ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/342869509

A revisited representation of the Red-black tree

Article *in* International Journal of Computer Aided Engineering and Technology · January 2020

DOI: 10.1504/IJCAET.2022.10043175

CITATIONS
0

READS
228

2 authors:

Lynda Bounif
Ecole Nationale Supérieure d'Informatique
6 PUBLICATIONS **9** CITATIONS
SEE PROFILE

Zegour Djamel Eddine
Ecole Nationale Supérieure d'Informatique
56 PUBLICATIONS **217** CITATIONS
SEE PROFILE

All content following this page was uploaded by Lynda Bounif on 16 January 2024.

The user has requested enhancement of the downloaded file.

A revisited representation of the red-black tree

Lynda Bounif* and Djamel Eddine Zegour

LCSI Laboratory,
Ecole nationale Supérieure d'Informatique (ESI ex INI),
Algiers, Algeria
Email: l_bounif@esi.dz
Email: d_zegour@esi.dz
*Corresponding author

Abstract: Nowadays, the red-black tree is probably the most used balanced data structure. It searches, inserts and deletes keys in O log (n) time and it is highly recommended for applications with frequent updates. However this balanced binary tree is complicated in the implementation because of the different cases in both the insertion and deletion and the use of the red and black colours. That's why we propose in this paper a simpler representation of this structure. The new representation uses one extra bit in order to gain simplicity and efficiency. The red-black tree is seen as a partitioned binary tree and uses instead of colours three kinds of subtrees: 01, 10 and 11 subtrees. We show on one hand that we can simply express the insert algorithm and on the other hand that the performance of the algorithm is slightly better. Efficiency is due to the fact that the search branch used in the new representation is reduced by half. Therefore this representation can replace easily the standard algorithms of red-black tree in practice.

Keywords: binary search trees; balanced trees; red-black tree; data structure; search trees; partitioning; algorithms; efficiency; performance.

Reference to this paper should be made as follows: Bounif, L. and Zegour, D.E. (xxxx) 'A revisited representation of the red-black tree', *Int. J. Computer Aided Engineering and Technology*, Vol. X, No. Y, pp.xxx–xxx.

Biographical notes: Lynda Bounif is a PhD student at the High School of Computer Science in Algiers, Algeria [Ecole nationale Supérieure d'Informatique (ESI ex INI)]. She works on data structures and algorithms. She received her master's degree in the same school in 2014.

Djamel Eddine Zegour is a Professor and Director of LCSI Laboratory in the High School of Computer Science [Ecole nationale Supérieure d'Informatique (ESI ex INI)]. He received his PhD from Paris Dauphine University, France in 1988. He is an author of many books and publications in the field of algorithms and data structures.

Copyright © 20XX Inderscience Enterprises Ltd.

1 Introduction

The design of balanced trees is a crucial invention in the field of data structures triggered by the poor worst case performance of binary search trees (Sedgewick, 2002). In a balanced tree, the depth of all the nodes is kept logarithmic to the size of the entire tree. AVL tree was the first balanced data structure invented in 1962 (Adel'son-Vel'skii and Landis, 1962) with a maximum height of 1.44 Log (n), where n is the number of nodes. It gives very satisfying results particularly in search operations. After that many types of generalisation of this balanced binary tree were proposed (Foster, 1965, 1973).

A few years later John Hopcrof proposes the 2–3 trees which are considered as an amelioration of AVL trees (Sedgewick and Wayne, 2011). The 2–3 tree is a multiway tree where every node has either two or three children, the binary tree form of this data structure was made just after two years by Bayer (1972) naming his balanced tree: 'symmetric binary B-trees'. This work is considered as the original version of the red-black tree: the balanced binary tree proposed later by Guibas and Sedgewick (1978). The authors give a new form of the tree by using two colours: red and black to represent the balance in the tree with O (Log n) average and worst case search, insertion and deletion algorithms.

Red-black trees are powerful data structures used primarily for symbol table implementations within Python, Java, C++ and many other modern systems, it is also used in dictionaries and the associative arrays. Recently it has respond to some problems in real time systems, like the problem of synchronisation cycle and the time consumption of updating and searching operations in the real time data addressing of control system (Sicheng et al., 2014). It is also used to solve the problem of storing an ordered list (Holenderski et al., 2014), for arranging virtual memory area of Linux (Zhang and Liang, 2010) and for an improved version of K-means clustering algorithm which outperforms the traditional K-means algorithm in terms of running time (Kumar et al., 2011).

A simplification of the red-black tree is more than necessary because of the difficulty and complexity of its code in the implementation, AA tree (Andersson, 1993) was a powerful simplification of the red-black tree, it uses split and skew procedures to rebalance the tree while maintaining the same performance, it represents well a red-black tree by making a restriction that a red node can only be a right child. Another alternative to give simpler algorithms were introduced in the same year. It is a simple and easy data structure comparing to the red-black tree (Smith and Graham, 1993), its idea is inspired from the 2–3 tree, the original structure of the red-black tree, with some similarities with the binary search tree and different balancing algorithms. This tree used the field type instead of the colour in order to specify whether the right son of the node is horizontally or vertically linked. A few years later an implementation example is in (Wiener, 2005) where a complete C# implementation of red-black trees is presented including all the implementation details for insertion and deletion and then a revisited version of red-black trees has been proposed later (Sedgewick, 2008) where the code is considerably reduced. Moreover several simple implementations of red-black trees can be found in Okasaki (1999) and Kahrs (2011). However all these types of improvements require in general some limited restrictions of the original tree and do not represent completely the original structure which still remains powerful.

The design space and analysis of balanced trees is still a very rich area, not yet fully explored; recently, the authors in the paper (Haeupler et al., 2015) presented a new common framework of balanced search trees under the name: rank balanced tree, in addition, they presented a new self-balanced binary data structure: weak AVL tree which combines the best proprieties of both AVL and red-black tree and can handle search, insertion and deletion operations in O (log n) time. This balanced tree structure is highly recommended in real time systems, but in addition to the obligation to satisfy a large number of inequalities corresponding to the number of updates; this data structure is not implemented in practice. A generalisation of the weak AVL tree was introduced after one year (Sen et al., 2016), it studied the possibility to rebalance just after insertions not deletions but uses log log n + O(1) as a balance information per node with periodic rebuilding.

In order to explore more the red-black tree structure, recent research works are done, for example a note on the largest number of red nodes in red-black tree is presented (Zhu et al., 2017), it gives at the first time an O (n2log n) time dynamic programming solution to calculate the number of red nodes and then improves the result to O (log n) recursive and non-recursive algorithms. Another study on red-black trees consists on the extension to red-black trees (Zouana and Zegour, 2018), where the balanced tree uses three different colours instead of two: green, red and black, it tolerates some imbalance in the tree which decreases the number of restructuring operations in practice but it increases the height of the tree and then influences the search time.

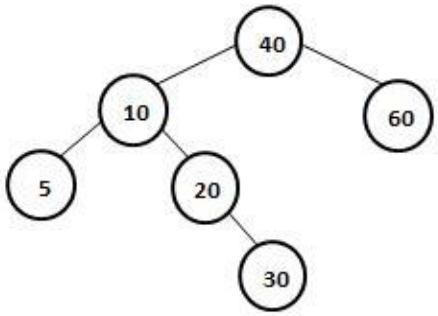
Our main purpose in this work is to represent the red-black tree structure in a different and simple way while preserving the logarithmic complexity of the insert and delete operations. The novel representation defines a mathematical partition of the whole tree on a set of subtrees of depth smaller or equals to 1. Instead of using the colour as a balance parameter in the red-black tree, we use two bits to distinguish between root nodes of subtrees and the other nodes. We will show on the one hand that with such a representation we can easily and simply express algorithms compared to red black tree ones, and on the other hand that the operations performances are slightly better. The rest of the paper is structured as follows: Section 2 introduces some preliminaries on the trees and types of trees; Section 3 will present the idea of our new balanced tree, and Section 4 gives the proposed insertion algorithm in details. Section 5 shows the results of the implementation of our proposed tree and the discussion by comparing with the red-black tree. Finally, Section 6 makes a conclusion and looks forward to the future research.

2 Preliminaries

2.1 Binary search tree (Cormen et al., 2009)

A binary search tree is a tree organised in a binary tree structure, where each node contains a key and a data, with pointers to left and right children, and also to its parent node. When a pointer is missing, the attribute contains the value NIL. We note that the root node has a NIL parent. In the binary search tree the keys are stored such as they respect the binary search tree propriety: Let x a node in a binary search tree, y and z nodes in the left and right subtrees of x respectively, then: y.key $\leq$ x.key and z.key $\geq$ x.key.

Figure 1 Example of a binary search tree



Operations on a binary search tree

The most important operations are the search, insertion and deletion operations: firstly the search operation of a key k starts from the root, and create a path downward in the tree, for each node traversed x; we compare its key with the key k, if k is smaller than x.key, we continue the search in the left subtree of x, symmetrically if k is greater than x.key, the search continue in the right subtree. The running time of the search operation is O (h) where h is the height of the tree.

The insertion and deletion operations are done after a search operation; the insertion of a new node n with key k is always done in the leaf node, after searching for its key in the tree, and this key does not exist in that tree, we insert it as a left or right child or a leaf node.

The deletion operation of a node n has three different cases:

Case 1 If n has no children then we simply remove it by modifying its parent to replace n with Nil as its child.

Case 2 If n has just one child we modify n's parent to replace n by n's child.

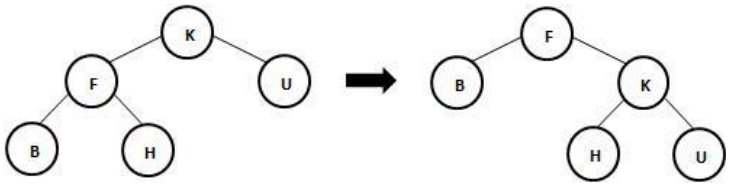
Case 3 If n has 2 children then we find n's successor y, we replace n by y, and then delete n.

A binary search tree is efficient only if it is balanced, this balance is ensured by restructuring operations, and we define in the section below their principle.

Restructuring operations

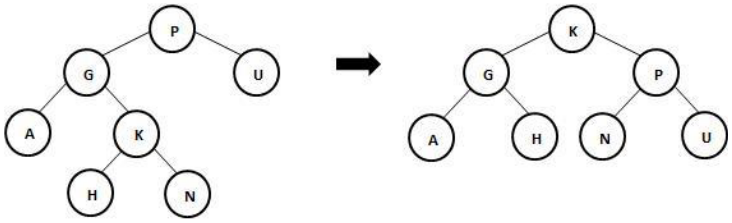
In order to keep the binary search tree as balanced as possible, we need to restructure the tree search time when it becomes unbalanced, we call these operations: the restructuring operations. In this paper we present two kind of restructuring operations which are done after an insertion of a new node in the initial tree or a removal of a node from it. *A single rotation:* Figure 2 shows an example of a single rotation.

Figure 2 Single right rotation of node K



A double rotation: Figure 3 indicates a double rotation.

Figure 3 Double rotation: left oration of node P followed by right rotation of node K



We denote by Dir the direction of a rotation. So, Dir = 1 means a right rotation while Dir = 0 denotes a left rotation. If x references a node then x. Link[0] denotes a left child of node x while x.Link[1] denotes the right child of node x.

The C code of the rotation is the following:

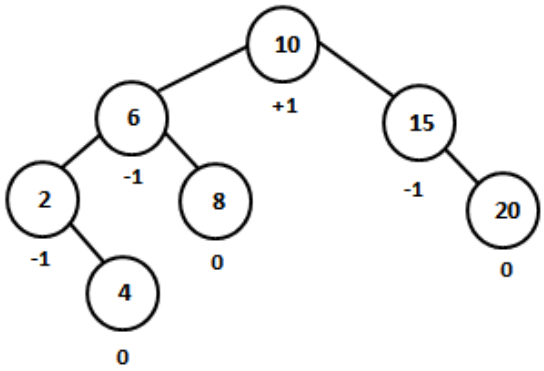
```
Rotate (x, dir)
{
    save = x->link[!dir];
    x->link[!dir] = save->link[dir];
    save->link[dir] = x;
    return save;
}
```

As examples, Figure 2 corresponds to a right rotation of node K, so we apply rotate (K, 1) while Figure 3 corresponds to a left rotation of node K followed by right rotation of node P, so we apply Rotate (K, 0) and Rotate (P, 1) respectively.

2.2 AVL tree

It is the first balanced binary data structure. It is named after its two inventors, Adel'son-Vel'skii and Landis (1962). In an AVL tree, the heights of the two child subtrees of any node differ by at most one, so each node alone in the tree represents an AVL tree, A balance factor is then added in each node in order to maintain the balance of the tree, it can takes only the values: 0, –1 or +1, so the balancing actions have to be done when the balance factor becomes 2 or –2 (Joshi, 2010). Figure 4 gives an example of AVL tree.

Figure 4 Example of an AVL tree



We can note that operations of search, insertion and deletion take O log (n) time in the average and the worst case. Also the rebalancing operations consist of tree rotations, which restore the unbalanced tree after an update operation. Two rotations at most suffice to restore the tree in the insertion case, while a deletion can take O log (n) rotations.

2.3 Splay tree

This is an intriguing height balanced binary tree since it does not require the storage of any balance information; however its height could be linear in the worst case and this is the major disadvantage of this data structure (Sleator and Tarjan, 1985). The restructuring operation of a splay tree is called splaying and the main goal is to move the recently accessed items to the root by doing rotations bottom-up along the access path.

To splay a tree at any node x we repeat the three cases below until x becomes the root:

- Zig case: it is applied if the parent of x is the root, a single rotation is performed to make x the new root and the process terminates.
- Zig-Zig case: when the parent of x is not the root and x and p(x) are either both right or both left children, two rotations are applied, the first consists on rotating the edge joining p(x) and its parent, and then the second rotation deals with the edge joining x with p(x).
- Zig-Zag case: this case occurs when x is not the root and x is a left child and p(x) is the right child or vice versa, in this case we rotate the tree on the edge between x and p(x) and then between the edge resulting between x and its grandparent.

2.4 Weak AVL tree

This binary balanced search tree was recently discovered (Haeupler et al., 2015), it uses the ranks instead of the heights and combines the best proprieties of AVL and red-black tree. Every AVL tree is a weak AVL tree, and every weak AVL tree can be assigned colours in order to get a red-black tree. In a weak AVL tree each node x has an integer rank: R(x) such that the rank of its parent is greater that its rank: R (p) > r(x). We define

the rank difference of a node x as the rank of its parent minus its rank, so we have: RD(x) = R (P) – R(x).

The weak AVL tree must respect theses three rules:

- every external node has rank 0
- the rank difference of any node expect the root is 1 or 2;
- an internal node with two external children cannot be a 2, 2 node.

Figure 5 Example of a weak AVL tree

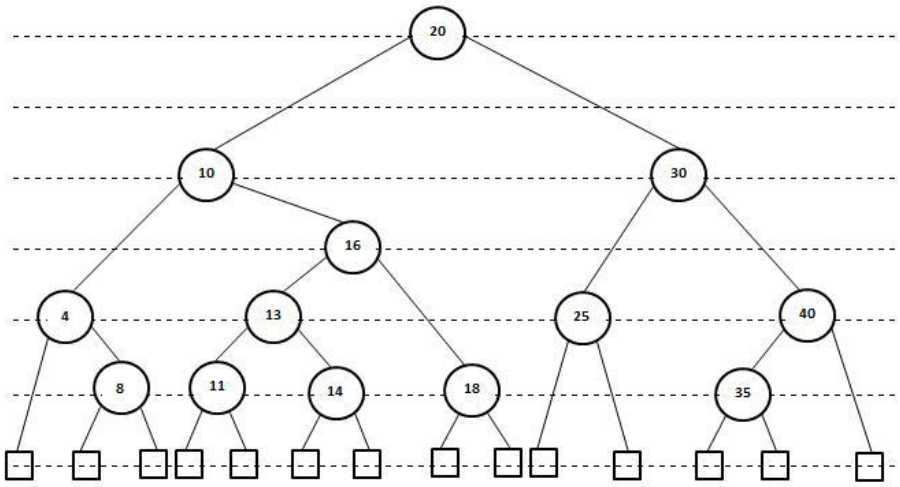


Figure 5 shows an example of a weak AVL tree with rank = 5. We notice that: the node root is 2, 2 node while the node 10 and 30 are 1, 2 and 2, 2 respectively.

2.5 Relaxed AVL tree (ravl)

This tree is motivated by a real-time data base application storing customer information. The database used at the beginning a red-black tree which causes a failure of the whole system. Ravls trees are issued from the rank balanced tree framework, such that the rank difference between a node and its parent can be non-constant. The restructuring operations are just done after insertions not deletions; however the worst case access time remains logarithmic in the number of insertions (Sen et al., 2016). Ravls trees needs periodic rebuilding, and saves log log m + 1 bits of balance information per node, with m is the number of insertions.

2.6 Red-black tree

A red-black tree is a binary search tree with one extra bit of storage per node: its colour, which can be either red or black. In a red-black tree, any path contains the same number of black nodes and does not contain two consecutive red nodes. A colour field is then

added in each node. A binary search tree is a red-black tree if it satisfies the following red-black properties (Edmonds, 2008):

- every node is either red or black
- the root is black
- every leaf (NIL) is black
- if a node is red, then both its children are black.

For each node, all paths from the node to descendant leave contain the same number of black nodes.

Lemma (Cormen et al., 2009): A red-black tree with n internal nodes has height at most 2 Log (n+1).

Proof: the proof is divided in two parts, the first one consists on showing that a subtree rooted at any node n has at least $2^{bh(n)} - 1$ internal node and then the second part complete the proof considering h the height of the global tree. *Part 1:* the proof is by induction on the height of n.

Hypothesis: 'Each internal node n has at least $2^{bh(n)} - 1$ internal node'.

If n is nil then its height is 0 indeed the subtree rooted at n contains $2^{bh(n)} - 1 = 2^{\circ} - 1 = 0$ internal nodes.

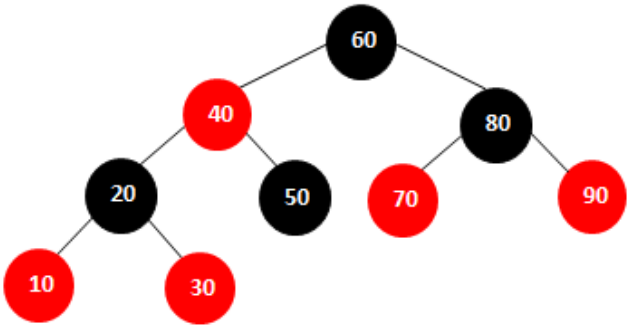
For the inductive step, we consider an internal node x with two children having black-height of bh(n) or bh(n) – 1 depending on its colour. Considering ch(n) the child, applying the hypothesis, it has at least $2^{bh(n)-1} - 1$ internal nodes. Thus the subtree rooted by n contains at least: $2^{bh(n)-1} - 1 + 2^{bh(n)-1} - 1 + 1 = {}^{2bh(n)} - 1$ internal nodes.

Part 2: let H be the height of the whole tree. The number of black nodes on any simple path to a leaf having x nodes is at least x/2. Thus the black height of the root must be at least h/2 thus:

$$N \geq 2^{h/2} - 1 \rightarrow N + 1 \geq 2^{h/2} \rightarrow \text{Log}(N+1) \geq h/2 \rightarrow h \leq 2\ \text{Log}(N+1).$$

An example of a red-black tree can be shown in Figure 6.

Figure 6 A red-black tree example insertion in a red-black tree (see online version for colours)



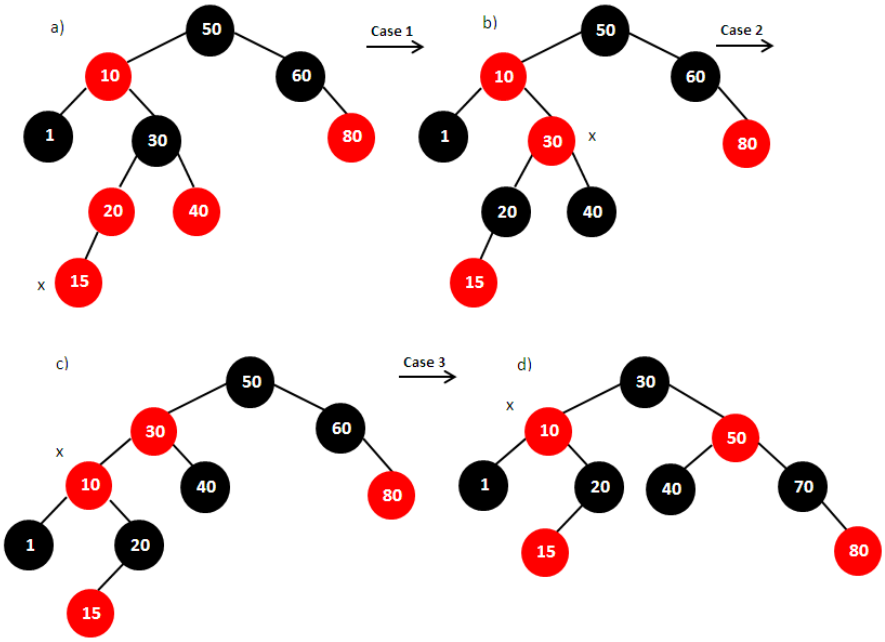
In order to deal with the red-black tree in coding we need to consider the T.NIL as a black node, holding arbitrary left, right and key values. We call it a black sentinel.

A newly element is always inserted as a red leaf. If its parent is also red, a maintenance algorithm is applied. The insert algorithm follows then three cases:

- the case 1 consists of a colour flip, when the sibling of the newly inserted node is red
- the case 2 and 3 are both performed when this sibling node is black, the process is repeated since the newly inserted node x is not a root node and the colour of its parent is red.

We can notice that the root of the global tree becomes black and the end of the algorithm. Suppose that we insert the node x with the value 15 in the tree, this node will have the colour red as mentioned in Figure 7(a), since its parent having the key 20 is also red, the violation of propriety 3 occurs, in this case the sibling of the node x having the key 40 is also red, so we are in the case 1: we recolor the grand parent of x and its children, then the pointer x is moved up the tree, as mentioned in Figure 7(b). In Figure 7(b) we remark that x and its parents are red but x's sibling is black, and since x is the right child of its parent, the case 2 is applied: a left rotation is done [see Figure 7(c)].

Figure 7 Insertion cases in a red-black tree (see online version for colours)



Now since x is the left child of its parent the case 3 can be applied, we perform a right rotation, and this gives the tree in Figure 7(d). In Figure 7(d) the tree is a legal red-black tree, so the insertion fixup terminates.

The insertion is preceded by the search algorithm which stores the path traversed from the root to the parent of the new node. We use Stack and dir as two tables containing all the nodes traversed from the root to the parent of the newly inserted node, and the directions left or right taken respectively. Let H be the index of the two tables.

We give here after a C implementation of the fixup function which restores the red-black tree proprieties proceeded by a pseudo code of the research process (see online version for colours).

```
Search for a given value Value in a tree having as root Root.
If value is not found, a stack is built with the (stack, dir) couples of the traversed subtrees from
the tree root. Otherwise, exit.
Create node n with value Value and a red colour and link it to its parent stack [H -1] If the
stack is empty Root = n; Exit
If the parent of the newly inserted node is black, there is no violation and the algorithm
terminates, else we call the fixup function here after:
bool stop = false ;
While (stack [H -1] ->colour == 1 && !stop)
{
    d = (dir[H - 2]) == 0 ? 1: 0;
    xPtr = Stack[ H - 2];
    yPtr= xPtr->link[d]                    //yPtr is the sibling
    if (sibling->colour == 1)              //Case 1
      {
        xPtr->colour = 1;
        P->colour = 0;
        yPtr->colour = 0;
        H = H – 2;
      }
      else
{
        if (dir[H – 1] != d)               //Case 2: No double rotation
        {
          yPtr = stack[H – 1];
        }
        else                               //Case 3: double rotation
        {
          xPtr = stack[H – 1];
          Single_rotate (xPtr,!d) ;
        }
        xPtr = stack[H – 2];
        xPtr->colour = 1;
        yPtr->colour = 0;
        Single_rotate (xPtr, d);
```

```
            if (xPtr == root)
            {
               root = yPtr;
            }
            else
            {
               stack[H – 3]->link[dir[H – 3]] = yPtr;
            }
            stop = true;
         }
      }
   root->colour = 0;
```

The main disadvantage of the red-black tree is the complexity of implementation due to the use of the red and black colours and its different cases, which lead to the use of the standard library implementation like STL set in C++ or tree set in Java. In addition when we plan to build the red-black tree only once and then perform only search operations, AVL tree gives better performance in this case and then it is recommended to use the AVL tree rather than the red-black tree.

3 Contribution

3.1 Description

The main idea of the new representation is the partitioning of the whole tree in small subtrees of depth 0 or 1. Any subtree can hold one node, two nodes or three nodes. The intersection of any two subtrees is empty and the union of theses subtrees gives the global tree. Any root of a subtree is called a compound node and any child – if any – of a compound node is called a single node.

Two bits are used to represent the different kinds of nodes in the tree. So, we define four types of nodes:

1 00-node: denotes a single node

2 01-node: denotes a compound node with no children

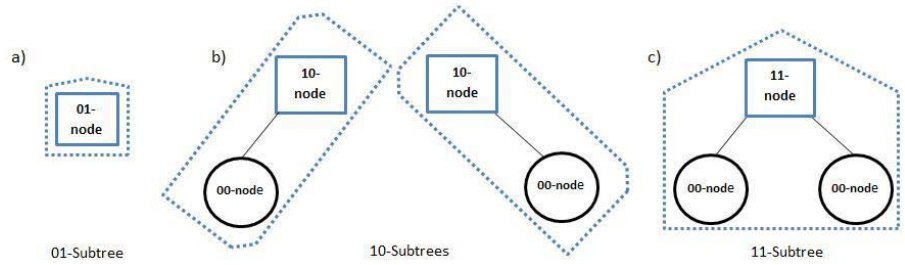
3 10-node: denotes a compound node with exactly one child

4 11-node: denotes a compound node with two children.

Figure 8 shows the kinds of subtrees and nodes. The square designates a compound node and the circle a single node. The surrounded nodes together represent subtrees. The type of a subtree is the type of its root.

Now, we can formally give two simple ways for defining the new representation.

Figure 8 Kinds of subtrees and nodes (see online version for colours)

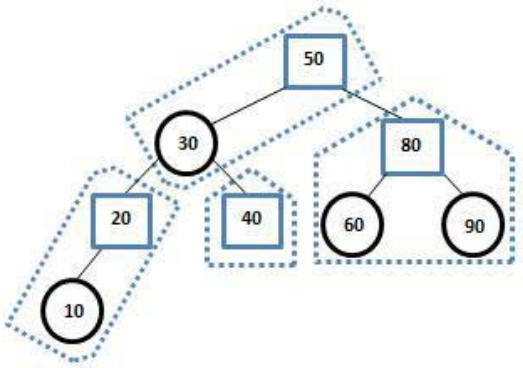


Definition 1 (in terms of nodes)

- a tree node can be 00-node, 01-node, 10-node or 11-node
- each path from any node to its descending nodes leaves must have the same number of compound nodes (01-node or 10-node or 11-node)
- every subtree can have at most three nodes. The root is a compound node and its children are single nodes
- every path from any subtree to any descendent leaf subtree must have the same number of subtrees.

Figure 9 denotes an example of the proposed tree.

Figure 9 Example of the proposed tree (see online version for colours)



Some terms on the new representation are useful to describe the insert algorithm. We can define them by using examples as follows:

- the subtree rooted at 50 denotes a root subtree
- the subtrees rooted at 20, 40 and 80 denote leaf subtrees
- the subtrees rooted at 20 and 40 denote sibling subtrees
- the subtree rooted at 80 has not a sibling subtree
- the subtree rooted at 50 is the parent subtree of the subtree 80.

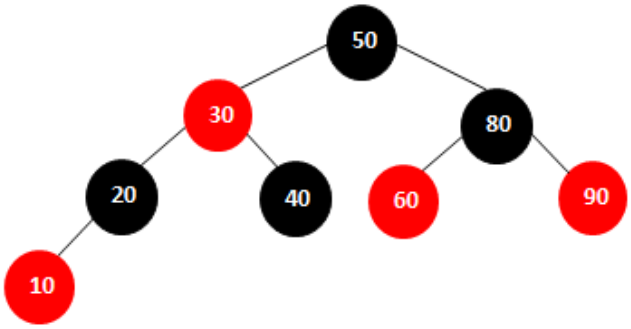
Lemma: The maximum height of the proposed tree is 2 Log n.

Proof: The minimum number of any node in the tree of height h satisfies the recurrence: $n_0 = 1$, $n_2 = 2$, $n_3 = 4$, $n_k = 2* n_{k-2} + 1$ for any k >= 2. By induction $n_k >= 2^{h/2}$ which gives: h <= 2 log n_k.

3.2 The new representation versus red-black tree

The proposed tree is a balanced binary tree respecting all red-black tree rules but in a different way of representation. Indeed, a 00-node denotes a red node. An 11-node, 10-node or 01-node denotes black nodes. Implicitly, the root of the proposed tree is a compound node (11-node or 10-node or 01-node). The root of a red-black tree is a black node. Furthermore, since in any path of the proposed tree, we have subtrees, thus it is not possible to have two consecutive single nodes (00-nodes). Two consecutive nodes of a red-black tree cannot be red.

Figure 10 The red-black tree corresponding of the tree in Figure 9 (see online version for colours)



We do not have to consider the nil nodes as compound nodes, while in a red-black tree they are considered as black nodes (sentinels).

4 The insertion algorithm

A new element is always inserted into a leaf subtree. If the depth of this latter does not exceed one, the insert process terminates. Otherwise, a violation occurs. It is solved as follows:

- If the compound node had only one child (10-subtree), a single rotation or a double rotation is made. The insert process terminates.
- If the compound node had two children (11-subtree), two subtrees are created rooted by the two children. The old compound node is integrated to the parent subtree as a single node. This can involves a new violation in the parent subtree which is solved in the same manner. The insert process can continue in cascade.

Figure 11 Insertion of a new element without violation (see online version for colours)

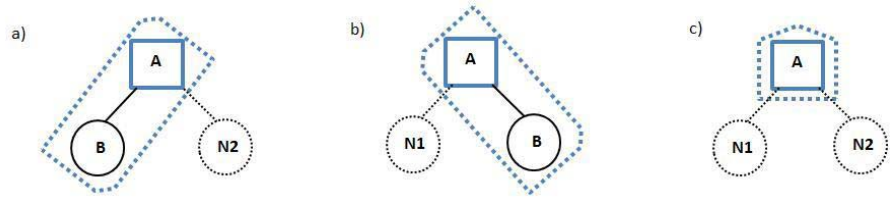
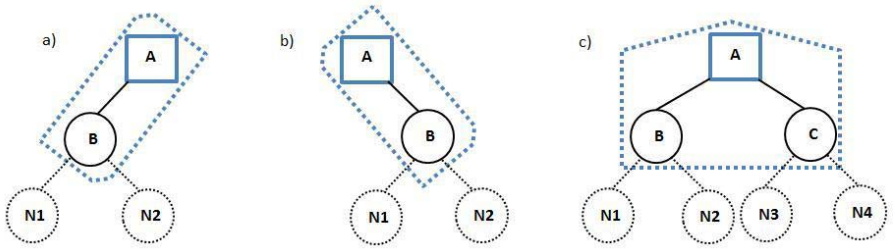


Figure 12 Insertion of a new element with violation (see online version for colours)



For more details, Figure 11 and Figure 12 give the different cases. The circled dashed lines denote the new inserted element. Several circles mean several possible cases. In the case of a leaf subtree N1, N2, N3, N4 represent newly created nodes, otherwise they represent compound nodes.

In the cases of Figure 11, the new element inserted does not involve a violation since it can be added to the subtree while preserving its depth. The insert process terminates.

However, in the cases of Figure 12, the new element inserted causes a violation as the depth of the subtree becomes greater than 1. It can be solved as follows:

a If the leaf subtree is a 10-subtree [Figure 12(a) and 13(b)], a restructuring operation is performed. The insert process terminates. It consists of reorganising the subtree by making a single or a double rotation. Figures 13(a) and 14(b) gives the result of the restructuring operation for the case of Figure 13(a) when inserting n1 and n2 respectively.

b If the leaf subtree is an 11-subtree [Figure 12(c)], a partitioning operation is performed. Two subtrees are created rooted by the two children. The old compound node is integrated to the parent subtree as a single node. This can involve a new violation in the parent subtree which is solved in the same manner. The insert process can then continue in cascade. Figure 14 gives the result of the partitioning operation for the case of Figure 12(c) when inserting n1.

Figure 13 The restructuring operations (see online version for colours)

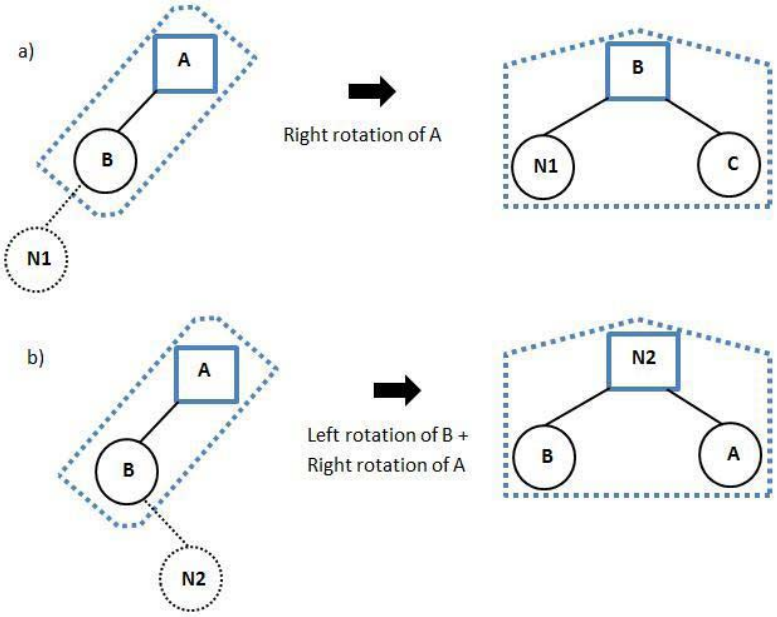
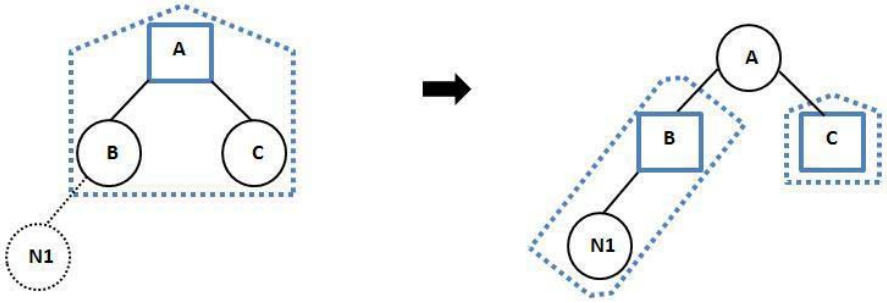


Figure 14 The partitioning operation (see online version for colours)



We can notice that inserting a new element is always preceded by a search step. A stack REP is then built to save the path from the root to the leaf. In our context, we save only the traversed subtrees. For each subtree we save the couple (r, Dir), r denotes the root of the subtree and Dir the direction from node r for which we move down the tree. Let K be the index of the tables.

The insert algorithm is the following (see online version for colours):

Search for a given value Value in a tree having as root Root.

If value is not found, a stack is built with the (r, Dir) couples of the traversed subtrees from the tree root. Otherwise, exit.

Create node n with value Value and type 00

If the stack is empty Root =n; Exit

```
Bool stop =False;

r = REP[K – 1]; Dir = dir[K – 1];

while (!Stop)
{
    if(r->link[Dir] == NULL || r->link[Dir]== n)      // (1) Cases without violation {

       r->link[Dir] = n ;
       if(r->type == 01) r->type = 10; else r->type = 11;
       stop = true;
    }
    else                                              // Cases with violation
    {
        Add (n, r.Link[Dir] )
        if (r->type == 10)                            // (2) Restructuration
        {
           if ( (r->link[Dir]->link[!Dir] != NULL) &&
              (r->link[Dir]->link[!Dir]->type == 0) ) //(3)
                 r->link[Dir] = single_rotate (r->link[Dir], Dir);
           q = single_rotate(r, !Dir); sauv = r;
           r->type = 0;
           if (q->link[!Dir] != NULL) q->type = 11 ; else q->type = 10;
           if (K <= 2) Root = q;                      //(4)
           else
           {
              K--;
              r = REP[ K – 1];
              Dir = dir [ K – 1];
              if(r->link[Dir] != sauv) Add (n, r.Link[Dir] )
              else r->link[Dir] = q;
           }
           stop = true;
        }
        else                                          //Partition
        {
           r->link[Dir]->type = 10;
           r->link[!Dir]->type = 01;
           if (K <= 2)                                //(5)
           {
              Root = r;
```

```
            r->type = 01;
            stop = true;
          }
          else                                  // Cascading
          {
            r->type = 00;
            n = r; K--;
            r = REP[K – 1];
            Dir = dir[K – 1];
          }
        }
    }
}
Void Add(n, r)                                  // Add node n to the left or right of node r
    If (r->data < n.data) r->Link[0] = n;
```

4.1 Comments

If the search failed, a new node n is created with the new key to be inserted. We attribute to it a 00 type.

If the stack is empty, this means that it is the first item to put in the tree. Node n becomes then the root of the tree and the algorithm terminates.

For each iteration of the loop, the algorithm begins by linking the new element n into the current subtree having r as root. If there is no violation [test (1)], the algorithm terminates.

The first condition of test (1) denotes the fact that the newly element is inserted in a leaf subtree while the second condition of test (1) can be satisfied after a partition operation and more precisely when we have a cascading, it stipulates that r point directly the node newly integrated to a no leaf subtree.

If test (1) is not satisfied, a violation occurs. The new element is first linked to the current subtree. If the type of the latter subtree is 10 [test (2)], a restructuring operation is performed.

The test (3) means that the new element is not inserted (or not integrated for a no leaf subtree) in the same direction of node r.Link [Dir]. It is the case of a double rotation.

A restructuring operation is always followed by updating the parent subtree if this exists.

Otherwise, q becomes the new root of the tree [test (4)].

The loop continues only when a partitioning operation is performed because the root of the current subtree becomes a single node and will be therefore linked to the parent subtree. To perform a new iteration of the loop, n becomes the old root r and we need to pop the stack to have the root and the Dir of the parent subtree.

4.2 Example of the insertion

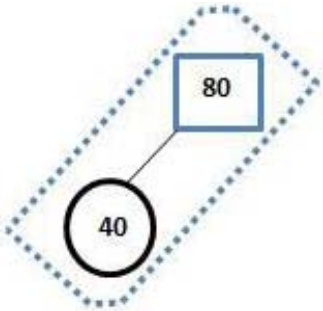
For more clarification of the proposed algorithm we give here after an example of the insertion of some elements.

- Insert 80: a leaf subtree is created with one node.

Figure 15 The tree after inserting the key 80 (see online version for colours)

- Insert 40: searching gives (r = @80, Dir = 0) as stack content. test (1) is satisfied. 40 is inserted into the leaf subtree rooted at 80.

Figure 16 The tree after inserting the key 40 (see online version for colours)

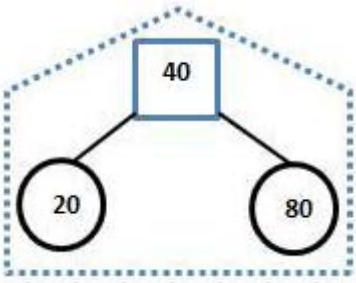


- Insert 20: searching gives (r = @80, Dir = 0) as stack content.

  Test (1) is not verified and the subtree type is 10. A violation occurs, which is solved by a restructuring operation (test (2)). In this case a right rotation of node 80 is performed because [test (3)] is not satisfied.

  As there is no parent subtree, 40 is the new root.

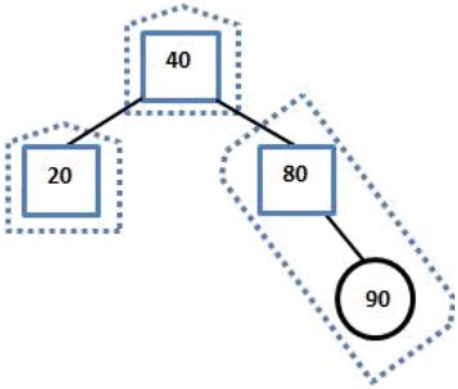
Figure 17 The tree after inserting the key 20 (see online version for colours)



- Insert 90: searching gives (r= @40, Dir = 1).

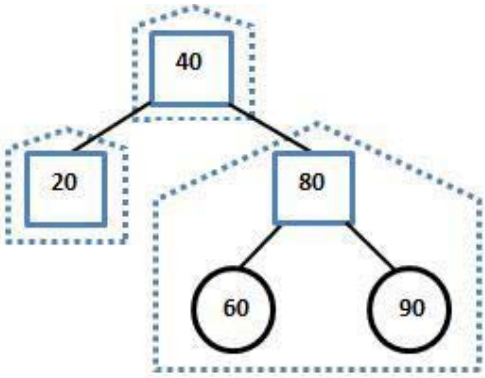
  90 is inserted into the leaf subtree with root 40. Test (1) is not satisfied, therefore a violation occurs. As the type of node 40 is 11, a partitioning operation is performed. Nodes 20 and 80 become compound nodes and 40 becomes a single node. Test (5) is satisfied, 40 becomes again a compound node.

Figure 18 The tree after inserting the key 90 (see online version for colours)



- Insert 60: 60 is inserted into the leaf subtree 80.

Figure 19 The tree after inserting the key 60 (see online version for colours)



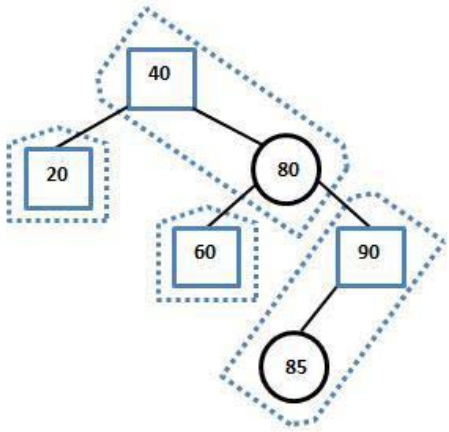
- Insert 85: searching gives (r = @40, Dir = 1), (r = @80, Dir = 1) as a stack content.

  The first iteration of the loop begins with r = @80 and Dir =1.

  Test (1) is not satisfied, therefore a violation occurs. As the type of node 80 is 11, a partitioning operation is performed. Nodes 60 and 90 become compound nodes and 80 becomes a single node. Test (5) is not satisfied, a second iteration of the loop is lunched again with n=@80, r=@40 and Dir=1.

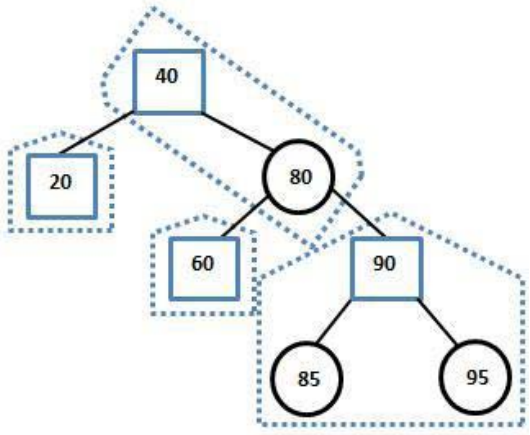
  Test (1) is satisfied, node 80 is added to subtree 40.

Figure 20 The tree after inserting the key 85 (see online version for colours)



- Insert 95: 95 is inserted into subtree 90 as a right child.

Figure 21 The tree after inserting the key 95 (see online version for colours)

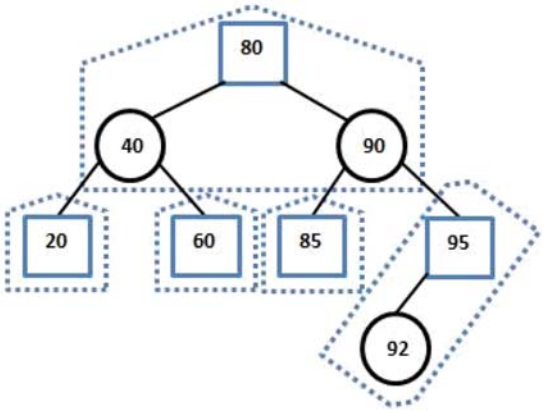


- Insert 92: searching gives (r= @40, Dir = 1), (r = @90, Dir = 1). The first iteration of the loop begins with r=@90 and Dir=1.

  Test (1) is not satisfied, therefore a violation occurs. As the type of node 90 is 11, a partitioning operation is performed. Nodes 85 and 95 become compound nodes and 90 becomes a single node. Test (5) is not satisfied, a second iteration of the loop is lunched with n=@90, r=@40 and Dir=1.

  Test (1) is not satisfied, a new violation occurs. As type of node 40 is 10 a restructuring operation is performed. A single left rotation of node 40 is made. Test (4) is satisfied, node 80 becomes the new root.

Figure 22 The tree after inserting the key 92 (see online version for colours)



Comparison between the two codes

- Both algorithms use a stack to save the search path and the left or right directions taken at each level.
- The search path used by the red-black tree holds all the nodes traversed in the search phase while the search path of the new representation holds only roots of traversed subtrees. This involves space saving of up to 50%. However, some additional tests are added.
- Reducing the length of the search stack will certainly contribute to the effectiveness of the insertion algorithm.
- The new representation uses four node types while the red-black tree structure uses only two types: red and black colours. This makes the insertion algorithm easier to code and to understand.
- As opposed to the red-black tree which deals with the global tree, the new representation deals with small subtrees linked together. This facilitates the understanding of the evolution of the insert process.

5 Experimental results

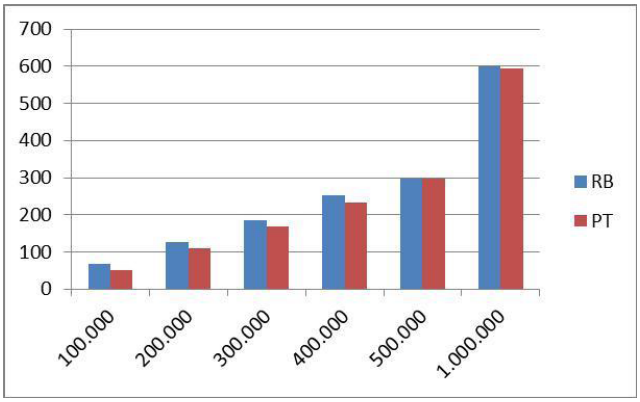
All the algorithms are implemented in C++ language. We considered the experiment where we build the red-black tree and the new representation by inserting random integer values starting from 100.000 integers to 1.000.000. We repeat theses operations ten times and we calculate the execution time of the insert process. At the end we calculate the average of the execution time of these operations.

Table 1 shows the execution times in milliseconds taken by each tree. The column N indicates the size of the tree already generated as well as the number of insertion operations. The other values in columns RB and PT represent the values of the average execution time of the operations.

Table 1 Simulation results of the insertion and deletion algorithms for both the red-black and the proposed tree

N	*RB (insert)*	*PT (insert)*
10.000	67	50
100.000	128	111
200.000	184	170
300.000	252	234
400.000	297	296
500.000	67	50

Figure 23 The insertion execution time for red-black tree and the proposed tree (see online version for colours)



As we can notice from the results in Table 1 and the graph in Figure 23 the new representation confirms it superiority comparing to the original balanced data structure, for example inserting 200.000 random values in the red-black tree takes 128 milliseconds while in the proposed tree, it takes only 111 milliseconds, and when we insert 400.000 random values in a RB tree, we take 252 milliseconds, while it consumes only 234 milliseconds in the proposed tree.

The execution time gained can be due to the reduction of the search path involved by the new representation. In our context this stack holds only the roots of subtrees rather than all the nodes, and then we gain half the space used for a stack in a standard red-black tree and the algorithm uses less iterations to insert one key in the tree.

The results obtained are very satisfactory since we did not pay any price for the simplicity using the joined subtrees.

6 Conclusions

We presented in this paper a simpler representation of the red-black tree, the most used self-balancing tree. It consists of a binary search tree partitioned in subtrees with depth zero or one having one, two or three nodes. On the one hand, we have shown that the proposed balanced tree is very simple to understand and to implement in practice. On the

other hand, we provided the insert algorithms of both the red-black tree and the proposed tree, and then we have provided the insert algorithms in C++ language.

As opposed to the red-black tree implementation we do not need to store all the nodes traversed in the search path to perform an insert operation. Only the roots of the subtrees are saved. This reduced by half the search path and has certainly contributed to the improvement of the insert algorithm as the simulation results show. We intend in the future to give more theoretical bounds on this structure and give the deletion algorithm.

References

Adel'son-Vel'skii, G.M. and Landis, E. (1962) 'An algorithm for the organization of information', English translation in *Soviet Mathematics Doklady*, Vol. 3, No. 5, pp.1259–1262.

Andersson, A. (1993) 'Balanced search trees made simple', *Workshop on Algorithms and Data Structures*, August, pp.60–71, Springer, Berlin, Heidelberg.

Bayer, R. (1972) 'Symmetric binary B-trees: data structure and maintenance algorithms', *Acta Informatica*, Vol. 1, No. 4, pp.290–306.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C. (2009) *Introduction to Algorithms*, MIT Press, USA.

Edmonds, J. (2008) *How to Think about Algorithms*, Cambridge University Press, USA.

Foster, C.C. (1965) 'Information retrieval: information storage and retrieval using AVL trees', *Proceedings of the 1965 20th National Conference*, ACM, August, pp.192–205.

Foster, C.C. (1973) 'A generalization of AVL trees', *Communications of the ACM*, Vol. 16, No. 8, pp.513–517.

Guibas, L.J. and Sedgewick, R. (1978) 'A dichromatic framework for balanced trees', *19th Annual Symposium on Foundations of Computer Science*, IEEE, October, pp.8–21.

Haeupler, B., Sen, S. and Tarjan, R.E. (2015) 'Rank-balanced trees', *ACM Transactions on Algorithms (TALG)*, Vol. 11, No. 4, p.30.

Holenderski, M., Bril, R.J. and Lukkien, J.J. (2014) 'Red-black trees with relative node keys', *Information Processing Letters*, Vol. 114, No. 11, pp.591–596.

Joshi, B.K. (2010) *Data Structures and Algorithms in C++*, Tata McGraw Hill Education.

Kahrs, S. (2001) 'Red-black trees with types', *Journal of Functional Programming*, Vol. 11, No. 4, pp.425–432.

Kumar, R., Puran, R. and Dhar, J. (2011) 'Enhanced k-means clustering algorithm using red black tree and min-heap', *International Journal of Innovation, Management and Technology*, Vol. 2, No. 1, p.49.

Okasaki, C. (1999) *Purely Functional Data Structures*, pp.11–14, Cambridge University Press, UK.

Sedgewick, R. (2002) *Algorithms in Java, Parts 1-4*, Addison-Wesley Professional, USA.

Sedgewick, R. (2008) 'Left-leaning red-black trees', *Dagstuhl Workshop on Data Structures*, September, p.17.

Sedgewick, R. and Wayne, K. (2011) *Algorithms*, 4th ed., Addison-Wesley Professional, USA.

Sen, S., Tarjan, R.E. and Kim, D.H.K. (2016) 'Deletion without rebalancing in binary search trees', *ACM Transactions on Algorithms (TALG)*, Vol. 12, No. 4, p.57.

Sicheng, A., Kehe, W., Huan, Z. and Yi, L. (2014) 'Advanced red-black algorithm for real-time data addressing of control system', *Sensors & Transducers*, Vol. 178, No. 9, p.292.

Sleator, D.D. and Tarjan, R.E. (1985) 'Self-adjusting binary search trees', *Journal of the ACM (JACM)*, Vol. 32, No. 3, pp.652–686.

Smith, P.E. and Graham, J.H. (1993) 'A simple balanced search tree', *Proceedings of the 1993 ACM Conference on Computer Science*, ACM, March, pp.461–465, ACM.

Wiener, R. (2005) 'Generic red-black tree and its C# implementation', *Journal of Object Technology*, Vol. 4, No. 2, pp.59–80.

Zhang, H. and Liang, Q. (2010) 'Red-black tree used for arranging virtual memory area of Linux, *International Conference on Management and Service Science (MASS)*, IEEE, August, pp.1–3.

Zhu, D., Wu, Y., Wang, L. and Wang, X. (2017) 'A note on the largest number of red nodes in red-black trees', *Journal of Discrete Algorithms*, Vol. 43, pp.81–94, Doi:10.1016/j.jda.2017.03.001.

Zouana, S. and Zegour, D.E. (2018) 'Red green black trees: extension to red black trees', *Journal of Computers*, Vol. 13, No. 4, pp.461–471.

View publication stats