Implementation of Dictionaries using

AVL Tree

Kanimozhi Balaraman
Indiana State University
Terre Haute IN, USA

kbalaraman@cs.indstate.edu

December 11, 2011
Contents

1 Statement of Problem .................................................. 3

2 History ........................................................................... 4

3 Algorithm ........................................................................ 4
   3.1 Example of an AVL Tree .............................................. 5
   3.2 Height of an AVL tree .................................................. 5
      3.2.1 Proposition .......................................................... 5
      3.2.2 Justification .......................................................... 5
   3.3 Properties of an AVL tree ............................................. 6
      3.3.1 Property : 1 ........................................................... 6
      3.3.2 Property : 2 ........................................................... 6
   3.4 Rotation ...................................................................... 7
      3.4.1 Left Rotation ......................................................... 7
      3.4.2 Right Rotation ....................................................... 9
      3.4.3 Right-Left Rotation ............................................... 10
      3.4.4 Left-Right Rotation .............................................. 11

4 AVL Tree - Operations ...................................................... 12
   4.1 Insert Operation ......................................................... 13
      4.1.1 Steps: Insert Operation ......................................... 13
      4.1.2 Example ............................................................... 14
   4.2 Search Operation ....................................................... 15
   4.3 Delete Operation ....................................................... 17
      4.3.1 Example ............................................................... 17

5 Time Complexity ............................................................ 19
Abstract

The paper is to implement Sorted Dictionaries using AVL Tree.

Dictionaries map words to their definition. So, A dictionary can be implemented by mapping two sets A and B where the input member A is mapped to the output member B. This mapping can be done by a searching algorithm. AVL tree is used as the searching algorithm in this project. The searching algorithm matches the output member B for the given input memeber A.

AVL Tree is a self balancing binary search tree. AVL trees take $O(\log n)$ time for insert, delete and search operations. The tree must be rebalanced after insert and delete operations.

This project supports insert, delete and search operations of AVL tree. In insert operation the value 'A' is inserted for the given key 'K'. In search operation the value 'A' is retrieved for the given key 'K'. In delete operation the given value 'A' is removed from the dictionary.

1 Statement of Problem

In the implementation of Dictionary using Binary Search tree one of the problem is that they become badly unbalanced. So a balanced Binary Search tree like AVL tree can be used to implement Dictionary. AVL trees are always balanced. In AVL trees the heights of two child subtrees of any node differs by atmost one.

In worst case Binary Search tree takes $O(n)$ time for insert, search and delete operations. To avoid $O(n)$ time in worstcase for these operations a balanced binary tree called AVL tree can be used to implement the dictionary. AVL tree takes $O(\log n)$ time for insert, search and delete operations in worst case.

The height of Binary Search tree is $O(n)$ in worst case. But the height of AVL tree is $O(\log n)$ in both tasks.

\[ A - \text{Node Value} \]
\[ K - \text{Key Value} \]
best and worst cases where $n$ is the number of nodes in the tree.

2 History

AVL trees are invented by two Russian mathematicians Adel’son-Vel’skii and Landis in 1962. AVL tree was published in the paper named ”An algorithm for the organisation information”. AVL tree is a balanced binary search tree which emploees rotation to maintain balance. AVL tree was the first data structure to be invented.

3 Algorithm

In this project AVL tree is used to implement Dictionary. An AVL tree is a Balanced Binary Search tree that is either empty or has the following properties.

1. The height of the left and right subtrees of a node differ by atmost 1. The height of an AVL tree is always $O(\log n)$.

2. The left and right subtrees of a node are AVL trees.
3.1 Example of an AVL Tree

3.2 Height of an AVL tree

3.2.1 Proposition

The height of the tree with 'n' nodes is $O(\log n)$.

3.2.2 Justification

Let $n(h)$ be the minimum possible number of nodes for an AVL tree of height 'h'.

We can find that $n(1) = 1$ and $n(2) = 2$.

For $h \geq 3$, an AVL tree of height 'h' contains the following:

1. root node,

* 2. one AVL subtree of height $h-1$
3. the other AVL subtree of height h-1 or h-2

Therefore \( n(h) = 1 + n(h-1) + n(h-2) \)

Knowing \( n(h-1) \geq n(h-2) \), we get

So we can say that,

\[
\begin{align*}
n(h) &= 1 + n(h-1) + n(h-2) > 2n(h-2) \\
n(h) &> 2n(h-2) \\
&> 4n(h-4) \\
&> 8n(h-6) \\
&\ldots \\
&> 2^i n(h - 2i) \text{ where } i \text{ is the number of steps}
\end{align*}
\]

When \( i = h/2 - 1 \) we get,

\[
n(h) > 2^{h/2-1} n(2) = 2^{h/2}
\]

On taking logarithms, \( h < 2\log n(h) \).

So \( h \) will be less than \( n \), where \( n \) is the number of nodes in the tree.

Thus the height of an AVL tree is \( O(\log n) \).

### 3.3 Properties of an AVL tree

#### 3.3.1 Property : 1

If the AVL tree has ‘n’ nodes and the closest leaf to the root is at level ‘k’. Then the height of the tree can be atmost \( 2k-1 \).

#### 3.3.2 Property : 2

If the leaf node closest to the root node is at level k, then all the nodes from root to level k-2 have two children.
3.4 Rotation

The AVL tree may become unbalanced after insert and delete operation. Rotation is the basic mechanism that rebalance the unbalanced tree.

The rotation is an adjustment to the tree, around an item, that maintains the required ordering of items. A rotation takes a constant time. Nodes that are not in the subtree of the item rotated about are unaffected by the rotation.

There are four kinds of rotation.

1. Single Rotation
   (a) Left Rotation
   (b) Right Rotation

2. Double Rotation
   (a) Left-Right Rotation
   (b) Right-Left Rotation

3.4.1 Left Rotation

Left Rotation operation is performed when the tree is right heavy. So the tree is rotated left side to rebalance the tree.

ALGORITHM

Node RotateLeft(node x)

*{
Node y = x.right;
x.right = y.left;
y.left = x;
return y;
}

Consider a tree with nodes of the following structure.

- A - Root node.
- B - A’s Right child node.
- C - B’s Right child node.

After rotation the resulting tree structure will be as following.

- B becomes the root node.
- A becomes the B’s Left child node.
- C will be B’s Right Child node.
3.4.2 Right Rotation

In Right Rotation operation the tree is rotated right side to rebalance the tree. A right rotation is mirror to left rotation operation.

**ALGORITHM**

Node RotateRight(node x) {

Node y = x.left;
node x.left = y.right;
y.right = x;
return y;
}

Consider a tree with nodes of the following structure.

- A - Root node
- B - A's Left child node.
- C - B’s Left child node.

![Tree Structure](image)

After rotation the resulting tree structure will be as following.

* B becomes the root node.
• A becomes the B’s Right child node.
• C will be B’s Left Child node.

3.4.3 Right-Left Rotation

In this case two rotation need to be performed inorder to balance the tree. First the tree is rotated right side and then to the left side. Consider a unbalanced AVL tree of following structure.

• A - Root node
• B - A’s Right child node.
• C - B’s Left child node.

The first rotation performed on B and C nodes in the tree.

• A - Root node
• C - A’s Right child node.
• B - will become C’s Right child node.

*
A Left-Right rotation is the mirror image of Right-Left Rotation operation. In this case two rotation need to be performed inorder to balance the tree. The tree is first rotated on left side and then on right side.

Consider a imbalanced tree of following structure to perform Left-Right rotation.

- A - Root node
- B - A’s Left child node.
- C - B’s Right child node.
After left rotation performed on B and C nodes, the tree structure looks like the following.

- A - Root node
- C - will become the left child node of A
- B - will become the left child node of C.

Now the tree is rotated right side inorder to obtain the balanced AVL tree structure.

- C - Root node
- B - will become the left child node of C
- A - will become the Right child node of C.

4 AVL Tree - Operations

This project supports three basic operations of AVL tree.

1. Insert
2. Search
3. Delete
4.1 Insert Operation

A new node 'n' can be inserted if the node value is not already present in the tree.

Inserting a new node 'n' into an AVL tree changes the height of some of the nodes in the tree. On inserting a new node 'n', the only nodes whose height can be increased are the ancestors of new node 'n'. This is because, only the subtree of the ancestors of new node 'n' changes as the result of insertion process.

As the consequences of inserting a new node 'n', some of the ancestors of new node 'n' would have a height imbalance resulting in an unbalanced AVL tree. Once the tree is found to be unbalanced, travel up the tree and find the first unbalanced node 'x'. Perform appropriate rotate operation on 'x' with its child node 'y' and grandchild node 'z' inorder to rebalance the tree.

4.1.1 Steps: Insert Operation

• Insert the node 'n' in the appropriate position.

• Travel up the tree towards the root node 'r' and check for height imbalance of ancestors of newly added node

• If all nodes till 'r' are balanced

• then

• The newly added node did not cause imbalance

• else

• Find the first imbalanced node 'x' in the path

---

3n - new node that is inserted into an AVL tree
4x - First unbalanced node encountered in the path when travelling from 'n' to root node.
5y - child node of x in the path from 'n' to 'x'.
6z - grandchild of x in the path from 'n' to 'x'.
• Find 'y' - the child node of 'x' that lies in the path from 'n' to 'r'.
• Find 'z' - the child node of 'y' that lies in the path from 'n' to 'r'.
• Rotate 'x' with 'y' and 'z' using appropriate rotation operation.
• Now the tree is balanced

4.1.2 Example

Insert - 55

- Insert 55 in appropriate place in the tree.
- Travel up the tree from 55 to 45 (root node) and check all ancestors of 55 for height imbalance.
- Here node 78 has an height imbalance, as the difference in the height of its right and left subtree is 2.
- Perform rotation on the node 78, its child node 50 and its grandchild node 63 that lies in the path from 55 to 78.
• Left-Right Rotation is performed to make the tree balanced.

• Left Rotation is performed on node 63 and node 50.

• Now node 63 becomes the parent of node 50 and leftchild of 63 becomes the rightchild of node 50.

• Perform Right Rotation on node 63 and 78.

• Now 63 becomes the right child of node 45 and node 78 becomes the rightchild of 63.

• Now the tree is balanced.

• In insert operation only one rotation is sufficient to balance the tree.

In Insertion operation, after performing the rotation once no further rotations required. Because the height of the subtree remains the same.

4.2 Search Operation

The Search operation can be performed with the key value that need to be searched in the given AVL tree. The search operation returns a node from a tree if the node value matches with the key value. If the
key value does not match with any node value in the tree no value is returned.

Key Value to be searched - 23

4.2.1 Search Operation - Algorithm

1. if nodeValue = key
2. return key found
3. else if {
4. if(key < nodeValue)
5. nodeValue = nodeValue.LeftChild
6. Repeat from Step 1
7. else if(key > nodeValue)
8. nodeValue = nodeValue.RightChild
9. Repeat from Step 1
10. }
11. if the key value does not match with any nodeValue

*
13. return key value not found
14. }

4.3 Delete Operation

Deletion is more complicated than insertion. The deletion of a node from a tree may decrease the height of the tree which may lead to unbalanced tree structure. So appropriate rotation operation is performed to rebalance the tree.

The node to be deleted can be either of the three following cases.
1. Leaf Node - Delete the leaf node.
2. Node with one children - Delete the node and replace it with the only child.
3. Node with two children - Replace the node with left-most node in the right subtree.

After deleting a node 'n' from the AVL tree, the height of the ancestors of 'n' could be decreased. So the ancestors of 'n' should be checked for height balance.

- Let 'n' be the deleted node.
- Let 'c' be the first unbalanced node encountered while travelling up the tree from 'n' to the root node.
- Let 'b' be the child of 'c' with greater height.
- Let 'a' be the child of 'b' with greater height.

Perform rotations to restore balance at the subtree rooted at 'c'. The reconstructing may cause imbalance of another node higher in the tree which are the ancestors of 'n'. So continue checking for node height balance until the root of the tree is reached. In deletion operation more than one rotation maybe performed.
4.3.1 Example

```
In order to delete node 41 from the tree, replace node 41 with node 42 as it is the leftmost node in the right subtree.

Travel up the tree from the place where the node 42 was before deletion and check all the ancestors for imbalance.

As we travel up the tree node 43 was found to be height imbalanced.

Rotate 43 with its childnode 46 and its grandchild node 47.

After balancing node 43, check node 42 for height imbalance. It is found that the tree is balanced already.

As node 42 is the root node and all the nodes in the path between the root node and the deleted node is balanced, the deletion process is complete.
```
5 Time Complexity

AVL tree takes $O(\log n)$ time for insert, search and delete operations in both average and worst case.

References


