DSA LAB ASSIGNMENT 8

Name : Mahesh Jagtap

Reg No. 24MCS1017

Lab Scenario: B+-Tree

Somu is an aspiring database engineer working on a university project. His task is to manage a large dataset using a **B+-tree**, an efficient way to index and search for data. His professor gave him some rules to follow to make sure the tree stays balanced. Somu's goal is to practice inserting and deleting keys from the B+-tree while keeping it well-structured.

Here's the challenge Somu faces:

Lab Task: Help Somu Maintain the B+-tree!

Rules Somu Must Follow:

1. **B+-tree Rules**:
    - Each node in the tree can hold **up to 3 keys and 4 pointers**.
    - The tree should be **half-full** at all times:
        - **Leaf nodes** must have at least 2 keys.
        - **Non-leaf nodes** (internal nodes) must have at least 1 key (2 pointers).

The Task:

1. **Insertion Phase**:
    - Somu starts with an empty B+-tree. He needs to insert the following keys into the tree, **one by one**: **1, 3, 5, 7, 9, 2, 4, 6, 8, 10**.
    - As he inserts each key, the tree will need to **split nodes** when they get too full (i.e., have more than 3 keys).
    - After each insertion, Somu needs to make sure the tree stays balanced.
2. **Deletion Phase**:
    - Once all the keys have been inserted, Somu is asked to **remove** the following keys in order: **9, 7, 8**.
    - After each deletion, he needs to make sure no nodes become too empty.
    - If a node has too few keys, Somu must **merge** it with another node or **borrow** a key from its neighbor to keep the tree balanced.

Steps to Follow:

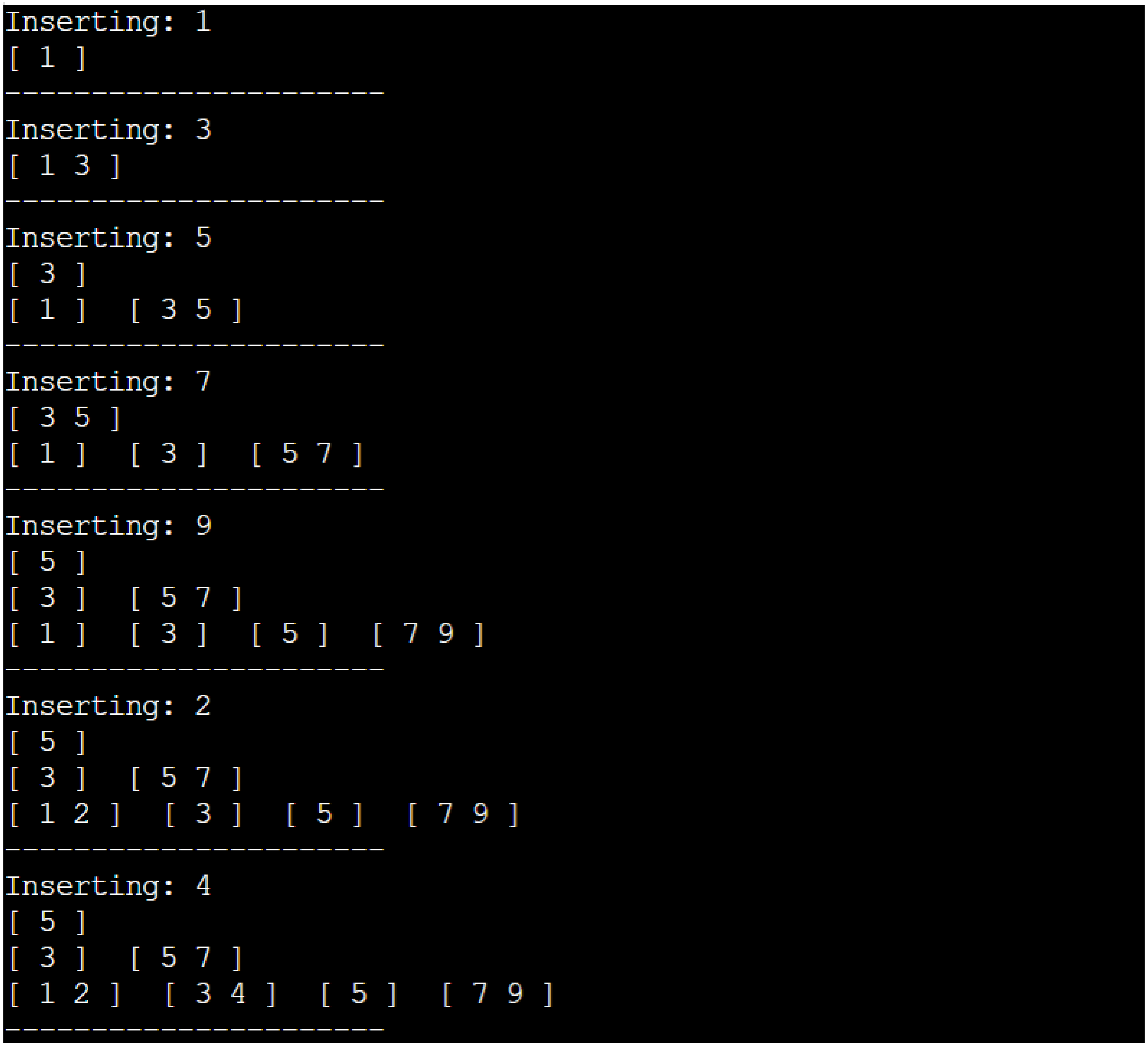
1. **Start with an Empty B+-tree**.
    - Draw or visualize each step as you insert keys one by one.
2. **Insert the Following Keys**: 1, 3, 5, 7, 9, 2, 4, 6, 8, 10.

    - After each insertion, check if the tree needs to split. If it does, record the split and how the keys are moved.
3. **Delete the Following Keys**: 9, 7, 8.
    - After each deletion, check if any node needs to borrow keys from its neighbor or merge with another node.

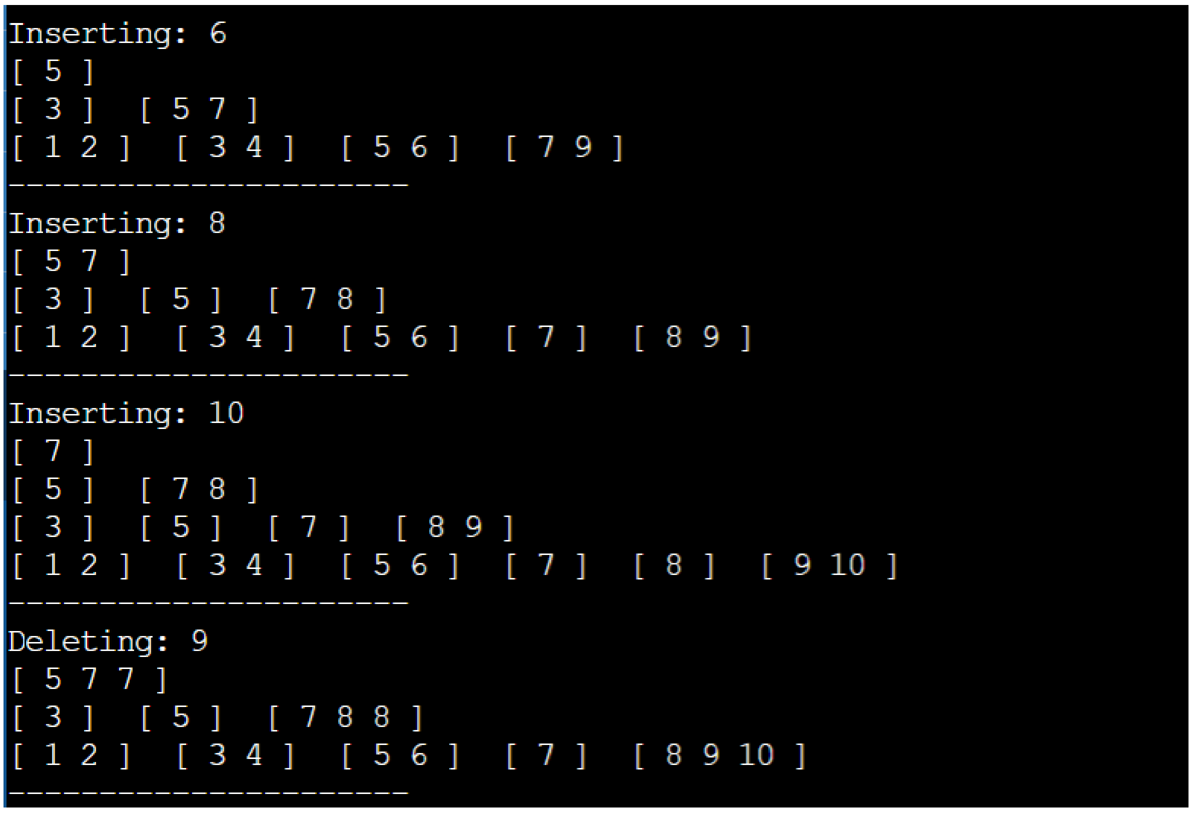
Code:

```
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>

using namespace std;

const int MAX_KEYS = 3;  // Maximum number of keys a node can hold
const int MIN_KEYS = 2;  // Minimum number of keys for leaf nodes

// Node structure for B+ Tree
class BPlusTreeNode {
public:
    vector<int> keys;  // Keys in the node
    vector<BPlusTreeNode*> children;  // Pointers to children (if not a leaf)
    bool isLeaf;  // True if the node is a leaf

    // Constructor
    BPlusTreeNode(bool leaf = true) : isLeaf(leaf) {}

    // Check if the node is full
    bool isFull() {
        return keys.size() >= MAX_KEYS;
    }

    // Check if the node is underfilled
    bool isUnderfilled() {
        return keys.size() < MIN_KEYS;
    }
};

// B+ Tree class
class BPlusTree {
public:
    BPlusTreeNode* root;

    BPlusTree() {
        root = new BPlusTreeNode();
    }

    // Insertion wrapper
    void insert(int key) {
        BPlusTreeNode* newRoot = insertRecursive(root, key);
        if (newRoot != nullptr) {
            // Tree height increases, create a new root
            BPlusTreeNode* newTreeRoot = new BPlusTreeNode(false);
            newTreeRoot->keys.push_back(newRoot->keys[0]);
            newTreeRoot->children.push_back(root);
            newTreeRoot->children.push_back(newRoot);
            root = newTreeRoot;
        }
    }

    // Deletion wrapper
    void remove(int key) {
        removeRecursive(root, key);
        if (root->keys.empty() && !root->isLeaf) {
            root = root->children[0]; // Shrink the tree height if the root is empty
        }
    }

    // Display the tree (Level-order traversal for better visualization)
    void printTree() {
        if (!root) return;
        queue<BPlusTreeNode*> q;
        q.push(root);

        while (!q.empty()) {
            int levelSize = q.size();
            while (levelSize--) {
                BPlusTreeNode* node = q.front();
                q.pop();

                // Print current node's keys
                cout << "[ ";
                for (int key : node->keys) {
                    cout << key << " ";
                }
                cout << "]  ";

                // Add children to the queue for the next level
                if (!node->isLeaf) {
                    for (BPlusTreeNode* child : node->children) {
                        q.push(child);
                    }
                }
            }
            cout << "\n";  // Newline for each level
        }
    }

private:
    // Insertion helper
    BPlusTreeNode* insertRecursive(BPlusTreeNode* node, int key) {
        // Leaf node case: Insert the key
        if (node->isLeaf) {
            node->keys.insert(lower_bound(node->keys.begin(), node->keys.end(), key), key);
            if (node->isFull()) {
                return split(node);
            }
            return nullptr;
        } else {
            // Internal node case: Find the correct child to recurse into
                        int   i   =   upper_bound(node->keys.begin(),   node->keys.end(),   key)   -
  node->keys.begin();
            BPlusTreeNode* newChild = insertRecursive(node->children[i], key);

            if (newChild != nullptr) {
                // Insert new key and child into the current internal node
                node->keys.insert(node->keys.begin() + i, newChild->keys[0]);
                node->children.insert(node->children.begin() + i + 1, newChild);
                if (node->isFull()) {
                    return split(node);
                }
            }
            return nullptr;
        }
    }

    // Split the node
    BPlusTreeNode* split(BPlusTreeNode* node) {
        BPlusTreeNode* newNode = new BPlusTreeNode(node->isLeaf);
        int mid = node->keys.size() / 2;

        newNode->keys.assign(node->keys.begin() + mid, node->keys.end());
        node->keys.erase(node->keys.begin() + mid, node->keys.end());

        if (!node->isLeaf) {
            newNode->children.assign(node->children.begin() + mid, node->children.end());
            node->children.erase(node->children.begin() + mid, node->children.end());
        }

        return newNode;
    }

    // Deletion helper
    void removeRecursive(BPlusTreeNode* node, int key) {
        // Leaf node case
        if (node->isLeaf) {
            auto it = find(node->keys.begin(), node->keys.end(), key);
            if (it != node->keys.end()) {
                node->keys.erase(it);
            }
        } else {
            // Internal node case: Find the correct child to recurse into
                        int   i   =   upper_bound(node->keys.begin(),   node->keys.end(),   key)   -
  node->keys.begin();
            removeRecursive(node->children[i], key);

            // Balance the node after deletion
            if (node->children[i]->isUnderfilled()) {
                balance(node, i);
            }
        }
    }
    // Balance the tree after deletion
    void balance(BPlusTreeNode* node, int idx) {
        if (idx > 0 && node->children[idx - 1]->keys.size() > MIN_KEYS) {
            // Borrow from left sibling
            BPlusTreeNode* leftSibling = node->children[idx - 1];
            BPlusTreeNode* currNode = node->children[idx];

            currNode->keys.insert(currNode->keys.begin(), node->keys[idx - 1]);
            node->keys[idx - 1] = leftSibling->keys.back();
            leftSibling->keys.pop_back();

            if (!currNode->isLeaf) {
                                                currNode->children.insert(currNode->children.begin(),
leftSibling->children.back());
                leftSibling->children.pop_back();
            }
        } else if (idx < node->children.size() - 1 && node->children[idx + 1]->keys.size() >
MIN_KEYS) {
            // Borrow from right sibling
            BPlusTreeNode* rightSibling = node->children[idx + 1];
            BPlusTreeNode* currNode = node->children[idx];

            currNode->keys.push_back(node->keys[idx]);
            node->keys[idx] = rightSibling->keys[0];
            rightSibling->keys.erase(rightSibling->keys.begin());

            if (!currNode->isLeaf) {
                currNode->children.push_back(rightSibling->children[0]);
                rightSibling->children.erase(rightSibling->children.begin());
            }
        } else {
            // Merge with sibling
            if (idx > 0) {
                merge(node, idx - 1);
            } else {
                merge(node, idx);
            }
        }
    }

    // Merge two child nodes
    void merge(BPlusTreeNode* node, int idx) {
        BPlusTreeNode* leftChild = node->children[idx];
        BPlusTreeNode* rightChild = node->children[idx + 1];

        leftChild->keys.push_back(node->keys[idx]);
                         leftChild->keys.insert(leftChild->keys.end(),    rightChild->keys.begin(),
rightChild->keys.end());
        if (!leftChild->isLeaf) {
             leftChild->children.insert(leftChild->children.end(), rightChild->children.begin(),
rightChild->children.end());
        }

        node->keys.erase(node->keys.begin() + idx);
        node->children.erase(node->children.begin() + idx + 1);

        delete rightChild;
    }
};

// Main function
int main() {
    BPlusTree tree;

    // Insertion phase
    vector<int> insertKeys = {1, 3, 5, 7, 9, 2, 4, 6, 8, 10};
    for (int key : insertKeys) {
        cout << "Inserting: " << key << endl;
        tree.insert(key);
        tree.printTree();
        cout << "---------------------\n";
    }

    // Deletion phase
    vector<int> deleteKeys = {9, 7, 8};
    for (int key : deleteKeys) {
        cout << "Deleting: " << key << endl;
        tree.remove(key);
        tree.printTree();
        cout << "---------------------\n";
    }

    return 0;
}
```

Output:



```
Inserting: 1
[ 1 ]
----------------------
Inserting: 3
[ 1 3 ]
----------------------
Inserting: 5
[ 3 ]
[ 1 ]  [ 3 5 ]
----------------------
Inserting: 7
[ 3 5 ]
[ 1 ]  [ 3 ]  [ 5 7 ]
----------------------
Inserting: 9
[ 5 ]
[ 3 ]  [ 5 7 ]
[ 1 ]  [ 3 ]  [ 5 ]  [ 7 9 ]
----------------------
Inserting: 2
[ 5 ]
[ 3 ]  [ 5 7 ]
[ 1 2 ]  [ 3 ]  [ 5 ]  [ 7 9 ]
----------------------
Inserting: 4
[ 5 ]
[ 3 ]  [ 5 7 ]
[ 1 2 ]  [ 3 4 ]  [ 5 ]  [ 7 9 ]
----------------------
```



```
Inserting: 6
[ 5 ]
[ 3 ]  [ 5 7 ]
[ 1 2 ]  [ 3 4 ]  [ 5 6 ]  [ 7 9 ]
----------------------
Inserting: 8
[ 5 7 ]
[ 3 ]  [ 5 ]  [ 7 8 ]
[ 1 2 ]  [ 3 4 ]  [ 5 6 ]  [ 7 ]  [ 8 9 ]
----------------------
Inserting: 10
[ 7 ]
[ 5 ]  [ 7 8 ]
[ 3 ]  [ 5 ]  [ 7 ]  [ 8 9 ]
[ 1 2 ]  [ 3 4 ]  [ 5 6 ]  [ 7 ]  [ 8 ]  [ 9 10 ]
----------------------
Deleting: 9
[ 5 7 7 ]
[ 3 ]  [ 5 ]  [ 7 8 8 ]
[ 1 2 ]  [ 3 4 ]  [ 5 6 ]  [ 7 ]  [ 8 9 10 ]
----------------------
```

Bonus Question:

Why do you think databases use B+-trees instead of simple binary search trees (BSTs)? How do B+-trees help in handling large datasets efficiently?

Answer:

Databases use **B+-trees** instead of **Binary Search Trees (BSTs)** for several reasons:

1. **Balanced Structure**: B+-trees remain balanced, ensuring consistent **O(log n)** time for search, insert, and delete operations.
2. **Efficient Disk Utilization**: They store multiple keys per node, reducing disk I/O operations, which is crucial for large datasets.
3. **Faster Range Queries**: B+-trees allow quick access to ranges of values, making them ideal for querying.
4. **Sequential Access**: Leaf nodes are linked, enabling easy sorted data access.
5. **Lower Height**: B+-trees have fewer levels, leading to faster search times.