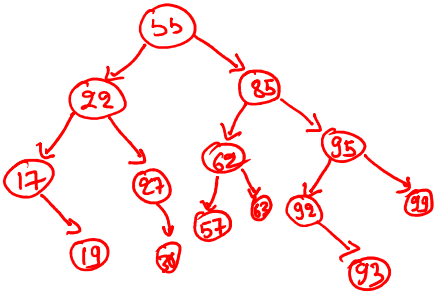
Today's topics

- Binary search trees
- AVL trees

Binary search tree

BST insertion

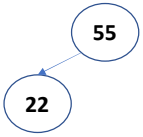
55 22 85 95 92 17 99 62 93 19 27 57 63 30



BST insertion

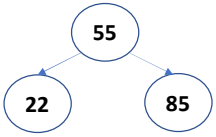
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



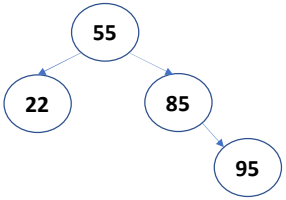
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



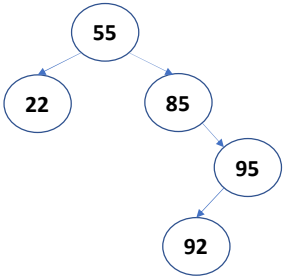
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



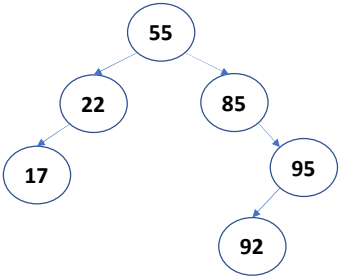
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



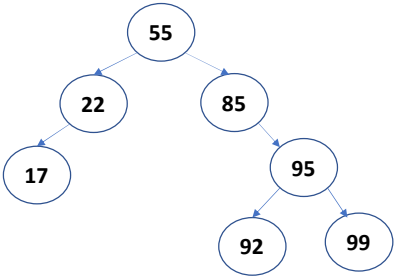
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



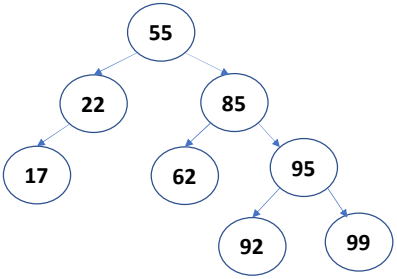
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



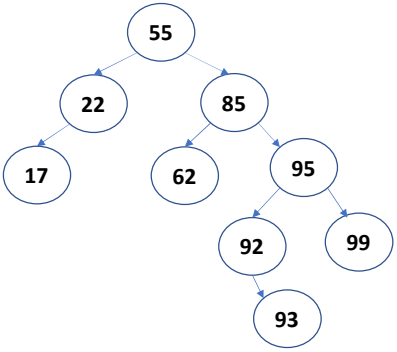
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



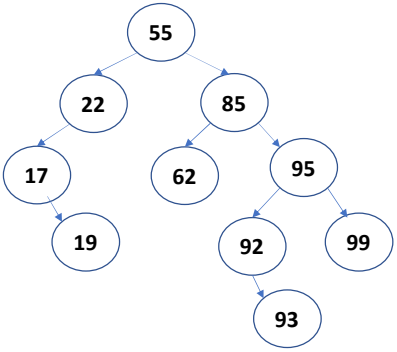
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



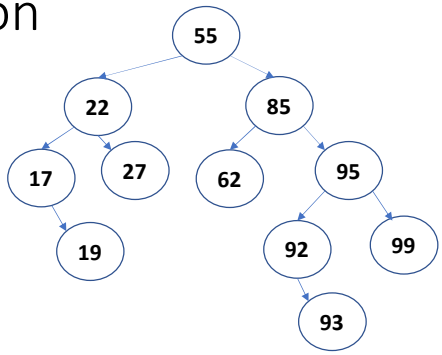
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



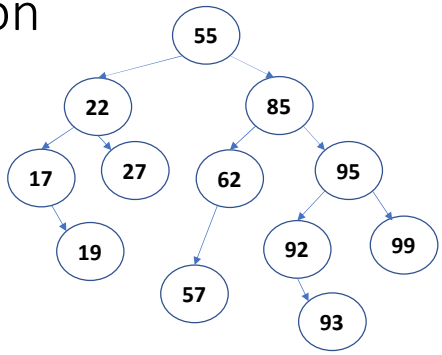
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



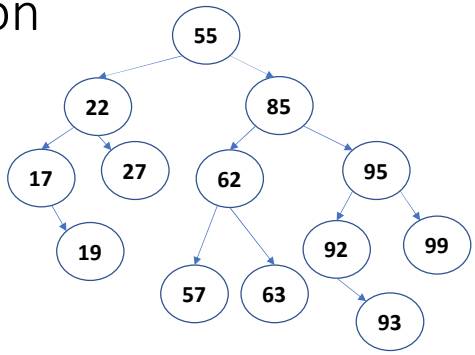
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



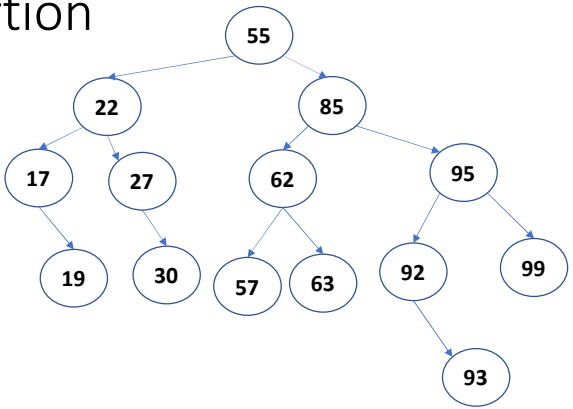
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST insertion (recursive algorithm)

- insert returns the root (possibly new) of the tree after inserting a node with a given value (val) (we are assuming val is also the key)
- Recursive step
- Base step

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

The insert routine returns the address of the root (possibly new) after the insertion. If the tree is empty, we allocate a new node with a given value and return its address. Otherwise, if the val is less than the key stored in the root, we recursively insert in the left subtree that returns the root (possibly new) of the left subtree, which is stored in root->left. If the val is greater than the key stored in root, we recursively insert in the right subtree and update root->right.

BST insertion (recursive algorithm)

- insert returns the root (possibly new) of the tree after inserting a node with a given value (val) (we are assuming val is also the key)

Recursive step

- If `val` is greater than or equal to the key stored at `root`
  - recursively insert in the right subtree
  - store root (possibly new) of the right subtree after insertion in root->right
- If `val` is less than the key stored at `root`
  - recursively insert in the left subtree
  - store root (possibly new) of the left subtree after insertion in root->left

Base step

- if the tree is empty (root == NULL), allocate and return a new node that stores val

BST insertion

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

  if (root == NULL) {
    /* allocate_node sets the left and right fields to NULL */
    root = allocate_node(val);
    return root;
  }

  if (val >= root->val) {
    root->right = insert(root->right, val);
  }
  else {
    root->left = insert(root->left, val);
  }
  return root;
}
```

Time complexity

- Complete binary tree $O(\log n)$
- Skewed tree $O(n)$

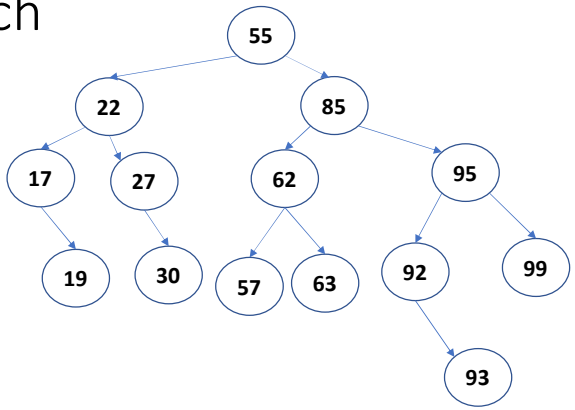
Search

Search

- The search routine returns a node in the BST that contains a given key value

- If none of the nodes contains the key, the search returns NULL

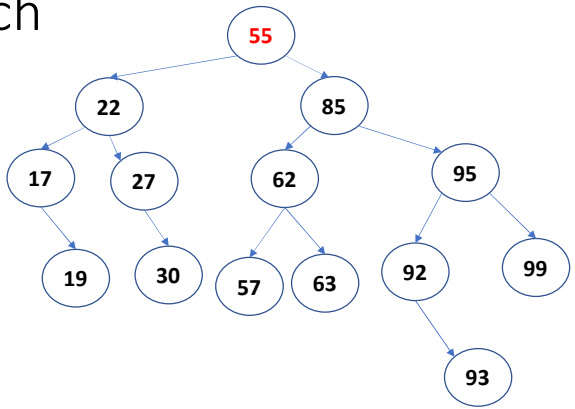
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

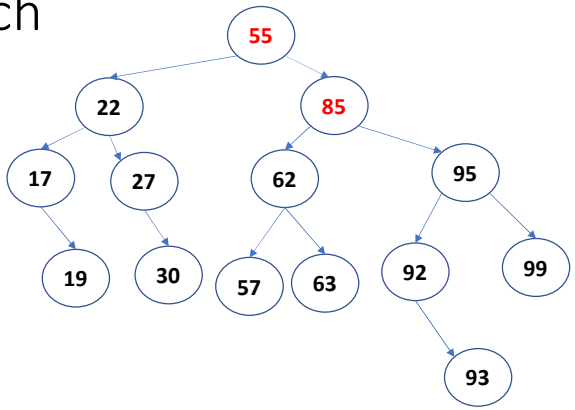
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

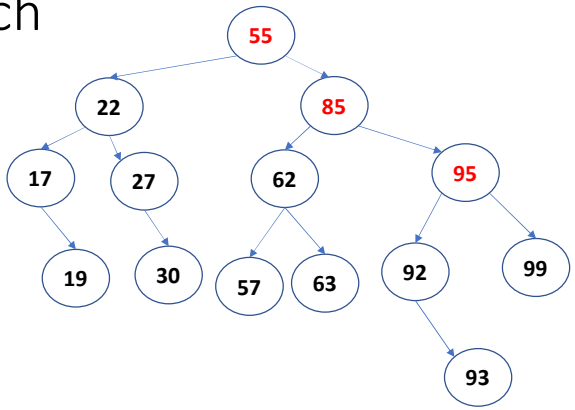
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

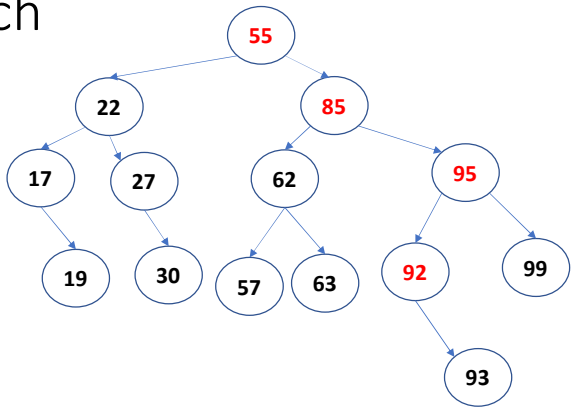
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

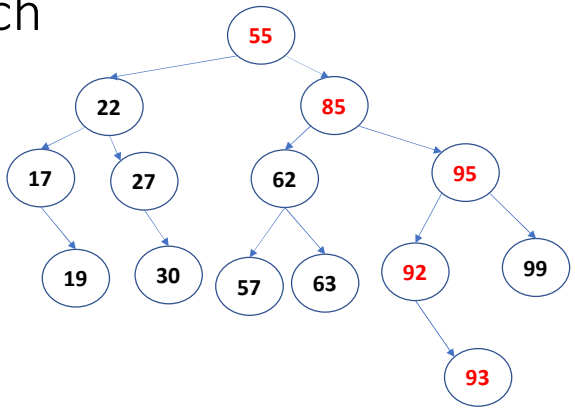
BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST search



search 93

55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST search (recursive algorithm)

- `search` returns a node that contains the key (`val`) being searched
- if `val` is not stored in any of the nodes, the `search` returns `NULL`
- Recursive step

```
search (root, val) {
    if (root->val > val)
        return search (root->left, val);
    else if (root->val < val)
        return search (root->right, val);
    else
        return root;
}
```

- Base step

```
if (root == NULL)
    return NULL
```

The search routine returns the address of the node that contains a given key. If the tree is empty, we return NULL. Otherwise, if the key is less than the key stored in the root, search returns the result of the search operation in the left subtree. If the key is greater than the key stored in the root, search returns the result of the search operation in the right subtree. Otherwise, if the key is the same as the root, it returns the address of the root.

BST search (recursive algorithm)

- `search` returns a node that contains the key (`val`) being searched
- if `val` is not stored in any of the nodes, the `search` returns `NULL`

Recursive step

- If `val` is greater than the key stored at `root`
  - recursively search in the right subtree
- If `val` is less than the key stored at `root`
  - recursively search in the left subtree

Base step

- if tree is empty (root == NULL), return `NULL`
- If the value of the key stored at `root` is `val`, `return root`

BST search

```
struct node *search(struct node *root, int val) {
  if (root == NULL) {
    return NULL;
  }
  if (root->val == val) {
    return root;
  }
  else if (val > root->val) {
    return search(root->right, val);
  }
  else {
    return search(root->left, val);
  }
}
```

Time complexity

- Complete binary tree $O(\log n)$
- Skewed tree $O(n)$

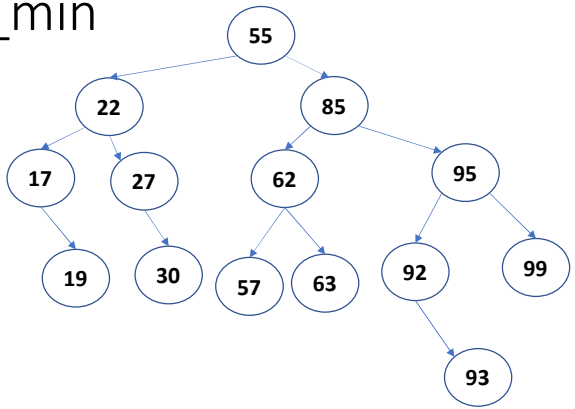
Find minimum

Find minimum (find_min)

- The goal of `find_min` is to find a node that stores the key with the minimum value in a BST

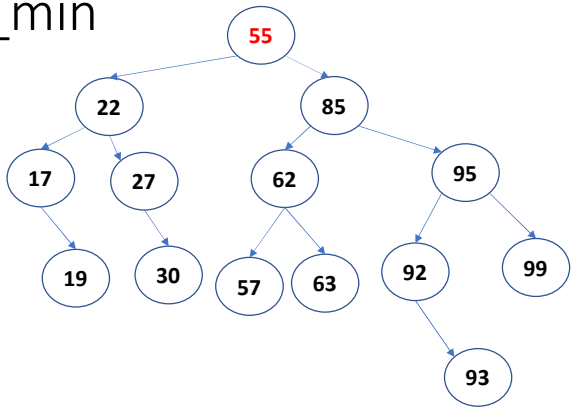
- The leftmost element in a BST contains the minimum key

BST find_min



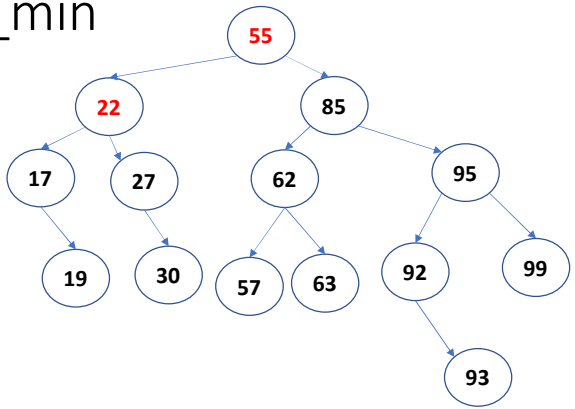
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST find_min



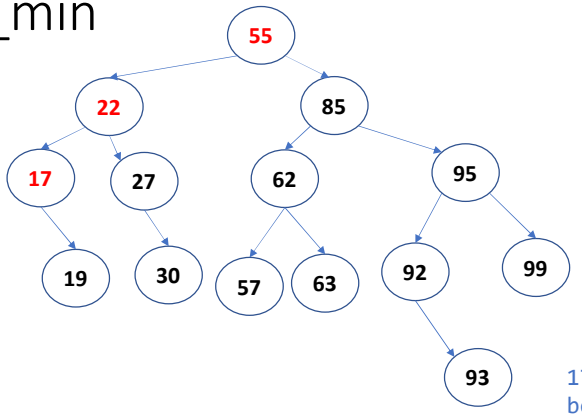
55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST find_min



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST find_min



17 is the minimum node because it has no left subtree.

55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST find_min

```
struct node* find_min(struct node *root) {
  if (root == NULL) {
     return NULL;
  }
  if (root->left == NULL) {
    return root;
  }
  else {
    return find_min(root->left);
  }
}
```

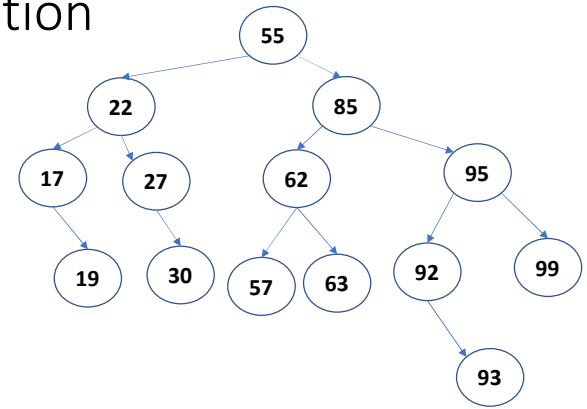
find_min returns the leftmost node.

Deletion

Delete

- The goal of the delete operation is to delete a node from a BST that stores the input key

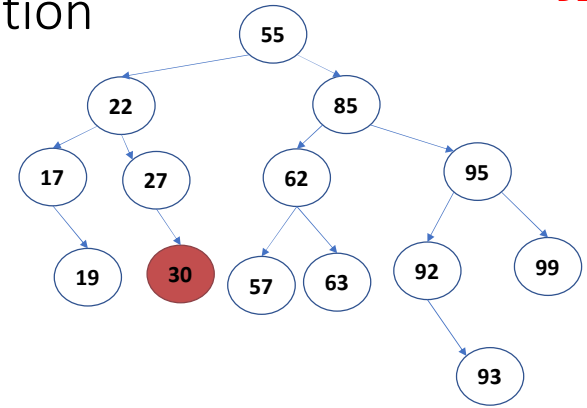
BST deletion



55 22 85 95 92 17 99 62 93 19 27 57 63 30

BST deletion

DELETING A LEAF NODE

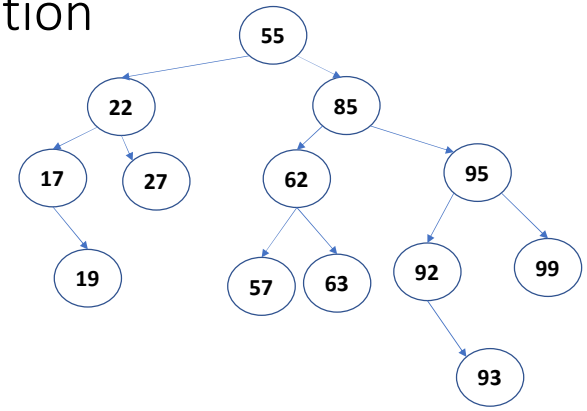


delete 30

55 22 85 95 92 17 99 62 93 19 27 57 63 30

The goal of the deletion is to delete a node with a given key value. If the node that needs to be deleted is a leaf, we simply unlink it from the parent by setting its reference in the parent to NULL.

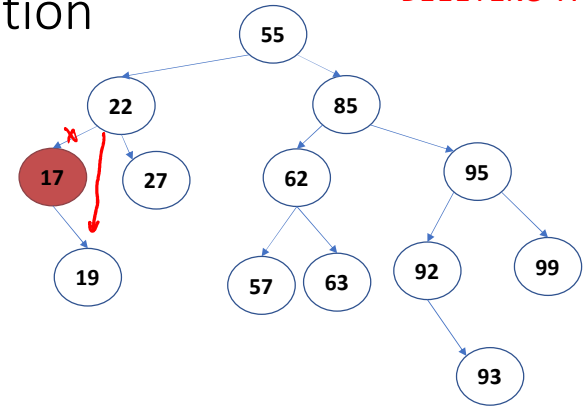
BST deletion



55 22 85 95 92 17 99 62 93 19 27 57 63 ~~30~~

BST deletion

DELETING A NODE WITH ONE CHILD

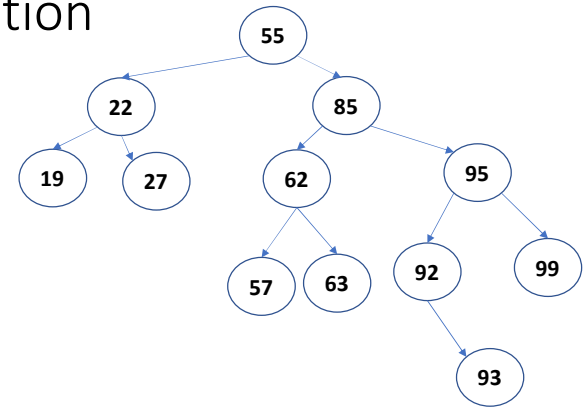


delete 17

55 22 85 95 92 17 99 62 93 19 27 57 63 ~~30~~

If the node that needs to be deleted has only one child, we replace its reference in the parent with the address of the child.

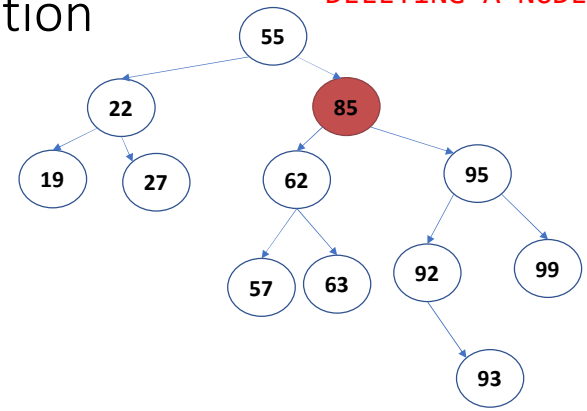
BST deletion



55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion

DELETING A NODE WITH TWO CHILDREN

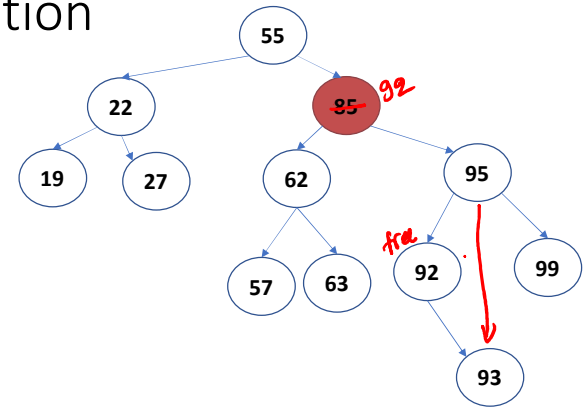


delete 85

55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

Now that we know how to delete a node with one or zero children let's look at the case when the node we want to delete, say n, has two children. One way of deleting n is to copy the value of some other node in n and delete the other node, ensuring that the key/value stored in n is not present in the tree. But we can't copy an arbitrary node; otherwise, the BST property will not hold at n. Let's assume that the smallest value in the right subtree of n is stored in node p. To delete n, we copy the value of p to n and delete p. This will not violate the BST property at n. Notice that p is the leftmost node in the right subtree and can have at most one child. Instead of copying the value of the p in n, we can also replace n with p directly by removing p from the right subtree, replacing the reference to n in its parent with p, and updating left and right children in p to point to the left and right children of n. This scheme is better when a node's size is large, and copying could be expensive.

BST deletion

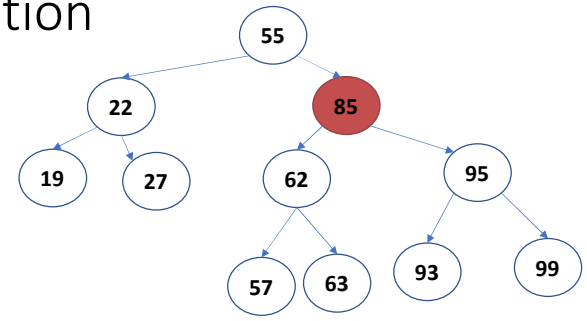


delete 85

delete a node with the minimum key in the right subtree (i.e., 92)

55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion

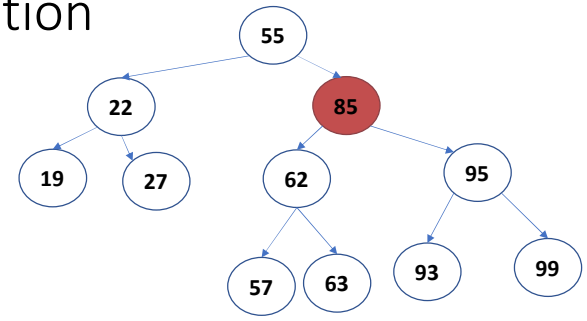


delete 85

delete a node with the minimum key in the right subtree (i.e., 92)

55 22 85 95 ~~92~~ ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion



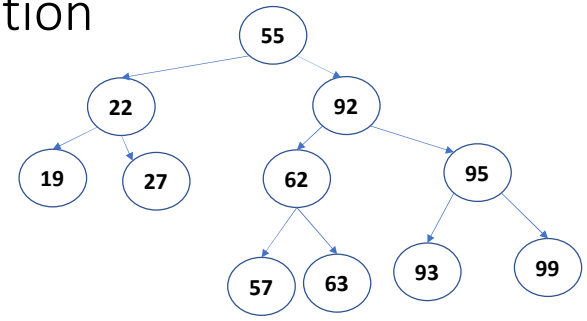
delete 85

delete a node with the minimum key in the right subtree (i.e., 92)

replace 85 with 92

55 22 85 95 ~~92~~ ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion



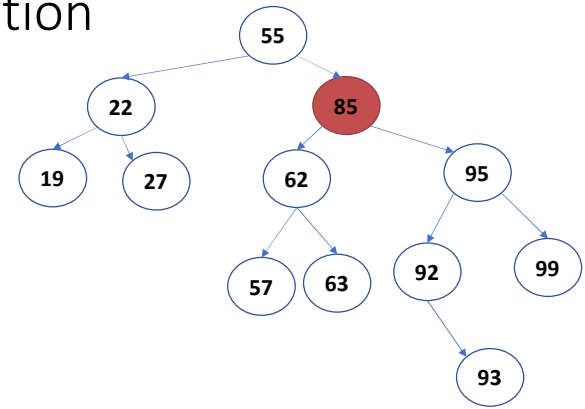
delete 85

delete a node with the minimum key in the right subtree (i.e., 92)

replace 85 with 92

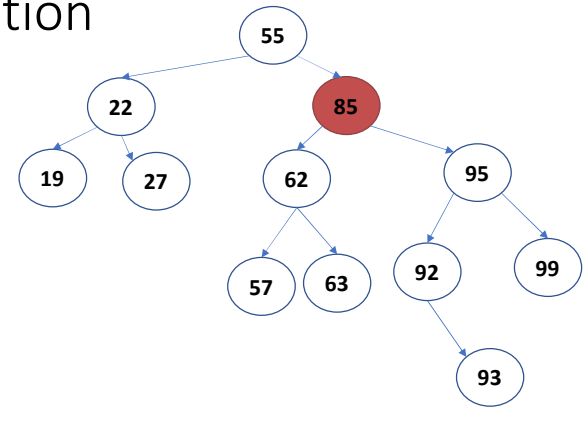
55 22 ~~85~~ 95 ~~92~~ ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion



55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion



delete 85

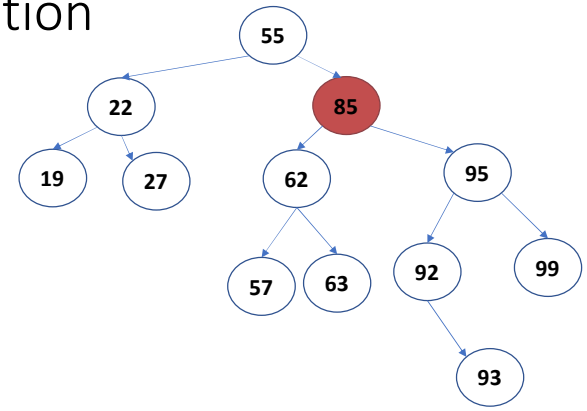
replace the value of the node with the value of the minimum key node in the right subtree (i.e., 92)

What is the other alternative?

55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

Another alternative is to replace the node with the largest node in the left subtree.

BST deletion



delete 85

replace the value of the node with the value of the minimum key node in the right subtree (i.e., 92)

What is the other alternative?

replace the value of the node with the value of the maximum key node in the left subtree (i.e., 63)

55 22 85 95 92 ~~17~~ 99 62 93 19 27 57 63 ~~30~~

BST deletion (recursive algorithm)

- delete returns the root (possibly new) of the tree after deleting a node that contains a given key (val)
- Recursive step

- Base step

BST deletion (recursive algorithm)

- delete returns the root (possibly new) of the tree after deleting a node that contains a given key (val)

Recursive step

- If `val` is greater than the value stored at `root`, recursively delete from the right subtree
  - store the root (possibly new) of the right subtree after the deletion in root->right
- If `val` is less than the value stored at `root`, recursively delete from the left subtree
  - store the root (possibly new) of the left subtree after the deletion in root->left

Base step

- if tree is empty (root == NULL) then val is not present, return NULL
- if val is stored at root
  - if root has both left and right children
    - recursively delete the node with the minimum key from the right subtree and copy the value of the deleted node in root
  - if root has at most one child
    - if root is a leaf node, return the empty tree
    - if root is not a leaf, return the child node

BST deletion

```
1.  struct node* delete(struct node *root, int val) {
2.    if (root == NULL) {
3.      return NULL;
4.    }
5.    if (root->val == val) {
6.      if (root->left == NULL) {
7.        struct node *ret = root->right;
8.        free(root);
9.        return ret;
10.     }
11.     else if (root->right == NULL) {
12.       struct node *ret = root->left;
13.       free(root);
14.       return ret;
15.     }
16.     else if (root->right != NULL && root->left != NULL) {
17.       struct node *min_node = find_min(root->right);
18.       root->val = min_node->val;
19.       root->right = delete(root->right, min_node->val);
20.       return root;
21.     }
22.   }
23.   else if (val > root->val) {
24.     root->right = delete(root->right, val);
25.   }
26.   else {
27.     root->left = delete(root->left, val);
28.   }
29.   return root;
30. }
```

As in the case of insertion, the delete routine returns the address of the root (possibly new) after the delete. If the node that needs to be deleted has only one child (line-6,11), we make the child the new root of the subtree that is returned to its caller. If the node that needs to be deleted has two children (line-16), the address of the root is not changed. In line-17, we find the minimum node in the right subtree; in line-18, we copy the value of the minimum node in the root; in line 19, we delete the minimum node from the right subtree and store the address of the root (possibly new) of the right subtree after the deletion in the root->right. Finally, the root is returned to its caller.

Time complexity

- Complete binary tree $O(\log n)$
- Skewed tree $O(n)$

AVL tree

References

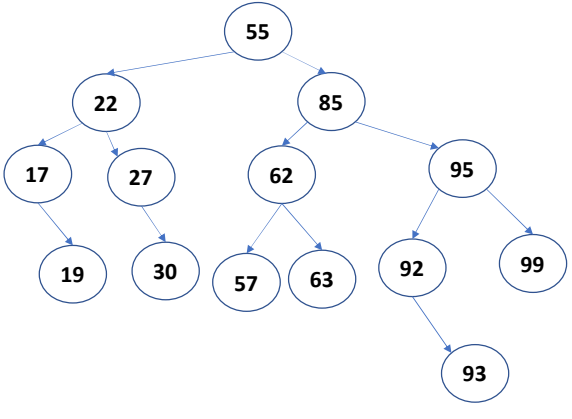
- Section-4.4 from Mark Allen Weiss

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

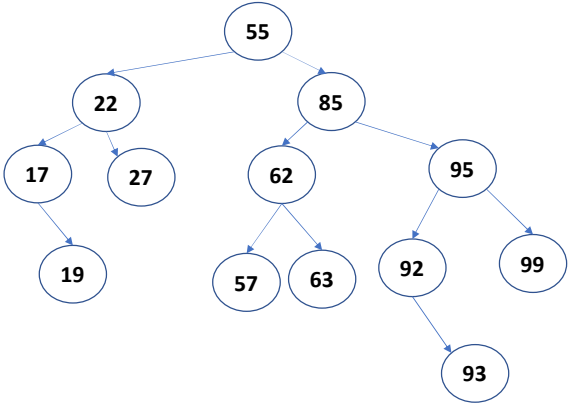
AVL tree



Is this an AVL tree? yes

This is an AVL tree because the height-balance property is satisfied at every node.

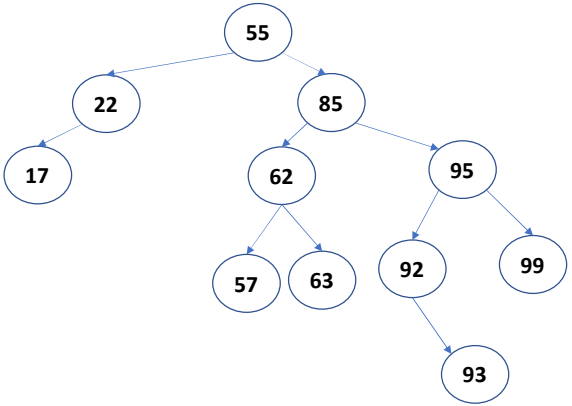
AVL tree



Is this an AVL tree? Yes

This is an AVL tree because the height-balance property is satisfied at every node.

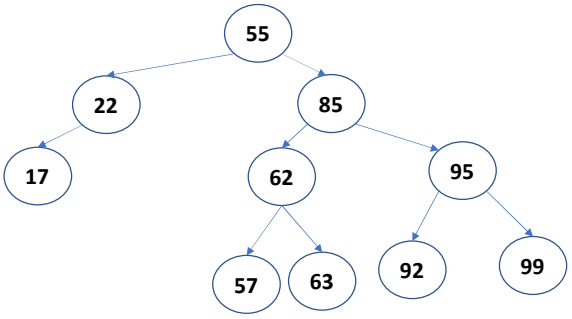
AVL tree



Is this an AVL tree? No

This is not an AVL tree because the height of the left subtree of 55 is 1, and the height of the right subtree of 55 is 3. So, the height balance property is not satisfied.

AVL tree



Is this an AVL tree? Yes

Balance factor

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

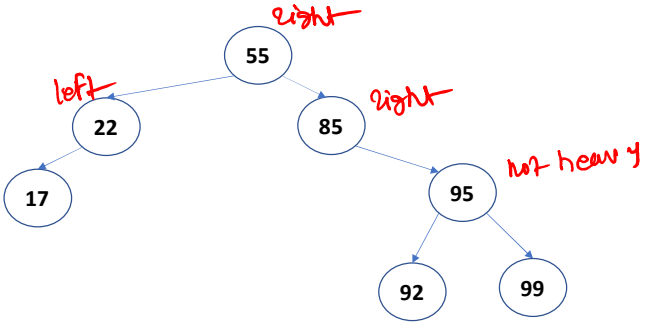
- A node is not balanced
  - if balance_factor < -1 or balance_factor > 1

- 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

AVL tree



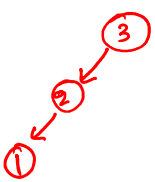
Identify which nodes are left heavy, right heavy, and not heavy.

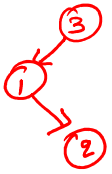
A node is heavy if the balance factor is not zero.

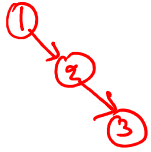
AVL tree

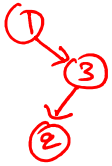
- What are all possible BSTs that store 1, 2 3?
- Which ones of them are AVL trees?
- How can we convert a non-AVL tree to an AVL tree?

AVL tree







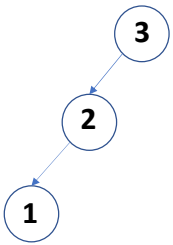


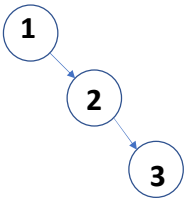
What are all possible BSTs that store 1, 2 3?

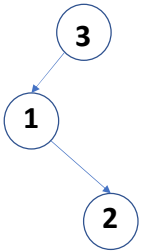
Which ones of them are AVL trees?

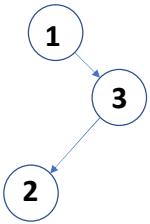
How can we convert a non-AVL tree to an AVL tree?

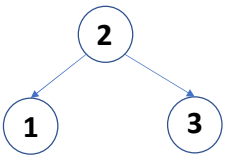
AVL tree











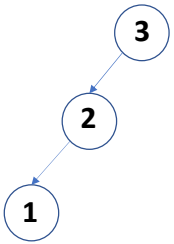
What are all possible BSTs that store 1, 2 3?

Which ones of them are AVL trees?

How can we convert a non-AVL tree to an AVL tree?

Out of these five possible BSTs for the given numbers, the below one is an AVL tree. If a BST is in the form of any of the above ones, the tree is converted into the tree at the bottom.

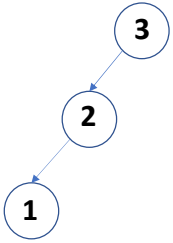
LL rotation

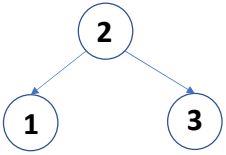


How can we convert this BST to an AVL tree?

After inserting node 1, node 3 becomes imbalanced. Here imbalance happened at 3 after inserting 2 in the left of its left child. LL stands for imbalance at node n due to an insertion in the left subtree of the left child of n.

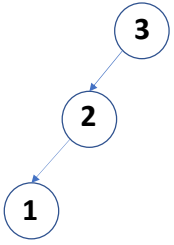
LL rotation

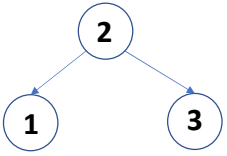




How can we identify the LL case?

LL rotation



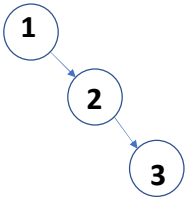


How can we identify the LL case?

```
if (balance_factor(root) > 1 and root->left is not right heavy)
```

In the case of LL rotation, to balance, we need to rotate the root node once in the right direction. Rotating a node in the right direction can be visualized as attaching a thread to the node and pulling it in the right downward direction such that the left child becomes the root.

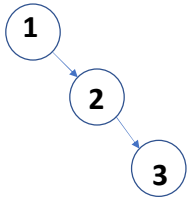
RR rotation

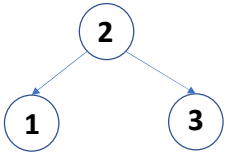


How can we convert this BST to an AVL tree?

RR stands for imbalance at node n due to an insertion in the right subtree of the right child of n.

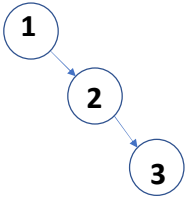
RR rotation

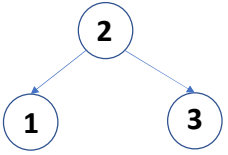




How can we identify the RR case?

RR rotation



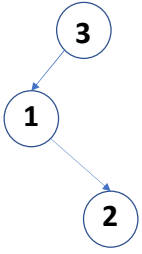


How can we identify the RR case?

```
if (balance_factor(root) < -1 and root->right is not left heavy)
```

In the case of RR rotation, to balance, we need to rotate the root node once in the left direction. Rotating a node in the left direction can be visualized as attaching a thread to the node and pulling it in the left downward direction such that the right child becomes the root.

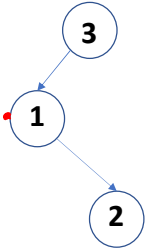
LR rotation

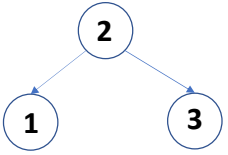


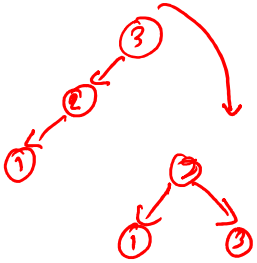
How can we convert this BST to an AVL tree?

LR stands for imbalance at node n due to an insertion in the right subtree of the left child of n.

LR rotation

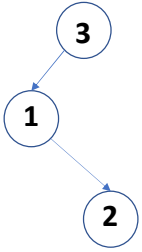


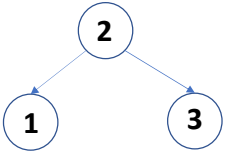




How can we identify the LR case?

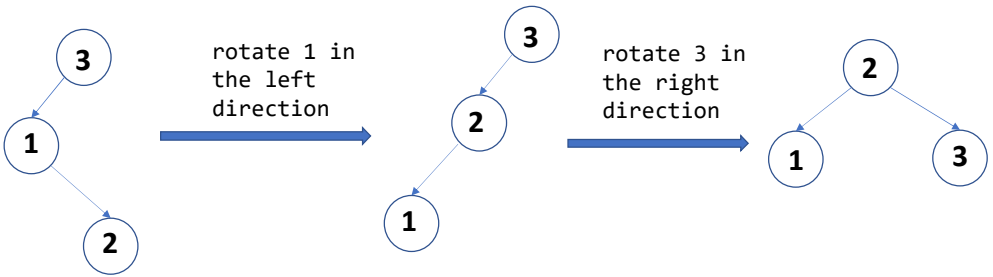
LR rotation





How can we identify the LR case?
if (balance_factor(root) > 1 and root->left is right heavy)

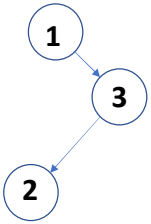
LR rotation




```
How can we identify the LR case?
if (balance_factor(root) > 1 and root->left is right heavy)
```

In the case of LR rotation, first, we need to rotate the left child once in the left direction and then rotate the root of the resulting tree in the right direction. As discussed earlier, rotating a node in the left direction can be visualized as attaching a thread to the node and pulling it in the left downward direction such that the right child becomes the root. Rotating a node in the right direction can be visualized as attaching a thread to the node and pulling it in the right downward direction such that the left child becomes the root.

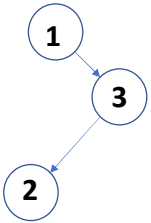
RL rotation

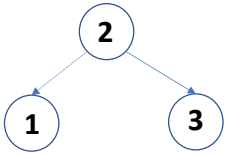


How can we convert this BST to an AVL tree?

RL stands for imbalance at node n due to the insertion in the left subtree of the right child of n.

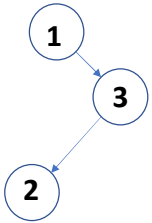
RL rotation

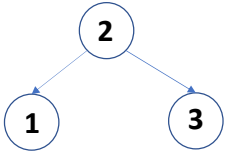




How can we identify the RL case?

RL rotation

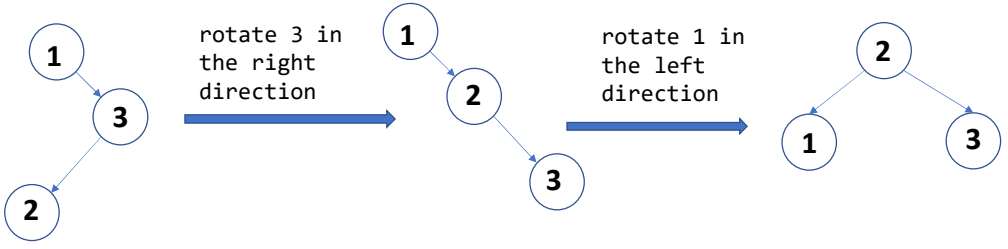




How can we identify the RL case?

```
if (balance_factor(root) < -1 and root->right is left heavy)
```

RL rotation



How can we identify the RL case?
if (balance_factor(root) < -1 and root->right is left heavy)

In the case of RL rotation, first, we need to rotate the right child once in the right direction and then rotate the root of the resulting tree in the left direction. As discussed earlier, rotating a node in the left direction can be visualized as attaching a thread to the node and pulling it in the left downward direction such that the right child becomes the root. Rotating a node in the right direction can be visualized as attaching a thread to the node and pulling it in the right downward direction such that the left child becomes the root.

AVL tree height

- What is the lower bound for the height of an AVL tree with n nodes?

AVL tree height (upper bound)

- Let's say f(h) represents the minimum number of nodes in an AVL tree of height h

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

$f(0) = 1$

$f(1) = 2$

$f(h) > 2f(h-2)$

$> 2^2 f(h-4)$

$> 2^3 f(h-6)$

$> 2^k f(h-2k)$

If h is even: $h - 2k = 0$

$f(h) > 2^{h/2} f(0)$

$\log_2(f(h)) > \frac{h}{2}$

$h < 2\log_2(f(h))$

$f(h) \leq n$

$h < 2\log_2(n)$

To compute the upper bound on the height of the tree, we first write a recurrence relation for the minimum number of nodes in an AVL. Let f(h) represents the minimum number of nodes in an AVL of height h. If the height of an AVL tree is h, the height of one of the subtrees must be h-1. The height of the other subtree would be either h-1 or h-2 because of the height balance property. To compute the minimum number of nodes, we need to take the height of the other tree as h-2. Thus, the final recurrence relation for the minimum number of nodes f(h) would be equal to f(h-1) + f(h-2) +1. Notice that f(h) <= n because it represents the minimum number of nodes in an AVL tree of height h. Solving this equation gives us the upper bound of h as O(log(n)).

AVL tree height (upper bound)

- Let's say f(h) represents the minimum number of nodes in an AVL tree of height h

$$f(0) = 1,\ f(1) = 2$$
$$f(h) = 1 + f(h-1) + f(h-2)$$
$$f(h) > 2f(h-2)$$
$$> 2^2 f(h-4)$$
$$> 2^3 f(h-6)$$
$$\dots$$
$$> 2^k f(h-2k)$$

If h is even, substitute $h = 2k$

$$f(h) > 2^{h/2} * f(0)$$
$$f(h) > 2^{h/2}$$
$$h < 2 * \log_2(f(h))$$

because, $f(h) <= n$

$$h < 2 * \log_2(n) = O(\log(n))$$

AVL tree height (upper bound)

- Let's say f(h) represents the minimum number of nodes in an AVL tree of height h

$f(0) = 1,\ f(1) = 2$

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

$f(h) > 2f(h-2)$

$> 2^2 f(h-4)$

$> 2^3 f(h-6)$

…

$> 2^k f(h-2k)$

If h is odd, substitute $h - 2k = 1$

$f(h) > 2^{h-1/2} * f(1)$

$f(h) > 2^{h-1/2} * 2$

$f(h) > 2^{h+1/2}$

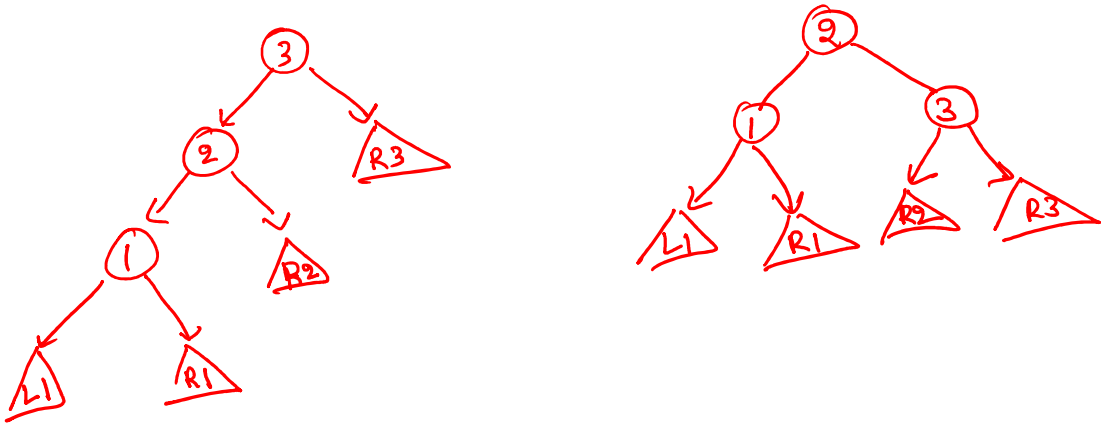
$h < (2 * \log_2(f(h))) - 1$

because, $f(h) <= n$

$h < (2 * \log_2(n)) - 1 = O(\log(n))$

Rotations

LL rotation




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

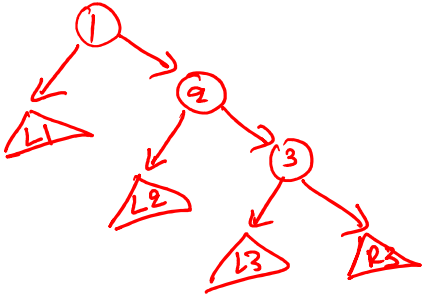
This is the general structure of an LL rotation. Notice that three nodes involved in the imbalance can have other balanced subtrees as children. After a rotation, as we discussed before, we need to find a place for these children in the rotated (balanced) tree. Notice that L1, R1, and R3 can stay in 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.

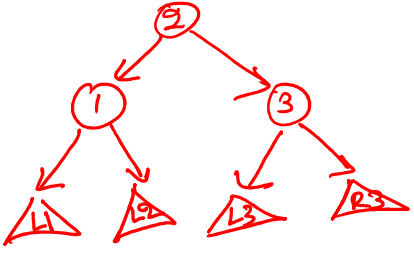
LL rotation

- 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
```

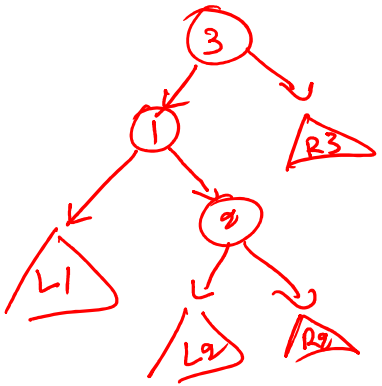
This is the general structure of an RR rotation. Notice that three nodes involved in the imbalance can have other balanced subtrees as children. After a rotation, as we discussed before, we need to find a place for these children in the rotated (balanced) tree. Notice that L1, L3, and R3 can stay in 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.

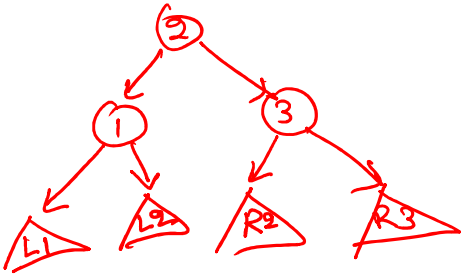
RR rotation

- 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
```

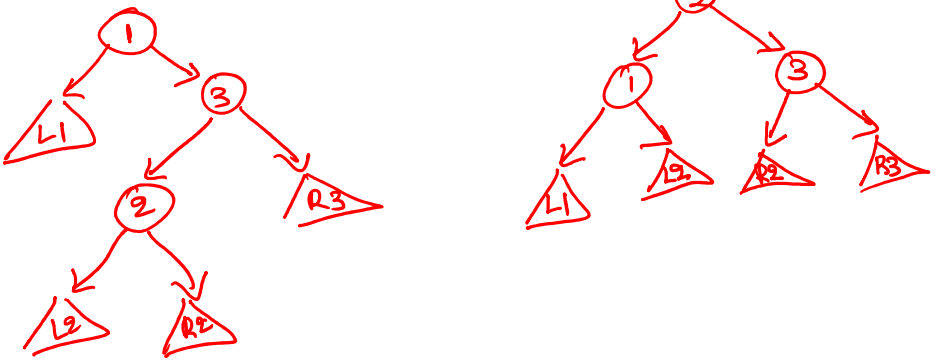
This is the general structure of an LR rotation. Notice that three nodes involved in the imbalance can have other balanced subtrees as children. After a rotation, as we discussed before, we need to find a place for these children in the rotated (balanced) tree. Notice that L1 and R3 can stay in the same 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.

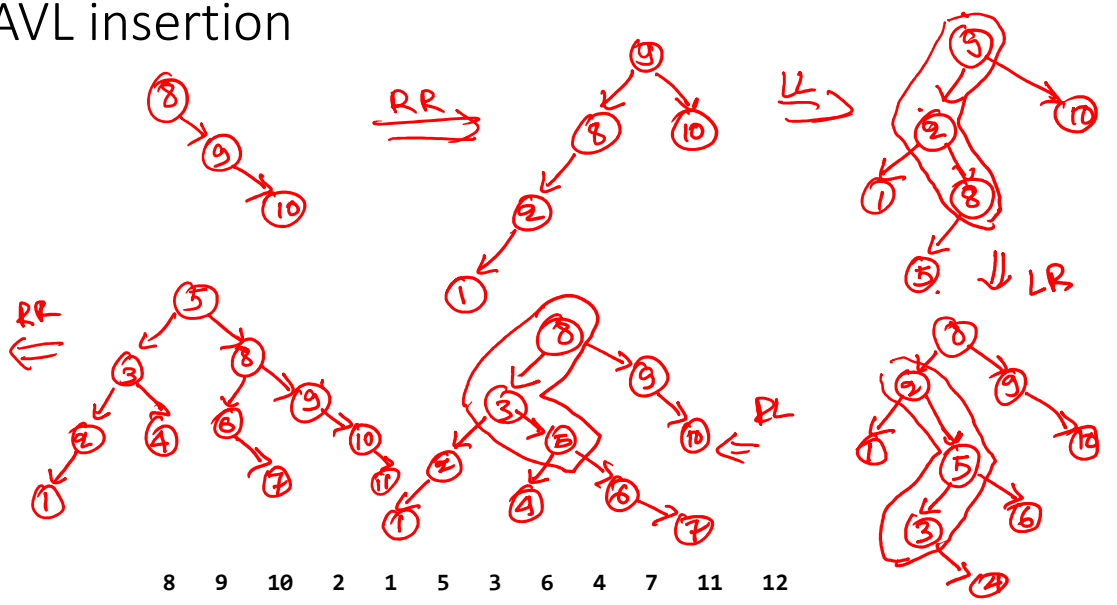
RL rotation

- 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

AVL 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

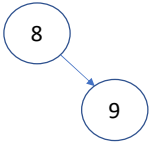
AVL insertion

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

AVL insertion

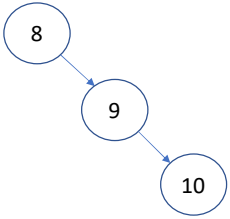
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



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

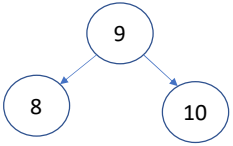
AVL insertion



need RR rotation

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

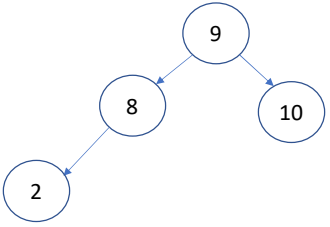
AVL insertion



after RR rotation

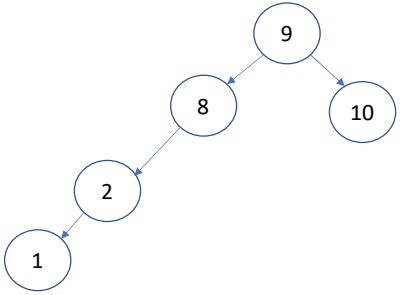
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



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

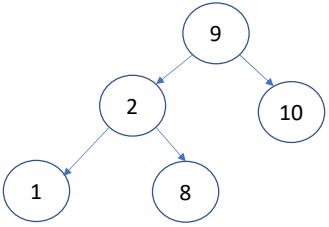
AVL insertion



need LL rotation

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

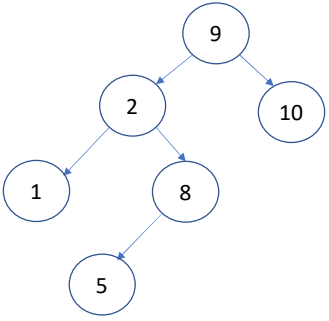
AVL insertion



after LL rotation

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

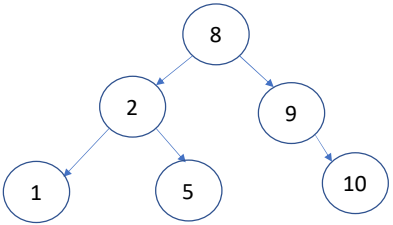
AVL insertion



need LR rotation

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

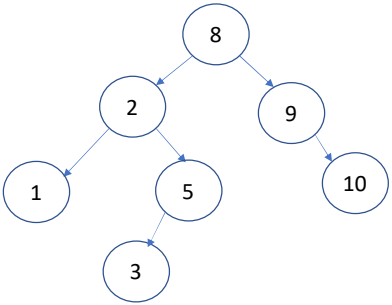
AVL insertion



after LR rotation

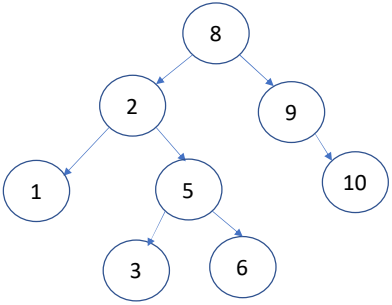
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



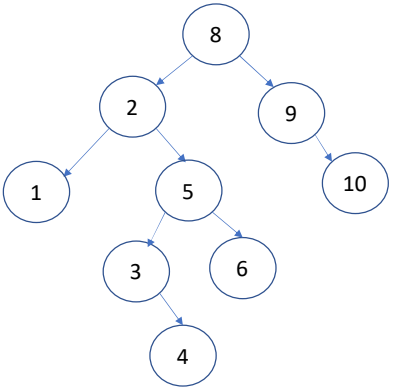
8 9 10 2 1 5 3 6 4 7 11 12

AVL insertion



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

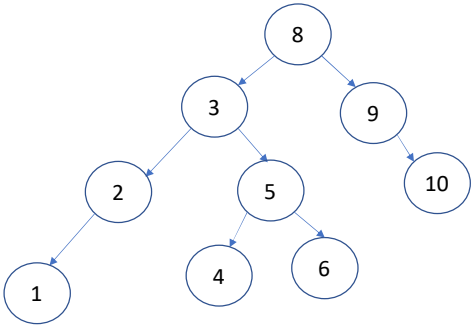
AVL insertion



need RL rotation

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

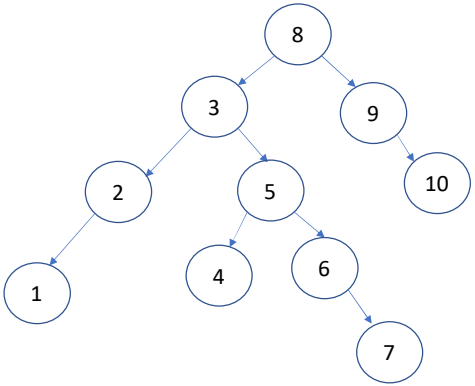
AVL insertion



after RL rotation

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

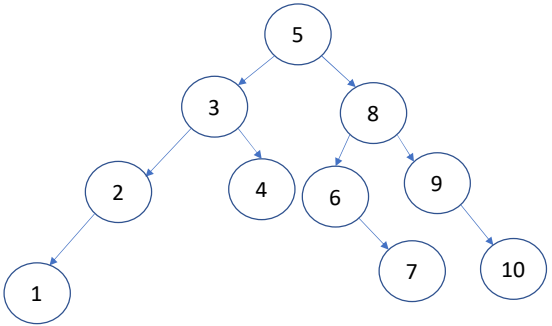
AVL insertion



need LR rotation

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

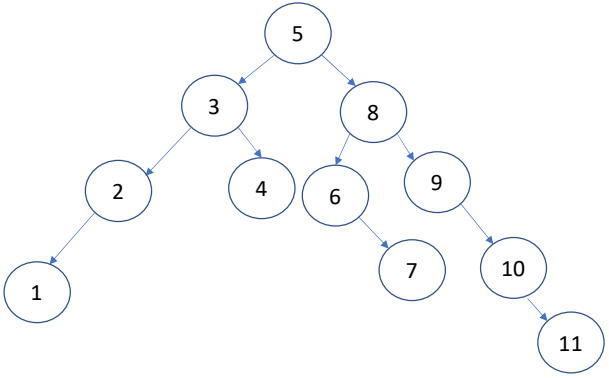
AVL insertion



after LR rotation

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

AVL insertion

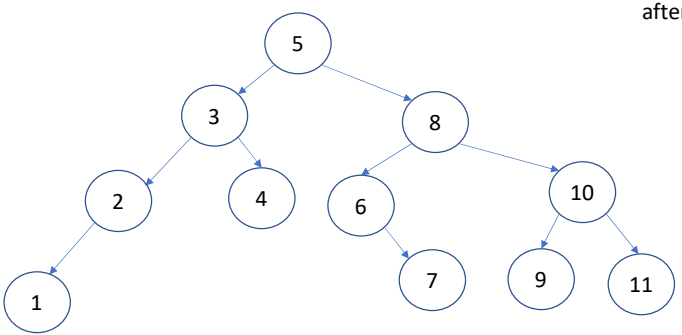


need RR rotation

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

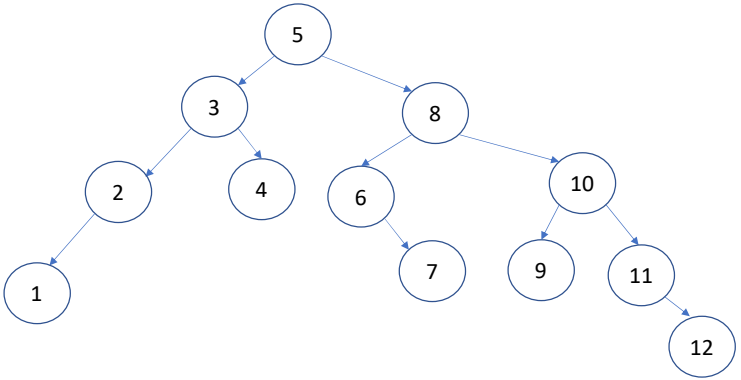
AVL insertion

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