VU Research Portal

Two Access Methods Using Compact Binary Trees

Jonge, W. de; Tanenbaum, A.S.; Riet, R.P. van de

published in

IEEE Transactions on Software Engineering
1987

DOI (link to publisher)

[10.1109/TSE.1987.233491](https://doi.org/10.1109/TSE.1987.233491)

document version

Publisher's PDF, also known as Version of record

[Link to publication in VU Research Portal](#)

citation for published version (APA)

Jonge, W. D., Tanenbaum, A. S., & Riet, R. P. V. D. (1987). Two Access Methods Using Compact Binary Trees. *IEEE Transactions on Software Engineering*, SE-13(7), 799-810. <https://doi.org/10.1109/TSE.1987.233491>

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal ?

Take down policy

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:

vuresearchportal.ub@vu.nl

Two Access Methods Using Compact Binary Trees

WIEBREN DE JONGE, ANDREW S. TANENBAUM, MEMBER, IEEE, AND REIND P. VAN DE RIET

Abstract—It is shown how a highly compact representation of binary trees can be used as the basis of two access methods for dynamic files, called BDS-trees and S-trees, respectively. Both these methods preserve key-order and offer easy and efficient sequential access. They are different in the way the compact binary trees are used for searching. With a BDS-tree the search is a digital search using binary digits. Although the S-tree search is performed on a bit-by-bit basis as well, it will appear to be slightly different. Actually, with S-trees the compact binary trees are used to represent separators at low storage costs. As a result, the fan-out, and thus performance, of a B-tree can be improved by using within each index page an S-tree for representing separators efficiently.

Index Terms—Access method, B-tree, file, searching, tree.

I. INTRODUCTION

IN many applications, files consist of many records, each containing a key by which the record is accessed. Examples are personnel files keyed on employee number, inventory files keyed on part number, and bank files keyed on account number. Typical operations are: given a key, read, write or delete the corresponding record; insert a new record in such a way that it can later be efficiently retrieved by its key; read the entire file sequentially, in key order; and perform a range query. To perform all these operations efficiently, the program (or database system) must be able to map a key onto the disk address of the corresponding record, without this mapping itself requiring many disk accesses.

Several methods for performing the mapping are known, including B-trees [1]–[3], Trie Hashing [12], Extendible Hashing [7], and Linear Hashing [10], [11]. The access methods that will be developed in this paper, are, like B-trees and Trie Hashing, order preserving, which makes sequential access to the records easy. Linear Hashing and Extendible Hashing do not have this property, and will not be considered further. B-trees, Trie Hashing, and our two methods all require some kind of data structure, or index, used to convert keys to disk addresses. This index must be updated as the file changes. The various methods use different data structures for the index. Actually, our methods mainly differ from B-trees and Trie Hashing in the way the index is represented.

In the following sections, we will describe our ideas in detail. In Section II, we present the necessary background information about access methods and keys. Then the bi-

nary digital search tree (BDS-tree), a first method using binary trees, is discussed and illustrated by an example in Section III. In Section IV we turn to a variation, the separator tree (S-tree), which offers better storage utilization. It is this variation that shows how separators can be represented by binary trees. Section V deals with the compact representation of the binary trees and the algorithms using this representation. In Section VI we compare our methods to Trie Hashing [12] and we show how the fan-out of B-trees can be increased by representing the search information within each page by using our S-trees. Finally, in Section VII we summarize our results.

II. KEYS, BUCKETS, SEPARATORS, AND FAN-OUT

As mentioned earlier, every record is assumed to have a (unique) key by which it is accessed. Throughout this paper, we will focus our attention on the key, but every record has a non-key portion as well, of course. A *key* is a string of bits, 0's, and 1's. Keys need not have any fixed maximum length.

Any set of distinct keys can be uniquely arranged in order, from smallest to largest. The ordering we will use is similar to lexicographic order. To compare two keys, examine each from the left, bit by bit, until they differ in some bit position. The key containing a 0 in that position is defined as the smaller of the two. A *prefix* of a key is a substring starting at the left end. Note that if one of the two keys is a prefix of the other, in our scheme zeros must be appended to the shorter until the comparison terminates.

If the actual record keys are integers, reals, characters, etc., to preserve key order, they must be converted to bit strings in such a way as to preserve this ordering. Sixteen bit numbers have the same ordering when viewed as unsigned integers compared numerically as when viewed as bit strings compared lexicographically, but signed integers and reals do not. Long and highly redundant keys lead to less compact search information. Therefore, if possible, also (an order preserving) text compression should be applied to the keys, before using them for an index. (This does not imply that the keys in the data records should be compressed too!) For example, character strings consisting exclusively of letters should be mapped with $a = 0$, $b = 1$, $c = 2$, etc., so each character can be represented in 5 bits, instead of, say, 8 bits.

In most applications, files are too large to be kept in main memory. Therefore, they are kept on disk or other random access secondary storage medium. We will call the unit of data transferred between memory and disk in

Manuscript received December 28, 1984; revised March 29, 1985.

The authors are with the Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands.

IEEE Log Number 8714556.

a single operation a *bucket*, page, or disk block. Typically a bucket is a sector or a track. We will use b to denote the capacity of a bucket, i.e., the number of records that can be stored in one bucket. Each bucket has a unique symbolical or physical address. A basic assumption beyond all access methods is that the time necessary to access a data structure in main memory is negligible compared to the time necessary to access a bucket on secondary storage.

While presenting our two methods in Sections III and IV we will assume, just for our convenience, that the index entirely fits in main memory. Since our way of representing the search information is very compact, this assumption may be realistic even for quite large files.

Still, despite our compact representation, an index may become too large to be entirely kept in main memory. An efficient way to structure such large indexes in a paged, multilevel index organization is offered by B^+ -trees [3]. A B^+ -tree organizes the data into a multilevel index part, containing search information, and a file with the records themselves. Within each page of the index, the search information consists of pointers and *separators* [2]. The performance of B^+ -trees can be improved by increasing the number of (separator, pointer)-pairs per index block, i.e., the fan-out of each index block since a higher fan-out leads to a smaller depth for the index tree of a certain file, and hence to fewer disk accesses to fetch the data.

Since our so-called S-trees of Sections IV and V will appear to be a very compact way of representing (separator, pointer)-pairs, the fan-out, and thus performance, of a B^+ -tree can be increased by using an S-tree for representing the search information within each index page (see Section VI). Thus, the reader has to keep in mind that the assumption about the index entirely fitting in main memory is not crucial for the usefulness of the ideas to be presented.

III. BINARY DIGITAL SEARCH TREES

The method to be presented in this section uses a binary tree to map keys onto bucket addresses. We have chosen to call this access method a Binary Digital Search Tree (BDS-tree), since a key is examined bit-by-bit during the search rather than being compared in its entirety to other keys [9]. As customary with binary trees we distinguish between *internal nodes*, which have exactly two descendants, and external nodes or *leaves*, which do not have descendants. Internal nodes are used to discriminate during the search, while only leaves may correspond to buckets on disk. For convenience, we assume that *each* leaf corresponds to a bucket. Therefore, we need to distinguish between buckets that contain at least one record, and those that are completely empty, since no actual disk storage will be allocated for the latter, even though they occur in the access tree. Empty buckets (and their corresponding leaves) will be called *dummies*.

Fig. 1 illustrates a small binary tree with four internal nodes and five leaves, one of which is a dummy. A known property concerning binary trees is that the number of

Fig. 1. A binary tree showing the paths.

leaves is one more than the number of internal nodes. This property underlies our search algorithm, as explained in Section V-D.

Another thing to note about Fig. 1 is the labeling of the edges. Left edges are labeled 0; right edges are labeled 1. Using these numbers, every bucket can be labeled by its *path* from the root. Bucket 1 has path 000, bucket 2 has path 001, bucket 3 (a dummy) has path 01, etc.

A. Searching

Given a tree of the kind shown in Fig. 1 and a record key, it is possible to determine which bucket the record belongs in. The algorithm is simple: starting from the root of the tree, the key is examined bit-by-bit from the left end. Each time a 0 is encountered, go left. Each time a 1 is encountered, go right. When a leaf is reached, stop. The corresponding bucket is where the record belongs. In other words, the key belongs in the bucket whose path is a prefix of the key. For example, key 00101 belongs in bucket 2 and key 110011 belongs in bucket 5.

Observe that *bucket numbers* are different from *bucket addresses*. The bucket numbers go from 1 to some maximum in tree-traversal order, that is, in the order the leaves would be encountered when traversing the entire tree. The issue of how to convert bucket numbers to bucket addresses will be discussed in detail later. For the time being just assume the existence of a table indexed by bucket number that contains the address of each bucket (with a special bucket address, say 0, for dummies). By simply scanning this table, it is possible to find all the buckets in order, and therefore read the file sequentially starting at the beginning or at an arbitrary bucket.

B. Insertions

When a new record is to be inserted, we can distinguish three cases:

- the required bucket is partially filled.
- the required bucket is a dummy (i.e., this is the first entry).
- the required bucket is full.

In the first case, insertion is trivial: the required bucket is read in, the new record inserted into it, and the bucket is rewritten to the disk. In the second case, the dummy bucket is converted into a real one. Since dummy buckets do not have disk space allocated to them, inserting a record in a dummy bucket will require allocating a new disk bucket. If a record with key 01100 were inserted into the tree of Fig. 1, the result would be Fig. 2(a).

The case of inserting a record into an already full bucket is more involved. The bucket must be split into two buckets, and all $b + 1$ keys (the b keys that previously filled

Fig. 2. Insertions.

the bucket, plus the new one to be inserted) are distributed over the two new buckets. As an example, consider what would happen if an insertion was executed in bucket 2 of Fig. 2(a) when this bucket is already full. The node would be converted to an internal node, with two buckets under it, as shown in Fig. 2(b). (Note that a split causes all subsequent buckets to be renumbered.) Whereas prior to the split, all keys starting with 001 went into bucket 2, after the split, those keys beginning with 0010 go into bucket 2, and those keys beginning with 0011 go into bucket 3. This redistribution applies both to the $b + 1$ keys of bucket 2 just prior to the split, and keys occurring subsequently.

In most cases, roughly half the keys will go into bucket 2 and the other half into bucket 3. However, in the worst case, all the keys might go into one of the buckets. For example, suppose $b = 4$, and prior to the split bucket 2 contained keys 0011000, 0011001, 0011011, and 0011110. Now key 0011101 is to be inserted. When all five keys are redistributed into buckets 2 and 3 of Fig. 2(b), unfortunately all five of them fall into bucket 3, which must itself be split, yielding Fig. 2(c). Fortunately, three of these go into bucket 3 and two go into bucket 4. Bucket 2 becomes a dummy. If the five keys had all started with 001110, bucket 4 in Fig. 2(c) would also have to be split (making bucket 3 a dummy), and its left child would have to have been split too, making its right child into a dummy as well [see Fig. 2(d)].

In general, if all the keys in a bucket to be split have long identical prefixes, many new levels may have to be created in the access tree to distinguish them, since each level corresponds to a single key bit. This tendency may or may not be of any importance in practice. In any event, Section IV deals with a variant of the BDS-tree that can guarantee a 50–50 split every time.

C. Deletions

In principle, deletions can be handled by just removing the record in question without changing the access tree structure. In practice, doing so might lead to many buckets with a utilization of less than 50 percent, thus wasting

Fig. 3. Deletions.

disk storage. A simple way to pursue a more acceptable degree of disk utilization is to make a check after each deletion as to see if the bucket being deleted from still is more than, say, half full. If not, it may be possible to combine the bucket with its immediate sibling. As an example, consider the effect of a deletion from bucket 6 in Fig. 2(c). If buckets 6 and 7 can now be merged into a single bucket, one bucket can be freed, and the tree reorganized as shown in Fig. 3(a). If a subsequent deletion from bucket 4 allows buckets 3 and 4 to be merged, we get Fig. 3(b), which in turn can be converted in Fig. 3(c) by merging bucket 3 with dummy bucket 2.

From Figs. 2 and 3 it can be seen how the access tree dynamically changes its shape as records are inserted and deleted from the file.

D. Some Properties of BDS-Trees

Assuming that the access tree can be kept in main memory, a successful search always takes exactly one disk access, since the search process yields the correct bucket every time. When searching for a key not present in the file, no disk access at all will be needed if the search terminates in a dummy bucket. When the search terminates in a nonempty bucket, exactly one disk access is needed to determine that the key is not present. Depending on what fraction of the (unsuccessful) searches end in dummy buckets, the mean number of disk accesses per read request will be between 0 and 1.

With BDS-trees a high storage utilization cannot be guaranteed. If every split would put exactly half the keys in one of the new buckets and the other half in the other new bucket, storage utilization in these new buckets will be 50 percent initially. As they subsequently fill up, the

efficiency will rise toward 100 percent. The mean efficiency would fall somewhere between these limits, typically about 70 percent [13].

However, if many bad splits occur, some buckets will be nearly empty, and storage utilization may be lower. For example, a 90–10 percent split would still yield a 50 percent storage utilization on the average, initially, but if the nearly full bucket is filled up and then split again, we have 1.1*b* entries spread over three buckets, for a 37 percent efficiency. Thus, a nonuniform key distribution usually leads to a storage utilization lower than 70 percent.

Note that a nonuniform key distribution does not always have a bad influence on the storage utilization or on the compactness of the tree. Suppose, for example, that a huge number of records whose keys begin with 00110 were to be inserted into Fig. 3(a). If these keys were randomly distributed starting at bit 6, few dummies would occur in the search tree, even though it is severely unbalanced. Furthermore, the storage utilization would be just the usual 70 percent. Put in other words, even if the key distribution exhibits large clusters, the storage utilization does not suffer. Also the storage costs of the search tree would not be affected, since the representation of an unbalanced tree costs no more bits than of a balanced one with the same number of leaves. To result in many “bad” splits, the key distribution must be more pathological; ordinary clustering is not enough.

It is also worth noting the effect of dummy buckets on the performance. As mentioned above, they make it possible to determine that certain keys are *not* present without making any disk accesses at all. Therefore, a higher percentage of dummies will usually lead to a lower average number of disk accesses required for an unsuccessful search. Since no disk storage is allocated for dummies, they do not waste disk space, but their negative effect is to increase the size of the access tree. However, as we will discuss later, this effect can be reduced to 3 bits per dummy.

As a final remark, we mention that, assuming some fixed bucket capacity, a certain set of keys has only one corresponding BDS-tree. Insertion of a certain set of keys in an initially empty BDS-tree will always result in the same, final BDS-tree, irrespective of the order in which these keys were inserted.

IV. SEPARATOR TREES

Although, in practice, the load factor for BDS trees may suffer from highly skewed key distributions, a modification of the method can guarantee that all splits are close to 50–50, for all key distributions, no matter how bad. The trees used here will be called S-trees (separator trees) to distinguish them from the BDS-trees used above.

A. Searching

With BDS-trees, each bucket is effectively assigned a range of keys, namely all the keys that begin with its path. Thus in Fig. 1, bucket 1 gets all keys beginning with 000, bucket 2 gets all keys beginning with 001, bucket 3 gets

<u>bucket</u>	<u>from</u>	<u>to</u>
1	00000...	00011...
2	00100...	00111...
3	01000...	01111...
4	10000...	10111...
5	11000...	11111...

(a)

<u>bucket</u>	<u>from</u>	<u>to</u>
1	00000...	00011...
2	00100...	01111...
3	-	-
4	10000...	10111...
5	11000...	11111...

(b)

Fig. 4. Bucket ranges for Fig. 1. (a) As a BDS-tree. (b) As an S-tree.

all keys beginning with 01, etc., as shown in Fig. 4(a). In an S-tree, only each nonempty bucket has a range, namely from its path up to but not including the path of the next nonempty bucket (or to 11111... for the rightmost nonempty bucket). To reiterate for emphasis, in an S-tree, unlike in a BDS-tree, the dummy buckets do not have ranges assigned to them. If a search leads to a dummy, the nearest nonempty bucket to the left must be inspected instead. A dummy always has a nonempty bucket to its left, since the insertion algorithm guarantees that the leftmost bucket (i.e., bucket 1) is nonempty. Fig. 4(b) shows the ranges assigned to each bucket of Fig. 1 when this figure is interpreted as an S-tree.

To make this point clearer, consider how a record with key 010 would be inserted into the S-tree of Fig. 1. As indicated in Fig. 4(b), this key falls within the range of bucket 2 and thus would go into that bucket instead of going into bucket 3, as would occur if the tree of Fig. 1 was interpreted as a BDS-tree. The reason is intuitively clear: to achieve a high storage efficiency, we want to prevent situations like this—a bucket with just one record.

Put in other terms, due to the way ranges are assigned to buckets, in any S-tree the union of the ranges of the currently existing nonempty buckets exhausts the entire key space. Barring overflows, no new buckets are ever created, since every key belongs in some existing bucket. Only when an overflow occurs is a new bucket created, and then, as we shall see shortly, the records can be evenly divided between the two buckets.

With BDS-trees, the union of the nonempty bucket ranges does not exhaust the key space, since dummy buckets “own” the keys starting with their paths. Hence, a dummy bucket must be converted to a real one when even one record falls in its range. By not assigning any range to the dummies in an S-tree, this problem is avoided.

B. Insertions

From the above reasoning, we can see that in S-trees new buckets are only created upon bucket overflow. Now we will show how it is possible to guarantee that the keys are evenly divided between the two buckets.

First, sort all $b + 1$ keys belonging to the overflowing

$k_0 = 10110000$
 $k_1 = 10110001$
 $k_2 = 10110010$
 $k_3 = 10110101$
 $k_4 = 10110111$

(a)

(b)

(c)

Fig. 5. (a) Five keys. (b) Split using 1011001 as separator. (c) Split using 101101 as separator.

bucket. Then find the median key K_m . Next find the *median separator* S_m , defined as the shortest prefix of the median key such that $K_{m-1} < S_m \leq K_m$. In effect, the median separator divides the sorted keys in two subsets, with all keys prior to the median key remaining in the overflowing bucket, and it and its successors being put in a newly created bucket. In other words, the median separator divided the original range of the overflowing bucket into two parts. The first part becomes the new range of the overflowing bucket and the other part becomes the range of the new, nonempty bucket to its right.

As an example of how this works, consider what would happen if the five keys of Fig. 5(a) landed in bucket 4 of Fig. 1, thereby causing an overflow (for $b = 4$). The median key in the sorted list is K_2 (10110010). The median separator is 1011001 because $10110001 < 1011001 \leq 10110010$ and no shorter prefix of K_2 (e.g., 101100) has this property. A path is then created for this separator, and it is inserted into the tree, as shown in Fig. 5(b). Note that it makes no difference here if K_0 would have been chosen to be 10001000 instead of 10110000.

A split does not always lead to the substitution of the overfull node by a subtree with two nonempty buckets. Consider, for example, the situation of Fig. 6(a) with bucket 2 containing the keys 00110, 10010, 10100, and 10111. If key 10110 has to be inserted, an overflow oc-

Fig. 6. Second kind of splitting.

curs. Then the median key is 10100, with separator 101. The result is that 00110 and 10010 stay in bucket 2, while the others go into the new bucket 5 as indicated in Fig. 6(b).

In the above examples the keys of the overflowing bucket were evenly divided by using the median separator. A variant of this idea is not to use the median key for determining the separator, but instead another key from some interval around the median key. (We apply here the idea proposed in [2].) In our example of Fig. 5, choosing K_3 instead of K_2 leads to a shorter separator (101101) and a tree [see Fig. 5(c)] with fewer dummies, although in both cases the keys split equally good. In general, a trade-off must be made between a potentially worse division of the keys between the buckets, which eventually leads to a lower load factor, and a longer separator, which means more dummies.

If the interval is chosen to cover the entire bucket, then the algorithm produces the shortest possible separator, just as the split algorithm of the BDS-tree actually does. Thus, the S-trees generated then will have the "shape" of the corresponding BDS-trees. A difference will be that some nonempty buckets in the BDS-tree correspond to dummies in the S-tree, since in an S-tree these dummies will not be converted to a nonempty bucket until the nearest nonempty bucket to their left overflows (see also Section IV-D).

C. Deletions

The only possible merge in a BDS tree is to combine two siblings (that is, two nodes having the same parent. In Fig. 6(a), buckets 1 and 2 are siblings, but 2 and 3 are not.) and replace them and their common parent with a single bucket. In an S-tree a bucket can be merged with the nearest nonempty bucket to its left or right (i.e., its left or right nonempty neighbor), even if they are not siblings. A merge in an S-tree is performed by adding the records of the right-hand bucket to the left-hand one, and converting the right-hand bucket to a dummy, which results in extending the range of the left-hand bucket as required.

If in Fig. 6(b), the records of buckets 2 and 4 fit together in one bucket, the merge will be performed by adding the records of bucket 5 to bucket 2. Then bucket 5 is converted to a dummy. Since buckets 4 and 5 both are dummies now, the tree can be reduced and will become as in Fig. 6(a). Note that, unlike in BDS-trees, it is impossible to reduce the tree further, since reducing buckets 4 and 5 of Fig. 6(a) to one nonempty bucket would unintentionally change the ranges of buckets 2 and 5.

D. Comparison to BDS-Trees

If the entire S-tree can be kept in main memory, exactly one disk access is needed, both for successful and unsuccessful searches. This performance is slightly worse than with BDS-trees, where some unsuccessful searches can be completed without any disk accesses at all. With S-trees, no key range is assigned to dummies, so if a key leads to a dummy, the nonempty bucket to its left must be inspected anyway.

Unlike with BDS-trees, a certain set of keys has (assuming some fixed bucket capacity) usually more than one corresponding S-tree. Precisely which S-tree will result from the insertion of a certain set of keys in an initially empty tree, depends on the order these keys are inserted.

If the split interval used is not too large, it is guaranteed that an overflow leads to two (approximately) half full buckets. Thus, in case of pure insertions, bucket utilization will be between 50 and 100 percent and mean load factor will be about 70 percent [13]. This situation is equal to that of B-trees and usually better than that of BDS-trees (see results below). In addition, it is possible for S-trees (as for B-trees) to achieve a utilization even higher than 70 percent by merging buckets with more than one adjacent neighbor, and by only performing a split if an overflow cannot be resolved by reallocating some keys to the neighboring buckets. (Of course, if some keys are reallocated to a neighboring bucket, the corresponding separator must be adapted.) However, doing so makes insertions and deletions slightly more expensive because overflows and merges then require more work.

Another difference is that S-trees tend to have more dummies than BDS-trees, depending on how large an interval around the median key is chosen. If a large interval around the median key is chosen, a larger choice of separators is available. This makes it possible to choose a short one. Since a short separator corresponds to a shallow tree, it will, in general, require fewer dummies than a long separator, which corresponds to a deep tree. Thus, the larger the interval inspected, the shorter the separator, and the fewer the dummies. We have made measurements on the percentage of dummies as a function of the interval searched (expressed as a fraction of the bucket capacity). As an example, Fig. 7(a) shows the results for a list of 50 000 Dutch words when using a bucket capacity of 50 (i.e., $b = 50$) and a 5-bit-per-character encoding. The 50 000 words were inserted in random order.

On the other hand, the larger the interval searched for a separator, the worse the split can be, and the lower the load factor. If the interval searched is 20 percent of the bucket (10 percent on either side of the median key), for example, the worst possible split is 60–40 percent. With a 50 percent interval, the split can be as bad as 25–75 percent. Finally, if 100 percent of the bucket is searched for the shortest separator, in the worst case one of the resulting two nonempty buckets has only one record. Experimental results show that with split intervals up to about 40 percent there is almost no degradation of the load fac-

Fig. 7. (a) Percentage of dummies as a function of the split interval. (b) Load factor as a function of the split interval.

tor from the 70 percent derived in [13], but with larger intervals, the load factor drops off. As an example, Fig. 7(b) shows the results for the list of 50 000 words. For the same list the BDS-tree gave a load factor of 46 percent and a percentage of dummies of almost 14 percent.

When inserting random numbers as keys, both BDS-trees and S-trees gave the expected average load factor of 70 percent. Of course the size of the split interval had in this case no influence on the average load factor of the S-tree. The percentage of dummies was negligible (i.e., much less than 1 percent) for BDS-trees. For S-trees a split interval of 40 percent appeared to be sufficient to make the percentage of dummies negligible too. But, for a 20 percent split interval the percentage of dummies was roughly 25 percent and for 0 percent split interval this percentage was even about 77 percent.

As we already mentioned before, when a 100 percent split interval is used with an S-tree (i.e., the whole bucket is searched for the shortest separator), the resulting tree is similar, but not identical to the BDS-tree corresponding to the same keys. If at a certain moment a key lands in a dummy, the action taken differs for a BDS-tree and an S-tree. In the former case, the dummy is simply converted into a real bucket. In the latter case, the key is put into the nearest real bucket to the left of the dummy, assuming there is room for it. Thus the shape of BDS-tree and the corresponding S-tree using a 100 percent split interval will be identical, only some of the leaves of the tree that are buckets in the BDS-tree may be dummies in the S-tree. Therefore, an S-tree used with a 100 percent split interval

will have a better load factor, but also a higher percentage of dummies than the corresponding BDS-tree. For the example of Fig. 7 the BDS-tree gave a load factor of 46 percentage and a percentage of dummies of 13 percent, whereas these numbers for the S-tree using a 100 percent split interval were 50 and 20 percent, respectively.

By way of illustration, consider the insertion of the keys of Fig. 5(a) into bucket 4 of Fig. 1 again. If an S-tree with a 100 percent split interval is used, the tree of Fig. 5(c) is produced. If a BDS-tree is used the resulting tree would have the same shape, but the bucket with path 100 would be a dummy and the bucket with path 101100 would be nonempty. More difference shows up if the next key to be stored is 10111100. For both methods the search algorithm reaches the dummy with path 10111. With an S-tree, the key is put into the nearest nondummy bucket to the left, alongside K_3 and K_4 , since there is sufficient room. With a BDS-tree, the dummy with path 10111 would just be converted to a real bucket immediately. After insertion of this key, the BDS-tree thus has one dummy less than the corresponding S-tree, but uses one nonempty bucket more for storing the same number of keys.

In summary, by adjusting the split interval of an S-tree, one can vary the load factor and the number of dummies. Up to about 40 percent split interval leaves the load factor about 70 percent, while lowering the percentage of dummies (and thus the size of the index). With a larger split interval, still fewer dummies are created, but the load factor deteriorates. With a 100 percent split interval an S-tree almost approximates the behavior of a BDS-tree.

V. REPRESENTATION OF BDS- AND S-TREES

Up until now, we have been completely silent about how the binary trees should be encoded. The obvious representation of an internal node as two pointers is not, in general, the best one. Similarly, we have said nothing about how (or even where) bucket addresses are represented. They obviously could be kept in the leaves of the tree, but this is not the only place; nor is it necessarily the best one. In the following sections, these issues will be discussed in detail. A compact representation for BDS- and S-trees will be developed, and search algorithms using this representation will be presented.

A. How to Represent Binary Trees Compactly

The straightforward way to represent a binary tree, such as that of Fig. 1, is to have each internal node consist of two pointers. The sign bit of each pointer could distinguish between a pointer to an internal node and a bucket address. The root node of Fig. 1 would then contain two internal pointers; its left child would contain one internal pointer and one external pointer (to the dummy bucket); its right child would contain two external pointers. However, this *standard representation* is not compact enough, since too many bits (namely, two pointers) are needed for each bit of a separator in the tree.

A completely different, and much more compact, rep-

<u>tree</u>	<u>linear representation</u>
Fig.1	000111011
Fig.2(a)	000111011
Fig.2(b)	00010111011
Fig.2(c)	0001010111011
Fig.2(d)	00010101001111011
Fig.3(a)	00010101111
Fig.3(b)	000101111
Fig.3(c)	0001111
Fig.5(b)	0001110010100011111
Fig.5(c)	00011100101001111
Fig.6(a)	000111011
Fig.6(b)	00011100111

Fig. 8. Linear representation of binary trees.

resentation of a binary tree can be obtained by a preorder tree traversal, emitting a 0 for every internal (split) node visited and a 1 for every bucket (dummy or not) visited. When traversing the tree of Fig. 1, we first encounter three internal nodes (including the root), then buckets 1 and 2, then the dummy 3, then the remaining internal node, and finally buckets 4 and 5, for a bit string of 000111011. Since every binary tree has one more bucket than internal nodes, the bit string will always have one less 0 bits than 1 bits. Fig. 8 shows the bit strings corresponding to all the example trees we have used so far. We will call this representation of a binary tree the *linear representation*. In this linear representation the first 1 bit corresponds to bucket number 1, the second 1 bit corresponds to bucket number 2, and so on.

B. Where to Represent Bucket Addresses

Bucket addresses can be stored in one of two ways. In the first way, the bucket address is included in the bit string itself, directly following the 1 bit corresponding to the bucket. For example, if aaaa, bbbb, cccc, etc. represent bucket addresses, the tree of Fig. 1 would be represented as

0001aaaa1bbbb1cccc01dddd1eeee.

This representation requires fixed-size bucket addresses, although any size can be used. Dummies can be represented by any convenient bit pattern of the same length as a bucket address (e.g., all 0's).

A more convenient way of storing the bucket addresses is in a separate table, indexed by bucket number. In Fig. 9(a) we show the linear representation of the tree of Fig. 6(b). Fig. 9(b) shows the *address table* used to convert a bucket number, 1 to 6, to a bucket address. A bucket number is just the position of its leaf bit in the linear representation when zeros (internal nodes) are neglected, whereas the bucket address is a physical or symbolical address. When the tree is modified, the bucket numbers may change, but the bucket addresses assigned to the various buckets do not change. If bucket 2 fills up and splits, buckets 3, 4, 5, and 6 will be renumbered 4, 5, 6, and 7, respectively, and their entries in the address table will have to be shifted downward one slot to reflect this renumbering, and to make room for the new bucket address. We will come back to this point later.

Fig. 9. (a) Linear representation for the tree of Fig. 6(b). (b) The accessory address table.

In the address table of Fig. 9(b), slots are also reserved for dummy buckets. An alternative representation that does not need entries in the address table for dummies is as follows. Associated with each tree is a bit map with as many bits as the tree has leaves. For each leaf (bucket) the corresponding bit is set to 0 if the bucket is empty, otherwise the bit is set to 1.

When a tree search terminates, the bucket number of the bucket found (how to get this number will be explained in Section V-D) is used to index into the bit map to fetch the corresponding bit. If that bit is a 1, the bucket found exists; if it is a 0, the bucket is a dummy. For example, the bit map for Fig. 6(b) is 110011. By scanning the bit map up to and including the bucket found, one can determine how many nondummy buckets exist prior to the one found. This number that will be called the *index number*, can be used as an index into a table that now contains only for each nondummy bucket a slot with its disk address. Fig. 10 shows the linear representation, the bit map and the address table for the tree of Fig. 6(b). In this tree the buckets 1-6 yield the index numbers: 1, 2, 2, 2, 3, 4. Always the first nonempty bucket has index number 1, the second nonempty bucket has index number 2, and so on. Each dummy has the same index number as the rightmost nonempty bucket to its left.

C. Storage Requirements

It is straightforward to calculate the number of bits required in the index per bucket in the file. If a tree has N buckets (including dummies), it will have $2N - 1$ nodes total, and thus $2N - 1$ bits are needed for its linear representation. Thus, each leaf (bucket) requires approximately 2 bits.

If the bit map is not used, then every bucket, dummy or not, has a slot in the address table. Slots corresponding to dummies will contain a string of 0's and slots corresponding to nonempty buckets contain its disk address ($\neq 0$). If we assume that a bucket address needs A bits for its representation, then the number of bits required per bucket is $A + 2$. If a fraction d of all the buckets are dummies, the number of bits per nondummy bucket is $(A + 2)/(1 - d)$.

If the bit map is used, the number of bits per leaf is increased from 2 to 3, but only nondummy buckets need a slot in the table for their disk addresses, so the total number of bits per nondummy bucket is $A + 3/(1 - d)$.

tree: 00011100111

bit map: 110011

table:

1	bucket address for 1
2	bucket address for 2
3	bucket address for 5
4	bucket address for 6
	⋮

Fig. 10. Representation for the tree of Fig. 6(b) when using a bit map.

Clearly, the bit map scheme is to be preferred whenever:

$$A + 3/(1 - d) < (A + 2)/(1 - d)$$

which occurs whenever $d > 1/A$. For typical values of $A = 16$ or 32, if more than 6 or 3 percent of the buckets are dummies, which is nearly always the case, the bit map method is better.

By way of example, suppose that the bit map method is used for an S-tree. As can be seen from Fig. 7, using a split interval of 30 percent of the bucket capacity typically leads to about 60 percent dummies, while the load factor will be the usual 70 percent. Thus, if a bucket address costs 2 bytes (i.e., for files up to about 65 000 buckets), the total cost per bucket would be about 3 bytes.

D. Search Algorithms

Now we will present in detail the BDS- and S-tree search algorithms operating on the above developed, compact representation using the bit map. The reader can easily derive from our description the algorithms for the variant without the bit map.

It may already be clear how a search proceeds once the bucket number has been found. The bucket number is used as an index in the bit map. If the bit map has a 0 for the bucket, it is a dummy; if it has a 1, it is a useful bucket.

For a BDS-tree, a dummy means the key is not present and the algorithm terminates. If the bit map entry is a 1 for a BDS-tree, the program must count the number of 1 bits in the bit map from the first bit up to and including the bit corresponding to the bucket found to determine the index number of that bucket. This index number indicates which slot in the table contains the required disk address.

For S-trees, the same procedure is followed as for BDS-trees, except that the count is applied no matter what the bit map value is since finding a dummy when searching an S-tree does not mean that the key is absent. One must look in the first nonempty bucket to the left of this dummy to find out. Since a dummy has the same index number as this first nonempty bucket to its left, always counting 1's up to its own bit in the bit map is justified.

It is now time to describe in detail how the bucket number can be found. This part of the search process has two inputs: the key to be found, and the linear representation of the tree. Two markers are used, one to mark the current position in the tree as it is traversed, and one to mark the current position in the key as it is scanned. Initially, the

tree marker is at the root of the binary tree and the key marker is at the first bit of the key.

Each time the bit pointed to by the tree marker is a 0 (i.e., an internal node), the bit pointed to by the key marker is used to decide whether to advance the tree marker to the left subtree or to the right subtree. If this bit of the key is a 0 the tree marker must go to the left, and if it is a 1, to the right subtree.

Advancing the tree marker to the left subtree can simply be accomplished by advancing it to one position further, since in the linear representation a left subtree is represented directly following the 0 bit of its parent. To advance the tree marker to the right subtree, the left subtree must be skipped. The fact that any subtree contains one more 1 than 0's, can be used to find the end of a subtree easily.

The above described search goes on until the bit pointed to by the tree marker finally is a 1 (i.e., a leaf). This 1 bit corresponds to the bucket found (which is possibly a dummy). Instead of afterward counting all 1 bits left to it in the linear representation to determine its bucket number, one can simply keep track of the number of ones passed-by while skipping subtrees.

Fig. 11 shows Pascal functions illustrating the algorithms for the representation using a bit map. The search algorithm spends its time mainly on skipping subtrees and on converting a bucket number to an index number. Since both these operations are very simple, it should be easy to build special hardware for it (or to microprogram these operations). Without special hardware, the searching process can still be made relatively fast (in software) by using two 256-entry tables. For example, computing the index number from a bucket number using the bit map can be done much faster by using a table giving the number of 1's for each possible bit configuration in a byte. Similarly, two other tables can help to make the skipping process work byte-wise instead of bit-wise for all but a few bits of each subtree. One of these tables gives the difference between the number of ones and zeros within each byte. For example, the byte 00101100 has a difference of -2. (Since this table is equally good for the computation of the index number, a separate table giving the number of ones for each possible byte is not required.) The other table contains the maximum difference between ones and zeros found when traversing the byte bit-by-bit from left to right. For example, the byte 01011100 has as maximum 2. This second table is used to check whether the last bit of the subtree is present in the current byte. If the maximum difference achievable within the current byte is greater than or equal to the difference to be searched for, then the end of the subtree is in this byte. Otherwise it is not, and the difference to be searched for must be adapted by subtracting the total difference in this byte, as given by the first table, before going on with the next byte. Without the check described above, the algorithm would skip too much if the last bit of the subtree is followed by more zeros than ones within the same byte.

The reason to keep the linear representation, the bit map

```

const dummyaddress = 0;
maxkey   = ....; { Pascal, not the algorithm, requires maximum sizes }
maxtree  = ....;
maxmap   = ....;
maxtable = ....;

type diskaddress = integer;
table  = array [1..maxtable] of diskaddress;
map    = array [1..maxmap]   of (dummy, nonempty);
treetype = array [1..maxtree] of (split, leaf);
keytype  = array [1..maxkey]  of 0..1;

function skipandcount (var treeptr : integer; tree : treetype) : integer;
{ skips a subtree and returns the number of leaves in it }
var diff, oldptr : integer;
begin
  diff := 0;
  oldptr := treeptr;
  repeat
    treeptr := treeptr + 1;
    if tree [treeptr] = leaf
    then diff := diff + 1
    else diff := diff - 1;
  until diff = 1;
  { the number of leaves can be computed from the size of the subtree }
  skipandcount := (treeptr - oldptr + 1) div 2;
end;

function findbucketnr (key : keytype; tree : treetype) : integer;
var keyptr, treeptr, bucketnr : integer;
begin
  bucketnr := 1;
  keyptr := 1;
  treeptr := 1;
  while tree [treeptr] = split do
    begin
      if key [keyptr] = 1
      then bucketnr := bucketnr + skipandcount (treeptr, tree);
      keyptr := keyptr + 1;
      treeptr := treeptr + 1;
    end;
  findbucketnr := bucketnr;
end;

function findindexnr (bucketnr : integer; bitmap : map) : integer;
var i, indexnr : integer;
begin
  indexnr := 0;
  for i := 1 to bucketnr do
    if bitmap [i] = nonempty then indexnr := indexnr + 1;
  findindexnr := indexnr;
end;

function findaddress (bdstree : boolean;
                     key       : keytype;
                     tree      : treetype;
                     bitmap    : map;
                     addresstable : table) : diskaddress;
var bucketnr : integer;
begin
  bucketnr := findbucketnr (key, tree);
  if bdstree and (bitmap [bucketnr] = dummy)
  then findaddress := dummyaddress
  else findaddress := addresstable [findindexnr (bucketnr, bitmap)];
end;

```

Fig. 11. Pascal version of the algorithms.

and the address table separated, was just to be able to use the relation between the number of ones and zeros in a subtree to skip it, and to use the improvements presented above.

E. Linearity of the Algorithms

From the description above it follows that the computational cost of the search algorithm is linear in the size of the tree. During a search the algorithm reads from the linear representation all bits up to and including the 1 bit representing the bucket (dummy or not) finally found. This means that on the average about half the bits of the linear representation will be read. Clearly, also half the bit map will be scanned on the average.

A separator will have to be added (deleted) only when a bucket overflows (underflows). A little thought reveals that the algorithms for adding or deleting separators are linear too. For example, in case of adding a new separator, it usually will be necessary to replace one leaf by a

small, new subtree. To make room for this subtree, on the average about half of the bits of the linear representation need to be shifted to the right. A similar shift is required in the bit map to make room for the dummy-or-not bits corresponding to the leaves of the new subtree. Finally, on the average half the table must be shifted one place to make room for the address of the newly created nonempty bucket.

Although the algorithms presented can be made to run reasonably fast, they are all linear, and thus our compact representation will be useful only for representing binary trees of limited size. As will be explained in the next section, large indexes will have to be divided into parts of restricted size anyway. Therefore, the linearity of the algorithms usually does not cause serious problems.

VI. COMPARISON TO TRIE HASHING AND B-TREES

What we did until now was in fact nothing more than developing a compact representation of search information (i.e., separators and pointers). We showed that separators can be represented in a binary tree, that binary trees can be represented in a compact way, and that rather fast, but linear algorithms exist that operate on this compact representation of search information.

A. Comparison to Trie Hashing

The variant of Trie Hashing using binary digits [12] is in many respects equal to BDS-trees, although they were developed independently. One difference is that Trie Hashing uses a so-called "digit number" in each node. As a result, for files up to 32K buckets and keys of at most 256 bits, Trie Hashing uses 5 bytes per node, while for such files BDS-trees would need in the *standard representation* only 4 bytes per node. Besides, in our case keys may also be longer than 256 bits, since BDS-trees impose no restriction on the length of the keys.

However, this is still not compact enough. Our experiments have convinced us that compactly implemented B^+ -trees are better than Trie Hashing, BDS-trees, and S-trees using the standard representation. Happily enough, we have shown how a much more compact *linear representation* of our trees can be implemented efficiently. A sequential representation is described in [12] for Trie Hashing too. But, that representation seems to be less compact than ours, and, more importantly, it is doubtful whether it is possible to make algorithms operating on it that are as simple and efficient as ours.

Also, our representation has the advantage that the address table is kept separate. For example, Trie Hashing simply has to fail if the file is so large that the search information cannot be kept in main memory. However, in our case one could put the address table on disk at the cost of an extra disk reference. Note that the address table then has become somewhat similar to the directory of Extendible Hashing [7]. (Still, in our opinion, it is better to page and structure large indices in the multilevel organization of a B^+ -tree, as described below.)

It was shown already that the storage utilization offered

by BDS-trees (and thus also offered by Trie Hashing with binary digits) may be hardly acceptable. Therefore, for most applications S-trees seem to be superior to both BDS-trees and Trie Hashing (see also Section IV-D).

B. Comparison to B-Trees

Although the comparison in [12] between Trie Hashing and B-trees is in favor of Trie Hashing, it does not suffice to have shown that S-trees compare favorably to Trie Hashing. There are several reasons for making a separate comparison between S-trees and B-trees.

First, the comparison in [12] was made assuming that the percentage of nil pointers will be negligible. However, nil pointers obviously have the same bad influence on the cost per effective node in Trie Hashing, as dummies have with BDS- and S-trees, and our experience is that the percentage of dummies (nil pointers) may be significant.

Furthermore, the comparison in [12] was made assuming that with Trie Hashing the whole index (trie) was in core, while with B-trees only the root page of the index was assumed to be there. A more fair comparison can be made in two ways. Either both methods may use the same amount of main memory, and the comparison shows which method needs less disk accesses than the other, or it is assumed that both methods have their index entirely in core, and the comparison shows which method needs less space for its index.

Still, even such a comparison would not be totally fair, since B-trees have a nice property not (yet) present with Trie Hashing and S-trees. Namely, with B-trees it is easy to trade main memory for access performance by having few or many of the index pages resident on disk. Below we will show that an index based on S-trees can be paged into the same multilevel organization as a B^+ -tree, and that the resulting method will give better performance than a Prefix B^+ -tree [2].

1) *Using the B^+ -Tree Index Organization:* Although an S-tree can be used for representing an index very compactly, it still may happen that a large index does not fit a certain prescribed amount of main memory. Therefore, it should be possible to have (part of) an index on disk.

Since the index organization of B^+ -trees has proven to be worthwhile, an easy way to get the ability to handle also very large indexes is to use the same paged, multilevel index organization for S-trees. (A somewhat different, but perhaps better way of structuring the index pages in a multilevel organization is described in [5].) Note that this proposal amounts to using S-trees for the representation of the (separator, pointer)-pairs within each index page and structuring the index pages in the organization of a B^+ -tree. So, the only difference with other compact variants of a B^+ -tree, such as the Simple Prefix B^+ -tree (SPB-tree) and the Prefix B^+ -tree (PB-tree) [2], then lies in the way the (separator, pointer)-pairs are represented.

2) *Compactness of S-Trees:* Some experiments were run to see how compact the representation of separators in an S-tree really is. In the comparison we assumed the

separators in the SPB- and PB-tree to be variable length bit strings represented by a one byte length field and the bit string. The length of a prefix was also expressed in bits, not bytes. Thus the implementation we assumed for the SPB- and PB-trees was squeezed as much as possible. (Using bit stuffing to make it possible to denote the end of the variable length bit string by a special bit pattern, requires about the same number of bits.) Still, it was found that representing separators in an S-tree needs roughly half as many bits as with PB-trees. As expected, the representation used with PB-trees appeared to be only slightly more compact than that of SPB-trees.

The data used were the list of 50 000 words mentioned before and a list of 200 000 names which we got from the Dutch PTT. Both experiments were run once with a 5-bit-per-character encoding and once with the characters in an 8 bit ASCII encoding. For both encodings S-trees appeared to be almost two times as compact as PB-trees.

In Section V-C we mentioned already that the representation of a separator in an S-tree typically may cost one byte. If 3 bytes are used for bucket addresses (i.e., for files up to 16 million buckets), the total cost per (separator, pointer)-pair is typically 4 bytes. Assuming that the size of a page is 1K byte, this means that, when using S-trees, index pages may have a maximum fan-out of about 250, and that the average fan-out can be roughly 175. (Remember that the average load factor of index pages will be about 70 percent too.) Thus, if only the root index page is kept in main memory, records of files up to roughly 44 Mbytes can be accessed in two disk accesses. In general, the same access performance is accomplished if all index levels except the lowest one are kept in core. Thus, each record in a file of $N * 30$ Mbytes can be accessed in two disk accesses if $N + 1$ index pages are kept in main memory. These figures would even be better if a pointer compression scheme would be used. In [5] it is shown how, in case of bucket addresses of 4 bytes, the actual cost per bucket address easily can be reduced to 2 bytes.

VII. SUMMARY AND CONCLUSION

We have shown how a highly compact representation of binary trees can be used for representing search information. Two access methods using this representation were developed, BDS-trees and S-trees. With both methods the index is represented so compactly that it usually will fit in main memory, which means that records usually can be retrieved in one disk access. Each of these two methods may have its own class of applications where it is better than the other. A point in favor of S-trees is, that its behavior can be influenced by varying the size of the split interval used. When a maximum split interval is chosen, S-trees behave almost as BDS-trees. Furthermore, we have shown how the fan-out of a B-tree can be improved by using S-trees for the representation of the (separator, pointer)-pairs.

In this paper the emphasis was on building indexes. However, the techniques presented can also be used, for

example, for storing pictures more efficiently than with current methods [5], [6], [8].

ACKNOWLEDGMENT

We would like to thank W. Litwin for his comments on a previous version [4], W. Koot for performing some simulations, the Dutch PTT for some test data, P. Scheuermann for his stimulating interest, and the referees for their valuable comments.

REFERENCES

- [1] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indexes," *Acta Inform.*, vol. 1, no. 3, pp. 173-189, 1972.
- [2] R. Bayer and K. Unterauer, "Prefix B-trees," *ACM Trans. Database Syst.*, vol. 2, no. 1, pp. 11-26, Mar. 1977.
- [3] D. Comer, "The ubiquitous B-tree," *Comput. Surveys*, vol. 11, no. 2, pp. 121-137, June 1979.
- [4] W. de Jonge, A. S. Tanenbaum, and R. P. van de Riet, "A fast, tree-based access method for dynamic files," *Vrije Univ., Inform. Rep. IR-70*, July 1981.
- [5] —, "A fast, tree-based access method," *Vrije Univ., Inform. Rep. IR-85*, July 1983.
- [6] W. de Jonge, Short note sent to P. Scheuermann, June 1984, 7 pp.
- [7] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, "Extendible hashing—A fast access method for dynamic files," *ACM Trans. Database Syst.*, vol. 4, no. 3, pp. 315-344, Sept. 1979.
- [8] I. Gargantini, "An effective way to represent quadrees," *Commun. ACM*, vol. 25, no. 12, pp. 905-910, Dec. 1982.
- [9] D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*. Reading, MA: Addison-Wesley.
- [10] P. Larson, "Linear hashing with partial expansions," in *Proc. 6th Int. Conf. Very Large Data Bases*, Montreal, Canada, Oct. 1980, pp. 224-232.
- [11] W. Litwin, "Linear hashing—A new tool for file and table access," in *Proc. 6th Int. Conf. Very Large Data Bases*, Montreal, Canada, October 1980, pp. 212-223.
- [12] —, "Trie Hashing," in *Proc. ACM-SIGMOD Int. Conf. Management of Data*, Ann Arbor, MI, 1981, pp. 19-29.
- [13] A. Yao, "On random 2-3 trees," *Acta Inform.*, vol. 9, no. 2, pp. 159-170, 1978.

Wiebren de Jonge was born in 1954 in Amsterdam, The Netherlands. He received the Bachelor's, Master's, and Ph.D. degrees from Vrije Universiteit in Amsterdam. His dissertation contains articles about statistical databases, keeping secrets, and digital signatures.

Besides being (co)author of several published papers concerning security and privacy, he is also (co)author of two published papers in the database field. Currently, he is mainly interested in this last field and has a tenure position at the Department

of Computer Science of Vrije Universiteit.

Dr. de Jonge is a member of the Association for Computing Machinery and NGI.

Andrew S. Tanenbaum (M'75) was born in New York City and raised in White Plains, NY. He received the Bachelor's degree from Massachusetts Institute of Technology, Cambridge, and the Ph.D. degree from the University of California at Berkeley.

He is currently on the faculty at the Vrije Universiteit in Amsterdam, The Netherlands, where he teaches and does research in the areas of computer architecture, operating systems, networks, and distributed systems. He is currently involved

in two major research projects, on advanced compiler technology and on distributed operating systems. The compiler project has led to the development of the Amsterdam Compiler Kit, which is a tool kit for building compilers for (at the moment) half a dozen languages and 10 different target machines. The kit is currently being used at more than 100 universities and companies around the world. The operating system research has led to the development of the Amoeba distributed operating system, which will soon be running at five sites in three countries, interconnected by a wide-area network. He is the author of three books (on computer organization, computer networks, and operating systems), and more than 40 published papers on a variety of subjects. He has also lectured in a dozen countries, and has worked for IBM and AT&T Bell Laboratories.

Dr. Tanenbaum is a member of the Association for Computing Machinery, the IEEE Computer Society, and Sigma Xi.

Reind P. van de Riet was born in Groningen, The Netherlands, on April 22, 1939. He received the Bachelor's degree in mathematics from Vrije Universiteit, Amsterdam, The Netherlands, in 1962, and the Ph.D. degree from the Universiteit van Amsterdam in 1968.

Since 1970 he has been Professor of Computer Science in the Department of Mathematics and Computer Science, and Chairman of the Computer Science Group, at Vrije Universiteit. He is the author of about 50 scientific papers. His current research interests include information systems, knowledge-based systems, and fifth generation computer systems.

Dr. van de Riet is Chairman of the Dutch Foundation for Research in Computer Science (SION), European regional Editor of *Data and Knowledge Engineering*, and Consulting Editor of *Future Generation Computer Systems*. He was Chairman of the Conference on Fifth Generation and Super Computers, Rotterdam, The Netherlands, in December 1984.