Red Green Black Trees: Extension to Red Black Trees

Seyfeddine Zouana*, Djamel Eddine Zegour
Laboratoire de la Communication dans les Systèmes Informatiques, Ecole nationale Supérieure d'Informatique, ESI (ex:INI), Oued-Smar, Algiers, Algeria.

* Corresponding author. Email: s_zouana, d_zegour@esi.dz
Manuscript submitted April 15, 2017; accepted July 5, 2017.
doi: 10.17706/jcp.13.4.461-470

Abstract: This paper propose an extended form of Red Black trees. It presents a new explicit balancing algorithm called Red Green Black trees. This structure tolerates some degree of imbalance that allows a decrease of the number of rebalancing relaxing the update operations. Through the use of three color nodes, the structure tolerates series of two nodes between Black nodes and defines a less balanced tree. It is interesting because the imbalance doesn't affect the update time and save the same level of performances of Red Black trees of *O(log(n))*. In fact, Red Green Black trees can provide better performances in environment where the restructuring is most frequent with Red Black trees.

Key words: Red black trees, balanced trees, rebalancing, restructuring.

1. Introduction

Binary Search Trees are a very important and largely used data structure to implement dictionaries mainly, but also to represent symbol tables and key indexes. They have simple updating algorithms and they can be maintained with only a limited number of restructuring per update. They are also used to implement task schedulers as interval trees and priority search trees. Binary Search Trees sort tasks automatically for the scheduler without the need to re-sort them. The keys are sorted on update, the lesser keys are inserted to the left sub-tree and the greater keys to the right sub-tree. The strategy of the scheduler is described by the tree distribution. This distribution is more significant when the tree is balanced.

Various balanced binary search trees are used, AVL trees and Red Black trees [1], [2] are the most suited examples. In fact, this is due to their almost complete balance giving high performances for search and update of keys. But, these almost balanced trees become poor when used to represent schedulers because of the great number of rebalancing they require. This problem becomes more complex when the tasks are concurrent and run with a set of relations. To reduce the number of restructuring, the structure must be relaxed.

Relaxation can be done by uncoupling the rebalancing form the update and lifting some of its cases or by tolerating some imbalance on the structure. While the first only postpones the rebalancing, the tolerance of some imbalance decreases majorly the number of rebalancing allowing faster updates. However, this goes against the search time as the height of the tree increases. This creates a big tradeoff between accelerating updates and slowing search. This idea can be applied to any balanced tree. AVL trees, for instance, are defined by the condition of the difference between the height of the left and the right sub-trees must at most be 1. We can define an imbalanced AVL[n] [3] by increasing this amount. Whereas, Red Black trees, where each node is either Red or Black and defined by the two restrictions of having the same number of Black nodes on

each path from root to leaf and the no succession of Red nodes is allowed. The imbalance can only be defined by allowing a series of Red nodes; for the first condition omitted, all the balance of the structure is gone. The imbalanced AVL[n] trees require the same height update of the perfectly balanced AVL which implies no gain in update. However, tolerating series of Red nodes decreases the number of Black nodes and the number of rebalancing as a consequence.

We propose a partially balanced binary search tree, called Red Green Black Trees, where we tolerate up to two nodes between two Black nodes extending the Red Black Trees condition of having at most one node between two Black nodes. The balance of the structure is defined by the number of Black nodes on each path. The formal definition of the structure is given in section 3. We discuss the different cases and restructuring of the insert/delete operations in section 4. To prove the preservation of the same level of performances, we give a little analysis of the worst case height of the tree in section 5. Some experimental results are given in section 6 to illustrate the effects of the generalization of RB trees.

2. Related Works

Balanced Binary Search Trees have great importance in both organizing and sorting data. From the beginning of their appearance, they give an efficient implementation to dictionaries and schedulers, though, they require too much maintenance (each update require a number of rebalance operations). The original papers of Adel'son Velski and Landis [1], and Bayer [4] introduced two self-balancing structures giving performance of $O(log(n))$ level. These two structures give amazing short search and update time. However, they require a significant number of maintenance operations. The balance of the structure can lower the performance when too much rebalancing cripples the application. The AVL trees has seen generalizing works such as AVL[n] [3] and HB[k] [5] that aimed to offer some control on both the degree of balance and the maintenance frequency, this problem has only been mentioned by Bayer in his introduction of the Symmetric Binary B-trees (SBB-trees) [6] which are a binarization of the B-trees where he defined a generalization idea of SBB[k] trees to limit the rebalancing. But no explicit algorithm has been given. These trees have been modernized by the dichromatic framework of Guibass [2] as Red Black trees. Arne Anderson [7] defined another generalized form of SBB trees where the black nodes limit small trees that are bound to be AVL. This structure didn't take a large use because of the number of conditions it requires. And RB trees and AVL trees preserved their position which led to relaxation works. Arne Anderson, for instance, gave a simple form for RB trees called AA trees that takes only the left rebalancing cases into account lifting half of the charge. Other works aimed to uncouple the update from the rebalancing and adapted them to more complex environments and applications such as: Relaxed Balanced RB trees [8] which define a relaxed balance by interleaving the updates, and Relativistic RB trees [9] that give a wait-free solution to the problem of having concurrent update operations.

3. Red Green Black Trees

We define a Red Green Black tree as an extension of the RB trees where we find three node colors. This provokes some imbalance on the structure as the balance conditions are lightened by the tolerance of successions of two nodes between black nodes. We preserve the condition of the same number of Black nodes on each path. So formally, we can define a Red Green Black tree as a tree where:

Each node has either a Red, Green or Black color.

- A nil pointer is conventionally a Red Green Black tree and is of Black color.
- Each path from Root-to-Leaf has the same number of Black nodes.
- A Green node can't have a Green node son and must at least has a Red node son.
- A Red node must have only Black node sons.

4. Maintenance Algorithms

The update operations are as simple as the Red Black trees operations. They are similar to any BST update operation and followed by some restructuring to ensure the preservation of the defined balance. The insertion and delete procedures are summarized as follows:

4.1. Insertion

Any new key is inserted to the tree by searching for its place designated by the nil pointer which makes it a leaf node. The new node is always of Red color. As the parent may be of Red color, this could lead to imbalance in the tree defined by two Red nodes in a row. We distinguish different cases to rebalance as follows (Fig. 1):

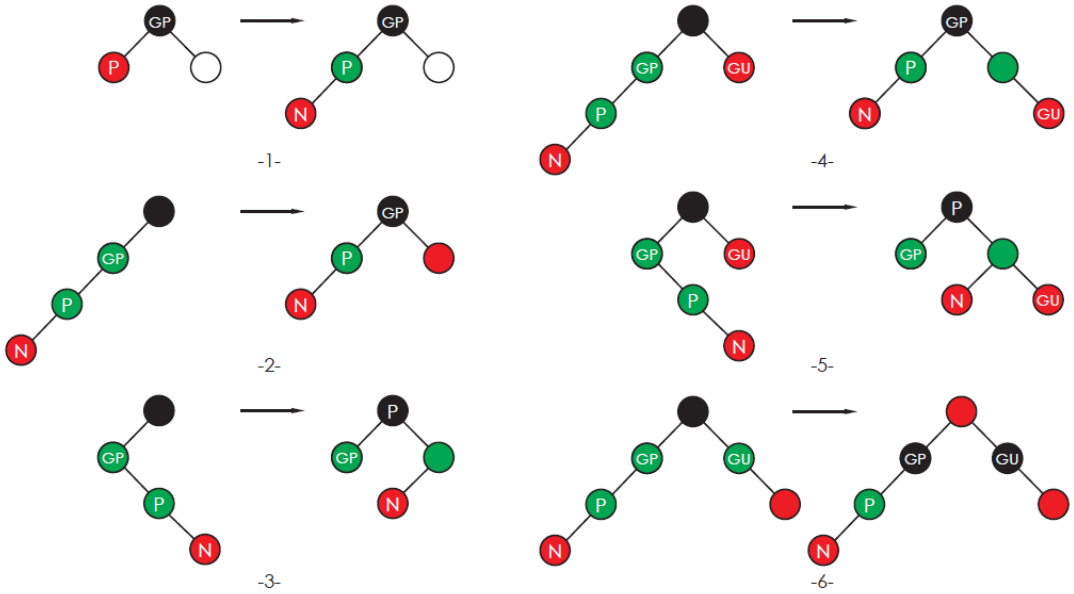


Fig. 1. Insertion cases.

- Case 1: Grand Parent is Black, the new node becomes Red and the Parent Green.
- Case 2: Grand Parent is Green, the sibling of the Grand Parent is (nil/Black). First, the Parent becomes Green. If the Parent is a left son of the Grand Parent then, we do a simple rotation on the Black node parent to the Grand Parent. The Grand Parent becomes Black and the previous Black node becomes Red.
- Case 3: Grand Parent is Green, the sibling of the Grand Parent is (nil/Black). First, the Parent becomes Green. If the Parent is a right son of the Grand Parent then, we do a double rotation on the Black node parent to the Grand Parent. The Parent becomes Black and the previous Black node becomes Green.
- Case 4: Grand Parent is Green, the sibling of the Grand Parent is Red. First, the Parent becomes Green. If the Parent is a left son of the Grand Parent then, we do a simple rotation on the Black node parent to the Grand Parent. The Grand Parent becomes Black and the previous Black node becomes Green.
- Case 5: Grand Parent is Green, the sibling of the Grand Parent is Red. First, the Parent becomes Green. If the Parent is a right son of the Grand Parent then, we do a double rotation on the Black node parent to the Grand Parent. The Parent becomes Black and the previous Black node becomes Green.
- Case 6: Grand Parent is Green, Parent is Red, Grand Uncle is Green. We flip the colors so as the Grand Parent and the Grand Uncle become Black, the Black node becomes Red. This color flip can provoke a series of rebalancing in a cascade phenomenon towards the root of the tree as it adds a Red node that may lead to the same previous situations.

4.2. Delete

We search for the node to delete like in any regular Binary Search Tree. If the node has non leaf children, we find the minimum element in the Right sub-tree. We substitute the keys. We then remove the node which we took its value. This node must have at most one leaf child. Then, the delete becomes much simpler. If the node to remove is Green, we just link its child in its place. If the node to remove is Black and its child is Green, we link its child in its place and color the child Black. If the node is Black and has no children, it means the node to remove is the root and the tree becomes null. If the node is Red and its Parent Green, we update the color of the Parent from Green to Red, else if the Parent was Black, we undergo some rebalancing as follows (Fig. 2):

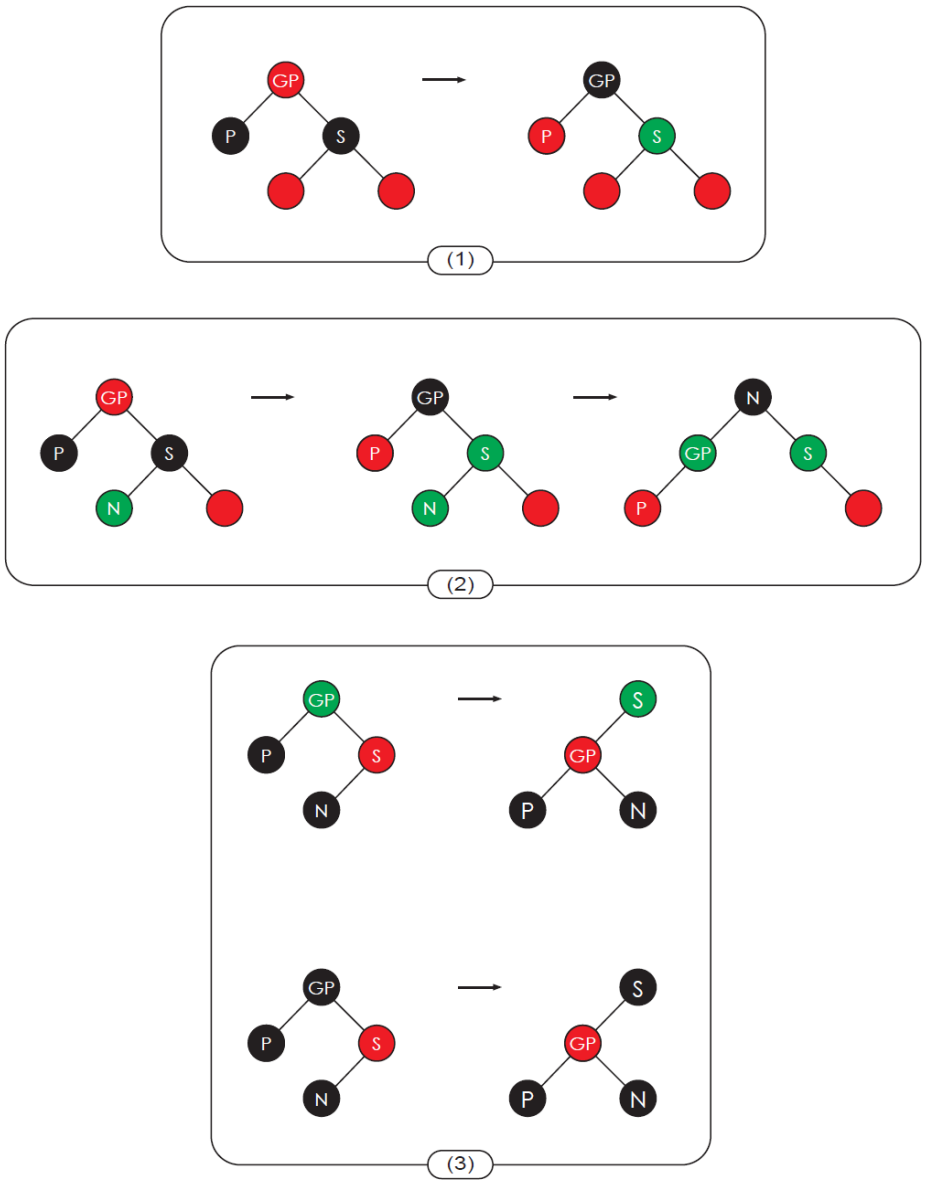


Fig. 2. Delete cases.

- Case 1: if the node P has Red Parent and Black sibling with no Green child, flip colors such as the Parent becomes Black, P becomes Red, its sibling Green.
- Case 2: if the node P has Red Parent and Black sibling with Green child, flip colors such as the Parent becomes Black, P becomes Red, its sibling Green, then rotate on the Parent to eliminate the series of Green nodes (a simple or double rotation depending on the case).

- Case 3: if the node P has a non-Red Parent GP which implies that it has a Green/Red sibling S. We find the first Black node N on the path by going to the other side of the Parent GP then crossing to the side of P from that node until finding the Black node. This Black node N has a Red Parent. The sibling S is linked to its Grand Parent. The non-Red Parent GP takes P and the found Black node N as children, the non-Red Parent GP becomes Red and child to the Red Parent. The Red Parent becomes Green and we continue with either case 1 or case 2.

The flip of colors of case 1 and case 2 can lead to the same situations as before. Repairing that require further rebalancing on the upper levels of the tree.

5. Worst Height of the Tree

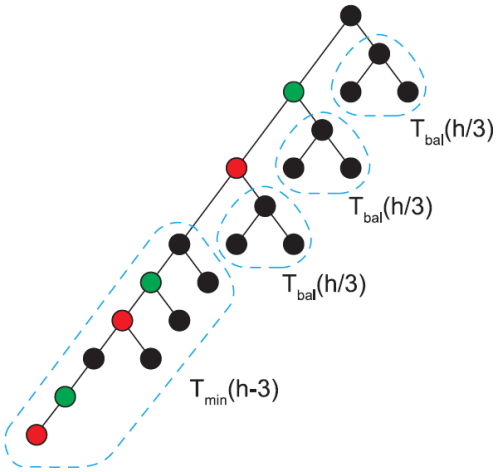


Fig. 3. Worst case tree.

Let the height of the RGB trees be the maximal number of nodes in any path from the root to a leaf. Then an RGB tree $T_{min}(h)$ of height h with the minimum number of nodes is of the form Fig. 3. Notice that $h=3.j$ where j is the number of Black nodes in each path from Root-to-Leaf. Notice that each node of the longest path of the root small tree is linked to the root of a perfectly balanced sub-tree $T_{bal}(h/3 -1)$ while the last one is also linked to a sub-tree $T_{min}(h-3)$. Let $N(T)$ be the number of nodes in the tree T. Then:

$$N(T_{min}(h)) = 3 \cdot N(T_{bal}(\frac{h}{3} - 1)) + 3 + N(T_{min}(h - 3))$$

since:

$$N(T_{bal}(k)) = 2^0 + 2^1 + \cdots + 2^{k-1} = 2^k - 1$$

we obtain:

$$N(T_{min}(h)) = 3 \cdot (2^{\frac{h}{3}-1} - 1) + 3 + N(T_{min}(h - 3)) = 3(2^{\frac{h}{3}-1}) + N(T_{min}(h - 3))$$
$$= 3(2^{\frac{h}{3}-1}) + 3(2^{\frac{h}{3}-2}) + 3(2^{\frac{h}{3}-3}) + \cdots + 3(2^0) = 3 \cdot \sum_{i=0}^{\frac{h}{3}-1} 2^i = 3(2^{\frac{h}{3}} - 1)$$

So the number of nodes of the tree $N(T)$ is bound by:

$$3(2^{\frac{h}{3}} - 1) \le N(T_{min}(h)) \le N(T) \le N(T_{bal}(h)) = 2^h - 1$$

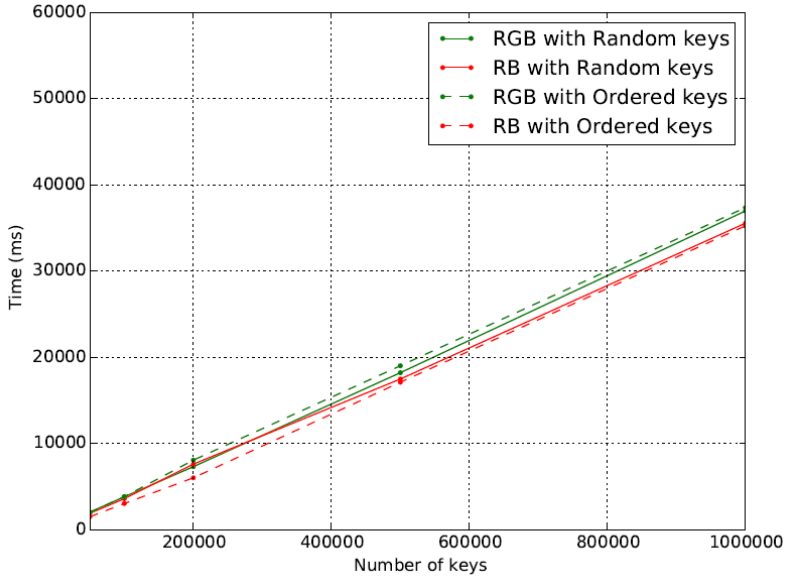
Which implies the in-equations: $h \le 3\log(N+3) - 3\log(3)$ and $\log(N+1) \le h$

The height of the RGB tree is at most *3log(N + 3) - 3log(3)* which is a little worse than RB trees height *2log(N+2) - 2*. This could lead to a large difference with large ordered sets.

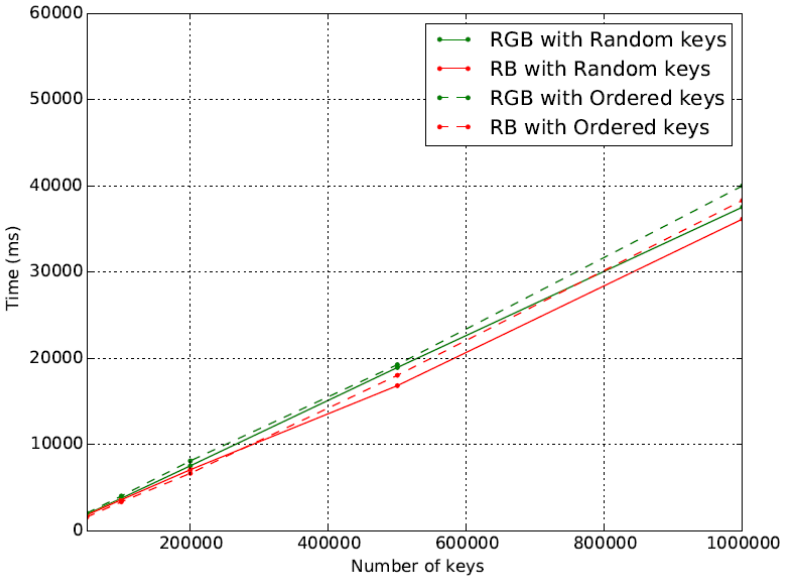
6. Experimentation

For a set of *N* keys, the RGB height is at most *3log(N + 3) - 3log(3)* preserving the *O(log(N))* performances of the RB trees. However, this cost is much less in practice. To define the behavior of the structure, we followed two main scenarios:

- Scenario 1: we insert files of randomly generated/ordered keys of sizes *n={50000, 100000, 200000, 500000 ,1000000}* . Then we delete them.
- Scenario 2: we insert the same files as in Scenario 1 and then we use those files to check if each key is in the set, we delete it. Else, we re-insert it.

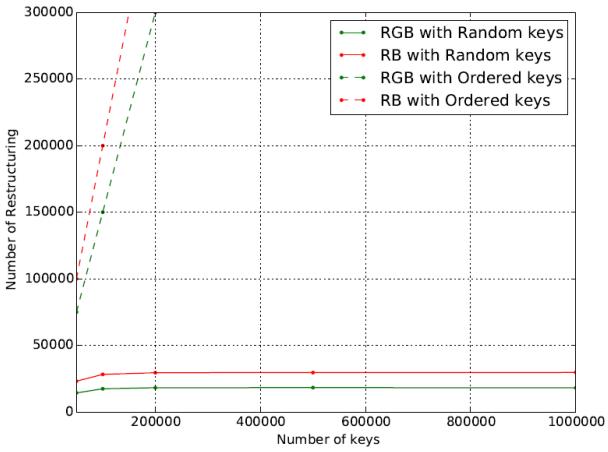


(a) Scenario 1

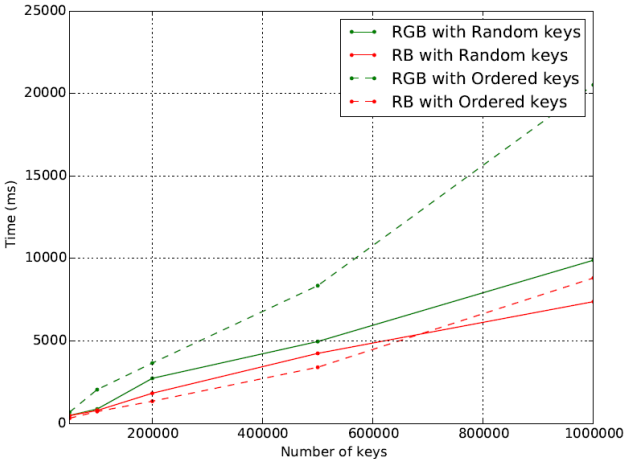


(b) Scenario 2

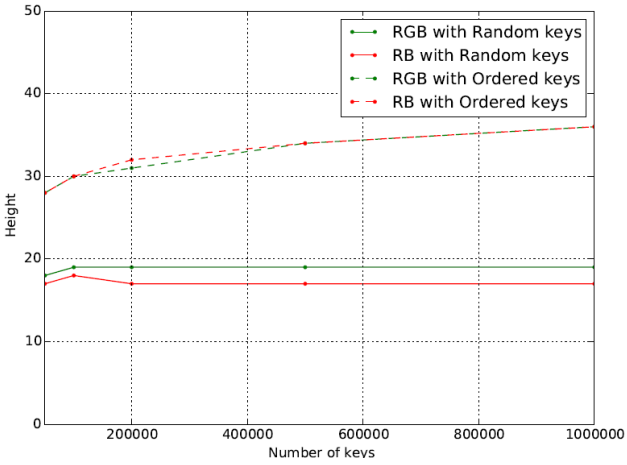
Fig. 4. Tree construction time.



(a) Number of Restructuring in the construction phase



(b) Tree search time



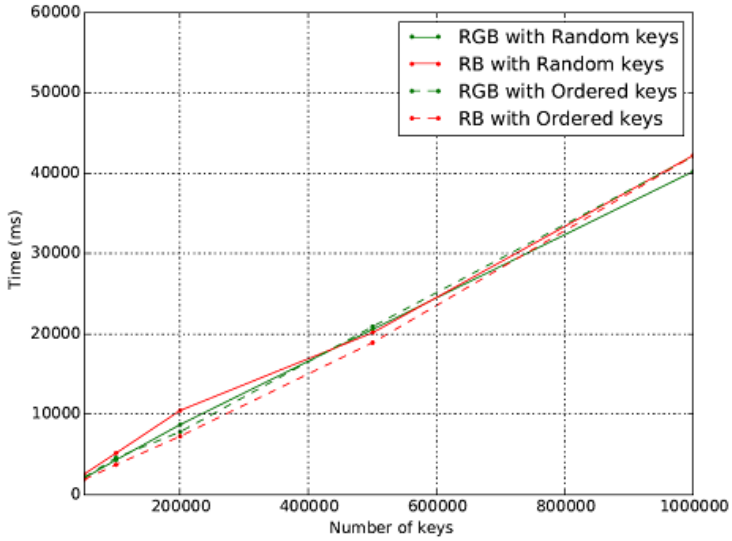
(c) Tree maximum height

Fig. 5. Evolution of number of rebalancing, search time and height of the tree.

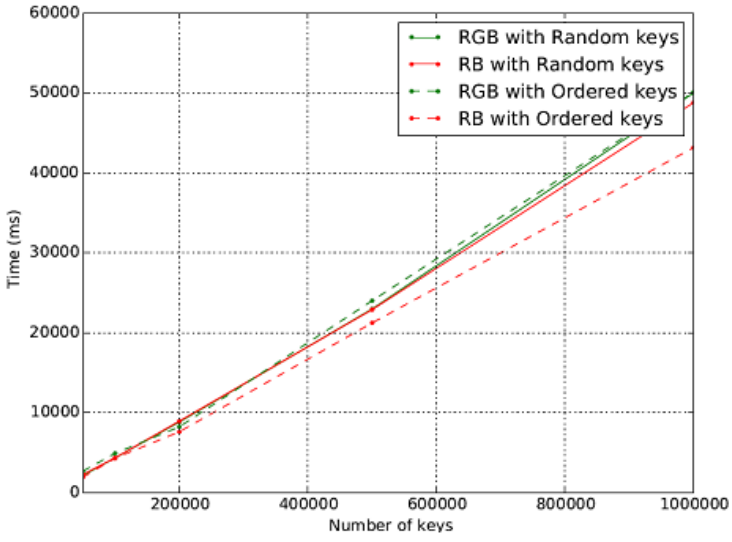
Those two scenarios allowed defining the performances of insert/delete operations and the behavior in a real life situation where update operations come randomly. The results for both scenarios are very interesting for both RGB trees and RB trees. When the keys are randomly generated, RGB and RB trees have slightly the same construction time (Fig. 4), the average insert operation in both trees takes about the same time with RGB trees taking an extended portion of time. But, in RGB trees, we observed that the number of restructuring (Fig. 5-a) is much less than RB trees. This is explained by the design of RGB trees that allows some imbalance on the structure by tolerating a series of two Red nodes (Red and Green), which implies a less balanced structure with a bigger height and a larger search time as a consequence. The increase in search time and height of the tree is due to the increase of the number of Red nodes in each path in accordance with theory. The maximum height (Fig. 5-b) increased by one third enlarging the search time (Fig. 5-c) which can be considered as a limitation for RGB trees. However, the RGB trees don't lose in performance globally and the gain in restructuring comes handy in environment where restructuring are costly. On the other hand, when the keys are ordered, which is the worst case for RGB trees, the height of the tree (Fig. 5-b) grows exponentially, the search time (Fig. 5-c) and the update time multiply because of the tolerated imbalance. This is not the case for RB trees that have almost perfect balance. We observed however a big increase of the number of restructuring (Fig. 5-a) for RB trees comparing to the randomly generated keys case suggesting that even the worst case RGB trees is to consider for their large gain in restructuring.

Like insertion, RGB trees and RB trees have about the same time in delete (Fig. 6). The RGB trees preserve their low number of restructuring (Fig. 7). We found that RGB trees have slightly better time for the delete operations even when the keys are ordered. In both scenarios, the RGB trees delete time (Fig. 6) is about the same of the RB trees delete time though it has a large difference in the needed number of restructuring (Fig. 7). It is interesting to note that in the Scenario 2 neither the time nor the number of restructuring increased. So the update time isn't affected by the order of the update operations.

Generally, we observed a major decrease in restructuring (rotations/ color flips) in RGB trees. This decrease provokes an increase of the height of the tree and consequently the search time. However, this increase isn't too big to change the total time of the update. Furthermore, the gain in restructuring is too significant making RGB updates very fast when the keys are randomly generated. If the keys are ordered, however, the update time increases gradually. The results show that RGB trees and RB trees have almost the same update time with a big difference in restructuring. The RGB update has a slightly bigger search phase with a very small restructuring phase. This characteristic makes the proposed structure more suitable with applications where restructuring is costly and search operations are too few.

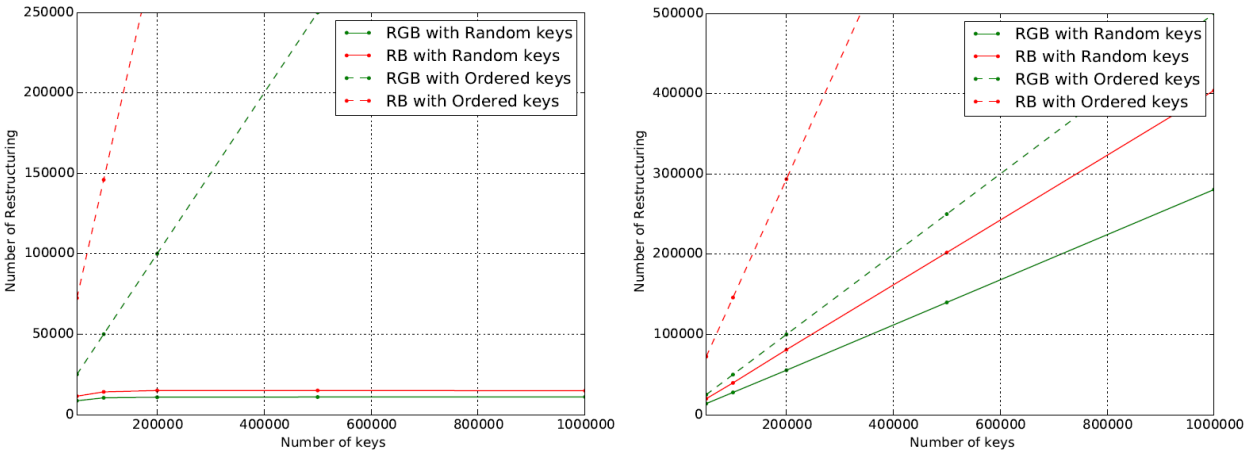


(a) Scenario 1



(b) Scenario 2

Fig. 6. Second phase time.



(a) Scenario 1 (b) Scenario 2

Fig. 7. Number of restructuring in phase 2.

7. Conclusion

In this paper, we proposed a new and explicit form for extending Red Black trees called Red Green Black trees (RGB trees). We detailed the insert/delete procedures to expose the different aspects and cases of the update operations. The RGB trees propose relaxed update operations thanks to the tolerated imbalance and the decrease in the number of restructuring. This structure is very interesting as a mean to define a faster update RB tree. The relaxation led to some degree of imbalance, increasing the height of the tree and slightly enlarging the search time. The increase is explained by the number of Red nodes where the worst case height is *3log(N + 3) - 3log(3)* and a little number of restructuring is needed *log(N + 3) - log(3)*. These results are in accordance with the conducted experiments. In fact, when the generated keys are random, the RGB trees give very fast update without losing much in balance. This characteristic is very useful when implementing schedulers which by nature have costly restructuring. It is true that the search time is increased comparing to RB trees. This is of course because of the increased height of the tree that could lead to worst case tree of the form of vines. However, this doesn't affect the update time because of the great gain in restructuring. Furthermore, in a randomized key environment, the experiment results assured that this case is quite improbable. This characteristic defines a major gain in the scheduler environment especially when there is concurrency of access and some relational condition to maintain. More investigation on this result would be appreciated between the research communities.

This structure can be extended to higher number of used colors, defining a generalized form of the RB trees where we can easily control of imbalance of the structure and adapt it to various environments. This generalization would lead to faster updates as the number of restructuring is decreased but would also provoke a considerable slower search time.

References

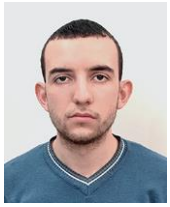
[1] Landis E. M., & Adelson-Velskii, G. (1962). An algorithm for the organization of information. *Doklady Akademii Nauk USSR, 146(2),* 263–266.

[2] Guibas, L. J., & Sedgewick, R. (1978). A dichromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science* (pp. 8–21).

[3] Foster, C. (1973). A generalization of avl trees. *Commun. ACM, 16(8),* 513–517.

[4] Bayer, R., & McCreight, E. (1970). Organization and maintenance of large ordered indices. *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control, SIGFIDET '70* (pp. 107–141).

[5] Karlton, P. L., Fuller, S. H., Scroggs, R. E., & Kaehler, E. B. (1976). Performance of height-balanced trees. *Commun. ACM, 19(1),* 23–28.
[6] Bayer, R. (1972). Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Inf., 1,* 290–306.
[7] Andersson, A. (1993). Balanced search trees made simple. *Proceedings of the 3rd Workshop on Algorithms and Data Structures* (pp. 60–71). Springer.
[8] Hanke, S., Ottmann, T., & Soisalon-Soininen, E. (1997). Relaxed balanced red-black trees. *Proceedings of CIAC: Vol. 1203. Lecture Notes in Computer Science* (pp. 193–204). Springer.
[9] Howard, P. W., & Walpole, J. (2014) Relativistic red-black trees. *Concurrency and Computation: Practice and Experience, 26(16),* 2684–2712.



Seyfeddine Zouana is a Ph.d student at high School of Computer Sciences, Algiers (ESI: Ecole Supérieure d'Informatique), which specializes in data structures. He graduated as a computer sciences engineer and received his master degree in 2013.

Djamel Eddine Zegour is a doctor in the Paris Dauphine University and a professor in High School of Computer Science, Algiers (ESI: Ecole Supérieure d'Informatique), which has thirty years of experience, specializing in data structures and programming paradigms. He is the author of several scientific publications and educational software.