PREPRINT. In Proc. 36th IEEE Symposium on Foundations of Computer Science, pages 655-663. ACM Press, 1995.

Sublogarithmic Searching Without Multiplications

Arne Andersson
Department of Computer Science
Lund University
Box 118, S–221 00 Lund, Sweden
arne@dna.lth.se

Abstract

We show that a unit-cost RAM with word length w can maintain an ordered set of w-bit integers (or binary strings) under the operations search, insert, delete, nearest neighbour in $O(\sqrt{\log n})$ worst-case time and range queries in $O(\sqrt{\log n}+$size of output$)$ worst-case time. The operations rely on AC^0 instructions only, thereby solving an open problem posed by Fredman and Willard. The data structure is simple.

We also present a static data structure that can process a set of $\Theta(\log n)$ searches in $O(\log n \log\log n)$ time.

1 Introduction

One of the most fundamental problems in computer science is to maintain an ordered set, supporting operations like insert, search, delete, neighbour and range search efficiently. Until a couple of years ago, it was believed that the information-theoretic lower bound of $\Theta(\log n)$ served as a general bound on the complexity per operation, where n denotes the number of stored elements. This belief was contradicted by Fredman and Willard who showed that, under the rather realistic assumption that the stored elements could be treated as binary strings or integers, operations could be performed in $O(\sqrt{\log n})$ amortized time [5].

The data structure by Fredman and Willard, the fusion tree, is of undisputed theoretical importance but some objections have been raised. First, in contrast to previous data structures like binary search trees [1] and van Emde Boas trees [15, 16], fusion trees make extensive use of multiplication. It may be argued that the cost of a multiplication is dependent on the length of the machine word, here denoted by w. Taking the cost of multiplication into account, it is even doubtful if fusion trees offer any improvement over van Emde Boas trees, which support searches and updates in $O(\log w)$ time. For this reason, Fredman and Willard posed as an open problem whether a sublogarithmic search cost could be achieved without multiplication. In particular, in order to argue that the cost is independent of the size of a machine word, one should only use AC^0 instructions.

A second objection against fusion trees is that they are complicated and involve high constant factors.

In this article we answer the question by Fredman and Willard affirmatively. We show that an ordered set can be maintained at a worst-case cost of $O(\sqrt{\log n})$ per update. The result is achieved by surprisingly simple methods. We only use instructions in AC^0, including comparison, addition, subtraction, bitwise AND and OR, and shift. By shift we mean shifting an entire word by a number of bit positions specified in a second word. Another advantage is that while fusion trees need to know a number of precomputed constants depending on w (these constants may be costly to compute), we only need to know the value of w.

Our data structure requires superlinear space, which can be reduced to linear, using hashing [3]. This will require the use of randomization, multiplication, and integer division but our data structure will still be much simpler than fusion trees.

Our data structure employs multiple comparisons, introduced by Paul and Simon [11] and also used in fusion trees (a description can be found in the textbook by Mehlhorn [7]). Our application of the technique is very simple: in particular we avoid the use of multiplication by means of a global lookup table.

Related work

On a high level, the techniques used here are similar to those used to obtain sorting in $O(n \log\log n)$ time [2]: A range reduction reduces the problem of maintaining long keys into that of maintaining short keys that can be packed into words and treated efficiently.

Similar observations have been made independently by Raman [12] who has shown that, if multiplications are used, a complexity of $O(1 + \log n/\log w)$ can be achieved. This implies a complexity of $o(\sqrt{\log n})$ for large word sizes.

Thorup has shown a number of related results for priority queues. In particular, he has developed a simple priority queue with a cost of $O(\log\log n)$ per operation without multiplications [14]. As a more general result, he has shown that there is a direct correspondence between the complexity of sorting and that of maintaining a priority queue [13].

It is intersting to note that while sorting can be done in $O(n \log\log n)$ time and a priority queue can be maintained in $O(\log\log n)$ time per operation, a corresponding bound cannot be achieved for searching. From a proof by Miltersen [9] a lower bound of $\Omega(\log^{1/3-o(1)} n)$ can be extracted [8] for the searching problem.

2 Main result

Theorem 1 *Given w as a parameter, there is a data structure for storing a set of n w-bit integers on a unit-cost RAM, which supports operations at the following costs:*

insert x: $O(\sqrt{\log n})$;

delete x: $O(\sqrt{\log n})$;

find x or report that x is not present: $O(1)$;

find the s largest or smallest elements: $O(s)$;

find the s largest elements $< x$ or the s smallest $> x$: $O(\sqrt{\log n} + s)$;

find all elements between x and y: $O(\sqrt{\log n}+$size of output$)$.

The representation can be deterministic or randomized. In the deterministic case, only AC^0 instructions are used, the costs above are worst-case, and $O(n2^{\epsilon w})$ space is used, for an arbitrary constant $\epsilon > 0$. In the randomized case, the costs are expected and $O(n)$ space is used.

If the cost of finding x or reporting that x is not present is allowed to be $O(\sqrt{\log n})$, the space complexity of the deterministic version decreases to $O(n + 2^{\epsilon w})$.

3 Packed B-trees

Apart from well-known data structures, we use a *packed B-tree*, derived from a traditional B-tree in a straightforward manner.

Lemma 1 *If $(k+1)D \leq w$ and $k \geq \log n$ then a set containing at most n k-bit integers can be maintained on a unit-cost RAM supporting operations insert x, delete x, or find x or its closest neighbour at a worst-case cost of $O\left(\log D + \frac{\log n}{\log D}\right)$.*

The data structure requires $O(n)$ space. Furthermore, the presence of a global lookup table of size $O\left(2^{D(k+1)}\right)$ is required. The table can be constructed in $O(D)$ time.

Proof: We use a packed B-tree.

In a B-tree node of degree d there are $d-1$ keys guiding the search and d pointers for branching. In a packed B-tree node all keys are packed in one machine word. Also, all pointers are packed in one word.

We use a packed B-tree with a maximal branching factor D. This implies that the degree of a node is roughly between $D/2$ and D. Each key is represented by a $(k+1)$-bit field. The first (leftmost) bit, the *test bit*, is 1, the following bits contain the key. The $d-1$ keys are stored in sorted left-to-right order in the rightmost $(d-1)(k+1)$ positions of a machine word. For a node v, the concatenated keys are denoted K_v. A packed B-tree keeps its nodes in an array of length n, so a pointer fits into $\log n$ bits. Hence, the d outgoing pointers can be packed in the rightmost $d \log n < kD < w$ bits of a machine word.

When searching for a k-bit key x in a packed B-tree, we first construct an integer X containing D copies of x. Again, each key is represented by a $(k+1)$-bit field. This time, the test bits are 0. X is created by a simple doubling technique: Starting with a word containing x in the rightmost part, we copy the word, shift the copy $k+1$ steps and unite the words with a bitwise OR. The resulting word is copied, shifted $2k+2$ steps and united, etc. Altogether X is generated in $O(\log D)$ time.

In order to determine the rank of x among the keys in node v, we perform a multiple comparison by the subtraction $R = (K_v - X)$ AND M, where M is a fixed mask in which all test bits are 1 and all other bits are 0. In R, each test bit corresponding to a key in K_v larger than x becomes 1, all other become 0. A multiple comparison is illustrated in Figure 1. Since the keys in K_v are sorted, there are only $D+1$ possible values of R; each value corresponds to a rank of x among the keys in K_v. This implies that we can construct a lookup table whose entries have the possible values of R as indices. (The same table can be used at all nodes in the tree and—of course—by more than one tree.) Looking in this table, we can determine the rank of x in K_v. (The address space required for the table is $O(2^{D(k+1)})$.) The table can be constructed in $O(D)$ time. Hence, formally, the use of packed B-trees requires a preprocessing time of $O(D)$. However, this can be taken care of during global rebuildings, see Section 4.

Having determined the rank as described above, we use that information to extract the pointer to the proper subtree by shift and bitwise logical operations.

The tree is maintained by traditional B-tree operations. Operations like adding a key (and the corresponding pointer) to a node, removing a key from a node, joining two nodes, and splitting a node are easily performed in constant time using shift and bitwise logical operations. □

How to implement a packed B-tree is illustrated in the Appendix.

A comment: In our application of packed B-trees $n = 2^k$. In that case we can actually omit the packed pointers; the key values themselves can be used as pointers instead. This option has not been utilized in our implementation.

4 Proof of Theorem 1

In our description below, we assume that n remains about the same during updates. When the value of n has changed significantly, our data structure has to be globally rebuilt. It can easily be arranged so that these rebuildings occur with large intervals; new versions of the data structure can be constructed as a background process. A more detailed discussion of global rebuilding can be found in the literature [10]. At each global rebuilding, we need to compute some constants of size $O(\sqrt{\log n})$, which can easily be computed in $o(n)$ time without multiplications.

4.1 The essential observation

If a packed B-tree can handle k-bit keys efficiently, a natural idea is to view our w-bit keys as consisting of k-bit characters and to store them in a trie of height w/k. At each internal node, the children represent a set of k-bit keys which can be stored in a packed

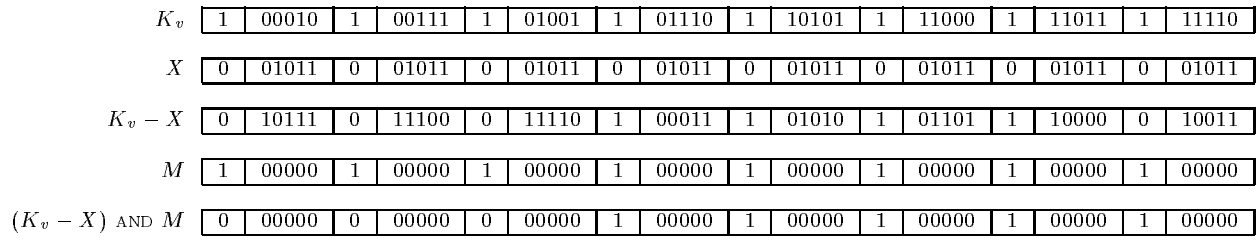

Figure 1: A multiple comparison

B-tree. When searching for a key x or x's nearest neighbour, we first search the trie in the usual way. If x is present, the search ends at a trie node. Otherwise, the trie traversal ends when trying to reach a non-existing child. Then, to locate x's nearest neighbour we make a local search among the children of the node, using the packed B-tree.

When inserting an element, we simply add a new child to a node in the trie; the corresponding k-bit key is inserted into the packed B-tree. Deletion is similar.

As an example, for $k = \Theta(w/\sqrt{\log n})$ the height of the trie as well as the degree of the packed B-tree becomes $\Theta(\sqrt{\log n})$ and the cost of an update or neighbour search becomes $\Theta(\log n/\log\log n)$. This simple data structure is coded in the Appendix.

4.2 The proof

The idea above can also be applied on a partial van Emde Boas tree, which gives the desired complexity.

First, we view each w-bit integer as composed of ϵw-bit short integers and we represent our set as a path-compressed trie of height $1/\epsilon$. Each node keeps an array of size $2^{\epsilon w}$ containing all possible outgoing edges. The children represent a set of ϵw-bit keys, these are stored in a partial van Emde Boas tre, VEB, as described below.

A van Emde Boas tree (VEB) [16] is a recursive trie structure where the length of the keys representing edges is halved at each recursive level. The data structure contains $O(n)$ nodes and $O(n)$ edges [15] and it supports neighbour searches efficiently. Normally, a VEB representing ϵw-bit keys would have a height of $O(\log(\epsilon w))$. However, instead of letting the VEB do all the work, it may be advantageous to cut off the recursion and switch to a packed B-tree. Cutting after $O(\sqrt{\log n})$ levels, the sizes of the integers have decreased to $\min(\log n, \frac{\epsilon w}{2^{\sqrt{\log n}}})$. If the size of the integers is at most $\log n$, we just complete the VEB in $O(\log\log n)$ additional recursive levels, the complexity follows immediately. Otherwise, at the bottom of the VEB we use packed B-trees with $k = \frac{\epsilon w}{2^{\sqrt{\log n}}}$ and $D = 2^{\sqrt{\log n}}$. The complexity follows from Lemma 1.

All elements are stored in a doubly-linked list. When a new element is inserted, we can find its neighbours in $O(\sqrt{\log n})$ time and update the list. With the list, we can perform range queries efficiently, as well as finding the largest and smallest elements.

The total number of internal nodes in the path-compressed trie is less than n and each node uses $O(2^{\epsilon w})$ space. There are less than $2n$ VEBs containing a total of at most $2n$ short integers. Each node in a VEB uses an address space of $O(2^{\epsilon w})$ and the total number of such nodes is $O(n)$. The packed B-trees use a total of $O(n)$ space and the global lookup table used for multiple comparisons use $O(2^{\epsilon w})$ space. In total, the required space is $O(n2^{\epsilon w})$.

If we use hashing, the space becomes proportional to the number of nodes in the used data structures plus the number of "active" entries in the global lookup table. These quantities are all $O(n)$. In the hashing case, we do not need the trie at the top level, we only combine one VEB with packed B-trees.

4.3 Reducing space

If we allow the cost of member queries to be $O(\sqrt{\log n})$ the space complexity can be reduced to $O(n + 2^{\epsilon w})$ or—if U denotes the size of the universe—to $O(n+U^{\epsilon})$. We use the data structure above, with two modifications.

First, we apply an idea similar to that of Willard [17] and Karlsson [6] to reduce the number of nodes in the trie by keeping (traditional) 2-3 trees of height $\Theta(\sqrt{\log n})$ at the bottom of the trie. All 2-3 trees have the same height and updates in the trie corresponds to splitting and merging of 2-3 trees. Most keys will be stored in the 2-3 trees, choosing suitable constants, we ensure that the number of keys stored in the trie is less than $\frac{n}{2^{2\sqrt{\log n}}}$.

Secondly, we replace ϵ in the proof above by $\frac{\epsilon}{2\sqrt{\log n}}$.

These modifications increase the cost of searches and updates by an additional term of $O(\sqrt{\log n})$. The B-trees use linear space and the trie uses a total space of $O\left(\frac{n}{2^{2\sqrt{\log n}}} \cdot 2^{\left(\frac{\epsilon w}{2\sqrt{\log n}}\right)}\right)$. To see that this is $O(n + 2^{\epsilon w})$ we distinguish two cases:

1. $n \le 2^{\epsilon w/2}$. Both terms are $O(2^{\epsilon w/2})$.
2. $n > 2^{\epsilon w/2}$. The second term is $O\left(2^{2\sqrt{\log n}}\right)$.

5 Batched Searching

The technique of combining range reduction with multiple comparisons can be used to obtain an efficient *static* data structure. The structure is only of theoretical interest, hardly more than a curiosity. The space cost and preprocessing time is rather high.

Theorem 2 *There is a data structure which stores n elements where $\Theta(\log n)$ arbitrary elements (or their closest neighbours) can be located in $O(\log n \log\log n)$ time. The data structure uses $O(n^{O(\log n)})$ space.*

Proof: (Sketch) We use a VEB where the recursion is cut off after $\log\log n$ levels. At the bottom level of the VEB we store n sets of short integers, the total number of integers is n. We can pack $\Theta(\log n)$ integers in one machine word. Each set is represented as a perfectly balanced binary search tree of height (at most) $\log n$. The trees are represented in an odd—and space consuming—way, as described below.

When searching for $\log n$ elements, we proceed as follows. For each element, we search down the VEB. The total cost of this is $O(\log n \log\log n)$. Each search ends up in one of the n search trees. This gives a combination of $\log n$ search trees. The total number of possible combinations is at most $n^{\log n}$. Each of these combinations is represented by one machine word, containing the $\log n$ root keys. Storing the machine words containing packed root keys in an array of size $n^{\log n}$, we can find the proper word in constant time. We pack our $\log n$ query elements in a machine word. Now, with a multiple comparison we can compare each key with its proper root key. There are $2^{\log n} = n$ possible outcomes of this comparison. This gives us a total of $n^{\log n} \cdot n$ possible combinations of subtrees, each combination is represented by one machine word. In this way we traverse all trees simultaneously; after at most $\log n$ steps our search is completed. The number of combinations increases by a factor of n at each step, the total number becomes at most $n^{2\log n}$. Altogether, if the VEB and the packed key tables are stored with perfect hashing [4], we can perform a set of log n searches in time $O(\log n \log\log n)$ using $n^{2\log n}$ space. (Choosing the parameters more carefully, the space can easily be reduced to $n^{\epsilon \log n}$ for an arbitrary positive constant ϵ.) □

6 Comments

We have shown that the information-theoretic barrier can be surpassed by an algorithm simple enough to be presented in an elementary textbook.

Our data structure can be extended to handle long strings. If we use a trie where $\Theta(w)$ bits are used for branching and where each internal node is represented by the data structure described above, a string contained in W machine words can be processed in $O(W + \sqrt{\log n})$ time. Hence, when handling long strings, multiple-precision numbers etc, the advantage over comparison-based algorithms becomes even more evident.

Acknowledgements

The author would like to thank Rolf Karlsson, Jesper Larsson, Stefan Nilsson, Ola Petersson, and Kurt Swanson for their valuable comments.

References

[1] G. M. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Dokladi Akademia Nauk SSSR*, 146(2):1259–1262, 1962.

[2] A. Andersson, T. Hagerup, S. Nilsson, and R. Raman. Sorting in linear time? In *Proceedings 27^{th} ACM Symposium on Theory of Computing*, pages 427–436. ACM Press, 1995.

[3] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[4] M. L. Fredman, J. Komlos, and E. Szemeredi. Storing a sparse table with $O(1)$ worst case access time. *Journal of the ACM*, 31(3):538–544, 1984.

[5] M. L. Fredman and D. E. Willard. Surpassing the information theoretic barrier with fusion trees. *J. Comput. Syst. Sci.*, 47:424–436, 1994.

[6] R. Karlsson. *Algorithms in a Restricted Universe.* Ph. D. Thesis, University of Waterloo, Canada, 1984.

[7] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching.* Springer-Verlag, 1984. ISBN 3-540-13302-X.

[8] P. B. Miltersen. Personal communication.

[9] P. B. Miltersen. Lower bounds for union-split-find related problems on random access machines. In *Proc. 26th Ann. ACM STOC*, pages 625–634, 1994.

[10] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science.* Springer Verlag, 1983. ISBN 3-540-12330-X.

[11] W. J. Paul and J. Simon. Decision trees and random access machines. In *Proc. International Symp. on Logic and Algorithmic, Zürich*, pages 331–340, 1980.

[12] R. Raman. Improved data structures for predecessor queries in integer sets. manuscript, 1995.

[13] M. Thorup. Equivalence between sorting and priority queues. Tech. report, DIMACS TR-95-12, DIMACS, 1995.

[14] M. Thorup. An $O(\log\log n)$ priority queue. Tech. report, DIKU-TR-95-5, Dept. of Computer Science, University of Copenhagen, 1995.

[15] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters*, 6(3):80–82, 1977.

[16] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Math. Syst. Theory*, 10:99–127, 1977.

[17] D. E. Willard. New trie data structures which support very fast search operations. *Journal of Computer and System Sciences*, 28:379–394, 1984.

APPENDIX: An implementation

Enclosing an implementation serves two purposes:

- The code itself illustrates the fact that a data structure surpassing the information theoretic barrier can be implemented without much effort.
- The reader can, by running the code, convince himself or herself that the involved constant factors are small.

Code is only given for insertion and search. The data structure is also somewhat simplified: instead of a trie containing VEBs containing packed B-trees, we only use a plain trie containing packed B-trees. Modifying the trie into a VEB and including code for deletion is rather simple. Our implementation assumes that a key of type `WORD` has its most significant bit in the leftmost position. The implementation is made without the use of any multiplications. (Multiplications introduced by the compiler, such as when indexing arrays, could be removed by hard-coding.)

As a simplification, the global rebuilding of the data structure is not made. Instead, the data structure is initialized with a given value of n and the parameters (height of trie, degree of B-tree etc) are set only then.

Since we just use a trie and not a VEB, the asymptotic complexity will be different. We split the machine word into $\Theta(\sqrt{\log n})$ fields. The height of the trie as well as the degree of the B-tree becomes $\Theta(\sqrt{\log n})$. This implies that the height of a packed B-tree, as well as the total cost of searching and updating, becomes $O(\log n / \log\log n)$. Still, the complexity is sublogarithmic, regardless of the word length.

We will not provide any experimental statistics but we have run the code on a SUN SPARCstation ELC using the `cc` and the `gcc` compiler with and without optimization on random input. The speed of the code, in terms of CPU time, was compared with that of the code for skip lists (W. Pugh, *Skip lists: a probabilistic alternative to balanced trees*, CACM 33(6), pages 668–676, 1990) announced by its inventor on the international network for anyone to fetch by anonymous ftp. Skip lists are known to be very fast in practice. Comparing execution times, we found that searching seem to be faster in our data structure even for small input sizes. Insertion takes about 2-3 times as long, the difference decreases as n increases. We conjecture that with a careful tuning of the code, our data structure will be very competitive also for updates.

More complete code will be made electronically available.

Comments on the code

Type declarations

Essentially, our data structure is a trie of degree $2^{\mathtt{k}}$. Each node contains an array of outgoing edges and a packed B-tree, containing the `k`-bit keys that are currently used for branching.

A variable of type `trie` represents a node in the trie. In this node, we have list pointers `min` and `max` to the minimum and maximum element in the current (sub-)trie. These two elements are not stored further down in the trie. In this way, each element will be stored as a `min` or `max` element of a trie node. This method guarantees that the total number of nodes in the trie will be $O(n)$ although path-compression is not used. When a trie node is created, its `min` and `max` pointer point at the same element. When a second element is inserted at the node, `min` and `max` will point at different elements. When a third element is inserted, the first child is created.

The array `Tnodes` contains $2^{\mathtt{k}}$ entries, each entry corresponds to a possible outgoing edge. When the array is allocated, we can not (at least in theory) afford to initialize all pointers as null pointers, instead we have to keep track of which pointers are used. For this purpose, we use two arrays; `checkpt` contains references to the used entries and `check` contains references back to `checkpt`. The outgoing edge with index `c` is ussed iff `check[c]` points within the active part of `checkpt` (indicated by `firstfree`) and `checkpt[check[c]]=c`. If deletions are to be made, the technique has to be a bit more, but not too, elaborate.

A packed B-tree inside a trie node contains at most $2^{\mathtt{k}}$ nodes, one for each possible child. (Note that this is a bit overestimated since each node, except possibly the root, contains more than one key.) The B-tree is stored in an array where the entries are of type `BtreeNode`. A `BtreeNode` representing a node of degree d has the following contents:

`size` $= d - 1$ (that is, `size` equals the number of keys in the node.

`key` contains the `size` keys used for branching. They are stored in sorted left-right order, right adjusted. Between each key there is a test bit, which is 1.

`pt` contains `size+1` pointers to subtrees. A pointer is just an index into the array of nodes. Since the length of this array is $2^{\mathtt{k}}$, a pointer requires only `k` bits. (Here we use `k+1` bits.)

Global variables

The use of `rTab`, and `rShift` is explained together with function `MultCmp`.

`minD` and `maxD` are the minimum and maximum degrees of a B-tree node.

`maxDepth` tells the maximum depth of a leaf in the trie. Currently only used during initialization.

`checkArraySize`, `trieArraySize`, and `nodeArraySize` are used for memory allocation. They tell the sizes of the arrays stored in a trie node.

`M` is the mask used at multiple comparisons to extract test bits. All test bits in `M` are ones, all other bits are zeroes.

`k` is the number of bits in a packed key. At each node in the trie, except possibly the root, `k` bits are used for branching. A packed B-tree contains `k`-bit keys.

`rootbits` is the number of bits used for branching at the root of the trie. (At the other levels `k` bits are used.) The reason that `rootbits` may differ from `k` is simply that w is not always divisible by `k`.

`mulk` and `mulk1` are arrays used for primitive multiplication simulation; `mulk[i]` contains the value of `k`$\cdot$`i` while `mulk1[i]` contains (`k+1`)$\cdot$`i`.

Macros and Subroutines

`EX(m,lo,hi)` Extract bits `lo` to `hi` from `m`. Bits are counted from right, starting with 0. Shift operations are used.

`EXF(m,pos,num)` Extract `num (k+1)`-bit fields, starting at field `pos`. Fields are numbered from right, starting with 0.

`FillKey(X)` Duplicate a `(k+1)`-bit field within a word in order to take part in a multiple comparison. The doubling technique is used.

`WORD MultCmp(key, X, size)` Multiple comparison. Compute the number of keys in `key` that are larger than `X`'s duplicated key. `size` tells the number of keys stored in `key`.

After the subtraction and masking with `M`, some of the rightmost test bits in `tmp` are ones. Using `tmp` as an index, the number of one bits can be found in `rTab`. Since the length of `rTab` is less than the universe, we make more than one access in `rTab`, using a part of `tmp` as index each time., The number of bits used each time is given by `rShift`.

The size of `rTab` and the value of `rShift`, depend on w and ϵ. They are computed in `Init`.

`WORD BtreeSearch(bt, X)` Search for the `k`-bit key stored in `X`. If present, the key is returned, otherwise a closest neighbour is returned.

(a) At the bottom of the tree
(b) If the key in `X` is smaller than all keys in the node, take smallest key.
(c) Go to child.

`AddField(keys, pos, X)` Viewing `*keys` as consisting of `(k+1)`-bit fields, insert the rightmost field in `X` into `*keys` at position `pos`, counting from right, starting at 0.

`WORD NewNode(bt)` Initialize a new B-tree node. The index of the new node is returned.

`SplitChild(bt, par, c, pos)` Split the child `c` of parent `par` into two B-tree nodes. A new node `s` is created. A reference to `s` is added in `par`. `bt` is the B-tree where the nodes are stored, `pos` is the position in node `par` where the new key is to be added.

`BtreeIns(bt, X, neighb)` Insert the `k`-bit field stored in `X` into the packed B-tree `*bt`. One of `X`'s closest neighbours is returned in `*neighb`. (There is no need to take care of insertion of duplicate keys.) Balancing is made top-down. The following comments refers to the corresponding comments in the code:

(a) The root too large, a split is made.
(b) Traverse down the tree.
(c) Compute position of child pointer.
(d) Extract child pointer
(e) Split the child in two nodes.
(f) Go to child.
(g) At the bottom of the tree, compute where to insert key.
(h) Finally, find a neighbour.

`LinkBefore (new, p), LinkAfter (new, p)` Insert node `new` into linked list before/after node `p`.

`boolean ChildExists(t, c)` Check if child `c` exists below trie node `t`. The arrays `check` and `checkpt` are used.

`list *TrieSearch(t, X)` Return a pointer to the list element containing `X` or, if `X` is not present, to a closest neighbour.

If `X` is present, only the trie will be traversed, `X` will be found as the `max` or `min` element of some trie node.

If `X` is not present, the trie traversal will end when trying to reach a non-existing child. Then, if there are no children, the `min` element of the current trie node serves as a closest neighbour. If there are children, we search the node's packed B-tree for a neighbour.

`MarkNewChild(t, c)` Mark a new child in the arrays `check` and `checkpt`.

`trie *NewTrie(min)` Initialize a new trie node and returns a pointer to the node. `min` is the element to be inserted as the new node's minimum element.

`CreateArrays(t)` When the first child of `t` is created, some further initialization is made. Among others, space is allocated for the arrays `check, ckeckpt,` and `Tnodes`.

`TrieIns(t, X)`, Main insertion procedure. Insert key `X` into trie `t`.

(a) Traverse down the trie.
(b) Return if `X` is already present.
(c) If there is only one min/max element add one and return.
(d) Check if the elements in `min` or `max` should be replaced. If so, the replaced key should be inserted at a lower level.
(e) Extract child pointer.
(f) If the trie node `tmp` has no child before, initialize arrays and create one new child. Insert the corresponding `k`-bit field `c` into the (empty) packed B-tree. Insert the element `X` into the linked list.
(g) Parameter `neighb` is not used in this case. The reason is that the B-tree is empty and there is no neighbour to be found. Instead we can take the `min` element of the current trie node (`tmp`) as the neighbour of the newly inserted one.
(h) If `tmp` has children before, but the child corresponding to the `k`-bit field `c` is not present, a new child is created in `tmp->Tnodes[c]` and `c` is inserted into the B-tree. After the B-tree insertion, `neighb` will contain a `k`-bit field corresponding to a closest neighbour to the

child `c`. Then, the newly inserted element `X`, stored in `tmp->Tnodes[c]->min`, should be inserted into the linked list as the closest neighbour of the maximum or minimum element in `tmp->Tnodes[neighb]`.

(i) Go down to child.

(j) Adjust the number of bits to be skipped as `X` is scanned left-to-right.

`WORD Divide(x, y, remainder)` Primitive division, used during initialization.

`Init (N, eps_inv)` Initialize global constants and tables. The value of `N` controls the parameters of the data structure. The height of the trie and the degree of the packed B-trees are set to $\Theta(\sqrt{\log N})$. `eps_inv` is the value of $1/\epsilon$. This parameter controls the size of `rTab`, the size of `rTab` should be at most $2^{\epsilon w}$. (A good choice for `eps_inv` is 2.

Since the height of the trie is $\Theta(\sqrt{\log N})$, the degree of each node is $\Theta\left(2^{\sqrt{\log N}}\right)$. Hence, the total space complexity will be $\Theta\left(n2^{\sqrt{\log N}} + 2^{\epsilon w}\right)$.

```
#define w 32                /* machine dependent */
#define TRUE 1
#define FALSE 0
#define boolean int
#define WORD unsigned long       /* w-bit words */

typedef struct list {
  WORD key;
  struct list *prev, *next;
} list;
typedef struct BtreeNode {
  WORD key, pt, size;
} BtreeNode;
typedef struct Btree {
  BtreeNode *nodes;
  WORD root, lastnode, height;
} Btree;
typedef struct trie {
  list *min, *max;
  WORD firstfree, *check, *checkpt;
  struct trie **Tnodes;
  Btree bt;
} trie;

WORD checkArraySize, trieArraySize, nodeArraySize, k,
  maxD, minD, rootbits, maxDepth, M,
  *rTab, rShift, rTabSize,
  mulk[w+1], mulk1[w+1];

/******* Macros for extracting parts of word *********/

#define EX(m,lo,hi) (m<<w-(hi)-1)>>w-(hi)+lo-1
#define EXF(m,pos,num) (m<<w-mulk1[pos+num]) \
        >>w-mulk1[pos+num]+mulk1[pos]
```

```
Swap(x, y)
WORD *x, *y;
{
  WORD tmp;

  tmp = *x;  *x = *y;   *y = tmp;
}

/************** Packed B-tree ************************/

FillKey(X)
     WORD *X;
{
  WORD d = 1;

  while (d < maxD-1) {
    *X= *X | (*X<<mulk1[d]);
    d+= d;
  }
  *X = EXF(*X, 0, maxD);
}

WORD MultCmp(key, X, size)
     WORD key, X, size;
{
  WORD tmp, pos = 0;

  tmp = EXF(((key-X) & M), 0, size);
  while (tmp > 0) {
    pos = pos+rTab[EX(tmp,0,rShift-1)];
    tmp = tmp>>rShift;
  }
  return pos;
}

WORD BtreeSearch(bt, X)
     Btree *bt; WORD X;
{
  WORD pos, height = bt->height, n = bt->root;

  FillKey(&X);
  while (TRUE) {
    pos = MultCmp(bt->nodes[n].key, X,
                  bt->nodes[n].size);
    if (height == 0) {                  /* a */
      if (pos == bt->nodes[n].size)     /* b */
        pos = bt->nodes[n].size-1;
      return EX(bt->nodes[n].key, mulk1[pos],
                mulk1[pos]+k-1);
    }
   n = EXF(bt->nodes[n].pt, pos, 1);  /* c */
   height--;
  }
}

AddField(keys, pos, X)
     WORD *keys, pos, X;
{
  WORD tmp;

  tmp = (((*keys>>mulk1[pos])<<k+1) | X)
        <<mulk1[pos];
  if (pos==0) *keys=0;
  else *keys = EXF(*keys, 0, pos);
  *keys = EXF((tmp|*keys), 0, maxD);
}
```

```
WORD NewNode(bt)
     Btree *bt;
{
  bt->lastnode++;
  bt->nodes[bt->lastnode].key = M;
  bt->nodes[bt->lastnode].pt = 0;
  bt->nodes[bt->lastnode].size = 0;
  return bt->lastnode;
}

SplitChild(bt, par, c, pos)
     Btree *bt; WORD par, c, pos;
{
  WORD s = NewNode(bt);

  AddField(&bt->nodes[par].key, pos,
           EXF(bt->nodes[c].key, minD-1, 1));
  AddField(&bt->nodes[par].pt, pos, s);
  bt->nodes[par].size++;
  bt->nodes[s].key = EXF(bt->nodes[c].key, 0, minD-1);
  bt->nodes[s].pt = EXF(bt->nodes[c].pt, 0, minD);
  bt->nodes[s].key = bt->nodes[s].key | M;
  bt->nodes[s].size = minD-1;
  bt->nodes[c].key = EXF(bt->nodes[c].key,minD,minD-1);
  bt->nodes[c].pt = EXF(bt->nodes[c].pt, minD, minD);
  bt->nodes[c].key = bt->nodes[c].key | M;
  bt->nodes[c].size = minD-1;
}

BtreeIns(bt, X, neighb)
     Btree *bt; WORD X, *neighb;
{
  WORD par, child, height, pos;

  FillKey(&X);
  if (bt->nodes[bt->root].size == maxD-1) {
    child = bt->root;                       /* a */
    bt->root = NewNode(bt);
    bt->nodes[bt->root].pt = child;
    bt->height++;
    SplitChild(bt, bt->root, child, 0);
  }
  par = bt->root;
  height = bt->height;
  while (height > 0) {                      /* b */
    pos = MultCmp(bt->nodes[par].key,       /* c */
                  X, bt->nodes[par].size);
    child = EXF(bt->nodes[par].pt, pos, 1); /* d */
    if (bt->nodes[child].size == maxD-1)
      SplitChild(bt, par, child, pos);      /* e */
    else {                                  /* f */
      height--;
      par = child;
    }
  }
  pos = MultCmp(bt->nodes[par].key,         /* g */
                X, bt->nodes[par].size);
  AddField(&bt->nodes[par].key, pos, (EXF(X, 0, 1)));
  bt->nodes[par].key = bt->nodes[par].key | M;
  bt->nodes[par].size++;
  if (pos == 0)                             /* h */
    *neighb = EX(bt->nodes[par].key, k+1, mulk[2]);
  else  *neighb = EX(bt->nodes[par].key,
     mulk1[pos-1],mulk1[pos-1]+k-1);
}
```

```
/***************** List ***************************/

LinkBefore(new, p)
     list *new, *p;
{
  new->prev = p->prev;
  new->next = p;
  if (new->prev != NULL)  new->prev->next = new;
  p->prev = new;
}

LinkAfter(new, p)
     list *new, *p;
{
  new->next = p->next;
  new->prev = p;
  if (new->next != NULL)  new->next->prev = new;
  p->next = new;
}

/***************** Trie ***************************/

boolean ChildExists(t, c)
     trie *t; WORD c;
{
  return (t->checkpt[c] < t->firstfree
  && t->check[t->checkpt[c]] == c);
}

list *TrieSearch(t, X)
     trie *t; WORD X;
{
  WORD c, skip = w - rootbits;

  if (t == NULL) return NULL;
  while (TRUE) {
    if (X <= t->min->key)  return t->min;
    if (X >= t->max->key)  return t->max;
    if (t->check == NULL)  return t->min;
    c = EX((X >> skip), 0, k-1);
    if (ChildExists(t, c))  t = t->Tnodes[c];
    else t = t->Tnodes[BtreeSearch(&t->bt, c)];
    skip -= k;
  }
}

MarkNewChild(t, c)
     trie *t; WORD c;
{
  t->check[t->firstfree] = c;
  t->checkpt[c] = t->firstfree;
  t->firstfree++;
}

trie *NewTrie(min)
     WORD min;
{
  trie *tmp;

  tmp = (trie *)malloc(sizeof(trie));
  tmp->check = NULL;
  tmp->min = (list *)malloc(sizeof(list));
  tmp->min->key = min;    tmp->max = tmp->min;
  tmp->min->prev = tmp->min->next = NULL;
  return tmp;
}
```

```
CreateArrays(t)
     trie *t;
{
  WORD neighb;
  t->check = (WORD *)malloc(checkArraySize);
  t->checkpt = (WORD *)malloc(checkArraySize);
  t->Tnodes = (trie **)malloc(trieArraySize);
  t->bt.nodes = (BtreeNode *)malloc(nodeArraySize);
  t->firstfree = t->bt.lastnode = 0;
  t->bt.root = NewNode(&t->bt);
  t->bt.height = 0;
}

TrieIns(t, X)
     trie **t; WORD X;
{
  WORD neighb, c, skip = w - rootbits; trie *tmp;

  if (*t == NULL) {
      *t = NewTrie(X);
      return;
    }
  tmp = *t;
  while (TRUE) {                        /* a */
    if (X == tmp->min->key || X == tmp->max->key)
      return;                           /* b */
    if (tmp->min == tmp->max) {         /* c */
      tmp->max = (list *)malloc(sizeof(list));
      tmp->max->key = X;
      LinkAfter(tmp->max, tmp->min);
      if (tmp->min->key > tmp->max->key)
        Swap(&tmp->min->key, &tmp->max->key);
      return;
    }
    if (X < tmp->min->key)              /* d */
      Swap(&X, &tmp->min->key);
    else if (X > tmp->max->key)
      Swap(&X,&tmp->max->key);
    c = EX((X >> skip), 0, k-1);        /* e */
    if (tmp->check == NULL) {           /* f */
      CreateArrays(tmp);
      tmp->Tnodes[c] = NewTrie(X);
      MarkNewChild(tmp, c);
      BtreeIns(&tmp->bt, c, &neighb);   /* g */
      LinkAfter(tmp->Tnodes[c]->min, tmp->min);
      return;
    }
    if (!ChildExists(tmp, c)) {         /* h */
      tmp->Tnodes[c] = NewTrie(X);
      MarkNewChild(tmp, c);
      BtreeIns(&tmp->bt, c, &neighb);
      if (c < neighb)
        LinkBefore(tmp->Tnodes[c]->min,
                   tmp->Tnodes[neighb]->min);
      else
        LinkAfter(tmp->Tnodes[c]->min,
                  tmp->Tnodes[neighb]->max);
      return;
    }
    tmp = tmp->Tnodes[c];           /* i */
    skip -= k;                      /* j */
  }
}
```

```
/***************** Initialization ******************/

WORD Divide(x, y, remainder)
     WORD x, y, *remainder;
{
  WORD out = 0;

  while (x >= y)  {
    x -= y;
    out++;
  }
  *remainder = x;
  return out;
}

Init (N, eps_inv)
     WORD N, eps_inv;
{
  WORD logn = 0, minDsqr, i, triedegree;

  do {
    N = N >> 1;
    logn++;
  } while (N != 0);
  minD = 3;
  minDsqr = 9;
  do {
    minD++;
    minDsqr += minD+minD-1;
  } while (minDsqr<logn);
  minD -= 1;
  maxD = minD+minD;
  k = Divide(w, maxD, &rootbits)-1;
  maxDepth = Divide(w, k, &rootbits)-1;
  if (rootbits > 0)  maxDepth++;
  else  rootbits = k;
  triedegree = 1 << k;
  checkArraySize = sizeof(WORD) << k;
  trieArraySize = sizeof(trie *) << k;
  nodeArraySize =
     (sizeof(WORD)+sizeof(WORD)+sizeof(WORD)) << k;
  mulk[0] = mulk1[0] = 0;
  for (i = 1; i <= maxD; i++) {
    mulk[i] = mulk[i-1]+k;
    mulk1[i] = mulk1[i-1]+k+1;
  }
  rTabSize = 1 << Divide (w, eps_inv, &i);
  rTab = (WORD*)malloc(sizeof(WORD)
          << Divide (w, eps_inv, &i));
  M = 1 << k;
  i = rTab[0] = 0;
  while (M <= rTabSize) {
    i++;
    rTab[M] = i;
    M = (M << k+1) | (1 << k);
  }
  rShift = mulk1[i];
  FillKey(&M);
}
```