International Journal of Innovative Research in Computer Science & Technology (IJIRCST)
ISSN: 2347-5552, Volume-6, Issue-6, November2018
DOI: 10.21276/ijircst.2018.6.6.2

Amortized Complexity Analysis for Red-Black Trees and Splay Trees

Isha Ashish Bahendwar, RuchitPurshottam Bhardwaj, Prof. S.G. Mundada

Abstract—**The basic conception behind the given problem definition is to discuss the working, operations and complexity analyses of some advanced data structures. The Data structures that we have discussed further are Red-Black trees and Splay trees.Red-Black trees are self-balancing trees having the properties of conventional tree data structures along with an added property of color of the node which can be either red or black. This inclusion of the property of color as a single bit property ensured maintenance of balance in the tree during operations such as insertion or deletion of nodes in Red-Black trees. Splay trees, on the other hand, reduce the complexity of operations such as insertion and deletion in trees by *splaying*or making the node as the root node thereby reducing the time complexity of insertion and deletions of a node. Furthermore, amortized analysis, which emerged from aggregate analysis, is an optimistic approach that can be used to calculate the amount of time and space required for the execution of various operations. Amortized analysis considers the number of operations required during the execution of an algorithm rather than the number of inputs required thus overlooking the worst case run time per operation, which can be too pessimistic.**

Keywords- Red-Black trees, Splay trees, Ammortization, Complexity, Insertion, Deletion

I. INTRODUCTION

Data structures refer to the schemes which are used widely to store data in computer’s memory in such a way that performing various operations on it becomes easier and efficient in terms of cost and time. There are various data structures in use such as stacks, queues, linked lists, trees, and many more. To make storage and accessibility of data better than basic data structures, Advanced data structures are used. Advanced Data Structures are improvement over Basic data structures reducing the complexities of operations on data to logarithmic complexities.

Manuscript Received November 09, 2018.

Isha Ashish Bahendwar, Computer Science and Engineering, Shri Ramdeobaba College of Engineering and Management, Nagpur, Maharashtra, Mobile No. 8390562363, bahendwaria@rknec.edu
RuchitPurshottam Bhardwaj, Computer Science and Engineering, Shri Ramdeobaba College of Engineering and Management, Nagpur, Maharashtra, Mobile No.9766893526
Prof. S. G. Mundada, Computer Science and Engineering, Shri Ramdeobaba College of Engineering and Management, Nagpur, Maharashtra, Mobile No. 9890597344

It stores data in more efficient and practical way than basic data structures. Advanced data structures include data structures like, Red Black Trees, B trees, B+ Trees, Splay Trees, K-d Trees, Priority Search Trees, etc. Each of them has its own special feature which makes it unique and better than the others.A Red-Black tree is a binary search tree with a feature of balancing itself after any operation is performed on it. Its nodes have an extra feature of color. As the name suggests, they can be either red or black in color. These color bits are used to count the tree’s height and confirm that the tree possess all the basic properties of Red-Black tree structure, The Red-Black tree data structure is a binary search tree, which means that any node of that tree can have zero, one, or two children such that the value of right child is smaller than the value of its parent node and value of left child is greater than that of its parent node. The operations that can be performed on Red-Black trees are: Search, Insert, and Delete.A Splay Tree is also a self-balancing tree data structure which has unique quality that recently accessed item is referred as the most frequently used item and becomes the root of the tree, so that next time if the same item is searched, it can be accessed as root node in easiest way. For many sequences of non-random operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. It possesses all the properties of a binary search tree, such as, its nodes can have either zero, one or two children and value of every left child is smaller than its parent nodes value and every right child’s value is greater than the value of its parent node’s value. Splaying is performed by rotations associated with steps. The steps that can be performed are: Zig step, Zag step, Zig-Zig, Zag-Zag step, Zig-Zag, Zag-Zig step

The operations that can be performed on Splay trees by using the above steps are: Search, Insert, and Delete.

II. EVOLUTION

A. *Red-Black Trees*

In 1972, Rudolf Bayer [2] invented a data structure that was a special order-4 case of a B-tree. These trees maintained all paths from root to leaf with the same number of nodes, creating perfectly balanced trees. However, they were not binary search trees. Bayer called them a "symmetric binary B-tree" in his paper and later they became popular as 2-3-4 trees or just 2-4 trees.[3]In a 1978 paper, "A Dichromatic Framework for Balanced Trees", [4] Leonidas J. Guibas and Robert Sedgewick derived the red-black tree from the symmetric binary B-tree.[5] The color "red" was chosen because it was the best-looking color produced by the color laser printer available to the authors while working at Xerox PARC.[6] Another response from Guibas states that since they had red and black color pens readily available, those two were the chosen colors for the depiction of the property of colors in the trees and hence the name 'Red-Black' Trees.[7]In 1993, Arne Andersson introduced the idea of right leaning tree to simplify insert and delete operations.[8]In 1999, Chris Okasaki showed how to make the insert operation purely functional. Its balance function needed to take care of only 4 unbalanced cases and one default balanced case.[9]The original algorithm used 8 unbalanced cases, but Cormen et al. (2001) reduced those 8 unbalanced cases to 6 unbalanced cases.[1] To ease it all, the insertion operation in Red-Black trees could be implemented in just 46 lines of code according to Sedgewick.[10][11] In contrast to the right leaning tree proposed by Arne Andersson, the year 2008 witnessed the proposition of left-leaning Red-Black trees which in turn simplified the algorithms.

B. Splay Trees

During the early years of advancements in Computer Science and data structures such as Binary Search trees, Daniel Sleator and Robert Tarjan contemplated and realized that many a times we are not interested in the individual time required for the execution of an operation but rather the total time for a sequence of operations. So, we could allow certain operations to be expensive as long as they are balanced out by a number of cheap operations. Around that time, the concept of self-adjusting list and move-to-front rule, which was Lastin First Out (LIFO) operation were prevalent. So, whenever we wanted to move a page out of the main memory, the Least Recently Used (LRU) page would be removed. In the case of Move-To-Front rule, whenever we accessed an item, the item would be moved to the front of the list so the next time it is accessed, it is cheap to do so. In 1985, a similar scheme was found for Binary Search Trees (BSTs) which got rid of the complicated rebalancing and was known as Splay Trees which were self-adjusting in nature. The idea was that every time we did access a node, we perform rotation along the access path that moves the accessed item all the way to the root of the tree. So, if it is accessed again soon, it is going to be cheap to access. We don't do it by rotating one at a time bottom up or one at a time top down, rather it is donedependingon the local structure and orientation of the tree because if we do it one at a time, it won't work. Thus, the concept of 'locality of reference' is associated with Splay Trees which were invented by Daniel Sleator and Robert Tarjan in 1985.No node in the tree gets pushed down by more than a certain additive constant and it is simple to program.

III. OPERATIONS ON TREES

A. Red-Black Trees

Rotations that can be performed on splay trees are:

i. Right Rotation

While rotating a tree in right direction about any node, its parent becomes its new right child and its left child remains the same. Its old right child becomes new right child's left node and so on, until a proper tree is formed.

ii. Left Rotation

While rotating a tree in left direction about any node, its parent becomes its new left child and its right child remains the same. Its old left child becomes new left child's right node and so on, until a proper tree is formed.

The operations that can be performed on Red Black trees are:

1. Search

When we need to search an element in a Red-Black tree, the value of element to be searched is compared with value of root node. Let's take the element to be searched as 'k'.If k is equal to root, return the root node, Otherwise, if k<root, traverse in left subtree or if k>root, traverse in the left subtree. In this way, compare nodes with k until the tree is over. If element is found in the tree, return it, otherwise inform that the searched element does not exist in that tree.

2. Insertion

When z lies in the left branch of the tree:
If child node i.e., the newly inserted node and its parent are red, then check color (sibling (parent (z))). If color (sibling (parent (z))) =Red, then simply recolor sibling (parent (z)), parent (z) and parent (parent (z)). Otherwise, check if z is in the right branch, then set z=parent(z) and left rotate at z, recolor parent(z), parent(parent(z)) and right rotate at parent(parent(z)).
Set root node as black.
When z lies in right branch of the tree, same as above clause with "right" and "left" exchanged.

3. Deletion

For x in left branch of the tree,
Check the color of w i.e., sibling (x)
Case 1:If color (w) = Red, then set color (w) = Black, Parent (x) = Red, Left Rotate at parent (x) and Set w = right child (parent (x))
Case 2:If color (w) = Black and both children are Black then set color (w) = Red and Set x = parent (x)
Case 3:If color (w) = Black, color (left child) = Red and color(w) = Black, then set color (Left child (w)) = Black, set color (w) = Red, Right rotate at w and Set w = right child (parent (x))
Case 4:If color (w) = Black and Right child (w) = Red then set color (w) = color (parent (x)), set color (parent (x)) = Black, set color (right child (w)) = Black, left rotate at parent (x), and set x as root node.
Set color (root node) = Black

B. Splay Trees

Table 1: Operations performed on Splay Trees

Operation	Rotation	Approach
Zig	Right	
Zag	Left	
Zig-Zig	Right Rotation, Right Rotation	Top-Down
Zag-Zag	Left Rotation, Left Rotation	Top-Down
Zig-Zag	Left Rotation, Right Rotation	Bottom-Up
Zag-Zig	Right Rotation, Left Rotation	Bottom-Up

Operations that can be performed on Splay Trees are:

1. *Search*

   i. Find the position of the element.
   ii. Splay the tree at that element according to its position using different operations and rotations mentioned above.
   iii. The resultant tree will have the searched element as its root.

2. *Insertion*

   i. Find an appropriate position for the element to be inserted according to the value of the nodes of tree and new element.
   ii. Insert the element at that position and splay the resultant tree at the newly inserted element.
   iii. The result will be a tree with newly inserted element as its root.

3. *Deletion*

   i. Find the element to be deleted in the tree.
   ii. Splay the tree at that element.
   iii. When the element to be deleted becomes the root of the tree, delete the element.
   iv. The result we will get are two separate sub-trees whose roots were left and right child of the root.
   v. If left sub-tree is not null, splay the left sub-tree at the node with maximum value. And join the right sub-tree to it.
   vi. If left sub-tree is null, return the right sub-tree as it is.

IV. AMORTIZED ANALYSIS AND COMPARISION

Amortized analysis is a way of evaluating the complexity of an algorithm i.e., how much memory and time it consumes to execute. It is not the average case analysis of any algorithm. Costly and less costly operations are considered in amortized analysis over whole series of operations in the algorithm. Many factors like length of input and different type of input affect the analysis and its performance. It is based on the number of operations or sequence of operations instead of input sample. It is worst case analysis on sample input.

A. Red-Black Trees

Amortized analysis of Red-Black Trees is calculated in the basic operations on the tree. While calculating the worst-case times, we generally sum the worst-case times of all the individual operations that are performed in order to arrive at a particular result. This might prove to be a pessimistic approach. Consequently, we are concerned with the total running time for that sequence of operations, not the individual running time of a single operation. Amortized analysis does just that. It takes the average of total running times of operations in a sequence rather than the total number of operations.

Terms to be considered are:

1. Credit of the nodes
2. Total credit

Note: Credits are given to the nodes which are BLACK in colour ONLY.
Basis of assigning credits:

Table 2: Basis of Assigning Credits

Orientation	Credit Assigned
A Black node with 1 Red Child	0
A Black node with 2 Black Children	1
A Black node with 2 Red Children	2

Bank's View of Amortization: Robert Tarjan first proposed the Bank's view of amortization. According to it, each time we perform a rotation after an update operation, we deposit some credit into the account and each time we complete the update operation without rebalancing the tree, we withdraw some credits from the account. The amount to be credited or debited from the account depends upon the type of rotation performed. Once the sequence of update operations has been completed, we get the lower and higher bounds of performance of that data structure, Red-Black trees in this case.
The following equations can be used to calculate the amortized complexity of Red-Black Trees which were proposed by Tarjan.

$$A_i = T_i - \Phi_i - \Phi_{i-1} \qquad (1)$$

Where, A_i= Amortized time of ith operation
T_i = Actual time of the ith operation
Φ = Is the potential function

$$\sum_{i=1}^{m} T_i = \sum_{i=1}^{m} (A_i - \Phi_i + \Phi_{i-1})$$
$$= \Phi_0 - \Phi_m + \sum_{i=1}^{m} A_i \qquad (2)$$

where, Φ_0 is the potential before the sequence of operations
Φ_m is the potential after complete sequence ofoperations.

These formulae are applied for insertion and deletion in the following manner.

Table 3: Bottom Up Insertion Algorithm in Red-Black Tree

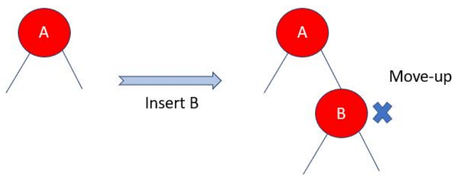
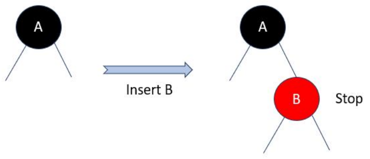
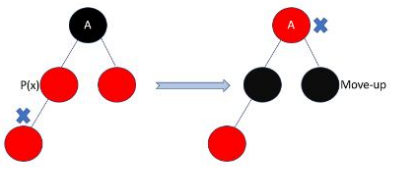
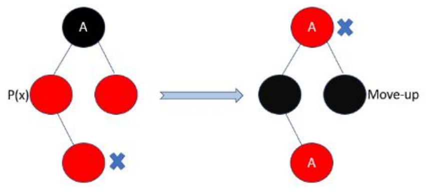
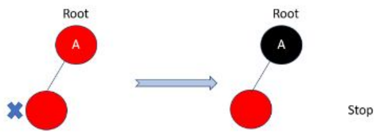
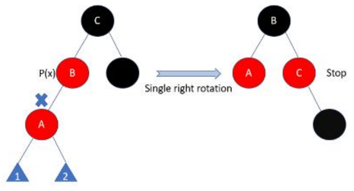
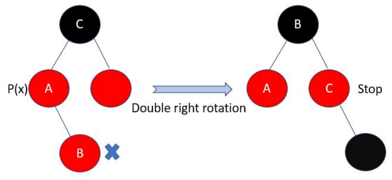
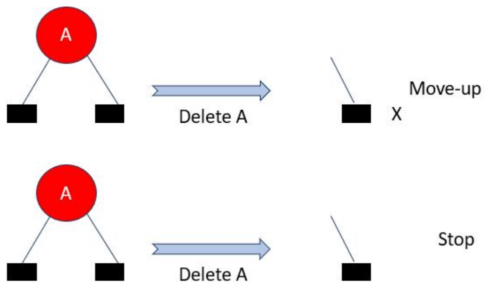
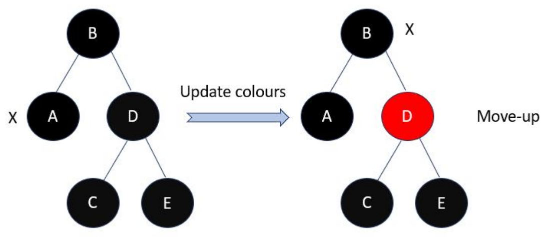
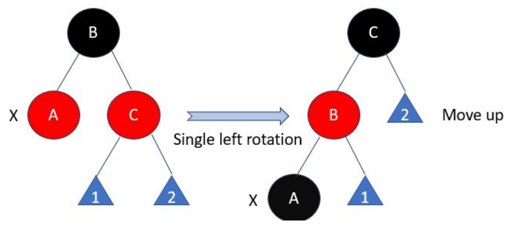
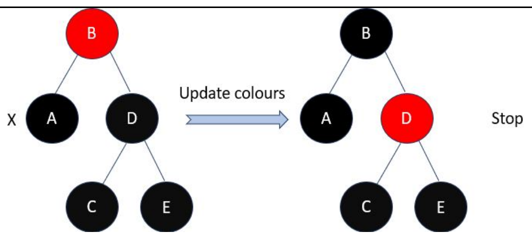
Case	Depiction
1.a	
1.b	
2.a	
2.b	
3.	
4.	 C P(x) B A 1 2 Single right rotation B A C Stop
5.	 C P(x) A B Double right rotation B A C Stop

Table 4: Rules for assigning credit to nodes during Insertion

Condition	Action on credits
Attach a node to a black node and terminate insertion (Refer Table 1 Case 1.a, 1.b)	-1
Update colors and move up (Refer Table 1 Case 2.a, 2.b)	-1
Perform single/double rotation and terminate (Refer Table 1 Case 4 and 5)	+2

Table 5: Bottom up Deletion Algorithm for Red-Black Tree

Case	Depiction
1	
2	 B X A D C E Update colours B X A D C E Move-up
3	 B X A C 1 2 Single left rotation C B 2 Move up X A 1
4	 B X A D C E Update colours B A D C E Stop

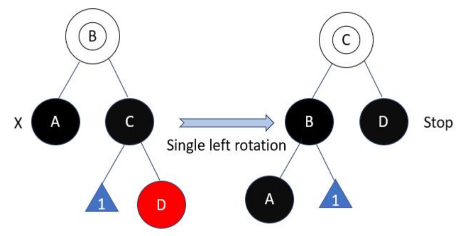
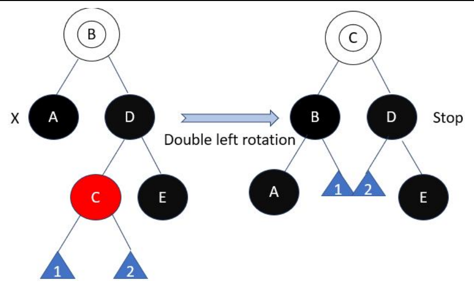
5	
6	

Table 6: Rules for assigning credit to nodes during Deletion

Condition	Action on credits
Delete a black node and move up (Refer Table 2 Case 1)	-1
Update colors and move up (Refer Table 2 Case 2)	-2
Update colors and terminate deletion (Refer Table 2 Case 4)	-1
Perform single rotation and terminate deletion (Refer Table 2 Case 5)	-1/+2
Perform double rotation and terminate deletion (Refer Table 2 Case 6)	+2

B. *Splay Trees*

Amortized analysis of splay trees is calculated on the basis of operation performed on the tree. In the case of Splay Trees, worst case complexity is O(n) for search or splaying nodes. But, using amortized analysis we get $O(\log_2 n)$ for search and splaying nodes.

Terms to be considered are:
1. Rank of the node
2. Credit of the node
3. Total credit

Amortized analysis for splay trees uses potential method to find out amortized complexity. Amortized complexity of a splay step iis given by:

$$A_i = T_i + (Cr_{\,i} - Cr_{\,i-1})$$

Where, Cr_i = Credit balance of tree after splay operation,
$Cr_{\,i-1}$ = Credit balance of tree before splay operation,

$T_{\,i}$ = number of levels the target node raises.
The value of T_i changes according to the operation being performed.

1. For Zig/Zag operation, $T_i = 1$
2. ForZig-Zig/Zig-Zag/Zag-Zig/Zag-Zagoperation, $T_i = 2$.
   $Cr(u) = r(u) = \log 2\,|T(u)|$

Credit of tree is summation of credits of all the nodes,

$$Cr_{\,i} = \sum_{i \epsilon Ti(u)} r_i\,(u)$$

Observations:
Amortized complexity Ai of a splay tree for,
Zig-Zig or Zag-Zag: $A_i < 3(r_i(u) - r_{i-1}(u))$,
Zig-Zag or Zag-Zig: $A_i < 2(r_i(u) - r_{i-1}(u))$,
Zig or Zag operation: $A_i < 1 + (r_i(u) - r_{i-1}(u))$.

V. EXAMPLES

A. *Red Black Trees*

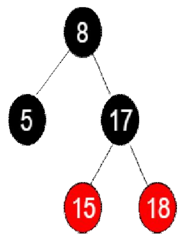


Fig. 1: Example of Red-Black Tree

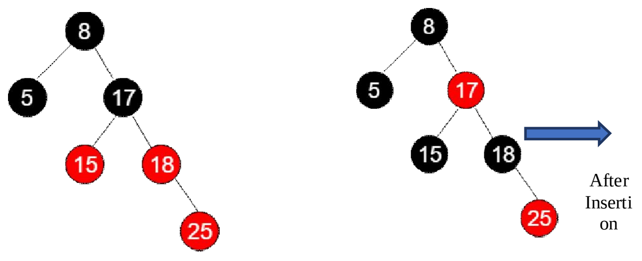


Fig. 2: Insertion Process

Example: **Insert** 25 in the following Red-Black Tree
The above insertion comes under case 2.a and 2.b

Table 7: Listing and Calculation of Potential during Insertion

Nodes	8	5	17	15	18	25	Potential
R_{before}	1	1	2	-	-	-	4
R_{after}	0	1	-	1	0	-	2

Assumption of $T_i$is done by the following table.

Table 8: T_iAssumption

Operation	Assumed Value of T_i
Left Rotate	1
Right Rotate	1
Recolor Nodes	Number of Nodes Recolored (E.g. If 2 nodes are recolored, then T_i= 2)

Based on the above table, the value of T_i= 4 in our case (z = node inserted. Nodes recolored:p(z), p(p(z)), sibling of p(z) and root (already black in color))
According to equation 1, Amortized Time, Ai = 4–2–4 = -2
Using Equation 2, T_i = 4 – 2 + (-2) = 0
Thus, the complexity is reduced from a linear complexity O(n) to a logarithmic complexity O(log n).

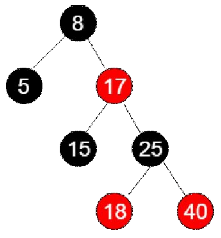


Fig. 3: Example of Red-Black Tree

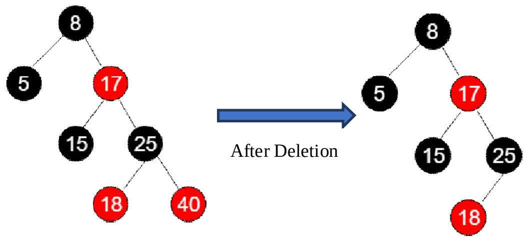


Fig. 4: Deletion Process

Example 2: **Delete**element 40 from the given Red-Black Tree

Table 9: Listing and Calculation of Potential duringDeletion

Nodes	8	5	17	15	25	18	40	Potential
R_{before}	0	1	-	1	2	-	-	4
R_{after}	0	1	-	1	-	1	-	3

Based on the assumption on Table Number 6, the assumed value of T_i is 0 (Since the node to be deleted is Red in color, no extra updating operation is performed and the node is directly deleted)
According to equation 1, Amortized Time, A_i=0–3 –4 =-7
Using equation 2, T_i = 4 – 3 + (-7) = -6
Thus, the complexity is reduced from a linear complexity O(n) to a logarithmic complexity O(log n).

B. Splay Trees

Example: **Splay** the given tree at 81.

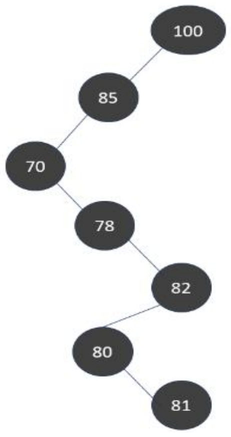


Fig. 5: Example of Splay Tree

Solution:

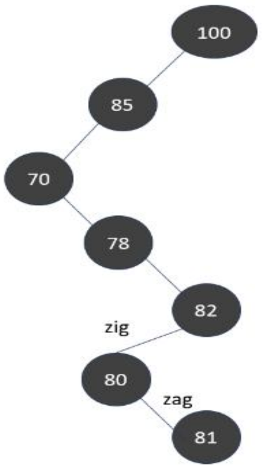


Fig. 6: Given splay tree

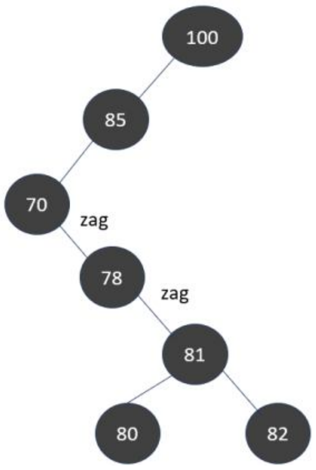


Fig. 7: Splay tree after Zig-Zag operation

Table 10: Potential Calculation

		100	85	70	78
R before	ZIG-ZAG	$Log_2 7$	$Log_2 6$	$Log_2 5$	$Log_2 4$
R after		$Log_2 7$	$Log_2 6$	$Log_2 5$	$Log_2 4$
R before	ZAG-ZAG	$Log_2 7$	$Log_2 6$	$Log_2 5$	$Log_2 4$
R after		$Log_2 7$	$Log_2 6$	$Log_2 1$	$Log_2 3$
R before	ZIG-ZIG	$Log_2 7$	$Log_2 6$	$Log_2 1$	$Log_2 3$
R after		$Log_2 1$	$Log_2 3$	$Log_2 1$	$Log_2 3$

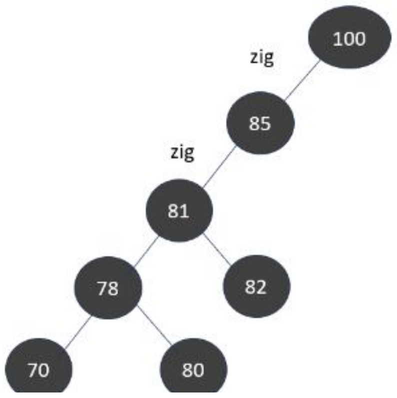


Fig. 8: Splay tree after Zag-Zag operation

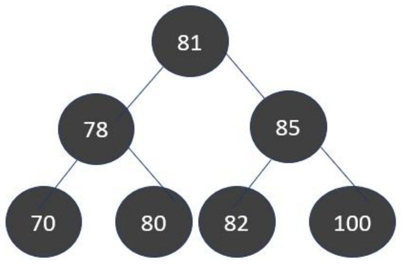


Fig. 9: Splay tree after Zig-Zig operation

For i = 1,
$A_1 = t_1 + Cr_1 - Cr_0$
$= 2 + [(\log_2 1 + \log_2 1 + \log_2 3) - (\log_2 2 + \log_2 3 + \log_2 1)]$
$= 1$

$A_1 = 1$ in amortized complexity
$A_1 < 2\ (r_1\ (81) - r_0\ (81))$
$A_1 < 2\ (\log_2 3 - 0)$
$1 < 2\ (\log_2 3)$
Therefore, True.

For i = 2,
$A_2 = t_2 + Cr_2 - Cr_1$
$= 2 + [(\log_2 5 + \log_2 4 + \log_2 1 + \log_2 1 + \log_2 3) - (\log_2 1 + \log_2 3 + \log_2 1 + \log_2 1 + \log_2 5)]$
$= 0$

$A_2 = 0$ in amortized complexity
$A_2 < 3\ (r_2\ (81) - r_1\ (81))$
$A_2 < 3\ (\log_2 5 - \log_2 3)$
$0 < 3\ (\log_2 5 - \log_2 3)$
Therefore, True.

For i = 3,
$A_3 = t_3 + Cr_3 - Cr_2$
$= 2 + [(\log_2 5 + \log_2 4 + \log_2 1 + \log_2 1 + \log_2 3) - (\log_2 1 + \log_2 3 + \log_2 1 + \log_2 1 + \log_2 5)]$
$= 0$

$A_3 = 0$ in amortized complexity
$A_3 < 3\ (r_3\ (81) - r_2\ (81))$
$A_3 < 3\ (\log_2 7 - \log_2 5)$
$0 < 3\ (\log_2 7 - \log_2 5)$
Therefore, True.

VI. APPLICATIONS

A. Red Black Trees

i. The scheduler of Linux kernel uses Red Black trees as its data structure. The scheduler was merged into the 2.6.23 release of Linux kernel and its name is Completely Fair Scheduler (CFS). It handles CPU resource allocation for executing processes and aims to maximize overall CPU utilization while also maximizing interactive performance.
ii. Java uses Red Black tree data structure to implement various tree classes of it. Some of them arejava.util.TreeMapandjava.util.TreeSet.
iii. C++ STL uses Red Black trees for map, multimap, and multiset.

B. Splay Trees

i. Search Engine Optimization
ii. Used for implementation of cache algorithms
iii. Can be used in Garbage collectors
iv. Used in Network Routers

VII. CONCLUSION

1.Red black trees and splay trees are two of the most beneficial advanced tree data structures which possess unique features. Red black trees are used in those cases where insertions and deletions are rare and accessibility of data needs to be fast and efficient.

2.The amortized analysis of red black tree takes into consideration the time taken by a sequence of operation to be performed during updating rather than the time taken for a single operation to take place. It also considers the cases where particular insertions or deletions fit into.

3. Splay trees are widely used in areas where recently accessed files and least recently accessed files are needed. Amortized analysis of splay trees is calculated and verified with the observations according to the operation performed.

ACKNOWLEGMENT

This idea for this paper was supported by the Computer Science and Engineering Department We thank our professors from Shri Ramdeobaba College of Engineering and Management who provided insight and expertise that greatly assisted the research, although they may not agree with all of the interpretations/conclusions of this paper.
We thank Prof. ShyamalMundada, Assistant Professor, RCOEM for assistance with the concept of Red-Black and Splay Trees.We would also like to show our gratitude to the Dr. M.B. Chandak, HOD, CSE, RCOEM for sharing their pearls of wisdom with us during the course of this research and providing us with any required resources, and we thank 3 “anonymous” reviewers for their so-called insights. Finally, we would like to mention that any errors in the paper are our own and should not tarnish the reputations of these esteemed persons.

REFERENCES

[1] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001). "Red–Black Trees". Introduction to Algorithms (second ed.). MIT Press. pp. 273–301. ISBN 0-262-03293-7.

[2] Rudolf Bayer (1972). "Symmetric binary B-Trees: Data structure and maintenance algorithms". Acta Informatica. 1 (4): 290–306. doi:10.1007/BF00289509.

[3] Drozdek, Adam. Data Structures and Algorithms in Java (2 ed.). Sams Publishing. p. 323. ISBN 0534376681.

[4] Leonidas J. Guibas and Robert Sedgewick (1978). "A Dichromatic Framework for Balanced Trees". Proceedings of the 19th Annual Symposium on Foundations of Computer Science. pp. 8–21. doi:10.1109/SFCS.1978.3.

[5] "Red Black Trees". eternallyconfuzzled.com. Retrieved 2015-09-02.

[6] Robert Sedgewick (2012). Red-Black BSTs. Coursera. A lot of people ask why did we use the name red–black. Well, we invented this data structure, this way of looking at balanced trees, at Xerox PARC which was the home of the personal computer and many other innovations that we live with today entering[sic] graphic user interfaces, ethernet and object-oriented programmings[sic] and many other things. But one of the things that was invented there was laser printing and we were very excited to have nearby color laser printer that could print things out in color and out of the colors the red looked the best. So, that’s why we picked the color red to distinguish red links, the types of links, in three nodes. So, that’s an answer to the question for people that have been asking.

[7] "Where does the term "Red/Black Tree" come from?". programmers.stackexchange.com. Retrieved 2015-09-02.

[8] Andersson, Arne (1993-08-11). Dehne, Frank; Sack, Jörg-Rüdiger; Santoro, Nicola; Whitesides, Sue, eds. "Balanced search trees made simple" (PDF). Algorithms and Data Structures (Proceedings). Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg. 709: 60–71. doi:10.1007/3-540-57155-8_236. ISBN 978-3-540-57155-1. Archived from the original on 2000-03-17.

[9] Okasaki, Chris (1999-01-01). "Red-black trees in a functional setting" (PS). Journal of Functional Programming. 9 (4): 471–477. doi:10.1017/S0956796899003494. ISSN 1469-7653.

[10] Sedgewick, Robert (1983). Algorithms (1st ed.). Addison-Wesley. ISBN 0-201-06672-6.

[11] Sedgewick, Robert; Wayne, Kevin. "RedBlackBST.java". algs4.cs.princeton.edu. Retrieved 7 April 2018.