AVL Trees

R. K. Ghosh

IIT Kanpur

Balanced Binary Search Trees

Balance Property

- An AVL tree is a height balanced binary tree.
- The height balance property must hold at each node.
- The height balance property means the balance factor (BF) or the difference in heights of left and right subtree is ± 1.
- Each node stores the balance factor (BF) in local variable.
- As insertions and deletions happen BF gets disturbed.
- BF is updated on every insertion and deletion.
- If the bound on BF is disturbed then rotations are performed to restore it.

Maintaining Balance Property

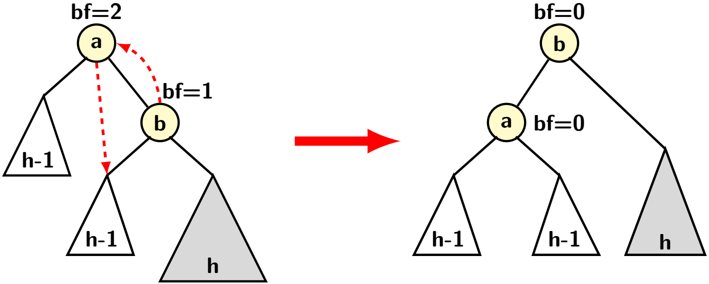
- There are two types of rotations:
  1. Single rotations, and
  2. Double rotations.
- Each of these rotations are performed through combinations two basic types of rotation:
  1. Left rotation, and
  2. Right rotation

AVL Tree Single Rotation LL Type

Define AVL tree node

```
typedef struct node {
    int info, ht;
    struct node *left,*right;
} Node;
```

AVL Tree Single Rotation LL Type



AVL Tree Single Rotation LL Type

Left rotation code

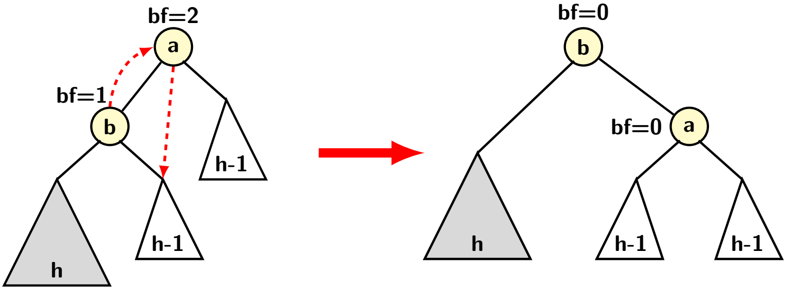
```
Node * rotateLeft(Node *x) {
    Node *y;
    y = x->right;
    x->right = y->left;
    y->left = x;
    x->ht = height(x);
    y->ht = height(y);
    return(y);
}
```

AVL Tree Single Rotation LL Type

Left rotation

```
Node * LL_rotation(Node *T) {
    T = rotateLeft(T);
    return T;
}
```

Single Rotation: RR Type



Single Rotation: RR Type

Right rotation code

```
Node * rotateRight(Node *x) {
    Node *y;
    y = x->left;
    x->left = y->right;
    y->right = x;
    x->ht = height(x);
    y->ht = height(y);
    return y;
}
```

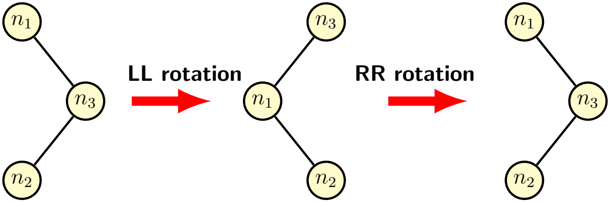
Single Rotation: RR Type

RR rotation

```
Node * RR_rotation(Node *T) {
    T = rotateRight(T);
    return T;
}
```

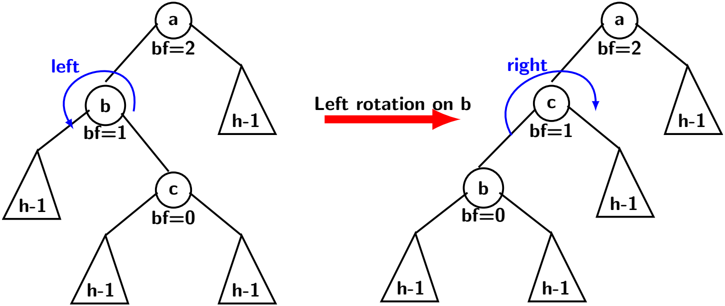
AVL Tree Double Rotations

Consider the following configuration of BST.

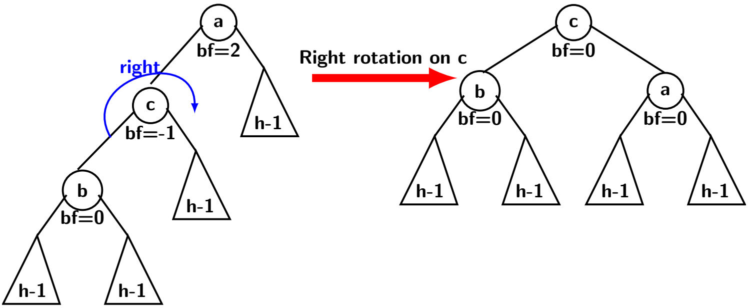


- In this case, tree remains unbalanced after completing LL rotation.
- If we try to apply RR rotation it return back to original configuration.

Double Rotation: LR type



Double Rotation: LR type (contd.)



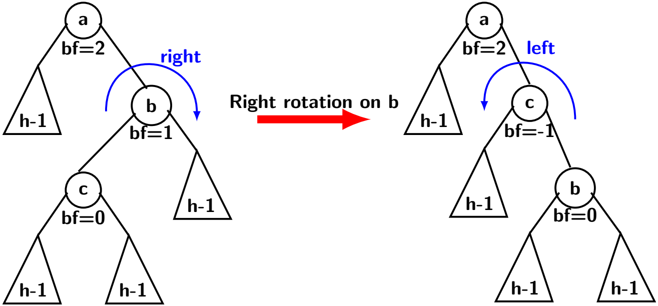
AVL Tree Double Rotations

Code for LR rotation

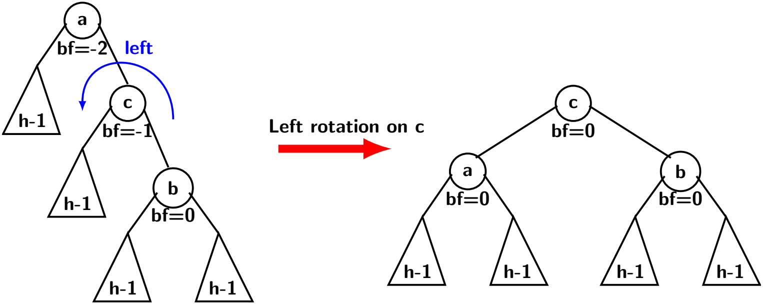
```
Node * LR_rotation(Node *T) {
    T->left = rotateLeft(T->left);
    T = rotateRight(T);

    return T;
}
```

Double Rotation: RL type



Double Rotation: RL type (contd.)



AVL Tree Double Rotations

Code for RL rotation

```
Node * RL_rotation(Node *T) {
    T->right = rotateRight(T->right);
    T = rotateLeft(T);
    return T;
}
```

AVL Tree: Computation of Height

Code for height

```
int height(Node *T) {
    int lh, rh;
    if(T == NULL)
        return 0;
    if(T->left == NULL)
        lh = 0;
    else
        lh = 1 + T->left->ht;
    if(T->right == NULL)
        rh = 0;
    else
        rh = 1 + T->right->ht;
    if(lh > rh)
        return lh;
    return rh;
}
```

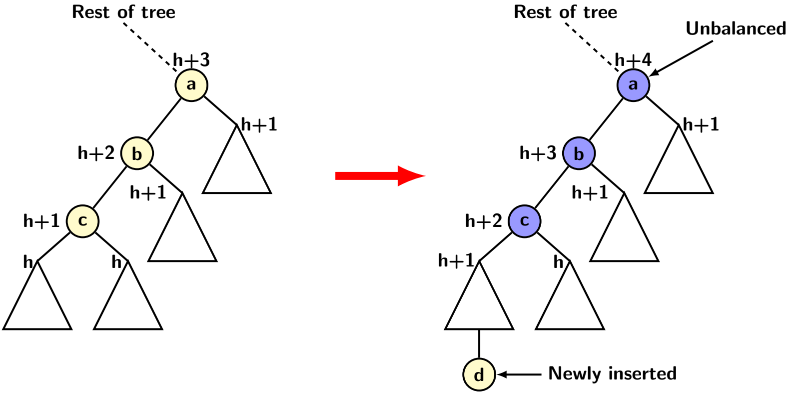
AVL Tree: Computation of BF

Code for BF

```
int BF(Node *T) {
    int lh, rh;
    if(T == NULL)
        return 0;
    if(T->left == NULL)
        lh = 0;
    else
        lh = 1 + T->left->ht;
    if(T->right == NULL)
        rh = 0;
    else
        rh = 1 + T->right->ht;
    return(lh - rh);
}
```

Where to Balance?

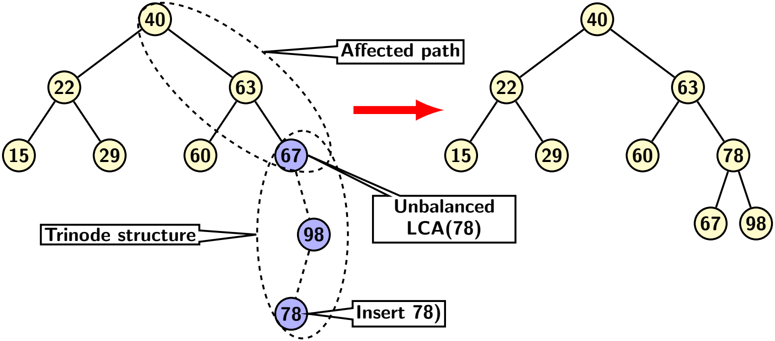
- When an insertion occurs, the BF may get disturbed at nodes between the parent up to the root.



Detecting the Tri-node Structure

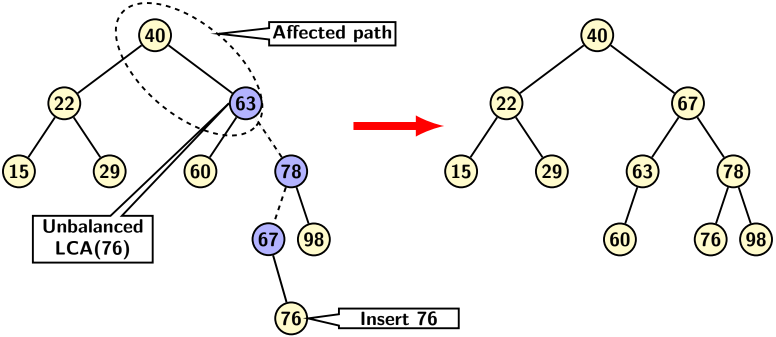
- The tri-node structure to start rotation would be:
  - The lowest ancestor a of newly inserted node where imbalance occurs.
  - The child of a at height height,i.e., b.
  - The grand child of a of higher height, i.e., c.
- The rotation operation will be applied recursively to tri-node structure along the affected part.

Trinode Structure with Insertion

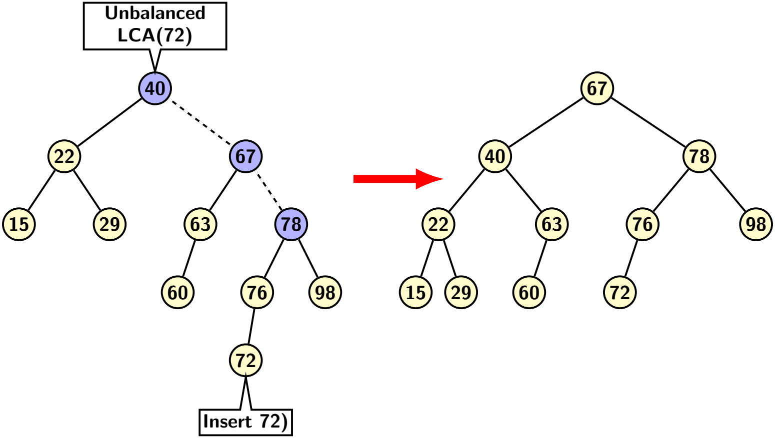


- Affected path where balance can get disturbed is from the LCA to the root of the tree.
- Rotation should be applied along the affected path.

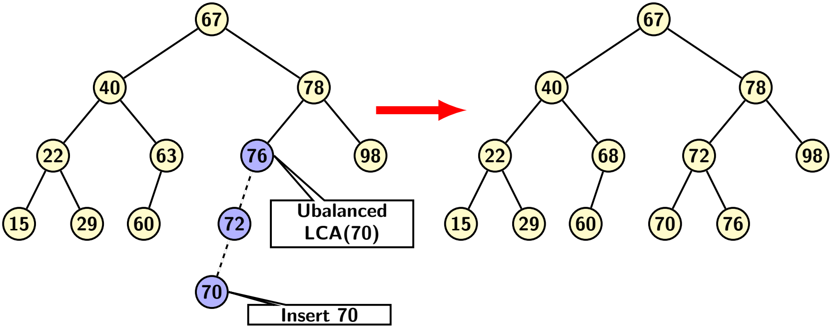
Insertion Example



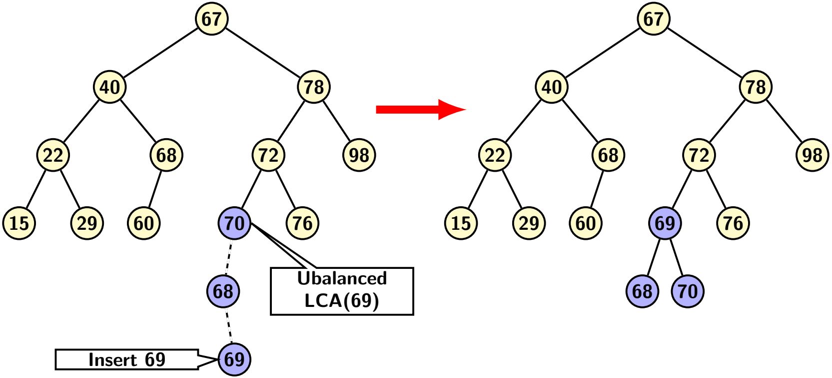
Insertion Example



Insertion Example



Insertion Example



Code for insertion

```
Node * insert(Node *T, int x) {
    if (T==NULL) {
        T = (Node*)malloc(sizeof(Node)); // Create new node
        T->info = x;
        T->left = NULL;
        T->right = NULL;
    }
    else if (x > T->info) {
            T->right = insert(T->right, x);// Insert right
            if (BF(T) == -2) // Rebalance
                if (x > T->right->info) // Right skew
                    T = RR_rotation(T);
                else // Right-left zig
                    T = RL_rotation(T);
    }
```

Code for insertion (contd.)

```
        else if(x < T->info) {
                // Insert into left subtree
                T->left = insert(T->left,x);
                if(BF(T) == 2) // Rebalance
                    if(x < T->left->info) // Left skew
                        T = LL_rotation(T);
                    else // Left-right zig
                        T = LR_rotation(T);
        }
        T->ht = height(T); // Update height
        return T;
}
```

Complexity

- Let us find the minimum number of internal nodes in an AVL tree of height h.
- For $h = 1$ $n(h) = 1$, for $h = 2$, $n(h) = 2$.
- For $h \geq 3$, we have one root node and two subtrees.

$$n(h) = 1 + n(h-1) + n(h-2)$$

- We know $n(h-1) > n(h-2)$, therefore, $n(h) > 2n(h-2)$.
- Apply the above inequality until i times: $n(h) > 2^i.n(h-2i)$.
- Set $i = (h/2) - 2$, then

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

- So, $n(h) > 2^{(h/2)-1}$. Taking log, we have $h \geq 2\log n(h) + 2$

Complexity

- Rotation is O(1). In most cases either a single or a double rotation fixes the tree.
- But the affected path is between the root and the lowest ancestor of the newly inserted node which becomes unbalanced.
- So, in the worst case the rotation may have to be applied O($\log n$) times.
- Insertion step itself is like BST, so it may also require time of O($\log n$).

Complexity

- Deletion is performed much like same manner as in a BST.
- The tri-node structure have to be identified for every physical node that is deleted.
- Then rotation should be applied to the affected path.
- So, principle of rotation remain same for insertion and deletion.
- Both require time of O($\log n$).

Summary

- AVL tree are balanced binary search trees.
- The idea is to use two basic rotation on a trinode structure to preserve balance after each inserion and deletion in a BST.
- Balance information is maintained at each node.
- Whenever balance factor at a node goes outside the range [-1,1] rotation is performed on trinode structure.
- The trinode structure is determined by:
  - LCA of the two subtree roots which becomes unbalanced,
  - Child of the LCA node at higher height
  - Child of child the LCA node at higher height
- All AVL operations can be performed in O($\log n$) time.