Towards an Optimal Data-Structure : CB-trees

T.V.Prabhakar & H.V.Sahasrabuddhe

Computer Sci. Dept., I.I.T. Kanpur, INDIA : 208016

ABSTRACT: This is a proposal for a new data-structure called chained B-trees (CB-trees). CB-trees exhibit a superior access cost curve compared to B-trees. They provide the same amount of space utilisation as B-trees and are not any more expensive to build. In this paper we define CB-trees and study their performance vis_a_vis B-trees through extensive simulation studies. Simulations were done through a novel technique which allows large random trees to be simulated in core.

1.INTRODUCTION: B-trees proposed by Bayer& McCreight [Bayer 72] are widely used for organising large indices. Simple maintenance algorithms, reasonable space utilisation and logarithmic access cost are the main reasons for their popularity. Several variants of the B-tree data-structure appeared in the literature, (eg. B*Trees, B+trees [Knuth 73, COMER 79], Digital B-trees [Lomet 81], Signature Trees [PRABHAKAR 83] etc.). The main thrust of these variants is in improving space utilisation and/or access costs. We propose here yet another data-structure based on the B-tree model. We call it a Chained-B-tree or a CB-tree for short. CB-trees exhibit an access cost curve (number of keys vs. average height) superior to that of a B-tree, provide the same amount of space utilisation and comparable construction cost. Nodes im a CB-tree might span over two disk pages and accessing all keys in such a node requires two page fetches. Thus all leaf nodes nodes are not at the same number of page fetches away from the root. However, they are different from height-balanced-multiway trees [Pasli 79].

In the next section we explain the motivation behind modifying the basic B-tree definition. Section 3 defines and discusses CB-trees. Here we also examine them in relation to other data structures proposed in the literature. A novel technique to simulate in core large randomly built tree structures is used to examine the performance of CB-trees vis-a-vis B-trees. In section 4 we briefly describe this technique and present simulation results. Section 5 presents a summary and conclusions to the paper.

2.MOTIVATION: A B-tree of order n has the following properties:

1:Every node except the root will have at least n keys and at most 2n keys.
2:A node with i keys will have (i+1) descendants.
3:The root may have as few as 2 descendants.
4:All leaves are at the same distance from the root.

A new key to be inserted is accomodated in a leaf node. If the leaf node oveflows (contains more than 2n keys) it is split into two nodes each with n keys and the centre key is pushed up to be accomodated at the parent node.

Proceedings of the Tenth International Conference on Very Large Data Bases.

Singapore, August, 1984

The tree grows in height when the root splits, the average height going up by one. Then the average height stays almost constant till the next split of the root. Variation of average height with number of keys can be seen in fig.5.1.

We observe that when the root is split to accomodate an up-coming key, all sub-trees eminating from the root are penalised by increasing their height. This is necessiated by the restriction that all leaves be at the same distance *from the root. If we relax this constraint, one way to accomodate an upcoming key is to chain a new node to the root and absorb it there as shown in fig.1.

Notice that only sub-trees m and (m+1) have their height increased by 1 and the rest of the sub-trees retain their earlier height. This results in a tree whose average height is less than if the node has been split. With this in mind we define the CB-tree.

3.1 CB-tree: In a CB-tree of order n
1.Every node is either a simple node(encompasses one disk page) or a compound node(encompasses two disk pages).
2.Every node except the root is at least half full. That is, a simple node will have a minimum of n keys and a maximum of 2n keys. A compound node will have at least (2n+1) keys and atmost 4n keys.
3.The root may have only one key (2 descendants).
4.All leaf nodes have the same number of nodes (either simple or compound) in the path to the root.

In a compound node the two pages are referred to as the main and twin. Disk pages for these two nodes need not be contiguous and can be assigned from anywhere on the disk. The twin can be reached only via the main page. Hence to access entries on a twin an extra page fetch is necessry. Leaf nodes are equi-distant from the root in node count but can differ by a ratio of 1:2 in page fetches.

A key insertion is done as in a B-tree observing the following rules:
Rule 1: Do not split an overflowing simple node if doing so will cause the root to split.
Rule 2: Always split an overflowing compound node.
Rule 3: Split a compound node even if it is not full when doing so will not cause the root to split.

When all the nodes in the path to the root are saturated a split at the leaf also splits the root . Rule 1 indicates, at such a time do not split the leaf but chain a new page converting it into a compound node. We observe that this operation retains the average height of keys in a CB-tree close to the height before insertion whereas a B-tree would have grown in height by 1. Thus in a CB-tree first the leaf nodes acquire chains and become compound nodes.

According to rule 2, a compound node is always split if it overflows. This might result in the root splitting if it is already a compound node which is full.

It is possible to have compound nodes in the tree even when their ancestor nodes are not saturated. This can occur as follows: Consider a node "a" with "x" and "y" as two of its descendants. Node "a" is saturated and so are its ancestors. Now "x" and "y" overflow and are converted to compound nodes, "x-x'" and "y-y'". If the compound node "x-x'" again overflows it is split into two simple nodes. A key 1 be pushed up and node "a" is converted to a compound node "a-a'". One can see that the compound node "a-a'" can accomodate more keys but it has a descendant which is also a compound node. In such a situation when the next key insertion occurs in "y-y'", it is split into two simple nodes nodes with the upcoming key accomodated at the compound node "a-a'". Fig.2 illustrates this sequence of events.

Among the data-structures reported in the literature of related interest are height-balanced-multiway trees [Pasli 79], Symmetric Binary B-trees [Bayer 73,Wirth 77] and Digital B-trees [Lomet 81]. Height-balanced-multiway trees are like AVL trees [Knuth 73] where the height of all sub-trees at any node may differ by one. Symmetric binary B-trees have been proposed to avoid pre-allocation of space to nodes which are anyway generally not full. Here a 2-3 tree node with 2 keys and 3 pointers is constructed out of 2 nodes with 1 key and 2 pointers. Digital B-trees have nodes which can span over

several contiguous disk pages. However, only one of these disk pages need be fetched for each data access. This is achieved by using bits of the keys to decide which disk block is to be fetched.

CB-trees are different from all these data-structures. Unlike in a height-balanced-multiway tree the longest path length in a CB-tree can be twice as much as the longest path length in a neighbouring sub-tree. CB-trees are not higher order generalisations of symmetric binary B-trees either where an order 2n node could be composed by 2 order n nodes. This is so because when an order n node gets full in a symmetric binary B-tree it automatically gets chained to another order n node to provide a full node of order 2n. In a CB-tree such chains are made only under certain conditions. Digital B-trees are clearly different since they use prefixes (digital searching) and are not comparison based. But like in a CB-tree an overflowing node in a Digital-B-tree can be split or compoundd. A compoundd digital B-tree node occupies two consecutive disc pages. After doubling, entries in its ancestor are modified so that only one of these two disc pages need be fetched at any time. The choice between doubling or splitting a node is decided by considering the utilisation of its ancestor and siblings. Note that in a CB-tree the decision to split or chain is arrived at by consulting the utilisation of its ancestors.

3.2 CB-tree Algorithms. Nodes in a CB-tree are composed of the following record structure:

```
node = record
        TWIN   : boolean ;
        KEYSHERE : 1..2n ;
        KEYS   : array [1..2n] of
                    key-type ;
        P      : array [0..2n] of
                    ^ node
       end ;
```

The 'TWIN' field on a node will indicate if it is a compound node. For a compound node the TWIN flag on main is set to 'true' and P[2N] points to its twin . On a twin node 'TWIN' is always set to false. KEYSHERE is a count for the number of keys on the node.

3.2.1 SEARCH: Searching for a key in the tree in similar to searching in a B-tree. Fig.3 gives a function to search for a key in a CB-tree.

The main advantage of a CB-tree over a B-tree is in the average page fetches required for a search. In both the tree, a search is completed in O (log N) time, for N keys in the tree. In the worst case a B-tree search will take $1+\lfloor \log_n (\frac{N+1}{2}) \rfloor$ page fetches. For a CB-tree the farthest node in the worst case may be $2\lfloor (\log_n (\frac{N+1}{2})) \rfloor$ pages away. At the same time there will be leaf nodes nearer to the root than this worst case, reducing the average height.The average height is sometimes better than the average height of a B-tree. We do not yet have an expression for the expected average height of a CB-tree. Experiments were conducted to compare the average heights of random B-trees and CB-trees. The next section gives details of these experiments.

3.2.2 INSERTION: Key insertion in a CB-tree is handled in such a way that overflowing nodes penalize as little as possible the rest of the tree. As explained before, an overflowing simple node gets split or gets converted to a compound node depending on its ancestors' occupancy. An overflowing compound node is always split. A compound node may sometimes gets split even when it does not overflow. Fig.4 sketches the algorithm for inserting a key in a CB-tree. Here the predicate THERE_IS_SPACE_HERE is true for a node if it is not full. That is, for a simple node THERE_IS_SPACE_HERE is true if it has less than 2n entries and for a compound node less than 2n entries on its twin(the main will always have 2n entries). Similarly THERE_IS_SPACE_ABOVE is true for any node if for any of its ancestors THERE_IS_SPACE_HERE is true.

Like in a B-tree an insertion will take O(log n) page reads and page writes. Accessing the entries on a twin page of a compound node requires an extra page fetch. If a key were to be inserted on the main page of a compound node the twin also need be fetched and rewritten.

3.2.3 Key Deletion: Deleting a key is very similar to key deletion in a B-tree. When a node underflows it is merged with its siblings to maintain 50% occupancy. An underflowing compound

node sets converted to a simple node.

4.1 SIMULATION RESULTS: In the absence of analytical results to evaluate CB-trees, we decided to examine their performance through simulation studies. Simulations were done using a novel technique which allows random trees of large size to be built in the main memory. (See [Yao 78] for the definition of random trees). This method of simulation does not store the key values in the tree. Only the structural information (number of keys at each node, in each sub-tree, pointers to the sub-tree) is retained. A key insertion is done by navigating through this to a page containing a random number in the range 1 to (N+1) and updating its key count (There are N keys in the tree). This way an order n tree with N keys can be simulated for structural information in O(N/n) memory locations. For example a million key tree of order 50 can be accomodated in under 100k bytes.

Our objective in these simulation studies was to examine CB-tree vis-a-vis B-trees for the following performance measures:

1. Average height
2. Storage utilisation
3. Construction cost

Average height of a tree is the average number of links traversed(including the link to reach the root) to execute a successful search operation. Storage utilisation is the ratio between the space occupied and space allotted. Construction cost is measured in pages read and written to build the tree.

Trees of medium order(15,20,25,30) were examined for their performance over an operating range extending to 100000 keys. Each tree was studied over 20 instances of randomly generated trees.

4.2 Average Height: Figures 5.1 to 5.4 give the average access cost curves for these trees. Also shown is the access cost curve of an 'ideal tree'. An ideal tree gives the best possible average height for a multiway tree. In here, the leaves may differ in height by one and only the unsaturated nodes are the farthest nodes.

The average height of a B-tree grows in sharp steps at the points where the root splits. After a root split the height stays constant for a large number of insertions(till the next split). CB-trees exhibit a more gradual increase in height with increase in tree size. They are like B-trees (in fact, they 'are' B-trees) till the point where the root of a B-tree would have been split. After that they grow differently for some time and again acquire the shape of B-trees. When the root splits in a B-tree, a CB-tree leaf acquires a chain becoming a compound node. Height of the CB-tree grows gradually as more and more nodes become compound nodes. When the CB-tree root sets saturated as a compound node it splits. This is the point when its height overshoots that of a B-tree(This overshoot is too small to observe in trees of order 20 and 25). But quickly after that it loses all its compound nodes and continues to grow like B-tree till one of the leaf nodes become a compound node.

Areas under the B-tree and CB-tree access cost curves were computed for comparison. Table 1 indicates the same along with the standard deviation recorded for each of these areas. If each of them is assumed to be normally distributed about their means, the random variable

$$T = \frac{\bar{X} - \bar{Y}}{(s_1^2/n_1 + s_2^2/n_2)^{1/2}}$$

where $s_1^2 = \sum (x_i - \bar{X})^2$ and

$$s_2^2 = \sum (y_i - \bar{Y})^2,$$

n_1, n_2 = number of samples in X and Y.

is approximately distributed as a Student-t with I degrees of freedom where I is calculated as [Green 78]

$$\frac{1}{I} = \frac{1}{(n_1 - 1)} * \left(\frac{s_1^2/n_1}{s_1^2/n_1 + s_2^2/n_2} \right)^2 + \frac{1}{(n_2 - 1)} * \left(\frac{s_2^2/n_2}{s_1^2/n_1 + s_2^2/n_2} \right)^2$$

The computed significance level for

the hypothesis: Area under the CB-tree acess cost curve is less than Area under B-tree access cost curve came out to negligibly small.

4.3 Storage Utilisation: No marked differences were observed among the number of pages that each tree occupies. Table 2 indicates the average storage utilisation for trees of order 25 at various points of their growth. Also shown are the standard deviations for these samples and the confidence level for the hypothesis:
0.95*Mean(B-tree) <= Mean(CB-tree) <= 1.05*Mean(B-tree)
These are computed using the t-statistic described earlier.

4.4 Construction Cost: This parameter does not particularly appear to be in favour of any of the two trees. Table 3 shows a sample of the page accesses (reads and writes) incurred per insertion for contructing these trees. Even though CB-trees required less page accesses at the maximum tree size simulated, it is more likely that the oscillatory behaviour indicated in the earlier readings to continue.

5.CONCLUSIONS: A data-structure (Chained B-trees) for index organisation has been proposed. It is obtained by modifying the B-tree definition to allow leaf nodes at different distances (measured in page fetches) away from the root. Simulation studies were conducted to examine its performance as compared to B-trees. The following conclusions were reached:

1) CB-trees exhibit a more gradual growth in their average height with increase in number of keys. Compared to a B-tree this growth curve is nearer to the best possible by a multiway tree.
2) CB-trees provide the same amount of space utilisation as B-trees.
3) It is no more expensive to construct a CB-tree than a B-tree

6.REFERENCES :

[BAYER 72] : Bayer,R. & McCreight,E., 'Organisation and Maintenance of Large Indices', Acta Informatica, Vol 1, 173-189

[BAYER 73] : Bayer,R. 'Symmetric Binary B-trees:Data Structure and Algorithms', Acta Informatica Vol 1;4, 290-306

[COMER 79] :Comer,D., 'The Ubiquitous B-Tree', ACM Computing Surveys, Vol 11(2),1979,121-138

[GREEN 78] : Green,J.R., Margerison,D., 'Statistical Treatment of Experimental Data', Elsevier/North-Holland Inc., First Revised Reprint

[KNUTH 73] : Knuth,D. 'Art of Programming, Vol 3', Addison Wesley

[LOMET 81] : Lomet,D.B. 'Digital B-trees', Proc. 7 Intnl. Conf. on Very Large Data Bases

[PAGLI 79] : Pagli,L. 'Height Balanced Multiway Trees', Info. Systems Vol.4, 290-306

[PRABHAKAR 83] : Prabhakar,T.V. & Sahasrabuddhe,H.V. 'Signature Trees - A Data Structure for Organising Large Indices', Proc. IEEE Intnl. Conf. on Sytems Man & Cybernetics, New-Delhi Dec 83-Jan 84

[WIRTH 77] : Wirth,N. 'Algorithms + Data Structures = Programs', Prentice-Hall Inc.

[Yao 78] : Yao,A.C. 'On Random 2-3 Trees', Acta Informatica Vol 9, 159-170

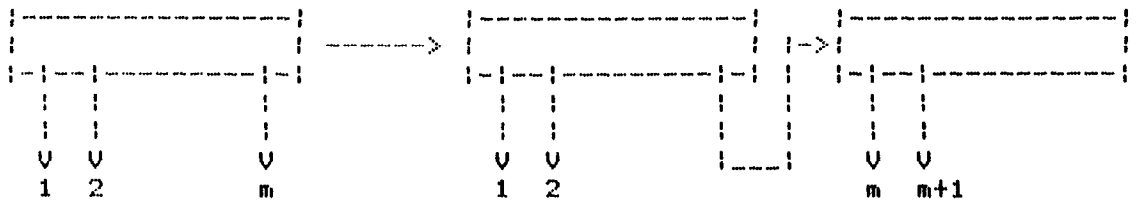
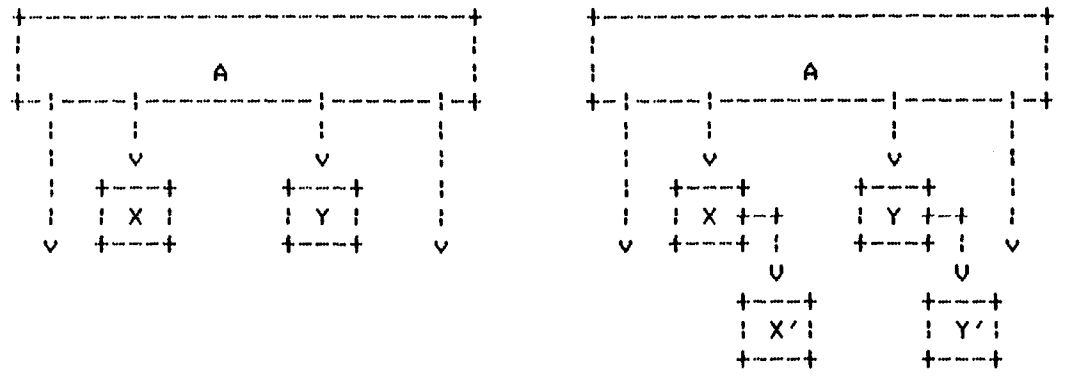
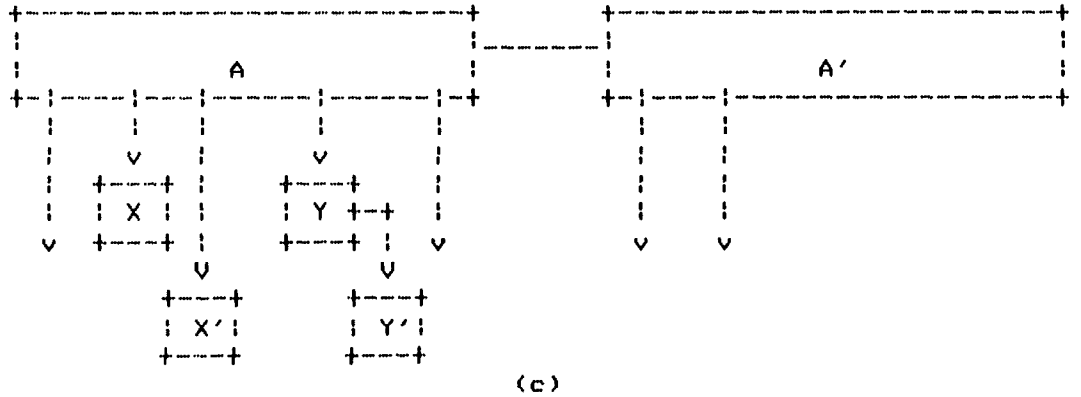


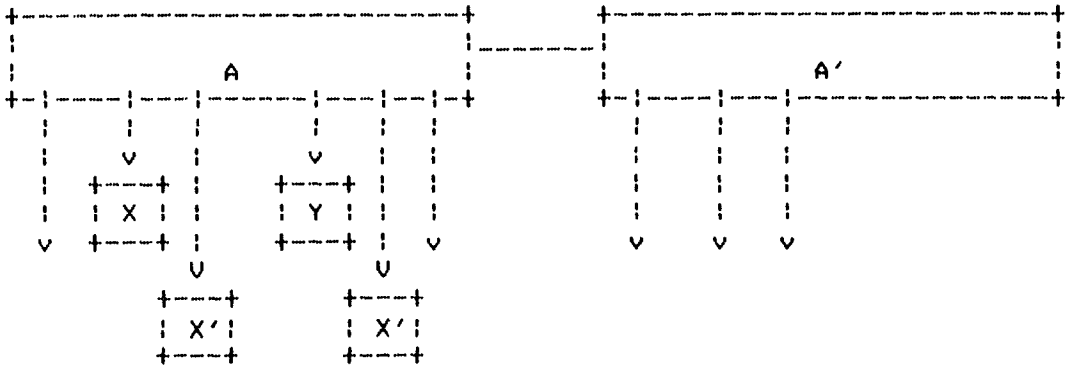
Fig. 1 Chaining Instead of Splitting



(a) (b)



(c)



(d)

Figure 2

```
function SEARCH(K:KEY-type ; X : ^ to A NODE ):boolean ;
{ SEARCH FOR K IN THE TREE POINTED BY X. SEARCH IS TRUE IF KEYS
 IS FOUND OR FALSE OTHERWISE }
   begin
       if X=nil then SEARCH := false
       else with X^ do
         begin
             case   K of
                   K in  {K[1],K[2],..K[KEYSHERE]} : SEARCH := true ;
                   K < K[1] : SEARCH := SEARCH(K,P[0]) ;
                   K < K[I] < K[I+1] : SEARCH := SEARCH(K,P[KEYSHERE]) ;
                   K > K[KEYSHERE] : SEARCH := SEARCH[K,P[KEYSHERE])
 end  end end ;
```

FIG.3 Function SEARCH

```
procedure INSERT ;
{ TO INSERT A KEY K IN THE SUB-TREE ROOTED AT X^ }
     begin
       if X=nil then PUSH THE KEY UP
       else
       with X^ do
         begin
           if TWIN then
             begin
               FIND THE APPROPRIATe SUB-TREE in THE
               COMPUND NODE THROUGH A BINARY SEARCH;
               INSERT KEY in THE SUB-TREE;
               if SUB-TREE SPLITS
               then
                 if (THERE_IS_SPACE_HERE) and (not THERE_IS_SPACE_ABOVE)
                       then ABSORB THE UPCOMING KEY
                  else
                    begin  SPLIT THE COMPOUND NODE
                      ABSORB THE UPCOMING KEY
                      PUSH A KEY UP
                    end
                else if (THERE_IS_SPACE_ABOVE) then begin
                                                     SPLIT THE COMPOUND
                                                     NODE.PUSH A KEY UP
                                                    end
             end
           else { node is a simple node }
             begin
               FIND APPROPRIATe SUB-TREE in NODE BY BINARY SEARCH;
               INSERT KEY in SUB-TREE;
               if (SUB-TREE SPLITS) then
                   if (THERE IS SPACE HERE) then ABSORB THE UPCOMING KEY
                   else if (THERE IS SPACE ABOVE)
                        then
                     begin SPLIT THE NODE;
                       ABSORB THE UPCOMING KEY;
                       PUSH A KEY UP
                     end
                   else
                     begin
                       CONVERT THE SIMPLE NODE to A COMPOU
                       NODE BY CHANING A NEW PAGE;
                       ABSORB THE UPCOMING KEY
                         end
       end    end
   end;
```

Fig.4 Procedure to Insert a key in a CB-tree

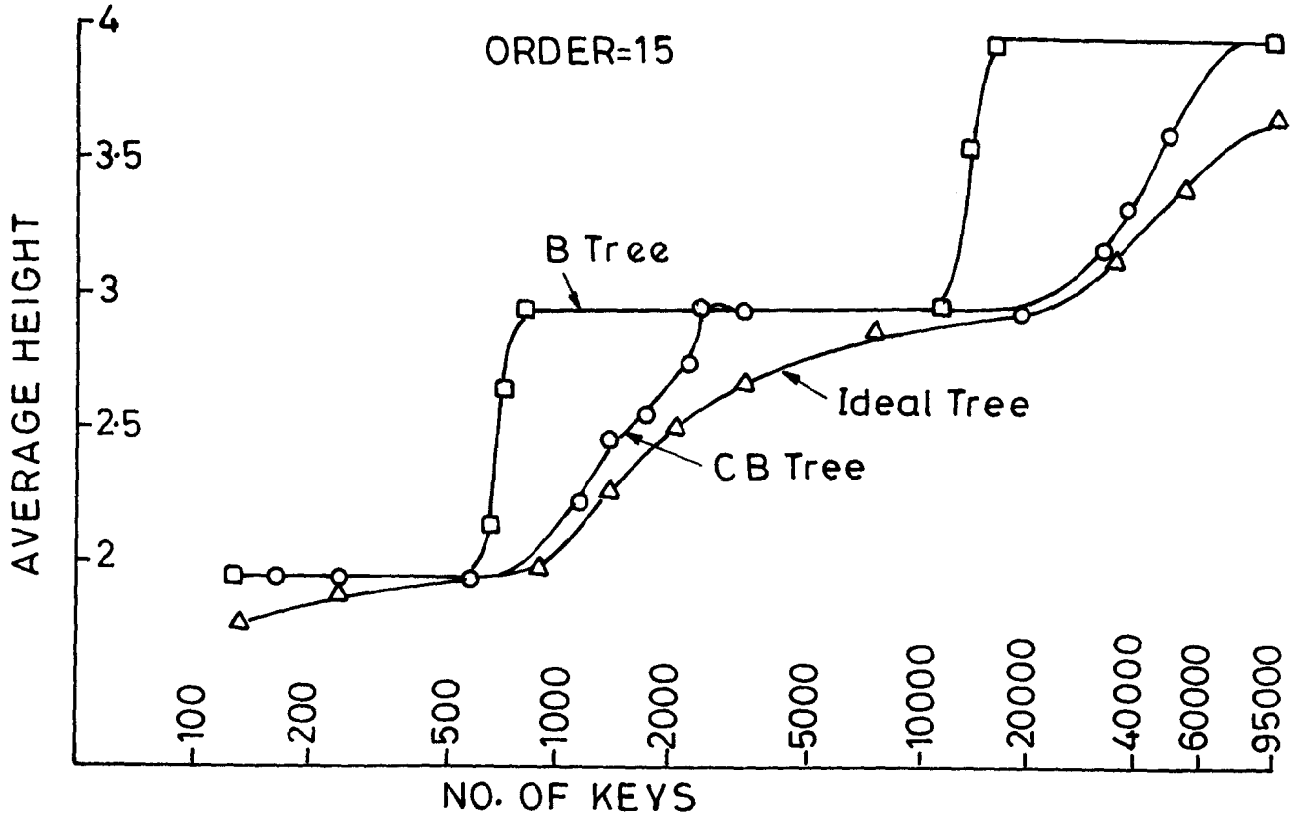


FIG. 5·1 ACCESS COST CURVES

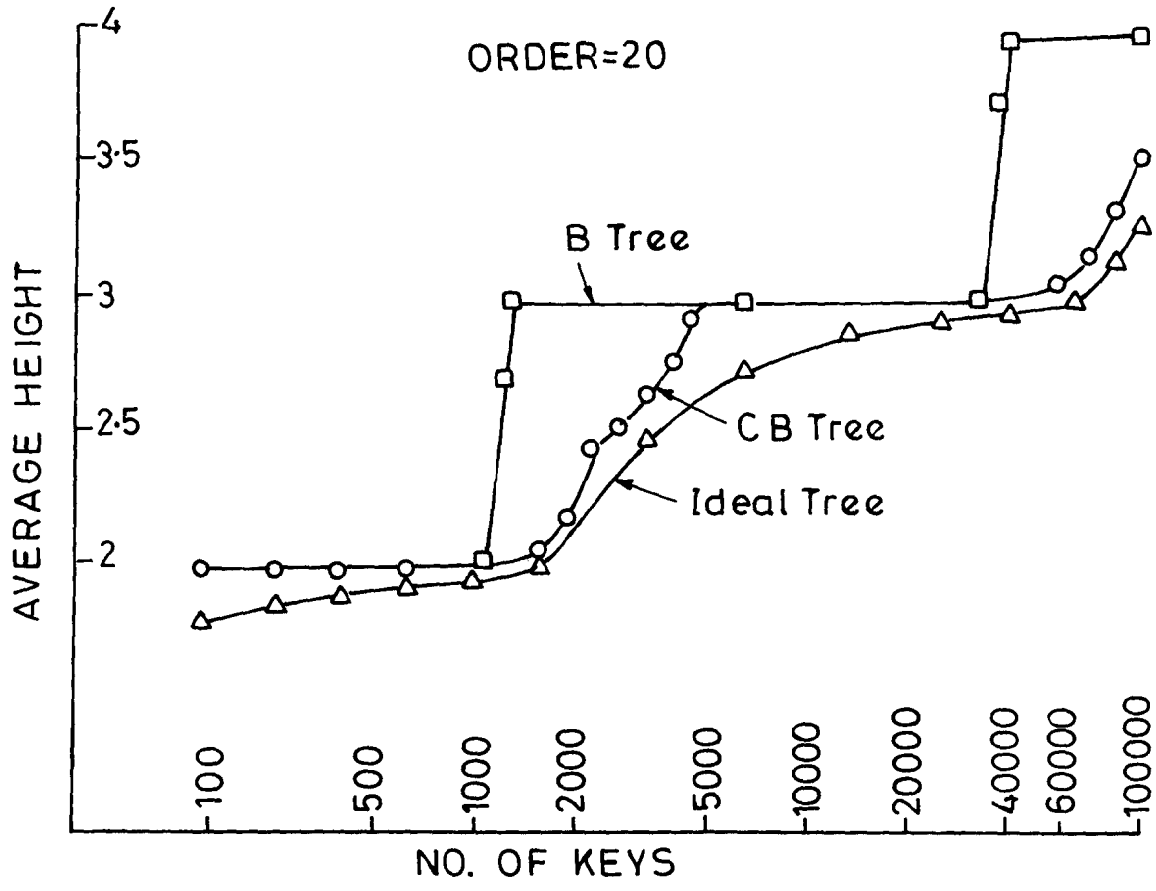


FIG. 5·2 ACCESS COST CURVES

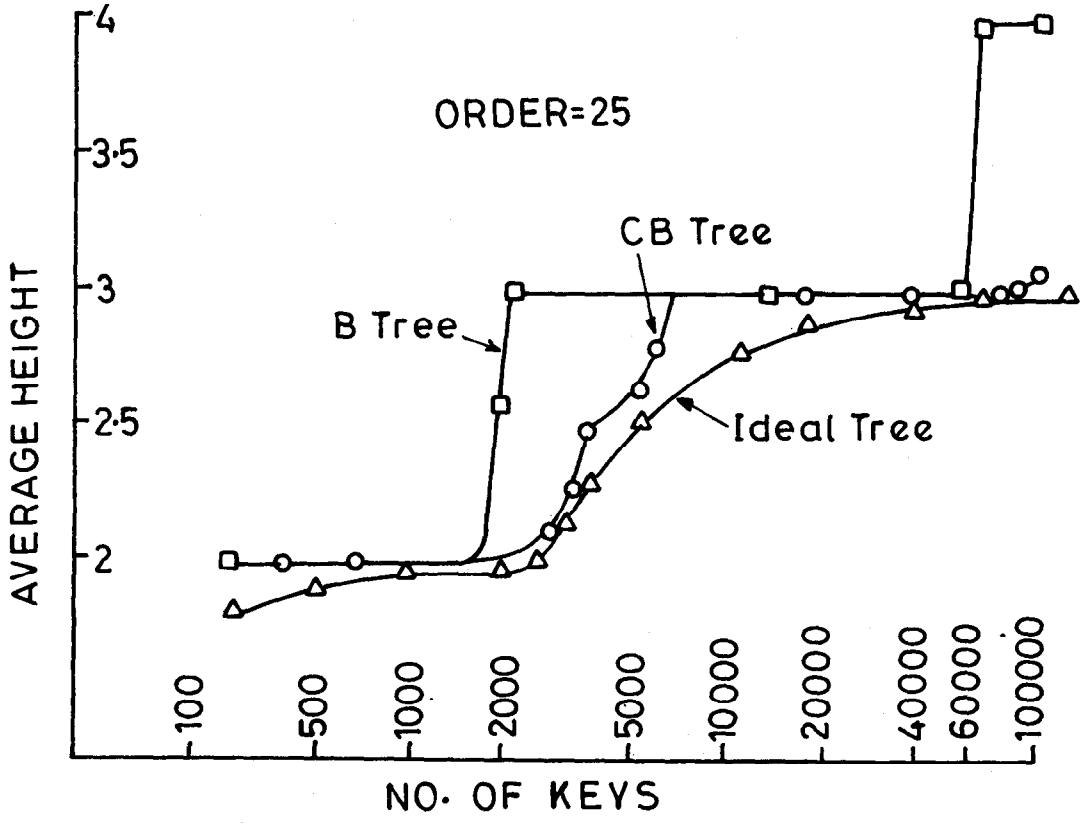


FIG.5·3 ACCESS COST CURVES

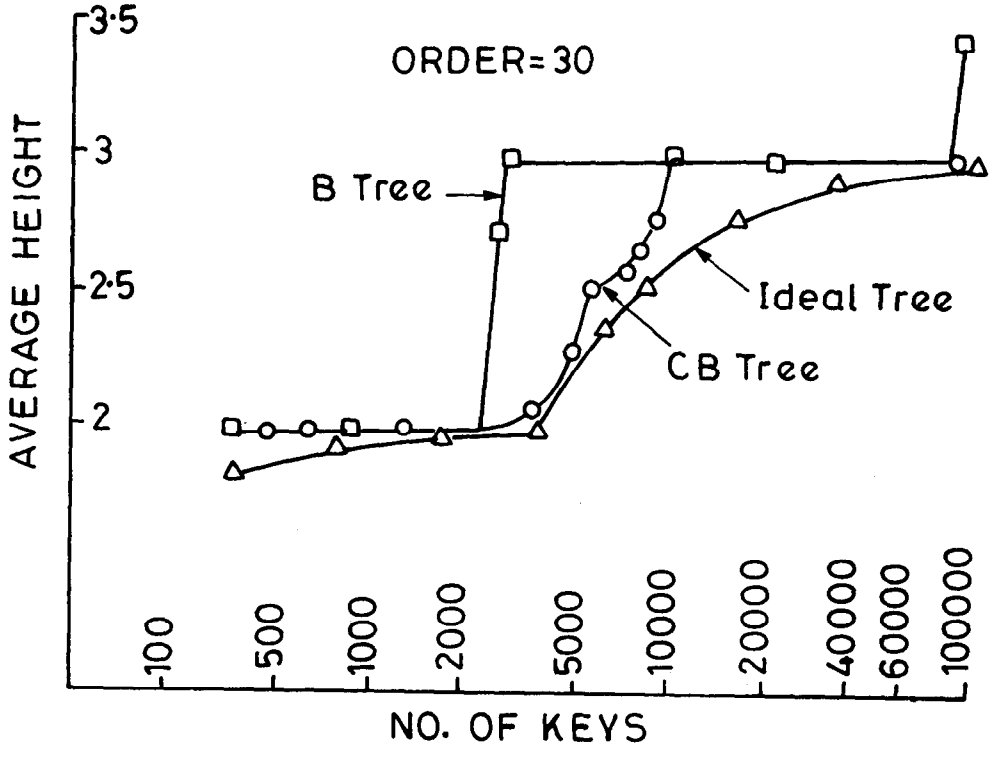


FIG.5·4 ACCESS COST CURVES

	B-TREE		CB-TREE	
ORDER	MEAN	S.D	MEAN	S.D.
15	367559.9200	1009.8646	336640.3100	2086.7847
20	364900.9800	1422.8034	312586.8800	1139.0156
25	345847.6600	105.21402	301769.3300	311.9264
30	281530.5200	1515.3457	276414.7400	189.1412

TABLE 1. AREAS UNDER THE ACCESS COST CURVE

KEYS	MEAN		STANDARD DEV.		CONF.
	B-TREE	CB-TREE	B-TREE	CB-TREE	
1037	0.683582	0.682783	0.042821	0.019812	99.6679
2500	0.689363	0.707390	0.021834	0.022676	98.7485
4957	0.691083	0.695623	0.012508	0.017007	99.9999
10841	0.696409	0.689499	0.009957	0.007964	99.9999
21497	0.696491	0.691716	0.006805	0.006014	99.9999
28827	0.688875	0.690629	0.006490	0.005213	99.9999
42628	0.693709	0.691429	0.005045	0.004914	99.9999
63036	0.685374	0.690657	0.004829	0.003951	99.9999
76655	0.690161	0.694073	0.002767	0.003542	99.9999
84530	0.692911	0.695727	0.003168	0.003900	99.9999
93215	0.695055	0.697747	0.003496	0.003475	99.9999
102793	0.693298	0.699631	0.003372	0.003213	99.9999

TABLE 2. STORAGE UTILISATION
ORDER OF THE TREE = 25

CONF. = 1.0 - SIGNIFICANCE LEVEL OF THE HYPOTHESIS:
0.95*MEAN(B-TREE) <= MEAN(CB-TREE) <= 1.05*MEAN(B-TREE)

KEYS	B-TREE		N-TREE	
	MEAN	ST.DEV.	MEAN	ST. DEV.
1037	3.00376	0.00357	3.03905	0.00309
2500	3.30908	0.03165	3.16890	0.01910
4957	3.68015	0.01623	3.61846	0.04221
10841	3.88472	0.00729	3.94463	0.04186
21497	3.97033	0.00386	4.00065	0.02111
28827	3.99311	0.00285	4.01528	0.01551
42628	4.01374	0.00194	4.02902	0.01058
63036	4.07762	0.01610	4.03887	0.00704
76655	4.25146	0.01290	4.05996	0.00771
84530	4.32637	0.01166	4.08162	0.00889
93215	4.39433	0.01061	4.10993	0.01041
102793	4.45627	0.00974	4.14360	0.01130

TABLE 3. PAGES READ|WRITES/INSERTION
ORDER OF THE TREE = 25