and 319 or meal and lame), they should also hash into different values. The reason for this is that key sets frequently have clusters or permutations that might otherwise result in collisions.

For example, the most common hash function (which we have used in the examples of this chapter) uses the division method, in which an integer key is divided by the table size and the remainder is taken as the hash value. This is the hash function $h(\text{key}) = \text{key} \% \text{tablesize}$. Suppose, however, that tablesize equals 1000 and that all the keys end in the same three digits (for example, the last three digits of a part number might represent a plant number and the program is being written for that plant). Then the remainder on dividing by 1000 yields the same value for all the keys, so that a hash clash occurs for each record except the first. Clearly, given such a collection of keys, a different hash function should be used.

It has been found that the best results with the division method are achieved when tablesize is prime (that is, it is not divisible by any positive integer other than 1 and itself). However, even if tablesize is prime, an additional restriction is called for. If r is the number of possible character codes on a particular computer (assuming an 8-bit byte, r is 256) and if tablesize is a prime such that $r \% \text{tablesize} = 1$, the hash function $\text{key} \% \text{tablesize}$ is simply the sum of the binary representation of the characters

in the key modulo tablesize. For example, suppose that r equals 256 and that tablesize equals 17, in which case $r \% \text{tablesize} = 1$. Then the key 27956, which equals $148 * 256 + 68$ (so that the first byte of its representation is 148 and the second byte is 68), hashes into $37956 \% 17$, which equals 12, which equals $(148 + 68) \% 17$. Thus two keys that are simply permutations (such as steam and mates) will hash into the same value. This may promote collisions and should be avoided. Similar problems occur if tablesize is chosen so that $r^k \% \text{tablesize}$ is very small or very close to tablesize for some small value of k .

Another hash method is the multiplicative method. In this method a real number c between 0 and 1 is selected; $h(\text{key})$ is defined as $\text{floor}(m * \text{frac}(c * \text{key}))$, where the function $\text{floor}(x)$, available in the standard library `math.h`, yields the integer part of the real number x and $\text{frac}(x)$ yields the fractional part. (Note that $\text{frac}(x) = x - \text{floor}(x)$). That is, multiply the key by a real number 0 and 1, take the fractional part of the product yielding a random number between 0 and 1 dependent on every bit of the key and multiply by m to yield an index between 0 and $m - 1$. If the word size of the computer is b bits, c should be chosen so that $2^b * c$ is an integer relatively prime to 2^b and c should not be too close to either 0 or 1. Also if r , as before, is the number of possible character codes, avoid values of c such that $\text{frac}((r^k) * c)$ is too close to 0 or 1 for some small value of k (these values yield similar hashes for keys with the same last k characters) and of values c of the form $i/(r - 1)$ or $i/(r^2 - 1)$ (these values yield similar hashes for keys that are character permutations). Values of c that yield good theoretical properties are 0.6180339887 [which equals $(\text{sqrt}(5) - 1)/2$] or 0.381966113 [which equals $1 - (\text{sqrt}(5) - 1)/2$]. If m is chosen as power of 2 such as 2^p , the computation of $h(\text{key})$ can be done quite efficiently by multiplying the one-word integer key by the one-word integer $c * 2^b$

to yield a two-word product. The integer represented by the most significant p bits of the integer in the second word of this product is then used as the value of $h(\text{key})$.

In another hash function, known as the midsquare method, the key is multiplied by itself and the middle few digits (the exact number on the number of digits allowed in the index) of the square are used as the index. If the square is considered as a decimal number, the table size must be a power of 10, whereas if it is considered as a binary number, the table size must be a power of 2. Alternatively, the number represented by the middle digits can be divided by the table size and the remainder used as the hash value. Unfortunately, the midsquare method does not yield uniform hash values and does not perform as well as the previous two techniques.

The folding method breaks up a key into several segments that are added or exclusive ORed together to form a hash value. For example, suppose that the internal bit string representation of a key is 010111001010110 and that 5 bits are allowed in the index. The three bit strings 01011, 10010, and 10110 are exclusive ored to produce 01111, which is 15 as a binary integer. (The exclusive or of two bits is 1 if the two bits are different and 0 if they are the same. It is the same as the binary sum of the bits, ignoring the carry). The disadvantage of the folding method is that two keys that are k -bit permutations of each other (that is, where both keys consist of the same groups of k bits in a different order) hash into the same k -bit value. Still another technique is to apply a multiplicative hash function to each segment individually before folding.

There are many other hash functions, each with its own advantages and disadvantages depending on the set of keys to be hashed. One consideration in choosing a hash function is efficiency of calculation; it does no good to be able to find an object on the first try if that try takes longer than several tries in an alternative method.

If the keys are not integers, they must be converted into integers before applying one of the foregoing hash functions. There are several ways to do this. For example, for a character string the internal bit representation of each character can be interpreted as a binary number. One disadvantage of this is that the bit representations of all the letters or digits tend to be very similar on most computers. If the keys consist of letters alone, the index of each letter in the alphabet can be used to create an integer. Thus the first letter of the alphabet (a) is represented by the digits 01 and the fourteenth (n) is represented by the digits 14. The key 'hello' is represented by the integer 0805121215. Once an integer representation of a character string exists, the folding method can be used to reduce it to manageable size. However, here too every other digit is a 0, 1 or 2, which may result in nonuniform hashes. Another possibility is to view each letter as a digit in base-26 notation so that 'hello' is viewed as the integer $8 * 26^4 + 5 * 26^3 + 12 * 26^2 + 12 * 26 + 15$.

One of the drawbacks of all these hash functions is that they are not order preserving; that is, the hash values of the two keys are not necessarily in the same order as the keys themselves. It is, therefore, not possible to traverse the hash table in sequential order by key. An example of a hash function that is order preserving is $h(\text{key}) = \text{key}/c$, where c is some constant chosen so that the highest possible key divided by c equals $\text{tablesize} - 1$. Unfortunately, order-preserving hash functions usually are severely nonuniform, leading to many hash clashes and a larger average number of probes to access an element. Note also that to enable sequential access to keys, the separate chaining method of resolving collisions must be used.

CHAPTER 5

RESULTS AND CONCLUSIONS

5.1 Introduction.

This chapter is intended to present the experimental results based on the algorithms discussed so far in the previous chapters. This chapter comprises four main sections of which the first one deals with the results of chapter one, the second discusses the experiments on chapter two and so on. Each section begins with a brief description of the experimental setup and verification procedure for each of the algorithms and justifies why a particular way of doing the experiments has been chosen when others have been discarded. Finally for the majority of the algorithms we will try to correlate the experimental results with the absolute theoretical results when the behaviour of an algorithm has already been established. In cases when sufficient theoretical results are lacking, we will try to establish some empirical relationships. It should be emphasized that one can go a long way without limit if one likes to experiment the algorithms in all possible respects. But we have chosen some representative properties of the algorithms as the basis of our experiments. In an actual implementation of an algorithm time and space complexity are the major factors to be considered for its acceptance. Consequently, these factors have been given the highest priority for algorithm evaluation.

5.2 Searching by comparison of keys.

The first chapter of this thesis discusses the search algorithms based on the comparison of keys kept ordered in a static table. The reason for the search table to be static is that both insertion and deletion from an ordered table is too costly to be worthy

in practical situation. Moreover, the algorithms of chapter one depend on the orderly relation among the keys so that the table should be kept ordered for the algorithms to work as intended. An ordered table of 987 elements has been chosen in order to evaluate the algorithms presented in chapter one. The basis for this choice is that 987 is a perfect fibonacciian number and the same table will be used to evaluate the performance of Fibonacciian search algorithm.

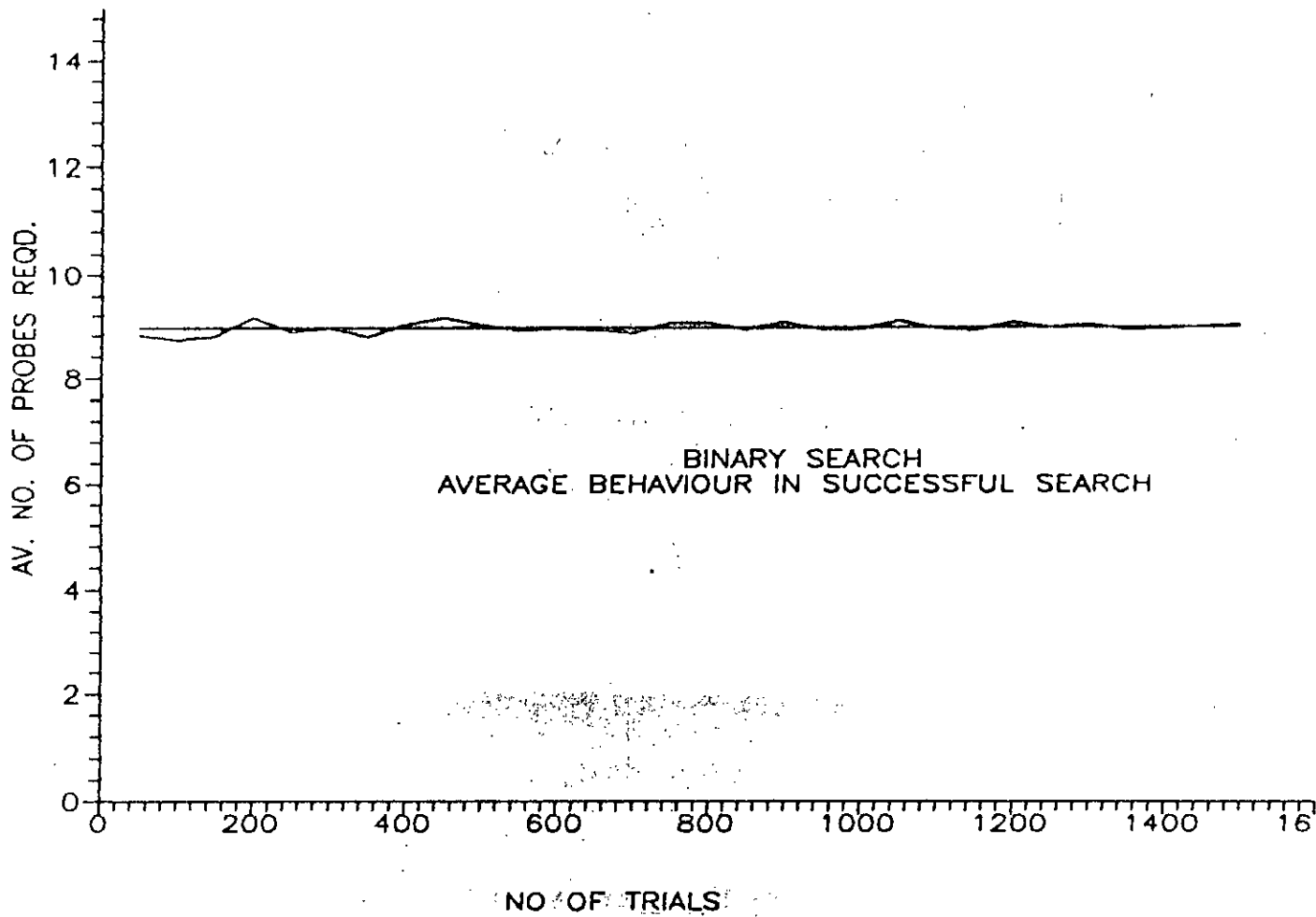
The programs of chapter one have been designed to obtain the following data:

- (i) Average number of probes required in a successful search.
- (ii) Average number of probes required in an unsuccessful search.
- (iii) Average time required for a successful search and
- (iv) Average time required for an unsuccessful search.

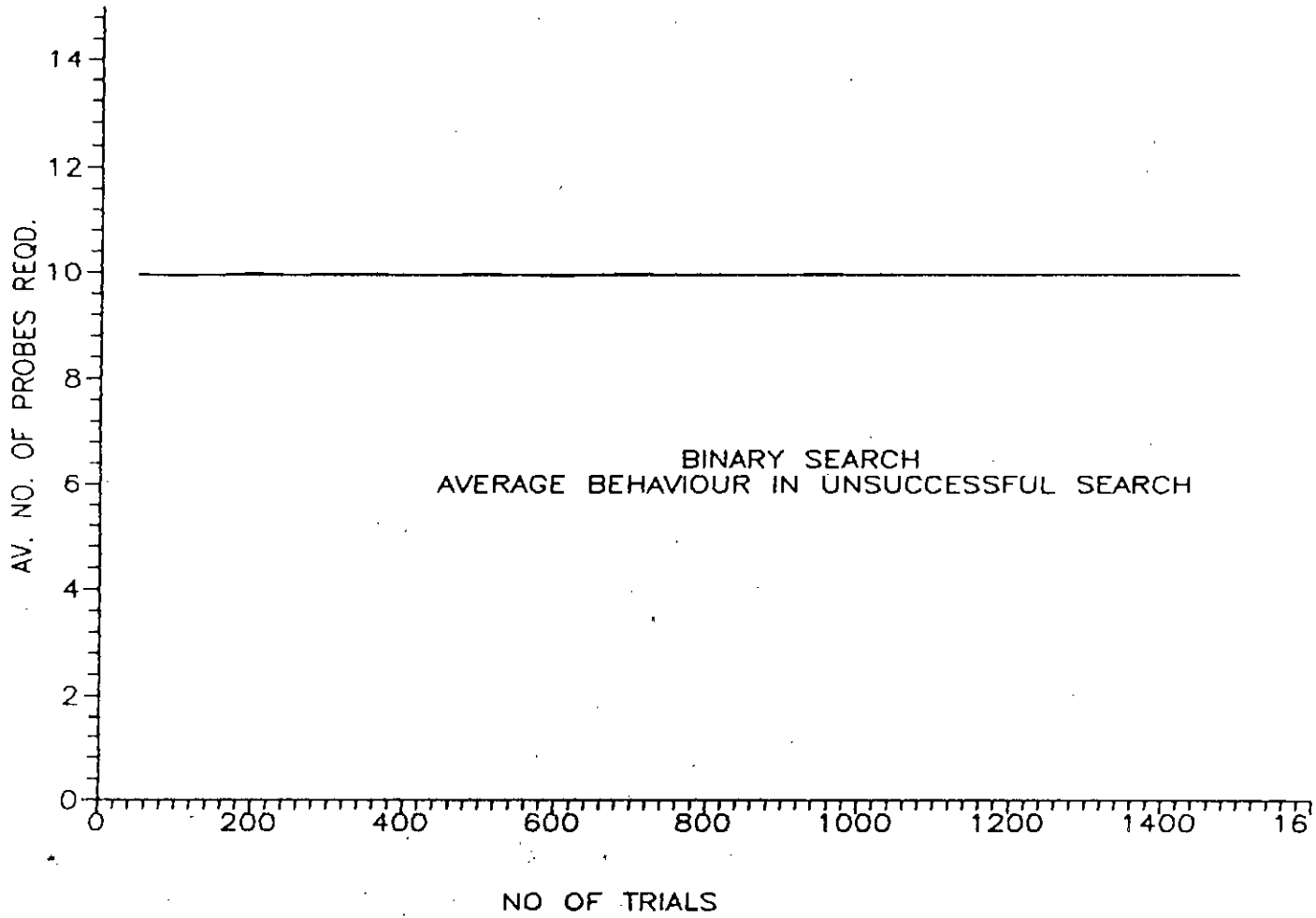
To determine the average number of probes required, attention has been given so that experimental results bear a true reflection of the approximate true average. For this reason thirty sets of data, each set comprising a sufficient number of trials (the first set consists of 50 trials, the second one consists of 100 trials and so on) have been employed to plot curves depicting the average behaviour of the algorithms. Each set of data has been collected after 10 iterations for the same reasons. Since time for each probe is too small to be counted, one thousand iterations for each probe have been taken to obtain a sufficiently accurate data to be relied upon.

The curves of Graph 1.1 through 1.4 show the average behaviour of Binary search algorithm. As derived in section 1.3, the average number of probes required for a successful search and unsuccessful search are

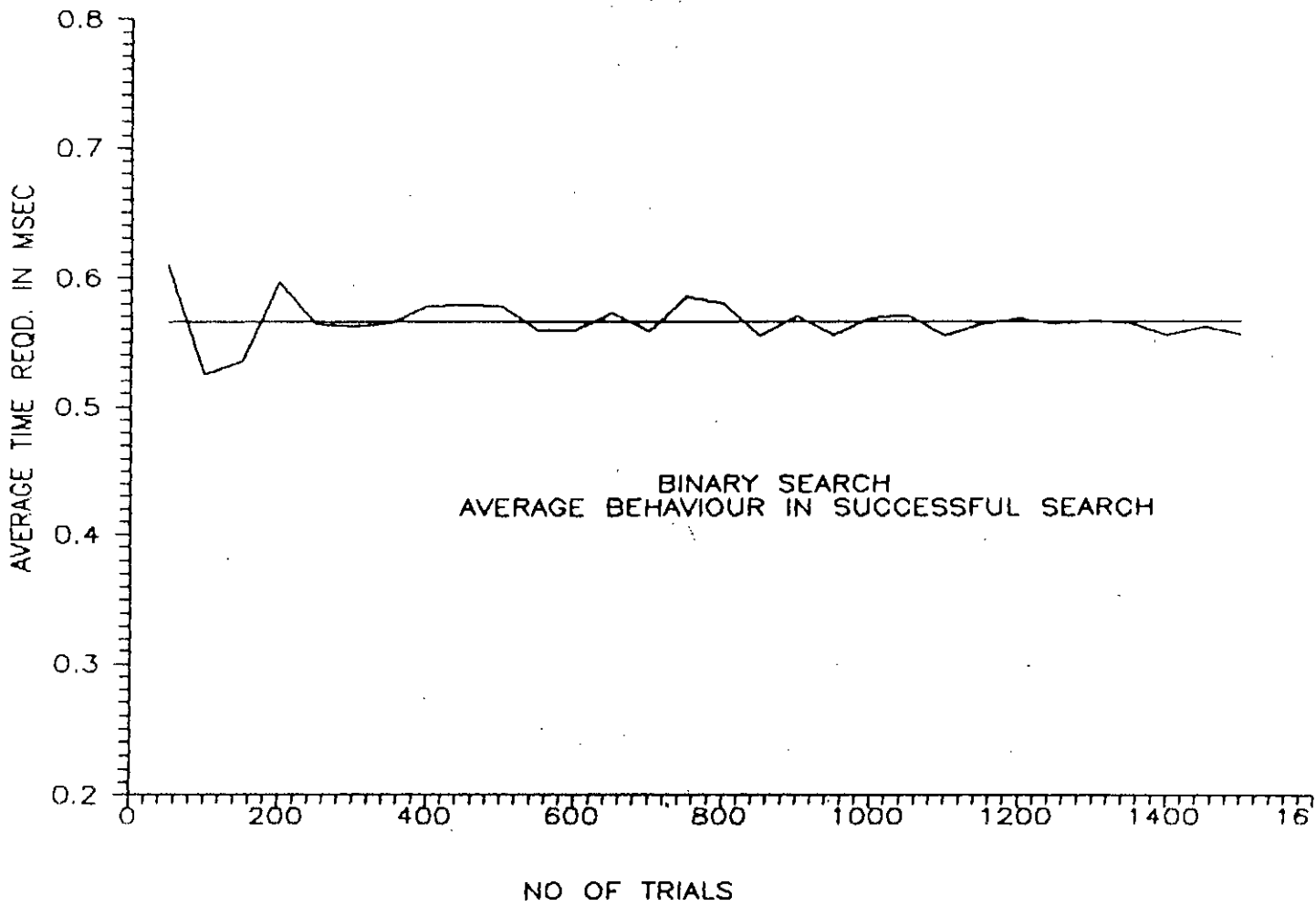
$$S_N = \left(1 + \frac{1}{N}\right) [\lg_2(N+1) + 2 - \theta - 2^{1-\theta}] - 1$$



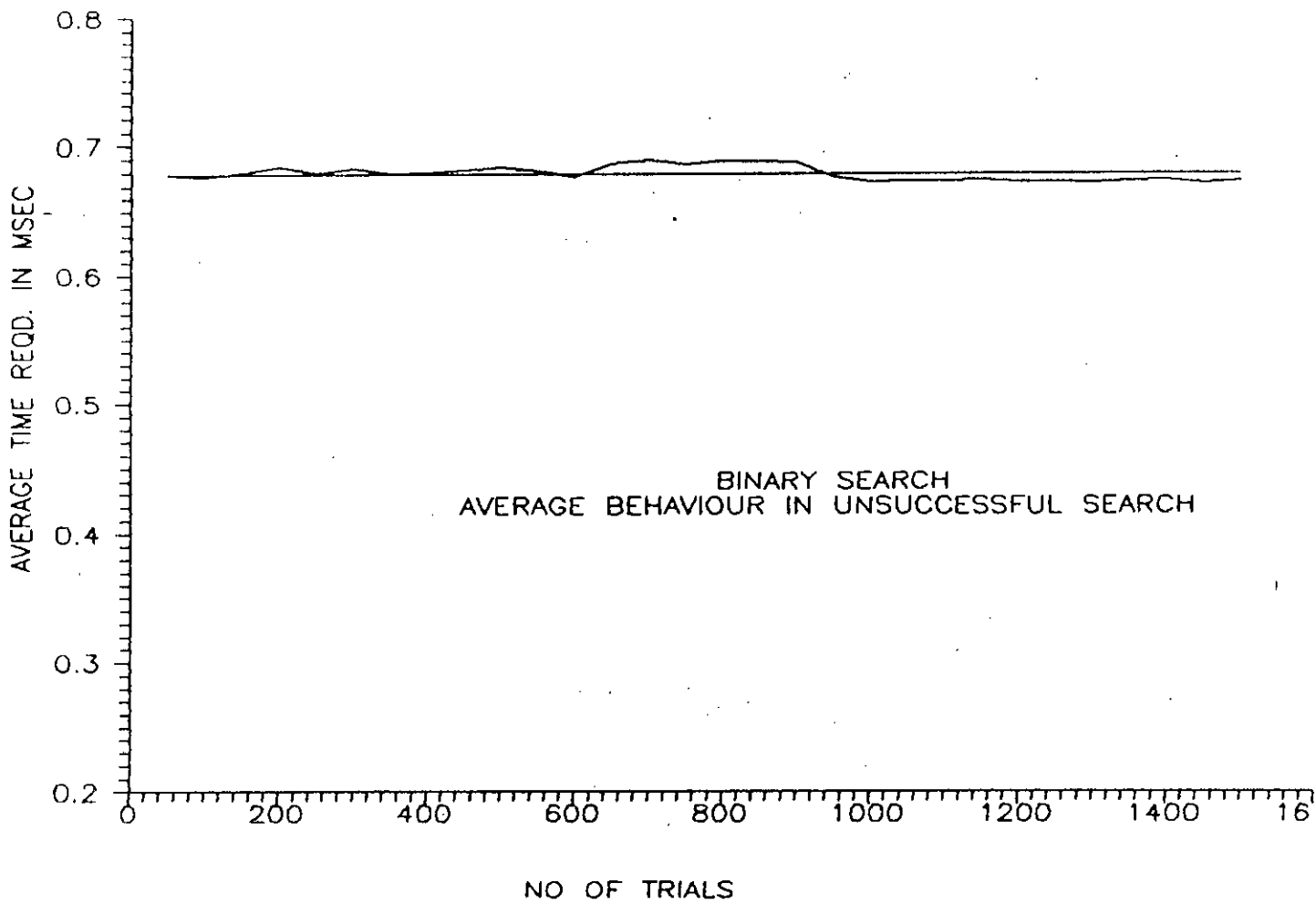
GRAPH 1.1 Average no. of probes reqd. in successful search in a Binary tree



GRAPH 1.2 Average no. of probes reqd. in unsuccessful search in a Binary tree



GRAPH 1.3 Average time reqd. in successful search in a Binary tree



GRAPH 1.4 Average time reqd. in unsuccessful search in a Binary tree.

and

$$U_N = \lg_2(N + 1) + 2 - \theta - 2^{1-\theta}$$

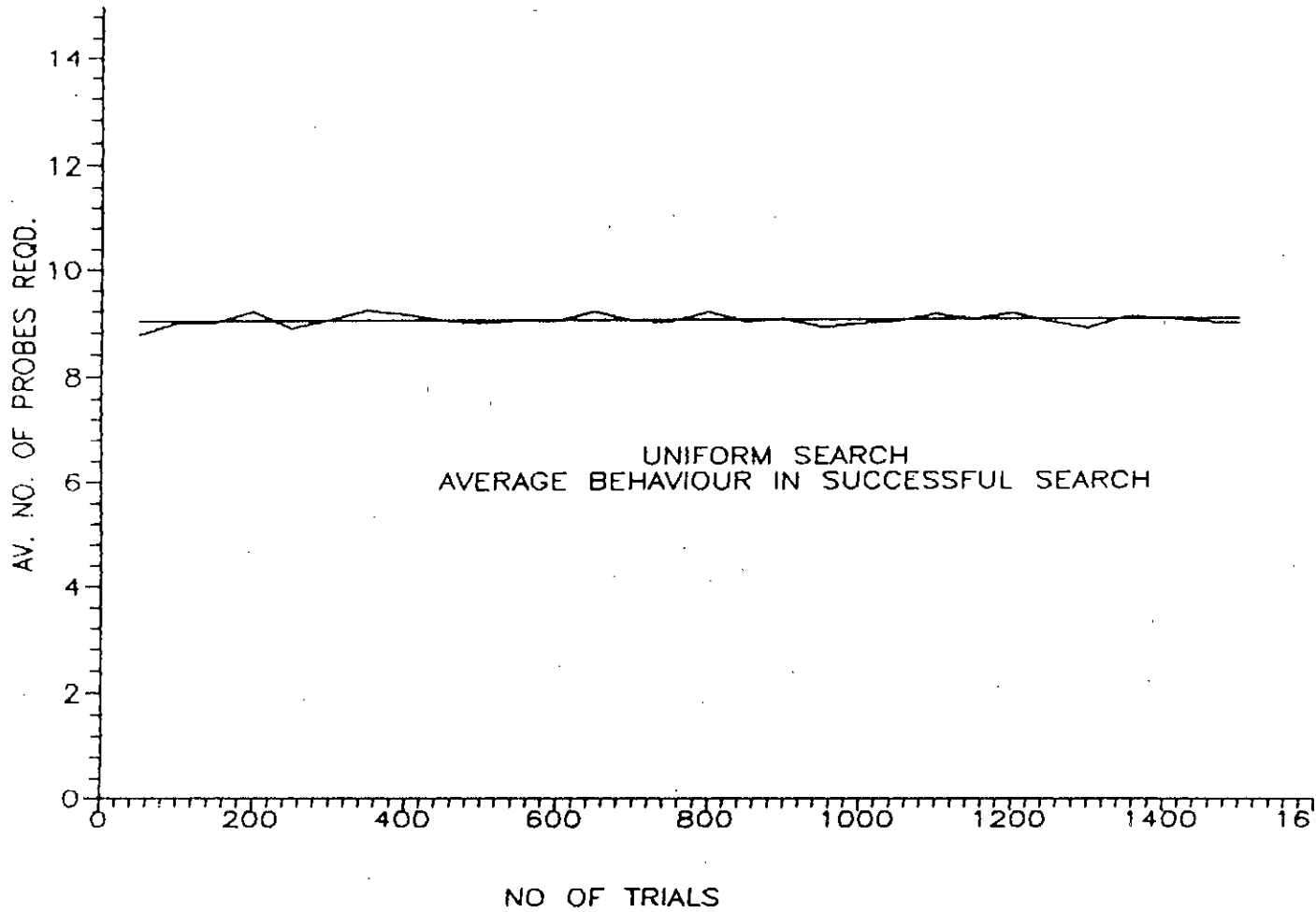
respectively where

$$\theta = \lg_2(N + 1) - \lfloor \lg_2(N + 1) \rfloor$$

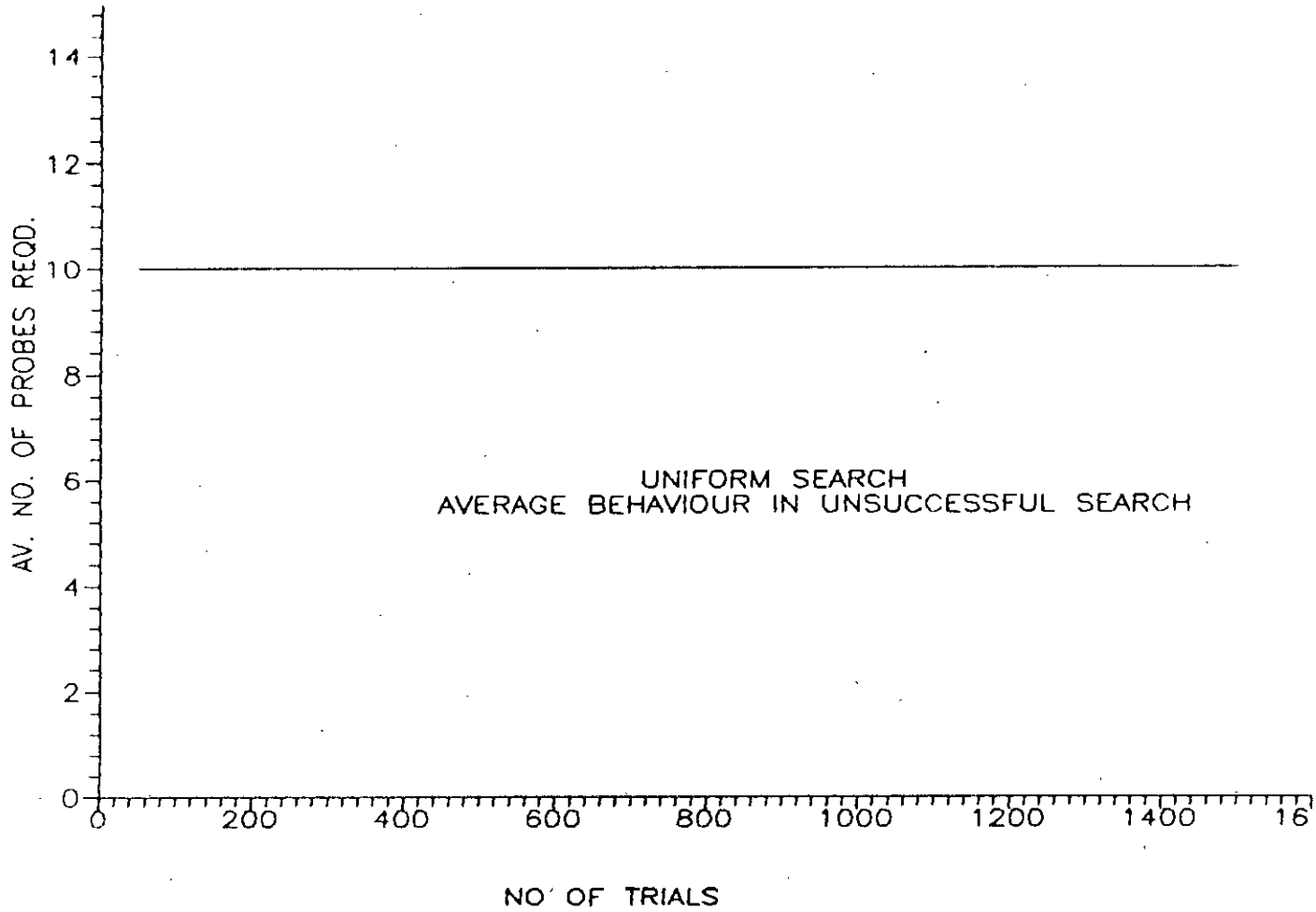
For the above mentioned table size the average number of probes required are 8.97262 and 9.96252 for successful and unsuccessful searches respectively. According to the curves of Graph 1.1 and 1.2, these values are 8.94751 and 9.95256 respectively which bear a close resemblance with the theoretical results. The average time required for successful and unsuccessful searches are 0.5656917 and 0.6788918 msec respectively as shown in the curves of Graph 1.3 and 1.4. These values are somewhat larger in magnitude than the true average because for the program requirement some extra instructions have been unintentionally embedded in the search loop which could not be avoided in the time calculation.

The next algorithm which has been tested is the uniform binary search. As stated in section 1.4, the external nodes all appear on the outermost level of the search tree. That is why the average number of probes for unsuccessful search remains constant in this situation, and the average number of probes for a successful search should be remain same as that for the binary search algorithm. As shown in the curves of Graph 1.5 and 1.6, the experimental results conform with this theoretical observation. Here average number of probes required for successful and unsuccessful searches are 9.0058183 and 10 respectively.

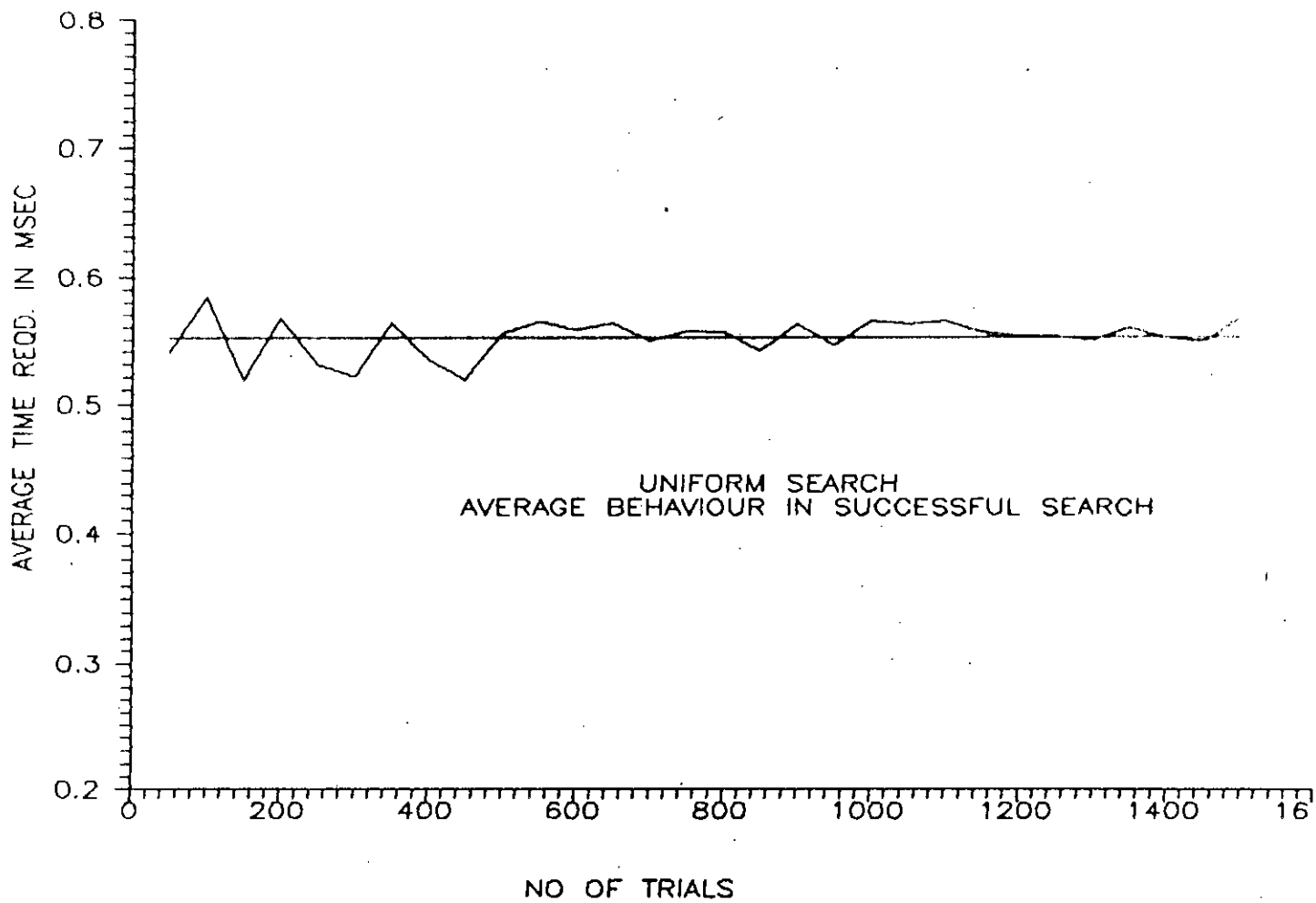
A table of various values of δ has been used to make the uniform search algorithm faster. The experimental curves of Graph 1.7 and 1.8 show that average time requirement in uniform search algorithm is lower than that for binary search algorithm.



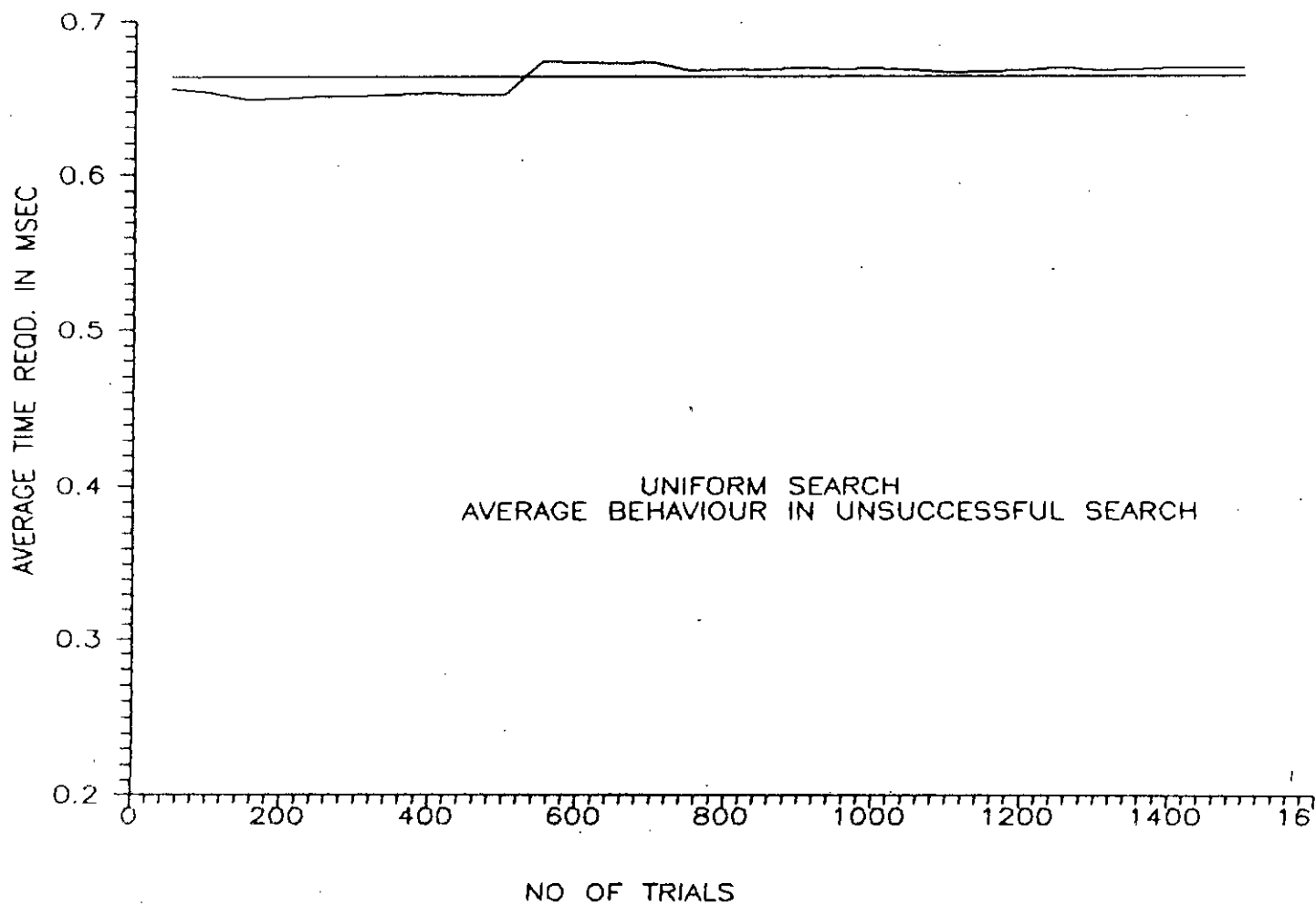
GRAPH 1.5 Average no. of probes reqd. in successful search in a Uniform search tree



GRAPH 1.6 Average no. of probes reqd. in unsuccessful search in a Uniform search tree



GRAPH 1.7 Average time reqd. in successful search in a Uniform search tree



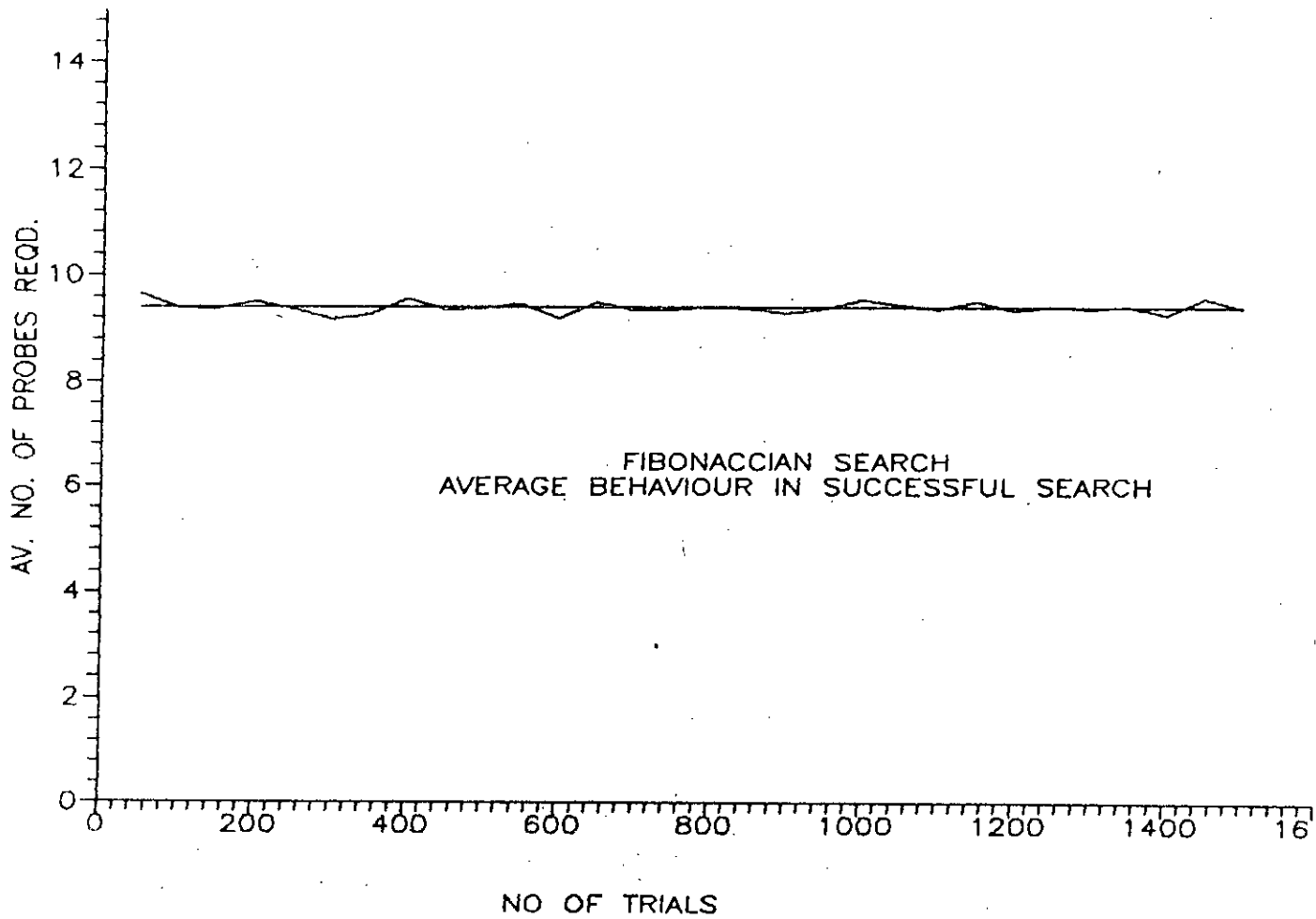
GRAPH 1.8 Average time reqd. in unsuccessful search in a Uniform search tree

These values for successful and unsuccessful searches are 0.551766 and 0.662638 msec respectively. This is due to the fact that in the binary search algorithm, every iteration requires a division operation to determine the middle element of an interval. But in uniform search and in Fibonacci search only addition operation are required in each iteration. In our experiments division operation has been avoided in uniform search by using a table of δ values to be used in each subsequent iteration of the algorithm.

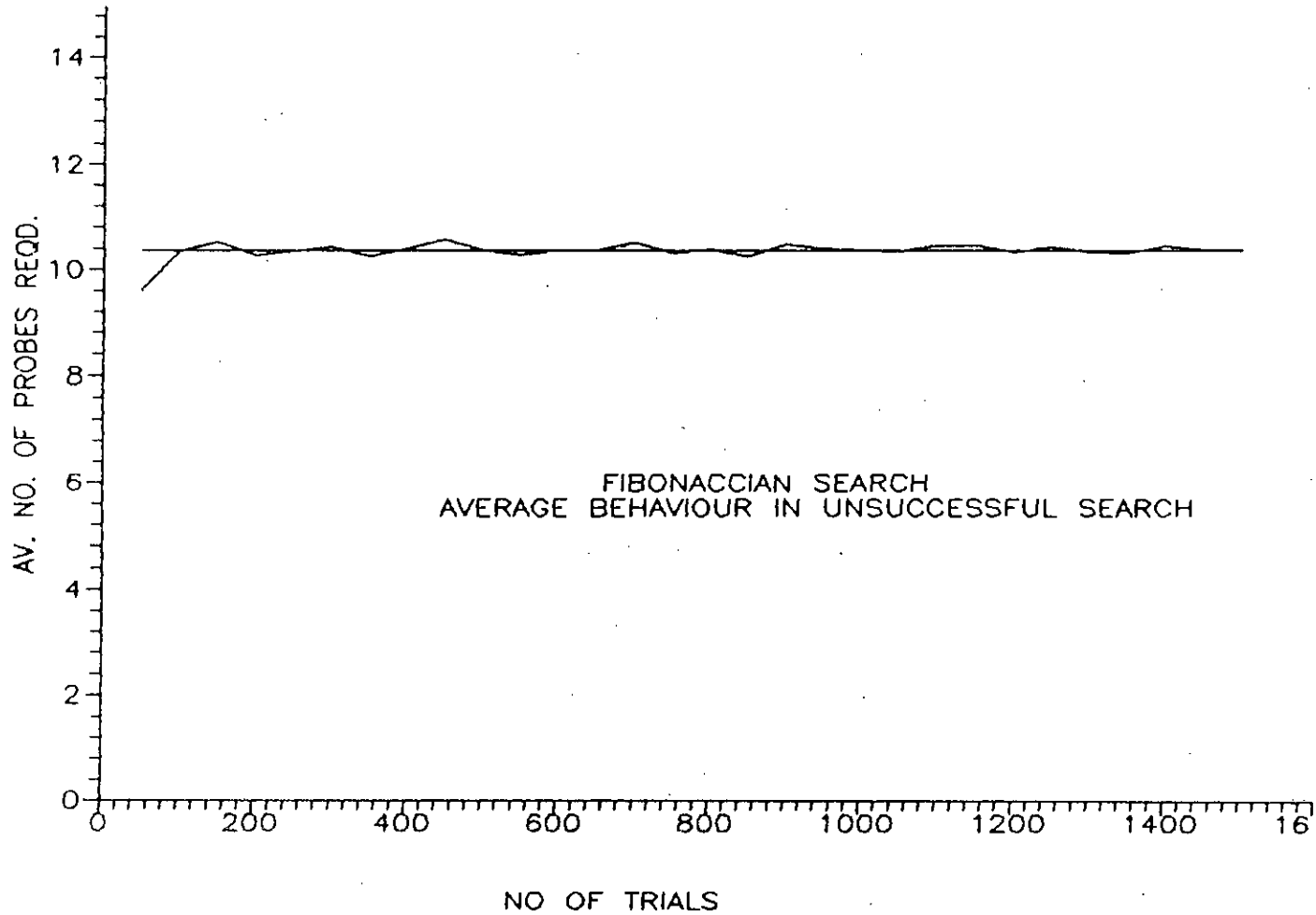
We will terminate this section after a brief look on Fibonacci search algorithm. As the experimental curves in Graph 1.9 and 1.10 show, the average number of probes required in successful search is 9.3932727 and that for an unsuccessful search is 10.353305. The curves in Graph 1.11 and 1.12 verify the fact that although Fibonacci search require a larger number of probes for both successful and unsuccessful searches, yet the time required for a single iteration is much lower than that for binary search algorithm because each iteration requires only addition and subtraction operations and these operations are much cheaper than division operations required in the binary search algorithm.

5.3 Search Trees.

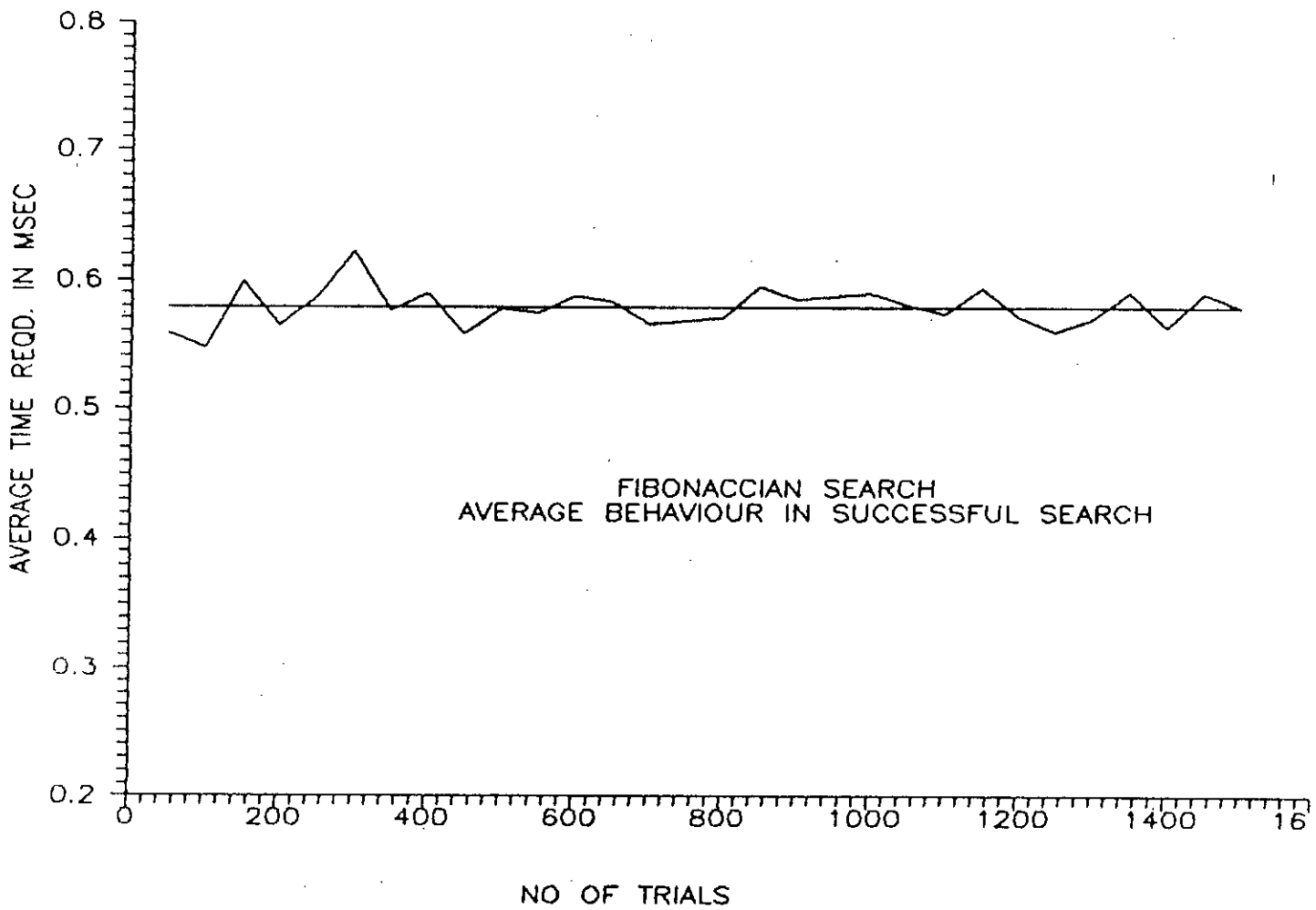
As discussed in chapter two an explicit binary tree is the appropriate data structure for the flexibility of frequent insertion and deletion, and the shape and size of the tree is extremely sensitive to the nature of insertions and deletions. In the worst case the dynamic tree can degenerate into a linear list if the search keys come into their natural order, and if we are extremely lucky the tree can become a perfect balanced tree. But we are interested in the average behaviour when the search keys come into truly random manner. Thus in the experiments on dynamic tree, a number of search trees of various sizes have been constructed on the assumption that the incoming keys are truly random



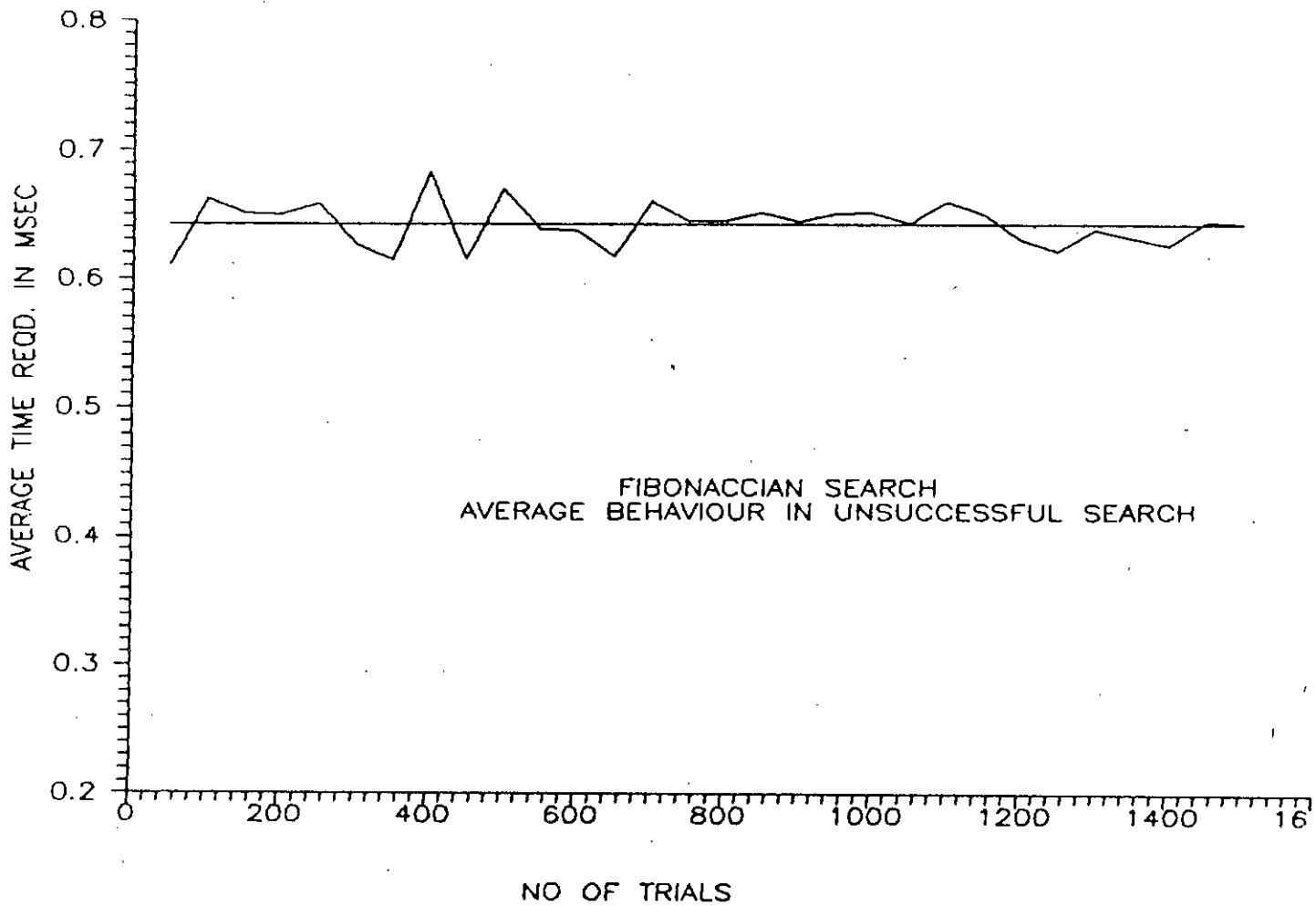
GRAPH 1.9 Average no. of probes reqd. in successful search in a Fibonacci search tree



GRAPH 1.10 Average no. of probes reqd. in unsuccessful search in a Fibonacci search tree



GRAPH 1.11 Average time reqd. in successful search in a Fibonacci search tree



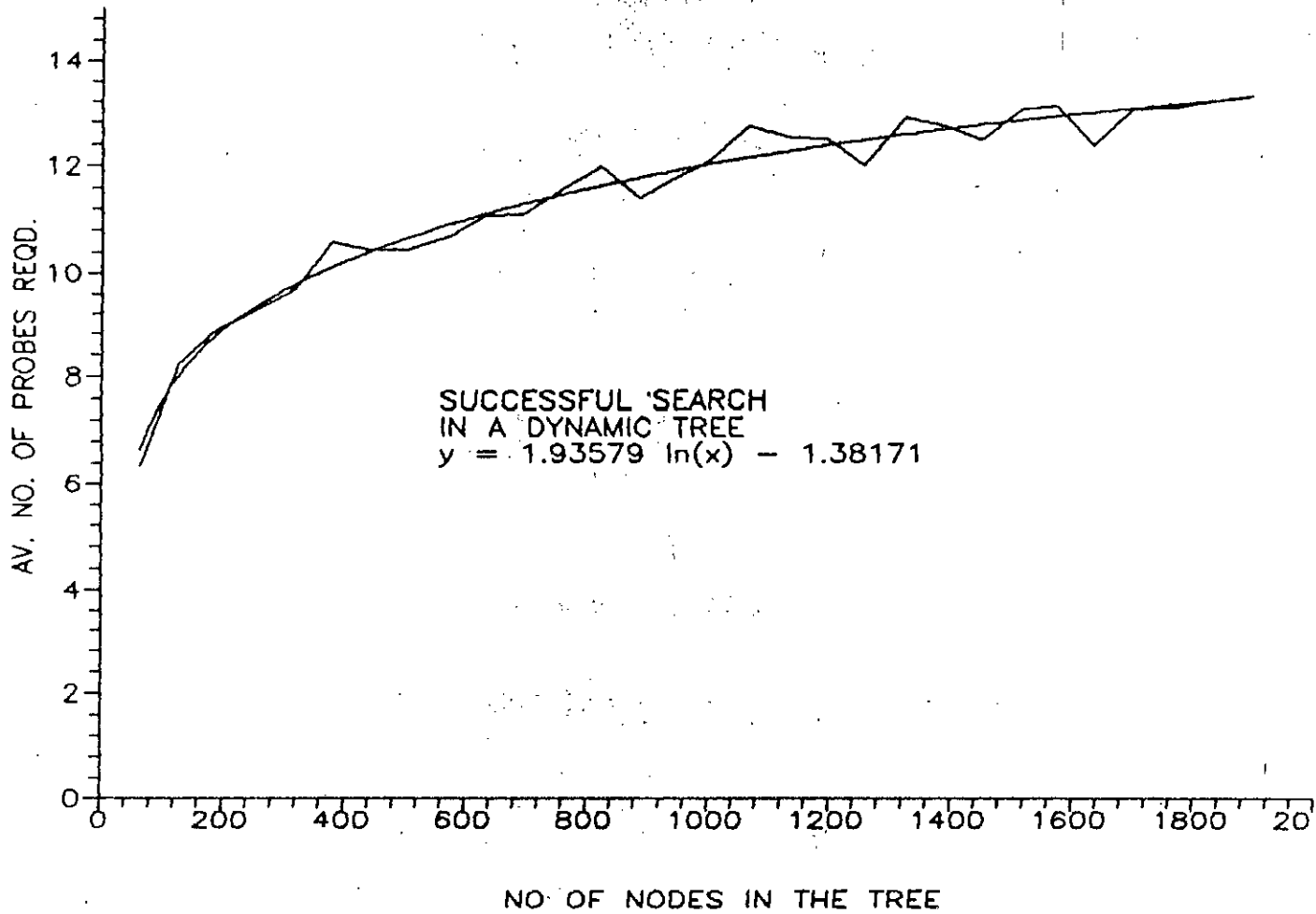
GRAPH 1.12 Average time reqd. in unsuccessful search in a Fibonacci search tree

and after a successful construction of such a tree sufficient number of trials have been taken to count the following measures:

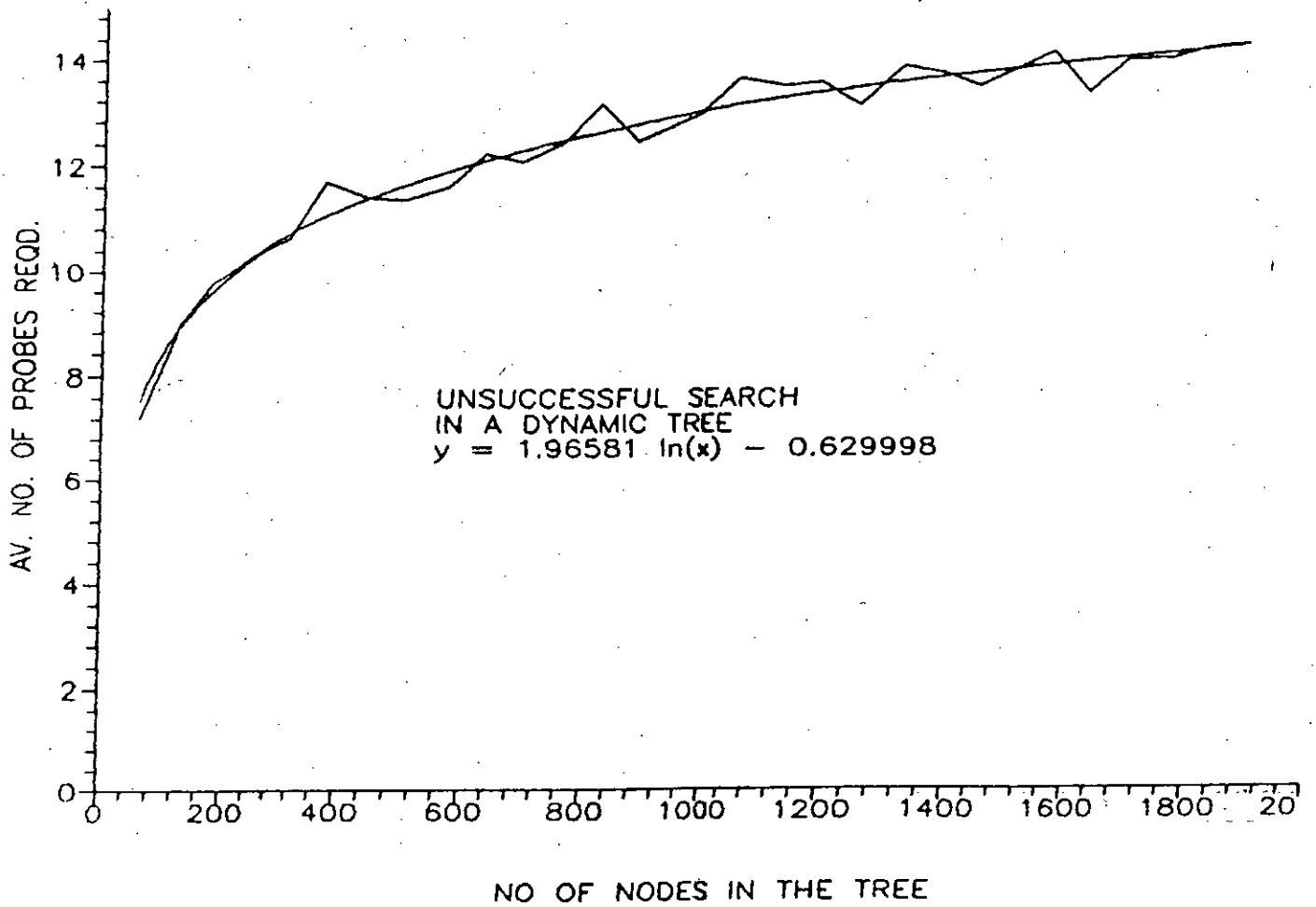
- (i) Average number of probes required in a successful search.
- (ii) Average number of probes required in an unsuccessful search.
- (iii) Average time required for a successful search and
- (iv) Average time required for an unsuccessful search.

Each of the above average measures for a tree of certain size were averaged for ten sets of data in order to satisfy the requirement that the experimental data should reflect the true average behaviour of a dynamic tree. For time calculation the same procedure as adopted in chapter one has been reinforced. The curves in Graph 2.1 through 2.4 depict the average behaviour of a dynamic search tree built from random keys. The average external path length of a dynamic tree is approximately $1.38 \log_2 n$ as derived in section 2.3. This means that the average number of probes required in a successful search is very close to this figure, and the experimental curves bear a close resemblance with this fact.

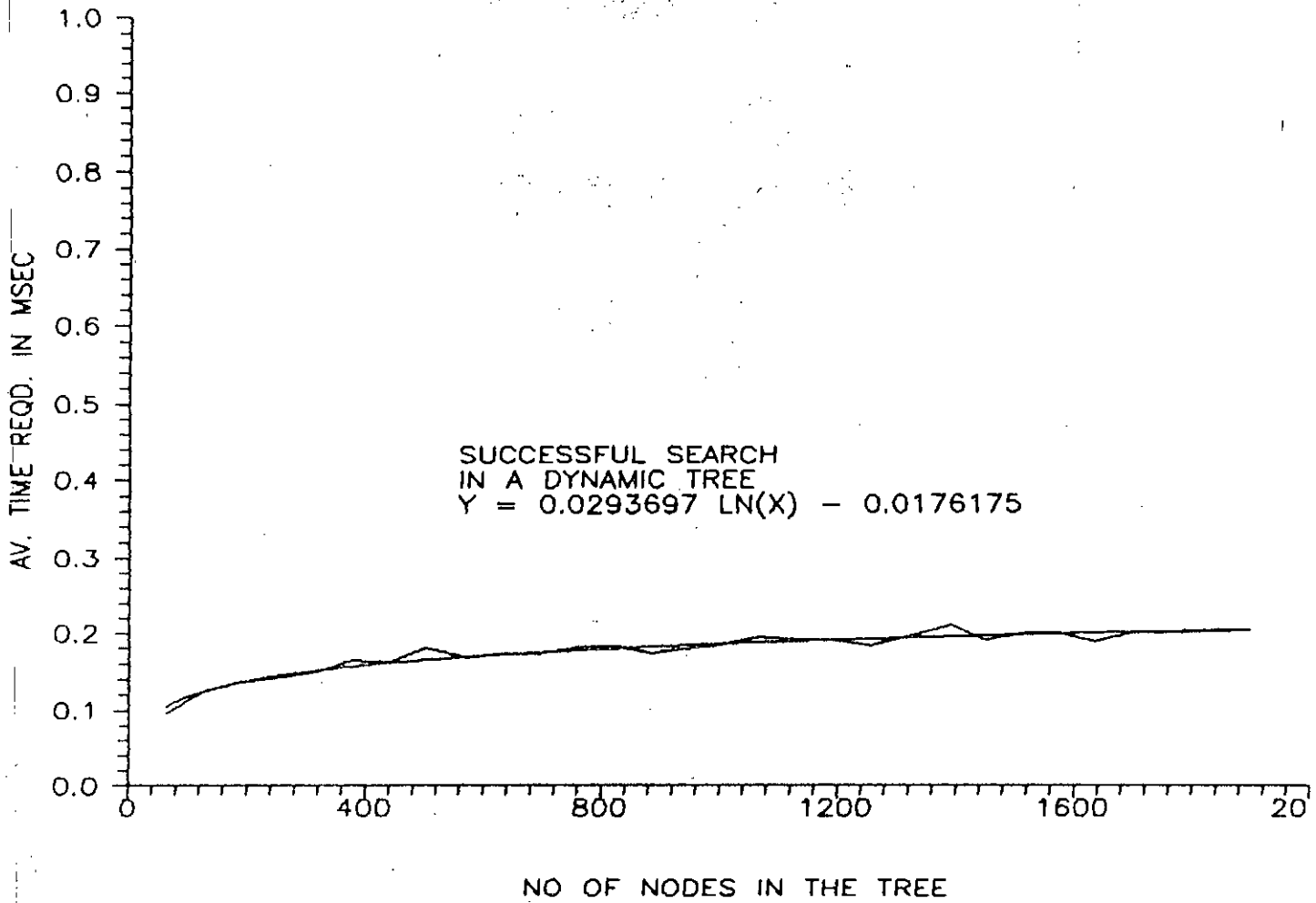
Next comes the question of random deletions from a dynamic search tree. From the discussions of chapter two it is evident that the average behaviour of a search tree after random deletions cannot be predicted to be the same as can be obtained if the tree is built afresh from random keys. The tree structure deviates slightly when random deletions are made as verified by the experimental curves of Graph 2.5 and 2.6. There is no known analysis of the average search time when random deletions and then insertions are made into a dynamic tree. We tried to investigate the average behaviour of a dynamic tree after random deletions and insertions are made. The experimental results show that as soon as random insertions are made into a dynamic tree having



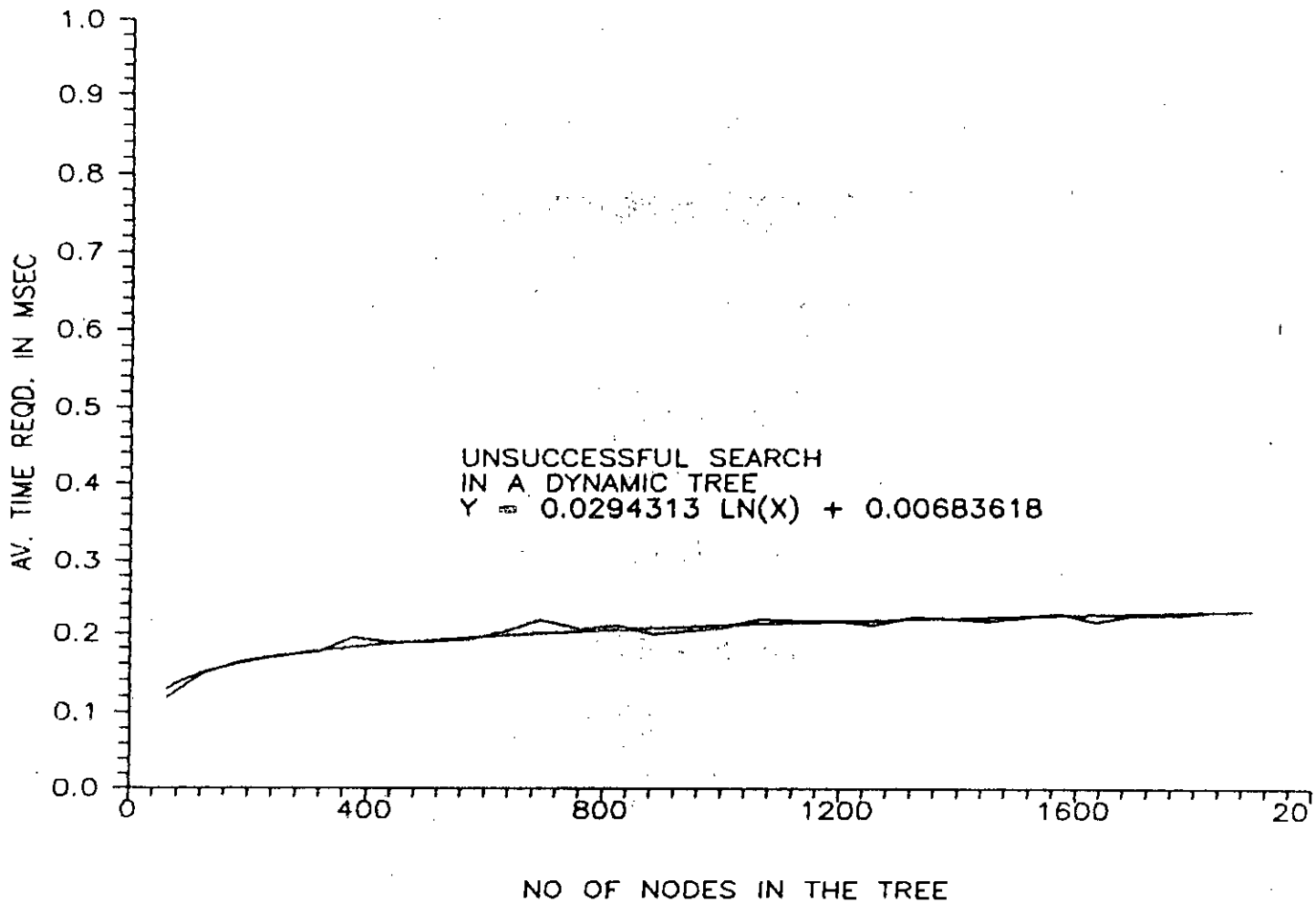
GRAPH 2.1 Average no. of probes reqd. in successful search in a Dynamic search tree



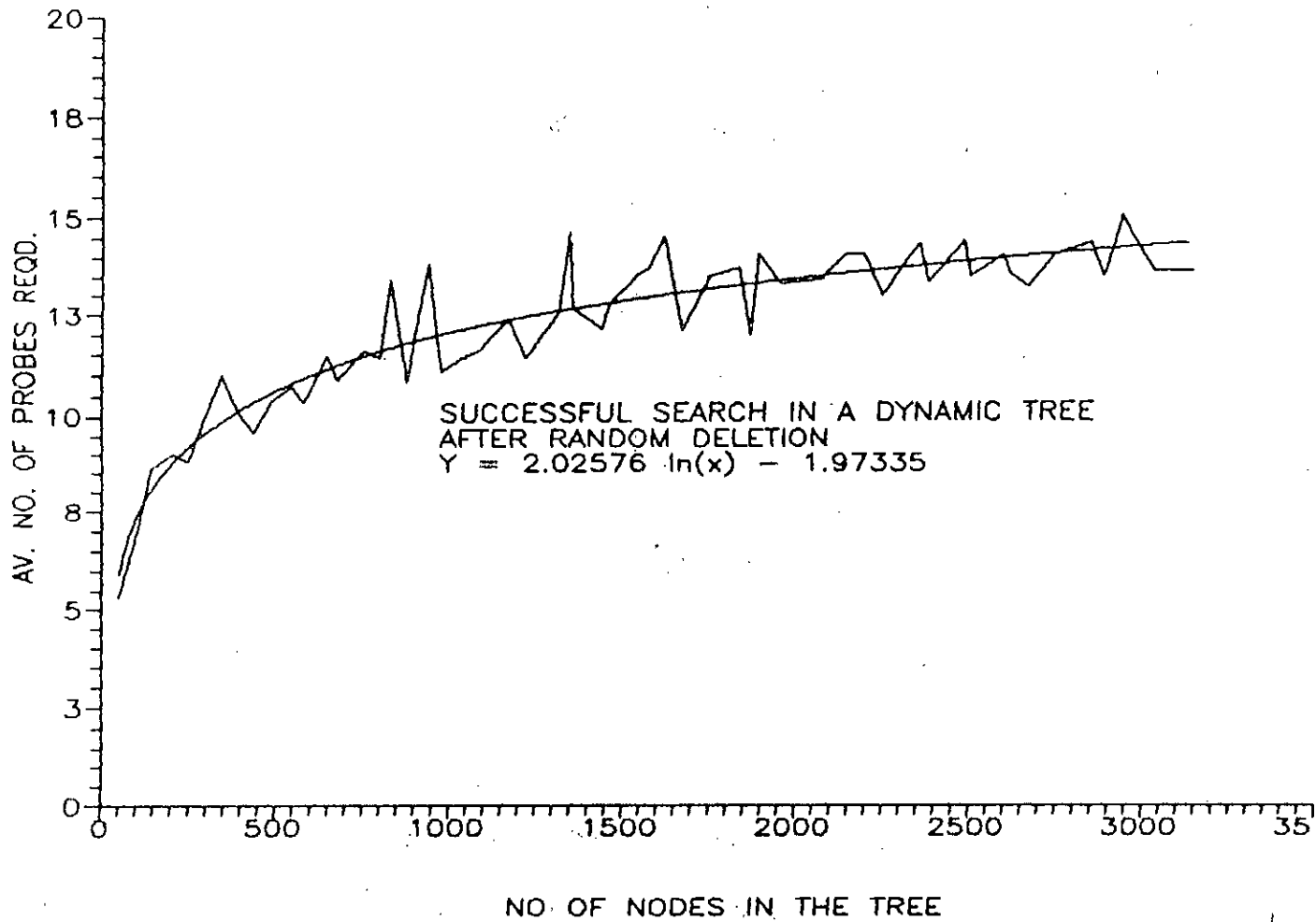
GRAPH 2.2 Average no. of probes reqd. in unsuccessful search in a Dynamic search tree



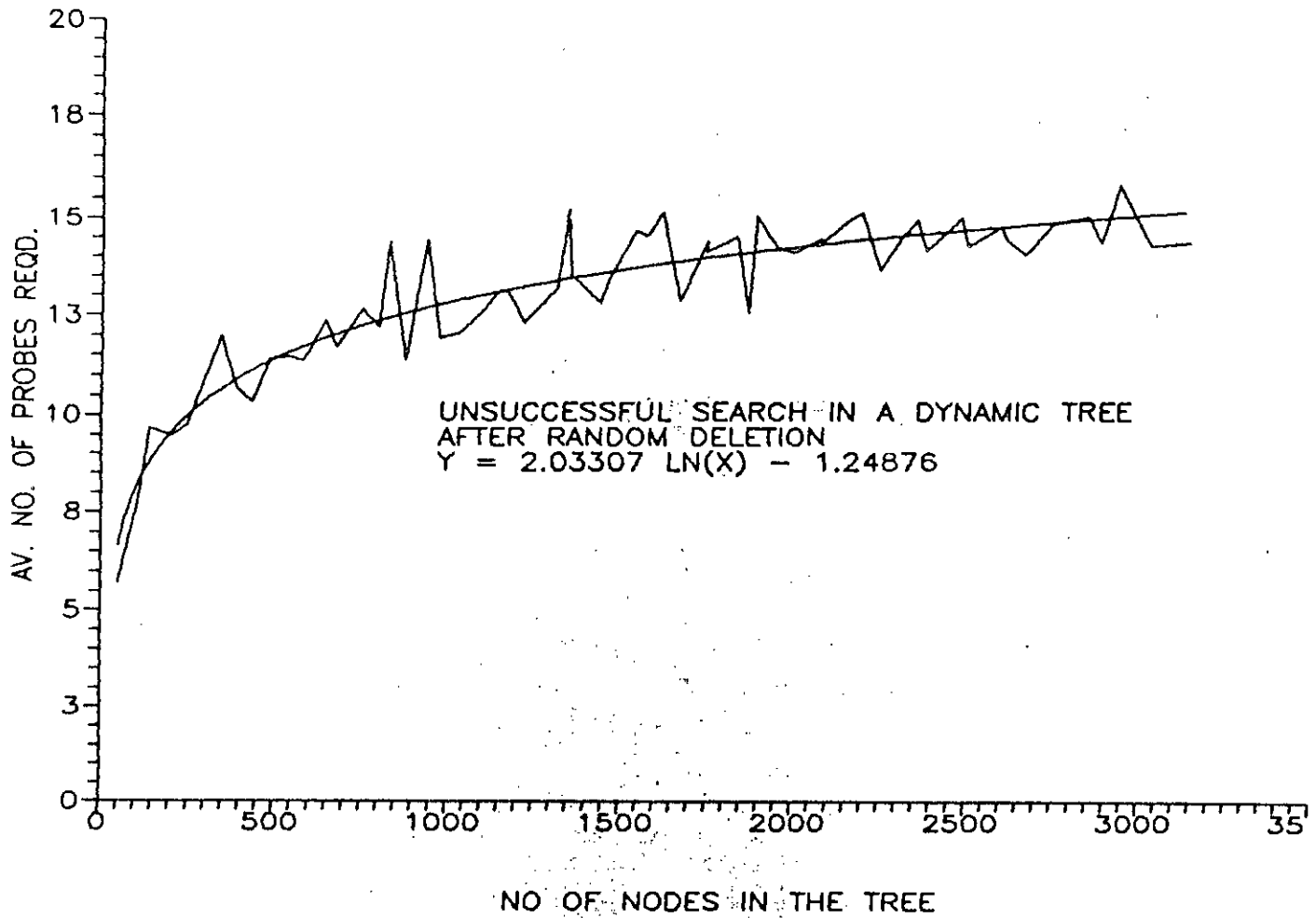
GRAPH 2.3 Average time reqd. in successful search in a Dynamic search tree



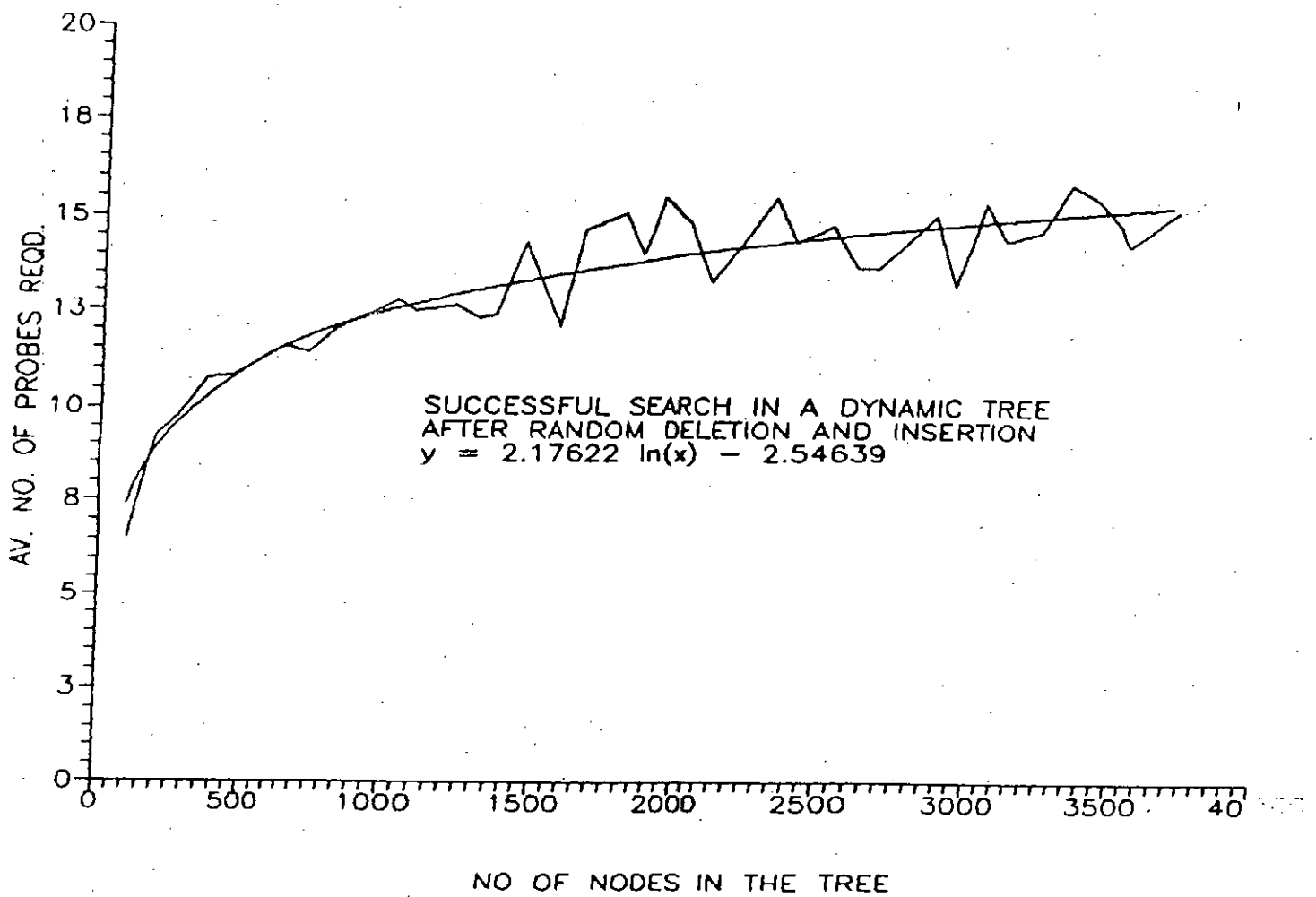
GRAPH 2.4 Average time reqd. in unsuccessful search in a Dynamic search tree



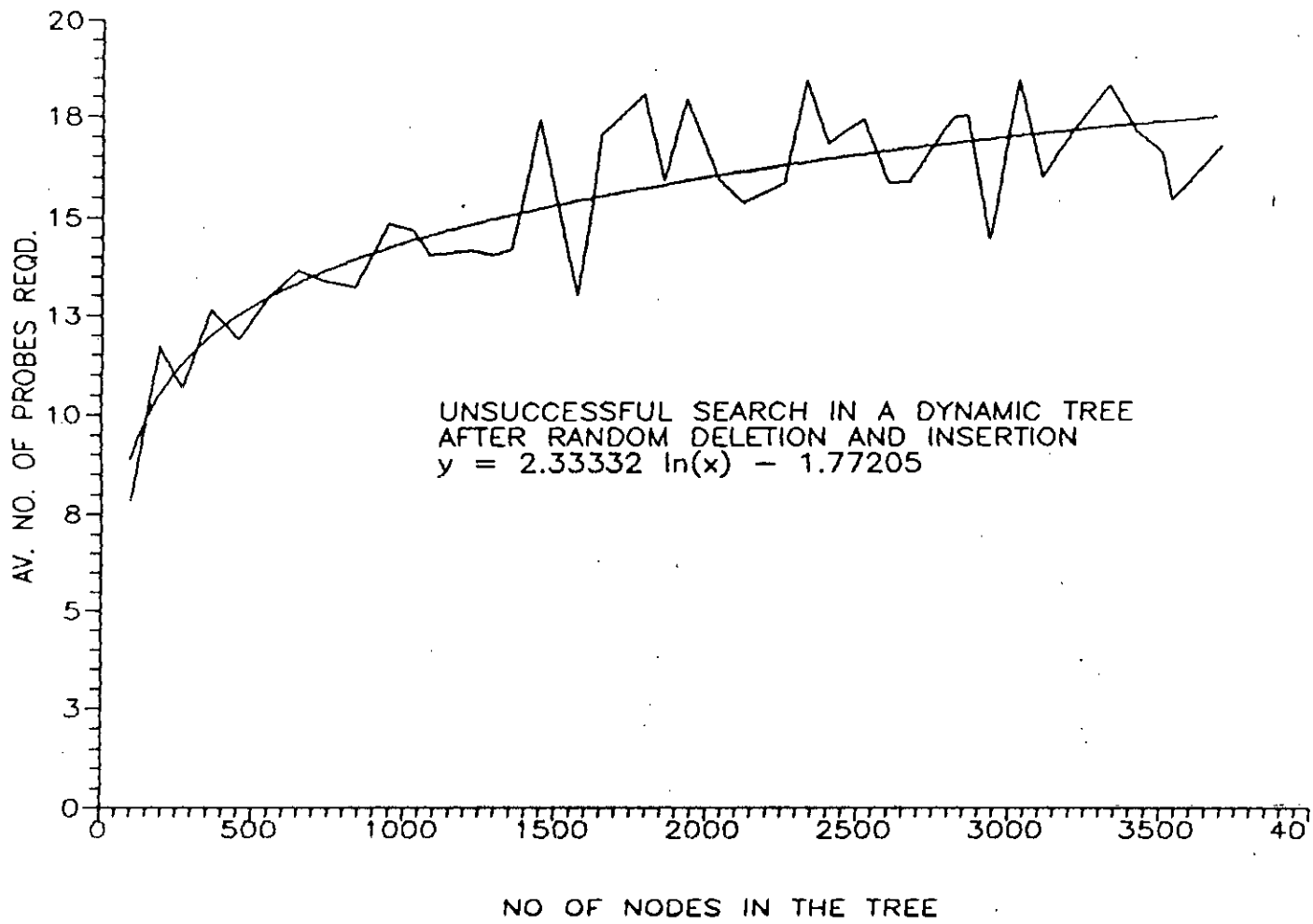
GRAPH 2.5 Average no. of probes reqd. in successful search in a Dynamic search tree after random deletions



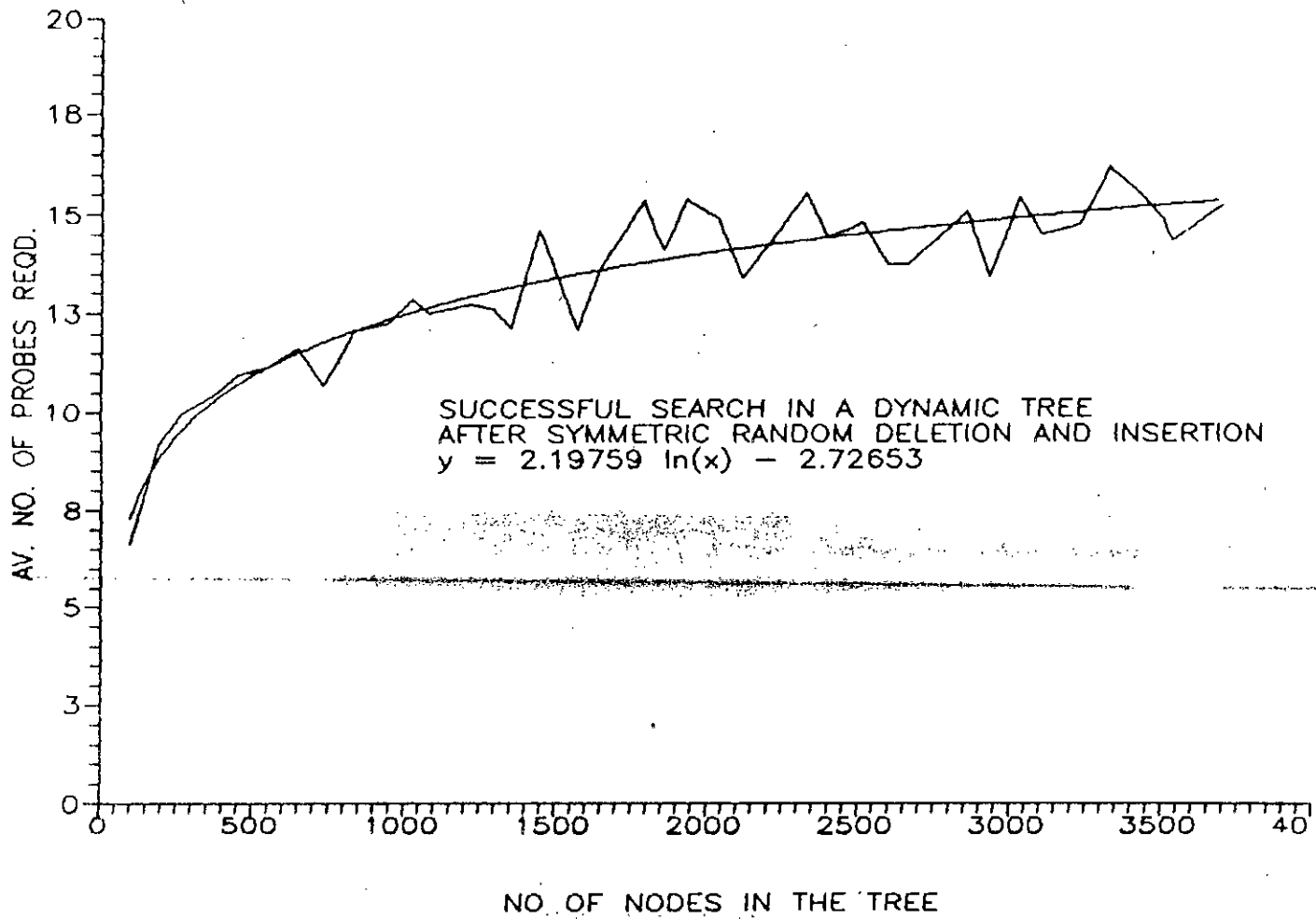
GRAPH 2.6 Average no. of probes reqd. in unsuccessful search in a Dynamic search tree after random deletions



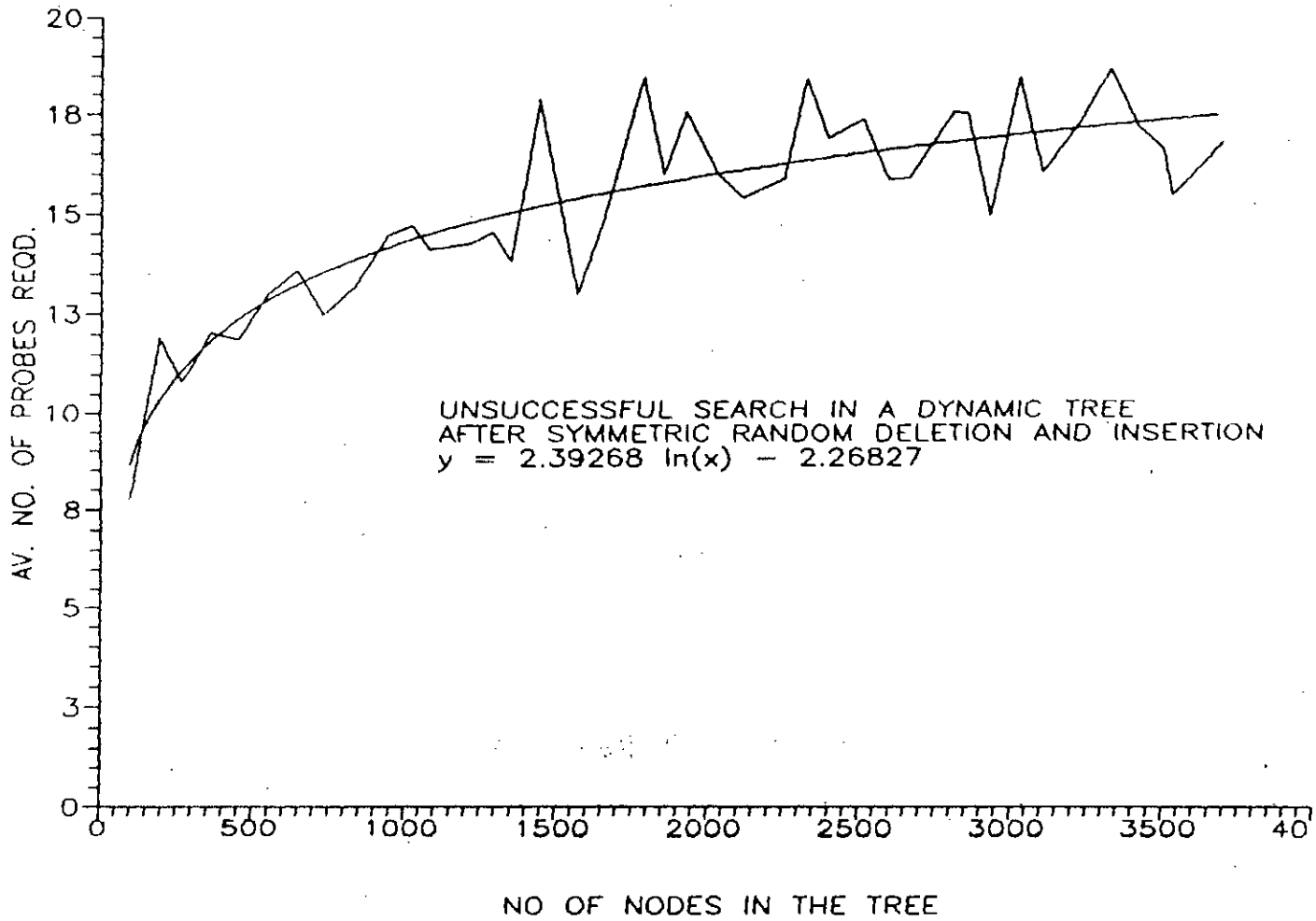
GRAPH 2.7 Average no. of probes reqd. in successful search in a Dynamic search tree after random deletions and insertions



GRAPH 2.8 Average no. of probes reqd. in unsuccessful search in a Dynamic search tree after random deletions and insertions



GRAPH 2.9 Average no. of probes reqd. in successful search in a Dynamic search tree after symmetric random deletions and insertions



GRAPH 2.10 Average no. of probes reqd. in unsuccessful search in a Dynamic search tree after symmetric random deletions and insertions

experienced a number of random deletions, the tree structure sharply deteriorates, and the average behaviour becomes worse as the curves of Graph 2.7 through 2.10 show. The theoretical analysis of this worse behaviour is still lacking but this can be inferred from the fact that random deletions change the relative distribution of values of a given tree shape and a random insertion after a number of random deletions destroys the randomness property of the tree.

A node can be deleted from a tree either by replacing it with its predecessor or by its successor. If only successor or predecessors are chosen for replacement the deletion process can be termed as asymmetric. But if predecessor and successor are chosen alternately for replacement then the deletion process can be termed as symmetric deletion. The curves of Graph 2.7 through 2.10 show that symmetric deletion causes no improvement in performance.

The rest of chapter two beginning from section 2.4 describes the procedure for building a binary search tree that will have an optimal weighted path length given the frequencies p_i and q_i . Algorithm 2.3 with the appropriate modification as enlightened at the very end of section 2.4 has been employed to construct the optimal search tree of Fig. 5.1. The frequencies of access of each element have been shown in this figure. The tree of Fig. 5.2 results when all the external frequencies are made zero and that of Fig. 5.3 results when all the internal frequencies are made zero. The trees of Fig. 5.2 and 5.3 show that both internal and external frequencies influence the tree structure by a considerable amount. The average cost of the trees are 4.147506, 2.991565 and 4.436492 respectively. Table 5.1, 5.2 and 5.3 shows the average cost for different selection of the roots for the trees of figure 5.1, 5.2 and 5.3 respectively. These tables show that the roots of the optimal trees give the minimum cost as expected since the optimality has been

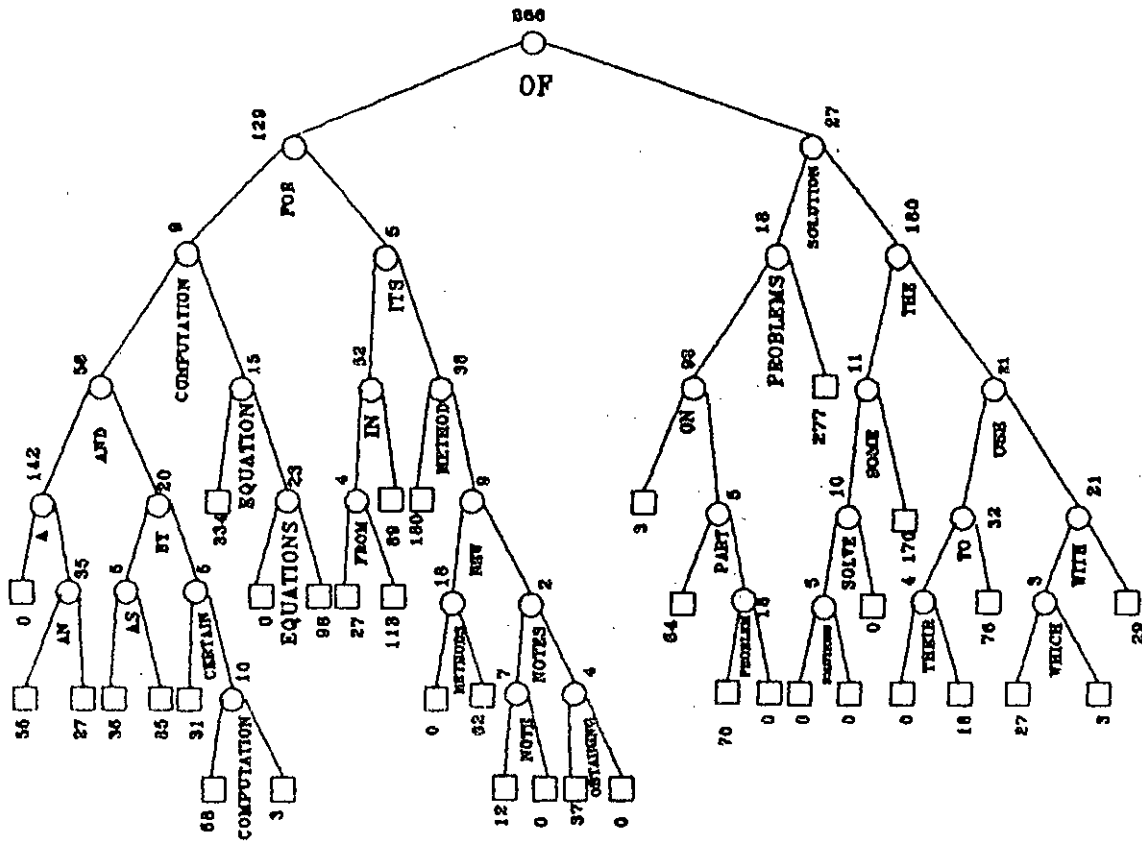


Fig. 5.1 An optimal Binary search tree

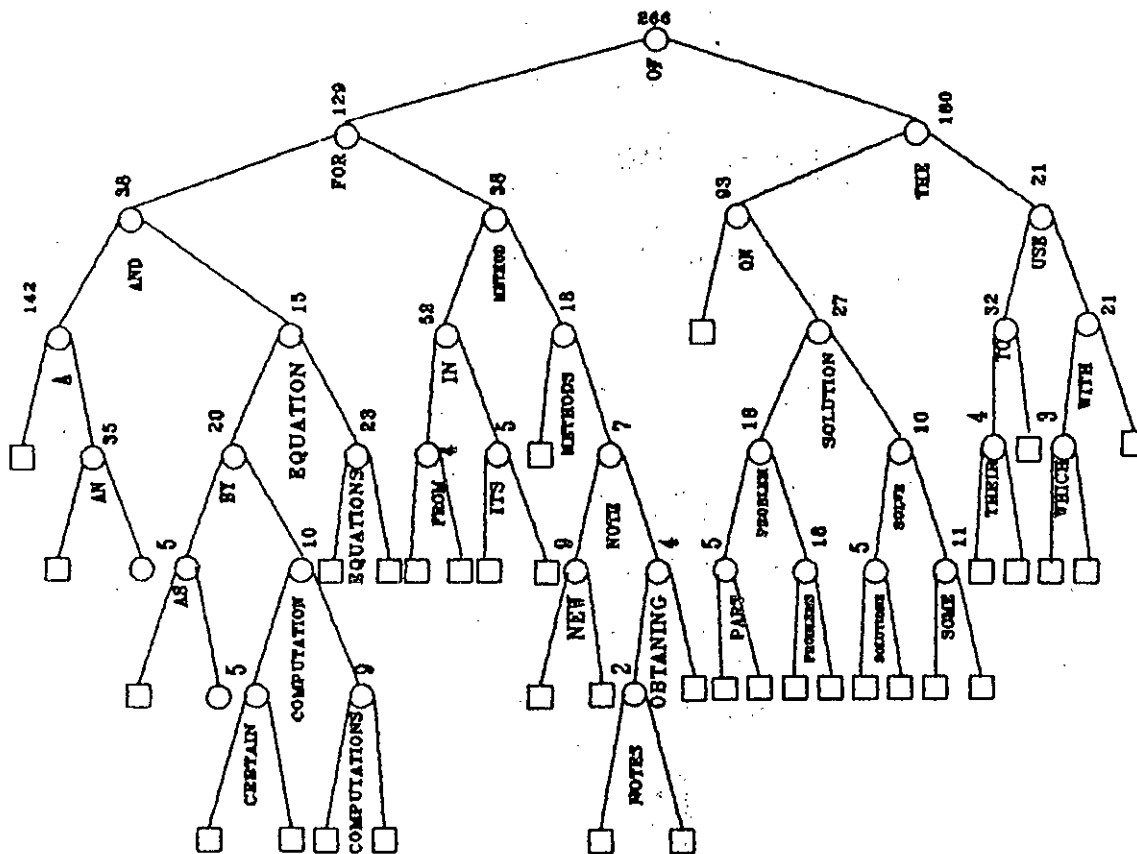


Fig. 5.2 Optimal tree of Fig. 5.1 with all external frequencies equal to zero

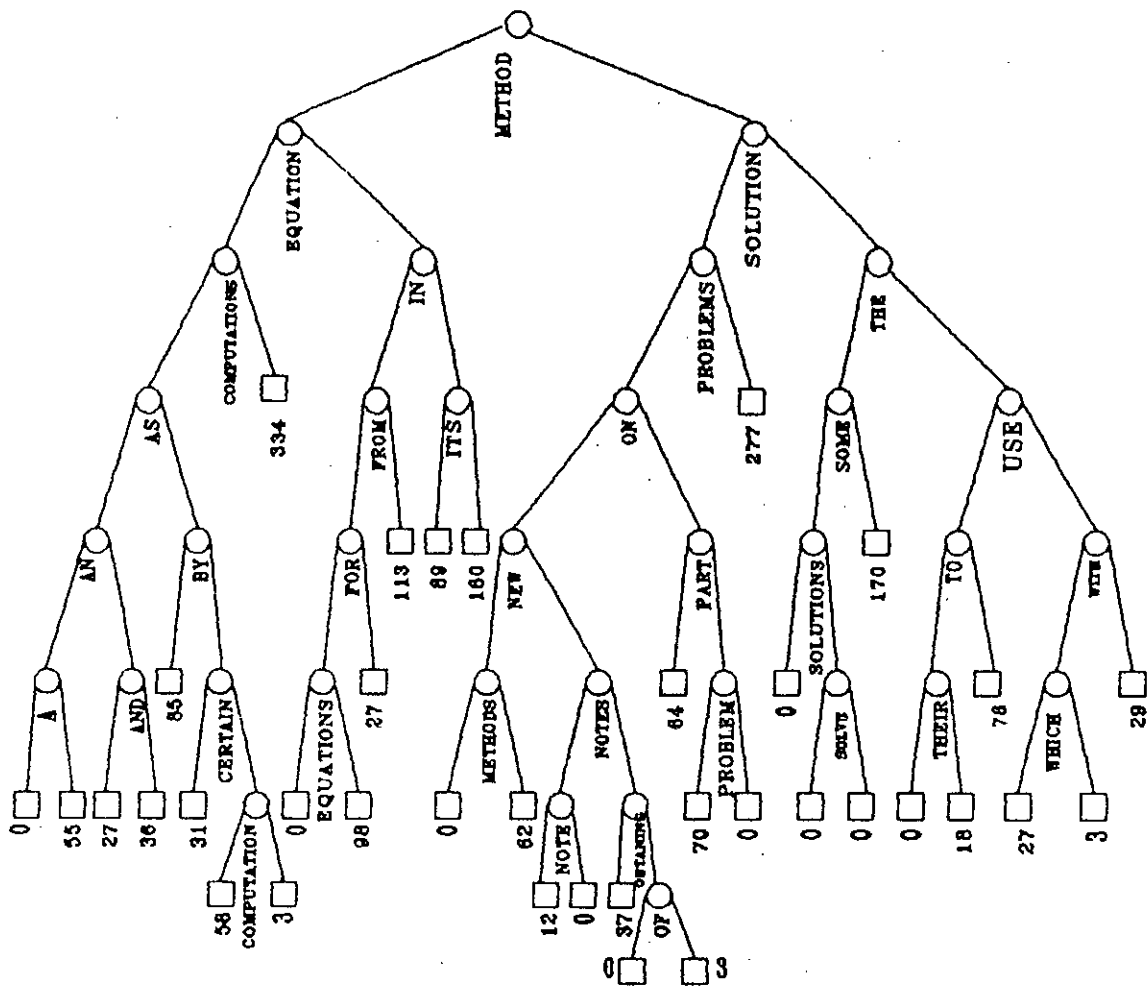


Fig. 5.3 Optimal tree of Fig. 5.1 with all internal frequencies equal to zero

ROOT	MINIMUM AVERAGE COST	INTERNAL FREQUENCY	EXTERNAL FREQUENCY
1	4.895986	142	55
2	4.796533	35	27
3	4.684915	58	36
4	4.686436	5	85
5	4.603406	20	31
6	4.596411	5	58
7	4.559002	10	3
8	4.476277	9	334
9	4.343370	15	0
10	4.396289	23	98
11	4.230536	129	27
12	4.337591	4	113
13	4.281630	52	89
14	4.324209	5	180
15	4.325426	38	0
16	4.377433	18	62
17	4.391119	9	12
18	4.417275	7	0
19	4.417883	2	37
20	4.433090	4	0
21	4.147506	266	3
22	4.330900	93	64
23	4.429440	5	70
24	4.433090	18	0
25	4.381083	18	277
26	4.482664	27	0
27	4.593674	5	0
28	4.592153	10	0
29	4.550487	11	170
30	4.652677	180	0
31	4.924878	4	18
32	4.898114	32	76
33	4.981448	21	27
34	5.058394	3	3
35	5.061436	21	29

TABLE 5.1

ROOT	MINIMUM AVERAGE COST	INTERNAL FREQUENCY	EXTERNAL FREQUENCY
1	3.529141	142	0
2	3.522239	35	0
3	3.384202	58	0
4	3.480828	5	0
5	3.412577	20	0
6	3.443252	5	0
7	3.421012	10	0
8	3.421779	9	0
9	3.410276	15	0
10	3.414877	23	0
11	3.193252	129	0
12	3.445552	4	0
13	3.331288	52	0
14	3.440951	5	0
15	3.368865	38	0
16	3.444785	18	0
17	3.483896	9	0
18	3.490798	7	0
19	3.511503	2	0
20	3.511503	4	0
21	2.991565	266	0
22	3.291411	93	0
23	3.490031	5	0
24	3.450153	18	0
25	3.467791	18	0
26	3.475460	27	0
27	3.570552	5	0
28	3.570552	10	0
29	3.598926	11	0
30	3.447853	180	0
31	3.858129	4	0
32	3.818252	32	0
33	3.860430	21	0
34	3.920245	3	0
35	3.915644	21	0

TABLE 5.2

ROOT	MINIMUM AVERAGE COST	INTERNAL FREQUENCY	EXTERNAL FREQUENCY
1	5.408770	0	55
2	5.251512	0	27
3	5.193044	0	36
4	5.109375	0	85
5	4.993952	0	31
6	4.974798	0	58
7	4.959677	0	3
8	4.822077	0	334
9	4.509073	0	0
10	4.628024	0	98
11	4.524194	0	27
12	4.489415	0	113
13	4.450605	0	89
14	4.445565	0	180
15	4.436492	0	0
16	4.495968	0	62
17	4.471774	0	12
18	4.489415	0	0
19	4.476815	0	37
20	4.481351	0	0
21	4.498488	0	3
22	4.470766	0	64
23	4.525202	0	70
24	4.628024	0	0
25	4.523690	0	277
26	4.793851	0	0
27	4.933468	0	0
28	4.933468	0	0
29	4.847782	0	170
30	5.051411	0	0
31	5.128024	0	18
32	5.116935	0	76
33	5.249496	0	27
34	5.336190	0	3
35	5.342742	0	29

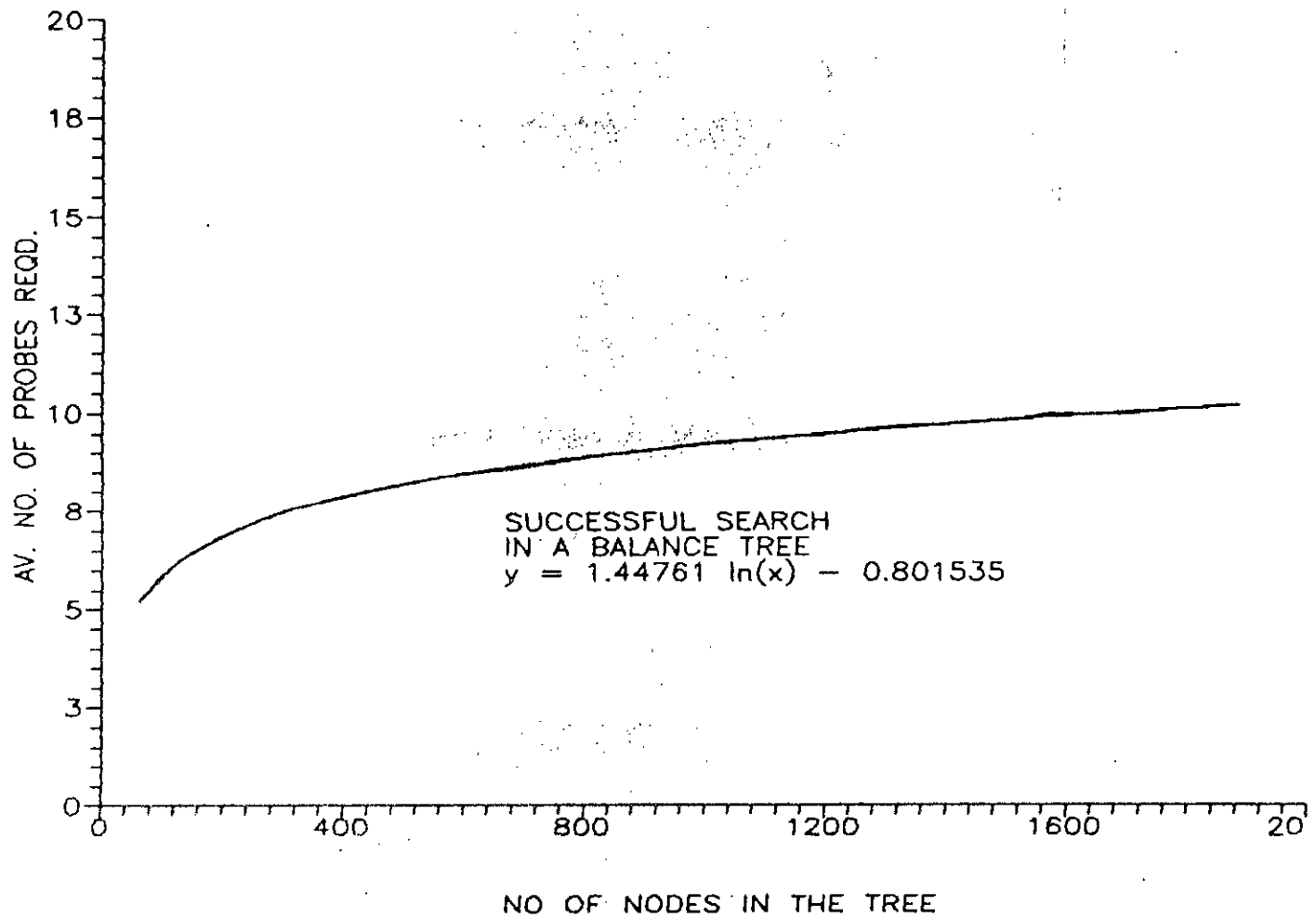
TABLE 5.3

defined in terms of minimum average cost. As calculated in section 2.4, the construction of the optimal search trees of Fig. 5.1, 5.2 and 5.3 required $O(n^2)$ operations for their construction.

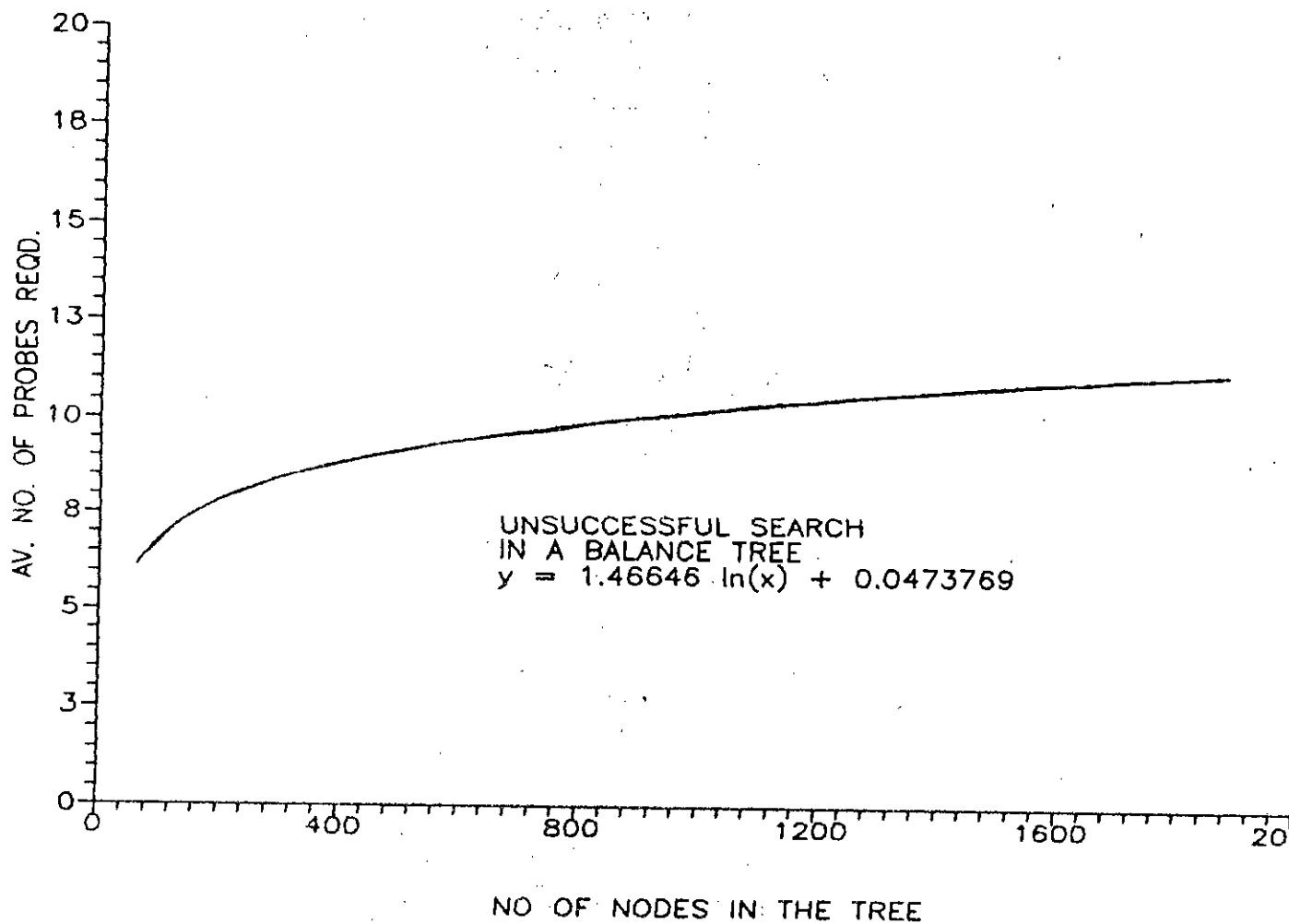
5.4 Balanced Trees.

The tree insertion algorithm discussed in chapter two will produce a good search tree when the input data is random, but there is still the annoying possibility that a degenerate tree will occur. The height of a balanced tree of n elements will be $O(\lg_2 n)$, so that search times are logarithmic, and insertion and deletions will require only local changes along a single path from the root to a leaf, requiring only time proportional to the height of the tree that is, $O(\lg_2 n)$. We have followed the same style in doing the experiments with balanced and 2-3-4 trees as done with dynamic trees in chapter two. Section 3.2.1 verifies the fact that in the worst case the number of probes required in a height balanced tree of n internal nodes will never be more than 45 percent higher than the optimum. As discussed in section (3.5), the fact that there are $n!$ possible orderings in which n keys can be inserted, and the perfectly balanced tree is obtained most of the times makes it extremely plausible that the average search time for a balanced tree is about $\lg_2 N + C$ comparisons for some small constant C .

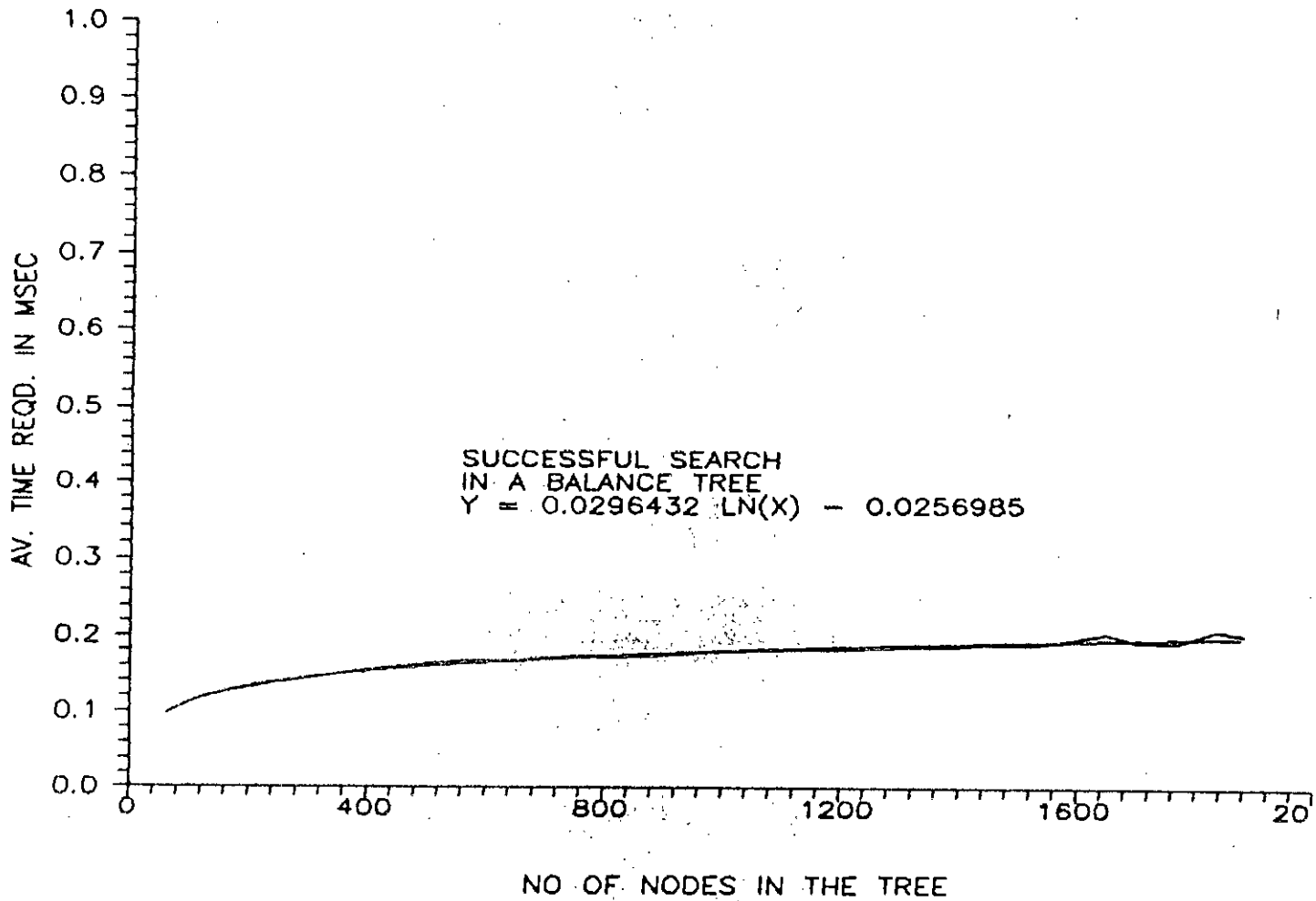
Experimental curves 3.1 and 3.2 support this conjecture. The timing curves shown in Graph 3.3 and 3.4 show the average time required for successful and unsuccessful search respectively. The real significance of balanced trees is their worse-case performance, and the fact that this performance is achieved at a very little cost. Experimental curves 3.5 show the percentage of times balancing required together with percentage of times single and double rotation required in times of insertion into a balanced tree. These figures support the fact that it is logical to construct balanced trees which has



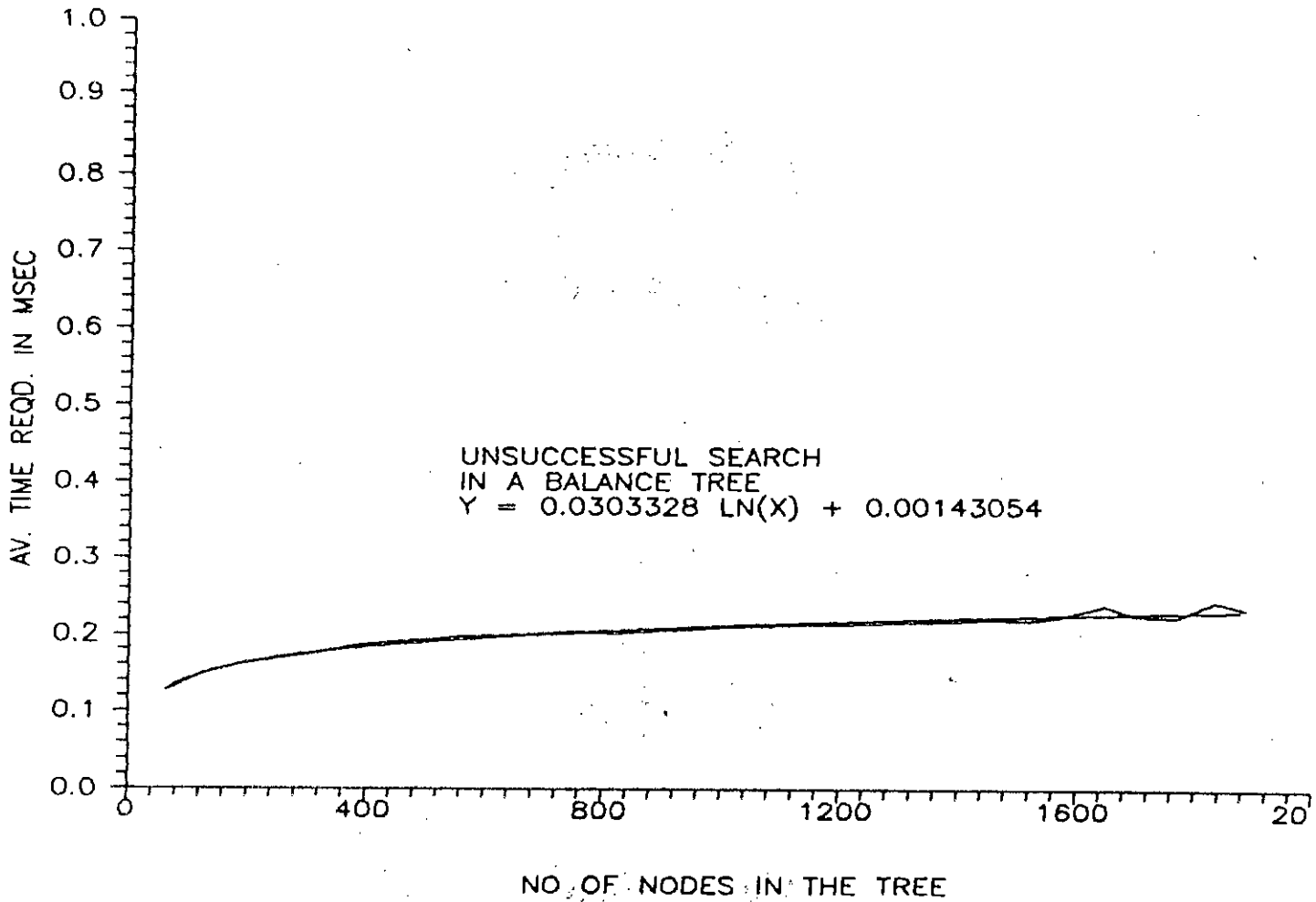
GRAPH 3.1 Average no. of probes reqd. in successful search in a Balanced tree



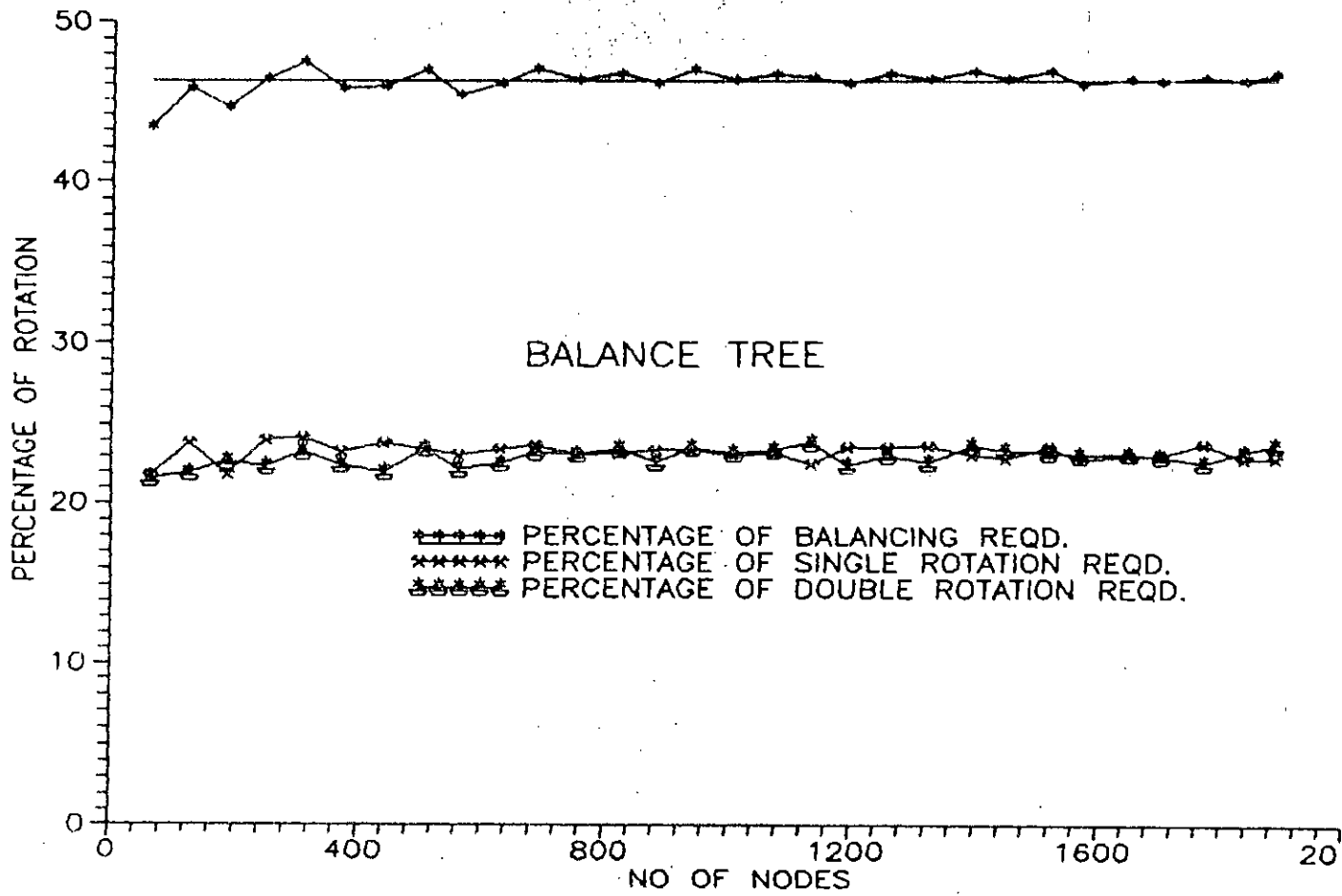
GRAPH 3.2 Average no. of probes reqd. in unsuccessful search in a Balanced tree



GRAPH 3.3 Average time reqd. in successful search in a Balanced tree



GRAPH 3.4 Average time reqd. in unsuccessful search in a Balanced tree



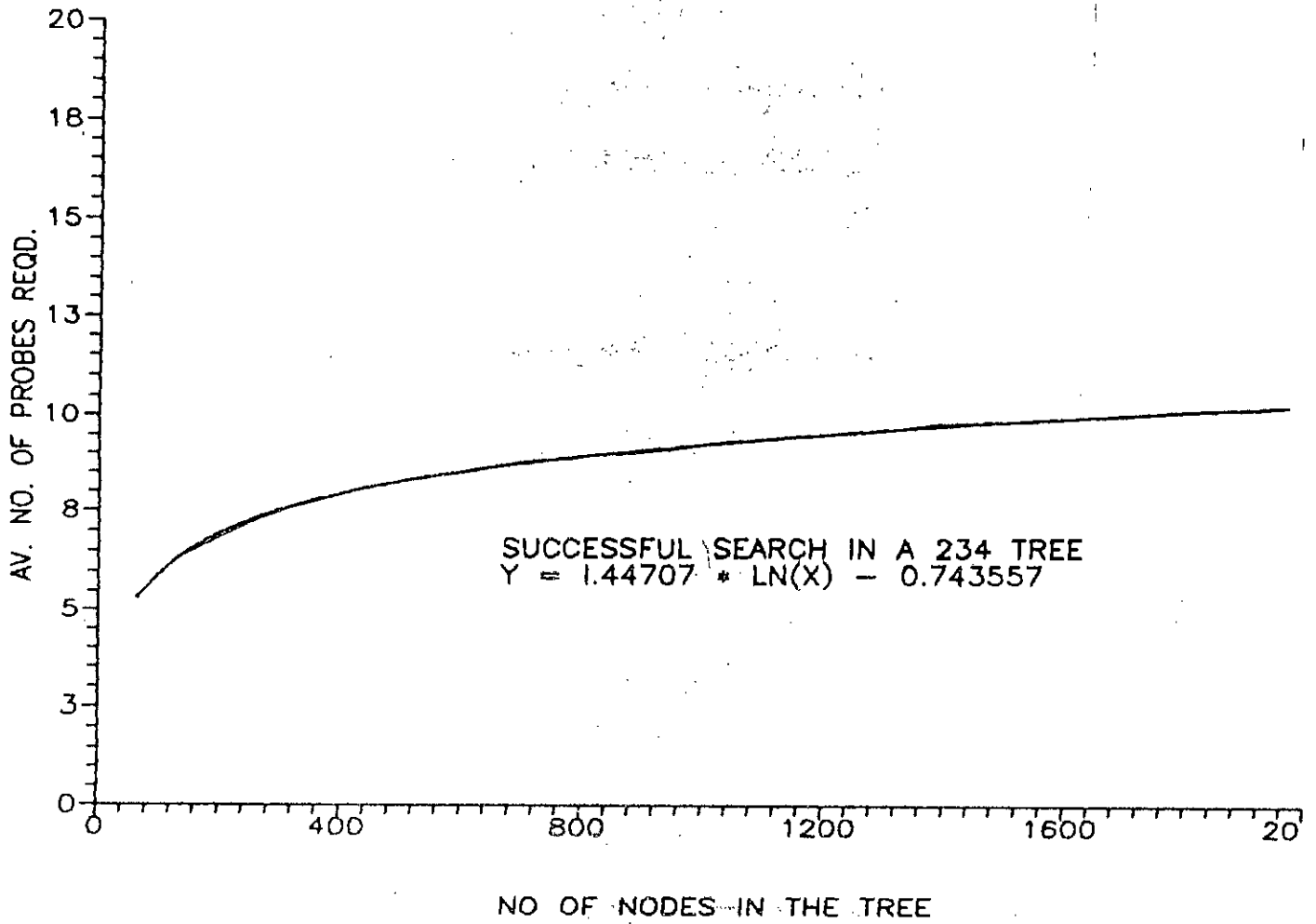
GRAPH 3.5 Percentage of rotation reqd. in a Balanced tree in times of insertion

guaranteed good worst-case performance because the insertion cost is not too high to discourage its use.

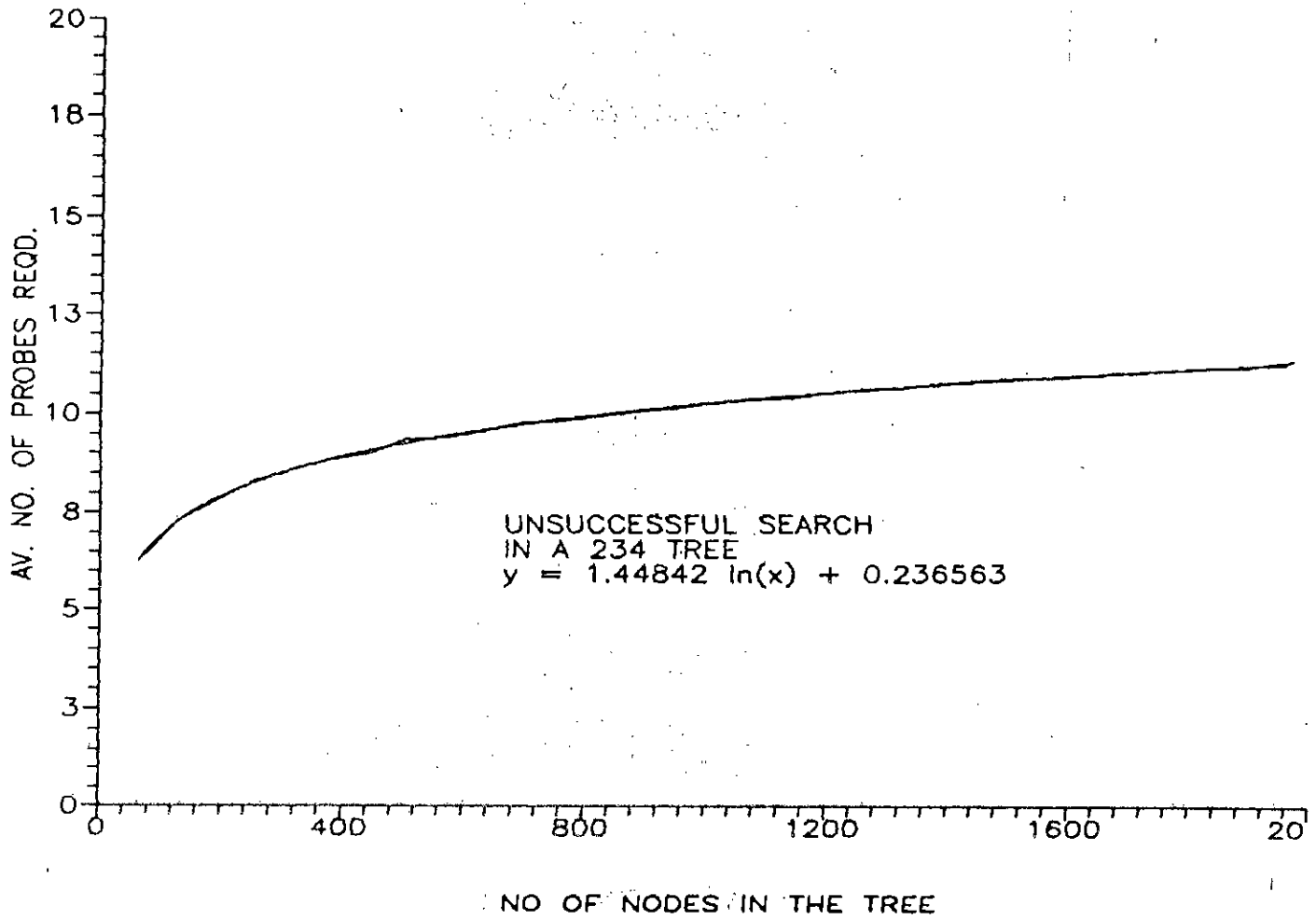
Next comes the 2-3-4 trees presented in section 3.6. The overhead incurred in manipulating the more complex 2-3-4 node structures in their direct representation is likely to make the algorithms slower than standard binary search. The primary purpose of using 2-3-4 trees is to provide insurance against a bad worst-case performance, but it would be unfortunate to have to pay the overhead cost for that insurance on every run of the algorithms. That is why we have represented 2-3-4 trees as standard binary trees by using one extra bit per node. The curves in graph 3.6 through 3.9 shows the average behaviour of 2-3-4 search tree. The behaviour is almost analogous to that of a balanced tree. But the behaviour of unsuccessful search is better for a 2-3-4 tree as verified by the experimental curves. The curves 3.10 shows the balancing requirement in times of insertion into a 2-3-4 search tree. The figures for single rotation and double rotation per insertion are significantly higher than those for balanced tree. But it should be noted that the algorithm to build a 2-3-4 search tree is much simpler than that of a balanced tree. Also a significant amount of overhead is required to adjust the balance factors along the search path in a balanced tree when a new node is inserted into it. Considering all these factors it can be concluded that total overhead requirement to insert a new node either into a balanced tree or into a 2-3-4 tree is almost identical. Since each key is inserted just once, but may be searched for many times in a typical application, the end result is that we get improved search times at relatively little cost.

5.5 Hashing Techniques.

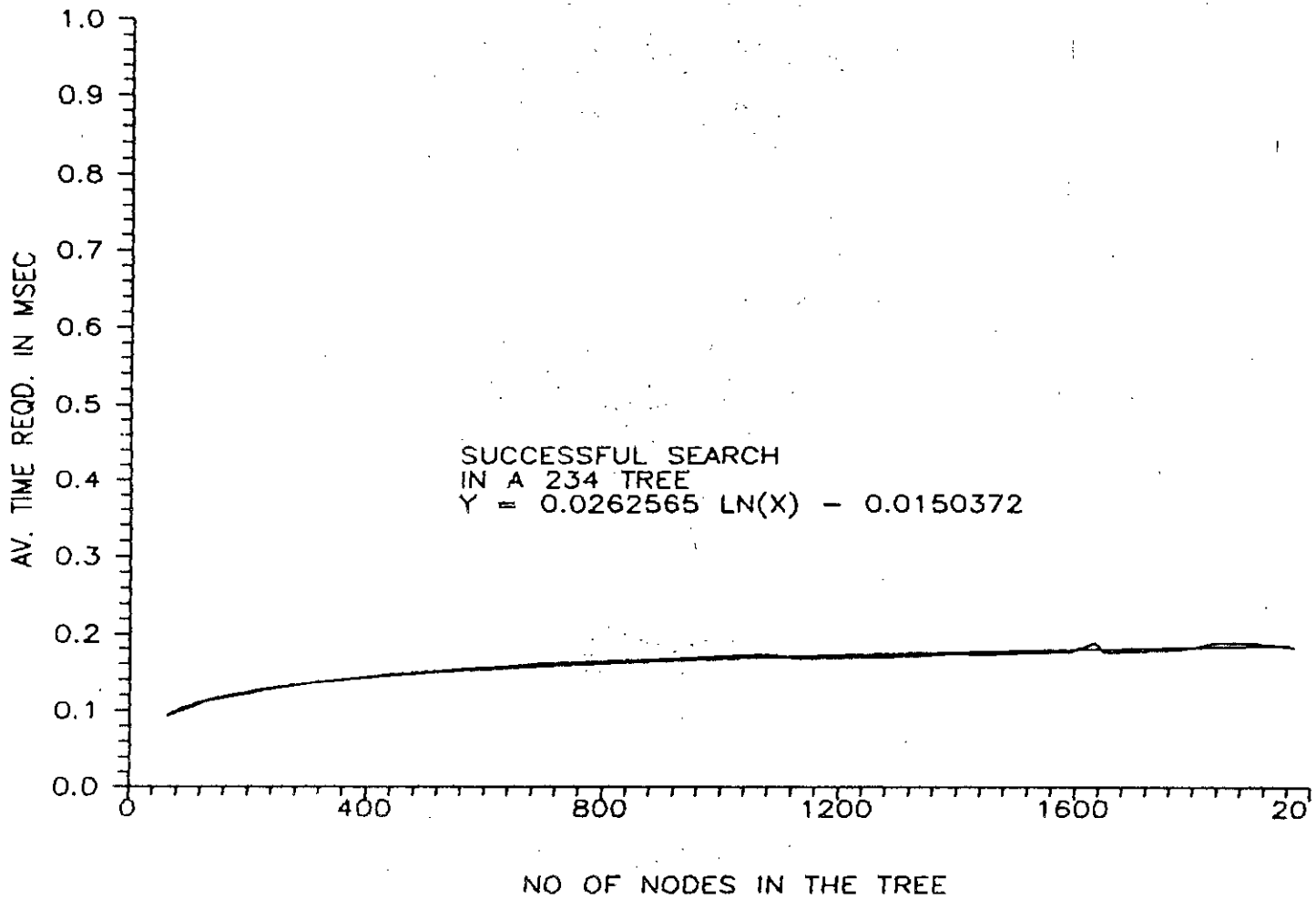
Chapter four deals with the various hashing techniques based on different collision resolution schemes. Collision resolution schemes can be classified into three categories



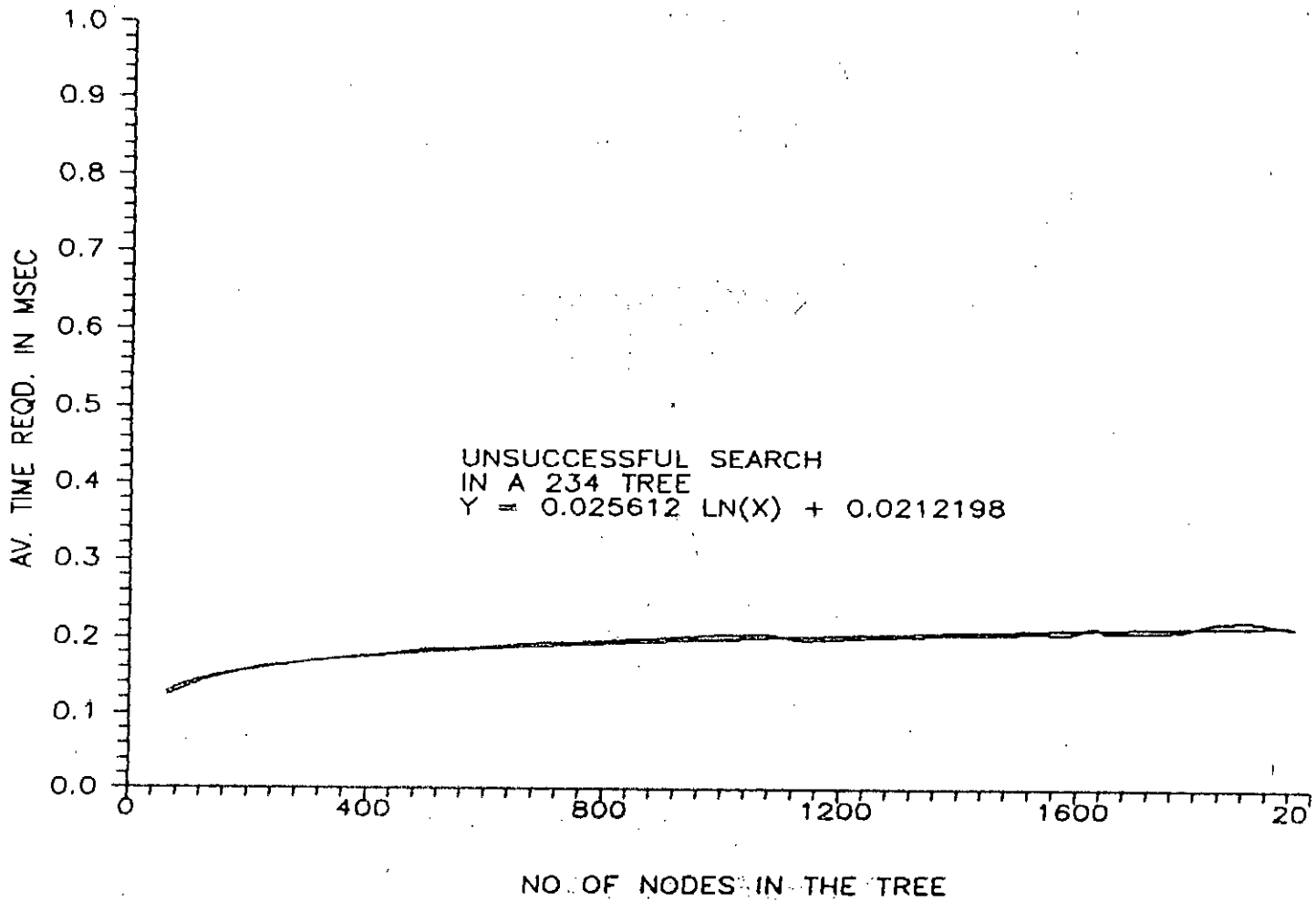
GRAPH 3.6 Average no. of probes reqd. in successful search in a 2-3-4 tree



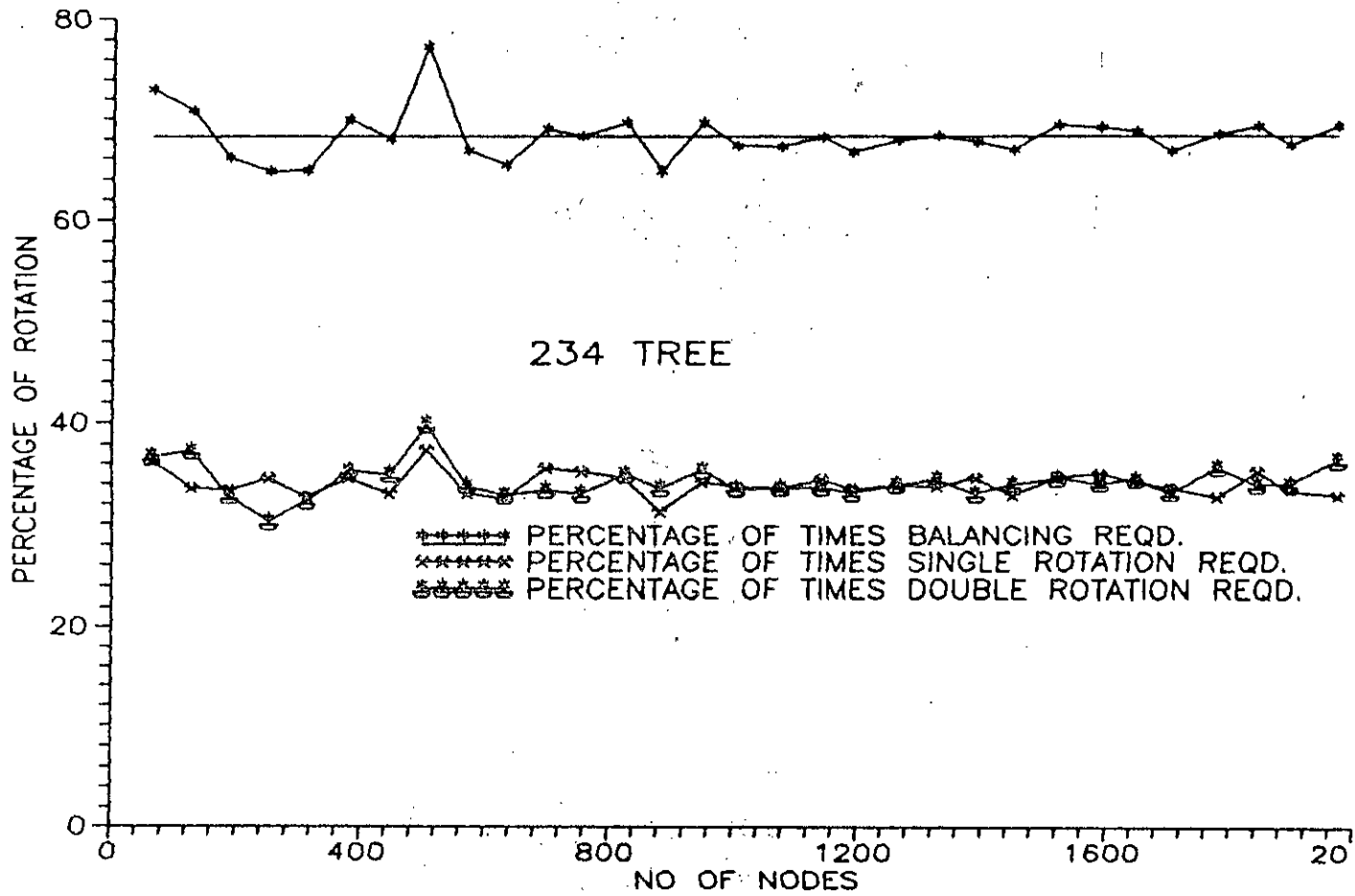
GRAPH 3.7 Average no. of probes reqd. in unsuccessful search in a 2-3-4 tree



GRAPH 3.8 Average time reqd. in successful search in a 2-3-4 tree



GRAPH 3.9 Average time reqd. in unsuccessful search in a 2-3-4 tree

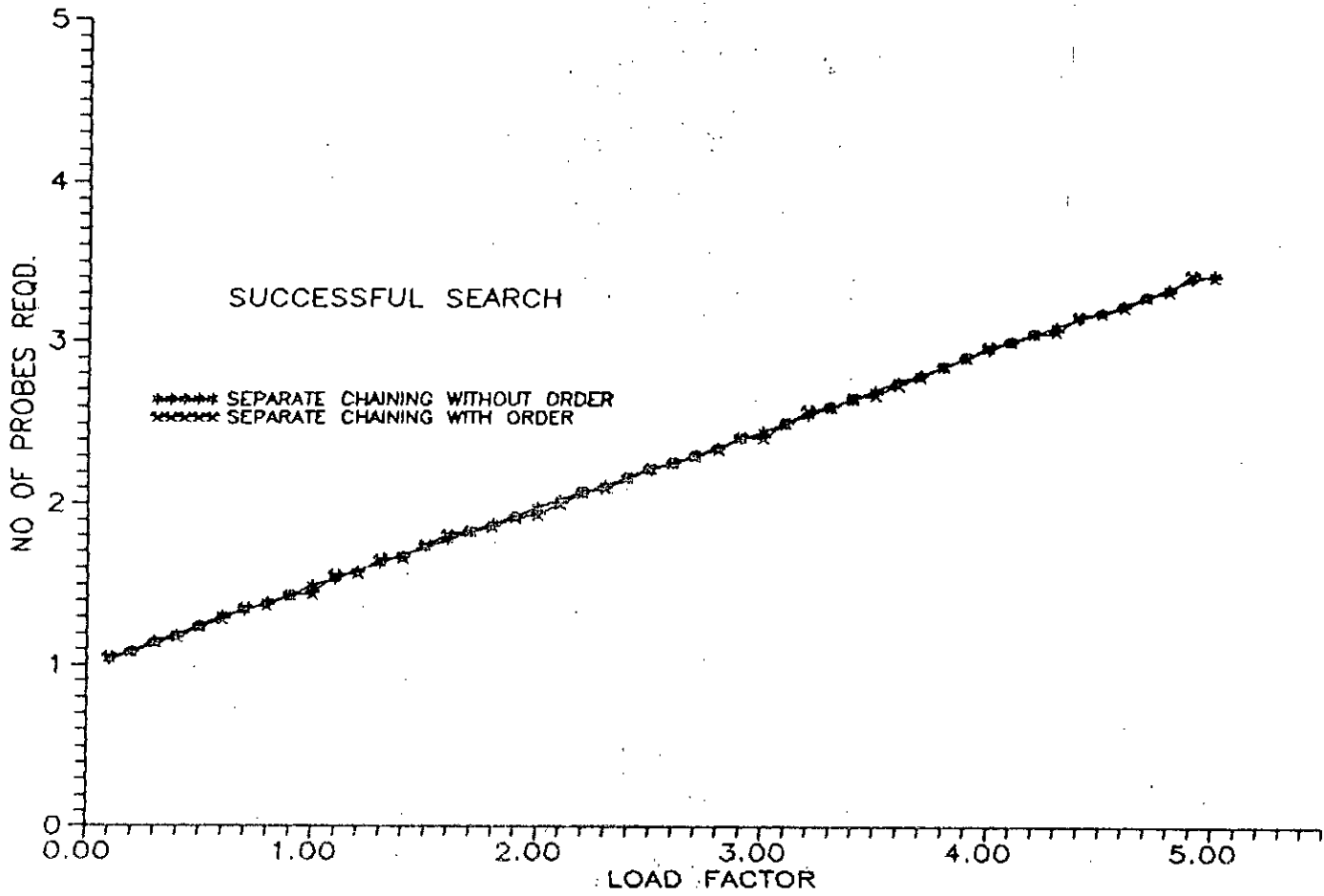


GRAPH 3.10 Percentage of rotation reqd. in a 2-3-4 tree in times of insertion

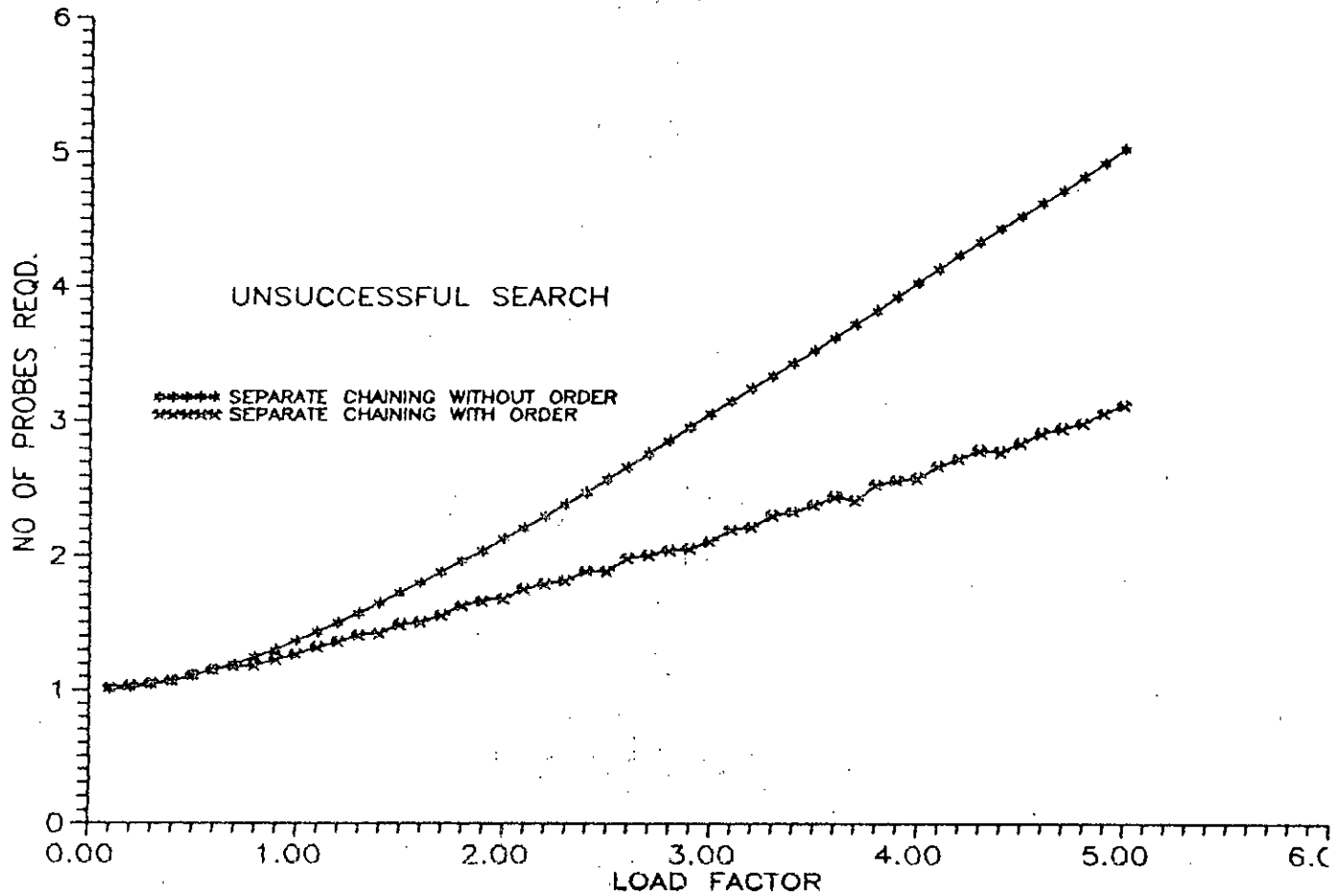
- (i) Chaining scheme,
- (ii) Linear open addressing,
- (iii) Double hashing

We have examined each of the above schemes along with variations thereof to improve certain properties of the corresponding hashing strategy.

A search table of length 1021 has been used to implement the algorithm presented in chapter four. Division hash function has been used to transform the keys into real addresses for simplicity and accuracy. Let us begin with the chaining method. Separate chaining is the natural starting point for the discussion of chaining method. As pointed out in section 4.3, if we keep each of the lists ordered by key, then time for unsuccessful searches can be reduced by a considerable amount. But since the keys come into random order, therefore time for successful search should not be affected by this variation. Graph 4.1 and 4.2 reflect these facts. Graph 4.2 shows that nearly any load factor convinces the use of ordered chain though each insertion in an ordered table is much costlier than that of in unordered chaining. Since insertion is a rare event in most practical situation, therefore, ordered chaining is preferable to unordered one specially when a large number of keys hashes into a short table, and unsuccessful searches are more common. In separate chaining, for the sake of speed we would like to make the no of list heads M rather large. But when M is large most of the lists will be empty and much of the space for the M list heads will be wasted. This suggests another approach named coalesced chaining which eliminates the overhead of the M list heads. Graph 4.7 and 4.8 compares the different collision resolution schemes. Here we see that separate chaining is faster than coalesced chaining for both successful and unsuccessful searches at any



GRAPH 4.1 Average no. of probes reqd. in successful search in separate chaining

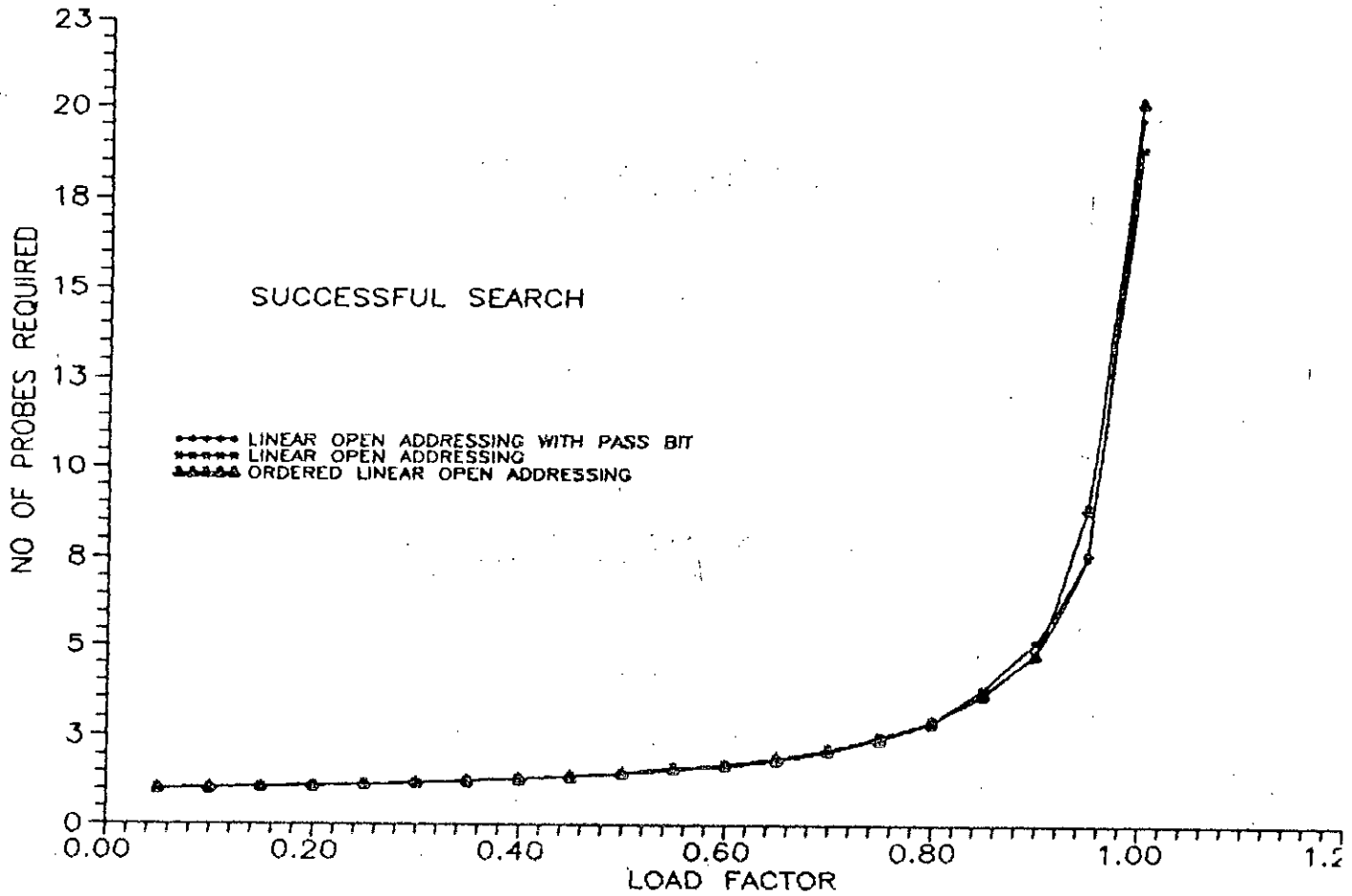


GRAPH 4.2 Average no. of probes reqd. in unsuccessful search in separate chaining

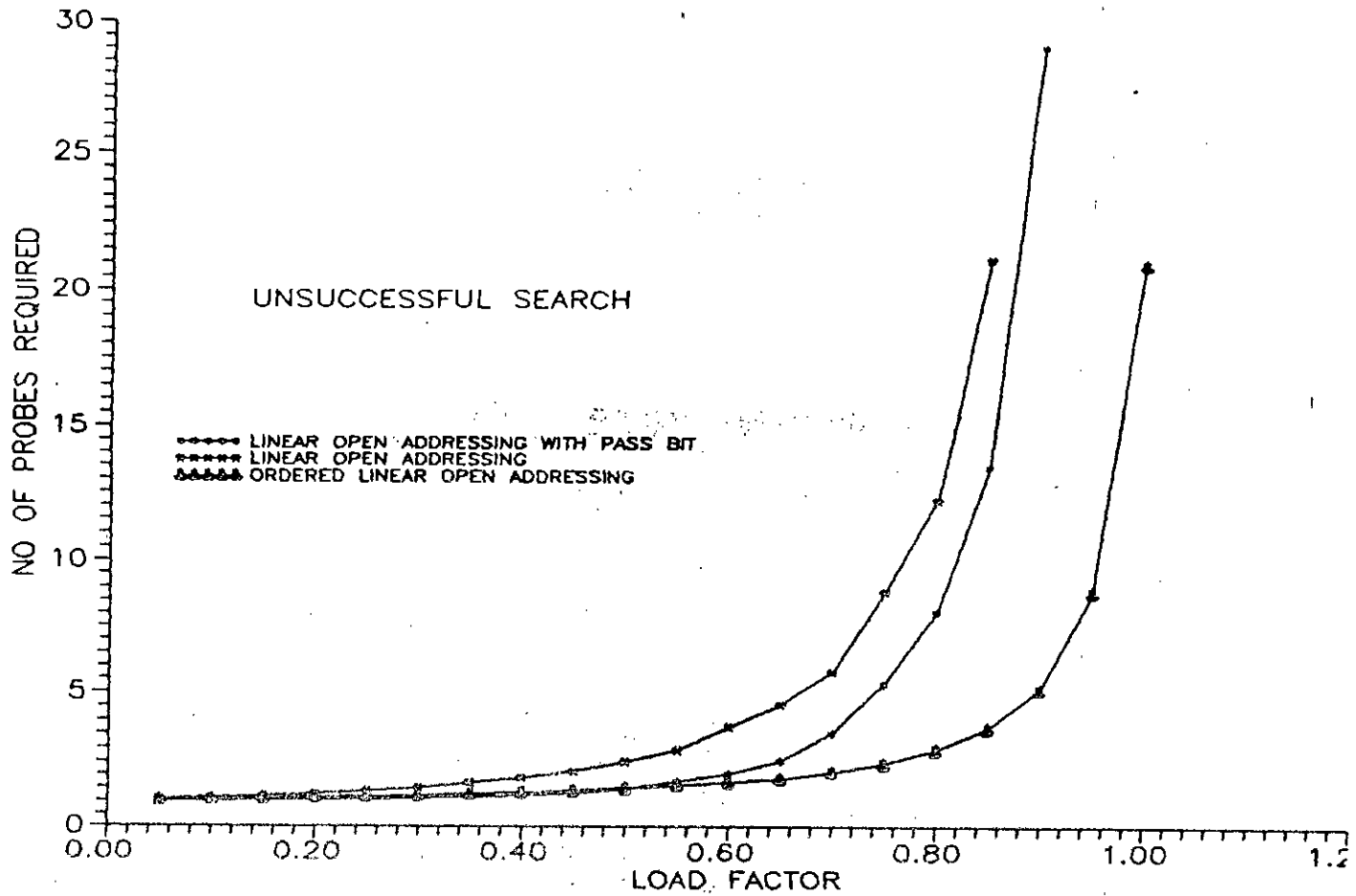
load factor, but if space is the major consideration then coalesced chaining is superior.

After chaining, the next collision resolution scheme is the linear probing which is the simplest of all the available methods. As discussed in section 4.4 in linear probing the performance degrades rapidly when the table gets full. To have a better performance we have examined this technique with two variations. The first one is the ordered linear probing where we have maintained an ordering relation among the elements in the table amounting to a considerable saving in unsuccessful searches, but like the ordered chaining method this does not make any change in performance for successful search. The next variation is the pass bit method which has been discussed in section 4.6. A one bit field in each table location can improve the performance in case of unsuccessful search under linear probing. Equation (4.15) shows that in linear probing when the table is full the average no of probes required in unsuccessful search is $1/2 (1 + M)$ where M is the table size. That is why we have tried to improve the performance giving strength on the unsuccessful search. Graph 4.3 and 4.4 shows the performance of the variations of linear probing in case of successful and unsuccessful searches respectively. Since each insertion in the ordered table is much costly, hence it is preferable only when insertion is less frequent event. Since a single bit in each table location improves the performance considerably and since a bit comparison is not so expensive compared to key comparison, therefore, pass bit method is the natural selection strategy in most of the situations.

Double hashing eliminates the problem of secondary clustering in linear probing using independent hash functions which is essentially indistinguishable from number of probes which would be required if the keys were inserted at random into the table. Section 4.5 deals with the behaviour of double hashing. Brent's algorithm on the other



GRAPH 4.3 Average no. of probes reqd. in successful search in different variations of Linear probing

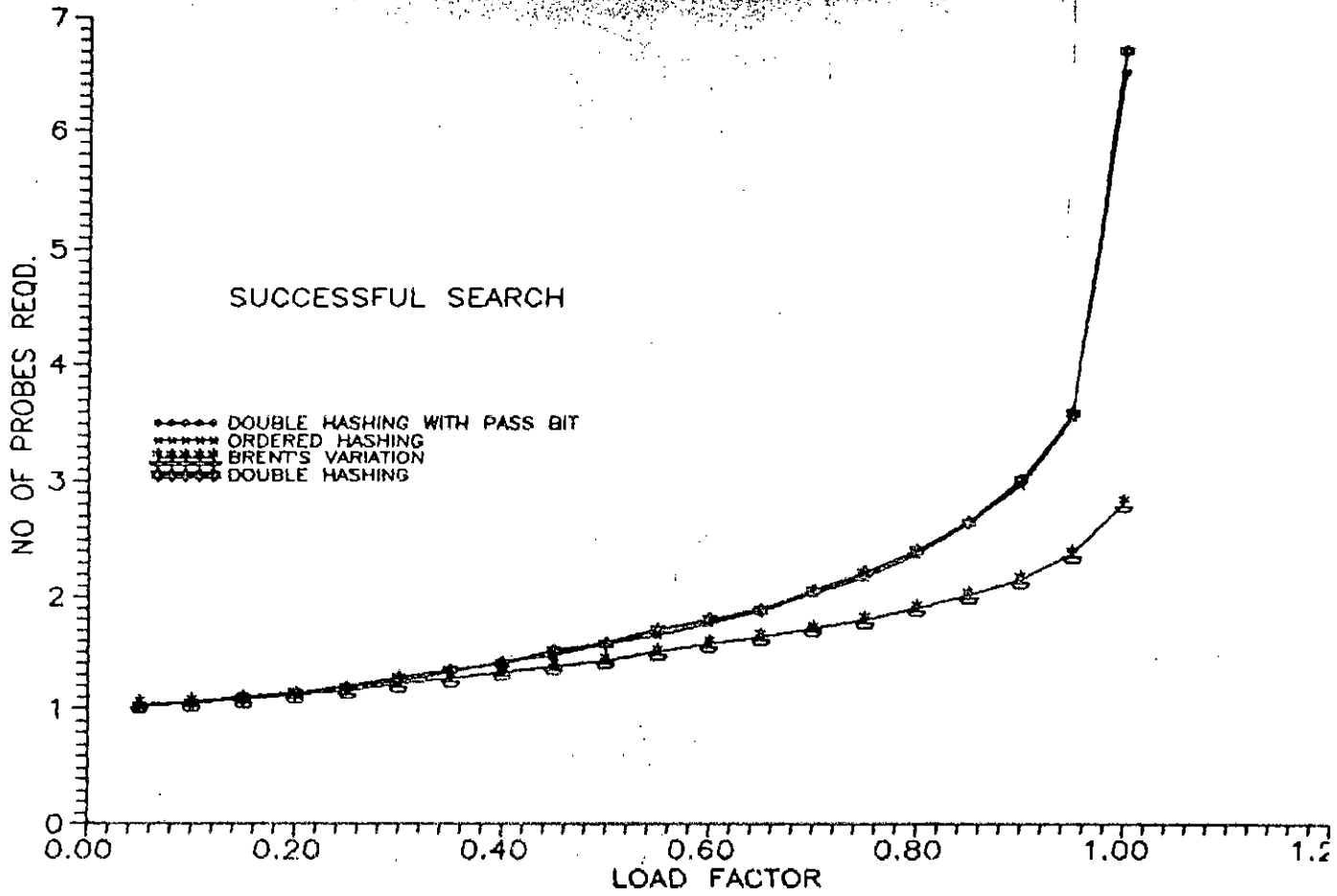


GRAPH 4.4 Average no. of probes reqd. in unsuccessful search in different variations of Linear probing

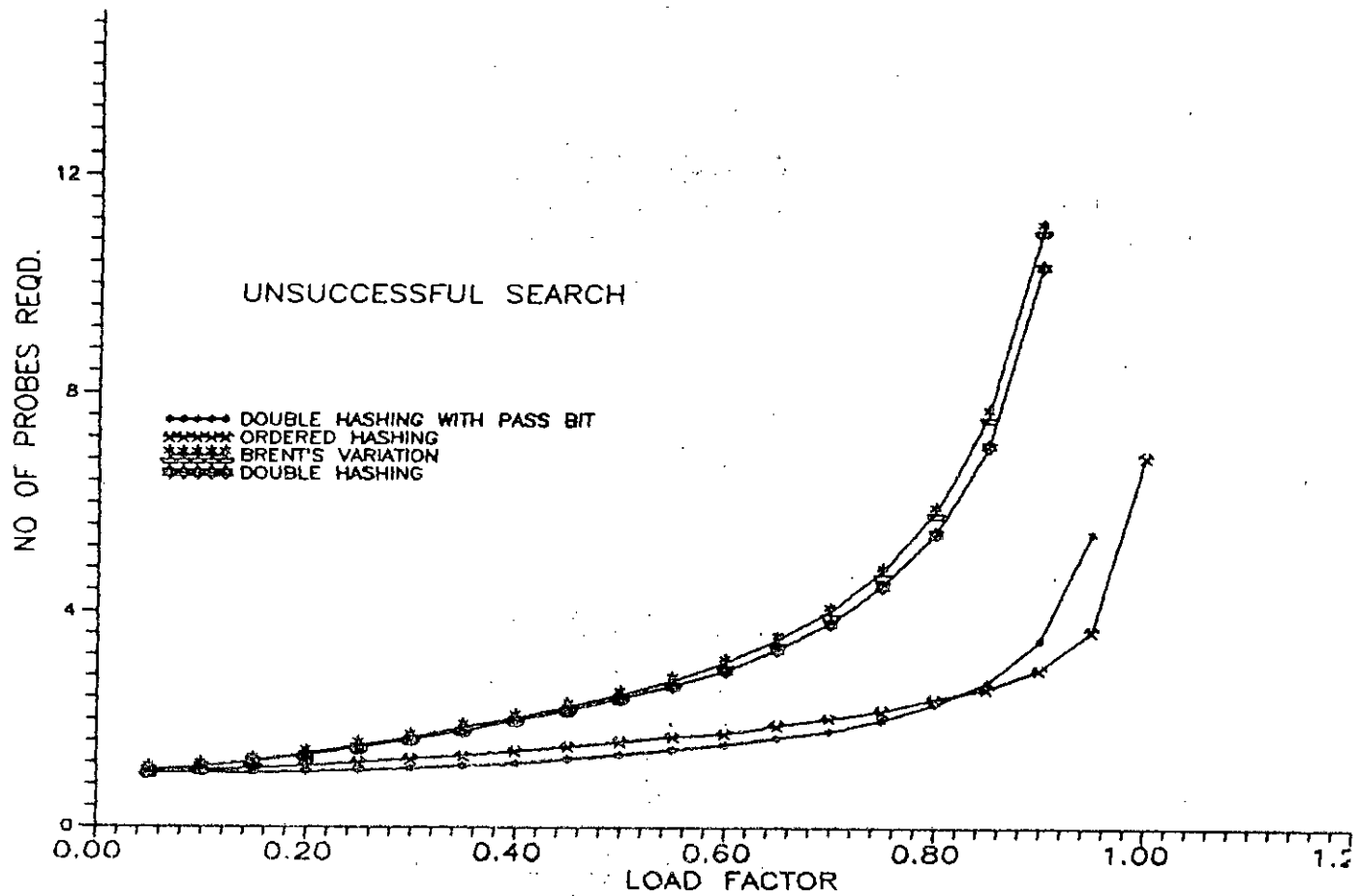
hand is a variation of double hashing which reduces the average number of probes per successful search but number of probes in an unsuccessful search is not reduced by Brent's variation, it remains at the same level as uniform hashing. Brent's algorithm is employed when successful searches are much more common than insertion as in the symbol table algorithm. Another variation of double hashing is the pass bit method which employs a single bit in the table location to improve the performance of unsuccessful search. Graph 4.5 and 4.6 shows the results for this different variations and conform to with the theoretical results. Here also if unsuccessful searches are more common then pass bit method is superior to ordered double hashing because the latter method is too costly to be worthy in practice, and the former one needs only an extra bit of memory for its realization.

Next we have shown the performance of all the major hashing schemes in graph 4.7 and 4.8. These figures show that when space is not the critical factor separate chaining is the best possible choice. But this method is particularly important when number of elements is greater than the table size i.e., when the load factor is greater than unity. Linear probing on the other hand is the simplest to implement, but its average behaviour when the table is nearly full discourages its use. In this situation double hashing comes to help us because it is the best among all the hashing methods if both space and time are taken into consideration although rehashing of the keys require a little bit more computational cost.

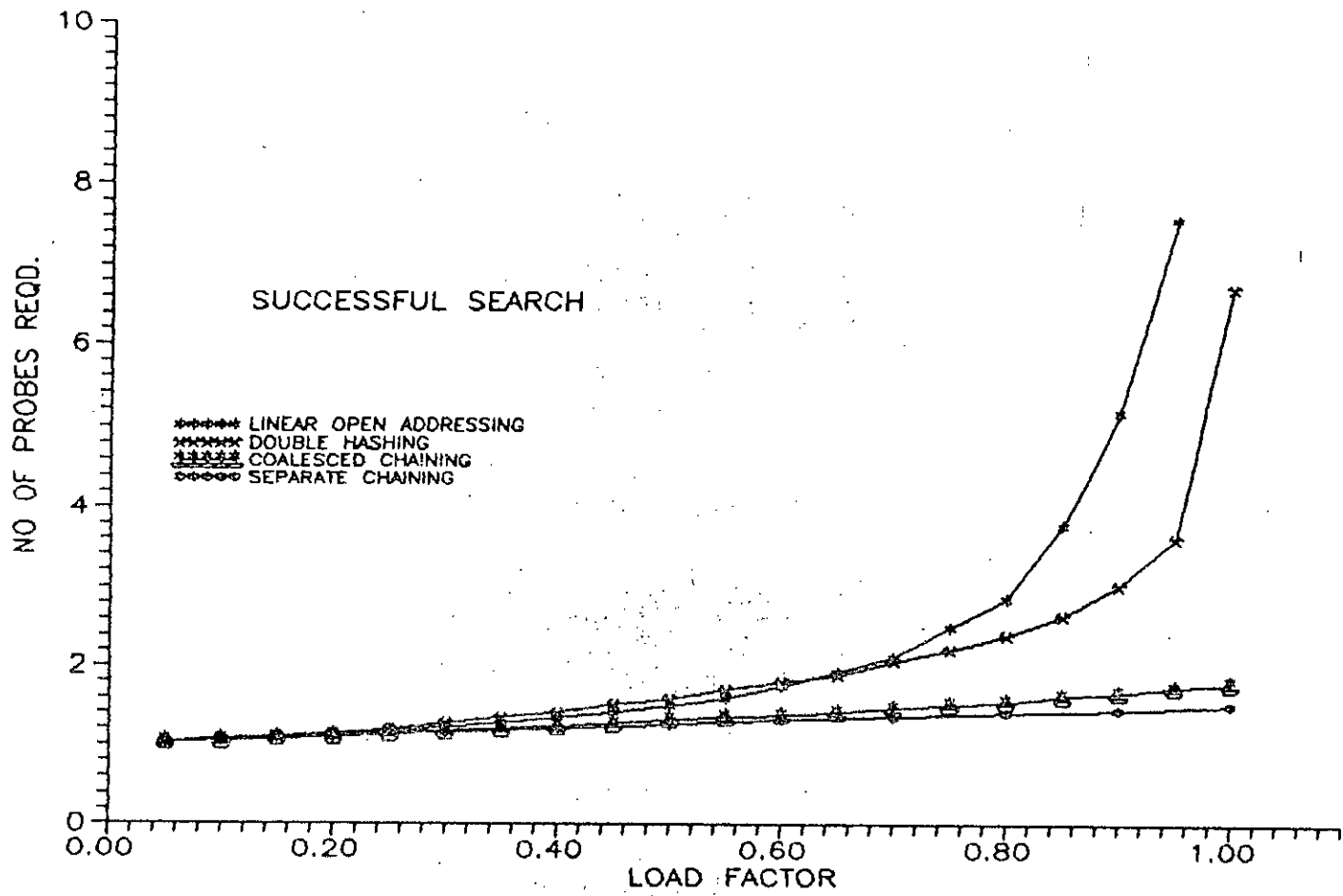
We will finish our discussion after a little remarks about external hashing. We were mainly interested in internal hashing, but have carried out some experiments out of curiosity. Attempt has been made to compare the behaviour of linear probing and double hashing in the context of external hashing. In our experiments bucket sizes of



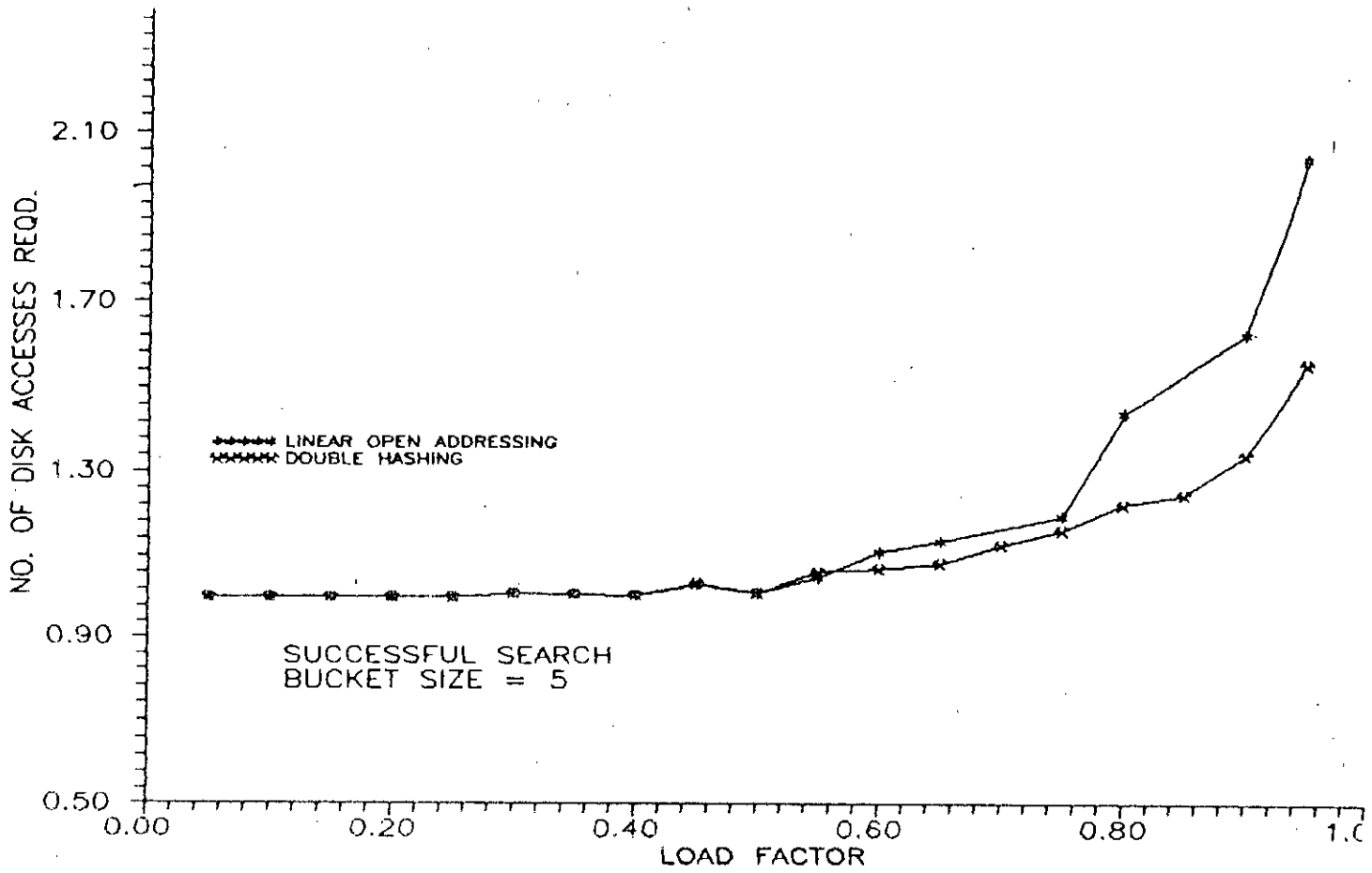
GRAPH 4.5 Average no. of probes reqd. in successful search in different variations of Double Hashing



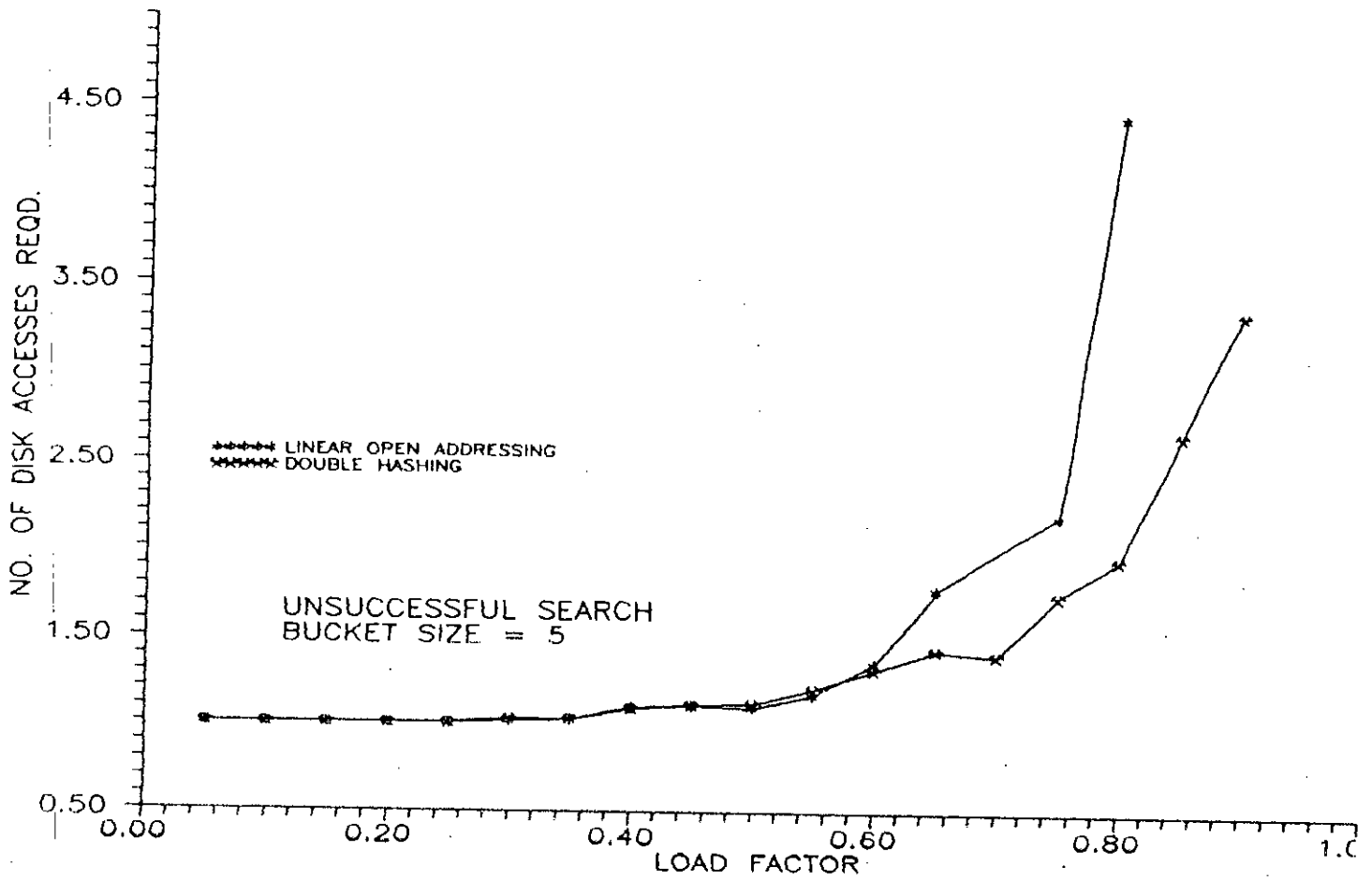
GRAPH 4.6 Average no. of probes reqd. in unsuccessful search in different variations of Double Hashing



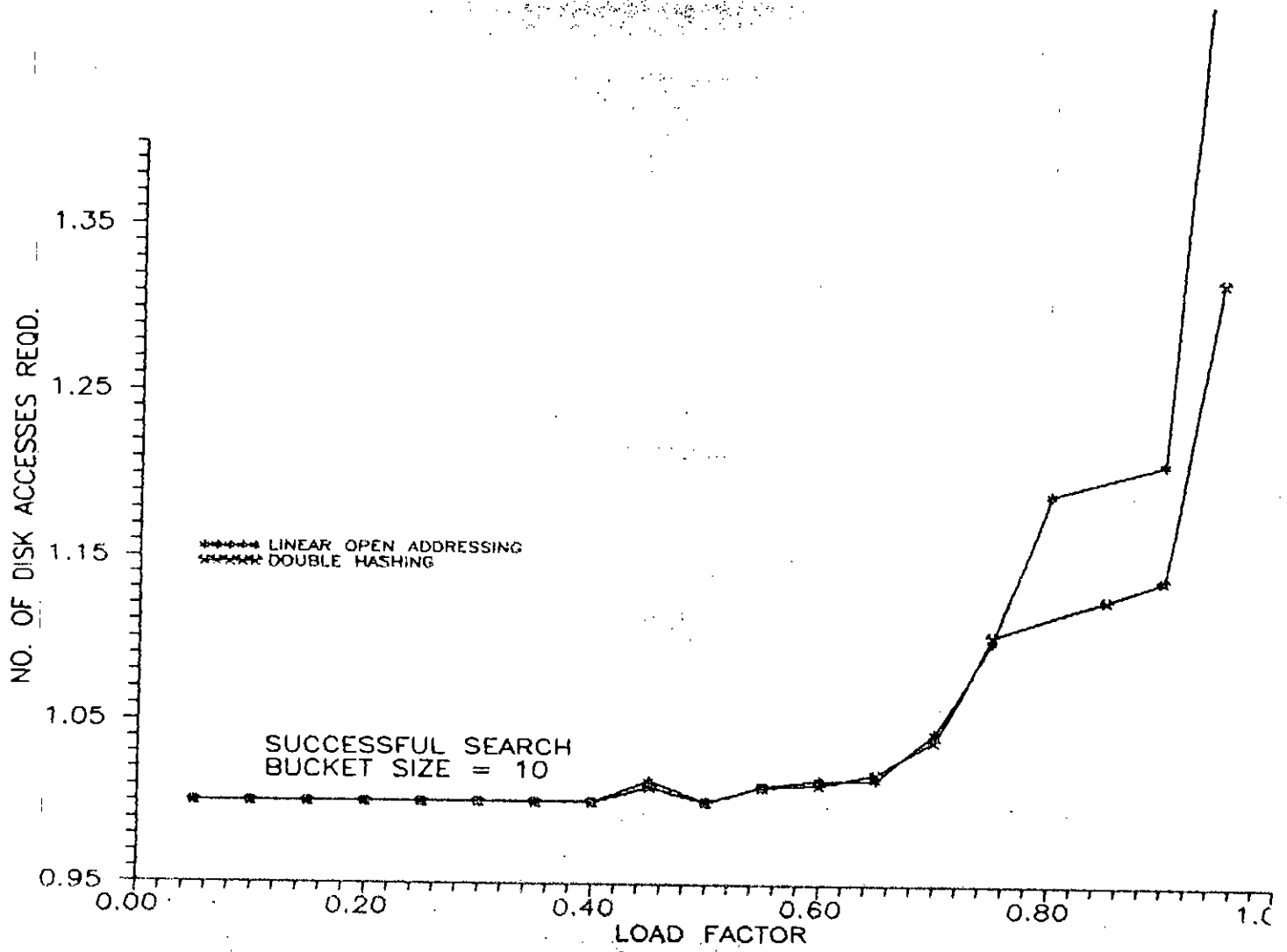
GRAPH 4.7 Average no. of probes reqd. in successful search in different Hashing schemes



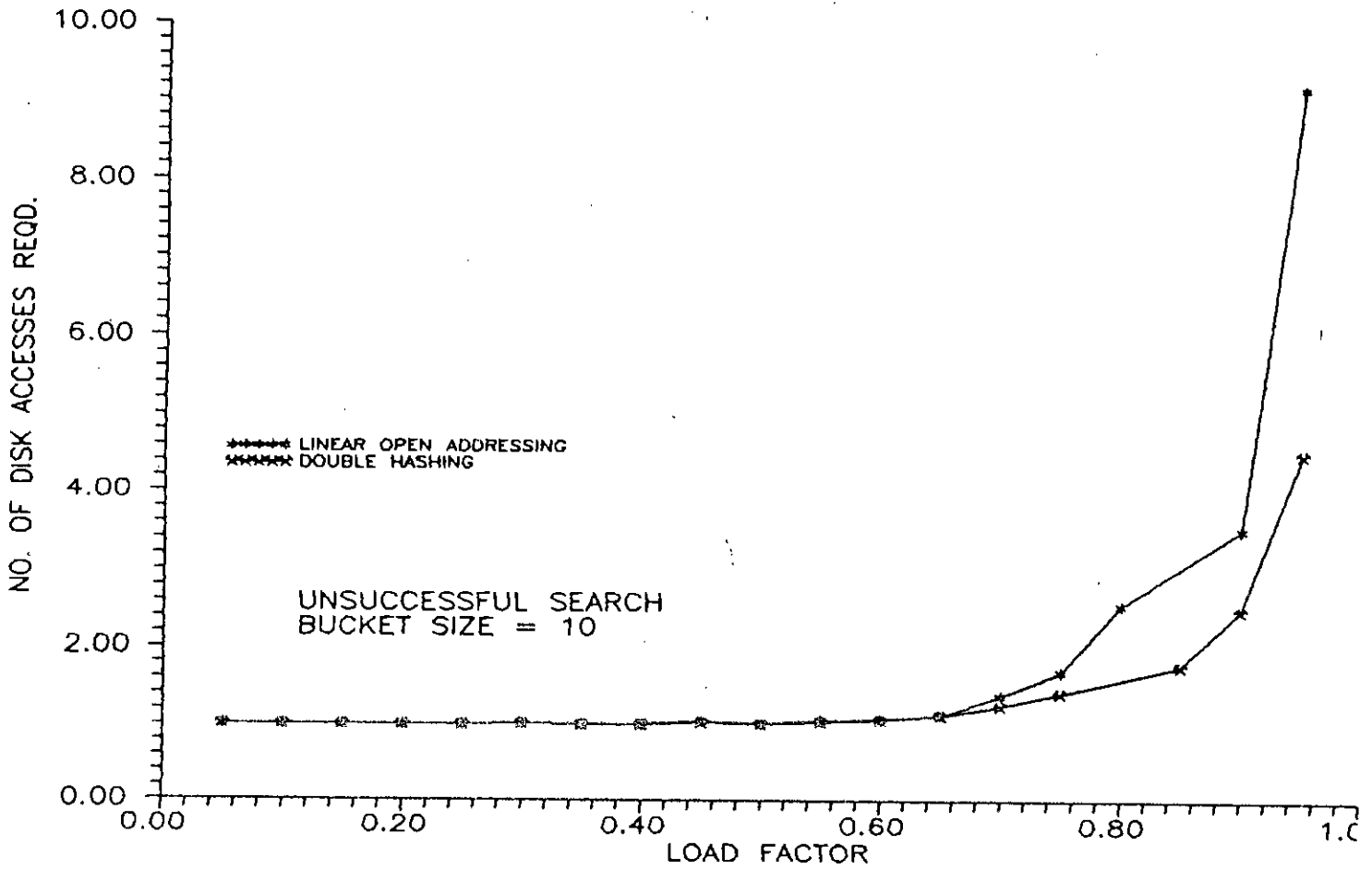
GRAPH 4.9 Average no. of disk access reqd. in successful search when
 Bucket size = 5



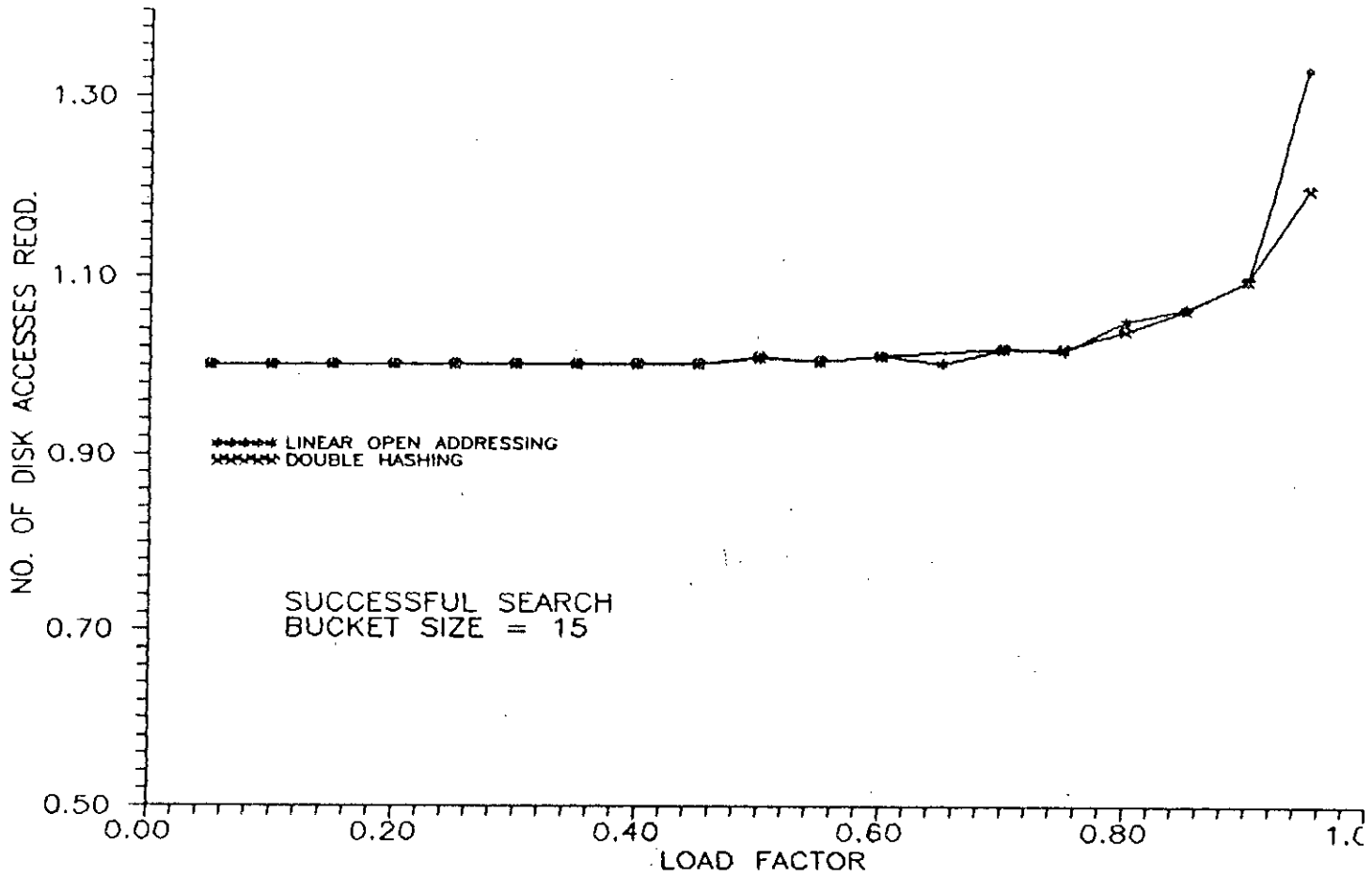
GRAPH 4.10 Average no. of disk access reqd. in unsuccessful search when Bucket size = 5



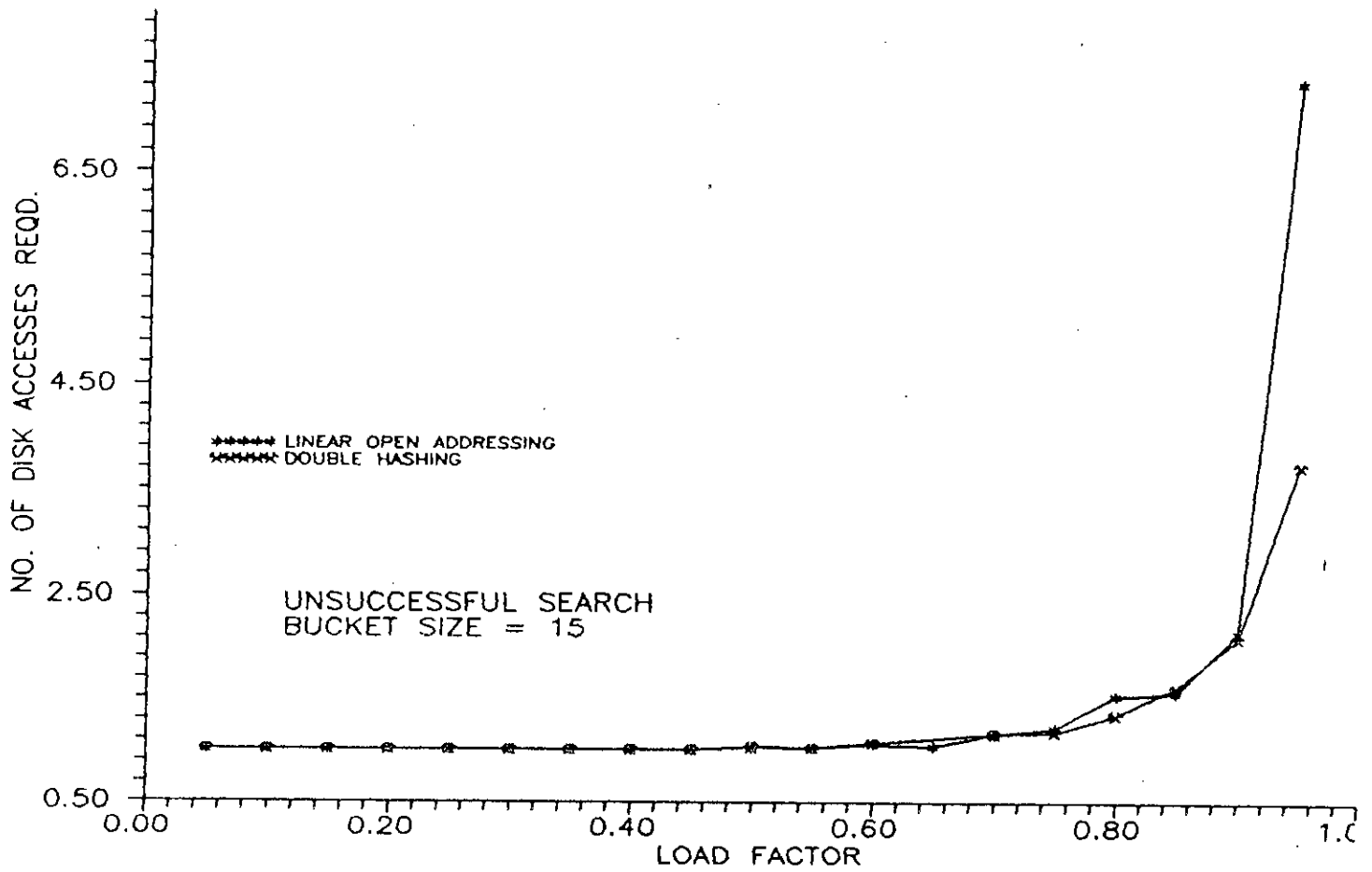
GRAPH 4.11 Average no. of disk access reqd. in successful search when Bucket size = 10



GRAPH 4.12 Average no. of disk access reqd. in unsuccessful search when Bucket size = 10



GRAPH 4.13 Average no. of disk access reqd. in successful search when
Bucket size = 15



GRAPH 4.14 Average no. of disk access reqd. in unsuccessful search when Bucket size = 15

5, 10 and 15 have been used. The experimental curves 4.9 through 4.14 show that double hashing is superior to linear probing at almost any load factor if number of disk accesses is the prime consideration. When dealing with external storage such as a disk, the number of buckets that have to be read from external storage is not the only determinant of access efficiency. Another important factor is dispersal of the buckets accessed that is, how far apart the buckets accessed are from each other. In general a major factor in the time it takes to read a block from a disk is the seek time. This is the time it takes for the disk head to move to the location of the desired data on the disk. If two buckets accessed one after other are far apart, more time is required than if they are close together. Given this fact, it would seem that linear rehashing is the most effective technique because although it may require accessing more buckets, the buckets it accesses are contiguous. At the very end of chapter four dynamic and extendible hashing techniques have been introduced to show that a single access is sufficient to bring any record from external memory with an appropriate modification of index structure.

5.6 Conclusions.

The experimental results on static table algorithms suggest the use of Uniform and Fibonacci search techniques in cases where a particular processor saves considerably in doing addition and subtraction operation than doing division operations. This conclusion has been drawn from the fact that the former two algorithms need only addition and subtraction operation for their implementation whereas the commonly used Binary search procedure uses division operation for every iteration and in most of the cases division operation is much costlier than addition and subtraction operation.

The simplest algorithm we have discussed for dynamic tables is dynamic tree search

algorithm. This algorithm is very simple to implement and its average search time is logarithmic when the input keys are perfectly random as verified by simulation experiments. But the tree shape deteriorates sharply as the input keys become nonrandom in nature. This worse behaviour becomes severe when some new keys are inserted after the deletion of some random keys. Therefore the adoption of dynamic search tree depends solely on the randomness property of the incoming keys and the frequency of deletions made upon it.

Two restricted tree structures namely, Balanced and 2-3-4 tree have been introduced to overcome the difficulties which may arise when the input keys are nonrandom in nature. It has been shown both analytically and experimentally that these algorithms behave logarithmically even when the input keys are nonrandom in nature. Moreover, the insertion cost in such trees is reasonably low so that one can easily adopt these algorithms in order to have an insurance against the bad worst case performance.

Under the proper conditions, hashing is unsurpassed in its efficiency as a table organization, since the average time for a search or an insertion is generally constant, independent of the size of the table. However, some important caveats are in order. First, hashing requires a strong belief in the law of averages, since in the worst case collision occurs every time, and hashing degenerates into linear search. Second, while it is easy to make insertions into a hash table, the full size of the table must be specified a priori, because it is closely connected to the hash function used; this makes it extremely expensive to change dynamically. If we choose too small a size the performance will be poor and the table may overflow, but if we choose too large a size much memory will be wasted. Third deletions from the table are not easily accommodated. Finally, the order of the elements in the table is unrelated to any natural order that may exist among the

elements, and so an unsuccessful search results only in the knowledge that the element sought is not in the table, with no information about how it relates to the elements in the table. If the problem at hand is not related to the above mentioned difficulties, Hashing technique is the appropriate choice without any question.

Among the various collision resolution schemes, separate chaining is the best choice if space is not a critical factor. In this scheme there is no difficulty with deletion and there is no chance of overflowing the table and it has the the least search time among all possible Hashing schemes. If space is the principal consideration then we can avoid the chaining method and adopt either Linear probing or Double Hashing scheme. Again Linear probing shows a poor behaviour as the table gets full even though this scheme is the simplest to implement. Therefore, if computation cost for second hash is reasonable, Double Hashing is more efficient than Linear probing. Moreover, after selecting any particular Hashing scheme, different variation thereof can be adopted as per requirement of the problem at hand. It should be mentioned that Hashing is particularly suited to external searching and with proper modification of the index table, any record can be retrieved from secondary storage device in only one access.

5.7 Suggestion for further study.

There are a large number of searching algorithms depending on the nature and characteristics of search procedure. Only some representative searching algorithms having wide applicability have been studied in this thesis. For example, algorithms using the digital properties of keys, algorithms related to graph search and algorithms commonly used in Artificial Intelligence have not been included in this thesis. So there is a wide scope to study these algorithms in future. Also the analytical results for the average behaviour of Balanced and 2-3-4 trees are still unknown. Only the empirical behaviour

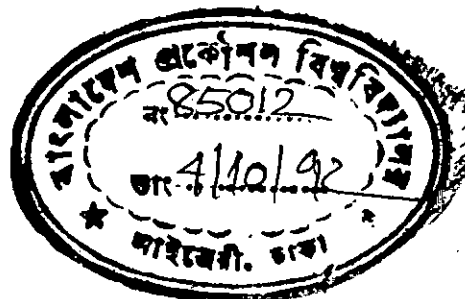
has been obtained by simulation experiments. One can attempt to find the behaviour of these algorithms analytically. Again the incorrectness of Hibbard's theorem has been pointed out through real examples and by logical reasoning. Attempt should be done to introduce mathematical support in this regard.

BIBLIOGRAPHY

- [1]. Aho, A. V., J. E. Hopcroft and J. D. Ullman : Data Structures and Algorithms, Addison - Wesley, Reading, Mass. 1983.
- [2]. Aho, A. V., J. E. Hopcroft and J. D. Ullman : The Design and Analysis of Computer Algorithms. Addison - Wesley, Reading, Mass. 1974.
- [3]. Amble, O. and D. E. Knuth : "Ordered Hash tables," Computer J., 18: 135-42, 1975.
- [4]. Amsbury, W : Data Structures from Arrays to Priority Queues, Wadsworth, Belmont, Ca., 1985.
- [5]. Baer, J. L. and B. Schwab : "A comparison of tree balancing Algorithms," Comm. ACM, 20(5), May 1977.
- [6]. Brent, R. P. : "Reducing the retrieval time of scatter storage techniques," Comm. ACM, 16(2), Feb. 1973.
- [7]. Brown, M : "A storage scheme for height- balanced trees." Inf. Proc. Lett., 7(5) : 231-32, Aug, 1978.
- [8]. Carter, J. L. and M. N. Wegman : "Universal classes of Hash functions," J. Comp. Sys. Sci. 18 : 143-154, 1979.
- [9]. Clampett, H. : "Randomized binary searching with tree structures." Comm. ACM. 7(3), 163-65, Mar, 1964.
- [10]. Fagin, R., J. Nievergelt, N. Pippenger and H. R. Strong : "Extendible hashing - A fast access method for dynamic files." ACM transaction on Database Systems, 4, 3 (September, 1979).

- [11]. Foster, C. C. : "A generalization of AVL trees," *Comm. ACM*, 16(8), Aug, 1973.
- [12]. Gonnet, G. H. and J. I. Munro : "Efficient ordering of Hash tables," *SIAM J. Comp.*, 8(3), Aug, 1979
- [13]. Guibas, L. J. and E. Szemerédi : "The analysis of Double Hashing," *J. Comp. Sys. Sci.*, 16 : 226-74, 1978
- [14]. Guibas, L. and R. Sedgwick : "A dichromatic framework for balanced trees," 19th Annual symposium on foundations of Computer Science, IEEE, 1978.
- [15]. Horowitz, E. and Sartaj Sahni : *Fundamentals of Computer Algorithms*, Galgotia Publications, 1990.
- [16]. Karlton, P. L. and S. H. Fuller, R. E. Scroggs, E. B. Kachler : "Performance of height-balanced Trees," *Comm. ACM*, 19(1) : 23-28, Jan, 1976.
- [17]. Knott, G. D. : "Hashing functions," *Computer Journal*, 18, Aug, 1975.
- [18]. Knuth, D. E. : *The Art of Computer Programming, Vol.1*, Addison - Wesley / Narosa, Indian Student Edition, 1989.
- [19]. Knuth, D. E. : *The Art of Computer Programming, Vol.3*, Addison - Wesley, 1973.
- [20]. Knuth, D. E. : "Optimum binary search trees," *ACTA Information*, 1:15-25, 1971.
- [21]. Larson, P. A. : "Analysis of uniform Hashing," *J. ACM*, 30(4) : 805 - 819, Oct, 1983.
- [22]. Larson, P. A. : "Dynamic Hashing," *BIT*, 18 : 184-201, 1978.
- [23]. Larson, P. A. : "Linear hashing with separators - A dynamic Hashing scheme achieving one access retrieval," Technical Report CS-84-23, University of Waterloo, Nov. 1984.
- [24]. Maurer W. and T. Lewis : "Hash table methods," *Comp. Surveys*, 7(1) : 5-19, Mar, 1975.

- [25]. Mehlhorn, K. : "Dynamic binary search," SIAM J. Comp., 8(2), May, 1979.
- [26]. Morris, R. : "Scatter storage techniques." Comm. ACM. 11(1), 38-44, Jan.1968.
- [27]. Nishihara, S. and K. Ikeda : " Reducing the retrieval time of hashing method by using predictors," Comm. ACM, 26(12), Dec, 1983.
- [28]. Reingold, E. M. and Willfred J. Hansen : Data Structures, Little, Brown and Company, 1983.
- [29]. Sedgewick, R. : Algorithms, Second Edition, Addison - Wesley, 1988
- [30]. Tanenbaum, A. M., Yedidyah Langsam, Moshe S. Augenstein : Data Structure Using C, Prentice - Hall, Inc. 1990.
- [31]. Vitter, J. S. : "Analysis of the search performance of Coalesced Hashing," J. ACM, 30(2), April, 1983.



Computer Algorithms
— S. S. S. S.