

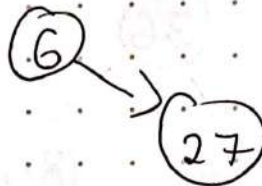
AVL Tree

(i) Insert: 6, 27, 19, 11, 36, 14, 81, 63, 75.

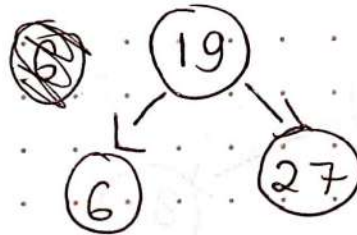
①



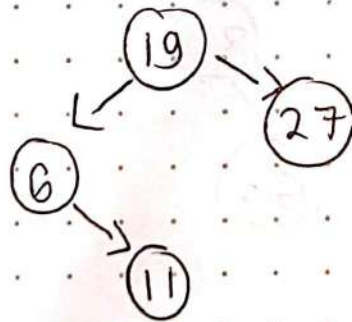
②



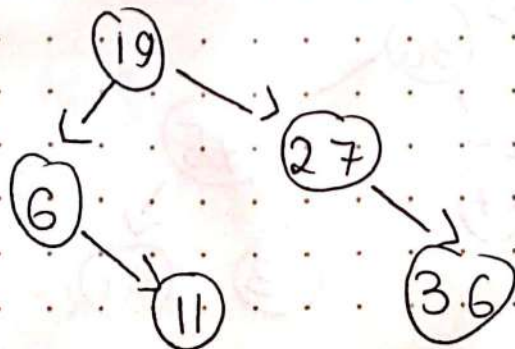
③



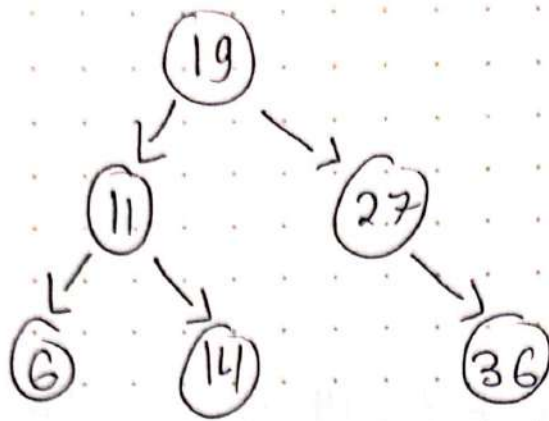
④



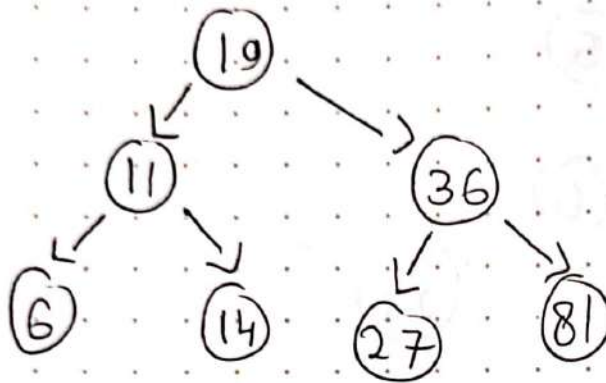
⑤



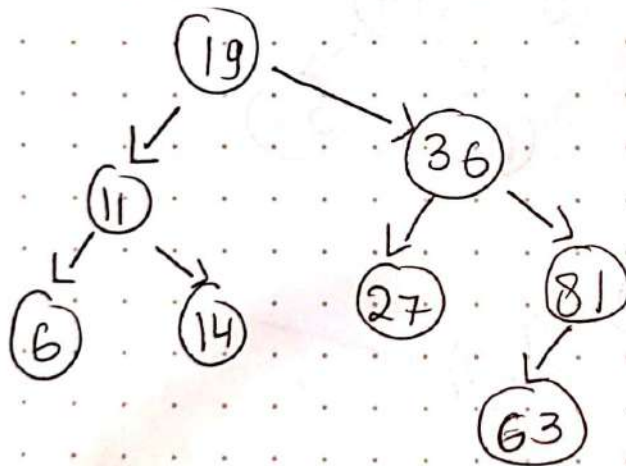
6



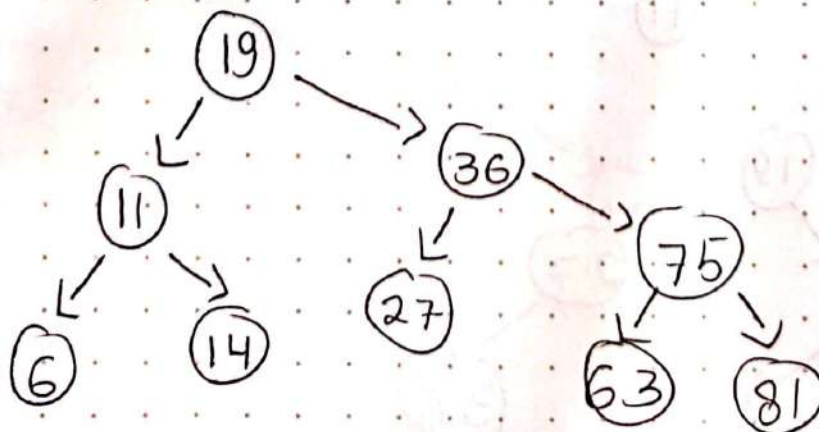
7



8

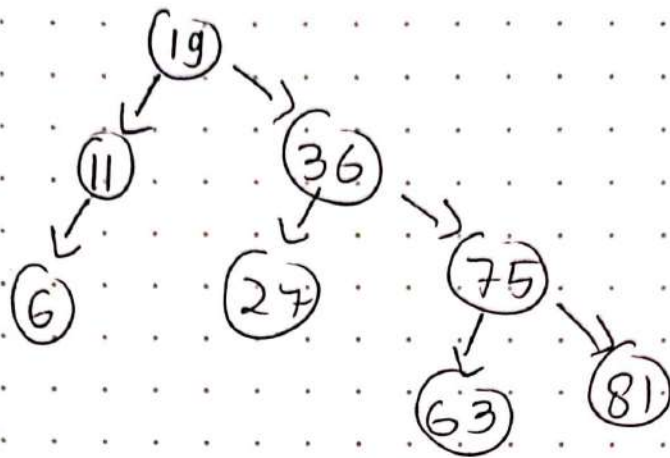


9

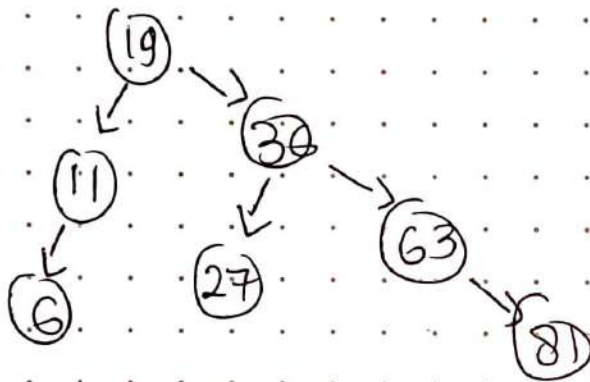


(ii) Delete: 14, 75, 36, 19, 11

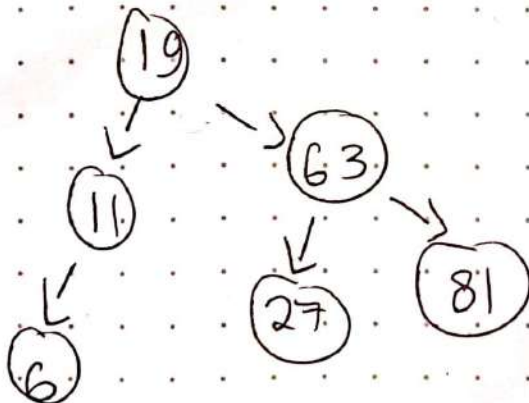
(1)



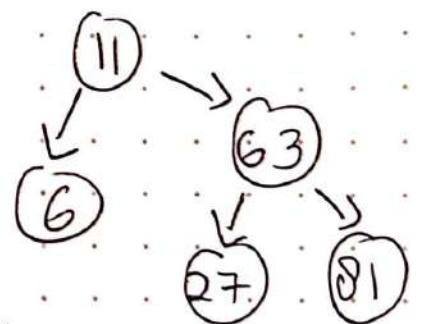
(2)



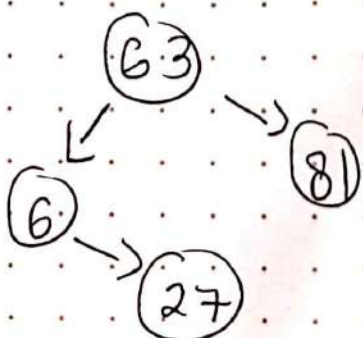
(3)



(4)

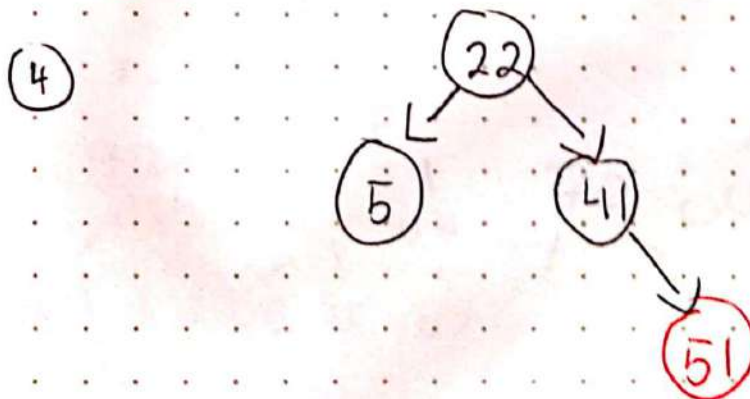
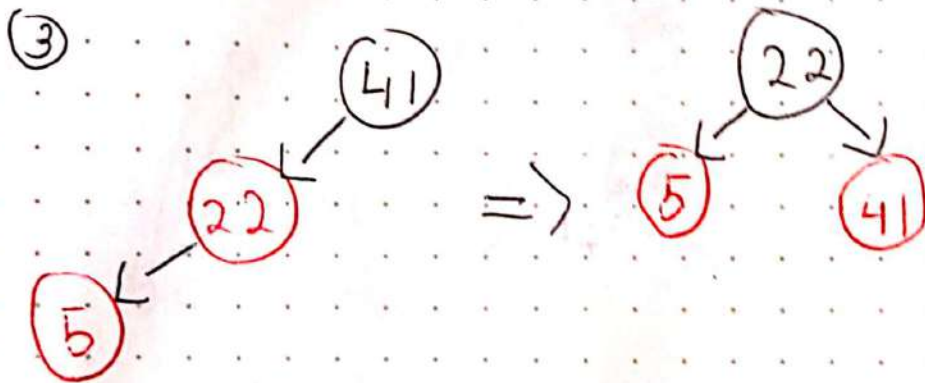
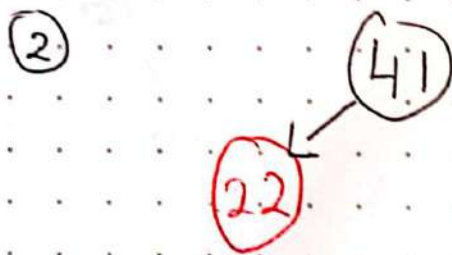
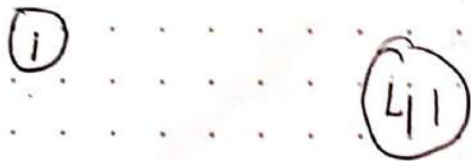


(5)

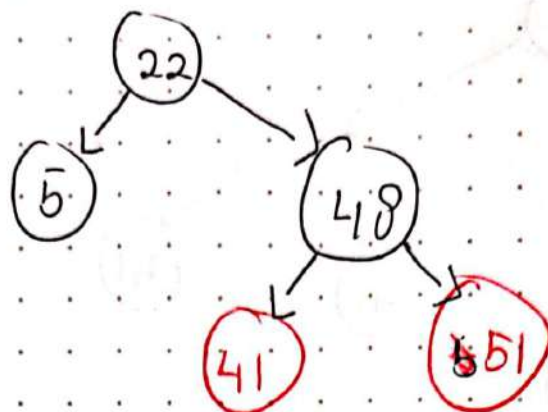


Red Black Tree

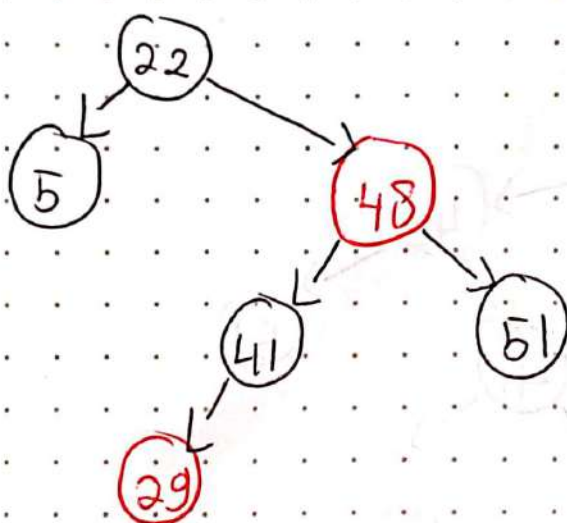
Sequence: 41, 22, 5, 51, 48, 29, 18, 21, 45, 3.



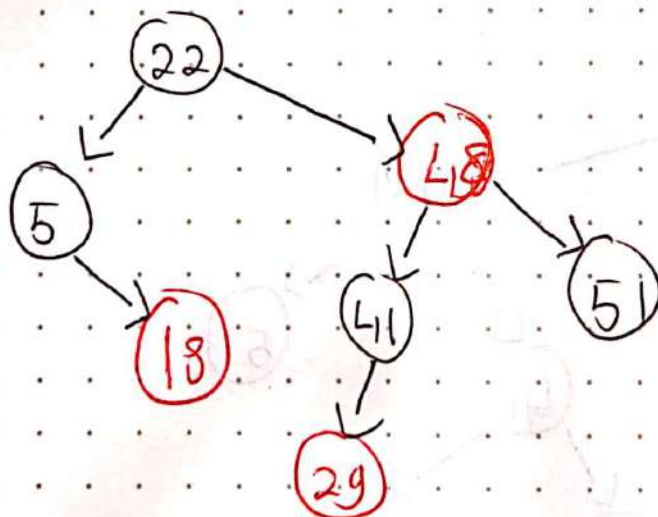
5



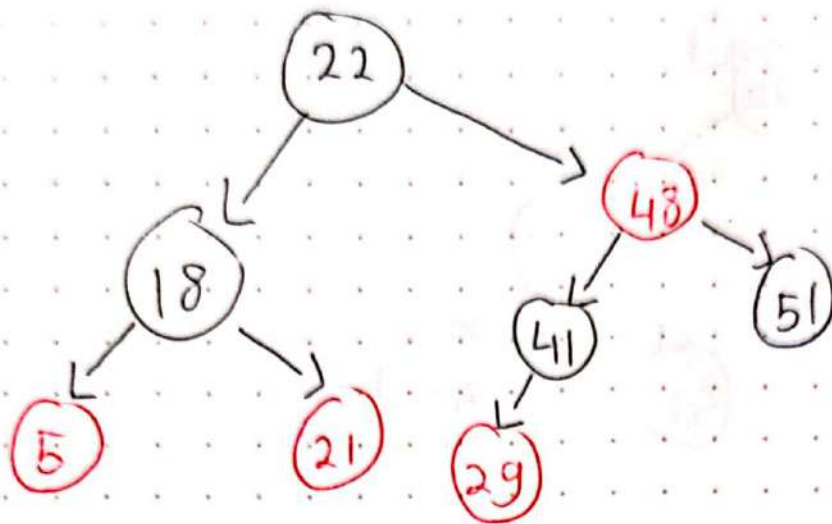
6



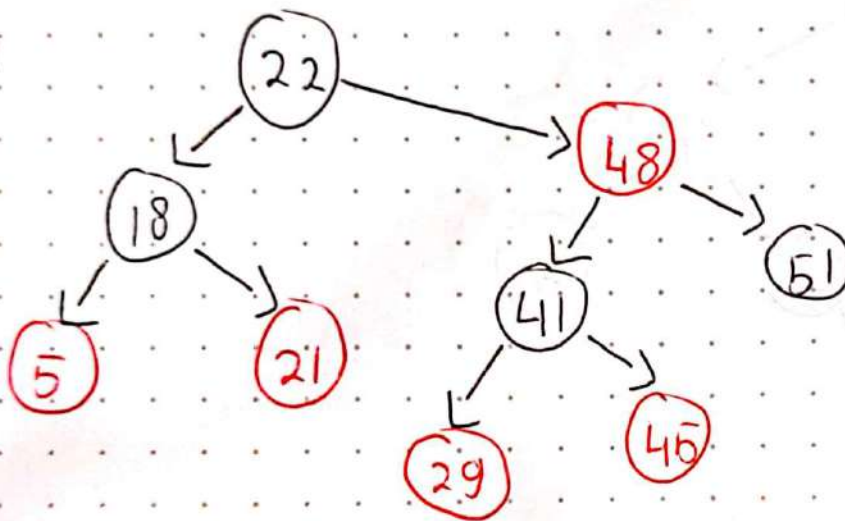
7



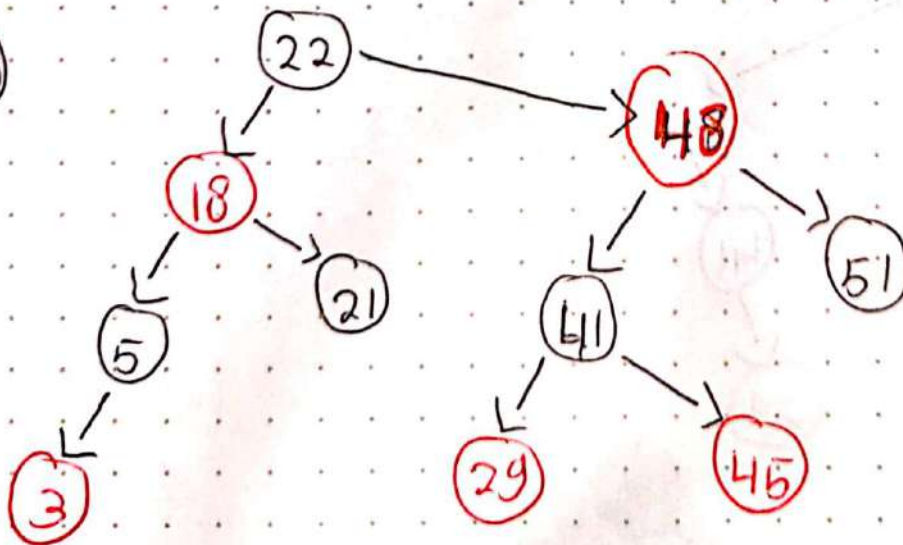
8



9



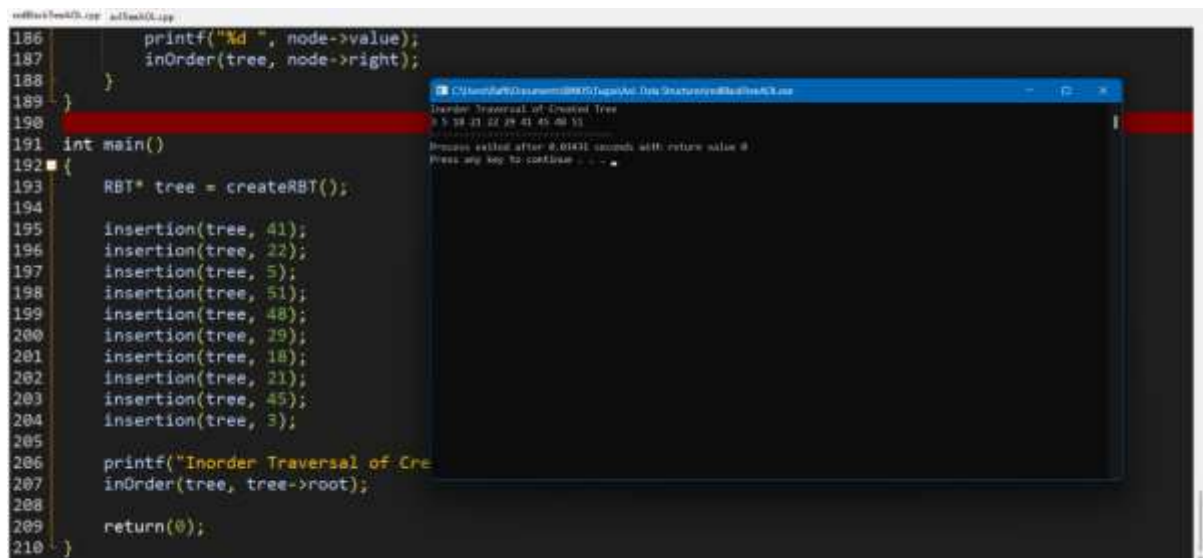
10



Nama: Raissa Raffi Darmawan
NIM: 2602177146

AoL Data Structures

1. Red Black Tree Code

The image shows a screenshot of a code editor with a dark theme. On the left, C++ code is visible, showing a function to insert nodes into a Red Black Tree and a main function that performs these operations. On the right, a terminal window displays the output of the program, showing the in-order traversal of the tree after several insertions. The code includes a function to print the value of a node and its right child, and a main function that creates a tree, inserts several values, and prints the in-order traversal. The terminal output shows the sequence of values: 3 5 18 21 22 29 39 41 45 48 51.

```
186     printf("%d ", node->value);
187     inOrder(tree, node->right);
188 }
189 }
190
191 int main()
192 {
193     RBT* tree = createRBT();
194
195     insertion(tree, 41);
196     insertion(tree, 22);
197     insertion(tree, 5);
198     insertion(tree, 51);
199     insertion(tree, 48);
200     insertion(tree, 29);
201     insertion(tree, 18);
202     insertion(tree, 21);
203     insertion(tree, 45);
204     insertion(tree, 3);
205
206     printf("Inorder Traversal of Cre
207     inOrder(tree, tree->root);
208
209     return(0);
210 }
```

```
C:\kennedy\Raissa\AoL Data Structures\RedBlackTree.exe
Inorder Traversal of Created Tree
3 5 18 21 22 29 39 41 45 48 51
Press any key to continue . . .
```

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

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

```
struct Node
```

```
{
```

```
    int value;
```

```
    int color;
```

```
    Node *parent, *left, *right;
```

```
};
```

```
struct RBT
```

```
{
```

```
    Node *root, *nil;
```

```
};
```

```

Node* createNewNode(int value)
{
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->value = value;
    newNode->parent = newNode->left = newNode->right = NULL;
    newNode->color = 1;

    return newNode;
}

```

```

RBT* createRBT()
{
    RBT* tree = (RBT*)malloc(sizeof(RBT));
    tree->nil = createNewNode(0);
    tree->nil->color = 0;
    tree->root = tree->nil;

    return tree;
}

```

```

void leftRotate(RBT* tree, Node* x)
{
    Node* y = x->right;
    x->right = y->left;
    if(y->left != tree->nil)
    {
        y->left->parent = x;
    }
    y->parent = x->parent;
    if(x->parent == tree->nil)
    {

```



```

        tree->root = y;
    }
    else if(x == x->parent->left)
    {
        x->parent->left = y;
    }
    else
    {
        x->parent->right = y;
    }

    y->left = x;
    x->parent = y;
}

```

```

void rightRotate(RBT* tree, Node* x)
{
    Node* y = x->left;
    x->left = y->right;

    if(y->right != tree->nil)
    {
        y->right->parent = x;
    }
    y->parent = x->parent;

    if(x->parent == tree->nil)
    {
        tree->root = y;
    }
    else if(x == x->parent->right)

```

```

    {
        x->parent->right = y;
    }
    else
    {
        x->parent->left = y;
    }

    y->right = x;
    x->parent = y;
}

void fixInsert(RBT* tree, Node* z)
{
    while(z->parent->color == 1)
    {
        if(z->parent == z->parent->parent->left)
        {
            Node* y = z->parent->parent->right;
            if(y->color == 1)
            {
                z->parent->color = 0;
                y->color = 0;
                z->parent->parent->color = 1;
                z = z->parent->parent;
            }
            else
            {
                if(z == z->parent->right)
                {
                    z = z->parent;
                }
            }
        }
    }
}

```

```

        leftRotate(tree, z);
    }
    z->parent->color = 0;
    z->parent->parent->color = 1;
    rightRotate(tree, z->parent->parent);
}
}
else
{
    Node* y = z->parent->parent->left;
    if(y->color == 1)
    {
        z->parent->color = 0;
        y->color = 0;
        z->parent->parent->color = 1;
        z = z->parent->parent;
    }
    else
    {
        if(z == z->parent->left)
        {
            z = z->parent;
            rightRotate(tree, z);
        }
        z->parent->color = 0;
        z->parent->parent->color = 1;
        leftRotate(tree, z->parent->parent);
    }
}
}

```

```

        tree->root->color = 0;
    }

void insertion(RBT* tree, int value)
{
    Node* z = createNewNode(value);
    Node* y = tree->nil;
    Node* x = tree->root;

    while(x != tree->nil)
    {
        y = x;
        if(z->value < x->value)
        {
            x = x->left;
        }
        else
        {
            x = x->right;
        }
    }
    z->parent = y;
    if(y == tree->nil)
    {
        tree->root = z;
    }
    else if(z->value < y->value)
    {
        y->left = z;
    }
    else

```



```

    {
        y->right = z;
    }

    z->left = tree->nil;
    z->right = tree->nil;
    z->color = 1;
    fixInsert(tree, z);
}

void inOrder(RBT* tree, Node* node)
{
    if(node != tree->nil)
    {
        inOrder(tree, node->left);
        printf("%d ", node->value);
        inOrder(tree, node->right);
    }
}

int main()
{
    RBT* tree = createRBT();

    insertion(tree, 41);
    insertion(tree, 22);
    insertion(tree, 5);
    insertion(tree, 51);
    insertion(tree, 48);
    insertion(tree, 29);
    insertion(tree, 18);
}

```

```

insertion(tree, 21);

insertion(tree, 45);

insertion(tree, 3);

printf("Inorder Traversal of Created Tree\n");

inOrder(tree, tree->root);

return(0);

}

```

2. AVL Tree Code

The screenshot shows a C++ IDE with the following code on the left and its execution output on the right.

```

179 }
180
181
182 root->height = max(height(root->left), height(root->right));
183 printf("Data not found\n");
184 return rebalance(root);
185 }
186
187 void menu()
188 {
189     puts("1. Insertion");
190     puts("2. Deletion");
191     puts("3. Transversal");
192     puts("4. Exit");
193     printf("Choose: ");
194 }
195
196 void inOrder(Data *root)
197 {
198     if(root)
199     {
200         inOrder(root->left);
201         printf("%d ", root->value);
202         inOrder(root->right);
203     }
204 }

```

The execution output on the right shows the following sequence of operations:

```

1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 4
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 21
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 45
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 3
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 3

```

The screenshot shows a C++ IDE with the following code on the left and its execution output on the right.

```

179 }
180
181
182 root->height = max(height(root->left), height(root->right));
183 printf("Data not found\n");
184 return rebalance(root);
185 }
186
187 void menu()
188 {
189     puts("1. Insertion");
190     puts("2. Deletion");
191     puts("3. Transversal");
192     puts("4. Exit");
193     printf("Choose: ");
194 }
195
196 void inOrder(Data *root)
197 {
198     if(root)
199     {
200         inOrder(root->left);
201         printf("%d ", root->value);
202         inOrder(root->right);
203     }
204 }

```

The execution output on the right shows the following sequence of operations:

```

4. Exit
Choose: 4
Insert: 41
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 71
1. Insertion
2. Deletion
3. Transversal
4. Exit
Delete: 4
Data not found
Data not found
Data not found
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 3

```

The screenshot shows a C++ IDE with a code editor on the left and a console window on the right. The code in the editor is as follows:

```
179 }
180
181
182 root->height = max(height(root->left), height(root->right));
183 printf("Data not found\n");
184 return rebalance(root);
185 }
186
187 void menu()
188 {
189     puts("1. Insertion");
190     puts("2. Deletion");
191     puts("3. Transversal");
192     puts("4. Exit");
193     printf("Choose: ");
194 }
195
196 void inOrder(Data *root)
197 {
198     if(root)
199     {
200         inOrder(root->left);
201         printf("%d ", root->value);
202         inOrder(root->right);
203     }
```

The console window on the right displays the output of the program:

```
C:\Users\Adriano\Documents\AVL\AVL.exe
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Inorder: 10 11 6 14 63 36 27 41 81 43
Preorder: 6 11 14 10 27 36 41 63 81
Postorder: 6 14 11 27 63 36 43 81 29
```

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

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

```
#include<string.h>
```

```
void pressEnter()
```

```
{
    char ch;
    scanf("%c", &ch);
    getchar();
}
```

```
struct Data
```

```
{
    int value;
    int height;

    Data *left, *right;
};
```

```
Data *newNode(int value)
```

```
{
```

```

    Data *temp = (Data*)malloc(sizeof(Data));
    temp->value = value;
    temp->height = 1;
    temp->left = temp->right = NULL;

    return temp;
}

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

int height(Data *root)
{
    if(root == NULL)
    {
        return(0);
    }
    return root->height;
}

int bf(Data *root)
{
    if(root == NULL)
    {
        return(0);
    }

    return height(root->left) - height(root->right);
}

```



```

Data *leftRotate(Data *root)
{
    Data *rightChild = root->right;
    Data *leftRightChild = rightChild->left;
    rightChild->left = root;
    root->right = leftRightChild;

    root->height = max(height(root->left),
height(root->right)) + 1;

    rightChild->height = max(height(rightChild->left),
height(rightChild->right)) + 1;

    return rightChild;
}

Data *rightRotate(Data *root)
{
    Data *leftChild = root->left;
    Data *rightLeftChild = leftChild->left;
    leftChild->right = root;
    root->left = rightLeftChild;

    root->height = max(height(root->left),
height(root->right)) + 1;

    leftChild->height = max(height(root->left),
height(root->right)) + 1;

    return leftChild;
}

```

```

Data *rebalance(Data *root)
{
    int factor = bf(root);

    if(factor > 1)
    {
        if(bf(root->left) >= 0)
        {
            return rightRotate(root);
        }
        else
        {
            root->left = leftRotate(root->left);
            return rightRotate(root);
        }
    }
    else if(factor < -1)
    {
        if(bf(root->right) <= 0)
        {
            return leftRotate(root);
        }
        else
        {
            root->right = rightRotate(root->right);
            return leftRotate(root);
        }
    }

    return root;
}

```

```
}
```

```
Data *insertion(Data *root, int value)
```

```
{
```

```
    if(root == NULL)
```

```
    {
```

```
        return newNode(value);
```

```
    }
```

```
    else if(value > root->value)
```

```
    {
```

```
        root->right = insertion(root->right, value);
```

```
    }
```

```
    else if(value < root->value)
```

```
    {
```

```
        root->left = insertion(root->left, value);
```

```
    }
```

```
        root->height = max(height(root->left),  
height(root->right)) + 1;
```

```
        return rebalance(root);
```

```
}
```

```
Data *pop(Data *root, int value)
```

```
{
```

```
    if(root == NULL)
```

```
    {
```

```
        return NULL;
```

```
    }
```

```
    else if(value > root->value)
```

```
    {
```

```
        root->right = pop(root->right, value);
```

```

}
else if(value < root->value)
{
    root->left = pop(root->left, value);
}
else
{
    if(root->left == NULL)
    {
        Data *temp = root->right;
        free(root);
        root = NULL;
        printf("Data Found\n");
        printf("Value %d was deleted\n", value);
        return temp;
    }
    else if(root->right == NULL)
    {
        Data *temp = root->left;
        free(root);
        root = NULL;
        printf("Data Found\n");
        printf("Value %d was deleted\n", value);
        return temp;
    }
    else
    {
        Data *temp = root->left;

        while(temp->right)
        {

```



```

        temp = temp->right;
    }

    root->value = temp->value;
    root->left = pop(root->left, value);
}

}

    root->height = max(height(root->left),
height(root->right)) + 1;
    printf("Data not found\n");
    return rebalance(root);
}

void menu()
{
    puts("1. Insertion");
    puts("2. Deletion");
    puts("3. Transversal");
    puts("4. Exit");
    printf("Choose: ");
}

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

```

```

    }
}

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

```

```

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

```

```

int main()
{
    Data *root = NULL;

    int option;

    do
    {

```

```
menu();  
scanf("%d", &option);  
  
switch(option)  
{  
    case 1:  
        printf("Insert: ");  
        int value;  
        scanf("%d", &value);  
        root = insertion(root, value);  
        break;  
    case 2:  
        printf("Delete: ");  
        int value1;  
        scanf("%d", &value1);  
        root = pop(root, value1);  
        break;  
    case 3:  
        printf("Preorder: ");  
        preOrder(root);  
        printf("\n");  
  
        printf("Inorder: ");  
        inOrder(root);  
        printf("\n");  
  
        printf("Postorder: ");  
        postOrder(root);  
        printf("\n");  
  
        pressEnter();  
}
```

```
        break;
    case 4:
        printf("Thank you\n");
        break;
    }

    }while(option != 4);

return(0);
}
```

redBlackTreeAOL.cpp avTreeAOL.cpp

```
186     printf("%d ", node->value);
187     inOrder(tree, node->right);
188 }
189 }
190
191 int main()
192 {
193     RBT* tree = createRBT();
194
195     insertion(tree, 41);
196     insertion(tree, 22);
197     insertion(tree, 5);
198     insertion(tree, 51);
199     insertion(tree, 48);
200     insertion(tree, 29);
201     insertion(tree, 18);
202     insertion(tree, 21);
203     insertion(tree, 45);
204     insertion(tree, 3);
205
206     printf("Inorder Traversal of Created Tree\n");
207     inOrder(tree, tree->root);
208
209     return(0);
210 }
```

C:\Users\Raffi\Documents\BINUS\Tugas\Aol. Data Structures\redBlackTreeAOL.exe

Inorder Traversal of Created Tree
3 5 18 21 22 29 41 45 48 51

Process exited after 0.03431 seconds with return value 0
Press any key to continue . . .

redBlackTreeAOL.cpp avTreeAOL.cpp

```
179 }
180
181
182 root->height = max(height(ro
183 printf("Data not found\n");
184 return rebalance(root);
185 }
186
187 void menu()
188 {
189     puts("1. Insertion");
190     puts("2. Deletion");
191     puts("3. Transversal");
192     puts("4. Exit");
193     printf("Choose: ");
194 }
195
196 void inOrder(Data *root)
197 {
198     if(root)
199     {
200         inOrder(root->left);
201         printf("%d ", root->value);
202         inOrder(root->right);
203     }
204 }
```

C:\Users\Raffi\Documents\BINUS\Tugas\Aol Data Structures\avTreeAOL.exe

```
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 6
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 27
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 19
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 11
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 36
```

redBlackTreeAOL.cpp avlTreeAOL.cpp

```
179     }
180
181     root->height = max(height(root->left), height(root->right));
182     printf("Data not found\n");
183     return rebalance(root);
184 }
185
186 void menu()
187 {
188     puts("1. Insertion");
189     puts("2. Deletion");
190     puts("3. Transversal");
191     puts("4. Exit");
192     printf("Choose: ");
193 }
194
195 void inOrder(Data *root)
196 {
197     if(root)
198     {
199         inOrder(root->left);
200         printf("%d ", root->value);
201         inOrder(root->right);
202     }
203 }
```

C:\Users\Raffa\Documents\BINUS\Tugas\Aol. Data Structures\avlTreeAOL.exe

```
4. Exit
Choose: 1
Insert: 63
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 75
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 2
Delete: 4
Data not found
Data not found
Data not found
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: _
```


redBlackTreeAOL.cpp avlTreeAOL.cpp

```
179     }
180
181
182     root->height = max(height(ro
183     printf("Data not found\n");
184     return rebalance(root);
185 }
186
187 void menu()
188 {
189     puts("1. Insertion");
190     puts("2. Deletion");
191     puts("3. Transversal");
192     puts("4. Exit");
193     printf("Choose: ");
194 }
195
196 void inOrder(Data *root)
197 {
198     if(root)
199     {
200         inOrder(root->left);
201         printf("%d ", root->value);
202         inOrder(root->right);
203     }
```

C:\Users\Raffi\Documents\BINUS\TugasAoL Data Structures\avlTreeAOL.exe

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