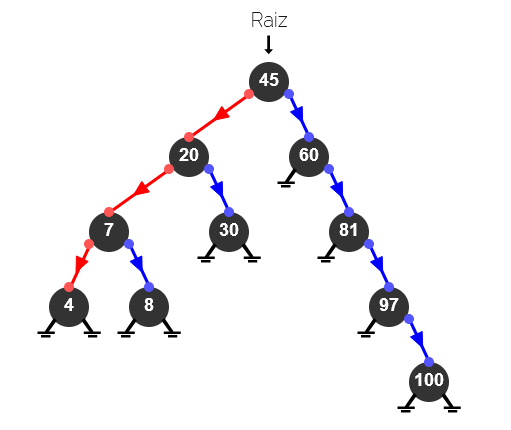
* Lista1:



# A class to store a BST node

class Node:

def \_\_init\_\_(self, data, left=None, right=None):

self.data = data

self.left = left

self.right = right

# Function to perform inorder traversal on the BST

def inorder(root):

if root is None:

return

inorder(root.left)

print(root.data, end=' ')

inorder(root.right)

# Helper function to find minimum value node in the subtree rooted at `curr`

def getMinimumKey(curr):

while curr.left:

curr = curr.left

return curr

# Recursive function to insert a key into a BST

def insert(root, key):

# if the root is None, create a new node and return it

if root is None:

return Node(key)

# if the given key is less than the root node, recur for the left subtree

if key < root.data:

root.left = insert(root.left, key)

# if the given key is more than the root node, recur for the right subtree

else:

root.right = insert(root.right, key)

return root

# Function to delete a node from a BST

def deleteNode(root, key):

# pointer to store the parent of the current node

parent = None

# start with the root node

curr = root

# search key in the BST and set its parent pointer

while curr and curr.data != key:

# update the parent to the current node

parent = curr

# if the given key is less than the current node, go to the left subtree;

# otherwise, go to the right subtree

if key < curr.data:

curr = curr.left

else:

curr = curr.right

# return if the key is not found in the tree

if curr is None:

return root

# Case 1: node to be deleted has no children, i.e., it is a leaf node

if curr.left is None and curr.right is None:

# if the node to be deleted is not a root node, then set its

# parent left/right child to None

if curr != root:

if parent.left == curr:

parent.left = None

else:

parent.right = None

# if the tree has only a root node, set it to None

else:

root = None

# Case 2: node to be deleted has two children

elif curr.left and curr.right:

# find its inorder successor node

successor = getMinimumKey(curr.right)

# store successor value

val = successor.data

# recursively delete the successor. Note that the successor

# will have at most one child (right child)

deleteNode(root, successor.data)

# copy value of the successor to the current node

curr.data = val

# Case 3: node to be deleted has only one child

else:

# choose a child node

if curr.left:

child = curr.left

else:

child = curr.right

# if the node to be deleted is not a root node, set its parent

# to its child

if curr != root:

if curr == parent.left:

parent.left = child

else:

parent.right = child

# if the node to be deleted is a root node, then set the root to the child

else:

root = child

return root

if \_\_name\_\_ == '\_\_main\_\_':

keys = [45, 20, 30, 60, 81, 97, 100, 7, 8, 4]

root = None

for key in keys:

root = insert(root, key)

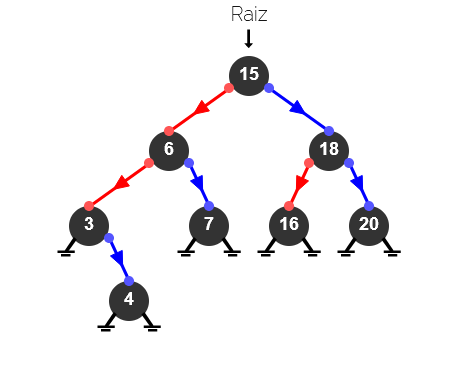
root = deleteNode(root, 7)

# inorder(root)

root = insert(root, 101)

inorder(root)

Lista2:



# A class to store a BST node

class Node:

def \_\_init\_\_(self, data, left=None, right=None):

self.data = data

self.left = left

self.right = right

# Function to perform inorder traversal on the BST

def inorder(root):

if root is None:

return

inorder(root.left)

print(root.data, end=' ')

inorder(root.right)

# Helper function to find minimum value node in the subtree rooted at `curr`

def getMinimumKey(curr):

while curr.left:

curr = curr.left

return curr

# Recursive function to insert a key into a BST

def insert(root, key):

# if the root is None, create a new node and return it

if root is None:

return Node(key)

# if the given key is less than the root node, recur for the left subtree

if key < root.data:

root.left = insert(root.left, key)

# if the given key is more than the root node, recur for the right subtree

else:

root.right = insert(root.right, key)

return root

# Function to delete a node from a BST

def deleteNode(root, key):

# pointer to store the parent of the current node

parent = None

# start with the root node

curr = root

# search key in the BST and set its parent pointer

while curr and curr.data != key:

# update the parent to the current node

parent = curr

# if the given key is less than the current node, go to the left subtree;

# otherwise, go to the right subtree

if key < curr.data:

curr = curr.left

else:

curr = curr.right

# return if the key is not found in the tree

if curr is None:

return root

# Case 1: node to be deleted has no children, i.e., it is a leaf node

if curr.left is None and curr.right is None:

# if the node to be deleted is not a root node, then set its

# parent left/right child to None

if curr != root:

if parent.left == curr:

parent.left = None

else:

parent.right = None

# if the tree has only a root node, set it to None

else:

root = None

# Case 2: node to be deleted has two children

elif curr.left and curr.right:

# find its inorder successor node

successor = getMinimumKey(curr.right)

# store successor value

val = successor.data

# recursively delete the successor. Note that the successor

# will have at most one child (right child)

deleteNode(root, successor.data)

# copy value of the successor to the current node

curr.data = val

# Case 3: node to be deleted has only one child

else:

# choose a child node

if curr.left:

child = curr.left

else:

child = curr.right

# if the node to be deleted is not a root node, set its parent

# to its child

if curr != root:

if curr == parent.left:

parent.left = child

else:

parent.right = child

# if the node to be deleted is a root node, then set the root to the child

else:

root = child

return root

if \_\_name\_\_ == '\_\_main\_\_':

keys = [15, 6, 18, 3, 7, 16, 20, 4]

root = None

for key in keys:

root = insert(root, key)

root = deleteNode(root, 3)

# inorder(root)

root = insert(root, 30)

inorder(root)