ISSN 2007-9737

Partitioned Trees

Fahd Mustapha Meguellati*, Djamel Eddine Zegour, Seyfeddine Zouana

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

{f_meguellati, d_zegour, s_zouana}@esi.dz

Abstract. We introduce the Partitioned Trees, a form of Partitioned Binary Search Tree parameterized to represent both Red-Black trees and a family of partially balanced Binary Search Trees. Partitioned Tree is interesting not only because it provides the same time and space complexity as Balanced Binary Search trees $O(\log n)$, but also because it's simple to implement, easily understandable, and highly adaptable in different fields where rebalancing is costly. We outline the various maintenance operations and insertion and deletion algorithms employed by the proposed data structure. Additionally, we conduct an in-depth analysis on the worst-case height of Partitioned Trees followed by a comparison of Partitioned Trees and Red-Black Trees. Our simulations confirm that Partitioned Trees exhibit superior performance compared to Red-Black Trees.

Keywords. Binary search trees, AVL-trees, red-black trees, restructuring, partitioning, departitioning.

1 Introduction

Binary search trees (BSTs) are a very popular and efficient structure for storing and retrieving data. However, this is only true if the tree is balanced. An unbalanced BST is no more efficient than a regular linked list. To keep a BST in optimal shape, many balancing algorithms have been proposed over the years. The first and most important are the AVL Trees and the Red-Black trees (RB Trees).

The AVL tree is a self-balanced BST that was invented by Adelson-Velskii and Landis in 1963 [1]. Subsequent to this, Foster and Caxton [12] conducted additional studies on it. AVL tree is simple to implement and best appropriate in lookup operations. However, it contains several maintenance cases involving single and double rotations to the left and right making their use hindered. As a result, much research was made to relax those constraints. For instance, Foster and Caxton [13] gives a generalization of AVL trees which allows unbalances up to a small integer thereby reducing the number of restructuring. Another enhancement to AVL trees was the One-Sided Height-Balanced tree (OSHB), which restricts the height of the node's children such that the right child never has a smaller height than the left one. The insertion and deletion algorithms for OSHB trees are in $O(log^2 n)$ time [17, 19]. Later, more sophisticated algorithms were proposed to achieve optimum performance in $O(log n)$ time for OSHB trees [23].

On the other hand, RB Trees are invented by Rudolf Bayer in 1972 [5] under the name Symmetric binary B-trees and presented as a class of B-trees. B-trees were discovered in turn by Bayer and McCreight [6, 7]. Symmetric binary B-trees were named RB Trees thereafter when Guibas and Sedgewick ([15] proposed a dichromatic framework for balanced trees. RB Trees implement the basic dictionary operations with a worst-case cost of $O(log n)$ per operation, at the cost of storing one extra bit (the color of the node) at each node. They are highly effective in applications with heavy update requirements; but have been criticized for their complexity in both understanding and implementation. To address these issues, various improvements have been proposed to either simplify implementation or improve performance, or both. Examples of simpler RB Trees implementations include Andersson [2] implementation of Bayer [4] binary B-trees and Sedgewick [24] related Left-Leaning

Red-Black trees (LLRB Trees), which simplify rebalancing through asymmetry and eliminate symmetric cases. Andersson [2] further made implementation easier by dividing rebalancing into two procedures (skew and split) and adding additional useful features. Ghiasi-Shirazi et al.[14] introduced the parity-seeking delete algorithm for RB Trees, which is an intuitive and comprehensible algorithm that restores balance to the deficient subtree and its sibling by either fixing the deficient subtree or elevating the deficiency to a higher level. To improve performance, [3, 22] attempted to decrease the maximum height of RB Trees, which is $2log(N + 2) - 2$ in the worst case. Others [10, 21, 20, 8, 18] sought to decouple updates from rebalancing, enabling a greater degree of concurrency and postponed processing. Additionally, Zegour [26] proposed an improvement to the delete algorithm of RB Trees that reduces color changes by roughly 29% and maintenance operations by about 11%. Combining this algorithm with insert and delete operations results in a 4% reduction in running time, while preserving the search performance of the standard algorithm.

The area of designing a balanced tree is abundant and has yet to be fully explored. A recent framework called Rank-Balanced Trees proposed by Haeupler and Tarjan in 2015 [16] allowed to represent AVL trees, RB Trees and its variants, and a novel balanced binary tree known as the weak AVL tree. However, the framework's drawback was its separate rules for defining commonly used balanced trees. This was addressed by Bounif and Zegour [9], which presented a unified representation for both AVL and RB Trees. In addition, Zouana and Zegour [28, 29] introduced the Red Green Black Trees which is an extension to RB Trees. They also proposed a generalized form of RB Trees that offers equivalent performance as RB Trees with a complexity of $O(logn)$, while requiring fewer maintenance operations and enhancing update speed.

Our main purpose in this work is to propose common algorithms for generating RB Trees and a family of partially balanced BSTs. A Partitioned Tree generates two kinds of nodes: Simple nodes and Class nodes. These last nodes form a partition on the tree with heights of either (n-1) or (n-2), 'n' being the parameter of the new structure. Two advantages make Partitioned Tree attractive. First, when 'n' is equal to 2 Partitioned Tree generates a data structure equivalent to RB Trees. When 'n' is larger than 2, the Partitioned Tree generates a family of suitable balanced BSTs. One extra byte of storage is needed to represent both the kind and the height of a node.

It's essential to note that our study focuses on a specific facet within the broader 'Partitioned Binary Search Trees'(P(h)-BST) project [25], distinct from previous research examined in references [9, 28, 29, 27], which explored various aspects of this larger project.

The paper is organized as follows: section 2 describes Partitioned trees and its definition. Section 3 presents the maintenance operations while section 4 and section 5 summarize the insertion and deletion operations. In section 6 we outline some important characteristics of the tree, including the worst-case height analysis. In section 7, we compare the classical algorithm of RB Trees as described by Cormen et al. [11] and the Partitioned Tree with the parameter of 2 . In section 8, we discuss some experimental results. Finally, section 9 concludes and looks forward to future research.

2 Partitioned Trees

Partitioned Tree, abbreviated as PT-n, represents a specialized Binary Search Tree organized into distinct Classes. Each Class functions as a sub-tree with a height of either (n-1) or (n-2), depending on the 'n' parameter. Within these sub-trees, the root node is classified as a Class node, while the remaining nodes are designated as Simple nodes. The PT-n tree maintains perfect balance solely among its Class nodes. Each node, besides containing data, includes a byte called 'control' to indicate its category and height. The node's height corresponds to the depth of the subtree rooted at that node within its respective Class. Using 3 bits to represent height (n = 7) allows classes to be constructed with a maximum height of (n - 1 = 6), accommodating up to $2^6 - 1 = 63$ elements. Formally, these trees can be defined as follows:

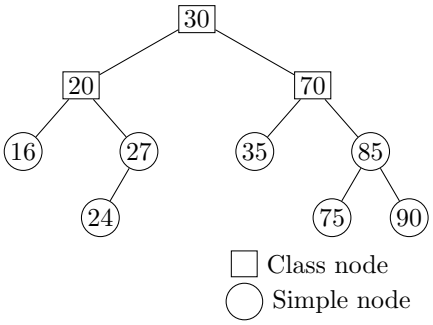
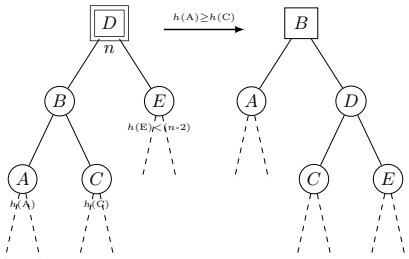
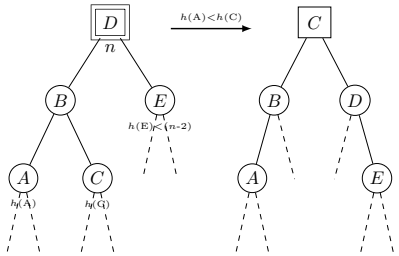


Fig. 1. PT of parameter 3 example



(a) Single rotation.



(b) Double rotation.

Fig. 2. The Restructuring operation. The overflow Class is represented by a double square

1. Nodes are of two kinds, either Simple or Class.
2. The root node is a Class node with a height between 0 and (n-1).
3. Each Root-to-leaf path contains an equal number of Class nodes.
4. Each Class node has a height of either (n-1) or (n-2).

The particular characteristic mentioned in property 4, where Class nodes have a height of either (n-1) or (n-2), contributes significantly to the enhanced balance within this structure compared to the variant discussed by Zouana and Zegour in [29].

In (Fig. 1), we present a visual representation of PT-3 (Partitioned Tree of parameter 3), with nodes organized into Classes as follows: {30} {20, 16, 27} and {70, 35, 85, 75, 90}. In PT-3, Class-30 acts as the root node, having a height of 0, in alignment with Property 2 of PT-n. This property allows the root node to possess a height between 0 and (n - 1). Both Class-20 and Class-70 exhibit heights of 2, signifying (n-1), where n=3, in accordance with Property 4. In illustrations of this paper, we depict Class nodes by square and Simple nodes by circle.

3 Maintenance Algorithms

PT-n aims to represent a large family of trees by tolerating some imbalance in the Classes. The balance of the structure is ensured by the third property: each direct path from the root to the leaf contains the same number of Class nodes. Thus, the maintenance of the structure is based on a set of rotations to distribute Simple nodes within the Class and eventually Partitioning or Departitioning the Class. As a result of reflection, we use three simple operations to maintain the structure's balanced.

3.1 Restructuring

Restructuring operations within PT-n involve single (Fig. 2a) or double rotations (Fig. 2b) centered on the Class node. These operations are specifically employed to reorganize Simple nodes within the Class when the height of the Class node reaches the defined parameter 'n', which is referred to as (Class overflow), thereby indicating a violation of the fourth property, which stipulates that each Class node must have a height equal to (n-1) or (n-2).

Restructuring becomes necessary when one of the child nodes within the Class node has a height less than (n - 2). (Fig. 2) provides visual

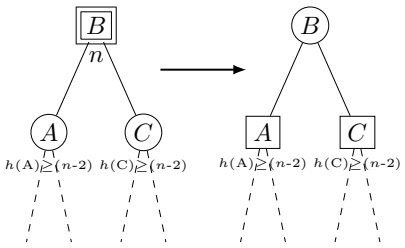


Fig. 3. The Partitioning operation

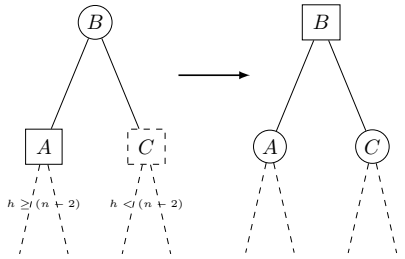


Fig. 4. The Departitioning operation. The underflow Class is represented by a dashed square

representations of the two primary restructuring operations, with analogous cases determined through symmetry.

- Single rotation (Fig. 2a) is used if the height of node A is greater or equal to the height of node C. By applying a right rotation on node D, the height of the Class shall be less than 'n'.
- Double rotation (Fig. 2b) is used when the height of node A is less than the height of node C. By applying a left rotation on node B followed by a right rotation on node D (a double rotation on node D), the height of the Class shall be less than 'n'.

3.2 Partitioning

Partitioning transforms the overflow Class into two Classes. By changing the kind of three nodes, the overflow Class is partitioned into two Classes (Fig. 3). Partitioning is used when both subtrees of the overflow Class are of height larger than or equal to (n-2), where Restructuring can't satisfy the fourth property. Partitioning does not require rotations and is done in $O(1)$.

3.3 Departitioning

Departitioning is the opposite operation of Partitioning, where two Classes are combined into one by switching the kind of three nodes (Fig. 4). It specifically targets Classes with heights lower than (n-2), which is referred to as 'Class underflow' and also constitutes a violation of Property 4.

This operation is typically performed during deletion operations, facilitating the merging of the underflow Class with its adjacent sister Class.

4 Insertion in PT-n

Insertion in PT-n is as simple as any BST insertion except that we do some maintenance operations in order to respect the structure properties. We can summarize the insertion in two steps.

Step 1: Perform a Binary Search to find the key location in the tree. Naturally, this location must be a leaf node within a leaf Class. we make the node of Simple kind. If the height of this leaf Class exceeds the specified parameter 'n' (Class overflow), which also constitutes a violation of Property 4, we proceed to Step 2.

Step 2: To address the imbalance resulting from the height overflow identified in Step 1, we address the issue by considering two cases based on the affected Class:

Case 1: If the overflowed Class has a son with a height less than (n-2). A Restructuring (Fig. 2) is performed.

Case 2: If the sons of the overflowed Class have a height equal to or greater than (n-2). A Partitioning (Fig. 3) is performed. After Partitioning, the mother Class acquires a new simple node and its height increases. This may trigger additional Restructuring and/or Partitioning with the mother Classes further up the tree, creating a cascade effect. To address this, we repeat Step 2 until Partitioning is no longer required.

5 Deletion in PT-n

The delete algorithm in PT-n is easy to implement due to the possibility of merging two Classes in a new one using the 'Departitioning' operation (Fig. 4) and redistributing the keys inside the new Class through a restructuring operation (Fig. 2). Deletion in PT-n can be summarized as follows.

Step 1: Perform a Binary Search to find the key's position in the tree. If it is an internal node, we permutate the key with its substitute leaf. Then, we delete the leaf. If the height of the Class, from which the key was deleted, is less than (n-2), indicating an underflow situation, which also constitutes a violation of Property 4, we proceed with Step 2.

Step 2: To preserve balance among the different portions of the tree following a Class underflow, it is necessary to perform a Departitioning operation (Fig. 4). This process involves three distinct cases:

Case 1: If the Class that underflows has a direct sister Class with a height of (n - 2), we conduct a Departitioning (Fig. 4), followed by an examination of the mother Class to determine if it is underflowing.

Case 2: If the Class that underflows has a direct sister with a height of (n - 1). We perform a Departitioning (Fig. 4) which leads to a Class with a height equal to 'n'. Therefore we must restructure (Fig. 2). If the height of the new Class is less than 'n', we proceed with the deletion process since a node has been removed from the mother Class. If the height of the new Class equals to parameter 'n', we perform Partitioning (Fig. 3) on the resulting Class.

Case 3: If the Class that underflows hasn't a direct sister Class (Fig. 5). The property of partial balance ensures that there is a corresponding sister Class. We need to transform the tree to find the direct sister Class. If the Class underflows on the right (or left) side of the parent, we can identify the sister Class by moving left (or right) from the parent and then selecting the rightmost (or leftmost) Class node. We change the kind of parent node and Simple direct sister node. The process of transformation may be perceived as a single rotation in which only two pointers are modified.

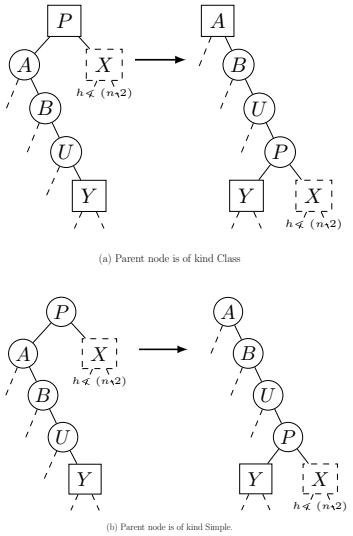


Fig. 5. Transforming operation

6 Analysis on Height

Defining the balance of BST can be challenging, but it is possible to determine the degree of balance/imbalance by establishing precise height intervals. A practical method to achieve this is by identifying the best and worst possible heights that a structure can have for a given set of elements (keys). This establishes a height interval and margin for the distribution of its items. It is well-known that a perfectly balanced tree has a logarithmic height in relation to its number of items. Therefore, it is crucial to analyze the worst-case scenario of the height that the structure can attain to ensure optimal performance.

6.1 Worst Tree Height

In the literature, the tree performance is often given as the worst case height. This height offers a proportional image to the possible tree operations:

Theorem 1. *In a PT-n, the worst case height is of* $\frac{n}{\log_2(n)} \log_2 (\frac{n-1}{n} N + 1)$ *where* N *is the number of keys on the tree.*

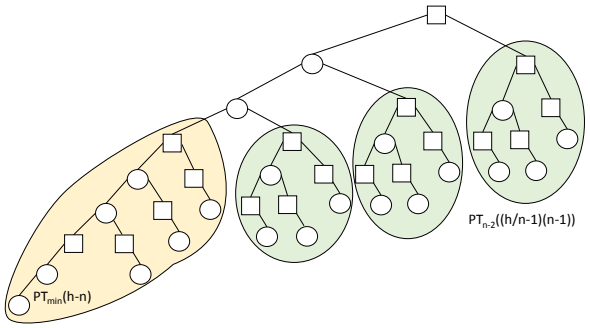


Fig. 6. Worst case tree for PT-3

Proof. Consider an PT-n $PT_{min}(h)$ of minimum number of nodes (Fig. 6) and height h (where h is the maximum number of nodes on any path from root to leaf). Notice that $h = n.k$ where k represents the number of Class nodes along the root-to-leaf path. The tree is produced through two main constraints: every small tree rooted by a Class node on the longest path is a small tree of maximum height and minimum number of nodes defined as a vine of n nodes (Class node included); and each small tree rooted by a Class node on any other path is a vine of $n-1$ nodes (Class node included). The tree is presented as each node of the longest path of the root small tree is linked to the root of a subtree T_{n-2} of height $(\frac{h}{n}-1)(n-1)$, where every Class represents a vine of $n-1$ nodes, except for the last one, which is linked to a subtree $PT_{min}(h-n)$. Let $N_b(T)$ be the number of nodes in the tree T. Then:

$$N_b(PT_{min}(h)) = n \cdot N_b(T_{n-2}((\frac{h}{n}-1)(n-1)) + n + N_b(PT_{min}(h-n)), \quad (1)$$

Since:

$$N_b(T_{n-2}(l)) = n.N_b(T_{n-2}(l-(n-1))+(n-1), \quad (2)$$

$$N_b(T_{n-2}(l)) = n^2 \cdot N_b(T_{n-2}(l-2(n-1)) + n(n-1)+(n-1), \quad (3)$$

$$N_b(T_{n-2}(l)) = n^{\frac{l}{n-1}-1}(n-1) + n^{\frac{l}{n-1}-2}(n-1) + \ldots + n^2(n-1) + n(n-1) + (n-1), \quad (4)$$

$$N_b(T_{n-2}(l)) = (n-1)\sum_{i=0}^{\frac{l}{n-1}-1} n^i = n^{\frac{l}{n-1}} - 1, \quad (5)$$

We obtain:

$$N_b(PT_{min}(h)) = n\cdot(n^{\frac{h}{n}-1}-1)+n+N_b(PT_{min}(h-n)) = n^{\frac{h}{n}} + N_b(T_{min}(h-n)), \quad (6)$$

$$N_b(PT_{min}(h)) = n^{\frac{h}{n}} + n^{\frac{h}{n}-1} + n^{\frac{h}{n}-2} + \ldots + n = n \cdot \sum_{i=0}^{\frac{h}{n}-1} n^i = \frac{n^{\frac{h}{n}}-1}{n-1}. \quad (7)$$

So the number of nodes of the tree N is bound by:

$$n\frac{n^{\frac{h}{n}}-1}{n-1} \leq N_b(PT_{min}(h)) \leq N \leq N_b(T_{bal}(h)) = 2^h - 1. \quad (8)$$

And for a tree of N keys, the worst height of a Partitioned Tree can be found through the following in-equation:

$$n\frac{n^{\frac{h}{n}}-1}{n-1} \leq N, \quad (9)$$

$$n^{\frac{h}{n}} - 1 \leq \frac{n-1}{n}N, \quad (10)$$

$$n^{\frac{h}{n}} \leq \frac{n-1}{n}N + 1, \quad (11)$$

$$\frac{h}{n} \leq log_n(\frac{n-1}{n}N + 1), \quad (12)$$

$$h \leq nlog_n(\frac{n-1}{n}N + 1), \quad (13)$$

$$h \leq n\frac{ln(\frac{n-1}{n}N+1)}{ln(n)}, \quad (14)$$

$$h \leq n\frac{ln(\frac{n-1}{n}N+1)}{ln(n)}\frac{ln(2)}{ln(2)}, \quad (15)$$

$$h \leq \frac{nln(2)}{ln(n)}\frac{ln(\frac{n-1}{n}N+1)}{ln(2)}, \quad (16)$$

$$h \leq \frac{n}{log_2(n)}log_2(\frac{n-1}{n}N + 1), \quad (17)$$

which implies the in-equations:

$$\log_2(N+1) \leq h \leq \frac{n}{\log_2(n)} \log_2\left(\frac{n-1}{n}N+1\right). \tag{18}$$

The height of the PT-n is at most $\frac{n}{\log_2(n)} \log_2\left(\frac{n-1}{n}N+1\right)$ which is a little worse than RB Trees height $2\log_2(N+2)-2$. Notice that for $n=2$ the worst height is that of RB Trees. This is easily provable as follows:

$$\begin{aligned}\frac{2}{\log_2(2)} \log_2\left(\frac{2-1}{2}N+1\right) &= 2\log_2\left(\frac{N+2}{2}\right)\\ &= 2\log_2(N+2) - 2\log_2(2)\\ &= 2\log_2(N+2) - 2.\end{aligned} \tag{19}$$

□

This further proves that PT-2 is equivalent to classic RB Trees.

6.2 Tree Distribution

Tree performance is commonly depicted by the height of the tree as it is seemingly difficult to know the actual distribution of the tree items. We must return to each tree definition to understand and construct an image on its behavior. PT-n are defined, similarly to RB Trees, through the use of two kinds of nodes, where one kind imposes a perfect balance and the other introduces some imbalance. These two properties permit the definition of a growth scheme and can describe the tree performance level. As imbalance is due to the existence of Simple nodes, this comes back to know the number and distribution of this type of nodes and compare them to the total number of nodes and tree height. By summing up these properties, we can give a precise indication on the tree presentation and its performance.

As discussed previously for a PT-n of N items, the worst height (Fig.7) is given by $h = \frac{n}{\log_2(n)} \log_2\left(\frac{n-1}{n}N+1\right)$ with $j = \frac{1}{\log_2(n)} \log_2\left(\frac{n-1}{n}N+1\right)$ Classes on each path. Thus, we know that the number of Class nodes is $2^j - 1$ and consequently, the number of Simple nodes is $N - 2^j + 1$ with a maximum height of $h - j$ Simple nodes. These parameters help in defining the tree presentation. Table 1 summarize and compare the presentation of PT of parameter $n = 2, 3$ with different heights. The tree height evolution is compared between the different parameters in (Fig.7). Notice that the different parameters tend to offer the same access performance with higher number of items as the height difference converges to a fixed amount.

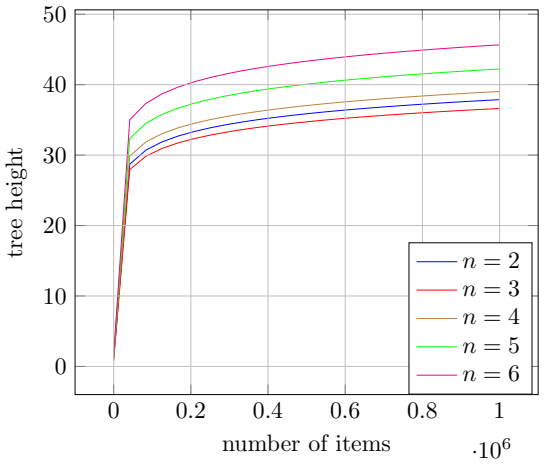


Fig. 7. Maximum PT-n height

Table 1. Comparison between PT-2 and PT-3 distributions

(a) PT-2

Height	Minimum items number N	Class nodes on the longest path j	Class nodes number on each 2^j-1	Simple nodes on the longest path $h-j$	Simple nodes number $N-2^j+1$
6	14	3	7	3	7
9	35	5	31	4	4
12	126	6	63	6	63
15	262	8	255	7	7
18	1022	9	511	9	511
21	2057	11	2047	10	10

(b) PT-3

Height	Minimum items number N	Class nodes on the longest path j	Class nodes number on each 2^j-1	Simple nodes on the longest path $h-j$	Simple nodes number $N-2^j+1$
6	12	2	3	4	9
9	39	3	7	6	32
12	120	4	15	8	105
15	363	5	31	10	332
18	1092	6	63	12	1029
21	3279	7	127	14	3152

The number of Class nodes on each path decreases with each tree parameter and decreasing the number of needed restructuring and maintenance. These two properties insure that PT-n have better performance with update operations.

Furthermore, (Fig.7) illustrates that when parameter $n = 3$, the tree produced by the PT-n structure exhibits a maximum tree height less then the maximum tree height of the RB Trees. This serves as an additional benefit of the PT-n structure over the RB Trees.

7 Comparing RB Trees and PT-2

This section aims to draw a comparison between RB Trees as described by Cormen et al. [11] and PT-2, emphasizing their similarities and differences. The key areas of comparison include the structures' definitions, as well as the fixing-up rules that follow insertion and deletion algorithms.

7.1 Comparing Definitions

An RB Tree is a BST that has the following properties:

1. Each node is either Black or Red.
2. The root node is Black.
3. Each Root-to-leaf path contains an equal number of Black nodes.
4. Each Red node must have Black children.

The definition of PT-2 is just an interpretation of RB Trees definition by replacing Class nodes by Black nodes and Simple nodes by Red nodes.

PT-2 is defined by:

a) Each node is either a Simple or a Class.

b) The root node is a Class.

c) Each Root-to-leaf path contains an equal number of Class nodes.

d) Each Class has a height of either 0 or 1.

The three properties (1,a), (2,b), and (3,c) are identical in both definitions. The only property that is not clear is the fourth property. It's worth mentioning that a Class with a height of either 0 or 1 is a Class containing at most, one level of Simple nodes. These Simple nodes have either Class nodes or external nodes (nil pointers) as their children. Under the assumption that Black nodes are present, these Simple nodes will only have Black nodes as children. Also, based on the requirement that Red nodes must have Black children, we can deduce that the maximum level difference between consecutive Black nodes in an RB Tree is 1. As a result, the two definitions are equivalent, making the RB Tree a special case of PT-n when $n = 2$.

7.2 Fixing Rules After Insertion

The insert algorithm in RB Trees has two steps: inserting a new Red node using the rules of BSTs, then fixing any property violations with fixup operations. Only the 4th property of RB Trees can be violated by a new Red node insertion. If the tree is empty, the 2nd property is violated and fixed by changing the root node color to Black.

If a child and parent node are both Red and the parent is a left child, the tree is fixed as follows:

a) if the parent's sibling is Red, turn the parent and sibling Black and grandparent Red. Checking for two consecutive Red nodes is continued from the grandparent node (Fig. 8(a)) and (Fig. 8(b)).

b) if the parent's sibling is Black and the current node is its right child, a left rotation is carried out on the parent node as illustrated in (Fig. 8(c)). This prepares the situation for the implementation of the following rule.

c) if the parent's sibling is Black, and the current node is its left child, a rotation to the right is performed on the grandparent node, as illustrated in (Fig. 8(d)).

The rules for the scenario where the parent node is a right child can be derived by interchanging the terms left and right in the above statements.

Regarding RB Trees, the PT-2 insertion algorithm operates in two stages. First, the new data is added according to the rules of BST's in a new Simple node. Then, if any property of PT-2 is violated, the tree is fixed with appropriate fixup operations. The 3rd property could not be violated as the newly inserted node is of kind Simple. If the insertion is applied to an empty tree, then the 2nd property is violated, which is simply fixed by changing the kind of the root node to Class. The only potential problem is the violation of the 4th property, i.e. the Class node overflows (it has height equal to parameter 2).

Assuming that the Class node overflows and that its left child has a height equal to 1, the tree is fixed using the following rules:

a) If the right son of the overflowed Class is of kind Simple, a Partitioning is applied (Fig. 9(a)) and (Fig. 9(b)). Checking for overflowed Class is continued from the mother Class.

b) If the right son of the overflowed Class is of kind Class, and its left son has a right child then a left rotation is performed on the left son of the overflowed Class (Fig. 9(c)). The situation is now prepared for the application of the next rule.

c) If the right son of the overflowed Class is of kind Class, and its left son has a left child then a right rotation is performed on the overflowed Class (Fig. 9(d)).

The rules for the case that the Class node overflows, and that its right child has a height equal to 1, are obtained by exchanging "left" and "right" in the above statements.

As demonstrated in both (Fig. 8) and (Fig. 9), it is evident that both the RB Tree and PT-2 use the same rules for fixing-up the tree after inserting a new node.

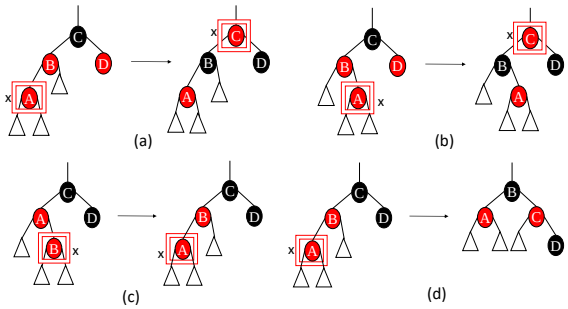


Fig. 8. Procedures for fixing-up RB Trees after inserting a new node. the Procedures are performed recursively. Only half of the procedures, specifically for when the parent node is the left child, are presented here. The remaining four procedures can be derived through symmetry. Subtrees are represented by a triangle symbol, while the node under examination is designated by the letter 'x' and a double square. Note that the implementation of RB trees considers only three cases, and rules (a) and (b) are simultaneously handled

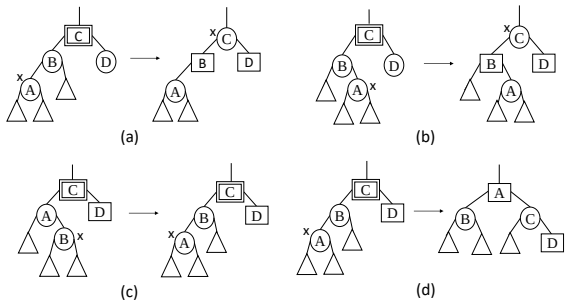


Fig. 9. Rules for fixing-up PT-2 following insertion of a new node. The rules are applied recursively. Each rule has a dual, obtained through symmetry, that is not shown. Subtrees are represented by a triangle. The node responsible for the overflows Class is denoted by the letter 'x'

7.3 Fixing rules after deletion

The delete operation in a RB Tree can take place at any node, including the root node, an internal node, or a leaf node. Initially, if the node to be deleted has two children, its value is replaced by either the maximum value in the left subtree or the minimum value in the right subtree, leading to the deletion of a node with one child (degree-1 node)

ISSN 2007-9737

or a leaf node. The actual deletion is then executed according to the following rules:

a) Deletion of a degree-1 node: Since a degree-1 node have only one child, the presence of a Black node in their subtree is prohibited. Additionally, considering that a node and its child cannot both be Red, it follows that a degree-1 node to be a Black node with a single Red child. In such case, the value of the Red child node is copied onto the degree-1 node, and the Red child node is deleted.

b) Deletion of a Red leaf node: The node is simply deleted, resulting in a tree that maintains the characteristics of RB Trees.

c) Deletion of a Black leaf node: Deleting a Black leaf node would result in a violation of the 3rd property of RB Trees, as the number of Black nodes in the left and right subtrees of its parent, would not be equal. In such a scenario, the fix-up operations continue until at least one of the rules in (Fig. 10) is applicable.

Regarding RB Trees, the deletion process in PT-2 can occur at the root, an internal node, or a leaf node. If the node being deleted has two children, its value is substituted with either the largest value in its left subtree or the smallest value in its right subtree. This transforms the deletion into a process involving either a leaf Class node with one Simple child or a leaf node. Then, the actual deletion is carried out according to the following rules:

a) Deleting a leaf Class with one Simple node: The leaf Class does not have a child on one side, preventing the existence of a Class node further down the subtree. The value of its Simple node child is copied to the leaf Class, and the Simple child node is deleted.

b) Deleting a Simple leaf node: The node is simply removed.

c) Deleting a Class leaf node: Deleting a Class leaf node would result in a violation of the 3rd property of ST-2, as the number of Class nodes in the left and right subtrees of its parent, would not be equal. In such instances, fix-up operations must be continued until at least one of the rules of (Fig. 11) becomes applicable.

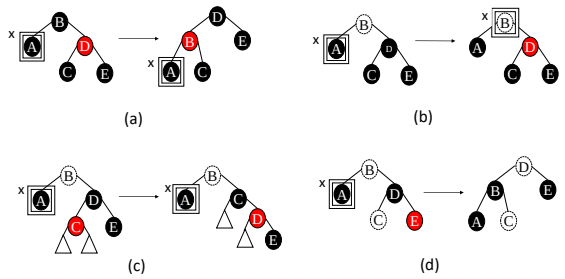


Fig. 10. Rules for fixing-up RB Trees after removing a Black leaf node. The steps are executed recursively. The root of the subtree with a reduced number of Black nodes is represented by a square and is referred to as 'x'. A node represented by a dashed circle shape can be either a Red node or a Black node. Each rule has a counterpart, which is derived through symmetry and not shown. Rule (a) prepares x's, Black sibling. Rule (b) is applied when the sibling and both of its children are Black, causing the deficiency to be passed up to the parent node. Rule (c) is applicable if the sibling and its right child are both Black, while the left child of the sibling is Red. In such cases, a right rotation is performed on node (D), and the right child of the resulting sibling is set to Red, thus setting up the context for the subsequent rule. Rule (d) is applicable in cases where the sibling node is of Black color and its right child node is of Red color. After executing rule (d), the deficiency is eliminated, and the algorithm ends

At first glance, it appears that both the RB Tree and PT-2 have the same rules for fixing up after deletion. However, the two trees handle Case (d) differently. (Fig. 12) illustrates in detail how both the RB Tree and PT-2 treat the Case (d).

We can already state that the PT-2 generates a tree that respects the properties of the RB Tree after a deletion operation, but it differs in terms of the number of Simple (Red) and Class (Black) nodes. This difference will impact the performance of the two trees, as demonstrated in the following section.

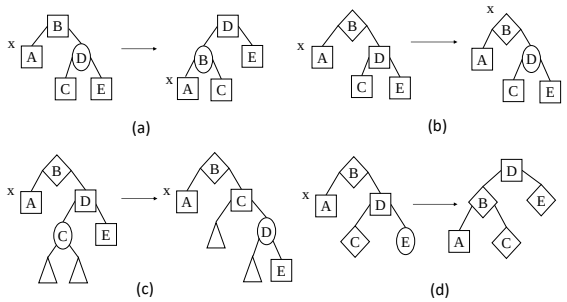


Fig. 11. Rules for fixing-up PT-2 after deleting a Class node are applied recursively. A node represented as a rhombic shape can be either a Class node or a Simple node. The root of the subtree with a reduced number of Class nodes is referred to as 'x'. Each rule has a dual, obtained through symmetry, that is not shown. Rule (a) transforms the tree to find a direct sister Class for the underflowed node. Rule (b) is applied when the direct sister Class has a height equal to zero, elevating the deficiency to the parent Class node. When the direct sister Class has a height of one, Rules (c) and (d) are applied. Rule (c) is applied if the left child of the direct sister is a Simple node. In this case, a Departitioning operation of Class (B) is performed followed by a right rotation on the node (D) and a Partitioning operation of node (B) to set up the context for the subsequent rule. Rule (d) is applied if the right child of the direct sister is a Simple node. In this case, a Departitioning operation of Class (A) is performed followed by a left rotation on the node (B) and potentially a Partitioning operation on the node (D). The algorithm ends after executing rule (d), having eliminated the deficiency

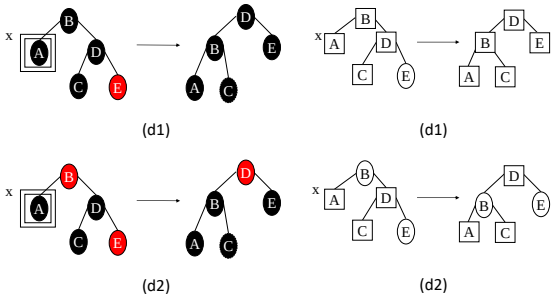


Fig. 12. Rules for fixing-up sub-cases of (d) by RB Tree and PT-2. The primary distinction is based on the manner in which the two trees alter the (Color/Kind) of the nodes after fixing-up. Specifically, in RB Tree, the root node is either Red or Black, and its children are always Black. In contrast, the root node of PT-2 tree is always of kind Class, and its children can be both either Simple or Class nodes. As a result, the number of Simple nodes in PT-2 is typically greater than the number of Red nodes in RB Tree

8 Experimental Tests

The performance evaluation of the Partitioned Trees (PT-n) were conducted in three stages. The first and the second stages consisted of evaluating the performance of the insertion and deletion algorithms of both PT-n and classical RB Trees. The last stage is reserved for comparing the performance of the two Trees under different frequency of (Insertion/Deletion) operations.

The performance metrics have been deliberately selected to remain independent of both algorithmic implementations and the measurement platform. These metrics include Average Search Height, Tree Height, and the total number of rotations. The Average Search Height is particularly significant as it reflects the efficiency of search, insertion, and deletion operations. A lower average height indicates quicker and more consistent performance, benefiting all tree-related tasks.

The simulation environment and the trees algorithms were implemented using the C programming language in Microsoft Visual Studio Community 17.6.5. RB Tree was implemented by adopting the method outlined in [11], featuring a nil node. Subsequently, Partitioned Trees with a nil node were also implemented. To ensure equitable comparison between the two structures, both implementations employ a stack for traversing trees. All experiments were conducted on an HP G62 notebook PC with an Intel Core i3-370 CPU running at 2.40GHz, and equipped with 8 GB of memory, operating on a 64-bit Windows 7 system.

The key sequences were generated in order to obtain more complete insight into PT-n and RB Trees performance. The sizes of the generated number sequences, denoted as N, vary within the set: $N \in \{100K, 200K, 300K, 400K, 500K, 1M\}$. This variation in sequence sizes was performed in order to establish how performance depends on the number of keys.

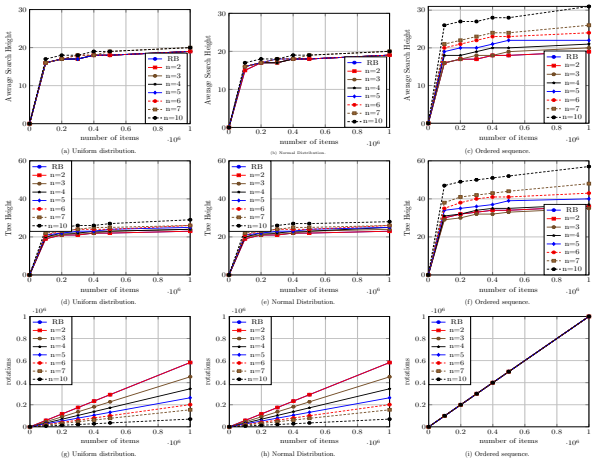


Fig. 13. Comparison of Insertion Algorithms in PT-n and RB Trees

Table 2. Comparison of Node Counts by Kind/Color After Insertions for PT-n and RB Trees using a Uniform Distribution

	Black/Class				Red/Simple			
N	RB	PT-2	PT-3	PT-10	RB	PT-2	PT-3	PT-10
100K	51,371	51,371	26,783	2,117	48,629	48,629	73,218	97,882
200K	102,642	102,642	53,555	4,232	97,358	97,358	146,445	195,768
300K	154,057	154,057	80,293	6,328	145,943	145,943	219,707	293,673
400K	205,270	205,270	107,072	8,447	194,730	194,730	292,929	391,553
500K	256,736	256,736	133,899	10,553	243,264	243,264	366,101	489,447
1M	513,454	513,454	267,711	21,107	486,546	486,546	732,289	978,893

Three groups of the key sequences have been used here, and they differ according to the way of key generation.

In the first group, random integer keys are generated using the C++ Standard Library's `'std::mt19937'` class, an implementation of the Mersenne Twister algorithm. This class is specialized for creating integer keys distributed uniformly across the integer range. To ensure distinct sequences of random numbers, `'std::mt19937'` employs various seed values during initialization, which are obtained from a random number generator device accessible through `'std::random_device'`. This approach guarantees that the generated keys remain within the integer value range, avoiding sequence repetition and eliminating the need for predefined numerical constraints.

In the second group, double-type keys are generated using the `'std::normal_distribution'` class from the C++ library, following a Gaussian or normal distribution with a mean of 0.0 and a standard deviation of 1.0. This results in keys clustering around 0.0, as expected for a Gaussian distribution.

In the third group, keys are generated from an ordered ascending sequence of integers, representing the worst-case scenario for conventional BSTs due to the creation of a degenerate tree. This experiment enhances our comprehensive understanding of the PT-n capacity to mitigate degenerate tree formation, ultimately augmenting the operational efficacy of key insertion and deletion procedures.

In the upcoming subsections, we will look into the description and analysis of the results obtained at each evaluation stage of the PT-n.

8.1 Comparing Insertion Algorithms for PT-n and RB Tree

We considered the following experiment:

1. Construct PT-n with parameters $n = \{2, 3, 4, 5, 6, 7, 10\}$, and create a Red-Black Tree using the same sequence (S) of N values. This sequence is initially generated using a random uniform distribution method, the Mersenne Twister, followed by generation in accordance with a normal distribution, and finally in an ascending ordered sequence. Then, compute:
   (a) The Average Search Height of both trees.
   (b) The height of both trees.
   (c) The total number of rotations.
   (d) The total number of nodes of each kind composing the trees.
2. Repeat ten times step (1) for N ={100K, 200K, 300K, 400K, 500K, 1M}.

Figure 13 illustrates the performance metrics of the insertion algorithm for PT-n and RB trees. The left column presents the outcomes for key sequences generated under a uniform distribution, while the middle column presents the outcomes of key sequences generated following a normal distribution.The right column presents

Table 3. Comparison of Node Counts by Kind/Color After Insertions for PT-n and RB Trees using a Normal Distribution

	Black/Class				Red/Simple			
N	RB	PT-2	PT-3	PT-10	RB	PT-2	PT-3	PT-10
100K	51,361	51,361	26,762	2,116	48,639	48,639	73,238	97,884
200K	102,767	102,767	53,563	4,223	97,233	97,233	146,437	195,777
300K	154,012	154,012	80,355	6,332	145,988	145,988	219,645	293,669
400K	205,414	205,414	107,140	8,429	194,586	194,586	292,860	391,570
500K	256,781	256,781	133,891	10,553	243,219	243,219	366,109	489,447
1M	513,702	513,702	267,714	21,116	486,546	486,546	732,286	978,884

Table 4. Comparison of Node Counts by Kind/Color After Insertions for PT-n and RB Trees using an Ordered Sequence

	Black/Class				Red/Simple			
N	RB	PT-2	PT-3	PT-10	RB	PT-2	PT-3	PT-10
100K	99,980	99,980	49,990	11,110	20	20	50,010	88,890
200k	199,980	199,980	99,990	22,220	21	21	100,010	177,780
300k	299,970	299,970	149,990	33,330	25	25	150,010	266,670
400k	399,980	399,980	199,990	44,440	22	22	200,010	355,560
500k	499,980	499,980	249,990	55,550	23	23	250,010	444,450
1M	999,980	999,980	499,990	111,110	24	24	500,010	888,890

the outcomes for key sequences generated following an ordered sequence. Tables 2, 3, and 4, respectively, depict the counts of nodes categorized by type or color after insertions for PT-n and RB Trees under uniform distribution, normal distribution, and ordered sequence.

It is clear from Figure 13 that both the RB Trees generated by the new PT-n structure with $n = 2$ and the standard RB Tree exhibit the same performance (average search height, tree heights, and total number of rotations). Additionally, Tables 2, 3, and 4 show that they also have the same counts of Color/Kind of nodes. Based on this data, we can attest that the insertion algorithms of both the RB Tree and PT-n with $n = 2$ are equivalent

When comparing PT-n for $n = \{3, 4, 5, 6, 7, 10\}$ to the RB tree with a key sequence generated under a uniform or a normal distribution, it becomes evident that there is no significant difference in terms of average search height, and the tree heights increases by 1 to 5 units for the PT-n tree, with the maximum increase of 5 units occurring when the parameter "n" is set to 10. Simultaneously, there is a decrease in the number of rotations, attributed to the fact that as the height of Class nodes increases, the number of Simple nodes within Class nodes also increases, resulting in fewer rotation operations needed to balance the tree.

However, in the case where keys are generated following an ordered sequence, one observes higher average search height and tree heights for PT-n with $n = \{5, 6, 7, 10\}$ compared to those of the RB tree, with only a slight reduction in the count of rotations. This reduction can reach up to 17 rotations less in favor of PT-n when $n = 10$, compared to the RB tree. This phenomenon arises from the absence of a mechanism for balancing Simple nodes within Class nodes. Since each Class node has a height equal to (n-1) or (n-2), the tree's height becomes the sum of the heights of the Class nodes in the longest branch.

8.2 Comparing Deletion Algorithms for PT-n and RB Tree

We considered the following experiment:

1. Construct PT-n with parameters $n = \{2, 3, 4, 5, 6, 7, 10\}$, and create a Red-Black tree using the same sequence (S1) of N values. This sequence is generated first using the random uniform distribution method, specifically the Mersenne Twister, followed by generation in accordance with a normal distribution, and finally in an ascending ordered sequence.
2. Generate a random sequence (S2) of approximately $N/2$ removal operations. Perform the following computations:
   (a) The Average Search Height of both trees.
   (b) The height of both trees.
   (c) The total number of rotations performed during the operations.
   (d) Count the total number of nodes of each type in the trees.
3. Repeat steps (1) and (2) ten times for each N value: $N = \{100K, 200K, 300K, 400K, 500K, 1M\}$.

Based on Figure 14, the performance of both RB trees, created using the new PT-n structure (with $n = 2$), and the standard RB tree, exhibits similarity in terms of average search height and tree heights. However, it's important to note that PT-2 requires

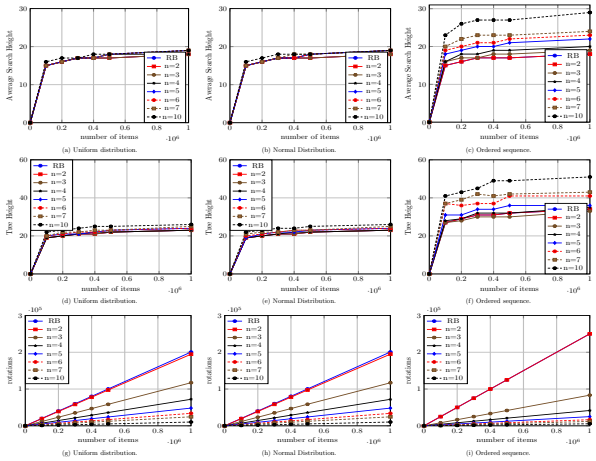


Fig. 14. Comparison of Deletion Algorithms in PT-n and RB Trees

Table 5. Comparison of Node Counts by Kind/Color After Deletions for PT-n and RB Trees using a Uniform Distribution

	Black/Class				Red/Simple			
N	RB	PT-2	PT-3	PT-10	RB	PT-2	PT-3	PT-10
100K	35,426	33,708	15,623	876	14,574	16,292	34,377	49,124
200K	70,917	67,441	31,258	1,748	29,083	32,559	68,742	98,252
300K	106,268	101,125	46,963	2,626	43,732	48,875	103,037	147,374
400K	141,704	134,779	62,589	3,512	58,296	65,221	137,411	196,488
500K	177,215	168,528	78,273	4,371	72,785	81,472	171,728	245,629
1M	354,219	336,885	156,473	8,702	145,782	163,115	343,527	491,298

Table 6. Comparison of Node Counts by Kind/Color After Deletions for PT-n and RB Trees using a Normal Distribution

	Black/Class				Red/Simple			
N	RB	PT-2	PT-3	PT-10	RB	PT-2	PT-3	PT-10
100K	35,392	33,683	15,649	871	14,609	16,317	34,352	49,129
200K	70,791	67,346	31,288	1,746	29,210	32,655	68,713	98,255
300K	106,290	101,098	46,935	2,625	43,711	48,903	103,066	147,376
400K	141,741	134,788	62,635	3,491	58,260	65,213	137,366	196,510
500K	177,174	168,538	78,244	4,368	72,827	81,463	171,757	245,633
1M	354,356	336,920	156,424	8,736	145,645	163,081	343,577	491,265

Table 7. Comparison of Node Counts by Kind/Color After Deletions for PT-n and RB Trees using an Ordered Sequence

	Black/Class				Red/Simple			
N	RB	PT-2	PT-3	PT-10	RB	PT-2	PT-3	PT-10
100K	49,976	49,976	24,988	5,528	24	24	25,012	44,472
200K	99,975	99,975	49,982	11,080	25	25	50,018	88,920
300K	149,972	149,972	74,988	16,635	28	28	75,012	133,365
400K	199,974	199,974	99,984	22,190	26	26	100,016	177,810
500K	249,971	249,971	124,985	27,745	29	29	125,015	222,255
1M	499,970	499,970	249,985	55,520	30	30	250,015	444,480

fewer rotation operations compared to the RB tree when keys are generated either uniformly or according to a normal distribution. This difference is due to PT-2 producing more Simple nodes and fewer Class nodes after deletion, as shown in Table 5 and Table 6 , because of the "d2" sub-rule in Figure 12. Consequently, after deleting nodes, PT-2 needs fewer restructuring operations since it doesn't have to fix the tree structure after removing Simple nodes. When keys are generated in order, both PT-2 and the RB tree have the same total rotations. This is because PT-2 applies sub-rule "d1" from Figure 12, which mirrors the RB tree's mechanism. This leads to both trees having identical node kind/color counts after deletions, as highlighted in Table 7.

When comparing PT-n, where $n = \{3, 4, 5, 6, 7, 10\}$, to the RB tree, it is evident that both tree structures exhibit similar behaviors when subjected to key sequences generated from either a uniform or normal distribution, as well as an ordered sequence. Specifically, we observe a noticeable reduction in the total number of rotations in favor of PT-n as the parameter "n" increases in comparison to the RB tree. This phenomenon can be attributed to the fact that with the increasing value of parameter "n" , the count of Simple nodes positioned beneath Class nodes also increases, as demonstrated in tables 5,6 and 7. Consequently, this rise in the number of Simple nodes leads to a reduced necessity for rotation operations, as the removal of Simple nodes no longer requires rotation to maintain the tree's balance.

Furthermore, as discussed in the 'Comparing insertion algorithm for PT-n and RB tree' subsection, we also observe a related phenomenon when keys are ordered. In this situation, PT-n with $n = \{5, 6, 7, 10\}$ display higher average search height and tree height compared to RB trees. This is due to the absence of a mechanism to balance Simple nodes within Class nodes in PT-n.

8.3 Comparing Performance of PT-n and RB Tree under Different Workloads

We considered the following experiment:

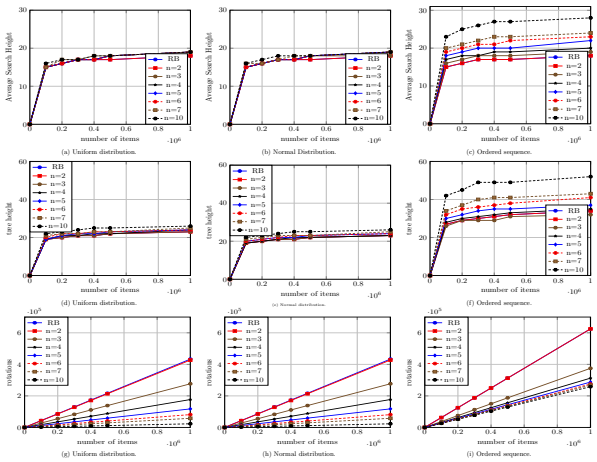


Fig. 15. Comparison of RB and PT-n Trees performance with 25% Insertions and 75% Deletions

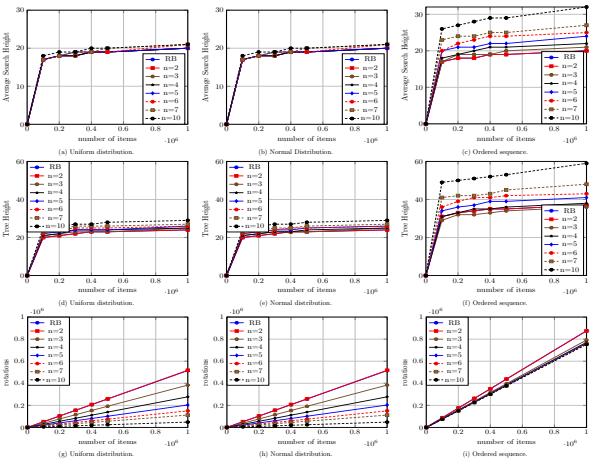


Fig. 16. Comparison of PT-n and RB Trees performance with 75% Insertions and 25% Deletions

1. Construct PT-n with parameters $n = \{2, 3, 4, 5, 6, 7, 10\}$ and create a Red-Black Tree using the same sequence (S1) of N values. This sequence is initially generated using a random uniform distribution method, the Mersenne Twister, followed by generation in accordance with a normal distribution, and finally in an ascending ordered sequence.
2. Generate a new sequence (S2) containing N values, following the same key generation process as the first sequence (S1). This sequence is designed to have varying workloads: (25% Insertion, 75% Suppression), and (75% Insertion, 25% Suppression). Then, compute :
   (a) The Average Search Height of both trees.
   (b) The height of both trees.
   (c) The total number of rotations performed during the operations.
3. Repeat step (1) and (2) ten times for each N value: $N = \{100K, 200K, 300K, 400K, 500K, 1M\}$.

Based on the data in Figures 15 and 16, it's evident that PT-2 and RB Tree share similar average search height and tree heights characteristics across diverse workloads. Nevertheless, they diverge significantly in terms of the total number of required rotations.

Specifically, when keys follow a uniform or normal distribution, PT-2 require fewer rotations than RB Tree with a 25% insertions and 75% deletions workload. Conversely, RB Tree needs fewer rotations than PT-2 when workload consist of 75% insertions and 25% deletions. This difference arises because PT-2 generate more 'Simple nodes' during deletions compared to RB Tree, which generate fewer 'Red nodes', as explained in the subsection titled "Comparing Deletion Algorithms for PT-n and RB tree." Consequently, a high volume of insertions require more rotation operations to balance PT-2 compared to RB Tree.

However, when keys are generated sequentially, PT-2 and RB Tree exhibit identical total numbers of rotations across various workloads. This phenomenon can be attributed to both tree types adhering to the same rules after insertion and deletion operations, as discussed in the subsections "Comparing Insertion Algorithms for PT-n and RB tree" and "Comparing Deletion Algorithms for PT-n and RB tree".

When comparing PT-3 and RB Tree, we observe similar average search height across varying workloads and key sequence generation methods, including uniform distribution, normal distribution, and ordered sequences. However, they diverge in terms of tree heights and total rotations. Specifically, PT-3 and RB Tree exhibit the same

height when keys are generated with a uniform or normal distribution. Conversely, when keys are generated in an ordered sequence, PT-3 exhibits a lower height by about one unit compared to RB Tree. Additionally, PT-3 requires fewer total rotations compared to RB Tree for various key generation methods and across varying workloads. This phenomenon stems from the increase of the "n" parameter in PT by 1 or 2 units, resulting in a more relaxed Class height and reduced rotation operations during updates, without a substantial increase in tree height.

For higher values of the parameter "n" in PT-n, such as $n = \{5, 6, 7, 10\}$, an increase in both the average search height and tree height is observed compared to the RB Tree, particularly when keys are generated in sequential order. This disparity arises from the lack of a mechanism to balance Simple nodes within Class nodes in PT-n.

9 Conclusion and Future Work

PT-n is interesting for its simple and easy-to-comprehend insertion and deletion algorithms. Empirical results indicate that the structure outperforms traditional RB Trees, requiring fewer restructuring operations. Notably, when parameter 'n' is set to three, the maximum height of the PT-3 is less then that of the RB Trees. Another key feature of the PT-n data structure is its flexibility, which stems from the utilization of the 'n' parameter. In practice, larger values of 'n' lead to the generation of less balanced trees with reduced maintenance requirements, while smaller 'n' values result in more balanced trees with increased maintenance needs. This feature makes PT-n particularly suitable for use in environments with expensive maintenance, such as schedulers. Nevertheless, the current implementation of the data structure requires the storage of 8 bits to consider both the height and the Kind of nodes, which may negatively impact storage efficiency. A potential solution could be to develop a method for calculating the height of the nodes without having to store it. Consequently, minimizing the storage demand to a single bit per node, freeing up 7 bits of storage per node.

References

1. **Adelson-Velskii, M., Landis, E. (1963).** An algorithm for the organization of information. Dokl.Akad. Nauk SSSR 146, Vol. 3, pp. 1259–1262.
2. **Andersson, A. (1993).** Balanced search trees made simple. In Proceedings of the WADS Conference (WADS'93), pp. 60–71.
3. **Andersson, A., Icking, C., Klein, R., Ottmann, T. (1990).** Binary search trees of almost optimal height. Acta informatica, Vol. 28, pp. 165–178.
4. **Bayer, R. (1971).** Binary b-trees for virtual memory. In Proceedings of the 1971 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control, pp. 219–235.
5. **Bayer, R. (1972).** Symmetric binary b-trees: Data structure and maintenance algorithms. Acta informatica, Vol. 1, pp. 290–306.
6. **Bayer, R., McCreight (1970).** Organization and maintenance of large ordered indices. In Proceedings of the 1970 ACM SIGFIDET (now SIGMOD) Workshop on Data Description, Access and Control, pp. 107–141.
7. **Bayer, R., McCreight (1972).** Organization and maintenance of large ordered indexes. Acta informatica, Vol. 1, pp. 173–189.
8. **Besa, J., Eterovic, Y. (2013).** A concurrent red-black tree. Journal of Parallel and Distributed Computing, Vol. 73, pp. 434–449.
9. **Bounif, L., Zegour, D. E. (2019).** Toward a unique representation for AVL and red-black trees. Computación y Sistemas, Vol. 23, No. 2.
10. **Boyar, J., Larsen, K. S. (1994).** Efficient rebalancing of chromatic search trees. Journal of Computer and System Sciences, Vol. 49, pp. 667–682.
11. **Cormen, T. H., Charles E. Leiserson, R. L. R., Stein, C. (2009).** Introduction to algorithms. MIT press, 3 edition.

12. **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.
13. **Foster, C. C. (1973).** A generalization of avl trees. Communications of the ACM, Vol. 16, No. 8, pp. 513–517. DOI: 0.1145/355609.362340.
14. **Ghiasi-Shirazi, K., Ghandi, T., Taghizadeh, A., Rahimi-Baigi, A. (2020).** Revisiting 2-3 red-black trees with a pedagogically sound yet efficient deletion algorithm: The parity-seeking delete algorithm. https://arxiv.org/abs/2004.04344. DOI: 10.48550/ARXIV.2004.04344.
15. **Guibas, L. J., Sedgewick, R. (1978).** A dichromatic framework for balanced trees. In 19th Annual Symposium on Foundations of Computer Science (sfcs 1978), IEEE, pp. 8–21.
16. **Haeupler, S., B. Siddhartha, Tarjan, R. (2015).** Rank-balanced trees.. ACM Transactions on Algorithms (TALG), Vol. 11, No. 4, pp. 30:1–30:26.
17. **Hirschberg, Daniel, S. (1976).** An insertion technique for one-sided height-balanced trees. Communications of the ACM, Vol. 19, No. 8, pp. 471–473.
18. **Howard, P. W., Walpole, J. (2014).** Relativistic red-black trees. Concurrency and Computation: Practice and Experience, Vol. 26, pp. 2684–2712.
19. **Kosaraju, S., Rao. (1978).** Insertions and deletions in one-sided height-balanced trees. Communications of the ACM, Vol. 21, No. 3, pp. 226–227.
20. **Larsen, K. S. (2002).** Relaxed red-black trees with group updates. Acta informatica, Vol. 38, pp. 565–586.
21. **Park, H., Park, K. (2001).** Parallel algorithms for red-black trees. Theoretical Computer Science, Vol. 262, pp. 415–435.
22. **Roura, S. (2013).** Fibonacci bsts: A new balancing method for binary search trees.. Theoretical Computer Science, Vol. 482, pp. 48–59.
23. **Räihä, Kari-Jouko, Zweben, S. H. (1979).** An optimal insertion algorithm for one-sided height-balanced binary search trees. Communications of the ACM, Vol. 22, No. 9, pp. 508–512.
24. **Sedgewick, R. (2008).** Left-leaning red-black trees. Dagstuhl Workshop on Data Structures, pp. 17.
25. **Zegour, D. (2022).** Partitioned binary search trees (p (h)-bst): A data structure for computer ram. Data Science with Semantic Technologies: Theory, Practice, and Application, pp. 139–177.
26. **Zegour, D. E. (2022).** Improving the red-black tree delete algorithm. https://doi.org/10.21203/rs.3.rs-1194654/v3.
27. **Zegour, D. E. (2023).** M-PBBST(n1, n2): Multiple partially balanced binary search trees in one. https://doi.org/10.21203/rs.3.rs-2857732/v1.
28. **Zouana, S., Zegour, D. E. (2018).** Red green black trees: Extension to red black trees.. J. Comput., Vol. 13, No. 4, pp. 461–470.
29. **Zouana, S., Zegour, D. E. (2019).** Partitioned binary search trees: a generalization of red black trees. Computación y Sistemas, Vol. 23, No. 4, pp. 1375–1391.

Article received on 14/02/2024; accepted on 14/02/2025.
**Corresponding author is Fahd Mustapha Meguellati.*