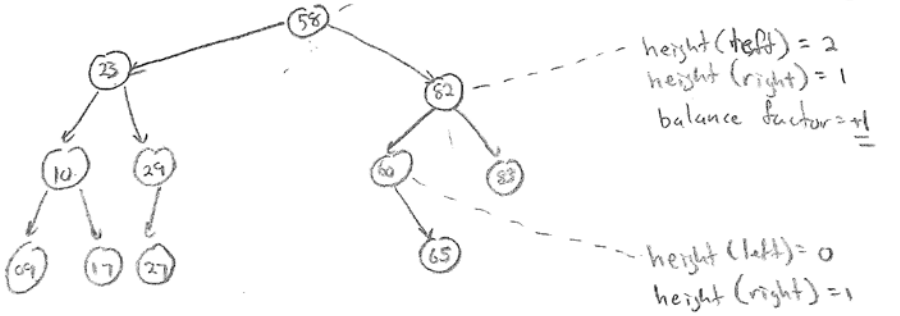
AVL Trees (Balanced Binary Search Tree)

- Named after Adelson-Vel'skii and Landis
- Height balanced tree: for all nodes x, the heights of the left and right subtrees differ by at most 1
- Typically uses a balance factor stored in each node

  balance factor = height (x → left) - height (x → right)
- A properly balanced AVL tree should ensure that all nodes balance factors are +1, -1, or 0,

Examples:



height (left) = 3
height (right) = 3
balance factor = 0

height (left) = 2
height (right) = 1
balance factor = +1

height (left) = 0
height (right) = 1
balance factor = -1

- To keep tree balanced additional work is needed for inserting and deleting nodes

Inserting Nodes in AVL Tree:

① Insert node using same method for inserting into unbalanced binary search tree
② Account for changes in balance factors that might result
③ If balance factor becomes +2 or -2, must rebalance subtree
   ↳ performed using rotations

Apply rotations starting at the first ancestor of inserted node

Let $\underline{x}$ = inserted node

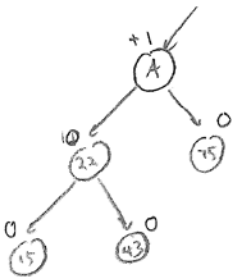
A = first ancestor of x with balance factor of +2 or -2

LL Rotation: If node x is within left subtree of the left subtree of A

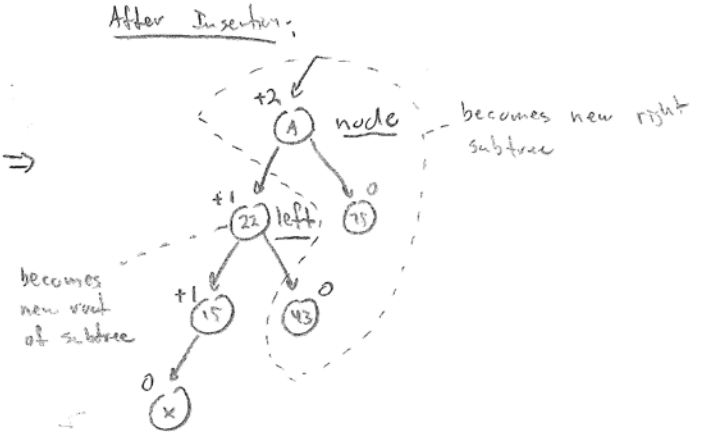
1. left = A → left
2. A → left = left → right
3. left → right = A
4. parent(A) = left
5. A → balance factor = 0
6. left → balance factor = 0

Example:

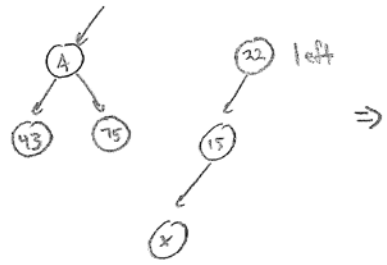
Before Insertion:



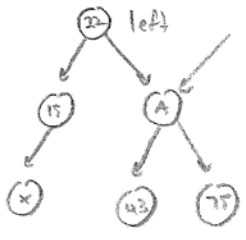
After Insertion:



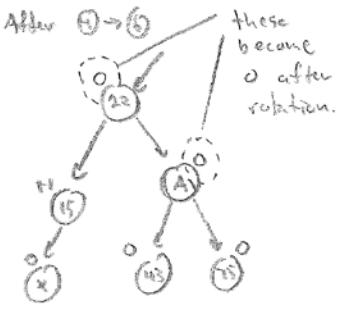
After ②



After ③



After ④ → ⑥

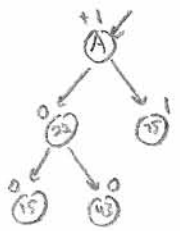


LR Rotation: If node x is within right subtree of the left subtree of A

① left = A → left
② grandchild = left → right
③ left → right = grandchild → left
④ grandchild → left = left
⑤ A → left = grandchild → right
⑥ grandchild → right = A
⑦ parent(A) = grandchild

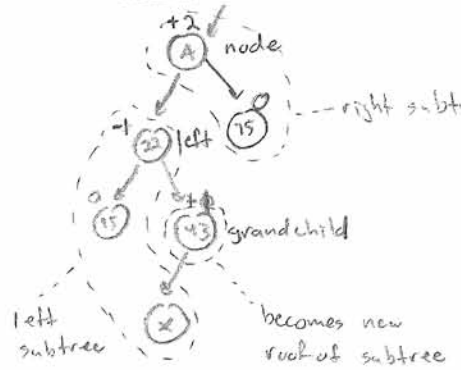
Example:

Before insertion:



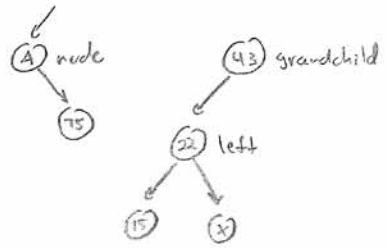
⇒

After insertion:

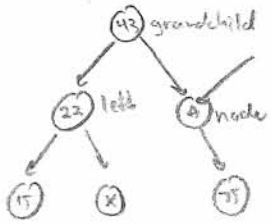


⇒

After ⑤→⑥:

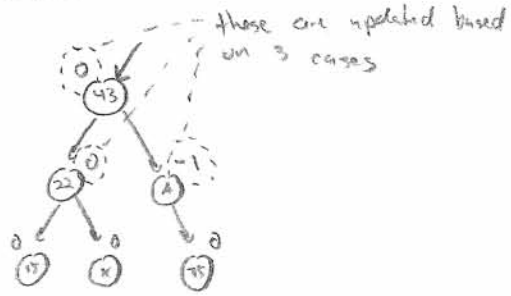


After ⑤→⑥:



⇒

After ⑦:



LR Rotation Updating Balance Factors

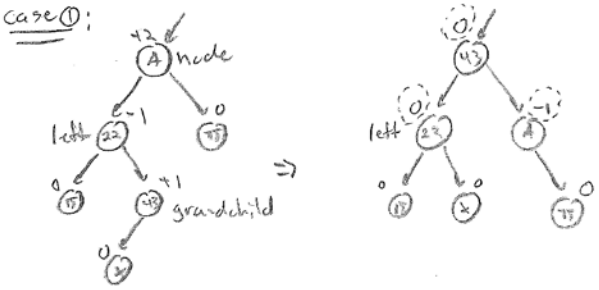
① if grandchild → balance factor = +1
A → balance_factor = -1
left → balance factor = 0

② else if grandchild → balance factor = 0
A → balance factor = 0
left → balance factor = 0

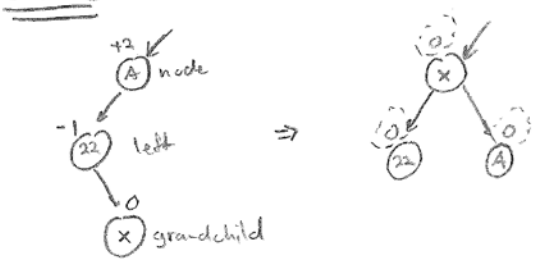
③ else
A → balance factor = 0
left → balance factor = 1

④ grandchild balance factor = 0

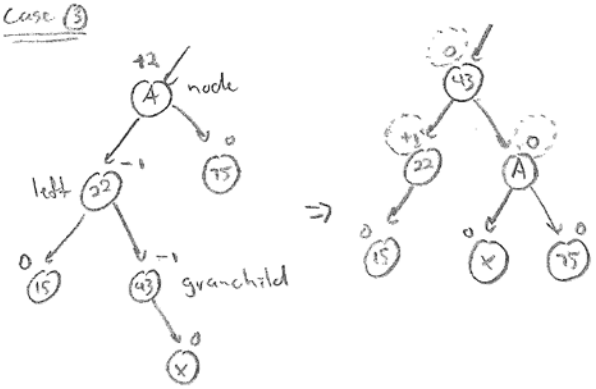
case ①:



case ②:



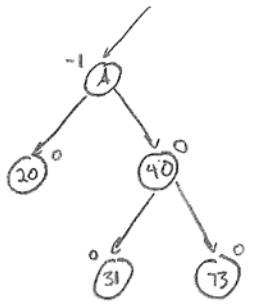
case ③



RR Rotation: If node x is within right subtree of right subtree of A

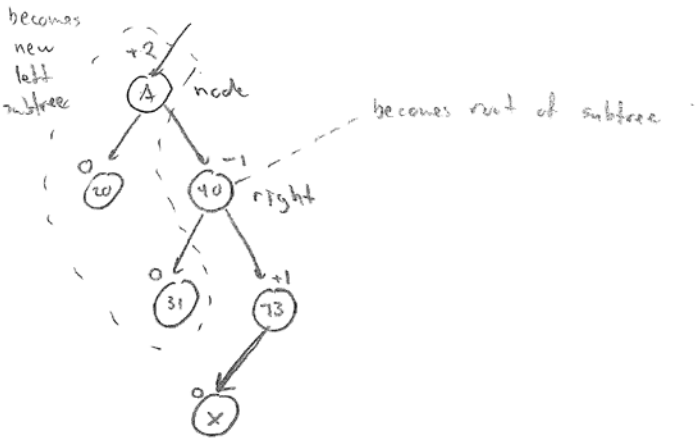
1. right = A → right
2. A → right = right → left
3. right → left = A
4. parent(A) = right
5. A → balance factor = 0
6. right → balance factor = 0

Before Insertion:



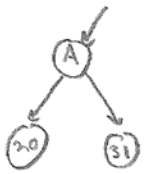
⇒

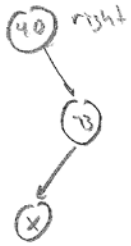
After Insertion:



⇐

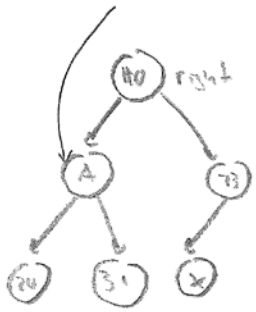
After (2):





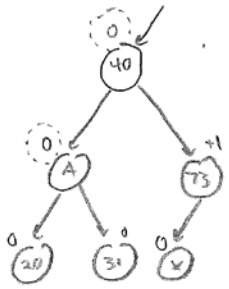
⇒

After (3):



⇒

After (4) & (6):

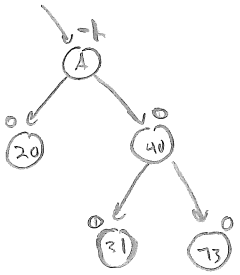


RL Rotation: If node x is within left subtree of right subtree of A

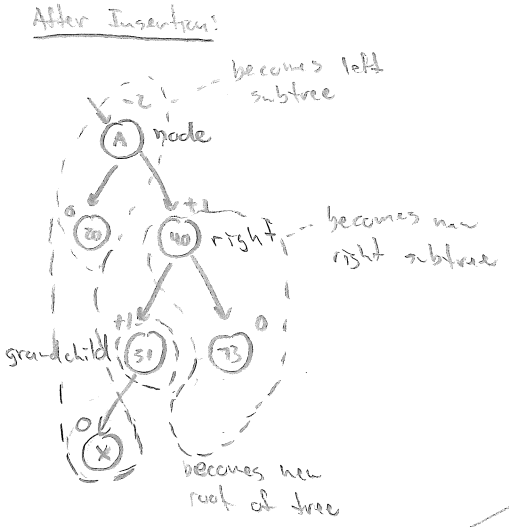
1. right = A→right
2. grandchild = right→left
3. right→left = grandchild→right
4. grandchild→right = right
5. A→right = grandchild→left
6. grandchild→left = A
7. parent(A) = grandchild

Example:

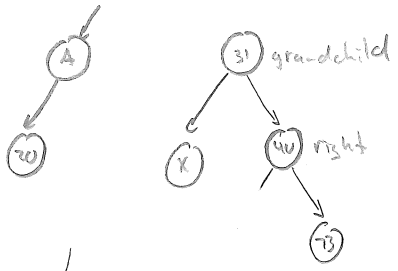
Before Insertion:



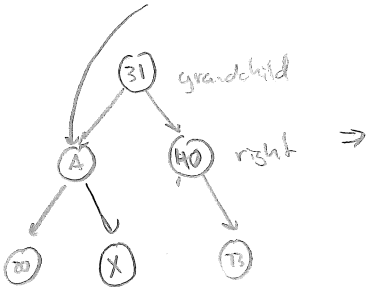
After Insertion:



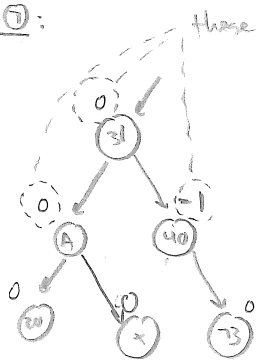
After 1→4:



After 5→6:



After 7:



these are updated based on 3 cases

RL Rotation updating Balance Factor

① if grandchild → balance factor = +1
A → balance factor = 0
right → balance factor = −1

② else if gra-dchild → balance factor = 0
A → balance factor = 0
right → balance factor = 0

③ else
A → balance factor = +1
right → balance factor = 0

④ gra-dchild → balance factor = 0

AVL Insertion:

- Calls insert function recursively to determine where to insert node
- Newly added node will always have a balance factor of 0
- Use additional variable in function to indicate if tree balance has change and we may need to rotate
  - balanced = 0 ≡ tree needs to be balanced
  - balanced = 1 ≡ tree is balanced

int bstree_insert(BsTree *tree, int data);

Operation: Insert new node in balanced AVL tree with provided data. calls a recursive insert() function to perform the insertion and balancing

Pseudo code:

① int balanced = 0

② return insert(tree, tree->root, data, &balanced)

int insert(BsTree *tree, BsTreeNode * node, int data, int *balanced);

Operation: Insert node into binary search tree maintaining AVL property

Pseudocode:

① if node is NULL, return bstree_ins_left(tree, NULL, data) *requires bstree_ins_left to initialize balance factor to 0

② compval = compare(data, node->data)

③ if node with same key found (compval == 0)

a) either return error or handle reinsertion of hidden nodes (i.e. lazy removal)

④ if key < node->data (compval < 0)

a) if node->left == NULL

I) if bstree_ins_left(tree, node, data) != 0, return error (error during insertion)

II) balanced = 0

b) else

I) if insert(tree, node->left, data, balanced) != 0, return error (error during insertion)

c) if not balanced

I) if node->balance factor == +1 (i.e. new insertion to left would result in balance factor of 2 if not rebalanced)

i) if node->left->balance factor == +1, LLRotate(node)

ii) else LRRotate(node)

iii) balanced = 1

II) else if node->balance factor == 0

i) node->balance factor = 1

III) else (node->balance factor == -1)

i) node->balance factor = 0

ii) balanced = 1

insert (cont.)

⑤ else (key > node->data)

a) if node->right == NULL
   I) if bstree_ins_left(true, node, data) != 0, return error
   II) balanced = 0 right

b) else
   I) if insert(true, node->right, data, balanced) != 0, return error

c) if not balanced
   I) if node balance factor is -1 (i.e. new insertion to right would result in balance factor of -2 if not rebalanced)
      i) if node->right->balance factor is -1, RRRotate(node)
      ii) else RLRotate(node)
      iii) balanced = 1
   II) else if node->balance factor is 0
      i) node->balance factor = -1
   III) else (node->balance factor is +1)
      i) node->balance factor = 0
      ii) balanced = 1

⑥ return success (0)

AVL Removal (The lazy method)

- Add additional variable to each node named hidden
- On removing a node, set hidden to 1
- if node reinserted, set hidden to 0

- works in cases when few values are removed
- Requires minor changes to insert, and lookup
- structure of AVL unchanged, so tree remains balanced (no rotations needed)