GROUP D

Assignment 08

Problem Statement:

Given sequence k = k1 <k2 < … < kn of n sorted keys, with a search probability pi for each key ki. Build the Binary search tree that has the least search cost given the access probability for each key?

Objectives:

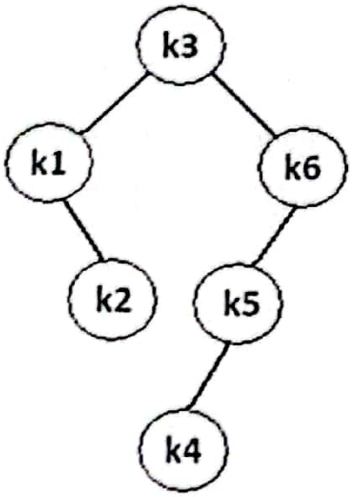
1. To understand concept of OBST.
2. To understand concept & features like extended binary search tree.

Theory:

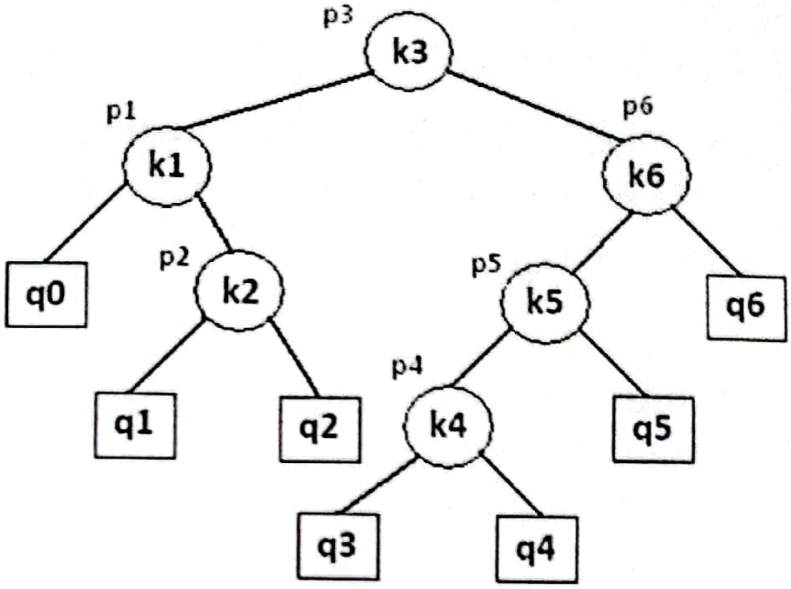
An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum.

For the purpose of a better presentation of optimal binary search trees, we will consider "extended binary search trees", which have the keys stored at their internal nodes. Suppose "n" keys k1, k2, … k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn.

An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:



Binary search tree



Extended binary search tree

In the extended tree:

- The squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree;
- The round nodes represent internal nodes; these are the actual keys stored in the tree;
- Assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.
- If the user searches a particular key in the tree, 2 cases can occur:
- 1 – the key is found, so the corresponding weight „p" is incremented;
- 2 – the key is not found, so the corresponding „q" value is incremented.

GENERALIZATION:

The terminal node in the extended tree that is the left successor of k1 can be interpreted as representing all key values that are not stored and are less than k1. Similarly, the terminal node in the extended tree that is the right successor of kn, represents all key values not stored in the tree that are greater than kn. The terminal node that is successes between ki and ki-1 in an inorder traversal represent all key values not stored that lie between ki and ki - 1.

ALGORITHMS

We have the following procedure for determining R(i, j) and C(i, j) with 0 <= i <= j <= n:

```
PROCEDURE COMPUTE_ROOT(n, p, q; R, C)
begin
for i = 0 to n do
C (i, i) ←0
W (i, i) ←q(i)
for m = 0 to n do
for i = 0 to (n – m) do
j ←i + m
W (i, j) ←W (i, j – 1) + p (j) + q (j)
*find C (i, j) and R (i, j) which minimize the
tree cost
end
The following function builds an optimal binary sea
rch tree
FUNCTION CONSTRUCT(R, i, j)
begin
*build a new internal node N labeled (i, j)
k ←R (i, j)
f i = k then
*build a new leaf node N‘‘ labeled (i, i)
else
*N‘‘ ←CONSTRUCT(R, i, k)
*N‘‘ is the left child of node N
if k = (j – 1) then
*build a new leaf node N‘‘‘‘ labeled (j, j)
else
*N‘‘‘‘ ←CONSTRUCT(R, k + 1, j)
*N‘‘‘‘ is the right child of node N
return N
end
```

Output: Create binary search tree having optimal searching cost.

Conclusion: This program gives us the knowledge OBST, Extended binary search tree.

GROUP D

Assignment 09

Problem Definition:

A Dictionary stores keywords & 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.

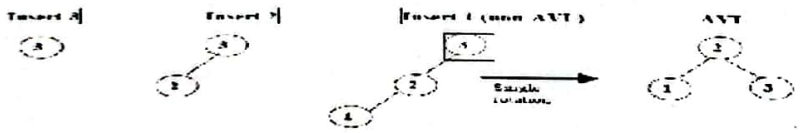
Prerequisite:

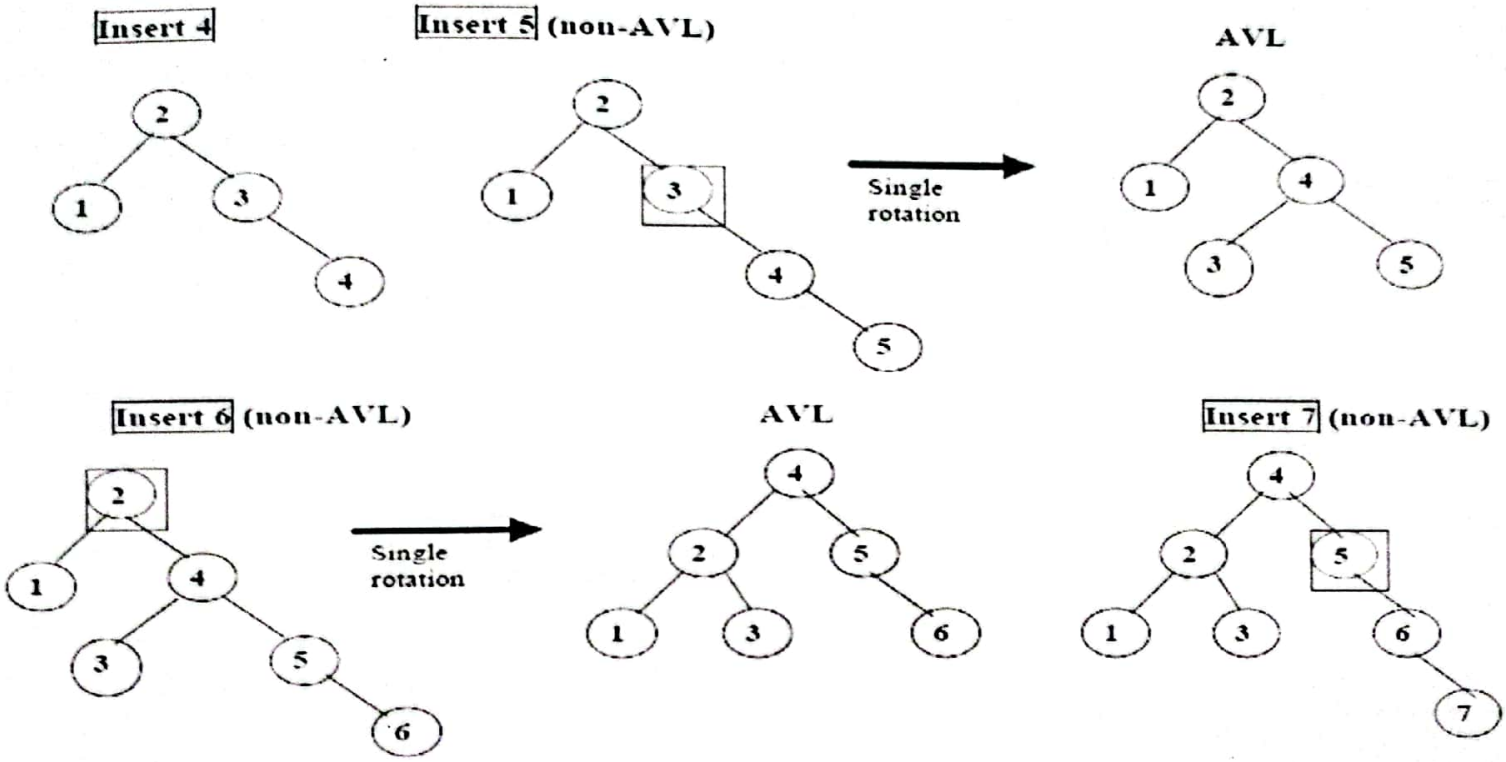
1. Basic concepts of thread
2. Concepts of in-Order & pre-Order traversals.

Theory :

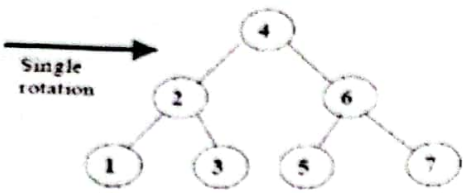
An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if Its balance factor is 0, 1, -1.

AVL (Adelson- Velskii and Landis) Tree: A balance binary search tree. The best search time, that is O (log N) search times. An AVL tree is defined to be a well-balanced binary search tree in which each of its nodes has the AVL property. The AVL property is that the heights of the left and right sub-trees of a node are either equal or if they differ only by 1

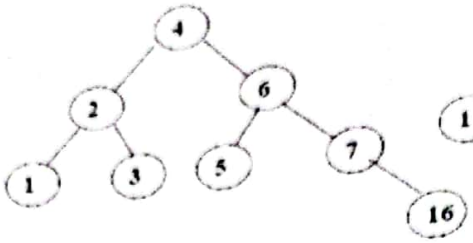



Insert 4
Insert 5 (non-AVL)
AVL
Single rotation
Insert 6 (non-AVL)
AVL
Single rotation
Insert 7 (non-AVL)

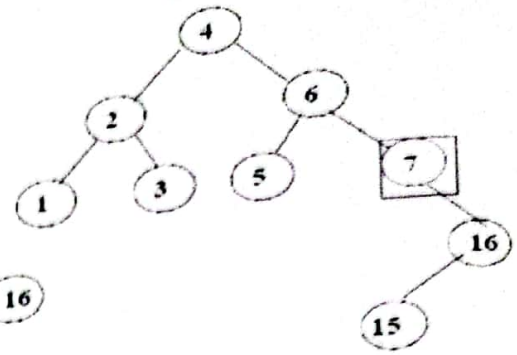
AVL



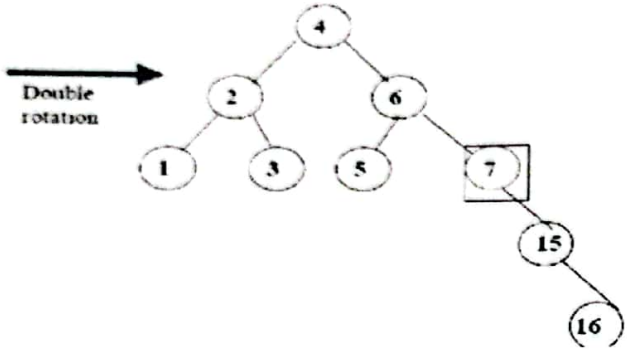
Insert 16



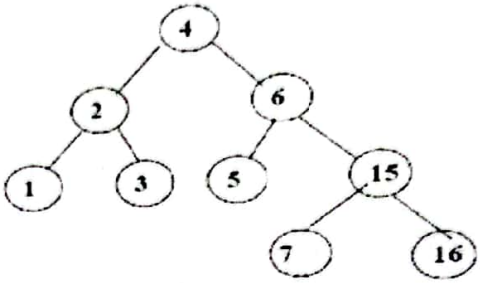
Insert 15 (non-AVL)



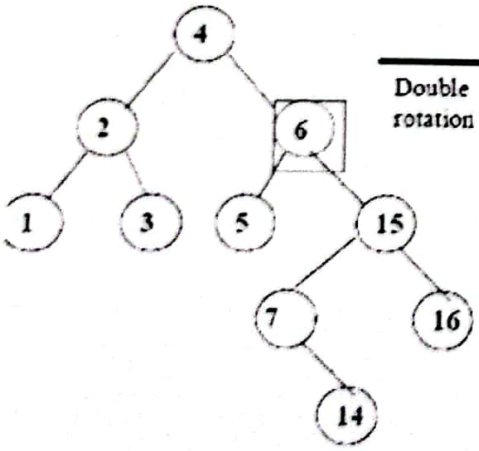
Step 1: Rotate child and grandchild



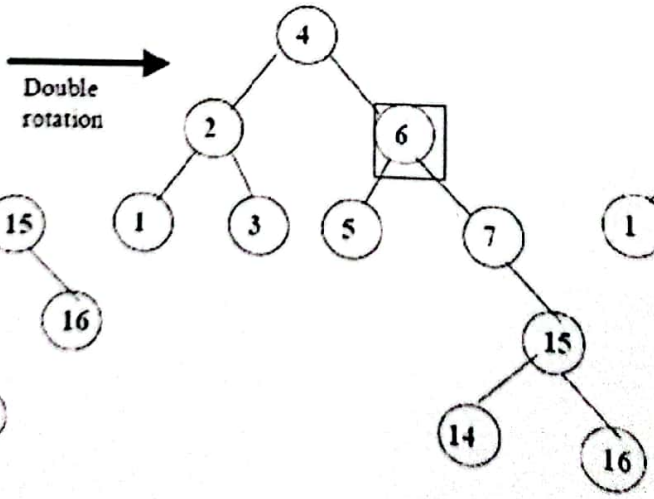
Step 2: Rotate node and new child (AVL)



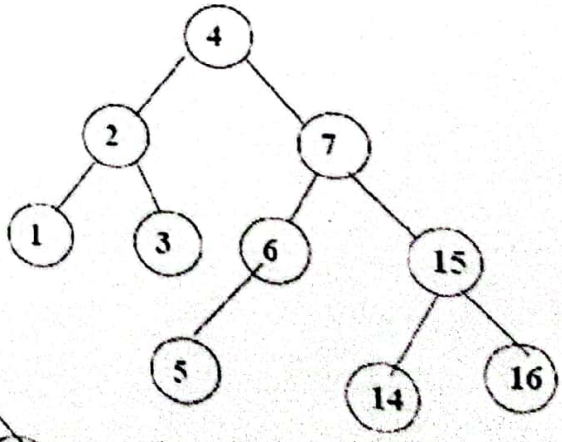
Insert 14 (non-AVL)



Step 1: Rotate child and grandchild



Step 2: Rotate node and new child (AVL)



An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1. An example of an AVL tree where the heights are shown next to the nodes

AVL tree is a height balance tree.

- The height of the right sub tree and height of the left sub tree for any node cannot differ by more than one.
- This process is usually done through rotation.

Operation on AVL Tree:-

Insertion of a node:-

- Inserting in AVL tree is same as in binary search tree. Here also we will search for the position where the new node is to be inserted and then insert the node.
- To restore the property of AVL tree we should convert the tree in such a way that, the new converted tree is balance tree i.e. the balance factor of each node should be -1, 0, 1.
- The new converted node should be a binary search tree with in order traversals same as that of original tree.
- The outline of the procedure to insert of a node is as- insert node to its proper place follow the same process as in binary search tree.
- Calculate the balance factor of the entire path starting from the inserted node to the root node.
- If the tree become unbalance after insertion then there is need to convert the above tree by performing rotations.

Deletion of a node at any position:-

- Read the node from the user which he wants to delete.
- Find out the node position and delete the node.
- Check for the balance factor.
- If tree is imbalance then perform the rotations.
- Stop.

Algorithm:

Insertion To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)). 1) Left Rotation 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

y x
/ \ Right Rotation / \
x T3 – – – – – – – > T1
y / \ <---------- / \
T1 T2 Left Rotation T2 T3

Keys in both of the above trees follow the following order

keys(T1) < key(x) < keys(T2) < key(y) < keys(T3) So BST property is not violated anywhere.

a) Left Left Case

T1, T2, T3 and T4 are subtrees. z y

/ \ / \

y T4 Right Rotate (z) x z

/ \ - - - - - - - - - -> / \ /

\ x T3 T1 T2 T3 T4

/ \

T1 T2

b) Left Right Case z z x

/ \ / \ / \

y T4 Left Rotate (y) x T4 Right Rotate(z) y

z / \ - - - - - - - - - -> / \ ----------> / \ / \

T1 x y T3 T1 T2 T3

T4 / \ / \

T2 T3 T1 T2

c) Right Right Case z y

/ \ / \

T1 y Left Rotate(z) z

x / \ ----------> / \ / \

T2 x T1 T2 T3 T4

/ \

T3 T4

d) Right Left Case

z z x

/ \ / \ / \

T1 y Right Rotate (y) T1 x Left Rotate(z) z x

/ \ - - - - - - - - - > / \ - - - - - - - - - > / \ /
\ x T4 T2 y T1 T2 T3 T4
/ \ / \
T2 T3 T3 T4

deletion. To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property (keys(left) < key(root) < keys(right)). 1) Left Rotation 2) Right Rotation

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

y x
/ \ Right Rotation / \
x T3 - - - - - - - - > T1 y

/ \ < - - - - - - - / \
T1 T2 Left Rotation T2 T3

Keys in both of the above trees follow the following order keys(T1) < key(x) < keys(T2) < key(y) < keys(T3)
So BST property is not violated anywhere.

Let w be the node to be deleted **1)** Perform standard BST delete for w. **2)** Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y. Note that the definitions of x and y are different from insertion here. **3)** Re-balance the tree by performing appropriate rotations on the subtree rooted with z. There can be 4 possible cases that needs to be handled as x, y and z can be arranged in 4 ways. Following are the possible 4 arrangements: a) y is left child of z and x is left child of y (Left Left Case) b) y is left child of z and x is right child of y (Left Right Case) c) y is right child of z and x is right child of y (Right Right Case) d) y is right child of z and x is left child of y (Right Left Case)

Like insertion, following are the operations to be performed in above mentioned 4 cases. Note that, unlike insertion, fixing the node z won"t fix the complete AVL tree. After fixing z, we may have to fix ancestors of z as well

a) Left Left Case

T1, T2, T3 and T4 are subtrees. z y
/ \ / \
y T4 Right Rotate (z) x z
/ \ - - - - - - - - - > / \ /
\ x T3 T1 T2 T3 T4
/ \
T1 T2

b) Left Right Case z z x
/ \ / \ / \
y T4 Left Rotate (y) x T4 Right Rotate(z) y z

/ \ - - - - - - - - - -> / \ - - - - - - - -> / \ /
\ T1 x y T3 T1 T2 T3 T4
/ \ / \
T2 T3 T1 T2

c) Right Case z y **Right**
/ \ / \
T1 y Left Rotate(z) z
x / \ ----------> / \ / \
T2 x T1 T2 T3 T4
/ \
T3 T4

d) Right Left Case
z z x
/ \ / \ / \
T1 y Right Rotate (y) T1 x Left Rotate(z) z x
/ \ - - - - - - - - - -> / \ - - - - - - - -> / \ /
\ x T4 T2 y T1 T2 T3 T4
/ \ / \
T2 T3 T3 T4

Unlike insertion, in deletion, after we perform a rotation at z, we may have to perform a rotation at ancestors of z. Thus, we must continue to trace the path until we reach the root.

Conclusion: In this way we have implemented dictionary programme in cpp