საქართველოს მეცნიერებათა ეროვნული აკადემიის მოამბე, ტ. 10, №3, 2016
BULLETIN OF THE GEORGIAN NATIONAL ACADEMY OF SCIENCES, vol. 10, no. 3, 2016

Informatics

The Modification of the Sedgewick's Balancing Algorithm

Koba Gelashvili[*], Nikoloz Grdzelidze[**], Giorgi Shvelidze[§]

[*] *Faculty of Exact and Natural Sciences, Tbilisi State University, Tbilisi, Georgia*
[**] *Education Management Information System, Tbilisi, Georgia*
[§] *Operative Technical Department, State Security Service of Georgia, Tbilisi*

(Presented by Academy Member Mindia Salukvadze)

ABSTRACT. A modification of the Sedgewick's balancing algorithm is developed. Unlike the Sedgewick's algorithm, the output of this algorithm is always a complete tree. Moreover, the method of generalization of both algorithms on Red-Black (shortly RB) and AVL trees with insignificant changes of node without augmenting its structure is proposed. In order to establish efficiency of the new modification in terms of running time, the numerical experiments were conducted on ordered and Red-Black trees of different sizes for benchmarking DSW algorithm, Sedgewick's algorithm along with its modification. Trees were created with the help of either ordered or random data. According to the results, it is clear that in case of RB- trees the proposed new algorithm is faster than the other ones. © *2016 Bull. Georg. Natl. Acad. Sci.*

Key words: binary search tree, RB-tree, AVL-tree, BST balancing algorithm, DSW algorithm, Sedgewick's balancing algorithm

1. Introduction

The issue of balancing the binary search trees is quite an old one for computer sciences. However, only two algorithms are more common (see [1, 2]). In 1976, A. Colin Day (see [3]) proposed an algorithm that balances the binary search tree in a linear time without using the extra memory. Note that all balancing algorithms, discussed here, result in binary search tree with the height as minimal as possible.

Theoretically, Day's algorithm is the fastest balancing algorithm but its variant - the DSW algorithm is more refined, because it creates complete tree (see [2]) - in which every level, except possibly the last, is completely filled, and the bottommost tree level is filled from left to right.

There is also an alternative approach proposed by Robert Sedgewick (see [1]). Even though his recursive algorithm has the worst case time complexity $n \cdot \log n$ and requires the node structure augmentation via adding the subtree size, in practical applications it is more effective than DSW algorithm. The tree with nodes

© 2016 Bull. Georg. Natl. Acad. Sci.

having subtree sizes as attributes are called ordered search trees or simply OST.

In general, Sedgewick's algorithm does not result in complete tree, but in one special case it gives complete tree and perfect tree simultaneously. The perfect tree means a binary tree in which all interior nodes have two children and all leaves have the same depth or same level.

If the number of nodes n in a tree equals to $2^k - 1$, where $k = 1,2,3,...$, then the height of the tree obtained by the Sedgewick's algorithm will be k. There exists unique tree with $n = 2^k - 1$ nodes and the height k, which is both complete and perfect tree simultaneously.

Currently, the balanced trees (Red-Black or RB Tree [4] and AVL tree [5]) are used more commonly. For example, lots of data structures used in high-level programming languages or in computational geometry, and the Completely Fair Scheduler used in current Linux kernels, are based on RB trees. In case of RB tree, the height of tree h and the number of nodes n are satisfying an inequality $h \le 2\lg(n+1)$. Therefore, the balancing algorithms on RB trees are not of great importance in general. But, the situation concerning RB trees has changed recently for the following reasons:

- The usage of greater data became necessary.
- In realistic tasks (in practice) the height of the RB tree is often very close to its theoretical upper bound.
- Computers are not switched in passive state immediately (e.g. hibernate) and naturally appears the possibility to rebalance the certain structure.
- The rebalancing algorithms might be faster using multi threads. For example, typical laptops are able to ensure 4-8 threads or even more.

Sedgewick's algorithm has the following drawbacks: it is not applicable directly for RB Trees, it is not known in which cases it works in a linear time, and it doesn't result in a complete tree in general.

Our variation of the Sedgewick's algorithm, called modified Sedgewick's algorithm or shortly MSA, works on RB Tree without loss of effectiveness and results in a complete tree. To use the RB tree structure more effectively the color field might be used as subtree size. After the process of rebalancing, the tree will be colored again and the properties of RB Tree will be restored.

The fragments of the C++ codes are used to describe the proposed algorithms. For clarity, trees are drawn using code of github: https://github.com/mooseman/pdlinkedlist/blob/master/draw-tree.c (author - Daniel Sleator, http://www.cs.cmu.edu/~sleator). The algorithms that we created are on github as well: https://github.com/zgnachvi/hds/tree/randomized, https://github.com/zgnachvi/hds.

It should be noted that the proposed and the basic (Sedgewick) algorithms still have the several drawbacks: it is not known in which cases it works in a linear time; temporarily or permanently the node requires extra field (besides data and pointers) to be used to store the subtree sizes.

We have to mention that the presented algorithm is applicable to AVL trees as well, but we have omitted this direction, as AVL trees are used rarely and they are extremely balanced.

2. Sedgewick's Algorithm on Ordered Search Tree

Let us describe the notations and approach used in our simple implementation of OST. Codes of algorithms are based on the information given below.

The node has the following structure:

```
template<typename T>
struct Node
```

```
{
   T key;
   Node* child[2];
   int bf;
   Node();
   Node(T keyValue);
};
```

For a given node *x*, the attribute `x->bf` contains the number of nodes in the subtree rooted at x. In some cases it is more convenient to use a function N(x), which coincides with `x->bf`, if x is an address of any existing node, or 0 if *x* is NULL.

For OST, we assume that templated class `Tree<T>` is created not using fictive node for leaves. If any node does not have left or right child, the address NULL is written in the appropriate field. The attribute "root" in tree class defines the address of root. For more simplicity suppose it is public.

On any subtree, linear ordering is defined naturally. Let *r* be arbitrarily given node of OST, and x be some node of the tree rooted at *r*. In given subtree, the rank of *x* is denoted by Rank(x) and means the number of nodes, whose keys are printed before `x->key` after the invoke

```
   inorder_walk(r);
```

where

```
void inorder_walk(node* h) {
   if (NULL != h)     {
      inorder_walk(h->child[0]);
      cout << h->key << endl;
      inorder_walk(h->child[1]);
   }
}
```

is the tree traversal simplest algorithm.

In the text below, we will be considering the order of node in the sense of above-mentioned rank. More precisely the k-th node means that its rank equals to k and the median of subtree is the node with the rank of half of the subtree size (number of elements-nodes). Let us describe the Sedgewick's algorithm and MSA in more detailed way.

The public method that balances the tree is as follows:

```
template<typename T>
void Tree<T>::balance() {
   root = balanceR(root);
}
```

It calls the Sedgewick recursive algorithm (private method):

```
template<typename T>
Node<T>* Tree<T>::balanceR(Node<T>* h)  {
   if (h == NULL || h->bf < 2) return h;
   h = partR(h, h->bf / 2);
   h->child[0] = balanceR(h->child[0]);
   h->child[1] = balanceR(h->child[1]);
```

```
   return h;
}
```

Here the auxiliary algorithm (private method) that partitions a binary search tree moving the k^{th} node to the root is applied.

```
template<typename T>
Node<T>* Tree<T>::partR(Node<T>* h, T k)  {
   int t = N(h->child[0]);
   if (t != k)  {
      int dir = (t < k)?1:0;
      h->child[dir] = partR(h->child[dir], k - dir*(t+1));
      h = ostRotate(h, !dir);
   }
   return h;
}
```

Obviously, partR (`Node<T>*  h, int k`) algorithm takes time proportional to the height of the subtree rooted at *h*. Since the dependences between heights of subtrees and source tree in the process of balancing are not known, the time assessment of balancing algorithms becomes undefined.

In other words, after calling `balanceR(x)`, the algorithm `partR()` will find the median node and move it to the root using rotations. Further, it will recursively repeat the same procedure on both sides of the root.

In the process of balancing, the cascade of calls will be fulfilled by the generic function:

```
template<typename T>
Node<T>* Tree<T>::ostRotate(Node<T>* x, bool dir)  {
   Node<T>* y = x->child[!dir];
   x->child[!dir] = y->child[dir];
   y->child[dir] = x;
   x->bf = N(x->child[0]) + N(x->child[1]) + 1;
   y->bf = N(y->child[0]) + N(y->child[1]) + 1;
   return y;
}
```

Rotation is generic in the sense that the second argument determines the direction for the rotation: rotate(h,1) rotates to the right and rotate(h,0) rotates to the left.

3. Modified Sedgewick's Algorithm on Ordered Search Tree

Formally, the following public method corresponds to MSA:

```
template<typename T>
void Tree<T>::balanceMod() {
   root = balanceMod(root);
}
```

which calls recursive MSA:

```
template<typename T>
Node<T>* Tree<T>::balanceMod(Node<T>* h){
   if (h == NULL || h->bf < 2) return h;
```

```
   int height = (int)log2(h->bf);
   if (h->bf <= 3 * (int)pow(2, height - 1) - 1) {
      h = partR(h, h->bf - (int)pow(2, height - 1) );
      h->child[1] = balanceR(h->child[1]);
      h->child[0] = balanceMod(h->child[0]);
   }
   else {
      h = partR(h, (int)pow(2, height) - 1);
      h->child[1] = balanceMod(h->child[1]);
      h->child[0] = balanceR(h->child[0]);
   }
   return h;
}
```

It is the recursive algorithm, the correctness of which can be easily proved using induction.

Rotations do not change ranks of any of the nodes. Consequently, rank represents invariant for both rotations and balancing algorithms. The proof is almost trivial, so we omit it.

As is mentioned above, Sedgewik's algorithm moves the median node to the root using rotations. Further, it will recursively repeat the same behavior on both sides of the root.

Unlike, MSA algorithm finds the rank of that node, which will become the new root after balancing tree as a complete one. Then it will pass this value (rank) to algorithm `partR`, which will move the node corresponding to the given rank to the root. Further, it will repeat the recursive behavior on both sides of the root.

Let us consider the issue of representation of the rank of the node y, which will become the new root after balancing a subtree rooted at node *h* as a complete one, as function of attributes of *h*.

Both subtrees before and after balancing, have the same number of nodes $n \equiv N(h)$. As a result, the height of the complete tree is equal to $h \equiv \lfloor \log(n) \rfloor$ and $2^h \le n < 2^{h+1}$. In the compete tree either right or left subtree of the root y will be definitevely a perfect tree. The rank of the root depends on the side of the perfect subtree.

If the left subtree of the node y is not a perfect tree, then a right subtree will be a perfect one with the height of *h*-2, with the number of elements $2^{h-1}-1$. And, the number of elements in a tree will satisfy the inequality:

$$1+\left(2^h-1\right)+\left(2^{h-1}-1\right) \ge n \quad \text{i.e.} \quad n \le 3\cdot 2^{h-1}-1 .$$

In this case the rank of the y is $n-2^{h-1} = n-\left(2^{h-1}-1\right)-1$, as `inorder_walk` algorithm will not process the ($2^{h-1}-1$) number of nodes in the right subtree and the root itself.

The case when the left subtree of the node y is a perfect one is easier to handle. The rank of root is 2^h-1, i.e. the number of nodes in the left subtree of the root y.

Thus,

$$Rank(h) = \begin{cases} n-2^{h-1}, & n \le 3\cdot 2^{h-1}-1 \\ 2^h-1, & n > 3\cdot 2^{h-1}-1 \end{cases}$$

In the subtree with the node h the rank of y is the same as after balancing. As a result, the rank of the future

root can be calculated via attributes of the current root.

The following lines of MSA

```
int height = (int)log2(h->bf);
if (h->bf <= 3 * (int)pow(2, height - 1) - 1) {
   h = partR(h, h->bf - (int)pow(2, height - 1) );
   . . .
   }
   else {
      h = partR(h, (int)pow(2, height) - 1);
      . . .
   }
```

are calculating the height of the future complete tree and the rank of the future root via attributes of the current root h. Then the rank will be passed to algorithm `partR`.

The condition $n \le 3 \cdot 2^{h-1} - 1$ determines the side, where the perfect subtree will be situated. therefore, the recursive MSA will invoke Sedgewick's algorithm in the direction of the perfect subtree, and again MSA in the opposite side.

In the following table, the first OST is created inserting keys {1-9} with the given order. The second tree is obtained from the first one using Sedgewick's algorithm. The third tree is obtained from the first one using MSA. It should be noted that after the balancing process the height gets its possible minimal value.

3. Algorithms on RB-Tree

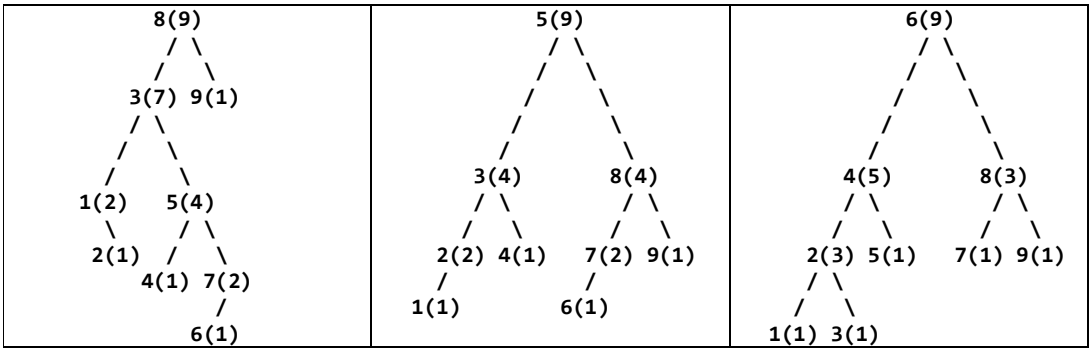
The structure of RB-tree's node is augmented by the field `Node* p`; which is used to store the address of parent node. The field bf is now used for color. The default constructor initializes it with value 0, meaning "RED". Each algorithm of RB-tree class takes into account the above-mentioned differences.

Single-byte "char" is used for the color in the RB-trees. However, if we want to expand Sedgewick's algorithm on RB-tree, it first should be transformed into OST. To this end, bf attribute should be of int type. Thus, a three-byte difference represents the cost to strengthen functionality of tree class.

The following private function, which works in a linear time and transforms RB-tree into OST-tree, after invoke writes the size of subtree rooted at parameter-node in the color field:

```
template<typename T>
int Tree<T>::updateSizes(Node<T>* h){
```

Table 1. In the lelft- OS tree; in the middle- tree, balanced using the Sedgewick's algorithm; in the right- tree, balanced using MSA



```
   if (h == NULL) return 0;
   int lN = updateSizes(h->child[0]);
   int rN = updateSizes(h->child[1]);
   h->bf = 1 + lN + rN;
   return h->bf;
}
```

Calling this function means the change of the "mode" in the tree, from RB-Tree into OST. As a result, we obtain opportunity of using above-mentioned balancing algorithms (attribute for parent node, inherited from RB-tree causes some changes). After the balancing is performed, it is necessary to return to the RB-tree. The following public method corresponds to the Sedgewick's algorithm on the RB-tree (in case of MSA, changes affect only the names of the balancing algorithms).

Here, RB-tree will be transformed into OST, obtained OST-tree will be balanced by Sedgewick's algorithm and then the height of the tree will be calculated. Finally, the private function of the class will be used to repaint again the tree:

```
template<typename T>
void Tree<T>::balance(){
   updateSizes(root);
   root = balanceR(root);
   int maxHeight = (int)log2(size);
   updateColors(root, maxHeight);
}
```

Function `updateColors` is very simple. A static variable is used to store the depth of the current node. The tree is traversed and only the lower nodes are colored in red. It operates in a linear time:

```
template<typename T>
void Tree<T>::updateColors(Node<T>* node, int maxHeight) {
   static int level = -1;
   if (node != NULL) {
      level++;
      node->bf = level < maxHeight ? 1 : 0;
      updateColors(node->child[0], maxHeight);
      updateColors(node->child[1], maxHeight);
      level–;
      }
}
```

The evolution of the RB-Tree (built with the data {1-6}) into a complete tree includes the following steps:

Table 2. From left to right: RB tree; corresponding OS tree; balanced OS tree; balanced RB tree

2(1) / \ / \ 1(1) 4(0) / \ / \ 3(1) 5(1) \ 6(0)	2(6) / \ / \ 1(1) 4(4) / \ / \ 3(1) 5(2) \ 6(1)	4(6) / \ / \ / \ 2(3) 6(2) / \ / / \ 5(1) 1(1) 3(1)	4(1) / \ / \ / \ 2(1) 6(1) / \ / / \ 5(0) 1(0) 3(0)

4. Numeral experiments

Numeral experiments were conducted on the computers of the following properties:

`Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 8.00 GB of RAM`.

To benchmark different balancing algorithms, DSW, Sedgewic and MSA are compared with each other on the two classes of the trees: on OST-trees and on the RB-trees. For each class, trees are created in two ways – with random data and with ordered data. In every subcase, 5 trees with sizes from 1000 to 1*E8 are selected, tested 20 times and the average time is taken.

In the tables below, every line corresponds to the balancing algorithms and columns correspond to the tree sizes. Each element shows the average time method requires to re-balance the tree of a given size. As we have two classes of trees and two ways to create the trees, we have four tables.

The following two tables show the results of the tests on OST- trees. In the first case, the tree is built with the random data. In the second case, with the ordered data.

As we can see, in most cases Sedgewick's method is faster, although, the difference decreases as the size of the tree increases.

The following two tables correspond to the more actual structure - RB-Tree. Balancing RB-trees occurs much faster via MSA.

Table 3. Average time for the OST- tree built with the random data

	1000	10000	100000	1000000	10000000	100000000
DSW	0,1234	1,5485	38,0266	455,544	4510,05	44519
Sedgewick	0,08275	0,64175	8,51495	118,487	1439,36	18332,5
MSA	0,16505	1,3371	15,8386	187,666	1887,54	18886

Table 4. Average time for the OST- trees built with the ordered data

	1000	10000	100000	1000000	10000000	100000000
DSW	0,03235	0,5917	6,841	108,509	1065,52	10044,4
Sedgewick	0,06395	0,62505	6,1622	58,8476	605,826	6478,43
MSA	0,0884	0,7524	6,5699	65,2096	639,261	6321,94

Table 5. Average time for the RB-Trees built with the random data

	1000	10000	100000	1000000	10000000	100000000
DSW	0,03545	0,44375	6,7467	273,303	3092,49	48030,1
Sedgewick	0,08365	1,0965	20,8714	273,696	2644,07	41896,1
MSA	0,08055	1,0372	20,4426	255,203	2623,6	38453,4

Table 6. Average time for the RB-Trees built with the ordered data

	1000	10000	100000	1000000	10000000	100000000
DSW	0,03525	0,41095	3,6737	82,8156	884,735	8848,1
Sedgewick	0,05155	0,551	5,9204	91,59	902,526	9310,06
MSA	0,0261	0,4845	3,74255	59,7136	874,461	7504,03

Acknowledgement

The presented work was supported in part by the Target Research Program Grant of Iv.Javakhishvili Tbilisi State University "The development of ordered container with improved functionality and efficiency "

ინფორმატიკა

სეჯვიკის დაბალანსების ალგორითმის მოდიფიკაცია

კ. გელაშვილი*, ნ. გრძელიძე, გ. შველიძე§**

** ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი, ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი, თბილისი, საქართველო*
*** განათლების მართვის საინფორმაციო სისტემა, თბილისი, საქართველო*
§ სახელმწიფო უსაფრთხოების სამსახური, ოპერატიულ ტექნიკური დეპარტამენტი, თბილისი, საქართველო

(წარმოდგენილია აკადემიის წევრის მ. სალუქვაძის მიერ)

დამუშავებულია სეჯვიკის დაბალანსების ალგორითმის მოდიფიკაცია. სეჯვიკის ალგორითმისგან განსხვავებით, ამ ალგორითმის შედეგი ყოველთვის არის სრული ზე. უფრო მეტიც, შემოთავაზებულია ამ მეთოდის განზოგადება წითელ-შავ და AVL ხეებზე, კვანძის უმნიშვნელო ცვლილებით მისი გაძლიერების გარეშე. იმისათვის რომ დავადგინოთ ახალი მოდიფიკაციის ეფექტურობა შესრულების დროის ტერმინებში, ჩატარებულია რიცხვითი ექსპერიმენტები დალაგებულ და წითელ-შავ ხეებზე DSW ალგორითმისთვის, სეჯვიკის ალგორითმისა და მისი მოდიფიკაციისთვის. ხეები აგებულია როგორც დახარისხებული, ასევე შემთხვევითი მონაცემებით. ტესტების შედეგების მიხედვით ცხადია, რომ წითელ-შავი ხის შემთხვევაში შემოთავაზებული ახალი ალგორითმი სწრაფია სხვებთან შედარებით.

REFERENCES:

1. *Sedgewick R.* (2001) Algorithms in C. Addison-Wesley Professional.
2. *Stout Q.F., Warren B.L.* (1986) Communications of the ACM. **29**, 9:902-908.
3. *Day C.* (1976) Computer Journal. XIX: 360-361.
4. *Cormen T., Leiserson C., Rivest R., Stein C.* (2009) Introduction to algorithms. The MIT Press,
5. *Adel'son-Velskii G.M., Landis E.M.* (1962) Doklady Akademii Nauk SSSR **146**: 263–266 (in Russian).

Received April, 2016