**UIT 2402 – ADVANCED DATA STRUCTURES AND ALGORITHM ANALYSIS**

**EX 2: B plus TREES**

**Name:** B.VASUNDHARA

**Roll no:** 3122 21 5002 119

B+ Tree is an extension of B Tree which allows efficient insertion, deletion, and search operations.

The leaf nodes of a B+ tree is linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree is used to store the large amount of data which cannot be stored in the main memory. Since, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

The internal nodes of B+ tree is often called index nodes.

**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:**

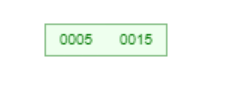
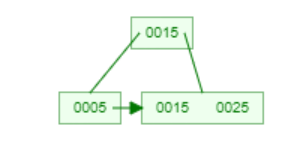
**Step 1:** Delete the key and data from the leaves.

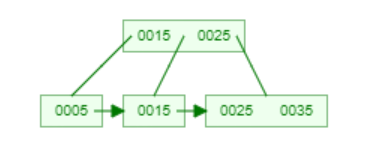
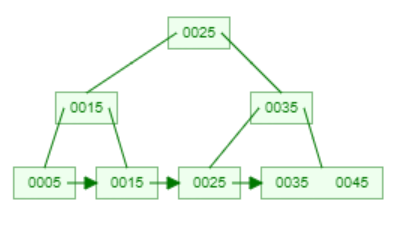
**Step 2:** if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

**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.

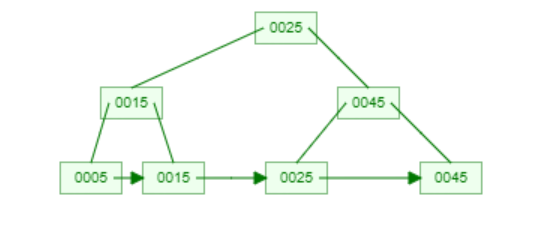
**Illustrations:**

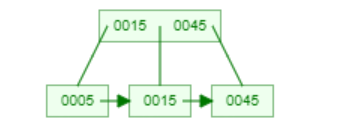
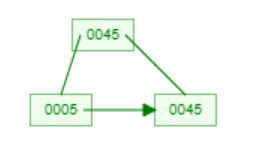
Insertion:

Deletion:



**Program code:**

import math

class Node:

    def \_\_init\_\_(self, order):

        self.order = order

        self.values = []

        self.keys = []

        self.nextKey = None

        self.parent = None

        self.check\_leaf = False

    def insert\_at\_leaf(self, leaf, value, key):

        if (self.values):

            temp1 = self.values

            for i in range(len(temp1)):

                if (value == temp1[i]):

                    self.keys[i].append(key)

                    break

                elif (value < temp1[i]):

                    self.values = self.values[:i] + [value] + self.values[i:]

                    self.keys = self.keys[:i] + [[key]] + self.keys[i:]

                    break

                elif (i + 1 == len(temp1)):

                    self.values.append(value)

                    self.keys.append([key])

                    break

        else:

            self.values = [value]

            self.keys = [[key]]

class BplusTree:

    def \_\_init\_\_(self, order):

        self.root = Node(order)

        self.root.check\_leaf = True

    def insert(self, value, key):

        value = str(value)

        old\_node = self.search(value)

        old\_node.insert\_at\_leaf(old\_node, value, key)

        if (len(old\_node.values) == old\_node.order):

            node1 = Node(old\_node.order)

            node1.check\_leaf = True

            node1.parent = old\_node.parent

            mid = int(math.ceil(old\_node.order / 2)) - 1

            node1.values = old\_node.values[mid + 1:]

            node1.keys = old\_node.keys[mid + 1:]

            node1.nextKey = old\_node.nextKey

            old\_node.values = old\_node.values[:mid + 1]

            old\_node.keys = old\_node.keys[:mid + 1]

            old\_node.nextKey = node1

            self.insert\_in\_parent(old\_node, node1.values[0], node1)

    def search(self, value):

        current\_node = self.root

        while(current\_node.check\_leaf == False):

            temp2 = current\_node.values

            for i in range(len(temp2)):

                if (value == temp2[i]):

                    current\_node = current\_node.keys[i + 1]

                    break

                elif (value < temp2[i]):

                    current\_node = current\_node.keys[i]

                    break

                elif (i + 1 == len(current\_node.values)):

                    current\_node = current\_node.keys[i + 1]

                    break

        return current\_node

    def find(self, value, key):

        l = self.search(value)

        for i, item in enumerate(l.values):

            if item == value:

                if key in l.keys[i]:

                    return True

                else:

                    return False

        return False

    def insert\_in\_parent(self, n, value, ndash):

        if (self.root == n):

            rootNode = Node(n.order)

            rootNode.values = [value]

            rootNode.keys = [n, ndash]

            self.root = rootNode

            n.parent = rootNode

            ndash.parent = rootNode

            return

        parentNode = n.parent

        temp3 = parentNode.keys

        for i in range(len(temp3)):

            if (temp3[i] == n):

                parentNode.values = parentNode.values[:i] + \

                    [value] + parentNode.values[i:]

                parentNode.keys = parentNode.keys[:i +

                                                  1] + [ndash] + parentNode.keys[i + 1:]

                if (len(parentNode.keys) > parentNode.order):

                    parentdash = Node(parentNode.order)

                    parentdash.parent = parentNode.parent

                    mid = int(math.ceil(parentNode.order / 2)) - 1

                    parentdash.values = parentNode.values[mid + 1:]

                    parentdash.keys = parentNode.keys[mid + 1:]

                    value\_ = parentNode.values[mid]

                    if (mid == 0):

                        parentNode.values = parentNode.values[:mid + 1]

                    else:

                        parentNode.values = parentNode.values[:mid]

                    parentNode.keys = parentNode.keys[:mid + 1]

                    for j in parentNode.keys:

                        j.parent = parentNode

                    for j in parentdash.keys:

                        j.parent = parentdash

                    self.insert\_in\_parent(parentNode, value\_, parentdash)

    def delete(self, value, key):

        node\_ = self.search(value)

        temp = 0

        for i, item in enumerate(node\_.values):

            if item == value:

                temp = 1

                if key in node\_.keys[i]:

                    if len(node\_.keys[i]) > 1:

                        node\_.keys[i].pop(node\_.keys[i].index(key))

                    elif node\_ == self.root:

                        node\_.values.pop(i)

                        node\_.keys.pop(i)

                    else:

                        node\_.keys[i].pop(node\_.keys[i].index(key))

                        del node\_.keys[i]

                        node\_.values.pop(node\_.values.index(value))

                        self.deleteEntry(node\_, value, key)

                else:

                    print("Value not in Key")

                    return

        if temp == 0:

            print("Value not in Tree")

            return

    def deleteEntry(self, node\_, value, key):

        if not node\_.check\_leaf:

            for i, item in enumerate(node\_.keys):

                if item == key:

                    node\_.keys.pop(i)

                    break

            for i, item in enumerate(node\_.values):

                if item == value:

                    node\_.values.pop(i)

                    break

        if self.root == node\_ and len(node\_.keys) == 1:

            self.root = node\_.keys[0]

            node\_.keys[0].parent = None

            del node\_

            return

        elif (len(node\_.keys) < int(math.ceil(node\_.order / 2)) and node\_.check\_leaf == False) or (len(node\_.values) < int(math.ceil((node\_.order - 1) / 2)) and node\_.check\_leaf == True):

            is\_predecessor = 0

            parentNode = node\_.parent

            PrevNode = -1

            NextNode = -1

            PrevK = -1

            PostK = -1

            for i, item in enumerate(parentNode.keys):

                if item == node\_:

                    if i > 0:

                        PrevNode = parentNode.keys[i - 1]

                        PrevK = parentNode.values[i - 1]

                    if i < len(parentNode.keys) - 1:

                        NextNode = parentNode.keys[i + 1]

                        PostK = parentNode.values[i]

            if PrevNode == -1:

                ndash = NextNode

                value\_ = PostK

            elif NextNode == -1:

                is\_predecessor = 1

                ndash = PrevNode

                value\_ = PrevK

            else:

                if len(node\_.values) + len(NextNode.values) < node\_.order:

                    ndash = NextNode

                    value\_ = PostK

                else:

                    is\_predecessor = 1

                    ndash = PrevNode

                    value\_ = PrevK

            if len(node\_.values) + len(ndash.values) < node\_.order:

                if is\_predecessor == 0:

                    node\_, ndash = ndash, node\_

                ndash.keys += node\_.keys

                if not node\_.check\_leaf:

                    ndash.values.append(value\_)

                else:

                    ndash.nextKey = node\_.nextKey

                ndash.values += node\_.values

                if not ndash.check\_leaf:

                    for j in ndash.keys:

                        j.parent = ndash

                self.deleteEntry(node\_.parent, value\_, node\_)

                del node\_

            else:

                if is\_predecessor == 1:

                    if not node\_.check\_leaf:

                        ndashpm = ndash.keys.pop(-1)

                        ndashkm\_1 = ndash.values.pop(-1)

                        node\_.keys = [ndashpm] + node\_.keys

                        node\_.values = [value\_] + node\_.values

                        parentNode = node\_.parent

                        for i, item in enumerate(parentNode.values):

                            if item == value\_:

                                parentNode.values[i] = ndashkm\_1

                                break

                    else:

                        ndashpm = ndash.keys.pop(-1)

                        ndashkm = ndash.values.pop(-1)

                        node\_.keys = [ndashpm] + node\_.keys

                        node\_.values = [ndashkm] + node\_.values

                        parentNode = node\_.parent

                        for i, item in enumerate(parentNode.values):

                            if item == value\_:

                                parentNode.values[i] = ndashkm

                                break

                else:

                    if not node\_.check\_leaf:

                        ndashp0 = ndash.keys.pop(0)

                        ndashk0 = ndash.values.pop(0)

                        node\_.keys = node\_.keys + [ndashp0]

                        node\_.values = node\_.values + [value\_]

                        parentNode = node\_.parent

                        for i, item in enumerate(parentNode.values):

                            if item == value\_:

                                parentNode.values[i] = ndashk0

                                break

                    else:

                        ndashp0 = ndash.keys.pop(0)

                        ndashk0 = ndash.values.pop(0)

                        node\_.keys = node\_.keys + [ndashp0]

                        node\_.values = node\_.values + [ndashk0]

                        parentNode = node\_.parent

                        for i, item in enumerate(parentNode.values):

                            if item == value\_:

                                parentNode.values[i] = ndash.values[0]

                                break

                if not ndash.check\_leaf:

                    for j in ndash.keys:

                        j.parent = ndash

                if not node\_.check\_leaf:

                    for j in node\_.keys:

                        j.parent = node\_

                if not parentNode.check\_leaf:

                    for j in parentNode.keys:

                        j.parent = parentNode

def printTree(tree):

    lst = [tree.root]

    level = [0]

    leaf = None

    flag = 0

    lev\_leaf = 0

    node1 = Node(str(level[0]) + str(tree.root.values))

    while (len(lst) != 0):

        x = lst.pop(0)

        lev = level.pop(0)

        if (x.check\_leaf == False):

            for i, item in enumerate(x.keys):

                print(item.values)

        else:

            for i, item in enumerate(x.keys):

                print(item.values)

            if (flag == 0):

                lev\_leaf = lev

                leaf = x

                flag = 1

record\_len = 3

bplustree = BplusTree(record\_len)

p='y'

while p == 'y':

    print('1.insert 2.delete 3.search')

    a = input("Enter your choice: ")

    if a == '1':

        b = int(input("Enter how many inputs: "))

        l1=[]

        for i in range(b):

            c = int(input("enter the number: "))

            l1.append(c)

        for i in l1:

            bplustree.insert(str(i),str(i))

        printTree(bplustree)

    elif a=='2':

        d = int(input("Enter the number you want to delete: "))

        bplustree.delete(str(d),str(d))

        print("now, output is ")

        printTree(bplustree)

    elif a =='3':

        e = int(input("enter the number to be searched: "))

        if (bplustree.find(str(e),str(e))):

            print("Element found")

        print("Element not found!!")

    else:

        raise Exception("Invalid input!!")

    p = input("Do you want to continue(y/n): ")

**OUTPUT:**

A screenshot of a computer

Description automatically generated