A Fast Method For Accessing Nodes In The Binary Search Trees

İbrahim Ates, Mustafa Akpınar, Beyza Eken, Nejat YUMUSAK

Sakarya University, Department of Computer Engineering, Serdivan-Sakarya, Turkey
ibrahimates@yahoo.com

Abstract: In this study, a method that makes easy to process in the search trees is presented. A data structure which uses this method is also explained. It is explained how this method is used for strings. Performance comparisons with other trees like AVL, RB tree are showed. A hash table and a balanced binary search tree are used to implement this data structure. It is built the categorized subtrees according to data. Hash table is used to access data in the subtrees. It is aimed to process on relatively less amount of data collections instead of large amount of data collections. In this way the numbers of the process will be decreased. It will make positive affect on the program performance.

Keywords: Balanced binary search tree, Hash table, AVL, RB Tree, Salkim Tree

Introduction

The study of data structures is core to computer science. A wide range of container structures have been developed to meet different problem situations. The focus on data structures that efficiently store large collections of data (Tremblay J. P., 2003; Ford W. and Top W. 2002).

Data structure determines performance of the software. When program use large collections of data then data structure selection is getting more important. The structures have operations to access items, insert items and remove items from the collection. Effectiveness of a program depends on performance of deletion, insertion and searching process (Robson R, 1999; Weiss M. A., 1994).

Data can be stored in a memory sequentially or associatively. Data structures which are stored sequentially save data with position in a memory. When a data with value is wanted to be found, O (n) times search will be needed. Data structures, which save data with value in memory, are more suitable in this situation. This type of data structures is called associative data structures. These data structures are updated by using values, instead of using positions in associated data structures. Tree, is an example of this kind of data type (Larsen Kim S., 2000). AVL (Adelson Velskii and Landis) and Red-Black trees are the most important examples of tree data structure.

AVL Trees

AVL trees are binary search trees which are locally balanced. Depth of the AVL trees is arranged as O (logn). This means that AVL trees have the same depth of the left and right sub trees. The difference between left and right sub trees of any node can be one or zero. Cost of AVL algorithms is O (logN), when these algorithms are used for building tree, deletion, insertion and searching process (Larsen Kim S., 2000, Gabarró J. and Messeguer X., 1998; Cameron H. and Wood D., 1994).

AVL (Adel'son-Vel'skii and Landis) trees are efficient data structures for implementing dictionaries. AVL trees are binary search trees which are locally balanced; that is, for any internal node, the heights of its left and right subtrees may differ by at most one. The local balance at each node guarantees that the height of an *n*-key search tree will always be bounded above by 1.44 log*(n* +2*)*. Since AVL trees are the most efficient method of balancing binary search trees, they are utilized in a wide variety of applications such as databases, operating systems, and symbol tables in compilers.

T, an AVL tree, is a binary tree in which the difference between the heights of the left and right subtrees of any node is at most one. Elements from a totally-ordered domain are stored in the leaves with smaller data to the left of larger ones. For each internal node *v*, we use *k (v)* to refer to the key value stored in it and *l(v)* and *r(v)* to denote the left and right children, respectively. Moreover, *k (v)* always equals the key value of the largest element stored in node *v*'s left subtree. Such trees are usually referred to the literature as external AVL trees.

When we insert a new node into an AVL tree, some external nodes are replaced by a new internal node (and two external nodes as its children), and the height of the parent of new node may have been increased by one. As a result, if the height of newly inserted node is increased, the property of AVL tree may be lost at the ancestors of this new node. When the insertion causes an AVL tree to loose its balance, applying exactly one of the four rotations—*single rotations LL or RR* and *double rotations LR or RL*—will restore it.

Red-Black Trees

The red–black tree is a balanced binary search tree whose height is O(log n) and dictionary operations such as search, insertion, and deletion are performed in O(log n) time in sequential computation, where n is the number of nodes in the red–black tree.

In Red-Black tree (RB tree), every node has RED or BLACK attributes. Tree operations, except insertion, are costed O (log n) in RB tree. Insertion of an element will violate balance of tree which must be rebalanced. Rebalance process can be achieved with a simple operation, called rotation (Park H. and Park K., 2001; Cameron H. and Wood D., 1994).

Let root(T) denote the root node of a red–black tree T and item(x) denote the item stored in node x. Let p(x) denote the parent of node x and $p^{n+1}(x)$ the parent of $p^n(x)$, $n \geq 1$. Let rchild(x) denote the right child of node x and lchild(x) the left child of x. The successor of node x is the node with the smallest item larger than item(x). The predecessor of node x is the node with the largest item smaller than item(x). Each node x has a space for its item, a bit for its color (red or black), and three pointers to p(x), lchild(x), and rchild(x). If a node does not have a parent or a child, nil is stored in the corresponding pointer. We will regard nil as a pointer to an external node (leaf) and the nodes holding items as internal nodes.

A red–black tree is a binary search tree satisfying the following red–black properties (Park H. and Park K., 2001).

1. Every node is either red or black.
2. Every external node (nil) is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

The red–black properties can be rewritten using nonnegative ranks instead of red and black colors

(a) If x is any node with a parent, $\mathrm{rank}(x) \leq \mathrm{rank}(p(x)) \leq \mathrm{rank}(x) + 1$.
(b) If x is any node with a grandparent, $\mathrm{rank}(x) < \mathrm{rank}(p^2(x))$.
(c) If x is an external node, $\mathrm{rank}(x) = 0$ and $\mathrm{rank}(p(x)) = 1$ if x has a parent.

The above conditions (a)–(c) are called balance conditions. The rank of node x corresponds to the number of black nodes in any simple path from x to a descendant leaf. Hence, $\mathrm{rank}(p(x)) = \mathrm{rank}(x) + 1$ if x is black and $\mathrm{rank}(p(x)) = \mathrm{rank}(x)$ otherwise. Note that rank(x) need not be stored in x. (Park H. and Park K., 2001; Cameron H. and Wood D., 1994).

Salkim Tree

A data structure is a systematic way of organizing and accessing data. It is focused on data structures that store large collections of data. It is needed new data structures that can efficiently add and remove items without involving the entire collection of elements. In this study, Salkim tree is proposed to address this problem.

A hash table and RB binary search tree are used together to build Salkim tree. Collision case of hash table is used to categorize data. Selected hash function generates same index for different data in same category. Data are stored in a special form of binary search tree. In this form, root has one element which provides connection between tree and hash table. Data are stored in meaningful subtrees instead of one tree. When a process is needed for an element, process will work in related subtrees instead of all trees.

Implementation

When this data structure is wanted to build for letters, records are generated for each letter in hash table (Hrádek J., 2003; Zobel J., 2001). For this aim, hash function is used to generate index. Index value shows location of each letter in hash table. The root addresses of each subtree are stored in hash table. Hash function is shown in equation 1.

$$H(x)=ascii(x)-65 \tag{1}$$

Address records of subtrees are generated statically in hash table. Initial value of address records are NULL.When a string is wanted to be added to the structure, firstly hash table is checked whether subtree is created or not. If related subtree is created then string will be added to this subtree, otherwise a root will be created and string will be added to this root. The address of the created root will be written to the related place in the hash table.

Searching process of an element; hash table is checked whether related subtree exists or not. If subtree does not exist, no need more completion, it can be said that element does not exist in structure. Otherwise searching process will continue in related subtrees.

For example, lets assume that 'train' word is wanted to search in structure, Firstly index value of 't' is calculated using hash function (index value of 't' is equal to 19). 19 th section of hash table is checked whether any address exists or not. If 19 th section value of hash table is NULL, then it can be said that 'train' does not exist in the structure, otherwise 'train' word will be searched in 't' subtree. If 'train' word is wanted to search in any tree, all trees must be searched though it does not exist. This situation increases the cost of searching process in an ordinary binary search tree.

For example, cost of searching an element, in a balanced tree with 26000 elements, is 15. Salkim trees's cost is 10 in the same situation (when all letter categories have 1000 element).

Assuming that number of element is N and number of element started with 'i' is Ni,

$$N=Na+Nb+ \dots +Ni+ \dots Nz \tag{2}$$

Assuming that all element is not started with 'i', it can be said that

$$\mathrm{Log}_2 N > \log_2 Ni \tag{3}$$

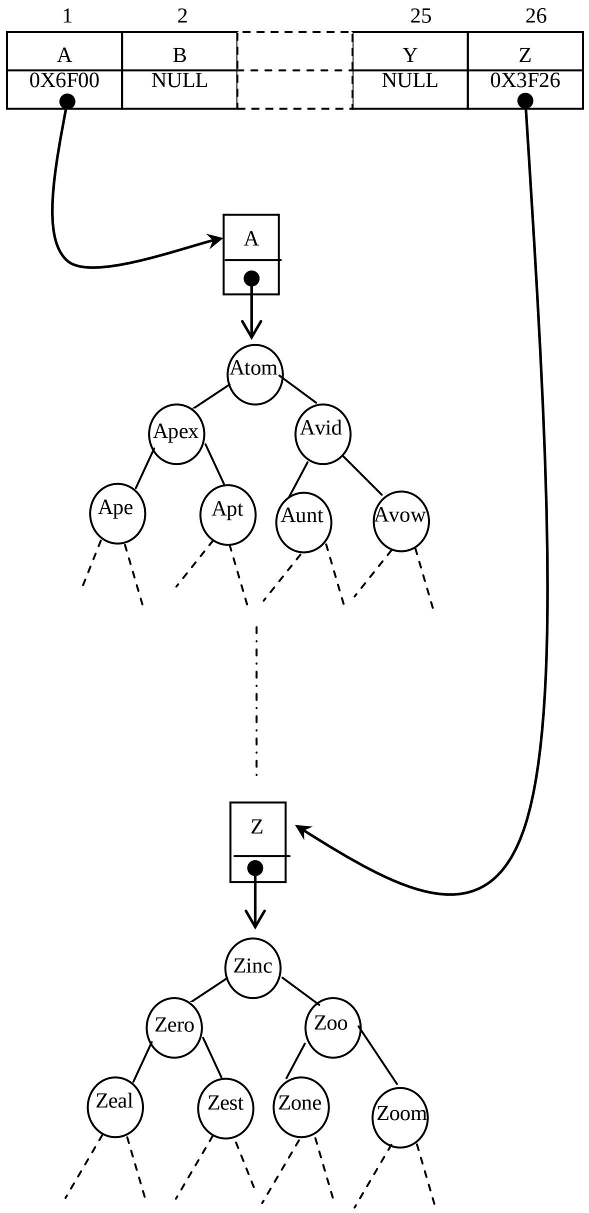

Figure 1. To access subtrees using hash table

Results

An application program is written to analyse building structure, inserting an item and searching an item performances of AVL, RB tree and Salkim.

Performances of building structure are examined for five different data sets. Amount of data in data sets are 25000, 275000,350000, 550000, 1100000. X axis of figures demonstrates these data sets. Y axis demonstrates process time. Build performance of those data structures is shown in Figure 2.

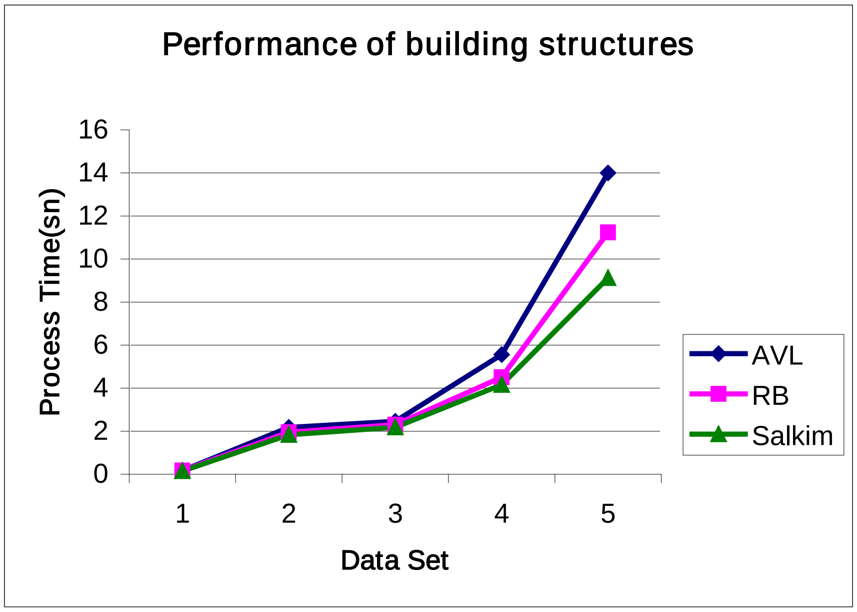

Figure 2. Performances of building structures

Table 1. Performance values of building structure test.

Data Set	Number of Data	AVL	RB	Salkim
1	25.000	0,17	0,17	0,15
2	275.000	2,173	1,956	1,833
3	350.000	2,46	2,303	2,18
4	550.000	5,56	4,506	4,156
5	1.100.000	14,006	11,237	9,124

Insertion performance of AVL, RB and Salkim tree is shown in Figure 3 and Table2. Note that all data structure was including 25000 elements before insertion test. Insertion performance is tested for four cases, in first case 100000, in second case 1000000, in third case 5000000, and in fourth case 10000000 elements are added into each structure.

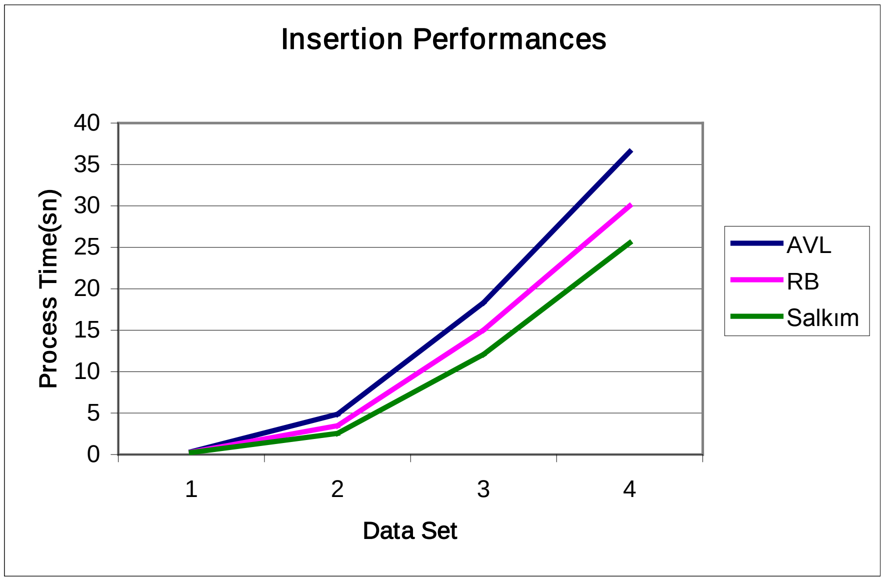

Figure 3. Performances of insertion item

Table 2. Performance values of insertion item test

Data Set	Number of inserted data	AVL	RB	Salkim
1	100.000	0,341	0,29	0,25
2	1.000.000	4,87	3,475	2,56
3	5.000.000	18,326	15,02	12,083
4	10.000.000	36,532	30,014	25,543

Search performance of structures is shown in Figure 4 and Table3. Note that all data structure was including 25000 elements before search test. Search performance of structures is tested for four cases. In first case 100000, in second case 1000000, in third case 5000000, and in fourth case 10000000 elements are searched on each structure.

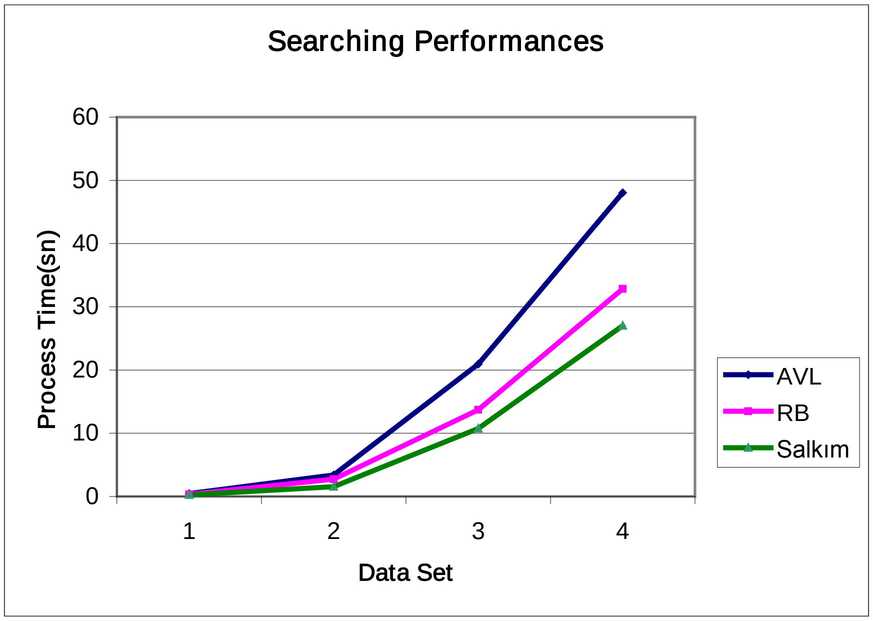

Figure 4. Performances of searching item

Table 3. Performance values of searching item test

Data Set	Number of searched data	AVL	RB	Salkim
1	100.000	0,451	0,341	0,25
2	1.000.000	3,395	2,744	1,542
3	5.000.000	20,92	13,71	10,752
4	10.000.000	48,037	32,837	27,01

Conclusion

Performance of Salkim tree is better than AVL and RB tree which are preferred in a lot of applications. Especially, search performance and insertion performance of Salkim tree's superiority is getting clearer when number of data increase.

Acknowledgement

We would like to thank people who are worked in Akcasu Software Corporation and Oea International Inc. Implementing of this study would be harder without their help.

References

Tremblay J. P., (2003), Tremblay J. P., "Data Structures and Software Development", Prentice Hall

Ford W. and Top W. (2002), "Data Structures with C++ using STL", Prentice Hall

Robson R., (1999), "Using the STL", Springer Publishing Company

Weiss M. A., (1994), "Data Structures and Algorithm Analysis in C++", Addison-Wessley

Larsen Kim S., (2000), "AVL Trees with Relaxed Balance", *Journal of Computer and System Sciences, Volume 61, Issue 3, Pages 508-522.*

Gabarró J. and Messeguer X., (1998), "Parallel dictionaries with local rules on AVL and brother trees", *Information Processing Letters, Volume 68, Issue 2, 30 October 1998, Pages 79-85.*

Cameron H. and Wood D., (1994), "Balance in AVL trees and space cost of brother trees", *Theoretical Computer Science, Volume 127, Issue 2, Pages 199-228*.

Park H. and Park K., (2001), "Parallel algorithms for red–black trees", *Theoretical Computer Science, Volume 262, Issues 1-2, Pages 415-435.*

Cameron H. and Wood D., (1994), "Insertion reachability, skinny skeletons and path length in red-black trees", *Information Sciences, Volume 77, Issues 1-2, Pages 141-152.*

Hrádek J., Kucha M. and Skala V., (2003), "Hash functions and triangular mesh reconstruction", *Computers & Geosciences, Volume 29, Issue 6, Pages 741-751.*

Zobel J., Heinz S. and Williams Hugh E., (2001), "In-memory hash tables for accumulating text vocabularies", *Information Processing Letters, Volume 80, Issue 6, Pages 271-277.*