**Exercise - 2**

**B plus TREES**

**Aim:** To write a Python code to implement insertion, deletion, and search operations of B plus trees.

**Algorithm:**

**Insertion:**

**Step 1:** Insert the new node as a leaf node.

**Step 2:** If the leaf doesn't have required space, split the node, and copy the middle node to the next index node.

**Step 3:** If the index node doesn't have required space, split the node, and copy the middle element to the next index page.

**Deletion:**

**Step 1:** Delete the key and data from the leaves.

**Step 2:** if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

**Step 3:** if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

**Program code:**

class BPlusTree:

    def \_\_init\_\_(self, order):

        self.order = order

        self.root = None

    def insert(self, key, value):

        if self.root is None:

            self.root = BPlusTreeNode(self.order, is\_leaf=True)

        if self.root.is\_full():

            old\_root = self.root

            self.root = BPlusTreeNode(self.order)

            self.root.children.append(old\_root)

            self.root.split\_child(0)

        self.root.insert\_non\_full(key, value)

    def delete(self, key):

        if self.root is None:

            return

        self.root.delete(key)

        if len(self.root.keys) == 0 and not self.root.is\_leaf:

            self.root = self.root.children[0]

    def print\_tree(self):

        self.root.print\_node()

    def search(self, key):

        if self.root is None:

            print("Tree is empty.")

            return

        result = self.root.search(key)

        if result:

            print(f"Key {key} found:")

        else:

            print(f"Key {key} not found.")

class BPlusTreeNode:

    def \_\_init\_\_(self, order, is\_leaf=False):

        self.order = order

        self.keys = []

        self.values = []

        self.children = []

        self.is\_leaf = is\_leaf

    def is\_full(self):

        return len(self.keys) == self.order - 1

    def insert\_non\_full(self, key, value):

        if self.is\_leaf:

            self.insert\_into\_leaf(key, value)

        else:

            index = self.find\_child\_index(key)

            if self.children[index].is\_full():

                self.split\_child(index)

                if key > self.keys[index]:

                    index += 1

            self.children[index].insert\_non\_full(key, value)

    def insert\_into\_leaf(self, key, value):

        index = 0

        while index < len(self.keys) and key > self.keys[index]:

            index += 1

        self.keys.insert(index, key)

        self.values.insert(index, value)

    def split\_child(self, child\_index):

        child = self.children[child\_index]

        new\_child = BPlusTreeNode(self.order, is\_leaf=child.is\_leaf)

        mid\_index = self.order // 2

        self.keys.insert(child\_index, child.keys[mid\_index])

        self.values.insert(child\_index, child.values[mid\_index])

        self.children.insert(child\_index + 1, new\_child)

        new\_child.keys = child.keys[mid\_index:]

        new\_child.values = child.values[mid\_index:]

        child.keys = child.keys[:mid\_index]

        child.values = child.values[:mid\_index]

    def find\_child\_index(self, key):

        index = 0

        while index < len(self.keys) and key > self.keys[index]:

            index += 1

        return index

    def delete(self, key):

        if self.is\_leaf:

            self.delete\_from\_leaf(key)

        else:

            index = self.find\_child\_index(key)

            if index < len(self.keys) and key == self.keys[index]:

                predecessor = self.children[index].get\_max\_key()

                self.keys[index] = predecessor

                self.children[index].delete(predecessor)

            else:

                self.children[index].delete(key)

            if len(self.children[index].keys) < (self.order - 1) // 2:

                self.fix\_child\_underflow(index)

    def delete\_from\_leaf(self, key):

        index = 0

        while index < len(self.keys) and key > self.keys[index]:

            index += 1

        if index < len(self.keys) and key == self.keys[index]:

            self.keys.pop(index)

            self.values.pop(index)

    def get\_max\_key(self):

        if self.is\_leaf:

            return self.keys[-1]

        else:

            return self.children[-1].get\_max\_key()

    def fix\_child\_underflow(self, child\_index):

        if child\_index > 0 and len(self.children[child\_index - 1].keys) > (self.order - 1) // 2:

            self.borrow\_from\_previous(child\_index)

        elif child\_index < len(self.children) - 1 and len(self.children[child\_index + 1].keys) > (self.order - 1) // 2:

            self.borrow\_from\_next(child\_index)

        elif child\_index > 0:

            self.merge\_with\_previous(child\_index)

        else:

            self.merge\_with\_next(child\_index)

    def borrow\_from\_previous(self, child\_index):

        child = self.children[child\_index]

        sibling = self.children[child\_index - 1]

        child.keys.insert(0, self.keys[child\_index - 1])

        child.values.insert(0, sibling.values.pop())

        self.keys[child\_index - 1] = sibling.keys.pop()

        if not child.is\_leaf:

            child.children.insert(0, sibling.children.pop())

    def borrow\_from\_next(self, child\_index):

        child = self.children[child\_index]

        sibling = self.children[child\_index + 1]

        child.keys.append(self.keys[child\_index])

        child.values.append(sibling.values.pop(0))

        self.keys[child\_index] = sibling.keys.pop(0)

        if not child.is\_leaf:

            child.children.append(sibling.children.pop(0))

    def merge\_with\_previous(self, child\_index):

        child = self.children[child\_index]

        sibling = self.children[child\_index - 1]

        sibling.keys.append(self.keys[child\_index - 1])

        sibling.values.extend(child.values)

        self.keys.pop(child\_index - 1)

        self.children.pop(child\_index)

        if not child.is\_leaf:

            sibling.children.extend(child.children)

    def merge\_with\_next(self, child\_index):

        child = self.children[child\_index]

        sibling = self.children[child\_index + 1]

        child.keys.append(self.keys[child\_index])

        child.values.extend(sibling.values)

        self.keys.pop(child\_index)

        self.children.pop(child\_index + 1)

        if not child.is\_leaf:

            child.children.extend(sibling.children)

    def print\_node(self, level=0):

        if self.is\_leaf:

            print(f"Level {level}: {self.keys}")

        else:

            print(f"Level {level}: {self.keys}")

            for child in self.children:

                child.print\_node(level + 1)

    def search(self, key):

        index = self.find\_child\_index(key)

        if index < len(self.keys) and key == self.keys[index]:

            return self.values[index]

        elif self.is\_leaf:

            return None

        else:

            return self.children[index].search(key)

#Driver Code

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

    order=int(input("Enter the order of b plus tree:"))

    b\_plus\_tree = BPlusTree(order)

    while True:

        ch=int(input("[1:Insert,2:Delete,3:Display,4:Exit,5:Search]\nEnter the choice:"))

        if ch == 1:

            n=int(input("Enter number of values to be inserted:"))

            for \_ in range(n):

                value=int(input("Enter the value:"))

                b\_plus\_tree.insert(value, f'Value {value}')

            print(f"After inserting:")

            b\_plus\_tree.print\_tree()

            print()

        elif ch == 2 :

            value=int(input("Enter value to be deleted:"))

            b\_plus\_tree.delete(value)

            print(f"After deleting:")

            b\_plus\_tree.print\_tree()

            print()

        elif ch == 3:

            print("Final Tree Structure:")

            b\_plus\_tree.print\_tree()

        elif ch == 4:

            print("Exiting..")

            break

        elif ch == 5:

            value=int(input("Enter key to be serched:"))

            b\_plus\_tree.search(value)

        else:

            print("Invalid choice")

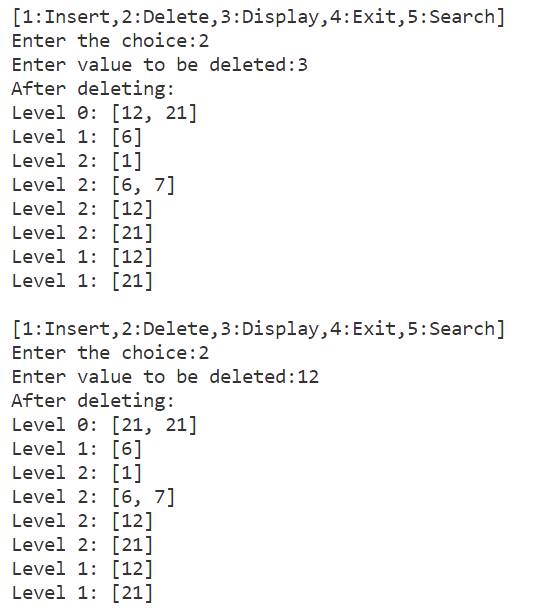
**Sample Input and Output:**

**Insertion:**

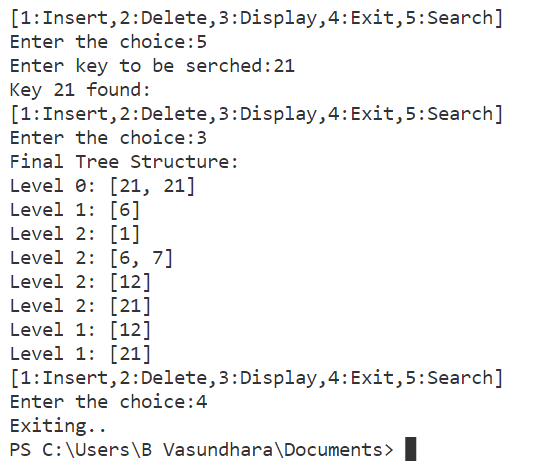
**A screenshot of a computer program

Description automatically generated with medium confidence**

**Deletion:**

****

**Searching and Display:**

****

**Result:**

Thus, the deletion, insertion and searching operations of B plus trees has been successfully implemented using Python code and the output is verified.