Today's topics

- AVL trees

DSA lab assignment

- Today, we will release the DSA second lab assignment
- You can talk to TAs during the Friday lab slot (28th April) if you have any queries regarding the installation
- You can also meet me during 4-5 pm on Friday (28th April) in my office if your queries are not resolved during the lab slot
- Post 28th April, queries related to installation will not be answered
- If you have any other doubts regarding the assignment, you can ask me after the class or during my office hours

AVL tree

References

- Section-4.4 from Mark Allen Weiss

AVL tree

- AVL tree is a BST that satisfies the **height-balance** property

- height-balance property: For each node in the tree, the heights of the left and right subtrees differ by at most one

Balance factor

- balance_factor(root) = height(root->left) – height(root->right)

- A node is not balanced
  - if balance_factor < -1 or balance_factor > 1

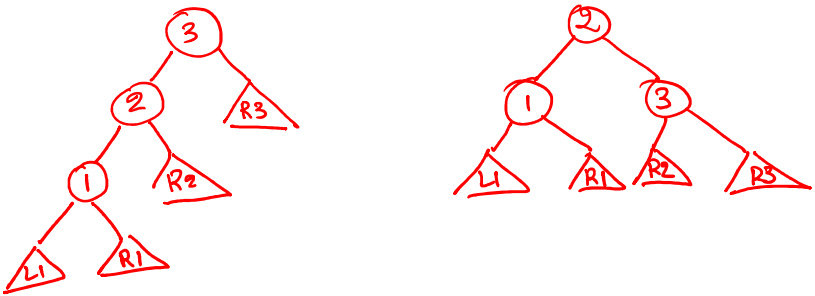
- If any node in a BST is not balanced, then the tree is not an AVL tree

Left heavy and right heavy

- A node is heavy if the balance_factor is not zero
- A node, n, is left heavy if the balance_factor(n) > 0
- A node, n, is right heavy if balance_factor(n) < 0

Rotations

LL rotation




```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

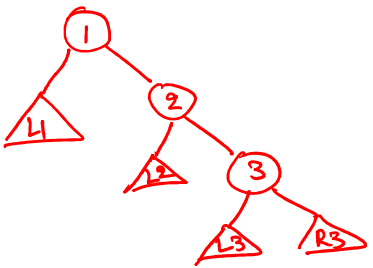
This is the general structure of an LL rotation. Notice that three nodes involved in the imbalance can have other balanced subtrees as children. After a rotation, as we discussed before, we need to find a place for these children in the rotated (balanced) tree. Notice that L1, R1, and R3 can stay in the same position as they were before the rotation. However, node 2 adopted a new child 3 and abandoned R2, so we need to find a new place for R2. Notice that there is a vacancy on the left of 3. Because all nodes in R2 are greater than 2 and less than 3, we can make R2 a left child of 3.

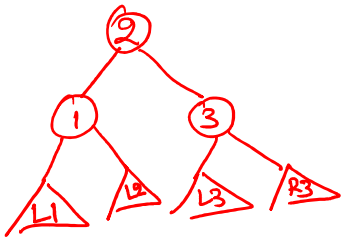
LL rotation

- We perform LL rotation on a node n, if
  - **balance_factor(n) > 1** and **n->left is not right heavy**

```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

RR rotation






```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

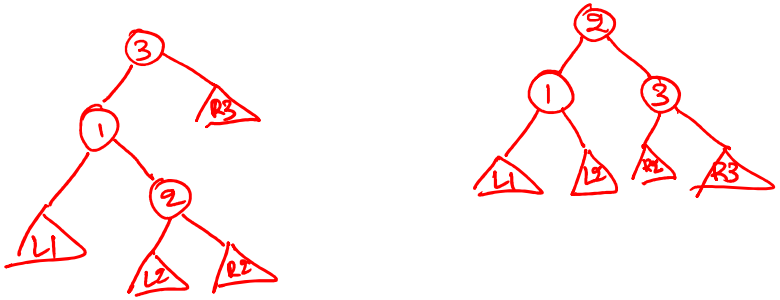
This is the general structure of an RR rotation. Notice that three nodes involved in the imbalance can have other balanced subtrees as children. After a rotation, as we discussed before, we need to find a place for these children in the rotated (balanced) tree. Notice that L1, L3, and R3 can stay in the same position as they were before the rotation. However, node 2 has now become the parent of its old parent (1) and abandoned L2, so we need to find a new place for L2. Notice that there is a vacancy in the right of 1. Because all nodes in L2 are greater than 1 and less than 2, we can make L2 the right child of 1.

RR rotation

- We perform RR rotation on a node n, if
  - **balance_factor(n) < -1** and **n->right is not left heavy**

```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

LR rotation




```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

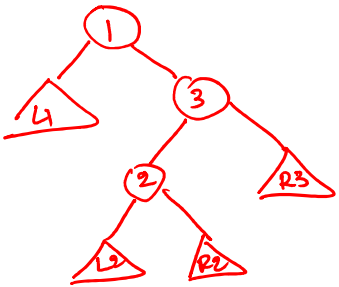
This is the general structure of an LR rotation. Notice that three nodes involved in the imbalance can have other balanced subtrees as children. After a rotation, as we discussed before, we need to find a place for these children in the rotated (balanced) tree. Notice that L1 and R3 can stay in the same position as they were before the rotation. However, node 2 has now become the parent of its previous parent (1) and grandparent(3) and abandoned L2 and R2, so we need to find a new place for L2 and R2. Notice that there is a vacancy on the right of 1 and the left of 3. Because all nodes in L2 are greater than 1 and less than 2, we can make L2 the right child of 1. Similarly, all the nodes in R2 are greater than 2 and left than 3; we can make R2 the left child of 3.

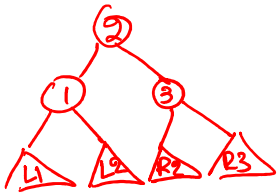
LR rotation

- We perform LR rotation on a node n, if
  - **balance_factor(n) > 1** and **n->left is right heavy**

```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

RL rotation






```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

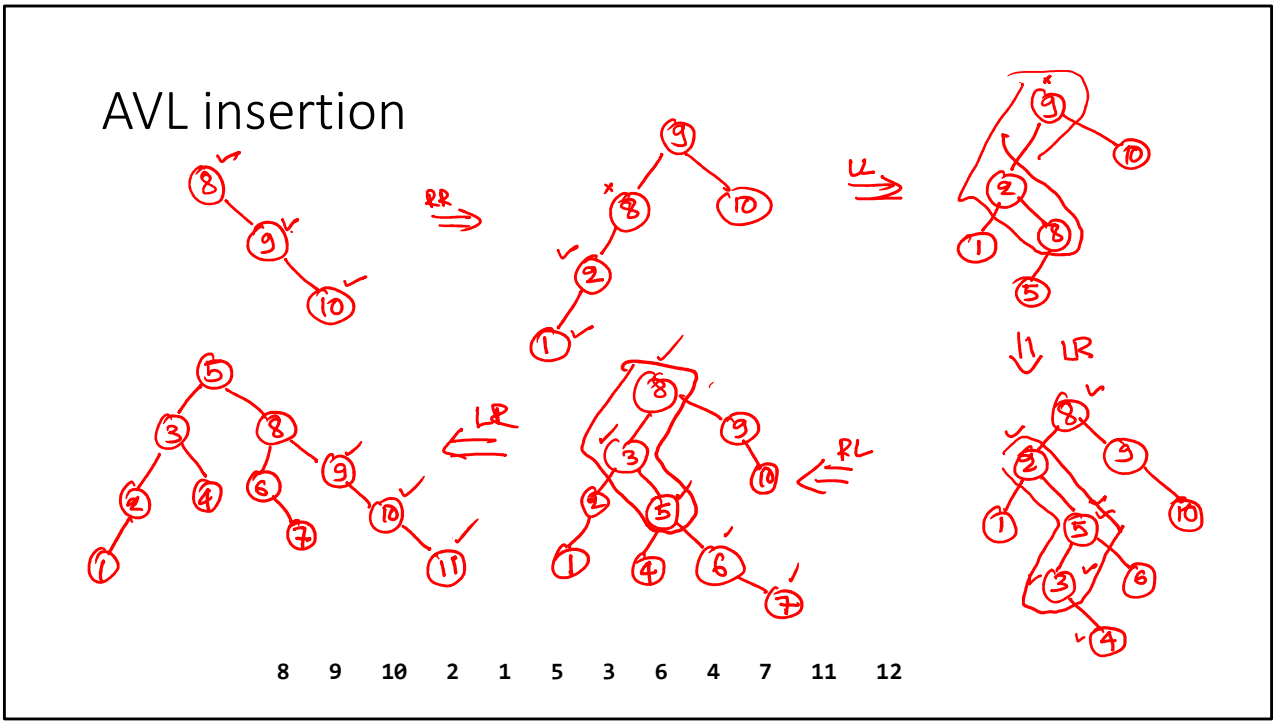
This is the general structure of an RL rotation. Notice that three nodes involved in the imbalance can have other balanced subtrees as children. After a rotation, as we discussed before, we need to find a place for these children in the rotated (balanced) tree. Notice that L1 and R3 can stay in the same position as before the rotation. However, node 2 has now become the parent of its previous parent (3) and grandparent(1) and abandoned L2 and R2, so we need to find a new place for L2 and R2. Notice that there is a vacancy on the right of 1 and the left of 3. Because all nodes in L2 are greater than 1 and less than 2, we can make L2 the right child of 1. Similarly, all the nodes in R2 are greater than 2 and less than 3; we can make R2 the left child of 3.

RL rotation

- We perform RL rotation on a node n, if
  - **balance_factor(n) < -1** and **n->right is left heavy**

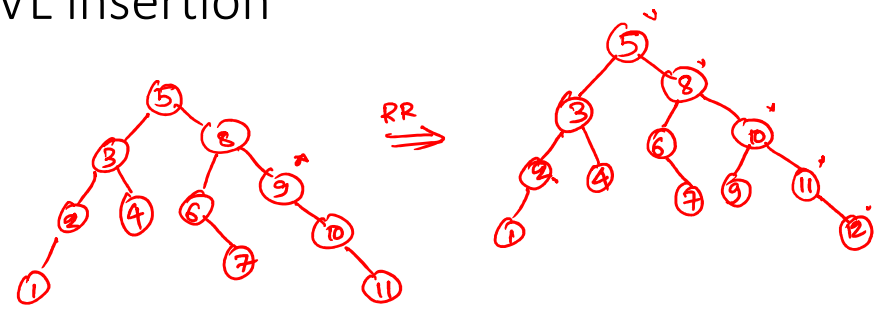
```
Left heavy : balance factor > 0
Right heavy : balance factor < 0
```

Insertion



In this example, we are inserting these elements individually in an AVL tree in the given order. After inserting a node n in the tree, when we return back from n to the root (like we did in the case of BST insertion), we check the balance at each intermediate node. If a node is imbalanced, we use the conditions that we discussed earlier to find its category, i.e., LL, RR, LR, and RL, and perform the rotation accordingly. After the rotation, we return the new root to its caller. We keep doing this for each intermediate node in the path until we reach the root. Notice that the maximum number of rotations in this scheme is equal to the height of the tree. Because each rotation requires a constant number of operations and the height of the tree is O(log(n)), the time complexity of the AVL insertion is O(log(n)). We will later show that the maximum number of rotations required for an insertion is actually one.

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

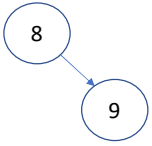
AVL insertion

8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion

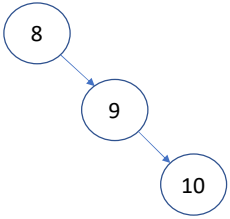
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

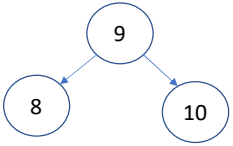
AVL insertion



need RR rotation

8 9 10 2 1 5 3 6 4 7 11 12

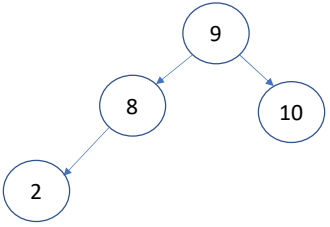
AVL insertion



after RR rotation

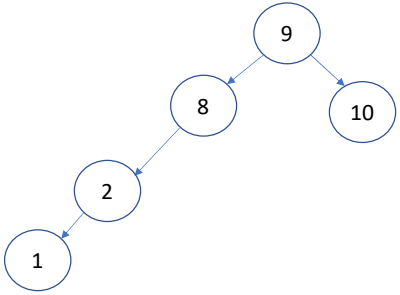
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

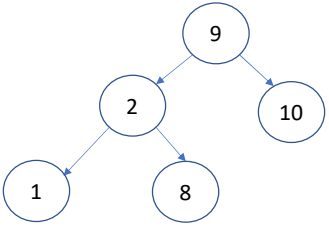
AVL insertion



need LL rotation

8 9 10 2 1 5 3 6 4 7 11 12

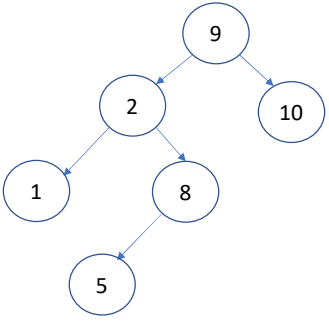
AVL insertion



after LL rotation

8 9 10 2 1 5 3 6 4 7 11 12

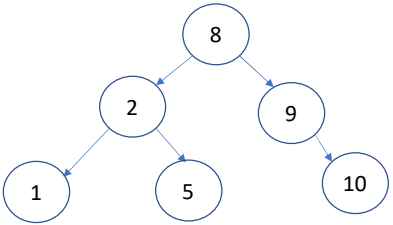
AVL insertion



need LR rotation

8 9 10 2 1 5 3 6 4 7 11 12

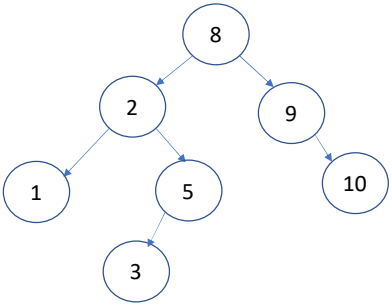
AVL insertion



after LR rotation

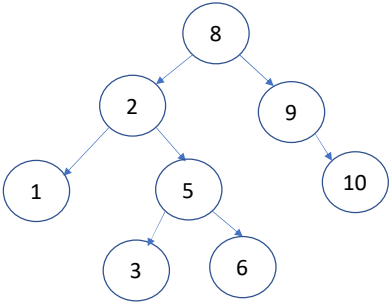
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



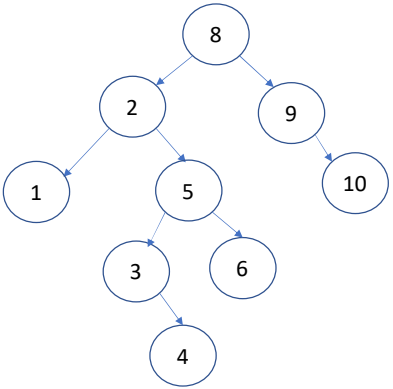
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



8 9 10 2 1 5 3 6 4 7 11 12

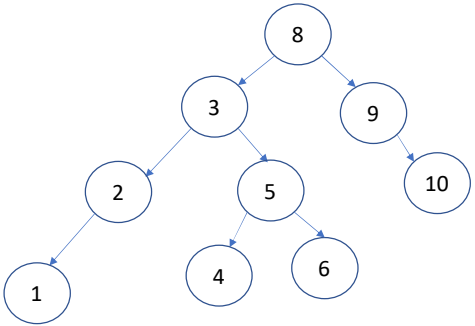
AVL insertion



need RL rotation

8 9 10 2 1 5 3 6 4 7 11 12

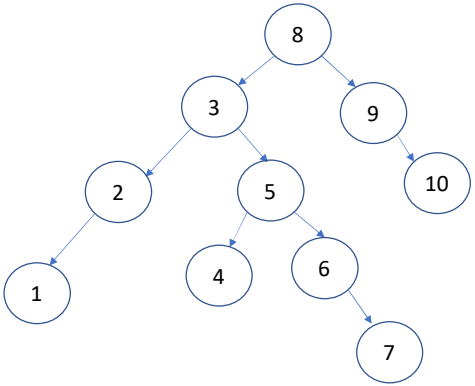
AVL insertion



after RL rotation

8 9 10 2 1 5 3 6 4 7 11 12

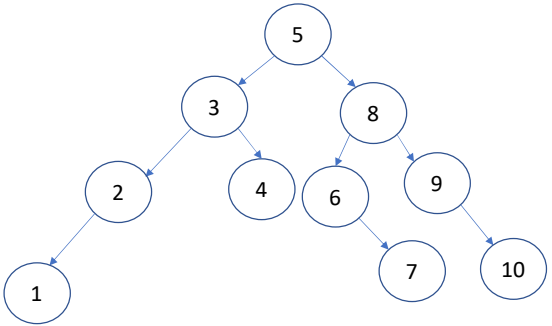
AVL insertion



need LR rotation

8 9 10 2 1 5 3 6 4 7 11 12

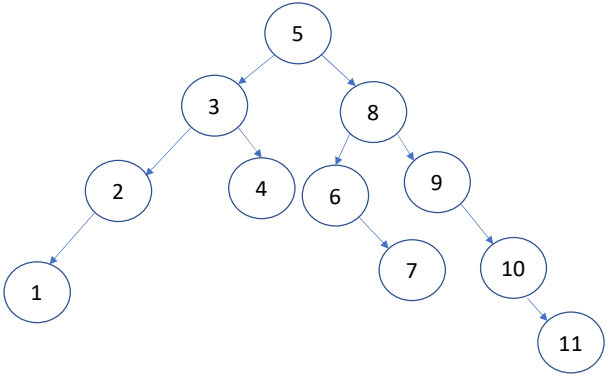
AVL insertion



after LR rotation

8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion

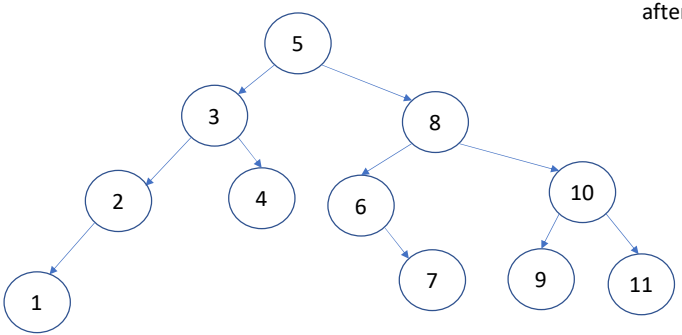


need RR rotation

8 9 10 2 1 5 3 6 4 7 11 12

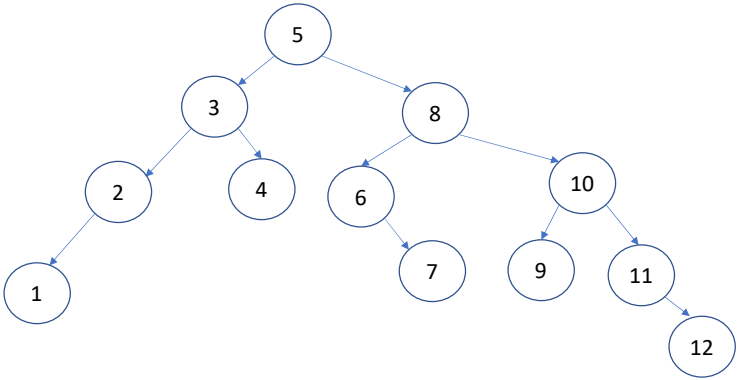
AVL insertion

after RR rotation



8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion

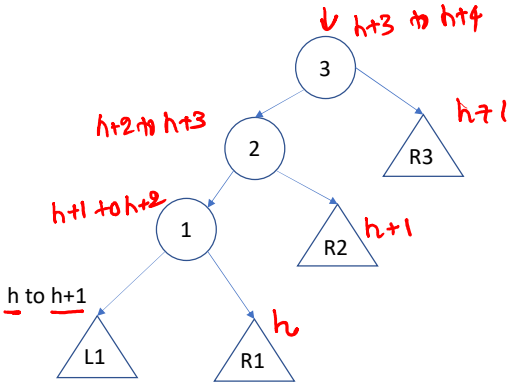


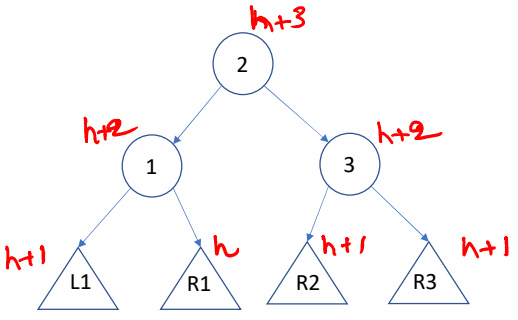
8 9 10 2 1 5 3 6 4 7 11 12

Insertion

- After insertion of node n, only the heights of the ancestors of n may change
  - As a consequence, some ancestors may violate the height-balance property
- What is the maximum number of single/double rotations required to make the tree balanced after insertion?
- What would be the time complexity of a single insertion?

Insertion (LL rotation)

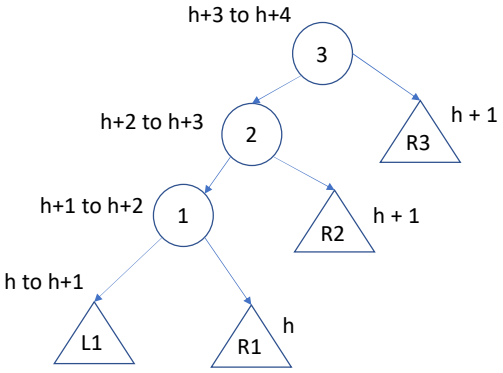


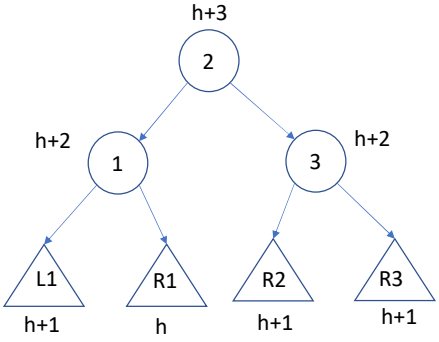


During insertion, an LL rotation at node n is triggered if we insert a node in the left subtree of the left child of n. Let's say after the insertion, the height of L1 becomes h+1 from h. Now a rotation at node 3 will be required if and only if the height of all intermediate nodes, i.e., 1, 2, and 3, change. The height of 1 would change due to the insertion in L1 only if the previous height of 1 was h+1. Notice that before the insertion, the height of 1 was greater than h because the height of L1 was h. Now the height of 1 can only change due to the insertion in L1 iff its previous height was h+1. Similarly, the height of 2 and 3 will change after the insertion iff their previous heights were h+2 and h+3. The new heights of 1, 2, and 3 are h+2, h+3, and h+4, respectively. Now, let's argue about the possible heights for R1. It can't be h+1 because, in that case, the previous height of 1 would be h+2, which is not the case here. Similarly, the height of R1 can't be h-1 because, in that case, node 1 becomes imbalanced after the insertion in L1, which is not the case here. Therefore, the only possible height for R1 is h. Similarly, the height of R2 can't be h+2 because, in that case, the height of 2 before the insertion would be h+3, which is not the case here. The height of R2 can't be h because, in that case, node 2 becomes imbalanced after the insertion, which is not the case here. Therefore, the only possible height for R2 is h+1. Let's look at node 3. The height of R3 can't be h+3 because, in that case, the height of node 3 before the insertion would be h+4, which is not the case. If the height of R3 is h+2, node 3

remains balanced after the insertion, which is not the case. Therefore, the only viable option for the height of R3 is h+1. After the rotation, the height of 1 is h+2 because the height of L1 is h+1; the height of 3 is h+2 because the heights of both R2 and R3 are h+1; the height of 2 is h+3 because the heights of both 1 and 3 are h+2. Notice that after the rotation, the height of the new root is the same as the height of the imbalanced node before the insertion. Therefore, no further rotation is required when we move upwards towards the root after the rotation.

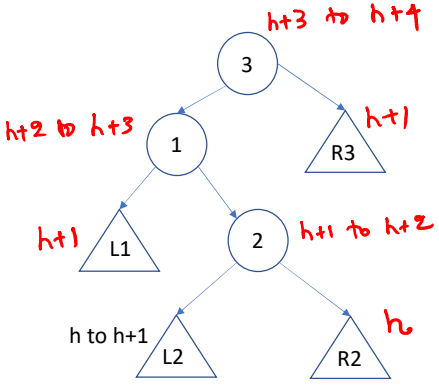
Insertion (LL rotation)

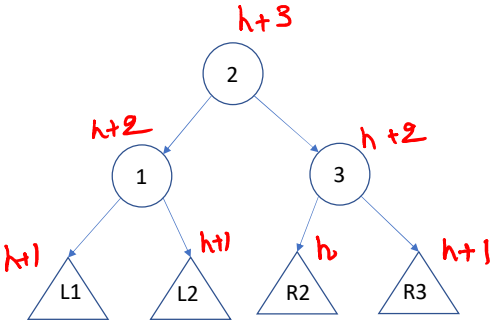




Height is restored as it was before the insertion. No further rotation is required.

Insertion (LR rotation)

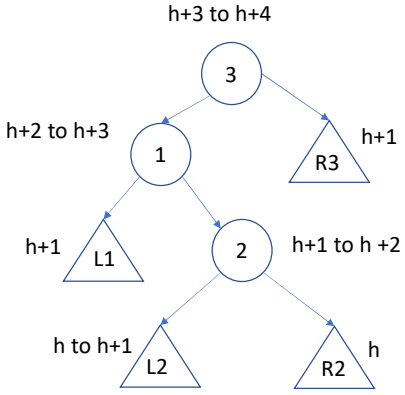


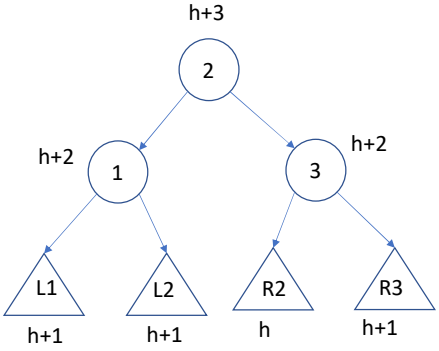


During insertion, an LR rotation at node n is triggered if we insert a node in the right subtree of the left child of n. Let's say after the insertion, the height of L2 becomes h+1 from h. Now a rotation at node 3 will be required if and only if the height of all intermediate nodes, i.e., 2, 1, and 3, change. The height of 2 would change due to the insertion in L2 only if the previous height of 2 was h+1. Notice that before the insertion, the height of 2 was greater than h because the height of L2 was h. Now the height of 2 can only change due to the insertion in L2 iff its previous height was h+1. Similarly, the height of 1 and 3 will change after the insertion iff their previous heights were h+2 and h+3. The new heights of 2, 1, and 3 are h+2, h+3, and h+4, respectively. Now, let's argue about the possible heights for R2. It can't be h+1 because, in that case, the previous height of 2 would be h+2, which is not the case here. Similarly, the height of R2 can't be h-1 because, in that case, node 2 becomes imbalanced after the insertion in L2, which is not the case here. Therefore, the only possible height for R2 is h. Similarly, the height of L1 can't be h+2 because, in that case, the height of 1 before the insertion would be h+3, which is not the case here. The height of L1 can't be h because, in that case, node 1 becomes imbalanced after the insertion, which is not the case here. Therefore, the only possible height for L1 is h+1. Let's look at node 3. The height of R3 can't be h+3 because, in that case, the height of node 3 before the insertion would be h+4, which is not the case. If the height of R3 is h+2, node 3

remains balanced after the insertion, which is not the case. Therefore, the only viable option for the height of R3 is h+1. After the rotation, the height of 1 is h+2 because the heights of both L1 and L2 are h+1; the height of 3 is h+2 because the height of R3 is h+1; the height of 2 is h+3 because the heights of both 1 and 3 are h+2. Notice that after the rotation, the height of the new root is the same as the height of the imbalanced node before the insertion. Therefore, no further rotation is required when we move upwards towards the root after the rotation.

Insertion (LR rotation)





Height is restored as it was before the insertion. No further rotation is required.

AVL insertion

- Each node also stores the height of that node
- The height of a leaf node is zero
- Insert the new node in the leaves similar to BST
- while returning from the leaf node to the root
  - Update the height of the nodes in the path
  - For each node n in the path from the leaf to the root until there is a rotation
    - Update the height of n
    - if a rotation is required (i.e., abs(balance_factor(n)) > 1), rotate the nodes and update the heights of the three nodes that are involved in the rotation
- Time complexity is O(log(n))

Type

```
struct node {
  int val;
  int height;
  struct node *left;
  struct node *right;
};
```

Helper functions

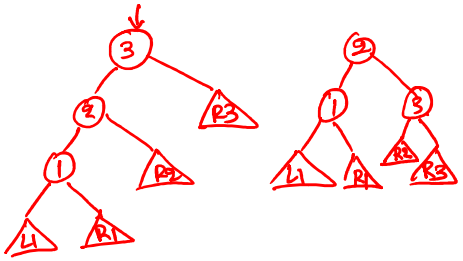
```
int get_balance(struct node *n) {
  int l_height = (n->left) ? n->left->height : -1;
  int r_height = (n->right) ? n->right->height : -1;
  return l_height - r_height;
}

void set_height(struct node *n) {
  int l_height = (n->left) ? n->left->height : -1;
  int r_height = (n->right) ? n->right->height : -1;
  n->height = (l_height > r_height) ? l_height + 1: r_height + 1;
}
```

(cond)? if-body : else-body syntax is syntactic sugar for a single line if and else bodies

LL rotation

```
struct node* LL(struct node *n) {
    struct node * new_root = n->left;
    n->left = new_root->right;
    new_root->right = n;
    set_height(n);
    set_height(new_root);
    return new_root;
}
```



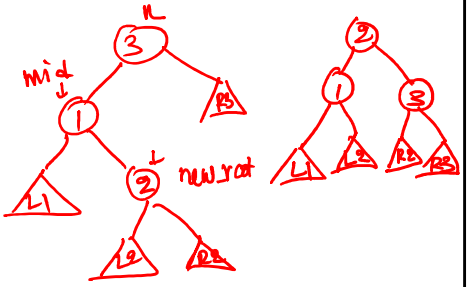
LL rotation

```
struct node* LL(struct node *n) {
  struct node *new_root = n->left;
  n->left = new_root->right;
  new_root->right = n;
  set_height(n);
  set_height(new_root);
  return new_root;
}
```

RR rotation

```
struct node* RR(struct node *n) {
  struct node *new_root = n->right;
  n->right = new_root->left;
  new_root->left = n;
  set_height(n);
  set_height(new_root);
  return new_root;
}
```

LR rotation




```
static struct node* LR(struct node *n) {
    struct node *mid = n->left;
    struct node *new_root = mid->right;
    n->left = new_root->right;
    mid->right = new_root->left;
    set_height(mid);
    set_height(n);
    set_height(new_root);
    return new_root;
}
```

```
new_root->right = n;
new_root->left = mid;
```

LR rotation

```
static struct node* LR(struct node *n) {
  struct node *mid = n->left;
  struct node *new_root = mid->right;
  mid->right = new_root->left;
  n->left = new_root->right;
  new_root->left = mid;
  new_root->right = n;
  set_height(n);
  set_height(mid);
  set_height(new_root);
  return new_root;
}
```

RL rotation

```
static struct node* RL(struct node *n) {
  struct node *mid = n->right;
  struct node *new_root = mid->left;
  mid->left = new_root->right;
  n->right = new_root->left;
  new_root->left = n;
  new_root->right = mid;
  set_height(n);
  set_height(mid);
  set_height(new_root);
  return new_root;
}
```

Insert

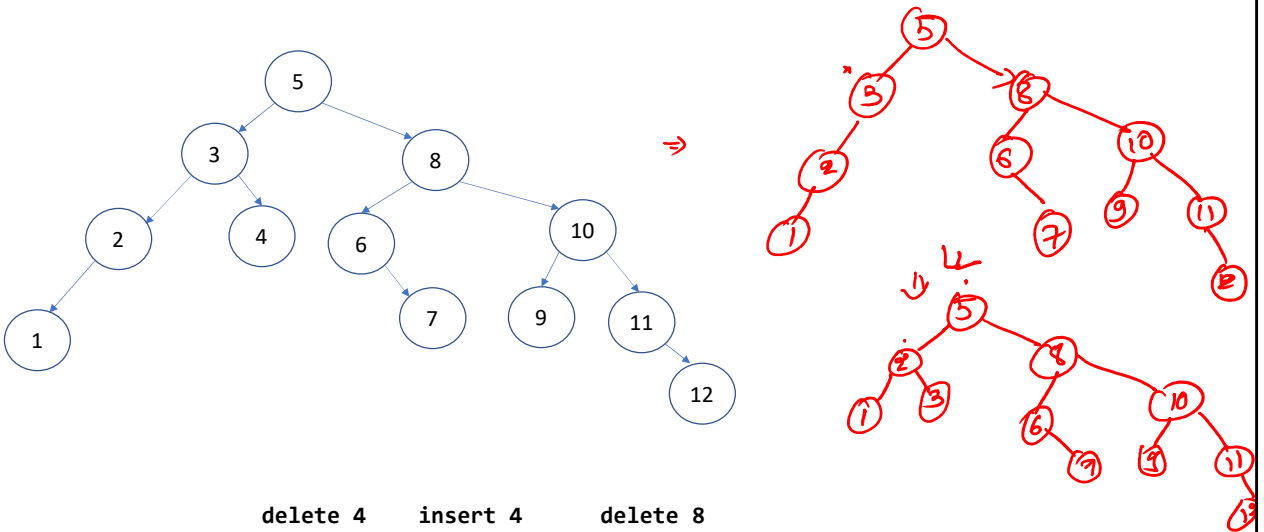
```
struct node* insert(struct node *root, int val) {
  struct node *n;

  if (root == NULL) {
    root = allocate_node(val);
    return root;
  }
  if (val >= root->val) {
    root->right = insert(root->right, val);
  }
  else {
    root->left = insert(root->left, val);
  }
  set_height(root);
  n = try_rotate(root);
  return n;
}
```

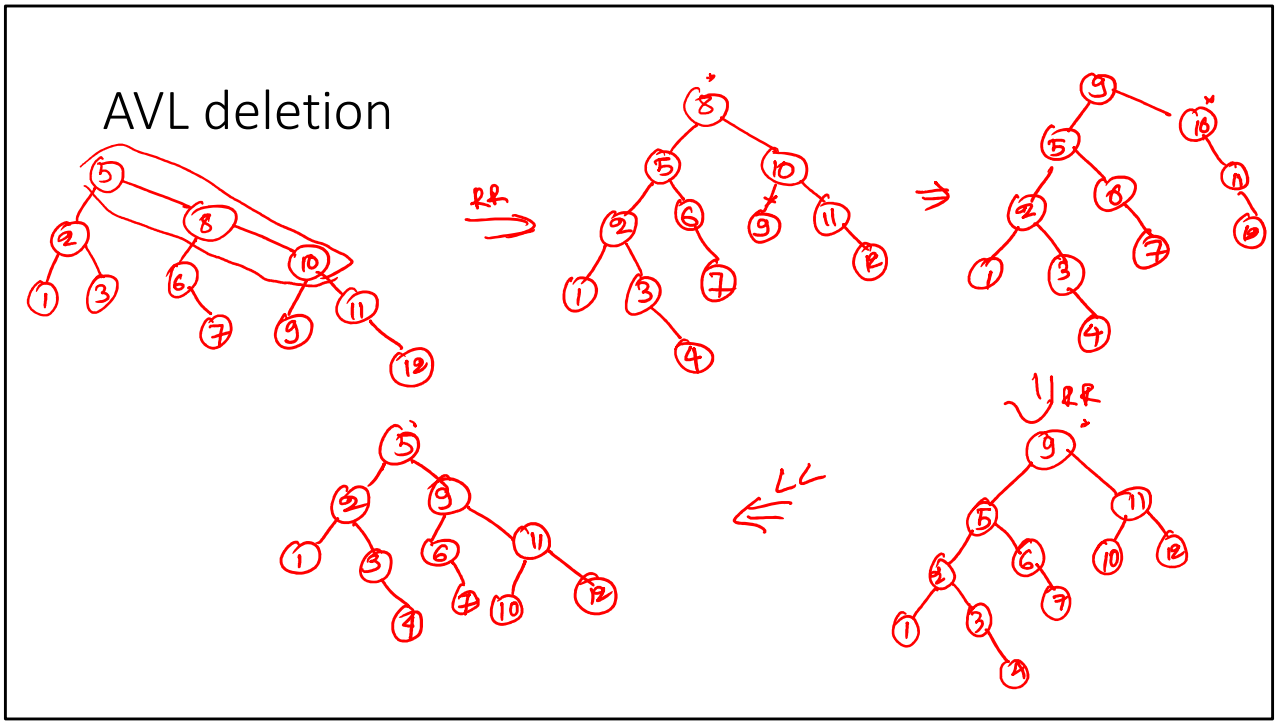
```
struct node* try_rotate(struct node *n) {
  int bal = get_balance(n);
  if (bal < -1) {
    bal = get_balance(n->right);
    if (bal > 0) {
      n = RL(n);
    }
    else {
      n = RR(n);
    }
  }
  else if (bal > 1) {
    bal = get_balance(n->left);
    if (bal < 0) {
      n = LR(n);
    }
    else {
      n = LL(n);
    }
  }
  return n;
}
```

Deletion

AVL deletion



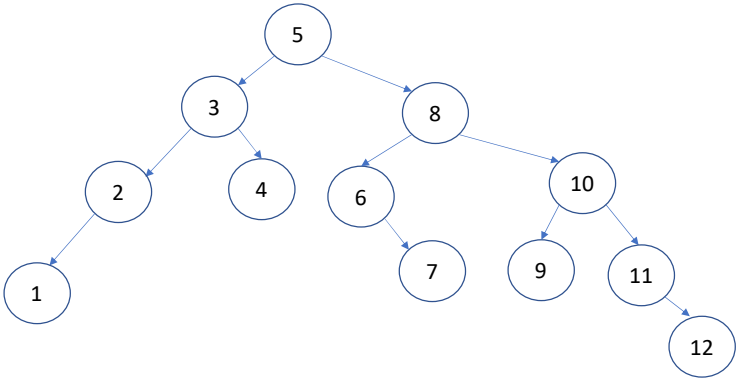
Deletion is similar to the BST except that the rotations may be required. After deleting 4, we will walk from 3 to root, checking the balance at each intermediate node and performing rotations if needed until we reach the root. Notice that in this case, after deleting 4, node 3 becomes imbalanced, so we perform an LL rotation at node 3. After the rotation, we check the balance of the parent node (5 in this case). Because 5 is also imbalanced, we perform a RR rotation at node 5. The final tree is shown on the next slide.



After deleting 4, the next goal is to insert 4 that don't require any rotations. Now, let's delete 8. Because 8 has two children, we will either delete the maximum node in the left subtree or the minimum node in the right subtree and copy the value of the deleted node to 8. Let's delete the minimum element in the right subtree, node 9, and replace 8 with 9. After deleting 9, we will walk from the original parent of 9 to the root and perform rotations if we encounter imbalanced nodes in the path. Notice that the parent of 9, node 10, is imbalanced now, so we perform a RR rotation. After the rotation, we check if the original parent of 10, node 9, is imbalanced. Because node 9 is imbalanced, we perform an LL rotation. Because node 9 was the root of the tree, the algorithm stops at this point.

AVL deletion

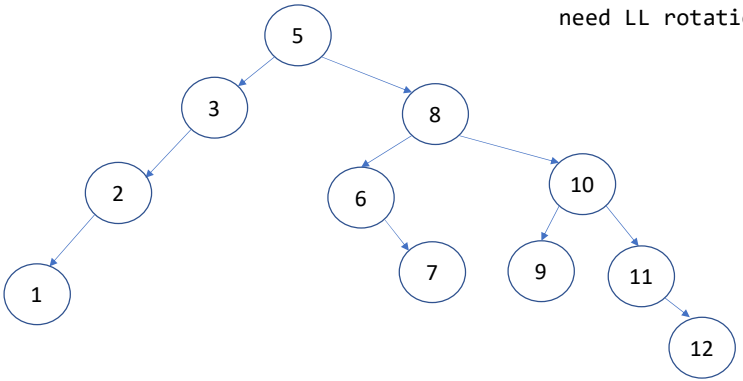
delete 4



8 9 10 2 1 5 3 6 4 7 11 12

AVL deletion

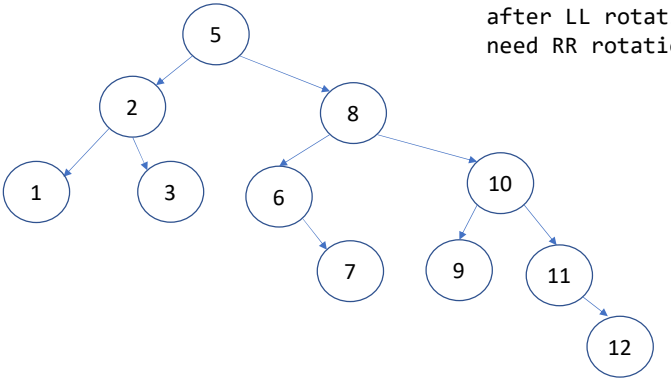
delete 4
need LL rotation at 3



8 9 10 2 1 5 3 6 ~~4~~ 7 11 12

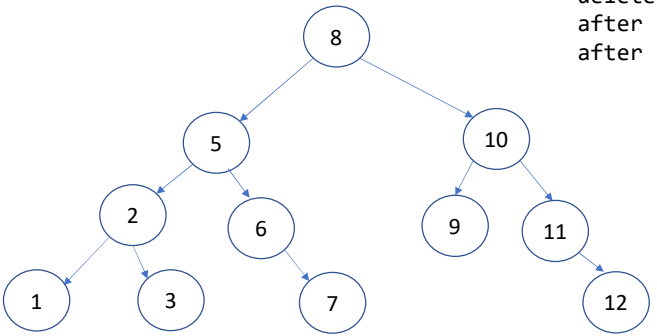
AVL deletion

delete 4
after LL rotation at 3
need RR rotation at 5



8 9 10 2 1 5 3 6 ~~4~~ 7 11 12

AVL deletion

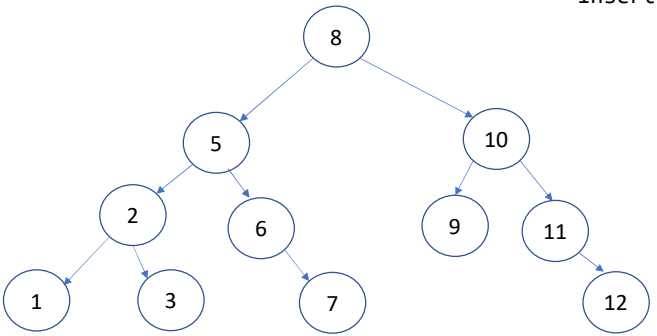


delete 4
after LL rotation at 3
after RR rotation at 5

8 9 10 2 1 5 3 6 ~~4~~ 7 11 12

AVL deletion

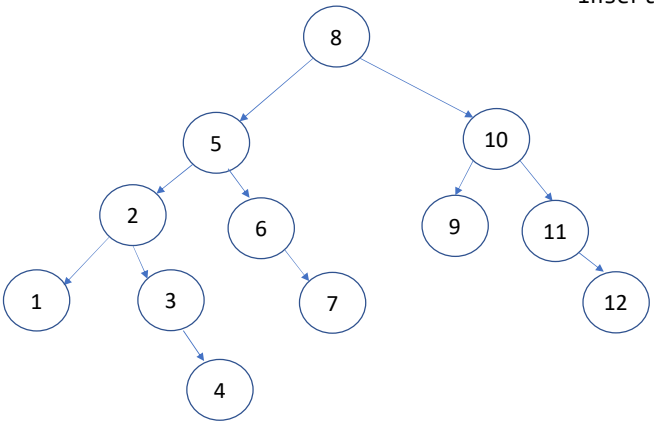
insert 4



8 9 10 2 1 5 3 6 ~~4~~ 7 11 12

AVL deletion

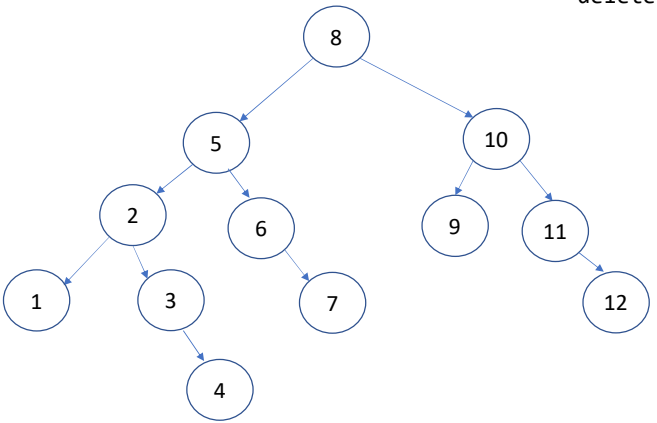
insert 4



8 9 10 2 1 5 3 6 ~~4~~ 7 11 12 4

AVL deletion

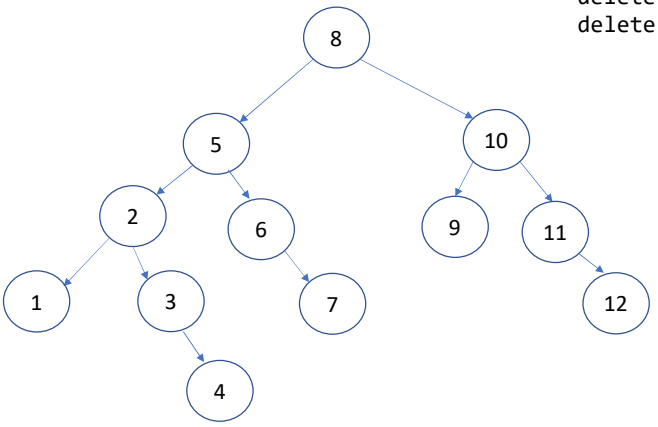
delete 8



8 9 10 2 1 5 3 6 ~~4~~ 7 11 12 4

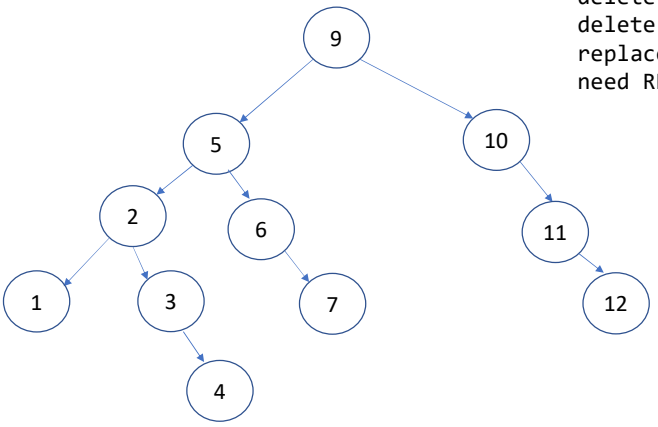
AVL deletion

delete 8
delete 9



8 9 10 2 1 5 3 6 ~~4~~ 7 11 12 4

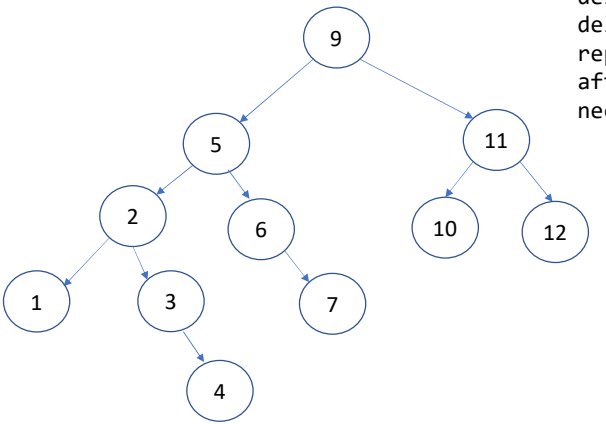
AVL deletion



delete 8
delete 9
replace 8 with 9
need RR rotation at 10

~~8~~ 9 10 2 1 5 3 6 ~~4~~ 7 11 12 4

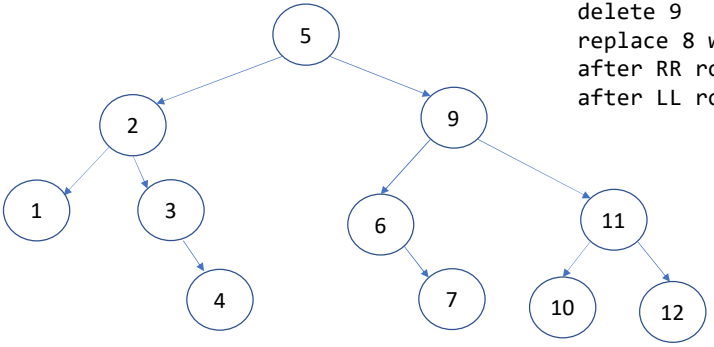
AVL deletion



delete 8
delete 9
replace 8 with 9
after RR rotation at 10
need LL rotation at 9

~~8~~ 9 10 2 1 5 3 6 ~~4~~ 7 11 12 4

AVL deletion



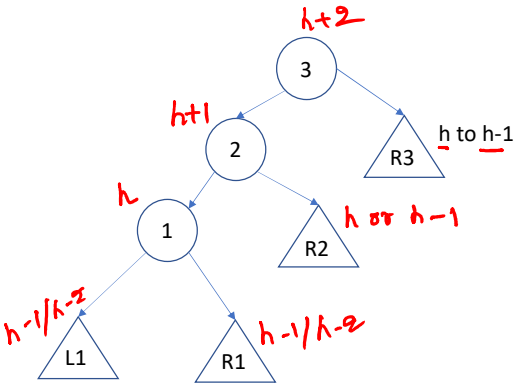

```
delete 8
delete 9
replace 8 with 9
after RR rotation at 10
after LL rotation at 9
```

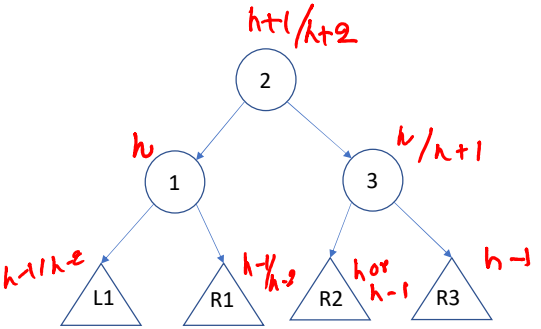
~~8~~ 9 10 2 1 5 3 6 ~~4~~ 7 11 12 4

Deletion

- After deletion of node n, only the heights of the ancestors of n in the original tree may change
  - As a consequence, some ancestors may violate the height-balance property
- What is the maximum number of single/double rotations required to make the tree balanced after deletion?
  - O(log n)
- What would be the time complexity of a single deletion?
  - O(log n), because at most O(log n) operations are required to find the node that needs to be deleted, and at most O(log n) rotations (a rotation needs constant time operations) are required.

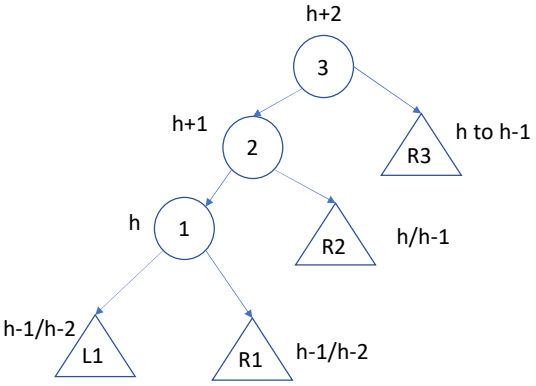
Deletion (LL rotation)

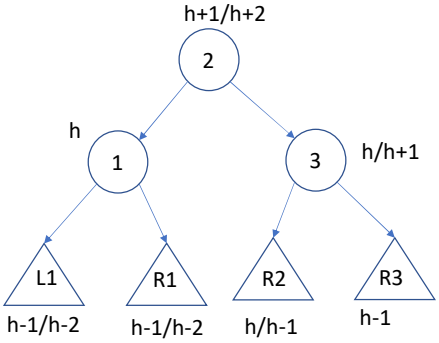




An LL rotation may be required at node 3 due to deletion if a node is deleted from the right subtree of 3, making it left heavy. Let's say we deleted a node from R3, and consequently, the height of R3 becomes h-1 from h. Because node 3 is imbalanced after the deletion, the height of 2 can't be h or h-1. The height of 2 can't be h-2 because, in that case, 3 was originally imbalanced. Therefore the only possible value for the height of 2 is h+1. Therefore, the height of 3 is h+2 before and after the deletion. Because we are doing LL rotation, node 2 is either left heavy or not heavy. If 2 is left heavy, in that case, the height of 1 will be h, and the height of R2 will be h-1. Notice that node 2 is not imbalanced. If node 2 is not heavy, then the heights of both node 2 and R2 will be h. The heights of L1 and R1 could be h-1 or h-2, but one of them must be h-1. After plugging the values of the heights of L1, R1, R2, and R3 in the rotated tree, we can compute the height of the new root. The height of 1 remains h because the height of either L1 or R1 must be h-1. The height of 3 will be h when the height of R2 is h-1, and it will be h+1 when the height of R2 is h. The height of 2 will be h+1 when the heights of both 1 and 3 are h. The height of 2 will be h+2 when the height of 3 is h+1. Because, after the rotation, the height of the new root can be h+1, which is one less than the original height of the imbalanced node, further rotations may be required.

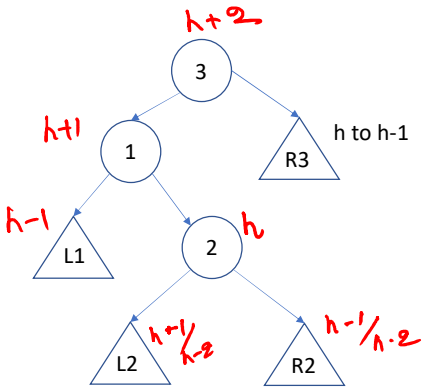
Deletion (LL rotation)

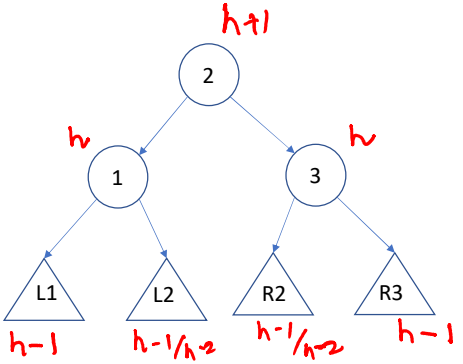




The height of the new root can be less than the original.
Further rotations may be required.

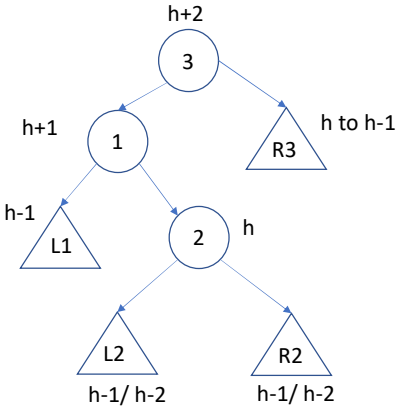
Deletion (LR rotation)

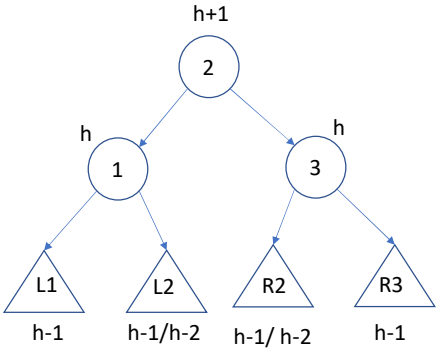




An LR rotation may be required at node 3 due to deletion if a node is deleted from the right subtree of 3, making it left heavy. Let's say we deleted a node from R3, and consequently, the height of R3 becomes h-1 from h. Because node 3 is imbalanced after the deletion, the height of 1 can't be h or h-1. The height of 1 can't be h-2 because, in that case, 3 was originally imbalanced. Therefore the only possible value for the height of 1 is h+1. Therefore, the height of 3 is h+2 before and after the deletion. Because we are doing LR rotation, node 1 is right heavy. If node 1 is right heavy, in that case, the height of 2 will be h, and the height of L1 will be h-1. Notice that node 1 is not imbalanced. The heights of L2 and R2 could be h-1 or h-2, but one of them must be h-1. After plugging the values of the heights of L1, L2, R2, and R3 in the rotated tree, we can compute the height of the new root. The height of 1 is h because the height of L1 is h-1. The height of 3 is h because the height of R3 is h-1. The height of 2 is h+1 because the heights of both 1 and 3 are h. Because, after the rotation, the height of the new root is h+1, which is one less than the original height of the imbalanced node, further rotations may be required. Let's also consider the case if we try LR rotation when node 1 is not heavy. If node 1 is not heavy, the heights of both L1 and 2 will be h. If we consider the height of L1 as h in the rotated tree, node 1 may become imbalanced because the height of L2 can also be h-2. Therefore, we perform an LL rotation when the left child of the imbalanced node is not heavy.

Deletion (LR rotation)





The height of the new root is not the same as the original.
Further rotations may be required.

AVL deletion

- Delete the node similar to BST
- while returning from the deleted node to the root
  - Update the height of the nodes in the path
  - For each node n in the path from the deleted node to the root
    - Update the height of n
    - if a rotation is required (i.e., abs(balance_factor(n)) > 1), rotate the nodes and update the heights of the three nodes that are involved in the rotation

- Time complexity is O(log(n))

AVL deletion

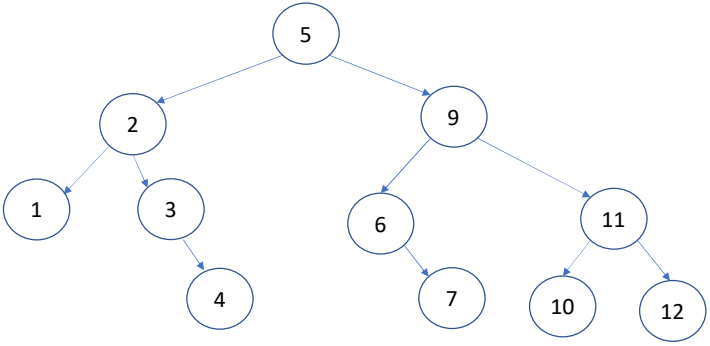
```
struct node* delete(struct node *root, int val) {
 if (root == NULL) {
  return NULL;
 }
 if (root->val == val) {
  if (root->left == NULL) {
   struct node *ret = root->right;
   free(root);
   return ret;
  }
  else if (root->right == NULL) {
   struct node *ret = root->left;
   free(root);
   return ret;
  }
  else if (root->right != NULL && root->left != NULL) {
   struct node *min_node = find_min(root->right);
   root->val = min_node->val;
   root->right = delete(root->right, min_node->val);
  }
 }
 else if (val > root->val) {
  root->right = delete(root->right, val);
 }
 else {
  root->left = delete(root->left, val);
 }
 set_height(root);
 struct node *n = try_rotate(root);
 return n;
}
```

Sorting

AVL sort

- To sort n numbers, insert all items in an AVL tree

AVL sort



AVL sort

- Insert all items in an AVL tree
- Perform the inorder traversal of the AVL tree
- Interestingly, the inorder traversal of a BST or AVL tree visits the nodes in the ascending order

AVL sort

- Time complexity of AVL sort

nlogn

- What would be time complexity of BST sort?

$O(n^2)$

The time complexity of the BST sort is n^2 because each insertion may take O(n) operations if the tree is skewed.