Bull. Nov. Comp. Center, Comp. Science, 30 (2010), 71–80
© 2010 NCC Publisher

On selection of data structures for use in WinALT simulating system*

M.B. Ostapkevich

Introduction

The importance of a proper selection of data structures can hardly be overestimated. It is crucial for the overall performance in certain problem domains, such as file systems, task and memory managers in operating systems, indexing in the DBMS, dictionaries in compression utilities.

The existing data structures are extremely versatile. Some of them have a wide spectrum of applications, for example, linear lists and hash tables are used in task managers and memory managers in operating systems. B-trees in [1] are used in DBMS, file systems. Tries [2] are used to represent dictionaries.

Other data structures have a narrow application. For example, distributed hash tables (DHT) [3], BATON trees [4], VBI trees [5] are used in peering systems (P2P). A quadtree along with its modifications such as regional quadtree, octal tree, R and R+ trees [6], K-dimensional tree (K-D) and its modifications such as adaptive K-D tree, K-D-B tree, augmented K-D tree, BSP tree [7], hB, LSD [8] are used in GIS, geophysical applications, image processing systems, and CAD.

This paper is aimed to selection of a data structure for the associative search by a key, which is intended for the use in the simulating system for fine-grain algorithms and structures called WinALT [9, 10]. In Section 1, the purpose and architecture of the WinALT is briefly described. Its parts are outlined, for which selecting the data structure is crucial for the overall performance of the system. Such requirements are formulated which the system imposes on the data structures employed. Section 2 is dedicated to a review of existing data structures that are potentially suitable for the considered simulating system. A brief description is given for each data structure. Advantages and drawbacks are outlined, and a conclusion is made whether such structure meets the above-formulated requirements or not. Based on analysis of tries, a trie with an original mechanism of a compact representation in memory is proposed in Section 3. Its pros and cons are pondered. The validity of the choice of a data structure for WinALT is confirmed by the test presented in Section 4. Finally, general conclusions are drawn, and the plans of further development are outlined.

*Supported by RAS under Grant 1.6.

1. Using data structures in WinALT

The simulating system for fine-grain algorithms and structures WinALT is intended for the simulation of models of all the main classes of fine-grain parallelism (further FGP) on a single processor or multiprocessor computers. There is a language for the analytical description of models, which is based on a mechanism of associative substitutions [14] that is applied simultaneously to data arrays. The system graphical environment can be used for the visual construction of models. A set of the FGP classes rapidly evolves thus making it impossible for building a system as a fixed set of functions. Instead, it should have an open architecture that would permit the user to add a support for the new FGP classes.

The WinALT open architecture is based on the Dynamically Configurable Modular System (DCMS) [11]. The DCMS is a middleware library built above OS. It facilitates the construction of extensible applications from a variable set of modules that communicate by the means of events. The most typical kinds of data in WinALT are cellular arrays (1D, 2D, and 3D arrays of cells with a contiguous representation in memory) and a variety of miscellaneous descriptors (substitution descriptors, instances of their applicability, variables in a model program, event handlers) indexed by keys of different kinds (a string of characters, a string concatenated with coordinates, a pointer). It is worth to mention that there is always no more than one data element that corresponds to a certain name in the system. There are never two cellular arrays with the same name or two cells with the same coordinates within one cellular array. Thus, we can make a conclusion that data elements form an indexed set, where the index is represented by identifiers: names of cellular arrays, coordinates of cells, and so on. The performance of simulations is a critical parameter for the system. That is why all its structures used for a simulation reside in the main memory.

Several parts of the system can be outlined, in which the selection of data structures has an impact on the overall performance of the system.

An event driven mechanism of inter-modular communications was chosen in order to attain an open architecture in the system. An event manager is a module that implements this mechanism. It allows one to register procedures as handlers of events and to generate events. At the moment of the event generation, a list of procedure addresses be obtained by an event identifier. The event generation is one of the most frequently used operations in the system. Its efficiency is mostly determined by a data structure for the search by a string key. An event identifier is a character string of unlimited length. Thus, the structure must be efficient for keys of any length. A sample of inter-modular interaction based on event driven mechanism is presented in Figure 1.

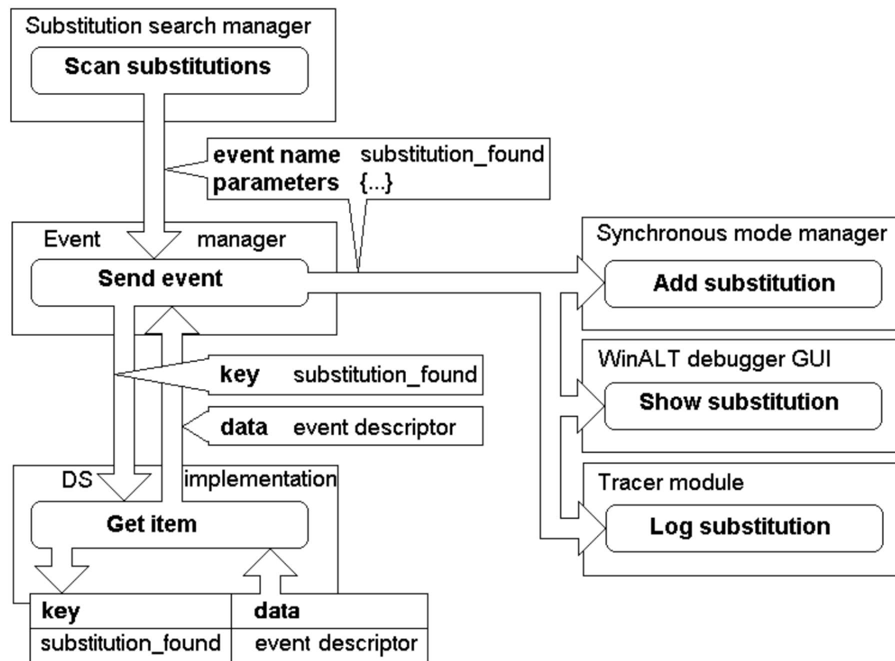

Figure 1. A sample of inter-modular event driven interaction

A subsystem of synchronous mode simulates an FGP device on sequential or parallel computers. One of its tasks is to provide a function to debug models. Collision is a situation when more than one rule is applicable to the same cell, and these rules require different values to be set as a new value for this cell. Collision detection is one of the main parts of a debugging process. In order to implement this mechanism, one has to accumulate all the found substitutions in a way that would permit to retrieve them by the name of a cellular array and coordinates. The duplication of all the modified cellular arrays would essentially increase the consumption of memory, because more properties than just a new value has to be stored for each applicable substitution. Particularly, references to a substitution descriptor and a relevant fragment of a model program must also be stored. Thus, a compact data structure must be used to quickly retrieve a description of an applicable substitution by a key that consists of a cellular object name and cell coordinates. The number of cellular arrays, cells and applicable substitutions depend only on models. There is no way for the system to limit them. Thus, the system must remain efficient with any number of those. There are several simulating modes in the system when in a contiguous mode the cellular arrays are sequentially examined, i.e., cell by cell. The keys used in this mode can be ordered. In other modes, the keys are just random. The system must remain efficient regardless of the character of the key distribution.

In addition to the two mentioned subsystems, there is an object manager in the system that needs an efficient data structure in order to attain a high performance, as one of its most used operations is the search of an object descriptor by the object name.

In all the considered cases there is a need for data structures that reside in the main memory and represent indexed sets. This structure must have a moderate consumption of memory and provide fast insertion and search of a data element by key. The following list of requirements to a data structure to be used in WinALT can be formulated:

- Data structure has to reside in the main memory.
- The data kept in the data structure is an indexed set (one key corresponds to no more than one data element).
- The number of data elements is unlimited.
- The principal operation is the data element search by a key.
- The key length is unlimited.
- The character of the key distribution in the stream of inserted data elements is unknown, it can be both sorted and random.

2. Overview of existing data structures

All the data structures that are suitable for a simulating system can be divided at the top level into two classes: structures that are represented by contiguous arrays and hash tables and lists. Contiguous arrays can be used for the associative search only for the key length below 16–24 bits, although with the best performance. They do not comply with the requirement of efficiency regardless of the number of data elements.

Hash tables [1] are also one of the fastest data structures. But there must be a priori estimation of the number of data elements in order to attain a high performance and moderate memory consumption at the same time. Thus, it does not meet the imposed requirements.

Among the lists, which are suitable for the search by a key, such structures as a linear list and trees can be mentioned. The linear list [12] is a list structure with a minimum memory consumption. Its main drawback is that it provides a descent performance only for a very small numbers of data items, so it violates the requirement of efficiency for any number of data elements.

Let us consider data structures that seem to be promising in the context of their use in the WinALT in some detail. Figure 2 shows the classification of trees.

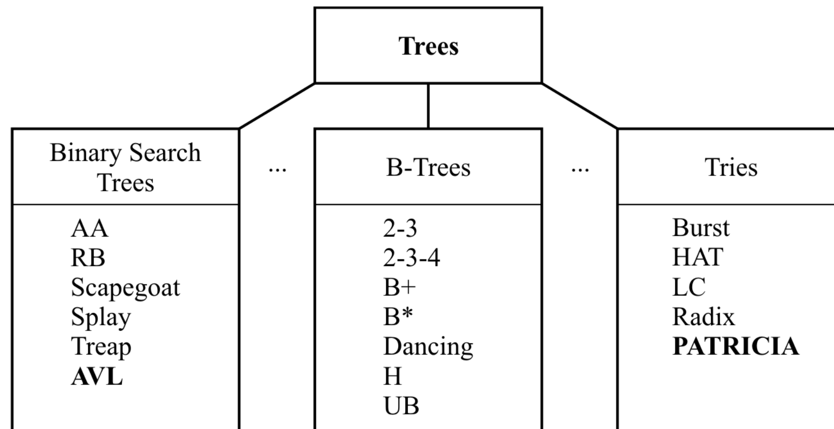

Figure 2. Classification of trees

Binary search trees. There are many kinds of binary search trees (further BST). AVL tree [16–18] is considered as one of the most efficient BSTs. An AVL tree meets all the imposed requirements as well as is possible with a BST. A common drawback of BSTs is that it is impossible to efficiently generalize them for higher arities. Also, there is virtually no way to introduce some form of data caching when most frequently used data elements move closer to a tree root and can be faster accessed.

B-trees [1,13] and their modifications do not comply with the requirement of efficient search when the structure resides in the main memory. In comparison with a BST, they have a greater memory consumption and more time for a search.

Tries. It is stated in [22] that a trie is a faster data structure for the search by a key than a BST. The advantage of a trie is that only some bits of a key should be compared in a node unlike BSTs, where a complete comparison of the keys must be done in every node. A trie can have different arities, and a caching version of a trie can be implemented. The major drawback of a trie is its huge memory consumption. Its binary version called the radix tree is less memory consuming, but is still more "greedy" than BSTs. Most of trie modifications introduce a certain mechanism of memory usage efficiency.

PATRICIA (Practical Algorithm To Retrieve Information Coded In Alphanumeric) [2] is the most well-known modification of a trie that has an excellent performance and a moderate memory consumption. Unlike the simplest trie and a radix trie, it has quite a small fraction of empty nodes. Nodes that have only one child merge with the nearest child node having two children. Each node contains a position of a bit or bits that must be used to select a subtree below. PATRICIA fully complies with a set of requirements and is considered to be one of the main candidates for tests.

3. The proposed tries modification

PATRICIA trie meets the requirements in a greater degree among all the considered data. It has a moderate memory consumption and a high performance. It allows the construction of trees with various arities. Its fault is that it has quite a complicated implementation from the standpoint of the size of source texts even for a binary case. For higher arities it is even worse. Moreover, PATRICIA still has some redundancies in the memory consumption because there is still a considerable number of nodes without data elements.

A modification of the trie named the expulsive tree is proposed to overcome the outlined faults. It retains all the important advantages of PATRICIA and has a similar performance for insertion and search. At the same time, it does not have empty nodes. Also, it has rather a simple implementation in terms of the source text size.

Empty nodes are excluded, because more than one position in a tree is allowed for the data element with a specified key. The algorithm was described in full detail in [23].

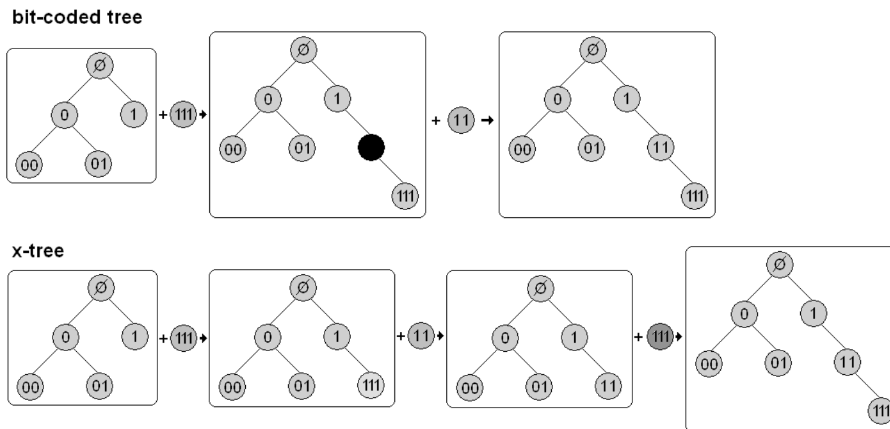

Figure 3. Insertion in the basic trie and in the expulsive tree

A sample of the data element insertion is depicted in Figure 3 both for the basic trie and the expulsive tree.

4. Comparison of efficiency of data structures

In order to confirm the correctness of the choice of a data structure we intend to measure its performance in a test. The following data structures were selected: AVL tree, PATRICIA trie and expulsive trie with arities 2, 4, 256.

The test program is written in C. It computes the number of occurrences of words in a text from an input file. The input file is generated by the Unix utility converting binary data to a text file (uuencode). The test is very simple and at the same time it generates the sequences of calls of the insertion and the search operations, which are similar to those in WinALT. The keys also resemble, those which are typically used in WinALT. In order to decrease the influence of other programs, the disk cache is purged before the test and the test itself is executed several times. As a result, a minimal time is then selected. The test is iterative. Parameters of the test are the minimal and the maximal sizes of a data set, delta of the size and the name of the input file. The results of the test are saved in the file and can be visualized by a graphic plotter component.

The results of tests are presented in Figures 4 and 5. The first one has approximately the same number of insertions and searches, which imitates how a data structure is used in a subsystem of the synchronous simulation mode. The second one issues considerably more searches than insertions, which is typical when using a data structure in the object manager and in the event manager.

In the test with a big fraction of insertions, the expulsive tree is better than AVL even in the binary case. The best performance is attained with the expulsive tree with arity 4. Its size in memory is just slightly bigger than that for PATRICIA trie.

In the second test, the expulsive tree with arity 2 is worse than AVL, while it is better for all the other arities. The expulsive tree with arity 4 is slightly worse than PATRICIA trie. The best performance is shown by the

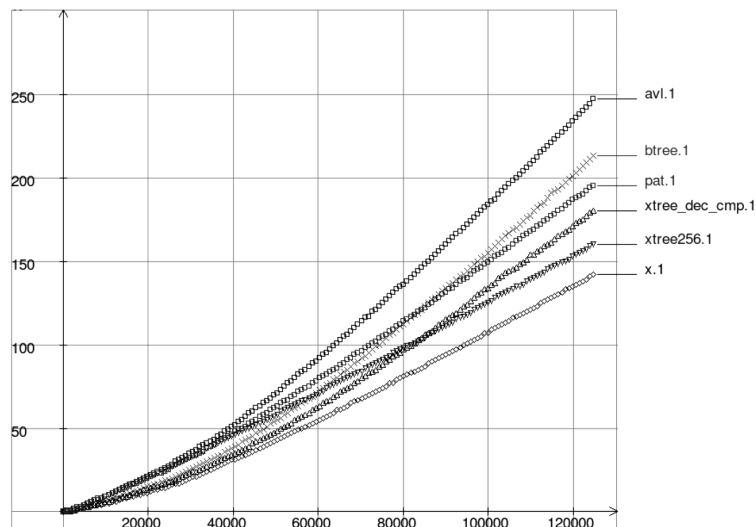

Figure 4. The results of the test with a big fraction of insertions

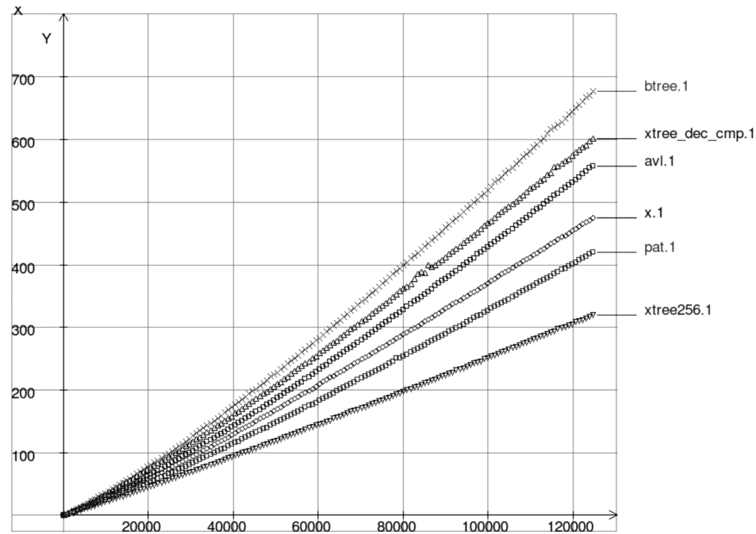

Figure 5. The results of the test with a small fraction of insertions

expulsive tree with arity 256, but its memory consumption is worse than for all the other tested trees.

A conclusion can be made that the best candidates to be used in WinALT are PATRICIA trie and the expulsive tree with arity 4. The expulsive tree can gain a certain decrease in access time (as compared to well-known non-hybrid data structures) in the case of a big fraction of insertions. In the case of a small fraction of insertions, the results are comparable to PATRICIA. It is also worth to mention that the algorithm for the expulsive tree is considerably simpler than that for PATRICIA trie.

Conclusion

The binary expulsive tree is used in the current stable version of WinALT. As the system evolves and the transition to new distributed platforms occurs, a set of WinALT requirements imposed on data structures will be refined. The analysis and the tests presented in this paper allow one to outline the steps for further development of the data used in WinALT. Two directions are planned: improvement of an expulsive tree and examination of hybrid trees. It is assumed that an expulsive tree with a variable arity and support for data caching might further decrease the time of the search and insertion with keeping a moderate consumption of memory. In order to verify this hypothesis, such kind of a tree and some hybrid trees, such as BURST trie [19], HAT trie [22], TST [19], hybrid AVL tree [16], string B-tree [13] should be implemented and tests of their performance and memory consumption should be performed.

References

[1] Main M., Savitch W. Data Structures and Other Objects Using C++. — Addison Wesley, 2000.

[2] Sedgewick A. Algorithms. — Addison-Wesley, 1983.

[3] Harren M., Hellerstein J.M., Huebsch R., Loo B., Shenker S., Stoica I. Complex Queries in DHT-based Peer-to-Peer Networks. — http://www.cs.rice.edu/Conferences/IPTPS02/191.pdf.

[4] Jagadish H.V., Beng Chin Ooi, Quang Hieu Vu. BATON: A Balanced Tree Structure for Peer-to-Peer Networks. — http://www.comp.nus.edu.sg/~ooibc/BATON.pdf.

[5] Jagadish H.V., Beng Chin Ooi, Quang Hieu Vu, Rong Zhang, Aoying Zhou. VBI-tree: a peer-to-peer framework for supporting multi-dimensional indexing schemes // Proc. Intl. Conf. on Data. — 2006.

[6] Samet H. Applications of Spatial Data Structures. — Addison Wesley, 1990.

[7] Gaede V., Gunthe O. Multidimensional access methods // ACM Computing Surveys. — 1998. — Vol. 30. — P. 170–231.

[8] Henrich A., Hans-Werner Six. The l-sd tree: spatial access to multi-dimensional point and non-point objects. — http://www.informatik.fernuni-hagen.de/import/pi3/PDFs/lsd-Tree-spatial-access.pdf

[9] Beletkov D., Ostapkevich M., Piskunov S., Zhileev I. WinALT, a software tool for fine-grain algorithms and structures synthesis and simulation // LNCS. – Springer, 1999. — No. 1662. — P. 491–496.

[10] WinALT home page.— http://winalt.sscc.ru/.

[11] Ostapkevich M. Event-driven tools for open system design // Bull. Novosibirsk Comp. Center. Special Ser. — Novosibirsk, 1999. — Iss. 1. — P. 15–22.

[12] Foster J.M. List Processing. — London: Macdonald&Co, 1967.

[13] Ferragina P., Grossi R. The string B-tree: A new data structure for string search in external memory and its applications // J. ACM. — 1999. — Vol. 46. — P. 236–280.

[14] Achasova S. M., et al. Parallel Substitutution Algorithm. Theory and Application. — Singapore: World Scientific, 1994.

[15] How to Build Patricia Trees. — http://goanna.cs.rmit.edu.au/~stbird/Tutorials/patricia.pdf.

[16] Bjornstrup J. Sorting and Searching using Hybrid AVL-Trees. — 1998. — (Technical Report; 10.1.1.21.5088.pdf).

[17] Evstigneev V.A. Application of Graph Theory in Programming. — Moscow: Nauka, 1985 (In Russian).

[18] Carrano F., Prichard J. Data Abstraction and Problem Solving with C++. — Addison Wesley, 2002.

[19] Zobel S.H.J., Williams H.E. Burst tries: A fast, efficient data structure for string keys // ACM Transactions on Information Systems. — Vol. 20. — P. 192–223. — (10.1.1.18.3499).

[20] Park G., Szpankowski W. Towards a Complete Characterization of Tries. — http://www.cs.purdue.edu/homes/spa/papers/profile-soda.ps.

[21] Knessl C., Szpankowski W. On the Number of Full Levels in Tries. — http://www.cs.purdue.edu/homes/spa/papers/fillup.ps.

[22] Askitis N., Sinha R. HAT-trie: A cache-conscious trie-based data structure for strings / G. Dobbie, ed. // Proc. XIIIth Australasian Computer Science Conference (ACSC2007). Ballarat, Victoria. — 2007. — P. 97–105.

[23] Ostapkevich M. Expulsive tree data structures for fast data search by a key // Bull. Novosibirsk Comp. Center. Ser. Comp. Science. — Novosibirsk, 1999. — Iss. 10. — P. 73–82.