A COMPARATIVE STUDY OF HEIGHT-BALANCED TREE

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science in Computer Science

By

Ronghuey Alice Wang

December, 1987

ACKNOWLEDGEMENT

I would like to express my sincere gratitude to Professor Stephen Huang for his advice and guidance in every stage of this work. Appreications are due to Professors Anne L. Simpson and Tiee-Jian Wu for serving on my committee.

Last but not least, special thanks to my parents and husband, Allen, for their love and encouragement.

A COMPARATIVE STUDY OF HEIGHT-BALANCED TREE

An Abstract of Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science in Computer Science

By

Ronghuey Alice Wang

December, 1987

ABSTRACT

Height-balanced trees (H-trees), a recently proposed data structure, is a variant of B-trees. An H(β, γ, δ) tree is defined by three parameters: β, the size of a node; γ, the minimal number of grandsons which a node must have; and δ, the minimal number of leaves which bottom nodes must have. The purpose of this research is to study H-trees empirically. Algorithms, to insert and delete elements, are implemented. The results of our experiments confirm the validity of existing theories. For example, by varying the parameters δ and γ, significant changes in the tree's performance are observed: the height of H-trees decreases as γ increases, and the storage utilization increases as δ increases. Moreover, comparisons of H-trees with other variants of B-trees are shown to demonstrate the superiority of H-trees. Specifically, the average storage utilization of H-trees may be higher than that of B-trees by almost 20%.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

Introduction and Definitions

1.1 Introduction

The B-tree [1] and its variants have been widely used in recent years as a data structure for storing large files of information, especially on secondary storage devices. The guaranteed small search, insertion, and deletion time for these structures makes them quite appealing for database applications. However, each of these has had a serious flaw. Here, a parameterized variant of B-trees is demonstrated to be able to eliminate the weakness of the other variants.

Knuth [4] defines B-trees as the followings:

A B-tree of order m is a tree that satisfies the following conditions:

1) Every node has $\leq m$ sons.

2) Every node, except for the root and leaves, has $\geq m / 2$ sons.

3) The root has at least two sons (unless it is a leaf).

4) All leaves appear on the same level, and carry no information.

5) A nonleaf node with k sons contains k - 1 keys.

B-trees can be considered as multiway trees of order $m \geq 3$ where each internal node has k sons and k - 1 keys for $\lceil m / 2 \rceil \leq k \leq m$. Each node corresponds to a page of fixed size which can hold up to m - 1 keys and m pointers. But pages may be only partially filled. Information is transferred between main memory and backup storage device in units of pages. Insertion of new key and deletion of unwanted keys is quite simple for B-trees. The insertion of a new key ensues by the recursive strategy of splitting an "overfull" node (or page) having m keys into two and then moving one of its keys upward. It can easily be seen that in the worst case each page is at least half filled. The average storage utilization is much better. Yao [6] has shown that after a sequence of random insertions into an initially empty B-tree of high order, the storage utilization is approximately $\ln 2 \approx 69$ percent. However, if the keys are inserted in ascending order into an initially empty B-tree, the worst case lower bond of 50 percent storage utilization becomes significant. An attempt to solve this "sparsity problem" for B-trees is the main idea of "overflow" introduced by Bayer and McCreight [1], which leads to the notion of B*-trees; instead of splitting an overfull node, look first at its left (or right) brother. Say the immediate right brother has only j keys and $j + 1$ sons where $j < m - 1$. Then the overfull node "flow[s] over" into its brother node, i.e., shifts its additional keys to the right or left brother node. If the brother node is already full, then split both nodes and create three new nodes, each about two-thirds full. The result of this modification is an increase in storage utilization; at least two-thirds of the available space is utilized. More discussion about B*-trees will be seen in the subsequent chapter. Basically, the scheme of the new variant also applies the idea of "overflow". However, instead of looking at both immediate right and left brothers, it examines a small number of neighboring brother

nodes around, so that the storage can be fully utilized.

The definition of B-trees is too general. For a given number of keys, it may construct a lot of different representations. Different representations with same number of keys may possess different heights, which could make severe difference in the access cost and storage utilization. For example, a skinny tree with larger height needs more time to access. Meanwhile, each node in the tree, if not fully utilized, causes poor storage utilization.

Generally, to solve this problem, most of the variants apply two approaches: either adding conditions on the tree to constrain its development more constrictively, or finding the subset containing all B-trees that are optimal with respect to some measure. B*-tree is a typical application of the first approach. Another example is the dense multiway tree.

Dense multiway tree [2] is featured in its unique parameter r. A time/space trade-off is available by adjusting the value of r. The tree is called weakly dense tree with little effort to insert and delete when $r = 1$. On the other hand, it becomes strongly dense as r approaches m and the storage utilization tends higher at the expense of higher cost for insertion and deletion. However, the time/space trade-off works out quite unsatisfactorily. Because the performance does not change "smoothly" as r changes, it is almost impossible to find a proper value of r which promises both existence of acceptable storage utilization and insertion/deletion time. Since the insertion and deletion time can still guarantee O(log N), however, the worst storage utilization $(r + 1) / (m + 1)$ becomes extremely small for weakly dense tree. On the contrary, the strongly dense tree is able to maintain larger worst

storage utilization ($r + 1$) / ($m + 1$) and provides terrible insertion and deletion algorithm O(N).

The new class of variant to be presented also adopts the first approach. Several parameters are provided to constrain the tree's construction. There also exists a time/space trade-off under its definition. Nevertheless, this trade-off is guaranteed to be "smooth," and meets the minimum requirement of at least 50 percent utilization and O(log N) insertion and deletion time which are the attraction of B-trees. Not only the "satisfactory" time/space trade-off, but also the definition, by its simplicity, generality, and wide application, is highlighted.

The definition of H-trees shall be introduced in the next section. More detailed insertion and deletion algorithms can be found in Chapter II. Chapter II also points out several properties of this class of trees. The tree's charm is displayed most in Chapter III by the comparisons between it and two other classes of trees mentioned above. Finally, a brief summary and conclusion is presented in the last chapter.

1.2 Definitions

The class of height-balanced tree [3] is defined by three parameters: β, the size of a node; γ, the minimal number of grandsons a node must have; and δ, the minimal number of leaves bottom nodes must have.

Huang [3] defines this class of tree as followings:

A height-balanced tree of order (β, γ, δ), or an H(β, γ, δ), is a B-tree of order β such that:

1) Every node, except the root and those nodes in the bottom level, must have at least γ grandsons.

2) All nodes on the bottom level must have at least δ leaves.

β, the order of a tree, is the maximum number of sons a node can have. The other extreme for the sons' requirement is α which stands for the minimum number of sons a node can have. As a subclass of B-trees, each node in the H-trees also carries the property of at least half full capacity. So α is at least $\beta / 2$. For the purpose of easy insertion and deletion, we always choose $\alpha = \lceil \beta / 2 \rceil$. Parameter γ restricts the number of grandsons each node can have in order to prevent the tree from becoming too sparse. It provides a generalization of the brother condition. Because many keys reside in the bottom level of the tree, the density of the node on the bottom level dominates the storage utilization of the whole tree. However, they are unable to be constrained by the restriction of having γ grandsons. Therefore, parameter δ is used to require the minimum number of

leaves these nodes must have.

By varying the parameters, H-trees can perform like many other trees. For example, the 2-3 tree, which requires each node to have two or three sons, is one case of H-trees with $\gamma = 5$. And the class of B-trees of order m can be obtained by choosing $\beta = m$, $\gamma = (m / 2)^2$ and $\delta = \lceil m / 2 \rceil$.

Figure 1.1 is an example of H(3, 5, 2) tree with 32 keys.

CHAPTER II

Insertion, Deletion Algorithms and Properties of Height-Balanced Tree

More detail about the Height-Balanced Tree is studied in this chapter. The algorithm to insert and delete a key in H-tree and its properties will be described. Moreover, several examples will be used to illustrate the properties. Before discussing the insertion and deletion algorithms of H-tree, several basic operations that are needed will be introduced in the subsquent section. First of all, let's look at the data type and notations which will be used later:

```
TYPE
    ptr     = ^node;
    item    = RECORD
                key     : integer;
                p       : ptr;
              END;
    node    = RECORD
                fptr    : ptr;          {father pointer}
                ns      : 0..maxi;      {number of sons}
                ngs     : 0..maxi;      {number of grandsons}
                p0      : ptr;
                son     : array [1..100] of item;
```

END;

2.1 Basic operations

Three basic operations mentioned in Huang [3] are summarized below:

1) SHIFT(B, N, m) Shift m sons from node B to node N, where nodes B and N are adjacent brother nodes with the same father F. Since the construction of H-tree also uses the idea of "overflow" introduced by Bayer and McCreight [1] and which leads to B*-tree, operation SHIFT is the one that serves this purpose. Node B can be either the left or right hand side brother of node N. An example of SHIFT(B, N, 1) can be found in Figure 2.1 where B is the left adjacent brother of N.

2) PACK(N, m) Reduce the number of sons of node N to m. However, the number of N's grandsons should not be changed. To preserve the requirement of an H-tree, m must be at least $\lceil ngs / \beta \rceil$. Figure 2.2 gives an example of PACK(N, 2).

3) MERGE(B, N) Node B is merged into its adjacent brother N. The MERGE operation is a special case of SHIFT with m = the number of sons of node B. The difference is that B vanishes after the operation. An example of MERGE operation is illustrated in Figure 2.3.

We found the following two additional operations useful in the explanation of the algorithms.

4) SPLIT(*N*, *NEW*, *m*) Split m sons of node *N* to the newly created subtree *NEW*, so that the number of sons of subtree *NEW* will be *m*. Figure 2.4 gives an example of SPLIT(*N*, *NEW*, 2).

5) UNPACK(*N*, *m*) Increase the number of sons of node *N* to *m*. This operation is the reverse of the PACK operation except that the number of sons of *N* is increased rather than reduced after the operation. Therefore, the number of *N*'s grandsons still remains the same. SPLIT operation will be performed for (*m* - number of sons of node *N*) times in UNPACK operation, and the *NEW* subtree after each SPLIT is going to be inserted into node *N*. An example of UNPACK(*N*, 4) is given in Figure 2.5.

The above two operations SPLIT and UNPACK are applied in most cases of propagating to the upper level.

2.2 Insertion and deletion algorithms

The insertion and deletion algorithms introduced below are to construct an H-tree which is based on "dynamic" construction analysis compared with the "static" analysis. Because the random insertions and deletions which constitute the dynamic analysis give a better understanding of the average behavior of classes of trees than the corresponding static analysis, the insertion and deletion algorithm of B-tree can be applied to insert and delete an H-tree. However, several major changes will be made to satisfy the conditions required by H-tree.

2.2.1 Insertion algorithm

The same algorithm for finding the node *N* on the bottom level where the key should be inserted into a B-tree can be applied to insert a new key into an H-tree. In order not to contravene any of the conditions of H-tree, caution must be exercised for inserting the new key after the node is found. In order to take care the different conditions seperately, the insertion algorithm is divided into two phases: the first phase deals with the nodes on the bottom level, and the second phase deals with higher levels.

Phase1 The problem in the first phase occurs when a node has to be split into two nodes. Each will have only α (or $\alpha + 1$ if β is even) sons. Because the value of δ is at least α, this may violate the condition that each node on the bottom level must have at least δ sons. To "overflow" to other nonfull brothers is one obvious way to solve this problem . In case all brothers are full, we just add another node and average all the keys over all brothers. If the father was full originally, it may be forced to split. This problem shall be handled in the second phase.

There is an obvious disadvantage in the above algorithm, though it works. That is, in the worst case, the operation cost is always β, no matter how small δ may be. Another algorithm whose cost is not only of function of δ, but also equal to that of B-trees when δ

= α, will be replaced.

The modified alogrithm works as follows: Suppose m is the number of full nodes around node N. Instead of waiting until all the brothers of node N are full, we average all the keys of these m full nodes (plus the new key) over one more node as soon as m is large enough. In other words, all the keys in the m nodes and m -1 keys between them in father F plus the newkey, that is $(\beta - 1)m + (m - 1) + 1$ keys in total, are distributed over $m + 1$ nodes. Considering the condition that each bottom node should have at least δ sons, the following inequality must hold:

$$(\beta - 1)m + (m - 1) + 1 \geq (\delta - 1)(m + 1) + m.$$

The smallest integer solution of the above inequality is

$$m = \lceil (\delta - 1) / (\beta - \delta) \rceil.$$

It may be worth pointing out that m is a fairly small number, except when δ is near its maximum value. Thus, the cost is quite small compared with averaging the keys over all brother nodes.

Phase 1 of the insertion algorithm is given below.

INSERTH_PHASE1(*key*, *N*, *NEWKEY*):

INPUT	*key*	The key to be inserted.
	N	Node found at the bottom level where the *key* should be inserted.
OUTPUT	*NEWKEY*	Subtree generated as input of Phase 2.

ALGORITHM

Step 1 Add *key* into node *N*. If *N* is not full, then go to Step 6.

Step 2 If *N* is the root, then go to Step 5. Otherwise, if all the brothers of *N* are full then go to Step 4.

Step 3 SHIFT 1 subtree of *N* to *B* via any nodes in between *N* and *B*, where *B* is the nearest nonfull brother of *N*, go to Step 6.

Step 4 If *F*^.ns > *m*, then average the keys in *N* and its *m* - 1 full brothers into *m* + 1 nodes. Otherwise, average the keys in the *F*^.ns full nodes into *F*^.ns + 1 nodes (UNPACK(*F*, *F*^.ns + 1)). Go to Phase 2 to insert the *NEWKEY* into *F*.

Step 5 SPLIT *N* into two nodes, and increase the height of the tree by creating a new root with the two nodes as its sons.

Step 6 STOP.

Phase 2 In the second phase, we are given a node *N* on level *L* of H(β, γ, δ) tree *T* of height *h*, a key called *newkey*, and a subtree of height *h* - *L*, called *newson*, such that there is an *i* which satisfies one of the following conditions:

1) $k'' < newkey < k' < N\text{^}.\text{key}[i]$,

where k' is any key in new son and k'' is any key in $N\text{^}.\text{son}[i]$;

2) $N\text{^}.\text{key}[i] < k' < newkey < k''$,

where k' is any key in new son and k'' is any key in $N\text{^}.\text{son}[i + 1]$.

We also assume that node N is the jth son of its father F if N is not the root of the tree.

The task on this level is to merge *NEWKEY* and *NEWSON* into node N or his brothers (if possible) or to go up one level if N must be split. At the end of this iteration, we either stop or we reduce the problem to a higher level. We present the iterative algorithm below. Note that node B is always the adjacent brother of node N.

INSERTH_PHASE2(*NEWKEY1*, *N*, *NEWKEY 2*)

INPUT	*NEWKEY1*	Subtree to be inserted
	N	Node where *NEWKEY1* should be inserted
OUTPUT	*NEWKEY2*	Subtree generated as the input to be inserted for the next iteration

ALGORITHM

Step 1 Add *NEWKEY1* into N. If N is not full, then go to Step 10.

Step 2 If N is the root, then go to Step 9. Otherwise, if N is splittable (i.e., if both the leftmost and rightmost half of the sons have at least γ grandsons), then SPLIT a

node as *NEWKEY2* and go to Step 8.

Step 3 If adjacent brother B is not full, then SHIFT(N, B, 1) and go to Step 10.

Step 4 If N has no more than β^2 grandsons, then PACK(N, β) and go to Step 10.

Step 5 SHIFT(N, B, 1). If B is splittable, then split B into two nodes as in Step 2, go to Step 8.

Step 6 If B has no more than β^2 grandsons, then PACK(B, β) and go to Step 10.

Step 7 Add a new node W between N and B. Distribute the $2\beta + 1$ sons in B and N into nodes B, W and N, such that each node has at least α sons and γ grandsons.

Step 8 $N := N^{\wedge}$.fptr, go to Step 1.

Step 9 SPLIT N into two nodes, and increase the height of the tree by creating a new root with the two nodes as its sons.

Step10 STOP.

The above insertion algorithms are somehow different from those in Huang [3] in the following aspects:

1) Whether N, the node to be inserted, is full or not, the *NEWKEY1* is always inserted in the very first step.
2) In Step 2, node N is checked to see if it is the root or not. If it is, it must be also full, then action to split the root and increase the height of the tree must be taken. Otherwise, continue.

3) In INSERTH_PHASE1, one more condition is taken with care. That is, when all the brothers of node N are full and their aggregate plus N ($F\hat{}$.ns) is smaller than m, then the number of nodes whose keys to be averaged into is $F\hat{}$.ns + 1 instead of m + 1.

2.2.2 Deletion algorithm

Similarly, the deletion algorithm is also divided into two phases: the first phase deals with the nodes on the bottom level (level h - 1); the second one handles the rest. The key we are deleting is assumed to be located at the bottom level, as in the case of B-trees. If not, a key on the bottom level which is the next higher or lower can be found to replace it.

Phase 1 This is almost the reverse of Algorithm INSERTH_PHASE1. The algorithm is slightly different in that we have to check not only for the number of sons, but also for the number of grandsons. Suppose a key is deleted from a node N on the bottom level. If node N has less than δ sons, there are two ways to solve this problem: either borrow keys from its brothers ("underflow"), if possible, or reconstruct nodes such that N has one less brother. This may require examining N's grandfather. Because of the requirement of minimum number of grandsons, after insuring that N has enough sons, we check that N's father has enough grandsons. The deletion algorithm is given below, with

$$w = \lceil (\beta - 1) / (\beta - \delta) \rceil = m + 1 .$$

DELETEH_PHASE1(*key*, *N*)

INPUT	*key*	key to be deleted
	N	Node contains the *key*
OUTPUT	*N*	Node in the higher level that needs to process because it fails to meet the definitions after the deletion

ALGORITHM

Step 1 Delete the *key* from node *N*. If *N* has at least δ sons, go to Step 5.

Step 2 If a nearest brother *B* of *N* that has more than δ sons is found, then SHIFT(*B*, *N*, 1) and go to Step 5.

Step 3 If *F*^.ns ≥ *w*, then redistribute *F*^.ns nodes, including *N* and all the other sons of *F*, and all the keys separating them over *w* - 1 of the nodes (PACK(*F*, *w* - 1)). Otherwise, redistribute them into *F*^.ns -1 nodes (PACK(*F*, *F*^.ns - 1)). Delete the extra node.

Step 4 If *N*'s grandfather *G* exists and does not have enough grandsons, then go to Step 6.

Step 5 If *N*'s father *F* has enough grandsons, go to Step 7. Otherwise, go to Step 6.

Step 6 If *F* is not the root, then

N := *F*;

F := *N*^.fptr;

go to Phase 2.

Step 7 STOP.

Phase 2 The deletion algorithm for H-trees is more difficult than the insertion algorithm compared with other kinds of trees. We present an iterative algorithm that goes up one level at a time. The invariant of each iteration is such that node N may have lost up to one son and β grandsons in the previous iteration; we go up one level, and the above invariant must be true for the new node N if the whole process has not been terminated.

DELETEH_PHASE2(N)

INPUT	N	Node fails to meet the definitions of H-trees
OUTPUT	N	Node in the higher level that needs to process because it fails to meet the definitions after the deletion

ALGORITHM

Step 1 If node N has at least α sons, go to Step 7.

Step 2 If shifting one son from adjacent node B to N will not violate any of the H-tree restrictions for node B (i.e., $B\text{^}.ns > \alpha$ and $B\text{^}.ngs \geq \gamma + \beta$), execute SHIFT($B, N$, 1) and go to Step 7.

Step 3 If $N\text{^}.ngs + B\text{^}.ngs \leq \beta^2$, then MERGE($B, N$), PACK($N, \beta$) if $N\text{^}.ns > \beta$, and go to Step 7.

Step 4 Let B' be another brother of N such that N, B, and B' are adjacent. If B' has at least β sons and at least $\beta(\beta - 1)$ grandsons, go to Step 6. Otherwise, PACK(B', β - 1).

Step 5 (B' has at most β -1 sons, and hence can absorb α son if we merge B and N.) MERGE(N, B), PACK(N, β + 1), SHIFT(N, B', 1), and go to Step 7.

Step 6 If node B' is also adjacent to N, then SHIFT(B', N, 1). Otherwise, SHIFT(B', B, 2) and SHIFT(B, N, 1).

Step 7 If node N has less than γ grandsons, go to Step 2. Otherwise, if F is the root, check to see if it has to be deleted, then go to Step 8. If F is not the root, then

$N := F$;

$F := N$^.fptr;

and go to Step 1.

Step 8 STOP.

The deletion algorithms above have some difference with those in Huang [3] in the following aspects:

1) Similar to 3) in the insertion algorithm comparisons, one more condition is added in Step 3 of DELETEH_PHASE1. If F^.ns is less than w, the redistribution node number, because it is not big enough for w - 1, will be F^.ns - 1.
2) In order to prevent the null pointer, node G's existence will be insured before the

number of its grandsons is checked in Step 4 of DELETEH_PHASE1. For the same reason, an examination is performed in Step 6 of DELETEH_PHASE1 to see if node F is the root.

3) In the Step 4 of Phase 2 in Huang's algorithm, two conditions are discussed. If $B'^{\wedge}.ns \geq \beta$ and $B'^{\wedge}.ngs \geq \beta(\beta - 1)$ then algorithm flows to Step 6, otherwise, if $B'^{\wedge}.ns < \beta$ then B' is going to be packed into $\beta - 1$ sons. However, the situation 'if $B'^{\wedge}.ns \geq \beta$ and $B'^{\wedge}.ngs < \beta(\beta - 1)$' is missing. Therefore, the above algorithm takes this condition into consideration in Step 4. No matter what value $B'^{\wedge}.ns$ might be, B' will be packed into $\beta - 1$ sons as long as its number of grandsons is less then $\beta(\beta - 1)$.

2.3 Examples and properties of Height-Balanced Tree

A Pascal program HTREE which is implemented to build a height-balanced tree consisting of the insertion and deletion procedures can be found in Appendex B. Several experiments are made using this program. It is easier for us to get a better idea and understanding of H-trees and their properties through the results of the experiments in the format of tables and figures.

Example 2.1

Table 2.1 and Figure 2.7 are the outcome of the execution of program HTree with $\beta = 3$, $\gamma = 5$, $\delta = 2$ and 128 random integers as input data which are listed in Figure 2.6. Figure 2.7 illustrates the final construction of this H-tree and Table 2.1 shows the information, including the height, number of nodes, storage utilization, about it as each increment of input data (keys). Also Figure 2.8 which is a double Y-axis chart with number of keys *N* as the X-axis and both the height and storage utilization as the Y-axis is drawn according to the data of Table 2.1.

Let's now look at some interesting phenomon in Figure 2.8 from the aspects of the two components of Y-axis: height and storage utilization. First, as we can see, the range of *N* becomes wider and wider as the height increases. This simply means the higher tree contains more keys then the lower one. Next, let's focus on the storage utilization. As with the same property of B-trees, the storage utilization of this H-tree moves cyclically between 0.5 and 1.0. However, the extreme values 0.5 and 1.0 only occure in the little number of keys where the curve is extremely unstable. As the number of keys *N* increases, the curve also becomes smooth. It stays steadily between 0.8 and 0.9 after about 100 key insertions. This gives us an idea of the value of the approximate average storage utilization of H-tree. This value which approaches 1.0 is significantly higher than other trees. More detailed comparisions between H-trees and other trees will be discussed in the following chapter.

Besides, it is obvious that the storage utilization curve drops sharply every time the height curve goes up one level. The increased nodes that result from every time the tree

getting higher are responsible for the sudden drop of storage utilization. However, the storage utilization usually keeps increasing, though not continuously, in the same level of height as the number of keys increases. The reason it does not increase continuously is that the increment of nodes causes the decline of the storage utilization.

The following examples demonstrate the relationships between the parameters, and how the two additional parameters δ and γ influence the performance of height-balanced tree.

Example 2.2

Tables 2.2 and 2.3 which contain several height-balanced trees with $\beta = 7$, and various degrees of δ and γ using two sets of input data, each with number of keys = 1024, individually are two seperate results of the execution of program HTREE. The existence of an H-tree requires specific values of δ and γ [3]. In other words, they must locate in the following ranges: $\alpha^2 \leq \gamma \leq (\beta^2 + 1) / 2$, $\alpha \leq \delta \leq (\alpha\beta - \beta + 1)$. Thus, $16 \leq \gamma \leq 25$ and $4 \leq \delta \leq 5$ for $\beta = 7$. Although both are based on different input data, the results are quite the same: except the height always remains 4,the average storage utilization and average accumulated insertion cost are increased; however, the average access cost declines as γ increases. On the other hand, the values in every catagory, except height,

increase when δ goes up to 5.

The increment of γ means a node owns more grandsons. This will result in fewer nodes in the whole tree which causes the higher storage utilization. As Chapter III will state more clearly, access cost is the number of nodes accessed to find the node containing a particular key, and insertion cost is the number of nodes visited to insert a key. According to Huang [3], the average access cost will be controlled by the height. However, the increment of γ results in more keys accommodated in each level which will decrease the height of the tree. Therefore, the access cost declines while the height decreases. Meanwhile, the increment of γ also results in more effort to insert a key because the number of nodes in each level increases.

Now, let's study the influence of the parameter δ on H-trees. The increment of δ gets the nodes residing on the bottom more sufficiently utilized. So, it is obvious that δ controls the storage utilization. Nevertheless, the increment of keys on the bottom level will also result in higher cost for access or insertion.

We have already found how δ and γ control the height, storage, and operation cost of the Height-Balanced Trees. Apparently, there exists the time/space trade-off when choosing the parameter values. However, the time/space trade-off is quite "smooth"; since

the changes of δ and γ only cause relatively small changes in performance. Thus, the class of trees gives arbitrarilly good performance by choosing suitable parameter values. Some properties concerning the H-trees can also be seen in the next chapter.

CHAPTER III

Comparisons of Height-Balanced Tree and Others

To illustrate that H-tree has satisfactory performance, comparisons of H-tree with other classes of trees are made in this chapter. The tree structures selected for comparisons are B*-tree and dense multiway tree. The properties and insertion algorithms of these two tree structures will be reviewed in the following section. Some important measures to judge the performance of the tree structures - height, storage utilization, access cost, and insertion cost - are used for comparisons. Also, some experimental results will be listed for comparison with the theoretical estimates, and between several classes of tree structures.

3.1 B*-tree

B*-tree is an improvement upon the basic B-tree structure. The modified tree by Knuth [4] utilizes at least two-thirds of the available space in every node. The definition and insertion algorithm of the B*-tree are detailed below.

A B*-tree of order m is a tree which satisfies the following properties:

1) Every node except the root has at most m sons.

2) Every node, except for the root and the leaves, has at least $(2m - 1) / 3$ sons.

3) The root has at least two and at most $2\lfloor(2m - 2) / 3 + 1\rfloor$ sons.

4) All leaves appear on the same level.

5) A nonleaf node with k sons contains $k - 1$ keys.

Suppose node N is found on the bottom level to insert a new key with B as its adjacent brother node. The insertion algorithm of B*-tree INSERTB*(N, *NEWKEY*) is given as below:

Step 1 Add key into N. If N is not full, then go to Step 5.

Step 2 If N is the root, then go to Step 4. Otherwise, if adjacent brother B is not full, then average the keys in nodes N and B (i.e., SHIFT(N, B, (m + B^.ns) / 2) and go to Step 5.

Step 3 Dsitribute evenly the $2m - 1$ keys in nodes N and B into N, B and *NEWKEY* which is just created, such that each node is about two-thirds full, and insert the new node into F,

$N := F$;

$F := N$^.fptr;

go to Step 1

(i.e., SHIFT(N, B, $(2m - 2) / 3$), SPLIT(B, *NEWKEY*, $(2m - 1) / 3$) and

INSERTB*(*F*, *NEWKEY*)).

Step 4 If *N*^.ns > 2(2*m* - 2) / 3 + 1 then split *N* into two nodes, and increase the height of the tree by creating a new root with the two nodes as its sons.

Step 5 STOP.

3.2 Dense Multiway Tree

Another tree structure, known as dense multiway tree (DMT), which also improves the storage utilization of a B-tree was developed by K. Culik, Th. Ottmann, and D. Wood [2]. A parameter *r* is introduced into the definition of B-trees. Dense Multiway Tree *T* is defined as follows.

An *m*-ary tree *T* is said to be *r*-dense, where $1 \leq r \leq m - 1$ iff the following hold:

1) The root of *T* is at least binary.
2) Each nonfull node different from the root has either only full brothers and at least one such brother or at least *r* full brothers
3) All leaves have the same depth.

In other words,

1) If a node *p* is the only son of its father *F*, then *p* must be full.

2) If a node p has k sons, then

i there are at least r full sons of p if $k \geq r + 1$;

ii otherwise, there is at most one nonfull node among the sons of p.

The insertion algorithm for dense multiway tree INSERTDMT(N, *NEWKEY*) operates as below, where N and B are exactly the same as those in the B*-tree algorithm.

Step 1 Add key into N. If N is not full, then go to Step 7.

Step 2 If N is the root, then go to Step 5. Otherwise, if an unsaturated brother B is found then SHIFT 1 subtree of N to B via any nodes inbetween N and B, go to Step 7.

Step 3 If F is unsaturated, then SPLIT the leftmost son of N as *NEWKEY* and go to Step 6.

Step 4 If N is not the rightmost son of F, then SPLIT the rightmost, otherwise, the leftmost, son of N as *NEWKEY* and go to Step 6.

Step 5 Create a new root with N as its only son and go to Step 3.

Step 6 $N := F$;

$F := N$^.fptr;

go to Step1

(i.e., INSERTDMT(F, *NEWKEY*)).

Step 7 STOP.

To insure that the SHIFT procedures in Step 2 of INSERTDMT, do not destroy the

strongly dense tree structure ($r = m - 1$), the SHIFT operation in DMT insertion is a little different from that of H-tree:

Step 1 If $r = m - 1$, then SIDESAT(N).

Step 2 SHIFT(N, B, 1).

Procedure SIDESAT(N, tf_right) makes rightmost (if tf_right = 'true') or leftmost (if tf_right = 'false') son of N saturated. Let tf_right = 'true', q be the rightmost son of N, and Bq be the immediate left brother of q which must be saturated.

The algorithm of SIDESAT operates as:

Step 1 If q is saturated, then go to Step 4.

Step 2 If q^.ns = 1, then make the leftmost son of Bq saturated and SHIFT m -1 sons to q (i.e., SIDESAT(Bq, (NOT tf_right)), SHIFT(Bq, q, m -1)) and go to Step 4.

Step 3 For i := 1 to m - q^.ns do
SIDESAT(Bq, tf_right);
SHIFT(Bq, q, 1).

Step 4 STOP.

To better illustrate the structural difference between H-tree, B*-tree and dense multiway tree, Figures 3.1 and 3.2 are demonstrated. They show three classes of tree

structures with equal height (1, 2, and 3) and number of keys (1, 3, and 11) respectively.

3.3 Comparisons

To make comparisons of the performance of these three classes of trees, there are several criteria that can help us to judge the performance of a tree structure T including:

1) Height

The height h of T is the number of edges in the longest path from root to a leaf. According to Huang [3], the maximal and minimal height for an H-tree with N keys are $\lfloor 2 + 2 \log_{\gamma}((N + 1) / 2\delta) \rfloor$ and $\lceil 1 + \log_{\beta}((N + 1) / 2) \rceil$ respectively. However, every r-dense m-ary tree with N keys is of height $h \leq 2 \log_{m}(N + 1) + 1$.

2) Storage utilization

Intuitively storage utilization u is the ration of the used portion of all the nodes.

$$u = \frac{\text{number of keys in } T}{(\text{number of nodes in } T)\ (\text{node size} - 1)}$$

where number of keys usually is denoted as N and node size is the maximum number of sons a node can have. It is equal to β in H-tree case. Therefore, (node size -1) represents the maximum number of keys a node can have.

The number of nodes visited during an operation of T also provides a measure of its performance.

3) Access Cost

Access cost (ac) is the number of nodes accessed in order to find the node containing a particular key. The range of average access cost for an H(β, γ, δ) tree of N keys ac yields:

$$h - (h - 1) / \beta \leq ac \leq h - 1 / \delta$$

4) Insertion Cost

Insertion cost is the number of nodes visited in order to insert a key into T. The cost is at most $2(h - 1) + \beta + (\delta - 1) / (\beta - \delta) + 1$ for H(β, γ, δ).

Since the H(3, 5, 2) tree is the H-tree with smallest order, we can afford a detailed study of this case. The competitors for B* tree and dense multiway tree are B*(3) and DMT(3, 1). Table 3.1 shows the theoretical estimates of the height, storage utilization, access cost and insertion cost for H(3, 5, 2), B*(3), and DMT(3,1).

To support the theoretical issue, three Pascal programs, HTREE, B* and DMT in appendix B, C, and D, which build these three classes of trees, are implemented according to respective insertion algorithm for realistic execution and proof.

Table 3.2 which is generated under the execution of the HTREE program illustrates the height, average storage utilization, average access cost, and average accumulated insertion cost of H-tree for ten different data sets each with $N = 128$. Tables 3.3 and 3.4 show the corresponding results derived from programs B* and DMT respectively, using the same data. The average storage utilization is the average of 128 storage utilization for each key insertion. Similarly, average accumulated insertion cost is the average of 128 accumulated insertion costs for each key insertion. No matter under which tree structure, the heights for $N = 128$ in these three tables all yield 5.

The averages of the ten data sets for Tables 3.2, 3.3, and 3.4 are collected into Table 3.5. In addition, the theoretical estimates according to Table 3.1 are also illustrated. We can see that, in Table 3.5, each experimental result is within the range of, or even perform better than the theoretical estimates.

Surprisingly important, we find that H-tree has the best performance in almost every category. This is stated more clearly in Table 3.6 which demonstrates the ratio of every figure in Table 3.5 to the figure of H-tree in each criteria. Among three classes of tree structures, B*-tree obviously has the worst performance because of lowest storage utilization and highest access and insertion costs. The access cost of dense multiway tree is about 3% lower than that of H-tree. On the contrary, H-tree has better storage utilization and insertion cost. Compared with the average storage utilization 0.69 in the case of B-tree [6], the improved B-tree variant - H-tree makes almost 20% progress in storage utilization.

Table 3.7 - Table 3.11 are corresponding to Table 3.2 - Table 3.6 except that N is increased to 256. They provide a confirmative support to the results of comparison because the ratio comparisons between the trees in Table 3.11 are quite similar with those in Table 3.6. Nevertheless, by comparing Table 3.5 and Table 3.10, there are some interesting differences. It is undoubtful that the height is raised because of the increment of N and all the operation costs are increased because of the increment of height. On the other hand, the storage utilization of H-tree and DMT also increases by the increment of N. However, this is not true for B*-tree. Reviewing the storage utilization of Table 2.2 and Table 2.3 which is with 1024 keys, we can conclude that the storage utilization of H-tree tends to 1 as N tends to the limit. This is another property of H-tree.

CHAPTER IV

Conclusions

We have presented a new class of balanced multiway trees, the height-balanced tree which is a generalization of B-trees. Some experimental results of this class of trees and the comparisons between it and others are given. Apparently, some strong evidence in favor of this new class has been demonstrated.

We have compared the storage utilization of the two classes - H-trees and B-trees for insertions. This has indicated that the H-trees do indeed have a better storage utilization than B-trees. In particular, we have shown that the average storage utilization tends to 1 as N and δ tend to the limit, but this is not true for B-trees. Furthermore, the experimental results prove that H-trees not only perform better in storage utilization than B-trees, but also perform better than B*-trees and even dense multiway trees. As a matter of fact, the results indicate that the actual application for H-trees shows even higher average storage utilization than the theoretical estimate.

As mentioned, H-trees with high parameters have excellent storage utilization and smaller heights. However, it becomes difficult to maintain these tree increases with an increase in the parameters γ and δ. Fortunately, the increment is relatively small compared to other trees. There is obviously some kind of trade-off between the amount of work for

insertion and the density of trees obtained by performing iterative insertions.

Though varying r, the basic parameter of dense multiway tree, can also allow a time/space trade-off. However, the performance of dense multiway tree is not satisfactory. And this is because no value of r guarantees both acceptable worst-case storage utilization and O(log N) insertion and deletion time. Besides, the time/space trade-off is not so "smooth" as that of H-trees. "Smooth" time/space trade-off means that small changes in the parameters cause relatively small changes in performance.

Not looking at brothers at all, as the scheme for B-trees, is very simple but generates quite sparse trees which waste the space of memory. On the other hand, looking at all brothers could be very messy and time consuming. However, this utilizes the storage most sufficiently. Thus we may consider these two classes of trees as the two endpoints of a whole variety of trees and insertion schemes in between; they are obtained by looking at more and more brothers.

Whether or not a scheme is appropriate depends on the actual environment. This is determined mainly by the number of search operations compared with the number of updates which have to be carried out and, furthermore, by the time which is necessary to settle a backup storage request.

The parameters δ and γ in H-trees are defined as user's specification. As we discussed in Chapter II, they can make significant changes to the tree's performance. Thus, it is convenient to change the parameters dynamically according to the user's need

under different environments. The user may select a small δ value to enjoy efficient insertion and deletion when there is quite sufficient storage. Once he has the need to utilize storage more efficiently, higher storage utilization can be traded by higher insertion and deletion costs simply by adjusting the value of δ higher without shutting down the whole system for reorganization. Similarly, this can be applied to the parameter γ.

H-tree features a variation of parameters. By selecting proper parameters, it can perform like many other trees. Therefore, it certainly provides a framework for comparing various trees. As shown in the previous chapter, H-tree has the better performance over B*-tree and dense multiway tree. After comparing them to H-trees, there is not much reason to search for efficent insertion algorithms for B*-trees and dense multiway trees.

REFERENCES

1. BAYER, R., And McCREIGHT, E. "Organization and maintenance of large ordered indexes," Acta Inf. 1 (1972), 173-189.

2. CULIK, K., OTTMANN, T., AND WOOD, D. "Dense multiway trees," ACM Trans. Database Syst. 6, 3 (1981), 486-512.

3. HUANG, S. "Height-Balanced Trees of Order (β, γ, δ)," ACM Trans. Database Syst. 10, 2 (1985), 261-284.

4. KNUTH, D. E. "The Art of Computer Programming, vol. 3, Sorting and Searching," Addison-Wesley, Reading, Mass., 1973.

5. ROSENBERG, A. L., AND SNYDER, L. "Time and space optimality in B-trees," ACM Trans. Database Syst. 6, 1 (1981), 174-193.

6. Yao, A. "On random 2-3 trees," Acta Inf. 9, 2 (1978), 159-170.

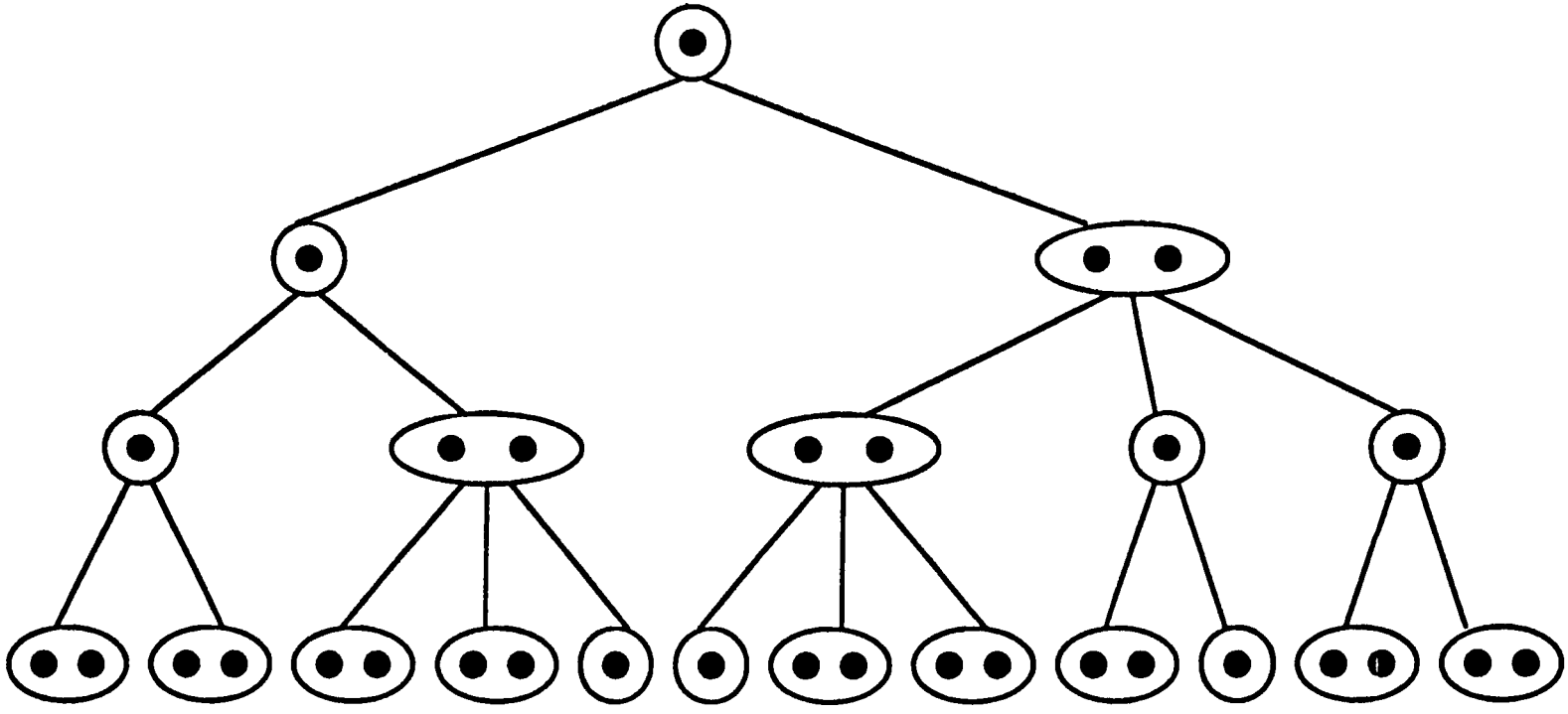

Figure 1.1 An H(3, 5, 2)-tree with 32 keys

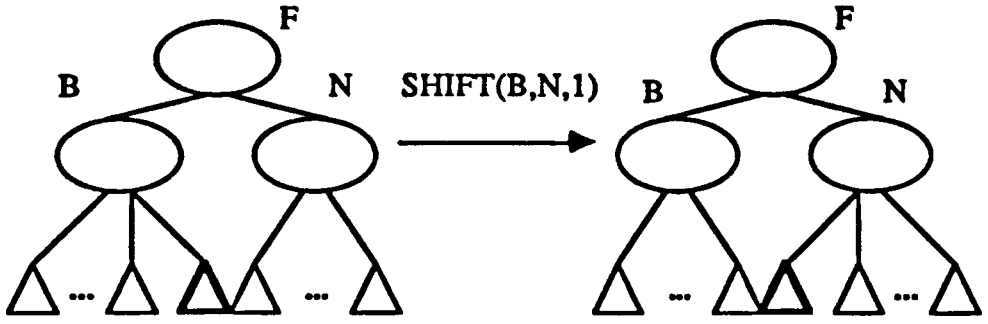

Figure 2.1 Example of operation SHIFT.

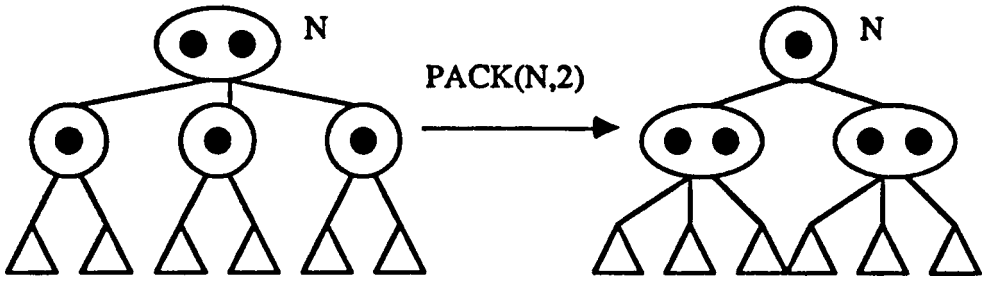

Figure 2.2 Example of operation PACK.

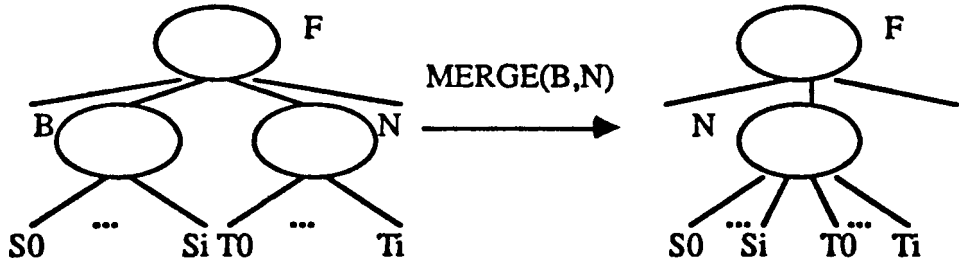

Figure 2.3 Example of operation MERGE.

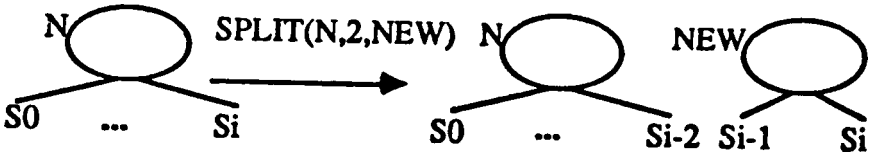

Figure 2.4 Example of operation SPLIT.

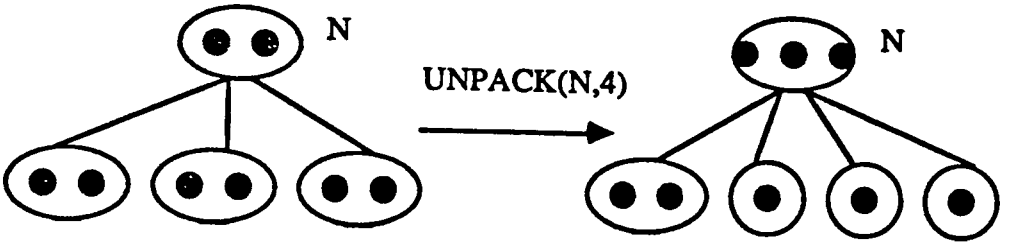

Figure 2.5 Example of operation UNPACK.

25 75 127 63 32 85 38 66 86 20 116 9 109 88 65 107 82 129 89 77 7 57 133 19 91 59
34 48 96 90 69 80 118 2 117 134 115 27 24 60 128 26 104 41 23 130 4 62 39 124 47
132 52 114 15 105 12 121 16 36 8 56 131 94 73 45 84 68 93 18 44 106 126 28 112 95
42 53 10 51 111 71 108 30 61 76 21 125 83 17 31 14 1 70 120 54 110 72 13 46 78 5
101 43 22 11 106 33 29 3 123 92 119 55 79 64 74 37 50 103 87 122 67 40 81 6 58 35
0 0

Figure 2.6 Randomly generated input data, in the range of 1 to 134, of example 2.1

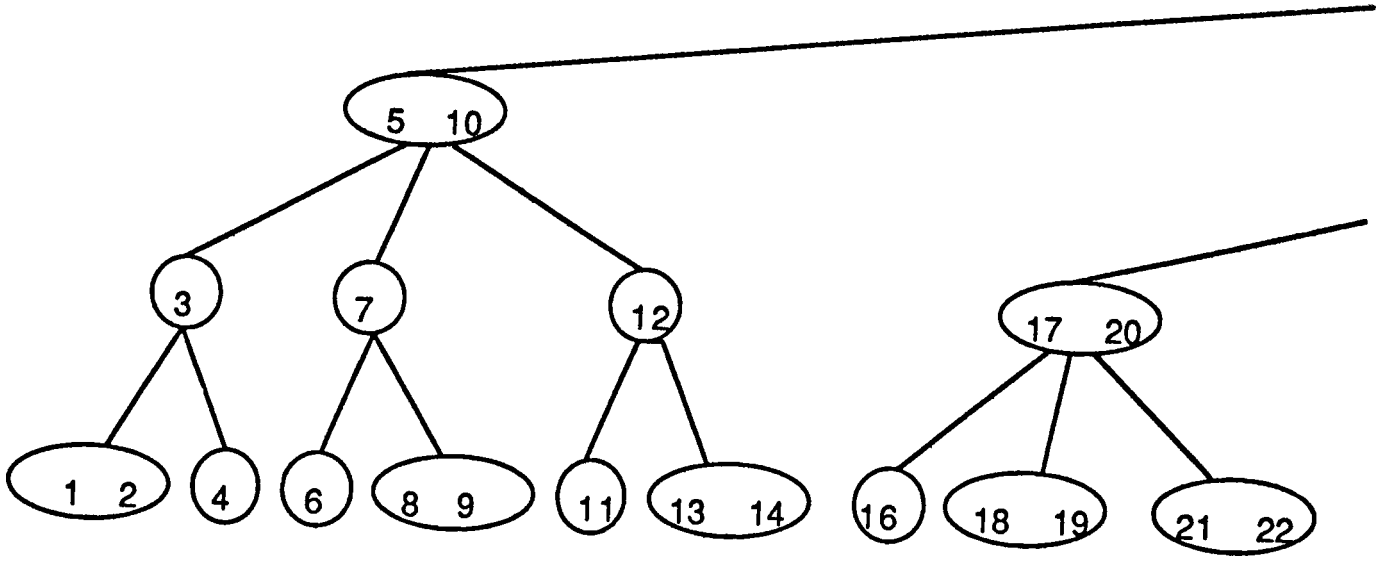

Figure 2.7 Construction of H(3, 5, 2) with input data in Figure 2.6

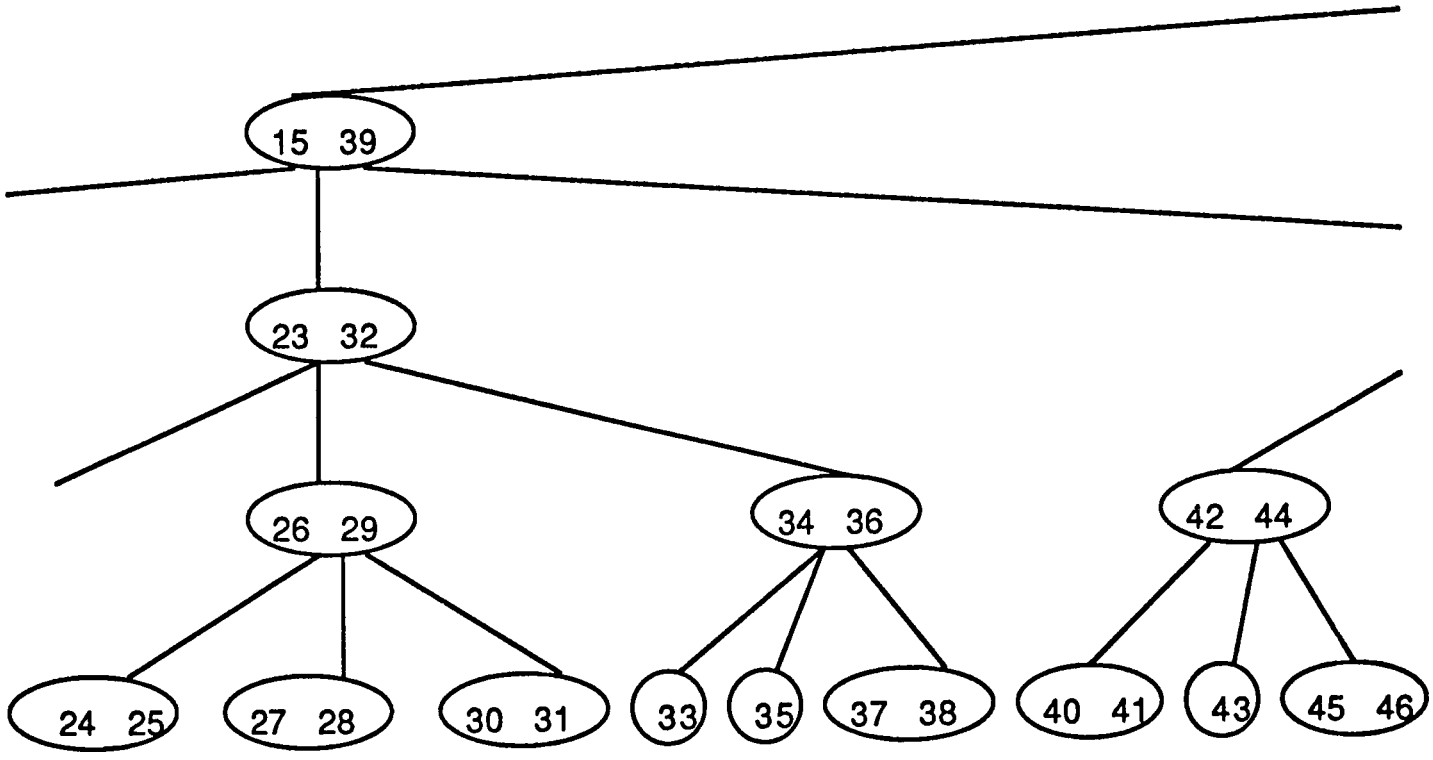

Figure 2.7 (cont.) Construction of H(3, 5, 2) with input data in Figure 2.6

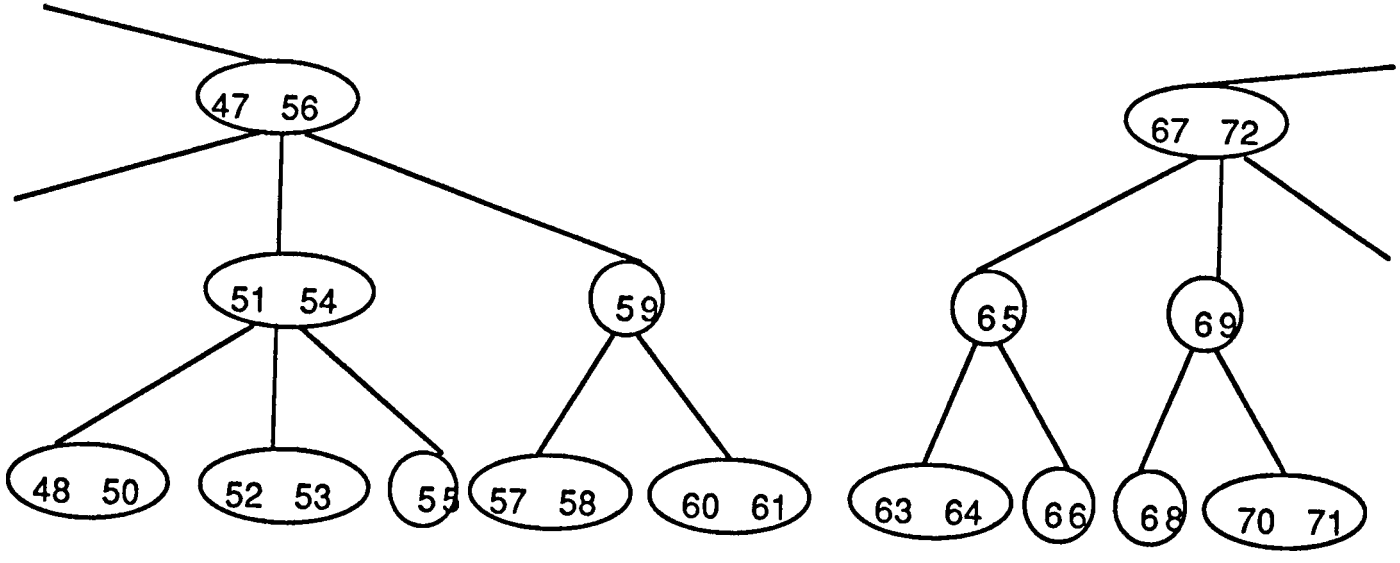

Figure 2.7 (cont.) Construction of H(3, 5, 2) with input data in Figure 2.6

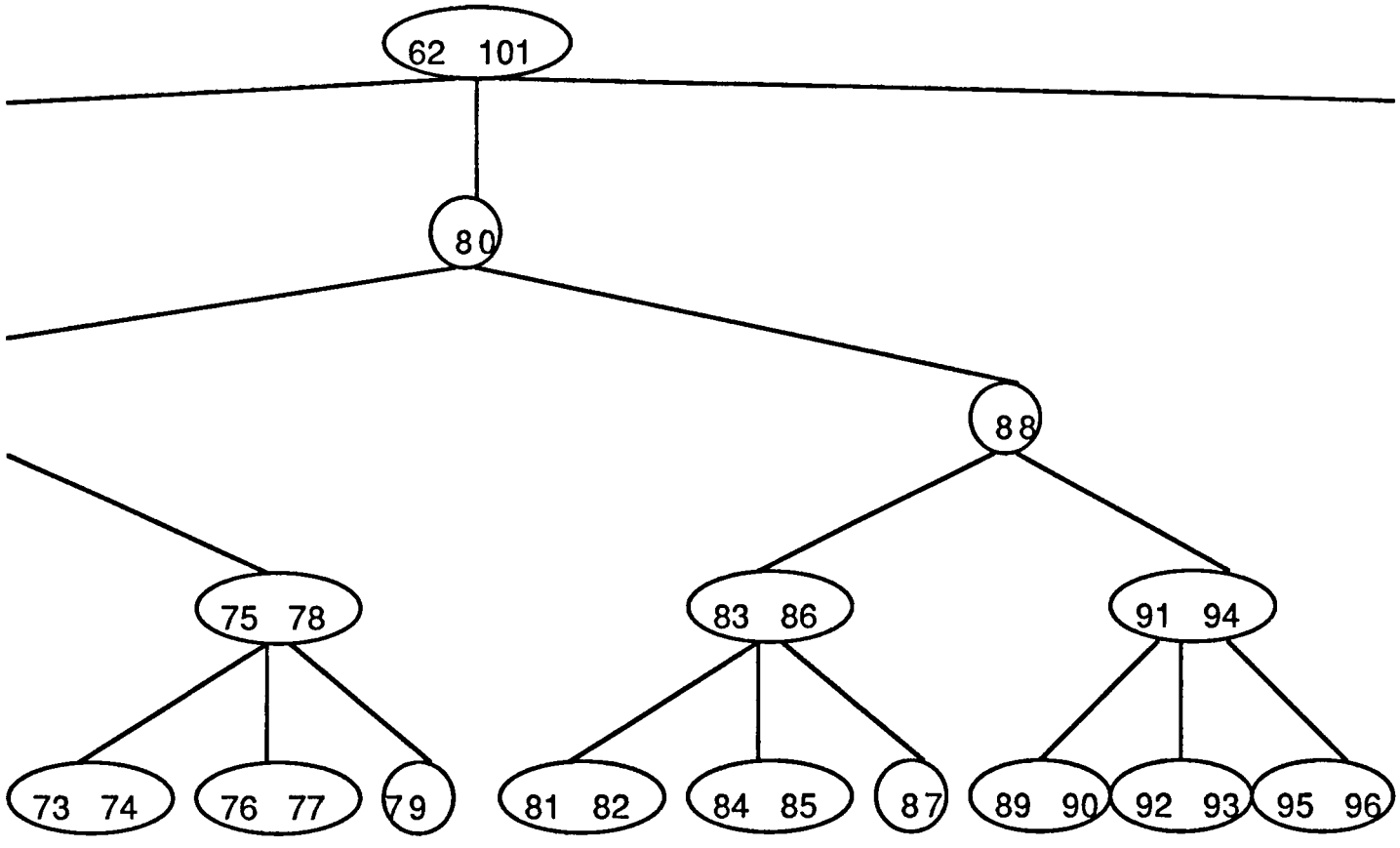

Figure 2.7 (cont.) Construction of H(3, 5, 2) with input data in Figure 2.6

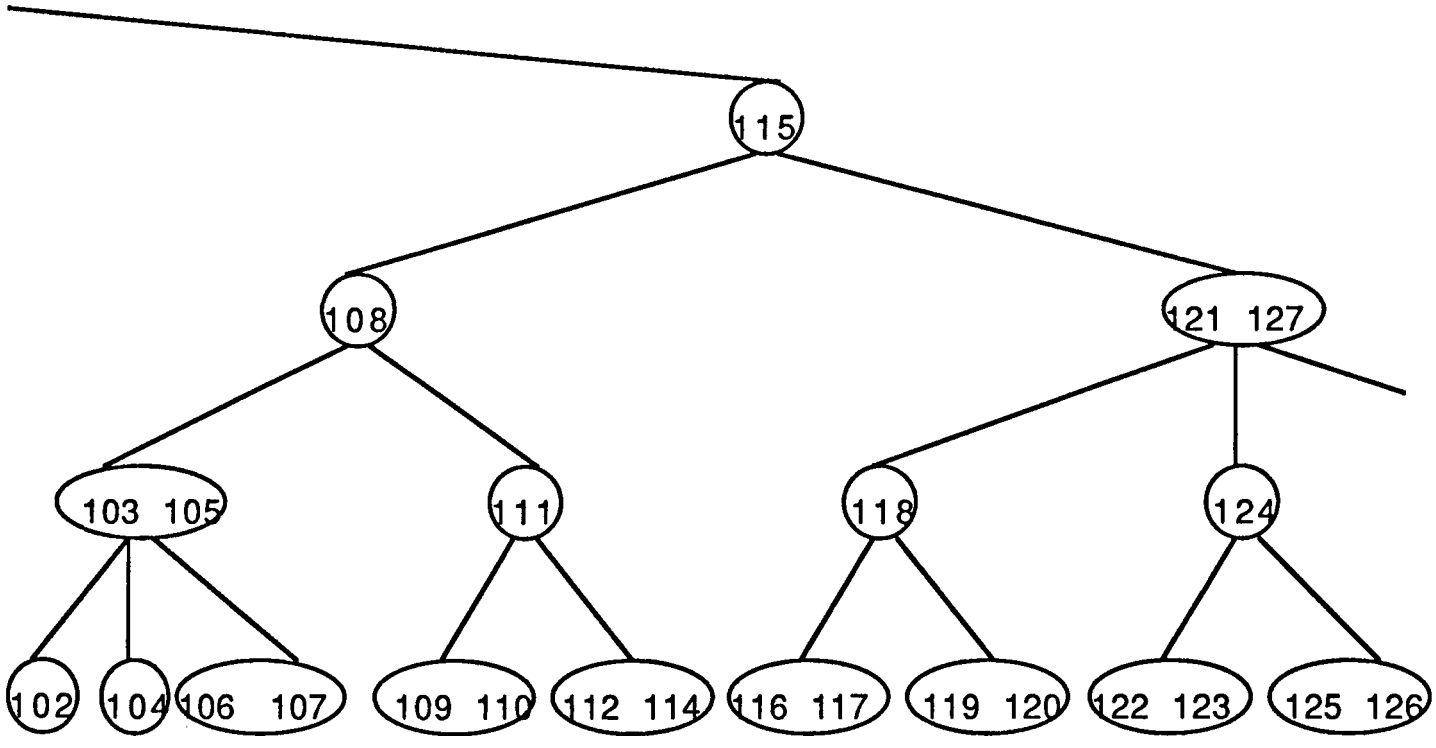

Figure 2.7 (cont.) Construction of H(3, 5, 2) with input data in Figure 2.6

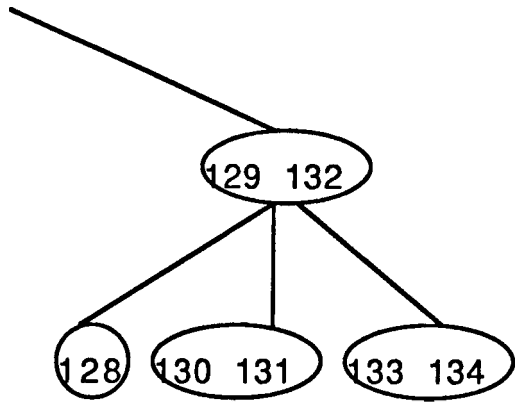

Figure 2.7 (cont.) Construction of H(3, 5, 2) with input data in Figure 2.6

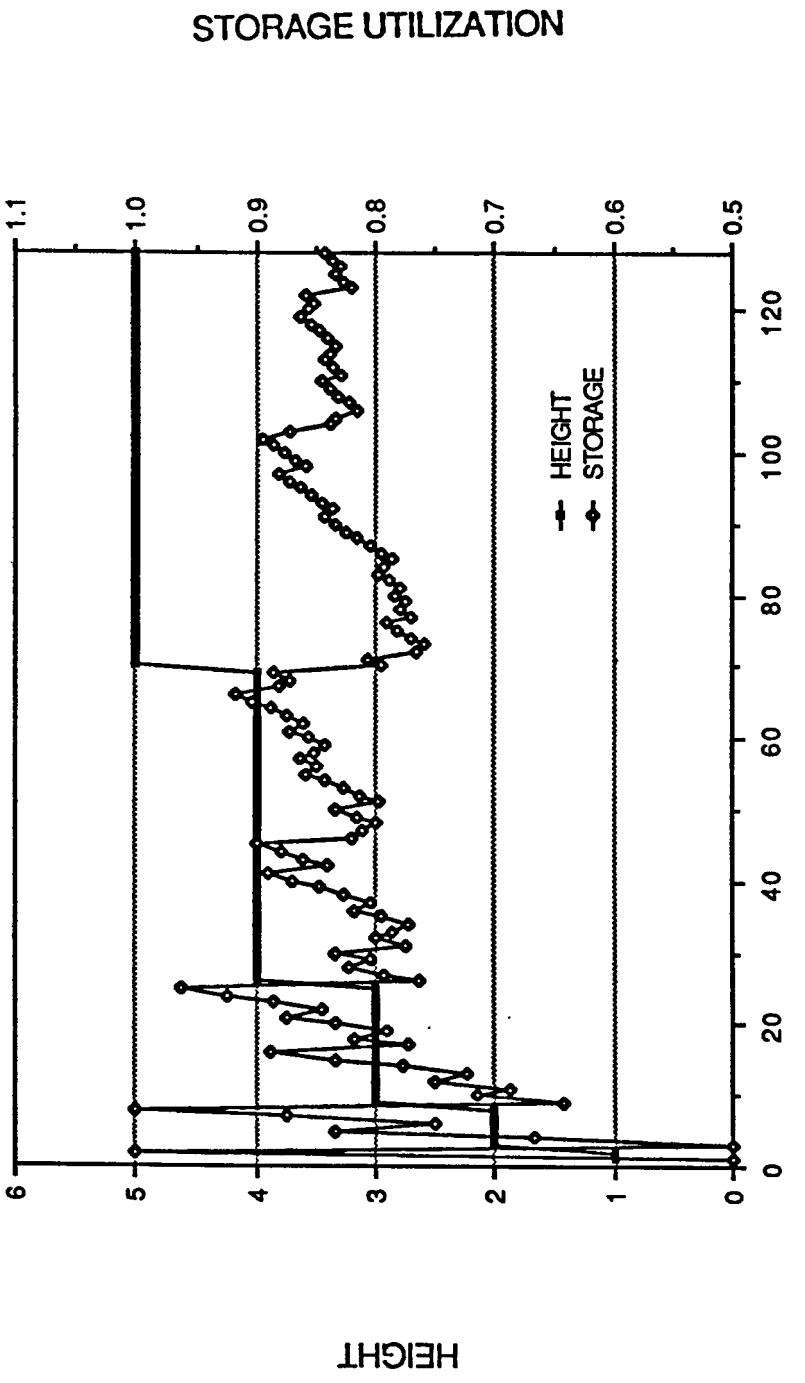

FIGURE 2.8 Relationship between height and storage utilization of Example 2.1

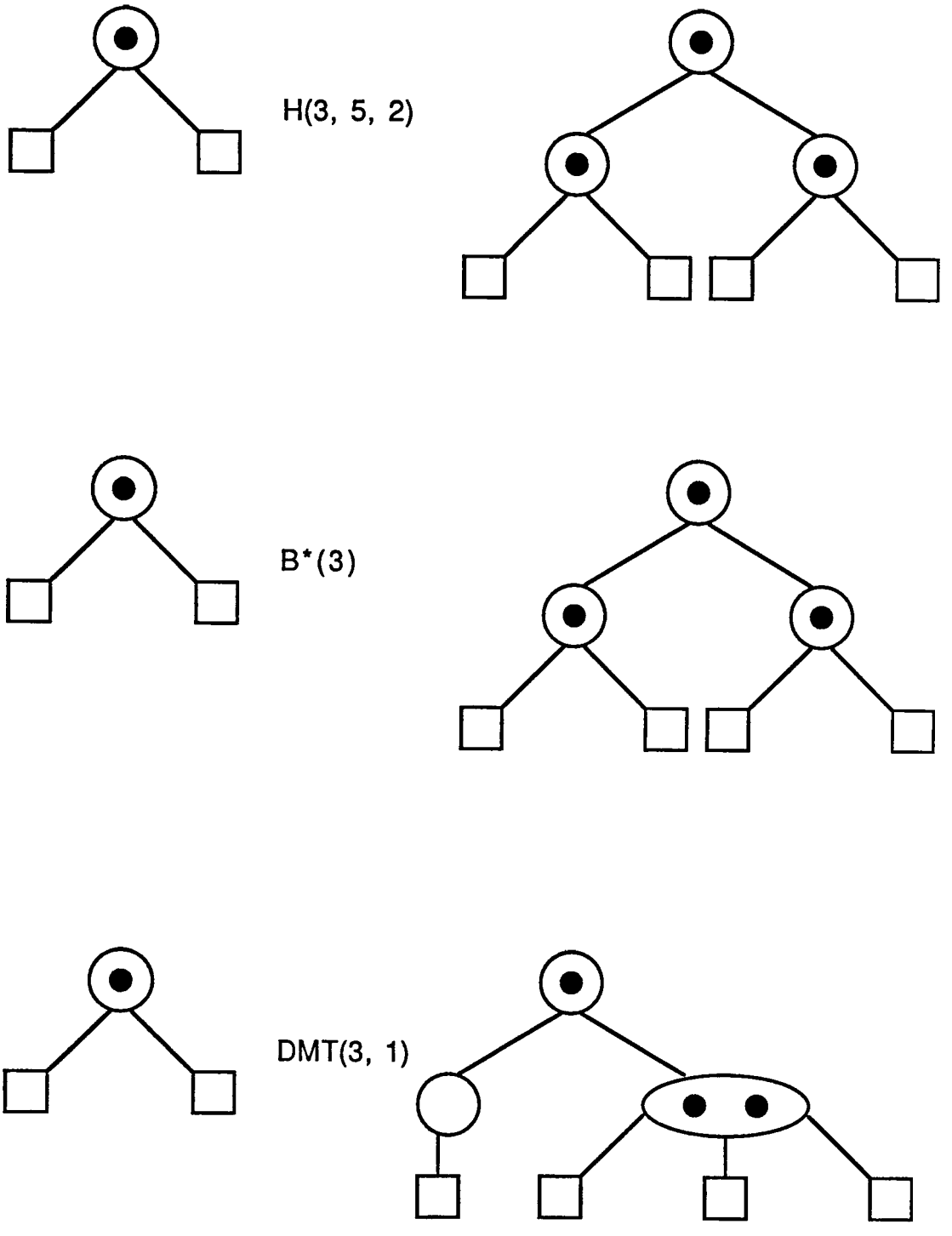

Figure 3.1 Three classes of trees with h=1, 2 and nk=1, 3

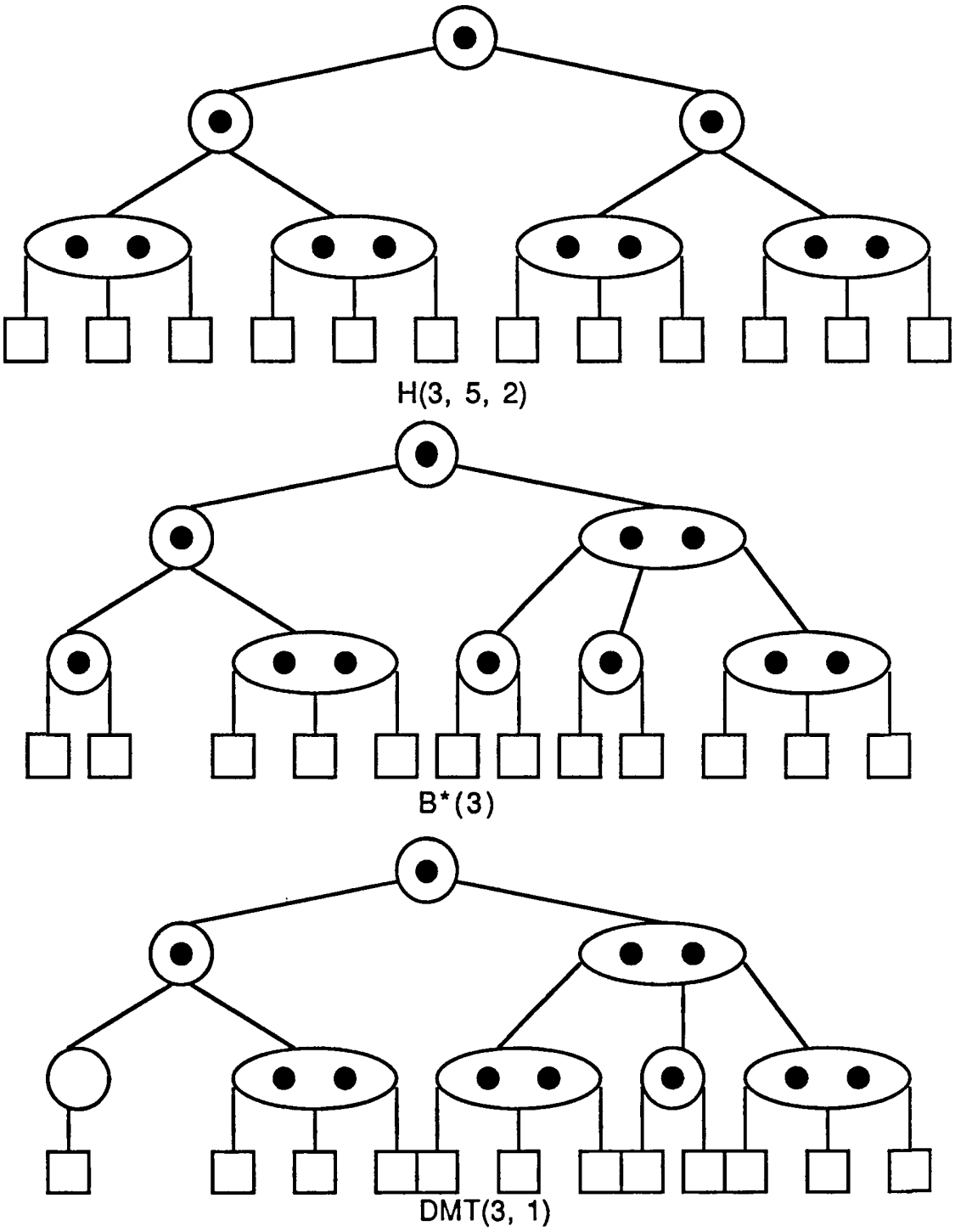

Figure 3.2 Three classes of trees with h=3 and nk=11

HEIGHT	NO. OF KEYS	NO. OF NODES	STORAGE UTILIZATION
1	1	1	0.5000
1	2	1	1.0000
2	3	3	0.5000
2	4	3	0.6667
2	5	3	0.8333
2	6	4	0.7500
2	7	4	0.8750
2	8	4	1.0000
3	9	7	0.6429
3	10	7	0.7143
3	11	7	0.7857
3	12	8	0.7500
3	13	9	0.7222
3	14	9	0.7778
3	15	9	0.8333
3	16	9	0.8889
3	17	11	0.7727
3	18	11	0.8182
3	19	12	0.7917
3	20	12	0.8333
3	21	12	0.8750
3	22	13	0.8462
3	23	13	0.8846
3	24	13	0.9231
3	25	13	0.9615
4	26	17	0.7647
4	27	17	0.7941
4	28	17	0.8235
4	29	18	0.8056
4	30	18	0.8333
4	31	20	0.7750
4	32	20	0.8000
4	33	21	0.7857
4	34	22	0.7727
4	35	22	0.7955
4	36	22	0.8182
4	37	23	0.8043
4	38	23	0.8261
4	39	23	0.8478
4	40	23	0.8696
4	41	23	0.8913
4	42	25	0.8400
4	43	25	0.8600
4	44	25	0.8800
4	45	25	0.9000
4	46	28	0.8214
4	47	29	0.8103
4	48	30	0.8000
4	49	30	0.8167
4	50	30	0.8333
4	51	32	0.7969
4	52	32	0.8125
4	53	32	0.8281
4	54	32	0.8438
4	55	32	0.8594
4	56	33	0.8485
4	57	33	0.8636
4	58	34	0.8529
4	59	35	0.8429
4	60	35	0.8571
4	61	35	0.8714
4	62	36	0.8611
4	63	36	0.8750
4	64	36	0.8889
4	65	36	0.9028
4	66	36	0.9167
4	67	38	0.8816
4	68	39	0.8718
4	69	39	0.8846

Table 2.1 Height, number of nodes, storage utilization of example 2.1

HEIGHT	NO. OF KEYS	NO. OF NODES	STORAGE UTILIZATION
5	70	44	0.7955
5	71	44	0.8068
5	72	47	0.7660
5	73	48	0.7604
5	74	48	0.7708
5	75	48	0.7813
5	76	48	0.7917
5	77	50	0.7700
5	78	50	0.7800
5	79	51	0.7745
5	80	51	0.7843
5	81	52	0.7788
5	82	52	0.7885
5	83	52	0.7981
5	84	53	0.7925
5	85	54	0.7870
5	86	54	0.7963
5	87	54	0.8056
5	88	54	0.8148
5	89	54	0.8241
5	90	54	0.8333
5	91	54	0.8426
5	92	55	0.8364
5	93	55	0.8455
5	94	55	0.8545
5	95	55	0.8636
5	96	55	0.8727
5	97	55	0.8818
5	98	57	0.8596
5	99	57	0.8684
5	100	57	0.8772
5	101	57	0.8860
5	102	57	0.8947
5	103	59	0.8729
5	104	62	0.8387
5	105	63	0.8333
5	106	65	0.8154
5	107	65	0.8231
5	108	65	0.8308
5	109	65	0.8385
5	110	65	0.8462
5	111	67	0.8284
5	112	67	0.8358
5	113	67	0.8433
5	114	68	0.8382
5	115	69	0.8333
5	116	69	0.8406
5	117	69	0.8478
5	118	69	0.8551
5	119	69	0.8623
5	120	70	0.8571
5	121	71	0.8521
5	122	71	0.8592
5	123	75	0.8200
5	124	75	0.8267
5	125	75	0.8333
5	126	77	0.8182
5	127	77	0.8247
5	128	78	0.8205

Table 2.1 (cont.) Height, number of nodes, storage utilization of example 2.1

GAMMA / DELTA	16	19	22	25
Height				
4	4	4	4	4
5	4	4	4	4
Average Storage Utilization				
4	0.8513	0.8573	0.8712	0.8936
5	0.8543	0.8597	0.8723	0.8965
Average Access Cost				
4	3.6797	3.6797	3.6611	3.6191
5	3.7236	3.6992	3.6748	3.6289
Average Accumulated Insertion Cost				
4	4.5938	4.7109	4.7598	4.9473
5	4.7578	4.7959	4.9063	5.1533

Table 2.2 H-trees with various degrees of delta and gamma

N= 1024, beta = 7

DELTA \ GAMMA	16	19	22	25
Height				
4	4	4	4	4
5	4	4	4	4
Average Storage Utilization				
4	0.8592	0.8560	0.8721	0.8922
5	0.8612	0.8601	0.8795	0.8990
Average Access Cost				
4	3.7080	3.7080	3.6650	3.6279
5	3.7080	3.7080	3.6641	3.6289
Average Accumulated Insertion Cost				
4	4.6396	4.6055	4.7178	4.9580
5	4.7686	4.7725	4.8809	5.0137

Table 2.3 H-trees with various degrees of delta and gamma

N= 1024, beta = 7

	H(3, 5, 2)	B*(3)	DMT(3, 1)
Height	log(N+1) 2+2log((N+1)/2 δ))	log(N+1) 1+log((N+1)/2))	log(N+1) 2log(N+1)+1
Storage Utilization	0.754	-	0.719 - 0.816
Access Cost	h-(h-1)/β h-1/ δ	-	-
Insertion Cost	2(h-1)+ β+(δ–1)/(β–δ)+1	-	-

Table 3.1 Theoretical estimates of height, storage utilization, access cost, and insertion cost of H(3, 5, 2), B*(3), and DMT(3,1).

DATA / H(3, 5, 2)	0	1	2	3	4	5	6	7	8	9	AVERAGE
Height	5	5	5	5	5	5	5	5	5	5	5
Average Storage Utilization	0.8241	0.8149	0.8118	0.8190	0.8292	0.8254	0.8346	0.8235	0.8109	0.8220	0.8216
Average Access Cost	4.1641	4.1719	4.1875	4.1563	4.1094	4.1484	4.1406	4.1719	4.1250	4.1484	4.1524
Average Accumulated Cost	4.9766	5.1641	5.1016	5.0547	5.0313	5.1406	5.0781	5.0234	5.0313	5.1094	5.0711

Table 3.2 Experimental results of 10 data, each with N = 128, by program HTREE(3, 5, 2)

DATA / B*(3)	0	1	2	3	4	5	6	7	8	9	AVERAGE
Height	5	5	5	5	5	5	5	5	5	5	5
Average Storage Utilization	0.7774	0.7898	0.7948	0.7794	0.7888	0.7873	0.7948	0.7961	0.7918	0.7906	0.7891
Average Access Cost	4.2969	4.2266	4.1953	4.2656	4.2500	4.2344	4.2656	4.2266	4.2109	4.2578	4.2430
Average Accumulated Cost	5.7578	5.7969	5.7109	5.8906	5.8438	5.8750	5.8516	5.7969	5.7578	5.7656	5.8047

Table 3.3 Experimental results of 10 data, each with N = 128, by program B*(3)

DATA / DMT(3, 1)	0	1	2	3	4	5	6	7	8	9	AVERAGE
Height	5	5	5	5	5	5	5	5	5	5	5
Average Storage Utilization	0.7774	0.7898	0.7948	0.7794	0.7888	0.7873	0.7948	0.7961	0.7918	0.7906	0.7891
Average Access Cost	4.2969	4.2266	4.1953	4.2656	4.2500	4.2344	4.2656	4.2266	4.2109	4.2578	4.2430
Average Accumulated Cost	5.7578	5.7969	5.7109	5.8906	5.8438	5.8750	5.8516	5.7969	5.7578	5.7656	5.8047

Table 3.4 Experimental results of 10 data, each with N = 128, by program DMT(3, 1)

	H(3, 5, 2)	B*(3)	DMT(3,1)
Height			
Theoretical	5 - 6	5 - 7	5 - 9
Experimental	5	5	5
Average Storage Utilization			
Theoretical	0.754	-	0.719 - 0.816
Experimental	0.82155	0.78908	0.81161
Average Access Cost			
Theoretical	3.667 - 4.50	-	-
Experimental	4.15235	4.24297	4.03750
Average Accumulated Insertion Cost			
Theoretical worst	13	15	-
Experimental	5.07111	5.80469	5.47971

Table 3.5 Tree Performance Comparisons derived from table 3.2, 3.3, and 3.4

	H(3, 5, 2)	B*(3)	DMT(3, 1)
Storage Utilization	1	0.9605	0.9880
Access Cost	1	1.0218	0.9723
Insertion Cost	1	1.1447	1.0806

Table 3.6 Comparisons in ratio derived from Table 3.5

DATA / H(3, 5, 2)	0	1	2	3	4	5	6	7	8	9	AVERAGE
Height	6	6	6	6	6	6	6	6	6	6	6
Average Storage Utilization	0.8338	0.8320	0.8147	0.8241	0.8295	0.8248	0.8240	0.8187	0.8188	0.8220	0.8238
Average Access Cost	4.8672	4.8438	4.9453	4.8516	4.8320	4.8477	4.8906	4.8706	4.8711	4.8867	4.8707
Average Accumulated Cost	5.9141	5.7930	5.8086	5.7930	5.8320	5.9258	5.7773	5.7059	5.8023	5.8203	5.8172

Table 3.7 Experimental results of 10 data, each with N = 256, by program HTREE(3, 5, 2)

DATA / B*(3)	0	1	2	3	4	5	6	7	8	9	AVERAGE
Height	6	6	6	6	6	6	6	6	6	6	6
Average Storage Utilization	0.7848	0.7989	0.7733	0.7978	0.7863	0.7891	0.7774	0.7771	0.7909	0.8043	0.7880
Average Access Cost	5.0273	4.9531	5.0352	4.9492	4.8789	5.0234	5.0898	5.0000	4.8867	4.9805	4.9824
Average Accumulated Cost	6.6055	6.5195	6.6992	6.6250	6.5820	6.6875	6.6797	6.6000	6.5625	6.6133	6.6174

Table 3.8 Experimental results of 10 data, each with N = 256, by program B*(3)

DATA / DMT(3, 1)	0	1	2	3	4	5	6	7	8	9	AVERAGE
Height	6	6	6	6	6	6	6	6	6	6	6
Average Storage Utilization	0.8185	0.8069	0.8199	0.8093	0.8098	0.8092	0.8226	0.8098	0.8167	0.8219	0.8147
Average Access Cost	4.7031	4.6875	4.7148	4.7383	4.6953	4.7070	4.7188	4.7020	4.7266	4.7109	4.7104
Average Accumulated Cost	6.2031	6.1719	6.2070	6.1992	6.1484	6.1797	6.2350	6.2745	6.3203	6.2268	6.2161

Table 3.9 Experimental results of 10 data, each with N = 256, by program DMT(3, 1)

	H(3, 5, 2)	B*(3)	DMT(3,1)
Height			
Theoretical	6 - 7	6 - 8	6 - 11
Experimental	6	6	6
Average Storage Utilization			
Theoretical	0.754	-	0.719 - 0.816
Experimental	0.82382	0.78799	0.81446
Average Access Cost			
Theoretical	4.333 - 5.50	-	-
Experimental	4.87066	4.98241	4.71043
Average Accumulated Insertion Cost			
Theoretical worst	15	18	-
Experimental	5.81723	6.61742	6.21612

Table 3.10 Tree Performance Comparisons derived from table 3.7, 3.8, and 3.9

	H(3, 5, 2)	B*(3)	DMT(3, 1)
Storage Utilization	1	0.9565	0.9886
Access Cost	1	1.0229	0.9671
Insertion Cost	1	1.1376	1.0686

Table 3.11 Comparisons in ratio derived from Table 3.10

APPENDIX A

Basic Subroutines Adapted by Program HTREE, DMT, and B*

The followings are some basic subroutines adpted by the preceding three programs: HTREE, DMT, and B*. For the purpose of briefness, only the functions of the subroutines and individual parameters are illustrated.

```
{------------------------------------------------------------ SHIFT
    Shift noshift sons from node Fromson which is the sonindex'th son of node
    Father to node Toson, where nodes Fromson and Toson are brother nodes
    with the same father Father.  If boolean varible right is true then Toson
    is on the r.h.s. of Fromson.  Otherwise, it is on the l.h.s. of Fromson.
------------------------------------------------------------}
PROCEDURE Shift(Fromson, Father:ptr; noshift,sonindex:integer; right:boolean);

{------------------------------------------------------------ MERGE
    Node Fromson which is the sonindex'th son of node Father is merged into
    its r.h.s. brother.
------------------------------------------------------------}
PROCEDURE Merge(Fromson, Father:ptr; sonindex:integer);

{------------------------------------------------------------ PACK
    Reduce the number of sons of node pn to m.
------------------------------------------------------------}
PROCEDURE Pack(PN:ptr; m:integer);

{------------------------------------------------------------ SPLIT
    Keep ii sons in p and split the remainings to the newly created node
    NewKey.
------------------------------------------------------------}
PROCEDURE Split(p:ptr; ii:integer; VAR Newkey:item);

{------------------------------------------------------------ INVERSE_PACK
    Increase the number of sons of node father to m.  Split the increased sons
    whose number is m-fathert.ns to the newly created node NewKey.
------------------------------------------------------------}
PROCEDURE Inverse_Pack(father:ptr; m:integer; VAR NewKey:item);

{------------------------------------------------------------ FOUNDBROTHER
    Among node a's sons, find a brother with index 'target' of a's sth son
    which satisfies the criteria.  If target is not found then target=-99 else
    variable right indicates which side of a's sth son target is located on.
------------------------------------------------------------}
PROCEDURE FoundBrother(a:ptr; s,criteria:integer; smaller:boolean;
                       VAR right:boolean; VAR target:integer);

{------------------------------------------------------------ FINDADJBROTHER
    Find an adjacent brother b of node a with index s which satisfies some
    criteria.  If found, then found:='true' and variable right indicates which
    side of a b is located on, else found:='false'.
------------------------------------------------------------}
PROCEDURE FindAdjBrother(a:ptr; s:integer; smaller:boolean; criteria:integer;
                         VAR found,right:boolean; VAR b:ptr);

{------------------------------------------------------------ MOVING
    Nodes with index s and target are two sons of node a.  This procedure
    moves 1 node out(/in) from s'th(/target'th) son to target'th(/s'th) son.
    Parameter targetright indicates the location of the node with index target.
------------------------------------------------------------}
PROCEDURE Moving(a:ptr; s,target:integer; targetright,out:boolean);
```

APPENDIX B

Main Part of Program HTREE - Construction of an H-tree

```
PROGRAM HTREE(input,output);
{Construction of H-tree}

{═══════════════════════════ INSERTION ═══════════════════════════}

{──────────────────────────────────────────────────────────── I1_STEP1
     Insert phase1 - step 1
──────────────────────────────────────────────────────────────────}
PROCEDURE I1_Step1(a:ptr; u:item; r:integer);

VAR     i:integer;

BEGIN
   WITH at DO
   BEGIN
      AddSon(a,1);
      FOR i:=ns DOWNTO r+2 DO son[i]:=son[i-1];
      son[r+1]:=u;
      IF u.p<>nil THEN u.pt.fptr:=a;
   END;
END; {I1_Step1}


{──────────────────────────────────────────────────────────── I2_STEP3
     Insert phase2 - step 3
──────────────────────────────────────────────────────────────────}

PROCEDURE I2_Step3(a:ptr; s:integer; VAR found:boolean);

VAR     right   : boolean;
        brother : ptr;

BEGIN
   FindAdjBrother(a,s,true,nn,found,right,brother);
   IF found Then Shift(a,at.fptr,1,s,right);
END; {I2_Step3}


{──────────────────────────────────────────────────────────── I2_STEP7
     Insert phase2 - step 7
──────────────────────────────────────────────────────────────────}

PROCEDURE I2_Step7(VAR a,b:ptr; s:integer; VAR w:item);

VAR     left    : ptr;

BEGIN
   IF s=at.fptr.ns THEN
   BEGIN
      Shift(a,at.fptr,at.ns-n,s,false);
      left:=b;
   END
   ELSE
   BEGIN
      Shift(b,bt.fptr,bt.ns-n,s+1,false);
      left:=a;
   END;
```

```
   IF odd(nn) THEN Split(left,n-1,w) ELSE Split(left,n,w);
END; {I2_STEP7}


{------------------------------------------------------------------ SEARCH
     Search key x on H-tree with root a; if not found, insert an item with key
     x in tree. If an item emerges to be passed to a lower level, then assign
     it to v; toph:="tree a has become heigher"; h:='continue'.
-------------------------------------------------------------------------}

PROCEDURE Search(x:integer; a:ptr; VAR toph,h:boolean; VAR v:item);

VAR     k,l,r   : integer;
        q       : ptr;

        PROCEDURE Insert_Phase2(a:ptr; VAR NewKey:item; i:integer);
        VAR     splitable       : boolean;
                Emptybrother    : boolean;
                ags,bgs         : integer;
                b               : ptr;
                l1,r1,k1        : integer;
        BEGIN
           with at do
           begin
                I1_Step1(a,NewKey,l);                                   {I2_Step1}
                IF ns<=nn-1 Then h:=false
                Else
                  IF fptr<>nil then
                  Begin
                   BinarySearch(fptr,son[1].key,l1,r1,k1);
                   TestSplit(a,splitable,ags);
                   IF (splitable) Then Split(a,n-1,Newkey)
                   Else
                   begin
                      I2_Step3(a,r1,Emptybrother);                      {I2_Step3}
                      IF (Emptybrother) then h:=false
                      Else
                         IF ags<=(nn*2) then
                         begin                                          {I2_Step4}
                            Pack(a,nn,false);
                            h:=false;
                         end
                         Else
                         begin                                          {I2_Step5}
                            IF r1=fptrt.ns then
                            begin
                               IF r1=1 then b:=fptrt.p0
                                       else b:=fptrt.son[r1-1].p;
                               shift(a,fptr,1,r1,false)
                            end
                            else
                            begin
                               b:=fptrt.son[r1+1].p;
                               shift(a,fptr,1,r1,true)
                            end;
                            TestSplit(b,splitable,bgs);
                            IF (splitable) Then
                            begin
                               Split(b,n-1,NewKey);
```

```
                        IF r1=fptrt.ns then r1:=r1-1 else r1:=r1+1;
                      end
                      else                                              {I2_Step6}
                        IF bgs<=(nn**2) then
                        begin
                           pack(b,nn,false);
                           h:=false;
                        end
                        else
                        begin
                           I2_Step7(a,b,r1,Newkey);                     {I2_Step7}
                           IF r1=at.fptrt.ns
                           then Insert_Phase2(bt.fptr,NewKey,r1-1)
                           else Insert_Phase2(bt.fptr,NewKey,r1);
                        end
                   end
               end
               End
               Else TopRoot(a,toph,h,v);
            IF h then
               Insert_Phase2(fptr,NewKey,r1);
      end;
   END;{Insert_Phase2}


   PROCEDURE Insert_Phase1;
   VAR     m,i1,r1,k1,BrotherGot   : integer;
           o                       : real;
           tf_right                : boolean;
           Newkey1                 : item;
   BEGIN                              {insert u to the right of at.son[r]}
           Toph:=false;
           WITH at DO
           BEGIN
{I1_STEP1}
              I1_Step1(a,v,r);                                          {I1_Step1}
              IF ns<=nn-1 THEN h:=false
              ELSE
                IF fptr<>nil then
                BEGIN
                   {node at is full; find a nonfull brother to shift}
                   BinarySearch(fptr,son[i].key,i1,r1,k1);
{I1_STEP2}
                   Foundbrother(fptr,r1,nn,true,tf_right,BrotherGot);
                   IF BrotherGot>=0 then
                   {nonfull brother found}
                   begin
{I1_STEP3}
                      Moving(fptr,r1,BrotherGot,tf_right,true);
                      h:=false;
                   end
                   ELSE                          {no nunfull brother exists}
{I1_STEP4}
                   BEGIN
                      o:=(Delta-1)/(nn-Delta);
                      m:=(Delta-1) div (nn-Delta);
                      IF m<0 then m:=1;
                      IF o>m then m:=m+1;
                      IF fptrt.nos>m then                {brother no > m}
```

```
                        I1_Step4(fptr,r1,m,NewKey1)
                     Else                                    {brother no <= m}
                     BEGIN
                        Inverse_Pack(fptr,fptr^.ns+2,NewKey1);
                        r1:=fptr^.ns;
                     END;
                  END;
               END
               ELSE TopRoot(a,toph,h,v);
               IF h and (at.fptr<>nil) then
                  Insert_Phase2(fptr,NewKey1,r1);
            END;
      END; {Insert_Phase1}


BEGIN  {search key x on node at; h=false}
   IF a=nil THEN
   BEGIN {item with key x is not in tree}
      IF root=nil then toph:=true else toph:=false;
      h:=true;
      WITH v DO
      BEGIN
         key:=x;
         P:=nil
      END;
   END
   ELSE
   WITH at DO
   BEGIN
      BinarySearch(a,x,l,r,k);
      IF l-r>1 THEN
      BEGIN {found}
         found:=true;
         h:=false
      END
      ELSE
      BEGIN  {item is not on this node}
         IF r=0 THEN q:=p0
         ELSE q:=son[r].p;
         Search(x,q,toph,h,v);
         IF h THEN Insert_Phase1;
      END
   END;
END {search};


{═══════════════════════════ DELETETION ═══════════════════════════}


PROCEDURE D1_Step2(a:ptr; s:integer; VAR found:boolean);

VAR     target  : integer;
        right   : boolean;

BEGIN
   FoundBrother(a,s,Delta,false,right,target);
   IF target>=0 THEN found:=true ELSE found:=false;
```

```
   IF found THEN Moving(a,s,target,right,false);
END; {D1_Step2}


{------------------------------------------------------------------ D1_STEP3}


PROCEDURE D1_Step3(a:ptr; VAR h:boolean);
VAR     r       : real;
        w       : integer;
BEGIN
   r:=(nn-1)/(nn-Delta);
   w:=(nn-1) div (nn-Delta);
   IF r<>w THEN w:=w+1;
   IF at.ns>=w THEN PACK(a,at.ns,false);
END; {D1_Step3}


{------------------------------------------------------------------ D1_STEP5}


PROCEDURE D1_Step5(a:ptr; VAR h:boolean);
VAR     gs              : integer;
BEGIN
   ComputeGS(a,gs);
   IF gs<Gama THEN h:=true              {D1_Step6}
   Else h:=false;
END; {D1_Step5}


{------------------------------------------------------------------ D2_STEP2}


PROCEDURE D2_Step2(a:ptr; s:integer; VAR found:boolean);
VAR     b               : ptr;
        right           : boolean;
        bgs             : integer;
BEGIN
   FindAdjBrother(a,s,false,n,found,right,b);
   IF found THEN
   begin
      ComputeGS(b,bgs);
      IF right THEN bgs:=bgs-bt.p0^.ns-1 ELSE bgs:=bgs-bt.son[bt.ns].pt.ns-1;
      IF bgs>=Gama THEN
         IF right THEN Shift(b,bt.fptr,1,s+1,false)             {brother found}
                  ELSE Shift(b,bt.fptr,1,s-1,true)
      Else
        IF (right) and (s<>0) THEN {try the left brother}
           BEGIN
              IF s=1 THEN b:=at.fptrt.p0 ELSE b:=at.fptrt.son[s-1].p;
              IF bt.ns>n THEN
              BEGIN
                 ComputeGS(b,bgs);
                 bgs:=bgs-bt.son[bt.ns].pt.ns-1;
```

```
                IF bgs>=Gama THEN                    {left brother is desired}
                   Shift(b,bt.fptr,1,s-1,true)
                ELSE found:=false                  {left brother is not desired}
             END
             ELSE found:=false;
          END
       ELSE found:=false;       {bothe right and left brothers are not desired}
   END;
END; {D2_Step2}


{------------------------------------------------------------ D2_STEP3}


PROCEDURE D2_Step3(VAR a:ptr; b:ptr; s:integer; VAR notq0:boolean;
                   VAR replacedby:ptr);
BEGIN
   IF s=at.fptrt.ns THEN
   BEGIN
      MERGE(b,bt.fptr,s-1);
      IF (at.fptrt.ns=0) AND (at.fptrt.fptr=nil) THEN
      BEGIN
         at.fptr:=nil;
         notq0:=true;
         replacedby:=a;
      END
      ELSE IF s-1=0 THEN at.fptrt.p0:=a ELSE at.fptrt.son[s-1].p:=a;
      IF at.ns>nn-1 THEN PACK(a,nn,false);
   END
   ELSE
   BEGIN
      MERGE(a,at.fptr,s);
      IF (bt.fptrt.ns=0) AND (bt.fptrt.fptr=nil) THEN
      BEGIN
         bt.fptr:=nil;
         notq0:=true;
         replacedby:=b;
      END
      ELSE IF s=0 THEN at.fptrt.p0:=b ELSE at.fptrt.son[s].p:=b;
      IF bt.ns>nn-1 THEN PACK(b,nn,false);
   END;
END; {D2_Step3}


{------------------------------------------------------------ D2_STEP5}


PROCEDURE D2_Step5(a,b:ptr; s:integer);
BEGIN
   IF s=at.fptrt.ns THEN
   BEGIN
      MERGE(b,bt.fptr,s-1);
      s:=s-1;
   END
   ELSE
   BEGIN
```

```
      MERGE(a,at.fptr,s);
      a:=b;
   END;
   PACK(a,nn+1,false);
   IF s=0 THEN Shift(a,at.fptr,1,0,true) ELSE Shift(a,at.fptr,1,s,false);
END; {D2_Step5}


{------------------------------------------------------------------ D2_STEP6}


PROCEDURE D2_Step6(a,b,b1:ptr; s:integer);
BEGIN
   IF s=at.fptrt.ns THEN
   BEGIN
      Shift(b1,b1t.fptr,2,s-2,true);
      Shift(b,bt.fptr,1,s-1,true);
   END
   ELSE IF s=0 Then
        BEGIN
           Shift(b1,b1t.fptr,2,s+2,false);
           Shift(b,bt.fptr,1,s+1,false);
        END
        ELSE
          Shift(b1,b1t.fptr,1,s-1,true);
END; {D2_Step6}


{------------------------------------------------------------------ DELETE}


PROCEDURE Delete(x:integer; a:ptr; s:integer; VAR h:boolean; VAR notq0:boolean;
                 VAR replacedby:ptr);

{search and delete key x in b-tree a; if a node underflow is necessary, balance
 with adjacent node if possible, otherwise merge; h:="node a is undersize"}

VAR     i,k,l,r : integer;
        q       : ptr;

        PROCEDURE Delete_Phase2(a:ptr; VAR h:boolean);
        VAR     ags     : integer;

                PROCEDURE D2STEP2;
                VAR     BrotherFound    : boolean;
                        b,b1            : ptr;
                        bgs             : integer;
                        l1,r1,k1        : integer;
                        nk              : item;

                BEGIN
                   BinarySearch(at.fptr,at.son[1].key,l1,r1,k1);
                   D2_Step2(a,r1,BrotherFound);                    {D2_Step2}
                   IF BrotherFound
                   THEN {D2_STEP7}
                   ELSE
```

```
            BEGIN
               IF r1=at.fptrt.ns                                        {D2_Step3}
               THEN IF r1=1 THEN b:=at.fptrt.p0
                            ELSE b:=at.fptrt.son[r1-1].p
               ELSE b:=at.fptrt.son[r1+1].p;
               Computegs(a,ags);
               Computegs(b,bgs);
               IF ags+bgs<=nn*nn
               THEN D2_Step3(a,b,r1,notq0,replacedby)
               ELSE
                 IF n>=2 THEN
                 BEGIN                                                  {D2_Step4}
                    IF r1=0 THEN b1:=at.fptrt.son[2].p
                    ELSE IF r1=1 Then b1:=at.fptrt.p0
                                 Else IF r1=at.fptrt.ns
                                      Then IF r1=2
                                           then b1:=at.fptrt.p0
                                           else b1:=at.fptrt.son[r1-2].p
                                      else b1:=at.fptrt.son[r1-1].p;
                    Computegs(b1,b1t.ngs);
                    IF (b1t.ns<nn-1) or (b1t.ngs<nn*(nn-1)) Then
                    BEGIN
                       IF b1t.ns<nn-2 then
                       begin
                          INVERSE_PACK(b1,nn-1,nk);
                          I1_Step1(b1,nk,b1t.ns);
                       end
                       Else IF b1t.ns>nn-2 then PACK(b1,nn-1,false);
                       D2_Step5(a,b,r1);                                {D2_Step5}
                    END
                    Else
                       D2_Step6(a,b,b1,r1)                              {D2_Step6}
                 end;
            end;
          END; {D2STEP2}


BEGIN
   IF at.ns<n-1 Then D2STEP2;                                           {D2_Step1}
   Computegs(a,ags);
   IF ags<Gama Then
      IF at.fptr=nil Then h:=false Else D2STEP2;
   IF at.fptr=nil Then
      IF at.ns=0 then h:=true
      else IF notq0 then h:=true else h:=false
   Else IF at.fptr<>root Then Delete_Phase2(at.fptr,h);
END; {Delete_Phase2}


PROCEDURE Delete_Phase1(a:ptr; s:integer; VAR h:boolean);
VAR     BrotherFound                    : boolean;
        gs                              : integer;
BEGIN
   IF at.ns<Delta-1 then                                                {D1_Step1}
   begin
      D1_Step2(at.fptr,s,BrotherFound);                                 {D1_Step2}
      IF (BrotherFound)
      then h:=false
      else
      begin
```

```
                D1_Step3(at.fptr,h);                                    {D1_Step3}
                IF at.fptrt.fptr<>nil then
                begin
                   ComputeGS(at.fptrt.fptr,gs);                         {D1_Step4}
                   IF gs>=Gama
                   Then D1_Step5(at.fptr,h)
                   else h:=true;
                end
                Else D1_Step5(at.fptr,h);
             end;
          end
          Else
            D1_Step5(at.fptr,h);                                        {D1_Step5}
          IF (at.fptrt.ns=0) and (at.fptrt.fptr=nil) Then
          begin
             at.fptrt.p0:=a;
             h:=true;
          end
          Else
          IF h Then IF at.fptr<>root then Delete_Phase2(at.fptr,h);     {D1_Step6}
       END; {Delete_Phase1}


       PROCEDURE del(p:ptr; VAR father:ptr; VAR h:boolean);
       VAR     q       : ptr;
               i       : integer;
       BEGIN
          with pt do
          begin
             q:=son[pt.ns].p;
             IF q<>nil then
                del(q,p,h);
             Else
             begin
                at.son[k].key:=pt.son[pt.ns].key;
                pt.fptr:=father;
                for i:=1 to fathert.ns do fathert.son[i].pt.fptr:=father;
                Subson(p,1);
                h:=pt.ns<DELTA-1;
                IF h then IF fptr=a then Delete_Phase1(p,r,h)
                                    else Delete_Phase1(p,pt.fptrt.ns,h);
             end
          end
       END; {del}

BEGIN {Delete}
   IF a=nil THEN
   BEGIN
      writeln('KEY IS NOT IN TREE');
      h:=false
   END
   ELSE
   WITH at DO
   BEGIN
      BinarySearch(a,x,l,r,k);
      IF r=0 THEN q:=p0 ELSE q:=son[r].p;
      IF l-r>1 THEN
      BEGIN {found, now delete son[k]}
         found:=true;
```

```
        IF q=nil THEN
        BEGIN                                                {a is a terminal node}
           SubSon(a,1);
           IF (p0<>nil) and (fptr<>nil) THEN h:=false ELSE h:=ns<n-1;
           FOR i:=k TO ns DO son[i]:=son[i+1];
           IF at.fptr<>nil then Delete_Phase1(a,s,h);
        END
        ELSE
           del(q,a,h);
     END
     ELSE
        Delete(x,q,r,h,notq0,replacedby);
  END
END {Delete};


{═══════════════════════════════════════════════════════════════════════}


BEGIN
  root:=nil;
  InputParameter;
  error:=false;
  read(x);
  WHILE (x<>0) DO
  BEGIN
     found:=false;
     search(x,root,toph,h,u);
     IF toph THEN
     BEGIN {insert new base node}
        q:=root;
        new(root);
        WITH roott DO
        BEGIN
           ns:=1;
           IF q<>nil then nos:=ns+1;
           p0:=q;
           son[1]:=u;
           IF p0<>nil then
           begin
              p0t.fptr:=root;
              son[1].pt.fptr:=root;
           end;
        END
     END;
     Printtree(root,1,error);
     read(x);
  END;
  read(x);
  WHILE (X<>0) DO
  BEGIN
     found:=false;
     notq0:=false;
     Delete(x,root,s,h,notq0,replacedby);
     IF h THEN
     BEGIN {base node size was reduced}
        IF roott.ns=0 THEN
        BEGIN
           q:=root;
           IF notq0 THEN root:=replacedby ELSE root:=qt.p0; {dispose(q)}
           IF (root<>nil) THEN IF (roott.fptr<>nil) THEN roott.fptr:=nil;
        END
     END;
     Printtree(root,1,error);
     read(x);
  END;
  PrintTable;
END.
```

APPENDIX C

Main Part of Program B* - Construction of an B*-tree

```
PROGRAM B*(input,output);
{Construction of B*- tree}


PROCEDURE Step1(a:ptr; u:item; r:integer);

VAR     i:integer;

BEGIN
   WITH at DO
   BEGIN
      AddSon(a,1);
      FOR i:=ns DOWNTO r+2 DO son[i]:=son[i-1];
      son[r+1]:=u;
      IF u.p<>nil THEN u.pt.fptr:=a;
   END;
END; {Step1}


{------------------------------------------------------------------ TOPROOT}


PROCEDURE TopRoot(a:ptr; VAR h,toph:boolean; VAR NewKey:item);
BEGIN
   IF odd(nn) THEN split(a,n-1,NewKey) ELSE split(a,n,NewKey);
   h:=false;
   toph:=true;
END; {TopRoot}


{------------------------------------------------------------------ SEARCH
     Search key x on B* tree with root a; if not found, insert an item with
     key x in tree. If an item emerges to be passed to a lower level, then
     assign it to v; h:='continue' or 'tree has become higher'.
-------------------------------------------------------------------------}

PROCEDURE Search(x:integer; a:ptr; VAR h,toph:boolean; VAR v:item);

VAR     k,l,r   : integer;
        q       : ptr;


        PROCEDURE Insert(a:ptr; VAR NewKey:item; i:integer);

        VAR     l1,r1,k1,shiftno        : integer;
                Bfound,tf_right         : boolean;
                Bgot,b                  : ptr;

        BEGIN                           {insert newkey to the right of at.son[r]}
                WITH at DO
                BEGIN
{STEP1}
                   Step1(a,Newkey,i);                                   {Step1}
                   IF ns<=nn-1 THEN h:=false
                   ELSE
                     IF a<>root then
                     BEGIN
```

```
                        ELSE
                        BEGIN
{STEP5}
                           q:=root;
                           new(root);
                           WITH root↑ DO
                           BEGIN
                              ns:=0;
                              nos:=1;
                              p0:=q;
                              p0↑.fptr:=root;
                           END;
                           SPLITOneL(a,NewKey,NewNode,InsL);
                           r1:=0;
                        END;
                        IF h THEN Insert(fptr,NewKey,NewNode,Insl,r1);
                     END
          END; {Insert}


BEGIN  {search key x on node at; h=false}
   IF a=nil THEN
   BEGIN {item with key x is not in tree}
      h:=true;
      WITH v DO
      BEGIN
         key:=x;
         P:=nil
      END;
   END
   ELSE
   WITH at DO
   BEGIN
      BinarySearch(a,x,l,r,k);
      IF l-r>1 THEN
      BEGIN {found}
         found:=true;
         h:=false
      END
      ELSE
      BEGIN  {item is not on this node}
         IF r=0 THEN q:=p0
         ELSE q:=son[r].p;
         IF (q<>nil) AND (q↑.ns=0) THEN
            IF q↑.p0<>nil THEN
            BEGIN
               q:=q↑.p0;
               r:=0;
            END
            ELSE
            BEGIN
               WHILE (q↑.ns=0) AND (r<>ns) DO
               BEGIN
                  r:=r+1;
                  y:=son[r].key;
                  son[r].key:=x;
                  x:=y;
                  q:=son[r].p;
               END;
               IF (r=ns) AND (q↑.ns=0) THEN
               WHILE (q↑.ns=0) AND (r>=0) DO
               BEGIN
                  y:=son[r].key;
                  son[r].key:=x;
                  x:=y;
                  r:=r-1;
                  IF r=0 THEN q:=p0 ELSE q:=son[r].p;
```

```
            END;
         END;
      Search(x,q,h,v);
      insl:=false;
      IF h THEN Insert(a,v,newnode,insl,r);
   END
END
END {search};

{═══════════════════════════════════════════════════════════════════}


BEGIN
   root:=nil;
   InputParameter;
   error:=false;
   read(x);
   WHILE (x<>0) DO
   BEGIN
      found:=false;
      search(x,root,h,u);
      IF h THEN
      BEGIN {insert new base node}
         q:=root;
         new(root);
         WITH root↑ DO
         BEGIN
            ns:=1;
            IF q<>nil then nos:=ns+1;
            p0:=q;
            son[1]:=u;
            IF p0<>nil then
            begin
               p0↑.fptr:=root;
               son[1].pt.fptr:=root;
            end;
         END
      END;
      Printtree(root,1,error);
      read(x);
   END;
   PrintTable;
END.
```

APPENDIX D

Main Part of Program DMT - Construction of an DMT-tree

```
PROGRAM DMT(input,output);
{Construction of Dense Multiway Tree}

{------------------------------------------------------------------ SIDESAT
    Ensure the lrightmost(/leftmost) son of p is saturated
------------------------------------------------------------------}

PROCEDURE SideSat(p:ptr; tf_r:boolean);
VAR     q,Bq    : ptr;
        index,i : integer;
BEGIN
   IF pt.p0<>nil THEN
   BEGIN
      IF tf_r THEN
      BEGIN
         q:=pt.son[pt.ns].p;
         index:=pt.ns-1;
      END
      ELSE
      BEGIN
         q:=pt.p0;
         index:=1;
      END;
      Bq:=pt.son[index].p;
      IF qt.ns<nn-1 THEN
         IF qt.ns=0 THEN
         BEGIN
            SideSat(Bq,(NOT tf_r));
            SHIFT(Bq,p,nn-1,index,tf_r);
         END
         ELSE
         FOR i:=1 TO nn-qt.ns-1 DO
         BEGIN
            SideSat(Bq,tf_r);
            SHIFT(Bq,p,1,index,tf_r);
         END;
   END;
END; {SideSat}


{================================ INSERTION ================================}


PROCEDURE Step1(a:ptr; u:item; newnode:ptr; VAR insl:boolean; r:integer);

VAR     i:integer;

BEGIN
   WITH at DO
   BEGIN
      AddSon(a,1);
      FOR i:=ns DOWNTO r+2 DO son[i]:=son[i-1];
      son[r+1]:=u;
      IF insl THEN
      BEGIN
         IF r=0 THEN p0:=newnode ELSE son[r].p:=newnode;
         IF newnode<>nil THEN newnodet.fptr:=a;
```

```
        insl:=false;
      END;
      IF u.p<>nil THEN u.pt.fptr:=a;
   END;
END; {Step1}


{------------------------------------------------------------------ SEARCH
    Search key x on DMT tree with root a; if not found, insert an item with
    key x in tree. If an item emerges to be passed to a lower level, then
    assign it to v; h:='continue' or 'tree has become heigher'.
------------------------------------------------------------------------}

PROCEDURE Search(x:integer; a:ptr; VAR h:boolean; VAR v:item);

VAR     k,l,r,y : integer;
        q       : ptr;
        newnode : ptr;
        insl    : boolean;

        PROCEDURE Insert(a:ptr; VAR NewKey:item; VAR NewNode:ptr;
                         VAR InsL:boolean; i:integer);

        VAR     l1,r1,k1,BrotherGot,t   : integer;
                tf_right,stop           : boolean;
                q                       : ptr;

        BEGIN                           {insert newkey to the right of at.son[r]}
                WITH at DO
                BEGIN
{STEP1}
                   Step1(a,Newkey,NewNode,InsL,i);                      {Step1}
                   IF ns<=nn-1 THEN h:=false
                   ELSE
                     IF a<>root then
                     BEGIN
                        {node at is full; find a nonfull brother to shift}
                        BinarySearch(fptr,son[1].key,l1,r1,k1);
                        IF r1=0 THEN t:=fptrt.p0t.ns
                                ELSE t:=fptrt.son[r1].pt.ns;
                        IF t=0 THEN
                        BEGIN
                           IF (r1=0) AND (fptrt.p0t.ns=0) THEN r1:=r1+1;
                           WHILE (fptrt.son[r1].pt.ns=0) AND (r1<>fptrt.ns) DO
                                 r1:=r1+1;
                           IF (r1=fptrt.ns) AND (fptrt.son[r1].pt.ns=0) THEN
                           BEGIN
                              stop:=false;
                              WHILE (not stop) DO
                                 IF fptrt.son[r1].pt.ns=0 THEN r1:=r1-1
                                 ELSE stop:=true;
                           END;
                        END;
{STEP2}
                        FoundBrother(fptr,r1,nn,true,tf_right,BrotherGot);
                        IF BrotherGot>=0 THEN
                        {nonfull brother found}
                        BEGIN
                           Moving(fptr,r1,BrotherGot,tf_right,true);
                           h:=false;
                        END
                        ELSE                            {no nunfull brother exists}
{STEP3}
                          IF fptrt.ns<nn-1 THEN SPLITOneL(a,NewKey,NewNode,InsL)
                          ELSE
{STEP4}
                            IF r1<>fptrt.ns
                            THEN SPLIT(a,ns-1,NewKey)
                            ELSE SPLITOneL(a,NewKey,NewNode,InsL);
                     END
```

```
                        {node at is full; find a nonfull brother to shift}
                        BinarySearch(fptr,son[1].key,l1,r1,k1);
{STEP2}
                        FindAdjBrother(a,r1,true,nn,BFound,tf_right,BGot);
                        IF BFound THEN
                        {nonfull brother found}
                        BEGIN
                           shiftno:=nn-((nn+Bgott.ns) div 2);
                           SHIFT(a,fptr,shiftno,r1,tf_right);
                           h:=false;
                        END
                        ELSE                              {no nunfull brother exists}
{STEP3}
                        BEGIN
                           IF r1=fptrt.ns THEN
                           BEGIN
                              IF r1=1 THEN b:=fptrt.p0
                                      ELSE b:=fptrt.son[r1-1].p;
                              SHIFT(a,fptr,nn-(2*nn-2) div 3,r1,false);
                           END
                           ELSE
                           BEGIN
                              b:=fptrt.son[r1+1].p;
                              SHIFT(a,fptr,nn-(2*nn-2) div 3,r1,true);
                           END;
                           SPLIT(b,(2*nn-1) div 3, NewKey);
                           IF r1=fptrt.ns THEN r1:=r1-1 ELSE r1:=r1+1;
                           INSERT(fptr,NewKey,r1);
                        END;
                     END
                     ELSE
{STEP5}
                       IF ns>(2*(2*nn-2) div 3)
                       THEN TopRoot(a,h,toph,v) ELSE h:=false;
                END
        END; {Insert_Phase1}


BEGIN  {search key x on node at; h=false}
   IF a=nil THEN
   BEGIN {item with key x is not in tree}
      IF root=nil THEN toph:=true ELSE toph:=false;
      h:=true;
      WITH v DO
      BEGIN
         key:=x;
         P:=nil
      END;
   END
   ELSE
   WITH at DO
   BEGIN
      BinarySearch(a,x,l,r,k);
      IF l-r>1 THEN
      BEGIN {found}
         found:=true;
         h:=false
      END
      ELSE
      BEGIN  {item is not on this node}
         IF r=0 THEN q:=p0
         ELSE q:=son[r].p;
         Search(x,q,h,toph,v);
         IF h THEN Insert(a,v,r);
      END
   END
END {search};
```

```
{═══════════════════════════════════════════════════════════════}

BEGIN
   root:=nil;
   InputParameter;
   error:=false;
   read(x);
   WHILE (x<>0) DO
   BEGIN
      found:=false;
      search(x,root,h,toph,u);
      IF toph THEN
      BEGIN {insert new base node}
         q:=root;
         new(root);
         WITH root↑ DO
         BEGIN
            ns:=1;
            IF q<>nil then nos:=ns+1;
            p0:=q;
            son[1]:=u;
            IF p0<>nil then
            begin
               p0↑.fptr:=root;
               son[1].pt.fptr:=root;
            end;
         END
      END;
      Printtree(root,1,error);
      read(x);
   END;
   PrintTable;
END.
```