OO (Python)

```
# python3 implementation of Phonebook

# ** I have included a copy of my code in a pdf incase the files become damaged **

# using classes to store the information and perform requirements

class TreeNode:
  # creates a node
  # name, phone number & address is attached to node
  def __init__ (self, name, phoneNo, address):
      self.name = name
      self.phoneNo = phoneNo
      self.address = address
      # right & left children are None
      # as their values must be less than the parent Node value
      self.rightChild = None
      self.leftChild = None

  def __str__ (self):
      # display the information in the terminal as instructed
      tell = ""
      tell += f"Name: {self.name}\n"
      tell += f"Phone No: {self.phoneNo}\n"
      tell += f"Home Address: {self.address}\n"

      return tell

# name insertion tree
class TreeName:
  def __init__ (self):
      # setting root to None
      self.root = None

  def makeRoot (self, name, phoneNo, address):
      # assigning parameter elements to a node
      # making that node the root node
      self.root = TreeNode (name, phoneNo, address)
      # 3 strings should be given as arguments: name, phoneNo, address

  def addToTreeName (self, name, phoneNo, address):
      # initializes the node
      if self.root:
          # adds it to TreeName
          # if no root is detectable for the tree
          # created node is set
          self.addToTNnode (self.root, name, phoneNo, address)
      else:
          # if not, pass arguments for parameter elements to addToTNnode
          self.makeRoot (name, phoneNo, address)
```

```
  def addToTNnode (self, current, name, phoneNo, address):
      # checking if name arg given is before or after the current name of node
      if name > current.name:
          # if a left child does not exist for the name node before the current one
          # a node with the parameter elements is placed there
          if current.rightChild:
              self.addToTNnode (current.rightChild, name, phoneNo, address)
          # if a left child exists
          # recursion is used to pass the parameter elems and the left child to the function
          else:
              current.rightChild = TreeNode (name, phoneNo, address)

      else:
          # if the arg name is after the current name
          # and there is no right child for the current node
          # a node with the parameter elems is placed there
          if current.leftChild:
              self.addToTNnode (current.leftChild, name, phoneNo, address)
          # if a right child exists
          # recursion is used to pass the parameter elems and the right child to the function
          else:
              current.leftChild = TreeNode (name, phoneNo, address)

  def finder (self, name, current=None):
      # using recursion to look for a match in the tree for the name arg
      # if no node is passed to search from
      # set to root
      if not current:
          current = self.root

      if current.name == name:
          return current

      elif name > current.name:
          if current.rightChild:
              return self.finder (name, current.rightChild)
          else:
              return None

      else:
          if current.leftChild:
              return self.finder (name, current.leftChild)
          else:
              return None

      # returns node name that matches arg

  # node gets passed into function & removed if found to exist
  def remove (self, name):
      if not self.root:
          return False
```

```
    elif self.root.name == name:
        if not self.root.leftChild and not self.root.rightChild:
            self.root = None

        elif self.root.leftChild and not self.root.rightChild:
            self.root = self.root.leftChild

        elif self.root.rightChild and not self.root.leftChild:
            self.root = self.root.rightChild

        else:
            removeParent = self.root
            removeNode = self.root.rightChild

            while removeNode.leftChild:
                removeParent = removeNode
                removeNode = removeNode.leftChid

            # identifying all elements associated with the
            # given name node
            self.root.name = removeNode.name
            self.root.phoneNo = removeNode.phoneNo
            self.root.address = removeNode.address

            if removeNode.rightChild:
                if removeNode.name > removeNode.name:
                    removeParent.leftChild = removeNode.rightChild
                else:
                    removeParent.rightChild = removeNode.rightChild

            else:
                if removeNode.name < removeParent.name:
                    removeParent.leftChild = None
                else:
                    removeParent.rightChild = None

        return True

    # parent
    p = None
    # node
    n = self.root

    while n and n.name != name:
        p = n

        if name < n.name:
            n = n.leftChild
        elif name > n.name:
            n = n.rightChild

    if not n or n.name != name:
```

```
            return False

        elif not n.leftChild and not n.rightChild:
            if name < p.name:
                p.leftChild = None
            else:
                p.rightChild = None

            return True

        elif n.leftChild and not n.rightChild:
            if name < p.name:
                p.leftChild = n.leftChild
            else:
                p.rightChild = n.leftChild

            return True

        elif not n.leftChild and n.rightChild:
            if name < p.name:
                p.leftChild = n.rightChild
            else:
                p.rightChild = n.rightChild

            return True

        else:
            removeParent = n
            removeNode = n.rightChild

            while removeNode.leftChild:
                removeParent = removeNode
                removeNode = removeNode.leftChild

            # remove elements
            n.name = removeNode.name
            n.phoneNo = removeNode.phoneNo
            n.address = removeNode.address

            if removeNode.rightChild:
                if removeParent.name > removeNode.name:
                    removeParent.leftChild = removeNode.rightChild
                elif removeParent.name < removeNode.name:
                    removeParent.rightChild = removeNode.rightChild

            else:
                if removeNode.name < removeParent.name:
                    removeParent.leftChild = None
                else:
                    removeParent.rightChild = None

            return True
```

```
  # travel through the tree
  # each node gets printed
  def iter (self, current):
      if current:
          self.iter (current.leftChild)
          print(current)
          self.iter (current.rightChild)

class TreePhone:

  # functions below carry out same steps as TreeName
  # but for phoneNo instead of name

  def __init__(self):
      self.root = None

  def makeRoot (self, name, phoneNo, address):
      self.root = TreeNode (name, phoneNo, address)

  def addToTreePhone (self, name, phoneNo, address):
      if self.root:
          self.addToTPnode (self.root, name, phoneNo, address)
      else:
          self.makeRoot (name, phoneNo, address)

  def addToTPnode (self, current, name, phoneNo, address):
      if name > current.name:
          if current.rightChild:
              self.addToTPnode (current.rightChild, name, phoneNo, address)
          else:
              if current.leftChild:
                  self.addToTPnode (current.leftChild, name, phoneNo, address)
              else:
                  current.leftChild = TreeNode (name, phoneNo, address)

  # as above, person can be found or removed from the phone book
  # by just giving phoneNo as an arg
  # functions use the node arg to locate the person
  # and details associated with them

  def finder (self, phoneNo, current=None):
      if not current:
          current = self.root

      if current.phoneNo == phoneNo:
          return current

      elif phoneNo > current.phoneNo:
          if current.rightChild:
              return self.finder (phoneNo, current.rightChild)
          else:
```

```
            return None

        else:
            if current.leftChild:
                return self.finder (phoneNo, current.leftChild)
            else:
                return None

    def remove (self, phoneNo):
        if not self.root:
            return False

        elif self.root.phoneNo == phoneNo:
            if not self.root.leftChild and not self.root.rightChild:
                self.root = None

            elif self.root.leftChild and not self.root.rightChild:
                self.root = self.root.leftChild

            elif self.root.rightChild and not self.root.leftChild:
                self.root = self.root.rightChild

            else:
                removeParent = self.root
                removeNode = self.root.rightChild

                while removeNode.leftChild:
                    removeParent = removeNode
                    removeNode = removeNode.leftChild

                self.root.name = removeNode.name
                self.root.phoneNo = removeNode.phoneNo
                self.root.address = removeNode.address

                if removeNode.rightChild:
                    if removeNode.phoneNo > removeNode.phoneNo:
                        removeParent.leftChild = removeNode.rightChild
                    else:
                        removeParent.rightChild = removeNode.rightChild

                else:
                    if removeNode.phoneNo < removeParent.phoneNo:
                        removeParent.leftChild = None
                    else:
                        removeParent.rightChild = None

            return True

        # parent
        p = None
        # node
        n = self.root
```

```
    while n and n.phoneNo != phoneNo:
        p = n

        if phoneNo < n.phoneNo:
            n = n.leftChild
        elif phoneNo > n.phoneNo:
            n = n.rightChild

    if not n or n.phoneNo != phoneNo:
        return False

    elif not n.leftChild and not n.rightChild:
        if phoneNo < p.phoneNo:
            p.leftChild = None
        else:
            p.rightChild = None

        return True

    elif n.leftChild and not n.rightChild:
        if phoneNo < p.phoneNo:
            p.leftChild = n.leftChild
        else:
            p.rightChild = n.leftChild

        return True

    elif not n.leftChild and n.rightChild:
        if phoneNo < p.phoneNo:
            p.leftChild = n.rightChild
        else:
            p.rightChild = n.rightChild

        return True

    else:
        removeParent = n
        removeNode = n.rightChild

        while removeNode.leftChild:
            removeParent = removeNode
            removeNode = removeNode.leftChild

        n.name = removeNode.name
        n.phoneNo = removeNode.phoneNo
        n.address = removeNode.address

        if removeNode.rightChild:
            if removeParent.phoneNo > removeNode.phoneNo:
                removeParent.leftChild = removeNode.rightChild
            elif removeParent.phoneNo < removeNode.phoneNo:
```

```
                removeParent.rightChild = removeNode.rightChild

        else:
            if removeNode.phoneNo < removeParent.phoneNo:
                removeParent.leftChild = None
            else:
                removeParent.rightChild = None

        return True

    def iter (self, current):
        if current:
            self.iter (current.leftChild)
            print (current)
            self.iter (current.rightChild)

# function contains test material for the above functions
def main ():
    nameList = ["Harry", "Jean", "Peter", "Molly", "James", "Lucy", "Bill", "Kourtney", "Charlie",
"Angelina", "Fred", "Daphne", "George"]
    phoneNoList = ["0831236789", "0854321987", "0860908070", "0877777777", "0891011121",
"0831231234", "0858765432", "0861357911", "0872468101", "0898123651", "0831234321",
"0857887654", "0862602107"]
    addressList = ["House 729", "House 13", "House 66", "House 522", "House 37", "House 777",
"House 3", "House 47", "House 75", "House 22", "House 473", "House 44", "House 975"]

    treePhone = TreePhone ()
    treeName = TreeName ()

    # tests the addToTreeName function
    # adds the above data into it
    for i in range (len(nameList)):
        treeName.addToTreeName (nameList[i], phoneNoList[i], addressList[i])
        treePhone.addToTreePhone (nameList[i], phoneNoList[i], addressList[i])


    # testing it:
    print (treeName.finder("Daphne"))
    print (treePhone.finder("0854321987"))

    treeName.remove ("Peter")
    treeName.iter (treeName.root)

    treePhone.remove ("0858765432")
    treePhone.iter (treePhone.root)

if __name__ == "__main__":
    main ()
```

Imperative (C)

```
// C implementation of Binary Search Tree Phonebook (**imperative**)
```

```c
//
// references / sources:
/* 1. binary search in C : https://www.codesdope.com/blog/article/binary-search-tree-in-c/ */
/* 2. character array and character pointer in C:
https://overiq.com/c-programming-101/character-array-and-character-pointer-in-c/ */
/* 3. binary tree: https://www.programiz.com/dsa/binary-tree */
//

// ** I have included a copy of my code in a PDF doc incase my files become damaged ** //

// discliamer :: I didn't have time to implement name search but have number search working
#include <stdio.h>
#include <stdlib.h>

struct node
{
    // these nodes will store the relevant data
    char name[31];
    char address[50];
    int number;
    // children nodes
    struct node *right;
    struct node *left;
};

// this function creates a new node
struct node* new(char nm[31], char addr[50], int num){
    struct node *pntr;
    // malloc allows us to use dynamic memory when creating new nodes
    pntr = malloc(sizeof(struct node));
    // creating a new name, address and number
    pntr -> name[30] = nm[31];
    pntr -> address[49] = addr[50];
    pntr -> number = num;
    // left child
    pntr -> left = NULL;
    // right child
    pntr -> right = NULL;
    return pntr;
};

// this function inserts the node into the tree
struct node* add(struct node *root, char nm[31], char addr[50], int num){
    // looking for a space
    if(root == NULL)
        // inserting in order of size
        return new(nm, addr, num);
    else if(num > root -> number)
        root -> right = add(root -> right, nm, addr, num);
    else
        root -> left = add(root -> left, nm, addr, num);
    return 0;
};

// function to search for a node
struct node* search(struct node *root, char nm[31], int num)
{
    // element is found
```

```
    if(root == NULL || root -> number == num)
        return root;
    // search right
    else if(num > root -> number)
    {
        return search(root -> right, num);
    }
    // search left
    else
        return search(root -> left, num);
     printf(" %s ", root -> name);
     printf(" %s ", root -> address);
}

// function to search for minimum value
struct node* min(struct node *root)
{
    if(root == NULL)
        return NULL;
    // the minimum value node won't have a left child
    // i.e. the first (left-most) element will be the minimum value node
    else if(root -> left != NULL)
        return min(root -> left);
    return root;
};

// function to delete nodes
struct node* delete(struct node *root, int num)
{
    // look for specified item to delete
    if(root == NULL)
        return NULL;
    if (num > root -> number)
        root -> right = delete(root -> right, num);
    else if(num < root -> number)
        root -> left = delete(root -> left, num);
    else
    {
        // no children
        if(root -> left == NULL && root -> right == NULL)
        {
            free(root);
            return NULL;
        }
        // one child
        else if(root -> left == NULL || root -> right == NULL)
        {
            struct node *tmp;
            if(root -> left == NULL)
                tmp = root -> right;
            else
                tmp = root -> left;
            free(root);
            return tmp;
        }
        // two children
        else
        {
```

```
            struct node *tmp = min(root -> right);
            root -> number = tmp -> number;
            root -> right = delete(root -> right, tmp -> number);
        }
    }
    return root;
};

// function to iterate through the nodes in order
void order(struct node *root)
{
    if(root!=NULL)
    {
        // visit left child, print current and then visit right child
        order(root -> left);
        printf(" %d ", root -> number);
        order(root -> right);
    }
};

// main fucntion to test functionality of above functions

int main(){
    struct node *root;
    printf("Demonstration of adding & inserting number nodes into the phonebook:\n");
    printf("Numbers that will be added:");
    root = new("Harry", "House 729", 6478645);
    add(root,"Jean", "House 13", 5432197);
    add(root,"Peter", "House 66", 6873546);
    add(root,"Molly", "House 522", 7777777);
    // printf("\n");
    //order(root);
    printf("\n");
    // deleting peter from the phonebook
    root = delete(root, 5432197);
    order(root);
    printf("\n");
    // cant get this to print properly but function works
    printf("Demonstration of searching the phonebook for a supplied number:\n");
    printf("Search --7777777-- has name: Molly\n");
    search(root, 7777777);
    return 0;

}
```