Data Structures : Binary Search Tree, AVL Tree

Dr. Sudhanshu Gupta

Trees

- A finite set of one or more nodes such that – There is a specially designated node called root node – The remaining nodes are partitioned into n>0 disjoint sets T1, T2 …Tn where each of these sets is a tree.

- Node : Stands for the item of information and branch to other items
- Degree : No. of sub trees of a node are called degree.
- Leaf Nodes: Nodes with degree zero
- Non terminals : All other non leaf nodes
- Siblings : child of same parent

Trees

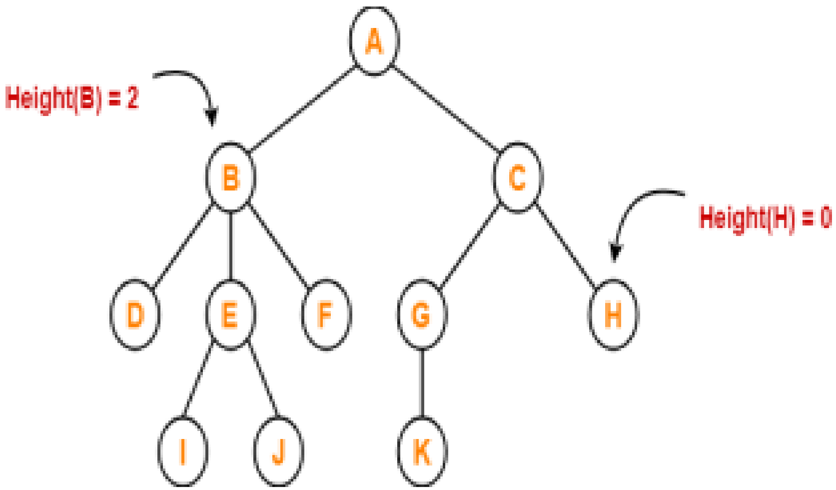
- Degree of tree : maximum degree of the nodes in the tree.

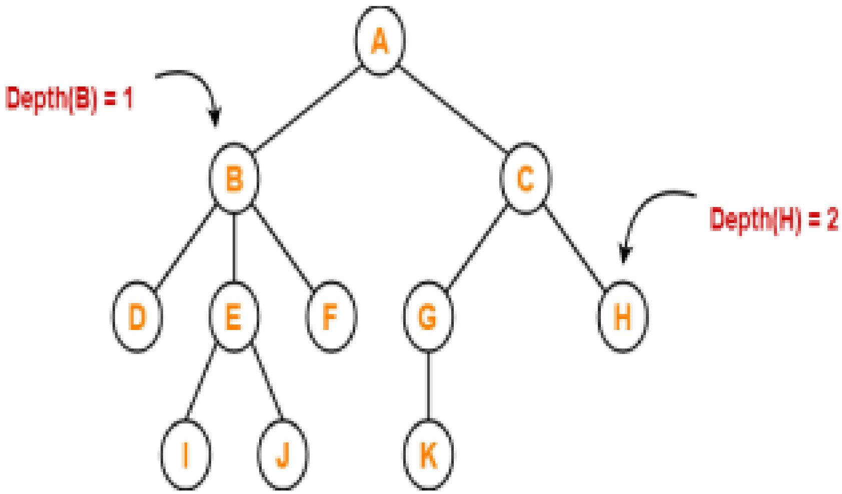
- Ancestors –All the nodes along the path from the root to that node .
- Level –The level of a node is defined by initially letting the root be at level l=0 its children are at l+1.
- A forest – is a set of n>0 disjoint tree if we remove root from a tree it is a forest .

Height and Depth of a tree

Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.

Total number of edges from root node to a particular node is called as **depth of that node**.




Depth(B) = 1
A
B
C
Depth(H) = 2
D
E
F
G
H
I
J
K

Parts of a Tree:nodes

Parts of a Tree: parent node

Parts of a Tree

Parts of a Tree

Parts of a Tree

root node

Parts of a Tree: sub tree

Traversal

- Systematic way of visiting all the nodes.
- Methods:
  - Preorder, Inorder, and Postorder
  - Traverse the left subtree before the right subtree.

– The name depends on when the node is visited. •

Preorder Traversal

  - Visit the node.
  - Traverse the left subtree.
  - Traverse the right subtree

Example: Preorder

43

64

31

20
40 56

89

28 33 47 59

Inorder Traversal

- Traverse the left subtree.
- Visit the node.
- Traverse the right subtree.

Example: Inorder

43

64

31

20
40 56

89

28 33 47 59

Postorder Traversal

- Traverse the left subtree.
- Traverse the right subtree.
- Visit the node.

Example: Postorder

43

64

31

20
40 56

89

28 33 47 59

Expression Tree

- A Binary Tree built with operands and operators.
- Also known as a parse tree.
- Used in compilers.

Example: Expression Tree (1/3+6*7/4)

/ /

1 3 *

4

– Postorder : Postfix Notation

- Notation
  - Preorder : Prefix Notation – Inorder : Infix Notation

6 7

Example: Infix

+

/

/

1

3 4

*

7

6

Example: Postfix

+

/

1 3

3

4

*

6 7

Example: Prefix

Binary Search Trees (BST)

- A recursively defined structure:
  - Contains no nodes, or
  - Comprised of three disjoint sets of nodes:
    - a root
    - a binary search tree (left subtree of the root)
    - a binary search tree (right subtree of the root)
- Every node entry has a unique key.
- Satisfies the binary search property:
  - All the keys in the left subtree of a node are less than the key of the node.
  - All the keys in the right subtree of a node are greater than the key of the node.

Binary Search Trees

branches

Root

B is the parent of F A, F

are children of B D is a

descendant of B

subtree rooted at D

G

B is an ancestor of

: **leaves**

: **internal nodes**

Binary Search Trees

height : 4

a path from the

root to a leaf

Root

-- 0 -- -- 1 -- -- 2 --

-- 3 -- -- 4 --

<u>height = length of longest path from the root to a leaf</u>

Binary Search Trees

- Used for storing and retrieving information
- Typical operations: insert, delete, search • A BST node contains:
  - A key (used to search)
  - The data associated with that key
  - Pointers to children, parent

- Leaf nodes have NULL pointers for children
- A BST contains a pointer to the root of the tree.

Key is an integer

BST Operations: Insert

- BST property must be maintained
- Algorithm to insert data with key k
  - Compare k to root key
  - If k < root key, go left
  - If k > root key, go right
  - Repeat until you reach a leaf. That's where the new node should be inserted.
- Running time:
  - The new node is inserted at a leaf position, so this depends on

the height of the tree.

1

- Worst case:
  - Inserting keys 1,2,3,... in this order will result in a tree that looks

2

    like a chain:
    - Tree has degenerated to list
    - Height : linear

3

Insert

- Create new node for the item.
- Find a parent node for the new node to be insterted.
- Attach new node as a leaf.

Insert

Insert

Storing binary tree in an array

- The root element is stored in the first location
- If a node is stored at i^{th} position then its left child is stored at $2*i+1^{th}$ position and its

right child is stored at $2*i+2^{th}$ position

Structure of a node

```
struct bstnode {
  int key;
  struct bstnode * lchild;
  struct bstnode * rchild;
```

}

Non recursive -1

```
Node_type * insert (node_type *root, node_type*new node ){
   node _type *p= root
   While (p! =NULL){
       If (newnode–>info<p->info){
           If (p->left != 0) { p=p->left;}
           Else { p->left =newnode; break; }
       }else if (newnode->info > p->info){
```

```
        If (p->right !=0){ p=p->right; }
        else{ p->right =newnode; break;}
    }else
        “duplicate record”
    }
```


Non recursive ..2

- }// end of while loop
- newnode-> left =newnode –>right=NULL
- If (root== NULL) root =newnode; • return root;

- }

Recursive

```
Node_type* insert
   (node_type*root,node_type *newnode){
If (root=NULL){
   root =newnode; root –>left =root->right =NULL;
} else if (newnode->info < root->info){
   root->left = insert(root->left,newnodes)
```

```
}else if (new node->info > root->info){
   root->right = insert(root->right,
newnode) }else{"duplicate key"}
return root
}
```


BST Operations: Insert

- Best case
  - The top levels of the tree are filled up completely
  - The height is then logn where n is the number of nodes in the tree. •

The height of a complete (i.e. all levels filled up) BST with n nodes is logarithmic.

  - Level i has 2i nodes, for i=0 (top level) through h (=height)
  - The total number of nodes, n, is then:

$$n = 2^0+2^1+...+2^h$$
$$= (2^{h+1}-1)/(2-1)$$

$= 2^h+1-1$

Solving for h gives us h ≈ logn

• An insert operation consists of two parts: – Search for the position

• best case logarithmic
• worst case linear

– Physically insert the node

• constant

12

4 14

2 8 16

Binary Search Trees

• Traversing a tree = visiting its nodes •

preorder

- visit root ,visit left subtree, visit right subtree • inorder
- visit left subtree, visit root, visit right subtree • postorder
- visit left subtree, visit right subtree, visit root

Preorder

Preorder (T)

```
{
  If T!=0 then
    Print (data(T))
    Call preorder (lchild(T))
    Call preorder (rchild(T))
}
```


Binary Search Trees

```
void print_inorder(Node *subroot ) {
```

```
        if (subroot != NULL) {
                print_inorder(subroot → left);
                printf(subroot→data);
                print_inorder(subroot →right);
        }
}
```

- There is exactly one call to print_inorder() for each node of the tree.

Binary Search Trees

4 14

2 8

16

6 10

in-order : 2 - 4 - 6 - 8 - 10 - 12 - 14

pre-order: 12 - 4 - 2 - 8 - 6 - 10 - 14 - 16

post-order: 2 - 6 - 10 - 8 - 4 - 16 - 14 - 12

level-order: 12 - 4 - 14 - 2 - 8 - 16 - 6 - 10

Binary tree traversal using stack

- 1) Create an empty stack S.
- 2) Initialize p node as root
- 3) Push the p node to S and set p = p->left until p is NULL
- 4) If p is NULL and stack is not empty then
  - a) Pop the top item from stack.
    b) Print the popped item, set p = popped_item- >right
    c) Go to step 3.

5) If p is NULL and stack is empty then we are done.

Search an element

- If target key is less than current node's key, search the left sub-tree.
- else, if target key is greater than current node's key, search the right sub-tree. • returns:
  - if found, pointer to node containing target key.
  - otherwise, NULL pointer.

Search Successful

43

Failed

Search

44

```
int search(struct node* node,int target){
  if(node==NULL) { return(0); }
  else {
   if(target == node->data){return(1); }
   else {
      if(target < node->data) {
            return(search(node->left,target));
      } else {
             return(search(node->right,target));
  } } }}
```

Delete a node from binary search tree

- Search for the node and Delete it
- The node to delete is a leaf (has no children).
  - Reset its parent's child pointer and deallocate memory
  - When the node to delete is a leaf, we want to remove it from the tree by setting the appropriate child pointer of its parent to null
  - or by setting root to null if the node to be deleted is the root, and it has no children.
- The node to delete has one child.
  - Reset its parent's child pointer, its child's parent pointer and deallocate memory

Delete leaf node

```
          50                               50
        /      \         delete(20)      /    \
      30        70       ---------->   30       70
     /  \      /  \                      \     /  \
   20    40  60    80                     40  60   80
```

Delete node with one child

```
    50                               50
 /      \         delete(30)       /   \
30       70       --------->      40     70
  \     /  \                            /  \
   40  60   80                         60   80
```


BST Operations: Delete

- The node to delete has two children
  - To delete the node, place its two children somewhere.

Restructure tree.

– The node to be deleted, x, has two children • Find the x's immediate successor, y. It is guaranteed to have at most one child

• Copy the y's contents over to x and delete y.

• Finding the immediate successor:

– The immediate successor will be in the right subtree.

– The immediate successor will be the smallest element in the right subtree.

– The smallest element in a BST is always the leftmost leaf.

Delete node with two children

```
    50                               60
  /     \         delete(50)       /    \
 40      70       ---------->    40      70
        /  \                               \
      60    80                              80
```

50

```
Delete(node_type **p){
 node_type *q;
 if(*p==0) Printf("empty node");
 elseif((*p)->left == 0){
     *p = (*p)->right; break;
 } elseif((*p->right==0)){
      *p = (*p)->left; break;
 } else {
```

```
    q=(*p->right);
    for( ;q->left;q=q->left);
    q->left =(*p)->left;
    *p = (*p)->right;
 }
}
```


Algorithm to delete :leaf node

```
struct node* delete(struct node* node, struct node* pnode, int
    target){
  struct node* rchild,* rchildparent;
  if(node==NULL){ return(pnode); }
  else{
    if(target == node->data){
        if(node->left == NULL && node->right == NULL)
            { if(pnode == NULL) {return(NULL);}
```

```
      if(pnode->left == node){ pnode->left = NULL;}
     else{ pnode->right = NULL; }
     return(pnode);
   } //end of if
```


Part 2: one child

```
if(node->left ==NULL ){
        if(pnode == NULL) {
             node = node->right;
              return(node);
        }
       if(pnode->left == node) {pnode->left =
       node->right;} else{ pnode->right = node->right; }
```

```
        return(pnode);
    }
```


Part 3: one child

- if(node->right ==NULL){
- if(pnode == NULL) {
- node = node->left;
- return(node);
- }
- if(pnode->left == node) {pnode->left = node->left;} • else{ pnode->right = node->left; }

- return(pnode);
- }

Part 4: two child

```
 rchild = node->right;
rchildparent=node;
 while(rchild->left != NULL) {
   rchildparent=rchild;
   rchild = rchild->left;
 }
 node->data=rchild->data;
 if(rchildparent == node) {
        node->right=rchild->right; //rchildparent->right=rchild->right; }
 else {
        rchildparent->left=rchild->right; //rchildparent->left=NULL;
```

}
free(rchild);
if(pnode ==NULL) { return(node); }
return(pnode); }

- else {
- if(target < node->data) {
- delete(node->left,node,target); • return(node);
- } else {
- delete(node->right,node,target); • return(node);
- } } } }

Sorting using Binary Search Trees

- Given a sequence of integers, insert each one in a BST
- Perform an inorder traversal. The elements will be accessed in sorted order.
- Running time:
  - In the worst case, the tree will degenerate to a list. Creation will take quadratic time and traversal will be linear. Total: $O(n^2)$
  - On average, the tree will be mostly balanced. Creation will take O(nlogn) and traversal will

again be linear. Total: O(nlogn)

Compare two binary trees

Equal (S,T) // return false if S and T are not equivalent ans=false

Case S=0 and T=0 ans =true

Case S!=0 and T!=0 :

```
If DATA (S) =DATA(T) {
    ans=equal (Lchild(S),Lchild(T));
    If (ans=true) {
      ans=equal(Rchild(S),Rchild(T))
    }
```

```
 return (ans)
end equal
```


Create a copy of binary tree

```
Copy (T){
  Q=0
  If (T != 0) {
    R =copy (L child(T))
    S=copy(R child (T))
    Call getnode(Q)
    Lchild(Q) =R
    Rchild(Q) =S
    Data(Q) =data(T)
 }
  return(Q)
End copy
```

Balanced Trees

- Force the subtrees of each node to have almost equal heights
- Place upper and lower bounds on the heights of the subtrees of each node. • Force the subtrees of each node to have similar sizes (=number of nodes)

Adelson-Velskii and Landis tree (AVL)

- It is a binary search tree
  - For every node, the heights of the left and right subtrees differ at most by one.
- Height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree) • An

AVL tree has balance factor calculated at every node

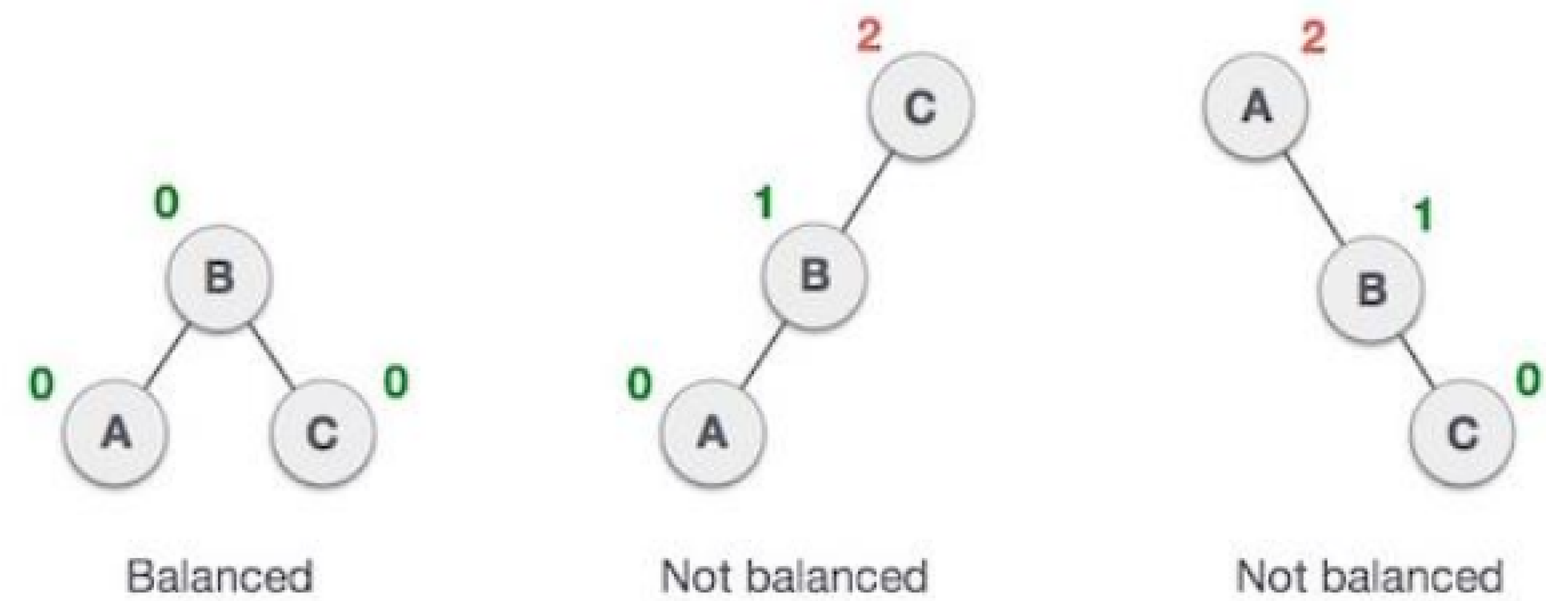
– For every node, heights of left and right subtree can differ by no more than 1

– Store current heights in each node

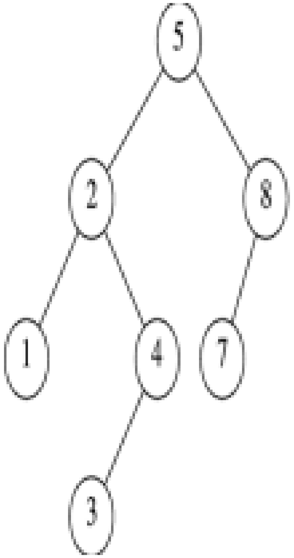
AVL tree

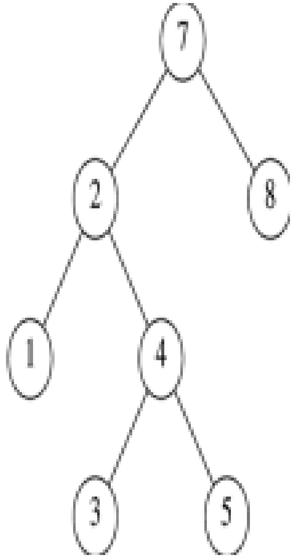
• Each node contains a value (-1, 1, 0) indicating which subtree is "heavier" • Insert and Delete restructure the tree to make it balanced (if necessary).

AVL tree : Balanced vs Non balanced subtrees



AVL tree : Balanced vs Non balanced subtrees





1

-1 1

0 0 1

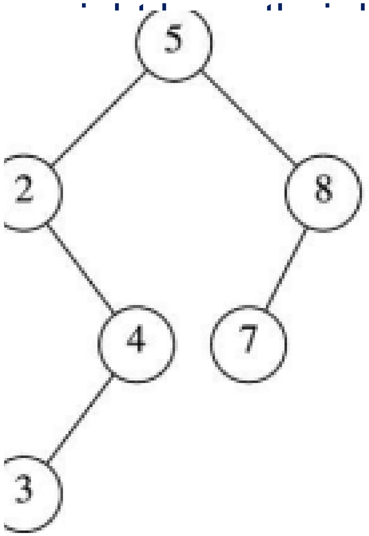
0

AVL property violated AVL tree

Insertion in AVL Tree

- Similar to binary search tree
  - – But may cause violation of AVL tree property
  - – Restore the destroyed balance
- After an insertion, only nodes that are on the path from the insertion point to the root ance altered

he deepest such node guarantees that the entire tree satisfies



7

6 8

6

Original AVL tree ~~Insert 6~~

Insertion Algorithm

- First, insert the new key as a new leaf just as in ordinary binary search tree
- Then trace the path from the new leaf towards the root. For each node x encountered, check if heights of left(x) and right(x) differ by at most 1 – If yes, proceed to parent(x)
  - If not, restructure by doing either a single rotation or a double rotation
- Once we perform a rotation at a node x, we won’t need to perform any rotation at any ancestor of x.

Cases for Rebalance

- Denote the node that must be rebalanced X –
  Case 1: an insertion into the left subtree of the left child of X
  - Case 2: an insertion into the right subtree of the left child of X
  - Case 3: an insertion into the left subtree of the right child of X
  - Case 4: an insertion into the right subtree of the right child of X
- Cases 1&4 are mirror image symmetries

with respect to X, as are cases 2&3

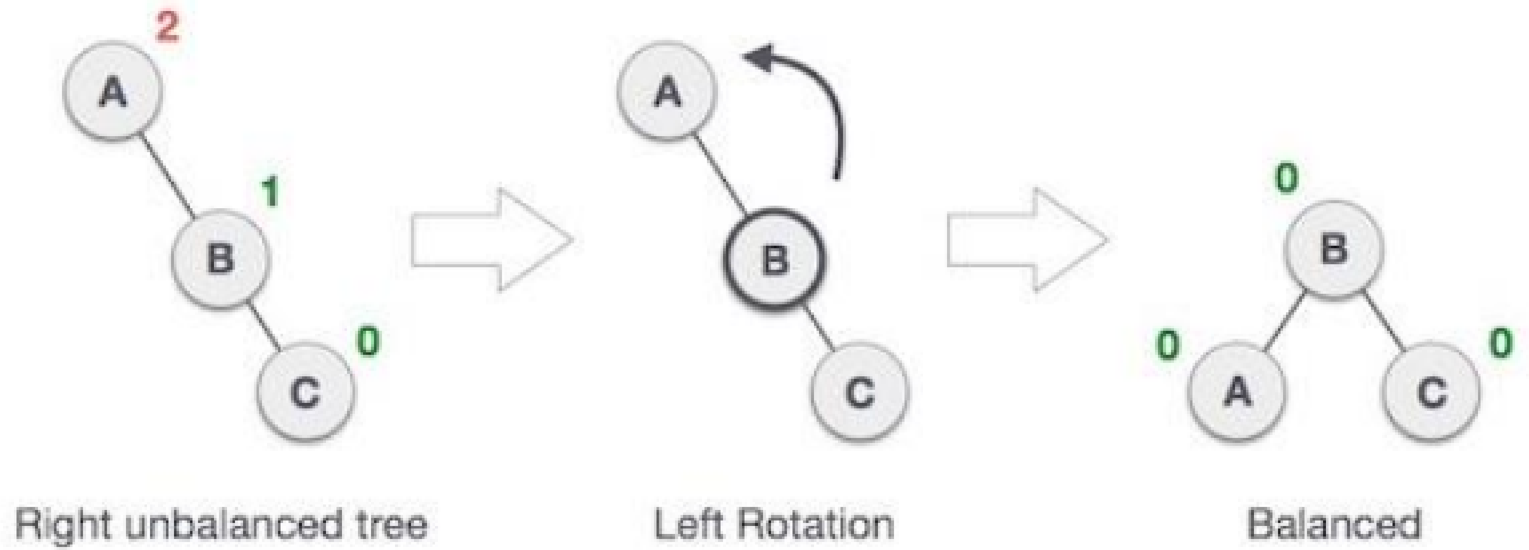
AVL trees: Fixing imbalances

or or TYPE 1 TYPE 2

Rotations

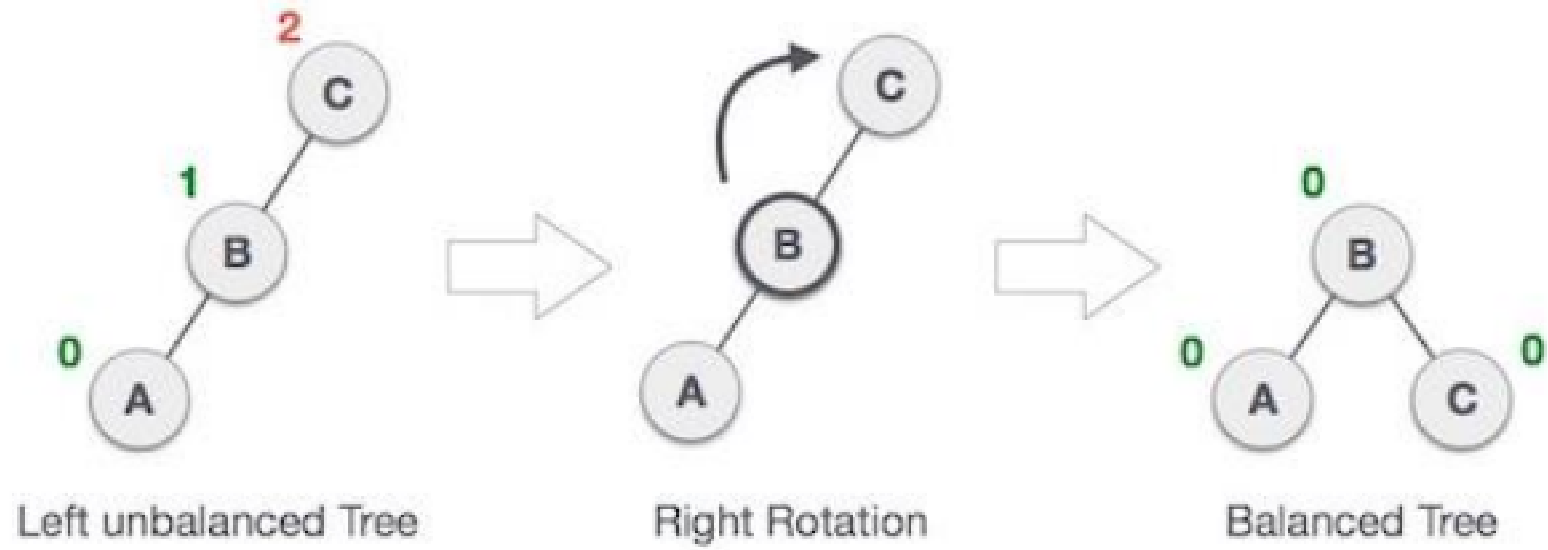
- Imbalance : Height difference between two subtrees of a node becomes greater than 1 or smaller than -1.
- Insertion occurs on the “outside” (i.e., left left or right-right) is fixed by single rotation of the tree
- Insertion occurs on the “inside” (i.e., left right or right-left) is fixed by double rotation of the tree

Left Rotation: If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree



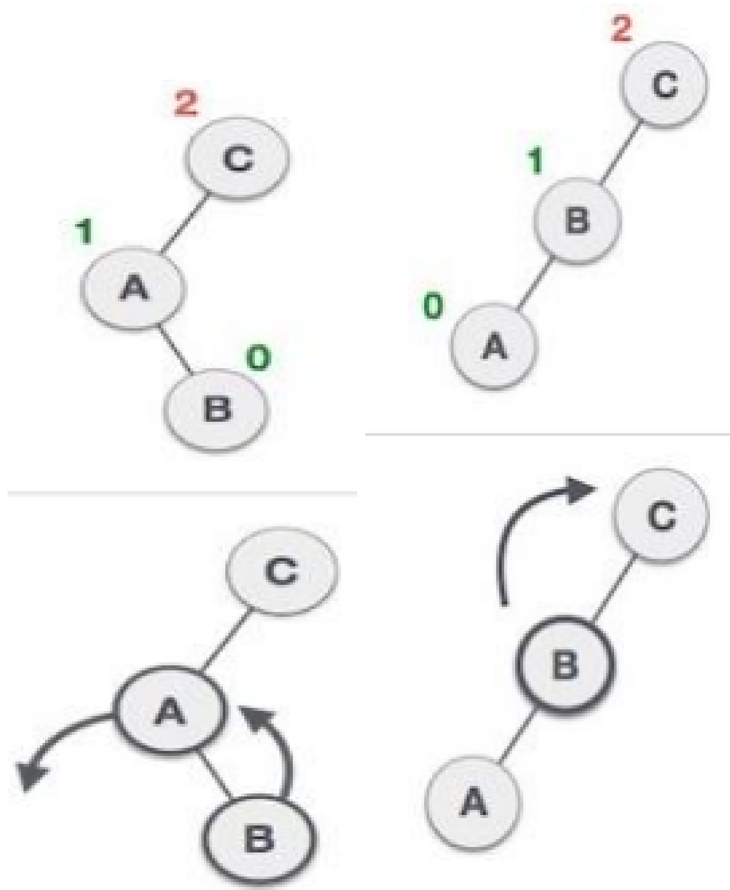
71

Right Rotation :Tree may become unbalanced, if a node is inserted in the left subtree of the left subtree.

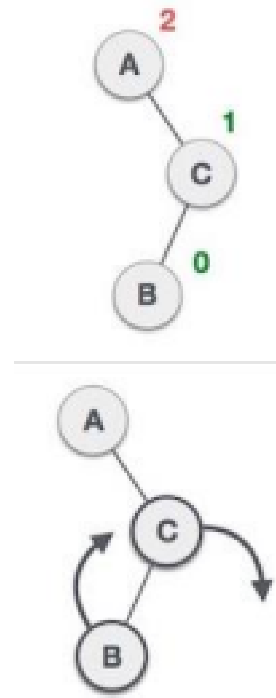


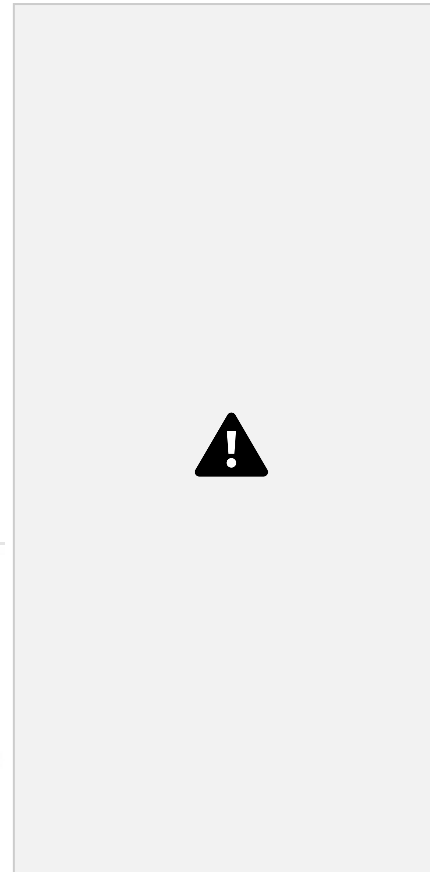
72

Left-Right Rotation : combination of left rotation followed by right rotation.



Right-Left Rotation : right rotation followed by left rotation.





AVL Trees: single rotation

D

node with imbalance

rotate at node 6

C

right

A B

A B C D

This is a single right rotation. A single left rotation is symmetric.

AVL Trees: Double rotation

node with *node with*

*imbalance*D

imbalance D

C

STEP2:

node 6

A

B C

A B

A B C D

right rotate at

STEP 1: left rotate at node 2

AVL Trees: Double rotation

- If you want to do it in one step, imagine taking 4 and moving it up ; in between 2 and 6, so that 2 and 6 become its new children:
- B and C will then be adopted by 2 and 6 respectively, in order to maintain the BST property.

imbalance

node with D

A

B C

A B C D

Sequentially insert 3, 2, 1, 4, 5, 6 to an AVL Tree:Single Rotation

3

2

2

3

2

2

1 3

1 3 Insert 4

1 3

Insert 5, violation at node 3

Insert 3, 2

1

Single rotation

4

4

Insert 1

2

violation at node 3

2

2

5

4

1 4

Insert 6,

2 5

violation at node 2

3 5

1 3 6 Single rotation

Single rotation

78

1 4

3 5

6

If we continue to insert 7, 16, 15, 14, 13, 12, 11, 10, 8, 9

4

4

violation at node 5

2 5

4

1 3 6

2 6

Insert 7,

2 6

7

4

1 3 5 7

2 6

Single rotation

1 3 5 7

Insert 16, fine

Insert 15

~~violation at node 7~~

15

16

1 3 5

Single rotation

But….

Violation remains

AVL tree: defining structure

```
struct node {
    int key;
    struct node *left;
    struct node *right;
    int height;
};
  // Get height of the tree
int height(struct node *N){
    if (N == NULL) return 0;
    return N->height;
```

```
}
 int max(int a, int b){ return (a > b)? a : b; }
```


AVL tree: create new node

```
struct node* newNode(int key)
{
    struct node* node = (struct node*)
                        malloc(sizeof(struct node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1; // new node is leaf
    return(node);
```

```
}
```