Algorithmica (2002) 33: 19–33
DOI: 10.1007/s00453-001-0102-y

Algorithmica

An Experimental Study of Compression Methods for Dynamic Tries[1]

S. Nilsson[2] and M. Tikkanen[3]

Abstract. We study an order-preserving general purpose data structure for binary data, the LPC-trie. The structure is a compressed trie, using both level and path compression. The memory usage is similar to that of a balanced binary search tree, but the expected average depth is smaller. The LPC-trie is well suited to modern language environments with efficient memory allocation and garbage collection. We present an implementation in the Java programming language and show that the structure compares favorably with a balanced binary search tree.

Key Words. Data structures, Trie, LPC-trie, Level compression, Relaxed level compression, Memory management, Binary search tree, Java.

1. Introduction. We describe a dynamic main memory data structure for binary data, the level- and path-compressed trie or LPC-trie. The structure is a dynamic variant of a static level-compressed trie or LC-trie [2]. The trie is a simple order-preserving data structure supporting fast retrieval of elements and efficient nearest neighbor and range searches. There are several methods for implementing dynamic trie structures in the literature [5], [8], [10], [17], [21]. One of the drawbacks of these methods is that they need considerably more memory than a balanced binary search tree. In this study we show how to avoid this problem by using trie compression. In addition to the well-known path-compression technique we utilize level compression, a technique that reduces both the size and the depth of a trie. We show how to make the restructuring in a dynamic trie more efficient by introducing a relaxed criterion for level compression. Relaxed level compression also reduces the depth and the amount of memory.

There are two main reasons why we choose to study dynamic trie structures using experimental rather than analytical methods. First, the form of a trie depends on the kind of data stored. Hence analytic studies always need to use a data model, such as uniform distribution, independent random samples from a certain distribution function, or Bernoulli-type processes. It is well known that tries have a low average depth for all these types of data, but it is not immediately clear how these results translate to real-world data. In this study we have used samples of English text, Internet routing tables, and geographic point data. Second, a dynamic trie structure relies heavily upon automatic memory management and we wanted to study how this affects the performance. In fact,

[1] Code, test data, and experimental results accompanying this article are available from URL http://www.nada.kth.se/~snilsson.

[2] KTH, Nada, SE-100 44 Stockholm, Sweden. snilsson@nada.kth.se.

[3] Nokia Networks, P.O. Box 323, FIN-00045 Nokia Group, Finland. matti.j.tikkanen@nokia.com.

Received October 29, 1998; revised December, 2000. Communicated by R. Battiti and A. Bertossi.
Online publication February 20, 2002.

we were initially doubtful whether it would be feasible to implement a dynamic level-compressed trie at all, since the amount of memory management seemed prohibitive. It is difficult to approach this type of memory problem analytically, since the memory allocation patterns depend not only on the data, but also on the updates that are actually performed.

In its original form, the trie [14], [15] is a data structure where a set of strings from an alphabet containing m characters is stored in an m-ary tree and each string corresponds to a unique path. In this article we only consider binary trie structures, thereby avoiding the problem of representing large internal nodes. Using a binary alphabet tends to increase the depth of the trie when compared with character-based tries. To counter this potential problem we use two different compression techniques, path compression and level compression.

The average case behavior of trie structures has been the subject of thorough theoretic analysis[12], [18], [23], [24]; an extensive list of references can be found in the *Handbook of Theoretical Computer Science* [20]. The expected average depth of a trie containing n independent random strings from a distribution with density function $f \in L^2$ is $\Theta(\log n)$ [9]. This result holds also for data from a Bernoulli-type process[11], [13].

The best known compression technique for tries is path compression. The idea is simple: paths consisting of a sequence of single-child nodes are compressed, as shown in Figure 1(b). A path-compressed binary trie is often referred to as a Patricia trie. Path compression may reduce the size of the trie dramatically. In fact, the number of nodes in a path-compressed binary trie storing n keys is $2n-1$. The asymptotic expected average depth, however, is typically not reduced [16], [18].

Level compression [2] is a more recent technique. Once again, the idea is simple: subtries that are complete (all children are present) are compressed, and this compression is performed top down, see Figure 1(c). Previously this technique has only been used in static data structures, where efficient insertion and deletion operations are not provided [4]. The level-compressed trie, LC-trie, has proved to be of interest both in theory and practice. It is known that the average expected depth of an LC-trie is $O(\log\log n)$ for data from a large class of distributions [3]. This should be compared with the logarithmic depth of uncompressed and path-compressed tries. These results also translate to good performance in practice, as shown by a recent software implementation of IP routing tables using a static LC-trie [22].

One of the difficulties when implementing a dynamic compressed trie structure is that a single update operation may cause a large and costly restructuring of the trie. Our solution to this problem is to relax the criterion for level compression and allow compression to take place even when a subtrie is only partly filled. This has several advantages. There is less restructuring, because it is possible to do a number of updates in a partly filled node without violating the constraints triggering its resizing. In addition, this relaxed level compression reduces the depth of the trie even further. In some cases this reduction can be quite dramatic.

Data structures based on bit manipulation have also been studied by Larson [19], who describes a linear hashing-based main memory data structure, and Analyti and Pramanik [1], who compare the search performance of Larson's data structure to two extendible hashing-based main memory structures. Neither of these use any compression methods, however. The ternary search tree of Bentley and Sedgewick [6] combines ideas from

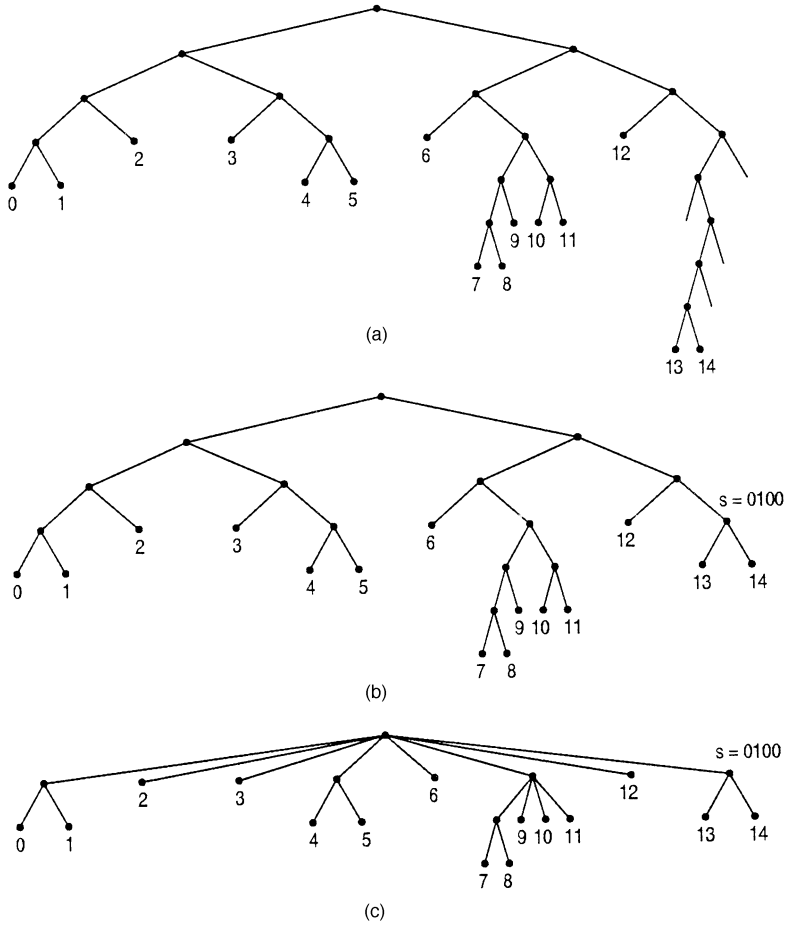

Fig. 1. (a) A binary trie; (b) a path-compressed trie; (c) a perfect LPC-trie.

tries and binary search trees. It is intended for character strings and does not support dynamic updates.

2. Compressing Binary Trie Structures. In this section we give a brief overview of binary tries and compression techniques. We start with the definition of a binary trie. We say that a string w is the *k-suffix* of the string u, if there is a string v of length k such that $u = vw$.

DEFINITION 1. A *binary trie* containing n elements is a tree with the following properties:

- If $n = 0$, the trie is empty.
- If $n = 1$, the trie is a leaf node that contains the element.

- If $n > 1$, the trie consists of an internal node with two children. The left child is a binary trie containing the 1-suffixes of all elements starting with 0 and the right child is a binary trie containing the 1-suffixes of all elements starting with 1.

Figure 1(a) depicts a binary trie storing 15 elements. In the figure the nodes storing the actual binary strings are numbered starting from 0. For example, node 14 stores a binary string whose prefix is 11101001.

We assume that all strings in a trie are prefix-free: no string can be a prefix of another. In particular, this implies that duplicate strings are not allowed. If all strings stored in the trie are unique, it is easy to ensure that the strings are prefix-free by appending a special marker at the end of each string. For example, we can append the string 1000... to the end of each string. A finite string that has been extended in this way is often referred to as a semi-infinite string or sistring.

A path-compressed binary trie is a trie where all subtries with an empty child have been removed.

DEFINITION 2. A *path-compressed binary trie* containing n elements is a tree with the following properties:

- If $n = 0$, the trie is empty.
- If $n = 1$, the trie is a leaf node that contains the element.
- If $n > 1$, the trie consists of an internal node containing two children and a binary string s of length $|s|$. This string equals the longest prefix common to all elements stored in the trie. The left child is a path-compressed binary trie containing the $(|s|+1)$-suffixes of all elements starting with $s0$ and the right child is a path-compressed binary trie containing the $(|s|+1)$-suffixes of all elements starting with $s1$.

Figure 1(b) depicts the path-compressed binary trie corresponding to the binary trie of Figure 1(a).

We use one more compression technique, level compression, which uses more than one bit for branching. We refer to this structure as a level- and path-compressed trie.

DEFINITION 3. A *level- and path-compressed trie*, or an *LPC-trie*, containing n elements is a tree with the following properties:

- If $n = 0$, the trie is empty.
- If $n = 1$, the trie is a leaf node that contains the element.
- If $n > 1$, the trie consists of an internal node containing 2^i children for some $i \geq 1$, and a binary string s of length $|s|$. This string equals the longest prefix common to all elements stored in the trie. For each binary string x of length $|x| = i$, there is a child containing the $(|s| + |x|)$-suffixes of all elements starting with sx.

A perfect LPC-trie is an LPC-trie where no empty nodes are allowed.

DEFINITION 4. A *perfect LPC-trie* is an LPC-trie with the following properties:

- The root of the trie holds 2^i subtries, where $i \geq 1$ is the maximum number for which all of the subtries are nonempty.
- Each subtrie is an LPC-trie.

Figure 1(c) provides an example of a perfect LPC-trie corresponding to the path-compressed trie in Figure 1(b). Its root is of degree 8 and it has four subtries storing more than one element: a child of degree 4 and three children of degree 2.

3. Implementation. We have implemented the LPC-trie in the Java programming language. Java is widely available, has well-defined types and semantics, offers automatic memory management, and supports object-oriented program design. The speed of a Java program is typically slower than that of a carefully implemented C program. This is mostly due to the immaturity of currently available compilers and runtime environments. We see no reason why the performance of Java programs should not be competitive in the near future.

We have separated the details of the binary string manipulation from the trie implementation by introducing an interface `SiString` that represents a semi-infinite binary string. To adapt the data structure to a new data type, we only need to write a class that implements the `SiString` interface. In our code we give three implementations, one for ASCII character strings, one for short binary strings as found in Internet routing tables, and one for points in a plane with integer coordinates. In the `SiString` interface implementation for planar points we interleave the bits of 32-bit planar coordinates to produce a one-dimensional 64-bit coordinate. Bit interleaving is extensively discussed in [25].

One of the most important design issues is how to represent the nodes of the trie. We use different classes for internal nodes and leaves. The memory layout of a leaf is straightforward. A leaf contains a reference to a key, which is a sistring, and a reference to the value associated with this key. An internal node is represented by two integers, a reference to a `SiString` and an array of references to the children of the node. Instead of explicitly storing the longest common prefix string representing a compressed path, we use a reference to a leaf in one of the subtries. We need two additional integers, `pos`, that indicates the position of the first bit used for branching, and `bits`, that gives the number of bits used. The size of the array equals $2^{\texttt{bits}}$. The number `bits` is not strictly necessary, since it can be computed as the binary logarithm of the size of the array.

Replacement of the longest common prefix string with a leaf reference saves us some memory and provides us with access to the prefix string from an internal node. This is useful during insertions and when the size of a node is increased. An alternative would be to remove the references altogether. In this way we could save some additional memory. The drawback is that insertions might become slower, since we would always need to traverse the trie all the way down to a leaf. On the other hand, a number of substring comparisons taking place in the path-compressed nodes of the trie would be replaced with a single operation finding the first conflicting bit in the leaf, which might well balance the extra cost of traversing longer paths. The doubling operation, however, would clearly be more expensive if the references were removed.

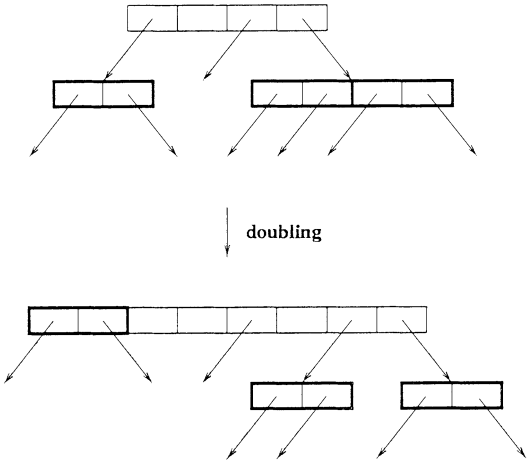

Fig. 2. Node doubling. The number of children in the root node is doubled. For each child of the original node, one of the following actions are taken. If the child is empty, a leaf, or an internal node with a nonempty path-compression string, then the subtree remains intact. Otherwise, the root node of the child is split in half.

The search operation is very simple and efficient. At each internal node, we extract from the search key the number of bits indicated by `bits` starting at position `pos`. The extracted bits are interpreted as a number that is used for indexing the child array. Note that we never inspect the longest common prefix strings during the search. It is typically more efficient to perform only one test for equality when reaching the leaf.

Insertions and deletions are also straightforward. They are performed in the same way as in a standard path-compressed trie. When inserting a new element into the trie, we either find an empty leaf where the element can be inserted or there will be a mismatch when traversing the trie. This mismatch may happen when we compare the path-compressed string in an internal node with the string to be inserted or it may occur in a leaf. In both cases we insert a new binary node with two children, of which one contains the new element and the other the previous subtrie. The only problem is that we may need to resize some of the nodes on the traversed path to retain proper level compression in the trie. We use two different node-resizing operations to achieve this: halving and doubling. Figure 2 illustrates how the doubling operation is performed and Figure 3 shows the halving.

We first discuss how to maintain a perfect LPC-trie during insertions and deletions. If a subtrie of an internal node is deleted, we need to compress the node to remove the empty subtrie. If the node is binary, it can be deleted altogether; otherwise we halve the node. Note that it may also be necessary to resize some of the children of the halved node to retain proper compression.

On the other hand, it may be possible to double the size of a node without introducing any new empty subtries. This will happen if each child of the node is full. We say that a node is *full* if it is an internal node with an empty path-compression string. Note that it may be possible to double the node more than once without introducing empty subtries.

When a node is doubled, we must split all of its full children of degree greater than 2. A split of a child node of degree 2^i leads to the creation of two new child nodes of degree

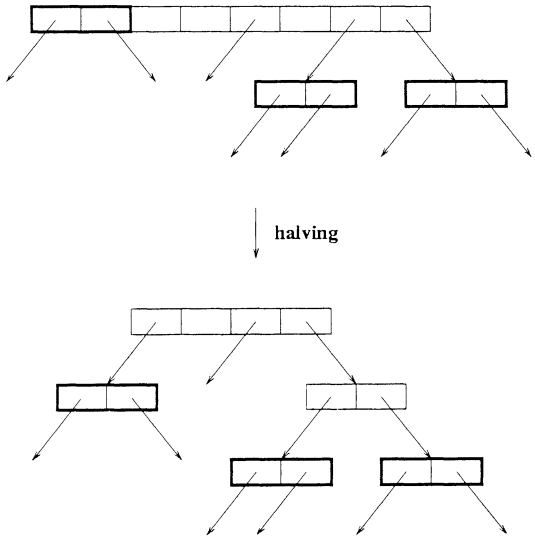

Fig. 3. Node halving. The number of children of the root node is halved. For each pair of children in the original node, one of the following actions are taken. If both children are empty, they are represented as a single empty child. If only one of the children in the pair is empty, the pair is represented by the nonempty child. If both children are nonempty, a new binary node is created to represent the pair.

2^{i-1}, one holding the 1-suffixes of all elements starting with 0 and one the 1-suffixes of all elements starting with 1. Once again, notice that it may also be necessary to resize the new children to retain the perfect level compression.

In order to efficiently determine when to resize, we use two additional numbers in each internal node. One of the numbers indicates the number of null references in the array and the other the number of full children. For full implementation details we refer the reader to the source code.

The requirement of perfect level compression may lead to very expensive update operations. Consider a trie with a root of degree 2^i. Further assume that all subtries except one contain two elements and the remaining subtrie only one. Inserting an element into the one-element subtrie will lead to a complete restructuring of the trie. Now, when removing the same key, we once again have to rebuild the trie completely. A sequence of alternating insertions and deletions of this particular key is therefore very expensive.

To reduce the risk of very expensive update operations we do not require our LPC-trie to be perfect. A node is doubled only if the resulting node has few empty children. Similarly, a node is halved only if it has a substantial number of empty children. We use two thresholds: `low` and `high`. A node is doubled if the ratio of nonempty children to all children in the *doubled* node is at least `high`. A node is halved if the ratio of nonempty children to all children in the *current* node is less than `low`. These values are determined experimentally. In our experiments we found that the thresholds 25% for `low` and 50% for `high` gave a good performance.

A relatively simple way to reduce the space requirements of the data structure is to use a different representation for internal nodes with only two children. For small nodes

we need no additional data, since it is cheap to decide when to resize the node. This will give a noticeable space reduction if there are many binary nodes in the trie. It also reduces the total amount of memory allocation, since child arrays with only two pointers no longer exist. In order to keep the code simple, we have not currently implemented this optimization.

4. Experimental Results. We have compared different compression strategies for binary tries: mere path compression, path and perfect level compression, and path and relaxed level compression. To give an indication of the performance relative to comparison-based data structures, we also implemented a randomized binary search tree, or treap [26]. Furthermore, we translated the red-black tree implementation from the textbook of Cormen et al. [7] to Java. We tried to keep true to the original implementation, only fixing some minor bugs.

A binary trie may of course hold any kind of binary data. In this study we have chosen to inspect ASCII character strings, short binary strings from Internet routing tables, and geometric point data. In addition, we evaluated the performance for uniformly distributed binary strings.

We refrained from low-level optimizations. Instead, we made an effort to make the code simple and easy to maintain and modify. Examples of possible optimizations that are likely to improve the performance on many current Java implementations include: avoiding the `instanceof` operator, performing function inlining and removing recursion, and hard coding string operations. All of these optimizations could be performed automatically by a more sophisticated Java environment.

4.1. *Method*. The speed of a program is highly dependent on the runtime environment. In particular, the performance of the insert and delete operations depends heavily on the quality of the memory management system. It is easier to predict the performance of a search, since this operation requires no memory allocation. The search time is proportional to the average depth of the structure. The timings reported in the experiments are actual clock times on a multi-user system.

When automatic memory management is used, it becomes harder to estimate the running time of an algorithm, since a large part of the running time is spent within a machine-dependent memory manager. There is clearly a need for a standard measure. A simple measure is to count the allocations of different size memory blocks. Accounting for memory blocks that are deallocated, or in the case of a garbage collected environment no longer referenced, is more difficult but clearly possible. To interpret these measures we need, of course, realistic models of automatic memory managers. We take the simple approach of counting the number of objects of different sizes allocated by the algorithm: the number of leaves, internal nodes, and arrays of child pointers. Even this basic information turned out to be useful in evaluating the performance and tuning the code.

It is somewhat difficult to measure the size of the data structure, since the internal memory requirements of references and arrays in Java are not specified in the language definition. The given numbers pertain to an implementation, where a reference is represented by a 32-bit integer, and an array by a reference to memory and an integer

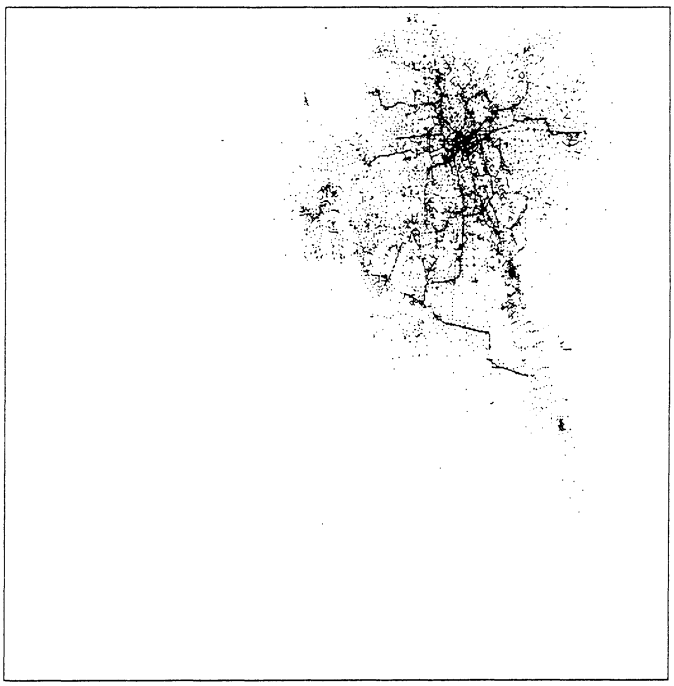

Fig. 4. Drill holes in Munich.

specifying the size of the array. We only count the size of the actual data structure, not the size of the stored elements.

We perform four different test sequences: `Put`, `Get`, `Rem`, and `Upd`. `Put` inserts all of the elements into an empty data structure. `Get` searches for each of the elements in a full data structure; that is all searches are successful. `Rem` removes the elements, one by one, from the data structure. `Upd` is a random sequence of insertions and deletions. For each operation we randomly either insert or delete an element that is chosen at random from the data set. Before starting the measurement we perform 1,000,000 random updates starting with an empty data structure. The measurement is performed on 100,000 update operations. For all of these sequences we measure the elapsed time. For `Put`, `Rem`, and `Upd` we also measure the number of memory allocations. `Get` does not allocate any memory.

We used the JDK (Java Development Kit) version 1.1.5 compiler from SUN to compile the program into byte code. The experiments were run on a SUN Ultra Sparc II with two 296-MHz processors and 512 MB of RAM. We used the JDK 1.1.5 runtime environment with default settings. The test data consisted of binary strings from Internet routing tables, ASCII strings from the Calgary Text Compression Corpus, a standardized text corpus frequently used in data compression research, and geometric point data [27] as depicted in Figure 4.

4.2. *Discussion*. Table 1 shows the average and maximum depths and the sizes of the data structures tested. We also give timings for inserting all of the elements (`Put`),

Table 1. Experimental results. The size figures in parentheses refer to a more compact trie representation.

	Depth		Put	Get	Rem	Upd	Size
	Aver.	Max.	(sec)	(sec)	(sec)	(sec)	(kB)
RANDOM, bit strings (50,000 unique entries)							
Treap	18.6	40	4.0	2.6	2.9	7.7	1000
RedBlack tree	14.0	18	3.3	1.6	1.9	4.9	1200
Path-compressed trie	15.9	20	9.9	2.6	6.0	14.3	1199
Perfect LPC	3.7	8	5.6	0.8	4.0	6.1	1674 (1101)
Relaxed LPC (50/75)	2.0	5	5.1	0.4	1.7	3.6	1276 (966)
Relaxed LPC (25/75)	2.0	5	5.6	0.6	1.1	3.0	1276 (966)
Relaxed LPC (25/50)	1.6	4	4.3	0.4	1.1	2.8	1155 (930)
MAE-EAST, routing table (38,470 entries, 38,367 unique entries)							
Treap	17.9	34	2.4	1.4	1.8	7.3	767
RedBlack tree	14.8	26	2.5	1.1	1.1	4.3	921
Path-compressed trie	18.6	24	6.7	2.0	4.5	16.1	921
Perfect LPC	5.8	13	4.6	0.8	4.6	8.3	1264 (844)
Relaxed LPC (50/75)	3.7	7	5.7	0.5	3.0	5.0	1031 (834)
Relaxed LPC (25/75)	3.7	7	5.6	0.5	1.6	4.9	1031 (834)
Relaxed LPC (25/50)	2.9	5	4.8	0.4	2.3	4.1	978 (843)
DRILL HOLES, point data (19,461 unique entries)							
Treap	16.7	32	1.0	0.7	0.8	6.4	389
RedBlack tree	12.8	17	0.8	0.6	0.6	4.2	467
Path-compressed trie	20.8	30	3.7	1.0	2.5	17.2	467
Perfect LPC	9.9	17	3.0	0.5	2.2	12.2	694 (442)
Relaxed LPC (50/75)	6.8	11	2.4	0.4	1.6	8.0	582 (452)
Relaxed LPC (25/75)	6.0	12	2.9	0.4	1.2	7.9	582 (452)
Relaxed LPC (25/50)	5.2	9	2.3	0.3	1.7	6.5	549 (445)
BOOK1, text (16,622 lines, 16,542 unique entries, 768,770 characters)							
Treap	16.7	32	1.4	1.3	1.3	9.5	330
RedBlack tree	12.7	17	1.1	1.0	1.0	6.9	397
Path-compressed trie	20.2	41	4.0	1.6	2.8	21.2	396
Perfect LPC	14.3	28	3.5	1.0	2.3	16.9	674 (391)
Relaxed LPC (50/75)	10.4	23	2.7	0.9	1.7	13.8	610 (412)
Relaxed LPC (25/75)	10.4	23	3.1	0.9	1.7	13.2	610 (412)
Relaxed LPC (25/50)	9.0	18	2.5	0.8	1.5	11.9	585 (401)

retrieving them (`Get`), deleting them one by one (`Rem`), and performing a random sequence of 100,000 updates (`Upd`). The timings should be carefully interpreted since the update operations, in particular, depend very much on the memory management system.

Starting with the two comparison-based data structures, the treap and the red-black tree, we note that the performance is similar. The red-black tree has a slightly lower average depth and, typically, a much lower maximum depth. Note that the slightly better performance of red-black trees cannot be completely explained by the difference in average depth. It is probably due to the iterative implementation of the red-black tree and its more aggressive use of code inlining. We conclude that even in a language like

Java with relatively costly method calls the gains obtained by recursion elimination and inlining are rather small. The red-black tree implementation uses parent pointers which explains its larger size compared with the treap.

The path-compressed trie is not an attractive alternative to a binary search tree. Even though the average depth and the size of the trie are comparable with those of the binary search trees, the execution times are markedly slower. This is because we need to allocate new memory when the trie is restructured during insertions and deletions. Also the searches are slower. Hence we may conclude that the bit extraction and comparison in an internal trie node is more expensive than the plain comparison in a binary search tree node. It should be noted, however, that our implementation of a path-compressed trie has some unnecessary overhead, since we have implemented it as a special case of the level-compressed trie. Thus, we perform some bookkeeping activities that are not necessary in a trie without level compression. Finally, note that the size of the path-compressed trie is a simple function of the number of elements n, since the number of leaves is n and number of internal nodes is $n - 1$. In our implementation the size of the path-compressed trie and the red-black tree are, in fact, equal.

We study three variants of the relaxed LPC-trie. The first parameter `low` indicates the upper bound of the ratio of null pointers to all pointers in the *current* node and the second parameter `high` the lower bound of the ratio of non-null pointers to all pointers in the *doubled* node.

In general, the LPC-trie behaves best for uniformly distributed data, but even for English text the performance is good. Level compression leads to a significant reduction of the average path length. Even in the least favorable case, the text, the size reduction compared with a path-compressed trie is more than a factor two. For the search operation this reduction in average depth is enough to make the LPC-trie the fastest, even for text data. For the update operations the additional cost of restructuring the trie is competitive with the binary search trees only for data with a high degree of randomness.

When comparing the different variants of the LPC-trie, we note that the differences in time correlate well with the differences in average depth. There are some noticeable exceptions, however. For example, the `Rem` sequence for the mae-east routing table is performed much faster by the 25/75 variant than by the other LPC-trie variants. The 25/75 variant performs much less memory allocation than the other variants, as can be seen in Figure 5. However, in general the LPC-tries are robust. Low–high tolerance between 25/50 and 50/75 works well. The key point is to provide some slack.

Figure 6 shows memory profiles of the different update sequences for English text. The amount of memory allocation needed to maintain the level compression is very small: the number of internal nodes allocated only slightly exceeds the number of leaves. However, we see that the algorithm frequently allocates arrays containing only two elements. To create a binary node, two memory allocations are needed: one to create the object itself and one to create the array of children. If we used a different memory layout for binary nodes we could reduce the number of array allocations considerably. For deletions, very little memory management is needed. Comparing with Table 1 we may conclude that the level compression reduces the average depth of the trie structure from 20 to 9 using very little restructuring.

For the routing table data and the geometric data the situation is different. In Figures 5 and 7 we see that the number of internal node and array allocations clearly

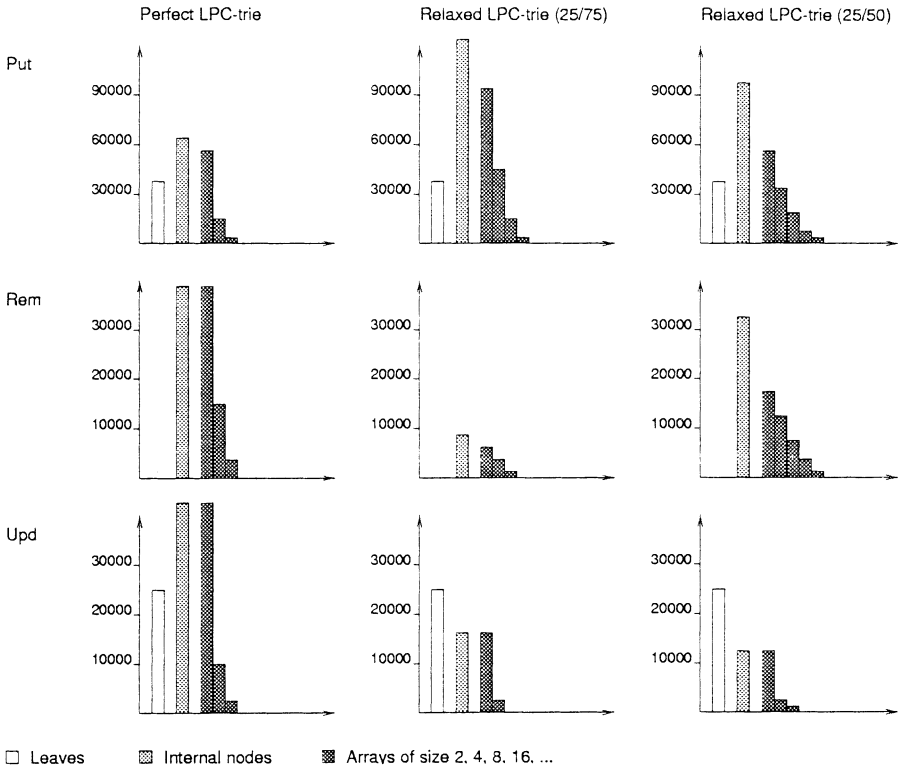

Fig. 5. Memory profiles for an mae-east routing table (38,470 entries).

exceeds the number of leaves. However, the extra work spent in restructuring the trie pays off. For example, in Table 1 the average depth of the routing table stored in a path-compressed trie is 18, the corresponding perfect LPC-trie has depth 6, and the relaxed LPC-trie depth 3. In this particular Java environment this reduction in depth is enough to compensate for the extra restructuring cost. The insertions times are, in fact, slightly faster for the level-compressed tries, compared with a tree using only path compression. We also note that the naive implementation of the doubling and halving algorithms results in more memory allocation than would be strictly necessary. Thus, it should be possible to improve running time by coding these operations more carefully.

In our implementation, the size of the trie structure is larger than the size of a corresponding binary search tree. However, by using a more compact memory representation for binary nodes as discussed in Section 3 we could achieve memory usage very similar to the treap. This optimization will also eliminate a large part of the memory allocation, since we no longer need to allocate arrays containing only two elements. As can be seen in the memory profiles, this type of allocation is rather frequent.

There are many other possible optimizations. For data with a very skewed distribution such as the English text, one might introduce a preprocessing step, where the strings are

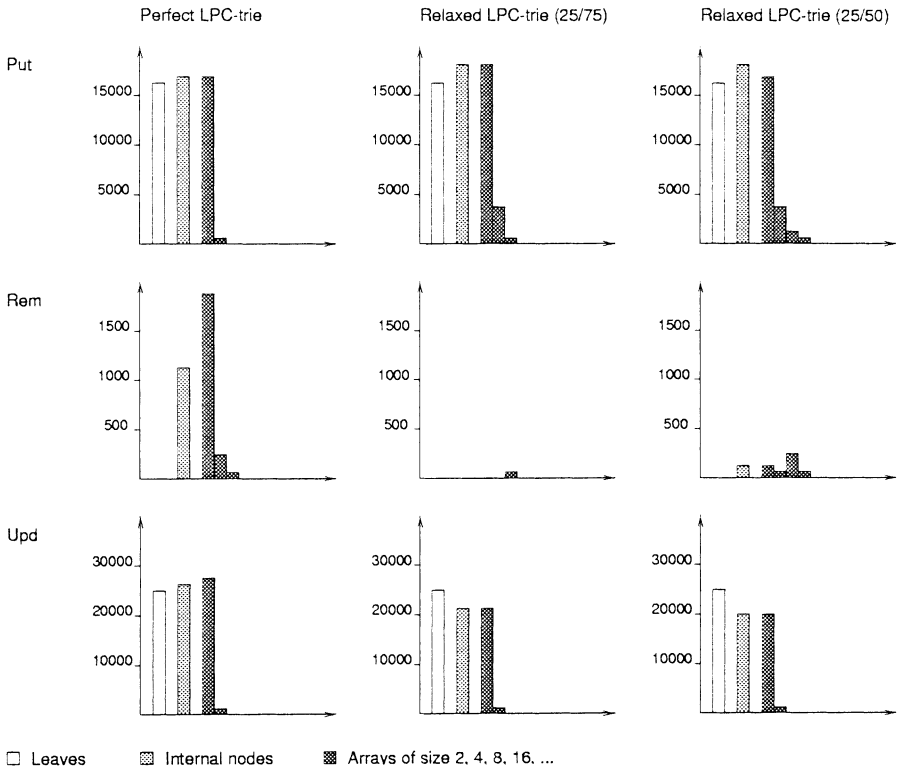

Fig. 6. Memory profiles for book1 (16,622 entries).

compressed, resulting in a more even distribution [4]. For example, order-preserving Huffman coding could be used.

5. Conclusions and Further Research. For both integers and text strings, the average depth of the LPC-trie is much less than that of the balanced binary search tree, resulting in better search times. In our experiments, the time to perform the update operations was similar to the binary search tree. Our LPC-trie implementation relies heavily on automatic memory management. Therefore, we expect the performance to improve when more advanced Java runtime environments become available. The space requirements of the LPC-trie are also similar to the binary search tree. We believe that the LPC-trie is a good choice for an order-preserving data structure when very fast search operations are required.

The standard measure of performance in theoretical studies of trie structures, the average depth, explains much of the observed performance in our experiments. The second most important factor seems to be memory allocation. The cases where we have noticeable differences in time and average depth can be explained by large differences in the amount of memory allocations. Additional factors that are often mentioned as

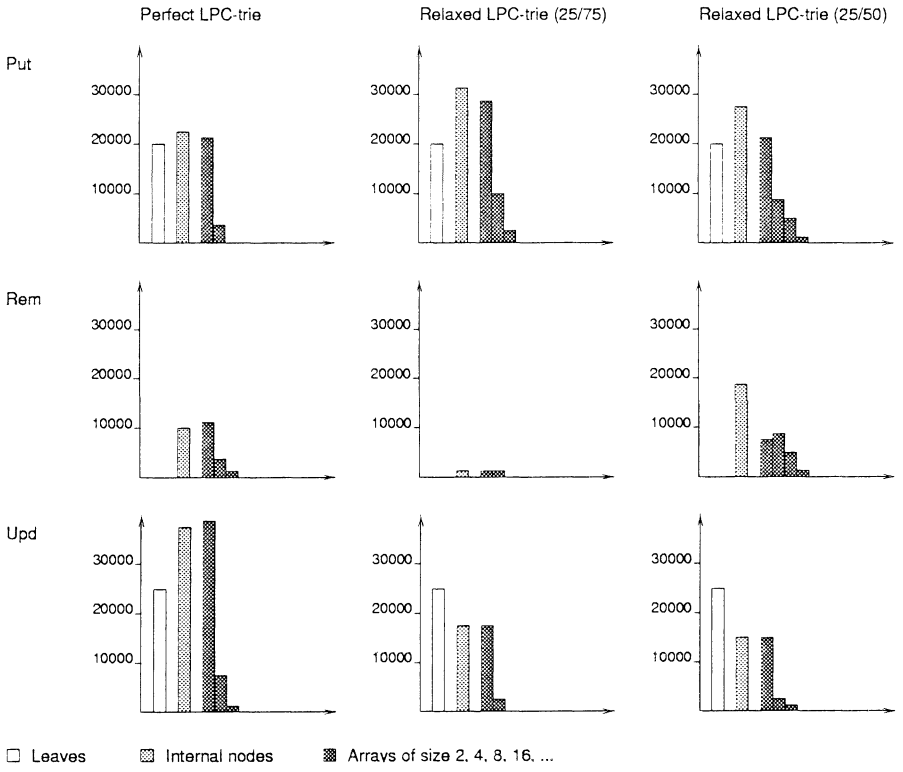

Fig. 7. Memory profiles for drill holes in Munich (19,461 entries).

bottlenecks in object-oriented languages such as the overhead for dispatched method calls seemed to be less important.

The experiments indicate that the data structure is efficient and robust. However, both doubling and halving may introduce new child nodes that also need to be resized. This might lead to poor performance for some data distributions and update sequences. An interesting topic for further research is to find amortized bounds for a sequence of update operations. To achieve this we believe that it may be necessary either to restrict the scope of the resizing operations or to permit resizing to occur also during searches.

Another interesting topic for further research is to compare the LPC-trie with the ternary search trees [6]. This data structure is static, however. To make a thorough comparison the ternary search tree must be extended with insertions and deletions.

Acknowledgments. We thank Petri Mäenpää, Ken Rimey, Eljas Soisalon-Soininen, and Peter Widmayer for comments on an earlier draft of this paper. We also thank the anonymous referees for helpful comments.

Tikkanen's work has been carried out in the HiBase project that is a joint research project of Nokia Networks and the Helsinki University of Technology. The HiBase

project has been financially supported by the Technology Development Centre of Finland (Tekes). Nilsson's work was carried out while he was with Helsinki University of Technology.

References

[1] A. Analyti, S. Pramanik. Fast search in main memory databases. *SIGMOD Record*, 21(2): 215–224, June 1992.

[2] A. Andersson, S. Nilsson. Improved behaviour of tries by adaptive branching. *Information Processing Letters*, 46(6):295–300, 1993.

[3] A. Andersson, S. Nilsson. Faster searching in tries and quadtrees—an analysis of level compression. In *Proceedings of the Second Annual European Symposium on Algorithms*, pages 82–93. Lecture Notes in Computer Science, vol. 855. Springer-Verlag, Berlin, 1994.

[4] A. Andersson, S. Nilsson. Efficient implementation of suffix trees. *Software – Practice and Experience*, 25(2):129–141, 1995.

[5] J.-I. Aoe, K. Morimoto. An efficient implementation of trie structures. *Software – Practice and Experience*, 22(9):695–721, 1992.

[6] J. L. Bentley, R. Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 360–369, 1997.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest. *Introduction to Algorithms*. McGraw-Hill, New York, 1990.

[8] J. J. Darragh, J. G. Cleary, I. H. Witten. Bonsai: a compact representation of trees. *Software—Practice and Experience*, 23(3):277–291, 1993.

[9] L. Devroye. A note on the average depth of tries. *Computing*, 28(4):367–371, 1982.

[10] J. A. Dundas III. Implementing dynamic minimal-prefix tries. *Software—Practice and Experience*, 21(10):1027–1040, 1991.

[11] P. Flajolet. On the performance evaluation of extendible hashing and trie searching. *Acta Informatica*, 20:345–369, 1983.

[12] P. Flajolet. Digital search trees revisited. *SIAM Journal on Computing*, 15(3):748–767, 1986.

[13] P. Flajolet, M. Régnier, D. Sotteau. Algebraic methods for trie statistics. *Annals of Discrete Mathematics*, 25:145–188, 1985.

[14] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–500, 1960.

[15] G. H. Gonnet, R. A. Baeza-Yates. *Handbook of Algorithms and Data Structures*, second edition. Addison-Wesley, Reading, MA, 1991.

[16] P. Kirschenhofer, H. Prodinger. Some further results on digital search trees. In *Proceedings of 13th ICALP*, pages 177–185. Lecture Notes in Computer Science, vol. 26. Springer-Verlag, Berlin, 1986.

[17] D. E. Knuth. *TEX: The Program*. Addison-Wesley, Reading, MA, 1986.

[18] D. E. Knuth. *The Art of Computer Programming*, vol. 3. Addison-Wesley, Reading, MA, 1973.

[19] P.-A. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4): 446–457, 1988.

[20] J. van Leeuwen. *Handbook of Computer Science*, vol. A. Elsevier, Amsterdam, 1990.

[21] K. Morimoto, H. Iriguchi, J.-I. Aoe. A method of compressing trie structures. *Software – Practice and Experience*, 24(3):265–288, 1994.

[22] S. Nilsson, G. Karlsson. Fast address lookup for Internet routers. In *Proceedings of the 4th IFIP International Conference on Broadband Communications* (*BC* '98). Chapman & Hall, London, 1998.

[23] B. Pittel. Asymptotical growth of a class of random trees. *The Annals of Probability*, 13(2):414–427, 1985.

[24] B. Pittel. Paths in a random digital tree: limiting distributions. *Advances in Applied Probability*, 18:139–155, 1986.

[25] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, Reading, MA, 1989.

[26] R. Seidel, C. R. Aragon. Randomized binary search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[27] F. Wagner, A. Wolff. A combinatorial framework for map labeling. In *Proceedings of the Symposium on Graph Drawing* '98. Springer-Verlag, Berlin, 1998.