Subject: Data Structures and algorithms Lab

Assignment Number: 08

Name: Shubham Chemate

Roll Number: 21118

Problem Statement:

A Dictionary stores keywords and its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

Learning Objectives:

- To understand concept of height balanced tree data structure
- To understand procedure of creating height balanced tree

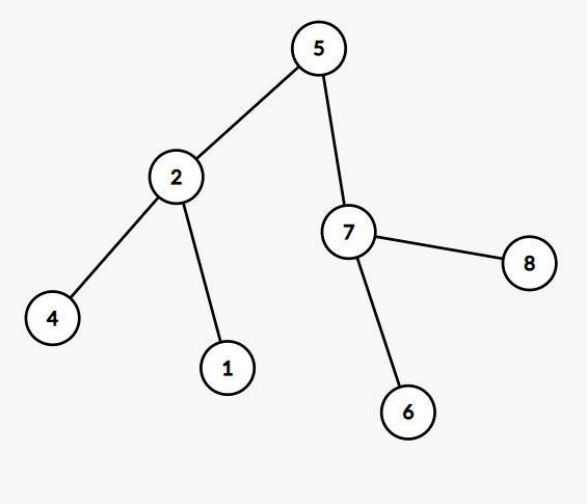
Learning Outcomes:

- Define class AVL for using object-oriented features
- Analyse working of various operations on AVL trees

Binary Search Trees:

In computer science, a binary search tree (BST), also called an ordered or sorted binary tree, is a rooted binary tree whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree.

The typical binary search tree is shown below:



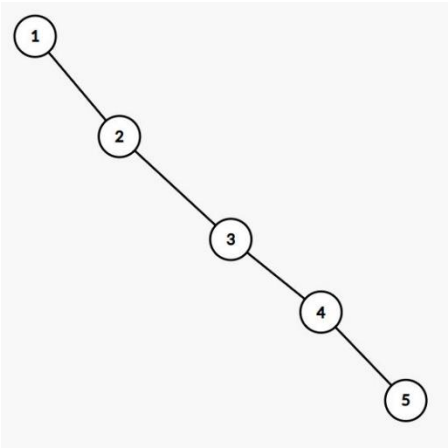
The time complexity of general operations of BST is:

Operation	Worst Case Time
Search	O(h)
Insert	O(h)
Delete	O(h)

Note that height of tree plays very important role in case of binary search tree.

Problem with General BST's:

Consider inserting keys 1, 2, 3, 4, 5 in general binary search tree. Each insertion will take O(h) time. Here height will be O(n) after any insertion. So, insertion in the BST will be same as insertion in any sorted linked list and will not provide any benefit in terms of time complexity. Search and delete operations will also require O(n) time which will be same as any other data structure.



Self-Balancing Binary Search Trees:

As discussed earlier, height of tree plays very important role in deciding complexity of general operations. Self-balancing binary search trees are the trees which keep their height as minimum as possible (O(logn)) after each operation.

The data structures which implements these trees are:

- 2–3 tree
- AA tree
- AVL tree
- B-tree
- Red–black tree
- Scapegoat tree
- Splay tree
- Tango tree
- Treap
- Weight-balanced tree

The time complexity of general operations of self-balancing BST is:

Operation	Worst Case Time
Search	O(log(n))
Insert	O(log(n))
Delete	O(log(n))

AVL Tree as self-balancing binary search tree:

- In computer science, an AVL tree (named after inventors Adelson-Velsky and Landis) is a self-balancing binary search tree. It was the first such data structure to be invented.
- In an AVL tree, the heights of the two child subtrees of any node differ by at most one; if at any time they differ by more than one, rebalancing is done to restore this property.
- Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.
- For lookup-intensive applications, AVL trees are faster than red–black trees because they are more strictly balanced.

Balance factor:

- In a binary tree the balance factor of a node X is defined to be the height difference.

  BalanceFactor(X) = Height(LeftSubtree(X)) – Height(RightSubtree(X))

- A binary tree is defined to be an AVL tree if the invariant

  BalanceFactor(X) lies in range [-1, 1] for every node X.

- A node X with BalanceFactor(X) < 0 is called "left-heavy", one BalanceFactor(X) > 0 is called "right-heavy", and one with BalanceFactor(X) = 0 is sometimes simply called "balanced".

Operations on AVL tree:

- Display Operation:
  - On performing inorder traversal on AVL trees we can get values of keys in sorted order.
  - Other traversals are also possible in AVL trees, just like normal binary search trees.
- Searching for key
  - Searching for a specific key in an AVL tree can be done the same way as that of any balanced or unbalanced binary search tree.
  - The number of comparisons required for successful search is limited by the height h and for unsuccessful search is very close to h, so both are in O (log n).
- Insertion Operation
  - When inserting a node into an AVL tree, you initially follow the same process as inserting into a Binary Search Tree.
  - After this insertion if a tree becomes unbalanced, only ancestors of the newly inserted node are unbalanced. This is because only those nodes have their sub-trees altered.[12] So it is necessary to check each of the node's ancestors for consistency with the invariants of AVL trees: this is called "retracing". This is achieved by considering the balance factor of each node.
  - Since with a single insertion the height of an AVL subtree cannot increase by more than one, the temporary balance factor of a node after an insertion will be in the range [–2, +2]. For each node checked, if the temporary balance factor remains in the range from –1 to +1 then only an update of the balance factor and no rotation is necessary. However, if the temporary balance factor becomes less than –1 or greater than +1, the subtree rooted at this node is AVL unbalanced, and a rotation is needed.

- Pseudocode for insertion operation:

Algorithm insertAVL(k, e, T):
Input: A key-element pair, (k, e), and an AVL tree, T
Output: An update of T to now contain the item (k, e)
$v \leftarrow$ IterativeTreeSearch(k, T)
if v is not an external node **then**
return "An item with key k is already in T"
Expand v into an internal node with two external-node children
v.key $\leftarrow k$
v.element $\leftarrow e$
v.height $\leftarrow 1$
rebalanceAVL(v, T)

- Delete Operation
  - Since with a single deletion the height of an AVL subtree cannot decrease by more than one, the temporary balance factor of a node will be in the range from −2 to +2.
  - If the balance factor remains in the range from −1 to +1 it can be adjusted in accord with the AVL rules.
  - If it becomes ±2 then the subtree is unbalanced and needs to be rotated. (Unlike insertion where a rotation always balances the tree, after delete, there may be BF(Z) ≠ 0.
  - Pseudo-code for deletion:

Algorithm removeAVL(k, T):
Input: A key, k, and an AVL tree, T
Output: An update of T to now have an item (k, e) removed
$v \leftarrow$ IterativeTreeSearch(k, T)
if v is an external node **then**
return "There is no item with key k in T"
if v has no external-node child **then**
Let u be the node in T with key nearest to k
Move u's key-value pair to v
$v \leftarrow u$
Let w be v's smallest-height child
Remove w and v from T, replacing v with w's sibling, z
rebalanceAVL(z, T)

- Rebalancing Operations in AVL-tree / Rotation Techniques:
  - Right Rotation: A single rotation applied when a node is inserted in the left subtree of a left subtree. In the given example, node C now has a balance factor of 2 after the insertion of node A. By rotating the tree right, node B becomes the root resulting in a balanced tree.

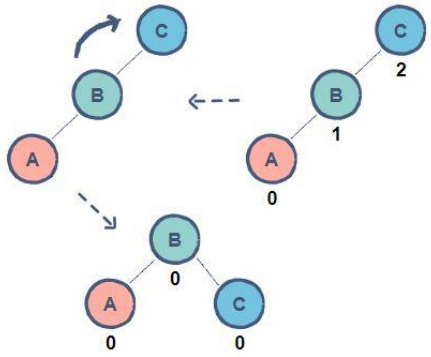


Fig: Right Rotation in AVL tree

- Left Rotation: A single rotation applied when a node is inserted in the right subtree of a right subtree. In the given example, node A has a balance factor of 2 after the insertion of node C. By rotating the tree left, node B becomes the root resulting in a balanced tree.

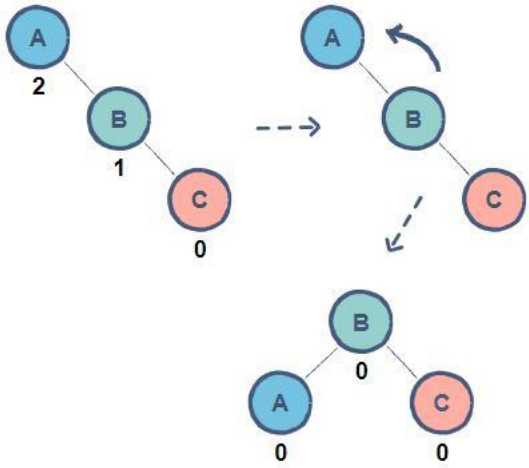


Fig: Left Rotation in AVL tree

- Left-Right Rotation: A double rotation in which a left rotation is followed by a right rotation. In the given example, node B is causing an imbalance resulting in node C to have a balance factor of 2. As node B is inserted in the right subtree of node A, a left rotation needs to be applied. However, a single rotation will not give us the required results. Now, all we have to do is apply the right rotation as shown before to achieve a balanced tree.

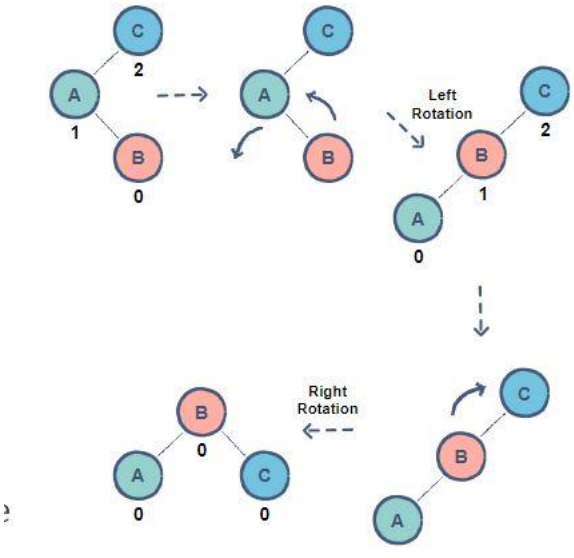


Fig: Left-Right Rotation in AVL tree

- Right-Left Rotation: A double rotation in which a right rotation is followed by a left rotation. In the given example, node B is causing an imbalance resulting in node A to have a balance factor of 2. As node B is inserted in the left subtree of node C, a right rotation needs to be applied. However, just as before, a single rotation will not give us the required results. Now, by applying the left rotation as shown before, we can achieve a balanced tree.

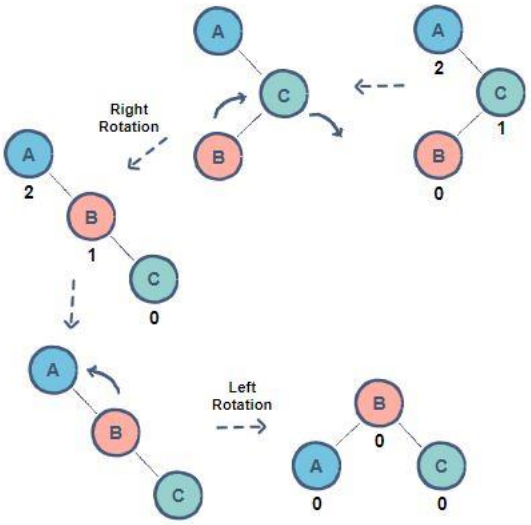


Fig: Right-Left rotation in AVL Tree

Pseudocode:

Algorithm rebalanceAVL(v, T):
Input: A node, v, where an imbalance may have occurred in an AVL tree, T
Output: An update of T to now be balanced
v.height $\leftarrow 1 + \max\{v.\text{leftChild}().\text{height}, v.\text{rightChild}().\text{height}\}$
while v is not the root of T **do**
$v \leftarrow v.\text{parent}()$
if $|v.\text{leftChild}().\text{height} - v.\text{rightChild}().\text{height}| > 1$ **then**
Let y be the tallest child of v and let x be the tallest child of y
$v \leftarrow \text{restructure}(x)$ // trinode restructure operation
$v.\text{height} \leftarrow 1 + \max\{v.\text{leftChild}().\text{height}, v.\text{rightChild}().\text{height}\}$

Applications of AVL trees:

- AVL trees are mostly used for in-memory sorts of sets and dictionaries.
- AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
- It is used in applications that require improved searching apart from the database applications.

Conclusion:

In this study assignment, I have learned about self-balancing binary trees and how we can perform dictionary operations on them. I learned about AVL trees, their operations and time complexities of operations.

Resources:

- https://en.wikipedia.org/wiki/AVL_tree
- https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree
- https://en.wikipedia.org/wiki/Binary_search_tree
- Algorithm Design and Application book by, by M. T. Goodrich and R. Tamassia, Wiley, 2015