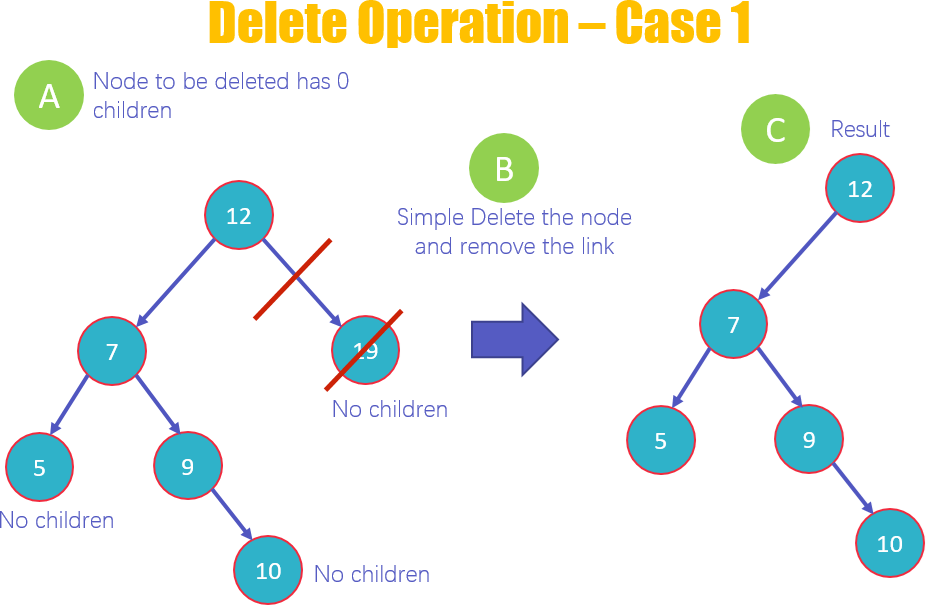
# Delete Operation

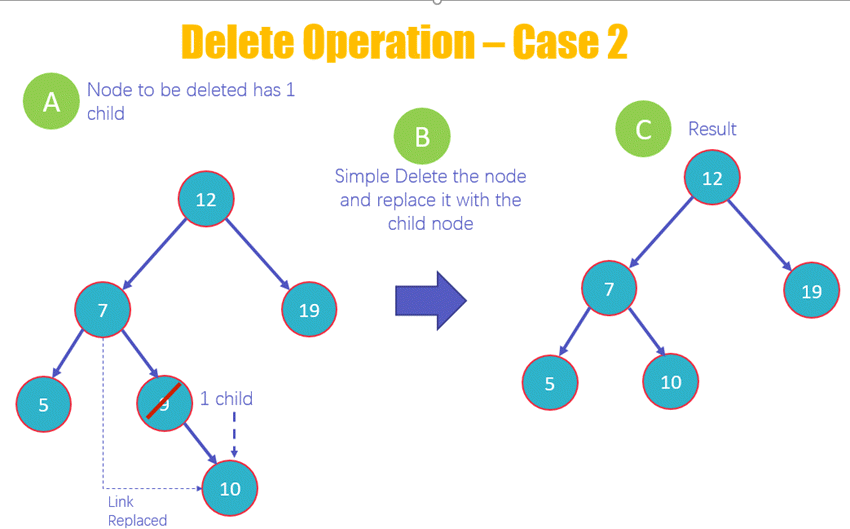
For deleting a node from a BST, there are some cases, i.e., deleting a root or deleting a leaf node. Also, after deleting a root, we need to think about the root node.

If we want to delete a leaf node, we can just delete it, but if we want to delete a root, we need to replace the root’s value with another node. Let’s take the following example:

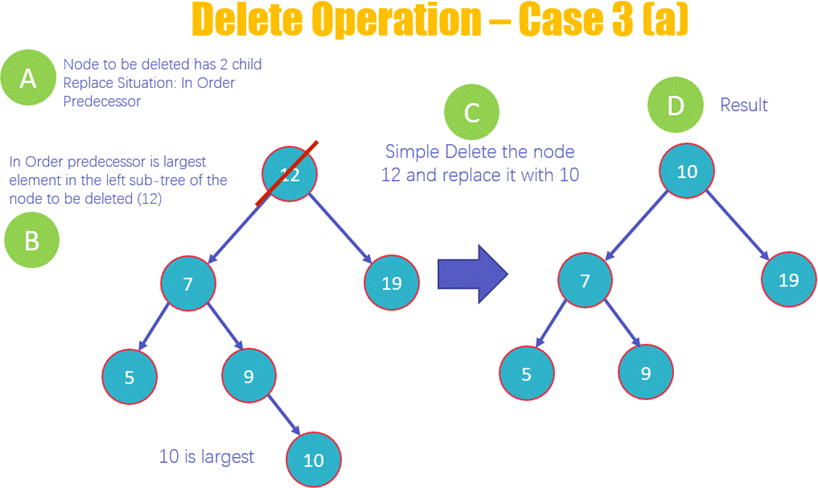
* Case 1- Node with zero children: here just need to delete the node which has no further children on the right or left.
* Case 2 – Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value.
* Case 3 Node with two children: it works on the following two rules
  + 3a – Replace the parent node with largest value in left subtree
  + 3b – Replace the parent node with smallest key in right subtree



* This is the first case of deletion in which you delete a node that has no children. As you can see in the diagram that 19, 10 and 5 have no children. But we will delete 19.
* Delete the value 19 and remove the link from the node.
* View the new structure of the BST without 19



* This is the second case of deletion in which you delete a node that has 1 child, as you can see in the diagram that 9 has one child.
* Delete the node 9 and replace it with its child 10 and add a link from 7 to 10
* View the new structure of the BST without 9



* Here you will be deleting the node 12 that has two children
* The deletion of the node will occur based upon the in-order predecessor rule, which means that the largest element on the left subtree of 12 will replace it.
* Delete the node 12 and replace it with 10 as it is the largest value on the left subtree
* View the new structure of the BST after deleting 12

## Delete Operation Case-3b

50 60

/ \ delete(50) / \ 40 70 ---------> 40 70

/ \ \

60 80 80

* Delete a node 50 that has two children
* The deletion of the node will occur based upon the In Order Successor rule, which means that the smallest element on the right subtree of 50 will replace it
* Delete the node 50 and replace it with 60 as it is the smallest value on the right subtree
* View the new structure of the BST after deleting 50

### Algorithm

Step1: if root is empty then print tree is empty

Step2: if not find the position of the data to be deleted Step3: if node with zero children just need to delete the node

Step4: Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value.

Step5: Node with two children: it works on the following two rules

* + Replace the parent node with largest value in left subtree
  + Replace the parent node with smallest key in right subtree

### Implementation

**class** BST:

**def** init (self, value): self.left = **None** self.right = **None** self.value = value

**def** insert(self, data):

**if** self.value:

**if** data < self.value:

**if** self.left **is None**: self.left = BST(data)

#### else:

self.left.insert(data)

**elif** data > self.value:

**if** self.right **is None**: self.right = BST(data)

#### else:

self.right.insert(data)

#### else:

self.value = data

**def** inorder(self):

**if** self.left: self.left.inorder()

print(self.value, end=**" "**) **if** self.right:

self.right.inorder()

**def** delete(self, data):

**if** self.value **is None**: print(**"tree is empty"**) **return**

**if** data < self.value:

**if** self.left:

self.left = self.left.delete(data)

#### else:

print(**"the given node is not present in tree"**) **elif** data > self.value:

**if** self.right:

self.right = self.right.delete(data)

#### else:

print(**"the given node is not present in tree"**)

#### else:

**if** self.left **is None**: temp = self.right self = **None return** temp

**if** self.right **is None**: temp = self.left self = **None return** temp

node = self.right

**while** node.left: node = node.left

self.value = node.value

self.right = self.right.delete(node.value)

**return** self root=BST(10) list=[6,12,1,16,98,3,7]

**for** i **in** list: root.insert(i)

root.inorder() root.delete(6) print()

print(**"After deleting the node"**) root.inorder()

### Output:

Tree elements are 1 3 6 7 10 12 16 98

After deleting the node 1 3 7 10 12 16 98

\*\* Process exited - Return Code: 0 \*\* Press Enter to exit terminal

## Time and space Complexity

The recursive function searches for the node to be deleted by dividing the subtrees at each level and moving on to the next level depending on the value of the node. In this way, the number of function calls takes the form of logN. Hence, the time complexity for search is **O(logN).**

The space complexity is **O(logN)** as well because every time the function is called, it allocates space in the stack in memory to determine the next function call.

# What is Tree Traversal?

Traversal means visiting nodes in some specific manner.

## Types of Tree Traversal

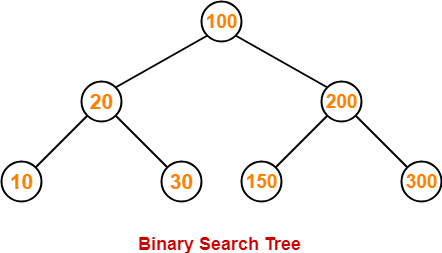
* Pre-order traversal
* Post-order traversal
* In-order traversal

1. For **In-order**, you traverse from the **left** subtree to the **root** then to the **right** subtree.
2. For **Pre-order**, you traverse from the **root** to the **left** subtree then to the **right** subtree.
3. For **Post-order**, you traverse from the **left** subtree to the **right** subtree then to the **root**.

Here is another way of representing the information above: Inorder => Left, Root, Right.

Preorder => Root, Left, Right. Post order => Left, Right, Root.

Consider the following binary search tree-



Now, let us write the traversal sequences for this binary search tree-

#### Preorder Traversal-

**inorder Traversal-**

#### Postorder Traversal-

100 20 10 30 200 150 300

10 20 30 100 150 200 300

10 30 20 150 300 200 100

Applications of Tree traversal in BST

1. In-order traversal gives nodes in non-decreasing order.
2. Post-order traversal is also useful to get the postfix expression of an expression tree
3. Pre-order traversal is also used to get prefix expression on an expression tree.

### Implementation of Tree traversal in BST Algorithm for Post-order

* 1. Start from the root node.
  2. If the root is empty, return.
  3. Traverse the left sub-tree recursively.
  4. Traverse the right sub-tree recursively.
  5. Print the root node.
  6. Stop.

### Algorithm for In-order

1. Start from the root node.
2. If the root is empty, return.
3. Traverse the left sub-tree recursively.
4. Print the root node.
5. Traverse the right sub-tree recursively.
6. Stop.

### Algorithm for Pre-order

1. Start from the root node.
2. If the root is empty, return.
3. Print the root node.
4. Traverse the left sub-tree recursively.
5. Traverse the right sub-tree recursively.
6. Stop.

Python Program to Implement Tree Traversal in BST

**class** BST:

**def** init (self, value): self.left = **None** self.right = **None** self.value = value

**def** insert(self, data):

**if** self.value:

**if** data < self.value:

**if** self.left **is None**: self.left = BST(data)

#### else:

self.left.insert(data)

**elif** data > self.value:

**if** self.right **is None**: self.right = BST(data)

#### else:

**else**:

self.right.insert(data)

self.value = data

**def** preorder(self):

print(self.value,end=**" "**) **if** self.left:

self.left.preorder()

**if** self.right:

self.right.preorder()

**def** inorder(self):

**if** self.left:

self.left.inorder() print(self.value,end=**" "**) **if** self.right:

self.right.inorder()

**def** postorder(self):

**if** self.left:

self.left.postorder()

**if** self.right:

self.right.postorder() print(self.value,end=**" "**)

root=BST(10) list=[20,4,30,15,6]

**for** i **in** list: root.insert(i)

print(**"preorder is"**) root.preorder() print() print(**"postorder is"**) root.postorder () print() print(**"inorder is "**) root.inorder()

### Output:

preorder is

10 4 6 20 15 30

postorder is

6 4 15 30 20 10

inorder is

4 6 10 15 20 30

\*\* Process exited - Return Code: 0 \*\* Press Enter to exit terminal

## Time and Space Complexity

The time complexity for tree traversal is **O(N)** because the function recursively visits all nodes of the tree. The space complexity for tree traversal is **O(N)** because the stack holds memory continuously while using a recursive function.