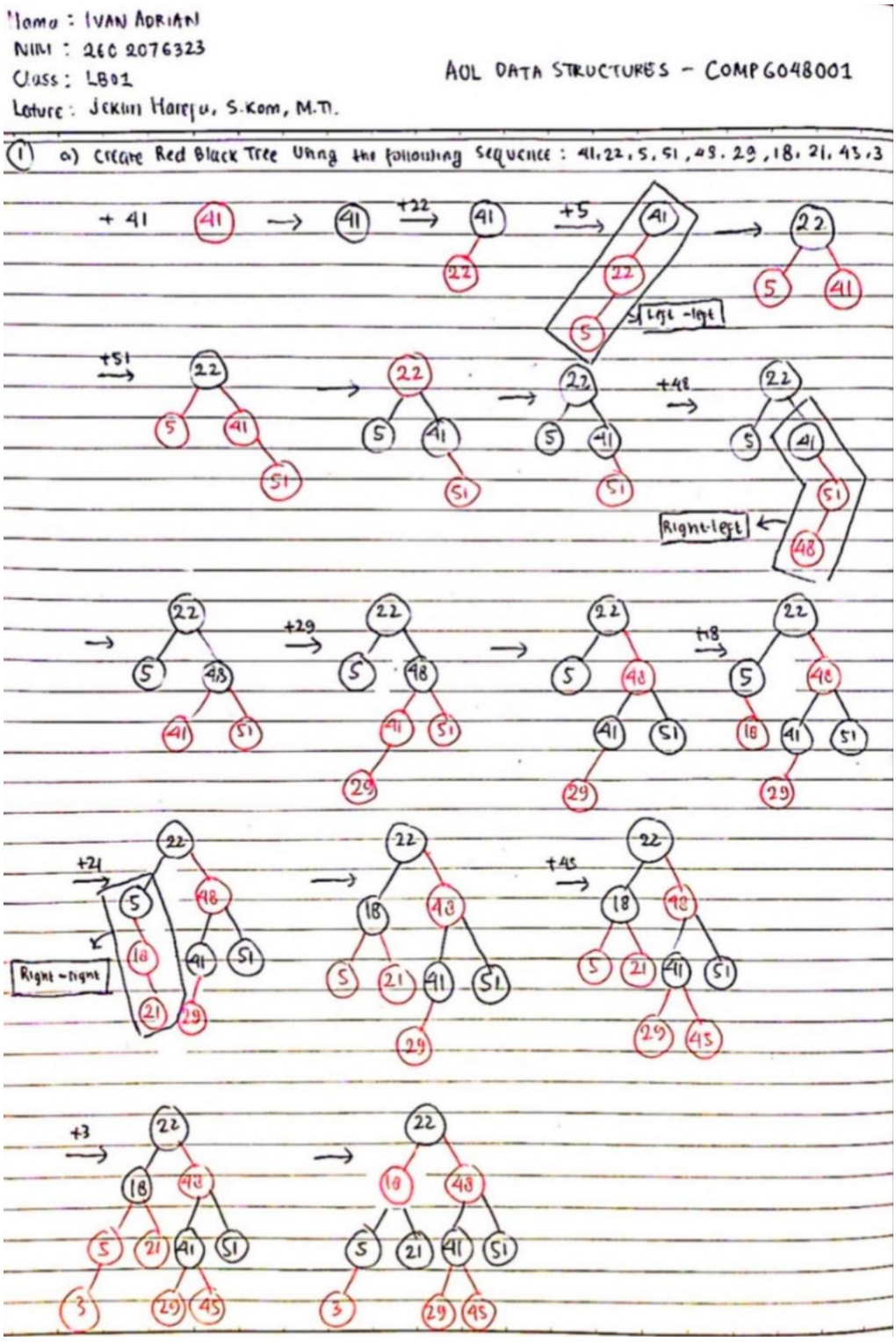
Nama : Ivan Adrian

NIM : 2602076323

Class : LB01

1. RED BLACK TREE

1a. Simulation:



1b. Implementation in C Pogram (Menggunakan VSCODE)

```
// RED BLACK TREE
// Nama : Ivan Adrian
// NIM : 2602076323
// Kelas : LB01

#include <stdio.h>
#include<string.h>
#include <stdlib.h>

enum Color{RED, BLACK};

struct Value{

    int value;
    Value* parent;
    Value* left;
    Value* right;
    enum Color redOrBlack;

};

void leftRotation(Value** root, Value* current){

    Value *childs = current->right;
    current->right = childs->left;

    if(childs->left != NULL){
        childs->left->parent = current;
    }

    childs->parent = current->parent;

    if(current->parent == NULL){
        *root = childs;
    }
    else if(current == current->parent->left){
        current->parent->left = childs;
    }
    else{
        current->parent->right = childs;
```

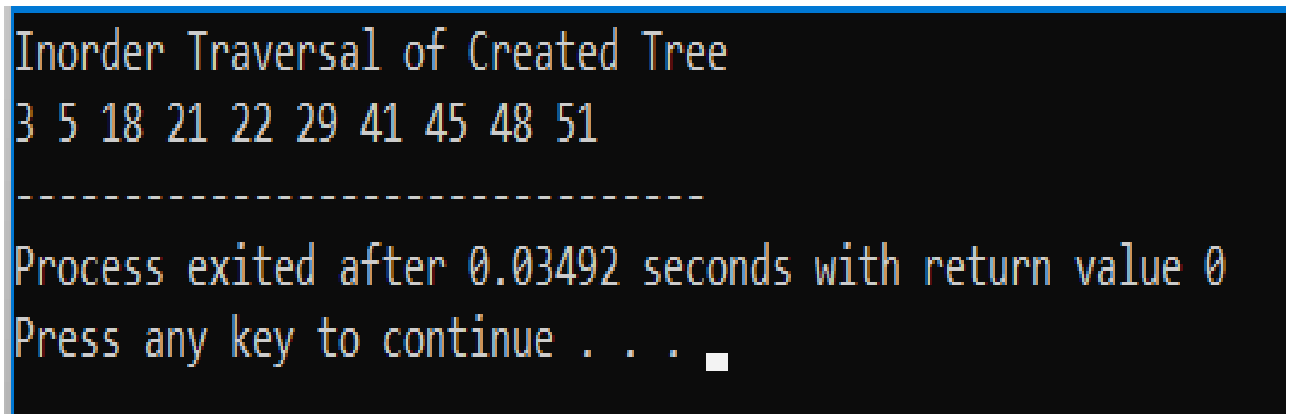
```
    }

    childs->left = current;
    current->parent = childs;
}

void rightRotation(Value** root, Value*current){

    Value *childs = current->left;
    current->left = childs->right;

    if(childs->right != NULL){
        childs->right->parent = current;
    }

    childs->parent = current->parent;

    if(current->parent == NULL){
        *root = childs;
    }
    else if(current == current->parent->right){
        current->parent->right = childs;
    }
    else{
        current->parent->left = childs;
    }

    childs->right = current;
    current->parent = childs;
}

void violateRepairation(Value** root, Value* current){

    while(current->parent != NULL && current->parent->redOrBlack ==
RED){
        if(current->parent == current->parent->parent->left){
            Value* uncle = current->parent->parent->right;
            if(uncle != NULL && uncle->redOrBlack == RED){
                current->parent->redOrBlack = BLACK;
                uncle->redOrBlack = BLACK;
                current->parent->parent->redOrBlack = RED;
                current = current->parent->parent;
```

```
            }
            else{
                if(current == current->parent->right){
                    current = current->parent;
                    leftRotation(root, current);
                }

                current->parent->redOrBlack = BLACK;
                current->parent->parent->redOrBlack = RED;
                rightRotation(root, current->parent->parent);
            }
        }
        else{
            Value* uncle = current->parent->parent->left;
            if(uncle != NULL && uncle->redOrBlack == RED){
                current->parent->redOrBlack = BLACK;
                uncle->redOrBlack = BLACK;
                current->parent->parent->redOrBlack = RED;
                current = current->parent->parent;
            }
            else{
                if(current == current->parent->left){
                    current = current->parent;
                    rightRotation(root, current);
                }

                current->parent->redOrBlack = BLACK;
                current->parent->parent->redOrBlack = RED;
                leftRotation(root, current->parent->parent);
            }
        }
    }

    (*root)->redOrBlack = BLACK;
}

void insertValue(Value** root, int value){

    Value *newItem = (Value*)malloc(sizeof(Value));

    newItem->value = value;
    newItem->parent = NULL;
```

```
    newItem->left = NULL;
    newItem->right = NULL;
    newItem->redOrBlack = RED;

    Value* current = *root;
    Value* parent = NULL;

    while(current != NULL){
        parent = current;
        if(newItem->value < current->value){
            current = current->left;
        }
        else{
            current = current->right;
        }
    }

    newItem->parent = parent;

    if(parent == NULL){
        *root = newItem;
    }
    else if(newItem->value < parent->value){
        parent->left = newItem;
    }
    else{
        parent->right = newItem;
    }

    violateRepairation(root, newItem);
}

void printInorder(Value *root){

    if (root == NULL){
        return;
    }
    else if(root != NULL){
        printInorder(root->left);
        printf("%d ", root->value);
        printInorder(root->right);
    }
```

```
}

int main() {
    Value *root = NULL;

    int value[] = {41, 22, 5, 51, 48, 29, 18, 21, 45, 3};
    int nValue = sizeof(value) / sizeof(value[0]);

    int i = 0;
    while(i < nValue){
        insertValue(&root, value[i]);
        i++;
    }

    puts("Inorder Traversal of Created Tree");
    printInorder(root);

    return 0;
}
```

Output:



Explanation of Code Red Black Tree:

```c
// RED BLACK TREE
// Nama : Ivan Adrian
// NIM : 2602076323
// Kelas : LB01

#include <stdio.h>
#include<string.h>
#include <stdlib.h>

enum Color{RED, BLACK};
// mendefinisikan dua warna RED atau BLACK

struct Value{

    int value;
    // menyimpan nilai dari simpul
    Value* parent;
    // menunjuk ke simpul induk dari simpul saat ini
    Value* left;
    // menunjuk ke simpul anak kiri dari simpul saat ini
    Value* right;
    // menunjuk ke simpul anak kanan dari simpul saat ini
    enum Color redOrBlack;
    // menyimpan informasi mengenai warna simpul

};
```

```c
void leftRotation(Value** root, Value* current){
    // melakukan rotasi ke kiri pada sebuah node

    Value *childs = current->right;
    // childs merupakan simpul baru setelah rotasi
    current->right = childs->left;
    // mengubah anak kanan dari simpul current menjadi anak kiri dari simpul childs

    if(childs->left != NULL){
    // jika anak kiri dari childs bukan NULL
        childs->left->parent = current;
        // mengubah pointer parent dari anak kiri tersebut menjadi simpul current
    }

    childs->parent = current->parent;
    // mengubah pointer parent dari childs menjadi pointer parent dari simpul current

    if(current->parent == NULL){
    // jika simpul current adalah akar (tidak memiliki parent)
        *root = childs;
        // maka simpul childs menjadi akar baru
    }
    else if(current == current->parent->left){
    // jika simpul current adalah simpul kiri dari simpul parentnya
        current->parent->left = childs;
        // mengubah pointer anak kiri dari parent simpul current menjadi childs
    }
```

```c
    else{
    // jika simpul current adalah simpul kanan dari simpul parentnya
        current->parent->right = childs;
        // mengubah pointer anak kanan dari parent simpul current menjadi childs
    }

    childs->left = current;
    // mengubah pointer anak kiri dari childs menjadi simpul current
    current->parent = childs;
    // mengubah pointer parent dari simpul current menjadi childs
}
```

```
67  void rightRotation(Value** root, Value*current){
68      // melakukan rotasi ke kanan pada sebuah node
69
70      Value *childs = current->left;
71      // childs merupakan simpul baru setelah rotasi
72      current->left = childs->right;
73      // mengubah anak kiri dari simpul current menjadi anak kanan dari simpul childs
74
75      if(childs->right != NULL){
76      // jika anak kanan dari childs bukan NULL
77          childs->right->parent = current;
78          // mengubah pointer parent dari anak kanan tersebut menjadi simpul current
79      }
80
81      childs->parent = current->parent;
82      // mengubah pointer parent dari childs menjadi pointer parent dari simpul current
83
84      if(current->parent == NULL){
85      // jika simpul current adalah akar (tidak memiliki parent)
86          *root = childs;
87          // maka simpul childs menjadi akar baru
88      }
89      else if(current == current->parent->right){
90      // jika simpul current adalah simpul kanan dari simpul parentnya
91          current->parent->right = childs;
92          // mengubah pointer anak kanan dari parent simpul current menjadi childs
93      }
```

```
94      else{
95      // jika simpul current adalah simpul kiri dari simpul parentnya
96          current->parent->left = childs;
97          // mengubah pointer anak kiri dari parent simpul current menjadi childs
98      }
99
100     childs->right = current;
101     // mengubah pointer anak kanan dari childs menjadi simpul current
102     current->parent = childs;
103     // mengubah pointer parent dari simpul current menjadi childs
104 }
```

```
106 void violateRepairation(Value** root, Value* current){
107     // memperbaiki pelanggaran aturan pada struktur RED BLACK TREE,
108     // setelah melalukan operasi penyisipan yang mungkin menyebabkan pelanggaran
109
110     while(current->parent != NULL && current->parent->redOrBlack == RED){
111     // selama simpul current memiliki parent dan parentnya memiliki warna RED
112     // berarti masih ada pelanggaran sifat-sifat Red-Black Tree yang perlu dibenahi
113         if(current->parent == current->parent->parent->left){
114         // memeriksa apakah parent adalah left child dari grandParent
115             Value* uncle = current->parent->parent->right;
116             // mendeklrasikan pointer uncle yang menunjuk ke right child dari grandParent
117             if(uncle != NULL && uncle->redOrBlack == RED){
118             // memerika apakah uncle tidak kosong dan berwarna merah
119                 current->parent->redOrBlack = BLACK;
120                 // mengubah warna parent menjadi BLACK
121                 uncle->redOrBlack = BLACK;
122                 // mengubah warna uncle menjadi BLACK
123                 current->parent->parent->redOrBlack = RED;
124                 // mengubah warna grandParent menjadi RED
125                 current = current->parent->parent;
126                 // menetapkan current menjadi grandParent untuk melanjutkan perbaikan lebih lanjut
127             }
```

```
128             else{
129             // jika kondisi di atas tidak terpenuhi
130             // artinya masih terjadi pelanggaran
131                 if(current == current->parent->right){
132                 // jika current adalah right child dari parent
133                     current = current->parent;
134                     // menetapkan current menjadi parent
135                     leftRotation(root, current);
136                     // melakukan rotasi ke kiri pada simpul parent
137                 }
138
139                 current->parent->redOrBlack = BLACK;
140                 // mengubah warna simpul parent menjadi BLACK
141                 current->parent->parent->redOrBlack = RED;
142                 // mengubah warna simpul grandParent menjadi RED
143                 rightRotation(root, current->parent->parent);
144                 // melakukan rotasi ke kanan pada simpul grandParent
145             }
146         }
```

```
        else{
        // jika kondisi di atas tidak terpenuhi
        // artinya masih terjadi pelanggaran
            Value* uncle = current->parent->parent->left;
            // mendeklrasikan pointer uncle yang menunjuk ke left child dari grandParent
            if(uncle != NULL && uncle->redOrBlack == RED){
            // memerika apakah uncle tidak kosong dan berwarna merah
                current->parent->redOrBlack = BLACK;
                // mengubah warna parent menjadi BLACK
                uncle->redOrBlack = BLACK;
                // mengubah warna uncle menjadi BLACK
                current->parent->parent->redOrBlack = RED;
                // mengubah warna grandParent menjadi RED
                current = current->parent->parent;
                // menetapkan current menjadi grandParent untuk melanjutkan perbaikan lebih lanjut
            }
```

```
            else{
            // jika kondisi di atas tidak terpenuhi
            // artinya masih terjadi pelanggaran
                if(current == current->parent->left){
                // jika current adalah left child dari parent
                    current = current->parent;
                    // menetapkan current menjadi parent
                    rightRotation(root, current);
                    // melakukan rotasi ke kanan pada simpul parent
                }

                current->parent->redOrBlack = BLACK;
                // mengubah warna simpul parent menjadi BLACK
                current->parent->parent->redOrBlack = RED;
                // mengubah warna simpul grandParent menjadi RED
                leftRotation(root, current->parent->parent);
                // melakukan rotasi ke kiri pada simpul grandParent
            }
        }
    }

    (*root)->redOrBlack = BLACK;
    // setelah perbaikan selesai dilakukan,
    // warna root diubah kembali menjadi BLACK
    // untuk memastikan bahwa properti Red-Black Tree Tetap terjaga
}
```

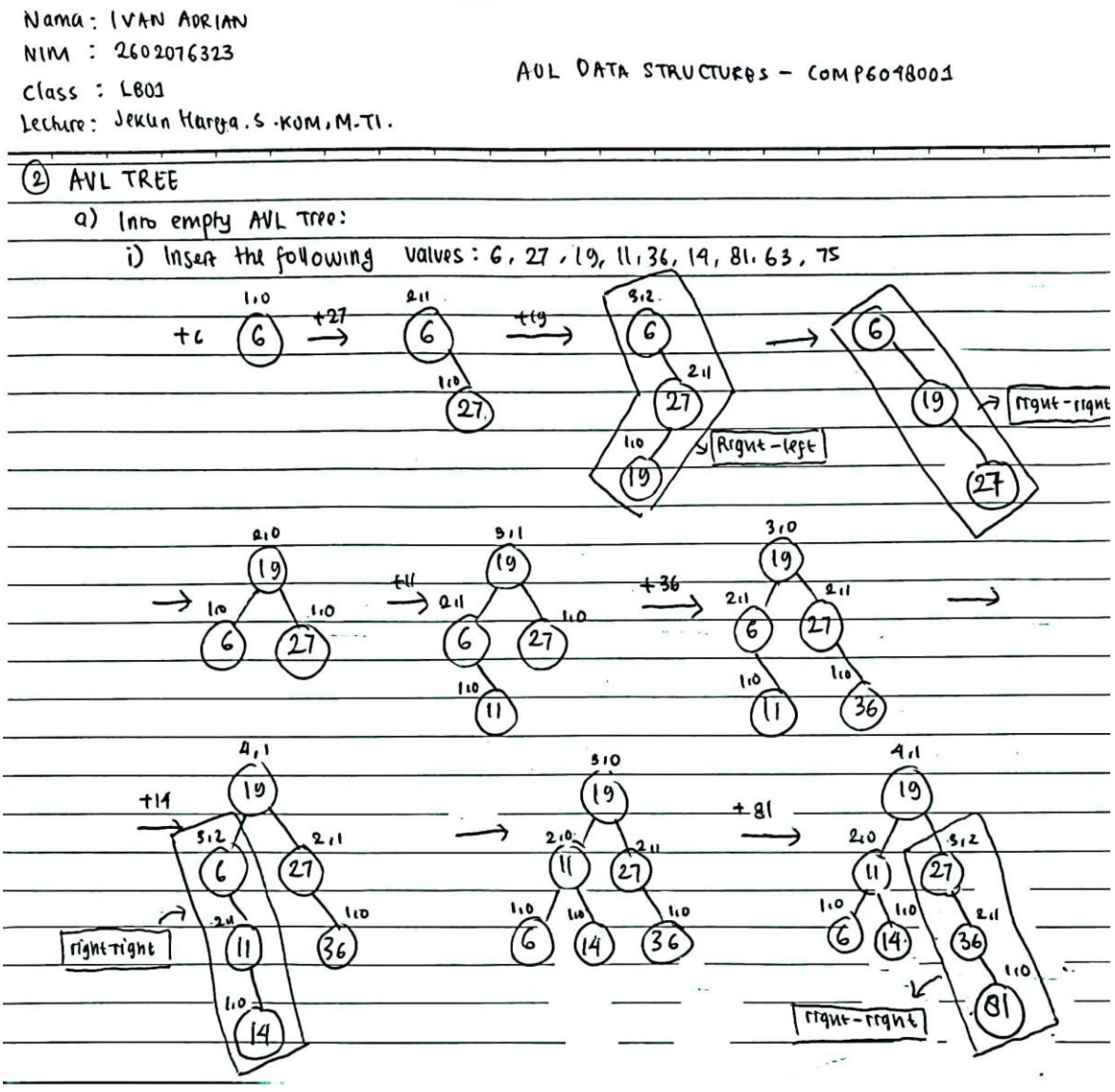
```
void insertValue(Value** root, int value){
    // fungsi untuk menyisipkan nilai baru ke dalam RBT

    Value *newItem = (Value*)malloc(sizeof(Value));
    // mengalokasikan memori yang diperlukan

    newItem->value = value;
    // menetapkan nilai value simpul baru sesuai dengan nilai yang ditentukan
    newItem->parent = NULL;
    // mengatur parent menjadi NULL
    newItem->left = NULL;
    // mengatur left child menjadi NULL
    newItem->right = NULL;
    // mengatur right child menjadi NULL
    newItem->redOrBlack = RED;
    // mengatur warna simpul baru menjadi RED sesuai aturan RBT

    Value* current = *root;
    // mengatur pointer current sebagai root
    Value* parent = NULL;
    // mengatur pointer parent menjadi NULL
```

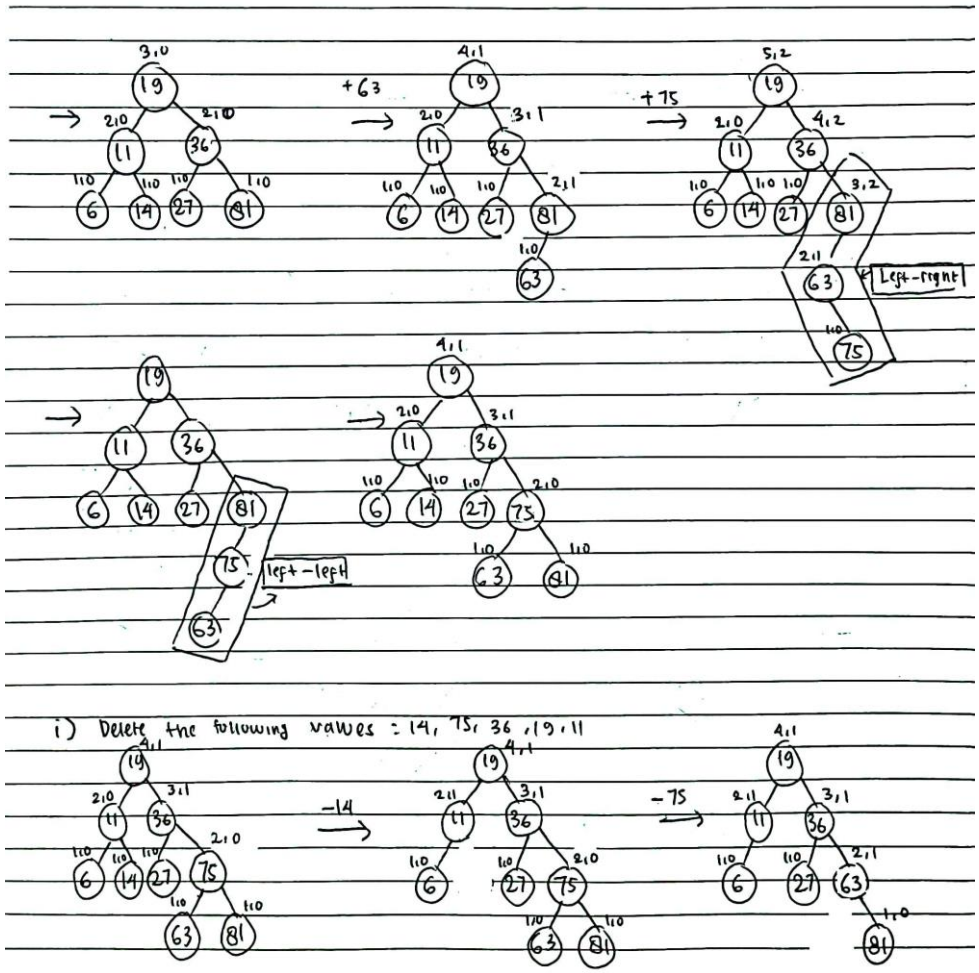
```
212     while(current != NULL){
213     // memulai loop while untuk mencari posisi yang tepat untuk menyisipkan simpul baru
214     // loop ini berjalan selama current tidak NULL.
215         parent = current;
216         // setiap kali melangkah ke simpul selanjutnya
217         // menyimpan simpul saat ini dalam parent
218         if(newItem->value < current->value){
219         // jika newItem->value lebih kecil dari nilai simpul saat ini
220             current = current->left;
221             // maka simpul berikutnya yang akan dijelajahi adalah anak kiri dari simpul saat ini
222         }
223         else{
224         // jika tidak,
225             current = current->right;
226             // simpul berikutnya adalah anak kanan
227         }
228     }
229
230     newItem->parent = parent;
231     // setelah mencapai posisi yang tepat untuk menyisipkan simpul baru
232     // mengatur pointer parent dari simpul baru
233     // sebagai simpul parent yang terakhir disimpan.

235     if(parent == NULL){
236     // jika parent NULL
237         *root = newItem;
238         // berarti simpul baru adalah simpul pertama dalam pohon
239         // dalam hal ini, mengatur root yang ditunjuk oleh pointer root ke simpul baru
240     }
241     else if(newItem->value < parent->value){
242     // jika newItem->value lebih kecil dari nilai simpul parent->value
243         parent->left = newItem;
244         // mengatur simpul baru sebagai anak kiri dari parent
245     }
246     else{
247     // Jika tidak,
248         parent->right = newItem;
249         // mengatur simpul baru sebagai anak kanan dari simpul parent
250     }
251
252     violateRepairation(root, newItem);
253     // setelah simpul baru disisipkan ke dalam pohon
254     // melalakukan perbaikan
255     // untuk memastikan bahwa sifat-sifat Red-Black Tree tetap terjaga
256 }

258 void printInorder(Value *root){
259     // mencetak nilai dari setiap node dalam struktur tree secara inorder
260
261     if (root == NULL){
262     // memerika apakah item adalah NULL
263     // yaitu jika mencapai lead atau mencetak subtree kosong
264         return;
265         // jika ya, maka fungsi akan mengembalikan (base case) dan keluar dari rekursi
266     }
267     else if(root != NULL){
268     // jika item tidak NULL
269         printInorder(root->left);
270         // memanggil rekursif fungsi printInorder untuk mencetak subtree kiri dari root
271         printf("%d ", root->value);
272         // mencetak nilai dari root saat ini
273         printInorder(root->right);
274         // memanggil rekursif fungsi printInorder untuk mencetak subtree kanan dari root
275     }
276
277 }
```

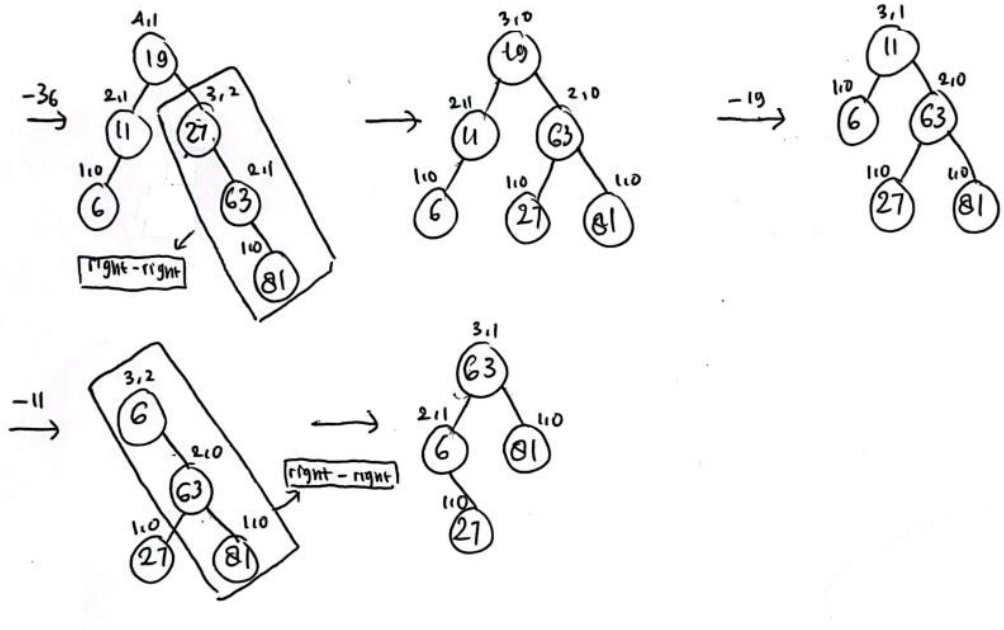
```c
int main() {
    Value *root = NULL;
    // inisialisasi root sebagai NULL

    int value[] = {41, 22, 5, 51, 48, 29, 18, 21, 45, 3};
    // mendefinisikan sebuah array value yang berisi beberapa nilai
    int nValue = sizeof(value) / sizeof(value[0]);
    // penghitungan jumlah elemen dalam array menggunakan sizeof agar lebih dinamis

    int i = 0;
    while(i < nValue){
    // perulangan while untuk memasukkan setiap nilai dari array
        insertValue(&root, value[i]);
        i++;
    }

    puts("Inorder Traversal of Created Tree");
    printInorder(root);
    // pencetakan hasil penelusuran inorder (inorder traversal) dari tree yang telah dibuat

    return 0;
}
```

2. AVL TREE

2a.simulation




+63
+75
left-right
left-left
i) Delete the following values : 14, 75, 36, 19, 11
-14
-75


-36
right-right
-19
-11
right-right

2b.Implementation in C program

```
// AVL TREE
// Nama : Ivan Adrian
// NIM : 2602076323
// Kelas : LB01

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<time.h>
#include<windows.h>

struct Value{

    int value;
    int height;
    int balanceFactor;

    Value* left;
    Value* right;

};

Value* createNode(int value){

    Value* newValue = (Value*) malloc(sizeof(Value));
    newValue->value = value;
    newValue->right = newValue->left = NULL;
    newValue->height = 1;

    return newValue;

}

int max(int getHeightLeft, int getHeightRight){

    if(getHeightLeft > getHeightRight){
        return getHeightLeft;
    }

    return getHeightRight;
```

```
}

int getHeight(Value* current){

    if(current == NULL){
        return 0;
    }

    return current->height;

}

int setBalanceFactor(Value* current){

    if(current == NULL){
        return 0;
    }

    return getHeight(current->left) - getHeight(current->right);
}

int setHeight(Value* current){

    if(current == NULL){
        return 0;
    }

    return max(getHeight(current->left), getHeight(current->right))
+ 1;

}

Value* updateNode(Value* current){

    if(current == NULL){
        return current;
    }

    current->height = setHeight(current);
    current->balanceFactor = setBalanceFactor(current);
```

```
    return current;

}

Value* leftRotation(Value* current){

    Value *pivot = current->right;
    Value *childLeft = pivot->left;
    pivot->left = current;
    current->right = childLeft;
    current = updateNode(current);
    pivot = updateNode(pivot);

    return pivot;
}

Value* rightRotation(Value* current){

    Value *pivot = current->left;
    Value *childRight = pivot->right;
    pivot->right = current;
    current->left = childRight;
    current = updateNode(current);
    pivot = updateNode(pivot);

    return pivot;
}

Value* rotation(Value* current){

    if(current == NULL){
        return current;
    }

    // LL (Left-Left Case)
    if(current->balanceFactor > 1 && current->left->balanceFactor >=
0){
        return rightRotation(current);
    }

    // RR (Right-Right Case)
```

```
    else if(current->balanceFactor < -1 && current->right->balanceFactor <= 0){
        return leftRotation(current);
    }

    // LR (Left-Right Case)
    else if(current->balanceFactor > 1 && current->left->balanceFactor < 0){
        current->left = leftRotation(current->left);
        return rightRotation(current);
    }

    // RL (Right-Left Case)
    else if(current->balanceFactor < -1 && current->right->balanceFactor > 0){
        current->right = rightRotation(current->right);
        return leftRotation(current);
    }

    return current;

}

Value* insertValue(Value* root, int value){

    if(root == NULL){
        return createNode(value);
    }
    else if(value < root->value){
        root->left = insertValue(root->left, value);
    }
    else if(value > root->value){
        root->right = insertValue(root->right, value);
    }

    return rotation(updateNode(root));

}

Value* searchValue(Value* root, int value){

    if(root == NULL || root->value == value){
```

```
        return root;
    }
    else if(value < root->value){
        return searchValue(root->left, value);
    }
    else{
        return searchValue(root->right, value);
    }

}

Value* deleteValue(Value* root, int deletedValue){

    if(root == NULL){
        return root;
    }
    else if(deletedValue < root->value){
        root->left = deleteValue(root->left, deletedValue);
    }
    else if(deletedValue > root->value){
        root->right = deleteValue(root->right, deletedValue);
    }
    else if(deletedValue == root->value) {
        if(root->left == NULL && root->right == NULL){
            // Node to delete has no children
            free(root);
            root = NULL;
        }
        else if(root->left != NULL && root->right == NULL){
            // Node to delete has only left child
            Value *temp = root->left;
            free(root);
            root = temp;
        }
        else if(root->left == NULL && root->right != NULL){
            // Node to delete has only right child
            Value *temp = root->right;
            free(root);
            root = temp;
        }
        else if(root->left != NULL && root->right != NULL){
            // Node to delete has both left and right children
```

```
            Value *temp = root->left;
            while (temp->right) {
                temp = temp->right;
            }
            root->value = temp->value;
            root->left = deleteValue(root->left, temp->value);
        }
    }

    return rotation(updateNode(root));

}

void preOrder(Value* root){

    if(root){
        printf("%d ", root->value);
        preOrder(root->left);
        preOrder(root->right);
    }

}

void inOrder(Value* root){

    if(root){
        inOrder(root->left);
        printf("%d ", root->value);
        inOrder(root->right);
    }

}

void postOrder(Value* root){

    if(root){
        postOrder(root->left);
        postOrder(root->right);
        printf("%d ", root->value);
    }

}
```

```c
void printTraversal(Value* root){

    printf("Preorder: ");
    preOrder(root);
    puts("");
    printf("Inorder: ");
    inOrder(root);
    puts("");
    printf("Postorder: ");
    postOrder(root);
    puts("");

}

void exitScreen(){

    system("cls");
    puts("Thank You");
    printf("Press Enter to exit");
    getchar();
    exit(0);

}

void menuList(){

    int inputNumber;
    Value *root = NULL;

    do{
        system("cls");
        puts("1. Insertion");
        puts("2. Deletion");
        puts("3. Traversal");
        puts("4. Exit");
        printf("Choose: ");

        scanf("%d", &inputNumber);
        getchar();

        int valueToInput;
```

```
        int valueToDelete;

        switch(inputNumber){
            case 1:
                printf("Insert: ");
                scanf("%d", &valueToInput);
                getchar();
                root = insertValue(root, valueToInput);
                printf("Value %d was inserted\n", valueToInput);
                break;
            case 2:
                printf("Delete: ");
                scanf("%d", &valueToDelete);
                getchar();

                if(searchValue(root, valueToDelete) != NULL){
                    root = deleteValue(root, valueToDelete);
                    puts("Data Found");
                    printf("Value %d was deleted\n",
valueToDelete);
                }
                else if(searchValue(root, valueToDelete) == NULL){
                    puts("Data not found");
                }
                root = deleteValue(root, valueToDelete);

                break;
            case 3:
                printTraversal(root);
                break;
            case 4:
                exitScreen();
                break;
        }
        puts("Press enter to continue...");
        getchar();
    }while(inputNumber >= 1 && inputNumber <= 4);

}

int main(){
```

```
    menuList();
    return 0;

}
```

i.Main Menu

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose:
```

ii.Insertion

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 6
Value 6 was inserted
```

Insert 6

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 27
Value 27 was inserted
```

Insert 27

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 19
Value 19 was inserted
```

Insert 19

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 11
Value 11 was inserted
```

Insert 11

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 36
Value 36 was inserted
```

Insert 36

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 14
Value 14 was inserted
```

Insert 14

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 81
Value 81 was inserted
```

Insert 81

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 63
Value 63 was inserted
```

Insert 63

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 1
Insert: 75
Value 75 was inserted
```

Insert 75

iii.Deletion

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 2
Delete: 14
Data Found
Value 14 was deleted
```

Delete 14

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 2
Delete: 75
Data Found
Value 75 was deleted
```

Delete 75

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 2
Delete: 36
Data Found
Value 36 was deleted
```

Delete 36

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 2
Delete: 19
Data Found
Value 19 was deleted
```

Delete 19

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 2
Delete: 11
Data Found
Value 11 was deleted
```

Delete 11

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 2
Delete: 7
Data not found
```

Data Not Found

iv.Traversal

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 3
Preorder: 19 11 6 14 36 27 75 63 81
Inorder: 6 11 14 19 27 36 63 75 81
Postorder: 6 14 11 27 63 81 75 36 19
```

After Insertion

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 3
Preorder: 63 6 27 81
Inorder: 6 27 63 81
Postorder: 27 6 81 63
```

After Deletion

v.Exit

```
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 4
Thank You
Press Enter to exit

--------------------------------
Process exited after 3.862 seconds with return value 0
Press any key to continue . . .
```

Explanation of Code AVL TREE:

```c
// AVL TREE
// Nama : Ivan Adrian
// NIM : 2602076323
// Kelas : LB01

#include<stdio.h>
#include<string.h>
#include<stdlib.h>
#include<time.h>
#include<windows.h>

struct Value{

    int value;
    int height;
    int balanceFactor;

    Value* left;
    Value* right;

};
```

```c
Value* createNode(int value){
    // Membuat simpul baru dengan nilai yang ditentukan

    Value* newValue = (Value*) malloc(sizeof(Value));
    // Mengalokasikan memori untuk struktur Value baru
    newValue->value = value;
    // Menetapkan nilai value simpul baru sesuai dengan nilai yang ditentukan
    newValue->right = newValue->left = NULL;
    // Mengatur pointer right dan left menjadi NULL, menandakan simpul baru tidak memiliki anak
    newValue->height = 1;
    // Mengatur ketinggian simpul baru menjadi 1, karena simpul ini adalah leaf node

    return newValue;
    // Mengembalikan pointer ke simpul baru yang telah dibuat

}
```

```c
int max(int getHeightLeft, int getHeightRight){
    // fungsi max menerima dua parameter bilangan yang akan dibandingkan

    if(getHeightLeft > getHeightRight){
    // memeriksa apakah nilai getHeightLeft lebih besar dari getHeightRight
    // jika iya, maka kembalikan nilai getHeightLeft sebagai nilai maximum
        return getHeightLeft;
    }

    return getHeightRight;
    // jika tidak, maka kembalikan nilai getHeightRight sebagai nilai maksimum

}

```

```
54  int getHeight(Value* current){
55      // Menghitung tinggi simpul (node)
56
57      if(current == NULL){
58          // Jika simpul saat ini adalah NULL,
59          // artinya tidak ada simpul,
60          // maka tinggi simpul tersebut adalah 0
61          return 0;
62      }
63
64      return current->height;
65      // Mengembalikan tinggi simpul saat ini
66
67  }
```

```
69  int setBalanceFactor(Value* current){
70      // Mengatur faktor keseimbangan simpul
71
72      if(current == NULL){
73          // Jika simpul saat ini adalah NULL, artinya tidak ada simpul, maka faktor keseimbangan adalah 0
74          return 0;
75      }
76
77      return getHeight(current->left) - getHeight(current->right);
78      // Mengembalikan selisih tinggi antara anak kiri dan anak kanan simpul saat ini sebagai faktor keseimbangan
79
80  }
```

```
82  int setHeight(Value* current){
83      // Mengatur tinggi simpul
84
85      if(current == NULL){
86          // Jika simpul saat ini adalah NULL, artinya tidak ada simpul, maka tinggi simpul tersebut adalah 0
87          return 0;
88      }
89
90      return max(getHeight(current->left), getHeight(current->right)) + 1;
91      // Mengembalikan tinggi maksimum antara anak kiri dan anak kanan simpul saat ini ditambah 1 sebagai tinggi simpul
92
93  }
```

```
95  Value* updateNode(Value* current){
96      // Memperbarui simpul dengan tinggi dan faktor keseimbangan yang terkini
97
98      if(current == NULL){
99          // Jika simpul saat ini adalah NULL, artinya tidak ada simpul, maka kembalikan simpul tersebut tanpa perubahan
100         return current;
101     }
102
103     current->height = setHeight(current);
104     // Mengatur tinggi simpul saat ini menggunakan fungsi setHeight
105     current->balanceFactor = setBalanceFactor(current);
106     // Mengatur faktor keseimbangan simpul saat ini menggunakan fungsi setBalanceFactor
107
108     return current;
109     // Mengembalikan simpul yang diperbarui
110
111 }
```

```
113 Value* leftRotation(Value* current){
114     // Melakukan rotasi ke kiri pada simpul saat ini
115 
116     Value *pivot = current->right;
117     // Simpan simpul anak kanan sebagai pivot
118     Value *childLeft = pivot->left;
119     // Simpan simpul anak kiri dari pivot
120     pivot->left = current;
121     // Jadikan simpul saat ini menjadi anak kiri dari pivot
122     current->right = childLeft;
123     // Jadikan anak kiri pivot menjadi anak kanan simpul saat ini
124     current = updateNode(current);
125     // Perbarui simpul saat ini
126     pivot = updateNode(pivot);
127     // Perbarui pivot
128 
129     return pivot;
130     // Kembalikan pivot sebagai simpul yang telah di-rotate
131 
132 }
```

```
134 Value* rightRotation(Value* current){
135     // Melakukan rotasi ke kanan pada simpul saat ini
136 
137     Value *pivot = current->left;
138     // Simpan simpul anak kiri sebagai pivot
139     Value *childRight = pivot->right;
140     // Simpan simpul anak kanan dari pivot
141     pivot->right = current;
142     // Jadikan simpul saat ini menjadi anak kanan dari pivot
143     current->left = childRight;
144     // Jadikan anak kanan pivot menjadi anak kiri simpul saat ini
145     current = updateNode(current);
146     // Perbarui simpul saat ini
147     pivot = updateNode(pivot);
148     // Perbarui pivot
149 
150     return pivot;
151     // Kembalikan pivot sebagai simpul yang telah di-rotate
152 
153 }
```

```
155 Value* rotation(Value* current){
156     // Fungsi untuk melakukan rotasi pada simpul saat ini berdasarkan faktor keseimbangan
157 
158     if(current == NULL){
159         // Jika simpul saat ini adalah NULL, artinya tidak ada simpul, maka kembalikan simpul tersebut tanpa perubahan
160         return current;
161     }
162 
163     // LL (Left-Left Case)
164     if(current->balanceFactor > 1 && current->left->balanceFactor >= 0){
165         // Jika faktor keseimbangan simpul saat ini lebih besar dari 1 (terlalu banyak simpul di anak kiri)
166         // dan faktor keseimbangan anak kiri tidak negatif (simpul lebih banyak di anak kiri dari anak kanan atau seimbang)
167         // maka lakukan rotasi ke kanan (rightRotation)
168         return rightRotation(current);
169     }
170 
171     // RR (Right-Right Case)
172     else if(current->balanceFactor < -1 && current->right->balanceFactor <= 0){
173         // Jika faktor keseimbangan simpul saat ini kurang dari -1 (terlalu banyak simpul di anak kanan)
174         // dan faktor keseimbangan anak kanan tidak positif (simpul lebih banyak di anak kanan dari anak kiri atau seimbang)
175         // maka lakukan rotasi ke kiri (leftRotation)
176         return leftRotation(current);
177     }
```

```
    // LR (Left-Right Case)
    else if(current->balanceFactor > 1 && current->left->balanceFactor < 0){
        // Jika faktor keseimbangan simpul saat ini lebih besar dari 1 (terlalu banyak simpul di anak kiri)
        // dan faktor keseimbangan anak kiri negatif (simpul lebih banyak di anak kanan dari anak kiri)
        // maka lakukan rotasi ke kiri (leftRotation) pada anak kiri dan kemudian rotasi ke kanan (rightRotation) pada simpul saat ini
        current->left = leftRotation(current->left);
        return rightRotation(current);
    }

    // RL (Right-Left Case)
    else if(current->balanceFactor < -1 && current->right->balanceFactor > 0){
        // Jika faktor keseimbangan simpul saat ini kurang dari -1 (terlalu banyak simpul di anak kanan)
        // dan faktor keseimbangan anak kanan positif (simpul lebih banyak di anak kiri dari anak kanan)
        // maka lakukan rotasi ke kanan (rightRotation) pada anak kanan dan kemudian rotasi ke kiri (leftRotation) pada simpul saat ini
        current->right = rightRotation(current->right);
        return leftRotation(current);
    }

    return current;
    // Jika tidak terjadi rotasi, kembalikan simpul saat ini tanpa perubahan

}
```

```
Value* insertValue(Value* root, int value){
    // Fungsi untuk menyisipkan nilai baru ke dalam pohon AVL

    if(root == NULL){
        // Jika pohon masih kosong, buat simpul baru dengan nilai tersebut
        return createNode(value);
    }
    else if(value < root->value){
        // Jika nilai yang akan disisipkan lebih kecil dari nilai simpul saat ini,
        // sisipkan nilai ke anak kiri simpul saat ini secara rekursif
        root->left = insertValue(root->left, value);
    }
    else if(value > root->value){
        // Jika nilai yang akan disisipkan lebih besar dari nilai simpul saat ini,
        // sisipkan nilai ke anak kanan simpul saat ini secara rekursif
        root->right = insertValue(root->right, value);
    }

    return rotation(updateNode(root));
    // Lakukan rotasi dan update keseimbangan setelah sisip nilai

}
```

```
Value* searchValue(Value* root, int value) {
    // Fungsi untuk mencari nilai tertentu dalam pohon AVL

    if(root == NULL || root->value == value){
        // Jika pohon kosong atau nilai simpul saat ini sama dengan nilai yang dicari, kembalikan simpul saat ini
        return root;
    }
    else if(value < root->value){
        // Jika nilai yang dicari lebih kecil dari nilai simpul saat ini,
        // Lanjutkan pencarian secara rekursif pada anak kiri simpul saat ini
        return searchValue(root->left, value);
    }
    else{
        // Jika nilai yang dicari lebih besar dari nilai simpul saat ini,
        // Lanjutkan pencarian secara rekursif pada anak kanan simpul saat ini
        return searchValue(root->right, value);
    }
}
```

```
244 Value* deleteValue(Value* root, int deletedValue) {
245     // Fungsi untuk menghapus simpul dengan nilai tertentu dari pohon AVL
246
247     if(root == NULL){
248         // Jika pohon kosong, kembalikan simpul saat ini
249         return root;
250     }
251     else if(deletedValue < root->value){
252         // Jika nilai yang akan dihapus lebih kecil dari nilai simpul saat ini,
253         // lanjutkan pencarian secara rekursif pada anak kiri simpul saat ini
254         root->left = deleteValue(root->left, deletedValue);
255     }
256     else if(deletedValue > root->value){
257         // Jika nilai yang akan dihapus lebih besar dari nilai simpul saat ini,
258         // lanjutkan pencarian secara rekursif pada anak kanan simpul saat ini
259         root->right = deleteValue(root->right, deletedValue);
260     }
261     else if(deletedValue == root->value) {
262         // Jika nilai yang akan dihapus sama dengan nilai simpul saat ini

264         if(root->left == NULL && root->right == NULL){
265             // Node to delete has no children
266             // Hapus simpul saat ini dari memori dan atur pointer root menjadi NULL
267             free(root);
268             root = NULL;
269         }
270         else if(root->left != NULL && root->right == NULL){
271             // Node to delete has only left child
272             // Simpan pointer ke anak kiri simpul saat ini
273             // Hapus simpul saat ini dari memori dan atur pointer root menjadi anak kiri
274             Value *temp = root->left;
275             free(root);
276             root = temp;
277         }
278         else if(root->left == NULL && root->right != NULL){
279             // Node to delete has only right child
280             // Simpan pointer ke anak kanan simpul saat ini
281             // Hapus simpul saat ini dari memori dan atur pointer root menjadi anak kanan
282             Value *temp = root->right;
283             free(root);
284             root = temp;
285         }

286         else if(root->left != NULL && root->right != NULL){
287             // Node to delete has both left and right children
288             // Cari nilai terbesar di anak kiri simpul saat ini sebagai pengganti simpul saat ini
289             // Ganti nilai simpul saat ini dengan nilai terbesar tersebut
290             // Hapus nilai terbesar tersebut dari anak kiri simpul saat ini
291             Value *temp = root->left;
292             while (temp->right) {
293                 temp = temp->right;
294             }
295             root->value = temp->value;
296             root->left = deleteValue(root->left, temp->value);
297         }
298     }
299
300     return rotation(updateNode(root));
301     // Lakukan rotasi dan pembaruan setelah operasi penghapusan
302 }
```

```c
void preOrder(Value* root){
    // Traversal PreOrder: mengunjungi simpul saat ini terlebih dahulu, kemudian anak kiri, dan anak kanan

    if(root){
        // Jika simpul saat ini tidak kosong
        printf("%d ", root->value);
        // Cetak nilai simpul saat ini
        preOrder(root->left);
        // Lakukan rekursi pada anak kiri
        preOrder(root->right);
        // Lakukan rekursi pada anak kanan
    }

}
```

```c
void inOrder(Value* root){
    // Traversal InOrder: mengunjungi anak kiri terlebih dahulu, kemudian simpul saat ini, dan anak kanan

    if(root){
        // Jika simpul saat ini tidak kosong
        inOrder(root->left);
        // Lakukan rekursi pada anak kiri
        printf("%d ", root->value);
        // Cetak nilai simpul saat ini
        inOrder(root->right);
        // Lakukan rekursi pada anak kanan
    }

}
```

```c
void postOrder(Value* root){
    // Traversal PostOrder: mengunjungi anak kiri terlebih dahulu, kemudian anak kanan, dan simpul saat ini terakhir

    if(root){
        // Jika simpul saat ini tidak kosong
        postOrder(root->left);
        // Lakukan rekursi pada anak kiri
        postOrder(root->right);
        // Lakukan rekursi pada anak kanan
        printf("%d ", root->value);
        // Cetak nilai simpul saat ini
    }

}
```

```c
void printTraversal(Value* root){

     printf("Preorder: ");
     preOrder(root);
     // Menampilkan hasil penelusuran Preorder
     puts("");
     printf("Inorder: ");
     inOrder(root);
     // Menampilkan hasil penelusuran Inorder
     puts("");
     printf("Postorder: ");
     postOrder(root);
     // Menampilkan hasil penelusuran Postorder
     puts("");

}
```

```c
void exitScreen(){

    system("cls");
    // Membersihkan layar konsol (tergantung pada sistem operasi)
    puts("Thank You");
    // Menampilkan teks "Thank You" di layar konsol
    printf("Press Enter to exit");
    // Menampilkan teks "Press Enter to exit" di layar konsol
    getchar();
    // Menunggu pengguna menekan tombol Enter
    exit(0);
    // Keluar dari program dengan status 0 (menandakan keluar dengan sukses)

}
```

```c
void menuList(){

    int inputNumber;
    Value *root = NULL; // deklrasi dan insialisasikan bahwa root awalnya adalah NULL

    do{
        system("cls");
        puts("1. Insertion");
        // Menampilkan pilihan untuk melakukan penambahan data
        puts("2. Deletion");
        // Menampilkan pilihan untuk melakukan penghapusan data
        puts("3. Traversal");
        // Menampilkan pilihan untuk melakukan menampilkan hasil traversal data
        puts("4. Exit");
        // Menampilkan pilihan untuk keluar dari program
        printf("Choose: ");
        // Menampilkan pesan untuk memilih opsi

        scanf("%d", &inputNumber);
        // Membaca input dari pengguna untuk memilih opsi
        getchar();
        // Menggunakan getchar() untuk membersihkan karakter newline (\n) dari input

        int valueToInput;
        int valueToDelete;

        switch(inputNumber){
            case 1:
                printf("Insert: ");
                // Meminta pengguna memasukkan nilai yang akan ditambahkan
                scanf("%d", &valueToInput);
                // Membaca nilai yang akan ditambahkan
                getchar();
                // Menggunakan getchar() untuk membersihkan karakter newline (\n) dari input
                root = insertValue(root, valueToInput);
                // Menambahkan nilai ke dalam pohon
                printf("Value %d was inserted\n", valueToInput);
                break;
            case 2:
                printf("Delete: ");
                // Meminta pengguna memasukkan nilai yang akan dihapus
                scanf("%d", &valueToDelete);
                // Membaca nilai yang akan dihapus
                getchar();
                // Menggunakan getchar() untuk membersihkan karakter newline (\n) dari input

                if(searchValue(root, valueToDelete) != NULL){
                    root = deleteValue(root, valueToDelete);
                    // Menghapus nilai dari pohon
                    puts("Data Found");
                    printf("Value %d was deleted\n", valueToDelete);
                }
                else if(searchValue(root, valueToDelete) == NULL){
                    puts("Data not found");
                    // Menampilkan pesan bahwa nilai tidak ditemukan
                }
                root = deleteValue(root, valueToDelete);

                break;
            case 3:
                printTraversal(root);
                break;
            case 4:
                exitScreen();
                // Keluar dari program
                break;
        }
        puts("Press enter to continue...");
        getchar();
    }while(inputNumber >= 1 && inputNumber <= 4);
    // Melakukan pengulangan selama inputNumber berada dalam rentang 1-4

}

int main(){

    menuList();
    // memanggil fungsi untuk menampilkan list menu
    return 0;

}
```