Efficient Binary Search Tree: An Alternative to Red-Black Trees with Simplified Algorithms and Enhanced Deletion Performance

DJAMEL-EDDINE ZEGOUR
Ecole Supérieure d'Informatique,
LCSI Laboratory,
BP 68M, Oued Smar, 16309, El Harrach, Alger,
ALGERIA

Abstract: - A new data structure is suggested, which is made of two classes of nodes - simple nodes and class nodes. Class nodes create a perfectly balanced trees when considered exclusively, while simple nodes introduce a small imbalance. The new data structure is advantageous because: On one hand, it is equivalent to a Red-Black tree but has simpler algorithms. For those who are unaware, Red-black trees were designed to guarantee that all operations take logarithmic time, therefore, they are effective balanced binary search trees. By simplifying the algorithms and keeping the equivalence, the new structure is expected to offer better code readability and easier implementation in contrast with traditional Red-Black trees. And on the other hand, it is more powerful with deletion operations compared to the Red-Black trees delete algorithm. The deletion in Red-Black Trees may sometimes involve complicated cases to maintain balance and satisfy the various properties of the tree. But the new structure proposes a better algorithm for deletions and could then offer a better performance for applications relying heavily on deletions. Implementation of the given data structure allows you to develop software systems that can accommodate and retrieve large datasets.

Key-Words: - Binary search trees, AVL-trees, Red-Black trees, 2-4 trees, Partitioning, Modern applications.

Received: April 21, 2025. Revised: July 9, 2025. Accepted: August 22, 2025. Published: February 5, 2026.

1 Introduction

Binary Search Trees (BSTs) are one of the most popular data structures used in computer science
They are dynamic and hierarchical structures that represent a collection of items ordered in a linear order. The crucial property of a binary search tree is that all values stored in the left subtree of any node 'x' are smaller than the value stored at 'x', while all values stored in the right subtree of 'x' are greater than the value stored at 'x' (referred to as the binary search tree property).

Many books on data structures and algorithms cover binary search trees as an important topic. Some notable references include widely known books, [1], [2], along with modern books that emphasize practical aspects, [3], [4], [5].

Binary search trees offer efficient performance when the tree is balanced or nearly balanced. In such cases, the cost of finding any element becomes logarithmic (log N). Throughout this paper, 'N' represents the number of nodes in the tree, and 'lg' denotes the base-two logarithm.

There are two approaches to maintaining a balanced binary search tree when inserting and deleting items. One approach involves keeping the tree balanced using rotations, while the other approach involves periodically balancing the entire tree.

Among the data structures that enable the maintenance of a balanced BST, AVL trees, [6], Red-Black trees, [7], and AA-trees, [8], are noteworthy. It is worth mentioning that the latter two are essentially new versions of SBB (Symmetric Binary B-trees) and BB (Binary B-trees) proposed in [9], [10]. SBB

and BB represent 2-4 trees and 2-3 trees, respectively, as binary search tree structures. Among the efficient algorithms for globally balancing a BST, the widely used and renowned algorithm is known as DSW, [11].

Currently, Red-Black trees are gaining popularity as a data structure. They are even dedicated entire chapters in many books on data structures and are implemented and integrated into various programming languages (e.g., Java, C). Red-Black trees are also utilized for implementing dictionaries and associative arrays. The maintenance operations primarily involve restructuring and recolouring. Restructuring involves one or two rotations, while recolouring involves one, two, or three colour changes. During an insert operation, at most one restructuring and lg N recolorings are performed. During a delete operation, at most two restructurings and lg N recolorings are performed.

When developing Red-Black trees, Guibas and Sedgewick began with the SBB structure, which represents the BST representation of a 2-4 tree proposed five years earlier. However, our approach differs in that it partitions a binary search tree into classes (subtrees) of height 0 or 1. Moreover, a single additional bit in each node is sufficient to distinguish between the two types of nodes generated by this method. The tree's balance is ensured by the partitioning operation, which employs a mechanism similar to that of B-trees, [12]. In our view, such a partitioned BST precisely corresponds to the BST representation of a 2-4 tree, thus providing an alternative to Red-Black trees.

This paper aims to present a new balanced binary search tree. We will demonstrate that this new structure is equivalent to a Red-Black tree but features simpler algorithms, as reasoning is based on subsets rather than colours. Furthermore, the delete process in the new structure is slightly different but more efficient than that of Red-Black trees, as it reduces type changes and rotations. A type change simply corresponds to a colouring. Consequently, the delete process in the new structure saves execution time while maintaining search performance. In a previous work, [13], we presented a more general data structure also based on the partitioning of the tree.

The proposed data structure can play a significant role in several modern applications by enabling the creation of software systems that efficiently manage and query large datasets.

After introducing the tree structure through an illustrative example in Section 2, our main objective is to describe the insertion and deletion algorithms in detail (Sections 3 and 4), using explanatory diagrams and relevant code snippets. Each maintenance case is examined in detail, with a detailed analysis of how the algorithms work. Explanations, accompanied by many useful images, illustrate step-by-step how these algorithms work. We conclude with remarks in Sections 5 and 6. Specifically, Section 5 compares the maintenance operations of our structure to those of Red-Black trees, especially during insertion and deletion operations. Additionally, Section 5 provides a summary of the code for the maintenance operations and highlights the simplicity of the algorithm descriptions in the new structure compared to Red-Black trees. Section 6 further demonstrates the superiority of the deletion algorithm over the red-black tree algorithm through simulations. Two patterns are considered: repeated deletion of items and repeated insertions/deletions of items from a pre-built tree. Section 7 highlights several modern applications where our data structure can drive significant improvements, and finally, Section 8 concludes the paper.

2 The New Structure

Mathematically, the new structure introduces a partition of the tree such that each partition class contains a binary search tree of depth 0 or 1.
The structure is composed of two types of nodes: class nodes and simple nodes. A single bit differentiates them by indicating the node type.

Here are the formal rules for defining the new binary tree (Figure 1):
- Each node must be either a simple node or a class node.
- Each class has a depth of 0 or 1, i.e., a class with 1, 2, or 3 elements.

- Each direct path from a node to a leaf must contain the same number of class nodes.

To understand these concepts, Figure 1 presents an example of the proposed data structure and all the nomenclature used in this paper. The existing class nodes are: 2, 34, 39, 51, 94, 40, 35, 31, 43, 54, 71, and 15. The parent class of {39}, {51, 43, 54}, and {94} is class {40, 71}. The highest-order class, called {35}, houses the root class {2, 15}, {34}, {39}, {51, 43, 54}, and {94}. {2, 15} and {34} are direct siblings, while {51, 43, 54} and {39} are essentially siblings. We will refer to class {40, 71} as class 40. Therefore, the children of classes 40, 51, and 51 are identical. Class 35 has no children. The number of elements in class 40 is two, and for class 35, the value is one.

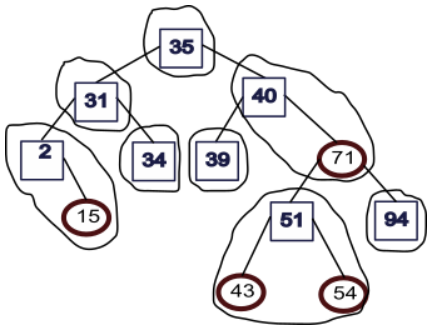


Fig. 1: New tree example and definitions
Source: created by the author

In the remainder of the article, we have adopted the following conventions for the figures.
- A class enclosed by a solid line indicates a valid class.
- A class enclosed by a highlighted solid line indicates an overflow or an invalid class.
- A class enclosed by a dotted line indicates an underflow class.

3 Insert Operation

Inserting a new node into a binary search tree (BST) consists of two important steps: First, we add the new node into the appropriate leaf class of the tree. Next, we only proceed to the second step if that leaf class starts to get overloaded. The insertion process ends with a restructuring and may trigger a cascade. These maintenance steps will be described in detail in the following sections.

Case 1: Restructuring a class

Restructuring happens when we override a class after an insertion, which typically occurs when the class reaches a height of 2 and has only one child. You can see this situation illustrated in Figures 2(a1) and 2(b1), where node A lacks a right child. Similar scenarios arise when a class has no left child either.

To fix the imbalance, we need to perform some rotations to balance the class. For instance, in Figure 2(a1), we carry out a double rotation—first turning left and then right—around node C. After this maneuver, node C becomes the new root of the class, while the original root is reduced to a single node. The result of the restructuring operation applied to Figure 2(a1) is depicted in Figure 2(a2). In Figure 2(b1), a single rotation (right) is performed around node B, which transforms into a class node, while the former root is converted into a simple node. The outcome is shown in Figure 2(b2).

When no arrow is depicted within a class, it indicates that the arrow points either to a NULL node (in the case of a leaf class) or to a class node (in the case of a non-leaf class).

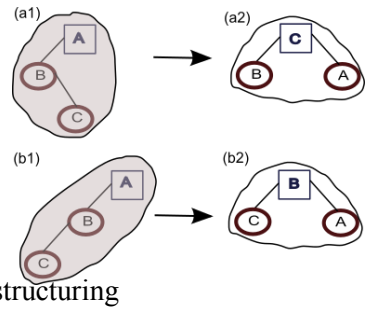


Fig. 2: Restructuring
Source: created by the author

The code for restructuring (A) is as follows:

```
P := Balancing(A)
Ass_Kind(P, 'Class')
Ass_Kind(Left(P), 'Simple')
Ass_Kind(Right(P), 'Simple')
```

In the context of the new structure, the Balancing procedure always takes a class with one child as input. It performs a single rotation if possible, or else it performs two rotations.

The functions Right(P) and Left(P) return the left and right child of node P, respectively. The function Ass_Kind(P, K) assigns the kind K to node P.

Case 2: Partitioning a class

Partitioning is triggered when a class becomes overloaded and the class node has two children. This scenario occurs when there is only one node at level two. An example can be seen in Figure 3(a1), where class node A has two children, B and C. Similar cases arise when D is located to the right of B, to the left of C, or the right of C.

Partitioning involves transforming one class into two classes and entails a simple modification of three nodes' kinds: the two child nodes become class nodes, and the root node becomes a simple node that now belongs to the mother class. If the parent class grows to a height of 2, it can trigger a series of partitioning operations that follow(cascading). Unlike restructuring, partitioning does not require any rotations. In Figure 3(a1), the class is partitioned to form two new classes, as shown in Figure 3(a2). Node A becomes a simple node in the mother class, while B and C become class nodes.

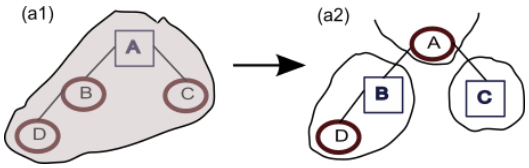


Fig. 3: Partitioning

Source: created by the author

The code for Partitioning (A) is as follows:

```
Ass_Kind(A, 'Simple')
Ass_Kind(Left(A), 'Class')
Ass_Kind(Right(A), 'Class')
```

This code modifies the kinds of three nodes in the partitioning process. Node A is assigned the kind 'Simple', while its left child is assigned the kind 'Class', and its right child is also assigned the kind 'Class'.

Insert algorithm

In the insert process, an element is always added to a leaf class. If the leaf class overflows, the algorithm depicted in Algorithm 1 is applied. The algorithm utilizes a stack (Branch) that contains all the nodes traversed from the root of the tree (Tree) to the parent (Parent) of the newly inserted element. The first Pop operation performed on the stack retrieves the parent, which is a simple node.

Comment and analysis

We can make the following remarks:

- The first Pop operation retrieves a class node.
- The second Pop operation retrieves the parent of that class node. This allows updating the link when a restructuring is performed.
- In the while condition, the variable Parent always refers to a simple node.

The new item is always inserted into a leaf class. If the leaf class overflows, meaning it reaches a height of 2, it is either restructured if the class node has one child, or it is partitioned. In the case of restructuring, the process is stopped. However, if partitioning is performed, the process continues upward in the tree since the class node becomes a new item (simple node) in the mother class. The process terminates when no further partitioning operations are necessary.

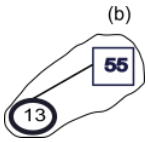
It can be observed that at most one restructuring operation is performed, and at most log (N) partitionings are performed. Therefore, the overall results are comparable to those of Red-Black trees.

Example scenario

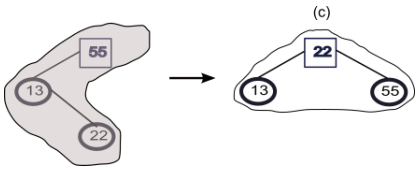
Below, we present the insert process step by step using an example.

1. Insert (55) → (a): A class is created with one element.

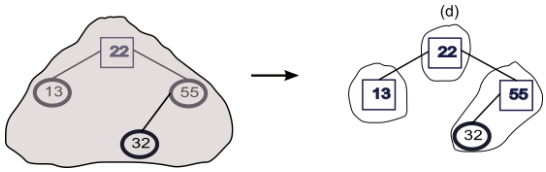
2. Insert (13) → (b): 13 is inserted into class 55.



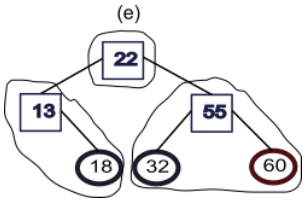
3. Insert (22) → (c): 22 is inserted into class 55 to the right of the node with value 13. Class 55 overflows since the maximum height of a class is equal to one. As class 55 has only one child, a restructuring is performed.



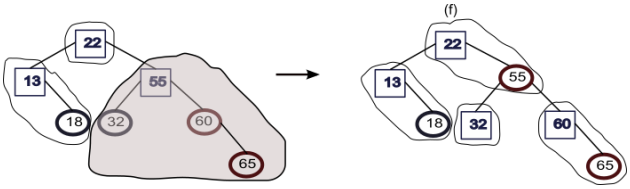
4. Insert (32) → (d): 32 is inserted to the left of the node with value 55. Once again, class 22 overflows and partitioning is performed as the class node with value 22 has two children.



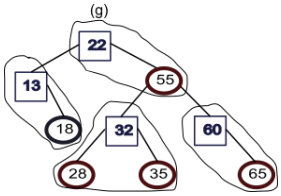
5. Insert (18), Insert (60) → (e): 18 is inserted into the leaf class 13. 60 is inserted into the leaf class 55.



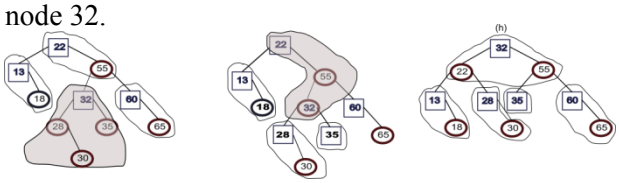
6. Insert (65) -> (f): 65 is inserted to the right of the node with the value 60. Class 55 overflows and a partitioning operation is performed as it has two children. Nodes 32 and 60 become class nodes, and their parent 55 becomes a simple node that now belongs to class 22.



7. Insert (28), Insert (35) → (g): 28 and 35 are inserted into leaf class 32.



8. Insert (30) → (h): 30 is inserted into leaf class 32 to the right of the node with the value 28. This class overflows, and the situation is resolved by a partitioning operation. Nodes 28 and 35 become class nodes, and 32 becomes a simple node that is added to the mother class 22. Once again, mother class 22 overflows, and as it has only one child (node 55), it is restructured by a double rotation around node 32.



4 Delete Operation

The delete operation proceeds in two steps:

Step 1: Similar to a BST.
The node is always deleted from a leaf class.
- If the deleted node is a simple node inside this leaf class, the delete process terminates.
- If the deleted node is a class node that has one child, it becomes a class node. The delete process terminates.
- If the deleted node is a class node that has no children, the leaf class underflows, and the delete process continues to step 2.

Step 2: Going upward the tree from the deleted node by considering the following cases:

Case 1: Departitioning

Departitioning is invoked when a class underflows, i.e., it becomes empty (after a delete operation), and its direct sister class exists and contains only one element. This situation is illustrated in Figure 4(a1) (or 4(b1)), where class C underflows and its direct sister class B contains only one element.

Departitioning involves eliminating the direct sister class. One way to achieve this is by transforming the single element of the sister class

into a simple node and integrating it into the mother class. If the mother class contains more than one element (Figure 4(a1)), the parent node of the direct sister class must be converted from a simple node into a class node. In other words, an element is removed from the mother class, and the delete process terminates.

However, if the mother class contains only one element (Figure 4(b1)), the parent node of the direct sister class is already a class node. In this case, the departitioning process continues upward through the tree, treating the mother class as an underflow class. This is the only scenario in which the delete process cascades.

Departitioning does not require restructuring and simply involves modifying the type of one or two nodes: the sister class of the underflow class becomes a simple node, while its parent can be transformed into a class node. Figure 4(a2) and Figure 4(b2) show the results of the departitioning operation corresponding to Figure 4(a1) and Figure 4(b1), respectively. In the case of Figure 4(b2), the delete process continues upward through the tree.

It is important to note that if C is a leaf class, it should not be present in Figure 4(a2) and Figure 4(b2).

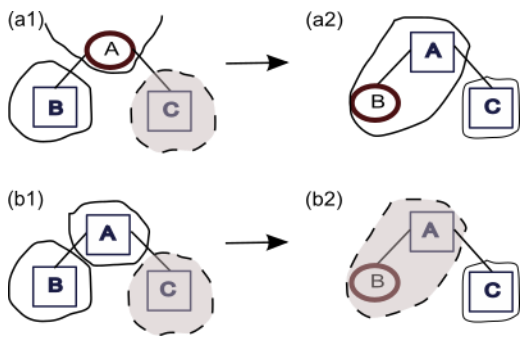


Fig. 4: The underflow class has a direct sister class with no children

Source: created by the author

The code for Departitioning (A, Dir) is as follows: (A is the parent of the underflow class. Dir has a value of 'Left' if the underflow class is to the left, 'Right' otherwise)

```
Ass_Kind(A, 'Class')
Case Dir of
    Left: Ass_Kind(Right(A), 'Simple')
    Right: Ass_Kind(Left(A), 'Simple')
End
```

The code for Departitioning correctly assigns the kind of nodes based on the direction of the underflow class. When the underflow class is to the left, the right child of node A is assigned the kind 'Simple'. When the underflow class is to the right, the left child of node A is assigned the kind 'Simple'. This code effectively updates the types of nodes during the departitioning process.

Case 2: Reorganizing

This operation is invoked when the underflow class has a direct sister class with one child, and their parent is a simple node. Figure 5(a1) in Appendix represents such a situation where C is the underflow class and B is its direct sister class with a left child, while their parent is a simple node. Figure 5(b1) in Appendix represents a similar situation in which the direct sister class has a right child. In these situations, we consider the direct sister class augmented with its parent node. The parent node can be seen as an overflow class (without considering the kinds of nodes) that are resolved using the same approach as in the insert process, i.e., by one or two rotations and possibly some type changes. The delete process terminates. Figure 5(a2) and Figure 5(b2) in Appendix show the augmented classes. A reorganization of these augmented classes gives the situations depicted in Appendix in Figure 5(a3) and Figure 5(b3), where the conflict is resolved.

Notice also that if C is a leaf class, it should not be present in Figure 5(a3) and Figure 5(b3) in Appendix.

The code of Reorganizing (A, Dir) is as follows: (A is the parent of the underflow class. Dir has a value of 'Left' if the underflow class is to the left, and 'Right' otherwise)

```
P := Balancing(A, Double)
{ Double is true if a double rotation is performed }
If Double Then
  Begin
    Ass_Kind(P, 'Class')
```

```
    Case Dir of
    Left : Ass_Kind( Right(P), 'Simple' )
    Right : Ass_Kind( Left(P), 'Simple' )
   End
  End
```

The Balancing procedure is implemented with two additional output parameters:
- Dir is set to 'Left' when the class to balance does not have a left child, and 'Right' otherwise.
- Double is set to be true if a double rotation is performed, and false otherwise.

Recall that the Balancing procedure always performs one rotation when possible, otherwise it performs two rotations. If two rotations are performed, the output parameter Double is set to true. Therefore, in the Reorganizing process, the node types change only when a double rotation is performed.

Note that in the case of Figure 5(a3) in Appendix, no type change is performed.

Case 3: Restructuring & Partitioning

This operation is invoked in the following two situations:
- The underflow class has a direct sister class with two children, and their parent is a simple node. This is illustrated in Figure 6(a1) in Appendix.

- The underflow class has a direct sister class with one or two children, and their parent is a class node. This is depicted in Appendix in Figure 7(b1), Figure 7(c1), and Figure 7(d1).

These cases simply represent the negation of case 2 (Reorganizing), i.e., NOT (The underflow class has a direct sister class with one child and their parent is a simple node). This can be equivalently stated as follows: the underflow class has a direct sister class with two children, OR their parent is a class node.

In these situations, we also consider the direct sister class augmented with its parent node, which represents an overflow class. Figure 6(a2), Figure 7(b2), Figure 7(c2), and Figure 7(d2) in Appendix show the augmented classes of class B. All of these situations are resolved in the same way, i.e., by restructuring-partitioning. The delete process terminates, and the resulting new situations are shown in Appendix in Figure 6(a3), Figure 7(b3), Figure 7(c3), and Figure 7(d3).

The code of Restructuring-Partitioning (A) is simply the following: (A is the parent of the underflow class)

```
P := Balancing (A)
Ass_Kind(P, Kind(A))
Ass_Kind(Left(P), 'Class')
Ass_Kind(Right(P), 'Class')
```

The code of Restructuring-Partitioning performs the restructuring and partitioning operations in the tree. It starts by calling the Balancing procedure on the parent node A to obtain the balanced node P. The kind of P is set to the same as A, indicating that it is still a class node.

Then, the left child of P is assigned the kind 'Class', indicating that it becomes a class node after the restructuring. Similarly, the right child of P is also assigned the kind 'Class'.
This code turns the underflow class and its parent into two new class nodes, all while keeping the tree's overall structure intact. It makes sure that the restructuring and partitioning operations are applied properly, which helps the deletion process move forward and effectively handle any underflow situations.

Case 4: Transforming

This case occurs when the underflow class does not have a direct sister class. In this situation, the parent of the underflow class must be a class node, and its sibling node, which becomes a simple node, must have a class node to its right if the underflow class is to the right (or a class node to its left if the underflow class is to the left).

To resolve this situation, a simple rotation is performed inside the class rooted at the parent node. This rotation involves interchanging the kinds of the parent node and its child. The process then continues according to cases 1, 2, or 3.

Figure 8(a1) depicts such a situation where a right rotation of node A is performed to resolve the underflow. The result of this rotation is shown in Figure 8(a2).

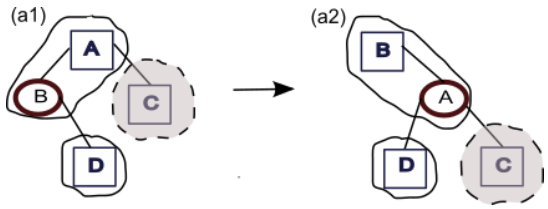


Fig. 8: The underflow class does not have a direct sister class

Source: created by the author

The corrected text is as follows:
The code of Transforming (A, Dir) is as follows: (A is the parent of the underflow class, Dir has the value 'Left' if the underflow class is to the left, 'Right' otherwise)
Left_Rotate(A) (Right_Rotate(A)) performs a left (right) rotation around node A.

```
Ass_Kind(A, 'Simple')
Case Dir of
Left: Begin
        Ass_Kind(Right(A), 'Class');
        Left_Rotate(A)
      End
Right: Begin
        Ass_Kind(Left(A), 'Class');
        Right_Rotate(A) End
      End
```

The code performs a transformation of the parent node and its child to resolve the underflow situation. If the underflow class is to the left, it assigns the kind 'Class' to the right child of the parent node and performs a left rotation around the parent node. If the underflow class is to the right, it assigns the kind 'Class' to the left child of the parent node and performs a right rotation around the parent node. This code effectively interchanges the kinds of the parent node and its child to reorganize the tree structure and continue the delete process accordingly.

Delete algorithm

In the delete process, an element is first removed from a leaf class. If the leaf class underflows, the pseudo-algorithm depicted in Appendix in Algorithm 2 is applied. The algorithm utilizes the stack (Branch) containing all the traversed nodes from the root of the tree (Tree) to the parent (Parent) of the underflow class. The pop operation on the stack retrieves this parent node.

The algorithm also includes a procedure called Sibling(Node), which returns the sibling (brother) of a given node Node.

Comment and analysis

An item is always deleted from a leaf class. If the leaf class contains more than one element, the delete process terminates. Otherwise, the leaf class underflows and the tree is reorganized according to the following steps:

If the underflow class does not have a direct sister class, a transforming operation (simple rotation) is performed to establish a direct sister class. The process then continues to step (ii).

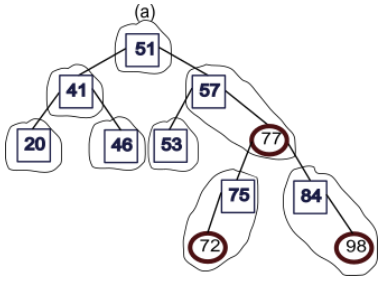
(ii) If the underflow class has a direct sister class with only one node (no children), a departitioning operation is performed. If the parent of the sister class is not the only node in the mother class, the delete process is stopped. Otherwise, the mother class is treated as an underflow class, and the process continues upward up the tree in the same way.
(iii) If the underflow class has a direct sister class with more than one node (one child or two children), two subcases are considered:
- If the sister class has one child and its parent is a simple node, a reorganizing operation is performed.
- Otherwise, a restructuring-partitioning operation is performed, involving the sister class augmented by its parent node.

In general:
- An underflow class can be null or a failing class.
- A failing class is only created in the case of a split operation.
- During a conflict, we always choose a node from the parent class.

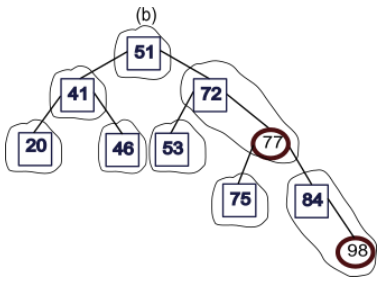
It can be observed that at most one transforming operation, at most one reorganizing operation, and most log (N) departitionings are performed. In terms of Red-Black trees, the transforming operation is considered as restructuring. Thus, the overall results are similar to those of Red-Black trees.

Example scenario

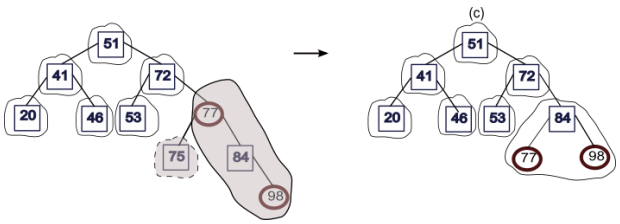
We start from the following initial tree:



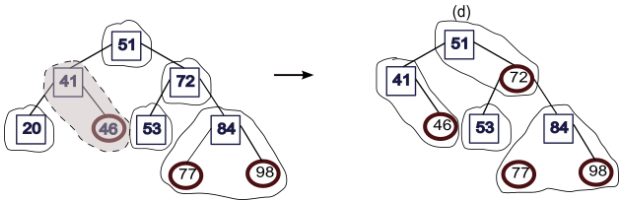
1. Delete (57) → (b): 72 is removed from class 75 to replace 57.



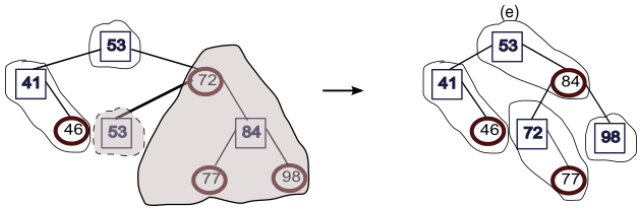
2. Delete (75) → (c): Class 75 underflows and becomes empty. Since it has a direct sister class with one child and their parent is a simple node, a reorganizing operation is performed. A left rotation is carried out on the node with the value 77.



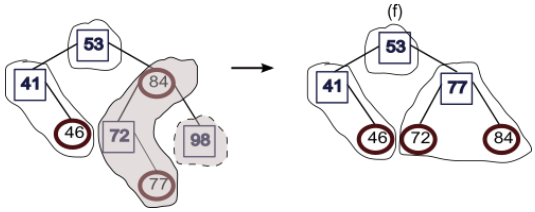
3. Delete (20) → (d): Class 20 underflows and its direct sister class includes one element. A departitioning operation is performed to create the new class {41, 46}. As the parent of the underflow class was a class node, the process continues, and now class 41 becomes the new underflow class. Since its direct sister class also contains one element, a departitioning operation is performed, resulting in the class {51, 72}. As this new class becomes the root class, the process terminates.



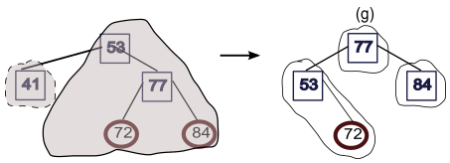
4. Delete (51) → (e): 53 replaces 51. Class 53 becomes empty and has a direct sister class with two children. A restructuring followed by a partitioning is performed. Since the parent node is simple, the delete process terminates.



5. Delete (98) → (f): Class 98 underflows. This class has a direct sister class with one child. A restructuring operation is performed, and the delete process terminates as the parent of the two classes is a simple node.



(f) Delete (46, 41) → (g): Deleting 46 and 41 results in an underflow situation. This time, it is resolved by a restructuring-partitioning operation.



5 EBST with Simplified Algorithms

The main goal of this paper is to show that the structure we've proposed can be a viable alternative to Red-Black trees, providing simpler algorithms in

the process. To accomplish this, we address the following points:
- Equivalence between the two structures
- Correspondence between the maintenance operations of the new structure and those of Red-Black trees
- Summary of the code for maintenance operations
- Simplicity of the insertion and deletion algorithms

Equivalence between the two structures

The new tree structure becomes equivalent to a Red-Black tree when a class node is replaced by a black node and a simple node is replaced by a red node.

Correspondence between the maintenance operations of the new structure and those of Red-Black trees

Figure 9 and Figure 10 depict the correspondences between the Red-Black tree cases (RB cases) and the new structure cases (EBST cases) in the insert and delete processes, respectively.

Regarding the insert algorithm, it is evident that EBST case 1 corresponds precisely to RB case 1, and EBST case 2 corresponds exactly to both RB cases 2 and 3.

For the delete algorithm, it is apparent that EBST cases 1 and 3 correspond exactly to RB cases 1 and 4, respectively. However, EBST cases 2 and 3 differ entirely from RB cases 2 and 3.

The "Reorganizing" operation covers the following sub-cases of RB cases 2 and 3:
- When the parent is coloured red, and the sibling node S has only one red child in the same direction as S. Two nodes are said to be in the same direction if they are both either left children or right children.
- When the parent is coloured red, and the sibling node S has only one red child in the reverse direction compared to S.

The "Restructuring-partitioning" operation encompasses other sub-cases of RB cases 2 and 3.
The sub-cases of RB case 2 include:
- When the parent is coloured red, and the sibling node S (black) has two red children.
- When the parent is coloured black, and the sibling node S (black) has two red children.
- When the parent is coloured black, and the sibling node S (black) has one red child in the same direction as S.

There is only one sub-case for RB case 3:
- When the parent is coloured black, and the sibling node (black) has one red child in the reverse direction of S.

The restructuring-partitioning operation resolves the conflicts in the same manner as Red-Black tree delete cases 2 and 3. However, the reorganizing operation differs. The reorganizing operation can avoid three types of changes compared to Red-Black tree case 2 and modify node types differently compared to Red-Black tree case 3. As a result, it generates a new form of the tree. Fewer class nodes are generated compared to Red-Black tree black nodes, and more simple nodes are generated compared to red nodes.

In the next section, we will demonstrate that this alteration yields significant performance benefits. We will observe a reduction in the number of type changes and rotations, thereby improving execution time while preserving search performance.

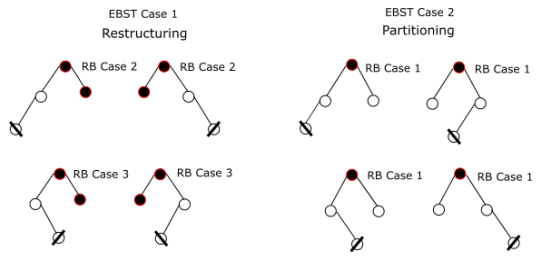


Fig. 9: EBST cases – RB cases correspondence - Insert process

Source: created by the author

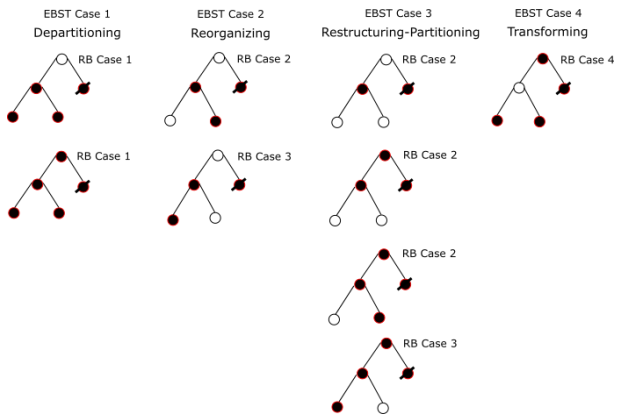


Fig. 10: EBST cases – RB cases correspondence - Delete process

Source: created by the author

Summary of the code for maintenance operations

The new structure's maintenance operations are simple to implement. All operations are type changes, with some initially balancing a class (subtree). Each operation consists of a maximum of three type changes and a maximum of two rotations.

The insert operation uses one of two procedures, Restructuring (A) and Partitioning (A).

- Restructuring (A) performs one or two rotations to balance a class A overflow with one child (subtree), thereby changing the types of nodes.
- Partitioning (A) takes a class A overflow with two children and turns it into two classes, thereby changing the types of nodes again.

The delete operation uses four procedures: Departitioning (A, Dir), Reorganizing (A, Dir), Restructuring-partitioning (A), and Transforming (A, Dir).

- Departitioning (A, Dir) is the inverse of Partitioning. It eliminates one class without requiring rotations. A represents the parent of the eliminated class, and Dir takes the value 'Left' if the underflow class is on the left, and 'Right' otherwise.
- Reorganizing (A, Dir) is essentially a restructuring procedure. A type change occurs only when a double rotation is performed. This procedure involves restructuring an augmented class.
- Restructuring-partitioning (A) combines a balancing operation with an immediate partitioning operation. Similar to Reorganizing, it aims to restructure an augmented class.
- Transforming (A, Dir) executes a rotation around A (the mother class) in the appropriate direction. A represents the mother class of the underflow class, and Dir takes the value 'Left' if the underflow class is on the left, and 'Right' otherwise.

Simplicity of the insertion and deletion algorithms

A simple analysis of the two data structures reveals that the algorithms for Red-Black trees involve rotations and colour changes, while the algorithms for the new structure involve subsets and maintenance operations on these subsets (such as Structuring, Partitioning, Departitioning, etc.).

Fig. 11: Simplicity of the insertion algorithm

Source: created by the author

Figure 11 illustrates the sequences of operations performed on both data structures during the insertion of 5000 elements. In the context of the new

structure, "N" denotes that no maintenance operation is executed. "P" signifies that a partitioning operation is executed, and "S" indicates the execution of a Restructuring operation (which involves one or two rotations). As for the Red-Black tree, "R" denotes a single rotation, and "D" signifies a double rotation. "M" is used to represent a colour change, while "Mi" represents the change of "i" colours.

We believe that the new structure is simpler and more pedagogical compared to the Red-Black tree. The same observation applies to the deletion process as well.

6 EBST with Enhanced Deletion

The second objective of this paper is to demonstrate that the deletion algorithm in the new structure outperforms the standard algorithm. As mentioned in the previous section, the deletion algorithm constructs a completely different tree that favours the presence of simple nodes over class nodes. This difference leads to more efficient data deletions.

To achieve this, we implemented the search, insert, and delete algorithms for the two data structures using the Delphi programming language. Our simulation was primarily based on the following two patterns:

(i) Deleting all N items from an existing tree in a random order.

(ii) Deleting/inserting N random items from/into an existing tree with N items.

During the simulation, we analyzed the number of type (or colour) changes, rotations, and the overall execution time. The tree nodes stored integers within the range of [0, 999,999,999].

We will demonstrate that in applications involving frequently repeated deletions, the deletion algorithm in the new structure performs more efficiently. Specifically, we observed a significant reduction in the number of type changes and a slight decrease in the number of rotations. The execution times observed further confirm the superiority of the new structure over the Red-Black tree when items are repeatedly deleted.

In most applications utilizing Red-Black trees, insert and delete operations are commonly performed. Here, we will demonstrate that the new structure outperforms the Red-Black tree as well. The new structure's delete algorithm entails fewer type changes and rotations compared to the Red-Black tree. Additionally, the new structure's insert algorithm incurs fewer type changes and rotations. However, it is an interesting trade-off as the overall execution time is reduced while maintaining search performance.

The simulation tables are presented below:

- The "N" column indicates the sizes of the initially generated Red-Black trees and new trees.
- The "RB" and "EBST" columns present the results obtained from the Red-Black tree and the new structure, respectively.
- The "(RB-EBST)/RB" column represents this improvement as a percentage of the Red-Black tree.

The values provided are the average results obtained from three tests. The last lines of the tables show the average values for the respective columns. All times are measured in milliseconds.

For each simulation pattern, we will begin by outlining the simulation algorithm.

Deleting in random order all the items of an existing tree

The following experiment was conducted:

1. Build a Red-Black tree and the new structure using the same sequence (S1) of N random integer values.
2. Generate a random sequence (S2) of N delete operations. Each deletion removes a randomly chosen existing item from the tree.
3. (a) Execute sequence S2 separately on the Red-Black tree and the new structure.

(b) Calculate the following for each delete algorithm:

- The total number of colour (or type) changes and rotations performed.
- The overall execution time of the delete algorithm for each type of tree.

2. Repeat steps 1 to 3 three times for N ranging from 100,000 to 1,000,000, with a step increment of 100,000 nodes.

Type (or colour) changes

Table 1 displays the number of colour (or type) changes performed by the Red-Black tree delete algorithm and the new structure delete algorithm in columns "RB" and "EBST" respectively. These operations were carried out while deleting all the items from the corresponding previously built trees in a random order.

Column (RB - BEST) /RB" represents the gain as a percentage achieved by the new delete algorithm compared to the Red-Black tree algorithm. It is noteworthy that the average percentage gain is 14.71% (as shown in the last column of Table 1). For instance, when deleting all the items from a tree with 500,000 nodes (row 5 in Table 1), the Red-Black tree delete algorithm performs 981,032 colour changes, whereas the new structure delete algorithm performs 836,385 type changes. In other words, the new algorithm achieves a savings of 144,647 type changes.

Table 1. Color (Type) changes – Delete algorithm

N	RB	EBST	(RB-EBST)/RB
100,000	195,980	167,246	14,66%
200,000	392,866	335,089	14,71%
300,000	589,062	502,581	14,68%
400,000	785,156	669,839	14,69%
500,000	981,032	836,385	14,74%
600,000	1,177,913	1,004,196	14,75%
700,000	1,374,298	1,171,353	14,77%
800,000	1,570,100	1,340,041	14,65%
900,000	1,765,978	1,505,916	14,73%
1,000,000	1,961,009	1,672,878	14,69%
			14,71%

Source: created by the author

Rotations

Table 2 presents the numbers of rotations performed by the Red-Black tree delete algorithm and the new structure delete algorithm in columns "RB" and "EBST" respectively. Column "(RB - BEST) /RB" represents the gain as a percentage achieved by the new delete algorithm compared to the Red-Black tree algorithm. On average, the gain in rotations is minor, approximately 1.82%. However, it is worth noting that the new structure delete algorithm does have an impact on the number of rotations performed.

Table 2. Rotations – Delete algorithm

N	RB	EBST	(RB-EBST)/RB
100,000	48,895	48,078	1,67%
200,000	98,487	96,582	1,93%
300,000	147,589	144,811	1,88%
400,000	196,208	192,706	1,78%
500,000	245,209	240,707	1,84%
600,000	294,592	289,205	1,83%
700,000	343,900	337,662	1,81%
800,000	392,848	385,720	1,81%
900,000	441,743	433,544	1,86%
1,000,000	490,376	481,430	1,82%
			1,82%

Source: created by the author

Execution time

Table 3 displays the execution times in milliseconds for the Red-Black tree and the new structure, presented in columns "RB" and "EBST" respectively. The last column represents the gain in milliseconds, as a percentage, achieved by the new structure compared to the Red-Black tree.

Table 3. Execution time - Delete algorithm

N	RB	EBST	(RB-EBST)/RB
100,000	900	854	5,47%
200,000	1,947	1,827	6,55%
300,000	3,026	2,844	6,41%
400,000	4,124	3,895	5,89%
500,000	5,255	4,911	6,99%
600,000	6,379	5,990	6,50%
700,000	7,573	7,161	5,76%
800,000	8,692	8,250	5,36%
900,000	9,973	9,407	6,02%
1,000,000	11,129	10,494	6,05%
			6,10%

Source: created by the author

The table demonstrates that the new structure's delete algorithm outperforms the Red-Black tree delete algorithm in terms of execution time. The average gain in execution time, as shown in the last column, is 6.10%. For instance, according to row 5, when performing 500,000 insert/delete operations on both data structures with 500,000 nodes, the new structure saves 343 milliseconds compared to the Red-Black tree algorithm.

We conducted the following experiment involving deleting/inserting N random items from/into an existing tree with N items:
1. First, we built a Red-Black tree and the new structure using the same sequence (S1) of N random integer values.
2. Next, we created a random sequence (S2) of N insert and delete operations. An insertion adds a new item to the tree, while a deletion removes an existing item chosen randomly from the tree.
3. (a) We ran sequence S2 separately on the Red-Black tree and the new structure.
(b) We computed the following:
- The total number of colour (or type) changes and rotations performed by each delete algorithm.
- The overall time taken by both the insert and delete algorithms for each type of tree.
4. We repeated steps 1-3 three times for N values ranging from 100,000 to 1,000,000 in increments of 100,000 nodes.

It's important to note that the number of inserted items in sequence S2 is equal to the number of deleted items.

Type (or colour) changes

Table 4 in Appendix presents the results of the number of colour changes performed on the Red-Black tree (RB) and the new structure (EBST) during the process of deleting/inserting items in random order on the corresponding trees that were previously built.
In each column, there are sub-columns for "Global," "Delete," and "Insert," representing the total number of colour (type) changes made by both the insert and delete operations, by only the delete algorithm, and by only the insert algorithm, respectively.
The column "(RB-EBST)/RB" shows the percentage difference between these numbers. The last row provides the average values. Positive values indicate a favourable outcome for the new structure.

The results demonstrate a significant gain (averaging 19.17%) in type changes made by the new delete algorithm compared to the Red-Black tree's delete algorithm. On the other hand, the insertions in the new structure require more type changes (averaging 7.16%) compared to the Red-Black tree. The average gain overall is 4.36%.

For example, when performing 500,000 insert/delete operations on both types of trees with 500,000 nodes (Table 4, Row 5, Appendix), the Red-Black tree undergoes 754,182 colour changes, while the new structure experiences 721,369 type changes. This indicates a 4.35% saving in the new structure. Specifically, the Red-Black tree's delete algorithm performs 338,566 colour changes, while the new structure's delete algorithm performs 273,606 type changes. Thus, the new structure achieves a 19.19% reduction in colour changes during the deletion process. However, the alteration of the tree caused by the new structure's delete algorithm requires more type changes (7.18%) during insertions to maintain the properties of the new structure. These results confirm the analysis conducted in section 6 effectively.

Rotations

Table 5 (Appendix) displays the number of rotations performed by the Red-Black tree (RB) delete algorithm and the new structure (EBST) delete algorithm while Deleting/inserting items in random order on the corresponding trees that were previously built.

In each column, there are sub-columns for "Global," "Delete," and "Insert," representing the total number of rotations made by both the insert and delete operations, by only the delete algorithm, and by only the insert algorithm, respectively.

The column "(RB-EBST)/RB" provides the percentage differences between these numbers. The

last row presents the average values. Positive values indicate a preference for the new structure.

The results will show a very small gain of 0.88% more rotations by the new delete algorithm than that of the Red-Black tree delete algorithm. On the other hand, insertions in this new structure have to perform 1.26% more rotations than those in the Red-Black tree. There is an average loss overall.

Take, for instance, 500,000 insert/delete operations on both trees of 500,000 nodes (see Table 5, Row 5, in Appendix). The Red-Black tree will have to perform 226,684 rotations; the new structure will perform 227,482 rotations. This shows a saving of only 0.35% that goes to the Red-Black tree. More specifically: 98,304 rotations are performed by the delete algorithm of this new structure; that of the Red-Black tree performs delete rotations totaling just 97,389. In effect, this makes for a reduction of rotation during deletion by about 0.93% for the new structure.

The findings suggest that there is only a minimal effect on the number of rotations, with a slight advantage for the Red-Black tree in terms of efficiency.

Execution time

Table 6 shows the execution times in milliseconds for both the Red-Black tree (RB) and the new structure, which we'll refer to as the EBST. In the column labeled "(RB-EBST)/RB," you'll find the percentage of time saved, measured in milliseconds, thanks to the new structure across all operations. As you can see, the data clearly demonstrates that the EBST outperforms the Red-Black tree when it comes to execution time. On average, there is a 5.12% improvement in execution time, as noted in the final column. For example, in row 5 of Table 6, the new structure reduces the time by 170 milliseconds when executing 500,000 insert/delete operations on both data structures with 500,000 nodes.

These findings further confirm that the delete algorithm of the new structure maintains search performance even when insertions and deletions occur simultaneously. In these situations, items are searched prior to being added or removed. The following section will explore this aspect in greater detail.

Table 6. Execution time – Insert/Delete algorithms

N	RB	EBST	(RB-EBST)/RB
100,000	629	599	4,87%
200,000	1,333	1,271	4,63%
300,000	2,067	1,958	5,30%
400,000	2,807	2,655	5,40%
500,000	3,582	3,412	4,76%
600,000	4,333	4,112	5,09%
700,000	5,177	4,911	5,13%
800,000	5,963	5,650	5,25%
900,000	6,791	6,433	5,28%
1,000,000	7,661	7,245	5,44%
			5,12%

Source: created by the author

7 Modern Applications

The proposed data structure is a self-adjusting binary search tree, which is evidence that the tree must remain balanced. This property facilitates faster insertion, deletion, and find operations and therefore will operate in O(log(n) time. Although this data structure has been debated in the context of theory and algorithms in the context of Computer Science, these self-balancing binary search trees are robust data structures that can be useful in countless modern applications to increase efficiency, accessibility, and resource management. Below are some modern applications:

- Database Management: This efficiently implements associative containers in the form of symbol tables, which is indispensable in indexing in relational databases.
- File Systems: This can be used for file lookups in file systems. In its modified form, the self-balancing binary search tree provides efficient indexing of directories in file systems.
- Dynamic Memory Allocation: This is useful for managing free memory blocks efficiently while improving the utilization of memory resources in dynamic memory allocation systems.

- Networking: This is useful for maintaining sorted lists of IP addresses in a routing table in a networking infrastructure so that a lookup or update can be completed efficiently.
- Compilers: The self-balancing binary search tree is effective for implementing a symbol table for variable scope and function declarations in compilers.
- Web Applications: Session management can occur with self-balancing binary search trees by sorting the sessions by access time, allowing for effective eviction policies in caching.

8 Conclusion

This paper introduces a new balanced binary search tree based on class partitioning with the additional overhead of one extra bit. Each class in the tree represents a height 0 or 1 sub-tree with up to 3 items. The data structure consists of two kinds of nodes: simple nodes and class nodes. Class nodes ensure the tree is balanced, and simple nodes allow for the tree to be unbalanced. When the height of a class is 2 and overflows, it is either partitioned to maintain balance, if it is already balanced, or restructured to bring it into balance. Restructuring will ensure that a class is balanced by using the sub-tree. We think that the new insertion and deletion algorithms are more straightforward and easier to understand than those traditionally used on Red-Black trees, since they are described in terms of subsets as opposed to colours. Overall, where insertions and deletions are occurring together, which is true for most applications using Red-Black trees, the new structure decreases the number of type changes (colour changes) by 4.36% with a slight 0.34% increase in rotations. The new structure also provides a 5.12% improvement in execution time with equally efficient search execution.

Furthermore, the delete algorithm in the new structure outperforms the one used in Red-Black trees in the following ways:

- It reduces type changes by an average factor of 14.71% when items are consistently deleted and by 9.17% on average when insertions and deletions are interspersed.
- It reduces rotations by an average factor of 1.82% when items are consistently deleted and by 0.88% on average when insertions and deletions are interspersed.
- It improves execution time by an average factor of 6.10% when items are consistently deleted.

We also believe that the new structure shows great promise by combining algorithmic simplicity with improved efficiency, making it a compelling choice for various applications.

Declaration of Generative AI and AI-assisted Technologies in the Writing Process

The authors wrote, reviewed and edited the content as needed and verifies that none utilized artificial intelligence (AI) tools were used. The authors take full responsibility for the content of the publication.

Acknowledgements

We would like to express special thanks to my dear colleague Pr W.K HIDOUCI for his advice and helpful comments and suggestions.

References

[1] Knuth, D. E. (1973). *The art of computer programming.* Vol III: Sorting and Searching, Addison Wesley, Reading, Mass.

[2] Aho A. V, J. E. Hopcroft, J. D. Ullman (1983). Data Structures and Algorithms. Addison Wesley, Reading, Mass.

[3] Sedgewick R. (2004): Fundamentals data structures. Addison – Wesley. 13th Printing.

[4] Brass P. (2008). Advanced Data Structures. Cambridge University Press.

[5] A. Drozdek, Data Structures and Algorithms in C++, 2nd ed. Pacific Grove, CA, USA: Brooks/Cole, 2001.

[6] Adel'son-Velskii, G. M., and Y. M. Landis (1962). An algorithm for the organization of information. *Dokl. Akad. Nauk SSSR* 146, pp. 263-266.

[7] Guibas, L.J. and Sedgewick R (1978). A Dichromatic Framework for Balanced Trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science,* October 16-18, 1978.

[8] Anderson A. Balanced Search trees made simple (1993). Workshop on algorithms and Data Structures, pages 60-71, Springer Verlag.

[9] Bayer R (1971). Binary B-Trees for virtual memory. *Proc. ACM Sigifidet Workshop on data description, access and control,* pp.219-235.

[10] Bayer, R (1972). Symmetric Binary B-Trees: Data Structure and Maintenance Algorithms. *Acta Informatica* 1(4):290-306, 1972.

[11] Stout Q. F. and WARREN B. L (1986). Tree Rebalancing in Optimal Time and Space. *Communications of the ACM.* Vol. 29 Number 9.

[12] Bayer, R. and E. M. McCreight (1972). Organization and maintenance of large ordered indices. *Acta Informatica,* 1:3, pp173-189.

[13] Zegour D.E, Partitioned Binary Search Trees (P(h)-BST): A Data Structure for Computer RAM; Book: Data Science with Semantic Technologies: Theory, Practice, and Application ‘Chapter 6), 2022, Willey.

Contribution of Individual Authors to the Creation of a Scientific Article (Ghostwriting Policy)

The author contributed in the present research, at all stages from the formulation of the problem to the final findings and solution.

Sources of Funding for Research Presented in a Scientific Article or Scientific Article Itself

No funding was received for conducting this study.

Conflict of Interest

The author has no conflicts of interest to declare that are relevant to the content of this article.

APPENDIX

Algorithm 1 : Insertion Maintenance Algorithm

```
While (Parent <> Nil) Do
  Begin
    Pop(Branch,Grand_parent) { Grand_parent Always <> Null }
    Pop (Branch, Parent)
    If the class rooted at Grand_parent has only one child
    Then
      Begin
        Q:=RESTRUCTURING (Grand_parent)
        If Parent <> Null
        Then Modify node Parent to point now Q
        Else Tree := Q
        Exit
      End
    Else { Two Children }
      Begin
        PARTITIONING (Grand_parent)
        If Parent <> Null
        Then
          If Kind (Parent) = 'Class' Then Exit
      End
  End
```

Algorithm 2 : Deletion Maintenance Algorithm

```
While (Parent <> Nil) Do
  Begin
    Grand_Parent := Top(Branch)
    C':= Sibling(C )
    If Kind (C') = 'Simple'
    Then
      Begin
        C'':=TRANSFORMING (C') { C'' is the new direct sister class of C }
        If Grand_Parent <> Null
        Then Modify node Grand_Parent to point now C'
        Else Tree := Parent
        Grand_Parent := C'  ;  C':=C''
      End
    If C' has no children
    Then
      Begin
        DEPARTITIONING (Parent)
        If Kind(Parent)= 'Class'
```

```
            Then   Pop(Branch, Parent)
            Else Exit
        End
    Else { Class with more than one element }
        Begin
            If (Kind (Parent) = 'Simple') AND C' has one child
            Then Q:= REORGANIZING(Parent)
            Else  Q: = RESTRUCTURING_PARTITIONING(Parent)

            If Grand_Parent <> Null
            Then Modify node Grand_Parent to point now Q
            Else Tree := Q
            Exit
        End
End
```

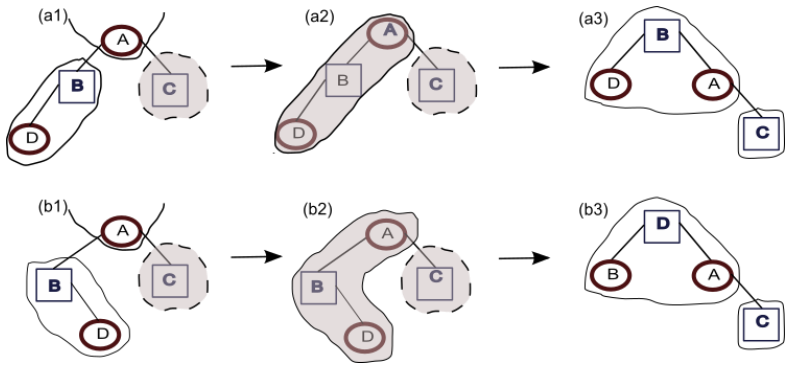


Fig. 5: The underflow class has a direct sister class with one child and its parent is a simple node

Source: created by the author

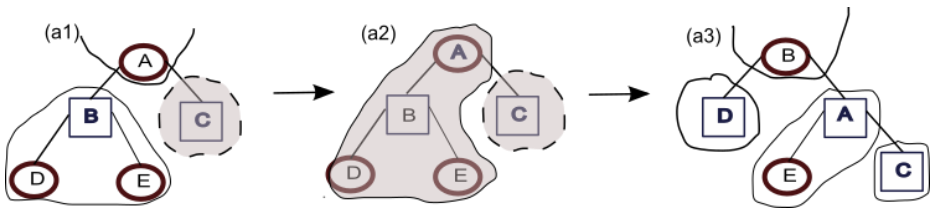


Fig. 6: The underflow class has a direct sister class with two children and their parent is a simple node

Source: created by the author

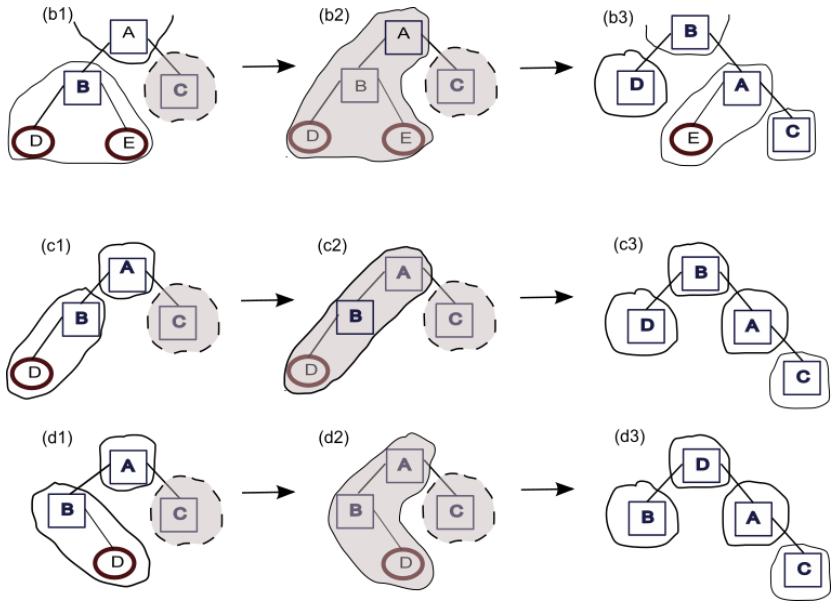


Fig. 7: The underflow class has a direct sister class with one or two children and their parent is a class node

Source: created by the author

Table 4. Color (Type) changes – Insert/Delete algorithms

	RB			**EBST**			**(RB-EBST)/RB**		
	Colour changes			**Type changes**			**Differences in %**		
N	Global	Delete	Insert	Global	Delete	Insert	Global	Delete	Insert
100,000	151,660	68,234	83,427	145,200	55,224	89,976	4,26%	19,07%	-7,28%
200,000	302,258	135,653	166,605	288,957	109,423	179,534	4,40%	19,34%	-7,20%
300,000	452,344	202,894	249,450	432,655	163,785	268,871	4,35%	19,28%	-7,22%
400,000	604,294	271,266	333,028	577,898	219,772	358,125	4,37%	18,98%	-7,01%
500,000	754,182	338,566	415,616	721,369	273,606	447,763	4,35%	19,19%	-7,18%
600,000	906,212	406,909	499,302	865,308	328,945	536,363	4,51%	19,16%	-6,91%
700,000	1,053,219	473,216	580,003	1,007,121	382,052	625,069	4,38%	19,26%	-7,21%
800,000	1,204,109	540,355	663,754	1,152,573	437,116	715,457	4,28%	19,11%	-7,23%
900,000	1,354,989	608,614	746,375	1,295,935	491,812	804,123	4,36%	19,19%	-7,18%
1,000,000	1,508,199	678,133	830,066	1,442,786	548,196	894,590	4,34%	19,16%	-7,21%
							4,36%	**19,17%**	**-7,16%**

Source: created by the author

Table 5. Rotations – Insert/Delete algorithms

	RB			EBST			(RB-EBST)/RB Differences in %		
N	Global	Delete	Insert	Global	Delete	Insert	Global	Delete	Insert
100,000	45,589	19,825	25,765	45,724	19,628	26,096	-0,30%	0,99%	-1,27%
200,000	90,727	39,266	51,461	91,043	38,906	52,137	-0,35%	0,92%	-1,30%
300,000	135,808	58,913	76,895	136,266	58,292	77,974	-0,34%	1,05%	-1,38%
400,000	181,279	78,716	102,564	181,906	77,980	103,925	-0,34%	0,93%	-1,31%
500,000	226,684	98,304	128,380	227,482	97,389	130,093	-0,35%	0,93%	-1,32%
600,000	272,183	118,099	154,084	273,161	116,978	156,184	-0,36%	0,95%	-1,34%
700,000	316,965	137,713	179,252	318,049	136,400	181,649	-0,34%	0,95%	-1,32%
800,000	362,239	155,641	206,598	363,454	155,416	208,038	-0,33%	0,14%	-0,69%
900,000	407,935	177,249	230,686	409,304	175,466	233,838	-0,33%	1,01%	-1,35%
1,000,000	453,581	197,075	256,505	455,105	195,174	259,931	-0,34%	0,96%	-1,32%
							-0,34%	**0,88%**	**-1,26%**

Source: created by the author