stichting
mathematisch
centrum

MC

AFDELING INFORMATICA
(DEPARTMENT OF COMPUTER SCIENCE)

IW 219/83 FEBRUARI

T. KRIJNEN & L.G.L.T. MEERTENS

MAKING B-TREES WORK FOR $\mathcal{B}$

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, Kruislaan 413, Amsterdam, The Netherlands.

The Mathematical Centre, founded 11th February 1946, is a non-profit institution for the promotion of pure and applied mathematics and computer science. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1982 CR. Categories: E.2, D.3.3, D.3.4

MAKING B-TREES WORK FOR *B*★

by

Timo Krijnen
Lambert Meertens†

ABSTRACT

The programming language *B* has a semantically very powerful type system. No attention was given to implementation efficiency during language design. This paper shows how to implement the operations of the type system efficiently by the use of B-trees.

KEY WORDS & PHRASES: data storage representations, B-trees, programming languages, data types and structures, *B*.

★ This paper is not for review; it is intended for publication elsewhere.

† Address until September 1983: Courant Institute of Mathematical Sciences, New York University, New York.

0. Introduction

B is the "working title" for a simple but powerful programming language, designed for use in personal computing. The aim in designing *B* was to offer an alternative to BASIC, supporting the use of present-day programming methodology. During the design of *B*, run-time efficiency issues were completely ignored. Instead, undivided attention was given to the ease of learning and using the language. This somewhat unusual attitude was motivated by the experience that concern for efficiency during language design is all too easily premature. Efficiency depends on the implementation chosen. The constraints of a particular implementation approach force one to complicate the language design. These complications may effectively prohibit the adoption of a better implementation than originally considered. Also, with increasing processor speed, the historically explicable emphasis on efficiency serves mainly to worsen the imbalance between hardware and software cost. A substantial part of the effort in using computer systems is spent on waging a combat against limitations introduced for the sake of efficiency.

One of the results of this attitude is that *B* is equipped with a conceptually simple, but semantically very powerful, system of types and associated operations. This is described in Section 1. Now the design of *B* is finished [MEE], we are addressing the question of how to implement the operations of the type system efficiently. The lack of concern for efficiency during the design does not mean that we feel that efficiency is unimportant. On the contrary: the intention has been to revise the language design if it is found that minor modifications would allow major efficiency gains.

As far as the type system is concerned, no need for modifications has come to light. By combining various techniques described in the literature in a creative way, and adding a few new techniques, it is possible to perform each primitive operation on texts, lists and tables in time O(log *N*), where *N* is the "semantic" size of the largest value featuring in the operation as operand or result. Not only does this give a good time efficiency, but it also reduces the use of storage. The main techniques are the use of B-trees to represent *B* values (no pun intended), and the extensive sharing of (sub)values.

1. THE TYPE SYSTEM OF *B*

B has two basic types: "numbers" and "texts", and three generic types: "compounds", "lists" and "tables". This paper is not concerned with the implementation of numbers (which is not completely trivial) or of compounds (which present no implementation problems).

A text is a sequence of characters. There is no restriction on the length of texts, and a text variable may receive in its lifetime texts of varying lengths. The operation `^` joins two texts: `'ab'^'cd' = 'abcd'`. Repetition is achieved with the operation `^^`: `'abc'^^3 = 'abcabcabc'`. A subtext may be selected with the two "trimming" operations `@` and `|`. For example, `'protracted'@4 = 'tracted'`, and `'tracted'|5 = 'tract'` (and so `'protracted'@4|5 = 'tract'`). A subtext selection on a variable gives an assignable variable. A subtext assignment such as `'long'` to `v@4|5` has the same effect as the ordinary assignment of `v|3^'long'^v@9` to `v`. So this can be handled in terms of other primitives. The function `#` gives the length of a text, and `n th'of t` gives the `n`'th character of `t`.

A list is a multiset (bag) of other values, all of the same type. The operations on a list are insertion and deletion of an entry. The presence of an entry may be tested. The function `min` returns the smallest entry in a list. If an additional argument is given to `min`, the least value exceeding that argument is returned. The function `max` is similarly defined. It is possible, with a `FOR` command, to iterate through all entries of a list, in order from low to high. The functions `#` and `th'of` are defined as for texts, where `th'of` counts the entries from low to high. (On all *B* values

of a given type, a natural ordering is defined.) Finally, the "range" `{p..q}` gives a list whose entries form the range of consecutive integers `{p; p+1; ... ; q}`.

A table is a map (associative array) from "keys" to "associates". A value may be stored and retrieved under a given key, and an entry (key-associate pair) may be deleted. The function `keys` returns the list of all current keys of a table. The functions `#` and `th'of` are also defined on tables; `th'of` returns the *associate* of the `n`'th key, counting from low to high keys. Likewise, a `FOR` command iterates through the associates in the order of the keys.

An amendment must be made with respect to the claim on the O(log N) time complexity of the primitive operations. The functions `min` and `max` and the test `in` are also defined on characters in texts and associates in tables. Although an O(log N) implementation is possible (for tables, e.g., by also maintaining the inverted table), it is expected that the overall run-time efficiency would thereby decrease. We propose, therefore, not to attempt to choose a data structure allowing an o(N) implementation.

The design of the type system was mainly the work of R.B.K. Dewar and the second author. It has a striking resemblance to the type system of SETL. And yet, it was not found by adapting SETL types to the design requirements of *B*, but by writing (in SETL!) and running a program that considered a large power set of candidate type systems and weeded out less successful ones, weighing "ease of learning" and "ease of use" on the basis of prepared input data. If the resemblance is not a coincidence, this can only mean that the criterion used in the selection program codified the visionary insight of the architect of SETL, J.T. Schwartz.

2. B-TREES

B-trees were invented by Bayer and McCreight [BAY] as an efficient data structure for maintaining large indexed files. To define B-trees, we first introduce as auxiliary notion "A-trees".

An "A-tree" for a sequence S of "entries" is empty if S is empty. Otherwise it—in a concrete representation, its root node—has the format

$$(p_1)\ e_1\ (p_2)\ \ldots\ (p_{n-1})\ e_{n-1}\ (p_n),$$

where

- the "children" p_i are again A-trees—concretely represented by pointers to nodes (or nil);
- the number of children n is at least 2;
- the e_i are entries, and the "inorder" traversal of the tree visits exactly the entries of S in the same order.

For example, (() a ()) b () c (() d (() e ()) f ()) is an A-tree for the sequence abcdef.

The "height" of an empty A-tree is 0, and that of a non-empty A-tree is one more than the largest height of its children. The height of the A-tree given above for abcdef is three: the heights of its subtrees for a, def and e are one, two and one, respectively.

Let a and b be two integers, such that $a \geqslant 2$ and $b \geqslant 2a-1$. An A-tree is in the class of "(a,b)-trees" if it is empty or

- $2 \leqslant n \leqslant b$, and
- its children are (a,b)-trees, *all of the same height*, and if they are not empty, *their* number of children n satisfies, for each of them, $a \leqslant n \leqslant b$.

So all nodes have $a \leqslant n \leqslant b$, except for the root, which may have fewer children if $a > 2$.

The classes of (a,b)-trees form together the B-trees. Because of the bounds on the number of children for each node and the equal distance from the root of all bottom nodes, a B-tree is bal-

anced. (A bottom node is a node, all of whose children are empty.) The height is at most $1 + \log((N+1)/2) / \log a = O(\log N)$, where N is the length of S. If the entries in the sequence S are sorted, then, for each non-empty subtree, all entries in a child p_i are at least e_{i-1} if $i \geqslant 2$ and at most e_i if $i \leqslant n-1$. This makes it possible to search for an entry, starting from the root, such that an entry can be found (or its absence detected) by visiting $O(\log N)$ nodes. The most valuable property of B-trees is that they allow $O(\log N)$ algorithms for insertion and deletion in S, maintaining "(a,b)-treeness". These algorithms will now be given, because further algorithms refer to (parts) of them. The most illuminating exposition, to our taste, is that in which the (a,b)-tree property may temporarily be lost and then is restored. This loss occurs for at most one node at a time, through "underflow" (n one too small) or "overflow" (n one too large). Restoration can be achieved by applying either a "*rotation*" or a "*merger*" on underflow, or by "*splitting*" on overflow.

Restoring on underflow:

- If the root underflows, take its only child as the new root.
- If another node underflows, it has a parent, and since only one node can underflow at a time, it must have at least one sibling. Assume, without loss of generality, that it has a right sibling. The parent has then the form ... $(t)\ x\ (u)$ ... , where t is the subtree whose top node underflows. Writing out one more level, we have

$$\ldots((p_1)e_1(p_2)\ldots(p_{m-1})e_{m-1}(p_m))\ \ x\ \ ((q_1)f_1(q_2)\ldots(q_{n-1})f_{n-1}(q_n))\ldots,$$

where $m = a-1$ and $a \leqslant n \leqslant b$.

- If $n \geqslant a+1$, the right sibling can spare a child. So a "*rotation*" can be applied, replacing the above by

$$\ldots((p_1)e_1(p_2)\ldots(p_{m-1})e_{m-1}(p_m)x(q_1))\ \ f_1\ \ ((q_2)\ldots(q_{n-1})f_{n-1}(q_n))\ldots$$

by transferring x from the parent and q_1 from the right sibling to the underflowing node, and f_1 from the right sibling to the parent. This restores (a,b)-treeness.

- If $n = a$, then $m+n = 2a-1 \leqslant b$, so the underflowing node and its sibling fit into one node. The "*merger*" of both nodes changes the above into:

$$\ldots((p_1)e_1(p_2)\ldots(p_{m-1})e_{m-1}(p_m)x(q_1)f_1(q_2)\ldots(q_{n-1})f_{n-1}(q_n))\ldots.$$

Here the parent has surrendered x for the merger of its children. It has now one child less, so the underflow condition may have been pushed one level up and necessitate restoring there, and so on, till the root.

Restoring on overflow:

If a node z overflows, it has the form

$$((p_1)e_1(p_2)\ldots(p_{n-1})e_{n-1}(p_n),$$

where $n = b+1 \geqslant 2a$. "*Split*" the node into $(s)\ x\ (t)$, where s and t are two new nodes and x is an entry, thus:

$$((p_1)e_1(p_2)\ldots(p_{a-1})e_{a-1}(p_a))\ \ e_a\ \ ((p_{a+1})e_{a+1}(p_{a+2})\ldots(p_{n-1})e_{n-1}(p_n)).$$

- If z has a parent, replace (z) in the parent by $(s)\ x\ (t)$. This increases the number of children in the parent by one, so it may now overflow in turn, and so on.
- If z is the root, create a new root consisting of $((s)\ x\ (t))$.

A refinement on overflow is to delay splitting by having excess entries transferred to a sibling by rotation if it still has room left. Only when the two siblings are both full, will they be split into three nodes. This improves storage utilization and may break a chain of overflows. It can be extended to

reorganizations involving more than two siblings. Similar refinements are possible on underflow, but these tend to degrade storage utilization.

The algorithms for insertion and deletion are now very simple:

- Locate the place where the entry is to be inserted or deleted in the tree.
  For insertion, this place is always at a bottom node.
  For deletion, if the entry is not in a bottom node, we can replace it by its predecessor (or successor); the place where an entry is to be deleted is then the original place of the predecessor, which is always at a bottom node.
- Insert or delete it.
- Restore as necessary.

Although the search for the entry needs time O(log N), the cost of rearranging nodes has a much higher weight. Luckily, the O(log N) bound on the number of node rearrangements is only a worst case bound. The expected number of rearrangements for a random insertion or deletion is a constant, depending on a and b but not on N.

Because of the confusion that exists in the terminology with regard to B-trees, we give the relationship between ours and that used by others. The notion of "(a,b)-tree" is taken from [HUD]. However, their (a,b)-trees are properly speaking not B-trees, as introduced by Bayer and McCreight. They are B^+-trees [COM], a variant in which all entries are stored in bottom nodes. The upper "internal" nodes of the tree serve only as an index. The same is true for the "2-3 trees" of Hopcroft as described in [AHU], where the index-part of the tree slightly differs from the organization in [HUD]. In this paper we use the term (a,b)-tree exclusively for proper B-trees, as described above.

The "class $\tau(k,h)$ of B-trees" of [BAY] corresponds then to the class of $(k+1,2k+1)$-trees of height h. The "B-trees of order m" of [KNU] are the $(\lceil m/2 \rceil, m)$-trees. "Weak" [HUD] (or "hysterical" [MAI]) B-trees have $a < (b+1)/2$. B-trees as originally defined in [BAY] (and also in [COM]) are all "strict": $a = (b+1)/2$. The B-trees of [KNU] are strict if their order is odd, and weak if their order is even.

Weakness of B-trees is a strength: the slight additional freedom makes it possible, e.g., to create a B-tree by N insertions and deletions with only $O(N)$ node rearrangements, rather than $O(N \log N)$ ([HUD], [MAI]). We consider the choice of a and b in section 11.

3. REPRESENTING *B* VALUES BY B-TREES: fiRST APPROXIMATION

A text can be represented by a B-tree for the sequence of characters that it is. (Of course, the entries in the sequence will only be ordered, not sorted.) At first sight it appears strange to use a B-tree representation for an elementary type as that of texts. And yet, this representation makes it possible to realize the important operations in time O(log N), which is not true of simpler representations (e.g., doubly linked lists).

For a list, the ordering on its entries can be used to keep the list entries sorted. A list is then represented by a B-tree for the sorted sequence. This also underlies the B-tree representation for sorted lists of [BRO] and [HUD].

For a table, an entry is a key-associate pair, and the entries can be sorted on the key field. This corresponds to the originally foreseen use of B-trees: although tables are "internal values" and files "external", there is no semantic difference between a table and an indexed file. (So in a *B*-dedicated programming environment, no separate objects for "files" are needed: "ordinary" *B* values will serve the purpose.)

With this representation, many operations on *B* values can already be implemented so as to take time O(log N): insertion, deletion, presence testing and the functions `min` and `max` for lists,

and storing, retrieval and deletion under a key for tables. Iteration with a FOR command takes a constant average time for each step. For building texts from a sequence of N characters an $O(N)$ algorithm like the one in [ROS] can be used.

4. ACCOMMODATING RANGES

An expression of the form {p..q} gives rise to a list with q-p+1 entries. Building a conventional B-tree for this list requires a time linear in the size of the list. One might use a special representation for such a list, but then a subsequent insertion or deletion might force the building of a full-fledged B-tree. To obtain a sublinear time also under such conditions, we introduce a special node type for lists of contiguous elements. The header of each node contains a marker that assumes one of the values "normal" and "range". If the marker is "normal", the node is a B-tree node as described above. If it is "range", it contains two values p and q, and it represents the entries p, p+1, ... q.

The algorithms for handling B-trees must now be adapted to cater for these special nodes. The adaptation is such that range nodes will always occur at the bottom level of the tree. When originally created, there is only one level, which is then also the bottom level. Also, the number of represented entries in a range node, in the sequel called its "size", is at all times at least a (compared to $a-1$ for normal nodes). If the number was originally less, a normal node can accommodate the entries instead; since the node is then the root, there are no underflow problems. Insertion and deletion proceed now as usual, until an entry has to be inserted or deleted in a range node.

On deletion in a range node z, test first if the size of z is at most b, in which case the node with the entry deleted is converted to a normal node and we are done. Otherwise, split z into a node s, an entry x and a node t, such that the entries represented by s are consecutive, and likewise for t (so x is adjacent to the hole caused by the deletion). Then s, x and t represent together at least $2a-1$ entries, so at most one of the two nodes can be short, meaning that its size is less than $a-1$. Convert the short node, if any, to normal format and rotate as on underflow, if necessary, thus transferring entries to the short node until its size is $a-1$. The parent node may now overflow, and this is remedied as described before.

On insertion of an entry x in a range node z, if the size of z is at most $b-2$, the node with the entry inserted is converted to a normal node and we are done again. Otherwise, split z into two range nodes s and t, such that the entries of s are at most, and those of t at least x. Now s, x and t represent together again at least $2a-1$ entries, so we may proceed as on deletion.

5. SHARED B-TREES

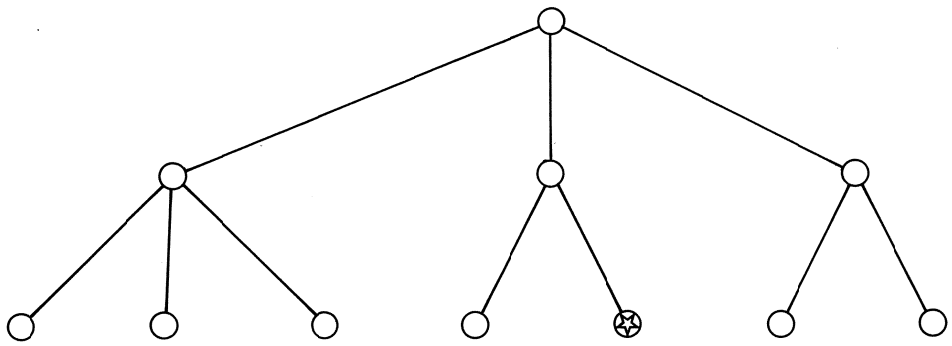
To make copying require time $O(\log N)$, and even $O(1)$, copies are made by copying pointers. Thus, a B-tree may be shared between "owners". For B, the scheme described in [HIB] can be used. The absence of circularity in B-values makes it feasible to use reference counts systematically, faithfully representing at all times the number of live owners of a shared object. This obviates the need for garbage collection. More importantly, B has a "value semantics", as opposed to an "object semantics", meaning that modifications to a shared object may not modify the value as perceived by the other owners. For example, after

```
PUT v IN v'
INSERT 0 IN v,
```

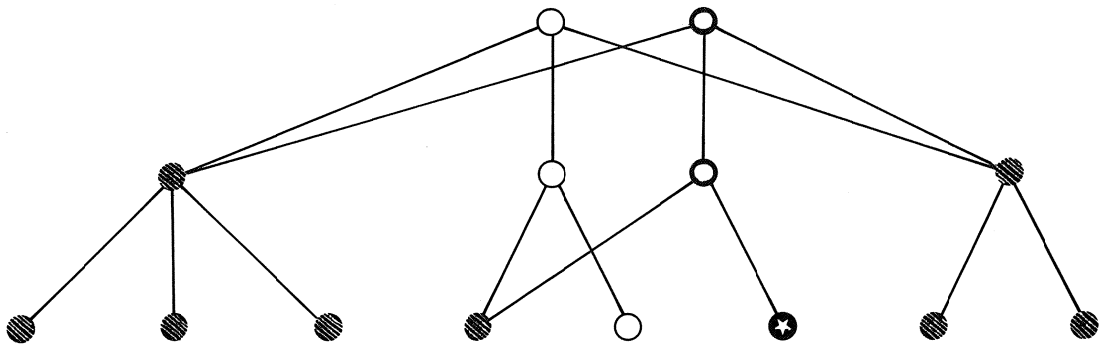
the values of v and v' are different. To obtain the correct semantics, the minimum run-time information that is needed is a "sharing" bit. That information is also supplied by a reference count.

The reference counts are not bound to the B-trees as a whole, but to each node. The latter obviously causes more storage overhead per node, but it makes up for this by allowing more sharing of storage, and is of essence in maintaining an O(log N) cost. For example, if information about sharing were not available for each node, the insertion in the above example would necessitate copying the full B-tree first, which takes more time and more storage. (Furthermore, the joining and trimming of texts could not be kept O(log N).) Now, physical copies have only to be made of the O(log N) nodes on the path from root to the position of modification.

The effect of this "desharing" along a path in a tree is pictured in fig. 1.



(a) the original shared tree,
the position of modification is marked



(b) the path to the position of modification is "deshared",
new nodes are thicker, shared nodes are shaded

Figure 1. Desharing a path in a shared tree.

In all algorithms it must be understood without explicit mention that all copies of B-(sub)trees are made by copying pointers and increasing reference counts, and that desharing takes place whenever dictated by the value semantics.

6. THE COST OF COMPARISONS

Until now, we have tacitly assumed that a comparison takes constant time. Indeed, in a typical *B* program most comparisons are between numbers or small texts. In the general case, however, the values to be compared may be arbitrarily complex themselves.

The comparison of two values, represented by pointers to B-trees, will start with the question of whether the two pointers are equal. In many cases this will immediately exhibit equality of complex values. But if two equal values of size N come into being through different operations, comparison may take up to a constant multiplied by N. It would seem that the O(log N) time requirement cannot be saved. We may, however, "amortize" the comparison cost by ascribing it to the (already incurred) construction cost of one of the two values. For this, we add a constant time to the cost of each of the elementary constructing operations. Once two values have been found equal, the representation of one may be replaced by that of the other, so that further comparisons are immediate and the added costs really only count for one future comparison.

If the values are not equal, this will, on the average, be found after comparing an initial part of small size, bounded by a constant. Moreover, the larger the initial parts that are still equal, the more the sharing of constituent equal parts will occur.

Note that the argument is partly probabilistic. It can be frustrated by writing a program that involves a great number of comparisons between different values with large equal initial parts. There are also other examples invalidating the argument. They are exemplified by

```
'abab'^^n = 'ab'^^(2*n)
```

Using the algorithms as proposed, the comparison will require time O(N). The above argument does not apply here, since the original construction cost was sublinear (see section 8). These counter-examples are so particular that it is pragmatically justifiable to ignore this. (Similar counter-examples do not exist for the only other sublinear constructor `{p..q}`, if the comparison algorithm is still linear in the number of nodes visited in the presence of range nodes, which is not hard to achieve.)

Although the actual comparison cost is, of course, still linear in N, the expected *total* cost of a sequence of operations, involving arbitrarily complex comparisons, is then the same as though the comparisons took logarithmic time

7. SIZE fiELDS.

We associate with each node of a B-tree a field containing the size (number of entries) of the dependent tree. Then the function # takes time O(1). This function is among the most heavily used in *B*. The function `th'of` can be implemented in time O(log N) now. The size fields are also necessary for the trimming algorithm in section 8.

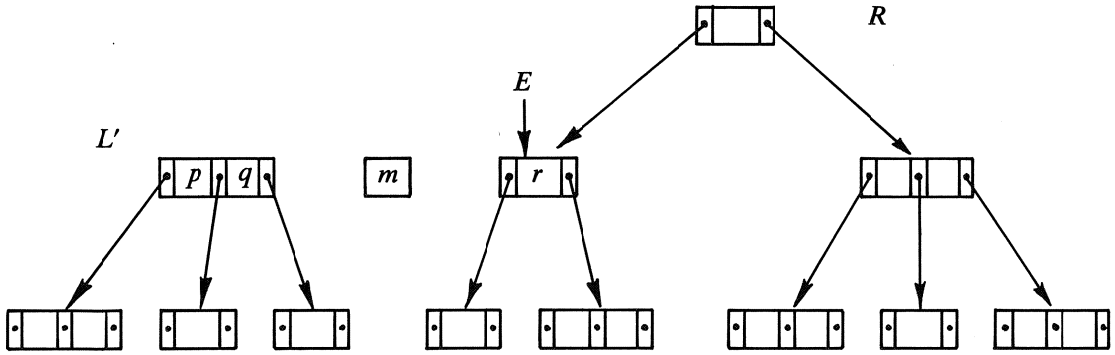
This obliges us, of course, to keep the size fields up to date through all B-tree operations, but that is a routine matter. (Implementing a `th'of` function is exercise 6 of section 6.2.4 of [KNU]. The solution given there requires more space.)

8. JOINING AND TRIMMING TEXTS

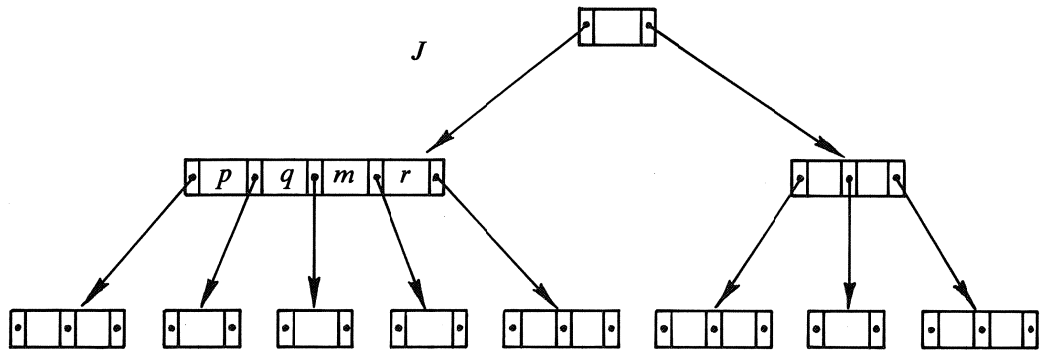
The most important operation on texts is joining (concatenation). In [AHU] an algorithm is given for joining (2,3) B^+-trees. It is easily extended to arbitrary B^+-trees. The extension to B-trees is less trivial. This is exercise 8 in section 6.2.4 of [KNU], where only a hint to a solution by C. H. Crane for AVL-trees is given. An algorithm for general B-trees will now be described. The problem is to construct, given two B-trees L and R for two sequences S and T, a new B-tree J for

the sequence S followed by T. If L is empty, we take $J = R$. Assume that L is not empty. Remove the rightmost entry m from L, using the O(log N) algorithm for deleting an entry. This gives us a new B-tree L'. We have to join now, from left to right, L', m and R.

If the height of L' is less than or equal to that of R, find the left-edge node E of R whose height (counting from the bottom) is the same as that of the root of L'. (A left-edge node is a node lying on the path from the root if only leftmost branches are followed.) The root of L', m and E are now made to form a new node taking the place of E. This is pictured in fig. 2.



(a) before



(b) after

Figure 2. Merging the root of L', entry m and node E.

The root of R then becomes the root of a tree whose entries are those of J. The B-tree invariant is now, in general, violated: the number of children of the new node may be as large as $2b$. Restore the invariant, if necessary, by splitting the node and proceeding as in the insertion algorithm.

If the height of L' is greater than that of R, this is the mirror image of the above situation.

As in the deletion and insertion algorithms, the O(log N) node rearrangements bound covers the worst case, but the expected (average) number of rearrangements is constant if L and R are not shared. As a practical remark, the algorithm could also start by deleting the leftmost element from R, and this is on the average faster if the size of R is less than that of L.

The operation ^^ for repeating a text n times can be implemented by mimicking the well-known O(log n) method for exponentiation by repeated multiplication. If this is done straightforwardly, we obtain an O($\log^2 N$) algorithm. Instead, delete first the rightmost entry r from the text to be repeated, and call the resulting tree Q_1. (If $n = 0$, or the text is empty, an empty tree is of course returned straightaway.) Form Q_2 by joining (as above) Q_1, r and again Q_1 together. In general, the entries of Q_k will form a k-fold repetition of the text, but with the last character omitted. Starting from the pair Q_1, Q_2, a sequence of pairs can be formed in O(log n) steps to yield Q_n, as follows. Expand n as a binary number $b_1b_2...b_m$, put $n_i = b_1b_2...b_i$ (so that $n_1 = 1$ and $n_m = n$), and form, for $k = n_2, n_3, \ldots, n_m$, the pair Q_k and Q_{k+1} from the previous pair, using the knowledge that Q_{2j} can be formed by joining Q_j, r and again Q_j together, and Q_{2j+1} by joining Q_j, r and Q_{j+1}. Finally, insert r in Q_n as rightmost element. (For example, if $n = 13 = 1101_2$, then from Q_1 & Q_2 the pair Q_3 & Q_4 is formed, from that Q_6 & Q_7, and finally Q_{13} & Q_{14}.)

The almost equal height of the trees Q_k and Q_{k+1} makes this algorithm efficient. The restoring of the B-tree invariant in each join takes then only time O(1). So the preliminary deletion, the pair forming, and the final insertion can all be done in time O(log N). Forming Q_{2j+1} from Q_{2j} and Q_1, makes the cost of restoring the invariant grow too much, giving the upper bound O($\log^2 N$). It should be remarked that the O(log N) time bound implies an O(log N) bound on the storage space used; thus an expression such as `'x'^^123456789` will use a modest amount of space.

The trimming of texts is the next operation that must be considered. Since the operations @ and | are each other's mirror image, we treat only the latter. The problem is, given a B-tree T for a sequence S, to construct a new B-tree C for the sequence consisting of the first p entries of S. In [AHU] an algorithm for (2, 3)-B$^+$-trees is given. The extension to B-trees is again not obvious. For *weak* B-trees we have found a relatively simple one-pass top-down algorithm in the spirit of [OTT]. The trick is to ensure that on descending the immediate parent root has at least $a + 1$ children, so that robbing it of one child cannot cause underflow.

If p equals 0 or the length of S, we can immediately return empty or T. Otherwise there exists a path in T from the root down, such that for each node on the path there is an i, $1 \leqslant i \leqslant n$, with the properties that at least one entry in (p_i) is among the first p entries of S, and for $i \neq n$ either e_i is exactly the p'th entry of S or e_i is not among the first p entries of S. The trimming algorithm travels down along this path in one pass.

To start the process, go down the path in T while it stays on the left edge. In this way we reach the highest node from which at least one entry goes to C. A "trimmed" copy of this node, upto and including p_i, becomes the root of C. Each time a node is encountered, a possibly "trimmed" copy R is made, again upto and including p_i. It is chained into the path growing from the root of C. Before descending further, it may be necessary to first rebalance the already existing part of the tree C, because R can have too few children. This can be done with the same means as in the previous algorithms: transfer, by rotation, some entries and children from the left sibling L of R to R, or merge L and R into one node. Because the latter case (merging) robs the parent of one child, it must have (if it is not the root) at least $a + 1$ children. So we have to take care that this is also true for R, which will assume the role of parent on the next step. (From this it can be seen that the left sibling L always exists.) The weakness of the tree makes it possible to do an extra rotation to achieve this in the boundary case.

After each trimming of a node the path down proceeds with the rightmost child p_i of R. The process ends when the number of entries in C reaches p, or when the node under consideration in T is not at the bottom level and e_i is exactly the p'th entry of T. In the latter case, after rebalancing the "trimmed" copy of the node, e_i must be inserted as the rightmost element of C, which can still be done in one downward pass.

For strict B-trees, this algorithm breaks down if L and R both have a children, since neither a rotation nor a merger can then be applied. However, this can be remedied by drawing the left sibling of L into the process of rotating or merging. This excludes the use of this algorithm for $(2,3)$-trees. It also poses problems at the root of other strict B-trees, but these can be solved.

9. GENERIC OPERATIONS AND THE FUNCTION keys

Many operations on texts, lists and tables are generic. This genericity propagates in user-defined units, as in:

```
YIELD last a:
    CHECK #a >= 1
    RETURN (#a) th'of a.
```

For texts the function `last` yields the last character, for lists it gives the last entry, and for tables the last associate is delivered.

The implementation of this genericity is facilitated by a uniform data structure for texts, lists and tables. The absence of run-time type distinctions saves time and code.

Each entry in a B-tree can be seen as a record-like data structure with two field-selecting operations: K and A (for "key" and "associate"). These can be represented by offsets. For a text, an entry is a single character; the K-operation is undefined (it will never be applied to an entry), and the A-operation selects the single field. For a list entry, the K- and A-operation both select its single field. For a table entry, the K-operation selects the key, and the A-operation the associate.

Texts, lists and tables can now be uniformly represented by a triple $<t,K,A>$, where t is a (pointer to) a B-tree as developed until now.

An important by-product of this representation is that the heavily used function `keys` for tables becomes almost free: `keys` $<t,K,A>$ is simply $<t,K,K>$. Without the triple representation, we would still have an $O(N)$ cost for this function.

For functions like `min` a run-time type distinction is desired for reasons of efficiency. This can be done once at the top: if $K = A$, the $O(\log N)$ algorithm for lists is invoked; otherwise $K \neq A$, and the $O(N)$ traversal of texts or tables is called for.

10. FINGERS

In many algorithms, handling texts, lists or tables, the entries in the tree are accessed (almost) sequentially. Under these conditions we can speed up the average search time considerably, without using high-level optimizations, if we remember the place of the last accessed entry in a "finger", and start the next search from there. This would also make the second accesses immediate in constructions like:

```
PUT t[n]+1 IN t[n]
```

and

```
IF e in t: REMOVE e FROM t
```

By using fingers in level-linked (a,b)-trees [BRO] and [HUD] reduce the average access time for an entry for such cases from $\Theta(\log N)$ to $\Theta(1)$. Where their (a,b)-trees differ from the proper B-trees (see section 2), the added child-to-parent and sibling-to-sibling links for their level-linked (a,b)-trees would be less effective if added to proper B-trees in speeding up searches in the vicinity of fingers. Also, it is envisaged that the size of most values in a typical B program will be small, restricting the height of the corresponding trees. For example, the average height of trees for values

with less than a thousand entries will be smaller than four if the mean number of children per node of the trees is at least ten. In that case we cannot expect much gain from the use of fingers. The extra cost for the other operations in maintaining the parent-to-child and sibling-to-sibling links in level-linked (a,b)-trees will probably overshadow such meagre gain.

It is possible to adapt the finger scheme to proper (a,b)-trees as follows: a (full) "finger" is the stack of pointers containing the most recently inspected path, and the entry giving rise to this path. Through this stack we have the possibility of going back to the parent along this path, retaining much of the savings. To avoid space overhead the finger can be woven into the tree by using pointer reversal techniques as in the Deutsch-Schorr-Waite algorithm [S&W]. The representation of an (a,b)-tree becomes a *pair* of pointers: one to the node in which the last accessed entry resides, and another to the parent node (which contains a reversed pointer to the grandparent, etc., until the root is reached). To enable proper use of the size fields, these must be updated to the weight of the actual (instead of virtual) dependent subtrees. However, this scheme requires desharing along the finger in all cases, even if no modification takes place.

For our application it is better to use a "little finger": a pointer to the last accessed entry (and its node), *without the possibility of going to the parent node* other than through the root. The node visit cost for the average random access search is even less for these "little" fingers than for the "full" ones: asymptotically $h-1$ as against $2(h-1)$, where h is the height of the tree. (In the case of the full fingers we cannot decide to initiate the search at the root for a random access, since the type of access is not clear in advance. This is impossible anyway if pointer reversal is used.) For a sequential search the little finger is less effective than the full variant, of course: on the average we need to visit $(h-1)\ /\ (r-1)$ and $2\ /\ (r-1)$ nodes respectively, where r is the mean number of children of the nodes. Little fingers behave better than their full counterparts even if a small part of the accesses is random.

11. IMPLEMENTATION CONSIDERATIONS

Many factors play a role in determining the optimum values for a and b, the bounds on the number of children in a node. The choice for $(2,3)$-trees saves some storage and code. However, the use of extra fields in each node (reference count, size of dependent tree, and "normal" or "range" mark) will lead to considerable storage overhead per entry for low bounds. Increasing the bounds arbitrarily, on the other hand, leads to a substantial waste of space if—as is the normal case—the majority of values have a small size. An argument in favour of large bounds is that the number of node rearrangements during an insertion or deletion is inversely proportional to the average number of children [BAY]; this leads to more memory-manager calls for low branching orders. If (parts of) the trees are stored on secondary memory, large bounds reduce the number of expensive memory accesses. If the bounds are arbitrarily increased, however, the time for rearranging nodes will eventually dominate. It seems to us that the best way of choosing a and b in practice is to run a collection of diverse, representative programs for various trial values. In any case, it is worthwhile to choose weak B-trees, since these improve worst-case costs drastically in some situations that might realistically occur. Some analytical calculations, assuming "typical" values for the costs of direct memory copying and of memory-manager calls, and a distribution of sizes where low sizes are predominant, suggests the choice $a = 6$ and $b = 12$. These calculations do not take into account the possibility of choosing different bounds for texts, list and tables, in such a way that all nodes get the same physical size. The latter will simplify memory management and make it more efficient. This may save more time than is lost by not choosing optimal uniform bounds.

At the bottom level of the B-tree all children are guaranteed to be empty. This represents a considerable waste of storage space, since the large majority of the nodes are at this level. Following a suggestion of [KNU], these "nil pointers" may be omitted, and (independently of this) a larger

number of entries may be kept in the bottom-level nodes. As long as values fit in one bottom node, this is, effectively, a linear representation. For small values, this is the most efficient representation.

12. DISCUSSION.

The issue of implementing variable-length data types has received too little attention in the literature. The reason for this is, undoubtedly, that existing programming languages have offered no facilities for data of variable length, because of a supposed "inherent inefficiency" in allocating memory for such data. This means, however, that in practice, if programmers are faced with a problem that lends itself naturally to expression in terms of lists or tables, say, they are forced to choose a less straightforward formulation of their problem, possibly introducing programming errors in the process, or to choose arbitrary limitations on the problem size (a major nuisance of many application programs), or to implement the missing type in terms of the available fixed-length types, which requires a substantial programmer effort and will result in a less efficient program than if the programming language itself had provided the necessary facilities.

Given the existence of variable-length data types in *B*, one may wonder if the use of B-trees is not a form of overkill. A previous experimental implementation of *B* uses a much simpler strategy: all data of variable length are allocated in contiguous memory segments. If extension of the length causes overflow, a new segment is requested that will accomodate eight more items, so that allocation is not necessary for each extension. For values of small length, this is satisfactory, but beyond a certain length it becomes too painful. One possible worry was that a B-tree implementation, although better for long values, would be too painful for small ones. However, some preliminary results indicate that this is not the case. We compared the times, required by the old and the proposed method, both written in C, for a typical operation such as

```
PUT t^'a' IN t.
```

The B-tree implementation was not tuned to optimal values for a and b, but used $a = 6$ and $b = 12$. As long as the length of `t` was on the average less than 200, the B-tree version was on the average faster. After that, the old version is faster (but by less than a factor of three), until the length of `t` approaches 3000, after which it is clearly overtaken. In this case, neither implementation is aware that the new value of `t` could be computed *in situ*, provided that the old value was not shared. An optimizing implementation could find this out, but this optimization is not foreseen for the next version. However, for the semantically equivalent operation

```
PUT 'a' IN t@(#t+1)
```

it is clear that no "desharing" is needed, and both implementations use this fact. Here, the B-tree version was faster over the full range of lengths.

Another implementation of variable-length data types is that used for SETL [SSS]. There an attempt is made to automatically select efficient data representations during compile time for undeclared program variables. Various data structures are used, like dynamic arrays and several variants of "breathing" linked hash tables. In a way, our approach is exactly the opposite: these authors propose to implement several data structures so that they can choose the best one for each different application, whereas we implement only one data structure that applies reasonably well to three different data types.

REFERENCES

[AHU] AHO, A.V. HOPCROFT, J.E. & ULLMANN, J.D., *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

[BAY] BAYER, R. & McCREIGHT E., Organization and maintenance of large ordered indexes, Acta Informatica 1 (1972), 173-189.

[BRO] BROWN, M.R. & TARJAN R.E., Design and analysis of a data structure for representing sorted lists, SIAM J. Computing 9 (1980), 594-614.

[COM] COMER, D., The ubiquitous B-tree, Computing Surveys 11 (1979), 121-137.

[HIB] HIBBARD, P.G., KNUEVEN P. & LEVERETT B.W., A stackless run-time implementation scheme, in Proc. 4th Intern. Conf. on the Design and Implementation of Algorithmic Languages, Dewar R.B.K.(Ed.), Courant Institute of Math. Sc., New York, 1976.

[HUD] HUDDLESTON, S. & MEHLHORN K., A new data structure for representing sorted lists, Acta Informatica 17 (1982), 157-184.

[KNU] KNUTH, D.E., *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, Mass. 1975.

[MAI] MAIER, D. & SALVETER S.C., Hysterical B-trees, Information Processing Letters 12 (1981), 199-202.

[MEE] MEERTENS, L.G.L.T., *Draft Proposal for the B Programming Language—Semi-Formal Definition*, Mathematical Centre, Amsterdam, 1981.

[OTT] OTTMANN, Th. & SCHRAPP M., 1-Pass top-down update schemes for balanced search trees, Forschungsbericht 107, Univ. Karlsruhe, 1981.

[ROS] ROSENBERG, A.L. & SNYDER L., Time- and space-optimality in B-trees, ACM Trans. on Database Systems 6 (1981), 174-183.

[SSS] SCHONBERG, E., SCHWARTZ J.T. & SHARIR M., Automatic data structure selection in SETL, in ACM Principles of Programming Languages 6 (1979), 197-210.

[S&W] SCHORR, H. & WAITE W.M., An efficient and machine-independent procedure for garbage-collection in various list-structures, Comm. ACM 10 (1967), 501-506.