Weak AVL Tree

November 21, 2021

Pranavkumar Mallela : 2020CSB1112 ,
Akshat Toolaj Sinha : 2020CSB1068 ,
Pratham Kundan : 2020CSB1114

Instructor:
Dr. Anil Shukla

Teaching Assistant:
Mr. Ravi Bhatt

Summary: This reports explains a new Data Structure named Weak Adelson-Velskii and Landis (WAVL) Tree. It discusses Insertion and Deletion using bottom-up re-balancing. Also given are proofs and intuition of various properties and operations of the WAVL tree.

---

1. Introduction

After learning and implementing the AVL tree within this course, we were motivated to study more about rank balanced trees. Weak AVL tree is a rank balanced tree that resembles both red black trees and AVL trees. In fact, it exhibits the best properties of both trees when it comes to complexity of various operations.

If no deletions occur, the WAVL tree formed is exactly the same as the corresponding AVL tree. With deletions included , its height is at least that of an AVL tree, with the same number of insertions but no deletions. WAVL trees are a proper subset of red black trees but follow a different balance rule and different re-balancing algorithms.

In the WAVL tree, insertions and deletions take at most two rotations and O(1) in the worst case; red black trees need three rotations in the worst case. The research paper taken as reference states that "..we know of no other balanced binary search tree in which deletions can be done in only two rotations". Like in other balanced trees , the WAVL tree has a balance rule that makes its height a constant factor of the logarithm of its size or O(Log(N)) where N is number of nodes in the tree.

Our main goal was to learn, implement and analyse this new data structure, taking reference from the research paper written by BERNHARD HAEUPLER(Carnegie Mellon University), SIDDHARTHA SEN(Microsoft Research), ROBERT E. TARJAN(Princeton University & Microsoft Research) [2]. This report contains the following:

1. Definitions of the common terminology used in WAVL trees
2. Analysis of Insertion using top-down rebalancing
3. Analysis of Deletion using top-down rebalancing
4. Comparison of WAVL tree vs AVL tree

2. WAVL Tree Terminology

2.1. Nomenclature of WAVL Trees

This sub-section gives some basic properties of the WAVL tree and the balancing rule associated with the same.

A binary search tree is a collection of Nodes which follow some properties. Let us say we have a node X,

- Value associated with the node : $X \rightarrow Value$
- Pointer to Left Child represented by Left(X) or L(X)
- Pointer to Right Child represented by Right(X) or R(X)
- Pointer to parent represented by Parent(X) or P(X)
- Rank of the node(defined later in this section)

If memory address pointed by pointer is non-essential, we denote it by NULL pointer.
Nodes are classified in three categories:

1. Binary : Both child pointers are non-NULL
2. Unary : A node with only one child pointer as non-NULL
3. Leaf: Both child pointers are NULL

We also define height and size of the WAVL Tree:

1. Height or H(X) = max(H(L(X)) , H(R(X))) +1 , Base Case : H(NULL) =-1
2. Size or S(X) = S(L(X)) + S(R(X)) +1 , Base Case : S(NULL) =0

In our implementation of the WAVL tree rotations play a vital role in performing insertions and deletions. The below figures demonstrate:

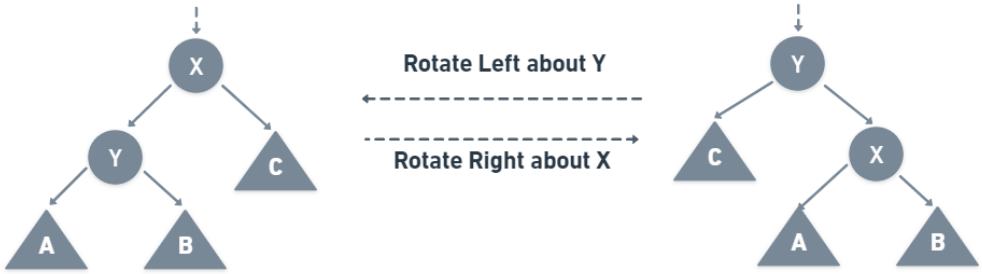


Figure 1: Rotation

To perform Right Rotate, we do RightRotate(X). To perform Left Rotate we do LeftRotate(Y).

2.2. Rank Balancing Rules

Firstly, let us define some terminology to help understand rank balancing rules:

- We assign a non-negative-integer called Rank or RK(X) to every node of the WAVL Tree which resembles closely to height of node in the respective tree
- NULL nodes have rank -1
- Rank difference of Non-NULL node is defined as RK(P(X)) - RK(X)
- A non-root node is called a k-child if its rank difference is k
- A node is p,q if its child has rank differences p and q, order of children is not taken into consideration here

So Rank Balancing rules in WAVL Tree are:

1. All rank differences are 1 or 2
2. Every leaf has rank 0

We will see in further sections how these rules make the WAVL Tree a Balanced Binary Search tree.

Theorem 2.1. *Let r, h and s denote the rank, height and size of a WAVL tree respectively, then* $h \leq r \leq 2h$ *and* $r \leq 2lgs$

Proof. Valid rank difference in a WAVL Tree is 1 or 2. Assume that rank denotes the floor number in a building and that leaves are at the ground floor. We can figure out that from a current floor, we can only go one or two floors up. So to find range of rank, we always make the optimal choice to find the bound, as to find the lower bound, we go only one floor up.

$$h \leq r \tag{1}$$

By a similar argument, one can infer that:

$$r \leq 2h \tag{2}$$

From Eqn (1) and Eqn (2) above, we get the required inequality.
To prove the height bound, we firstly find minimum number of nodes in a tree which has a rank r *i.e* s_r
We find a recurrence relation that s_r follows and then solve it. The recurrence relation is:

$$s_r = 1 + 2s_{r-2} \ , s_0 = 1, s_1 = 2$$

Intuition is given below:

Let us say we have a tree of rank r, having minimum number of vertices s_r. Now, if we are to have minimum number of vertices, we should make the sub-tree rooted at this position to be rank $r-2$, as we will miss the node of rank $r-1$ altogether. Hence, we get the recurrence relation given above.

Now, to get a closed form formula of s_r , we make two cases:

When r is even :

$$\begin{gathered} s_2 = 1 + 2s_0 \\ s_4 = 1 + 2s_2 = 1 + 2 + 4s_0 \\ s_6 = 1 + 2s_4 = 1 + 2 + 4 + 8s_0 \\ s_r = 1 + 2 + 4 + ...2^{\frac{r}{2}-1} + 2^{\frac{r}{2}} s_0 \\ s_r = \sum_{l=1}^{\frac{r}{2}} 2^l + 2^{\frac{r}{2}} s_0 \\ \boldsymbol{s_r = \sum_{l=1}^{\frac{r}{2}} 2^l + 2^{\frac{r}{2}}} \end{gathered}$$

From this we can see that:

$$2^{\frac{r}{2}} \leq s_r$$

When r is odd:

$$\begin{gathered} s_3 = 1 + 2s_1 \\ s_5 = 1 + 2s_3 = 1 + 2 + 4s_1 \\ s_7 = 1 + 2s_5 = 1 + 2 + 4 + 8s_1 \\ s_r = 1 + 2 + 4 + ...2^{\frac{r-1}{2}-1} + 2^{\frac{r-1}{2}} s_1 \\ \boldsymbol{s_r = \sum_{l=1}^{\frac{r-1}{2}} 2^l + 2^{\frac{r-1}{2}} s_1} \end{gathered}$$

Similar to the statement above:

$$\begin{gathered} 2^{\frac{r-1}{2}} + 2^{\frac{r-1}{2}} \leq s_r \\ 2^{\frac{r+1}{2}} \leq s_r \\ 2^{\frac{r}{2}} \leq s_r \end{gathered}$$

So, in general we have:

$$\begin{gathered} 2^{\frac{r}{2}} \leq s_r \\ r \leq 2lg(s) \end{gathered}$$

Using the relation proved, we can see that:

$$h \leq 2lg(s)$$

As seen in [2], AVL tree has a height (h)

$$h \leq log_\phi(n) \ , \ \phi = \frac{1+\sqrt{5}}{2}$$

So, height of the WAVL Tree can be more than AVL Tree, thus increasing the search time than in AVL Tree.

3. WAVL Operations

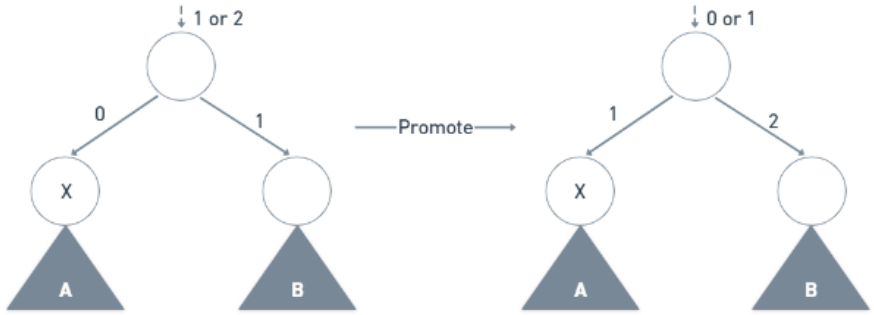
3.1. Pseudocode

1. Promote Function increases Rank of Node by 1 unit.
2. Demote Function decreases Rank of Node by 1 unit.

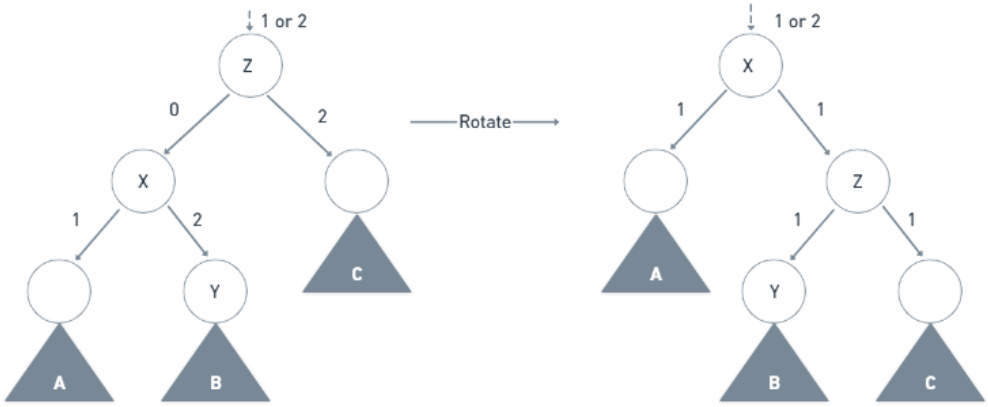
Algorithm 1 Insert(root, x)

```
1: if root = NULL then
2:    return GetNode(x)
3: else if x is greater than root → value then
4:    Insert(root → right, x)
5: else
6:    Insert(root → left, x)
7: end if
8: if root is a 0,1 node then
9:    return Promote(root)
10: else if root is a 0,2 node then
11:    if right side rank difference = 0 then
12:       x = right child of root
13:       y = left child of x
14:    else
15:       x = left child of root
16:       y = right child of x
17:    end if
18:    if y = NULL or y is a 2-child then
19:       if right side rank difference = 0 then
20:          LeftRotate(root)
21:          Demote(left child of root)
22:       else
23:          RightRotate(root)
24:          Demote(right child of root)
25:       end if
26:       return root
27:    else if y is a 1-child then
28:       if right side rank difference = 0 then
29:          LeftRotate(x)
30:          RightRotate(root)
31:       else
32:          RightRotate(x)
33:          LeftRotate(root)
34:       end if
35:       Demote(left child of root)
36:       Demote(right child of root)
37:       return Promote(root)
38:    end if
39: else
40:    return root
41: end if
```

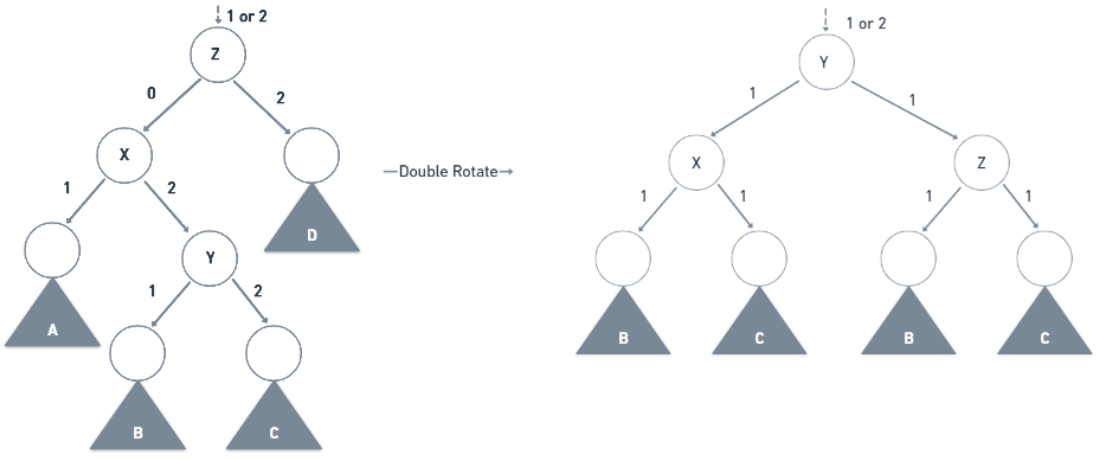
Visual description of Insertion Re-balancing:



(a) Promote.



(b) Single Rotate



(c) Double Rotate

Figure 2: Insertion Re-balancing. (Mirror cases also exist)

When a new element is inserted in WAVL Tree, it can only be inserted at a leaf node or unary node :

1. Unary Node A unary node has to be 2,1 otherwise, it would violate the properties of WAVL Tree and we attach our element at 2,1 making it 1,1 and insertion ends here as no property is violated in the tree.

2. Leaf Node : Attaching an element makes the leaf a 0,1 node. Hence, we start promotion bottom-up until we get a 0,2 node and we perform Single Rotate, Double Rotate. Else, we get a 1-child in our path, which

completes the re-balancing as well as the insertion.

Algorithm 2 Delete(Root, x)

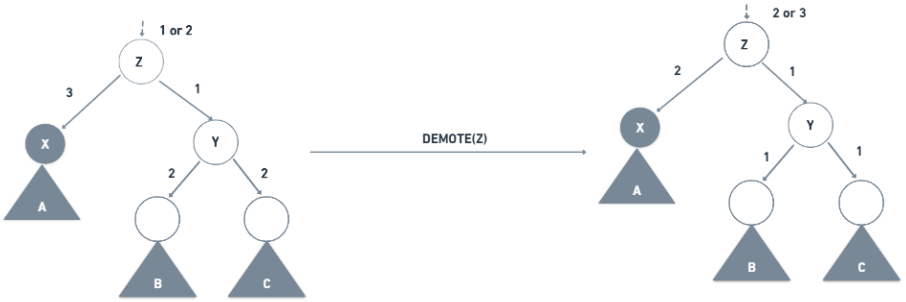
```
1: X = search(x)
2: if X is a leaf then
3:    P = P(X)
4:    remove X
5:    Rebalance(P)
6:    return Root
7: else if X is not a leaf then
8:    Swap values of X with predecessor/successor of X
9:    Delete(X,x)
10:   return Root
11: else
12:   return Root
13: end if
```

Algorithm 3 Rebalance(X)

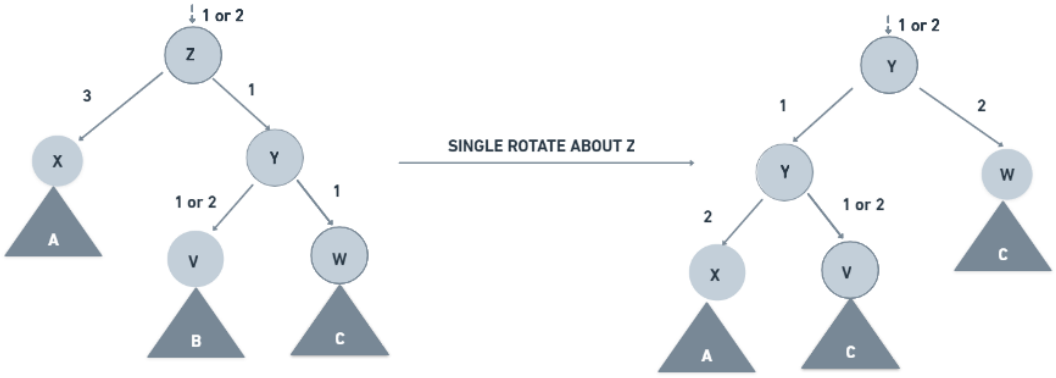
```
1: Y = Sibling(X)
2: while x is a 3-child and y is a 2-child or 2,2 do
3:    if Y is not a 2-child then
4:       Demote(Y)
5:    end if
6:    Demote(X)
7:    X = P(X)
8:    Y = Sibling(X)
9: end while
10: if P(X) is 1,3 and Y is not 2,2 then
11:   if Y is left child then
12:      if Y→left is a 1 node then
13:         LeftRotate about P(Y)
14:         Demote(P(Y)) and Promote(Y)
15:      else
16:         LeftRotate about Y and RightRotate ABOUT P(Y)
17:         DoubleDemote(P(Y))
18:         Demote(Y)
19:         DoublePromote(P(X))
20:      end if
21:   else if Y is right child then
22:      if Y→right is a 1 node then
23:         RightRotate about P(Y)
24:         Demote(P(Y)) and Promote(Y)
25:      else
26:         RightRotate about Y and LeftRotate ABOUT P(Y)
27:         DoubleDemote(P(Y))
28:         Demote(Y)
29:         DoublePromote(P(X))
30:      end if
31:   end if
32: end if
```

Visual description of Deletion Re-balancing:

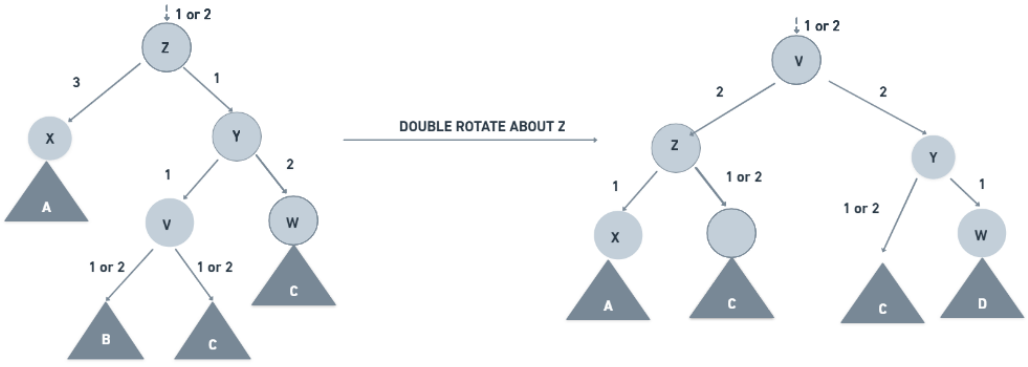
(a) Demote.



(b) Demote



(c) Single Rotate



(d) Double Rotate

Figure 3: Deletion Re-balancing. (Mirror cases also exist)

When a element is deleted, one of the following three cases arises:

1. Leaf : We start demotion while going bottom-up until we get a node in which we can apply Single Rotate / Double Rotate of Deletion

2. Unary Node : We replace the unary node with its neighboring leaf. We delete the leaf node and continue re-balancing.
3. Binary Node : We find the inorder successor of the element to be deleted and then swap the values to make it one of Case 1 or 2 and then delete it from the whole tree.

The number of rebalancing steps are actually exponentially smaller in rank k, hence we can infer that most rebalancing steps occur lower in the tree.

Theorem 3.1. *In a WAVL Tree with Bottom-up rebalancing , the number of deletion rebalancing steps of rank k is $O(d/b_1^k)$, where d is the number of deletions and b_1 is the plastic constant.*

Theorem 3.2. *In a WAVL Tree with Bottom-up rebalancing , the number of insertion rebalancing steps of rank k is $O(m/b_1^k)$, where m is the number of deletions and b_1 is the plastic constant.*

Theorem 3.1 and 3.2 are proved using Amortized Analysis. The general approach for proving is :
Assigning potential to a specific type of node of some rank k. We define potential in such a way that it is exponential in rank , but increases by a constant amount or O(1) per rebalancing. Then we limit our potential function to Rank k to show the given Theorem.
Formal proof is given in [2]

4. Comparison with other Balanced BST

Theorem 4.1. *AVL Trees forms a proper subset of WAVL Tree*

Proof. Inspecting our Bottom-up insertion algorithm carefully , we can see that is exactly the same as the original insertion algorithm for AVL Tree. So the tree obtained by doing only insertion on an empty WAVL tree will give us the AVL Tree on which insertions are done in the same order. They only differ after at least one deletion.
To prove given theorem , firstly we will show that every node in a AVL Tree is 1,1 or 1,2, where rank of node is equal to height of the node. Observe that , in AVL tree , Balance factor : $|bf| \leq 1$, so atleast one of rank difference of child has to be 1, otherwise, we will skip one height level in between, which is not allowed in AVL tree as rank of a node is equal height,

So valid rank difference are 1,1 and 1,2 (2,1 is equivalent to this)

So AVL is a subset of WAVL Tree. WAVL is formed by relaxing the rank balance rule of AVL tree, i.e by including 2,2 nodes.
We can also show that WAVL Trees form a proper subset of Red-Black Trees similarly. As the number of deletions increase, the height of WAVL degrades from AVL Tree to Red-Black Tree.

We know that Bottom-Up Rebalancing in Insertion or Deletion takes O(log n) Rank Changes and O(1) Rotations amortized. [2]

5. Conclusions

We have used rank and rank differences of a node X to obtain Balanced Binary Search Trees .We have shown/cited many of its similarities with AVL and Red-Black tree. We have also implemented Insertion and Deletion using Bottom-up Rebalancing.
Future improvements can be:

1. Insertion and Deletion using Top-Down Rebalancing
2. Use of Negative/Fractional Ranks
3. Improvement in Rank Rule

6. Bibliography and citations

Acknowledgements

We thank our instructor, Dr. Anil Shukla and our Teaching Assistant Mr. Ravi Bhatt Sir for their guidance and useful suggestions throughout our project.

References

[1] Tamassia Goodrich. Weak avl trees. "https://www.ics.uci.edu/ goodrich/teach/cs165/notes/WeakAVLTrees.pdf", 2015.

[2] TARJAN HAEUPLER, SEN. Rank-balanced trees, 2014.

[3] Wikipedia. Weak avl trees. 2015.