A-2

Name: Muhammad Bilal

Roll No:23K-0026

Sec: BCS-3F

Code:

```cpp
#include <iostream>
#include <string>
#include <chrono>
#include <vector>
#include<functional>
using namespace std;

// Placeholder classes for BST, AVL, and B-Tree
struct Record
{
    int id;
    string name;
    int age;
};
class AVLNode
{
public:
    Record data;
    AVLNode *left;
    AVLNode *right;
    int height;

    AVLNode(Record rec) : data(rec), left(nullptr), right(nullptr), height(1) {}
};

class Node
{
public:
    Record data;
    Node *left;
    Node *right;

    Node(Record record) : data(record), left(nullptr), right(nullptr) {}
};
```

```
class BST
{
private:
    Node *root;
    void inorder(Node *node)
    {
        if (node != nullptr)
        {
            inorder(node->left);
            inorder(node->right);
        }
    }

    Node *insert(Node *root, Record record)
    {
        if (root == nullptr)
        {
            return new Node(record);
        }

        if (record.id > root->data.id)
        {
            root->right = insert(root->right, record);
        }
        else
        {
            root->left = insert(root->left, record);
        }
        return root;
    }

    Node *search(Node *root, int id)
    {
        if (root == nullptr || root->data.id == id)
        {
            return root;
        }

        if (id < root->data.id)
        {
            return search(root->left, id);
        }
        else
```

```
        {
            return search(root->right, id);
        }
    }


    Node *getmin(Node *node)
    {
        while (node && node->left)
        {
            node = node->left;
        }
        return node;
    }

    Node *deleteNode(Node *root, int id)
    {
        if (root == nullptr)
        {
            return root;
        }

        if (id < root->data.id)
        {
            root->left = deleteNode(root->left, id);
        }
        else if (id > root->data.id)
        {
            root->right = deleteNode(root->right, id);
        }
        else
        {

            if (root->left == nullptr)
            {
                Node *temp = root->right;
                delete root;
                return temp;
            }
            else if (root->right == nullptr)
            {
                Node *temp = root->left;
                delete root;
```

```
                return temp;
            }
            Node *temp = getmin(root->right);
            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data.id);
        }
        return root;
    }

public:
    BST() : root(nullptr) {}

    void insert(Record record)
    {
        root = insert(root, record);
    }

    Record *search(int id)
    {
        Node *node = search(root, id);
        return node ? &node->data : nullptr;
    }

    void update(int id, const string &name, int age)
    {
        Node *node = search(root, id);
        if (node)
        {
            node->data.name = name;
            node->data.age = age;
        }
    }

    void remove(int id)
    {
        root = deleteNode(root, id);
    }

    bool isEmpty() {
        return root == nullptr;
    }
};
```

```
// AVL Tree class
class AVLTree
{
private:
    AVLNode *root;

    void inorder(AVLNode *node)
    {
        if (node != nullptr)
        {
            inorder(node->left);
            inorder(node->right);
        }
    }

    // Helper function to calculate the height of a node
    int Nodeheight(AVLNode *node)
    {
        if (!node)
            return 0;
        int leftHeight = Nodeheight(node->left);
        int rightHeight = Nodeheight(node->right);
        return 1 + max(leftHeight, rightHeight);
    }

    // Helper function to calculate the balance factor of a node
    int balanceFactor(AVLNode *node)
    {
        if (!node)
            return 0;
        return Nodeheight(node->left) - Nodeheight(node->right);
    }

    // Right rotation
    AVLNode *RRrotation(AVLNode *root)
    {
        AVLNode *newRoot = root->right;
        root->right = newRoot->left;
        newRoot->left = root;
        root->height = Nodeheight(root);
        newRoot->height = Nodeheight(newRoot);
        return newRoot;
    }
```

```
// Left rotation
AVLNode *LLrotation(AVLNode *root)
{
    AVLNode *newRoot = root->left;
    root->left = newRoot->right;
    newRoot->right = root;
    root->height = Nodeheight(root);
    newRoot->height = Nodeheight(newRoot);
    return newRoot;
}

// Left-Right rotation
AVLNode *LRrotation(AVLNode *root)
{
    root->left = RRrotation(root->left);
    return LLrotation(root);
}

// Right-Left rotation
AVLNode *RLrotation(AVLNode *root)
{
    root->right = LLrotation(root->right);
    return RRrotation(root);
}

AVLNode *insert(AVLNode *root, Record record)
{
    if (!root)
        return new AVLNode(record);

    if (record.id < root->data.id)
    {
        root->left = insert(root->left, record);
    }
    else if (record.id > root->data.id)
    {
        root->right = insert(root->right, record);
    }
    else
    {
        return root;
    }
```

```
    root->height = Nodeheight(root);
    int balance = balanceFactor(root);

    if (balance > 1 && record.id < root->left->data.id)
        return LLrotation(root);
    if (balance < -1 && record.id > root->right->data.id)
        return RRrotation(root);
    if (balance > 1 && record.id > root->left->data.id)
        return LRrotation(root);
    if (balance < -1 && record.id < root->right->data.id)
        return RLrotation(root);

    return root;
}

AVLNode *search(AVLNode *root, int id)
{
    if (!root || root->data.id == id)
        return root;
    if (id < root->data.id)
        return search(root->left, id);
    return search(root->right, id);
}

AVLNode *deleteNode(AVLNode *root, int id)
{
    if (!root)
        return root;

    if (id < root->data.id)
    {
        root->left = deleteNode(root->left, id);
    }
    else if (id > root->data.id)
    {
        root->right = deleteNode(root->right, id);
    }
    else
    {
        if (!root->left)
        {
            AVLNode *temp = root->right;
```

```
                delete root;
                return temp;
            }
            else if (!root->right)
            {
                AVLNode *temp = root->left;
                delete root;
                return temp;
            }

            AVLNode *temp = root->right;
            while (temp && temp->left)
                temp = temp->left;

            root->data = temp->data;
            root->right = deleteNode(root->right, temp->data.id);
        }

        if (!root)
            return root;

        root->height = Nodeheight(root);
        int balance = balanceFactor(root);

        if (balance > 1 && balanceFactor(root->left) >= 0)
            return LLrotation(root);
        if (balance > 1 && balanceFactor(root->left) < 0)
            return LRrotation(root);
        if (balance < -1 && balanceFactor(root->right) <= 0)
            return RRrotation(root);
        if (balance < -1 && balanceFactor(root->right) > 0)
            return RLrotation(root);

        return root;
    }

public:
    AVLTree() : root(nullptr) {}

    void insert(Record record)
    {
        root = insert(root, record);  // Insert record into the tree
    }
```

```
    Record *search(int id)
    {
        AVLNode *node = search(root, id);
        return node ? &node->data : nullptr;
    }

    void update(int id, const string &name, int age)
    {
        Record *record = search(id);
        if (record)
        {
            record->name = name;
            record->age = age;
        }
    }

    void remove(int id)
    {
        root = deleteNode(root, id);
    }

    void printTree()
    {
        if (root == nullptr)
        {
            cout << "The AVL Tree is empty.\n";
        }
        else
        {
            inorder(root);
        }
    }
};

class BTreeNode {
public:
    vector<Record> elements;
    vector<BTreeNode*> subNodes;
    int minDegree;
    bool isLeaf;

    BTreeNode(int _minDegree, bool _isLeaf) {
```

```
        minDegree = _minDegree;
        isLeaf = _isLeaf;
    }

    void insertIntoNonFullNode(Record record) {
        int index = elements.size() - 1;

        if (isLeaf) {
            elements.push_back({});
            while (index >= 0 && record.id < elements[index].id) {
                elements[index + 1] = elements[index];
                index--;
            }
            elements[index + 1] = record;
        } else {

            while (index >= 0 && record.id < elements[index].id) {
                index--;
            }
            index++;



            if (subNodes[index]->elements.size() == 2 * minDegree - 1) {
                splitChildNode(index, subNodes[index]);
                if (record.id > elements[index].id) {
                    index++;
                }
            }
            subNodes[index]->insertIntoNonFullNode(record);
        }
    }



    void splitChildNode(int index, BTreeNode* fullChild) {
        BTreeNode* newChild = new BTreeNode(fullChild->minDegree, fullChild-
>isLeaf);



        for (int i = 0; i < minDegree - 1; i++) {
            newChild->elements.push_back(fullChild->elements[i + minDegree]);
        }
```

```
        if (!fullChild->isLeaf) {

            for (int i = 0; i < minDegree; i++) {
                newChild->subNodes.push_back(fullChild->subNodes[i + minDegree]);
            }
        }



        fullChild->elements.resize(minDegree - 1);
        if (!fullChild->isLeaf) {
            fullChild->subNodes.resize(minDegree);
        }



        subNodes.insert(subNodes.begin() + index + 1, newChild);
        elements.insert(elements.begin() + index, fullChild->elements[minDegree -
1]);
    }

    BTreeNode* searchNode(int id) {
        int i = 0;
        while (i < elements.size() && id > elements[i].id) {
            i++;
        }

        if (i < elements.size() && id == elements[i].id) {
            return this;
        }

        if (isLeaf) {
            return nullptr;
        }

        return subNodes[i]->searchNode(id);
    }



    BTreeNode* deleteRecordNode(BTreeNode* node, int id) {
        if (!node) return node;

        int i = 0;
        while (i < node->elements.size() && id > node->elements[i].id) {
```

```
            i++;
        }

        if (i < node->elements.size() && id == node->elements[i].id) {
            if (node->isLeaf) {
                node->elements.erase(node->elements.begin() + i);
            } else {
                if (node->subNodes[i + 1]->elements.size() >= node->minDegree) {
                    BTreeNode* temp = node->subNodes[i + 1];
                    while (!temp->isLeaf) {
                        temp = temp->subNodes[0];
                    }
                    node->elements[i] = temp->elements[0];
                    deleteRecordNode(node->subNodes[i + 1], temp-
>elements[0].id);
                } else if (node->subNodes[i]->elements.size() >= node->minDegree)
{
                    BTreeNode* temp = node->subNodes[i];
                    while (!temp->isLeaf) {
                        temp = temp->subNodes[temp->elements.size()];
                    }
                    node->elements[i] = temp->elements[temp->elements.size() -
1];
                    deleteRecordNode(node->subNodes[i], temp->elements[temp-
>elements.size() - 1].id);
                } else {
                    BTreeNode* leftChild = node->subNodes[i];
                    BTreeNode* rightChild = node->subNodes[i + 1];
                    leftChild->elements.push_back(node->elements[i]);
                    for (int j = 0; j < rightChild->elements.size(); j++) {
                        leftChild->elements.push_back(rightChild->elements[j]);
                    }
                    for (int j = 0; j < rightChild->subNodes.size(); j++) {
                        leftChild->subNodes.push_back(rightChild->subNodes[j]);
                    }
                    node->elements.erase(node->elements.begin() + i);
                    node->subNodes.erase(node->subNodes.begin() + i + 1);
                    deleteRecordNode(leftChild, id);
                }
            }
        } else {
            if (node->subNodes[i]->elements.size() < node->minDegree) {
                if (i > 0 && node->subNodes[i - 1]->elements.size() >= node-
>minDegree) {
```

```
                    BTreeNode* child = node->subNodes[i];
                    BTreeNode* leftSibling = node->subNodes[i - 1];
                    child->elements.insert(child->elements.begin(), node-
>elements[i - 1]);
                    node->elements[i - 1] = leftSibling->elements[leftSibling-
>elements.size() - 1];
                    leftSibling->elements.pop_back();
                    if (!leftSibling->isLeaf) {
                        child->subNodes.insert(child->subNodes.begin(),
leftSibling->subNodes[leftSibling->subNodes.size() - 1]);
                        leftSibling->subNodes.pop_back();
                    }
                } else if (i < node->elements.size() && node->subNodes[i + 1]-
>elements.size() >= node->minDegree) {
                    BTreeNode* child = node->subNodes[i];
                    BTreeNode* rightSibling = node->subNodes[i + 1];
                    child->elements.push_back(node->elements[i]);
                    node->elements[i] = rightSibling->elements[0];
                    rightSibling->elements.erase(rightSibling->elements.begin());
                    if (!rightSibling->isLeaf) {
                        child->subNodes.push_back(rightSibling->subNodes[0]);
                        rightSibling->subNodes.erase(rightSibling-
>subNodes.begin());
                    }
                } else {
                    if (i < node->elements.size()) {
                        BTreeNode* child = node->subNodes[i];
                        BTreeNode* rightSibling = node->subNodes[i + 1];
                        child->elements.push_back(node->elements[i]);
                        for (int j = 0; j < rightSibling->elements.size(); j++) {
                            child->elements.push_back(rightSibling->elements[j]);
                        }
                        for (int j = 0; j < rightSibling->subNodes.size(); j++) {
                            child->subNodes.push_back(rightSibling->subNodes[j]);
                        }
                        node->elements.erase(node->elements.begin() + i);
                        node->subNodes.erase(node->subNodes.begin() + i + 1);
                        deleteRecordNode(child, id);
                    } else {
                        BTreeNode* child = node->subNodes[i];
                        BTreeNode* leftSibling = node->subNodes[i - 1];
                        leftSibling->elements.push_back(node->elements[i - 1]);
                        for (int j = 0; j < child->elements.size(); j++) {
                            leftSibling->elements.push_back(child->elements[j]);
```

```
                        }
                        for (int j = 0; j < child->subNodes.size(); j++) {
                            leftSibling->subNodes.push_back(child->subNodes[j]);
                        }
                        node->elements.erase(node->elements.begin() + i - 1);
                        node->subNodes.erase(node->subNodes.begin() + i);
                        deleteRecordNode(leftSibling, id);
                    }
                }
            } else {
                deleteRecordNode(node->subNodes[i], id);
            }
        }

        return node;
    }

};

Record generateRandomRecord()
{
    Record record;
    record.id = rand() % 100000;
    record.name = "Name_" + to_string(rand() % 1000);
    record.age = rand() % 100 + 20;
    return record;
}
vector<Record> generateDummyData(int size)
{
    vector<Record> data;
    for (int i = 0; i < size; ++i)
    {
        data.push_back(generateRandomRecord());
    }
    return data;
}

void analyzePerformance(vector<Record> &data, BST &bst, AVLTree &avl, BTreeNode
&btreeRoot, int btreeMinDegree) {
    auto measureExecutionTime = [](auto operation) -> double {
        auto startTime = std::chrono::high_resolution_clock::now();
        operation();
        auto endTime = std::chrono::high_resolution_clock::now();
```

```
        return std::chrono::duration<double, std::milli>(endTime -
startTime).count();
    };

    // Insertion Performance
    cout << "Insertion Performance:\n";
    double bstInsertTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            bst.insert(record);
    });
    cout << "  BST: " << bstInsertTime << " ms\n";

    double avlInsertTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            avl.insert(record);
    });
    cout << "  AVL: " << avlInsertTime << " ms\n";

    double btreeInsertTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            btreeRoot.insertIntoNonFullNode(record);
    });
    cout << "  BTree: " << btreeInsertTime << " ms\n";

    // Search Performance
    cout << "\nSearch Performance:\n";
    double bstSearchTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            bst.search(record.id);
    });
    cout << "  BST: " << bstSearchTime << " ms\n";

    double avlSearchTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            avl.search(record.id);
    });
    cout << "  AVL: " << avlSearchTime << " ms\n";

    double btreeSearchTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            btreeRoot.searchNode(record.id);
    });
    cout << "  BTree: " << btreeSearchTime << " ms\n";
```

```
    // Update Performance
    cout << "\nUpdate Performance:\n";
    double bstUpdateTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            bst.update(record.id, "UpdatedName", record.age + 1);
    });
    cout << "  BST: " << bstUpdateTime << " ms\n";

    double avlUpdateTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            avl.update(record.id, "UpdatedName", record.age + 1);
    });
    cout << "  AVL: " << avlUpdateTime << " ms\n";

    double btreeUpdateTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            btreeRoot.searchNode(record.id); // Simulating update by searching
    });
    cout << "  BTree: " << btreeUpdateTime << " ms\n";

    // Delete Performance
    cout << "\nDelete Performance:\n";
    double bstDeleteTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            bst.remove(record.id);
    });
    cout << "  BST: " << bstDeleteTime << " ms\n";

    double avlDeleteTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            avl.remove(record.id);
    });
    cout << "  AVL: " << avlDeleteTime << " ms\n";

    double btreeDeleteTime = measureExecutionTime([&]() {
        for (const auto &record : data)
            btreeRoot.deleteRecordNode(&btreeRoot, record.id);
    });
    cout << "  BTree: " << btreeDeleteTime << " ms\n";
}
```

```cpp
int main() {
    const vector<int> datasetSizes = {5000, 10000, 50000};
    int btreeMinDegree = 3;

    cout << "Performance Analysis of BST, AVL Tree, and BTree:\n";

    for (auto size : datasetSizes) {
        cout << "\nDataset Size: " << size << endl;

        vector<Record> data = generateDummyData(size);
        BST bst;
        AVLTree avl;
        BTreeNode btreeRoot(btreeMinDegree, true);

        analyzePerformance(data, bst, avl, btreeRoot, btreeMinDegree);
    }
     // Time Complexity Analysis of Data Structures (BST, AVL, and B-Tree)
    // Expected Time Complexity:
    // Binary Search Tree (BST):
    // - Best case (balanced tree): O(log N)
    // - Worst case (unbalanced tree): O(N) (in case of a skewed tree)
    //
    // AVL Tree:
    // - Best and worst case: O(log N) for all operations because AVL trees are
balanced by definition.
    //
    // B-Tree:
    // - Best and worst case: O(log N) for insertion, search, and deletion
operations because B-trees maintain balanced nodes.

/*
Performance Table
Dataset Operation   BST(ms) AVL Time (ms)   B-Tree Time (ms)
5,000   Insertion   2.96     358.22                104.47
5,000   Search     0.62     0.73                      52.67
5,000   Update     0.89     1.12                      51.29
5,000   Delete     1.08     353.61                    208.50
10,000  Insertion   6.27    2101.47               544.98
10,000  Search    2.19      2.10                      267.06
10,000  Update    2.88      2.60                      214.27
10,000  Delete    2.57     2081.04                    387.30
50,000  Insertion 34.81    66315.50              12895.50
50,000  Search    13.27     77.71                     6683.36
```

```
50,000  Update    16.90     12.99                      6241.52
50,000  Delete    17.24    53582.80                 11668.30*/
/*Insertion Time: The BST is the fastest for insertion in all datasets, followed
by the B-Tree and then the AVL Tree, which is the slowest.
Search Time: The BST is the fastest for search operations, closely followed by
the AVL Tree. The B-Tree has significantly slower search times.
Update Time: The BST remains the fastest for updates, with the AVL Tree being
slightly slower, and the B-Tree again taking the longest time for updates.
Delete Time: The BST is the fastest for deletion operations as well, with the AVL
Tree taking much longer for larger datasets. The B-Tree also shows slower
performance for deletions compared to the BST and AVL Tree.
This table summarizes the performance of each tree type in terms of insertion,
search, update, and delete operations across different dataset sizes.*/

}
```