

Data Storage and Retrieval Using AVL Tree

Venkatakrishnan R
CH.EN.U4AIE20078

Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

Siva Jyothi Nath Reddy B
CH.EN.U4AIE20063

Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

Sarthak Yadav
CH.EN.U4AIE20058

Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

Shaik Huziafa Fazil
CH.EN.U4AIE20060

Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

Pravine Mukesh
CH.EN.U4AIE20050
Computer Science and Engineering (AI)
Amrita Vishwa Vidyapeetham, Chennai

1.Introduction

One of the most essential data structures is the tree, which is used to conduct operations like insertion, deletion, and searching of items efficiently. Construction of a well-balanced tree for sorting all data is not practicable when working with a huge number of data, though. As a result, only valuable data is saved as a tree, and the actual volume of data used changes over time as new data is inserted and old data is deleted. It is possible to conduct traversals, insertions, and deletions without utilizing either stack or recursion in some circumstances where the NULL link to a binary tree to special links is referred to as threads. In this project, we use the Height balance tree which is also known as the AVL tree.

2. AVL Tree

A binary search tree in which the height difference between the left and right subtrees of each node is less than or equal to one is known as an AVL tree. Adelson, Velskii, and Landi discovered the approach for balancing the height of binary trees, which is known as the AVL tree or Balanced Binary Tree.

AVL tree can be defined as follows :

Let T be a non-empty binary tree with the left and right subtrees TL and TR. If the following conditions are met, the tree is height balanced:

- TL and TR are height balanced
- $h_L - h_R \leq 1$, where $h_L - h_R$ are the heights of TL and TR

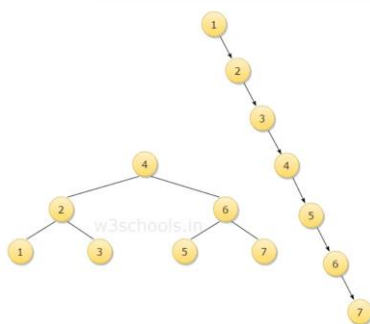
Depending on whether the height of the left subtree is larger, less than, or equal to the height of the right

subtree, the Balance factor of a node in a binary tree can be 1, -1, or 0.

3. Advantages of AVL Tree

Because AVL trees are height balance trees, operations such as insertion and deletion are quick. Consider the following scenario:

The binary tree will seem like the second figure if you have the following tree with keys 1, 2, 3, 4, 5, 6, 7:



The procedure for inserting a node with the key Q in a binary tree requires seven comparisons, but the algorithm for inserting the same key in an AVL tree requires three comparisons, as shown in the first picture.

4. Operations on AVL Tree

1) For Insertion

- First, use BST's (Binary Search Tree) insertion logic to add a new element to the tree.
- You must check the Balance Factor of each node once you have inserted the items.
- The algorithm will proceed to the next operation once the Balance Factor of each node has been determined to be 0 or 1 or -1.
- When any node's balance factor falls outside of the aforementioned three ranges, the tree is considered to be unbalanced. The algorithm will then proceed to the next operation after performing the appropriate Rotation to make it balanced.

2) For Deletion

- To begin, locate the node where k is stored.
- Secondly, remove the node's contents (Suppose the node is x)

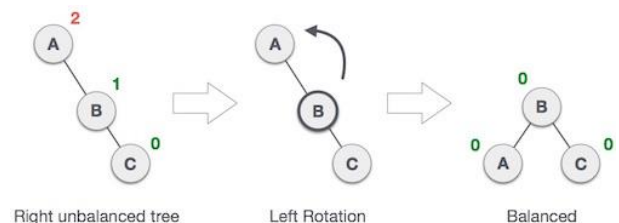
- Claim: In an AVL tree, eliminating a leaf can lower the size of a node. There are three scenarios that could occur:
 - When x has no children then, delete x.
 - When x has one child, let x' becomes the child of x.
 - Notice: x' cannot have a child, since subtrees of T can differ in height by at most one :
 - ❖ then replace the contents of x with the contents of x'
 - ❖ then delete x' (a leaf)
- When x has two children,
 - then find x's successor z (which has no left child)
 - then replace x's contents with z's contents, and
 - delete z

In all of the three cases, you will end up removing a leaf.

5. AVL Rotation

1) Left Rotation

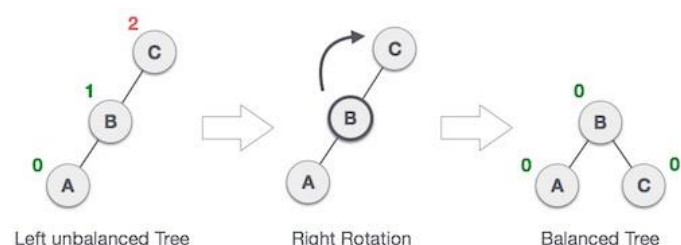
When a node is added into the right subtree of the right subtree, the tree becomes unbalanced, and we do a single left rotation.



As a node is inserted in the right subtree of A's right subtree, node A becomes unbalanced in this example. By making A the left-subtree of B, we may do the left rotation.

2) Right Rotation :

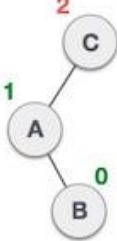
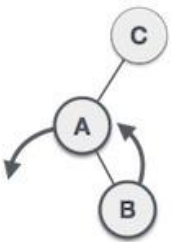
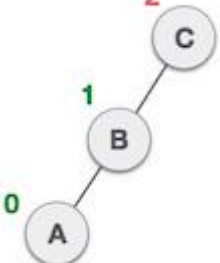
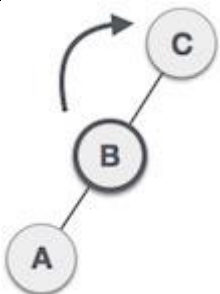
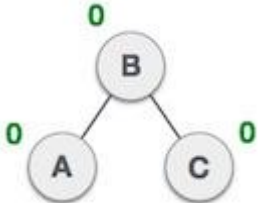
If a node is inserted in the left subtree of the left subtree, the AVL tree may become unstable. The tree must then be rotated correctly.



By completing a right rotation, the unbalanced node becomes the right child of its left child, as shown.

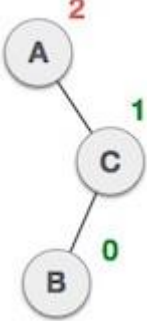
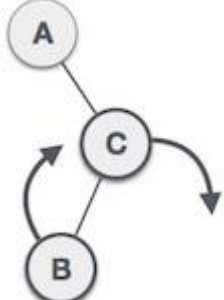
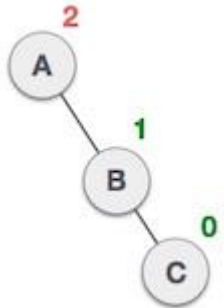
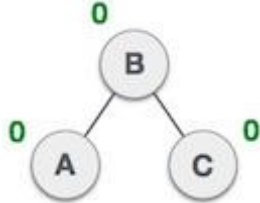
3) Left-Right Rotation

Double rotations are a more complicated variation of previously explained rotations. Take note of each action performed during rotation to better comprehend them. First, let's look at how to perform a Left-Right rotation. A combination of left and right rotations is known as a left-right rotation.

Action	State
A node has been added to the left subtree's right subtree. As a result, C is an unbalanced node. The AVL tree rotates left to right in these cases.	
On the left subtree of C, we first do the left rotation. As a result, A is the left subtree of B.	
Although Node C is still unbalanced, it is now due to the left-subtree of the left-subtree.	
We'll now right-rotate the tree, making B the new subtree's root node. C is now the left subtree of its own left subtree's right subtree.	
The tree is now balanced.	

4) Right-Left Rotation

Right-Left Rotation is the second type of double rotation. It consists of a combination of right and left rotations.

Action	State
A node has been added to the right subtree's left subtree. As a result, A becomes an unbalanced node with a balance factor of 2.	
To begin, we rotate C node to the right, making C the right subtree of its own left subtree B. B is now the right-hand subtree of A.	
Because of the right subtree of its right subtree, Node A is still imbalanced and requires a left rotation.	
The tree is now balanced.	

6. Methodology

6.1 Dataset :

The dataset of the project can be downloaded for https://www.kaggle.com/sudalairajkumar/novel-corona-virus-2019-dataset?select=covid_19_data.csv.

ObservationDate	Province/State	Country/Region	Confirmed	Deaths
02/11/20		Spain	2	
02/11/20	Chicago, IL	US	2	
02/11/20	San Benito, CA	US	2	
02/11/20	Santa Clara, CA	US	2	
02/11/20		Belgium	1	
02/11/20		Cambodia	1	
02/11/20	London, ON	Canada	1	
02/11/20		Finland	1	
02/11/20	Tibet	Mainland China	1	
02/11/20		Nepal	1	
02/11/20		Sri Lanka	1	
02/11/20		Sweden	1	
02/11/20	Boston, MA	US	1	
02/11/20	Los Angeles, CA	US	1	
02/11/20	Madison, WI	US	1	
02/11/20	Orange, CA	US	1	
02/11/20	San Diego County, CA	US	1	
02/11/20	Seattle, WA	US	1	
02/11/20	Tempe, AZ	US	1	
02/12/20	Hubei	Mainland China	33366	10
02/12/20	Guangdong	Mainland China	1219	
02/12/20	Henan	Mainland China	1135	
02/12/20	Zhejiang	Mainland China	1131	
02/12/20	Hunan	Mainland China	946	
02/12/20	Anhui	Mainland China	889	
02/12/20	Jiangxi	Mainland China	844	
02/12/20	Jiangsu	Mainland China	543	
02/12/20	Chongqing	Mainland China	518	
02/12/20	Shandong	Mainland China	497	
02/12/20	Sichuan	Mainland China	436	
02/12/20	Heilongjiang	Mainland China	378	
02/12/20	Beijing	Mainland China	352	
02/12/20	Shanghai	Mainland China	311	
02/12/20	Fujian	Mainland China	272	

6.2 Steps Followed:

We have five files namely TNode.java, Covid19.java, HashMap.java, CovidData.java, and AVLTrees.java.

- TNode.java : The TNode class consists of data such as the hashcode, confirmed rate of infection, recovered rate, death rate, and height. The class also consists of the node to be saved in the left and right node of the AVL Tree. This class consists of a constructor which will initialise the data which has been mentioned earlier.

```
//An AVL tree node
public class TNode {
    public int data, confirmed, dead, recovered, height;
    public String date;
    public TNode right, left;
    public TNode(int e, int confirm, int death, int recover, String da) {
        data = e;
        confirmed = confirm;
        dead = death;
        recovered = recover;
        date = da;
        height = 1;
        right = left = null;
    }
}
```

- Covid19.java : Creates array objects for the AVLTree class which consists of the basic operations such as insertion of a node which is determined by the balance factor and appropriate rotations are being done. The function 'insertall' present in Covid19.java is responsible for creating multiple AVL Trees with nodes as dates. This function calls another function called 'countrylist' which is used to list all countries which are present in the dataset. Since our dataset is a csv file we use "," as a delimiter. Since our dataset consists of repetition of a country because of various provinces in a country, we handle this by appending the data if the country repeats.

```
public void insertall() throws IOException {
    countrylist();
    // this will create multiple AVL trees with nodes as date
    AVLTrees[] arrr = ConTrees(country);
    System.out.println("AVLTrees of countries are created");
    for (int i = 0; i < count; i++) {
        h.put(arrr[i].root.data, country[i]);
    }
}
```

```
public void countrylist() throws FileNotFoundException {
    String filename = "G:/DSA/Covid-Data-Retrieval/covid_19_data.csv";
    Scanner sc = new Scanner(new File(filename));
    // sets the delimiter pattern
    sc.nextLine();
    String[] inter;
    while (sc.hasNext()) // returns a boolean value
    {
        String data;
        data = sc.nextLine();
        String data1[] = null;
        // since it's a csv file, delimiter is ,
        data1 = data.split(",");
        inter = data1[0].split("/|-");
        String datet = inter[0] + inter[1] + inter[2];
        // this will store all the countries in the country array
        if (Arrays.asList(country).contains(data1[1]) == false) {
            country[count++] = data1[1];
        }
        if (Arrays.asList(datee).contains(datet) == false) {
            datee[count1++] = datet;
        }
    }
    sc.close();
}
```

The function ‘ConTrees’ will read the data consisting of the countries and assigns the hashcode values for countries and dates. A Hashcode value is a fixed length numeric value that identifies data uniquely.

```
public AVLTrees[] ConTrees(String country[]) throws IOException {
    String line;
    // inserting data in AVLtrees of all countries using loop
    for (int i = 0; i < count; i++) {
        String filename = "G:/DSA/Covid-Data-Retrieval/covid_19_data.csv";
        BufferedReader br1 = new BufferedReader(new FileReader(filename));
        br1.readLine();
        arr[i] = new AVLTrees();
        while ((line = br1.readLine()) != null) // returns a boolean value
        {
            String data[] = new String[5];
            data = line.split(",");
            if (country[i].equals(data[1])) {
                String inter[] = data[0].split("/|-");
                String inter1 = inter[0] + inter[1] + inter[2];
                // inserting addition of hashcode of date and country as node in the trees
                arr[i].insert(inter1.hashCode() + data[1].hashCode(), Integer.parseInt(data[2]),
                    Integer.parseInt(data[3]), Integer.parseInt(data[4]), data[0]);
            }
        }
    }
    br1.close();
}
```

- AVLTree.java : The function ‘insert’ present in the class AVLTree will call upon the function ‘insert_rec’ and store it as variable ‘root’ which will consist of our data such as hashcode, confirmed rate of infection, recovered rate, and death rate with respect to different countries.

```
public void insert(int e, int confirm, int death, int recover, String date) {
    root = insert_rec(e, confirm, death, recover, date, root);
}
```

The insertion operation is based on value of the hashcode and balanced accordingly. The function ‘insert_rec’ will check whether the root node exists. If the root node exists the insertion process will be done, if not the root node will be created.

```
// Insert into BST
private TNode insert_rec(int e, int confirm, int death, int recover, String date, TNode r) {
    /*
     * 1. Perform the normal BST insertion returning a new node with the date,
     * confirmed, dead and recovered cases...
     */
    if (r == null)
        return (new TNode(e, confirm, death, recover, date));
    else {
        // Key is smaller than root's key
        if (e < r.data) {
            r.left = insert_rec(e, confirm, death, recover, date, r.left);
        }
        // Key is greater than root's key
        else if (e > r.data) {
            r.right = insert_rec(e, confirm, death, recover, date, r.right);
        }
        /*
         * if the current data is equal to the node in the tree ,confirmed ,death and
         * recovered case of that node will be added with current one
         */
        else if (e == r.data) {
            r.confirmed = r.confirmed + confirm;
            r.dead = r.dead + death;
            r.recovered = r.recovered + recover;
        } else
            return r;
    }
}
```

```
// Update height of current node...
// This node is compromised by an insertion ...
r.height = 1 + max(height(r.left), height(r.right));
// Check is the node is in balance ...
// that is ... the height difference between the left and right is 1
int balance = getbalance(r);
// Left Left case ...
if (balance > 1 && e < r.left.data)
    return rightRotate(r);
// Right Right case ...
if (balance < -1 && e > r.right.data)
    return leftRotate(r);
// Left Right case ...
if (balance > 1 && e > r.left.data) {
    r.left = leftRotate(r.left);
    return rightRotate(r);
}
// Right Left case ...
if (balance < -1 && e < r.right.data) {
    r.right = rightRotate(r.right);
    return leftRotate(r);
}
return r;
}
```

7. Results

We are creating an AVL Tree to store the data of COVID – 19.

The below snapshot shows time taken to create the AVL Tree for the given dataset :

```
CovidData [Java Application] C:\Users\ss261\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full\
AVLTrees of countries are created
Total creation time: 10.237
If details are required for all countries/date, enter 'All'
If not, enter the name of a specific country:
```

Now we will retrieve the details of COVID – 19 by entering the country name and specific date. This gives us the data of confirmed cases, deaths in country, and recovered cases of the above mentioned country. Also it shows the time taken to retrieve the data:

```
<terminated> CovidData [Java Application] C:\Users\ss261\p2\pool\plugins\org.eclipse.justj.openjdk.hotspot.jre.full\
AVLTrees of countries are created
Total creation time: 10.237
If details are required for all countries/date, enter 'All'
If not, enter the name of a specific country:
US
Enter the date for which details are required in dd-mm-yy:
10-06-2020
The number of confirmed cases in US on 10-06-2020 are 2000464
The number of deaths in US on 10-06-2020 are 112924
The number of recovered cases in US on 10-06-2020 are 533504
Total execution time: 0.001
```

8. References

https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.html

<https://www.javatpoint.com/binary-search-tree-vs-avl-tree>

<https://www.w3schools.in/data-structures-tutorial/avl-trees/>

<https://sci-hub.hkvisa.net/10.1145/800197.806043>