

**BACHELOR OF COMPUTER SCIENCE
FACULTY/SCHOOL OF SCHOOL OF COMPUTER SCIENCE
BINA NUSANTARA UNIVERSITY
JAKARTA**

ASSESSMENT FORM

Course: COMP6048001 - Data Structure

Method of Assessment: Case Study

Semester/Academic Year : 2/2022-2023

Name of Lecturer : Mayliana, S.Kom., M.T.I.

Date : Monday, June 5th, 2023

Class : LD01

Topic : Review II

Group Members :	<ol style="list-style-type: none">1. Fendy Wijaya2. Wilbert Yang3. Vincent Owen Bun4. Trinata Suryawan5. Bernard Owens Wiladjaja6. David
------------------------	---

Student Outcomes:

SO 2 - Mampu merancang, mengimplementasikan, dan mengevaluasi solusi berbasis komputasi untuk memenuhi serangkaian persyaratan komputasi dalam konteks ilmu computer

Able to design, implement, and evaluate a computing-based solution to meet a given set of computing requirements in the context of computer science

Learning Objectives:

LObj 2.2 - Mampu mengimplementasikan solusi berbasis komputasi untuk memenuhi serangkaian persyaratan komputasi tertentu dalam konteks ilmu computer

Able to implement a computing-based solution to meet a given set of computing requirements in the context of computer science

Learning Outcomes:

LO 3 - Apply data structures using C

No	Related LO-LOBJ-SO	Assessment criteria	Weight	Excellent (85 - 100)	Good (75-84)	Average (65-74)	Poor (0 - 64)	Score	(Score x Weight)
1	LO3-LObj 2.2 -SO 2	Ability to identify the problems to find the solution	10 %	Able to identify both the input and output in fully detail for the problem	Able to identify both the input and output but in less detail for the problem	Able to identify both the input and output but not in clear detail for the problem	Able to identify only the input or output for the problem		
2	LO3-LObj 2.2 -SO 2	Ability to design an algorithm for the problem	40 %	Able to design an algorithm for the problem in full detail	Able to design an algorithm but not have full detail	Able to design an algorithm for the problem but not have clear process	Able to design an algorithm but cannot be implemented in the problem		
3	LO3-LObj 2.2 -SO 2	Ability to solve the problem	50 %	Able to solve 76-100% the problem with fully functional feature	Able to solve 51-75% the problem and lack some features	Able to solve 26-50% of the problem and lack some features	Able to solve less than 25% of the problem		
		Total Score:	$\sum(\text{Score} \times \text{Weight})$						

Remarks:

ASSESSMENT METHOD

Instructions

- This case study is individual with 1 week processing
- Design an algorithm in pseudocode/code to solve the problem and write down the algorithm
- If in the case study does not have specific instruction about the algorithm that must be used, it means that students can determine the best algorithm to solve the given problem
- Report will be submitted in pdf format to binusmaya

Note for Lecturers:

- The lecture notifies this case study to the student from Week 1
- Deadline for the case study is one week after the lecture post it on binusmaya
- The student should submit the report to binusmaya no later than deadline
- If the student do plagiarism, their score for this case study will be zero

1. RED BLACK TREE

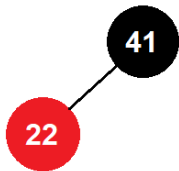
- a. (Bobot 10%, SO 2, LObj 2.2, LO 3) Create a Red Black Tree using the following sequence: 41,22,5,51,48,29,18,21,45,3

Answer:

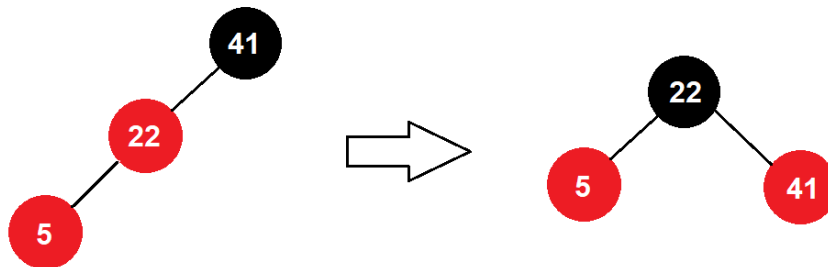
Insert 41



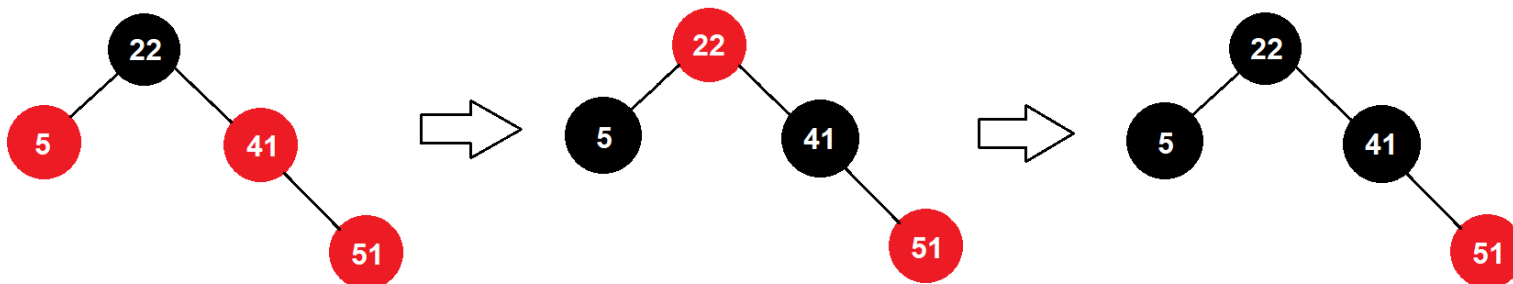
Insert 22



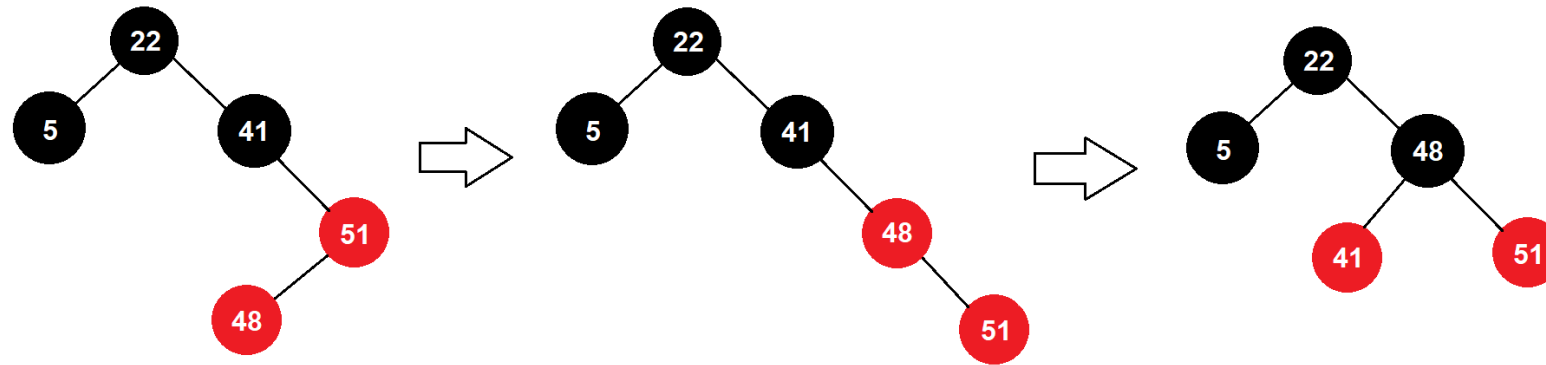
Insert 5



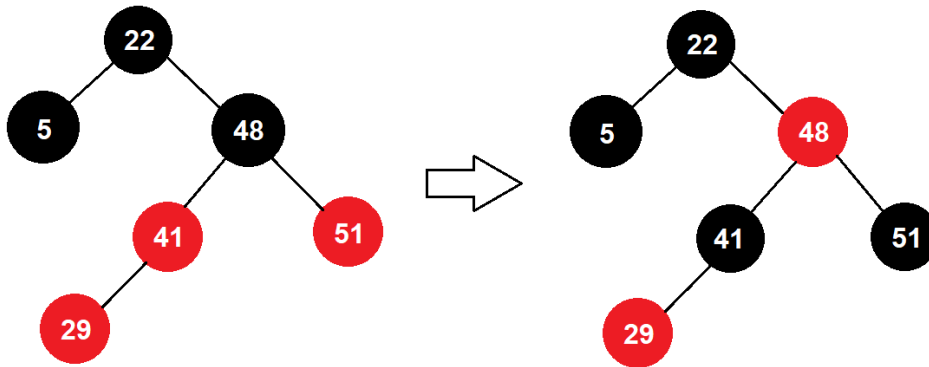
Insert 51



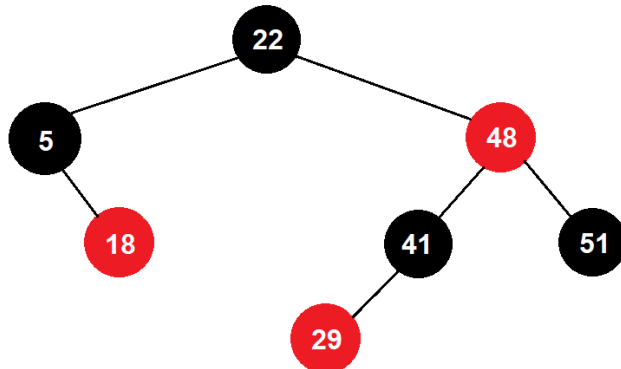
Insert 48



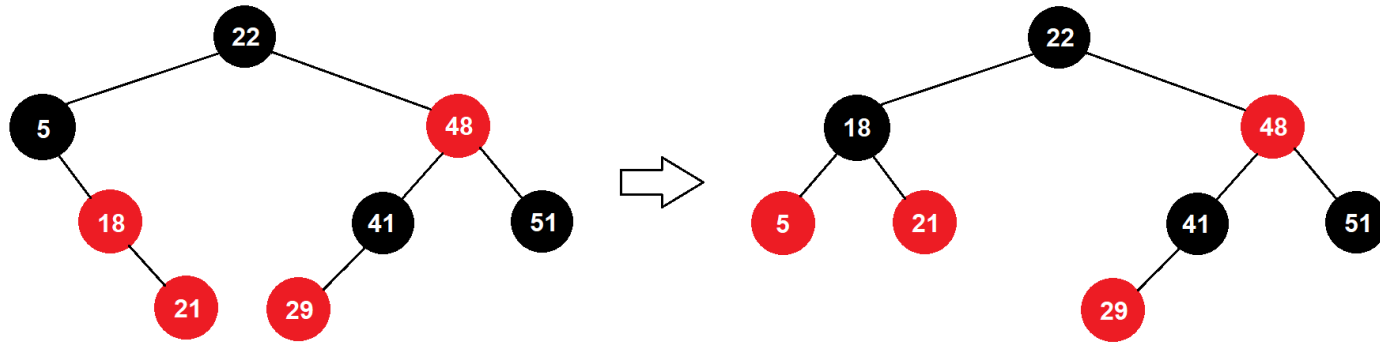
Insert 29



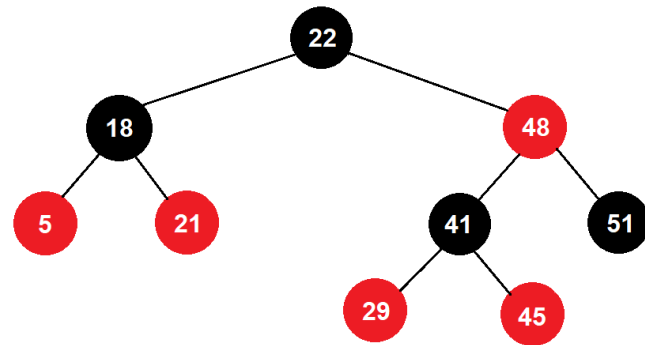
Insert 18



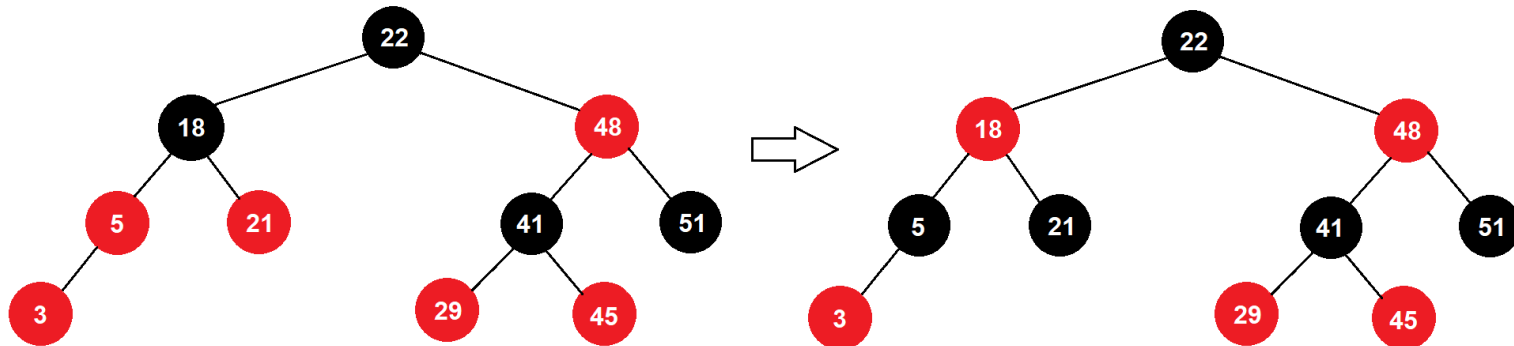
Insert 21



Insert 45



Insert 3



FINAL ANSWER

- b. (Bobot 20%, SO 2, LObj 2.2, LO 3) Try implementing (a) Red Black Tree insertion in C Program and print those datas with InOrder Traversal

```
Inoder Traversal of Created Tree
3 5 18 21 22 29 41 45 48 51
-----
Process exited after 0.1592 seconds with return value 0
Press any key to continue . . .
```

Answer:

Snippet of code is displayed below. For further exploration, we have already attached in this .zip file the .cpp program to execute the code.

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

struct RBT {
    int key;
    int color; //merah bervalue 1, hitam bervalue 0

    struct RBT* left;
    struct RBT* right;
    struct RBT* parent;
};

//fungsi untuk membuat node
struct RBT* createNode(int key) {
    struct RBT* newNode = (struct RBT*)malloc(sizeof(struct RBT));
    newNode->key = key;
    newNode->color = 1; //selalu berwarna merah (angka 1) saat pertama kali dibuat
```

```

//pointer diset ke NULL semua
newNode->parent = NULL;
newNode->left = NULL;
newNode->right = NULL;
//mengembalikan alamat dari node yang telah dibuat
return newNode;
}

//fungsi untuk melakukan left rotation
struct RBT* leftRotation(struct RBT* root, struct RBT* node) {
    struct RBT* rightNode = node->right; //anak kanan dari node saat ini akan disimpan dalam rightNode
    node->right = rightNode->left; // anak kanan dari node akan menyimpan anak kiri dari rightNode

    //jika anak kiri dari rightNode tidak kosong, maka akan dihubungkan dengan parentnya
    if (rightNode->left != NULL) {
        rightNode->left->parent = node;
    }

    //lalu menghubungkan rightNode dengan parent dari node
    rightNode->parent = node->parent;

    if (node->parent == NULL) {
        root = rightNode; /*jika jika parent dari node kosong, artinya node sekarang adalah root, maka
        rightNode yang akan menjadi root baru*/
    }
}

```



```

    } else if (node == node->parent->left) {
        node->parent->left = rightNode; /*jika node saat ini merupakan anak kiri dari parent, maka
dapat mengubah anak kiri parent menjadi rightNode*/
    } else {
        node->parent->right = rightNode; /*jika node kini adalah anak kanan dari parent, maka
mengubah anak kanan parent menjadi rightNode*/
    }
    /*kondisi saat ini berarti rightNode sekarang merupakan parent dari node, sehingga kita
menghubungkan node->parent ke rightNode, dan anak kiri rightNode menjadi node*/
    rightNode->left = node;
    node->parent = rightNode;

    //kemudian mengembalikan kembali alamat root untuk mengupdate rootnya
    return root;
}

//fungsi untuk melakukan right rotation
struct RBT* rightRotation(struct RBT* root, struct RBT* node) {
    struct RBT* leftNode = node->left; //anak kiri dari node disimpan dalam leftNode
    node->left = leftNode->right; //anak kanan dari leftNode akan menjadi anak kiri dari node

    /*jika anak kanan dari leftNode tidak kosong, anak kanan leftNode akan dihubungkan ke parent baru
yaitu node*/
    if (leftNode->right != NULL) {

```

```
    leftNode->right->parent = node;
}

//menghubungkan leftNode dengan parent dari node
leftNode->parent = node->parent;

if (node->parent == NULL) {
    root = leftNode; //jika parent node kosong, maka leftNode akan menjadi root baru
} else if (node == node->parent->left) {
    node->parent->left = leftNode; /*jika node berada di posisi anak kiri dari parent, maka
leftNode yang akan menggantikannya*/
} else {
    node->parent->right = leftNode; /*jika node berada di posisi anak kanan parent, maka leftNode
akan menjadi anak kanan baru dari parent*/
}

//sekarang leftNode menjadi parent dari node, sehingga diperlukan update sebagai berikut
leftNode->right = node; //node menjadi anak kanan dari leftNode
node->parent = leftNode; //parent dari node adalah leftNode

//mengupdate root
return root;
}
```

```

//fungsi untuk mengupdate tree, sehingga warna dan tree tidak memiliki violation sama sekali
struct RBT* update(struct RBT* root, struct RBT* node) {
    //akan mengulang proses jika node masih memiliki parent yang berwarna merah
    while (node->parent != NULL && node->parent->color == 1) {
        /*mengisi node uncle dengan mengecek apakah uncle tersebut merupakan anak kiri atau kanan
        dari grandparent (node->parent->parent)*/
        struct RBT* uncle = (node->parent->parent->right == node->parent)? node->parent->parent-
        >left : node->parent->parent->right;
        /*jika node saat ini memiliki uncle dan unclenya berwarna merah, maka hanya perlu mengupdate
        warna saja*/
        if (uncle != NULL && uncle->color == 1) {
            node->parent->color = 0;
            uncle->color = 0;
            node->parent->parent->color = 1;
            //grandparent akan menjadi node terbaru untuk menghilangkan violation yang masih ada
            node = node->parent->parent;
        }
        //jika parent node saat ini adalah anak kiri dari grandparent
        else if (node->parent == node->parent->parent->left) {
            /*dan jika node berada di anak kanan parent, hal ini menjadi left righ case dan
            membutuhkan double rotation*/
            if (node == node->parent->right) {
                node = node->parent;
                root = leftRotation(root, node);
            }
        }
    }
}

```

```

    }
    /*selanjutnya adalah left left case dengan melakukan rotasi dan mewarnai node untuk
menghilangkan violation*/
    node->parent->color = 0;
    node->parent->parent->color = 1;
    root = rightRotation(root, node->parent->parent);
}
//jika parent node saat ini adalah anak kanan dari grand parent
else {
    /*jika node saat ini adalah anak kiri dari parent, maka akan menjadi right left case,
diperlukan double rotation*/
    if (node == node->parent->left) {
        node = node->parent;
        root = rightRotation(root, node);
    }
    /*selanjutnya adalah right right case, melakukan rotasi kiri dan mewarnai kembali
node untuk menghilangkan violation*/
    node->parent->color = 0;
    node->parent->parent->color = 1;
    root = leftRotation(root, node->parent->parent);
}
}
root->color = 0; //untuk memastikan bahwa root selalu berwarna hitam
//mengupdate root

```

```

    return root;
}

/*fungsi insertion berikut masih mirip dengan cara insert pada BST, yang membedakan adalah tambahan
untuk mengupdate pointer anak curr menjadikan curr sebagai parent*/
struct RBT* insertion(struct RBT* curr, struct RBT* newNode) {
    if (curr == NULL) {
        return newNode;
    } else if (curr->key > newNode->key) {
        curr->left = insertion(curr->left, newNode);
        curr->left->parent = curr;
    } else if (curr->key < newNode->key) {
        curr->right = insertion(curr->right, newNode);
        curr->right->parent = curr;
    }
    return curr;
}

//fungsi ini untuk membungkus langkah insert dan update menjadi satu fungsi
struct RBT* insertAndUpdate(struct RBT* root, int key) {
    struct RBT* newNode = createNode(key);
    root = insertion(root, newNode);
    root = update(root, newNode);
    return root;
}

```

```

}

//fungsi untuk print tree secara inorder
void printInorder(struct RBT* node) {
    if (node != NULL) {
        printInorder(node->left);
        printf("%d ", node->key);
        printInorder(node->right);
    }
}

void testInorderColor(struct RBT* curr) {
    if (curr != NULL) {
        testInorderColor(curr->left);
        printf("%d %s\n", curr->key, (curr->color)?"red":"black");
        testInorderColor(curr->right);
    }
}

int main() {
    struct RBT* root = NULL;
    int keyList[] = {41, 22, 5, 51, 48, 29, 18, 21, 45, 3};
    int size = sizeof(keyList)/sizeof(int);
    for (int i = 0; i < size; i++) {

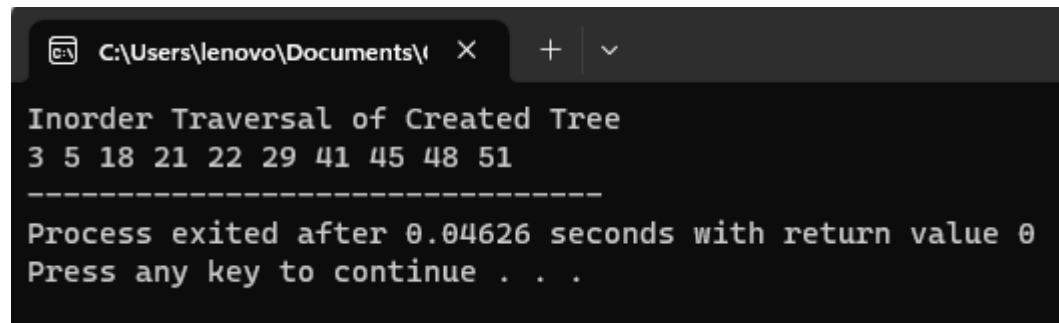
```

```

        root = insertAndUpdate(root, keyList[i]);
        root->color = 0; //memastikan kembali bahwa root berwarna hitam (0 -> hitam)
    }
    // printf("root %d\n", root->key);
    printf("Inorder Traversal of Created Tree\n");
    // testInorderColor(root);
    printInorder(root);
    return 0;
}

```

The output of the program will be shown by the picture below.



```

C:\Users\lenovo\Documents\
Inorder Traversal of Created Tree
3 5 18 21 22 29 41 45 48 51
-----
Process exited after 0.04626 seconds with return value 0
Press any key to continue . . .

```

Code can be executed at the separated file, titled “Nomor 1b (Red-Black Tree Implementation Code).cpp”. For better and clearer view of code documentation, it is suggested to open the .cpp file.

2. AVL TREE

a. Into empty AVL Tree:

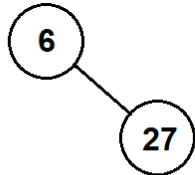
i. (Bobot 10%, SO 2, LObj 2.2, LO 3) Insert the following values: 6, 27, 19, 11, 36, 14, 81, 63, 75

Answer:

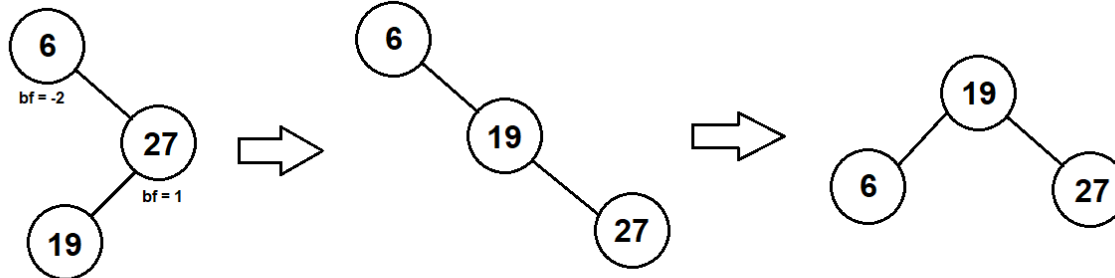
Insert 6



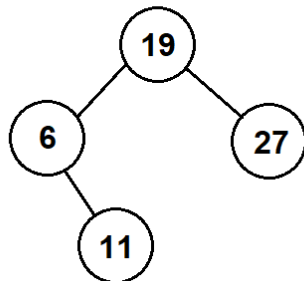
Insert 27



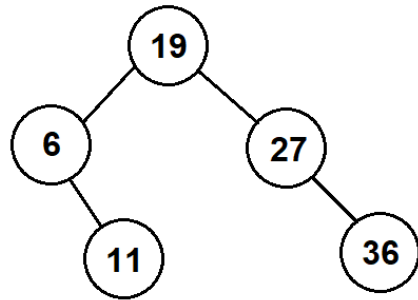
Insert 19



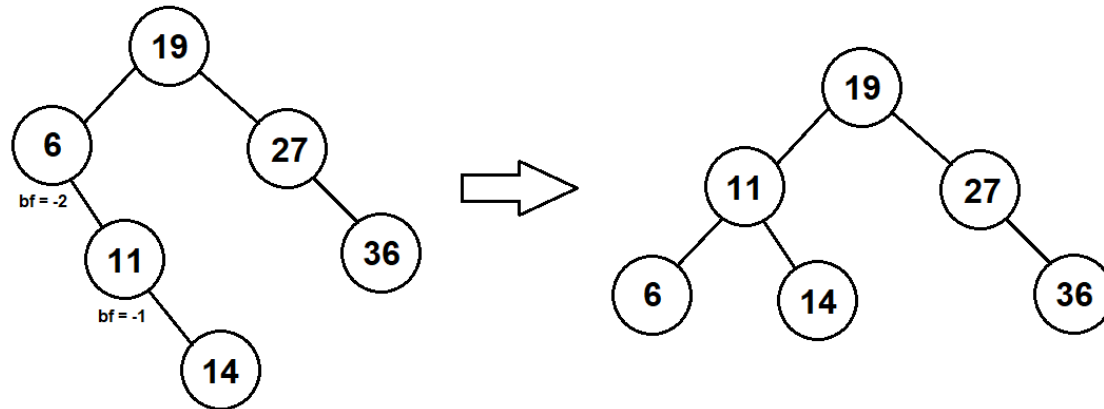
Insert 11



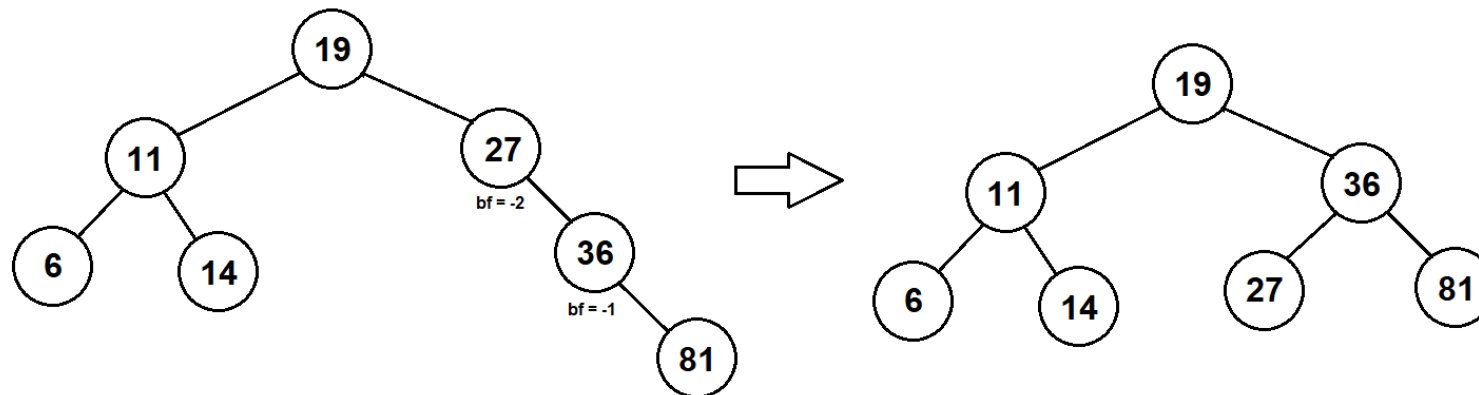
Insert 36



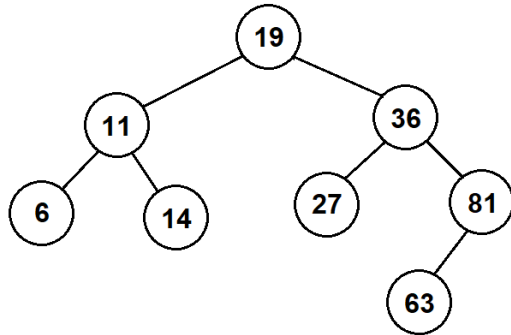
Insert 14



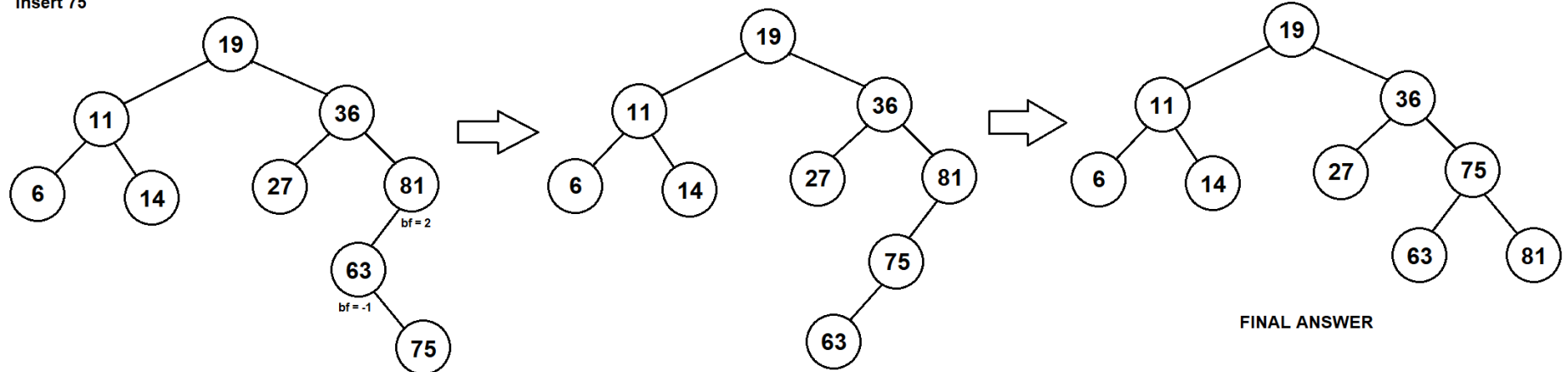
Insert 81



Insert 63



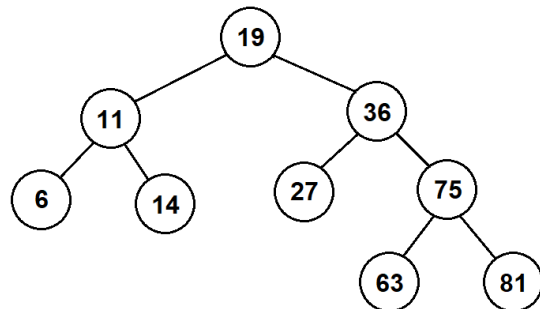
Insert 75



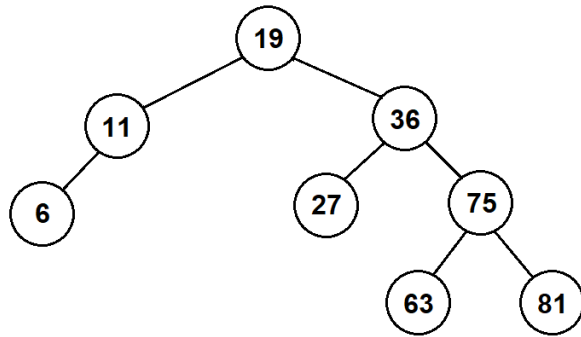
ii. (Bobot 10%, SO 2, LObj 2.2, LO 3) Delete the following values: 14, 75, 36, 19, 11

Answer:

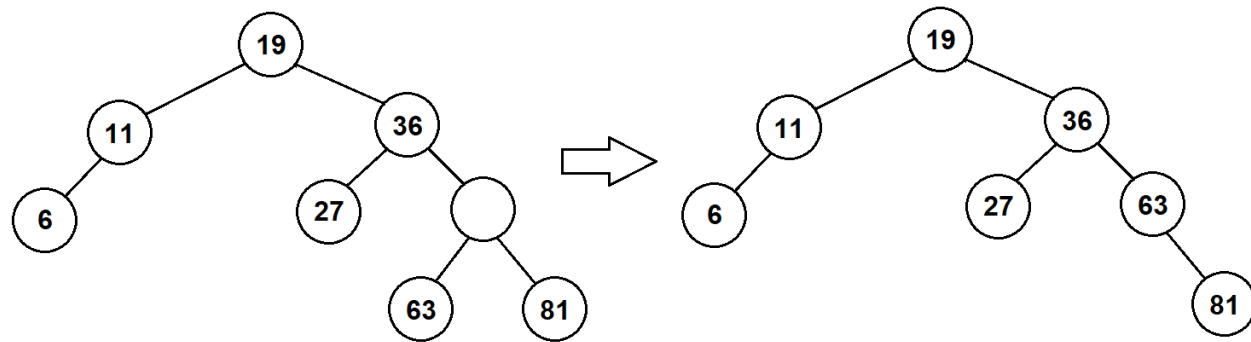
Initial condition before deletion



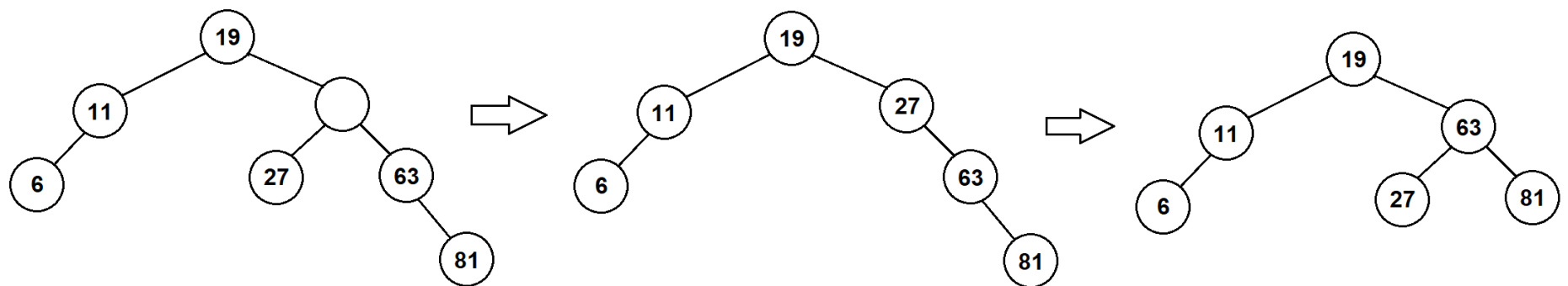
Delete 14



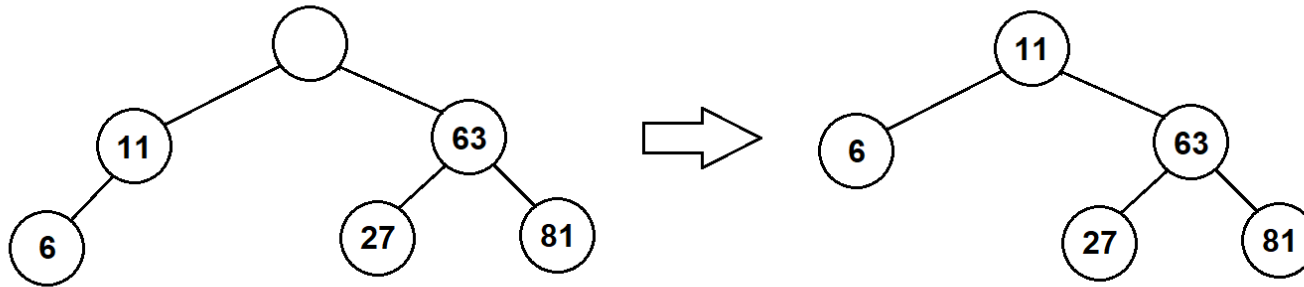
Delete 75



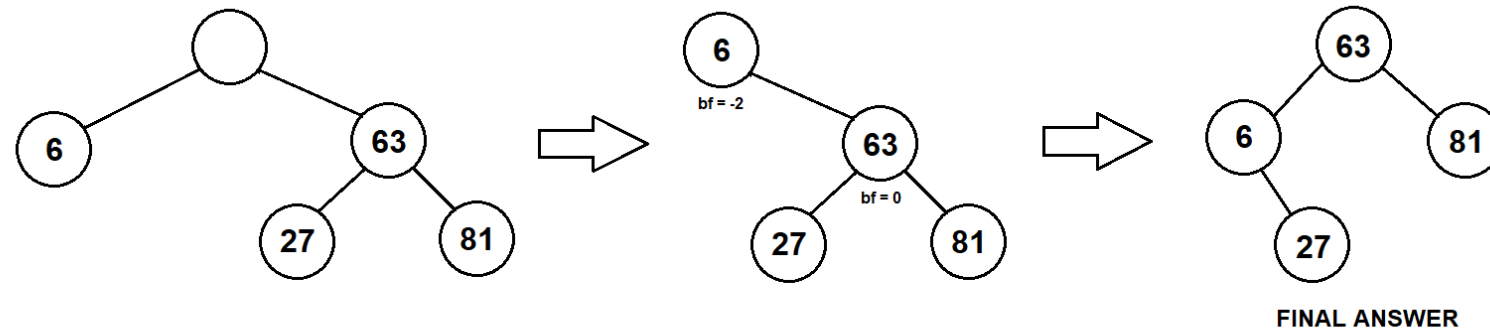
Delete 36



Delete 19



Delete 11



b. (Bobot 50%, SO 2, LObj 2.2, LO 3) Write a program to insert, delete and print datas from AVL Tree insertion in C Program

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

i. Insertion

In this menu, the program will asked the value that the user want to insert into AVL tree. Do the insertion in (a).

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

ii. Deletion

In this menu, the program will be asked the value that the user wants to delete from AVL tree. If the value is found in the tree, then it will be deleted; otherwise, the program gives the message 'data not found'. Do the deletion in (a).

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

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

iii. Traversal

In this menu, the program will print all data from AVL tree in preorder, inorder, and postorder.

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

iv. Exit

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

Answer:

Snippet of code is displayed below. For further exploration, we have already attached in this .zip file the .cpp program to execute the code.

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

struct AVL{
    int key;
    int height;
    struct AVL* right;
    struct AVL* left;
};

//fungsi untuk membuat node baru
struct AVL* createNode(int key){
    struct AVL* newNode = (struct AVL*)malloc(sizeof(struct AVL));
    newNode->key = key;
    newNode->height = 1;
    newNode->right = NULL;
    newNode->left = NULL;
```

```

        return newNode; //mereturn alamat dari node yang telah dibuat
    }

//fungsi untuk mengetahui angka maksimum dari dua angka yang diberikan
int max(int a, int b){
    if(a > b) {
        return a;
    } else {
        return b;
    }
}

//fungsi untuk mendapatkan ketinggian dari suatu node dalam tree
int getHeight(struct AVL* curr){
    if(curr == NULL) {
        return 0;
    } else {
        return curr->height;
    }
}

/*fungsi untuk mendapatkan balance factor (yaitu tinggi subtree kiri dikurangi tinggi subtree
kanan)*/
int getBalanceFactor(struct AVL* curr){

```

```

    if (curr == NULL) {
        return NULL;
    } else {
        return (getHeight(curr->left) - getHeight(curr->right));
    }
}

//fungsi untuk melakukan left rotation
struct AVL* leftRotate(struct AVL* curr){
    struct AVL* child = curr->right; //menyimpan anak kanan dari curr ke child
    struct AVL* subChild = child->left; //menyimpan anak kiri child ke subChild

    child->left = curr; /*menjadikan curr sebagai anak kiri dari child, kini child menjadi parent
dari curr*/
    curr->right = subChild; //dan subChild sebagai anak kanan dari curr

    //mengupdate ketinggian dari curr dan child
    curr->height = 1 + max(getHeight(curr->left), getHeight(curr->right));
    child->height = 1 + max(getHeight(child->left), getHeight(child->right));

    //mengupdate curr menjadi child dengan return child
    return child;
}

```



```

//fungsi untuk melakukan right rotation
struct AVL* rightRotate(struct AVL* curr){
    struct AVL* child = curr->left; //menyimpan anak kiri dari curr ke child
    struct AVL* subChild = child->right; //menyimpan anak kanan dari child ke subChild

    child->right = curr; /*menjadikan curr sebagai anak kanan dari child, child menjadi parent
dari curr*/
    curr->left = subChild; // subchild menjadi anak kiri dari curr

    //mengupdate ketinggian dari curr dan child
    curr->height = 1 + max(getHeight(curr->left), getHeight(curr->right));
    child->height = 1 + max(getHeight(child->left), getHeight(child->right));

    //mereturn child untuk mengupdate curr menjadi child
    return child;
}

//fungsi untuk menyeimbangkan pohon AVL
struct AVL* balance(struct AVL* curr){
    //mengupdate ketinggian curr terlebih dahulu
    curr->height = max(getHeight(curr->right), getHeight(curr->left)) + 1;
    int balanceFactor = getBalanceFactor(curr); //menyimpan balance factor dari curr

```

```
/*jika balance factor curr lebih besar dari pada 1 dan balance factor anak kiri dari curr
lebih besar sama dengan dari 0, maka ini merupakan left left case, sehingga dibutuhkan right
rotation untuk menyeimbangkannya*/
    if (balanceFactor > 1 && getBalanceFactor(curr->left) >= 0) {
        return rightRotate(curr);
    }
    /*jika balance factor curr lebih besar dari pada 1 dan balance factor anak kiri dari curr
    lebih kecil dari 0, maka ini merupakan left right case, dimana dibutuhkan double rotation*/
    else if (balanceFactor > 1 && getBalanceFactor(curr->left) < 0) {
        curr->left = leftRotate(curr->left); /*lakukan left rotation dulu untuk mengubahnya
        menjadi left left case, lalu lakukan right rotation untuk menyeimbangkannya*/
        return rightRotate(curr);
    }
    /*jika balance factor curr lebih kecil dari -1 dan balance factor anak kanan dari curr lebih
    kecil sama dengan dari 0, maka ini adalah right right case, jadi cukup melakukan rotateleft sekali
    untuk menyeimbangkannya*/
    else if (balanceFactor < -1 && getBalanceFactor(curr->right) <= 0) {
        return leftRotate(curr);
    }
    /*jika balance factor curr lebih kecil dari -1 dan balance factor anak kanan dari curr lebih
    besar dari 0, maka ini adalah right left case, diperlukan double rotation untuk menyeimbangkannya*/
    else if (balanceFactor < -1 && getBalanceFactor(curr->right) > 0) {
        curr->right = rightRotate(curr->right); /*pertama dilakukan right rotation dan dilanjutkan
        dengan left rotation untuk menyeimbangkan tree*/
```

```

        return leftRotate(curr);
    }
    //mereturn curr untuk mengupdate pointer yang harus menunjuk ke curr
    return curr;
}

/*fungsi ini menginsert node ke posisi sama seperti pada BST, namun akan selalu dibalance treenya
supaya tidak mengandung violation*/
struct AVL* insert(struct AVL* curr, struct AVL* newNode){
    if(curr == NULL){
        return newNode;
    }else if(newNode->key > curr->key){
        curr->right = insert(curr->right, newNode);
    }else if(newNode->key < curr->key){
        curr->left = insert(curr->left, newNode);
    }else{
        return curr;
    }
    return balance(curr); //balancing terjadi setiap kali direturn
}

/*fungsi untuk mendapatkan predecessor (value yang nilainya paling mendekati dan bernilai lebih
kecil dari node)*/
struct AVL* getPredecessor(struct AVL* curr) {

```

```

    struct AVL* predecessor = curr->left; /*traversal aka dimulai dari anak kiri dari current
node*/
    while (predecessor->right) { //melakukan traversal ke anak kanan hingga sampai leaf)
        predecessor = predecessor->right;
    }
    return predecessor; //mengembalikan predecessor yang sudah ditemukan
}

//fungsi untuk menghapus node
struct AVL* deleteNode(int key, struct AVL* curr){
    /*delete akan melakukan traversal yang sama seperti deletion pada BST, namun bedanya adalah
tree akan selalu dibalance pada setiap traversal*/
    if (curr == NULL) {
        return NULL;
    } else if (key < curr->key) {
        curr->left = deleteNode(key, curr->left);
    } else if (key > curr->key) {
        curr->right = deleteNode(key, curr->right);
    } else {
        //terdapat 3 kasus jika sudah ketemu node yang akan dihapus
        //pertama, node tersebut tidak memiliki anak, maka dapat langsung dihapus
        if (curr->left == NULL && curr->right == NULL) {
            free(curr);
            curr = NULL;

```

```

    }
    /*kedua, ketika node memiliki 1 anak (kiri atau kanan), sehingga harus memindahkan
posisi anak dari curr ke posisi curr*/
    else if (curr->right == NULL || curr->left == NULL) {
        struct AVL* childNode = (curr->right) ? curr->right : curr->left;
        free(curr);
        curr = childNode;
    }
    /*ketiga, ketika node memiliki 2 anak (kiri dan kanan), sehingga harus menggantikan
node saat ini dengan predecessor*/
    else {
        struct AVL* predecessor = getPredecessor(curr);
        curr->key = predecessor->key;
        curr->left = deleteNode(predecessor->key, curr->left);
    }
}

if(curr == NULL) {
    return NULL; /*jika node saat ini NULL maka tidak perlu melakukan proses balance dan
dapat langsung di return*/
}
return balance(curr); //dan akan dilakukan kembali proses balancing
}

```

```
//fungsi untuk print tree secara in order
```

```
void inOrder(struct AVL* curr){  
    if(curr != NULL){  
        inOrder(curr->left);  
        printf("%d ", curr->key);  
        inOrder(curr->right);  
    }  
}
```

```
//fungsi untuk print tree secara pre order
```

```
void preOrder(struct AVL* curr){  
    if(curr != NULL){  
        printf("%d ", curr->key);  
        preOrder(curr->left);  
        preOrder(curr->right);  
    }  
}
```

```
//fungsi untuk print tree secara post order
```

```
void postOrder(struct AVL* curr){  
    if(curr != NULL){  
        postOrder(curr->left);  
        postOrder(curr->right);  
        printf("%d ", curr->key);  
    }  
}
```

```

    }
}

/*fungsi search digunakan untuk mencari key, jika key ada, maka fungsi akan mereturn integer 1,
jika tidak maka akan mereturn integer 0*/
int search(struct AVL* curr, int keyToFind) {
    if (curr == NULL) {
        return 0;
    } else if (curr->key > keyToFind) {
        return search(curr->left, keyToFind);
    } else if (curr->key < keyToFind) {
        return search(curr->right, keyToFind);
    }
    return 1;
}

int main(){
    struct AVL* root = NULL;
    int option = 0, num = 0;
    do {
        printf("1. Insertion\n");
        printf("2. Deletion\n");
        printf("3. Traversal\n");
        printf("4. Exit\n");
    }
}

```

```
printf("Choose: ");
scanf("%d", &option);
getchar();

switch(option) {
    case 1:
        //opsi 1 jika ingin menginsert node baru
        printf("Insert: ");
        scanf("%d", &num); //mengisi value dari node baru
        getchar();
        root = insert(root, createNode(num)); /*kemudian dibuatkan node baru berisi
value dari input user*/

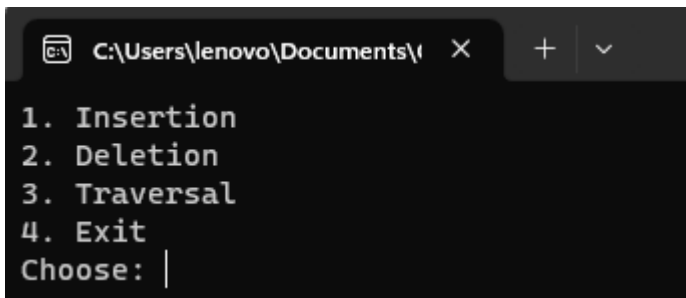
        break;
    case 2:
        //opsi 2 jika ingin menghapus node
        printf("Delete: ");
        scanf("%d", &num); //input node yang ingin dihapus
        getchar();
        if (search(root, num) == 1) {
            //jika node yang ingin dihapus ada
            printf("Data Found\n");
            root = deleteNode(num, root);
            printf("Value %d was deleted\n", num);
        }
    }
}
```



```
        } else {  
            //jika data tidak ditemukan, maka tidak perlu menghapus node apapun  
            printf("Data not found\n");  
        }  
  
        break;  
    case 3:  
        printf("Preorder: ");  
        preOrder(root);  
        printf("\n");  
  
        printf("Inorder: ");  
        inOrder(root);  
        printf("\n");  
  
        printf("Postorder: ");  
        postOrder(root);  
        printf("\n");  
  
        break;  
    default:  
        break;  
}  
printf("\n");
```

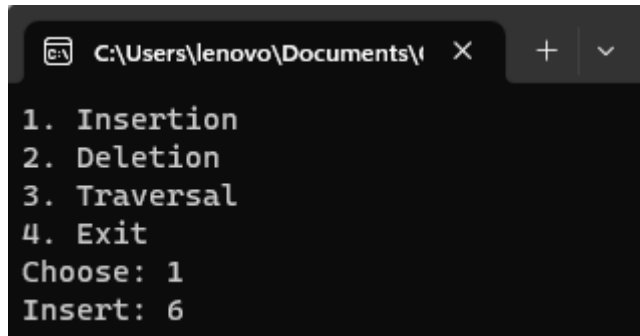
```
    } while (option != 4); // do-while loop akan berhenti jika opsi input adalah 4  
    printf("Thank you\n\n");  
    return 0;  
}
```

From the code above, the output would be displayed as the picture below.



```
C:\Users\lenovo\Documents\l X + v  
1. Insertion  
2. Deletion  
3. Traversal  
4. Exit  
Choose: |
```

(1) Insertion



```
C:\Users\lenovo\Documents\l X + v  
1. Insertion  
2. Deletion  
3. Traversal  
4. Exit  
Choose: 1  
Insert: 6
```

(2) Deletion

```
C:\Users\lenovo\Documents\l X + v
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 2
Delete: 14
Data Found
Value 14 was deleted

C:\Users\lenovo\Documents\l X + v
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 2
Delete: 7
Data not found
```

(3) Traversal

```
C:\Users\lenovo\Documents\l X + v
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
```

(4) Exit

```
C:\Users\lenovo\Documents\l X + v
1. Insertion
2. Deletion
3. Traversal
4. Exit
Choose: 4

Thank you
```

Code can be executed at the separated file, titled “Nomor 2b (AVL Tree Implementation Code).cpp”. For better and clearer view of code documentation, it is suggested to open the .cpp file.