Proceedings WAE'98, Saarbrücken, Germany, August 20–22, 1998
Ed. Kurt Mehlhorn, pp. 25–36

Implementing a dynamic compressed trie

Stefan Nilsson
Helsinki University of Technology
e-mail: Stefan.Nilsson@hut.fi

and

Matti Tikkanen
Nokia Telecommunications
e-mail: Matti.Tikkanen@ntc.nokia.com

ABSTRACT

We present an order-preserving general purpose data structure for binary data, the LPC-trie. The structure is a highly 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 to a balanced binary search tree.

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 implementations of dynamic trie structures in the literature [5, 6, 8, 17, 23]. One of the drawbacks of these methods is that they need considerably more memory than a balanced binary search tree. We avoid this problem by compressing the trie. In fact, the LPC-trie can be implemented using the same amount of memory as a balanced binary search tree.

In spatial applications trie-based data structures such as the quadtree and the octree are extensively used [30]. To reduce the number of disk accesses in a secondary storage environment, dynamic order-preserving data structures based on extendible hashing or linear hashing have been introduced. Lomet [22], Tamminen [33], Nievergelt et al. [24], Otoo [26, 27], Whang and Khrishnamurthy [35], Freeston [13], and Seeger and Kriegel [31] describe data structures based on extendible hashing. Kriegel and Seeger [19] and Hutflesz et al. [15] describe data structures based on linear hashing. All of the data structures mentioned above have been designed for a secondary storage environment, but similar structures have also been introduced for main memory. Analyti and Pramanik [1] describe an extendible hashing based main memory structure, and Larson [20] a linear hashing based one.

In its original form the trie [12, 14] is a data structure where a set of strings from an alphabet containing m characters is stored in a 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 of varying size. Using a binary alphabet tends to increase the depth of the trie when compared to 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 [11, 18, 28, 29]; an extensive list of references can be found in Handbook of Theoretical Computer

Science [21]. The expected average depth of a trie containing n independent random strings from a distribution with density function $f \in L^2$ is $\Theta(\log n)$ [7]. This result holds also for data from a Bernoulli-type process [9, 10].

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 1b. 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 1c. 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 to 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 [25].

One of the difficulties when implementing a dynamic compressed trie structure is that a single update operation might 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. The price we have to pay is the potentially increasing storage requirements. However, it is possible to get the beneficial effects using only a very small amount of additional memory.

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 *i-suffix* of the string u, if there is a string v of length i such that $u = vw$.

Definition 2.1. *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 consists of a node that contains the element.*
- *If $n > 1$, the trie consists of a 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 1a 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 insure 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.

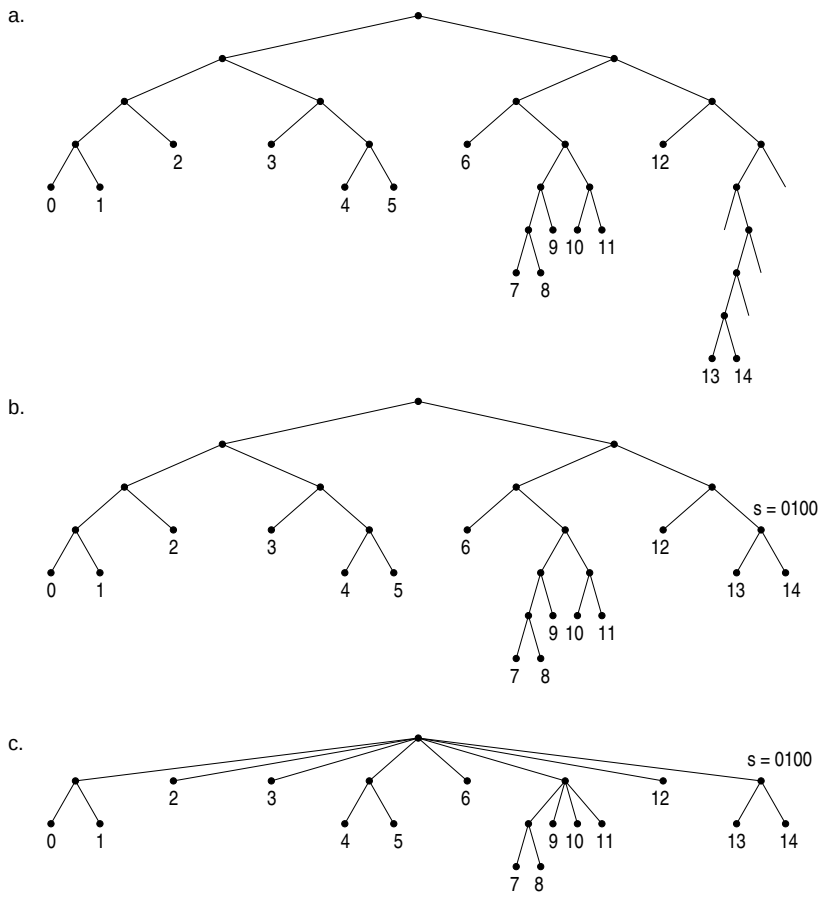

Figure 1: (a) A binary trie; (b) a path compressed trie; (c) a perfect LPC-trie.

Definition 2.2. *A* path compressed binary trie*, or Patricia trie, containing n elements is a tree with the following properties:*

- *If $n = 0$, the trie is empty.*
- *If $n = 1$, the trie consists of a node that contains the element.*
- *If $n > 1$, the trie consists of a 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 1b depicts the path compressed binary trie corresponding to the binary trie of Figure 1a. A natural extension of the path compressed trie is to use more than one bit for branching. We refer to this structure as a level and path compressed trie.

Definition 2.3. *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 consists of a node that contains the element.*
- *If $n > 1$, the trie consists of a 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 2.4. *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 non-empty.*
- *Each subtrie is an LPC-trie.*

Figure 1c provides an example of a perfect LPC-trie corresponding to the path compressed trie in Figure 1b. 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 an 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. Currently, 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 two implementations, one for ASCII character strings and one for short binary strings as found in Internet routers.

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 connected 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.

The replacement of longest common prefix strings with leaf references saves us some memory while providing us access to the prefix strings from internal nodes. 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 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.

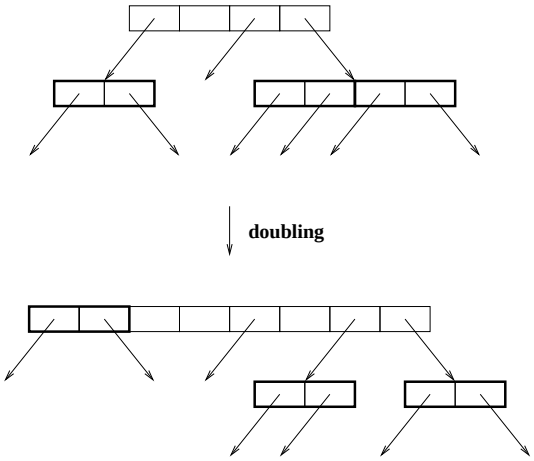

Figure 2: Node doubling.

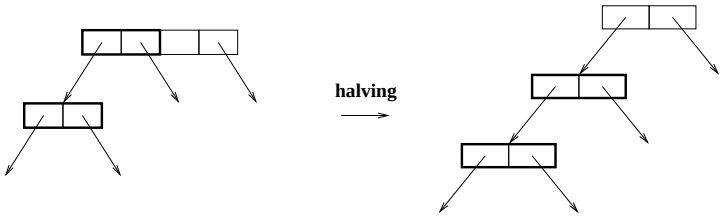

Figure 3: Node halving.

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 and this number is used as an index in the child array. Note that we do not 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 Patricia 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 might happen when we compare the path compressed string in an internal node with the string to be inserted or it might occur in a leaf. In both cases we insert a new binary node with two children, one contains the new element and the other contains 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

has at least two children and an empty path compression string. Note that it may be possible to perform the doubling operation more than once without introducing empty subtries.

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

In order to efficiently check if resizing is needed, 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.

If we require that the trie is perfectly level compressed, we might get 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 would result in a complete restructuring of the trie. Now, when removing the same key, we once again have to completely rebuild the trie. A sequence of alternating insertions and deletions of this particular key is therefore very expensive.

To reduce the risk of a scenario like this 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 non-empty children to all children in the *doubled* node is at least `high`. A node is halved if the ratio of non-empty 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` give 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. In order to keep the code clean, 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 [32]. A study of different balanced binary search trees is beyond the scope of this article. We just note that the update operations of a treap are very simple and that the expected average path for a successful search is relatively short, approximately $1.4 \log_2 n - 1.8$.

A binary trie may of course hold any kind of binary data. In this study, we have chosen to inspect ASCII character strings and short binary strings from Internet routing tables. 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 synchronized method calls, avoiding the `instanceof` operator, performing function inlining and removing recursion, performing explicit memory management, for example by reusing objects, and hard coding string operations. All of these optimizations could be performed by a more sophisticated Java environment.

4.1. Method

The speed of the 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 would be to count the allocations of different size memory blocks. This measure could be estimated both analytically and experimentally. 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. That is, the number of leaves, internal nodes, and arrays of children pointers. Even this crude information turned out to be useful in evaluating the performance and tuning the code.

It is also a bit tricky 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 specifying the size of the array.

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 and ASCII strings from the Calgary Text Compression Corpus, a standardized text corpus frequently used in data compression research. The code and test data are available at URL `http://www.cs.hut.fi/~sni`

4.2. Discussion

Table 1 shows the average and maximum depths and the size of the data structures tested. We also give timings for inserting all of the elements (Put), retrieving them (Get), and deleting them one by one (Remove). The timings should be carefully interpreted, however, because the insertion and deletion times in particular depend very much on the implementation of the memory management system.

We use two variants of the relaxed LPC-trie. In the first variant, the `low` value 50% indicates the upper bound of the ratio of null pointers to all pointers in the *current* node and the `high` value 75% the lower bound of the ratio of non-null pointers to all pointers in the *doubled* node. In the second variant, the `low` and `high` values are 25% and 50%, respectively. Note that in all test cases the second variant outperforms the first one for our test data, even though the second variant has a poorer fill ratio. There is an interesting tradeoff between the number of null pointers and the number of internal nodes.

The trie behaves best for uniformly distributed data, but even for English text the performance is satisfactory. Interestingly, our experimental results agree with theoretical results on the expected number of levels for multi-level extendible hashing with uniformly distributed keys [34]. Level compression leads to a significant reduction of the average path length. The path compressed (Patricia) trie does not offer any improvement over the treap for our test data.

Figure 4 shows memory profiles for a sequence of insert and delete operations 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. This comes from the fact that 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 would reduce the number of allocated arrays 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.

book1 (16622 lines, 16542 unique entries, 768770 characters)

	Depth		Put	Get	Remove	Size	
	Aver	Max	(sec)	(sec)	(sec)	(kB)	
Treap	16.2	30	1.5	1.3	1.3	323	
Patricia	20.2	41	3.9	1.3	2.6	388	
Perfect LPC	14.3	28	3.3	1.1	2.1	658	(382)
Relaxed LPC (50/75)	10.4	23	2.6	0.9	1.7	596	(403)
Relaxed LPC (25/50)	9.0	18	2.4	0.7	1.5	571	(391)

uniform random (50000 unique entries)

	Depth		Put	Get	Remove	Size	
	Aver	Max	(sec)	(sec)	(sec)	(kB)	
Treap	18.5	34	3.4	2.5	2.7	977	
Patricia	16.0	20	9.6	2.0	5.5	1171	
Perfect LPC	3.7	8	4.3	0.7	3.6	1635	(1076)
Relaxed LPC (50/75)	2.0	5	2.9	0.5	1.5	1246	(943)
Relaxed LPC (25/50)	1.6	4	2.1	0.5	1.2	1128	(908)

mae-east routing table (38470 entries, 38367 unique entries)

	Depth		Put	Get	Remove	Size	
	Aver	Max	(sec)	(sec)	(sec)	(kB)	
Treap	17.5	32	2.1	1.3	1.6	749	
Patricia	18.6	24	4.9	1.6	4.1	899	
Perfect LPC	5.8	13	4.0	0.7	3.3	1235	(825)
Relaxed LPC (50/75)	3.7	7	5.3	0.5	2.5	1006	(814)
Relaxed LPC (25/50)	2.9	5	4.5	0.4	2.1	955	(823)

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

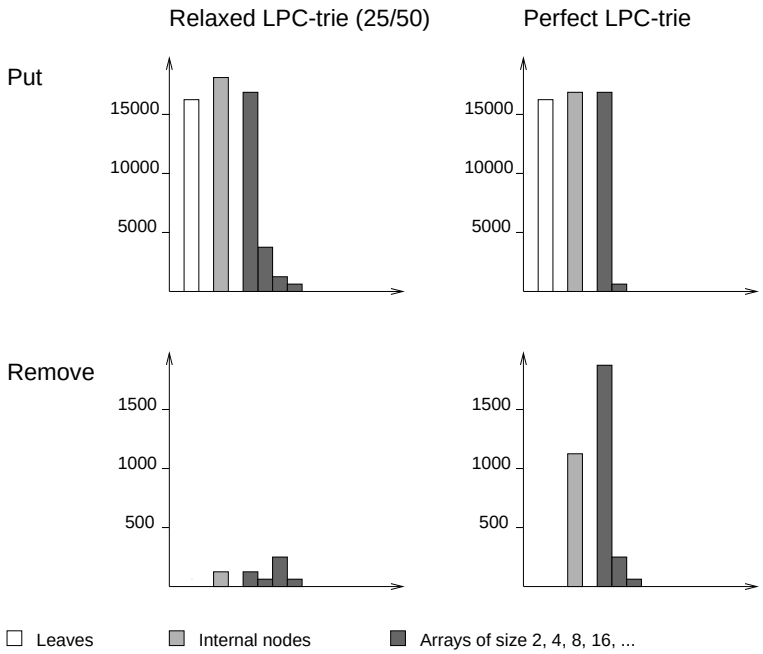

Figure 4: Memory profiles for book1 (16622 entries).

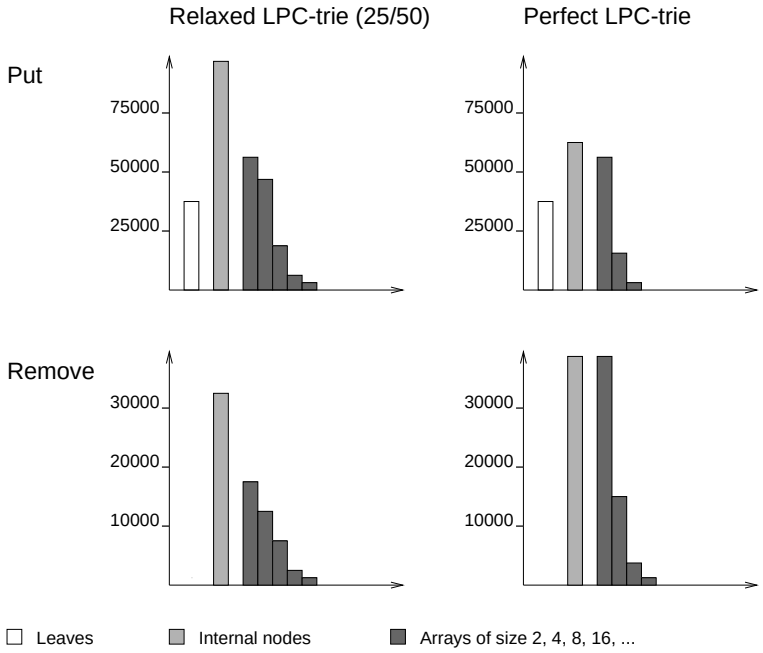

Figure 5: Memory profiles for mae-east (38470 entries).

For the routing table data the situation is different. In Figure 5 we see that the number of internal nodes and arrays allocated clearly exceeds the number of leaves. However, the extra work spent in restructuring the trie pays off. As can bee seen in Table 1, the average depth for of the Patricia tree is 18, the 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 as compared to 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 and there seems to be room for improvement in 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, using a more compact memory representation for binary nodes as discussed in Section 2 would give memory usage very similar to the treap. There are many other possible further optimizations. For data with a very skewed distribution such as the English text, one might introduce a preprocessing step, where the strings are 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 mature 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.

Further research is still needed to find an efficient node resizing strategy. Doubling and halving may introduce new child nodes that also need to be resized. In the current implementation, we recursively resize all of the new child nodes that fulfill the resizing condition. This may become too expensive for some distributions of data, because several subtries may have to be recursively resized. One way to avoid too expensive resizing operations is to delimit resizing to the nodes that lie along the search path of the update operation. This will make searches more expensive, but in an update intensive environment this might be the right thing to do. It is also possible to permit resizing to occur during searches in order to distribute the cost of resizing more evenly among operations.

Automatic memory management supported by modern programming language environments frees the application programmer from low level details in the design of algorithms that rely heavily upon dynamic memory. When the run time environment is responsible for memory management, it is possible to tailor and optimize the memory manager to take full advantage of the underlying machine architecture. This makes it possible to implement algorithms efficiently without explicit knowledge of the particular memory architecture. On the other hand, it is more difficult to take advantage of the fact that many algorithms need only a limited form of memory management that could be implemented more efficiently than by using a general purpose automatic memory manager. It also becomes more difficult to benchmark algorithms, when a large part of the run time is spent within the memory manager. There is clearly a need for a standardized measure to account for the cost of automatic memory management. A very interesting project would be to collect memory allocation and deallocation profiles for important algorithms and create performance models for different automatic memory management schemes. This should be of interest both to designers of general purpose data structures and automatic memory management schemes.

Acknowledgements

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

Tikkanen's work has been carried out in the HiBase project that is a joint research project of Nokia Telecommunications and Helsinki University of Technology. The HiBase project has been financially supported by the Technology Development Centre of Finland (Tekes).

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, 1994. LNCS 855.

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

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

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

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

[10] P. Flajolet, M. Régnier, D. Sotteau. Algebraic methods for trie statistics. *Ann. Discrete Math.*, 25:145–188, 1985.

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

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

[13] M. Freeston. The BANG file: a new kind of grid file. *ACM SIGMOD Int. Conf. on Management of Data*, 260–269, 1987.

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

[15] A. Hutflesz, H.-W. Six, P. Widmayer. Globally order preserving multidimensional linear hashing. *Proc of the 4th Int. Conf. on Data Engineering*, 572–587, 1988.

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

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

[18] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, 1973.

[19] H.-P. Kriegel, B. Seeger. Multidimensional Order Preserving Linear Hashing with Partial Expansions, *Proceedings of International Conference on Database Theory (Lecture Notes in Computer Science)*, Springer Verlag, Berlin, 203–220, 1986.

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

[21] J. van Leeuwen. *Algorithms and Complexity*, volume A of *Handbook of Computer Science*. Elsevier, 1990.

[22] D.B. Lomet. Digital B-trees. *Proc of the 7th Int. Conf. on Very Large Databases, IEEE*, 333–344, 1981.

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

[24] J. Nievergelt, H. Hinterberger, K.C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. *ACM TODS*, 9(1), 38–71, 1984.

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

[26] E.J. Otoo. A mapping function for the directory of a multidimensional extendible hashing. *Proc of the 10th Int. Conf. on Very Large Databases*, 491–506, 1984.

[27] E.J. Otoo. Balanced multidimensional extendible hash tree. *Proc 5th ACM SIGACT-SIGMOD Symposium on the Principles of Databases*, 491–506, 1985.

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

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

[30] H. Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1989.

[31] B. Seeger, H.P. Kriegel. The buddy-tree: an efficient and robust access method for spatial database systems. *Proc of the 16th Int. Conf. on Very Large Databases*, 590–601, 1990.

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

[33] M. Tamminen. The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica*. Mathematics and Computer Science Series No. 34, Helsinki, Finland, 1981.

[34] M. Tamminen. Two levels as good as any. *Journal of Algorithms*. 6(1):138–144, 1985.

[35] K.-Y. Whang, R. Krishnamurthy. *Multilevel grid files*. Research Report RC 11516 (#51719), IBM Thomas J. Watson Research Center, 43, 1985.