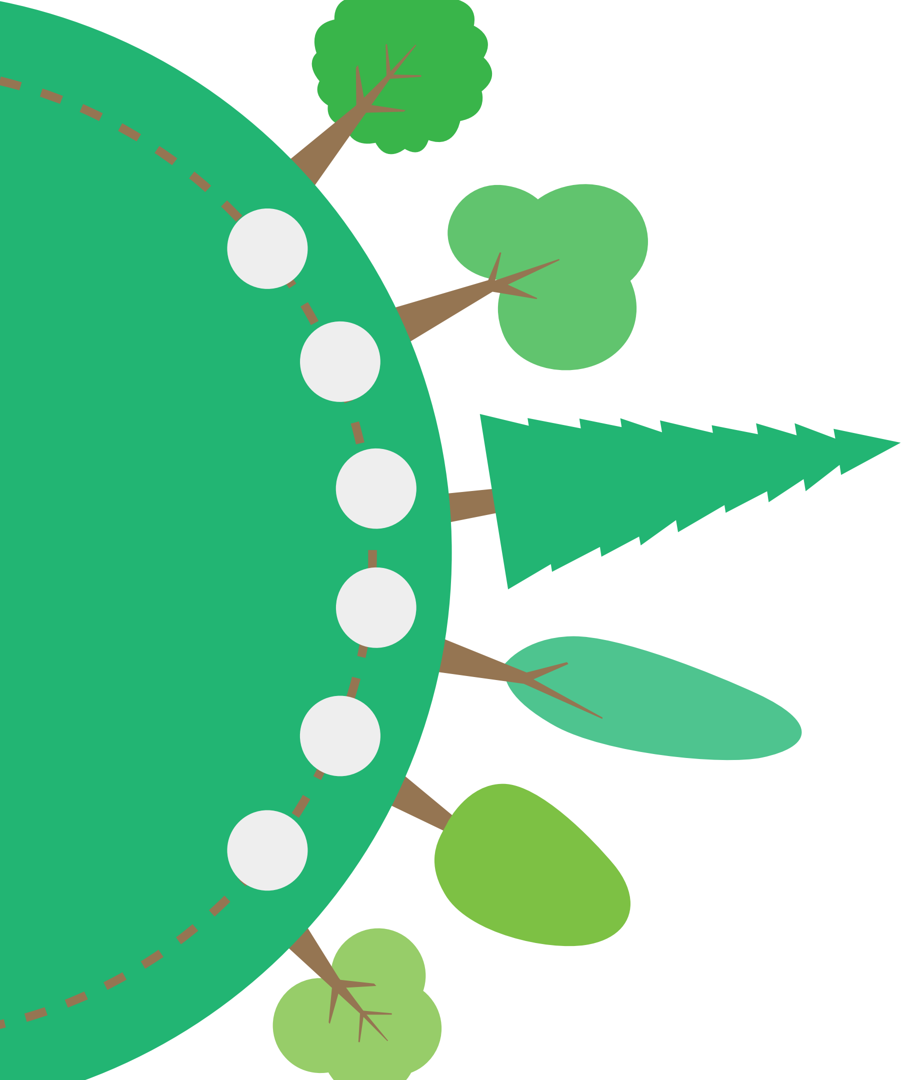


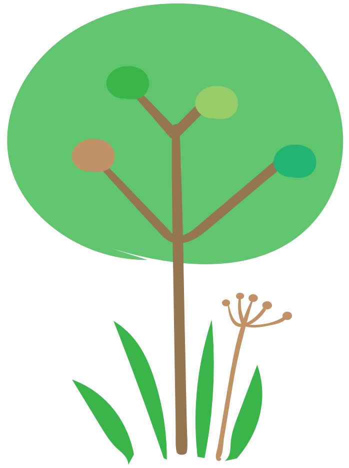
BST & AVL Trees

Team Summer

BST & AVL Tree: History

In **1960**, BST, or Binary Search Trees, were discovered by several researchers, including **P.F. Windley, Andrew Donald Booth, Andrew Colin and Thomas N. Hibbard.** Attributed to **Conway Berners-Lee and David Wheeler**

Then, in **1962**, AVL Trees, named after inventors **A**delson-**V**elsky and **L**andis, were invented by **Georgy Adelson-Velsky and Evgenii Landis** for the efficient organization of information.



Binary Search Tree

- It is a node-based binary tree that allows us to maintain a sorted list of numbers.

Properties

- All nodes of left subtree are less than the root node
- All nodes of right subtree are more than the root node
- Both subtrees of each node are also BSTs i.e. they have the above two properties

AVL Tree

- It is a self-balancing binary search tree
- Each node maintains a balance factor whose value is either -1, 0 or +1.

Balance Factor

- Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.
- If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

Balance Factor = (Height of Left Subtree - Height of Right Subtree)

BST: Original Version

A Insertion in a BST

1. **if node == null**, create a new node with the value of the key field equal to K. We return this newly created node directly from here.

2. **if K <= node.key**, it means K must be inserted in the left subtree of the current node. We repeat(recur) the process from step 1 for the left subtree.

3. **else K > node.key**, which means K must be inserted in the right subtree of the current node. We repeat(recur) the process from step 1 for the right subtree.

4. **Return the current node.**

BST: Original Version

B Searching in a BST

1. **if node == null**, K is not present in the tree. We return null to signify that K was not found.

2. **if K < node.key**, it means K must be present in the left subtree of the current node. We repeat(recur) the process from step 1 for the left subtree.

3. **else if K > node.key**, it means K must be present in the right subtree of the current node. We repeat(recur) the process from step 1 for the right subtree.

4. **else K == node.key**, we have found K in the tree. We can return the current node to signify K was found in this particular node.

BST: Original Version

C Deletion in a BST

1. If the **node becomes NULL**, it will return NULL.

2. **if K > node.key**, it means K must be present in the right subtree of the current node. So, call the above remove function with root->right

3. **else if K < node.key**, it means K must be present in the left subtree of the current node. So, call the above function with root->left

c Deletion in a BST

BST: Original Version

4. This part will be executed only if the **if K == node.key**. The actual removal starts from here

   a. **Case 1: Leaf node.**

      i. Check if both left and right children are NULL, then replace the leaf node with NULL and simple free the allocated space.

   b. **Case 2: Node has right child.**

      i. Replace the root node with root->right and free the right node

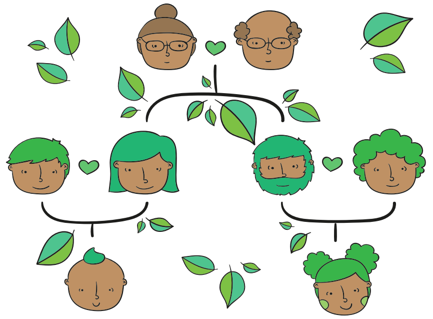
   c. **Case 3: Node has left child.**

      i. Replace the node with root->left and free the left node

   d. **Case 4: Node has both left and right children.**

      i. Find the min value in the right subtree. Replace node value with min and again call the remove function to delete the node which has the min value. Since we find the min value from the right subtree call the remove function with root->right.

AVL Tree: Original Version



A Balance Factor

1. The **balance factor** is known as the difference between the height of the left subtree and the right subtree.

2. **Balance factor(node) = height(node->left) – height(node->right)**

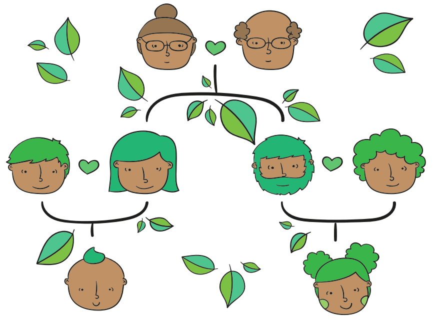
3. Allowed values of BF are **–1, 0, and +1.**

4. The **value –1** indicates that the right subtree contains one extra, i.e., the tree is right heavy.

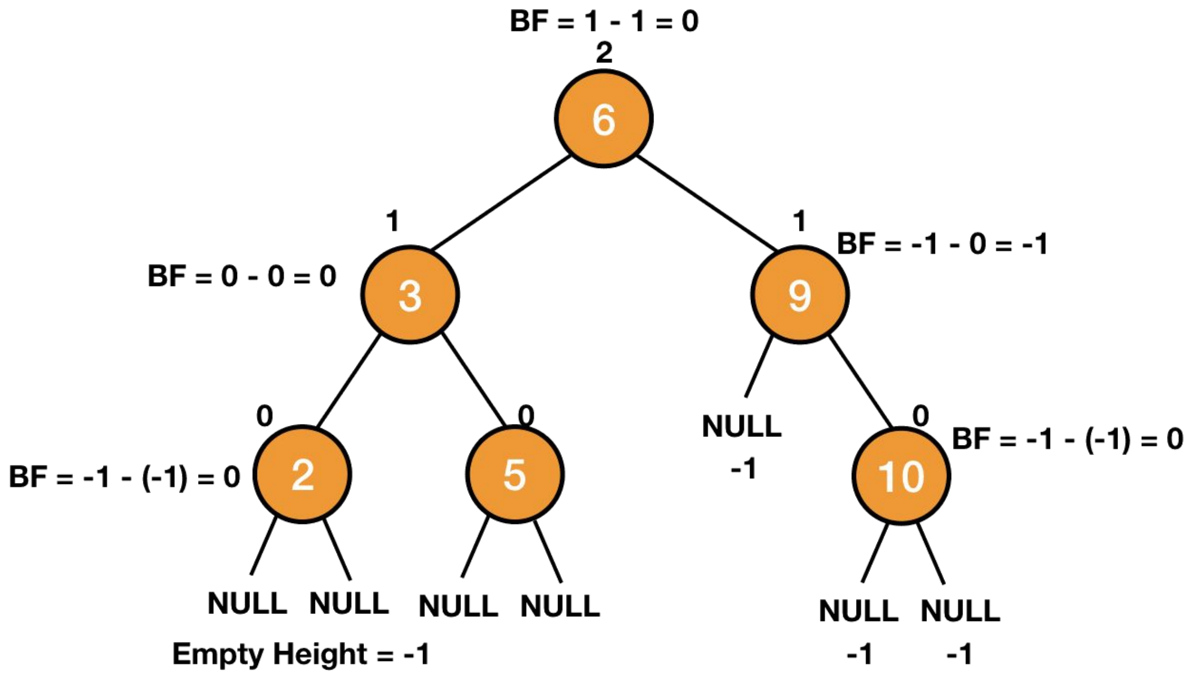
5. The **value +1** indicates that the left subtree contains one extra, i.e., the tree is left heavy.

6. The **value 0** shows that the tree includes equal nodes on each side, i.e., the tree is perfectly balanced.

AVL Tree: Original Version



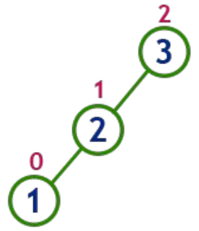
A Balance Factor



AVL Rotations

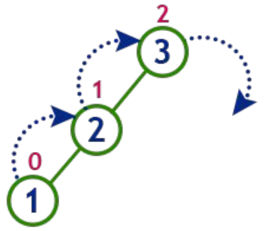
LL Rotation

- If a node is inserted in the left subtree of the left subtree, the tree needs a left left rotation

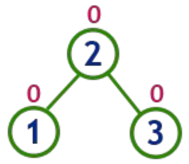


Tree is imbalanced

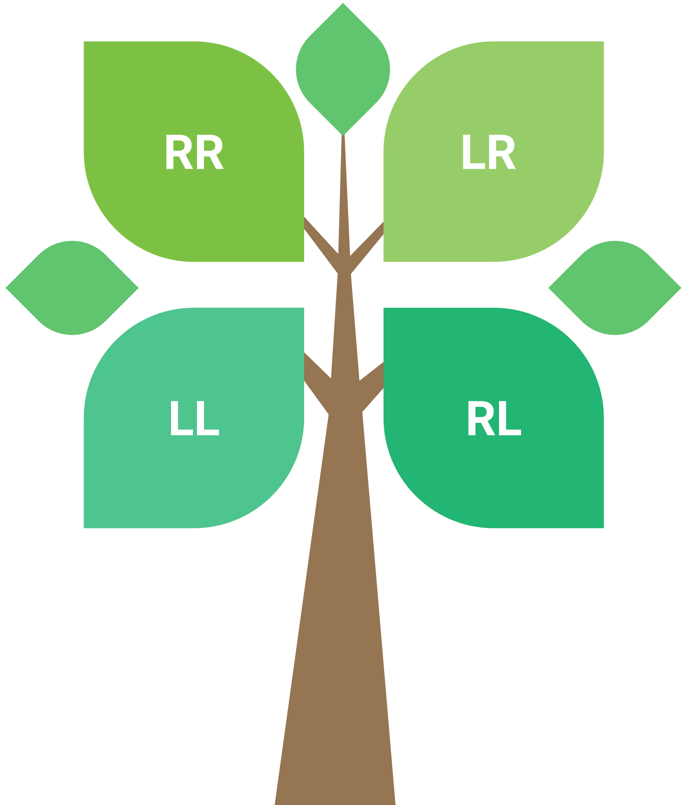
because node 3 has balance factor 2



To make balanced we use LL Rotation which moves nodes one position to right



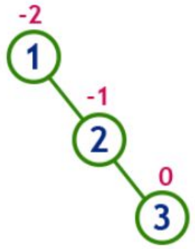
After LL Rotation Tree is Balanced



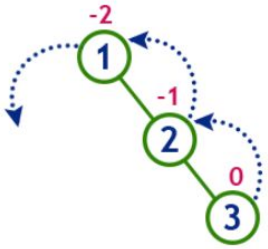
AVL Rotations

RR Rotation

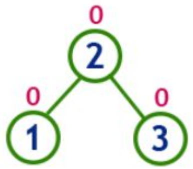
- If a node is inserted in the right subtree of the right subtree, the tree needs a right right rotation



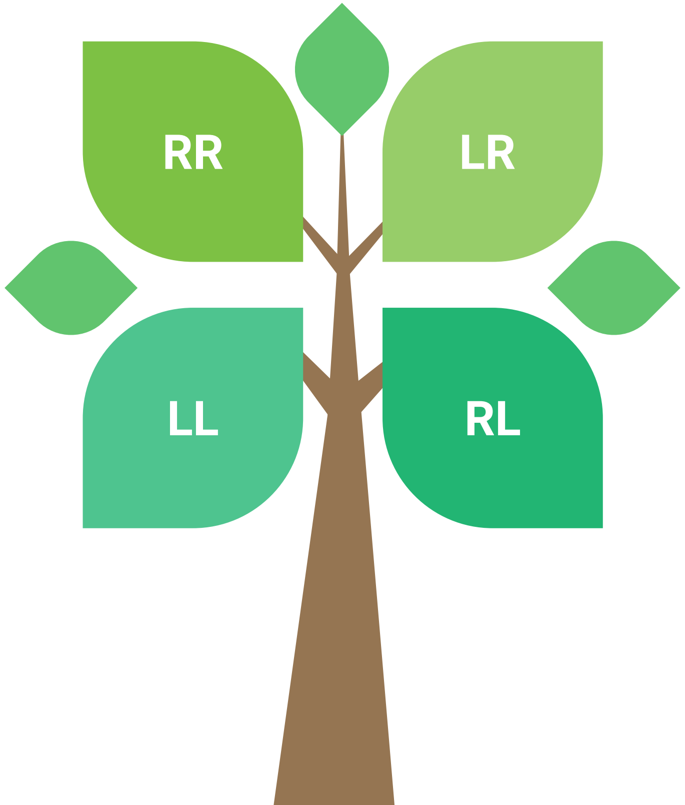
Tree is imbalanced



To make balanced we use RRRotation which moves nodes one position to left



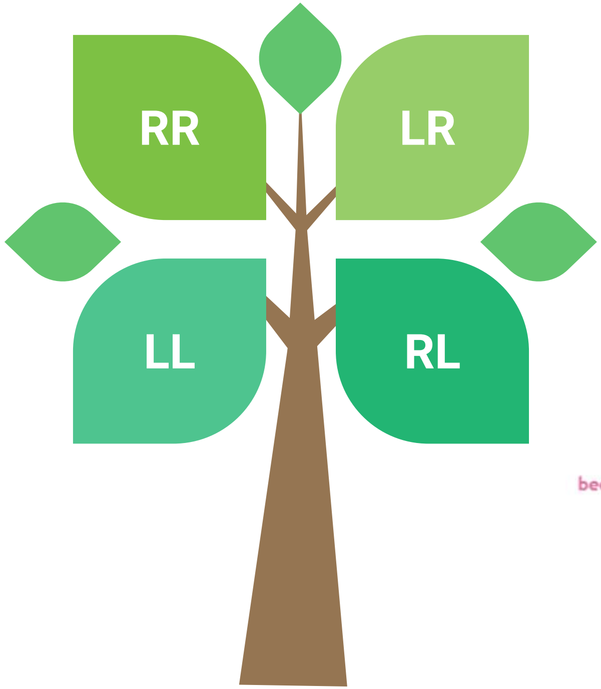
After RR Rotation Tree is Balanced

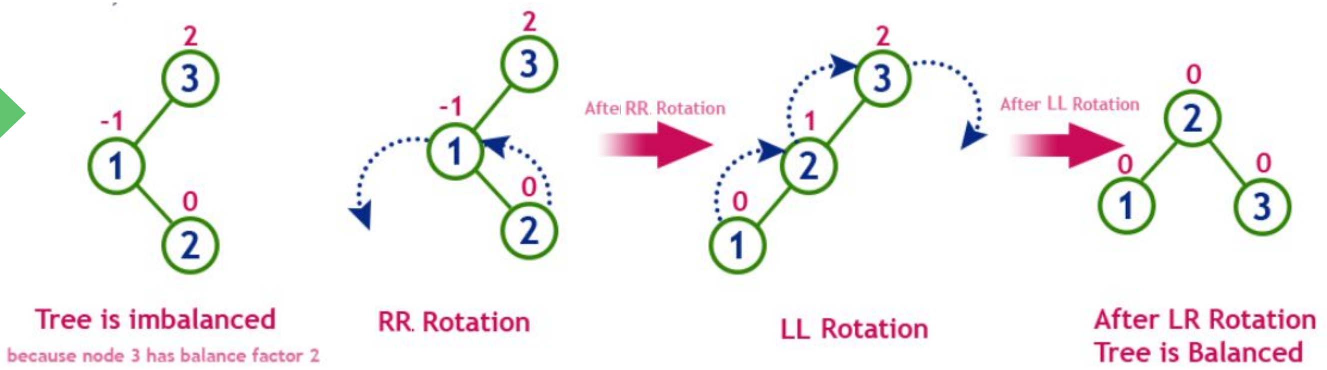


AVL Rotations

LR Rotation

- When a node has been inserted into the right subtree of the left subtree, this causes the AVL tree to perform left-right rotation.

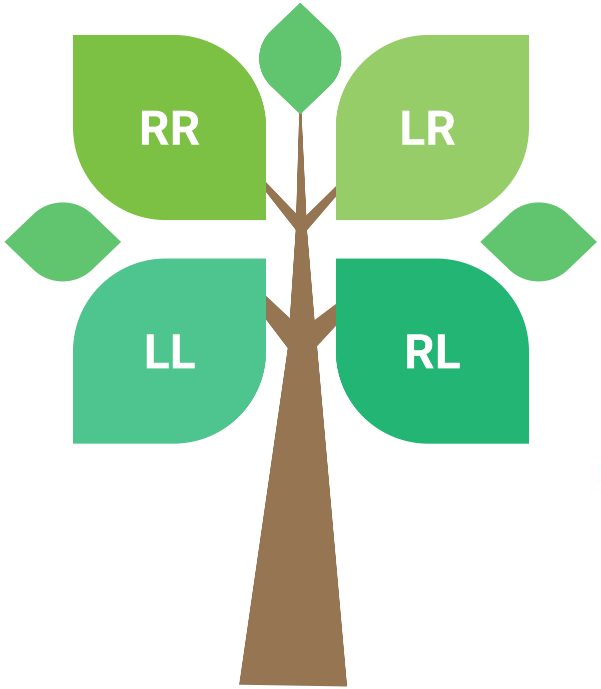


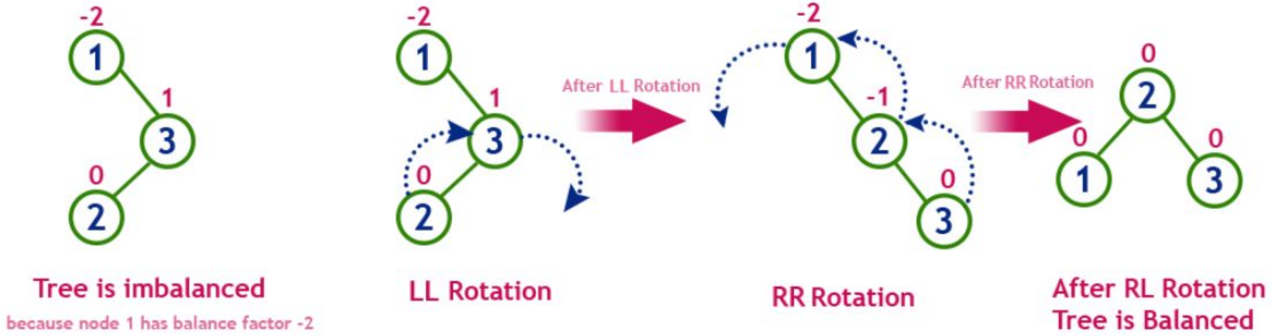


AVL Rotations

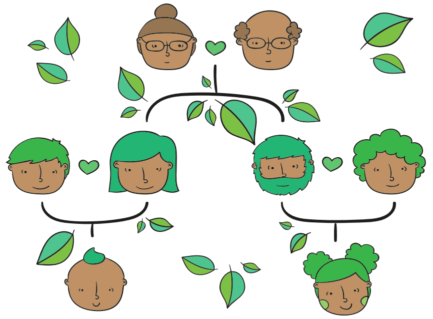
RL Rotation

- When a node has been inserted into the left subtree of the right subtree, this causes the AVL tree to perform right-left rotation.





AVL Tree: Original Version



B Insertion in AVL

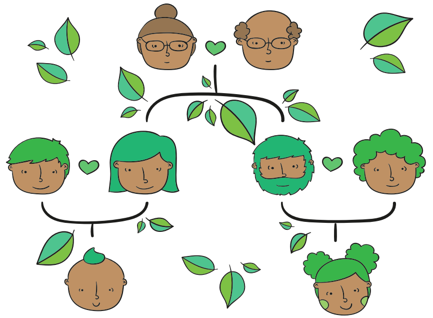
1. **if root == null**, create a new node with the value of the key field equal to **K**. We return this newly created node directly from here. Important: A **new node** is always inserted as a leaf node with balance factor equal to 0.

2. Go to the appropriate leaf node to insert **K** using the following recursive steps. Compare **K** with **root.key** of the current tree.

   a. If **K** < **root.key**, call insertion algorithm on the left subtree of the current node until the leaf node is reached.

   b. Else if **K** > **root.key**, call insertion algorithm on the right subtree of current node until the leaf node is reached.

AVL Tree: Original Version



B Insertion in AVL

3. Update **balanceFactor** of the nodes.

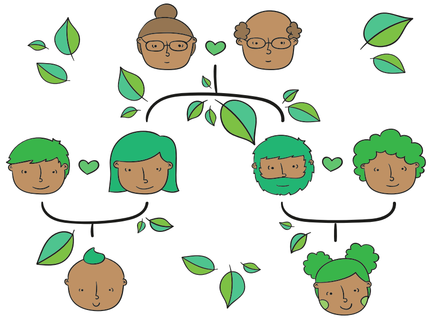
4. If the nodes are unbalanced, then rebalance the node.

    a. If **balanceFactor** > 1, it means the height of the left subtree is greater than that of the right subtree. So, do a right rotation or left-right rotation

        i. If **K** < **leftChildKey** do right rotation.

        ii. Else, do left-right rotation

AVL Tree: Original Version



B Insertion in AVL

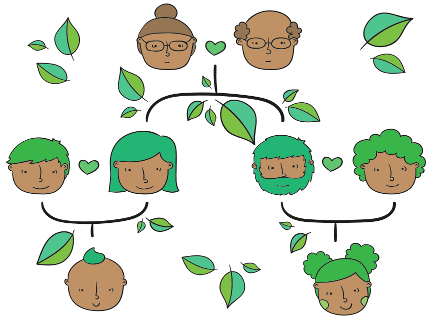
4. If the nodes are unbalanced, then rebalance the node.

    a. If **balanceFactor** < -1, it means the height of the right subtree is greater than that of the left subtree. So, do right rotation or right-left rotation

        i. If **K** > **rightChildKey** do left rotation.

        ii. Else, do right-left rotation

AVL Tree: Original Version



c Deletion in AVL

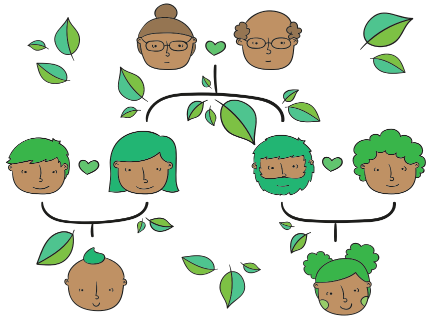
1. Locate **nodeToBeDeleted** (recursion is used to find **nodeToBeDeleted**)

   a. **if node == null**, we return node.

   b. **if K > node.key**, it means K must be present in the right subtree of the current node. So, call the above remove function with root->right

   c. **else if K < node.key**, it means K must be present in the left subtree of the current node. So, call the above function with root->left

AVL Tree: Original Version



c Deletion in AVL

2. This part will be executed only if the **if K == node.key**. The actual removal starts from here

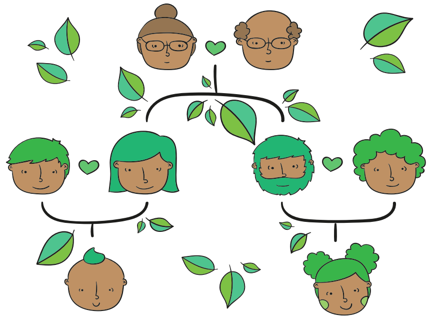
   a. **Case 1: When the node to be deleted is a leaf node**

      i. In this case, the node to be deleted contains no subtrees i.e., it’s a leaf node. Hence, it can be directly removed from the tree.

   b. **Case 2: When the node to be deleted has one subtree**

      i. In this case, the node to be deleted is replaced by its only child thereby removing the specified node from the BST.

AVL Tree: Original Version

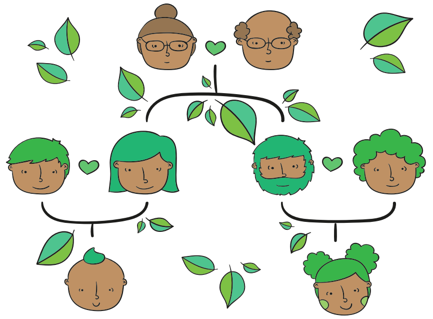


c Deletion in AVL

c. **Case 3: When the node to be deleted has both the subtrees.**
   i. In this case, the node to be deleted can be replaced by one of the two available nodes:
      1. It can be replaced by the node having the largest value in the left subtree (Longest left node or Predecessor).
      2. Or, it can be replaced by the node having the smallest value in the right subtree (Smallest right node or Successor).

3. Update **balanceFactor** of the nodes.

AVL Tree: Original Version

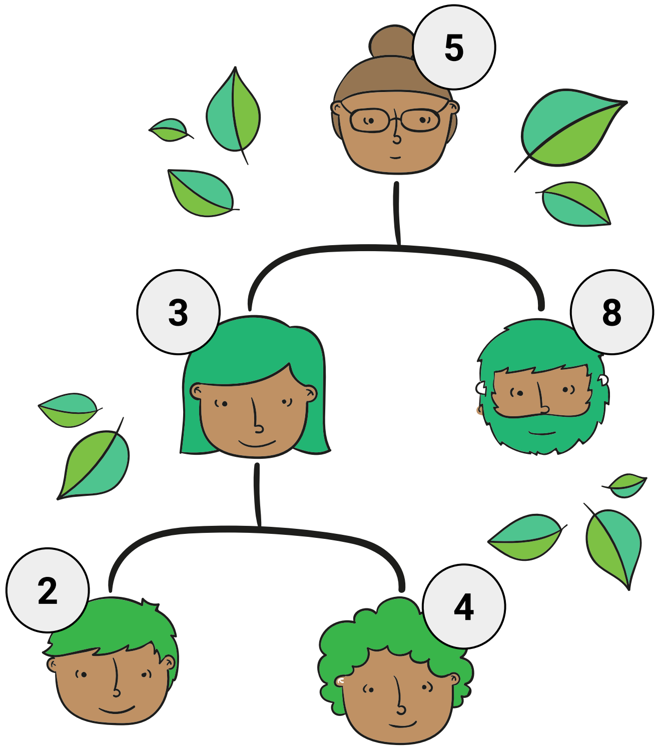


c Deletion in AVL

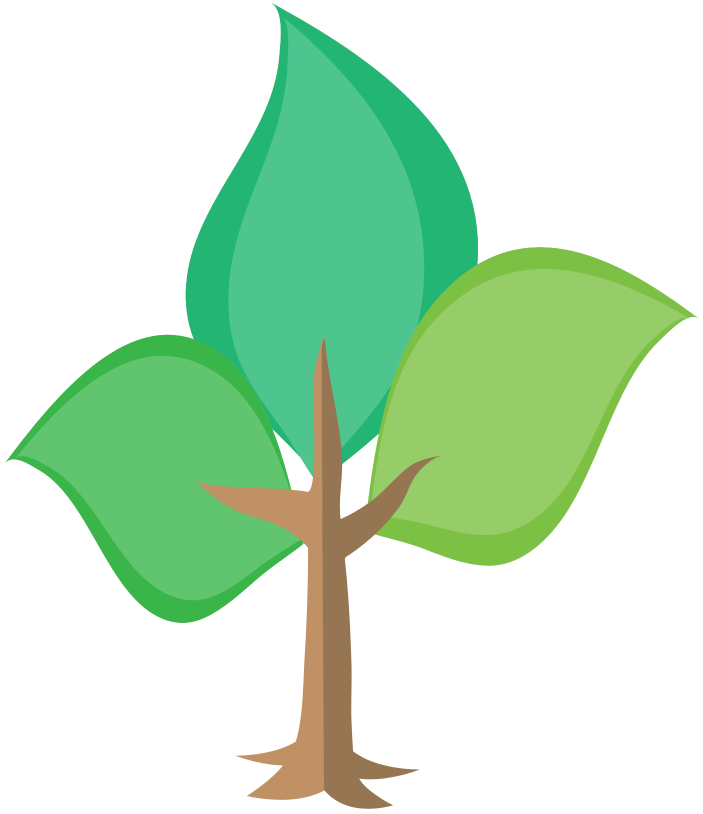
4. Rebalance the tree if the balance factor of any of the nodes is not equal to -1, 0 or 1.
    a. If **balanceFactor** of **currentNode** > 1,
        i. If **balanceFactor** of **leftChild** >= 0, do right rotation
        ii. Else do left-right rotation.
    b. If **balanceFactor** of **currentNode** < -1,
        i. If **balanceFactor** of **rightChild** <= 0, do left rotation.
        ii. Else do right-left rotation.

BST Variants on the Internet

1. AA Tree
2. AVL Tree
3. B- Tree
4. Red Black Tree
5. Scapegoat Tree
6. Splay Tree
7. Tango Tree
8. Treap
9. Weight Balance Tree



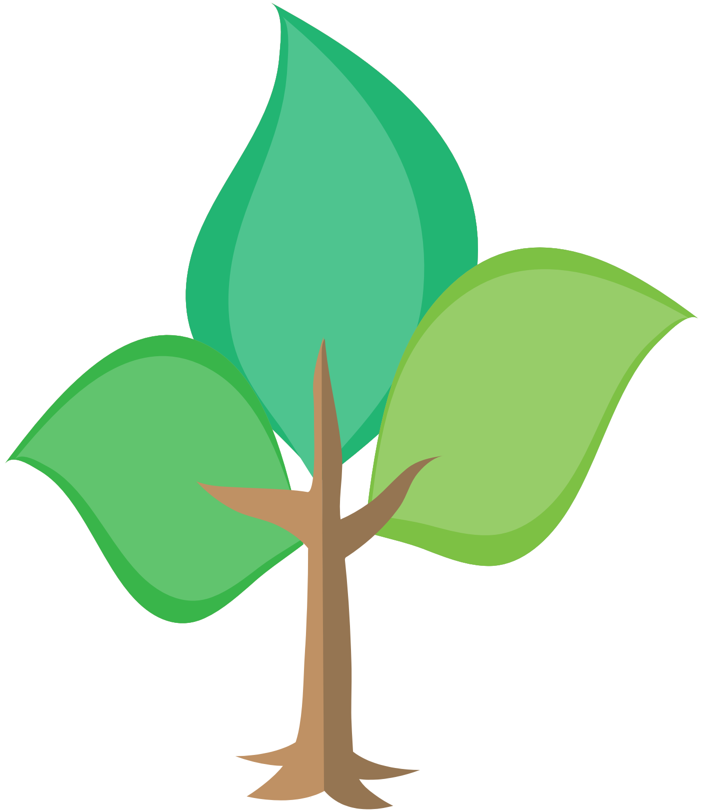
Complexity: Time and Space



Binary Search Trees

OPERATION	Best case	Average case	Worst case	Space
SEARCH	O(1)	O(log n)	O(n)	O(n)
INSERT	O(1)	O(log n)	O(n)	O(n)
DELETE	O(log n)	O(log n)	O(n)	O(n)

Complexity: Time and Space

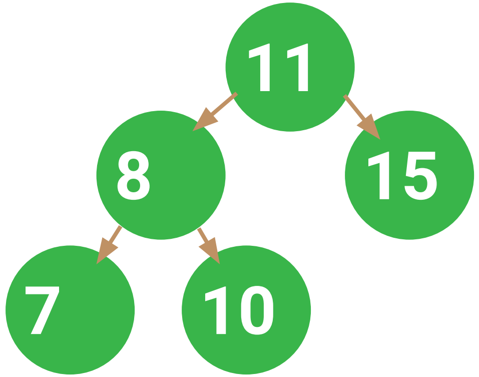


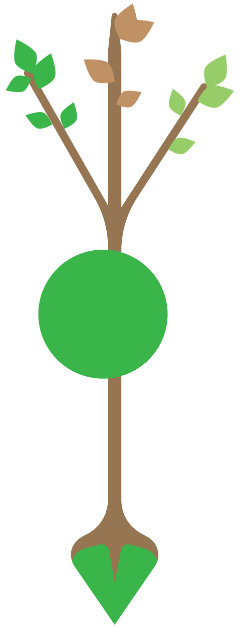
AVL Trees

OPERATION	Best case	Average case	Worst case	Space
SEARCH	O(1)	O(log n)	O(log n)	O(n)
INSERT	O(log n)	O(log n)	O(log n)	O(n)
DELETE	O(log n)	O(log n)	O(log n)	O(n)
TRAVERSAL	O(log n)	O(log n)	O(log n)	O(n)

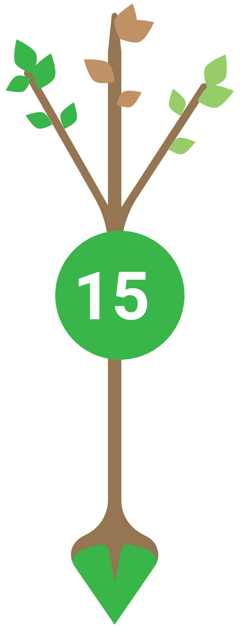
SEARCH SIMULATION

BS TREE

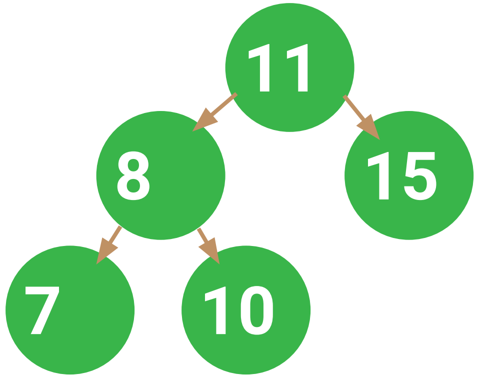




SEARCH SIMULATION

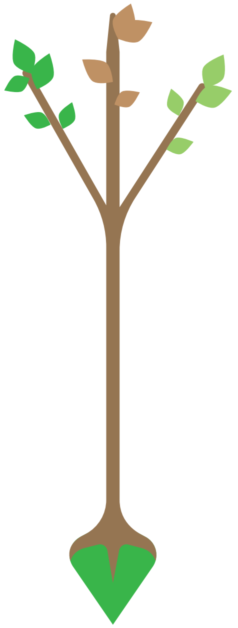


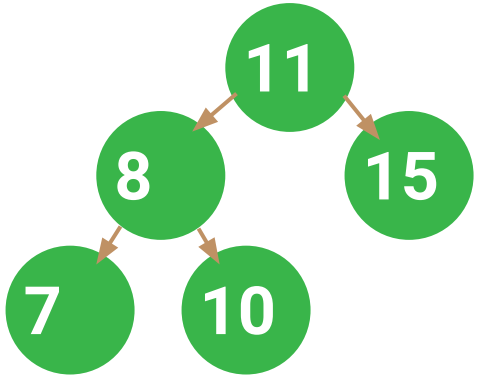
BS TREE

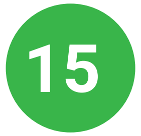


SEARCH SIMULATION

BS TREE

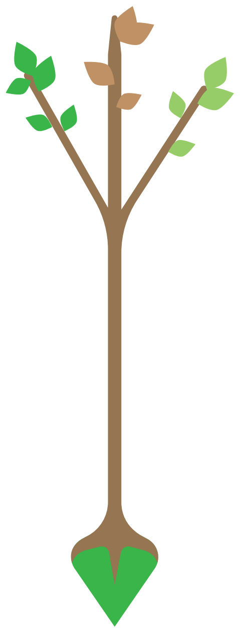




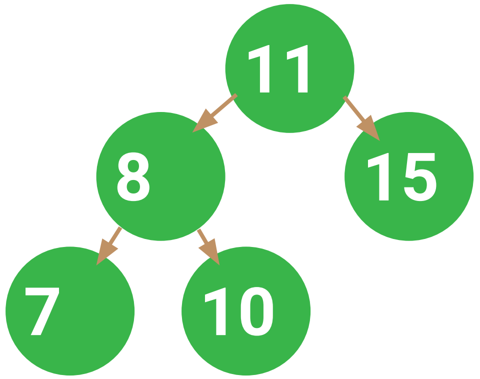


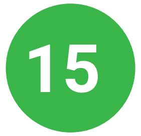
Step 1 - Read the search element from the user.

SEARCH SIMULATION



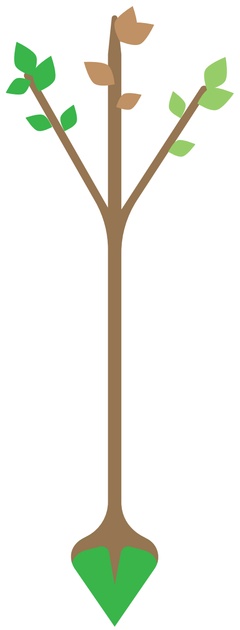
BS TREE



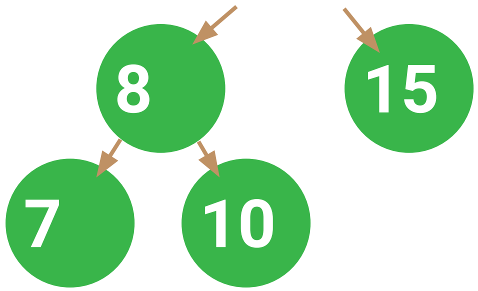


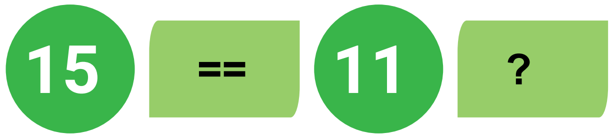
Step 2 - Compare the search element with the value of root node in the tree.

SEARCH SIMULATION



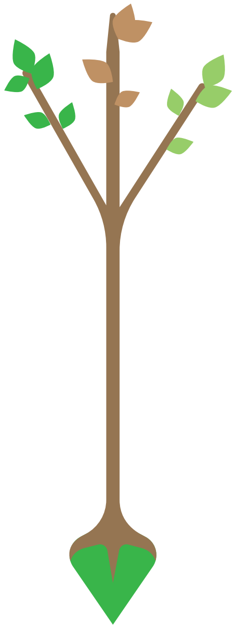
BS TREE



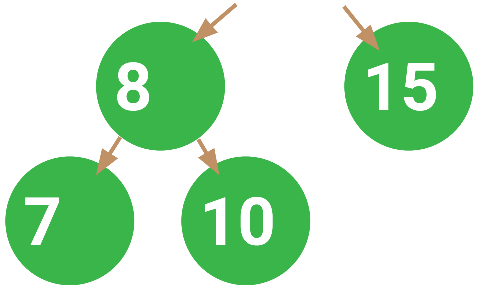


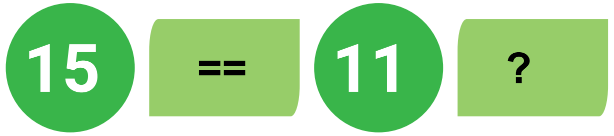
Step 2 - Compare the search element with the value of root node in the tree.

SEARCH SIMULATION



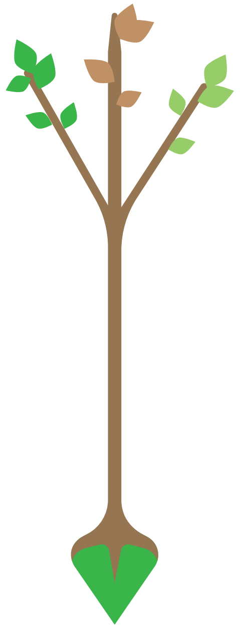
BS TREE

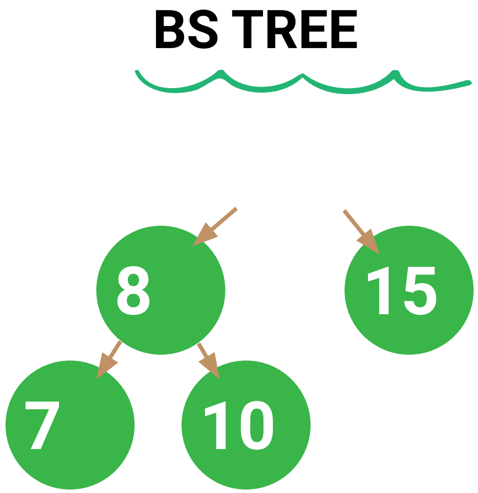


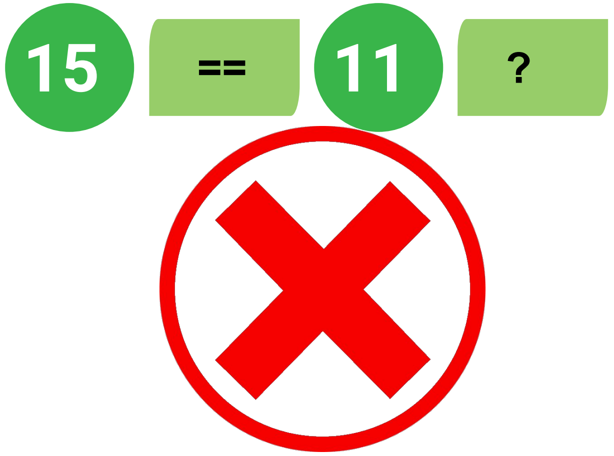


Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

SEARCH SIMULATION



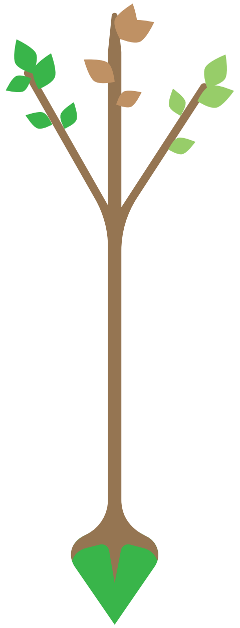


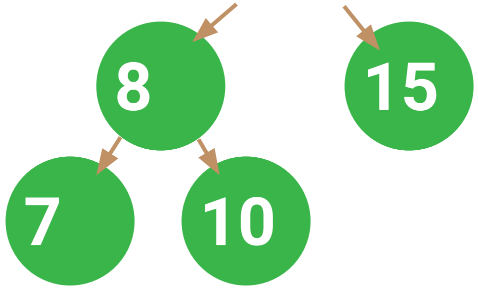


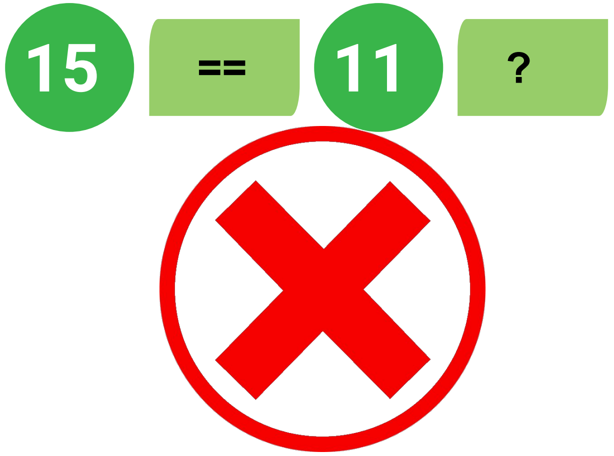
Step 3 - If both are matched, then display "Given node is found!!!" and terminate the function

SEARCH SIMULATION

BS TREE



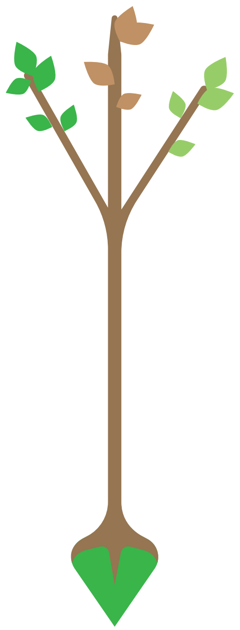


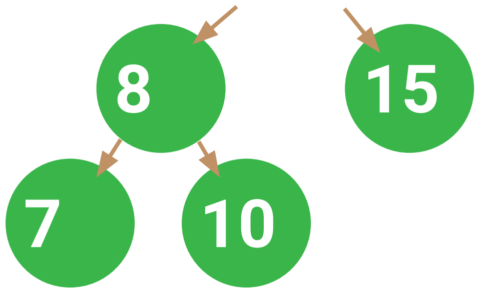


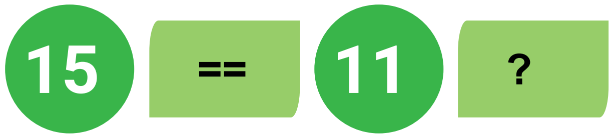
Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

SEARCH SIMULATION

BS TREE



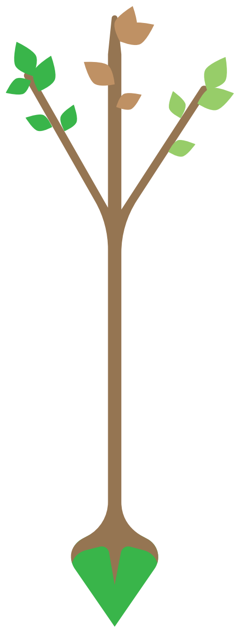


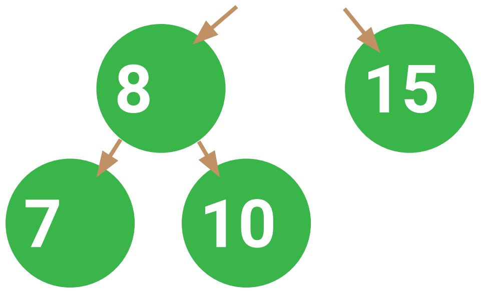


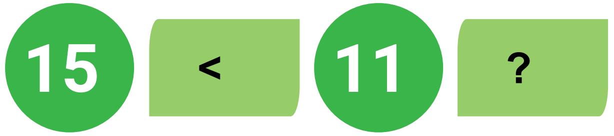
Step 4 - If both are not matched, then check whether search element is smaller or larger than that node value.

SEARCH SIMULATION

BS TREE



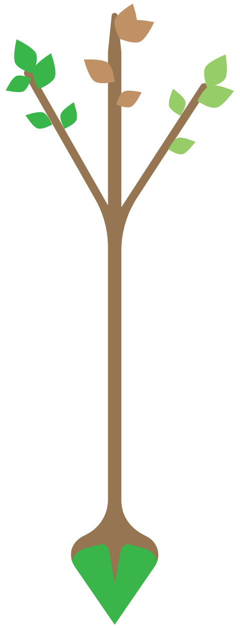


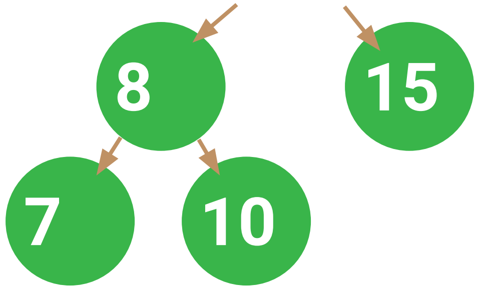


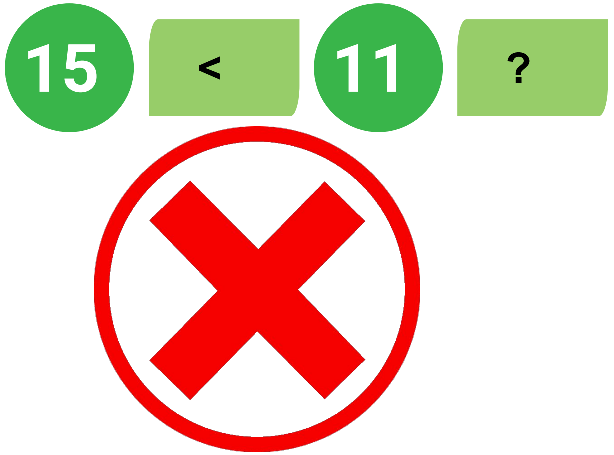
Step 5 - If search element is smaller, then continue the search process in left subtree.

SEARCH SIMULATION

BS TREE



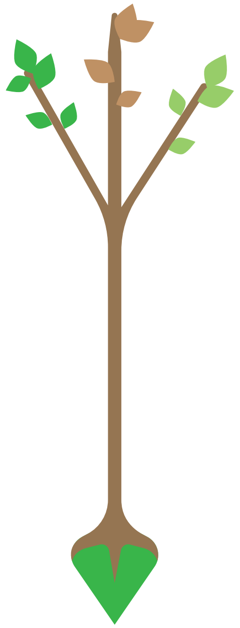


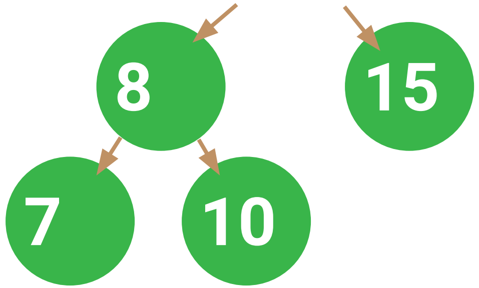


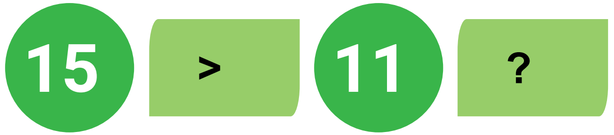
Step 5 - If search element is smaller, then continue the search process in left subtree.

SEARCH SIMULATION

BS TREE



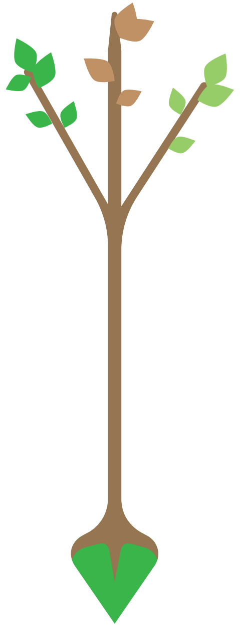


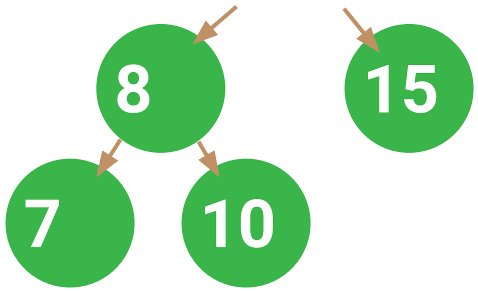


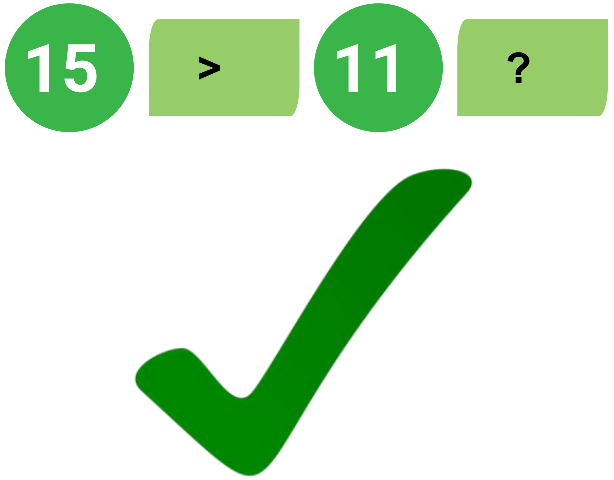
Step 6- If search element is larger, then continue the search process in right subtree.

SEARCH SIMULATION

BS TREE

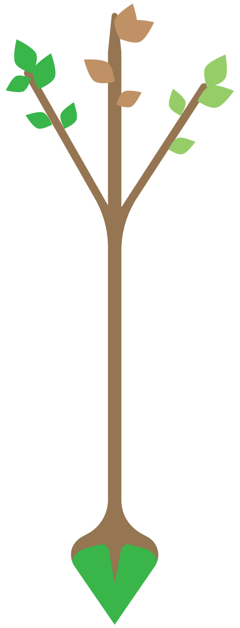




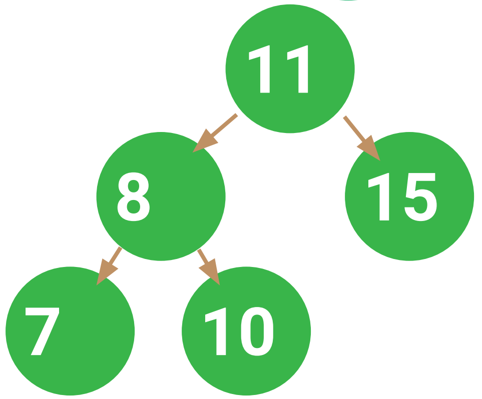


Step 6- If search element is larger, then continue the search process in right subtree.

SEARCH SIMULATION

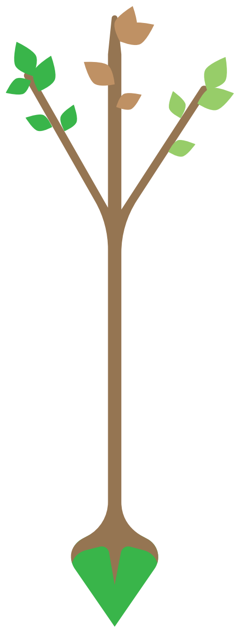


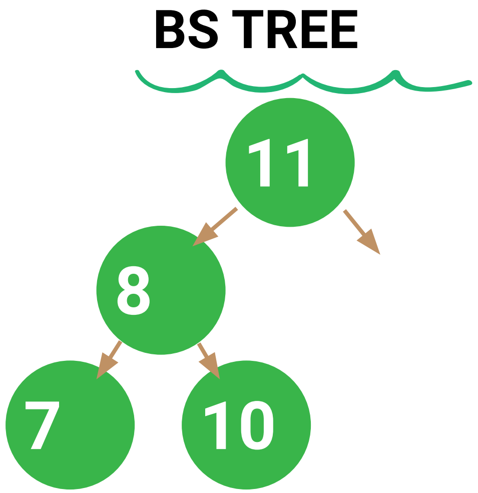
BS TREE

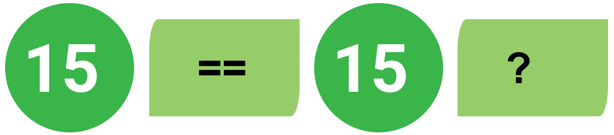


Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

SEARCH SIMULATION

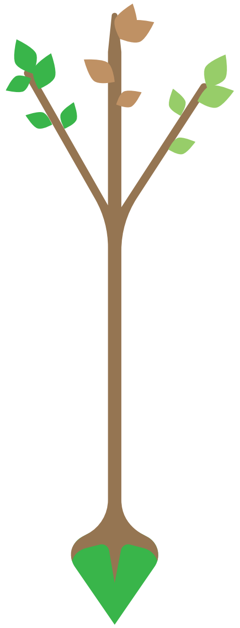




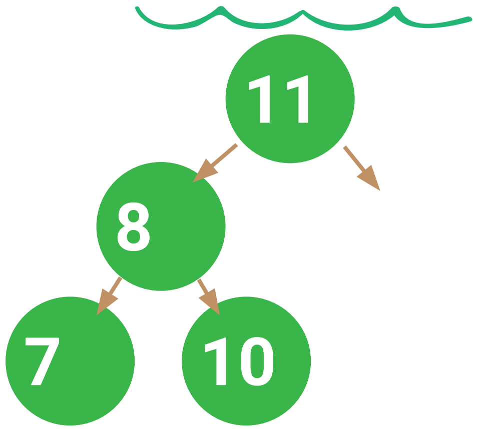


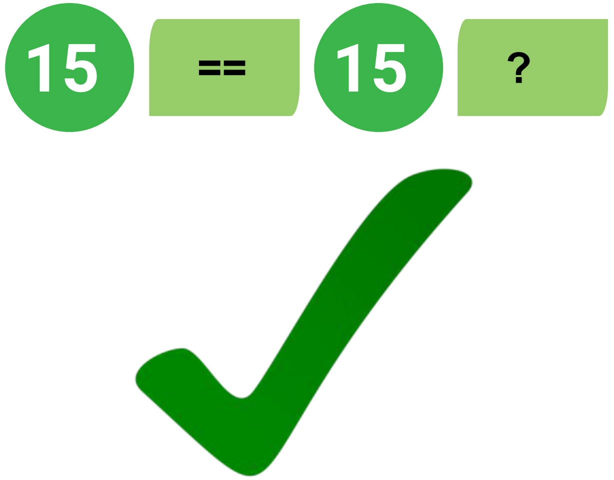
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

SEARCH SIMULATION



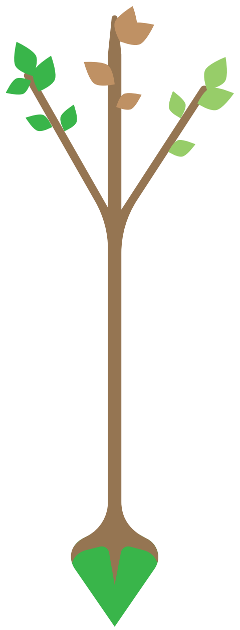
BS TREE



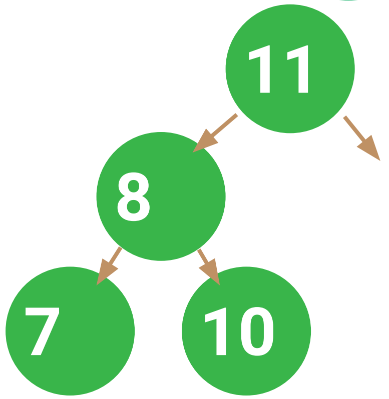


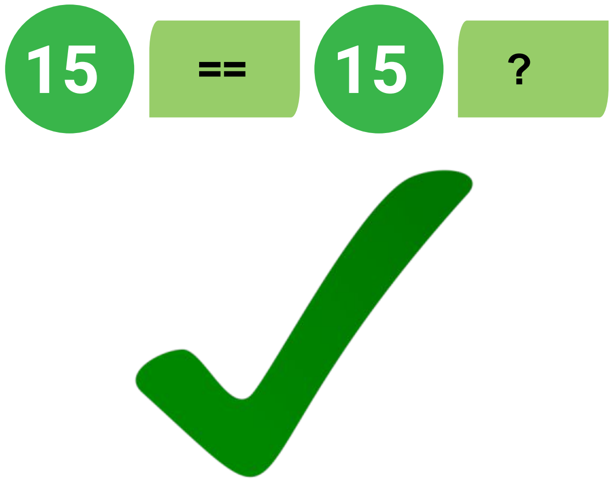
Step 7 - Repeat the same until we find the exact element or until the search element is compared with the leaf node

SEARCH SIMULATION



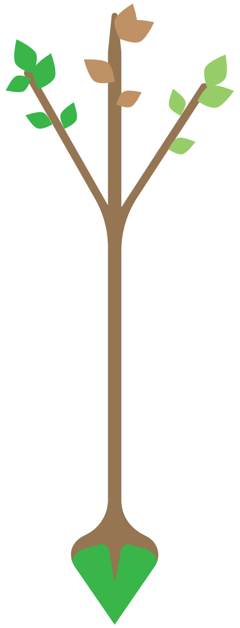
BS TREE



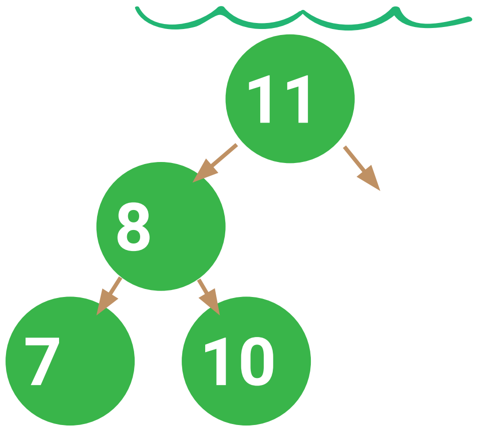


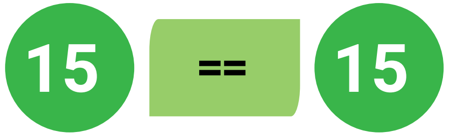
Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

SEARCH SIMULATION



BS TREE



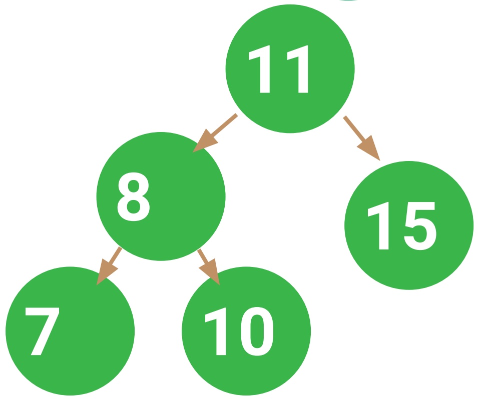


Element is found!

Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

SEARCH SIMULATION

BS TREE

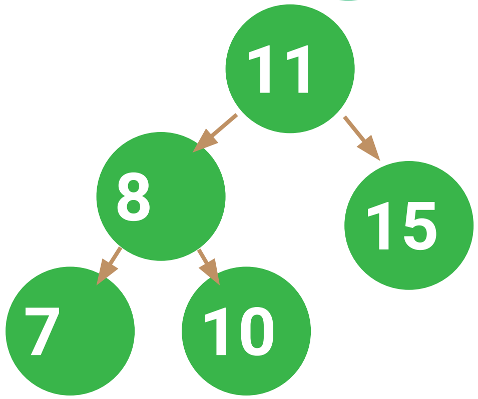


Step 8 - If we reach to the node having the value equal to the search value then display "Element is found" and terminate the function.

SEARCH SIMULATION

Function terminate

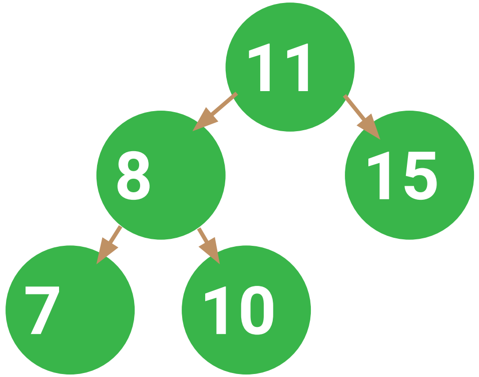
BS TREE

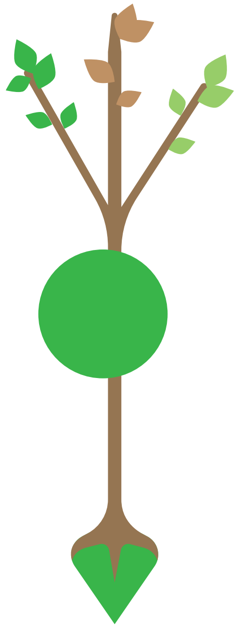


Step 9 - If we reach to the leaf node and if it is also not matched with the search element, then display "Element is not found" and terminate the function.

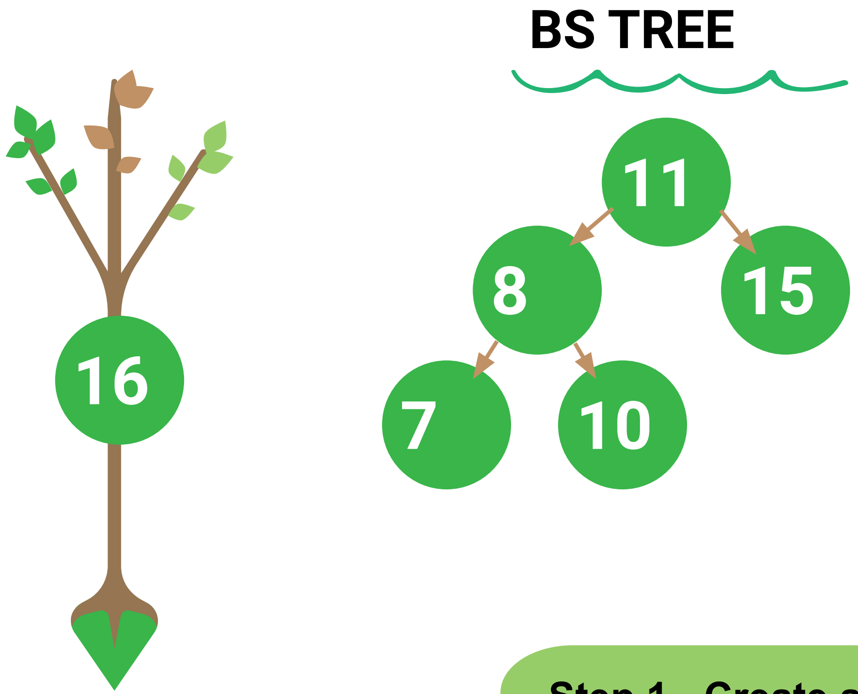
INSERT SIMULATION

BS TREE





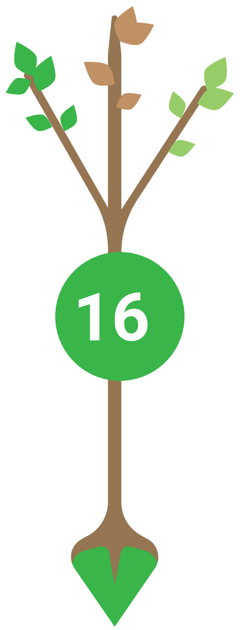
INSERT SIMULATION

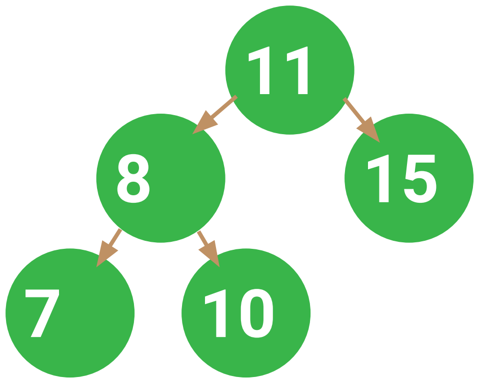


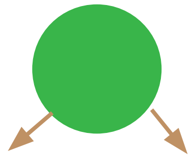
Step 1 - Create a newNode with given value and set its left and right to NULL.

INSERT SIMULATION

BS TREE



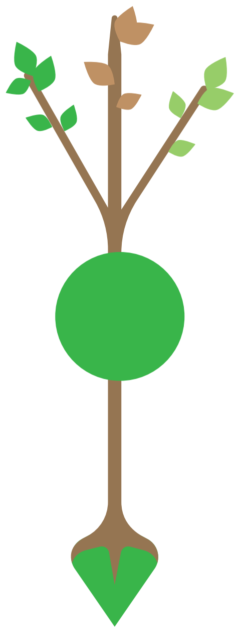


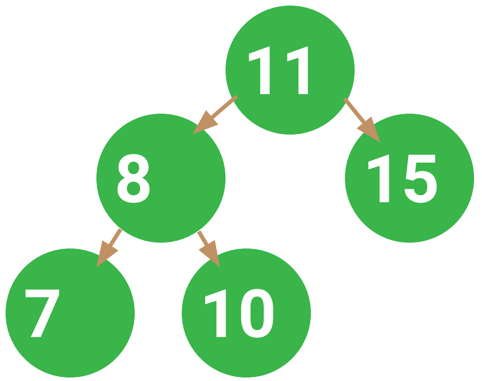


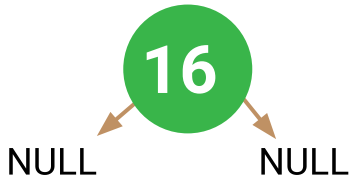
Step 1 - Create a newNode with given value and set its left and right to NULL.

INSERT SIMULATION

BS TREE



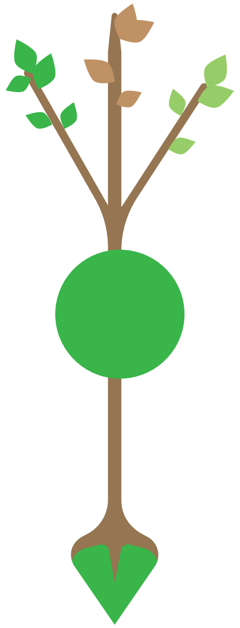


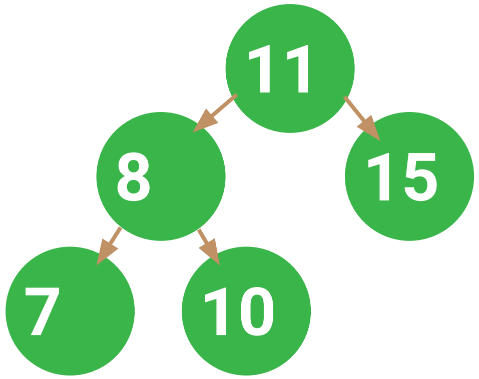


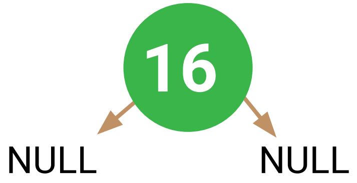
Step 1 - Create a newNode with given value and set its left and right to NULL.

INSERT SIMULATION

BS TREE

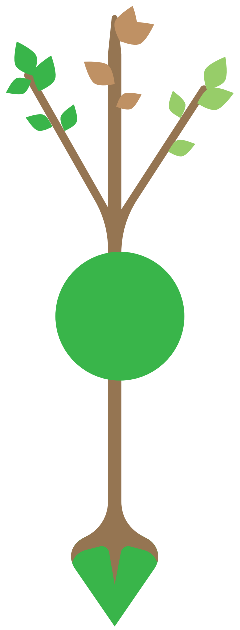




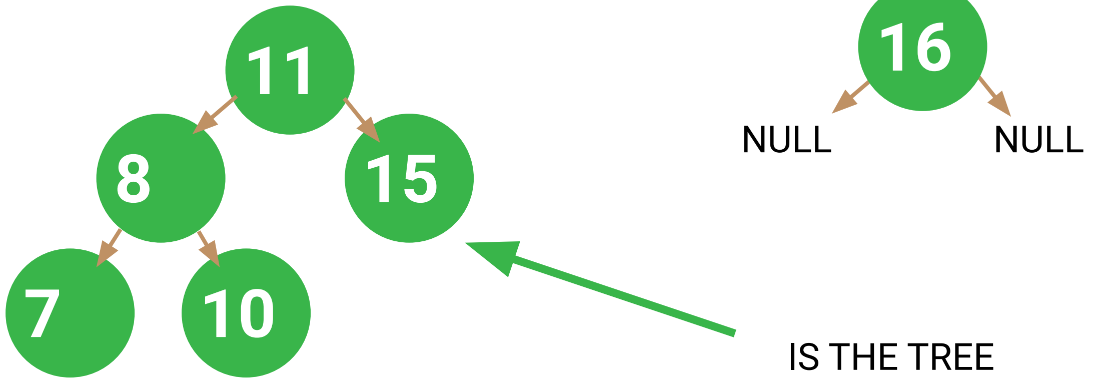


Step 2 - Check whether tree is Empty.

INSERT SIMULATION



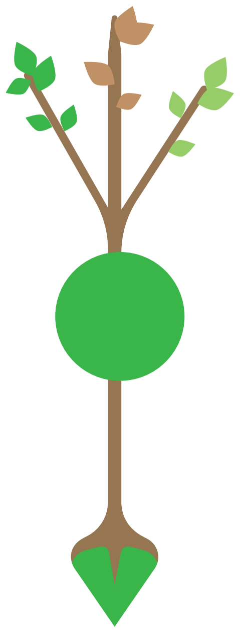
BS TREE

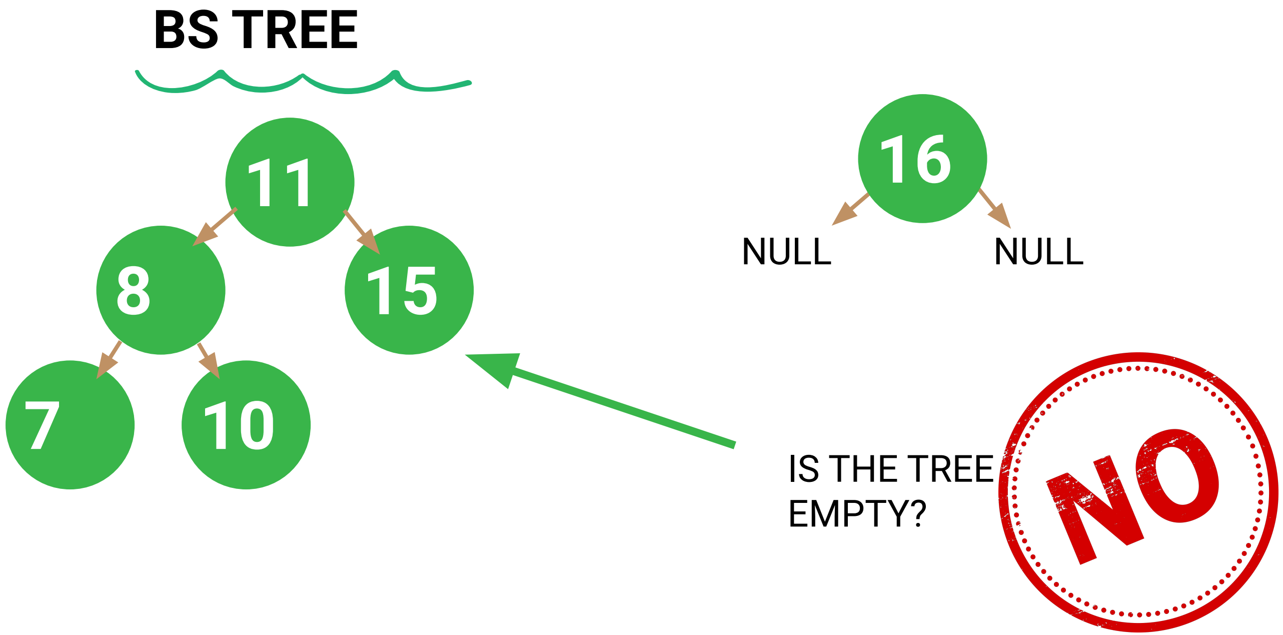


IS THE TREE
EMPTY?

Step 2 - Check whether tree is Empty.

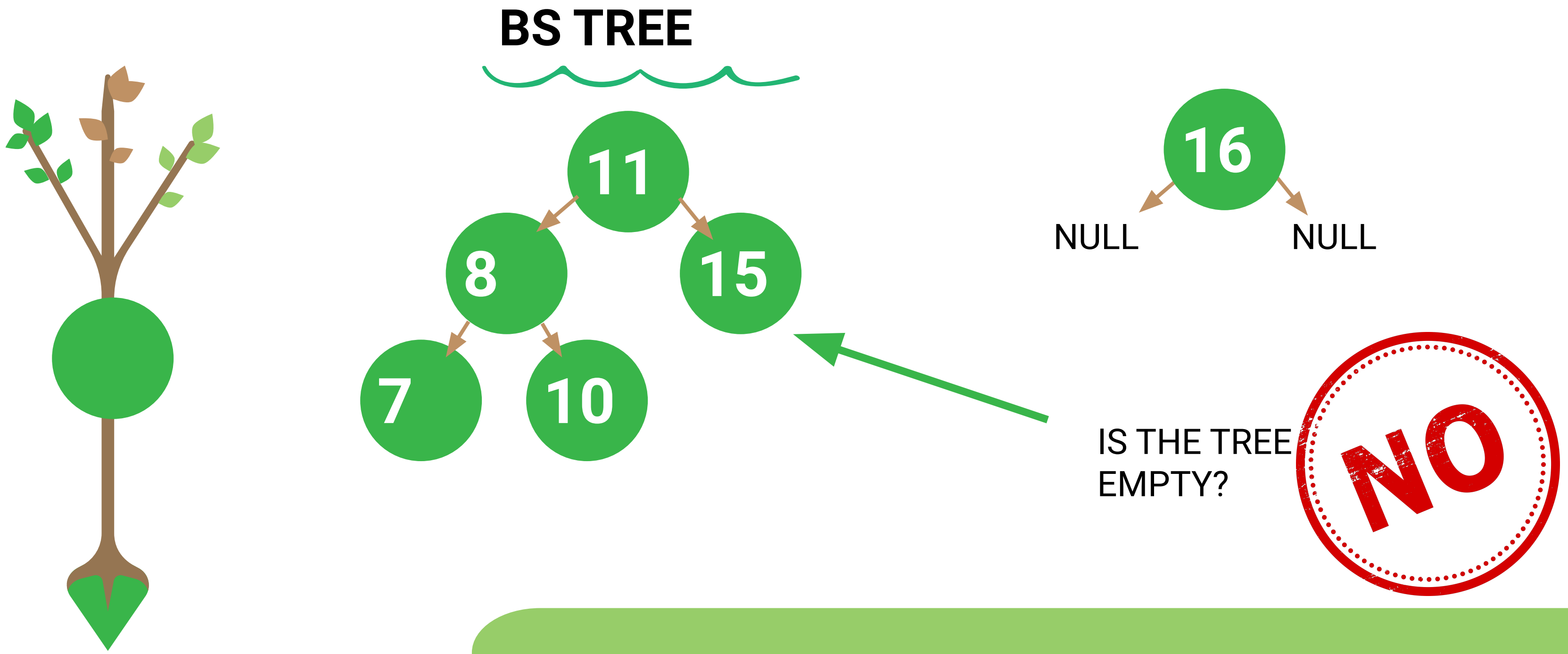
INSERT SIMULATION





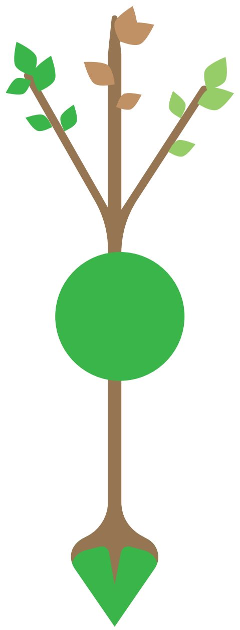
Step 2 - Check whether tree is Empty.

INSERT SIMULATION

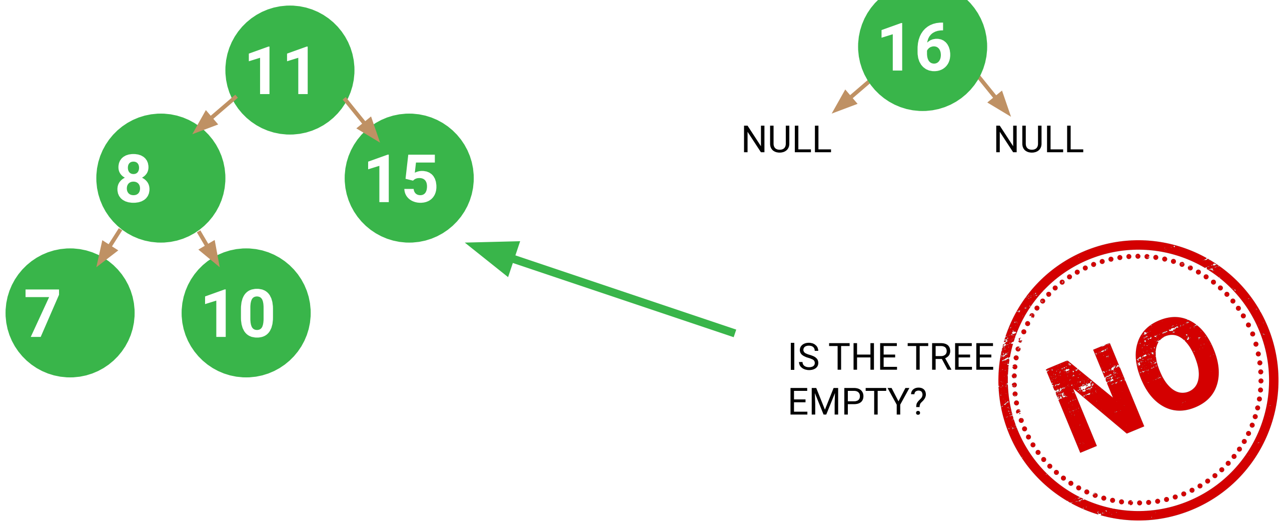


Step 3 - If the tree is Empty, then set root to newNode.

INSERT SIMULATION



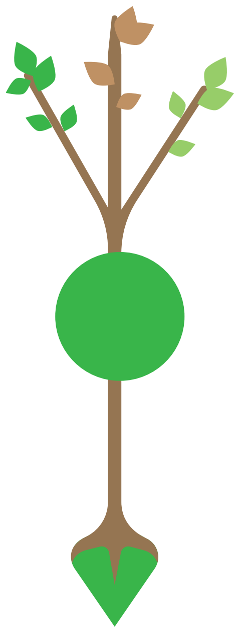
BS TREE

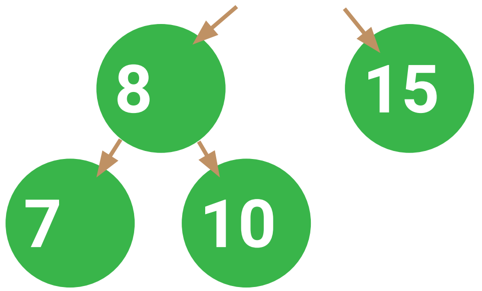


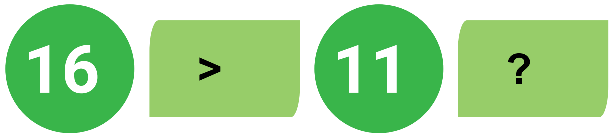
Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).

INSERT SIMULATION

BS TREE



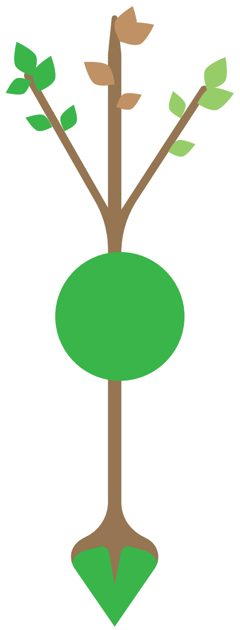


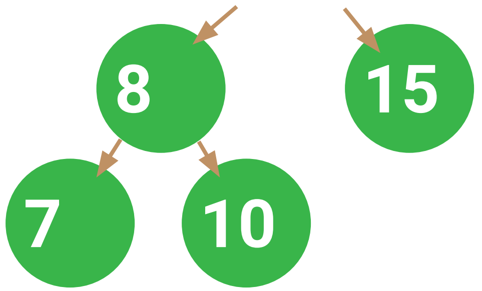


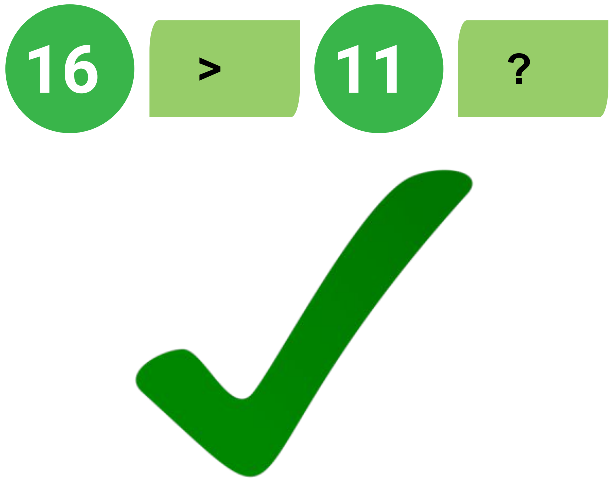
Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).

INSERT SIMULATION

BS TREE



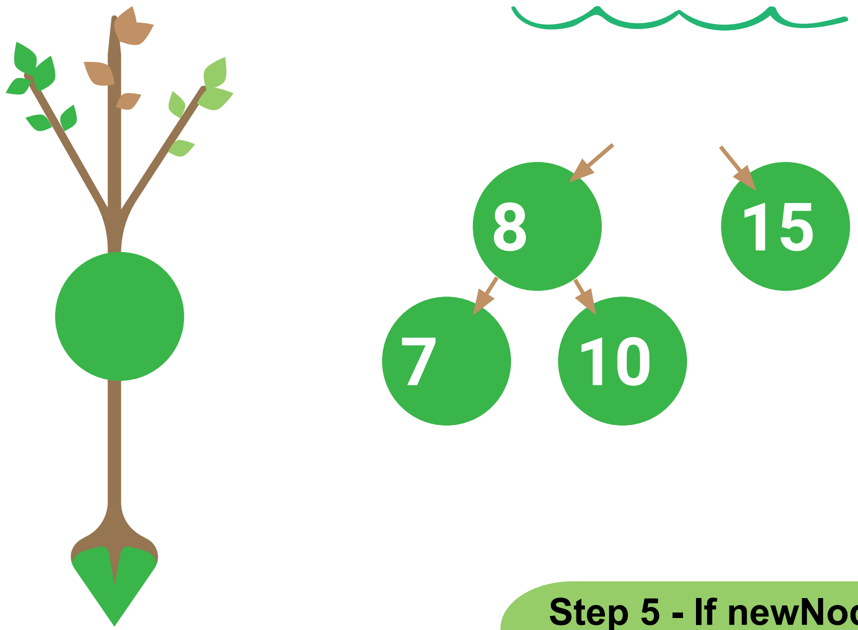


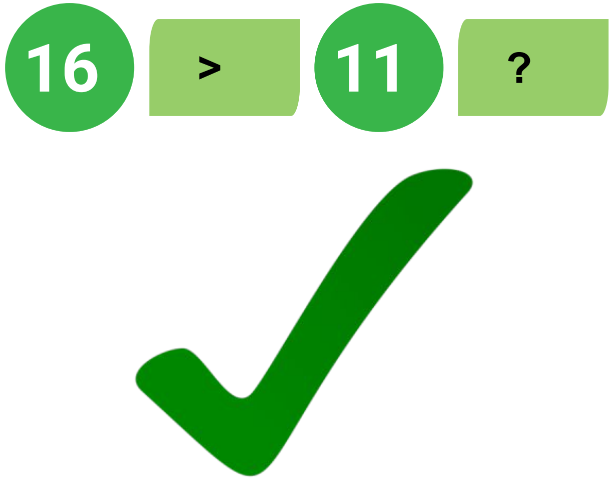


Step 4 - If the tree is Not Empty, then check whether the value of newNode is smaller or larger than the node (here it is root node).

INSERT SIMULATION

BS TREE

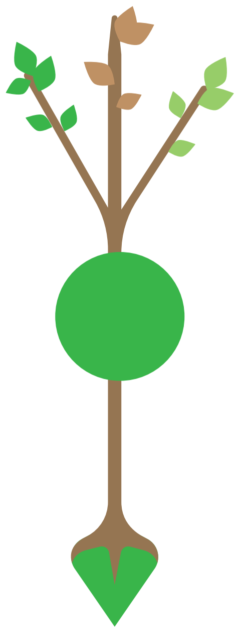


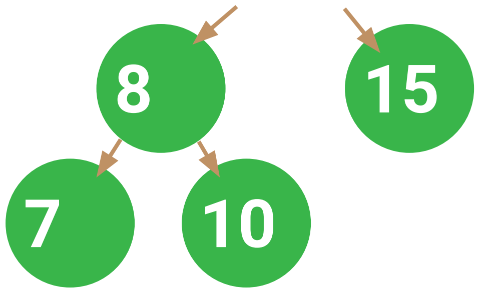


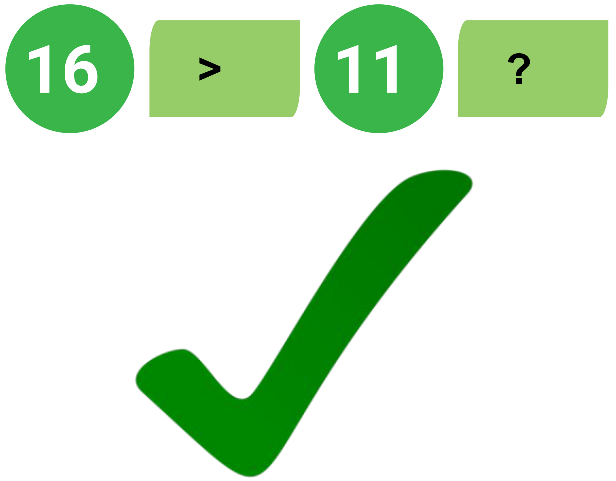
Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.

INSERT SIMULATION

BS TREE



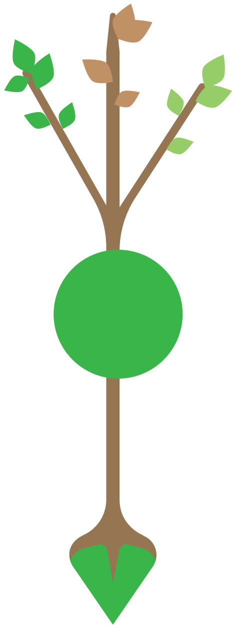


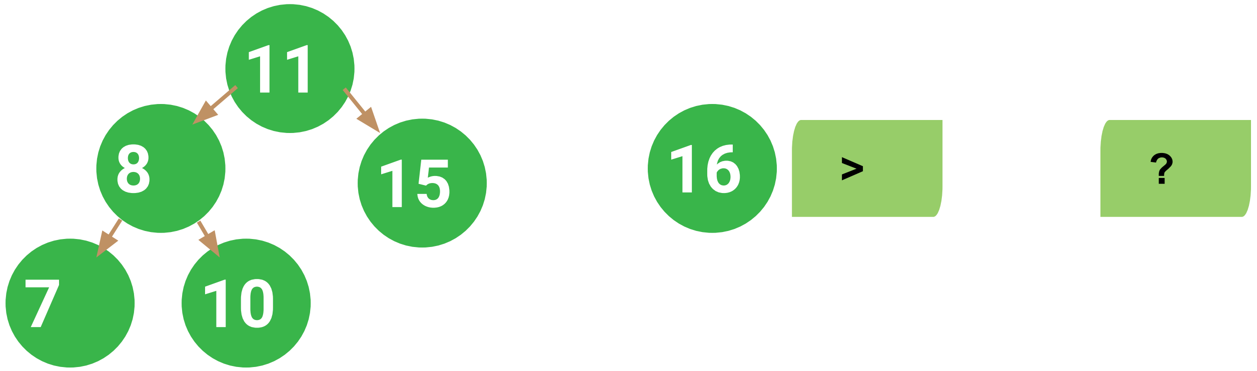


Step 5 - If newNode is smaller than or equal to the node then move to its left child. If newNode is larger than the node then move to its right child.

INSERT SIMULATION

BS TREE

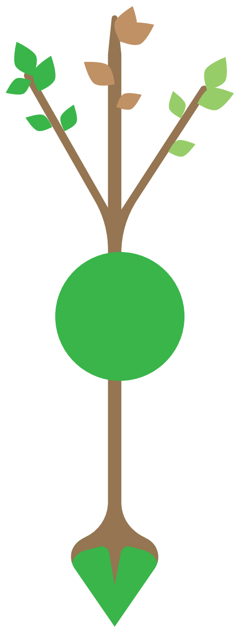


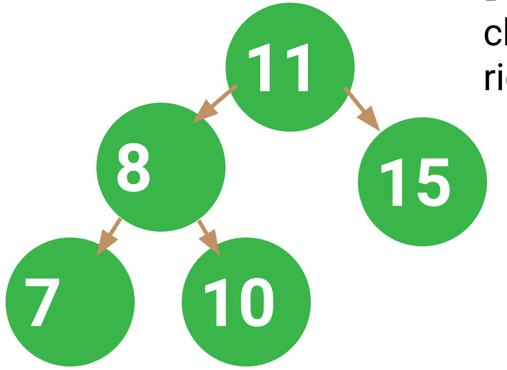


Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

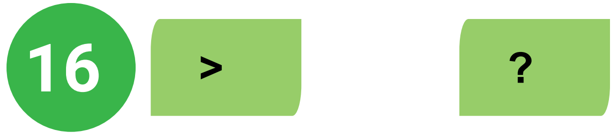
INSERT SIMULATION

BS TREE



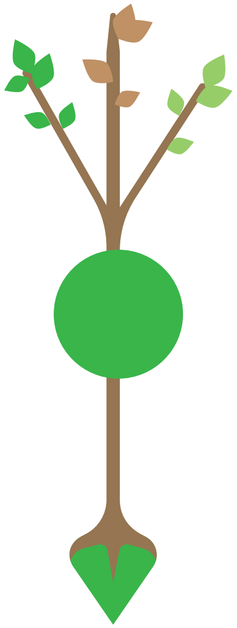


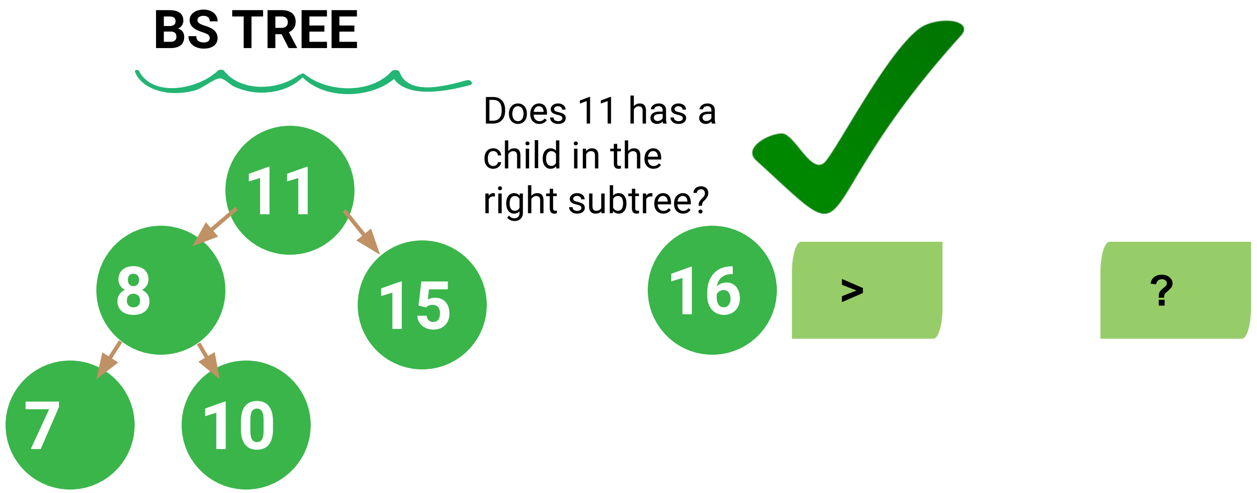
Does 11 has a child in the right subtree?



Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

INSERT SIMULATION

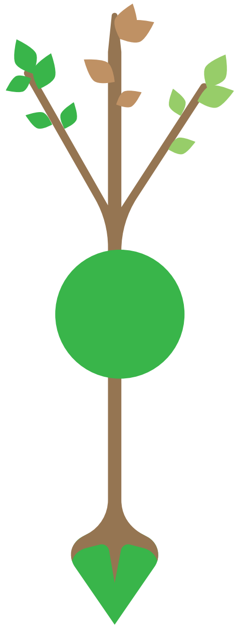


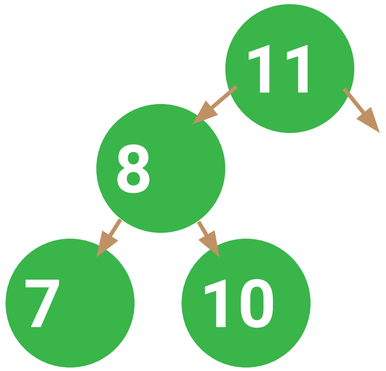


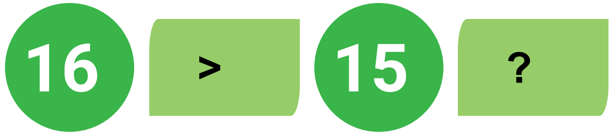
Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

INSERT SIMULATION

BS TREE



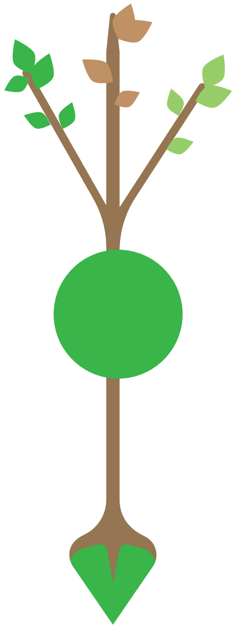


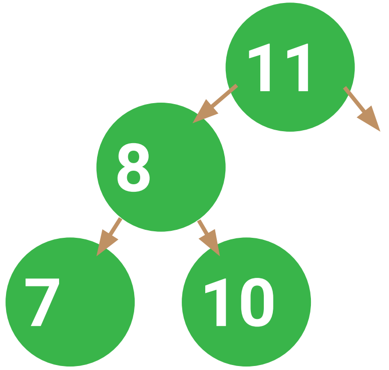


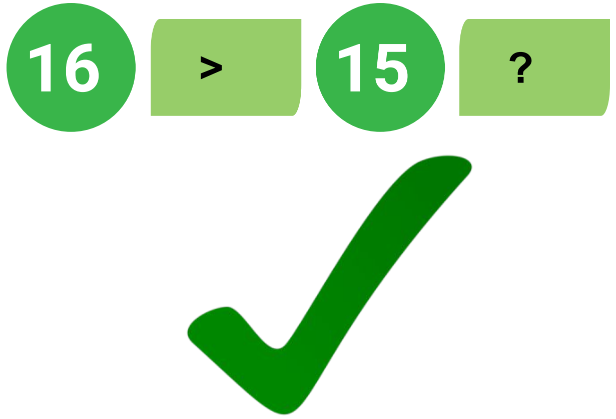
Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

INSERT SIMULATION

BS TREE



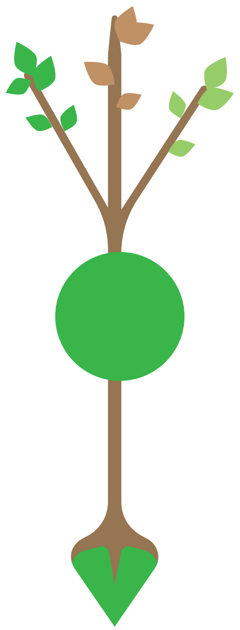


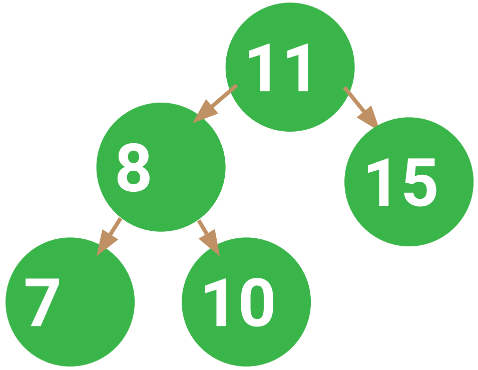


Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

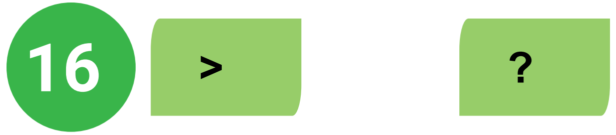
INSERT SIMULATION

BS TREE





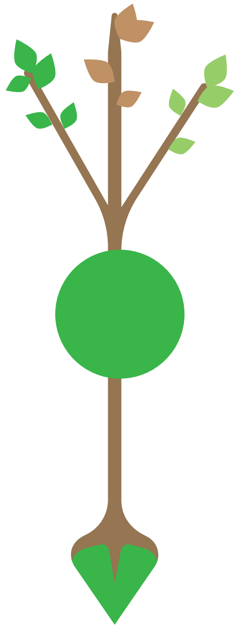
Does 15 has a child in the right subtree?

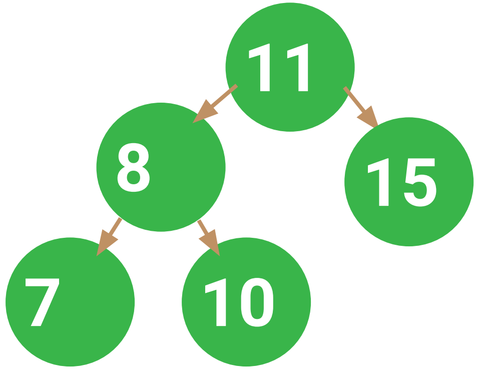


Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

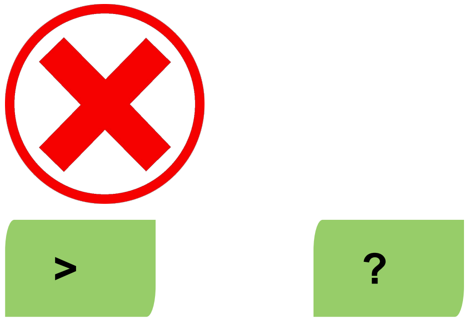
INSERT SIMULATION

BS TREE





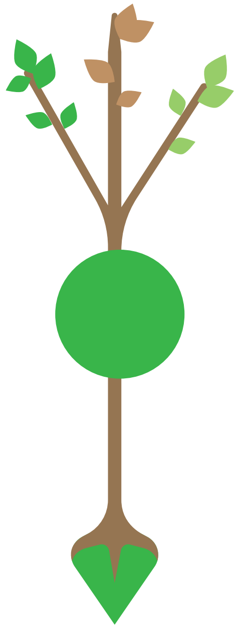
Does 15 has a child in the right subtree?

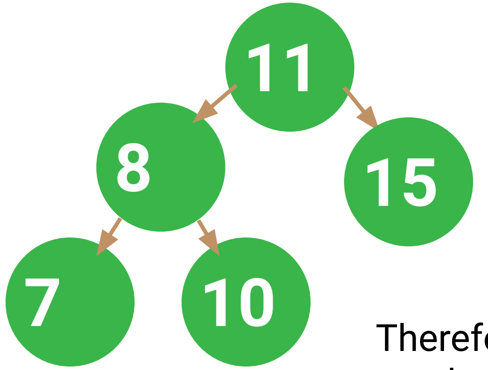


Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

INSERT SIMULATION

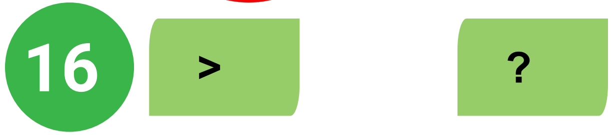
BS TREE





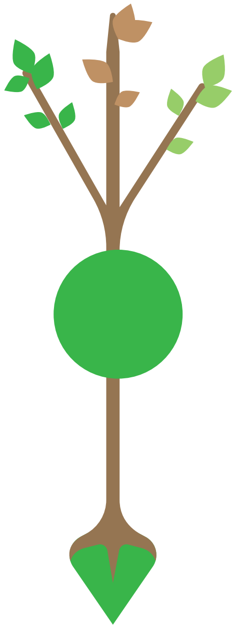
Does 15 has a child in the right subtree?

Therefore we reached the leaf node

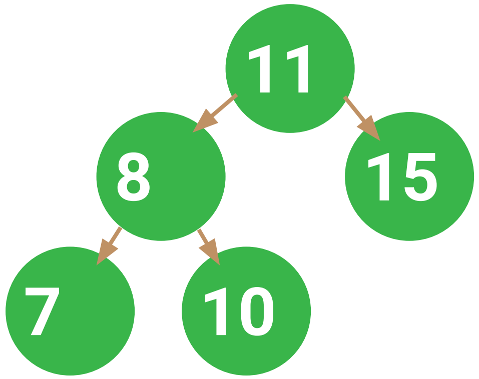


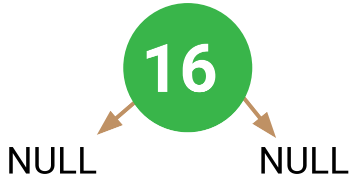
Step 6- Repeat the above steps until we reach to the leaf node (i.e., reaches to NULL).

INSERT SIMULATION



BS TREE

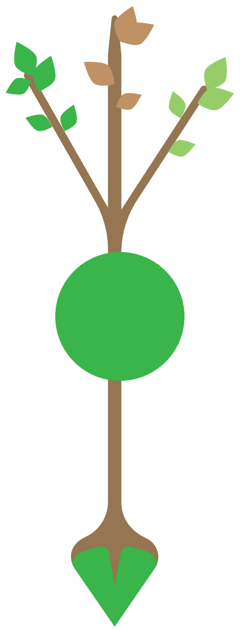


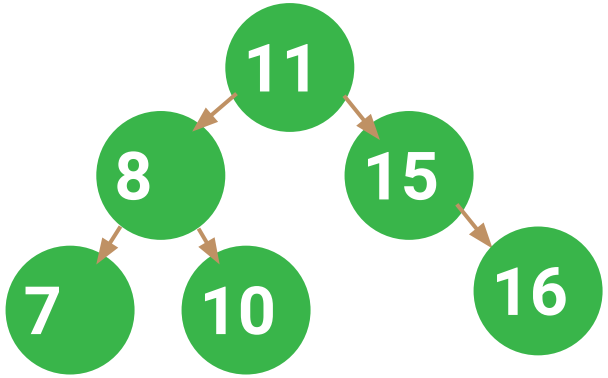


Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

INSERT SIMULATION

BS TREE





Step 7 - After reaching the leaf node, insert the newNode as left child if the newNode is smaller or equal to that leaf node or else insert it as right child.

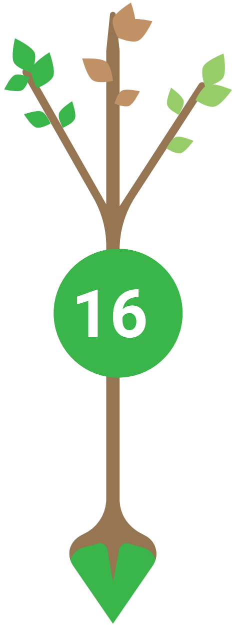
DELETE SIMULATION

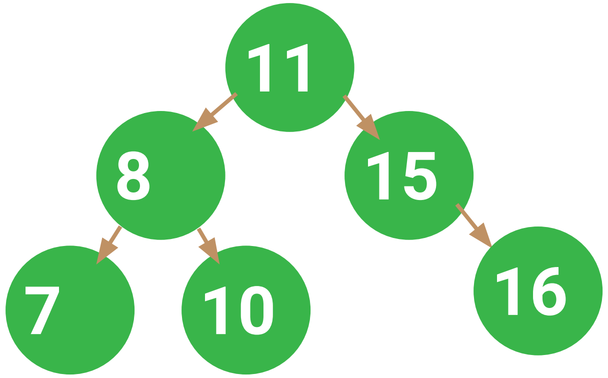
BS TREE - 3 cases

1. Case 1: Deleting a Leaf node (A node with no children)
2. Case 2: Deleting a node with one child
3. Case 3: Deleting a node with two children

DELETE SIMULATION

BS TREE - CASE 1

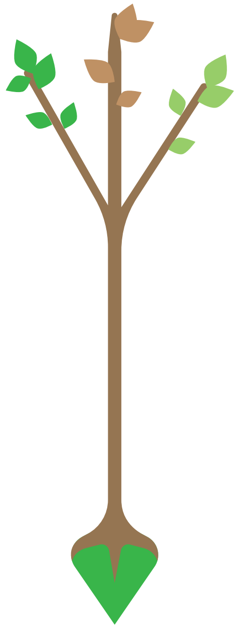


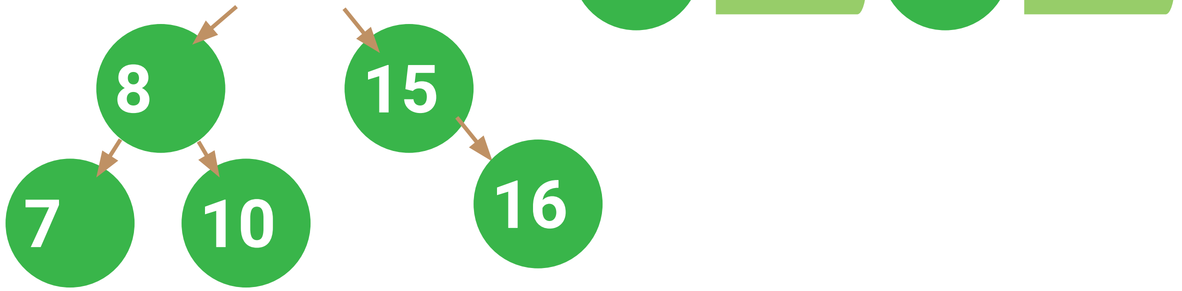


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 1

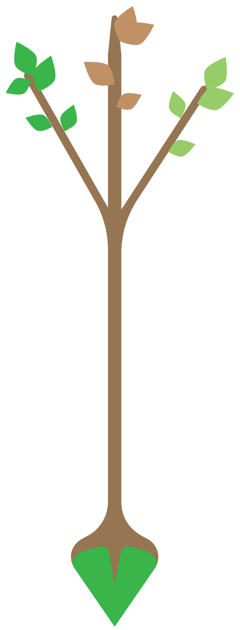


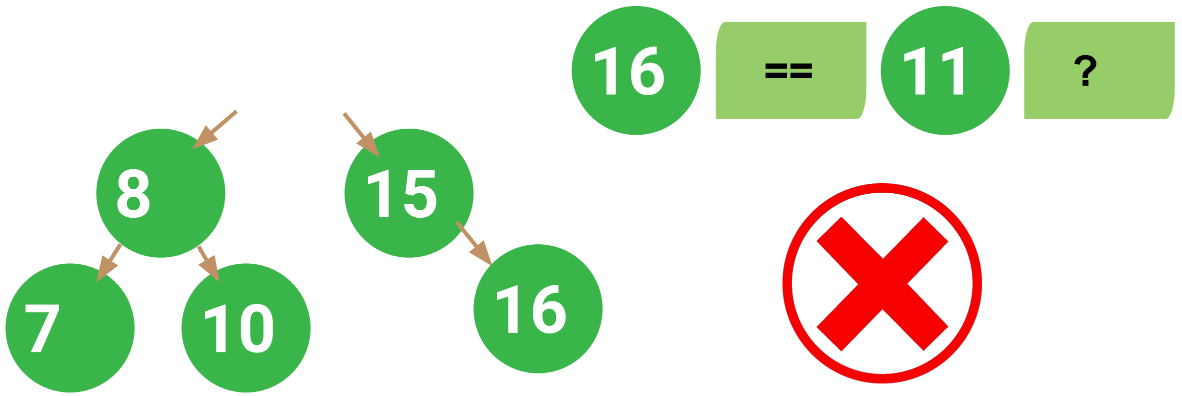


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 1

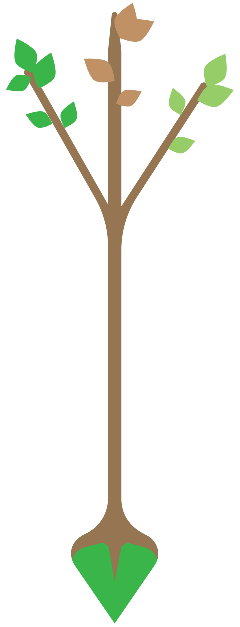


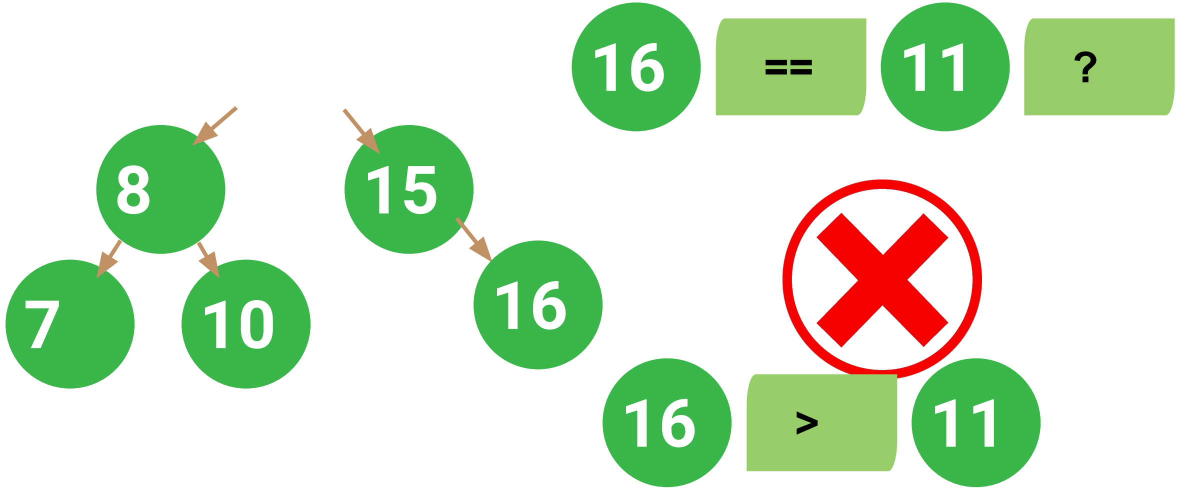


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 1

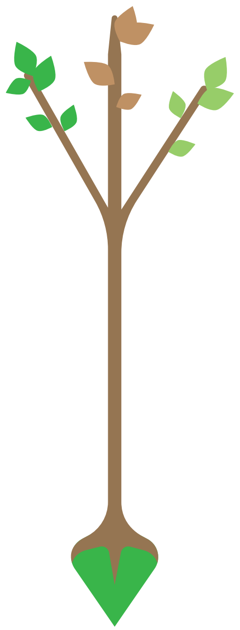


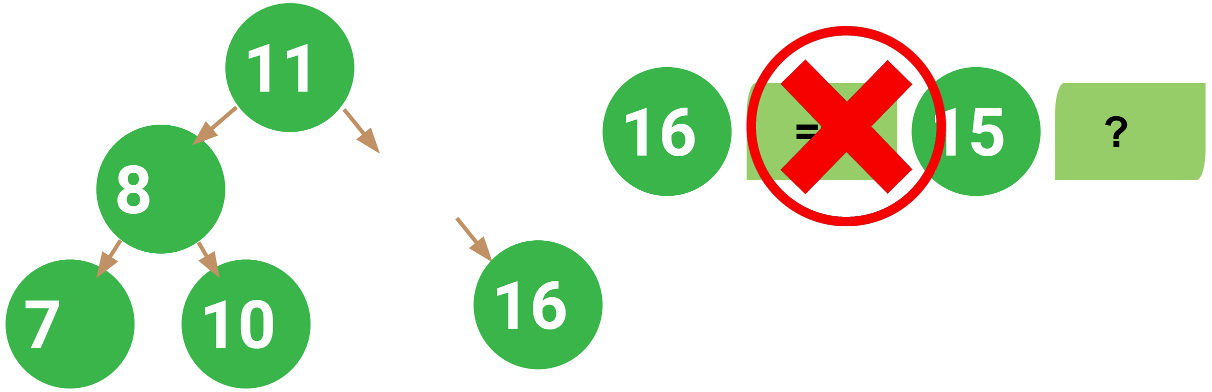


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 1

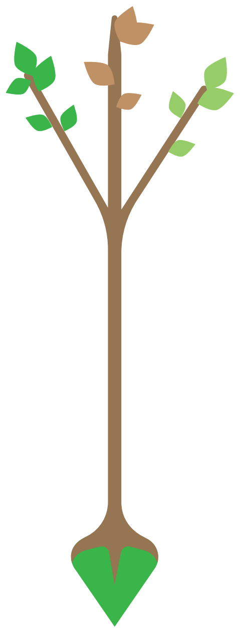


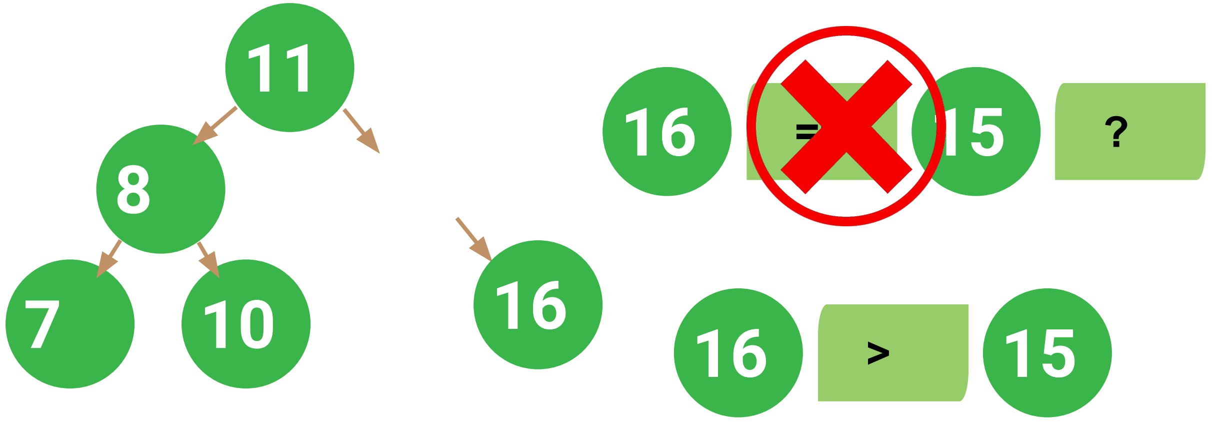


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 1

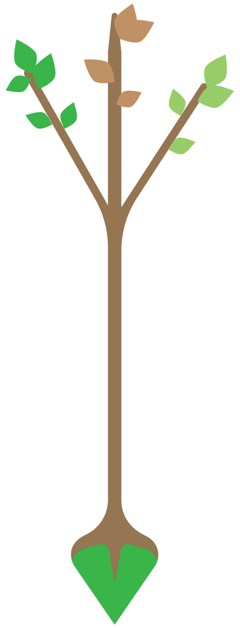


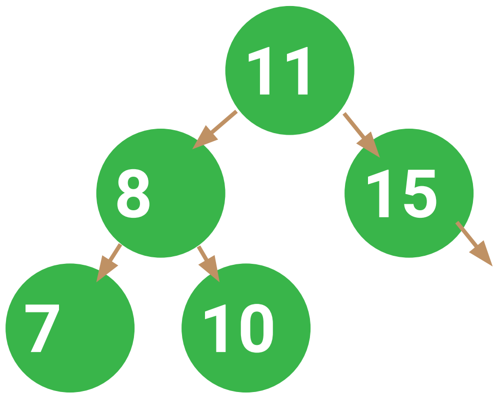


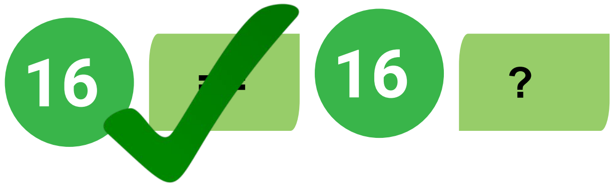
Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 1



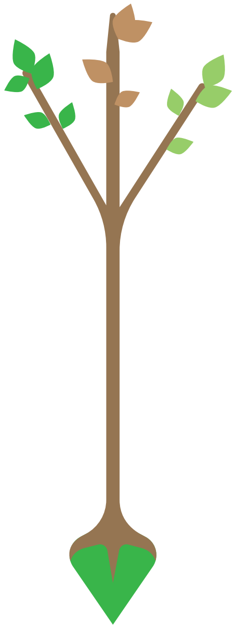


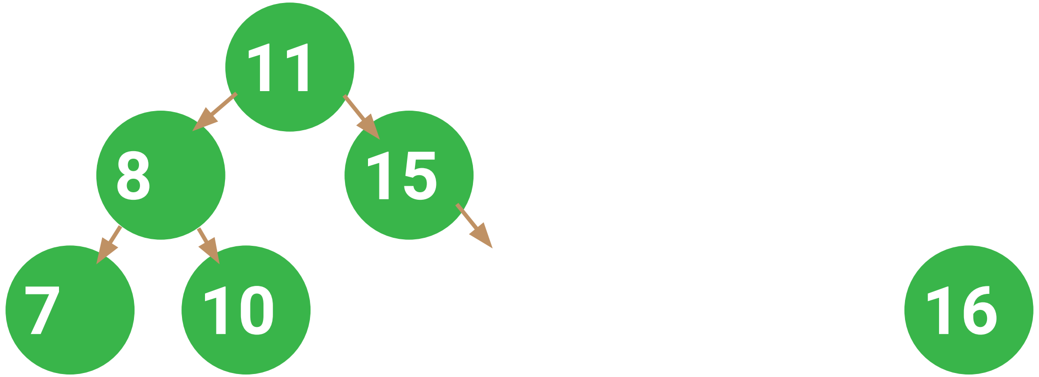


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 1

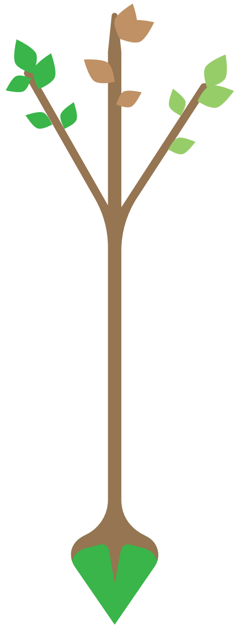


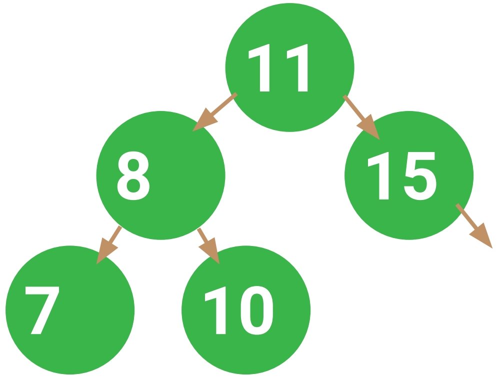


Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

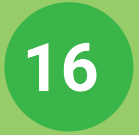
DELETE SIMULATION

BS TREE - CASE 1





FREE

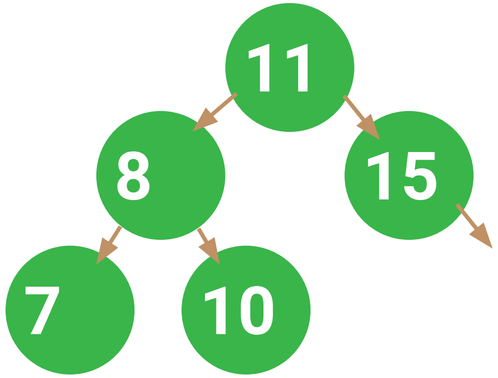


Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

DELETE SIMULATION

BS TREE - CASE 1

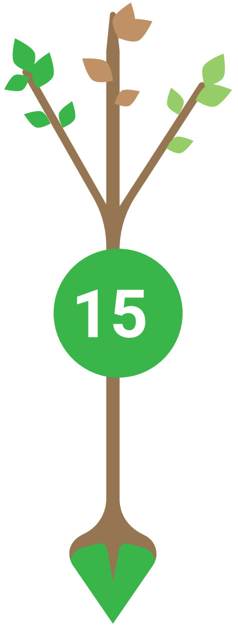
Function terminate

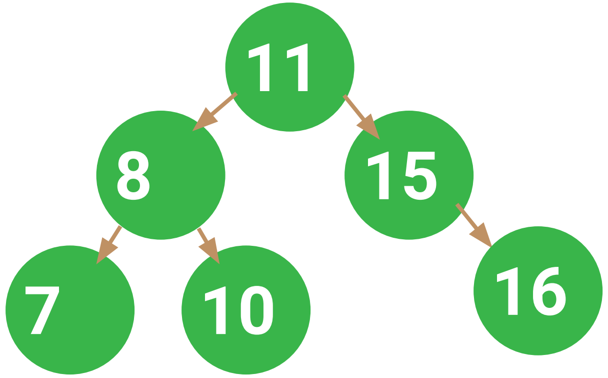


Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.

DELETE SIMULATION

BS TREE - CASE 2

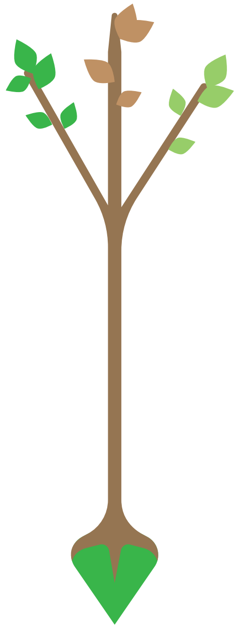


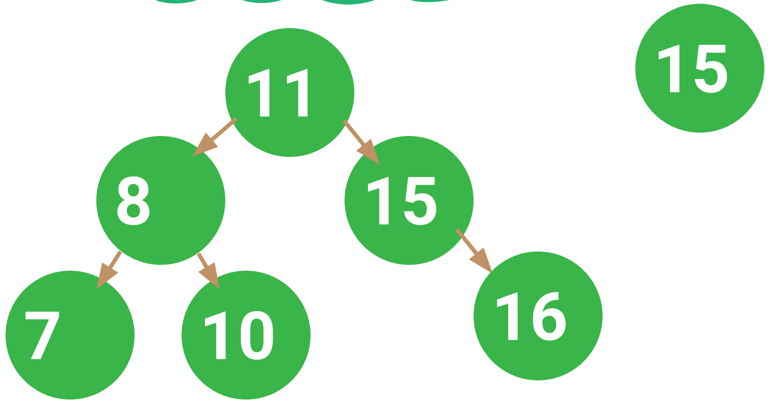


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 2

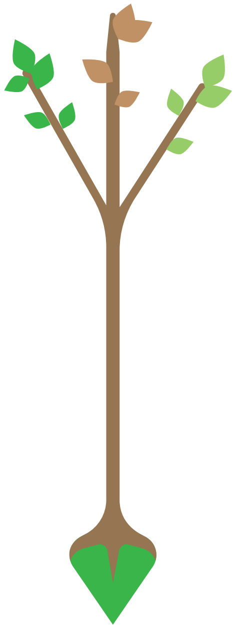


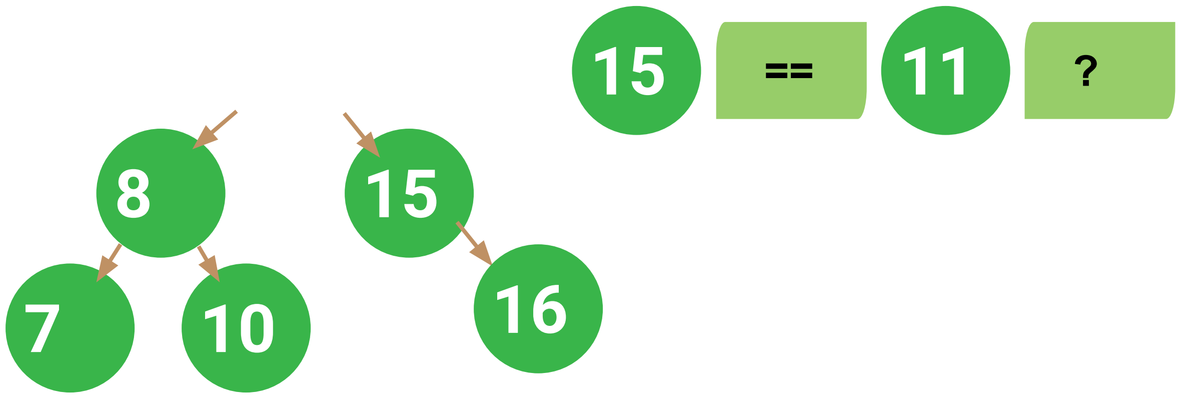


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 2

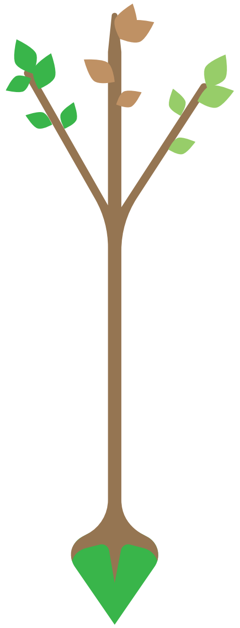


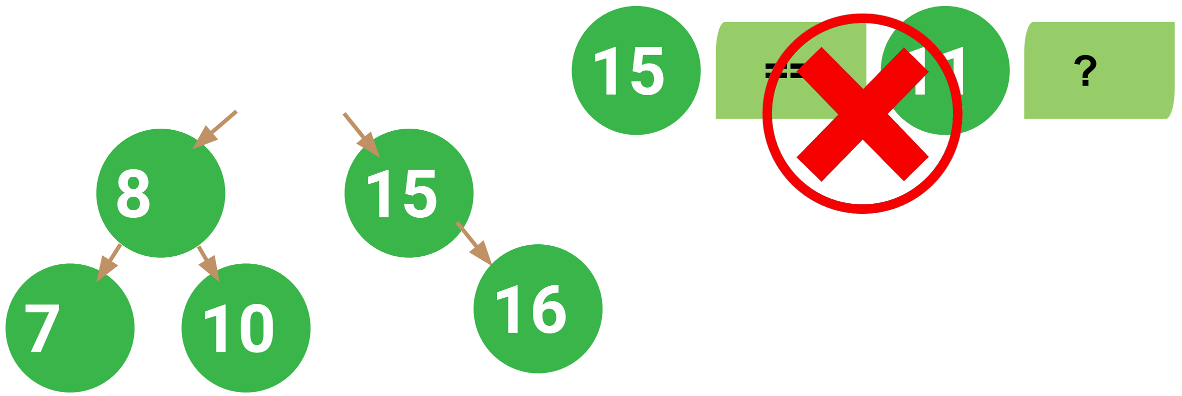


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 2

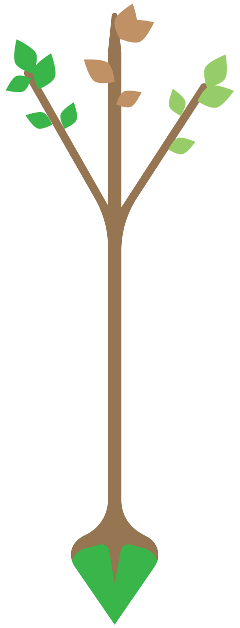


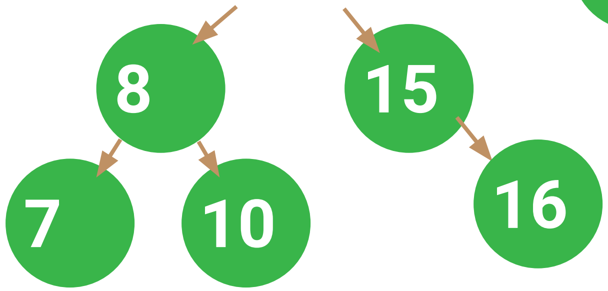


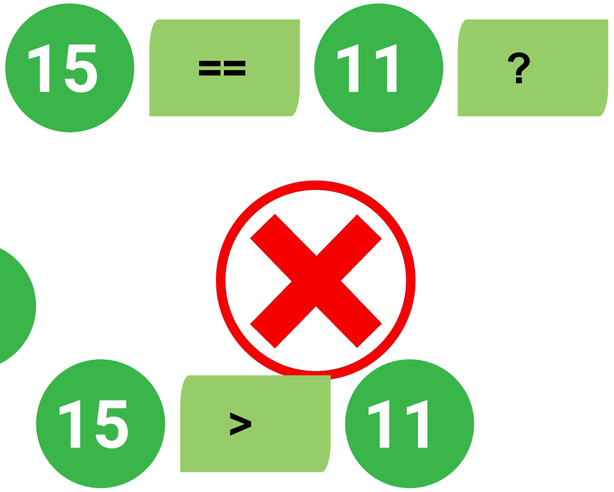
Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 2



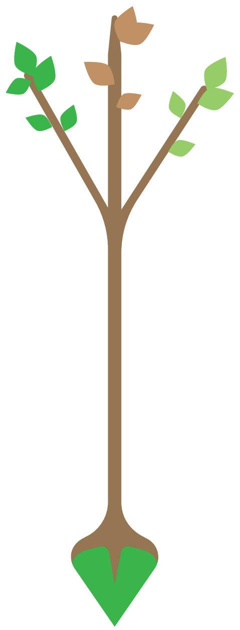


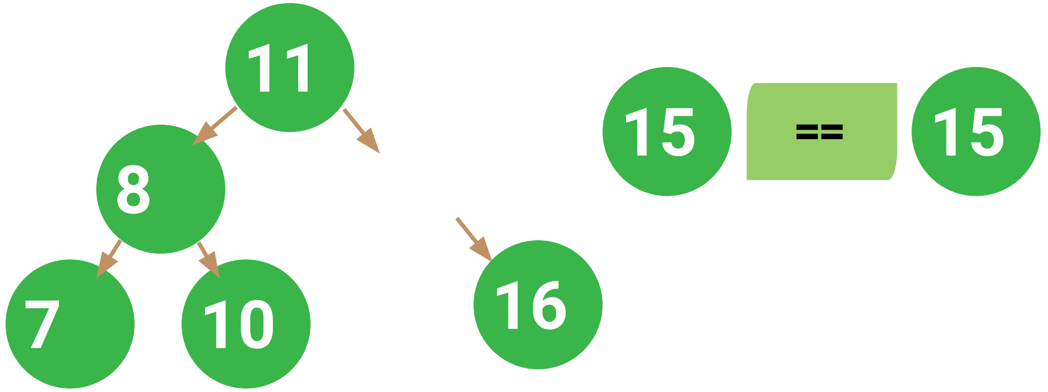


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 2

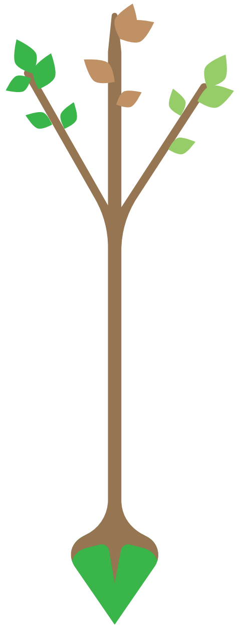


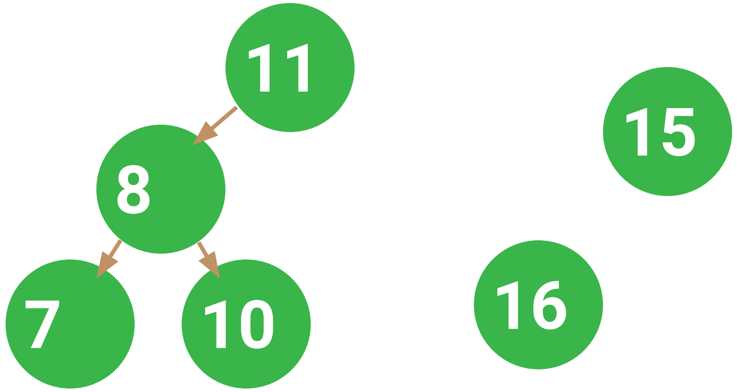


Step 1 - Find the node to be deleted using search operation

DELETE SIMULATION

BS TREE - CASE 2

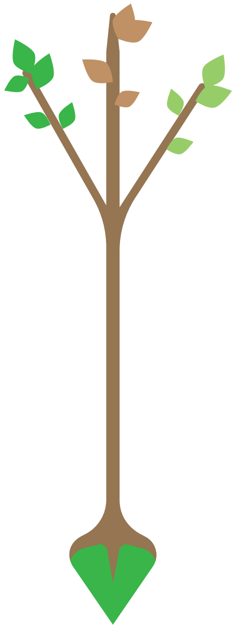


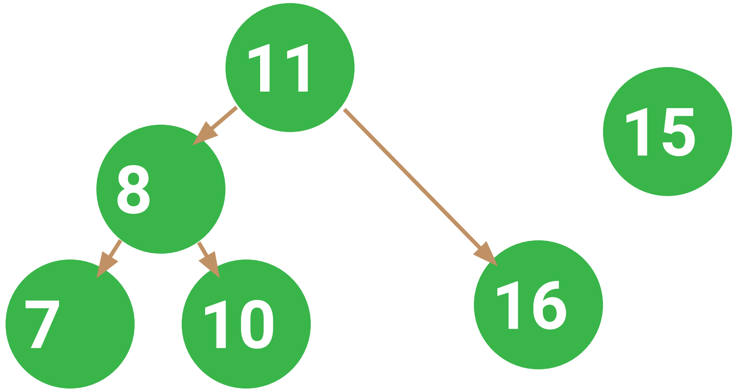


Step 2 - If it has only one child then create a link between its parent node and child node.

DELETE SIMULATION

BS TREE - CASE 2

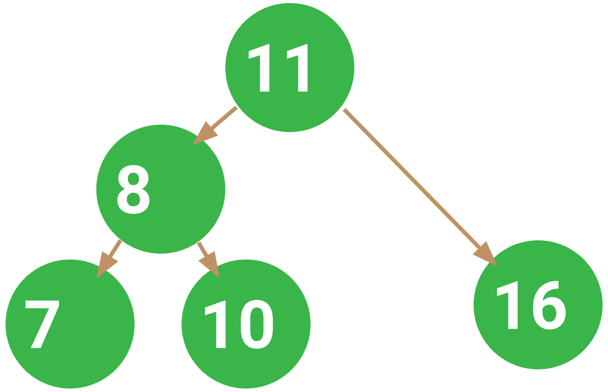


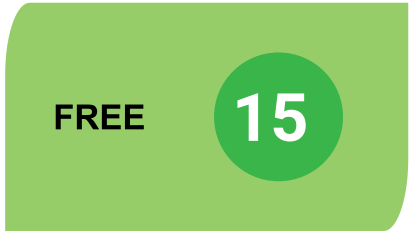


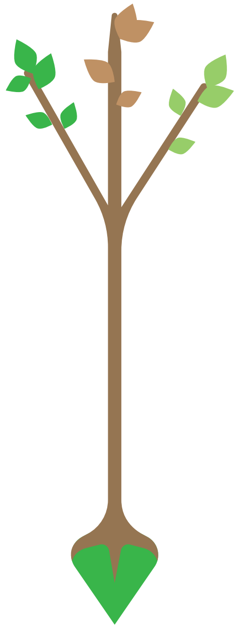
Step 2 - If it has only one child then create a link between its parent node and child node.

DELETE SIMULATION

BS TREE - CASE 2





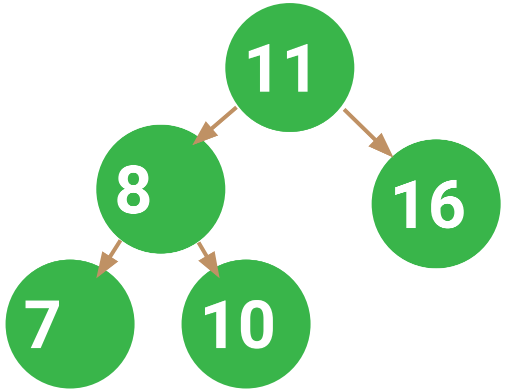


Step 3 - Delete the node using free function and terminate the function.

DELETE SIMULATION

BS TREE - CASE 2

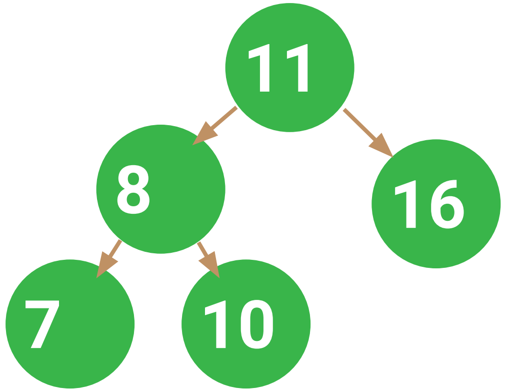
Function terminate



Step 3 - Delete the node using free function and terminate the function.

DELETE SIMULATION

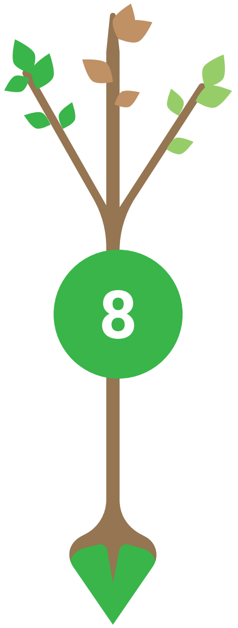
BS TREE - CASE 3

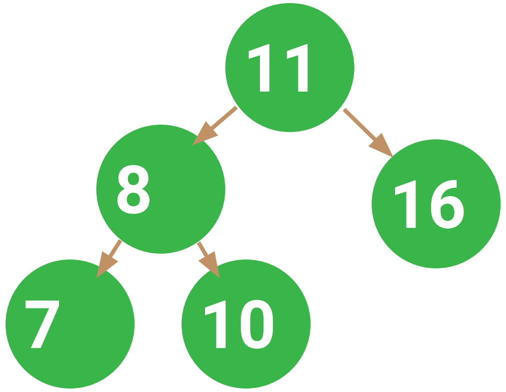


Step 1 - Find the node to be deleted using search operation.

DELETE SIMULATION

BS TREE - CASE 3

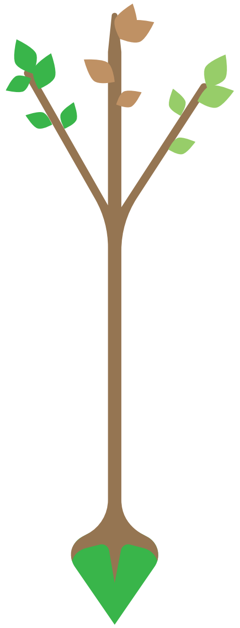


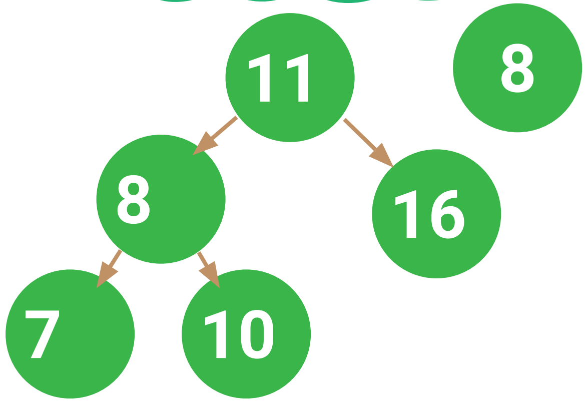


Step 1 - Find the node to be deleted using search operation.

DELETE SIMULATION

BS TREE - CASE 3

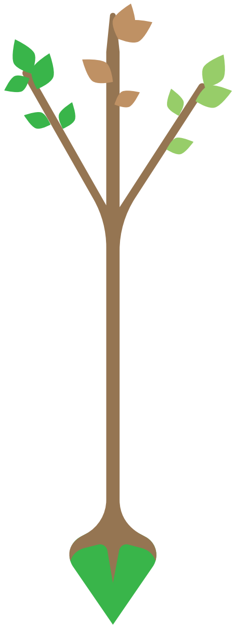


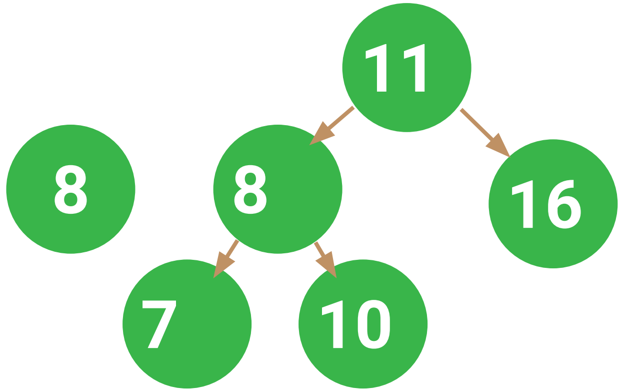


Step 1 - Find the node to be deleted using search operation.

DELETE SIMULATION

BS TREE - CASE 3

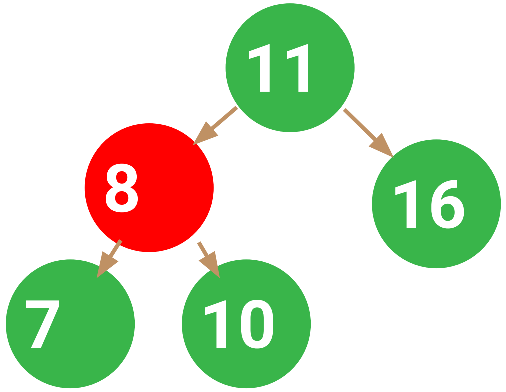


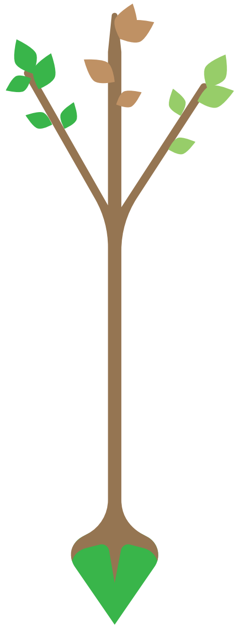


Step 1 - Find the node to be deleted using search operation.

DELETE SIMULATION

BS TREE - CASE 3

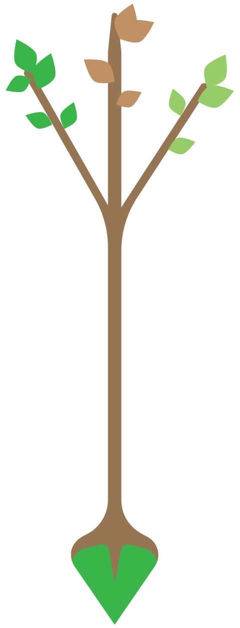


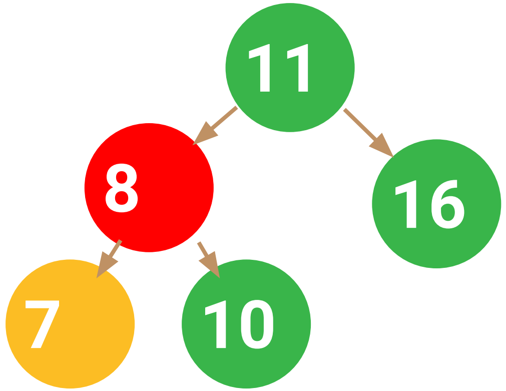


Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

DELETE SIMULATION

BS TREE - CASE 3

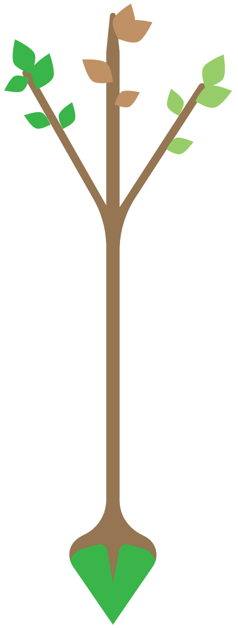


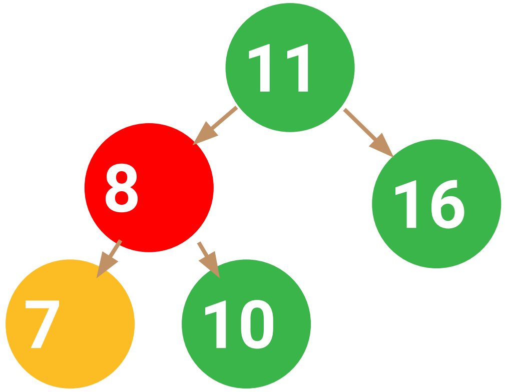


Step 2 - If it has two children, then find the largest node in its left subtree (OR) the smallest node in its right subtree.

DELETE SIMULATION

BS TREE - CASE 3

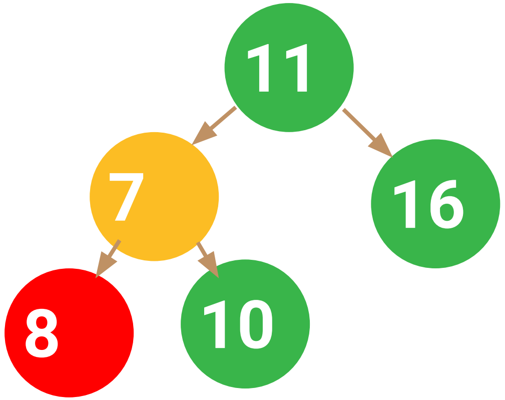


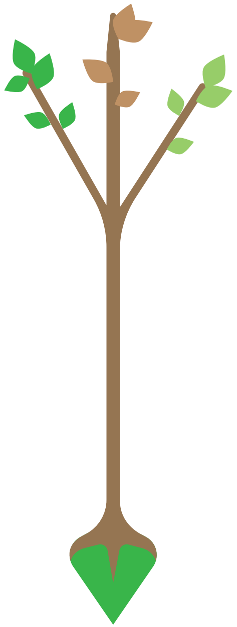


Step 3 - Swap both deleting node and node which is found in the above step

DELETE SIMULATION

BS TREE - CASE 3

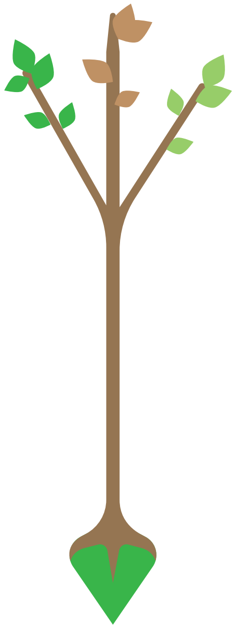


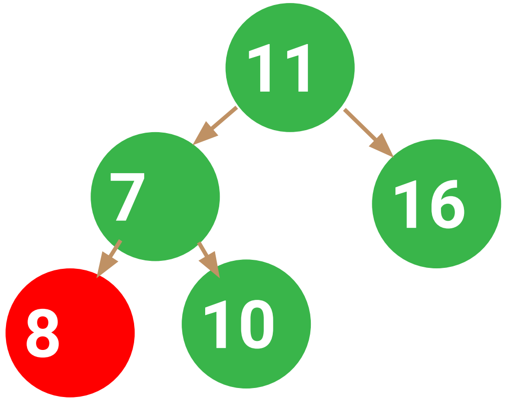


Step 3 - Swap both deleting node and node which is found in the above step

DELETE SIMULATION

BS TREE - CASE 3

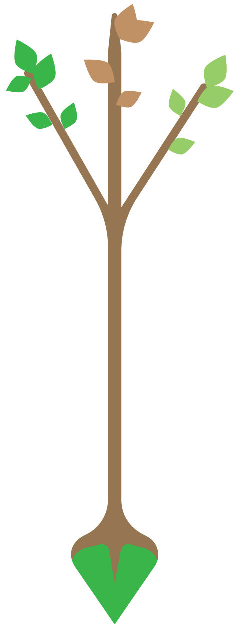


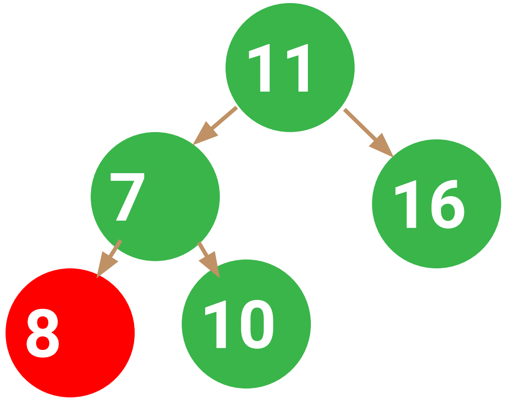


Step 4 - Then check whether deleting node came to case 1 or case 2 or else goto step 2

DELETE SIMULATION

BS TREE - CASE 3

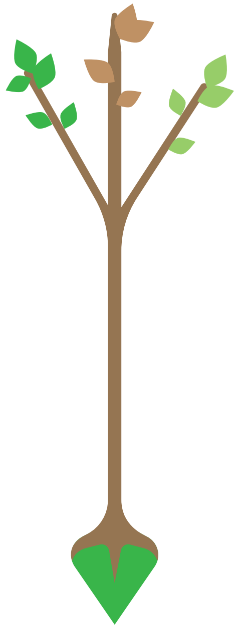


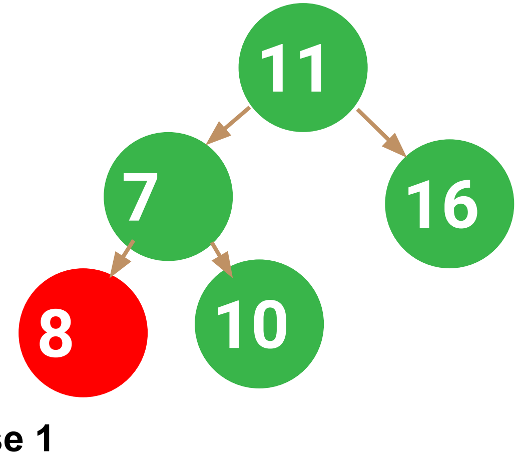


Step 5 - If it comes to case 1, then delete using case 1 logic.
Step 6- If it comes to case 2, then delete using case 2 logic.

DELETE SIMULATION

BS TREE - CASE 3

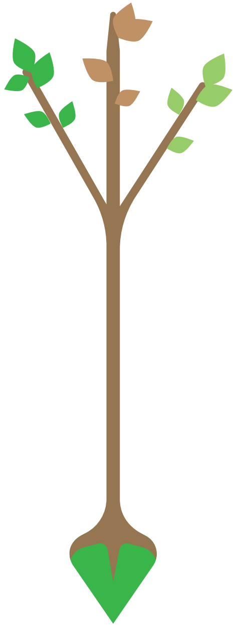


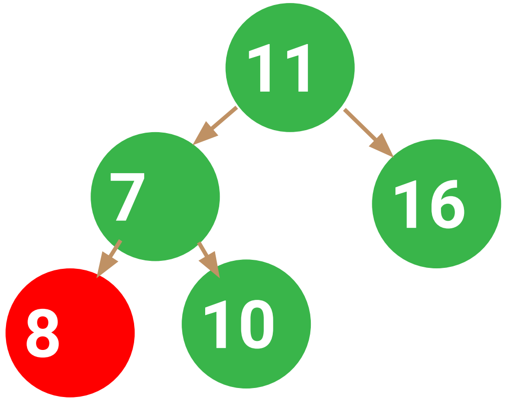


Step 5 - If it comes to case 1, then delete using case 1 logic.
Step 6- If it comes to case 2, then delete using case 2 logic.

DELETE SIMULATION

BS TREE - CASE 3



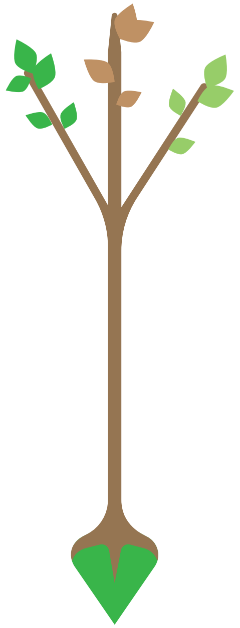


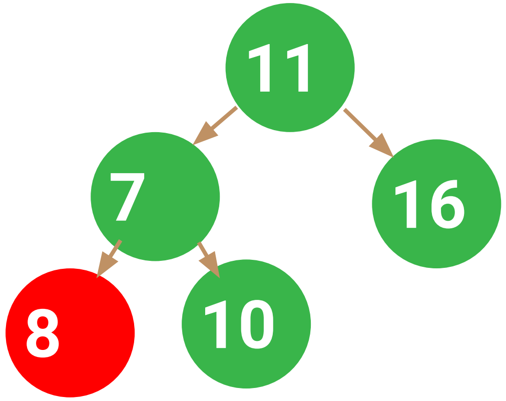
case 1

Step 5 - If it comes to case 1, then delete using case 1 logic.
Step 6- If it comes to case 2, then delete using case 2 logic.

DELETE SIMULATION

BS TREE - CASE 3



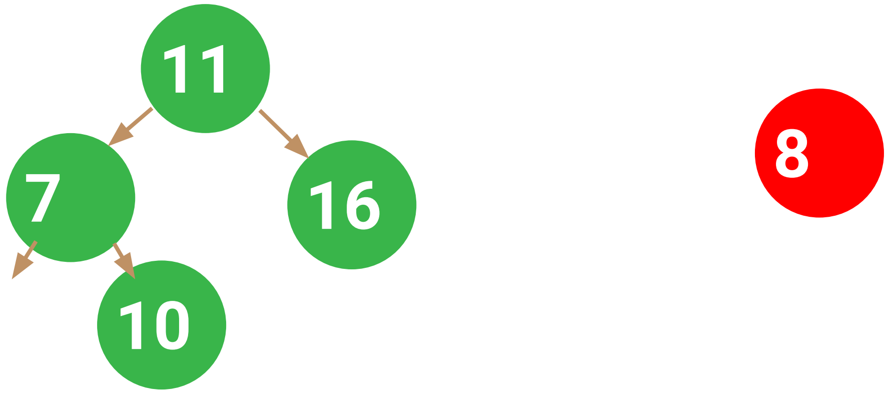


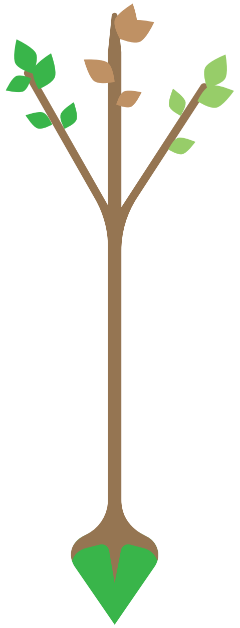
case 1

Step 7 - Repeat the same process until the node is deleted from the tree.

DELETE SIMULATION

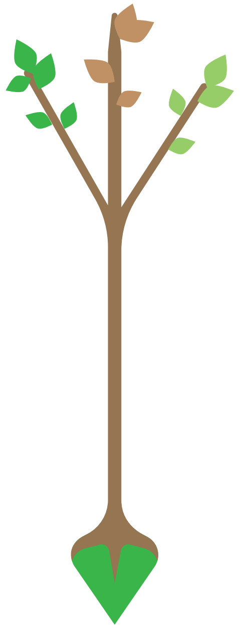
BS TREE - CASE 3

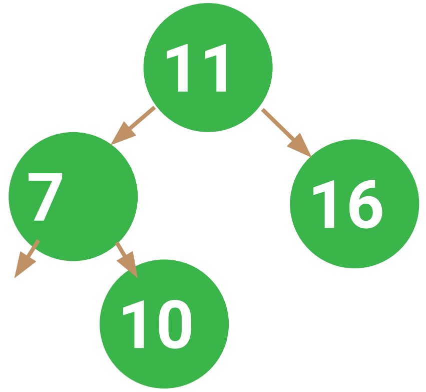


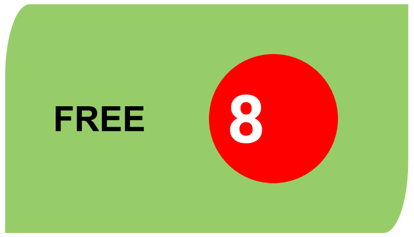


DELETE SIMULATION

BS TREE - CASE 3

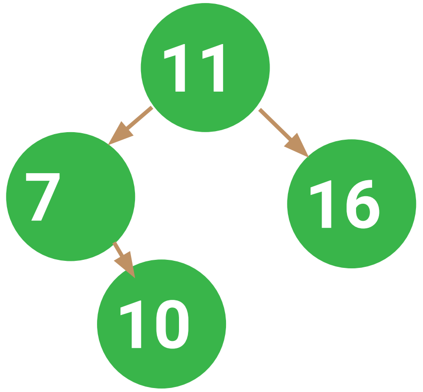






DELETE SIMULATION

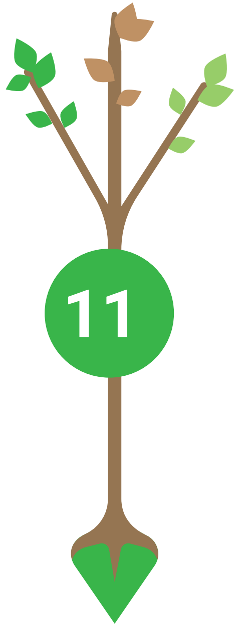
BS TREE - CASE 3



Function terminate

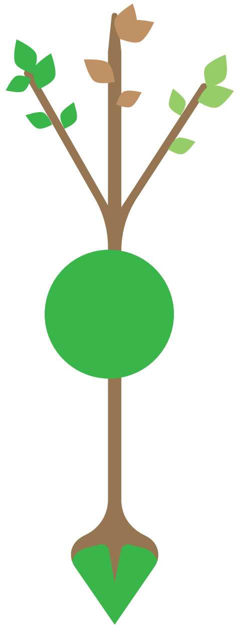
SEARCH SIMULATION

AVL TREE



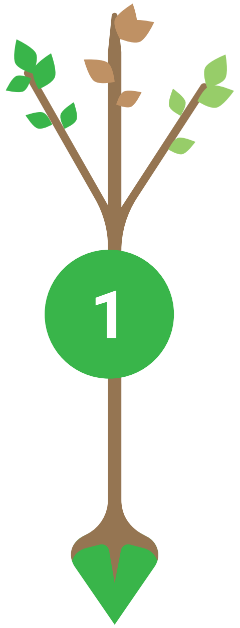
INSERTION SIMULATION

AVL TREE



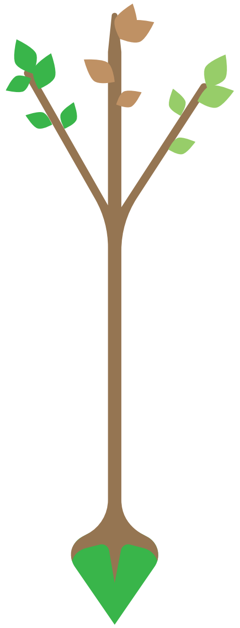
INSERTION SIMULATION

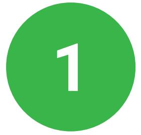
AVL TREE



INSERTION SIMULATION

AVL TREE

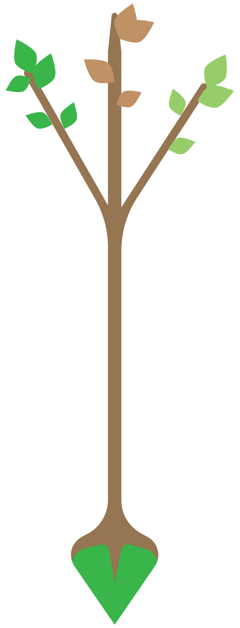


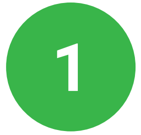


Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

INSERTION SIMULATION

AVL TREE

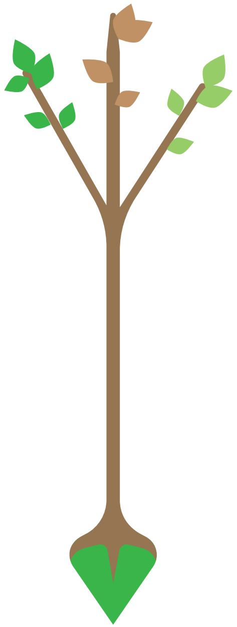




Step 2 - After insertion, check the Balance Factor of every node.

INSERTION SIMULATION

AVL TREE



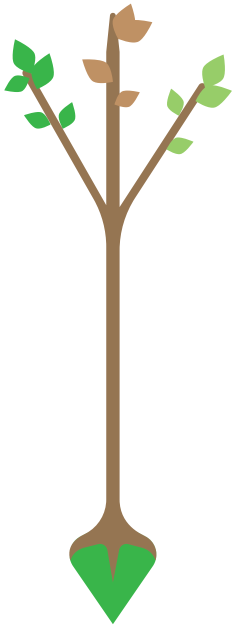
1

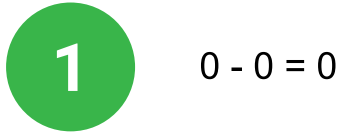
HL - HR = BF

Step 2 - After insertion, check the Balance Factor of every node.

INSERTION SIMULATION

AVL TREE

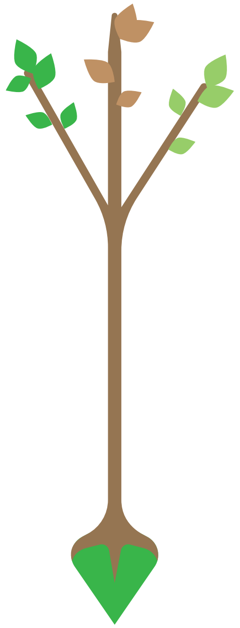


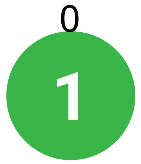


Step 2 - After insertion, check the Balance Factor of every node.

INSERTION SIMULATION

AVL TREE

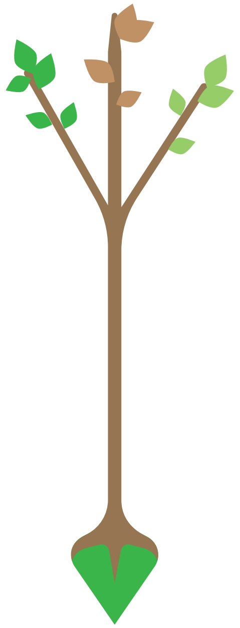


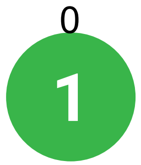


Step 2 - After insertion, check the Balance Factor of every node.

INSERTION SIMULATION

AVL TREE

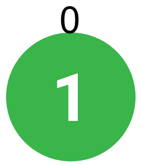


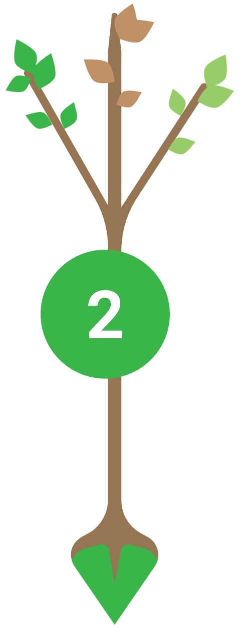


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

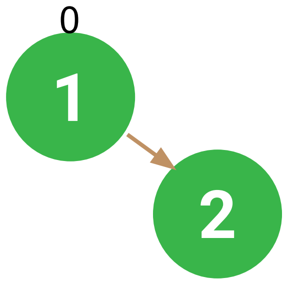


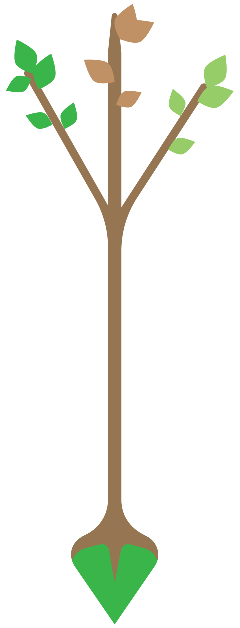


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

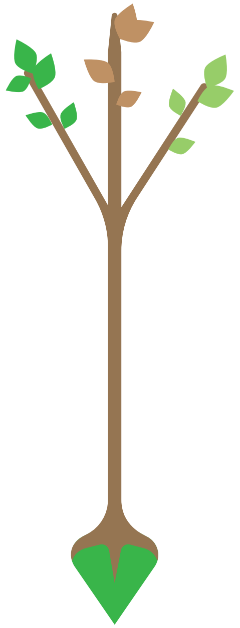


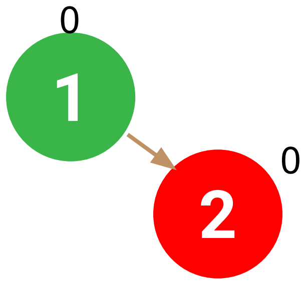


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

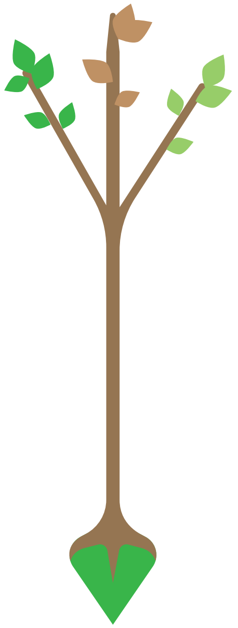


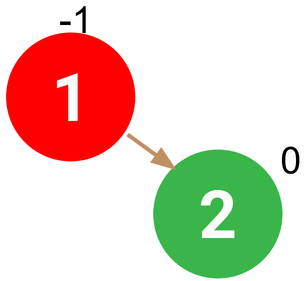


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

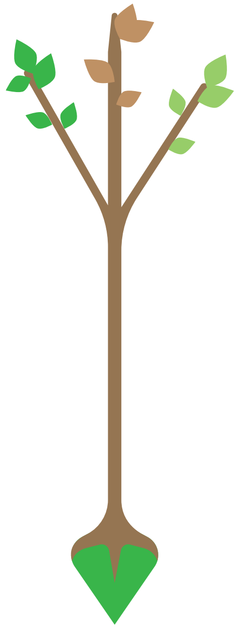


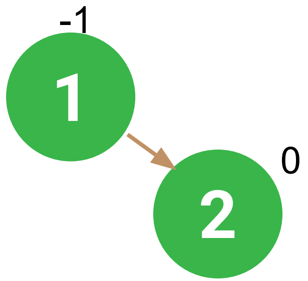


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

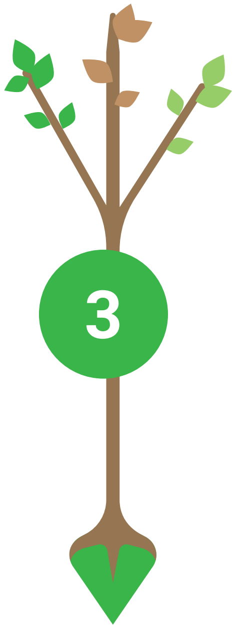


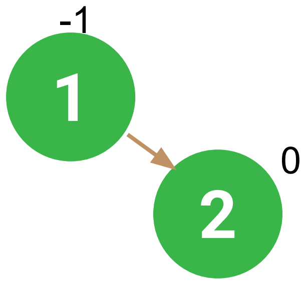


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

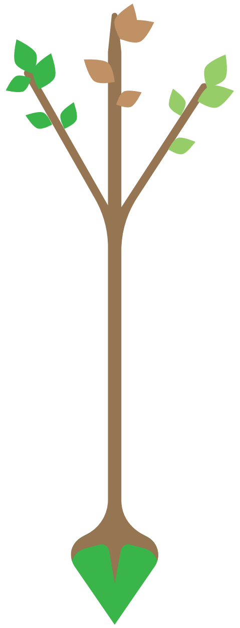


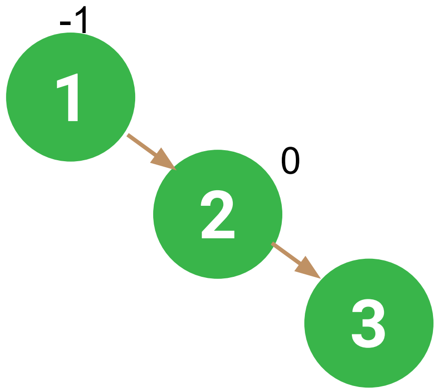


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

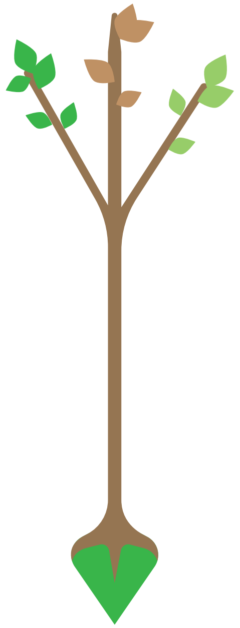


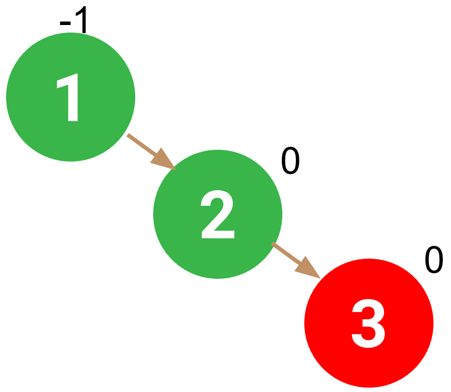


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

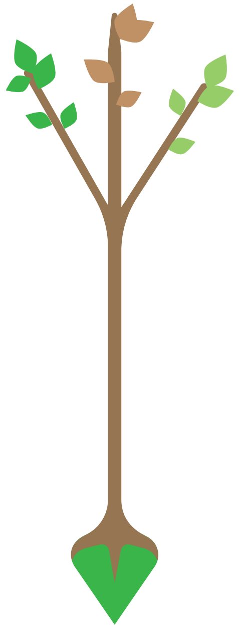


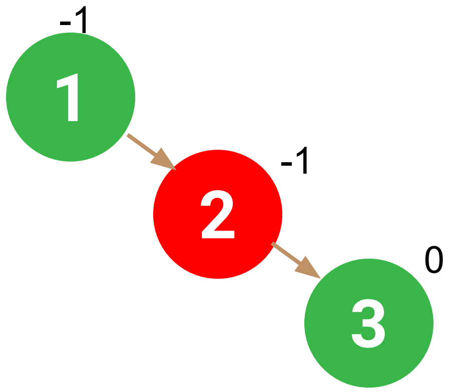


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

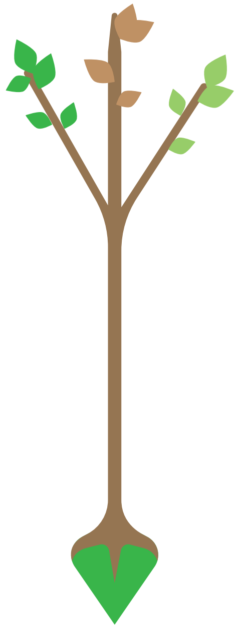


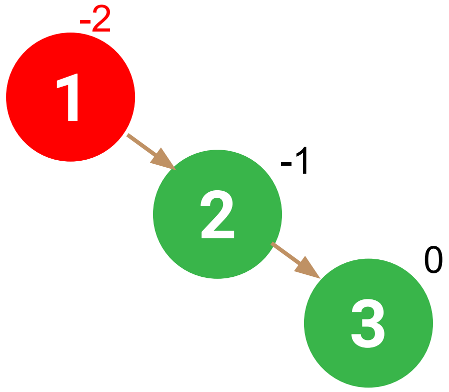


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

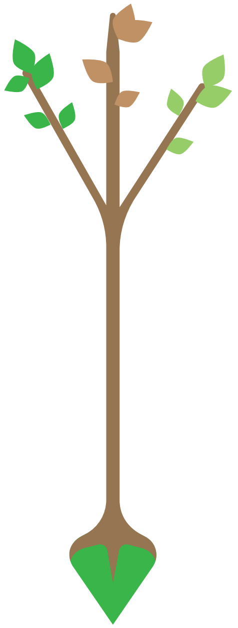


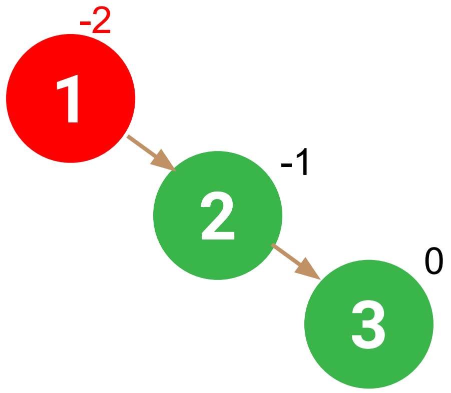


Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

INSERTION SIMULATION

AVL TREE

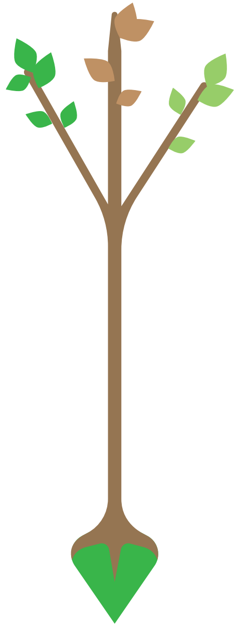


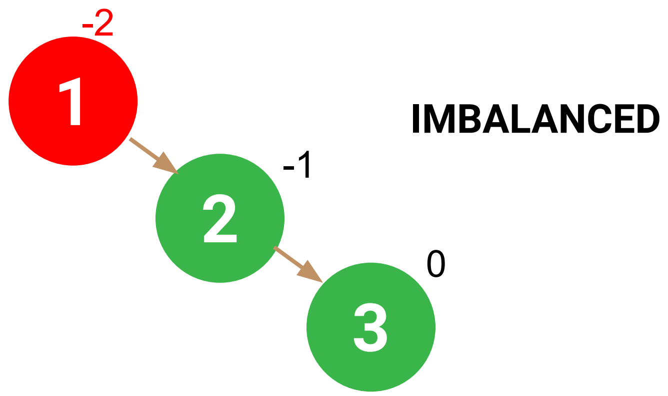


Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

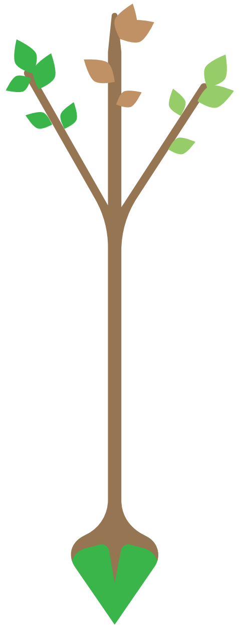


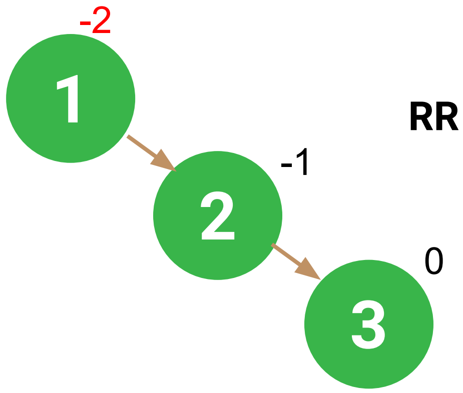


Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

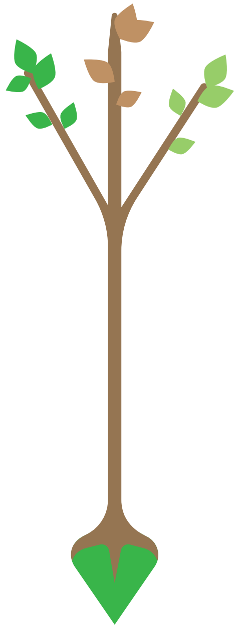




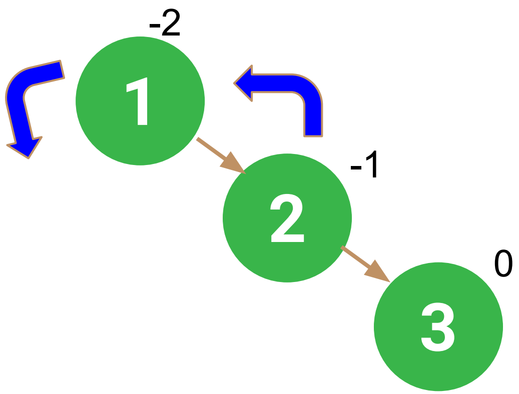
RR Rotation

Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION



AVL TREE

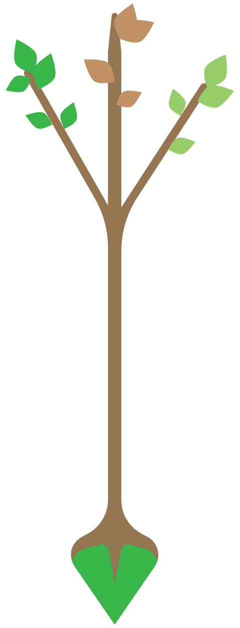


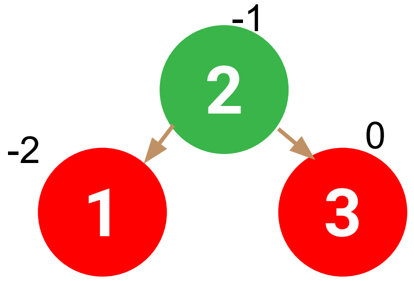
LL Rotation - BF > 1 && Left data > elem
RR Rotation - BF < -1 && right data < elem
LR Rotation - BF > 1 && Left data > elem
RL Rotation - BF < -1 && right data > elem

Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

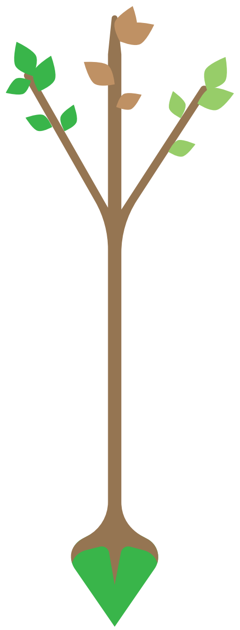


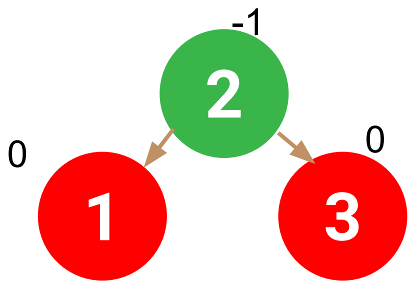


Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

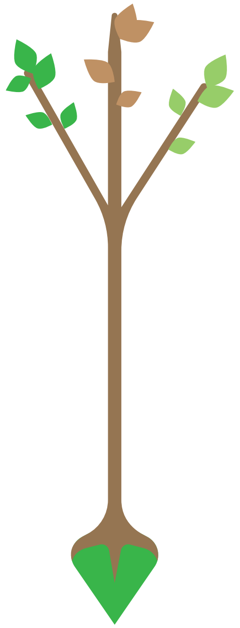


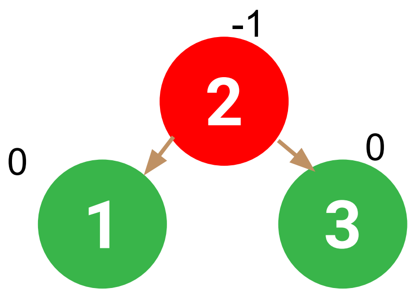


Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

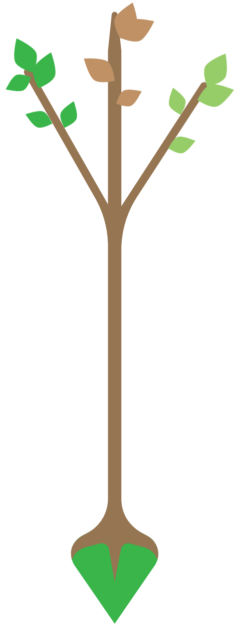


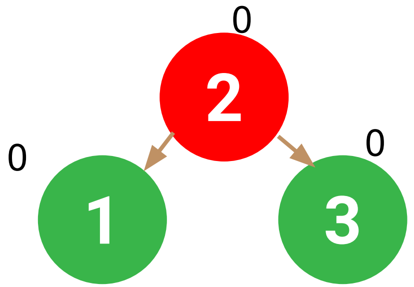


Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

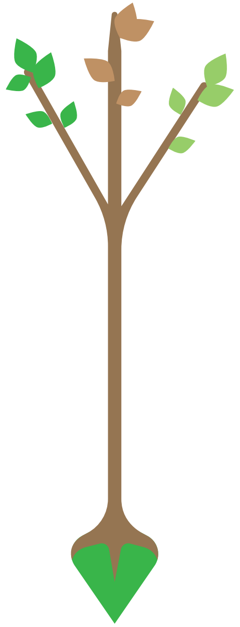


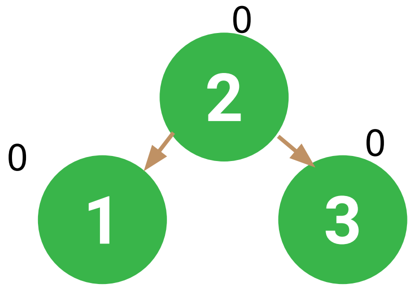


Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

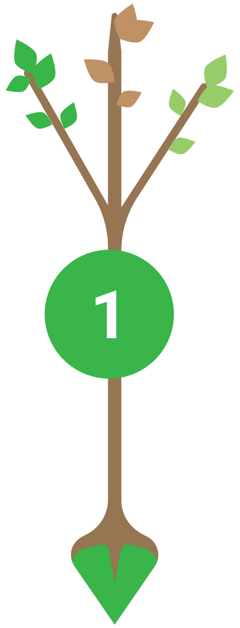


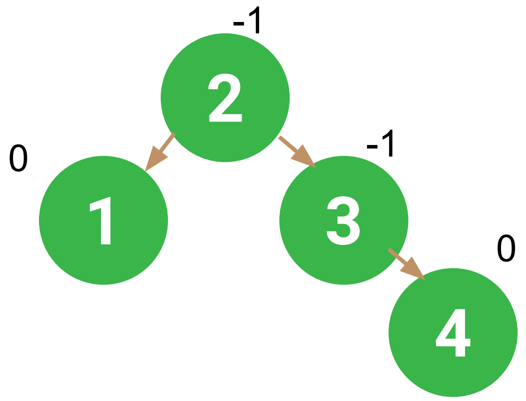


Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

DELETE SIMULATION

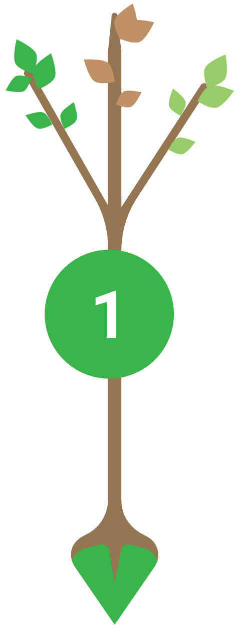
AVL TREE

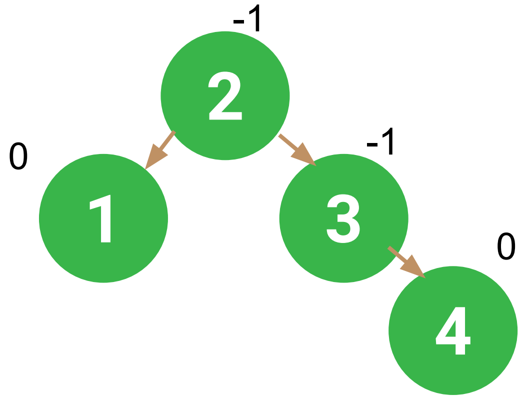




DELETE SIMULATION

AVL TREE

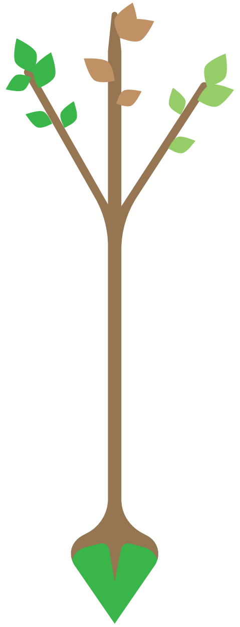


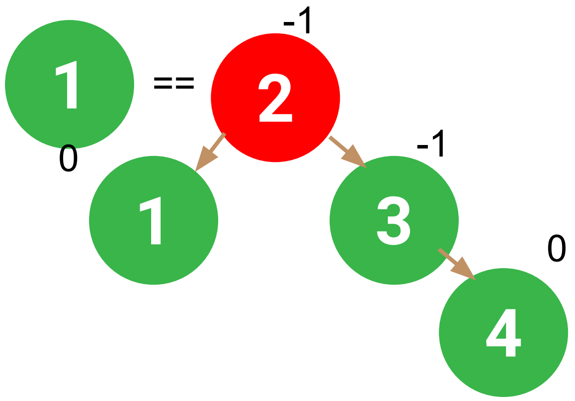


The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

DELETE SIMULATION

AVL TREE

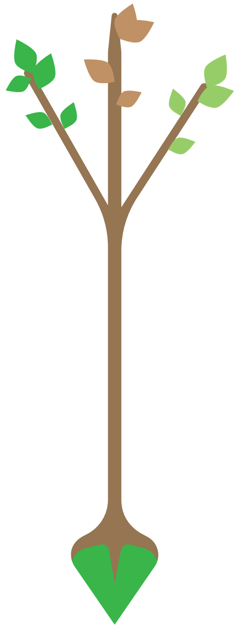


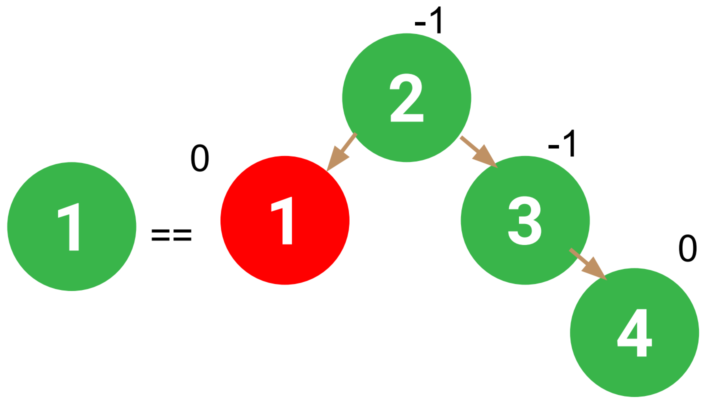


The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

DELETE SIMULATION

AVL TREE

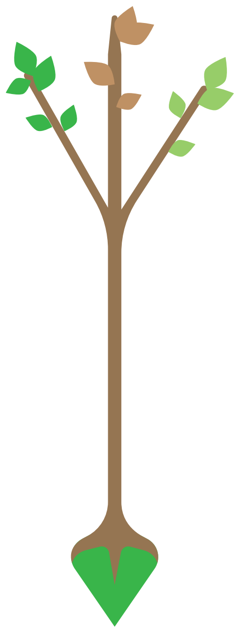


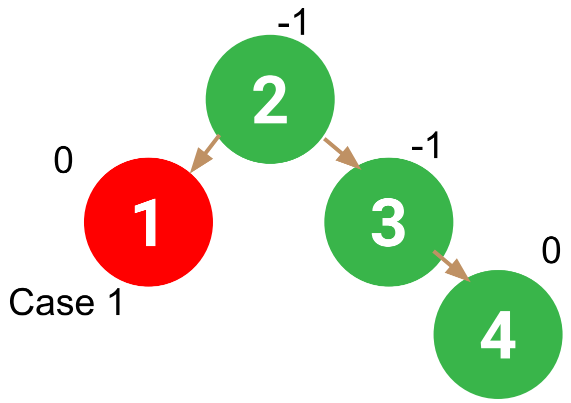


The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

DELETE SIMULATION

AVL TREE

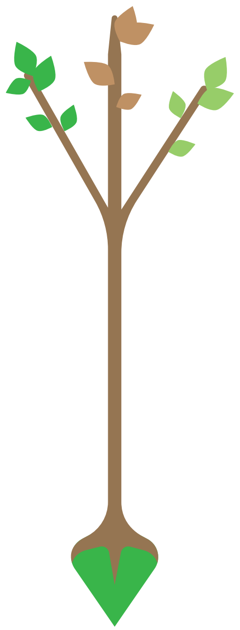


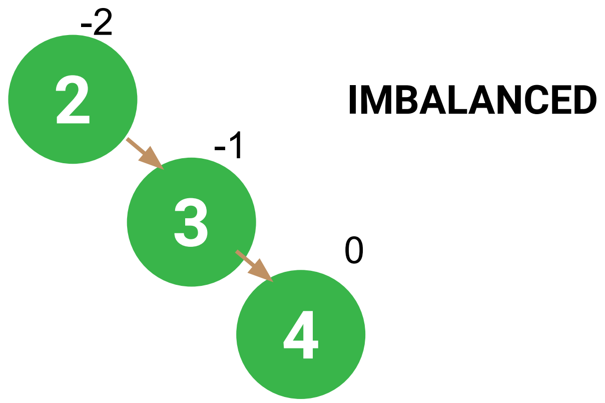


The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

DELETE SIMULATION

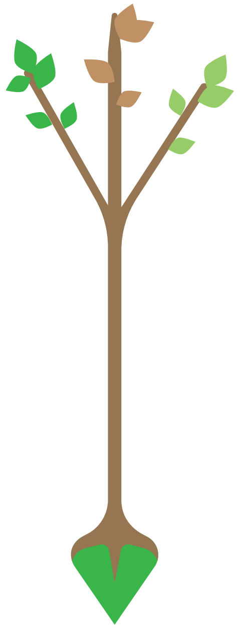
AVL TREE



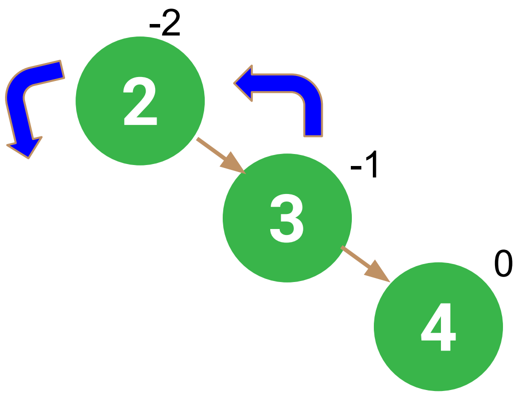


The deletion operation in AVL Tree is similar to deletion operation in BST. But after every deletion operation, we need to check with the Balance Factor condition. If the tree is balanced after deletion go for next operation otherwise perform suitable rotation to make the tree Balanced.

DELETE SIMULATION



AVL TREE



LL Rotation - BF > 1 &&
balanceFactor((*trav)->left) >= 0

RR Rotation - BF < -1 &&
balanceFactor((*trav)->right) <= 0

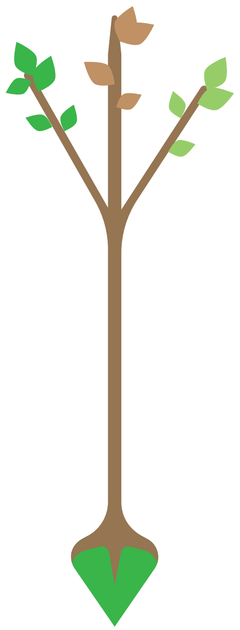
LR Rotation - BF > 1 &&
balanceFactor((*trav)->left) < 0

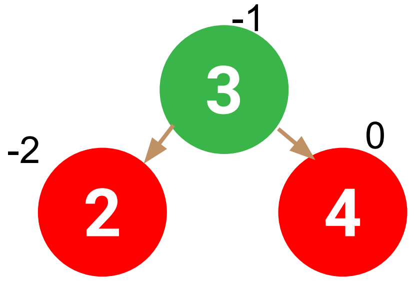
RL Rotation - BF < -1 &&
balanceFactor((*trav)->right) > 0

If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

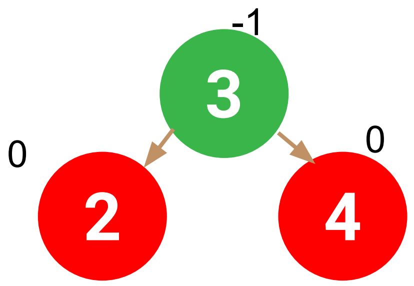


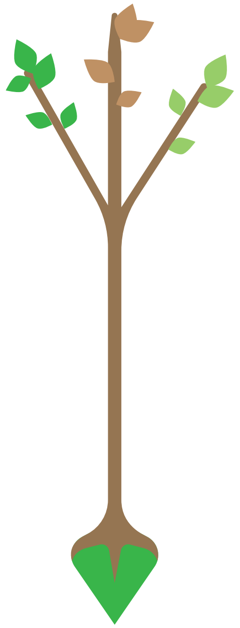


Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

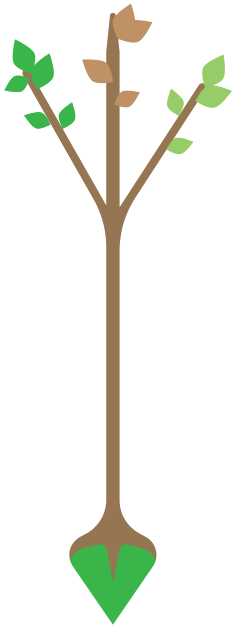


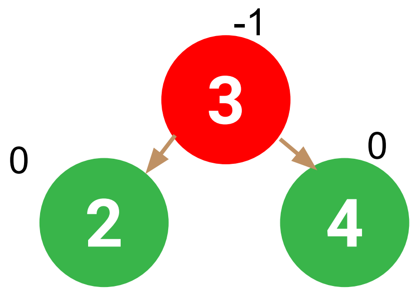


If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

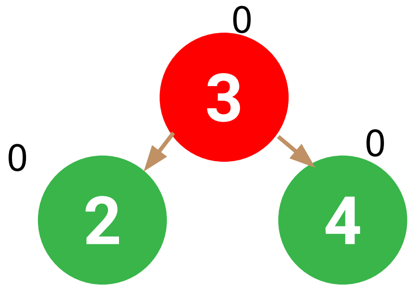


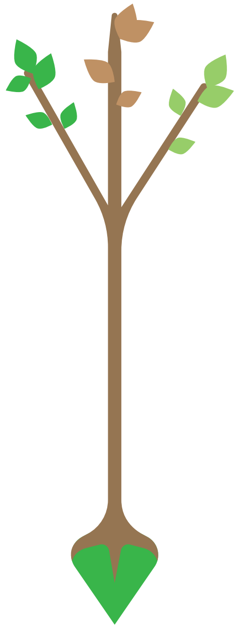


If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

INSERTION SIMULATION

AVL TREE

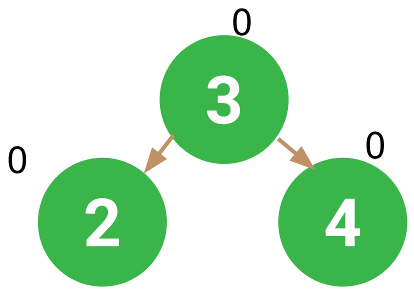


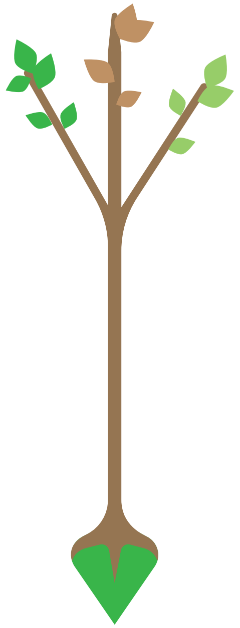


If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

DELETE SIMULATION

AVL TREE





If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

BST Insert: Internet Code vs Streamline Code

```
struct node *insert(struct node *node, int key) {
    if (node == NULL) return newNode(key);

    if (key < node->key)
        node->left = insert(node->left, key);
    else
        node->right = insert(node->right, key);

    return node;
}

struct node *newNode(int item) {
    struct node *temp = (struct node *)malloc(sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}
```

```
void insertBST(BST *B, int data){
    BST temp, *trav;

    for(trav = B; *trav != NULL && (*trav)->elem != data; ){
        trav = (data < (*trav)->elem)? &(*trav)->LC : &(*trav)->RC;
    }

    if(*trav == NULL){
        temp = (BST)calloc(1, sizeof(ctype));
        if(temp != NULL){
            temp->elem = data;
            *trav = temp;
        }
    }
}
```

BST Delete: Internet Code vs Streamline Code

```
struct node *deleteNode(struct node *root, int key) {
    if (root == NULL) return root;
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    else {
        if (root->left == NULL) {
            struct node *temp = root->right;
            free(root);
            return temp;
        } else if (root->right == NULL) {
            struct node *temp = root->left;
            free(root);
            return temp;
        }
        struct node *temp = minValueNode(root->right);
        root->key = temp->key;
        root->right = deleteNode(root->right, temp->key);
    }
    return root;
}
```

```
void deleteBST(BST *B, int data){
    BST temp, *trav;

    for(trav = B; *trav != NULL && (*trav)->elem != data; ){
        trav = (data < (*trav)->elem)? &(*trav)->LC : &(*trav)->RC;
    }

    if(*trav != NULL){
        if((*trav)->LC == NULL){
            temp = *trav;
            *trav = temp->RC;
            free(temp);
        }else if((*trav)->RC == NULL){
            temp = *trav;
            *trav = temp->LC;
            free(temp);
        }else{
            (*trav)->elem = deleteMin(&(*trav)->RC);
        }
    }
}
```

BST Delete: Internet Code vs Streamline Code

```
struct node *minValueNode(struct node *node) {
    struct node *current = node;

    while (current && current->left != NULL)
        current = current->left;

    return current;
}
```

```
struct node *maxValueNode(struct node *node) {
    struct node *current = node;

    while (current && current->right != NULL)
        current = current->right;

    return current;
}
```

```
int deleteMin(BST *B){
    BST temp, *trav;
    int min;

    for(trav = B; (*trav)->LC != NULL; trav = &(*trav)->LC){}
    temp = *trav;
    min = (*trav)->elem;
    *trav = temp->RC;
    free(temp);

    return min;
}
```

```
int deleteMax(BST *B){
    BST *trav, temp;
    int max;

    for(trav = B; (*trav)->RC != NULL; trav = &(*trav)->RC){}
    temp = *trav;
    max = (*trav)->elem;
    *trav = temp->LC;
    free(temp);

    return max;
}
```

BST Search: Internet Code vs Streamline Code

```
struct node* search(struct node* root, int key){
	if (root == NULL || root->key == key)
		return root;

	if (root->key < key)
		return search(root->right, key);

	return search(root->left, key);
}
```

```
int isMember(BST S, int data){
	BST trav;

	for(trav = S; trav != NULL && trav->elem != data; ){
		trav = (data < trav->elem)? trav->LC : trav->RC;
	}

	return (trav != NULL)? 1 : 0;
}
```

AVL Insert: Internet Code vs Streamline Code

```
struct Node *insertNode(struct Node *node, int key) {
	//Find the correct position to insert the node
	if (node == NULL)
		return (newNode(key));

	if (key < node->key)
		node->left = insertNode(node->left, key);
	else if (key > node->key)
		node->right = insertNode(node->right, key);
	else
		return node;

	 //Update the balance factor of each node
		node->height = 1 + max(height(node->left),
			height(node->right));
		int balance = getBalance(node);
```

```
void insertAVL(AVL *A, int elem) {
	AVL *trav, temp;
	int BF;
	//Find the correct position to insert the node
	for(trav = A; *trav != NULL && (*trav)->data != elem; ) {
		trav = (elem < (*trav)->data)? &(*trav)->left : &(*trav)->right;
	}

	if (*trav == NULL) {
		temp = (AVL)calloc(1,sizeof(struct node));
		if (temp != NULL) {
			temp->data = elem;
			temp->height = 0;
			*trav = temp;
		}
	}
	//Update the balance factor of each node
	(*trav)->height = max(height((*trav)->left), height((*trav)->right))+1;
	BF = balanceFactor(*trav);
```

AVL Insert: Internet Code vs Streamline Code

```
    //Balance the tree
    /*LL Rotation*/
    if (balance > 1 && key < node->left->key)
          return rightRotate(node);
      /*RR Rotation*/
    if (balance < -1 && key > node->right->key)
          return leftRotate(node);
     /*LR Rotation*/
    if (balance > 1 && key > node->left->key) {
          node->left = leftRotate(node->left);
          return rightRotate(node);
    }
     /*RL Rotation*/
    if (balance < -1 && key < node->right->key) {
          node->right = rightRotate(node->right);
          return leftRotate(node);
    }

    return node;
}
```

```
    //Balance the tree
    /*LL Rotation*/
    if (BF > 1 && (*trav)->left->data > elem){
          rightRotate(trav);
    /*RR Rotation*/
    }else if (BF < -1 && (*trav)->right->data < elem){
          leftRotate(trav);
    /*LR Rotation*/
    }else if (BF > 1 && (*trav)->left->data < elem){
          leftRotate(&(*trav)->left);
          rightRotate(trav);
    /*RL Rotation*/
    }else if (BF < -1 && (*trav)->right->data > elem){
          rightRotate(&(*trav)->right);
          leftRotate(trav);
    }

    return A;
}
```

```c
struct Node *newNode(int key) {
    struct Node *node = (struct Node *)

    malloc(sizeof(struct Node));
    node->key = key;
    node->left = NULL;
    node->right = NULL;
    node->height = 1;

    return (node);
}

int getBalance(struct Node *N) {
    if (N == NULL)
        return 0;

    return height(N->left) - height(N->right);
}

int height(struct Node *N) {
    if (N == NULL)
        return 0;

    return N->height;
}

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

```c
int balanceFactor(AVL A) {
    return (height(A->left) - height(A->right));
}

int height(AVL A) {
    return (A!=NULL)?A->height:0;
}

int max(int A, int B) {
    return(A>B)?A:B;
}
```

```
struct Node *leftRotate(struct Node *x) {
  struct Node *y = x->right;
  struct Node *T2 = y->left;

  y->left = x;
  x->right = T2;

  x->height = max(height(x->left), height(x->right)) + 1;
  y->height = max(height(y->left), height(y->right)) + 1;

  return y;
}
```

```
void leftRotate(AVL *A) {
        AVL x, y;

        x = (*A)->right;
        y = x->left;

        x->left = (*A);
        (*A)->right = y;

        (*A)->height = max(height((*A)->left), height((*A)->right))+1;
        x->height = max(height(x->left), height(x->right))+1;

        (*A) = x;
}
```

```
struct Node *rightRotate(struct Node *y) {
  struct Node *x = y->left;
  struct Node *T2 = x->right;

  x->right = y;
  y->left = T2;

  y->height = max(height(y->left), height(y->right)) + 1;
  x->height = max(height(x->left), height(x->right)) + 1;

  return x;
}
```

```
void rightRotate(AVL *A) {
        AVL x, y;

        x = (*A)->left;
        y = x->right;

        x->right = (*A);
        (*A)->left = y;

        (*A)->height = max(height((*A)->left), height((*A)->right))+1;
        x->height = max(height(x->left), height(x->right))+1;

        (*A) = x;
}
```

AVL Delete: Internet Code vs Streamline Code

```
struct Node *deleteNode(struct Node *root, int key) {
    //Find the node to be deleted
    if (root == NULL)
        return root;

    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        //Delete node
        if ((root->left == NULL) || (root->right == NULL)) {
            struct Node *temp = root->left ? root->left : root->right;

            if (temp == NULL) {
                temp = root;
                root = NULL;
            } else
                *root = *temp;
                free(temp);
        } else {
            struct Node *temp = minValueNode(root->right);
            root->key = temp->key;
            root->right = deleteNode(root->right, temp->key);
        }
    }
```

```
void deleteAVL(AVL *A, int elem) {
    AVL *trav, temp;
    int BF;

    //Find the node to be deleted
    for(trav = A; *trav != NULL && (*trav)->data != elem; ) {
        trav = (elem < (*trav)->data)? &(*trav)->left : &(*trav)->right;
    }

    //Delete node
    if(*trav != NULL){
        if((*trav)->left == NULL){
            temp = *trav;
            *trav = (*trav)->right;
            free(temp);
        }else if((*trav)->right == NULL){
            temp = *trav;
            *trav = (*trav)->left;
            free(temp);
        }else{
            (*trav)->data = deleteMin(&(*trav)->right);
        }
    }
```

AVL Delete: Internet Code vs Streamline Code

```
    //Update the balance factor of each node
    if (root == NULL)
        return root;

    root->height = 1 + max(height(root->left), height(root->right));
    int balance = getBalance(root);

    //Balance the tree
    /*LL Rotation*/
    if (balance > 1 && getBalance(root->left) >= 0)
        return rightRotate(root);
    /*RR Rotation*/
    if (balance > 1 && getBalance(root->right) <= 0) {
        return leftRotate(root);
    }
    /*LR Rotation*/
    if (balance < -1 && getBalance(root->left) < 0)
        root->left = leftRotate(root->left);
        return rightRotate(root);
    /*RL Rotation*/
    if (balance < -1 && getBalance(root->right) > 0) {
        root->right = rightRotate(root->right);
        return leftRotate(root);
    }

    return root;
}
```

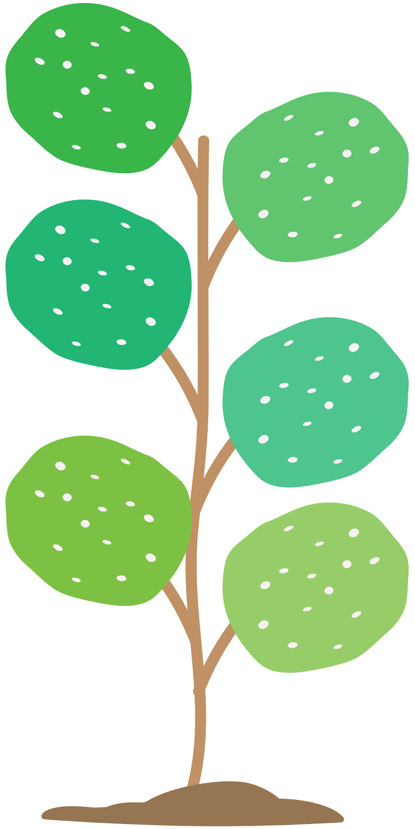
```
    //Update the balance factor of each node
    if(*trav != NULL){
        (*trav)->height = max(height((*trav)->left), height((*trav)->right))+1;
        BF = balanceFactor(*trav);

        //Balance the tree
        /*LL Rotation*/
        if (BF > 1 && balanceFactor((*trav)->left) >= 0){
            rightRotate(trav);
        /*RR Rotation*/
        }else if (BF < -1 && balanceFactor((*trav)->right) <= 0){
            leftRotate(trav);
        /*LR Rotation*/
        }else if (BF > 1 && balanceFactor((*trav)->left) < 0){
            leftRotate(&(*trav)->left);
            rightRotate(trav);
        /*RL Rotation*/
        }else if (BF < -1 && balanceFactor((*trav)->right) > 0){
            rightRotate(&(*trav)->right);
            leftRotate(trav);
        }
    }
}
```

AVL Delete: Internet Code vs Streamline Code

```
struct Node *minValueNode(struct Node *node) {
    struct Node *current = node;

    while (current->left != NULL)
        current = current->left;

    return current;
}

struct Node *maxValueNode(struct Node *node) {
    struct Node *current = node;

    while (current->right != NULL)
        current = current->right;

    return current;
}
```

```
int deleteMin(AVL *A){
        AVL *trav, temp;
        int min;

        for(trav = A; (*trav)->left != NULL; trav = &(*trav)->left){}
        temp = *trav;
        min = temp->data;
        *trav = temp->right;
        free(temp);

        return min;
}

int deleteMax(AVL *A){
        AVL *trav, temp;
        int max;

        for(trav = A; (*trav)->right != NULL; trav = &(*trav)->right){}
        temp = *trav;
        max = temp->data;
        *trav = temp->left;
        free(temp);

        return max;
}
```

References



Binary search tree(bst). Programiz. (n.d.). Retrieved October 4, 2022, from https://www.programiz.com/dsa/binary-search-tree

Binary search tree: Set 1 (search and insertion). GeeksforGeeks. (2022, August 25). Retrieved October 4, 2022, from https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

Avl Tree. Programiz. (n.d.). Retrieved October 4, 2022, from https://www.programiz.com/dsa/avl-tree

Data Structure and algorithms - AVL trees. Tutorials Point. (n.d.). Retrieved October 4, 2022, from https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm

Thank You :)