Multi Level Hashing With Separate Chaining using AVL trees

An Innovative Assignment Report

Submitted for Department Elective Course

Advanced Data Structures 2CSDE75

By

Shrey Viradiya, 18BCE259
Computer Science & Engineering
Institute of Technology
Nirma University

Vishva Patel, 18BCE260
Computer Science & Engineering
Institute of Technology
Nirma University

NIRMA UNIVERSITY

INSTITUTE OF TECHNOLOGY

NAAC ACCREDITED 'A' GRADE

Computer science and engineering department
Institute of Technology, Nirma University
Ahmedabad-382481
November 2020

Table of contents

I. Code

1. AVL.h

```
#pragma once
#include <iostream>
#include <cstring>
#include <string>
#include <fstream>
#include <stdexcept> // std::runtime_error
#include <sstream>
#include "AVLutilities.h"
// This header file contains the code for AVL self balancing tree

class AVL
{
private:
    struct AVLnode* root;
public:
    AVL();
    ~AVL();
    void AddData(std::string filename, int isHeading);
    void insert(int key);
    void traverse(int mode);
    void deleteKey(int key);
    int search(int key);
    void PrettyPrinting();
    int size();
};

AVL::AVL()
{
    root = nullptr;
}

AVL::~AVL()
{
    using namespace std;
    releaseMemoryTree(root);
```

```
}

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

int AVL::size() {
    return count(root);
}

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

    while (node)
    {
        if (node->key == key)
        {
            return 1;
        }
        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);
    }

    // Close file
    myFile.close();
}

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

    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;
    printBT("", root, false);
}

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

2. AVLutilities.h

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

struct AVLnode
{
    int key;
    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 << 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 << 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 << 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)
{

    if (node == nullptr) {
        node = new struct AVLnode;
        node->key = key;
        return node;
    }
    if (key > node->key)
        node->right = insertObject(node->right, key);
    else if (key < node->key)
        node->left = insertObject(node->left, key);
    else
    {
        //std::cout << "Key Already Found" << 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;
}

int findMin(struct AVLnode* root)
{
    while (root->left != nullptr)
        root = root->left;
    return root->key;
}

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
        {
            int minimum = findMin(node->right);
            node->key = minimum;
            node->right = delete_node(node->right, minimum);
        }
    }

    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 << std::endl;

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

int count(AVLnode* tree)
{
    int c = 1;              //Node itself should be counted
    if (tree == nullptr)
        return 0;
    else
    {
        c += count(tree->left);
        c += count(tree->right);
        return c;
    }
}
```

3. FourLevelHash.h

```
#pragma once
#include <iostream>
#include <utility>
#include <vector>
#include <list>
#include "AVL.h"

using namespace std;

class FourLevelHash {
    //    list < list < list < list <long long>* >* >* > *table;
    vector<int> hashLevel;
    vector<vector<vector<vector<AVL> > > > table;
public:
    explicit FourLevelHash(vector<int> v) {
        hashLevel = std::move(v);

        table = vector<vector<vector<vector<AVL> > > >(hashLevel[0]);
        for (int i = 0; i < hashLevel[0]; i++) {
            table[i] = vector<vector<vector<AVL> > >(hashLevel[1]);
            for (int j = 0; j < hashLevel[1]; j++) {
                table[i][j] = vector<vector<AVL> >(hashLevel[2]);
                for (int k = 0; k < hashLevel[2]; k++) {
                    table[i][j][k] = vector<AVL>(hashLevel[3]);
                    for (int o = 0; o < hashLevel[3]; o++)
                    {
                        table[i][j][k][o] = AVL();
                    }
                }
            }
        }
    }

    static int hashFunction(long long x, int hashNumber) {
        return x % (long long)hashNumber;
    }

    void insertItem(long long key) {
```

```
        int id1 = hashFunction(key, hashLevel[0]);
        int id2 = hashFunction(key, hashLevel[1]);
        int id3 = hashFunction(key, hashLevel[2]);
        int id4 = hashFunction(key, hashLevel[3]);
        table[id1][id2][id3][id4].insert(key);
    }

    void displaySizes() {
        for (auto i = 0; i < hashLevel[0]; i++) {
            for (auto j = 0; j < hashLevel[1]; j++) {
                for (auto k = 0; k < hashLevel[2]; k++) {
                    for (auto o = 0; o < hashLevel[3]; o++) {
                        cout << i << "-->" << j << "-->" << k <<
"-->" << o << "-->" << table[i][j][k][o].size()
                            << endl;
                    }
                }
            }
        }
    }

    void findNumber(long long key) {
        int id1 = hashFunction(key, hashLevel[0]);
        int id2 = hashFunction(key, hashLevel[1]);
        int id3 = hashFunction(key, hashLevel[2]);
        int id4 = hashFunction(key, hashLevel[3]);

        /*int k = 0;
        std::vector<long long>::iterator i;
        for (i = table[id1][id2][id3][id4].begin(); i !=
table[id1][id2][id3][id4].end(); i++) {
            k++;
            if (*i == key)
                break;
        }*/
        int result = table[id1][id2][id3][id4].search(key);


        if (result) {
```

```
            cout << "Found the number in HashTable" << endl;
        }
        else {
            cout << "Not Found" << '\n';
        }
    }
};
```

4. SingleHash.h

```
#pragma once

#include <iostream>
#include <vector>

using namespace std;

class SingleHash {
    int buckets;
    std::vector<long long>* table;

public:
    explicit SingleHash(int v) {
        buckets = v;
        table = new vector<long long>[buckets];
    }

    void insertItem(long long key) {
        int index = hashFunction(key);
        table[index].push_back(key);
    }

    void deleteItem(long long key) {
        int index = hashFunction(key);

        std::vector<long long>::iterator i;
        for (i = table[index].begin(); i != table[index].end(); i++)
{
            if (*i == key)
                break;
        }

        if (i != table[index].end()) {
            table[index].erase(i);
        }
    }

    int hashFunction(long long x) {
```

```
        return x % (long long)buckets;
    }

    void displayHash() {
        for (int i = 0; i < buckets; i++) {
            cout << i;
            for (auto x : table[i])
                cout << " --> " << x;
            cout << endl;
        }
    }

    void displaySizes() {
        for (int i = 0; i < buckets; i++) {
            cout << i << "-->" << table[i].size() << '\n';
        }
    }

    void findNumber(long long key) {
        int index = hashFunction(key);
        int k = 0;
        std::vector<long long>::iterator i;
        for (i = table[index].begin(); i != table[index].end(); i++)
{
            k++;
            if (*i == key)
                break;
        }

        if (i != table[index].end()) {
            cout << "Found the number in list " << index << " at
index " << k << '\n';
        }
        else {
            cout << "Not Found" << '\n';
        }
    }

    long long getNumber(int index, int lst) {
```

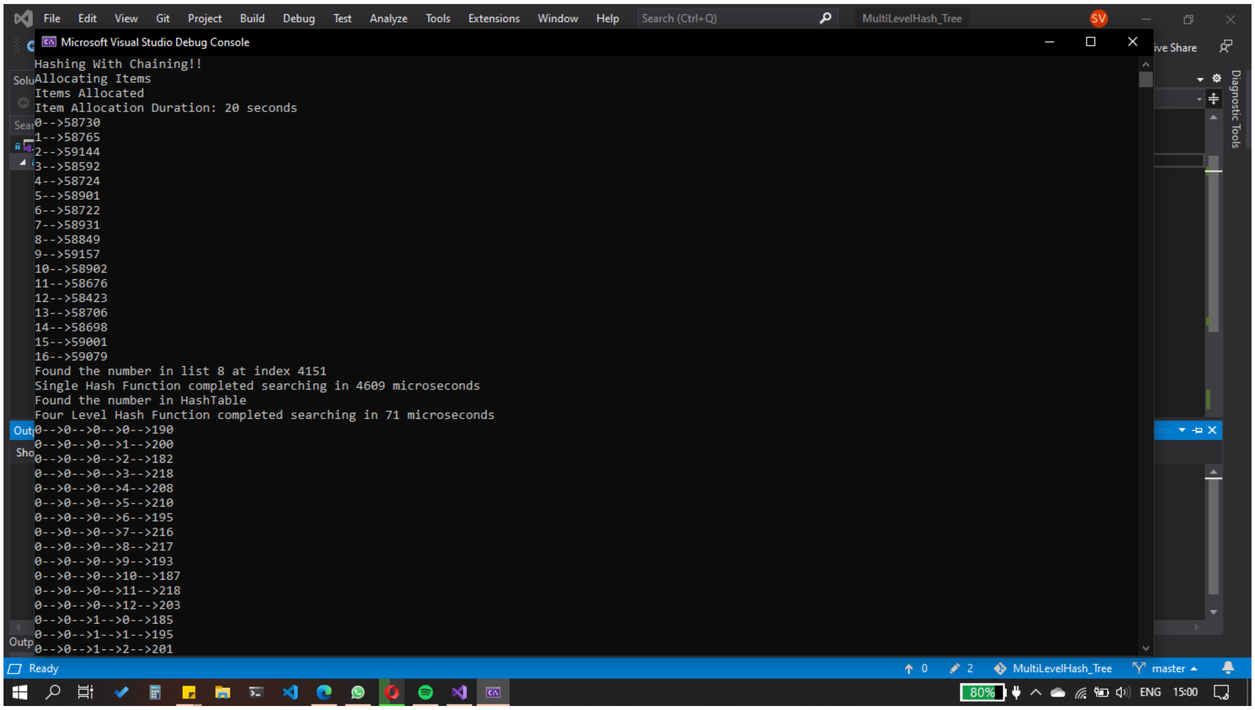
```
        auto i = table[lst].begin();
        advance(i, index - 1);
        return *i;
    }
};
```

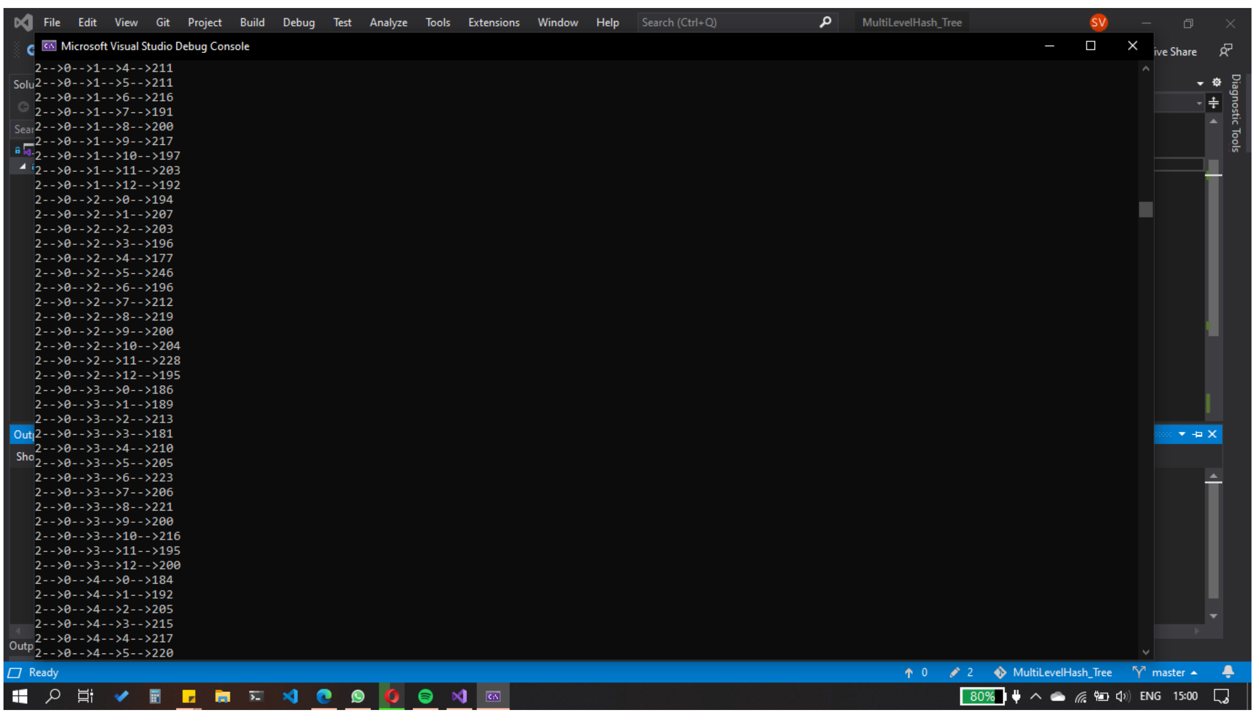
5. Main.cpp

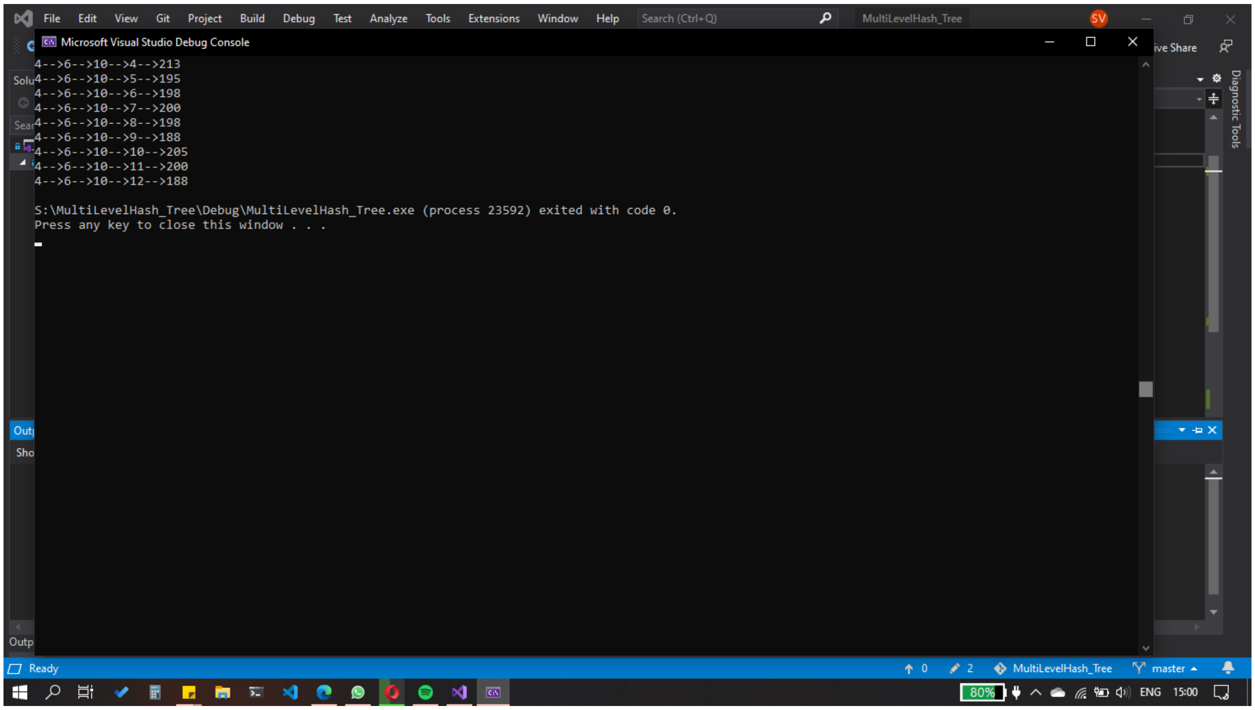
```
#include <iostream>
#include <random>
#include <chrono>
#include "SingleHash.h"
#include "FourLevelHash.h"


int main() {
    SingleHash H(17);
    vector<int> primes = { 5, 7, 17, 29 };
    FourLevelHash H4(primes);


    std::random_device rd;
    std::mt19937 mt(rd());
    std::uniform_real_distribution<double> dist(1.0, 1000000000.0);

    std::cout << "Hashing With Chaining!!" << std::endl;

    int size = 1000000;
    long long find;
    auto start1 = std::chrono::high_resolution_clock::now();

    cout << "Allocating Items" << endl;
    for (auto i = 0; i < size; i++) {
        auto temp = (long long)(dist(mt));
        if (i == 70000)
            find = temp;
        H.insertItem(temp);
        H4.insertItem(temp);
    }
    cout << "Items Allocated" << endl;
    auto stop1 = std::chrono::high_resolution_clock::now();

    auto duration1 =
std::chrono::duration_cast<std::chrono::seconds>(stop1 - start1);
    cout << "Item Allocation Duration: " << duration1.count() << "
seconds" << endl;
```

```
    H.displaySizes();
    //H4.displaySizes();

    auto start2 = std::chrono::high_resolution_clock::now();
    H.findNumber(find);
    auto stop2 = std::chrono::high_resolution_clock::now();

    auto duration2 =
std::chrono::duration_cast<std::chrono::microseconds>(stop2 -
start2);
    cout << "Single Hash Function completed searching in " <<
duration2.count() << " microseconds" << endl;

    auto start3 = std::chrono::high_resolution_clock::now();
    H4.findNumber(find);
    auto stop3 = std::chrono::high_resolution_clock::now();

    auto duration3 =
std::chrono::duration_cast<std::chrono::microseconds>(stop3 -
start3);
    cout << "Four Level Hash Function completed searching in " <<
duration3.count() << " microseconds" << endl;

    return 0;
}
```

II. Snapshot of the output







As seen in the output, the sizes of chains in the single level hash in quite high data is nearly 60000 in each. But that of each bucket in Multilevel is nearly 200. That is also handled by AVL trees. So searching for an element in SingleLevel took 4609 microseconds and in MultiLevel it took 71 microseconds. 65 times faster!!!

III. Conclusion

MultiLevel hash function is a really important data structure for handling a large number of keys. This data structure is used in database management systems to retrieve record data quickly. The given hash implementation is written in C++ with time to lookup for a given particular key.