ISSN 2007-9737

# Toward a Unique Representation for AVL and Red-Black Trees

Lynda Bounif, Djamel Eddine Zegour

Ecole Nationale Supérieure d'Informatique,
Laboratoire de la Communication dans les Systèmes Informatiques,
Algeria

{l_bounif, d_zegour}@esi.dz

**Abstract.** We propose a unique representation of both AVL and Red-Black trees with the same time and space complexity. We describe all the maintenance operations and also the insertion and deletion algorithms. We give the implementation of the proposed tree and the results. We then make a comparison of the three structures. The simulation results confirm the performance of the new representation relatively to AVL and Red-Black trees.

**Keywords.** Balanced binary trees, red-black trees, AVL trees, binary search tree, partitioning, data structures.

## 1 Introduction

Binary search trees are an efficient data structure for loads applications in computer science but have a poor worst case performance [1]. The good remedy for that is the perfect balanced tree with a height at most *log(n)* [2]. But unfortunately keeping a perfect balance of a tree is rude and so expensive in practice. One of the reason to introduce balanced trees is because of the costs are guaranteed to be logarithmic while ensuring that the tree remains almost balanced.

Balanced trees assure a random search, insertion and deletion operations in time proportional to *log(n)*. The first type of these trees is the AVL tree [2]; it is simple and deals well in lookup operations. After that many alternatives of generalization, simplification or complement studies of this first balanced tree have been proposed [3, 4, 5, 6, 7].

A novel kind of generalization of the AVL tree is the Red-Black tree. It is one of the most important and used self-balancing data structures. It behaves well in update-intensive applications, since it performs *log(n)* operations in the insertion process and at most two restructurings in the deletion one. The Red-Black tree was originally obtained from the 2-3 trees as an amelioration of AVL trees [8]. The first version was designed in 1972 by Rudolf Bayer [6] under the name: "Symmetric Binary B-trees", where the author compares the structure with the class of B-trees. A few years later Leonidas J. Guibas and Robert Sedgewick [9] proposed a new form of the original structure where the tree balance is expressed using Red and Black colors.

Because of the difficulty to implement the Red-Black tree in practice, especially in the deletion process, some works were proposed to simplify the corresponding algorithms. AA tree is a powerful simplification of Red-Black trees with the same performance and much more approach and coding simplicity [10]. Moreover, several simple implementations of Red-Black trees can be found in [11, 12]. Recently, the majority of works in terms of AVL and Red-Black trees aim basically to simplify rather than get a good performance. In the AVL tree case, [13] introduces a new simpler insertion and deletion algorithms for AVL trees by using virtual nodes. A brief study of AVL trees using this concept is presented in [14]. In the same spirit, work [15] gives a new algorithm and explains how to easily maintain the balance factor after an updating operation. When it comes to Red-Black trees, a revisited version has been proposed [16] where the code is considerably reduced compared to the implementation proposed in [17].

The design of a balanced tree is still a rich area, and not yet fully explored. A recent proposition with improvements for binary search trees are proposed in 2015 [18].

The main idea is to assign a non-negative integer rank to each node and impose eight rank

rules to give the AVL tree, a new kind of balanced tree and different kinds of Red-Black trees: the standard version [9] equivalent to the symmetric binary B-trees [6], the binary B-tree [5], the left leaning trees [16]. Their rank-based framework generalizes the dichotomies framework of Guibas and Sedgewick [9]. It is a very interesting work since it provides not only a new framework for defining height-based balance but also a new sort of balanced binary tree: the weak AVL tree. However, in addition to the obligation of satisfying loads of inequalities corresponding to the number of inserted and deleted cases, the framework gives separate rules to define common balanced trees rather than a unique hybridization of the most important and useful ones.

Our main purpose in this work is to represent at the same time the most used balanced binary trees: AVL and Red-Black trees. In other words, we propose common algorithms for the two data structures. Only one parameter suffices to switch between the two structures. The new representation is a binary search tree partitioned either in one class or in classes of heights 0 and 1. Each class holds an AVL tree. When only one class exists, it generates predictably an AVL tree. Otherwise, the new structure is equivalent to a Red-Black tree with totally different and simple algorithms. One extra byte of storage allows representing both the kind and the height of a node.

The rest of the paper is structured as follows: section 2 introduces the tree terminology. Section 3 describes our contribution. Section 4 presents the maintenance operations while section 5 gives insertion and deletion algorithms of the proposed balanced tree. In the section 6 we give the implementation of the structure, the results of implementations and the discussion. Section 7 shows the applications of the new structure. Finally section 8 makes a conclusion and looks forward to the future research.

## 2 Tree Terminology

We present here after some basic definitions used across the paper.

These definitions are related mainly to the binary search tree and the partitioning problem on graphs.

### 2.1 Binary Search Tree

It is an organized tree in a binary representation where each node contains a key, a data, the left child and the right child which can be missing nodes.

Consider x a node in a binary search tree. If *y* is a node in the left sub-tree of *x*, then $y.key \leq x.key$. If *y* is a node in the right sub-tree of *x*, then $y.key \geq x.key$. It is called the binary-search-tree property which allows us to print out all the keys in a binary search tree in sorted order by the simple recursive algorithm: the in-order tree walk. [19].

In what follows we define some concepts of the binary search tree:

- A leaf or an external node is a node with no children, while a unary (respectively a binary) node is a node with one child (respectively two children). They denote the internal nodes.
- The height of a node *x*: *h(x)* is the max of the height of its left and right children plus one in the case *x* is not a missing node, otherwise, *h(x)* equals *-1*.
- The size of a node: *s(x)* is the number of its descendants including itself.
- The search process starts with the root. First we compare the searched key with the root's key. Next we go to the left (respectively right) sub-tree if the searched key is less (respectively greater) than the root's key. We reach the end of the search when we find the desired key or we reach a missing node.
- Update operations concern the insertion and deletion. They are both preceded by a search process. For the insertion, when a missing node is reached, it is replaced by a new node with the key to insert. As for the deletion, the process is more complicated since after the search, multiple scenarios may arise. If the node is a leaf we replace it by a missing node, and if it is a unary node we replace it by its child. However if this node is an internal node, we find its in-order predecessor node or its in-order successor node and then we switch


**Fig. 1.** The right rotation of the node G

between the node we found and the item we want to delete.

- The restructuring operations allow us to maintain the binary tree balanced. We generally use single or double rotations after an update operation. See figure 1.

### 2.2 The Partitioning Problem

The tree partitioning problem arises when information must be allocated to blocks of memory, whose capacity is limited.

Assume a tree *T = (V, E)*. A partition of *T* is defined as a collection of k clusters of nodes $C_i$ witch we name class, while *i* varies between 1 and K, such that: $\bigcup_{i=1}^{k} Ci = VC_i \cap C_j = \emptyset$

As a result, the union of the sub-trees gives the whole tree and the intersection of two given sub-trees is null.

An edge *(i, j)* of *T* is said to be cut by a partition of *T* if nodes *i* and *j* are in different clusters.

An optimal partition of *T*: $P_t(opt) = \{C_1, C_2, C_3, \ldots C_k\}$

Is one in which each cluster $C_i$ satisfies the weight constraint:

$$\sum_{j \in Cj} Wj \leq = W$$

### 2.3 Basic Operations

Here after are some basic operations used in the representation of our proposed tree:

- Lc(P): Left child of node P
- Rc(P): Right child of node P
- Ass_Rc(P, Q): Make Q a right child of node P
- Ass_Lc(P, Q): Make Q a left child of node P
- Kind(P): Kind of node P
- Ass_Kind (P, A_Kind): Make A_kind the new kind of node P
- Height(P): Height of simple node P inside the class it belongs
- Height2(P): Height of class node P
- Ass_height (P, H): Make H the new height of node P
- Rotation(P, Dir): makes a left rotation around node P if Dir=1 and a right rotation if Dir=0. It returns the node that replaces P
- KindSwap(P, F): swaps kinds of nodes P and F
- KindFlip(P, F, Dir): is invoked after F= Rotation(P, Dir). It transforms new children of F into class nodes and attributes to F the initial kind of P.

## 3 The Partitioned Binary Tree

### 3.1 The Basic Idea

The notion of partitioning a graph in a form of a tree is studied earlier by [20]. The application of the algorithm can be: in the allocation of computer information to physical storage space or in finding a suboptimal partition of any connected graph.

The proposed work is a binary search tree partitioned in classes. Each class is in fact a sub tree holding an AVL tree of height *H* or *H-1*. The root node of this sub-tree is a class node; the other nodes are simple. Furthermore, the new structure is perfectly balanced considering only class nodes.

Beside the data field, a node contains a byte called a code to designate both its kind and its height. The height of a node is in fact the depth of the sub-tree rooted at this node inside the class it belongs. The storage of a byte in a node delivers a range of benefits:

- To minimize the number of the requirements for a unique framework
- To detect easily the type of the balanced tree used
- To trigger the restructuring operations after an update

– To control and detect the type of a node at any time in the tree for the operations inside and outside a class.

### 3.2 The Rules

In this section we give formally the requirements that a binary tree must have in order to be considered as a partitioned tree, we emphasis on the fact that these requirements must be satisfied for AVL and Red-Black tree case on the globalism, not a separate rule for each type like the case of [18]. In consequence we have the same representation of both AVL and Red-Black tree.

Formally, the new structure should respect these four rules:

– Rule 1: Every node can be either a simple or a class node.

We describe two categories of nodes: simple and class nodes. The class node encloses a sub-tree of simple nodes of height one or infinity of nodes.

– Rule 2: Every class must have a height equals to *H - 1* or *H - 2*.

A variable H is considered as an integer initialized at the beginning, and the variation of this integer can give different kinds of balanced trees. In this work we focus on the main ones. When *H = 2* the tree is considered as a Red-Black tree, and when the limit of H tends to the infinity the tree will be an AVL. Indeed, there is only one class node which is the root of the class. All the others are simple nodes. Consequently, in the definition above, by replacing simple nodes by red nodes and class nodes by black ones, we obtain exactly the definition of a Red-Black tree.

When the height of a class is equal to 0, this means the black node has not a red child. When the height of a class is equal to 1, this means that the black node has one or two red children. All the simple nodes have 0 as height. Classes define mathematically a partition on the tree. In other words, the intersection of any two classes is empty and the union of all the classes gives the whole tree.

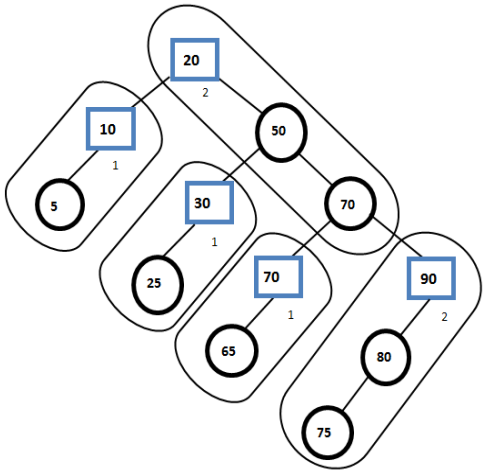


**Fig. 2.** The representation of the new structure *H = 3*

In order to simplify the presentation of the trees in the figures below, class nodes are represented inside blue squares and simple nodes inside circles. Furthermore, classes are surrounded.

Figure 3(a) shows the new structure as a structure equivalent to an AVL tree. There is only one class containing an AVL tree. Values under the nodes designate the height of the nodes.

– Rule 3: Every simple node must have a height equals to *0* or *H - 2*.
– Rule 4: Every direct path from any node to a leaf must contain the same number of class nodes.

This rule corresponds to having the same black node in a Red-Black tree. We can notice that when the tree has only class nodes it is a perfect balanced tree since any path from the root to a leaf has the same number of class nodes.

Figure 2 shows an example of our representation when *H = 3*, we can see that the height of every class is 2 or 1. Moreover every direct path from the root to a leaf has exactly two classes.

#### a) AVL Tree Case

It is straightforward to observe that for $H = \infty$ the new structure generates an AVL Tree.

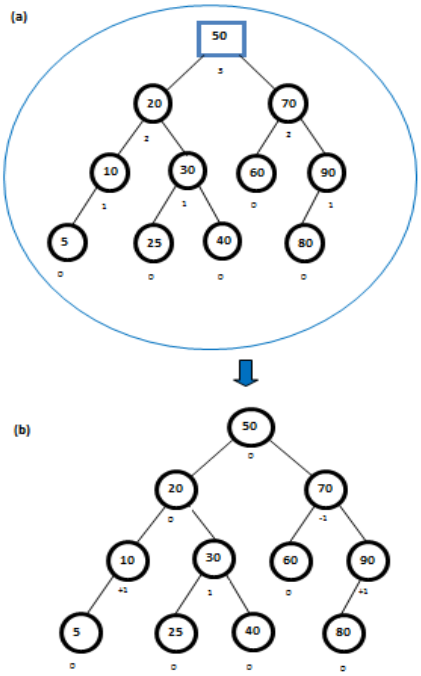


**Fig. 3.** The new structure as an AVL tree

Indeed, there is only one class node which is the root of the class. All the others are simple nodes. In practice, there is no limit for the height of the unique class and each path from any node to a leaf contains the same number of class nodes (0 or 1).

The minimum number of nodes in an AVL tree of code *C* with height *k* satisfies the recurrence:

$n_0 = 1, n_2 = 2, n_3 = 4, n_k = 1 + n_{k-1} + n_{k-2}$ for any $k \geq 2$. This requrence corresponds to Fibonachi trees, $n_k = F_{k+3} - 1$. We have $F_{k+2} > \emptyset^k$ where $\emptyset$ is the golden ratio [21].

$F_{k+3} = 1 + F_{k-2} + F_{k-1} \rightarrow F_{k+3} - 1 = F_{k-2} + F_{k-1}$.

$F_{k+2} > \emptyset^k \rightarrow F_{k+3} - 1 > \emptyset^k \rightarrow k < \log_{\emptyset} n \rightarrow k < 1.4404 \log n$

**b) Red-Black Case**

It is also pretty straightforward to notice that for $H = 2$ the tree generates a data structure equivalent to a Red-Black tree.

The subtree rooted at any node n has at least $2^{bh(n)} - 1$ internal nodes.

If N is nil, then its height is 0. 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 color. 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: $2^{bh(n)-1} - 1 + 2^{bh(n)-1} - 1 + 1 = 2^{bh(n)} - 1$ internal nodes [19].

We know also that at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least *H/2*; thus, $n > 2^{h/2} - 1 \rightarrow H < 2.\log(n + 1)$

**Lemma 3.1**: The maximum height of the partitioned tree is *2 Log n*

**Proof:** The minimum number of any node in PBT 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* $k \geq 2$.

By induction $n_k \geq 2^{H/2}$ which gives: $h \leq 2 \log n_k$.

## 4 Maintenance Operations

We can classify maintenance operations into two categories: those that are applied inside a class and those outside classes. *Restructuring*, *AVL_tree_insert* and *AVL_tree_delete* are operations of the first category. Operations of the second category are: *Partitioning*, *Departitioning*, *Restructuring-Partitioning* and *Transforming*.

### 4.1 Operations inside a Class

We give here after the various basic maintenance operations used to perform insertion and deletion algorithms on the new structure in terms of operations defined above.

**a) Restructuring**

Restructuring consists simply in rebalancing the tree after a tree property violation. It uses Restructure operation which performs a rotation and updates heights of the turned nodes.

A KindSwap operation can also be performed in Restructure operation. Furthermore, Restructure

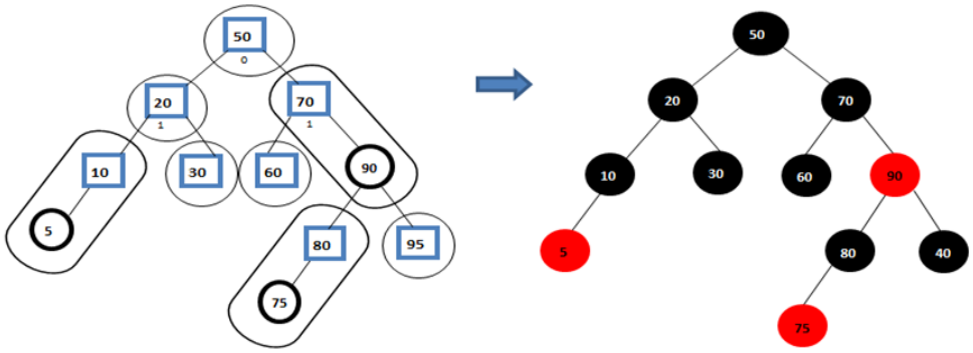


**Fig. 4.** The new structure as a Red-Black tree

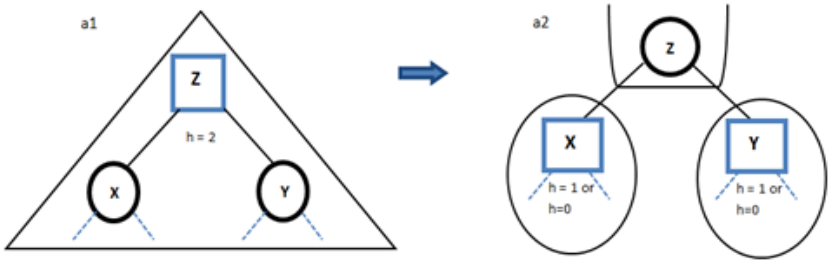


**Fig. 5.** The Partitioning operation

can be preceded by a Reverse_balance operation which consists in reversing the balance of a given: node.

```
Function Restructuring (PBT P, int Dir): PBT
Variables
S: PBT
Begin
   If(Dir = 0)
      If(Height(Rc(Rc(P)))–Height( Lc(Rc(P)))=1)
         Ass_Rc(P,Reverse_Balance(Rc (P),1 ))
         S ← Restructure (P, 0)
   Else If(Dir = 1)
      If(Height (Rc(Lc(P)))–Height(Lc(Lc(P)))= 1)
        Ass_Lc (P,Reverse_Balance (Lc (P), 0 ))
        S ← Restructure(P, 1)
   Return S
End
Function Restructure (PBT P, int Dir): PBT
Variables
F: PBT
Max1, Max2: INT
Begin
   F ← Rotation (P, Dir)
   Max1 = Max( Height(Lc(P)), Height(Rc(P)) )
      Ass_Height(P, Max1 + 1)
   Max2 = Max( Height(Lc(F), Height(Rc(F)) )
   Ass_Height(F, Max2 + 1)
   If (REDBLACK Or (Kind (P) = Class) )
      KindSwap(P, F)
   Return F
END

Function Reverse_Balance (PBT P, int Dir): PBT
Variables
F: PBT
Begin
   F ← Rotation (P, Dir)
   Max1 = Max(Height (Lc(P)),Height(Rc(P)) )
   Ass_Height(P,Max1 + 1)
   Max2 = Max( Height (Lc(F), Height(Rc(F)) )
   Ass_Height(F, Max2+ 1)
   Return F
End
```

**b) AVL Tree Insert**

AVL tree insert algorithm uses the *Restructuring* operation defined above to rebalance the tree each time the tree becomes unbalanced in the sense of AVL trees. In our purpose, the algorithm is expressed with the height instead of balance in each tree node.

This algorithm is applied when an item is inserted into the tree. Stack Branch holds the path traversed by search process from the tree root toward the parent of the new inserted node. Nodes are popped in order to update their height fields. If the balance of a node becomes (in absolute value) greater than 1, the tree is restructured and the process is stopped. The algorithm is the following:

```
Function Avl_insert (PBT Root)
Begin
   Avl_Insert ← Root
   Continue ← True
   Repeat
        Pop(Branch, P)
        Kind_P ← Kind (P)
        Update P's height
        If ( |Height (Lc (P)) - Height (Rc (P)) | > 1)
             If ( Height (Lc (P)) > Height (Rc (P)) )
                  Q ← Restructuring (P, 1)
             Else Q ← Restructuring (P, 0)
             If (Kind_P = Simple)
                       Pop (Branch, Parent)
                      Modify node Parent to point Q
             Else Avl_Insert ← Q
             Continue ← False
        Else Continue ← (Kind_P = Class)
    Until (Not Continue)
   End
```

**c) AVL_tree_delete**

*AVL_tree_delete* algorithm also uses *Restructuring* operation. The algorithm below is applied when an item is deleted from a leaf class rooted at Root. Stack Branch holds the path traversed by the search process from the root of the entire tree toward the parent of the deleted node. Nodes are popped in order to update their height fields. If the balance of a node becomes (in absolute value) greater than 1, the tree is restructured. The process can continue upward the tree.

```
Function Avl_Delete (PBT Root)
Begin
   Avl_Delete ← Root
   Continue ← True
   Repeat
        Pop (Branch, P)
        Kind_P ← Kind_ (P)
        Save_height ← Height (P)
        Update P's height
        If ( |Height (Lc(P)) - Height (Rc) | > 1 )
             If ( Height (Lc (P)) > Height (Rc (P)) )
                  Q ← Restructuring (P, 1)
             Else Q ← Restructuring (P, 0)
             If (Kind_P = Simple)
                  Parent ← Top (Branch)
                  Modify node Parent to point now Q
            Else Avl_Delete ← Q
            P ← Q
   Until (Not Save_height – Height(P) = 0) Or
         (Kind_P = Class))
End
```

**4.2 Operations Outside a Class**

Naturally, operations between classes concern only the structure equivalent to a Red-Black tree.

During the process of insertion, an item is always inserted into a leaf class. The class can overflow, i.e. its height reaches 2. A *Restructuring* is performed if the class has only one child. Otherwise, a *Partitioning* operation is performed.

During the process of deletion, an item is always removed from a leaf class. The leaf class can underflow, i.e. its height reaches -1. Several cases occur:

- The underflow class has not a direct sister class (or its sibling node is a simple node). A *Transforming* operation is performed.
- The underflow class has a direct sister class (or its sibling node is a class node) with no child. A *Departitioning* operation is performed.

ISSN 2007-9737



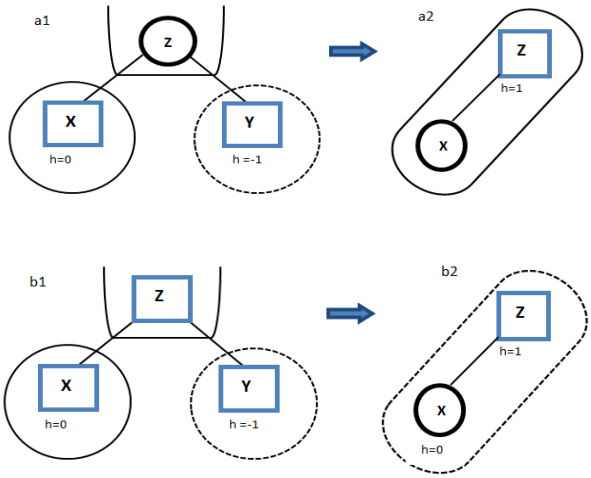


**Fig. 6.** The Departitioning operation

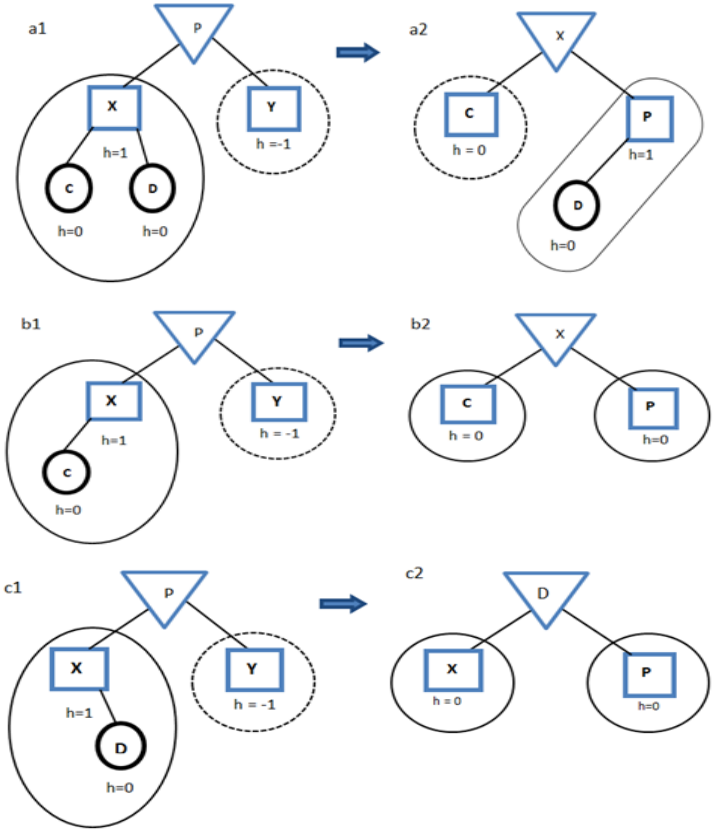


**Fig. 7.** The Restructuring-Partitioning operation

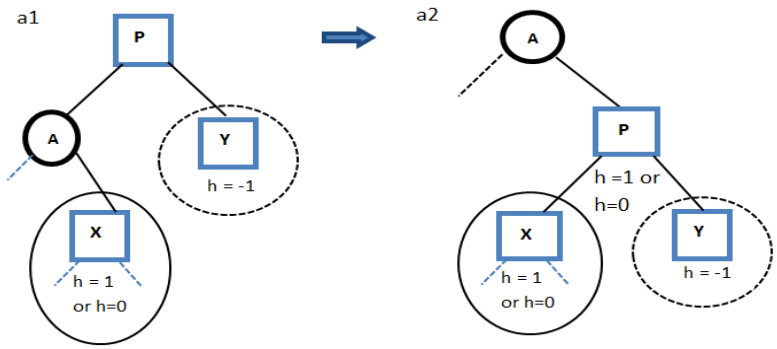


**Fig. 8.** The Transforming operation

– The underflow class has a direct sister class (or its sibling node is a class node) with one or two children. A *Restructuring-Partitioning* operation is performed.

We describe henceforth, all the operations mentioned.

### a) Partitioning

It consists in transforming one class into two classes. In Figure 5(a1), after an insertion operation in class *Z*, this is partitioned since its height reaches *2*. Dashed lines correspond to the four possible cases. Node *Z* becomes a simple node and its two children *X* and *Y* become class nodes (Figure 5(a2)). Node *Z* becomes thus a new leaf in the mother class.

This operation corresponds simply to the modification of three nodes' kinds. Partitioning does not require rotations and is done in *O(1).*

### b) Departitioning

In Figures 6(a1) and 6(b1), after a delete operation, the height of class *Y* becomes -1 while its direct sister class has a height equal to 0. A *Departitioning* operation holds. Node *Z* is deleted from the mother class as depicted in Figures 6(a2) and 6(b2). Figure 6(b2) shows a situation where the conflict is not yet resolved. This means that the process continues since the mother class has no child. Two nodes' kinds will be modified: the parent node and its child. The parent node becomes a class node while its child becomes a simple node.

As *Partitioning*, *Departitioning* does not require rotations and works in *O(1) time.*

### c) Restructuring-Partitioning

Restructuring-Partitioning is undertaken when a class underflows, its direct sister class exists and has one or two children. Such situations are depicted in figures 7(a1), 7(b1) and 7(c1) where the underflow class is to the right of node *P*. As node *P* can be a simple or class node, it is represented inside a triangle. The conflict is first solved by possibly applying a *Reverse_Balance* operation on the sister class (a simple rotation). Second, a *Restructure_Partition* operation is performed. It performs a rotation, a *KindFlip* operation and updates heights of the concerned nodes. Figures 7(a2) and 7(b2) are new situations of Figures 7(a1) and 7(b1). Nodes *C* and *P* become class nodes and the kind of node *X* after the rotation is the one of *P* before the rotation. As node *X* had already a left child, a *Reverse_Balance* operation is not necessary. For Figure 7(c1), a reversing of balance of class *X* is first performed. As a consequence, the result is depicted in Figure 7(c2).

### d) Transforming

Recall that *Transforming* occurs when a class underflows while it does not have a direct sister class (the sibling node is simple).

It is the case of Figure 8(a1) where the underflow class is *Y* and its direct sister class is *A*. *P* is their parent node.

ISSN 2007-9737



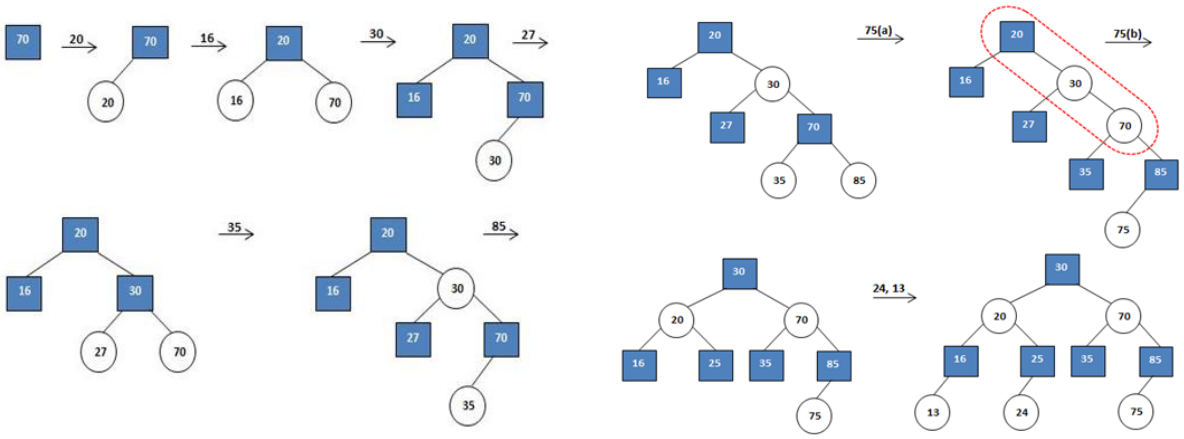


**Fig. 9.** A step by step insertion algorithm

If node *A* is a left child, then the right child of node A must be a class node with a height *0* or *1*. A first single right rotation of node *P* is performed in order to find a direct sister class of the underflow class. Figure 8 (a2) is the result of the *Transforming* process. Now, the underflow class has a direct sister class and the process continues either with a *Departitioning* or a *Restructuring- Partitioning*.

## 5 Insertion and Deletion Algorithms

Once the maintenance operations are presented, we can now give the algorithms of insertion and deletion of the new structure.

### 5.1 Inserting a New Element

In the insertion process an element is always added into a leaf class. In AVL tree case (Parameter *REDBLACK = False*), the process terminates. In Red-Black tree case (Parameter *REDBLACK = True*), if the height of this class becomes 2, the algorithm described below is applied. It uses a stack containing all the nodes traversed from the root (Tree) of the entire tree until the parent of the newly inserted node. The algorithm goes upward the tree from the inserted node towards the root of the tree by making either *Restructuring* or *Partitioning*. When *Restructuring* is performed, the process terminates. On the other hand, *Partitioning* can be in cascade.

The insertion algorithm begins with the root of the overflow class *(Root)* and its parent node *(Parent)*. The parent is used to update links when a restructuring is performed. Top operation gives the root of the mother class and its parent node without popping elements from stack Branch. Recall that function *AVL_INSERT(Root)* adjusts the balance of the class rooted at Root and returns the new root of the class.

***Repeat***
    *Save_Root ← Root*
    *Root' ← AVL_INSERT (Root)*
    ***If*** *(Root' <> Save_Root)*
        ***If*** *(Parent <> Null)*
            *Modify node Parent to point now Root'*
        ***Else*** *Tree ← Root'*
        *Continue ← False*
    ***ElseIf*** *(Height2 (Root') = 2 and REDBLACK)*
        *PARTITIONING (Root')*
        *Top (Branch, Root, Parent)*
***Until*** *(Empty (Branch)) or (Not continue)*

**Comment and Analysis**

The element is first inserted inside a class using the *AVL_INSERT* method.

**a) AVL Tree Case:** If the root of the class is modified by the *AVL_INSERT* function, a *restructuring* is done (in *AVL_INSERT*) and then test *(Root' <> Save_Root)* holds. Flag *Continue* is set to *False*. As a consequence, only one iteration of the Repeat loop is performed. If the root of the class is not modified, test "*Height2(Root') = 2 and REDBLACK*" fails since *REDBLACK* is false. Furthermore, the stack is empty as there is no *Restructuring* in *AVL_INSERT*.

**b) Red-Black Tree Case:** Function *AVL_INSERT* is called at each new iteration. It inserts the root of the partitioned class. Recall that *AVL_INSERT* performs at most one *Restructuring*. On the other hand, several *Partitioning* operations can be done. Indeed, each time a *Partitioning* is made, the root of the partitioned class migrates to mother class which can be again partitioned if its height reaches *2*.

**Scenario Example:** Figure 7 shows step by step the construction mechanism through an example when parameter *REDBLACK* is true, i.e. *H = 2.*

1. Insert (70): a class is created with one element.
2. Insert (20): 20 is inserted into class 70.
3. Insert (16): 16 is inserted into class 70 and causes a restructuring (right rotation of node 70). 20 becomes the root of the class. Indeed, class 70 overflows while it has one child.
4. Insert (30): 30 is inserted into class 20 and causes a Partitioning since class 20 overflows while it has two children.
5. Insert (27): 27 is inserted to the left of node 30 and this causes a restructuring (right rotation of node 70).
6. Insert (35): 35 augments the height of class 30 and this is balanced in the sense of an AVL tree. As this class overflows, it is partitioned to generate two other classes: 27 and 70. 30 is transformed into a simple node and belongs now to the mother class 20.
7. Insert (85): 85 is inserted into class 70 as a right child.
8. Insert (75): This case is represented by two trees (75a and 75b). a) 75 is inserted into class 70 and augments its height. Class 70 is balanced in the sense of an AVL tree but it overflows. It is then partitioned to generate two other classes: 35 and 85. b) Node 70 becomes a new leaf of class 20 and then overflows (class surrounded in red lines). As class 20 has one child, it is structured (left rotation of node 20). 30 becomes the new root.
9. Insert (24, 13): 24 is inserted in class 27 and 13 in class 16.

### 5.2 Deleting an Existing Element

An element is always removed from a leaf class. For the AVL tree case (Parameter *RED_BLACK* is False), the process terminates. However, for the Red-Black tree case (Parameter *RED_BLACK* is True), if the height of this class became equal to -1, i.e. it underflows, the algorithm described below is applied. This consists in going upward the tree from the removed node towards the root of the tree (Tree), by making one or several operations among the following:

- *Departitioning*
- *Transforming*
- *Restructuring-Partitioning*

When a *Restructuring-Partitioning* is performed, the process stops. On the other hand, *Departitioning* can be in cascade. *Transforming* is performed only one time but the process continues.

The algorithm uses a stack containing all the nodes traversed from the root until the parent of the newly removed node.

It has as input the root of the underflow class *(Root)*, its parent node *(Parent)* and its grandparent node *(Grandparent).* Parent is used to update links when a restructuring is performed. Grandparent is used when a maintenance operation is performed. *Top* operation gives the root of the mother class, its parent node and its grandparent node without popping elements from stack Branch.

Recall that function *AVL_DELETE(Root)* adjusts the balance of the class rooted at Root and returns the new root of this class.

***Repeat***
*Save_Root ← Root*
*Root' ← AVL_DELETE(Root)*

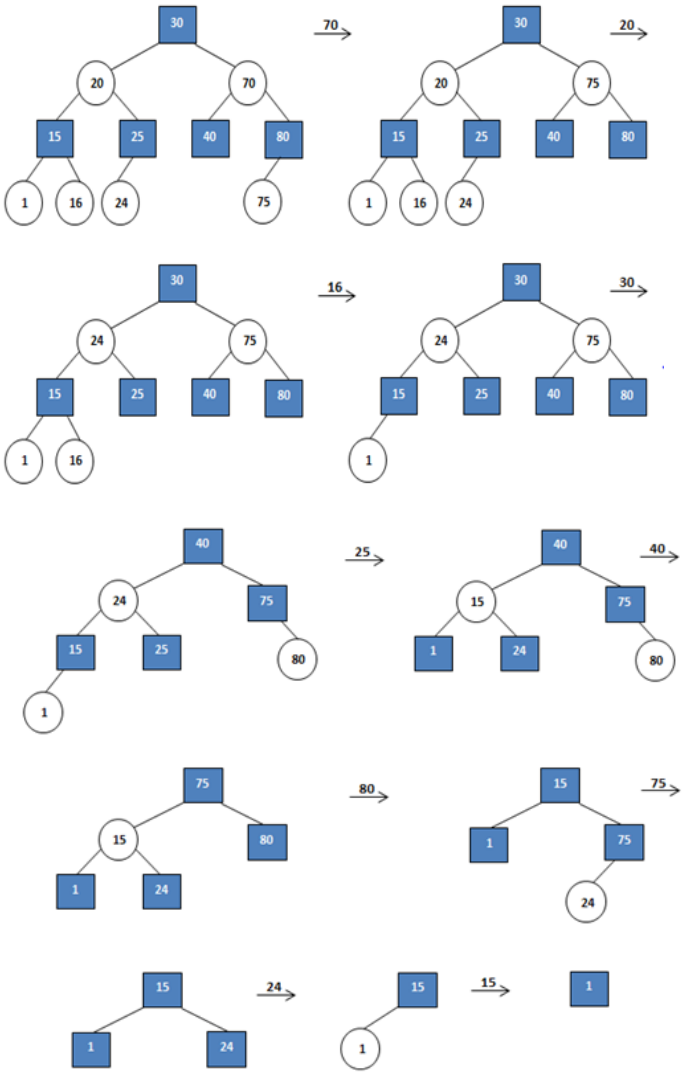


**Fig. 10.** A step by step deletion algorithm

```
If (Root' <> Save_Root) And (Root' <> Null)
    If (Parent <> Null)
        Modify node Parent to point now Root'
    Else Tree ← Root'
    Continue ← false
Else If ((Height2(Root')= -1) And REDBLACK)
  If (Lc(Parent)= Root')
        Sister ←Rc(Parent)
        Dir ← 0;
Else
        Sister ← Lc (Parent)
        Dir ← 1
If (Kind (Sister) = Simple)
   Parent2 ← Sister
   If (Dir = 1) New_Sister ← Rc (Sister)
   Else New_Sister ← Lc( Sister )
       Q ← TRANSFORMING (Parent, Dir)
   If (Grandparent <> Null)
```

```
        Modify node Grandparent to point Q
    Else Tree ← Q;
    Pop(Branch, X);     Push(Branch, Q);
    Push(Branch, X;   Grandparent ← Parent2
  If ( |(Height2(Root') – Height2 (New_Sister )|
     > 1)
      Q ← RESTRUCTURING_
      PARTITIONING (Parent, Dir)
      If (Parent == Tree) Tree ← Q
      Else If (Grandparent <> Null)
          Modify node Grandparent to point Q
      Else Tree ← Q
      Continue = False
  Else
      Kind_Parent ← Kind (Parent)
      DEPARTITIONING (Parent, Dir)
      Top( Branch, Root, Parent, Grandparent)
      If (Kind_Parent = Simple) Pop (Branch)
Else Continue ← False
Until (Not Continue)
```

**Comment and Analysis**

The element is first removed from a class using the *AVL_DELETE* function.

**a) AVL Tree Case**

If the root of the class is modified by the *AVL_DELETE* function, one or two restructurings are done and then test *"(Root' <> Save_Root)* And *(Root' <> Null)"* holds. Flag Continue is set to False. As a consequence, only one iteration of the Repeat loop is performed. If the root of the class is not modified, test "*Height2 (Root') = -1 and REDBLACK*" fails since *RED_BLACK* is false. Furthermore, the stack is empty as there is no restructuring in *AVL_DELETE.*

**b) Red-Black tree case**

Function *AVL_DELETE* is called at each new iteration. It always deletes a leaf inside a class. This leaf becomes the root of departitioned classes. Recall that *AVL_DELETE* performs at most two restructurings.

On the other hand, several *Departitioning* operations can be done. Indeed, each time a *Departitioning* is made, the root of the departitioned classes is removed from the mother class which can be again departitioned if its height reaches -1.

It is straightforward to observe that the deletion algorithm works as follows:

If the underflow class has not a direct sister class (test "*Kind(Sister) = Simple*"), a *Transforming* is first done to find its direct sister class. If the difference in heights between the underflow class and its direct sister class exceeds one in absolute value, a *Restructuring-Partitioning* is performed. Otherwise a *Departitioning* is performed.

**Scenario Example:** Figure 8 shows step by step the deletion mechanism through an example when parameter *REDBLACK* is true.

1. Delete (70): 70 is replaced by 75 (its in-order successor) and then this latter is removed from class 84.
2. Delete (20): Again, 20 is replaced by 24(its in-order successor) which is removed from class 25.
3. Delete (16): 16 is removed from class 15.
4. Delete (30): 30 is replaced by 40(its in-order successor). Class 40 underflows and has a direct sister class 84 with no children. They are then departitioned into the new class 75.
5. Delete (25): 25 is removed and causes an underflow of class 25. Class 25 has a direct sister class with one child. A *Restructuring-Partitioning* is performed (Right rotation of node 24 followed by a *KindFlip*)
6. Delete (40): 40 is replaced by 75(its in-order successor). 75 is removed from class 80. 80 becomes the new root.
7. Delete (80): 80 is removed and causes an underflow of class 80. Class 80 does not have a direct sister class. A *Transforming* is completed by a right rotation of node 75. Now, node 75 has at its left class 24 and at its right class 80. A *Departitioning* of node 75 is then performed.
8. Delete (75): 75 is removed from class 75. 24 becomes the new root.
9. Delete (24): class 24 underflows. Classes 24 and 1 are then departitioned in order to generate the new class 15.
10. Delete (15): 15 is removed from class 15. 1 becomes the new root.

## 6 Implementation

### 6.1 Description

The new balanced tree has one Boolean parameter: *RED_BLACK*. If this parameter is true, we are in the case of a Red-Black tree structure. Else, it is about an AVL tree. This parameter is used in both insertion and deletion algorithms, as well as in maintenance operations.

The new structure uses one additional byte in every node. Bit 1 is set to 1 if the node is a class node. Otherwise this bit is set to 0. Bits 2 to 8 hold node height. In this way, *Height(Code)* is simply *Code Mod 128*. Moreover, if Code ≥ 128 then it is a class node. Otherwise, it is a simple node.

### 6.2 Data Structure for the Proposed Tree

We give hence the pseudo code of the proposed tree.

```
Type TypeNode = (Simple, Class)
Type Ptr_node = * T_node
T_node ← record
Begin
    Data: Anykind
    Code: Byte
    Lc, Rc: Ptr_node
End

Function Kind (A: Ptr_node): Typenode
Begin
    If (A.Code >= 128)
        Kind ← Class
    Else Kind ← Simple
End

Procedure Ass_kind(A:Ptr_node, A_kind:
Typenode)
Begin
    If (A_kind = Simple)
        A.Code ← A.Code Mod 128
    Else A.Code = 128 + A.Code Mod 128;
End

Function Height2 (A: Ptr_node): integer;
Begin
   If (A = nil) Height2 ← -1
   Else Height2 ← A.Code Mod 128
End

Function Height (A: Ptr_node): integer;
Begin
    If (A=nil) Height ← -1
    Else If (A.Code >= 128) // Class node
      Height ← -1
    Else Height ← A.Code Mod 128
End

Procedure Ass_Height (P: Ptr_node; H: integer);
Begin
    If (P.Code < 128) P.Code ← H
    Else P.Code ← 128 + H
End
```

### 6.3 Experimental Tests

We considered the following experiment:
1. Build a Red-Black tree (RB), an AVL tree (AVL) and the two new binary search trees generated by the new structure (Z_AVL and Z_RB) with a same sequence *(S1)* of *N* random integer values.
2. Build a random sequence *(S2)* of about *N* insertion and removal operations.
3. (A) - Run sequence *S2* separately on each data structure.
(B) - Compute:

- The total number of rotations.
- The execution time made by both the insertion and removal algorithms in each kind of trees.

4. Repeat 1 – 3 three times for *N = 100 000* to *500 000* by step of *100 000* nodes.

### 6.4 Results

We have not shown the numbers of rotations performed by each data structure. These have been computed only to verify correctness. As expected, we obtained the same number of rotations in AVL and Z_AVL as well as in RB and Z_RB. We focused then our attention only on the execution time.

Table 1 shows in columns "AVL","RB", "Z_AVL" and "Z_RB" the execution times in milliseconds taken by each data structure. Column N denotes the size of trees initially generated as well as the number of inserted/ deleted operations. Values denote the average values of three tests.

First, simulation results confirm the superiority of Red-Black trees (Column RB) compared to AVL trees (Column AVL) in applications where insertions and deletions are very common. As an example to insert / delete 100 000 elements in a tree containing previously 500 000, AVL consumed 920 ms, while RB consumed 702 ms.

It is clear from the table above that the performance of the Red-Black tree generated by the new structure (Z_RB) gives the same results as the standard Red-Black tree (Column RB).

It is surprising that the performance of the AVL tree generated by the new structure (Column Z_AVL) is better than that of AVL trees (AVL). This could be explained by the fact that the new structure uses the height in nodes while the standard AVL uses the balance (0, +1 or -1).

Let us notice that the performance of the Red-Black tree generated by the new structure (Z_RB) is comparable to the performance of the AVL tree generated by the new structure (Z_AVL) because we used the same code.

**Table 1.** Results of the simulation tests

| N | AVL | RB | Z_AVL | Z_RB |
|---|---|---|---|---|
| 100,000 | 187 | 140 | 140 | 141 |
| 200,000 | 405 | 281 | 281 | 281 |
| 300,000 | 578 | 437 | 421 | 437 |
| 400,000 | 734 | 546 | 546 | 562 |
| 500,000 | 920 | 702 | 702 | 717 |

## 7 Applications

This new structure can be applied in all applications where AVL and Red-Black trees are used since it is equivalent to both structures and gives very good execution times. However it can be a very efficient structure for real time systems. In this context [22] demonstrates the usefulness of using both AVL and Red-Black tree in the priority queue in Dynamic Data-Driven Application Systems: when the system anticipates intensive search operations, the system will convert the tree to AVL, while when the system anticipates intensive updates operations, it convert the tree to Red-Black. This transformation can be done easily since we have the same code.

## 8 Conclusion and Future Work

In the current work we have described the possibility to connect the two most useful and intriguing balanced search trees, AVL and Red-Black trees in a simple way. This is accomplished through a binary search tree partitioned into classes that are in fact AVL sub-trees. The implementation of the algorithms gives satisfying results comparing to the previous propositions. Several applications of these combining algorithms are suggested like the real time systems, especially trees in the priority queue for Dynamic Data-Driven Application Systems: when the system anticipates intensive search operations the system will convert the tree to AVL. On the other side, when the system anticipates intensive update operations it convert the tree to Red-Black.

Our work gives simple insertion and deletion algorithms but its implementation requires a storage of 8 bits in order to take in consideration both the height and the type of the node in one way, and do not guarantee switching from one structure to another in real time systems. A possible amelioration of this proposition is to define a method in order to allow the structure switching from one structure to another in a dynamic environment.

## References

1. **Sedgewick, R. & Addison, W. (2002)**. Algorithms in Java, Parts 1-4. *Professional*, pp. 768.
2. **Adelson-Velskii, M. & Landis, E.M. (1963).** An algorithm for the organization of information. *Dokl. Akad. Nauk* SSSR 146, Vol. 3, pp. 1259–1262.
3. **Foster, C.C. (1965).** Information retrieval: information storage and retrieval using AVL trees. *Proceedings 20th national conference (ACM)*, pp. 192–205. DOI: 10.1145/800197.806043.
4. **Foster, C.C. (1973).** A generalization of AVL trees. *Communications of the ACM*, Vol. 16, No. 8, pp. 513–517. DOI: 10.1145/355609.362340.
5. **Bayer, R. (1971).** Binary B-trees for virtual memory. *Proc ACM SIGFIDET Workshop,* pp. 219–235. DOI:10.1145/1734714.1734731.

6. **Bayer, R. (1972).** Symmetric Binary B-Trees: Data structure and maintenance algorithms. *Acta Informatica,* Vol. 1, No. 4. pp. 290–306. DOI: 10.1007/BF00289509.
7. **Brown, M. (1978).** A storage scheme for height-balanced trees. *Inf. Process,* pp. 231–232.
8. **Sedgewick, R. & Wayne, K. (2011).** Algorithms 4 edition. Princeton University.
9. **Guibas, L.J. & Sedgewick, R. (1978).** A dichromatic framework for balanced trees. *In 19th Annual Symposium on Foundations of Computer Science IEEE*. DOI: 10.1109/SFCS.1978.3.
10. **Andersson, A. (1993).** Balanced search trees made simple. *Algorithms and Data Structures*. Springer Berlin Heidelberg, pp. 60–71. DOI: 10.1007/3-540-57155-8_236.
11. **Okasaki, C. (1999)**. *Purely functional data structures*. Cambridge University Press, pp. 1–203.
12. **Kahrs, S. (2011).** Red-black trees with types. *Journal of functional programming*, Vol. 11, No. 4, pp. 425–432. DOI: 10.1017/S0956796801004026.
13. **Rajeev, T. & Kumar, R. (2010).** *Balancing of AVL tree using virtual node.* RN 10 20. DOI: 10.5120/1331-1695.
14. **Chauhan, S., Thakur, S., & Rana, S. (2014).** A brief study of balancing of AVL tree. *International Journal of Research* 1, Vol. 11, pp. 406–408.
15. **Mondal, G. (2014).** A New Way of Inserting and Deleting the Node To and From the AVL search tree. *International Journal of Advance Research in Computer Science and Management Studies*, pp. 191–194.
16. **Sedgewick, R. (2008).** Left-leaning red-black trees. *Dagstuhl Workshop on Data Structures*, pp. 4–10.
17. **Wiener, R. (2005).** Generic Red-Black Tree and its C# Implementation. *Journal of Object Technology,* Vol. 4, No. 2, pp. 59–80.
18. **Haeupler, B., Siddhartha, S., & Tarjan, R.E. (2015).** Rank-balanced trees. *ACM Transactions on Algorithms (TALG)*, Vol. 11, No. 4. DOI: 10.1145/2689412.
19. **Cormen, T.H. (2009).** Introduction to algorithms. *MIT Press*, pp. 1292.
20. **Lukes, J.A. (1974).** *Efficient Algorithm for Partitioning of Trees*. IBM J. Res. Develop.
21. **Knuth, D.E. (1973).** *The Art of Computer Programming, Sorting and Searching.* Addison-Wesley, Vol. 3.
22. **Kumar, N.C., Vyas, S., Shidal, J.A., Cytron, R., Gill, D.C., Zambreno, J., & Jonesa, P.H. (2012).** Improving system predictability and performance via hardware accelerated data structures. *Proc. Computer Science*, Vol. 9, pp. 1197–1205. DOI: 10.1016/j.procs.2012.04.129.

*Article received on 04/12/2017; accepted on 07/09/2018.*
*Corresponding author is Lynda Bounif.*