Acta Informatica 1, 173–189 (1972)

Organization and Maintenance of Large Ordered Indexes

R. Bayer and E. McCreight

Received September 29, 1971

Summary. Organization and maintenance of an index for a dynamic random access file is considered. It is assumed that the index must be kept on some pseudo random access backup store like a disc or a drum. The index organization described allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ where I is the size of the index and k is a device dependent natural number such that the performance of the scheme becomes near optimal. Storage utilization is at least 50% but generally much higher. The pages of the index are organized in a special datastructure, so-called B-trees. The scheme is analyzed, performance bounds are obtained, and a near optimal k is computed. Experiments have been performed with indexes up to 100000 keys. An index of size 15000 (100000) can be maintained with an average of 9 (at least 4) transactions per second on an IBM 360/44 with a 2311 disc.

1. Introduction

In this paper we consider the problem of organizing and maintaining an index for a dynamically changing random access file. By an *index* we mean a collection of index elements which are pairs (x, α) of fixed size physically adjacent data items, namely a key x and some associated information α. The key x identifies a unique element in the index, the associated information is typically a pointer to a record or a collection of records in a random access file. For this paper the associated information is of no further interest.

We assume that the index itself is so voluminous that only rather small parts of it can be kept in main store at one time. Thus the bulk of the index must be kept on some backup store. The class of backup stores considered are *pseudo random access devices* which have a rather long access or wait time—as opposed to a true random access device like core store—and a rather high data rate once the transmission of physically sequential data has been initiated. Typical pseudo random access devices are: fixed and moving head discs, drums, and data cells.

Since the data file itself changes, it must be possible not only to search the index and to retrieve elements, but also to delete and to insert keys—more accurately index elements—economically. The index organization described in this paper always allows retrieval, insertion, and deletion of keys in time proportional to $\log_k I$ or better, where I is the size of the index, and k is a device dependent natural number which describes the page size such that the performance of the maintenance and retrieval scheme becomes near optimal.

In more illustrative terms theoretical analysis and actual experiments show that it is possible to maintain an index of size 15000 with an average of 9 retrievals, insertions, and deletions per second in real time on an IBM 360/44 with a 2311 disc as backup store. According to our theoretical analysis, it should be possible to maintain an index of size 1500000 with at least two transactions per second on such a configuration in real time.

The index is organized in pages of fixed size capable of holding up to $2k$ keys, but pages need only be partially filled. Pages are the blocks of information transferred between main store and backup store.

The pages themselves are the nodes of a rather specialized tree, a so-called B-tree, described in the next section. In this paper these trees grow and contract in only one way, namely nodes split off a brother, or two brothers are merged or "catenated" into a single node. The splitting and catenation processes are initiated at the leaves only and propagate toward the root. If the root node splits, a new root must be introduced, and this is the only way in which the height of the tree can increase. The opposite process occurs if the tree contracts.

There are, of course, many competitive schemes, e.g., hash-coding, for organizing an index. For a large class of applications the scheme presented in this paper offers significant advantages over others:

i) Storage utilization is at least 50% at any time and should be considerably better in the average.

ii) Storage is requested and released as the file grows and contracts. There is no congestion problem or degradation of performance if the storage occupancy is very high.

iii) The natural order of the keys is maintained and allows processing based on that order like: find predecessors and successors; search the file sequentially to answer queries; skip, delete, retrieve a number of records starting from a given key.

iv) If retrievals, insertions, and deletions come in batches, very efficient essentially sequential processing of the index is possible by presorting the transactions on their keys and by using a simple prepaging algorithm.

Several other schemes try to solve the same or very similar problems. AVL-trees described in [1] and [2] guarantee performance in time $\log_2 I$, but they are suitable only for a one-level store. The schemes described in [3] and [4] do not have logarithmic performance. The solution presented in this paper is new and is related to those described in [1–4] only in the sense that the problem to be solved is similar and that it uses a data organization involving tree structures.

2. *B*-Trees

Def. 2.1. Let $h \geqq 0$ be an integer, k a natural number. A directed tree T is in the class $\tau(k, h)$ of *B-trees* if T is either empty ($h = 0$) or has the following properties:

i) Each path from the root to any leaf has the same length h, also called the *height* of T, i.e., $h =$ number of nodes in path.

ii) Each node except the root and the leaves has at least $k+1$ sons. The root is a leaf or has at least two sons.

iii) Each node has at most $2k+1$ sons.

Number of Nodes in B-Trees. Let $N_{\min}$ and $N_{\max}$ be the minimal and maximal number of nodes in a B-tree $T \in \tau(k, h)$. Then

$$N_{\min} = 1 + 2\left((k+1)^0 + (k+1)^1 + \cdots + (k+1)^{h-2}\right) = 1 + \frac{2}{k}\left((k+1)^{h-1} - 1\right)$$

for $h \geqq 2$. This also holds for $h=1$. Similarly one obtains

$$N_{\max} = \sum_{i=0}^{h-1} (2k+1)^i = \frac{1}{2k}\left((2k+1)^h - 1\right); \qquad h \geqq 1.$$

Upper and lower bounds for the number $N(T)$ of nodes of $T \in \tau(k, h)$ are given by:

$$N(T) = 0 \qquad \text{if } T \in \tau(k, 0); \tag{2.1}$$

$$1 + \frac{2}{k}\left((k+1)^{h-1} - 1\right) \leqq N(T) \leqq \frac{1}{2k}\left((2k+1)^h - 1\right) \quad \text{otherwise.}$$

Note that the classes $\tau(k, h)$ need not be disjoint.

3. The Data Structure and Retrieval Algorithm

To repeat, the pages on which the index is stored are the nodes of a B-tree $T \in \tau(k, h)$ and can hold up to $2k$ keys. In addition the data structure for the index has the following properties:

i) Each page holds between k and $2k$ keys (index elements) except the root page which may hold between 1 and $2k$ keys.

ii) Let the number of keys on a page P, which is not a leaf, be l. Then P has $l+1$ sons.

iii) Within each page P the keys are sequential in increasing order: $x_1, x_2, \ldots, x_l$; $k \leqq l \leqq 2k$ except for the root page for which $1 \leqq l \leqq 2k$. Furthermore, P contains $l+1$ pointers $p_0, p_1, \ldots, p_l$ to the sons of P. On leaf pages these pointers are undefined. Logically a page is then organized as shown in Fig. 1.

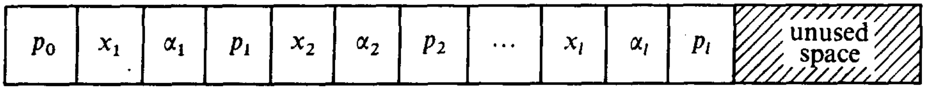


Fig. 1. Organization of a page

The α_i are the associated information in the index element (x_i, α_i). The triple (x_i, α_i, p_i) or—omitting α_i—the pair (x_i, p_i) is also called an *entry*.

iv) Let $P(p_i)$ be the page to which p_i points, let $K(p_i)$ be the set of keys on the pages of that maximal subtree of which $P(p_i)$ is the root. Then for the B-trees considered here the following conditions shall always hold:

$$(\forall y \in K(p_0))(y < x_1), \tag{3.1}$$

$$(\forall y \in K(p_i))(x_i < y < x_{i+1}); \quad i = 1, 2, \ldots, l-1, \tag{3.2}$$

$$(\forall y \in K(p_l))(x_l < y). \tag{3.3}$$

Fig. 2 is an example of a B-tree in $\tau(2, 3)$ satisfying all the above conditions. In the figure the α_i are not shown and the page pointers are represented graphically. The boxes represent pages and the numbers outside are page numbers to be used later.

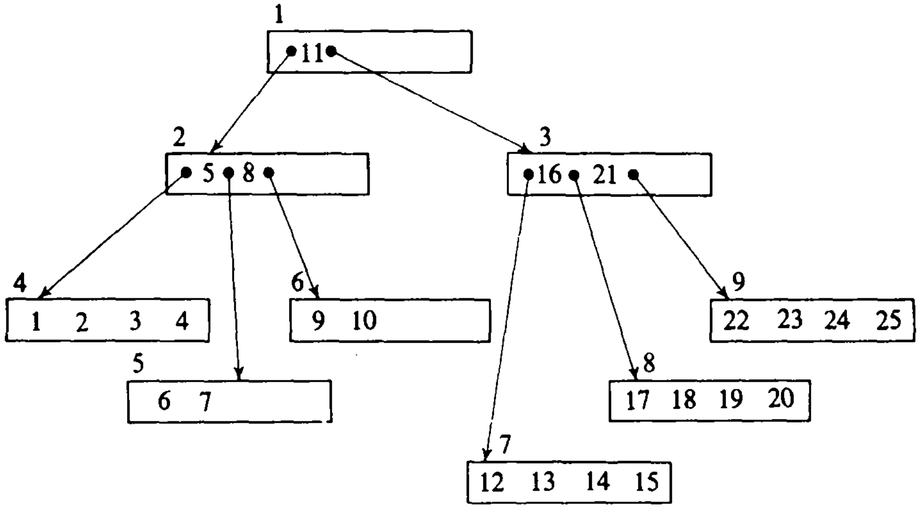


Fig. 2. A data structure in $\tau(2, 3)$ for an index

Retrieval Algorithm. The flowchart in Fig. 3 is an algorithm for retrieving a key y. Let p, r, s be pointer variables which can also assume the value "undefined" denoted as u. r points to the root and is u if the tree is empty, s does not serve any purpose for retrieval, but will be used in the insertion algorithm. Let $P(p)$ be the page to which p is pointing, then $x_1, \ldots, x_l$ are the keys in $P(p)$ and $p_0, \ldots, p_l$ the page pointers in $P(p)$.

The retrieval algorithm is simple logically, but to program it for a computer one would use an efficient technique, e.g., a binary search, to scan a page.

Cost of Retrieval. Let h be the height of the page tree. Then at most h pages must be scanned and therefore fetched from backup store to retrieve a key y. We will now derive bounds for h for a given index of size I. The minimum and maximum number $I_{\min}$ and $I_{\max}$ of keys in a B-tree of pages in $\tau(k, h)$ are:

$$I_{\min} = 1 + k\left(2\,\frac{(k+1)^{h-1} - 1}{k}\right) = 2(k+1)^{h-1} - 1$$

$$I_{\max} = 2k\left(\frac{(2k+1)^{h} - 1}{2k}\right) = (2k+1)^{h} - 1 .$$

This is immediate from (2.1) for $h \geqq 1$. Thus we have as sharp bounds for the height h:

$$\begin{gathered} \log_{2k+1}(I+1) \leqq h \leqq 1 + \log_{k+1}\left(\frac{I+1}{2}\right) \quad \text{for} \quad I \geqq 1, \\ h = 0 \quad \text{for} \quad I = 0. \end{gathered} \tag{3.1}$$

4. Key Insertion

The algorithm in Fig. 4 inserts a single key y into an index described in Section 3. The variable s is a page pointer set by the retrieval algorithm pointing to the last page that was scanned or having the value u if the page tree is empty.

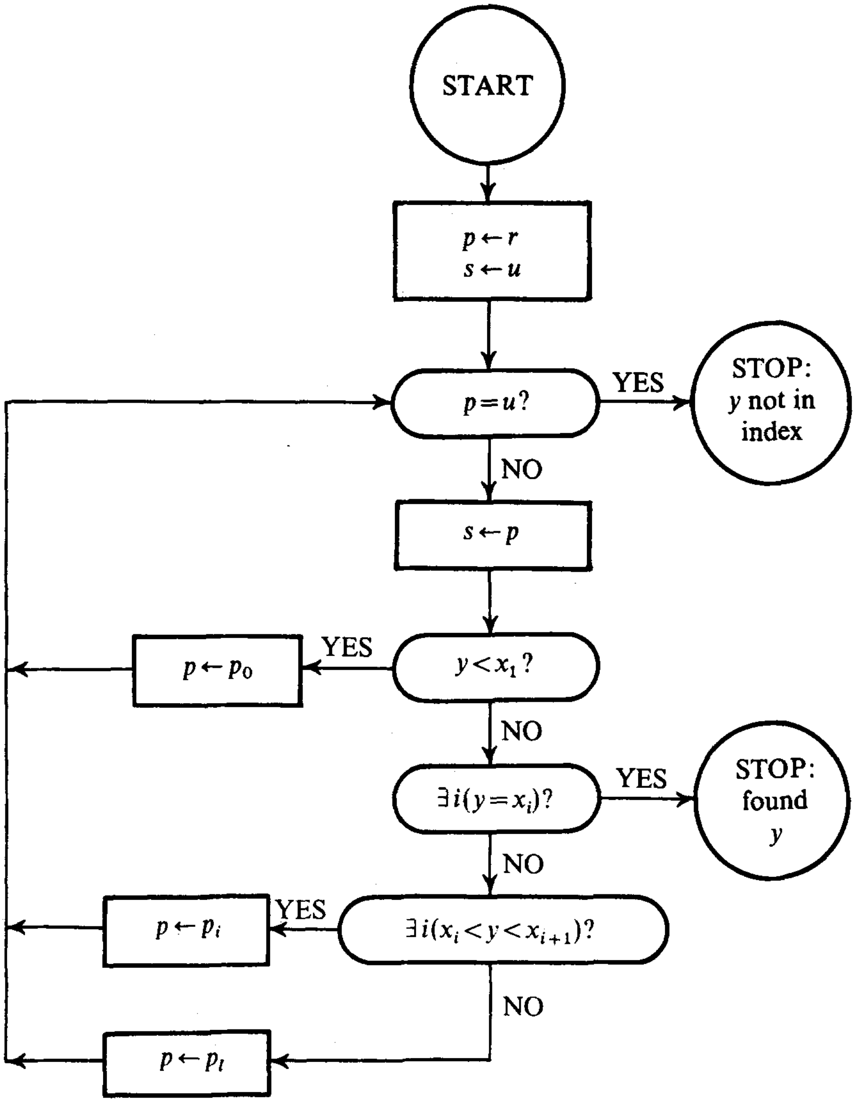


Fig. 3. Retrieval algorithm

Splitting a Page. If a page P in which an entry should be inserted is already full, it will be split into two pages. Logically first insert the entry into the sequence of entries in P—which is assumed to be in main store—resulting in a sequence

$$p_0, (x_1, p_1), (x_2, p_2), \ldots, (x_{2k+1}, p_{2k+1}).$$

Now put the subsequence $p_0, (x_1, p_1), \ldots, (x_k, p_k)$ into P and introduce a new page P' to contain the subsequence

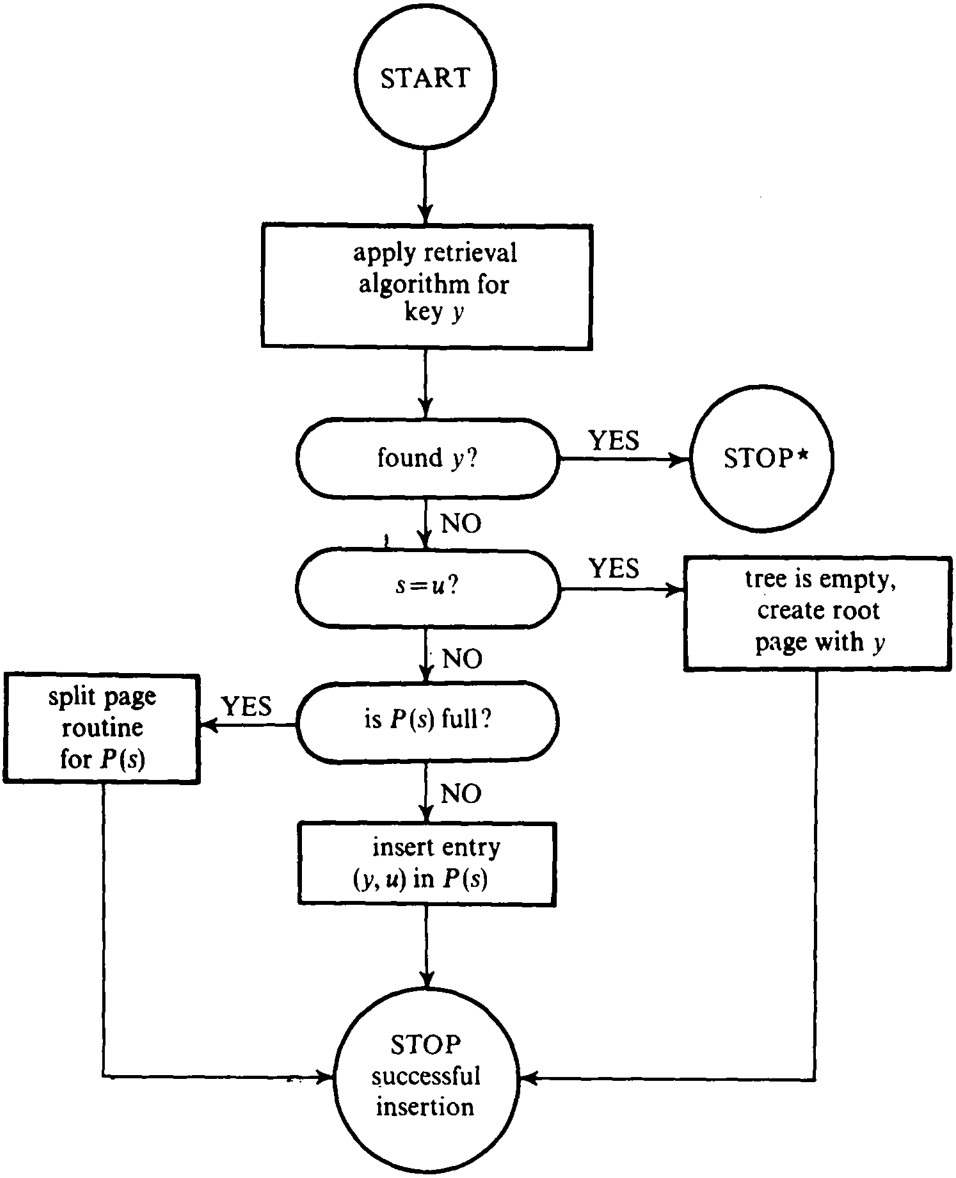
$$p_{k+1}, (x_{k+2}, p_{k+2}), (x_{k+3}, p_{k+3}), \ldots, (x_{2k+1}, p_{2k+1}).$$

Let Q be the father page of P. Insert the entry (x_{k+1}, p'), where p' points to P', into Q. Thus P' becomes a brother of P.

Inserting (x_{k+1}, p') into Q may, of course, cause Q to split too, and so on, possibly up to the root. If the splitting page P is the root, then we introduce a new root page Q containing p, (x_{k+1}, p') where p points to P and p' to P'.

Note that this insertion process maps B-trees with parameter k into B-trees with parameter k, and preserves properties (3.1), (3.2), and (3.3).

To illustrate the insertion process, insertion of key 9 into the tree in Fig. 5 with parameter $k = 2$ results in the tree in Fig. 2.


START
apply retrieval algorithm for key y
found y?
YES
STOP*
NO
$s=u$?
YES
tree is empty, create root page with y
NO
is $P(s)$ full?
YES
split page routine for $P(s)$
NO
insert entry (y, u) in $P(s)$
STOP successful insertion

* Key y is already in index, take appropriate action.

Fig. 4. Insertion algorithm

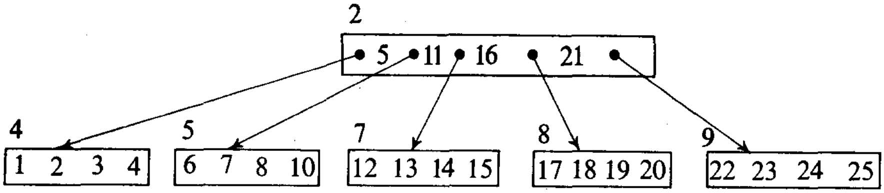


Fig. 5. Index structure in $\tau(2, 2)$

5. Cost of Retrievals and Insertions

To analyze the cost of maintaining an index and retrieving keys we need to know how many pages must be fetched from the backup store into main store and how many pages must be written onto the backup store. For our analysis we make the following assumption: Any page, whose content is examined or modified during a single retrieval, insertion, or deletion of a key, is fetched or paged out respectively exactly once. It will become clear during the course of this paper that a paging area to hold $h+1$ pages in main store is sufficient to do this.

Any more powerful paging scheme, like e.g., keeping the root page permanently locked in main store, will, of course, decrease the number of pages which must be fetched or paged out. We will not, however, analyze such schemes, although we have used them in our experiments.

Denote by $f_{\min}$ ($f_{\max}$) the minimal (maximal) number of pages fetched, and by $w_{\min}$ ($w_{\max}$) the minimal (maximal) number of pages written.

Cost of Retrieval. From the retrieval algorithm it is clear that for retrieving a single key we get

$$f_{\min}=1; \quad f_{\max}=h; \quad w_{\min}=w_{\max}=0.$$

Cost of Insertion. For inserting a single key the least work is required if no page splitting occurs, then

$$f_{\min}=h; \quad w_{\min}=1.$$

Most work is required if all pages in the retrieval path including the root page split into two. Since the retrieval path contains h pages and we have to write a new root page, we get:

$$f_{\max}=h; \quad w_{\max}=2h+1.$$

Note that h always denotes the height of the old tree. Although this worst bound is sharp, it is not a good measure for the amount of work which must generally be done for inserting one key.

If we consider an index in which keys are only retrieved or inserted, but no keys are deleted, then we can derive a bound for the average amount of work to be done for building an index of I keys as follows:

Each page split causes one (or two if the root page splits) new pages to be created. Thus the number of page splits occurring in building an index of I items is bounded by $n(I)-1$, where $n(I)$ is the number of pages in the tree. Since each page has at least k keys, except the root page which may have only 1, we get: $n(I) \leqq \frac{I-1}{k}+1$. Each single page split causes at most 2 additional pages to be written. Thus the average number of pages written per single key insertion due to page splitting is bounded by

$$(n(I)-1)\cdot\frac{2}{I}<\frac{2}{k}.$$

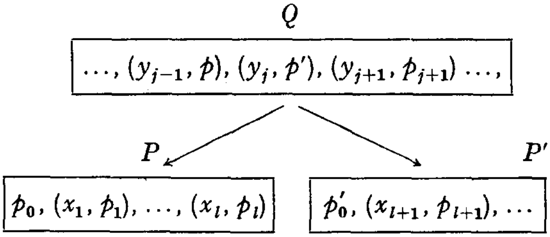
A page split does not require any additional page retrievals. Thus in the average for an index without deletions we get for a single insertion:

$$f_a=h;\quad w_a<1+\frac{2}{k}.$$

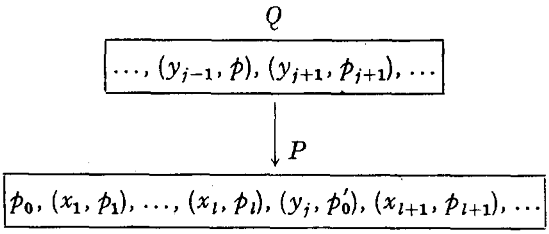
6. Deletion Process

In a dynamically changing index it must be necessary to delete keys. The algorithm of Fig. 6 deletes one key y from an index and maintains our data structure properly. It first locates the key, say y_i. To maintain the data structure properly, y_i is deleted if it is on a leaf, otherwise it must be replaced by the smallest key in the subtree whose root is $P(p_i)$. This smallest key is found by going from $P(p_i)$ along the p_0 pointers to the leaf page, say L, and taking the first key in L. Then this key, say x_1, is deleted from L. As a consequence L may contain fewer than k keys and a catenation or underflow between L and an adjacent brother is performed.

Catenation. Two pages P and P' are called *adjacent brothers* if they have the same father Q and are pointed to by adjacent pointers in Q. P and P' can be catenated, if together they have fewer than $2k$ keys, as follows: The three pages of the form

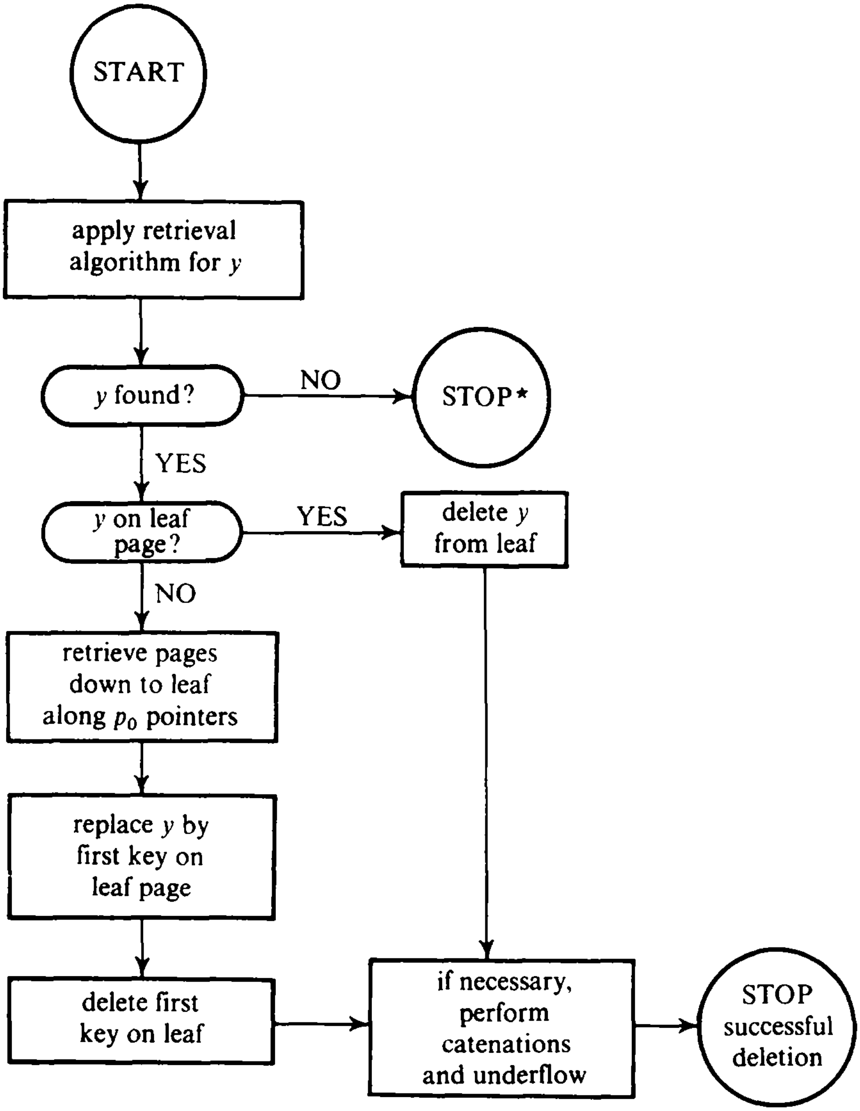

Q
$\dots,(y_{j-1},p),(y_j,p'),(y_{j+1},p_{j+1})\dots,$
P
P'
$p_0,(x_1,p_1),\dots,(x_l,p_l)$
$p_0',(x_{l+1},p_{l+1}),\dots$

can be replaced by two pages of the form:


Q
$\dots,(y_{j-1},p),(y_{j+1},p_{j+1}),\dots$
P
$p_0,(x_1,p_1),\dots,(x_l,p_l),(y_j,p_0'),(x_{l+1},p_{l+1}),\dots$

As a consequence of deleting the entry (y_j, p') from Q it is now possible that Q contains fewer than k keys and special action must be taken for Q. This process may propagate up to the root of the tree.

Underflow. If the sum of the number of keys in P and P' is greater than $2k$, then the keys in P and P' can be equally distributed, the process being called an underflow, as follows:


START
apply retrieval algorithm for y
y found?
NO
STOP*
YES
y on leaf page?
YES
delete y from leaf
NO
retrieve pages down to leaf along p_0 pointers
replace y by first key on leaf page
delete first key on leaf
if necessary, perform catenations and underflow
STOP successful deletion

* The key to be deleted is not in index, take appropriate action.

Fig. 6. Deletion algorithm

Perform the catenation between P and P' resulting in too large a P. This is possible since P is in main store. Now split P "in the middle" as described in Section 4 with some obvious minor modifications.

Note that underflows do not propagate. Q is modified, but the number of keys in it is not changed.

To illustrate the deletion process consider the index in Fig. 2. Deleting key 9 results in the index in Fig. 5.

7. Cost of Deletions

For a successful deletion, i.e., if the key y to be deleted is in the index, the least amount of work is required if no catenations or underflows are performed and y is in a leaf. This requires:

$$f_{\min}=h; \qquad w_{\min}=1.$$

If y is not in a leaf and no catenations or underflows occur, then

$$f=h; \qquad w=2.$$

A maximal amount of work must be done if all but the first two pages in the retrieval path are catenated, the son of the root in the retrieval path has an underflow, and the root is modified. This requires:

$$f_{\max}=2h-1; \qquad w_{\max}=h+1.$$

As in the case of the insertion process the bounds obtained are sharp, but very far apart and assumed rarely except in pathological examples. To obtain a more useful measure for the average amount of work necessary to delete a key, let us consider a "pure deletion process" during which all keys in an index I are deleted, but no keys are inserted.

Disregarding for the moment catenations and underflows we may get $f_1=h$ and $w_1=2$ for each deletion at worst. But this is the best bound obtainable if one considers an example in which keys are always deleted from the root page.

Each deletion causes at most one underflow, requiring $f_2=1$ additional fetches and $w_2=2$ additional writes.

The total number of possible catenations is bounded by $n(I)-1$, which is at most $\frac{I-1}{k}$. Each catenation causes 1 additional fetch and 2 additional writes, which results in an average

$$f_3=\frac{1}{I}\left(\frac{I-1}{k}\right)<\frac{1}{k}$$
$$w_3=\frac{2}{I}\left(\frac{I-1}{k}\right)<\frac{2}{k}.$$

Thus in the average we get:

$$f_a\leqq f_1+f_2+f_3<h+1+\frac{1}{k}$$
$$w_a\leqq w_1+w_2+w_3<2+2+\frac{2}{k}=4+\frac{2}{k}.$$

8. Page Overflow and Storage Utilization

In the scheme described so far utilization of back-up store may be as low as 50% in extreme cases—disregarding the root page—if all pages contain only k keys. This could be improved by avoiding certain page splits.

An *overflow* between two adjacent brother pages P and P' can be performed as follows: Assume that a key must be inserted in P and P is already full, but P' is not full. Then the key is inserted into the key-sequence in P and an underflow as described in Section 6 between the resulting sequence and P' is performed. This avoids the need to split P into two pages. Thus a page will be split only if both adjacent brothers are full, otherwise an overflow occurs.

In an index without deletions overflows will increase the storage utilization in the worst cases to about 66%. If both insertions and deletions occur, then the storage utilization may of course again be as low as 50%. For most practical applications, however, storage utilization should be improved appreciably with overflows.

One could, of course, consider a larger neighborhood of pages than just the adjacent brothers as candidates for overflows, underflows, and catenations and increase the minimal storage occupancy accordingly.

Bounds for the cost of insertions for a scheme with overflows are easily derived as:

$$f_{\min} = h; \qquad w_{\min} = 1;$$

$$f_{\max} = 3h - 2; \quad w_{\max} = 2h + 1.$$

For a pure insertion process one obtains as bounds for the average cost:

$$f_a < h + 2 + \frac{2}{k}; \quad w_a < 3 + \frac{2}{k}.$$

It is easy to construct examples in which each insertion causes an overflow, thus these bounds cannot be improved very much without special assumptions about the insertion process.

9. Maintenance Cost for Index with Insertions and Deletions

The main purpose of this paper is to develop a data structure which allows economical maintenance of an index in which retrievals, insertions, and deletions must be done in any order. We will now derive bounds on the processing cost in such an environment.

The derivation of bounds for retrieval cost did not make any assumptions about the order of insertions or deletions, so they are still valid. Also, the minimal and maximal bounds for the cost of insertions and deletions were derived without any such assumptions and are still valid. The bounds derived for the average cost, however, are no longer valid if insertions and deletions are mixed.

The following example shows that the upper bounds for the average cost cannot be improved appreciably over the upper bounds of the cost derived for a single retrieval or deletion.

Example. Consider the trees T_2 in Fig. 2 and T_5 in Fig. 5. Deleting key 9 from T_2 leads to T_5, and inserting key 9 in T_5 leads back to T_2. Consider a sequence of alternating deletions and insertions of key 9 being applied starting with T_2.

Case 1. No page overflows, but only page splits occur:

i) Each deletion of key 9 from T_2 requires:
3 retrievals to locate key 9, namely pages 1, 2, 6.
1 retrieval of brother 5 of page 6 to find out that pages 5 and 6 can be catenated.
2 pages, namely 5 and 2 are modified and must be written. Pages 6 and 3 are deleted from the tree T_2.
Thus $f=5$ and $w=2$. But $f=5=2h-1=f_{\max}$ and $w=2=h-1=w_{\max}-2$.

ii) Each insertion of key 9 into T_5 requires:
2 retrievals to locate slot for 9 in page 5.
5 pages must be written, namely 1, 2, 3, 5, 6.
Thus

$$f=2=h=f_{\max}$$

$$w=5=2h+1=w_{\max}.$$

Case 2. Consider a scheme with page overflows.

i) Deletion of key 9 leads to the same results as in Case 1.

ii) Insertion of key 9 requires:
2 retrievals to locate slot for 9 on page 5.
2 retrievals of brothers 4 and 7 of 5 to find out that 5 must be split.
5 pages must be written as in Case 1.
Thus:

$$f=4=3h-2=f_{\max}$$

$$w=5=2h+1=w_{\max}.$$

Analogous examples can be constructed for arbitrary h and k.

From the analysis it is clear that the performance of our scheme depends on the actual sequence of insertions and deletions. The interference between insertions and deletions may degrade the performance of the scheme as opposed to doing insertions or deletions only. But even in the worst cases this interference degrades the performance at most by a factor of 3.

It is an open question how important this interference is in any actual applications and how relevant our worst case analysis is. Although the derivable cost bounds are worse, the scheme with overflows performed better in our experiments than the scheme without overflows.

10. Choice of *k*

The performance of our scheme depends on the parameter k. Thus care should be taken in choosing k to make the performance as good as possible.

To obtain a very rough approximation to the performance of the scheme we make the following assumptions:

	Retrieval	Insertion in index without deletions and without overflows	Deletion in index without insertions, with or without overflows	Insertion in index without deletions, but with overflow	Insertion in index with deletions, without overflow	Deletion in index with insertions, with or without overflows	Insertion in index with deletion, with overflow
min	$f=1$ $w=0$	$f=h$ $w=1$	$f=h$ $w=1$	$f=h$ $w=1$	$f=h$ $w=1$	$f=h$ $w=1$	$f=h$ $w=1$
Average as derived in paper	$f \leqq h$ $w=0$	$f=h$ $w<1+\frac{2}{k}$	$f<h+1+\frac{1}{k}$ $w<4+\frac{2}{k}$	$f \leqq h+2+\frac{2}{k}$ $w \leqq 3+\frac{2}{k}$	$f=h$ $w \leqq 2h+1$	$f \leqq 2h-1$ $h-1 \leqq u \leqq h+1$	$f \leqq 3h-2$ $w \leqq 2h+1$
max	$f=h$ $w=0$	$f=h$ $w=2h+1$	$f=2h-1$ $w=h+1$	$f=3h-2$ $w=2h+1$	$f=h$ $w=2h+1$	$f=2h-1$ $w=h+1$	$f=3h-2$ $w=2h+1$

f = number of pages fetched
w = number of pages written
I = size of index set
h = height of B-tree
k = parameter of B-tree of pages
u = best upper bound obtainable for w

Fig. 7. Table of costs for a single retrieval, insertion, or deletion of a key

i) The time spent for each page which is written or fetched can be expressed in the form:

$$\alpha + \beta(2k+1) + \gamma \ln(\nu k+1)$$

α fixed time spent per page, e.g., average disc seek time plus fixed CPU overhead, etc.

β transfer time per page entry.

γ constant for the logarithmic part of the time, e.g., for a binary search.

ν factor for average page occupancy, $1 \leqq \nu \leqq 2$.

We assume that modifying a page does not require moving keys within a page, but that the necessary channel subcommands are generated to write a page by concatenating several pieces of information in main store. This is the reason for our assumption that fetching and writing a page takes the same time.

i) The average number of pages fetched and written per single transaction in an environment of mixed retrievals, insertions, and deletions is approximately proportional—see Fig. 7—to h, say δh. The total time T spent per transaction can then be approximated by:

$$T \approx \delta h\big(\alpha + \beta(2k+1) + \gamma \ln(\nu k+1)\big).$$

Approximating h itself by: $h \approx \log_{\nu k+1}(I+1)$ where I is the size of the index, we get: $T \approx T_a = \delta \log_{\nu k+1}(I+1)\,\big(\alpha + \beta(2k+1) + \gamma \ln(\nu k+1)\big)$.

Now one easily obtains the minimum of T_a if k is chosen such that:

$$\frac{\alpha}{\beta} = 2\Big(\frac{(\nu k+1)}{\nu} \ln(\nu k+1)\Big) - (2k+1) = f(k, \nu).$$

Neglecting CPU time, k is a number which is characteristic for the device used as backup store. To obtain a near optimal page size for our test examples we assumed $\alpha = 50$ ms and $\beta = 90$ μs. According to the table in Fig. 8 an acceptable choice should be $64 < k < 128$. For reasons of programming convenience we chose $k = 60$ resulting in a page size of 120 entries.

k	$f(k, 1)$	$f(k, 1.5)$	$f(k, 2)$
2.00000$E+00$	1.59167$E+00$	2.39356$E+00$	3.04718$E+00$
4.00000$E+00$	7.09437$E+00$	9.16182$E+00$	1.07750$E+01$
8.00000$E+00$	2.25500$E+01$	2.74591$E+01$	3.11646$E+01$
1.60000$E+01$	6.33292$E+01$	7.42958$E+01$	8.23847$E+01$
3.20000$E+01$	1.65769$E+02$	1.89265$E+02$	2.06334$E+02$
6.40000$E+01$	4.13670$E+02$	4.62662$E+02$	4.97915$E+02$
1.28000$E+02$	9.96831$E+02$	1.09726$E+03$	1.16911$E+03$
2.56000$E+02$	2.33922$E+03$	2.54299$E+03$	2.68826$E+03$
5.12000$E+02$	5.37752$E+03$	5.78842$E+03$	6.08075$E+03$
1.02400$E+03$	1.21625$E+04$	1.29881$E+04$	1.35748$E+04$
2.04800$E+03$	2.71506$E+04$	2.88062$E+04$	2.99818$E+04$
4.09600$E+03$	5.99647$E+04$	6.32806$E+04$	6.56343$E+04$
8.19200$E+03$	1.31269$E+05$	1.37906$E+05$	1.42617$E+05$
1.63840$E+04$	2.85235$E+05$	2.98514$E+05$	3.07938$E+05$
3.27680$E+04$	6.15877$E+05$	6.42442$E+05$	6.61292$E+05$
6.55360$E+04$	1.32258$E+06$	1.37572$E+06$	1.41342$E+06$

Fig. 8. The function $f(k, \nu)$ for optimal choice of k

The size of the index which can be stored for $k = 60$ in a page tree of a certain height can be seen from Fig. 9.

Height of page tree	Minimum index size	Maximum index size
1	1	120
2	121	14 640
3	7441	1 771 560
4	453 961	214 358 880

Fig. 9. Height of page tree and index size

11. Experimental Results

The algorithms presented here were programmed and their performance measured during various experiments. The programs were run on an IBM 360/44 computer with a 2311 disc unit as a backup store. For the index element size chosen (14 8-bit characters) and index size generally used (about 10000 index elements), the average access mechanism delay for this unit is about 50 ms, after which information transfer takes place at the rate of about 90 μs per index element. From these two parameters, our analysis predicts an optimal page size ($2k$) on the order of 120 index elements.

The programming included a simple demand paging scheme to take advantage of available core storage (about 1250 index elements' worth) and thus to attempt to reduce the number of physical disc operations. In the following section by *virtual disc read* we mean a request to the paging scheme that a certain disc page be available in core; a virtual disc read will result in a physical disc read only of there is no copy of the requested disc page already in the paging area of core storage. A *virtual disc write* is defined analogously.

At the time of this writing ten experiments had been performed. These experiments were intended to give us an idea of what kind of performance to expect, what kind of storage utilization to expect, and so forth. For us the specification of an experiment consists of choosing

1) whether or not to permit overflows on insertion,
2) a number of index elements per page, and
3) a sequence of transactions to be made against an initially empty index.

At several points during the performance of an experiment certain performance variables are recorded. From these the performance of the algorithms according to various performance measures can be deduced; to wit

1) % storage utilization
2) average number of virtual disc reads/transaction
3) average number of physical disc reads/transaction
4) average number of virtual disc writes/insertion or deletion
5) average number of physical disc writes/insertion or deletion
6) average number of transactions/second.

We now summarize the experiments. Each experiment was divided into several phases, and at the end of each of these the performance variables were measured. Phases are denoted by numbers within parentheses.

$E1$: 25 elements/page, overflow permitted.
(1) 10000 insertions sequential by key,
(2) 50 insertions, 50 retrievals, and 100 deletions uniformly random in the key space.

$E2$: 120 elements/page; otherwise identical to $E1$.

$E3$: 250 elements/page; otherwise identical to $E1$.

$E4$: 120 elements/page, overflow permitted.
(1) 10000 insertions sequential by key,
(2) 1000 retrievals uniformly random in key space,
(3) 10000 sequential deletions.

$E5$: 120 elements/page, overflow *not* permitted.
(1) 5000 insertions uniformly random in key space,
(2) 1000 retrievals uniformly random in key space,
(3) 5000 deletions uniformly random in key space.

$E6$: Overflow permitted; otherwise identical to $E5$.

$E7$: 120 elements/page, overflow permitted.
(1) 5000 insertions sequential by key,
(2) 6000 each insertions, retrievals, and deletions uniformly random in key space.

*E*8: 120 elements/page, overflow permitted.

(1) 15000 insertions uniformly random in key space,

(2) 100 each insertions, deletions, and retrievals uniformly random in key space.

*E*9: 250 elements/page; otherwise identical to *E*8.

*E*10: 120 elements/page, overflow permitted.

(1) 100000 insertions sequential by key,

(2) 1000 each insertions, deletions, and retrievals uniformly random in key space,

(3) 100 group retrievals uniformly random in key space, where a group is a sequence of 100 consecutive keys (statistics on the basis of 10000 transactions),

(4) 10000 insertions sequential by key, to merge uniformly with the elements inserted in phase (1).

	% Storage used	*VR/T**	*PR/T*	*VW/I* or *D*	*PW/I* or *D*	*T*/sec
*E*1 (1)	99.8	2.2	0	2.3	0.04	66.1
*E*1 (2)	91.5	4.4	1.62	2.7	1.5	6.6
*E*2 (1)	99.2	1.0	0	1.0	0.008	94.5
*E*2 (2)	87.3	2.5	1.15	1.3	1.1	6.7
*E*3 (1)	97.6	1.0	0	1.0	0.004	100.0
*E*3 (2)	84.7	2.4	1.08	1.3	1.1	5.2
*E*4 (1)	99.2	1.0	0	1.0	0.008	94.5
*E*4 (2)	99.2	2.0	—	—	—	19.5
*E*4 (3)	—	2.0	0.01	2.0	0	74.1
*E*5 (1)	67.1	1.0	0.55	1.0	0.56	17.0
*E*5 (2)	67.1	2.0	0.83	—	—	18.2
*E*5 (3)	—	4.0	0.68	2.2	0.65	12.4
*E*6 (1)	86.7	1.1	0.55	1.1	0.54	17.1
*E*6 (2)	86.7	2.0	0.79	—	—	24.3
*E*6 (3)	—	4.0	0.65	2.2	0.62	13.4
*E*7 (1)	96.9	1.0	0	1.0	0.008	111.9
*E*7 (2)	76.8	2.3	0.83	1.3	0.88	13.1
*E*8 (1)	84.5	1.3	0.87	1.3	0.85	10.1
*E*8 (2)	83.9	3.7	1.00	3.0	1.00	9.5
*E*9 (1)	86.4	1.1	0.84	1.0	0.82	8.5
*E*9 (2)	85.2	2.3	0.94	1.1	0.96	8.2
*E*10 (1)	99.8	1.9	0	1.9	0.008	91.7
*E*10 (2)	82.1	4.1	1.94	1.8	1.54	4.2
*E*10 (3)	82.1	4.0	0.03	—	—	75.7
*E*10 (4)	83.8	2.2	0.10	2.2	0.11	38.0

* This statistic is unnecessarily large for deletions, due to the way deletions were programmed. To find the *necessary* number of virtual reads, for sequential deletions subtract one from the number shown, and for random deletions subtract one and multiply the result by about 0.5.

References

1. Adelson-Velskii, G. M., Landis, E. M.: An information organization algorithm. DANSSSR, **146**, 263–266 (1962).
2. Foster, C. C.: Information storage and retrieval using AVL trees. Proc. ACM 20th Nat'l. Conf. 192–205 (1965).
3. Landauer, W. I.: The balanced tree and its utilization in information retrieval. IEEE Trans. on Electronic Computers, Vol. EC-12, No. 6, December 1963.
4. Sussenguth, E. H., Jr.: The use of tree structures for processing files. Comm. ACM, **6**, No. 5, May 1963.

Prof. Dr. R. Bayer
Dept. of Computer Science
Purdue University
Lafayette, Ind. 47907
U.S.A.

Dr. E M. McCreight
Palo Alto Research Center
3180 Porter Drive
Palo Alto, Calif. 94304
U.S.A.