EXERCISE – 2 : B – PLUS TREE

AIM:

To perform the insertion, deletion and searching operation in a B+ tree of order 5. Implement different test cases for insertion and deletion Analyze the time complexity of code for B+ Tree and express the same using asymptotic notation .

ALGORITHM:

Initialization:

- Create a root node initially containing no keys or pointers.
- Set the root as the current node.

Insertion:

- Start at the root node.
- If the current node is a leaf node, insert the new key-value pair into it, maintaining sorted order.
- If the leaf node overflows (i.e., exceeds its maximum capacity), split it into two leaf nodes and promote the median key to the parent node.
- If the parent node overflows as a result of promoting the median key, split it recursively.
- If the root node splits, create a new root node with the median key as its sole element.

Deletion:

- Start at the root node and find the appropriate leaf node containing the key to be deleted.
- Remove the key from the leaf node.

- If the leaf node underflows (i.e., falls below a minimum capacity), borrow a key from a neighboring node, or merge it with a neighboring node.
- Propagate changes upwards, adjusting parent nodes as necessary.
- If the root node becomes empty after deletion, delete it and set its child as the new root.

Searching:

- Start at the root node.
- Compare the search key with the keys in the current node to determine the appropriate child node to search next.
- Continue searching down the tree until reaching a leaf node or finding the desired key.

Splitting:

- When splitting a node, allocate a new node to hold the larger half of the keys.
- Adjust pointers to maintain the tree structure.
- Promote the median key to the parent node.
- If the parent node overflows as a result of promoting the median key, split it recursively.

Merging:

- When merging two nodes, move all keys and pointers from one node to another.
- Adjust pointers to maintain the tree structure.
- Update the parent node to reflect the merged node.

Updating pointers:

- Ensure that pointers between nodes are updated correctly after insertion, deletion, splitting, or merging operations.

Balancing:

- Maintain the balance property of the B+ tree by adjusting node sizes and redistributing keys when necessary.

CODE:

```
# Python Implementation
class Node:
    def __init__(self, leaf = False):
        self.keys = []
        self.values = []
        self.leaf = leaf
        self.next = None

class BPlusTree:
    def __init__(self, degree):
        self.root = Node(leaf = True)
        self.degree = degree

    def search(self, key):
        curr = self.root
        while not curr.leaf:
            i = 0
            while i < len(curr.keys):
                if key < curr.keys[i]:
                    break
                i += 1
            curr = curr.values[i]
        i = 0
        while i < len(curr.keys):
            if curr.keys[i] == key:
                return True
            i += 1
        return False

    # Insert key value pairs
    def insert(self, key):
        curr = self.root
        if len(curr.keys) == 2 * self.degree:
            new_root = Node()
            self.root = new_root
```

```
        new_root.values.append(curr)
        self.split(new_root, 0, curr)
        self.insert_non_full(new_root, key)
    else:
        self.insert_non_full(curr, key)

def insert_non_full(self, curr, key):
    i = 0
    while i < len(curr.keys):
        if key < curr.keys[i]:
            break
        i += 1
    if curr.leaf:
        curr.keys.insert(i, key)
    else:
        if len(curr.values[i].keys) == 2 * self.degree:
            self.split(curr, i, curr.values[i])
            if key > curr.keys[i]:
                i += 1
        self.insert_non_full(curr.values[i], key)

def split(self, parent, i, node):
    new_node = Node(leaf = node.leaf)
    parent.values.insert(i + 1, new_node)
    parent.keys.insert(i, node.keys[self.degree-1])
    new_node.keys = node.keys[self.degree:]
    node.keys = node.keys[:self.degree-1]
    if not new_node.leaf:
        new_node.values = node.values[self.degree:]
        node.values = node.values[:self.degree]

def steal_from_left(self, parent, i):
    node = parent.values[i]
    left_sibling = parent.values[i-1]
    node.keys.insert(0, parent.keys[i-1])
    parent.keys[i-1] = left_sibling.keys.pop(-1)
    if not node.leaf:
        node.values.insert(0, left_sibling.values.pop(-1))

def steal_from_right(self, parent, i):
    node = parent.values[i]
    right_sibling = parent.values[i + 1]
    node.keys.append(parent.keys[i])
    parent.keys[i] = right_sibling.keys.pop(0)
    if not node.leaf:
```

```
            node.values.append(right_sibling.values.pop(0))

    # Del the given key
    def delete(self, key):
        curr = self.root
        found = False
        i = 0
        while i < len(curr.keys):
            if key == curr.keys[i]:
                found = True
                break
            elif key < curr.keys[i]:
                break
            i += 1
        if found:
            if curr.leaf:
                curr.keys.pop(i)
            else:
                pred = curr.values[i]
                if len(pred.keys) >= self.degree:
                    pred_key = self.get_max_key(pred)
                    curr.keys[i] = pred_key
                    self.delete_from_leaf(pred_key, pred)
                else:
                    succ = curr.values[i + 1]
                    if len(succ.keys) >= self.degree:
                        succ_key = self.get_min_key(succ)
                        curr.keys[i] = succ_key
                        self.delete_from_leaf(succ_key, succ)
                    else:
                        self.merge(curr, i, pred, succ)
                        self.delete_from_leaf(key, pred)
            if curr == self.root and not curr.keys:
                self.root = curr.values[0]
        else:
            if curr.leaf:
                return False
            else:
                if len(curr.values[i].keys) < self.degree:
                    if i != 0 and len(curr.values[i-1].keys) >= self.degree:
                        self.steal_from_left(curr, i)
                    elif i != len(curr.keys) and len(curr.values[i + 1].keys) >=
self.degree:
                        self.steal_from_right(curr, i)
                    else:
```

```
                    if i == len(curr.keys):
                        i -= 1
                    self.merge(curr, i, curr.values[i], curr.values[i + 1])
                self.delete(key)

    def delete_from_leaf(self, key, leaf):
        leaf.keys.remove(key)
        if leaf == self.root or len(leaf.keys) >= self.degree // 2:
            return
        parent = self.find_parent(leaf)
        i = parent.values.index(leaf)
        if i > 0 and len(parent.values[i-1].keys) > self.degree // 2:
            self.rotate_right(parent, i)
        elif i < len(parent.keys) and len(parent.values[i + 1].keys) >
self.degree // 2:
            self.rotate_left(parent, i)
        else:
            if i == len(parent.keys):
                i -= 1
            self.merge(parent, i, parent.values[i], parent.values[i + 1])

    def get_min_key(self, node):
        while not node.leaf:
            node = node.values[0]
        return node.keys[0]

    def get_max_key(self, node):
        while not node.leaf:
            node = node.values[-1]
        return node.keys[-1]

    def merge(self, parent, i, pred, succ):
        pred.keys += succ.keys
        pred.values += succ.values
        parent.values.pop(i + 1)
        parent.keys.pop(i)
        if parent == self.root and not parent.keys:
            self.root = pred

    def fix(self, parent, i):
        node = parent.values[i]
        if i > 0 and len(parent.values[i-1].keys) >= self.degree:
            self.rotate_right(parent, i)
        elif i < len(parent.keys) and len(parent.values[i + 1].keys) >=
self.degree:
```

```python
            self.rotate_left(parent, i)
        else:
            if i == len(parent.keys):
                i -= 1
            self.merge(parent, i, node, parent.values[i + 1])

    # Balance the tree after deletion
    def rotate_right(self, parent, i):
        node = parent.values[i]
        prev = parent.values[i-1]
        node.keys.insert(0, parent.keys[i-1])
        parent.keys[i-1] = prev.keys.pop(-1)
        if not node.leaf:
            node.values.insert(0, prev.values.pop(-1))

    def rotate_left(self, parent, i):
        node = parent.values[i]
        next = parent.values[i + 1]
        node.keys.append(parent.keys[i])
        parent.keys[i] = next.keys.pop(0)
        if not node.leaf:
            node.values.append(next.values.pop(0))

    # Function to print Tree
    def print_tree(self):
        curr_level = [self.root]
        while curr_level:
            next_level = []
            for node in curr_level:
                print(str(node.keys), end =' ')
                if not node.leaf:
                    next_level += node.values
            print()
            curr_level = next_level

if __name__=="__main__":
# create a B + tree with degree 5
    tree = BPlusTree(5)

    tree.insert(1)
    tree.insert(2)
    tree.insert(3)
    tree.insert(4)
    tree.insert(5)
    # tree.insert(6)
```

```
    # tree.insert(7)
    # tree.insert(8)
    # tree.insert(9)
    tree.print_tree() # [4] [2, 3] [6, 7, 8, 9] [1] [5]
    tree.delete(3)
    tree.print_tree() # [4] [2] [6, 7, 8, 9] [1] [5]
```

ANALYSIS :

Insertion time complexity :

Average time complexity for insertion is O (log n)

n	f(n)	n**2	log n	f(n)/n	f(n)/n**2	f(n)/logn
1000	0.0067	1000000	9.966	0.0000067	6.7E-09	0.000672
10000	0.00135	100000000	13.288	1.35E-07	1.35E-11	0.000102
20000	0.032	400000000	14.288	0.0000016	8E-11	0.00224
30000	0.02	300000000	14.873	6.667E-07	6.67E-11	0.001345
40000	0.038	1600000000	15.288	9.5E-07	2.38E-11	0.002486

Deletion time complexity :

Average time complexity for deletion is O(Log n)

n	f(n)	log n	f(n)/log n	f(n)/n
100	0.243	6.643856	0.036575	0.00243
500	0.06	8.965784	0.006692	0.00012
700	0.072	9.451211	0.007618	0.000103
900	0.04	9.813781	0.004076	4.44E-05

RESULT :

The insertion, deletion and searching operation in a B+ tree of order 5 has been implemented and time complexity analysis has been done successfully.