Date: 04/03/2021

Practical 5:

2CSDE075 - Advanced Data Structures

Name: Shrey Viradiya

Roll No: 18BCE259

Aim:

Write a program to split a balance search tree at

1. Root
2. A given point of split.

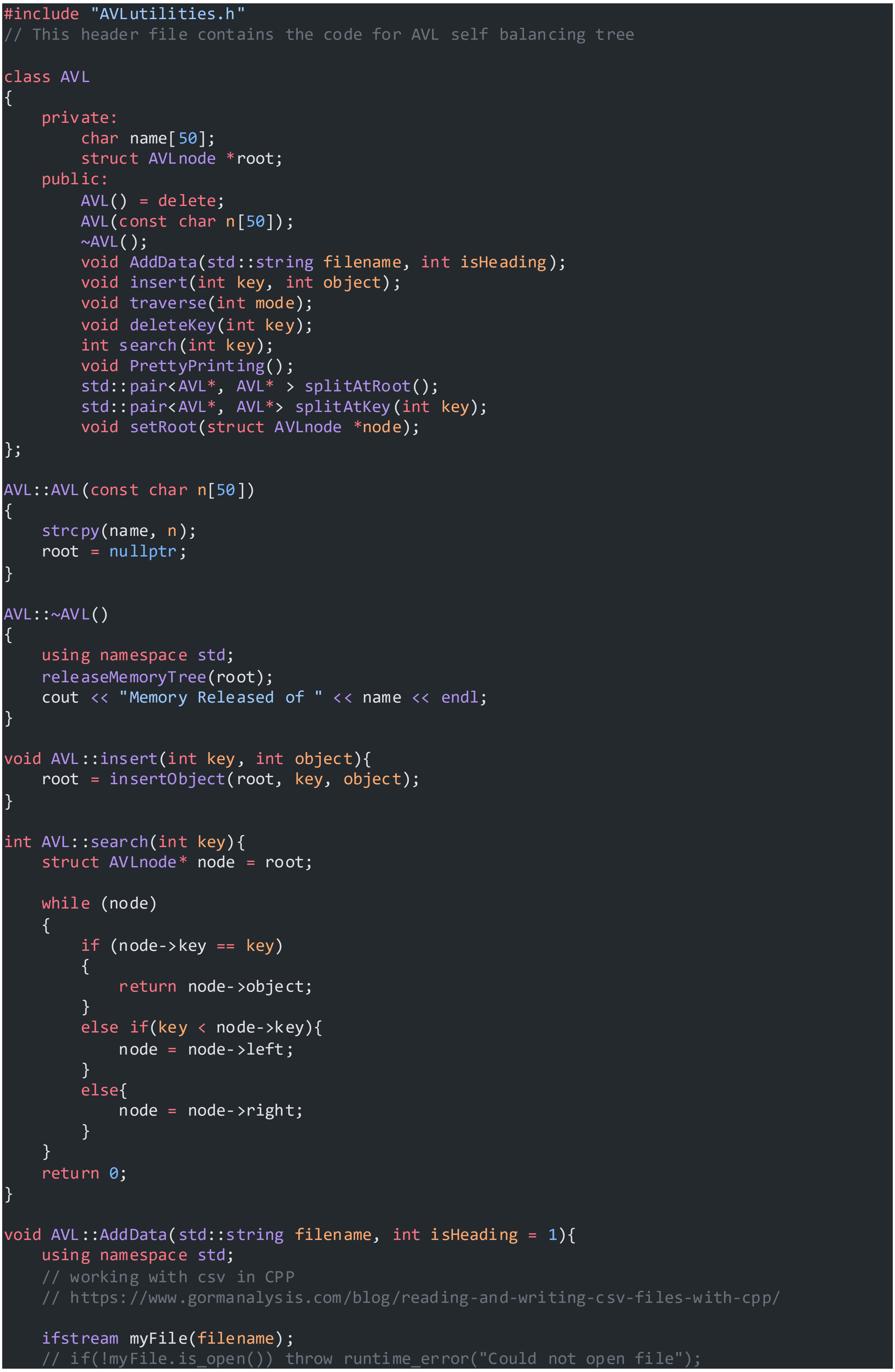
Code:

Prac5_Splitting.cpp

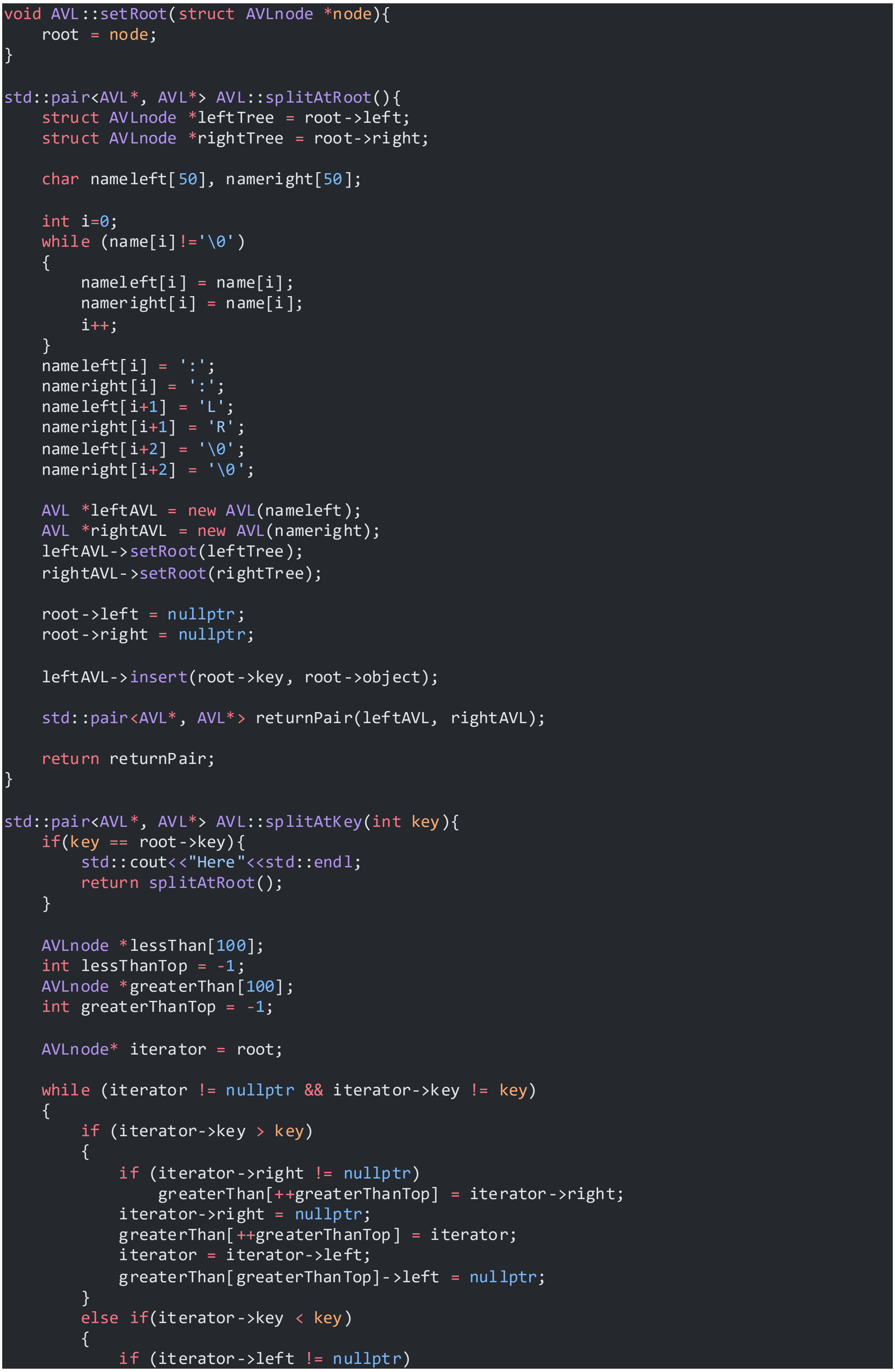
```cpp
// Write a program to split a balance search tree at
// i. Root
// ii. A given point of split.

#include "AVL.h"
#include <iostream>

int main(){

    AVL data("Ajino Motado");

    for (int i = 0; i < 10; i++)
    {
        data.insert(i, i*10);
    }

    data.PrettyPrinting();

    std::pair<AVL*, AVL*> temp = data.splitAtRoot();

    temp.first->PrettyPrinting();
    temp.second->PrettyPrinting();

    delete temp.first;
    delete temp.second;

    for (int j = 0; j < 10; j++)
    {
        AVL data2("Carume Satado");

        for (int i = 0; i < 10; i++)
        {
            data2.insert(i, i*10);
        }

        data2.PrettyPrinting();

        std::pair<AVL*, AVL*> temp2 = data2.splitAtKey(j);

        std::cout << "Splitting at " << j << std::endl;
        temp2.first->PrettyPrinting();
        temp2.second->PrettyPrinting();

        delete temp2.first;
        delete temp2.second;
    }
    return 0;
}
```

AVL.h

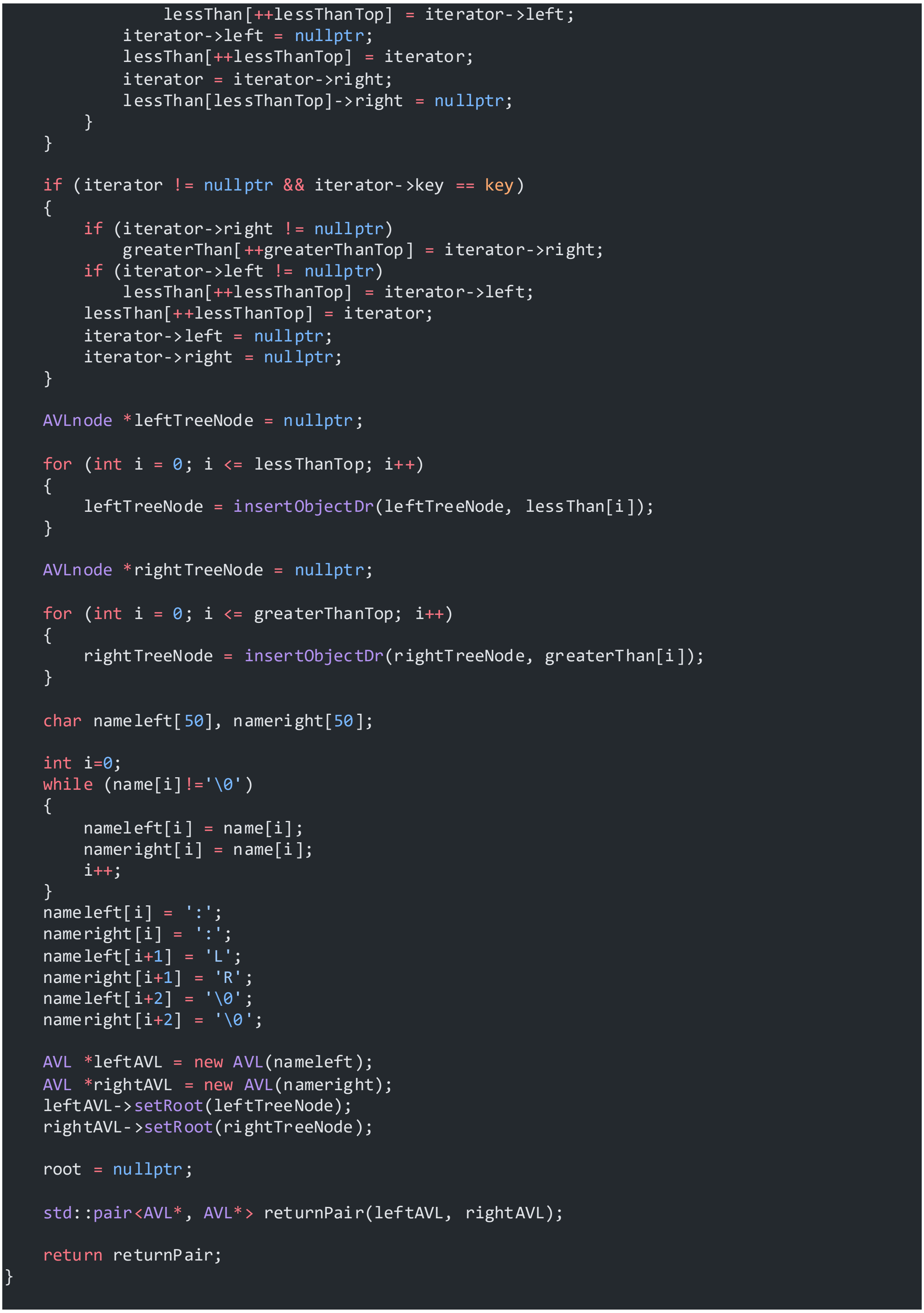
```cpp
#pragma once
#include <iostream>
#include <cstring>
#include <string>
#include <fstream>
#include <vector>
#include <stdexcept> // std::runtime_error
#include <sstream>
```



```cpp
#include "AVLutilities.h"
// This header file contains the code for AVL self balancing tree

class AVL
{
    private:
        char name[50];
        struct AVLnode *root;
    public:
        AVL() = delete;
        AVL(const char n[50]);
        ~AVL();
        void AddData(std::string filename, int isHeading);
        void insert(int key, int object);
        void traverse(int mode);
        void deleteKey(int key);
        int search(int key);
        void PrettyPrinting();
        std::pair<AVL*, AVL* > splitAtRoot();
        std::pair<AVL*, AVL*> splitAtKey(int key);
        void setRoot(struct AVLnode *node);
};

AVL::AVL(const char n[50])
{
    strcpy(name, n);
    root = nullptr;
}

AVL::~AVL()
{
    using namespace std;
    releaseMemoryTree(root);
    cout << "Memory Released of " << name << endl;
}

void AVL::insert(int key, int object){
    root = insertObject(root, key, object);
}

int AVL::search(int key){
    struct AVLnode* node = root;

    while (node)
    {
        if (node->key == key)
        {
            return node->object;
        }
        else if(key < node->key){
            node = node->left;
        }
        else{
            node = node->right;
        }
    }
    return 0;
}

void AVL::AddData(std::string filename, int isHeading = 1){
    using namespace std;
    // working with csv in CPP
    // https://www.gormanalysis.com/blog/reading-and-writing-csv-files-with-cpp/

    ifstream myFile(filename);
    // if(!myFile.is_open()) throw runtime_error("Could not open file");
    string line, word;
    int val;

    if (isHeading)  getline(myFile, line);

    // Read data, line by line
    while(getline(myFile, line))
    {
        // Create a stringstream of the current line
        stringstream ss(line);
        pair<int, int> data;

        // add the column data
        // of a row to a pair
        getline(ss, word, ',');
        data.first = stoi(word);

        getline(ss, word, ',');
        data.second = stoi(word);

        insert(data.first, data.second);
    }

    // Close file
    myFile.close();
}

void AVL::traverse(int mode = 1){
    using namespace std;

    cout << "\n\nPrinting The AVL tree: " << name << endl;
    cout << "===========================" << endl;
    cout << "Key --> Value" << endl;
    cout << "===========================" << endl;
    if (mode == 0){
        cout << "Preorder" << endl;
        traversePreorder(root);
    }
    else if (mode == 1){
        cout << "Inorder" << endl;
        traverseInorder(root);
    }
    else if (mode == 2){
        cout << "Postorder" << endl;
        traversePostorder(root);
    }
    else{
        cout << "Invalid Mode" << endl;
        cout << "Inorder" << endl;
        traverseInorder(root);
    }
    cout << "===========================\n" << endl;
}

void AVL::PrettyPrinting(){
    std::cout << "---------------------------------" << std::endl;
    std::cout << "\t" << name << std::endl;
    std::cout << "---------------------------------" << std::endl;
    printBT("", root, false);
}

void AVL::deleteKey(int key){
    root = delete_node(root, key);
}
```



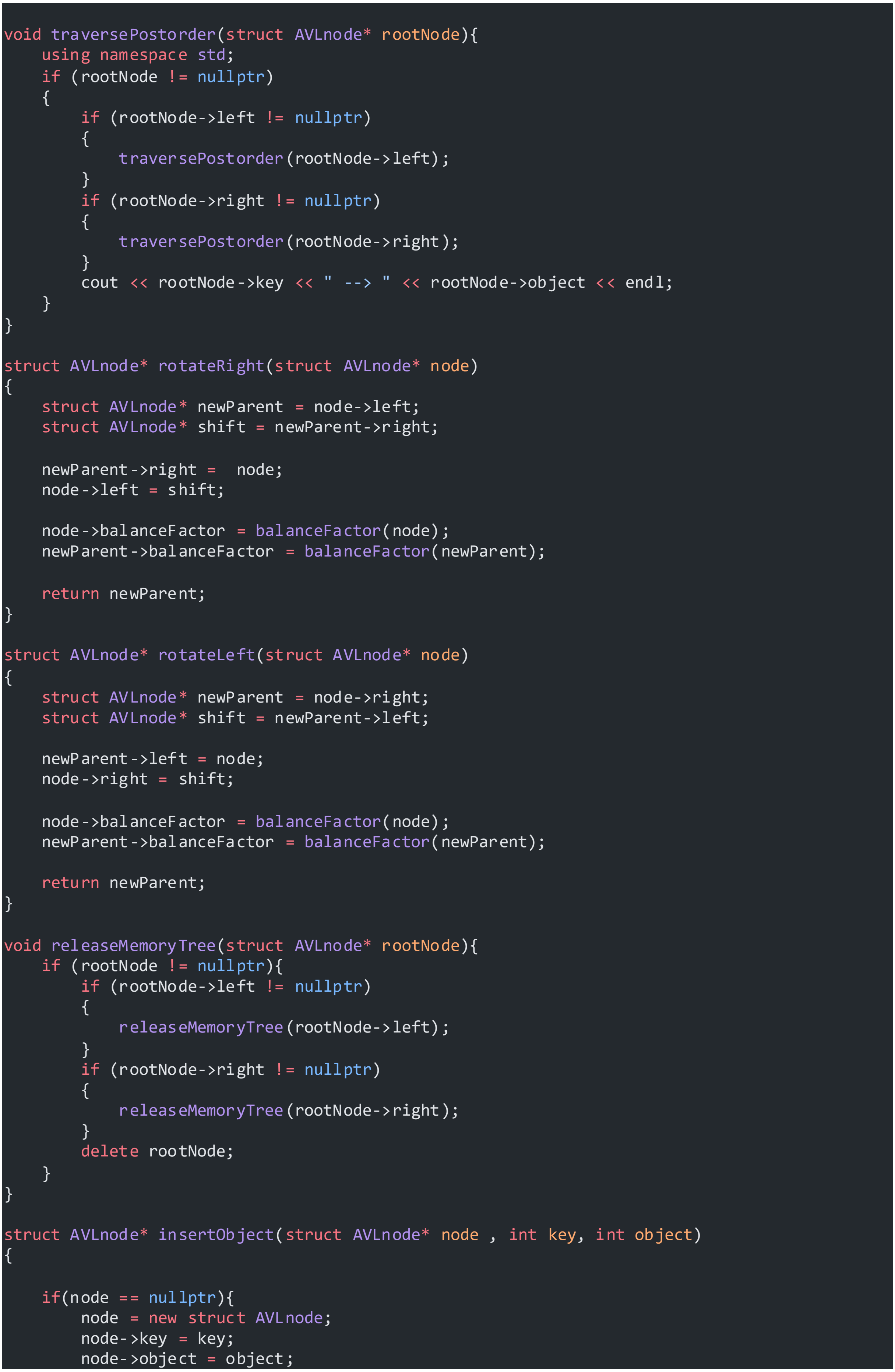
```cpp
void AVL::setRoot(struct AVLnode *node){
    root = node;
}

std::pair<AVL*, AVL*> AVL::splitAtRoot(){
    struct AVLnode *leftTree = root->left;
    struct AVLnode *rightTree = root->right;

    char nameleft[50], nameright[50];

    int i=0;
    while (name[i]!='\0')
    {
        nameleft[i] = name[i];
        nameright[i] = name[i];
        i++;
    }
    nameleft[i] = ':';
    nameright[i] = ':';
    nameleft[i+1] = 'L';
    nameright[i+1] = 'R';
    nameleft[i+2] = '\0';
    nameright[i+2] = '\0';

    AVL *leftAVL = new AVL(nameleft);
    AVL *rightAVL = new AVL(nameright);
    leftAVL->setRoot(leftTree);
    rightAVL->setRoot(rightTree);

    root->left = nullptr;
    root->right = nullptr;

    leftAVL->insert(root->key, root->object);

    std::pair<AVL*, AVL*> returnPair(leftAVL, rightAVL);

    return returnPair;
}

std::pair<AVL*, AVL*> AVL::splitAtKey(int key){
    if(key == root->key){
        std::cout<<"Here"<<std::endl;
        return splitAtRoot();
    }

    AVLnode *lessThan[100];
    int lessThanTop = -1;
    AVLnode *greaterThan[100];
    int greaterThanTop = -1;

    AVLnode* iterator = root;

    while (iterator != nullptr && iterator->key != key)
    {
        if (iterator->key > key)
        {
            if (iterator->right != nullptr)
                greaterThan[++greaterThanTop] = iterator->right;
            iterator->right = nullptr;
            greaterThan[++greaterThanTop] = iterator;
            iterator = iterator->left;
            greaterThan[greaterThanTop]->left = nullptr;
        }
        else if(iterator->key < key)
        {
            if (iterator->left != nullptr)
```



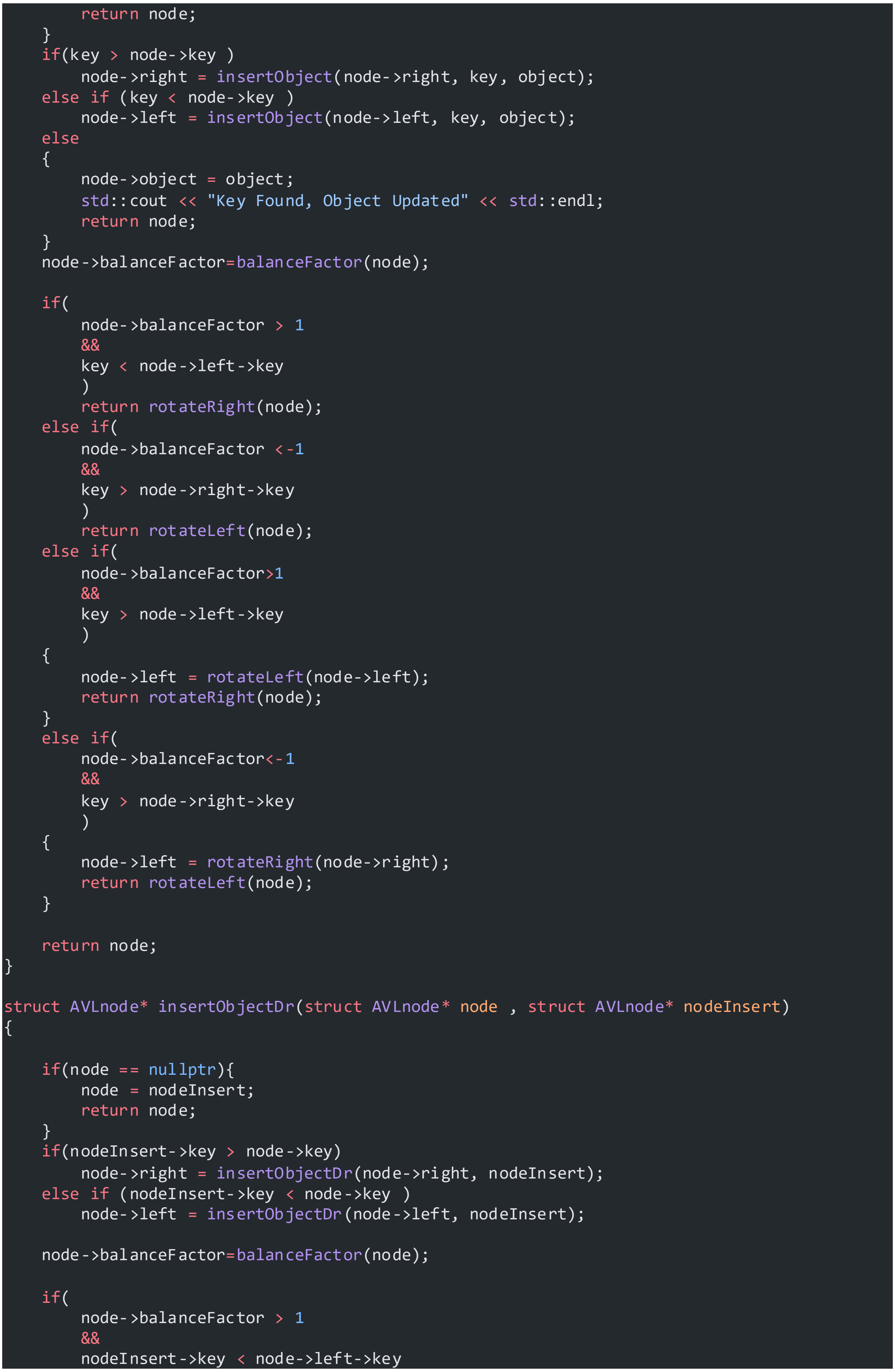
```cpp
                lessThan[++lessThanTop] = iterator->left;
            iterator->left = nullptr;
            lessThan[++lessThanTop] = iterator;
            iterator = iterator->right;
            lessThan[lessThanTop]->right = nullptr;
        }
    }

    if (iterator != nullptr && iterator->key == key)
    {
        if (iterator->right != nullptr)
            greaterThan[++greaterThanTop] = iterator->right;
        if (iterator->left != nullptr)
            lessThan[++lessThanTop] = iterator->left;
        lessThan[++lessThanTop] = iterator;
        iterator->left = nullptr;
        iterator->right = nullptr;
    }

    AVLnode *leftTreeNode = nullptr;

    for (int i = 0; i <= lessThanTop; i++)
    {
        leftTreeNode = insertObjectDr(leftTreeNode, lessThan[i]);
    }

    AVLnode *rightTreeNode = nullptr;

    for (int i = 0; i <= greaterThanTop; i++)
    {
        rightTreeNode = insertObjectDr(rightTreeNode, greaterThan[i]);
    }

    char nameleft[50], nameright[50];

    int i=0;
    while (name[i]!='\0')
    {
        nameleft[i] = name[i];
        nameright[i] = name[i];
        i++;
    }
    nameleft[i] = ':';
    nameright[i] = ':';
    nameleft[i+1] = 'L';
    nameright[i+1] = 'R';
    nameleft[i+2] = '\0';
    nameright[i+2] = '\0';

    AVL *leftAVL = new AVL(nameleft);
    AVL *rightAVL = new AVL(nameright);
    leftAVL->setRoot(leftTreeNode);
    rightAVL->setRoot(rightTreeNode);

    root = nullptr;

    std::pair<AVL*, AVL*> returnPair(leftAVL, rightAVL);

    return returnPair;
}
```

AVLutilities.h

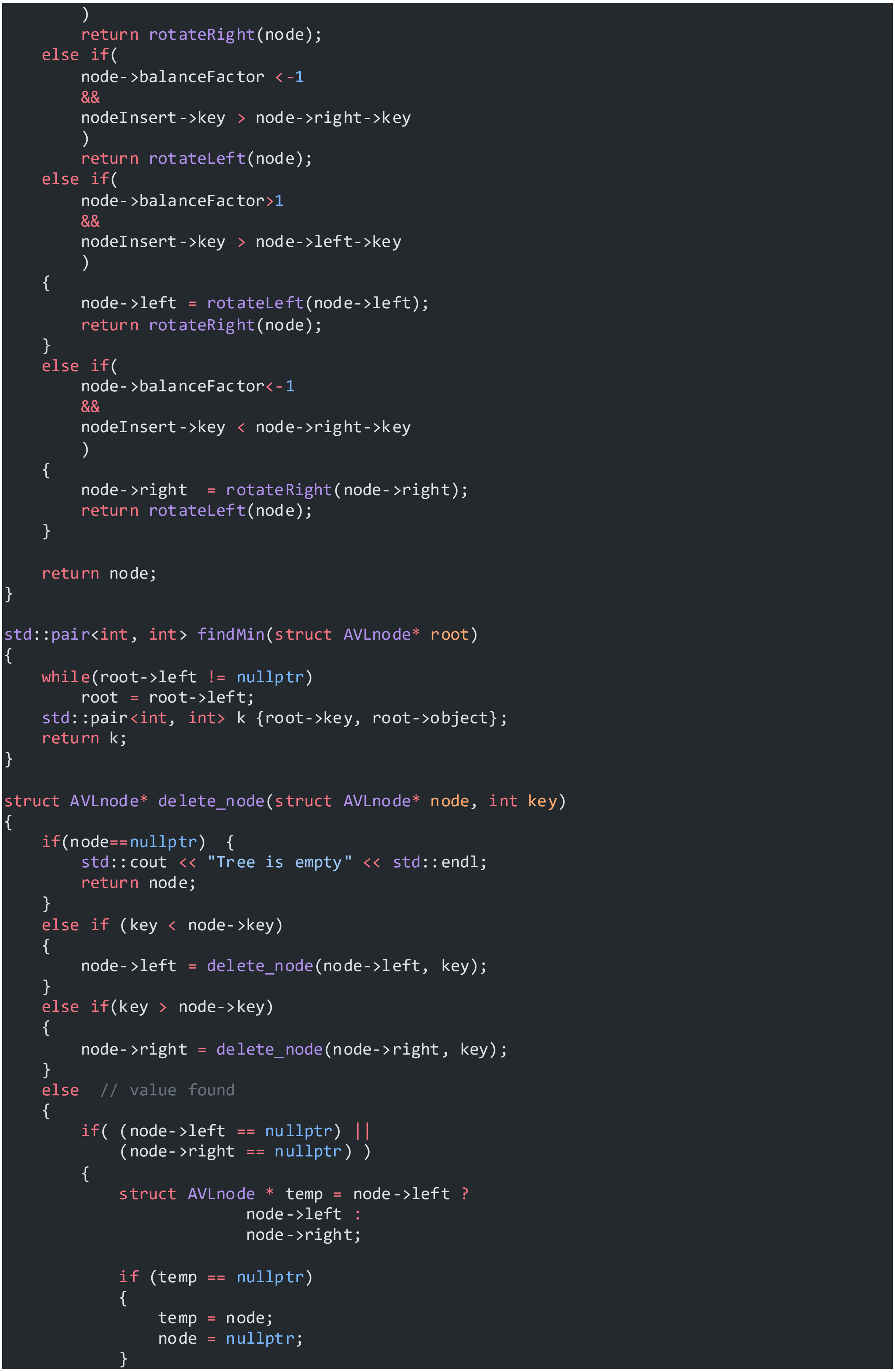
```
#pragma once
#include<iostream>
#include<utility>

struct AVLnode
{
    int key;
    int object;
    struct AVLnode *left = nullptr;
    struct AVLnode *right = nullptr;
    int balanceFactor = 0;
};

int height(struct AVLnode* node){
    if(node==nullptr)
        return 0;
    else
    {
        int lh = height(node->left);
        int rh = height(node->right);
        if (lh>rh)
            return lh+1;
        else
            return rh+1;
    }
}

int balanceFactor(struct AVLnode* node)
{
    return (height(node->left)-height(node->right));
}

void traversePreorder(struct AVLnode* rootNode){
    using namespace std;
    if (rootNode != nullptr)
    {
        cout << rootNode->key << " --> " << rootNode->object << endl;
        if (rootNode->left != nullptr)
        {
            traversePreorder(rootNode->left);
        }
        if (rootNode->right != nullptr)
        {
            traversePreorder(rootNode->right);
        }
    }
}

void traverseInorder(struct AVLnode* rootNode){
    using namespace std;
    if (rootNode != nullptr)
    {
        if (rootNode->left != nullptr)
        {
            traverseInorder(rootNode->left);
        }
        cout << rootNode->key << " --> " << rootNode->object << endl;
        if (rootNode->right != nullptr)
        {
            traverseInorder(rootNode->right);
        }
    }
}
```



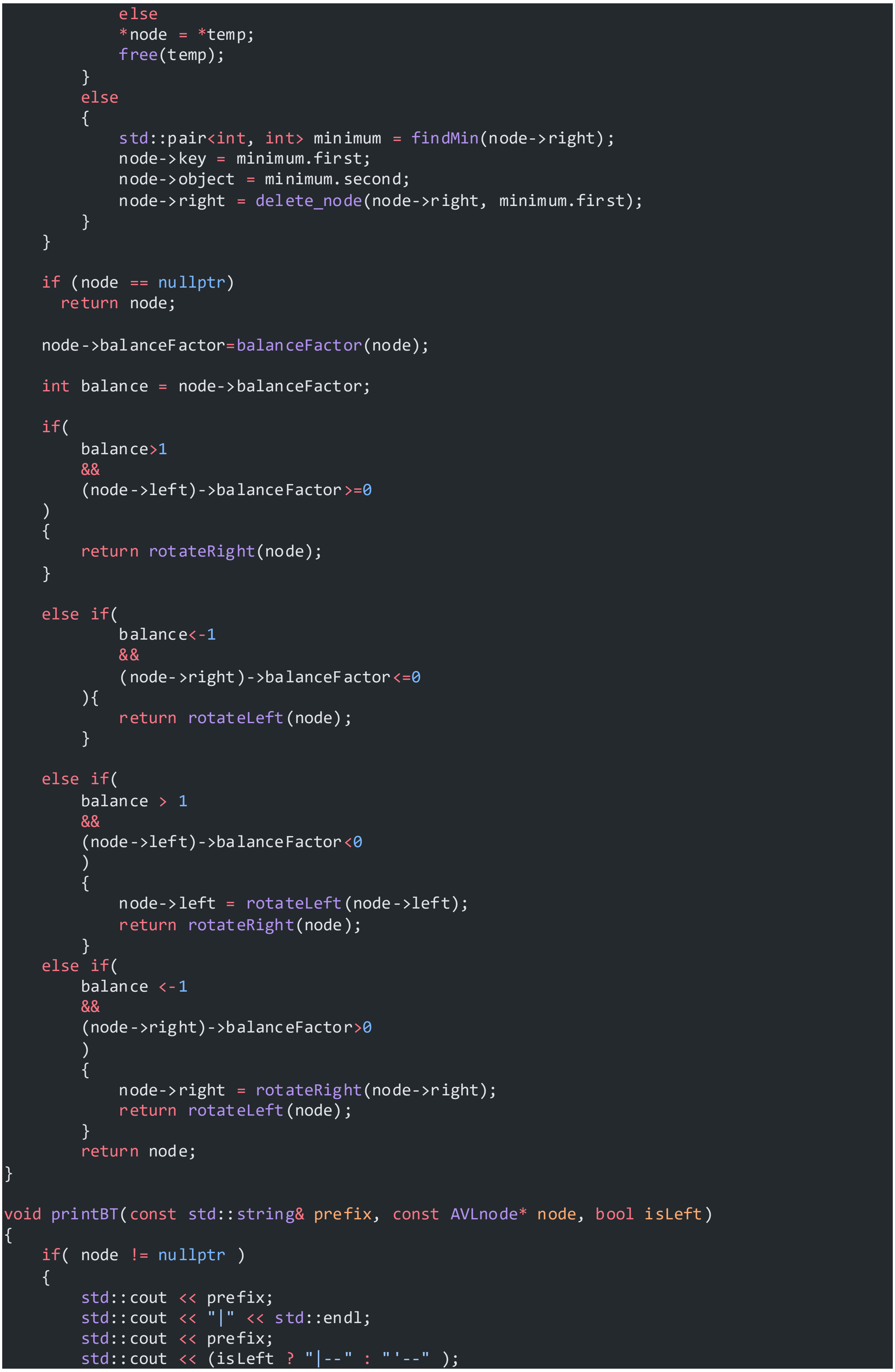
```
void traversePostorder(struct AVLnode* rootNode){
    using namespace std;
    if (rootNode != nullptr)
    {
        if (rootNode->left != nullptr)
        {
            traversePostorder(rootNode->left);
        }
        if (rootNode->right != nullptr)
        {
            traversePostorder(rootNode->right);
        }
        cout << rootNode->key << " --> " << rootNode->object << endl;
    }
}

struct AVLnode* rotateRight(struct AVLnode* node)
{
    struct AVLnode* newParent = node->left;
    struct AVLnode* shift = newParent->right;

    newParent->right =  node;
    node->left = shift;

    node->balanceFactor = balanceFactor(node);
    newParent->balanceFactor = balanceFactor(newParent);

    return newParent;
}

struct AVLnode* rotateLeft(struct AVLnode* node)
{
    struct AVLnode* newParent = node->right;
    struct AVLnode* shift = newParent->left;

    newParent->left = node;
    node->right = shift;

    node->balanceFactor = balanceFactor(node);
    newParent->balanceFactor = balanceFactor(newParent);

    return newParent;
}

void releaseMemoryTree(struct AVLnode* rootNode){
    if (rootNode != nullptr){
        if (rootNode->left != nullptr)
        {
            releaseMemoryTree(rootNode->left);
        }
        if (rootNode->right != nullptr)
        {
            releaseMemoryTree(rootNode->right);
        }
        delete rootNode;
    }
}

struct AVLnode* insertObject(struct AVLnode* node , int key, int object)
{

    if(node == nullptr){
        node = new struct AVLnode;
        node->key = key;
        node->object = object;
```



```
        return node;
    }
    if(key > node->key )
        node->right = insertObject(node->right, key, object);
    else if (key < node->key )
        node->left = insertObject(node->left, key, object);
    else
    {
        node->object = object;
        std::cout << "Key Found, Object Updated" << std::endl;
        return node;
    }
    node->balanceFactor=balanceFactor(node);

    if(
        node->balanceFactor > 1
        &&
        key < node->left->key
        )
        return rotateRight(node);
    else if(
        node->balanceFactor <-1
        &&
        key > node->right->key
        )
        return rotateLeft(node);
    else if(
        node->balanceFactor>1
        &&
        key > node->left->key
        )
    {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }
    else if(
        node->balanceFactor<-1
        &&
        key > node->right->key
        )
    {
        node->left = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

struct AVLnode* insertObjectDr(struct AVLnode* node , struct AVLnode* nodeInsert)
{

    if(node == nullptr){
        node = nodeInsert;
        return node;
    }
    if(nodeInsert->key > node->key)
        node->right = insertObjectDr(node->right, nodeInsert);
    else if (nodeInsert->key < node->key )
        node->left = insertObjectDr(node->left, nodeInsert);

    node->balanceFactor=balanceFactor(node);

    if(
        node->balanceFactor > 1
        &&
        nodeInsert->key < node->left->key
```

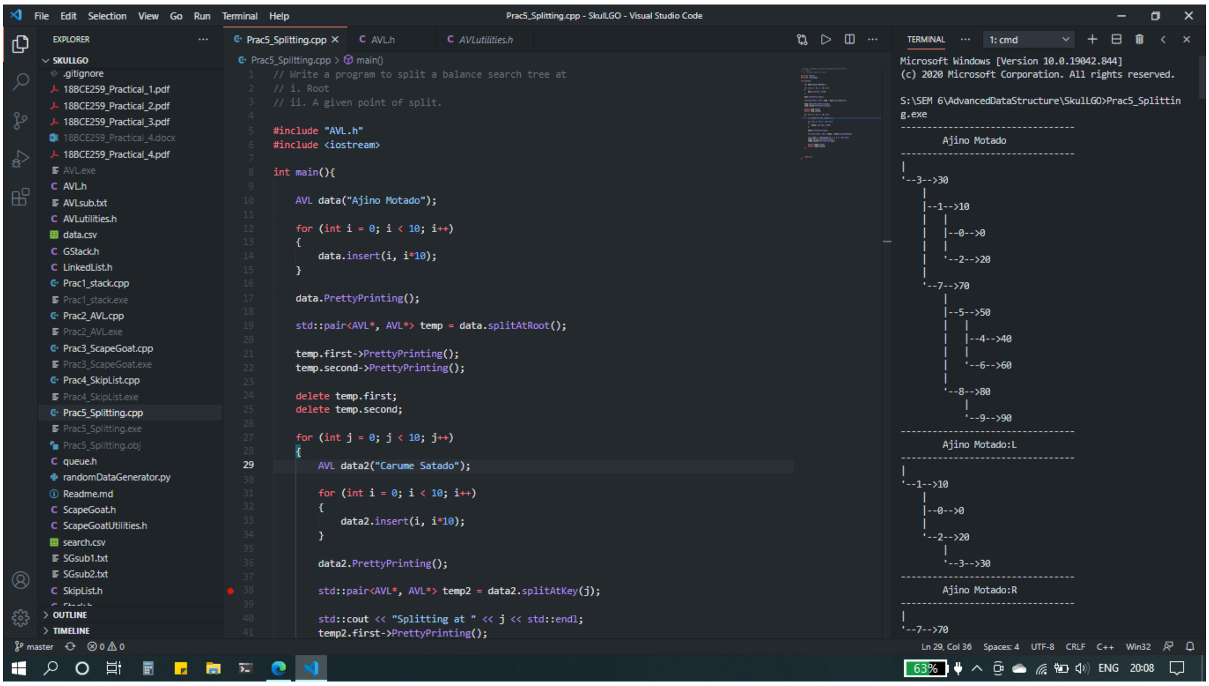


```
        )
        return rotateRight(node);
    else if(
        node->balanceFactor <-1
        &&
        nodeInsert->key > node->right->key
        )
        return rotateLeft(node);
    else if(
        node->balanceFactor>1
        &&
        nodeInsert->key > node->left->key
        )
    {
        node->left = rotateLeft(node->left);
        return rotateRight(node);
    }
    else if(
        node->balanceFactor<-1
        &&
        nodeInsert->key < node->right->key
        )
    {
        node->right  = rotateRight(node->right);
        return rotateLeft(node);
    }

    return node;
}

std::pair<int, int> findMin(struct AVLnode* root)
{
    while(root->left != nullptr)
        root = root->left;
    std::pair<int, int> k {root->key, root->object};
    return k;
}

struct AVLnode* delete_node(struct AVLnode* node, int key)
{
    if(node==nullptr)  {
        std::cout << "Tree is empty" << std::endl;
        return node;
    }
    else if (key < node->key)
    {
        node->left = delete_node(node->left, key);
    }
    else if(key > node->key)
    {
        node->right = delete_node(node->right, key);
    }
    else  // value found
    {
        if( (node->left == nullptr) ||
            (node->right == nullptr) )
        {
            struct AVLnode * temp = node->left ?
                         node->left :
                         node->right;

            if (temp == nullptr)
            {
                temp = node;
                node = nullptr;
            }
```



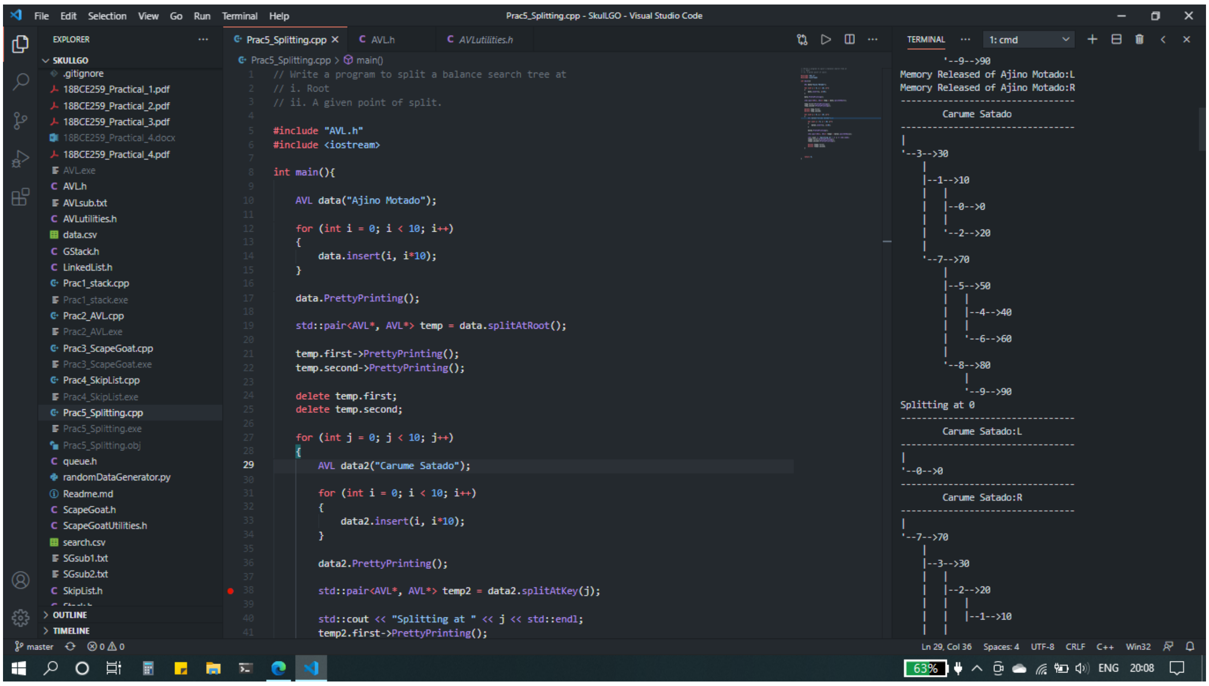
```
            else
            *node = *temp;
            free(temp);
        }
        else
        {
            std::pair<int, int> minimum = findMin(node->right);
            node->key = minimum.first;
            node->object = minimum.second;
            node->right = delete_node(node->right, minimum.first);
        }
    }

    if (node == nullptr)
      return node;

    node->balanceFactor=balanceFactor(node);

    int balance = node->balanceFactor;

    if(
        balance>1
        &&
        (node->left)->balanceFactor>=0
    )
    {
        return rotateRight(node);
    }

    else if(
            balance<-1
            &&
            (node->right)->balanceFactor<=0
        ){
            return rotateLeft(node);
        }

    else if(
        balance > 1
        &&
        (node->left)->balanceFactor<0
        )
        {
            node->left = rotateLeft(node->left);
            return rotateRight(node);
        }
    else if(
        balance <-1
        &&
        (node->right)->balanceFactor>0
        )
        {
            node->right = rotateRight(node->right);
            return rotateLeft(node);
        }
        return node;
}

void printBT(const std::string& prefix, const AVLnode* node, bool isLeft)
{
    if( node != nullptr )
    {
        std::cout << prefix;
        std::cout << "|" << std::endl;
        std::cout << prefix;
        std::cout << (isLeft ? "|--" : "'--" );
        // print the value of the node
        std::cout << node->key << "-->" << node->object << std::endl;

        // enter the next tree level - left and right branch
        printBT( prefix + (isLeft ? "|   " : "    ") , node->left, true);
        printBT( prefix + (isLeft ? "|   " : "    ") , node->right, false);
    }
}
```

Snapshot of the output:



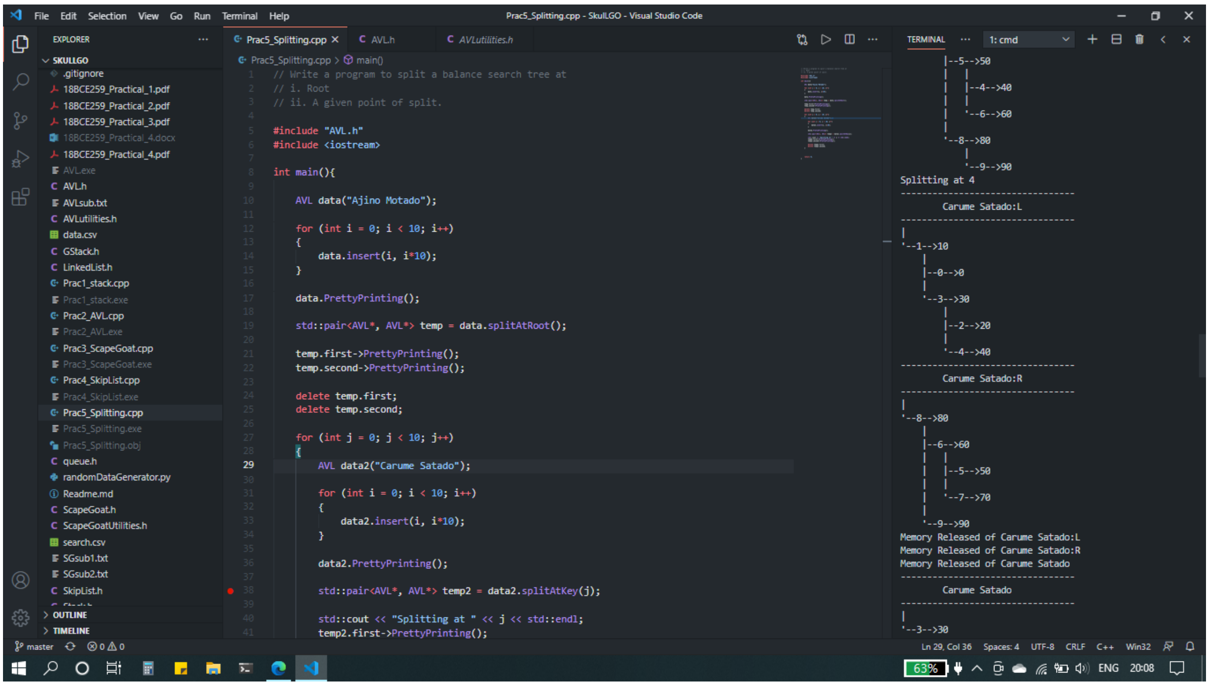
```cpp
// Write a program to split a balance search tree at
// i. Root
// ii. A given point of split.

#include "AVL.h"
#include <iostream>

int main(){

    AVL data("Ajino Motado");

    for (int i = 0; i < 10; i++)
    {
        data.insert(i, i*10);
    }

    data.PrettyPrinting();

    std::pair<AVL*, AVL*> temp = data.splitAtRoot();

    temp.first->PrettyPrinting();
    temp.second->PrettyPrinting();

    delete temp.first;
    delete temp.second;

    for (int j = 0; j < 10; j++)
    {
        AVL data2("Carume Satado");

        for (int i = 0; i < 10; i++)
        {
            data2.insert(i, i*10);
        }

        data2.PrettyPrinting();

        std::pair<AVL*, AVL*> temp2 = data2.splitAtKey(j);

        std::cout << "Splitting at " << j << std::endl;
        temp2.first->PrettyPrinting();
```

```
Microsoft Windows [Version 10.0.19042.844]
(c) 2020 Microsoft Corporation. All rights reserved.

S:\SEM 6\AdvancedDataStructure\SkullGO>Prac5_Splittin
g.exe
---------------------------------
        Ajino Motado
---------------------------------
|
'--3-->30
    |
    |--1-->10
    |   |
    |   |--0-->0
    |   |
    |   '--2-->20
    |
    '--7-->70
        |
        |--5-->50
        |   |
        |   |--4-->40
        |   |
        |   '--6-->60
        |
        '--8-->80
            |
            '--9-->90
---------------------------------
        Ajino Motado:L
---------------------------------
|
'--1-->10
    |
    |--0-->0
    |
    '--2-->20
        |
        '--3-->30
---------------------------------
        Ajino Motado:R
---------------------------------
|
'--7-->70
```


File Edit Selection View Go Run Terminal Help
Prac5_Splitting.cpp - SkullGO - Visual Studio Code
EXPLORER
SKULLGO
TERMINAL
1: cmd
master
Ln 29, Col 36 Spaces: 4 UTF-8 CRLF C++ Win32
63%
ENG 20:08

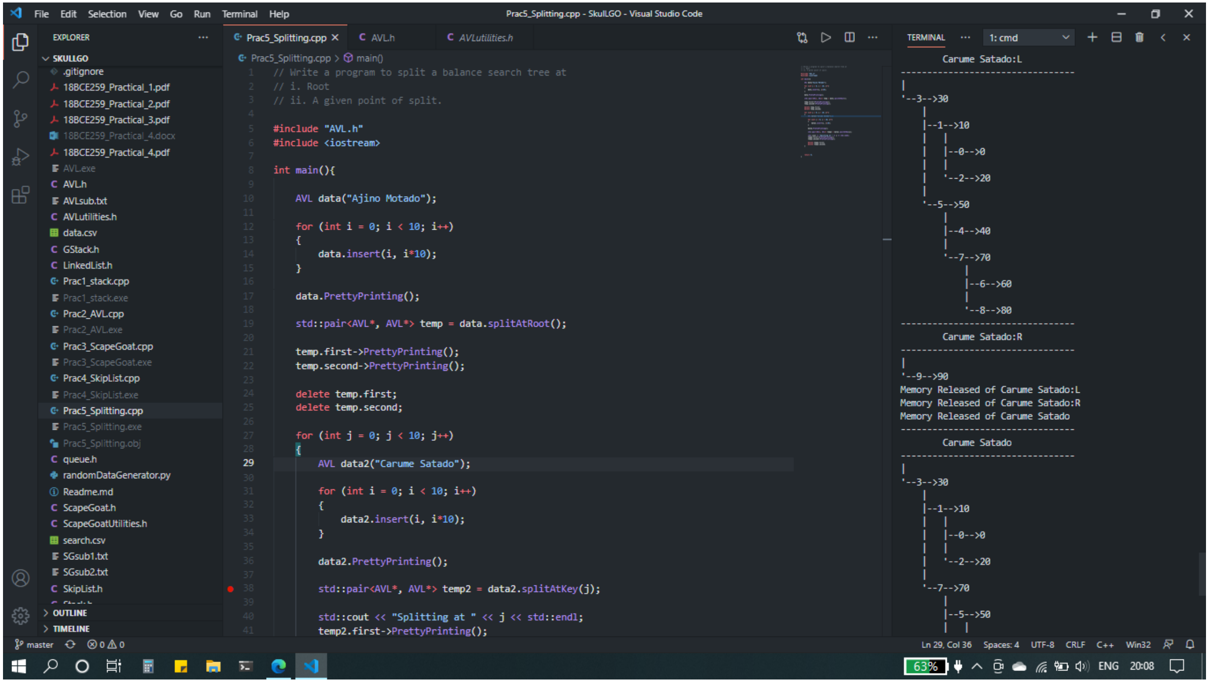
```cpp
// Write a program to split a balance search tree at
// i. Root
// ii. A given point of split.

#include "AVL.h"
#include <iostream>

int main(){

    AVL data("Ajino Motado");

    for (int i = 0; i < 10; i++)
    {
        data.insert(i, i*10);
    }

    data.PrettyPrinting();

    std::pair<AVL*, AVL*> temp = data.splitAtRoot();

    temp.first->PrettyPrinting();
    temp.second->PrettyPrinting();

    delete temp.first;
    delete temp.second;

    for (int j = 0; j < 10; j++)
    {
        AVL data2("Carume Satado");

        for (int i = 0; i < 10; i++)
        {
            data2.insert(i, i*10);
        }

        data2.PrettyPrinting();

        std::pair<AVL*, AVL*> temp2 = data2.splitAtKey(j);

        std::cout << "Splitting at " << j << std::endl;
        temp2.first->PrettyPrinting();
```

```
        '--9-->90
Memory Released of Ajino Motado:L
Memory Released of Ajino Motado:R
---------------------------------
        Carume Satado
---------------------------------
|
'--3-->30
    |
    |--1-->10
    |   |
    |   |--0-->0
    |   |
    |   '--2-->20
    |
    '--7-->70
        |
        |--5-->50
        |   |
        |   |--4-->40
        |   |
        |   '--6-->60
        |
        '--8-->80
            |
            '--9-->90
Splitting at 0
---------------------------------
        Carume Satado:L
---------------------------------
|
'--0-->0
---------------------------------
        Carume Satado:R
---------------------------------
|
'--7-->70
    |
    |--3-->30
    |   |
    |   |--2-->20
    |   |   |
    |   |   |--1-->10
    |   |
```



```cpp
// Write a program to split a balance search tree at
// i. Root
// ii. A given point of split.

#include "AVL.h"
#include <iostream>

int main(){

    AVL data("Ajino Motado");

    for (int i = 0; i < 10; i++)
    {
        data.insert(i, i*10);
    }

    data.PrettyPrinting();

    std::pair<AVL*, AVL*> temp = data.splitAtRoot();

    temp.first->PrettyPrinting();
    temp.second->PrettyPrinting();

    delete temp.first;
    delete temp.second;

    for (int j = 0; j < 10; j++)
    {
        AVL data2("Carume Satado");

        for (int i = 0; i < 10; i++)
        {
            data2.insert(i, i*10);
        }

        data2.PrettyPrinting();

        std::pair<AVL*, AVL*> temp2 = data2.splitAtKey(j);

        std::cout << "Splitting at " << j << std::endl;
        temp2.first->PrettyPrinting();
```

```
        |--5-->50
        |   |
        |   |--4-->40
        |   |
        |   '--6-->60
        |
        '--8-->80
            |
            '--9-->90
Splitting at 4
---------------------------------
        Carume Satado:L
---------------------------------
|
'--1-->10
    |
    |--0-->0
    |
    '--3-->30
        |
        |--2-->20
        |
        '--4-->40
---------------------------------
        Carume Satado:R
---------------------------------
|
'--8-->80
    |
    |--6-->60
    |   |
    |   |--5-->50
    |   |
    |   '--7-->70
    |
    '--9-->90
Memory Released of Carume Satado:L
Memory Released of Carume Satado:R
Memory Released of Carume Satado
---------------------------------
        Carume Satado
---------------------------------
|
'--3-->30
```



```cpp
// Write a program to split a balance search tree at
// i. Root
// ii. A given point of split.

#include "AVL.h"
#include <iostream>

int main(){

    AVL data("Ajino Motado");

    for (int i = 0; i < 10; i++)
    {
        data.insert(i, i*10);
    }

    data.PrettyPrinting();

    std::pair<AVL*, AVL*> temp = data.splitAtRoot();

    temp.first->PrettyPrinting();
    temp.second->PrettyPrinting();

    delete temp.first;
    delete temp.second;

    for (int j = 0; j < 10; j++)
    {
        AVL data2("Carume Satado");

        for (int i = 0; i < 10; i++)
        {
            data2.insert(i, i*10);
        }

        data2.PrettyPrinting();

        std::pair<AVL*, AVL*> temp2 = data2.splitAtKey(j);

        std::cout << "Splitting at " << j << std::endl;
        temp2.first->PrettyPrinting();
```

```
        Carume Satado:L
---------------------------------
|
'--3-->30
    |
    |--1-->10
    |   |
    |   |--0-->0
    |   |
    |   '--2-->20
    |
    '--5-->50
        |
        |--4-->40
        |
        '--7-->70
            |
            |--6-->60
            |
            '--8-->80
---------------------------------
        Carume Satado:R
---------------------------------
|
'--9-->90
Memory Released of Carume Satado:L
Memory Released of Carume Satado:R
Memory Released of Carume Satado
---------------------------------
        Carume Satado
---------------------------------
|
'--3-->30
    |
    |--1-->10
    |   |
    |   |--0-->0
    |   |
    |   '--2-->20
    |
    '--7-->70
        |
        |--5-->50
        |   |
```

Conclusion:

By Tree Splitting, we can get two different trees with the same properties of the original one. One might want to split a tree to reduce overhead in the system if it contains a lot of elements.