Bull. Nov. Comp. Center, Comp. Science, 10 (1999), 73–82
© 1999 NCC Publisher

Expulsive tree data structures for fast data search by a key

Mike Ostapkevich

We intend to draw a comparison between the different most known data indexing structures so as to outline the bottlenecks and faults from the point of view of their utilization in the fine grained algorithms simulating system WinALT [1]. Then we would like to propose some ways to eliminate these drawbacks and to give a description of a tree structure with both satisfactory memory consumption and acceptable access time for the considered type of simulating systems. The proposed data structure is called expulsive tree.

1. Introduction

The significance of data structure design can hardly be overestimated. An efficient implementation of any project is impossible without adequate data structures developed for the project. This is a well-known fact, it is reflected in the evolution of software design paradigms. The Johnson approach requires that the design ought to be started with the design of data structures. Object oriented approach [2] is more data oriented than its ancestors: modular programming and structured programming [3].

The design of data structures in a commonplace project is not done just from the scratch, it considerably relies upon a certain set of typical well-known data structures, such as contagious arrays, lists (unidirectional, bidirectional), hash tables, trees and graphs of more universal structure. Improvements of these structures may positively influence a wide set of applications. The design of some structures is oriented to the operation of data element search by a key.

The goals of this paper are as follows:

1) to investigate quite an important and widely used subclass of data structures intended for a data element search by a unique key and associated with the data element from the point of view of their utilization in the fine-grained algorithms simulating system;
2) to propose some ways to eliminate these drawbacks and to give a description of a so called expulsive tree structure with both satisfactory memory consumption and acceptable access time for the considered type of simulating systems.

The data structures for search by index are widely used in simulating systems as WinALT for a number of purposes. Two most important of them are: translation of symbolic names into handles for objects and sparse data arrays representation.

From the point of view of a fine-grained algorithm simulating system specifications, several things are vitally important. First, a simulating system ought to implement the fastest simulation possible. Second, it should be capable to represent objects with large dimensions to be able to simulate real-world problems. Last but not least, it must be capable to represent a multitude of small objects without huge memory overheads. These demands lead to the following criteria which reflect the requirements to the data structures used for the considered type of systems.

Two criteria of comparison are memory consumption and access time. The memory consumption is regarded from two principle points of view. First, we would like to minimize the average size of memory per one data element (when number of data elements unlimitedly increases). Second, we have an intention to minimize the total data structure memory size when the number of elements is very small (0 or 1). This demand is not nonsense for a simulating system (though it is not typical when we examine data structures), because in such a type of applications it is highly desirable to avoid limitations of the data structures number. We might want to create a huge number of them. Of course, in a commonplace case only few of them will grow strongly, others remain small. But we do not know which one of them will grow and at what rate. In a simulating system it depends on the modeling problem rather than on the system itself.

Only the simplistic and pure forms shall be considered in the paper. That means that such combinations as hash tables of lists or those of binary trees are beyond the scope of the article.

The notions used are as follows:

- data element is an element of a set or database that we need to keep.
- data item is a data of non-atomic type [4] that represents a data element in an indexing data structure. Normally that is a pair of data element and its key;
- indexing data structure is a set of data items that has operation for data element insertion, search and deletion;
- key is a value of atomic type that is a part of each data item. Key is unique among all the items in a set (or in one data structure) and is used to distinguish data elements. In the common case, there might be many keys (up to the number of properties in the type of data element) for the same data element, but here we tackle only the simplest case when there is only one such a key.

2. Well-known typical data structures

The purpose of this section is neither to give a brief introduction into the data structures, nor to present their detailed description, as both these tasks were accomplished decades ago [5–8]. Our intention here is just to mark some of the drawbacks for these data structures from the point of view of a fine-grained simulating system design, to make an effort in the next section to overcome partially some of them.

2.1. List (in the narrow sense of the term) is the simplest data structure. It is not used for data search because of the unacceptable access time. We would like to mention only that a list offers the best memory consumption among all the dynamically allocated data structures.

2.2. Hash table provides fast access to data, but the access time augments rapidly when more than 70 percent of positions in the table are occupied [6]. The same position in the table corresponds to more than one key value. The more positions are occupied, the greater possibility that a required cell is already occupied by another data item with another key value. Thus a collision must be resolved, and this takes more time.

This drawback results in a very undesirable feature that a hash table has. We need to know *a priori* the range of item quantity to exploit hash table efficiently. If we do not estimate it correctly, we will either create a memory consuming (many unused positions in the table) or time consuming (many collisions happen at access operations).

Of course, in the case when we do have such an estimation before execution (e.g., as an accumulated statistics of previously done executions) and when the distribution of the quantity has small dispersion, the hash table is usually the best choice. Many implementations of system level software modules rely upon hashes to minimize performance overhead [9].

In a simulating system a preliminary estimation is impossible, as most of data element numbers and their sizes are not static, as they depend on the simulating model, which in its turn often has a dynamic and unpredictable behaviour. Thus, a hash table, at, least in its most simplistic form, cannot be used in the considered type of systems.

2.3. Binary tree derives its name from the graph theory. It is represented by a set of interconnected nodes. A graph's arc denotes a connection. A node may have two types of connections: connections to parents (an input arc for a node) and connections to children (an output arc). All the nodes but one must have one parent (otherwise a tree graph turns into a forest). The node without a parent is called root node. There are two types of children: a left child and a right child. A node may have zero or one child of each type. A node that has no children is called a leaf or terminal node.

A typical node consists of pointer to a data element and two pointers to children. The root node is the entry or the point where each search in a tree is started from. Then zero or more transitions from a parent to a child node is done until a data element with a required key is found or it becomes evidènt that there the data element with this key is not in the tree.

The simplest idea to utilize such trees for fast data access is to sort their items by the value of their key in such a manner that the left child of each item has the key value inferior to that of this node. And each right child item has a key with a greater value [6]. In this case each transition to a child narrows the range of search key value. And due to the fact that there is no intersection of key values in the subtrees of the left and right children, there is no necessity to go back to a parent node to restart the search in another subtree.

If we have a balanced tree, then the estimation of average data access shall be $t = O(\log_2 N)$, where N is the number of items.

What are the advantages of such a tree? Its memory consumption is almost as good as that of the list. Really, for each element to keep we need to create only one item in the tree. Such an item occupies only three machine words (two pointers to children and one to a data element).

The list of faults is bigger:

- Even if the tree is balanced, the access time strongly depends on the number of elements in the structure. And besides, the comparison at each node requires a full (in the case of equality) comparison of keys.
- The procedure of tree balancing is rather time consuming. The worst case for such a tree is when data elements with monotonically ascending or descending keys are added to it which is not uncommon in simulating systems, such a tree degenerates into a unidirectional list if no care is taken to balance it.

2.4. Bitcoded binary tree. The drawback mentioned in the previous subsection concerning the complexity of comparison at each item may be eliminated easily. An element of data with the bit key length N may be represented by a chain of N nodes in a tree. The top one shall denote the very first bit, the next one will be relevant to the second bit. The bit's value is determined by the way we came to an item. If at the previous level we have selected the left branch, the value of bit in the current position is 0. Otherwise it is 1.

Now to reach an element if the length of a key is N bits, we need to perform N comparisons of bits. Such a comparison is a very cheap operation, it is represented in hardware by all contemporary processors.

So, bitcoded trees are excellent from the point of view of performance. They have no problems of binary trees with balancing, the access is fast and they do not depend so strongly on the number of elements kept in the structure. Instead, the number of comparisons is equal to the length of a key in bits.

Nevertheless, these trees have not vanquished the traditional binary trees. The most reasonable explanation of this fact lies possibly in their memory consumption. To keep a data element, at worst $3N$ words are to be allocated (where N is the bit length of a key). Most of items in such a tree have no data associated with this item.

The access time for bitcoded trees can be further improved if "multibranch" implementation is used instead of binary. Of course, in this case, the memory overheads are higher than for conventional binary bitcoded trees. An element occupies at worst $3NQ$ words (where N is the bit length of a key and Q is the number of branches, a power of 2, preferably 4, 16, 256).

The memory consumption for multibranch trees can be improved if Q is variable which has the maximal value for the root and then it decreases from layer to layer until it reaches 2.

3. Expulsive tree

3.1. The general description of the idea. As it was shown before, the biggest problem of bitcoded tree was inefficient memory usage. A lot of items do not contain data, they are just allocated to represent a part of a certain key. Let us note also that in bitcoded trees the position of an item, which holds an element with a certain key, is absolutely determined by this key.

Let us try to design a tree structure which has no unused (by the pointer to data elements) items. At the same time we want to keep as much of the bitcoded tree advantages as possible: fast access due to the simple comparison at each node and absence of necessity to perform tree balancing.

First, we give up the strict association between the key value and the position in the tree. A notion called "relevance" is introduced. A data element is absolutely relevant to the position in the tree's hierarchy if the position's key fully coincides with that of a data element in bitcoded tree. Some of the data items may be located in the positions which are not fully relevant to their key. The more the size of prefix of their coincidence, the greater "relevance" between a data elements and a position.

The extreme case is the root item of an expulsive tree which is relevant to any data item. Its left child is relevant to all the data elements that have zero in the very first position of their key. The right child is relevant to all the items with '1' prefix. The deeper we go down the tree, the greater are the size of prefix and the degree of relevance.

The general idea is to place a data item for a certain element to the first empty position which is relevant. This means that such a position has the least possible relevance among all the relevant unoccupied positions. At a certain moment a situation might occur when there are no relevant positions free for a certain data element that we need to add to a tree. This is possible when the keys may have a variable length (e.g., a text string). To place such an item we need to free a space for it, this is done by an operation called the expulsion (hence, the name of the tree). The expulsion excludes (or "expulses") the data item at the position which is absolutely relevant to that of an item to insert. After the expulsion, the inserted data item is placed into the absolutely relevant position. But at this moment we have a recently excluded item. A good idea is just to insert it again in the tree. Here an indirect recursion might be produced with insert and expulse procedures-functions.

One of the requirements that we expect to meet is the absence of empty items in a tree, items without references to data element. The insertion guarantees that no empty items are produced. But deletion of an item which has children produces an empty item. To avoid this situation we introduce a counterpart of expulsion operation, the unexpulsion. The unexpulsion takes an item which is absolutely relevant to its position and which has no children, and removes this item from its position. Then this item is placed into the position which became recently empty as the result of the deletion. Unlike expulsion, which actually increases the relevance of an expulsing item to its position, this operation reduces the relevance. Nevertheless this is rather small price that we pay for considerable memory consumption reduction.

Just as for bitcoded trees, the access time may be improved for expulsive trees by switching from binary to multibranch implementation. In this case, memory consumption augments as some slots in data items are not used. Still, it remains more moderate than that of binary bitcoded trees. Other way to improve performance is to combine expulsive tree with hash, but this topic is beyond the scope of the paper.

3.2. Benchmarks and comments. The tests just as the theoretical estimations play an important role in comparison of data structures. Many obscure real world things which were not taken into consideration in estimations (in other words, the details that were eliminated at a certain level of abstraction) alter the result of comparison. On the other hand, a test demonstrates only one particular case. Thus a special care is to be taken to implement a representative set of tests.

Two benchmarks were implemented: memory consumption test and performance benchmark. In both of them a text file is scanned and its words (chains of symbols between delimiters) are added to the tested data structure as keys. The integer data associated with a key is a number of word occurrences in the text.

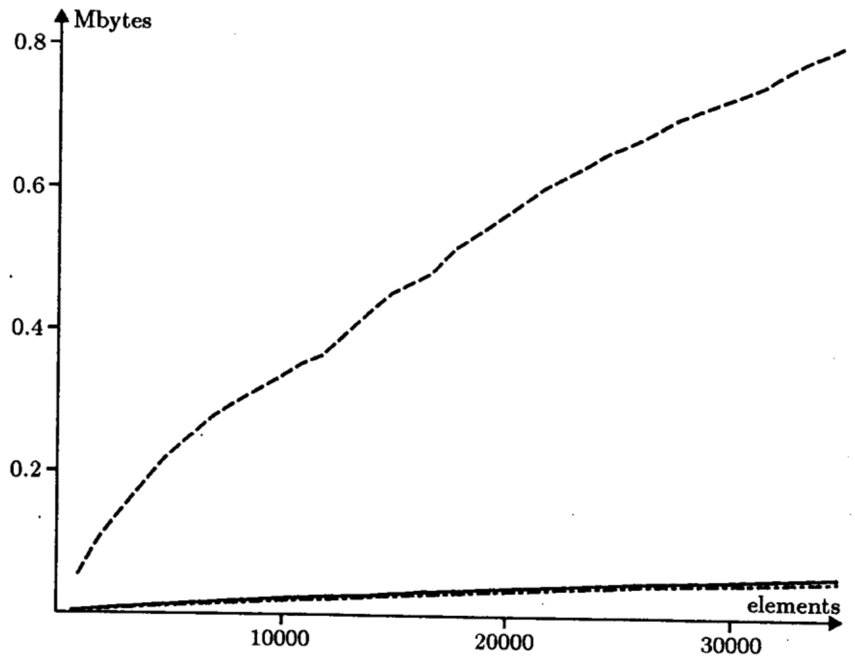

Figure 1. Memory consumption:
—— an expulsive tree, – – – a bitcoded tree, – · – a list

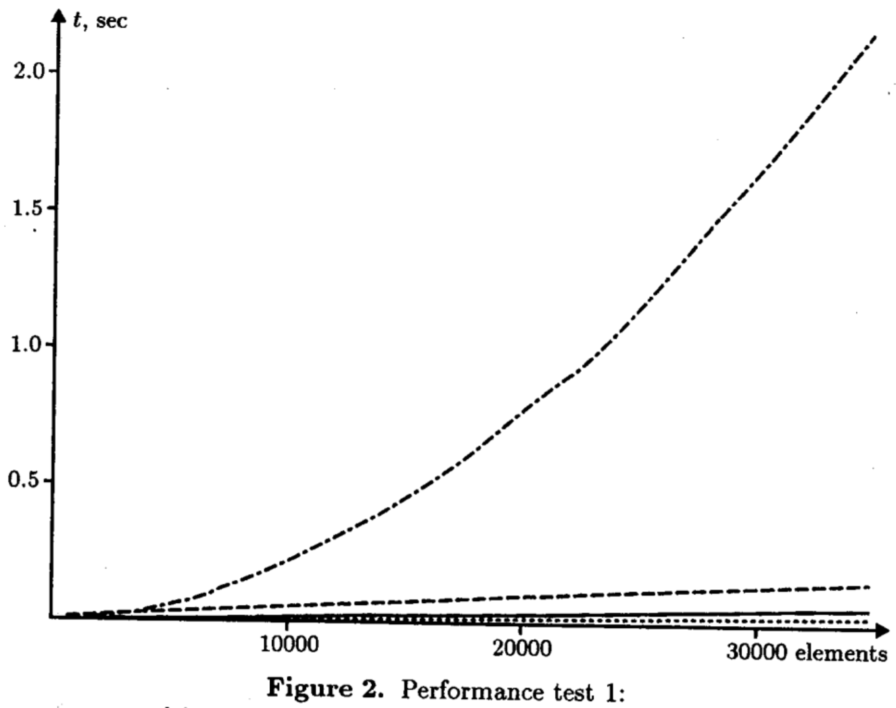

Figure 2. Performance test 1:
—— an expulsive tree, – – – a bitcoded tree, – · – a list, · · · a hash table

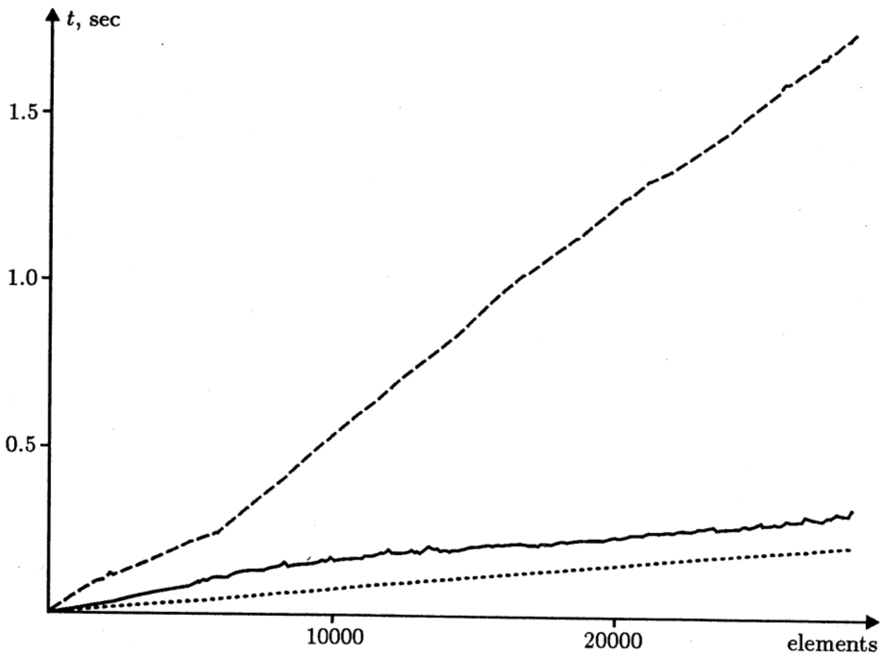

Figure 3. Performance test 2:
—— an expulsive tree, – – – a bitcoded tree, · · · a hash table

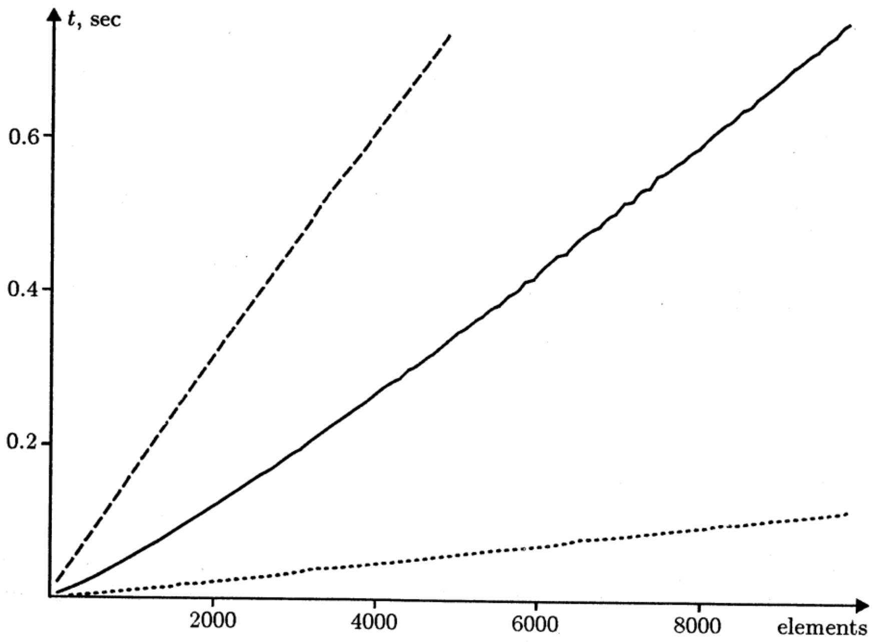

Figure 4. Performance test on short keys:
—— an expulsive tree, – – – a bitcoded tree, · · · a hash table

Memory consumption measurement results are presented in Figure 1. The measurement is done by monitoring the space allocated in the heap. All the calls of system heap allocation/deallocation functions are substituted by specially implemented monitoring functions. As it can be seen, the list offers the most moderate memory consumption. It is followed by the expulsive tree. The bitcoded tree is far more consuming. The hash table is not included into this test, as its size in memory does not depend on the number of stored data elements.

The performance benchmarks which are depicted in Figures 2 and 3, measures the difference of time between the start and the finish of word insertions into the examined data structure. Both tests are done iteratively starting with the minimal number of bytes read from a test file 1024 and increasing their quantity by 1024 at each step. In Figure 2, the comparison is shown for a list, bitcoded and expulsive trees, and hash table. In Figure 3, the statistics on bitcoded tree, expulsive tree, and hash table is represented. The fact that the expulsive tree is much faster than the bitcoded one can be justified by a big length of a key. In Figure 4, the results of another test are shown where the key is a 32 bit cardinal value. Evidently the difference in access time is not so big as for the benchmarks with symbolic strings as keys.

The comparison of a symbolic string key for expulsive tree in the performance test is currently implemented as full comparison of two character strings. Further performance improvement might be obtained in the case of partial comparison. The prefix of a character string which corresponds to the part of key relevant to current position need not to be verified again, as it is the same for all the children starting from a certain item.

All the source texts and the results of the benchmarks are available from the anonymous FTP login:

```
ftp://trans21.sscc.ru/pub/ostap/ds_article
```

4. Conclusion

The comparison of miscellaneous data structures from the point of view of their fitness for a fine-grained parallelism simulating system was done in the paper. It was shown that hash tables are unacceptable because of the necessity to estimate *a priori* the number of data elements, what is not possible for a simulating system. Among other data structures two have met the requirement of fast access: bitcoded tree and the proposed expulsive tree. The former may not be used for the considered type of simulating systems, as it has a tremendous memory consumption, while the latter eliminates this fault of the bitcoded tree and offers a satisfactory choice for the fine-grained algorithms simulating systems.

References

[1] Beletkov D.T., Ostapkevich M.B., Piskunov S.V., Zhileev I.V. WinALT, a software tool for fine-grain algorithms and structures synthesis and simulation // Lecture Notes in Computer Science. – 1999. – Vol. 1662. – P. 491–496.

[2] Booch G. Object Oriented Designs with Applications. – Benjamin: Cummings Publishing Company, 1991.

[3] Yourdon E. Techniques of Program Structure and Design. – New Jersey, Prentice-Hall, 1975.

[4] Zamulin A.V. Abstract Data Types in Computer Languages and Data Bases. – Novosibirsk: Nauka, 1987 (in Russian).

[5] Wirth N. Algorithms and Data Structure. – New Jersey, Prentice-Hall, 1986.

[6] Knuth D.E. The Art of Computer Programming. Volume 3. Sorting and Searching. – Addison-Wesley Publishing Company, 1973.

[7] Meyer B., Baudoin C. Methods of programmation. – Direction des metudes et recherche d'electricite de France, 1978.

[8] Kushnirenko A.G., Lebedev G.V. Programming for mathematicians. – Moscow: Nauka, 1988 (in Russian).

[9] The source texts of Linux kernel (www.linux.org).