Demo Praktikum 3 Struktur Data

Muhammad Alfan Mahdi – 5025221275

1. BPKA

Soal ini meminta untuk menghitung total dari penjumlahan dari setiap kuadrat dari jumlah per kolom.

Source Code:

```
#include<bits/stdc++.h>

using namespace std;

typedef struct AVLNode_t
{
    int data;
    struct AVLNode_t *left,*right;
    int height;
}AVLNode;

typedef struct AVL_t
{
    AVLNode *_root;
    unsigned int _size;
}AVL;

void avl_init(AVL *avl) {
    avl->_root = NULL;
    avl->_size = 0u;
}

int _getHeight(AVLNode* node){
    if(node==NULL)
        return 0;
    return node->height;
}

int _max(int a,int b){
    return (a > b)? a : b;
}

int _getBalanceFactor(AVLNode* node){
    if(node==NULL)
        return 0;
    return _getHeight(node->left)-_getHeight(node->right);
}
```

```
AVLNode* _rightRotate(AVLNode* pivotNode){
    AVLNode* newParrent=pivotNode->left;
    pivotNode->left=newParrent->right;
    newParrent->right=pivotNode;
    pivotNode->height=_max(_getHeight(pivotNode->left),_getHeight(pivotNode-
>right))+1;
    newParrent->height=_max(_getHeight(newParrent-
>left),_getHeight(newParrent->right))+1;
    return newParrent;
}

AVLNode* _leftCaseRotate(AVLNode* node){
    return _rightRotate(node);
}

AVLNode* _leftRotate(AVLNode* pivotNode){

    AVLNode* newParrent=pivotNode->right;
    pivotNode->right=newParrent->left;
    newParrent->left=pivotNode;

    pivotNode->height=_max(_getHeight(pivotNode->left),
                    _getHeight(pivotNode->right))+1;
    newParrent->height=_max(_getHeight(newParrent->left),
                    _getHeight(newParrent->right))+1;

    return newParrent;
}

AVLNode* _rightCaseRotate(AVLNode* node){
    return _leftRotate(node);
}

AVLNode* _leftRightCaseRotate(AVLNode* node){
    node->left=_leftRotate(node->left);
    return _rightRotate(node);
}

AVLNode* _rightLeftCaseRotate(AVLNode* node){
    node->right=_rightRotate(node->right);
    return _leftRotate(node);
}

AVLNode* _avl_createNode(int value) {
    AVLNode *newNode = (AVLNode*) malloc(sizeof(AVLNode));
    newNode->data = value;
    newNode->height=1;
```

```
    newNode->left = newNode->right = NULL;
    return newNode;
}

AVLNode* _search(AVLNode *root, int value) {
    while (root != NULL) {
        if (value < root->data)
            root = root->left;
        else if (value > root->data)
            root = root->right;
        else
            return root;
    }
    return root;
}

AVLNode* _insert_AVL(AVL *avl,AVLNode* node,int value){

    if(node==NULL)
        return _avl_createNode(value);
    if(value < node->data)
        node->left = _insert_AVL(avl,node->left,value);
    else if(value > node->data)
        node->right = _insert_AVL(avl,node->right,value);

    node->height= 1 + _max(_getHeight(node->left),_getHeight(node->right));

    int balanceFactor=_getBalanceFactor(node);

    if(balanceFactor > 1 && value < node->left->data)
        return _leftCaseRotate(node);
    if(balanceFactor > 1 && value > node->left->data)
        return _leftRightCaseRotate(node);
    if(balanceFactor < -1 && value > node->right->data)
        return _rightCaseRotate(node);
    if(balanceFactor < -1 && value < node->right->data)
        return _rightLeftCaseRotate(node);

    return node;
}

bool avl_find(AVL *avl, int value) {
    AVLNode *temp = _search(avl->_root, value);
    if (temp == NULL)
        return false;

    if (temp->data == value)
        return true;
```

```
    else
        return false;
}

void avl_insert(AVL *avl,int value){
    if(!avl_find(avl,value)){
        avl->_root = _insert_AVL(avl,avl->_root,value);
        avl->_size++;
    }
}

AVLNode* _findMinNode(AVLNode *node) {
    AVLNode *currNode = node;
    while (currNode && currNode->left != NULL)
        currNode = currNode->left;
    return currNode;
}

AVLNode* _remove_AVL(AVLNode* node,int value){
    if(node==NULL)
        return node;
    if(value > node->data)
        node->right=_remove_AVL(node->right,value);
    else if(value < node->data)
        node->left=_remove_AVL(node->left,value);
    else{
        AVLNode *temp;
        if((node->left==NULL)||(node->right==NULL)){
            temp=NULL;
            if(node->left==NULL) temp=node->right;
            else if(node->right==NULL) temp=node->left;

            if(temp==NULL){
                temp=node;
                node=NULL;
            }
            else
                *node=*temp;

            free(temp);
        }
        else{
            temp = _findMinNode(node->right);
            node->data=temp->data;
            node->right=_remove_AVL(node->right,temp->data);
        }
    }
```

```
    if(node==NULL) return node;

    node->height=_max(_getHeight(node->left),_getHeight(node->right))+1;

    int balanceFactor= _getBalanceFactor(node);

    if(balanceFactor>1 && _getBalanceFactor(node->left)>=0)
        return _leftCaseRotate(node);

    if(balanceFactor>1 && _getBalanceFactor(node->left)<0)
        return _leftRightCaseRotate(node);

    if(balanceFactor<-1 && _getBalanceFactor(node->right)<=0)
        return _rightCaseRotate(node);

    if(balanceFactor<-1 && _getBalanceFactor(node->right)>0)
        return _rightLeftCaseRotate(node);

    return node;
}

void avl_remove(AVL *avl,int value) {
    if(avl_find(avl,value)) {
        avl->_root=_remove_AVL(avl->_root,value);
        avl->_size--;
    }
}

void verticalSumUtil(AVLNode* node, int hd,map<int, int> &Map) {
    // Base case
    if (node == NULL) return;

    // Recur for left subtree
    verticalSumUtil(node->left, hd-1, Map);

    // Add val of current node to
    // map entry of corresponding hd
    Map[hd] += node->data;

    // Recur for right subtree
    verticalSumUtil(node->right, hd+1, Map);
}

void verticalSum(AVLNode *root)
{
    // a map to store sum of nodes for each
    // horizontal distance
    map < int, int> Map;
```

```
    map < int, int> :: iterator it;
    int total = 0;

    // populate the map
    verticalSumUtil(root, 0, Map);

    // Prints the values stored by VerticalSumUtil()
    for (it = Map.begin(); it != Map.end(); ++it)
    {
        total += pow(it->second, 2);
    }
    cout << total << endl;
}


int main() {
    AVL avlku;
    avl_init(&avlku);

    string input;
    int value;

    do {
        getline(cin, input);
        if (input == "HITUNG") {
            break;
        } else {
            string word = input.substr(0, input.find(' '));
            string number = input.substr(input.find(' ') + 1);
            int angka = stoi(number);
            avl_insert(&avlku, angka);
        }
    } while (input != "HITUNG");
    verticalSum(avlku._root);
}
```

Penjelasan:

- Kode ini merupakan implementasi dari pohon AVL (Adelson-Velskii and Landis), yang merupakan jenis pohon biner yang seimbang. Pohon AVL memastikan bahwa perbedaan ketinggian antara anak sub-pohon kiri dan kanan dari setiap simpul tidak lebih dari satu. Pohon AVL menggabungkan manfaat pohon pencarian biner dengan operasi efisien untuk memasukkan, mencari, dan menghapus elemen-elemen dalam pohon.
- Kode dimulai dengan pendefinisian struktur data untuk simpul AVL (AVLNode) dan pohon AVL (AVL). Setiap simpul AVL memiliki nilai data, pointer ke simpul anak kiri dan kanan, dan tinggi simpul tersebut dalam pohon. Pohon AVL memiliki pointer

ke akar (_root) dan variabel unsigned integer _size untuk menyimpan jumlah elemen dalam pohon.

- Fungsi avl_init digunakan untuk menginisialisasi pohon AVL dengan mengatur pointer akar menjadi NULL dan mengatur ukuran pohon menjadi 0.
- Fungsi _getHeight digunakan untuk mengembalikan tinggi simpul yang diberikan. Jika simpul adalah NULL, maka tingginya adalah 0.
- Fungsi _max digunakan untuk mengembalikan nilai maksimum antara dua bilangan.
- Fungsi _getBalanceFactor digunakan untuk menghitung faktor keseimbangan simpul yang diberikan. Faktor keseimbangan didefinisikan sebagai selisih antara tinggi anak sub-pohon kiri dan tinggi anak sub-pohon kanan.
- Fungsi _rightRotate dan _leftRotate digunakan untuk melakukan rotasi simpul kanan dan kiri pada pohon AVL. Rotasi digunakan untuk mempertahankan keseimbangan pohon setelah operasi penyisipan atau penghapusan.
- Fungsi _leftCaseRotate, _rightCaseRotate, _leftRightCaseRotate, dan _rightLeftCaseRotate digunakan untuk menangani kasus rotasi khusus yang terjadi saat pohon AVL tidak seimbang setelah penyisipan atau penghapusan.
- Fungsi _avl_createNode digunakan untuk membuat simpul baru dengan nilai yang diberikan.
- Fungsi _search digunakan untuk mencari simpul dengan nilai yang diberikan dalam pohon AVL.
- Fungsi _insert_AVL digunakan untuk menyisipkan simpul baru dengan nilai yang diberikan ke dalam pohon AVL. Fungsi ini menggunakan rekursi untuk mencari posisi yang tepat untuk menyisipkan simpul baru dan kemudian melakukan rotasi jika diperlukan untuk mempertahankan keseimbangan pohon.
- Fungsi avl_find digunakan untuk mencari apakah suatu nilai ada dalam pohon AVL.
- Fungsi avl_insert digunakan untuk menyisipkan nilai ke dalam pohon AVL jika nilai tersebut belum ada di dalamnya. Fungsi ini memanggil _insert_AVL dan meningkatkan ukuran pohon jika penyisipan berhasil.
- Fungsi _findMinNode digunakan untuk mencari simpul dengan nilai minimum dalam sub-pohon yang diberikan.
- Fungsi _remove_AVL digunakan untuk menghapus simpul dengan nilai yang diberikan dari pohon AVL. Fungsi ini menggunakan rekursi untuk mencari simpul yang akan dihapus, dan kemudian melakukan rotasi jika diperlukan untuk mempertahankan keseimbangan pohon.
- Fungsi _remove_AVL memiliki tiga kasus utama untuk menghapus simpul:
    - Jika simpul yang akan dihapus adalah simpul daun atau simpul dengan satu anak, maka simpul tersebut dapat dihapus langsung dengan mengubah pointer yang sesuai.
    - Jika simpul yang akan dihapus memiliki dua anak, maka kita mencari nilai minimum dari sub-pohon kanan simpul tersebut. Nilai minimum ini akan menggantikan nilai simpul yang akan dihapus. Kemudian, kita secara rekursif menghapus nilai minimum tersebut dari sub-pohon kanan.
    - Setelah menghapus simpul, kita perlu memperbarui tinggi dari simpul yang dipengaruhi dan melakukan rotasi jika diperlukan untuk mempertahankan

keseimbangan pohon. Fungsi _getBalanceFactor digunakan untuk memeriksa faktor keseimbangan simpul dan menentukan jenis rotasi yang diperlukan.

- Setelah itu, terdapat fungsi avl_remove yang digunakan untuk menghapus nilai dari pohon AVL. Fungsi ini memeriksa apakah nilai yang akan dihapus ada dalam pohon sebelum memanggil _remove_AVL untuk menghapusnya. Jika nilai ditemukan, _remove_AVL dipanggil dan ukuran pohon dikurangi.
- Fungsi verticalSumUtil digunakan untuk menghitung jumlah vertikal dari simpul-simpul dalam pohon AVL. Fungsi ini melakukan rekursi secara in-order melalui pohon, mengakumulasi nilai simpul pada setiap tinggi horizontal (hd) menggunakan map. Setiap kali kita menemukan simpul dengan hd yang sama, kita menambahkan nilai simpul tersebut ke entri map yang sesuai.
- Fungsi verticalSum menggunakan verticalSumUtil untuk menghitung jumlah vertikal dari seluruh simpul dalam pohon AVL. Setelah mengumpulkan nilai-nilai vertikal dalam map, fungsi ini mengiterasi map dan menghitung total kuadrat dari semua nilai. Total ini dicetak sebagai hasil akhir.
- Fungsi main adalah fungsi utama yang digunakan untuk menguji pohon AVL. Pada awalnya, pohon AVL diinisialisasi dan kemudian input dari pengguna diterima. Jika input adalah "HITUNG", proses pengisian nilai selesai dan fungsi verticalSum dipanggil dengan akar pohon AVL sebagai argumen. Nilai vertikal dari pohon AVL dicetak sebagai hasil akhir.

2. CUCKER

Soal ini meminta untuk menghitung selisih tingkat pertahanan antara parent node dengan sibling parent suatu node yang telah dipilih. Apabila parent node tidak mempunyai sibling, maka program akan menampilkan tingkat pertahanan parent node itu sendiri.

```
Source Code:
#include<iostream>

using namespace std;

typedef struct AVLNode_t
{
    int data;
    struct AVLNode_t *left,*right;
    int height;
}AVLNode;

typedef struct AVL_t
{
    AVLNode *_root;
    unsigned int _size;
}AVL;

void avl_init(AVL *avl) {
    avl->_root = NULL;
```

```
    avl->_size = 0u;
}

int _getHeight(AVLNode* node){
    if(node==NULL)
        return 0;
    return node->height;
}

int _max(int a,int b){
    return (a > b)? a : b;
}

int _getBalanceFactor(AVLNode* node){
    if(node==NULL)
        return 0;
    return _getHeight(node->left)-_getHeight(node->right);
}

AVLNode* _rightRotate(AVLNode* pivotNode){
    AVLNode* newParrent=pivotNode->left;
    pivotNode->left=newParrent->right;
    newParrent->right=pivotNode;
    pivotNode->height=_max(_getHeight(pivotNode->left),_getHeight(pivotNode-
>right))+1;
    newParrent->height=_max(_getHeight(newParrent-
>left),_getHeight(newParrent->right))+1;
    return newParrent;
}

AVLNode* _leftCaseRotate(AVLNode* node){
    return _rightRotate(node);
}

AVLNode* _leftRotate(AVLNode* pivotNode){

    AVLNode* newParrent=pivotNode->right;
    pivotNode->right=newParrent->left;
    newParrent->left=pivotNode;

    pivotNode->height=_max(_getHeight(pivotNode->left),
                    _getHeight(pivotNode->right))+1;
    newParrent->height=_max(_getHeight(newParrent->left),
                    _getHeight(newParrent->right))+1;

    return newParrent;
}
```

```
AVLNode* _rightCaseRotate(AVLNode* node){
    return _leftRotate(node);
}

AVLNode* _leftRightCaseRotate(AVLNode* node){
    node->left=_leftRotate(node->left);
    return _rightRotate(node);
}

AVLNode* _rightLeftCaseRotate(AVLNode* node){
    node->right=_rightRotate(node->right);
    return _leftRotate(node);
}

AVLNode* _avl_createNode(int value) {
    AVLNode *newNode = (AVLNode*) malloc(sizeof(AVLNode));
    newNode->data = value;
    newNode->height=1;
    newNode->left = newNode->right = NULL;
    return newNode;
}

AVLNode* _search(AVLNode *root, int value) {
    while (root != NULL) {
        if (value < root->data)
            root = root->left;
        else if (value > root->data)
            root = root->right;
        else
            return root;
    }
    return root;
}

AVLNode* _insert_AVL(AVL *avl,AVLNode* node,int value){

    if(node==NULL)
        return _avl_createNode(value);
    if(value < node->data)
        node->left = _insert_AVL(avl,node->left,value);
    else if(value > node->data)
        node->right = _insert_AVL(avl,node->right,value);

    node->height= 1 + _max(_getHeight(node->left),_getHeight(node->right));

    int balanceFactor=_getBalanceFactor(node);

    if(balanceFactor > 1 && value < node->left->data)
```

```
        return _leftCaseRotate(node);
    if(balanceFactor > 1 && value > node->left->data)
        return _leftRightCaseRotate(node);
    if(balanceFactor < -1 && value > node->right->data)
        return _rightCaseRotate(node);
    if(balanceFactor < -1 && value < node->right->data)
        return _rightLeftCaseRotate(node);

    return node;
}

bool avl_find(AVL *avl, int value) {
    AVLNode *temp = _search(avl->_root, value);
    if (temp == NULL)
        return false;

    if (temp->data == value)
        return true;
    else
        return false;
}

void avl_insert(AVL *avl,int value){
    if(!avl_find(avl,value)){
        avl->_root = _insert_AVL(avl,avl->_root,value);
        avl->_size++;
    }
}

AVLNode* _findMinNode(AVLNode *node) {
    AVLNode *currNode = node;
    while (currNode && currNode->left != NULL)
        currNode = currNode->left;
    return currNode;
}

AVLNode* _remove_AVL(AVLNode* node,int value){
    if(node==NULL)
        return node;
    if(value > node->data)
        node->right=_remove_AVL(node->right,value);
    else if(value < node->data)
        node->left=_remove_AVL(node->left,value);
    else{
        AVLNode *temp;
        if((node->left==NULL)||(node->right==NULL)){
            temp=NULL;
            if(node->left==NULL) temp=node->right;
```

```
            else if(node->right==NULL) temp=node->left;

            if(temp==NULL){
                temp=node;
                node=NULL;
            }
            else
                *node=*temp;

            free(temp);
        }
        else{
            temp = _findMinNode(node->right);
            node->data=temp->data;
            node->right=_remove_AVL(node->right,temp->data);
        }
    }

    if(node==NULL) return node;

    node->height=_max(_getHeight(node->left),_getHeight(node->right))+1;

    int balanceFactor= _getBalanceFactor(node);

    if(balanceFactor>1 && _getBalanceFactor(node->left)>=0)
        return _leftCaseRotate(node);

    if(balanceFactor>1 && _getBalanceFactor(node->left)<0)
        return _leftRightCaseRotate(node);

    if(balanceFactor<-1 && _getBalanceFactor(node->right)<=0)
        return _rightCaseRotate(node);

    if(balanceFactor<-1 && _getBalanceFactor(node->right)>0)
        return _rightLeftCaseRotate(node);

    return node;
}

void avl_remove(AVL *avl,int value) {
    if(avl_find(avl,value)) {
        avl->_root=_remove_AVL(avl->_root,value);
        avl->_size--;
    }
}

void avl_inorder(AVLNode *root) {
    if (root) {
```

```
        avl_inorder(root->left);
        cout << root->data << " ";
        avl_inorder(root->right);
    }
}

AVLNode* findParent(AVLNode *root, AVLNode *node) {
    if(root == NULL) return NULL;
    if(root == node) return root; // root is parent of itself (root is the
root)
    if(root->left == node || root->right == node) return root;
    if(node->data < root->data) return findParent(root->left, node);
    else return findParent(root->right, node);
}

AVLNode* findSibling(AVLNode *root, AVLNode *node) {
    AVLNode *parent = findParent(root, node);
    if (parent == NULL) {
        return NULL;
    }

    if (parent == node) {
        return parent;
    } else if (parent->left == node) {
        return parent->right;
    } else {
        return parent->left;
    }
}

int main() {
    AVL avlku;
    avl_init(&avlku);
    int n, t;
    cin >> n >> t;
    for(int i=0;i<n;i++){
        int x;
        cin >> x;
        avl_insert(&avlku,x);
    }
    for(int i=0;i<t;i++){
        int x, y, z;
        cin >> x;
        y = findParent(avlku._root, _search(avlku._root, x))->data;
        z = findSibling(avlku._root, _search(avlku._root, y))->data;
        if ( y > z) {
            cout << y - z << endl;
        } else if (z > y) {
```

```
            cout << z - y << endl;
        } else if (y == z) {
            cout << y << endl;
        }
    }

    return 0;
}
```

Penjelasan:

- Struct AVLNode_t dan AVL_t:
    - AVLNode_t adalah struktur yang merepresentasikan simpul dalam pohon AVL. Setiap simpul memiliki data (integer), pointer ke simpul anak kiri dan anak kanan, serta tinggi simpul tersebut dalam pohon.
    - AVL_t adalah struktur yang merepresentasikan pohon AVL secara keseluruhan. Struktur ini menyimpan pointer ke simpul akar dan jumlah elemen dalam pohon.
- Fungsi avl_init:
    - Fungsi ini menginisialisasi pohon AVL dengan mengatur pointer akar menjadi NULL dan jumlah elemen menjadi 0.
- Fungsi bantuan untuk operasi pada pohon AVL:
    - _getHeight: Fungsi ini mengembalikan tinggi (height) simpul yang diberikan.
    - _max: Fungsi ini mengembalikan nilai maksimum antara dua bilangan.
    - _getBalanceFactor: Fungsi ini mengembalikan faktor keseimbangan (balance factor) simpul yang diberikan.
    - _rightRotate, _leftCaseRotate, _leftRotate, _rightCaseRotate, _leftRightCaseRotate, _rightLeftCaseRotate: Fungsi-fungsi ini melakukan rotasi simpul dalam pohon AVL untuk memperbaiki keseimbangan saat melakukan operasi penambahan atau penghapusan simpul.
    - _avl_createNode: Fungsi ini membuat simpul baru dengan nilai yang diberikan dan mengembalikan pointer ke simpul tersebut.
    - _search: Fungsi ini mencari simpul dengan nilai yang diberikan dalam pohon AVL dan mengembalikan pointer ke simpul tersebut.
    - _insert_AVL: Fungsi ini melakukan operasi penambahan simpul dengan nilai yang diberikan ke dalam pohon AVL, mempertahankan sifat AVL setelah penambahan.
    - avl_find: Fungsi ini mencari apakah sebuah nilai ada dalam pohon AVL.
    - avl_insert: Fungsi ini memasukkan nilai ke dalam pohon AVL jika nilai tersebut belum ada.
    - _findMinNode: Fungsi ini mencari simpul dengan nilai minimum dalam subpohon yang diberikan.
    - _remove_AVL: Fungsi ini melakukan operasi penghapusan simpul dengan nilai yang diberikan dari pohon AVL, mempertahankan sifat AVL setelah penghapusan.

    - avl_remove: Fungsi ini menghapus sebuah nilai dari pohon AVL jika nilai tersebut ada.
- Fungsi avl_inorder:
    - Fungsi ini mencetak isi pohon AVL secara in-order, yaitu dengan urutan kiri-akar-kanan.
- Fungsi findParent dan findSibling:
    - findParent: Fungsi ini mencari dan mengembalikan pointer ke simpul parent dari simpul yang diberikan dalam pohon AVL.
    - findSibling: Fungsi ini mencari dan mengembalikan pointer ke simpul sibling (simpul dengan parent yang sama) dari simpul yang diberikan dalam pohon AVL
- Fungsi main:
    - Pada awalnya, kita mendeklarasikan sebuah objek avlku dengan tipe AVL, yang akan digunakan untuk menyimpan pohon AVL. Kemudian, kita memanggil fungsi avl_init untuk menginisialisasi objek tersebut dengan mengatur nilai _root menjadi NULL dan _size menjadi 0.
    - Selanjutnya, kita membaca dua angka dari input menggunakan cin >> n >> t;. Angka pertama (n) adalah jumlah elemen yang akan dimasukkan ke dalam pohon AVL, dan angka kedua (t) adalah jumlah pertanyaan yang akan dijawab.
    - Setelah itu, kita menggunakan loop for untuk memasukkan elemen-elemen ke dalam pohon AVL. Dalam setiap iterasi loop, kita membaca angka (x) dari input menggunakan cin >> x;, lalu memanggil fungsi avl_insert(&avlku, x) untuk memasukkan elemen tersebut ke dalam pohon AVL avlku.
    - Selanjutnya, kita menggunakan loop for untuk menjawab pertanyaan-pertanyaan. Dalam setiap iterasi loop, kita membaca angka (x) dari input menggunakan cin >> x;. Kemudian, kita mencari simpul y yang merupakan parent dari simpul dengan nilai x dalam pohon AVL menggunakan fungsi findParent. Selanjutnya, kita mencari simpul z yang merupakan sibling dari simpul y menggunakan fungsi findSibling.
    - Setelah mendapatkan nilai y dan z, kita membandingkannya untuk mencari selisih terkecil antara keduanya. Jika y lebih besar dari z, kita mencetak hasilnya dengan cout << y - z << endl;. Jika z lebih besar dari y, kita mencetak hasilnya dengan cout << z - y << endl;. Jika y dan z sama, kita mencetak y dengan cout << y << endl;.
    - Setelah menjawab semua pertanyaan, program selesai dan mengembalikan nilai 0.

3. MPD

Soal ini meminta untuk mengecek apakah inputan angka yang nantinya dimasukkan ke BST memengaruhi BST tersebut menjadi tidak stabil atau masih tetap stabil.

Source Code:

```
#include <iostream>

using namespace std;

int isBalance = 0;

typedef struct AVLNode_t
{
    int data;
    struct AVLNode_t *left,*right;
    int height;
}AVLNode;

typedef struct AVL_t
{
    AVLNode *_root;
    unsigned int _size;
}AVL;

void avl_init(AVL *avl) {
    avl->_root = NULL;
    avl->_size = 0u;
}

int _getHeight(AVLNode* node){
    if(node==NULL)
        return 0;
    return node->height;
}

int _max(int a,int b){
    return (a > b)? a : b;
}

int _getBalanceFactor(AVLNode* node){
    if(node==NULL)
        return 0;
    return _getHeight(node->left)-_getHeight(node->right);
}

AVLNode* _rightRotate(AVLNode* pivotNode){
    AVLNode* newParrent=pivotNode->left;
    pivotNode->left=newParrent->right;
    newParrent->right=pivotNode;
    pivotNode->height=_max(_getHeight(pivotNode->left),_getHeight(pivotNode-
>right))+1;
    newParrent->height=_max(_getHeight(newParrent-
>left),_getHeight(newParrent->right))+1;
```

```
    return newParrent;
}

AVLNode* _leftCaseRotate(AVLNode* node){
    return _rightRotate(node);
}

AVLNode* _leftRotate(AVLNode* pivotNode){

    AVLNode* newParrent=pivotNode->right;
    pivotNode->right=newParrent->left;
    newParrent->left=pivotNode;

    pivotNode->height=_max(_getHeight(pivotNode->left),
                    _getHeight(pivotNode->right))+1;
    newParrent->height=_max(_getHeight(newParrent->left),
                    _getHeight(newParrent->right))+1;

    return newParrent;
}

AVLNode* _rightCaseRotate(AVLNode* node){
    return _leftRotate(node);
}

AVLNode* _leftRightCaseRotate(AVLNode* node){
    node->left=_leftRotate(node->left);
    return _rightRotate(node);
}

AVLNode* _rightLeftCaseRotate(AVLNode* node){
    node->right=_rightRotate(node->right);
    return _leftRotate(node);
}

AVLNode* _avl_createNode(int value) {
    AVLNode *newNode = (AVLNode*) malloc(sizeof(AVLNode));
    newNode->data = value;
    newNode->height=1;
    newNode->left = newNode->right = NULL;
    return newNode;
}

AVLNode* _search(AVLNode *root, int value) {
    while (root != NULL) {
        if (value < root->data)
            root = root->left;
        else if (value > root->data)
```

```
            root = root->right;
        else
            return root;
    }
    return root;
}

AVLNode* _insert_AVL(AVL *avl,AVLNode* node,int value){

    if(node==NULL)
        return _avl_createNode(value);
    if(value < node->data)
        node->left = _insert_AVL(avl,node->left,value);
    else if(value > node->data)
        node->right = _insert_AVL(avl,node->right,value);

    node->height= 1 + _max(_getHeight(node->left),_getHeight(node->right));

    int balanceFactor=_getBalanceFactor(node);

    if(balanceFactor > 1 && value < node->left->data)
        return _leftCaseRotate(node);
    if(balanceFactor > 1 && value > node->left->data)
        return _leftRightCaseRotate(node);
    if(balanceFactor < -1 && value > node->right->data)
        return _rightCaseRotate(node);
    if(balanceFactor < -1 && value < node->right->data)
        return _rightLeftCaseRotate(node);

    return node;
}

bool avl_find(AVL *avl, int value) {
    AVLNode *temp = _search(avl->_root, value);
    if (temp == NULL)
        return false;

    if (temp->data == value)
        return true;
    else
        return false;
}

void avl_insert(AVL *avl,int value){
    if(!avl_find(avl,value)){
        avl->_root = _insert_AVL(avl,avl->_root,value);
        avl->_size++;
    }
```

```
}

AVLNode* _findMinNode(AVLNode *node) {
    AVLNode *currNode = node;
    while (currNode && currNode->left != NULL)
        currNode = currNode->left;
    return currNode;
}

AVLNode* _remove_AVL(AVLNode* node,int value){
    if(node==NULL)
        return node;
    if(value > node->data)
        node->right=_remove_AVL(node->right,value);
    else if(value < node->data)
        node->left=_remove_AVL(node->left,value);
    else{
        AVLNode *temp;
        if((node->left==NULL)||(node->right==NULL)){
            temp=NULL;
            if(node->left==NULL) temp=node->right;
            else if(node->right==NULL) temp=node->left;

            if(temp==NULL){
                temp=node;
                node=NULL;
            }
            else
                *node=*temp;

            free(temp);
        }
        else{
            temp = _findMinNode(node->right);
            node->data=temp->data;
            node->right=_remove_AVL(node->right,temp->data);
        }
    }

    if(node==NULL) return node;

    node->height=_max(_getHeight(node->left),_getHeight(node->right))+1;

    int balanceFactor= _getBalanceFactor(node);

    if(balanceFactor>1 && _getBalanceFactor(node->left)>=0)
        return _leftCaseRotate(node);
```

```
    if(balanceFactor>1 && _getBalanceFactor(node->left)<0)
        return _leftRightCaseRotate(node);

    if(balanceFactor<-1 && _getBalanceFactor(node->right)<=0)
        return _rightCaseRotate(node);

    if(balanceFactor<-1 && _getBalanceFactor(node->right)>0)
        return _rightLeftCaseRotate(node);

    return node;
}

void avl_remove(AVL *avl,int value) {
    if(avl_find(avl,value)) {
        avl->_root=_remove_AVL(avl->_root,value);
        avl->_size--;
    }
}

void avl_inorder(AVLNode *root) {
    if (root) {
        avl_inorder(root->left);
        cout << root->data << " ";
        avl_inorder(root->right);
    }
}

AVLNode* _insert_AVLnBalanced(AVL *avl,AVLNode* node,int value){

    if(node==NULL)
        return _avl_createNode(value);
    if(value < node->data)
        node->left = _insert_AVLnBalanced(avl,node->left,value);
    else if(value > node->data)
        node->right = _insert_AVLnBalanced(avl,node->right,value);

    node->height= 1 + _max(_getHeight(node->left),_getHeight(node->right));

    int balanceFactor=_getBalanceFactor(node);

    if(balanceFactor > 1 && value < node->left->data) {
        isBalance = 1;
        return _leftCaseRotate(node);
    }
    if(balanceFactor > 1 && value > node->left->data) {
        isBalance = 1;
        return _leftRightCaseRotate(node);
    }
```

```
    if(balanceFactor < -1 && value > node->right->data) {
        isBalance = 1;
        return _rightCaseRotate(node);
    }
    if(balanceFactor < -1 && value < node->right->data) {
        isBalance = 1;
        return _rightLeftCaseRotate(node);
    }
    return node;
}

void avl_insertnBalanced(AVL *avl,int value){
    if(!avl_find(avl,value)){
        avl->_root = _insert_AVLnBalanced(avl,avl->_root,value);
        avl->_size++;
    }
}

int main() {
    AVL avlku;
    avl_init(&avlku);
    int n, p;

    cin >> n;

    for(int i = 0; i < n; i++) {
        int x;
        cin >> x;
        avl_insert(&avlku, x);
    }
    cin >> p;
    avl_insertnBalanced(&avlku, p);
    if(isBalance == 1) {
        cout << "Tree tidak balance" << endl;
    } else {
        cout << "Tree tetap balance" << endl;
    }
}
```

Penjelasan:

- Pendefinisian Tipe Data
    - AVLNode_t: Struktur data untuk merepresentasikan node dalam AVL Tree. Berisi data integer, pointer ke node anak kiri dan kanan, serta tinggi node.
    - AVL_t: Struktur data untuk merepresentasikan AVL Tree secara keseluruhan. Berisi pointer ke akar pohon dan ukuran pohon.
- Fungsi avl_init

    - Fungsi ini menginisialisasi pohon AVL dengan mengatur pointer akar menjadi NULL dan ukuran menjadi 0.
- Fungsi-fungsi Utilitas
    - _getHeight: Fungsi ini mengembalikan tinggi (jumlah level) dari suatu node dalam pohon AVL. Jika node adalah NULL, maka tingginya adalah 0.
    - _max: Fungsi ini mengembalikan nilai maksimum antara dua bilangan.
    - _getBalanceFactor: Fungsi ini mengembalikan faktor keseimbangan dari suatu node dalam pohon AVL. Faktor keseimbangan didefinisikan sebagai selisih antara tinggi node anak kiri dan tinggi node anak kanan.
    - _rightRotate dan _leftRotate: Fungsi-fungsi ini digunakan untuk melakukan rotasi kanan dan rotasi kiri pada pohon AVL. Rotasi digunakan untuk menjaga keseimbangan pohon saat dilakukan operasi penambahan atau penghapusan node.
    - _leftCaseRotate, _rightCaseRotate, _leftRightCaseRotate, dan _rightLeftCaseRotate: Fungsi-fungsi ini adalah variasi rotasi yang digunakan untuk memperbaiki ketidakseimbangan khusus pada pohon AVL.
    - _avl_createNode: Fungsi ini digunakan untuk membuat node baru dengan data yang diberikan dan mengembalikan pointer ke node tersebut.
- Fungsi-fungsi Operasi Pohon AVL
    - _search: Fungsi ini mencari nilai yang diberikan dalam pohon AVL dan mengembalikan pointer ke node yang mengandung nilai tersebut. Jika nilai tidak ditemukan, maka mengembalikan NULL.
    - _insert_AVL: Fungsi ini digunakan untuk menyisipkan nilai baru ke dalam pohon AVL. Jika nilai sudah ada dalam pohon, tidak dilakukan penyisipan. Fungsi ini melakukan rotasi jika diperlukan untuk mempertahankan keseimbangan pohon setelah penyisipan.
    - avl_find: Fungsi ini memeriksa apakah suatu nilai ada dalam pohon AVL. Jika nilai ditemukan, mengembalikan true; jika tidak, mengembalikan false.
    - avl_insert: Fungsi ini memasukkan nilai baru ke dalam pohon AVL. Fungsi ini memanggil _insert_AVL dan meningkatkan ukuran pohon jika nilai berhasil ditambahkan.
    - _findMinNode: Fungsi ini mencari node dengan nilai terkecil dalam pohon AVL yang diberikan dan mengembalikan pointer ke node tersebut.
    - _remove_AVL: Fungsi ini menghapus nilai yang diberikan dari pohon AVL. Fungsi ini melakukan rotasi jika diperlukan untuk mempertahankan keseimbangan pohon setelah penghapusan.
    - avl_remove: Fungsi ini menghapus nilai dari pohon AVL. Fungsi ini memanggil _remove_AVL dan mengurangi ukuran pohon jika nilai berhasil dihapus.
    - avl_inorder: Fungsi ini melakukan penjelajahan inorder pada pohon AVL dan mencetak nilai node secara terurut.
- Fungsi main
    - Pertama, fungsi ini menginisialisasi pohon AVL dan membaca jumlah bilangan yang akan dimasukkan dari input pengguna.

- Kemudian, menggunakan perulangan, fungsi ini membaca bilangan-bilangan tersebut dan memasukkannya ke dalam pohon AVL menggunakan fungsi avl_insert.
- Setelah itu, fungsi ini membaca bilangan tambahan yang akan dimasukkan dan memanggil fungsi avl_insertnBalanced untuk memasukkannya ke dalam pohon AVL.
- Terakhir, fungsi ini memeriksa variabel isBalance dan mencetak pesan apakah pohon AVL masih seimbang atau tidak.

Kode tersebut adalah implementasi dasar dari pohon AVL dalam bahasa C++. Dalam penggunaannya, kode ini akan membentuk pohon AVL berdasarkan bilangan yang dimasukkan pengguna, memeriksa apakah pohon masih seimbang setelah penambahan bilangan tambahan, dan mencetak pesan sesuai hasilnya.

4. MTS

Soal ini meminta untuk membuat BST yang dapat diisi sendiri. Nantinya akan ada pilihan "buat" untuk membuat node baru, dan "cari" untuk mencari posisi node yang diinginkan.

```
Source Code:
#include<bits/stdc++.h>

using namespace std;

typedef struct AVLNode_t
{
    int data;
    struct AVLNode_t *left,*right;
    int height;
}AVLNode;

typedef struct AVL_t
{
    AVLNode *_root;
    unsigned int _size;
}AVL;

void avl_init(AVL *avl) {
    avl->_root = NULL;
    avl->_size = 0u;
}

int _getHeight(AVLNode* node){
    if(node==NULL)
        return 0;
    return node->height;
```

```
}

int _max(int a,int b){
    return (a > b)? a : b;
}

int _getBalanceFactor(AVLNode* node){
    if(node==NULL)
        return 0;
    return _getHeight(node->left)-_getHeight(node->right);
}

AVLNode* _rightRotate(AVLNode* pivotNode){
    AVLNode* newParrent=pivotNode->left;
    pivotNode->left=newParrent->right;
    newParrent->right=pivotNode;
    pivotNode->height=_max(_getHeight(pivotNode->left),_getHeight(pivotNode-
>right))+1;
    newParrent->height=_max(_getHeight(newParrent-
>left),_getHeight(newParrent->right))+1;
    return newParrent;
}

AVLNode* _leftCaseRotate(AVLNode* node){
    return _rightRotate(node);
}

AVLNode* _leftRotate(AVLNode* pivotNode){

    AVLNode* newParrent=pivotNode->right;
    pivotNode->right=newParrent->left;
    newParrent->left=pivotNode;

    pivotNode->height=_max(_getHeight(pivotNode->left),
                    _getHeight(pivotNode->right))+1;
    newParrent->height=_max(_getHeight(newParrent->left),
                    _getHeight(newParrent->right))+1;

    return newParrent;
}

AVLNode* _rightCaseRotate(AVLNode* node){
    return _leftRotate(node);
}

AVLNode* _leftRightCaseRotate(AVLNode* node){
    node->left=_leftRotate(node->left);
    return _rightRotate(node);
```

```
}

AVLNode* _rightLeftCaseRotate(AVLNode* node){
    node->right=_rightRotate(node->right);
    return _leftRotate(node);
}

AVLNode* _avl_createNode(int value) {
    AVLNode *newNode = (AVLNode*) malloc(sizeof(AVLNode));
    newNode->data = value;
    newNode->height=1;
    newNode->left = newNode->right = NULL;
    return newNode;
}

AVLNode* _search(AVLNode *root, int value) {
    while (root != NULL) {
        if (value < root->data)
            root = root->left;
        else if (value > root->data)
            root = root->right;
        else
            return root;
    }
    return root;
}

AVLNode* _insert_AVL(AVL *avl,AVLNode* node,int value){

    if(node==NULL)
        return _avl_createNode(value);
    if(value < node->data)
        node->left = _insert_AVL(avl,node->left,value);
    else if(value > node->data)
        node->right = _insert_AVL(avl,node->right,value);

    node->height= 1 + _max(_getHeight(node->left),_getHeight(node->right));

    int balanceFactor=_getBalanceFactor(node);

    if(balanceFactor > 1 && value < node->left->data)
        return _leftCaseRotate(node);
    if(balanceFactor > 1 && value > node->left->data)
        return _leftRightCaseRotate(node);
    if(balanceFactor < -1 && value > node->right->data)
        return _rightCaseRotate(node);
    if(balanceFactor < -1 && value < node->right->data)
        return _rightLeftCaseRotate(node);
```

```
    return node;
}

bool avl_find(AVL *avl, int value) {
    AVLNode *temp = _search(avl->_root, value);
    if (temp == NULL)
        return false;

    if (temp->data == value)
        return true;
    else
        return false;
}

void avl_insert(AVL *avl,int value){
    if(!avl_find(avl,value)){
        avl->_root = _insert_AVL(avl,avl->_root,value);
        avl->_size++;
    }
}

AVLNode* _findMinNode(AVLNode *node) {
    AVLNode *currNode = node;
    while (currNode && currNode->left != NULL)
        currNode = currNode->left;
    return currNode;
}

AVLNode* _remove_AVL(AVLNode* node,int value){
    if(node==NULL)
        return node;
    if(value > node->data)
        node->right=_remove_AVL(node->right,value);
    else if(value < node->data)
        node->left=_remove_AVL(node->left,value);
    else{
        AVLNode *temp;
        if((node->left==NULL)||(node->right==NULL)){
            temp=NULL;
            if(node->left==NULL) temp=node->right;
            else if(node->right==NULL) temp=node->left;

            if(temp==NULL){
                temp=node;
                node=NULL;
            }
            else
```

```
                *node=*temp;

            free(temp);
        }
        else{
            temp = _findMinNode(node->right);
            node->data=temp->data;
            node->right=_remove_AVL(node->right,temp->data);
        }
    }

    if(node==NULL) return node;

    node->height=_max(_getHeight(node->left),_getHeight(node->right))+1;

    int balanceFactor= _getBalanceFactor(node);

    if(balanceFactor>1 && _getBalanceFactor(node->left)>=0)
        return _leftCaseRotate(node);

    if(balanceFactor>1 && _getBalanceFactor(node->left)<0)
        return _leftRightCaseRotate(node);

    if(balanceFactor<-1 && _getBalanceFactor(node->right)<=0)
        return _rightCaseRotate(node);

    if(balanceFactor<-1 && _getBalanceFactor(node->right)>0)
        return _rightLeftCaseRotate(node);

    return node;
}

void avl_remove(AVL *avl,int value) {
    if(avl_find(avl,value)) {
        avl->_root=_remove_AVL(avl->_root,value);
        avl->_size--;
    }
}

void avl_inorder(AVLNode *root) {
    if (root) {
        avl_inorder(root->left);
        cout << root->data << " ";
        avl_inorder(root->right);
    }
}

void avl_postorder(AVLNode *root, vector<int> &temp) {
```

```
    if (root) {
        avl_postorder(root->left, temp);
        avl_postorder(root->right, temp);
        temp.push_back(root->data);
    }
}

int main() {
    AVL avlku;
    avl_init(&avlku);
    int t;
    cin >> t;
    for(int i = 0; i < t; i++) {
        string input;
        int value;
        cin >> input >> value;
        if(input == "buat") {
            avl_insert(&avlku, value);
        } else if(input == "cari") {
            if(avl_find(&avlku, value)) {
                vector<int> temp;
                avl_postorder(avlku._root, temp);
                for(int i = 0; i < temp.size(); i++) {
                    if(temp[i] == value) {
                        cout << "Ruangannya ada di urutan ke-" << temp.size()-
i << endl;
                        break;
                    }
                }
                //cout << "Ruangannya ada di urutan ke-" <<  << endl;
            } else {
                cout << "Lah, ruangannya mana?" << endl;
            }
        } else {
            cout << "Maksudnya gimana?" << endl;
        }
    }
}
```

Penjelasan:

- Pertama, kita mendefinisikan dua struktur data: AVLNode untuk merepresentasikan node dalam pohon AVL, dan AVL untuk merepresentasikan pohon AVL secara keseluruhan. Struktur AVLNode memiliki empat anggota: data untuk menyimpan nilai node, left dan right untuk menunjukkan anak kiri dan anak kanan dari node, dan height untuk menyimpan tinggi node tersebut. Struktur AVL memiliki dua anggota: _root untuk menunjukkan akar pohon AVL, dan _size untuk menyimpan jumlah node dalam pohon.
- Kemudian, terdapat beberapa fungsi utilitas yang digunakan dalam implementasi pohon AVL. Fungsi _getHeight digunakan untuk mengembalikan tinggi node. Fungsi _max digunakan

untuk mengembalikan nilai maksimum dari dua bilangan. Fungsi _getBalanceFactor menghitung faktor keseimbangan suatu node dengan mengurangi tinggi anak kanan dari tinggi anak kiri. Fungsi-fungsi _rightRotate dan _leftRotate digunakan untuk melakukan rotasi kanan dan rotasi kiri pada pohon AVL. Rotasi dilakukan untuk mempertahankan keseimbangan pohon setelah operasi penyisipan atau penghapusan. Fungsi-fungsi _leftCaseRotate, _rightCaseRotate, _leftRightCaseRotate, dan _rightLeftCaseRotate adalah variasi rotasi yang digunakan untuk memperbaiki ketidakseimbangan khusus pada pohon AVL. Fungsi _avl_createNode digunakan untuk membuat node baru dengan nilai yang diberikan.

- Selanjutnya, terdapat fungsi _search yang digunakan untuk mencari nilai tertentu dalam pohon AVL. Fungsi ini menggunakan perulangan untuk mencari node yang mengandung nilai yang dicari. Jika nilai ditemukan, fungsi mengembalikan pointer ke node tersebut. Jika tidak, fungsi mengembalikan NULL.
- Fungsi _insert_AVL digunakan untuk menyisipkan nilai baru ke dalam pohon AVL. Fungsi ini menggunakan rekursi untuk mencari posisi yang tepat untuk menyisipkan node baru. Setelah node baru disisipkan, fungsi memperbarui tinggi node dan memeriksa keseimbangan pohon. Jika pohon tidak seimbang, fungsi melakukan rotasi sesuai dengan kondisi yang ditemui.
- Fungsi avl_find digunakan untuk memeriksa apakah suatu nilai ada dalam pohon AVL. Fungsi ini memanggil fungsi _search dan mengembalikan nilai true jika nilai ditemukan, dan false jika tidak.
- Fungsi avl_insert digunakan untuk menyisipkan nilai baru ke dalam pohon AVL. Fungsi ini memanggil fungsi avl_find untuk memeriksa apakah nilai sudah ada dalam pohon sebelum disisipkan. Jika nilai belum ada, fungsi memanggil _insert_AVL untuk menyisipkannya.
- Fungsi _findMinNode digunakan untuk mencari node dengan nilai terkecil dalam pohon AVL. Fungsi ini digunakan saat melakukan penghapusan node dengan dua anak.
- Fungsi _remove_AVL digunakan untuk menghapus node dengan nilai tertentu dari pohon AVL. Fungsi ini menggunakan rekursi untuk mencari dan menghapus node yang sesuai. Setelah penghapusan dilakukan, fungsi memperbarui tinggi node dan memeriksa keseimbangan pohon. Jika pohon tidak seimbang, fungsi melakukan rotasi sesuai dengan kondisi yang ditemui.
- Fungsi avl_remove digunakan untuk menghapus nilai tertentu dari pohon AVL. Fungsi ini memanggil fungsi avl_find untuk memeriksa apakah nilai ada dalam pohon sebelum dilakukan penghapusan. Jika nilai ditemukan, fungsi memanggil _remove_AVL untuk menghapusnya.
- Fungsi avl_inorder digunakan untuk mencetak nilai-nilai dalam pohon AVL secara terurut. Fungsi ini melakukan penjelajahan inorder pada pohon dan mencetak nilai node.
- Fungsi avl_postorder digunakan untuk melakukan penjelajahan postorder pada pohon AVL dan menyimpan nilai-nilai node ke dalam vektor temp.
- Fungsi main adalah program utama yang melakukan interaksi dengan pengguna. Fungsi ini menginisialisasi pohon AVL dan membaca jumlah bilangan yang akan dimasukkan dari input pengguna. Selanjutnya, menggunakan perulangan, fungsi ini membaca bilangan-bilangan tersebut dan memasukkannya ke dalam pohon AVL menggunakan fungsi avl_insert. Jika input pengguna adalah "cari", fungsi mencari nilai yang diberikan dan mencetak posisi ruangan jika ditemukan. Jika input pengguna tidak valid, fungsi mencetak pesan kesalahan.