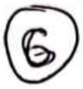
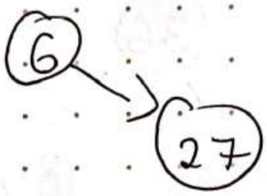
AVL tree

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

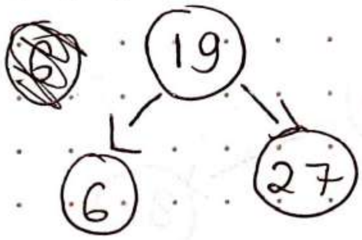
1



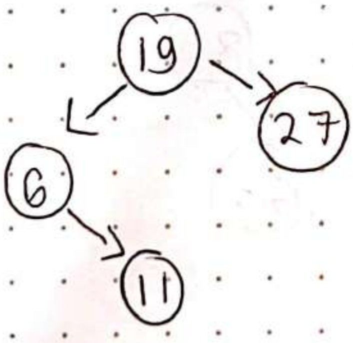
2



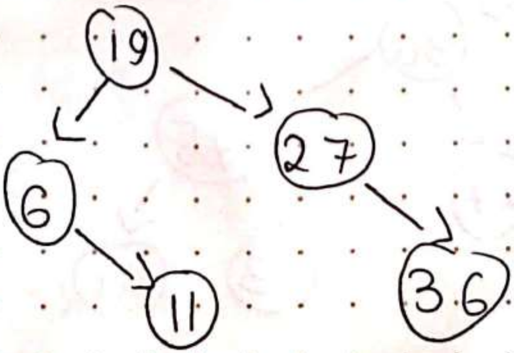
3



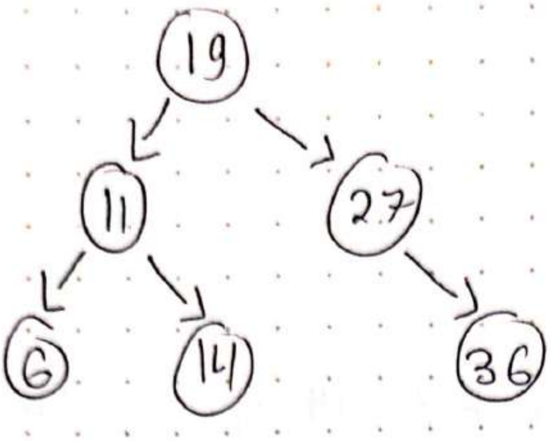
4



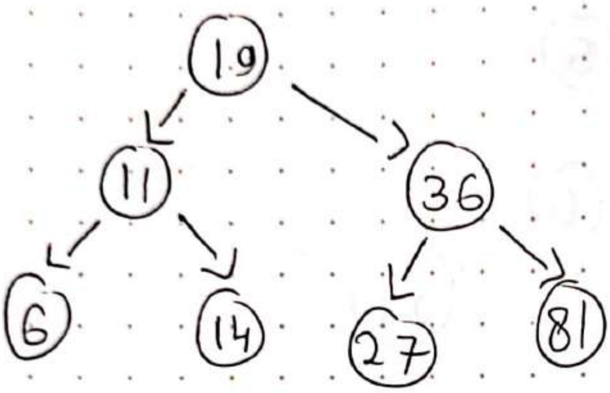
5



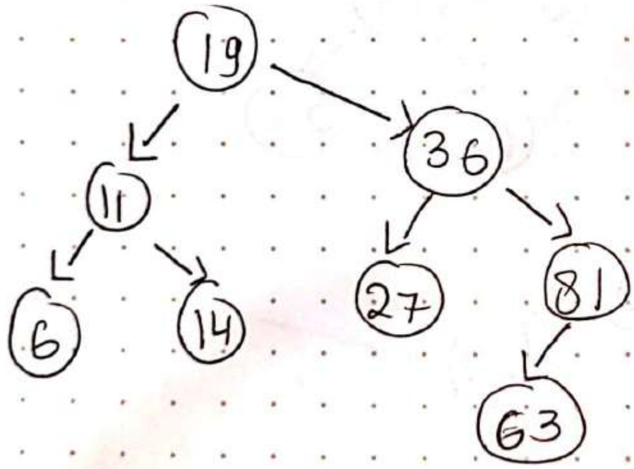
6


19
11
27
6
14
36

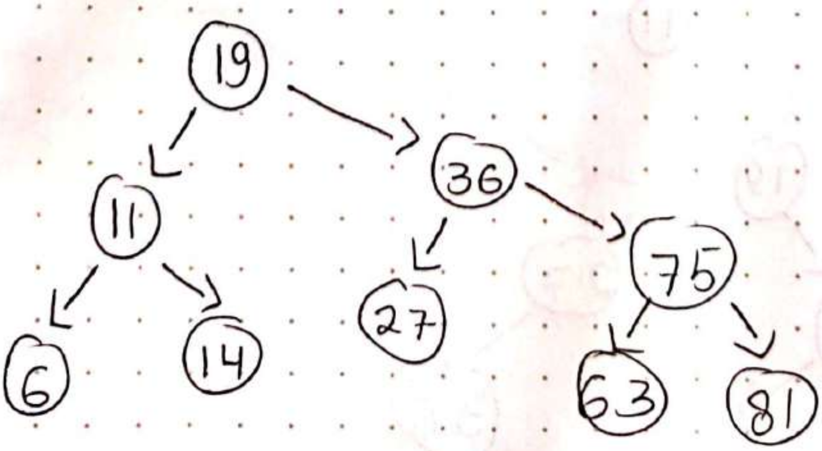
7


19
11
36
6
14
27
81

8

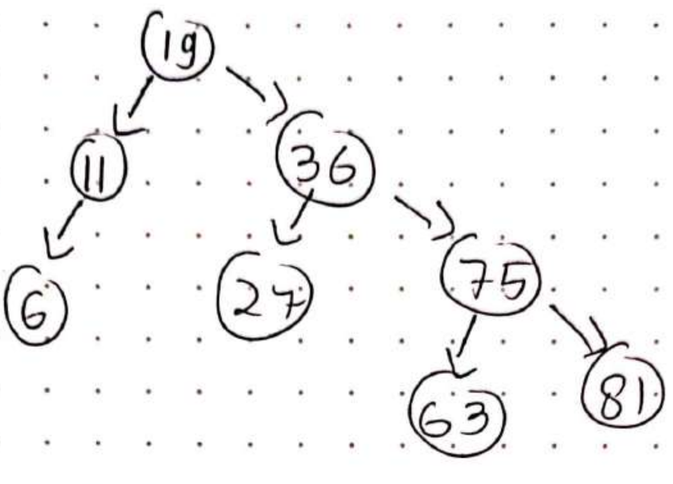

19
11
36
6
14
27
81
63

9

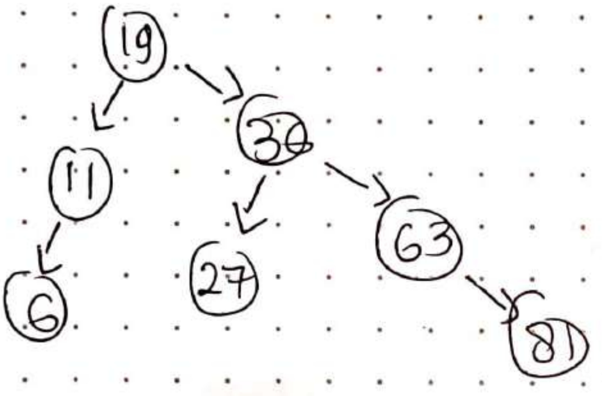

19
11
36
6
14
27
75
63
81

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

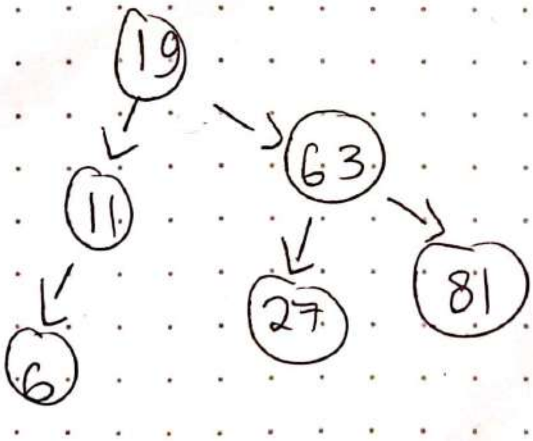
(1)



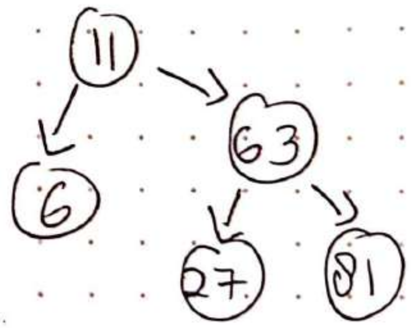
(2)



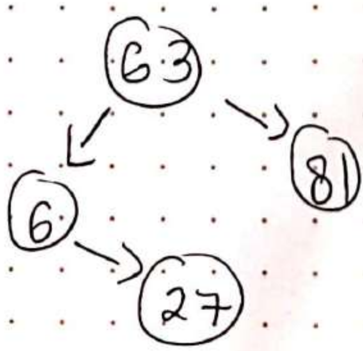
(3)



(4)



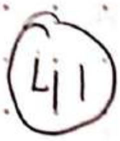
(5)



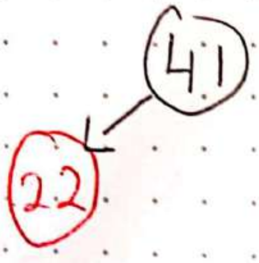
Red Black Tree

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

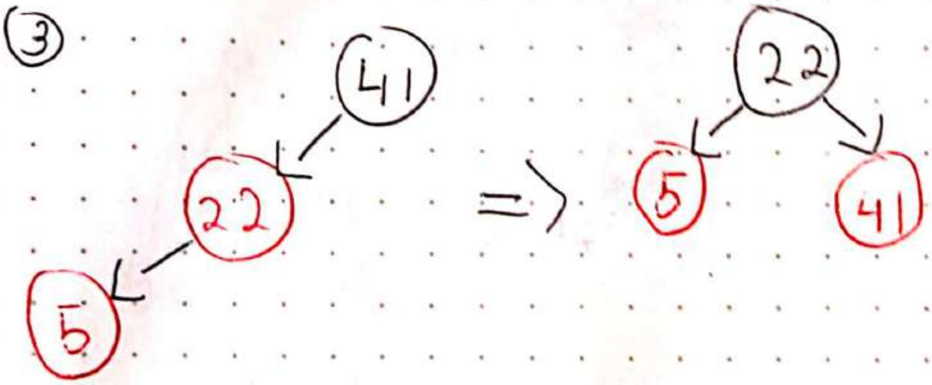
①



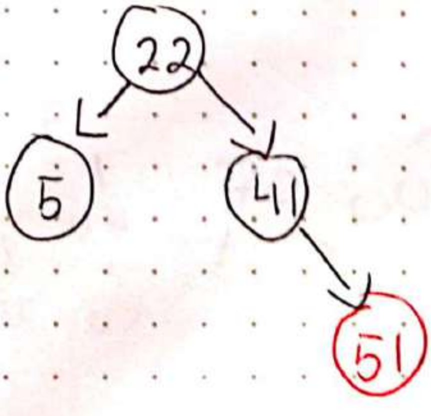
②



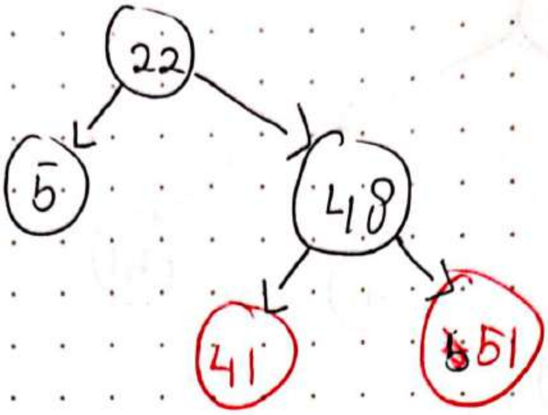
③



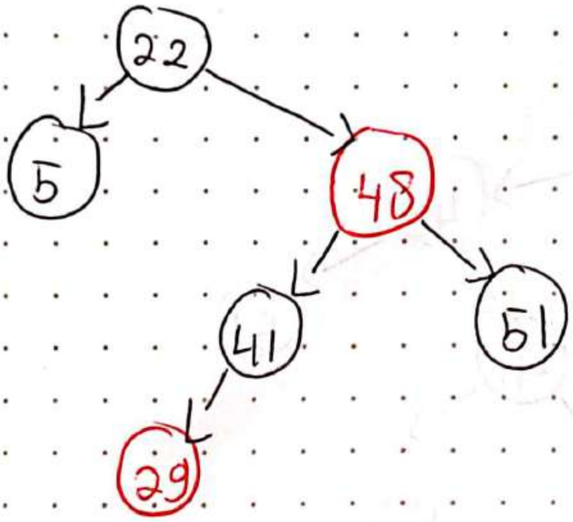
④



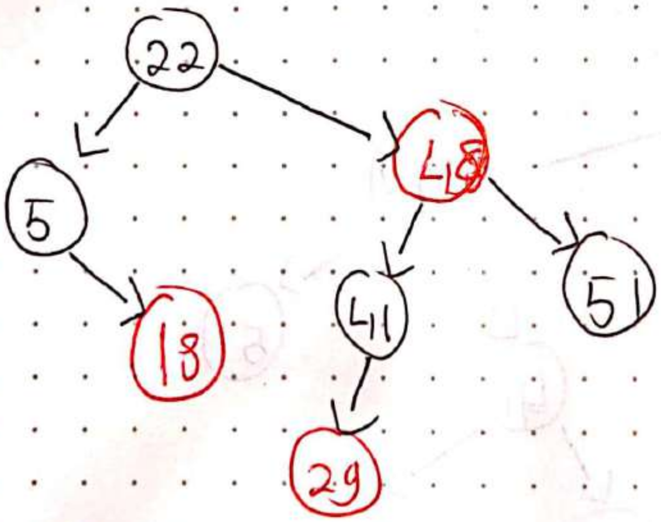
5



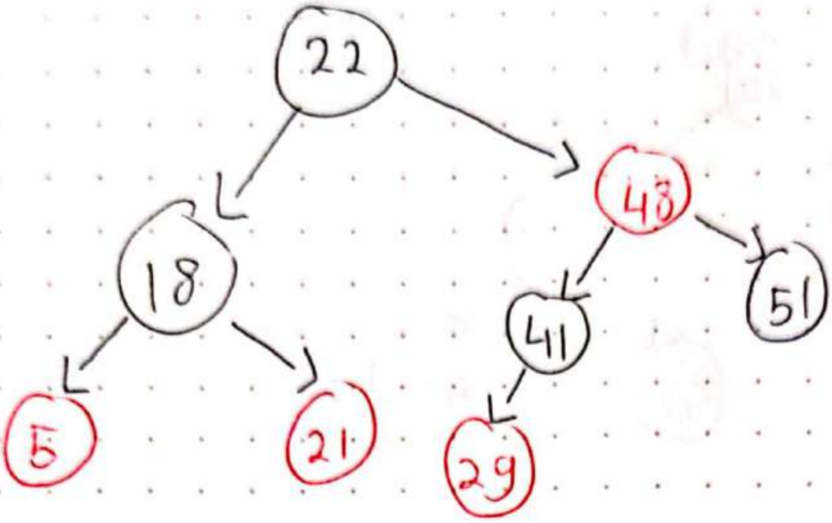
6



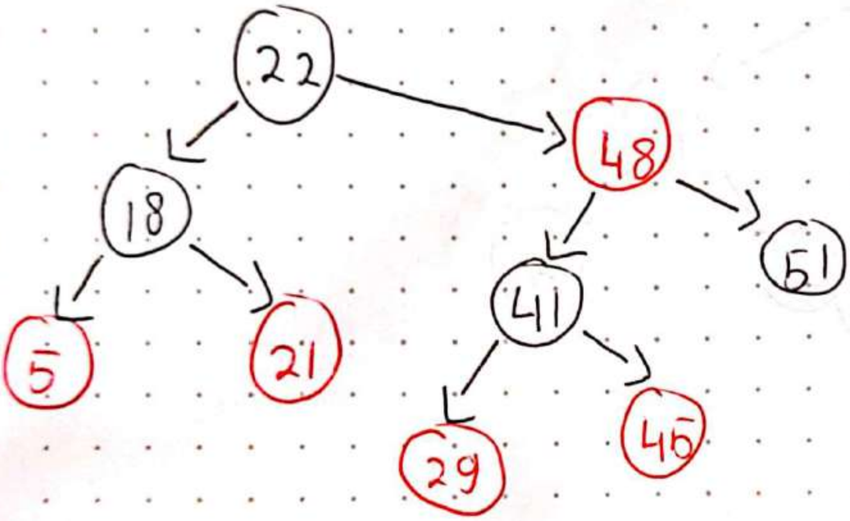
7



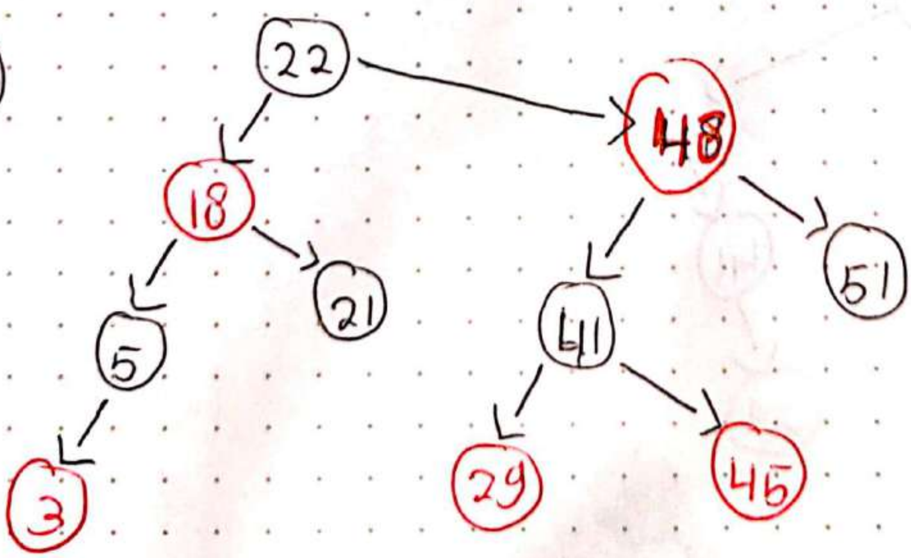
8


22
18
48
5
21
41
51
29

9


22
18
48
5
21
41
51
29
45

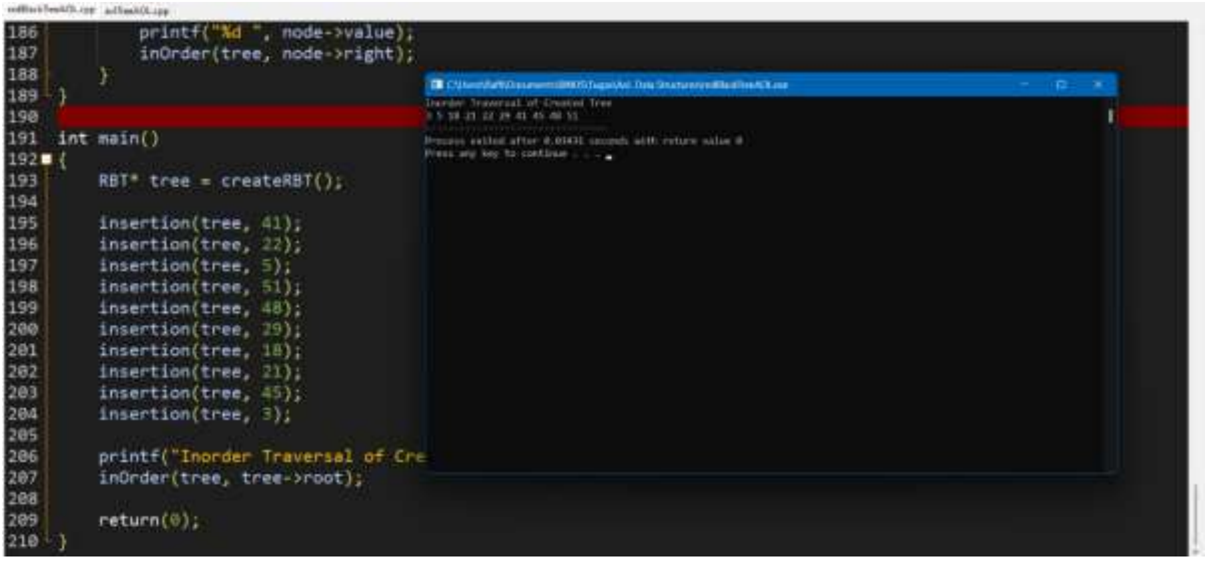
10


22
18
48
5
21
41
51
3
29
45

Nama: Raissa Raffi Darmawan
NIM: 2602177146

AoL Data Structures

1. Red Black Tree Code



```
            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 Cre
    inOrder(tree, tree->root);

    return(0);
}
```

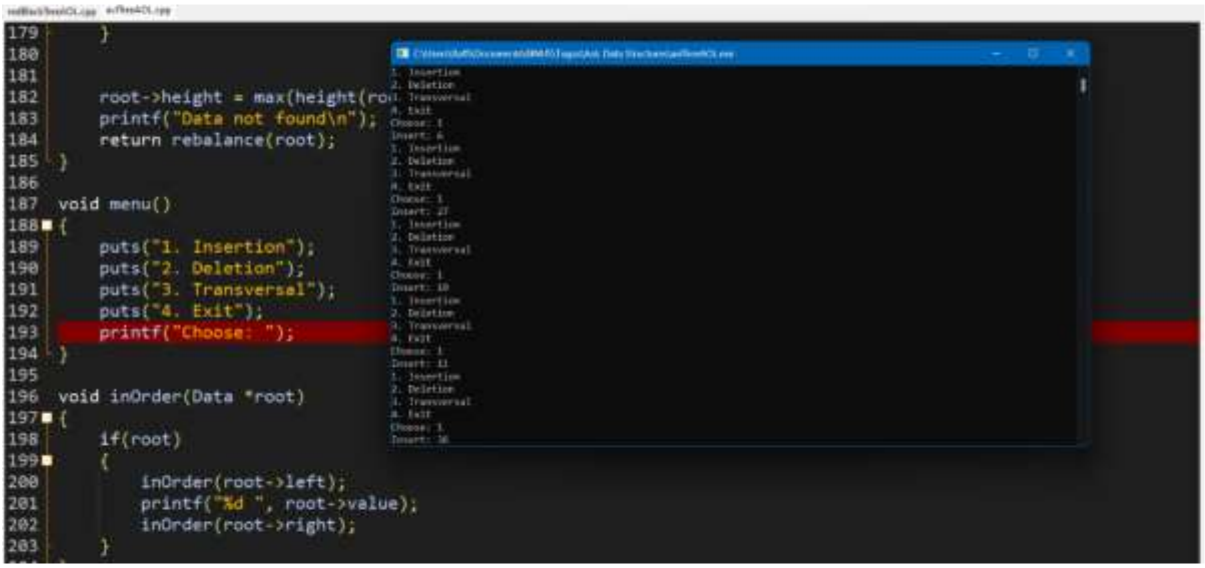
```
Inorder Traversal of Created Tree
3 5 18 21 22 29 41 45 48 51
--------------------------------
Process exited after 0.03431 seconds with return value 0
Press any key to continue . . .
```

```
#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);
            }
        }
    }
```

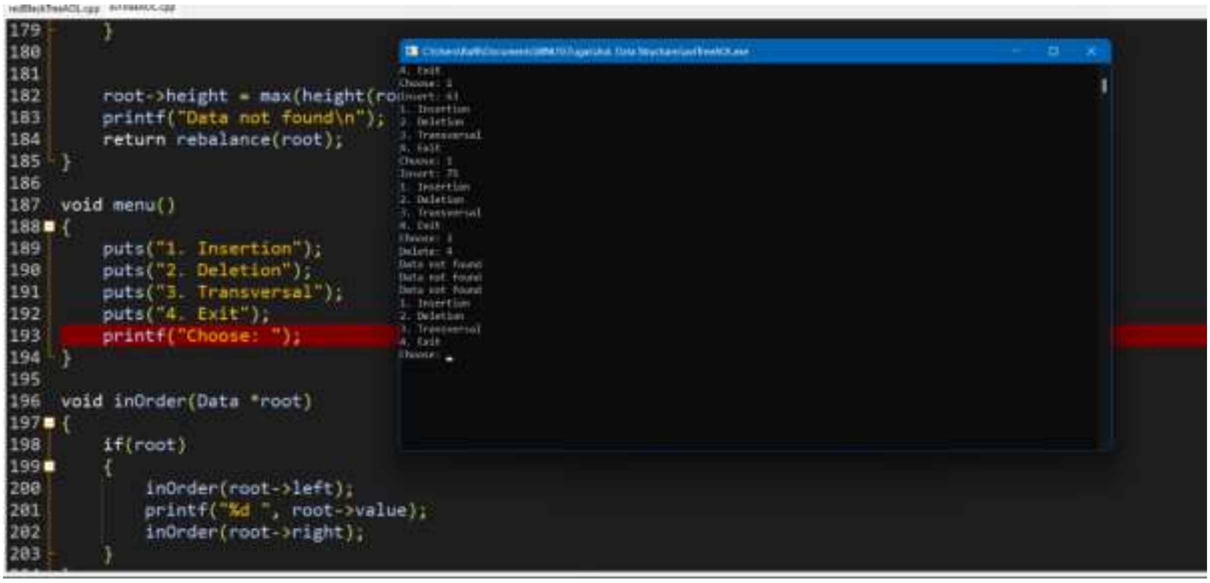
```
    tree->root->color = 0;
}

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



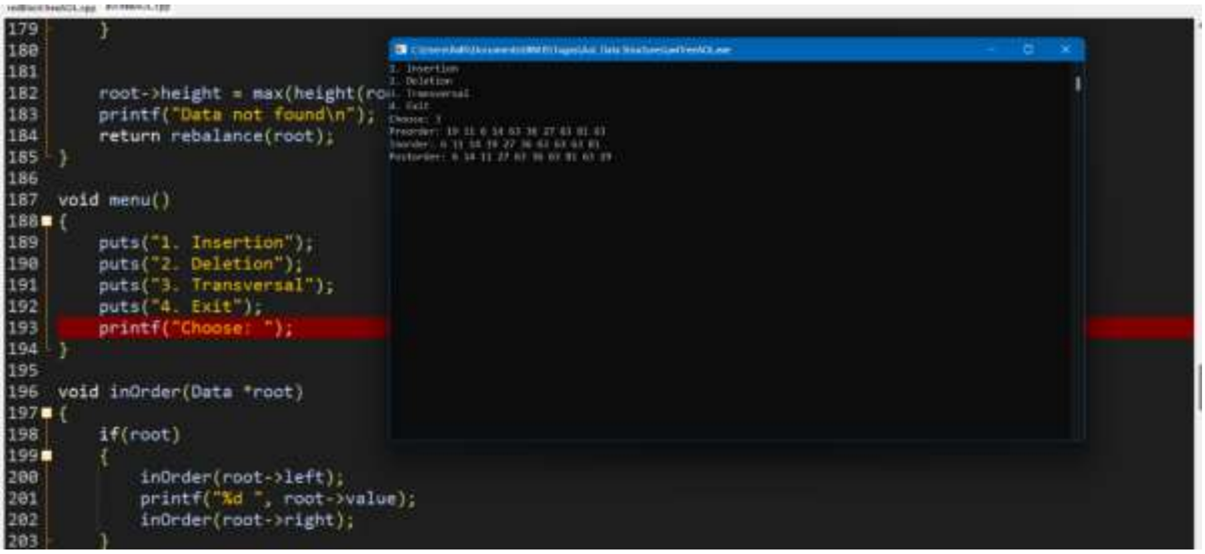
```
    }

    root->height = max(height(ro
    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);
    }
```

```
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: 18
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 11
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 1
Insert: 36
```



```
    }

    root->height = max(height(ro
    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);
    }
```

```
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: _
```



```
    }


    root->height = max(height(ro
    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);
    }
```

```
1. Insertion
2. Deletion
3. Transversal
4. Exit
Choose: 3
Preorder: 19 11 6 14 63 36 27 61 81 67
Inorder: 6 11 14 19 27 36 61 63 67 81
Postorder: 6 14 11 27 61 36 67 81 63 19
```

```
#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;
```

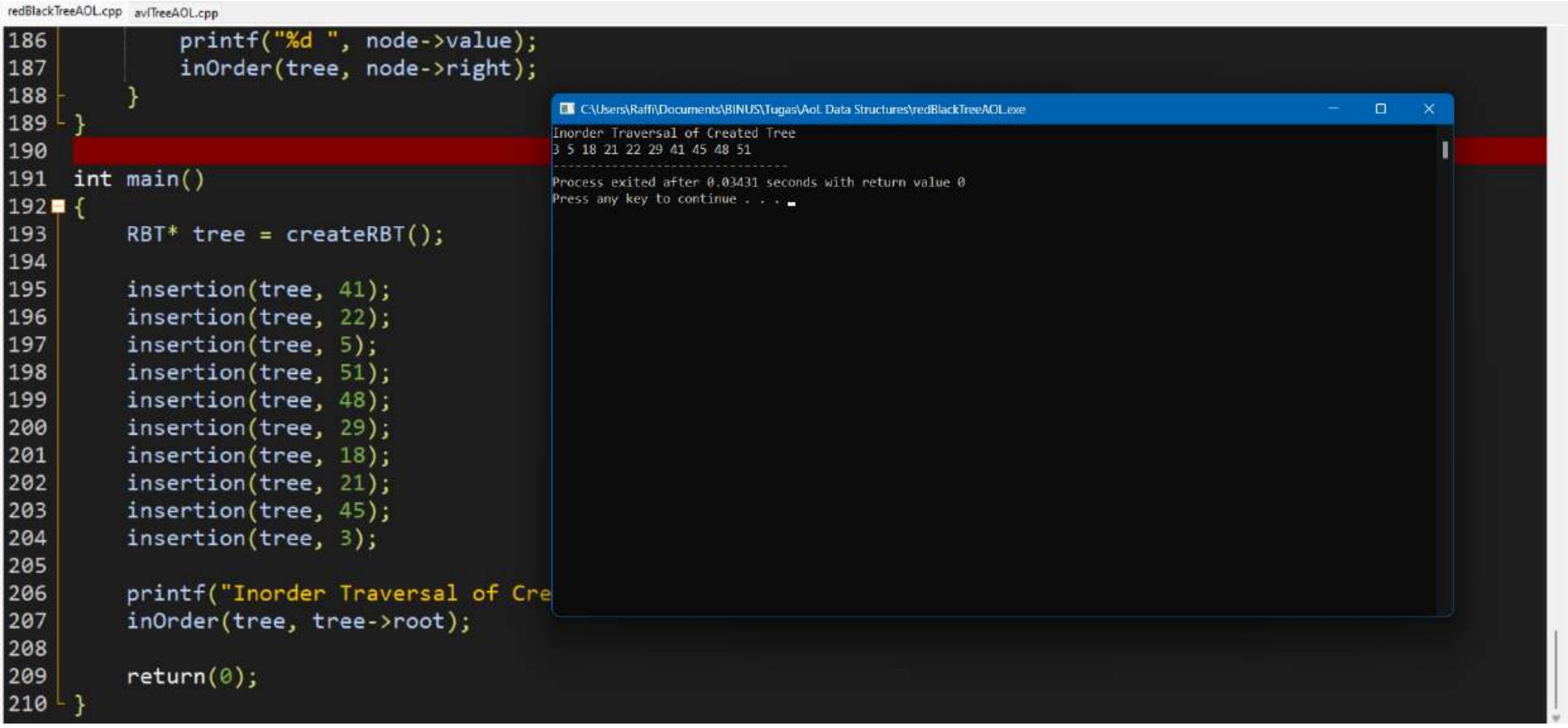
```
}

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
          {
```

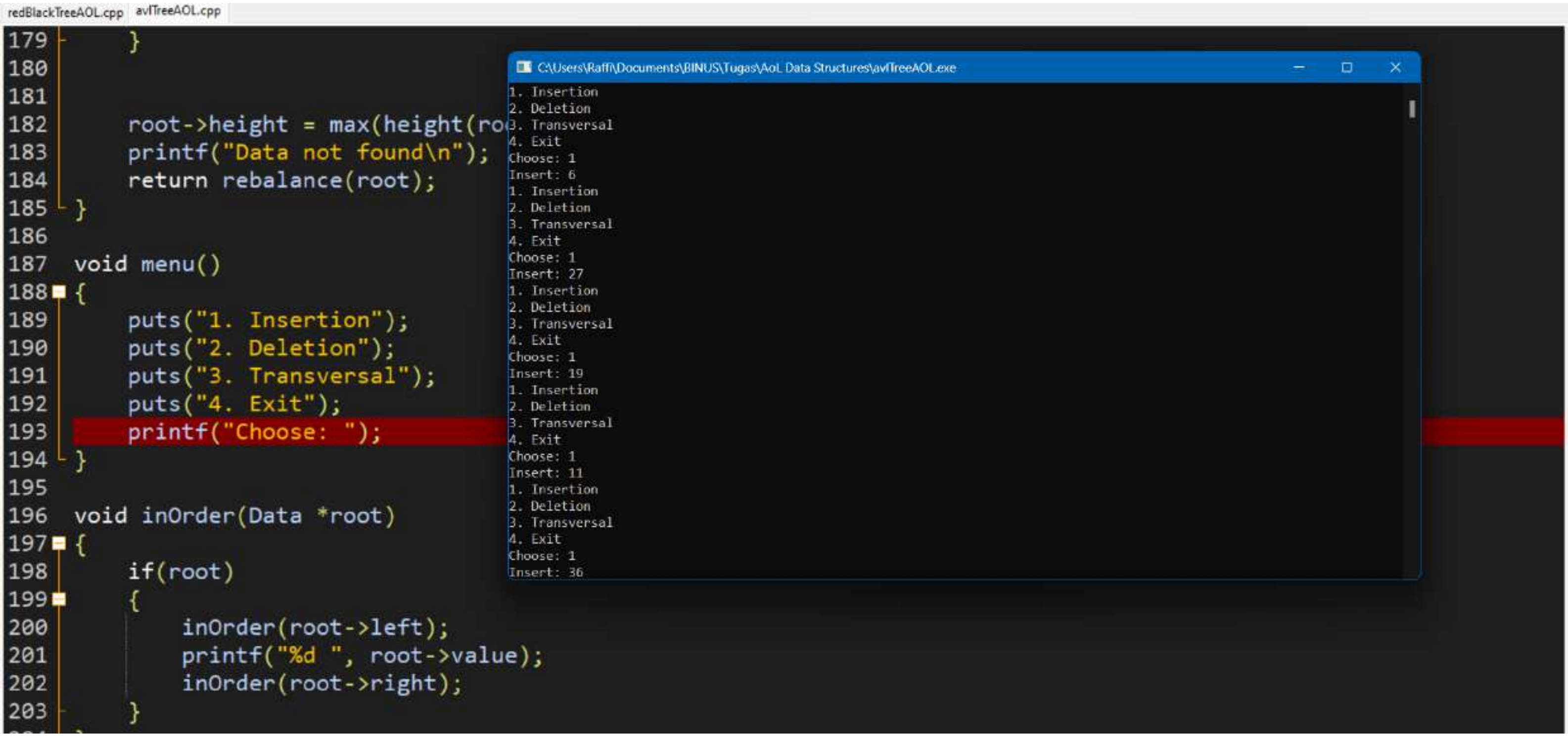
```
menu();
scanf("%d", &option);

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
avlTreeAOL.cpp
C:\Users\Raffi\Documents\BINUS\Tugas\AoL Data Structures\redBlackTreeAOL.exe

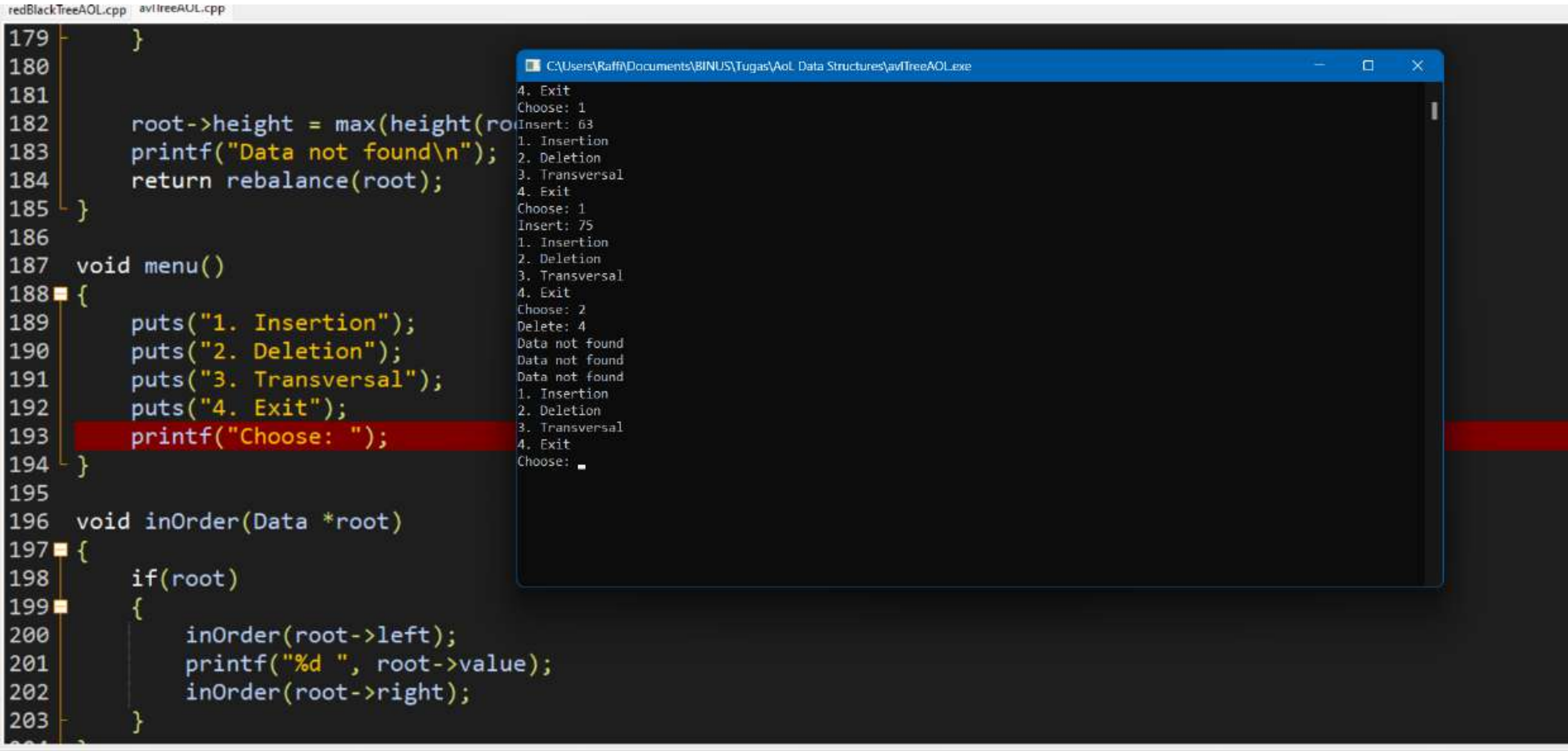
```
        printf("%d ", node->value);
        inOrder(tree, node->right);
    }
}

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 Cre
    inOrder(tree, tree->root);

    return(0);
}
```

```
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
avlTreeAOL.cpp
C:\Users\Raffi\Documents\BINUS\Tugas\AoL Data Structures\avlTreeAOL.exe

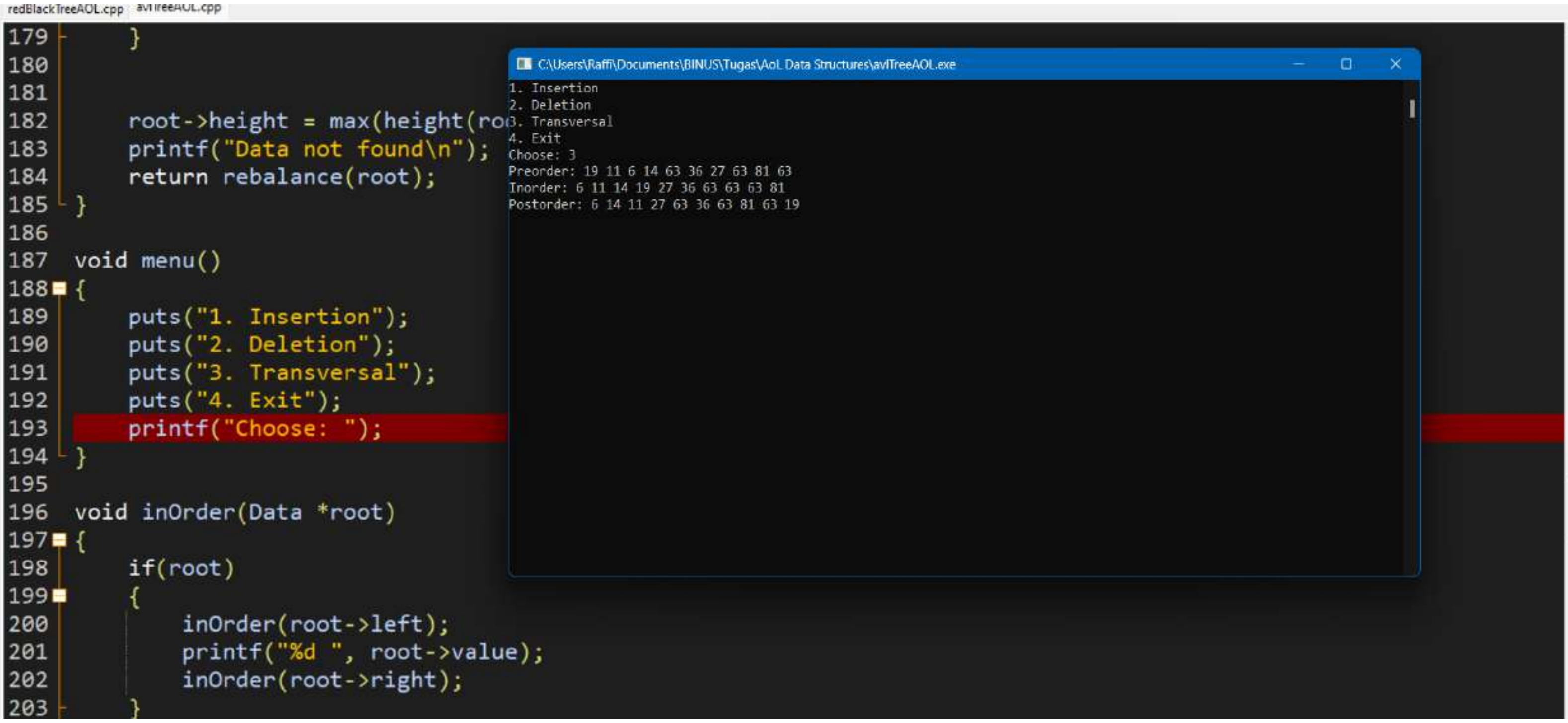
```
    }


    root->height = max(height(ro
    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);
    }
```

```
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
C:\Users\Raffi\Documents\BINUS\Tugas\AoL Data Structures\avlTreeAOL.exe

```
    }


    root->height = max(height(ro
    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);
    }
```

```
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
C:\Users\Raffi\Documents\BINUS\Tugas\AoL Data Structures\avlTreeAOL.exe

```
    }


    root->height = max(height(ro
    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);
    }
```

```
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 63 81
Postorder: 6 14 11 27 63 36 63 81 63 19
```