Assignment 3: Hashing and hash functions

Graded

Student

Arushi Yadav

Total Points

86 / 100 pts

Autograder Score

86.0 / 100.0

Failed Tests

1.21) test_21 (test_simple.HashingTester) (0/7)

1.26) test_26_comp (test_simple.HashingTester) (0/7)

Passed Tests

Check submitted files (0/0)

1.1) test_1 (test_simple.HashingTester) (1/1)

1.2) test_2 (test_simple.HashingTester) (1/1)

1.3) test_3 (test_simple.HashingTester) (2/2)

1.4) test_4 (test_simple.HashingTester) (2/2)

1.5) test_5 (test_simple.HashingTester) (1/1)

1.6) test_6 (test_simple.HashingTester) (5/5)

1.7) test_7 (test_simple.HashingTester) (2/2)

1.8) test_8 (test_simple.HashingTester) (1/1)

1.9) test_9 (test_simple.HashingTester) (5/5)

1.10) test_10 (test_simple.HashingTester) (1/1)

1.11) test_11 (test_simple.HashingTester) (1/1)

1.12) test_12 (test_simple.HashingTester) (2/2)

1.13) test_13 (test_simple.HashingTester) (5/5)

1.14) test_14 (test_simple.HashingTester) (5/5)

1.15) test_15 (test_simple.HashingTester) (7/7)

1.16) test_16 (test_simple.HashingTester) (1/1)

1.17) test_17 (test_simple.HashingTester) (4/4)

1.18) test_18 (test_simple.HashingTester) (4/4)

1.19) test_19 (test_simple.HashingTester) (7/7)

1.20) test_20 (test_simple.HashingTester) (1/1)

1.22) test_22_linear (test_simple.HashingTester) (7/7)

1.23) test_23_quadratic (test_simple.HashingTester) (7/7)

1.24) test_24_cubic (test_simple.HashingTester) (7/7)

1.25) test_25_chaining (test_simple.HashingTester) (2/2)

1.27) test_27_linear (test_simple.HashingTester) (1/1)

1.28) test_28_quadratic (test_simple.HashingTester) (1/1)

1.29) test_29_cubic (test_simple.HashingTester) (1/1)

1.30) test_30_chaining (test_simple.HashingTester) (1/1)

1.31) test_31_linear (test_simple.HashingTester) (1/1)

Autograder Results

Check submitted files (0/0)

All required files submitted!

1.1) test_1 (test_simple.HashingTester) (1/1)

1.2) test_2 (test_simple.HashingTester) (1/1)

1.3) test_3 (test_simple.HashingTester) (2/2)

1.4) test_4 (test_simple.HashingTester) (2/2)

1.5) test_5 (test_simple.HashingTester) (1/1)

1.6) test_6 (test_simple.HashingTester) (5/5)

1.7) test_7 (test_simple.HashingTester) (2/2)

1.8) test_8 (test_simple.HashingTester) (1/1)

1.9) test_9 (test_simple.HashingTester) (5/5)

1.10) test_10 (test_simple.HashingTester) (1/1)

1.11) test_11 (test_simple.HashingTester) (1/1)

1.12) test_12 (test_simple.HashingTester) (2/2)

1.13) test_13 (test_simple.HashingTester) (5/5)

1.14) test_14 (test_simple.HashingTester) (5/5)

1.15) test_15 (test_simple.HashingTester) (7/7)

1.16) test_16 (test_simple.HashingTester) (1/1)

1.17) test_17 (test_simple.HashingTester) (4/4)

1.18) test_18 (test_simple.HashingTester) (4/4)

1.19) test_19 (test_simple.HashingTester) (7/7)

1.20) test_20 (test_simple.HashingTester) (1/1)

1.21) test_21 (test_simple.HashingTester) (0/7)

```
Test Failed: 'Testcase failed databaseSize instruction 4\nPart test5: Comp failed\n' != 'Part test5: Comp passed
- Testcase failed databaseSize instruction 4
- Part test5: Comp failed
?                ^ ^^
+ Part test5: Comp passed
?                ^ ^^
 :
------------------------------------
Testcase failed databaseSize instruction 4
Part test5: Comp failed

------------------------------------
```

1.22) test_22_linear (test_simple.HashingTester) (7/7)

1.23) test_23_quadratic (test_simple.HashingTester) (7/7)

1.24) test_24_cubic (test_simple.HashingTester) (7/7)

1.25) test_25_chaining (test_simple.HashingTester) (2/2)

1.26) test_26_comp (test_simple.HashingTester) (0/7)

```
Test Failed: 'Testcase failed getBalance instruction 4\nPart test6: Comp failed\n' != 'Part test6: Comp passed\r
- Testcase failed getBalance instruction 4
- Part test6: Comp failed
?                ^ ^^
+ Part test6: Comp passed
?                ^ ^^
 :
------------------------------------
Testcase failed getBalance instruction 4
Part test6: Comp failed

------------------------------------
```

1.27) test_27_linear (test_simple.HashingTester) (1/1)

1.28) test_28_quadratic (test_simple.HashingTester) (1/1)

1.29) test_29_cubic (test_simple.HashingTester) (1/1)

1.30) test_30_chaining (test_simple.HashingTester) (1/1)

1.31) test_31_linear (test_simple.HashingTester) (1/1)

Submitted Files

BaseClass.h

Download

```cpp
#ifndef BASECLASS_H
#define BASECLASS_H

#include <string>
#include <vector>
struct Account {
    std::string id;
    int balance;
};

class BaseClass {
public:
    virtual void createAccount(std::string id, int count) = 0;
    virtual std::vector<int> getTopK(int k) = 0;
    virtual int getBalance(std::string id) = 0;
    virtual void addTransaction(std::string id, int count) = 0;
    virtual bool doesExist(std::string id) = 0;
    virtual bool deleteAccount(std::string id) = 0;
    virtual int databaseSize() = 0;
    virtual int hash(std::string id) = 0;



    std::vector<Account> bankStorage1d;
    std::vector<std::vector<Account>> bankStorage2d;

};

#endif // BASECLASS_H
```

Chaining.cpp

Download

```cpp
#include "Chaining.h"
using namespace std;


Chaining::Chaining(){
    bankStorage2d.resize(100001);
}


int Chaining::hash(std::string id) {
    int s=0;
    for(char c: id){
        s= s+ int(c);


    }

    // IMPLEMENT YOUR CODE HERE
    return s% bankStorage2d.size(); // Placeholder return value
}



std::vector<int> merge(const std::vector<int>& left, const std::vector<int>& right) {
    std::vector<int> merged;

    size_t leftIdx = 0, rightIdx = 0;

    while (leftIdx < left.size() && rightIdx < right.size()) {
        if (left[leftIdx] >= right[rightIdx]) {

            merged.push_back(left[leftIdx]);

            leftIdx++;
        } else {
            merged.push_back(right[rightIdx]);

            rightIdx++;

        }
    }


    while (leftIdx < left.size()) {
        merged.push_back(left[leftIdx]);

        leftIdx++;
    }

```

```
    while (rightIdx < right.size()) {

        merged.push_back(right[rightIdx]);

        rightIdx++;
    }

    return merged;
}

// Merge sort function
std::vector<int> mergeSort(const std::vector<int>& arr) {
    if (arr.size() <= 1) {

        return arr;
    }

    size_t mid = arr.size() / 2;

    std::vector<int> left(arr.begin(), arr.begin() + mid);

    std::vector<int> right(arr.begin() + mid, arr.end());


    left = mergeSort(left);
    right = mergeSort(right);

    return merge(left, right);
}


std::vector<int> Chaining::getTopK(int k) {

    std::vector<int> balances;

    for (const std::vector<Account>& bucket : bankStorage2d) {

        for (const Account& account : bucket) {

            balances.push_back(account.balance);
        }
    }


    balances = mergeSort(balances);


    if (k < balances.size()) {

        balances.resize(k);
    }

```

```
    return balances;
}


void Chaining::createAccount(std::string id, int count) {

    int m = hash(id);

    for (Account &x : bankStorage2d[m]){

        if(x.id== id){

            return ;
        }

    }

    bankStorage2d[m].push_back({id, count});


}




void Chaining::addTransaction(std::string id, int count) {
    // IMPLEMENT YOUR CODE HERE
    int index = hash(id);


    for (Account& account : bankStorage2d[index]) {
        if (account.id == id) {

            account.balance += count;
            return;
        }
    }

    bankStorage2d[index].push_back({id, count});
}



bool Chaining::deleteAccount(std::string id) {
    // IMPLEMENT YOUR CODE HERE
    int index = hash(id);


    for (auto x = bankStorage2d[index].begin(); x != bankStorage2d[index].end(); ++x) {

        if (x->id == id) {

```

```
            bankStorage2d[index].erase(x);

            return true;
        }
    }


    return false; // Placeholder return value
}



bool Chaining::doesExist(std::string id) {
    // IMPLEMENT YOUR CODE HERE
    int index = hash(id);


    for (const Account& account : bankStorage2d[index]) {

        if (account.id == id) {

            return true;

        }
    }



    return false; // Placeholder return value
}



int Chaining::getBalance(std::string id) {
    // IMPLEMENT YOUR CODE HERE


    int index = hash(id);


    for (const Account& account : bankStorage2d[index]) {
        if (account.id == id) {
            return account.balance;
        }
    }


    return -1;

     // Placeholder return value
}

```

```
int Chaining::databaseSize() {
    int s= 0;


    for (const std::vector<Account>& bucket : bankStorage2d) {

        s += bucket.size();
    }

    return s;

    // Placeholder return value
}


```

Chaining.h Download

```
#ifndef CHAINING_H
#define CHAINING_H

#include "BaseClass.h"
#include <iostream>
#include <vector>

class Chaining : public BaseClass {
public:
    Chaining();
    void createAccount(std::string id, int count) override;
    std::vector<int> getTopK(int k) override;
    int getBalance(std::string id) override;
    void addTransaction(std::string id, int count) override;
    bool doesExist(std::string id) override;
    bool deleteAccount(std::string id) override;
    int databaseSize() override;
    int hash(std::string id) override;

private:

    // Other data members and functions specific to Chaining
};

#endif // CHAINING_H
```

Comp.cpp

Download

```cpp
#include "Comp.h"

void Comp::createAccount(std::string id, int count) {

    // IMPLEMENT YOUR CODE HERE
}

std::vector<int> Comp::getTopK(int k) {
    // IMPLEMENT YOUR CODE HERE
    return std::vector<int>(); // Placeholder return value
}

int Comp::getBalance(std::string id) {
    // IMPLEMENT YOUR CODE HERE
    return 0; // Placeholder return value
}

void Comp::addTransaction(std::string id, int count) {
    // IMPLEMENT YOUR CODE HERE
}

bool Comp::doesExist(std::string id) {
    // IMPLEMENT YOUR CODE HERE
    return false; // Placeholder return value
}

bool Comp::deleteAccount(std::string id) {
    // IMPLEMENT YOUR CODE HERE
    return false; // Placeholder return value
}
int Comp::databaseSize() {
    // IMPLEMENT YOUR CODE HERE
    return 0; // Placeholder return value
}

int Comp::hash(std::string id) {
    return 0;
    // IMPLEMENT YOUR CODE HERE

}


// Feel free to add any other helper functions you need
// Good Luck!
```

Comp.h

Download

```cpp
#ifndef COMP_H
#define COMP_H

#include "BaseClass.h"
#include <iostream>
#include <vector>

class Comp : public BaseClass {
public:
    void createAccount(std::string id, int count) override;
    std::vector<int> getTopK(int k) override;
    int getBalance(std::string id) override;
    void addTransaction(std::string id, int count) override;
    bool doesExist(std::string id) override;
    bool deleteAccount(std::string id) override;
    int databaseSize() override;
    int hash(std::string id) override;

private:

    // Other data members and functions specific to Your implementation
};

#endif // COMP_H
```

CubicProbing.cpp

Download

```cpp

#include "CubicProbing.h"
#include <iostream>
#include <vector>
using namespace std;



CubicProbing:: CubicProbing (){
    int size=0;
    m=300007;
    bankStorage1d.resize(m);
    for (int i=0;i<m;i++){
        bankStorage1d[i].id="";
        bankStorage1d[i].balance=-1;
    }

    bankStorage1d.resize(m);

}



int CubicProbing::hash(std::string id) {

    // int sum = 0;


    // for (char c : id) {
    //     sum += int(c);
    // }

    // return sum % m;
    int prod=0;
    int n=id.length();
    for (int i=0;i<n;i++){
        prod+=(int)id[i];
    }
    return prod;
}

int CubicProbing::databaseSize() {
    // int size = 0;


    // for (const Account& account : bankStorage1d) {
    //     if (!account.id.empty()) {
    //         size++;
    //     }
```

```
    // }

    return size;

    // int size = 0;

    // for (int i=0 ;i<100000;i++) {

    //     if (!bankStorage1d[i].id.empty()) {
    //         size++;
    //     }
    // }


    // return size;
}




std::vector<int> merge(const std::vector<int>& left, const std::vector<int>& right) {


    std::vector<int> merged;


    size_t leftIdx = 0, rightIdx = 0;

    while (leftIdx < left.size() && rightIdx < right.size()) {

        if (left[leftIdx] >= right[rightIdx]) {

            merged.push_back(left[leftIdx]);

            leftIdx++;
        } else {

            merged.push_back(right[rightIdx]);

            rightIdx++;
        }
    }


    while (leftIdx < left.size()) {
        merged.push_back(left[leftIdx]);
        leftIdx++;
    }

    while (rightIdx < right.size()) {
```

```
        merged.push_back(right[rightIdx]);
        rightIdx++;
    }

    return merged;
}

// Merge sort function
std::vector<int> mergeSort(const std::vector<int>& arr) {
    if (arr.size() <= 1) {
        return arr;
    }

    size_t mid = arr.size() / 2;
    std::vector<int> left(arr.begin(), arr.begin() + mid);
    std::vector<int> right(arr.begin() + mid, arr.end());

    left = mergeSort(left);
    right = mergeSort(right);

    return merge(left, right);
}


std::vector<int> CubicProbing::getTopK(int k) {

    std::vector<int> balances;
    for (const Account& account : bankStorage1d) {
        if (!account.id.empty()) {
            balances.push_back(account.balance);
        }
    }


    balances = mergeSort(balances);


    if (k < balances.size()) {
        balances.resize(k);
    }

    return balances;
}





void CubicProbing::createAccount(std::string id, int count) {

    // int index = hash(id);
    // size++;
```

```
    // int originalIndex = index;

    // int i = 1;
    // while (!bankStorage1d[index].id.empty() && bankStorage1d[index].id != id) {



    //     index = (originalIndex + (i * i * i)) % m;

    //     i++;
    // }

    // if (bankStorage1d[index].id.empty() || bankStorage1d[index].id == id) {

    //     bankStorage1d[index] = {id, count};
    // }
    Account new_acc;
    new_acc.id=id;
    new_acc.balance=count;
    int ind=hash(id);
    if (bankStorage1d[ind].id=="" || bankStorage1d[ind].id=="del"){
        bankStorage1d[ind].balance=count;
        bankStorage1d[ind].id=id;
        size+=1;
    } else{
        int j=1;
        while (bankStorage1d[ind].balance!=-1){
            ind=(ind+j*j*j)%m;
            j++;
        }  bankStorage1d[ind].balance=count;
        bankStorage1d[ind].id=id;
        size+=1;
}
}


int CubicProbing::getBalance(std::string id) {

    if (doesExist(id)==false){
        return -1;
    } else{
        int ind=hash(id);
        int j=1;
        while (j>0){
            if (bankStorage1d[ind].id==id){
                return bankStorage1d[ind].balance;
            } ind=(ind+j*j*j)%m;
            j++;
        }
} return -1;
```

```
//  int index = hash(id);
//     int oriindx = index;
//     int i=1;
//     while (bankStorage1d[index].id!= id){
//        index= (index +(i*i*i))%m;
//        i++;
//        if (index== oriindx){
//         return -1;

//        }
//        return bankStorage1d[index].balance;



//     }

}





void CubicProbing::addTransaction(std::string id, int count) {

    // int index = hash(id);
    // int originalIndex = index;

    // int i = 1;
    // while (!bankStorage1d[index].id.empty()) {


    //     if (bankStorage1d[index].id == id) {



    //         bankStorage1d[index].balance += count;

    //         return;
    //     }

    //     index = (originalIndex + (i * i * i)) % m;
    //     i++;
    // }


    // Account newAccount = {id, count};
    // bankStorage1d[index] = newAccount;
    int ind=hash(id);
    if (doesExist(id)==true){
        int j=1;
        while (bankStorage1d[ind].id!=""){
```

```
            if (bankStorage1d[ind].id==id){
                bankStorage1d[ind].balance+=count;
                break;
            } ind=(ind+j*j*j)%m;
            j++;
        }
} else{
    createAccount(id,count);
}

}


bool CubicProbing::doesExist(std::string id) {



    // int index = hash(id);


    // int originalIndex = index;

    // int i = 1;
    // while (!bankStorage1d[index].id.empty()) {


    //     if (bankStorage1d[index].id == id) {

    //         return true;
    //     }


    //     index = (originalIndex + i * i * i) % m;


    //     i++;
    // }


    // return false;
    int ind=hash(id);
    int j=1;
    while (bankStorage1d[ind].id!=""){
        if (bankStorage1d[ind].id==id){
            return true;
        } ind=(ind+j*j*j)%m;
        j++;
    }
    return false;
}


```

```

bool CubicProbing::deleteAccount(std::string id) {


    // int index = hash(id);

    // int originalIndex = index;

    // int i = 1;
    // while (!bankStorage1d[index].id.empty()) {

    //     if (bankStorage1d[index].id == id) {


    //         bankStorage1d[index].id.clear();

    //         bankStorage1d[index].balance = 0;
    //         size--;

    //         return true;
    //     }


    //     index = (originalIndex + i * i * i) % m;
    //     i++;
    // }


    // return false;
     if (doesExist(id)==false){
        return false;
    } else{
        int ind=hash(id);
        int j=1;
        while(bankStorage1d[ind].id!=""){
            if (bankStorage1d[ind].id==id){
                bankStorage1d[ind].id="del";
                bankStorage1d[ind].balance=-1;
                size-=1;
                return true;
            } ind=(ind+j*j*j)%m;
            j++;
        }

    } return false;
}

```

CubicProbing.h Download

```cpp
#ifndef CUBICPROBING_H
#define CUBICPROBING_H

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

class CubicProbing : public BaseClass {
public:
    CubicProbing();
    void createAccount(std::string id, int count) override;
    std::vector<int> getTopK(int k) override;
    int getBalance(std::string id) override;
    void addTransaction(std::string id, int count) override;
    bool doesExist(std::string id) override;
    bool deleteAccount(std::string id) override;
    int databaseSize() override;
    int hash(std::string id) override;


private:
    // Other data members and functions specific to Quadratic Probing
    int size=0;
    int m=300007;

};

#endif // CUBICPROBING_H
```

LinearProbing.cpp

Download

```cpp
#include "LinearProbing.h"

LinearProbing:: LinearProbing(){
    int size=0;
    int m = 300007;
    bankStorage1d.resize(300007);
    for (int i=0;i<m;i++){
        bankStorage1d[i].id="";
        bankStorage1d[i].balance=-1;
    }

}



int LinearProbing::hash(std::string id) {
    int sum=1;
    int k=id.length();

    for (int i=0;i<k;i++){
        int c= (int)id[i];
        sum*=c;
        sum=sum%m;
    }
    return sum;
}



int LinearProbing::databaseSize() {

    // int count= 0;

    // for (auto account : bankStorage1d) {

    //     if (!account.id.empty()) {

    //         count++;
    //     }
    // }

    return size;
    // int size = 0;

    // for (int i=0 ;i<100000;i++) {

    //     if (!bankStorage1d[i].id.empty()) {
    //         size++;
    //     }
```

```
    // }


    // return size;
}



std::vector<int> merge(const std::vector<int>& left, const std::vector<int>& right) {
    std::vector<int> merged;
    size_t leftIdx = 0, rightIdx = 0;

    while (leftIdx < left.size() && rightIdx < right.size()) {

        if (left[leftIdx] >= right[rightIdx]) {

            merged.push_back(left[leftIdx]);

            leftIdx++;

        } else {
            merged.push_back(right[rightIdx]);

            rightIdx++;

        }
    }


    while (leftIdx < left.size()) {

        merged.push_back(left[leftIdx]);

        leftIdx++;
    }

    while (rightIdx < right.size()) {

        merged.push_back(right[rightIdx]);

        rightIdx++;
    }

    return merged;
}


std::vector<int> mergeSort(const std::vector<int>& arr) {
    if (arr.size() <= 1) {

        return arr;
    }
```

```

    size_t mid = arr.size() / 2;

    std::vector<int> left(arr.begin(), arr.begin() + mid);


    std::vector<int> right(arr.begin() + mid, arr.end());

    left = mergeSort(left);

    right = mergeSort(right);

    return merge(left, right);
}





std::vector<int> LinearProbing::getTopK(int k) {

    std::vector<int> vec;
    for (const Account& account : bankStorage1d) {
        if (!account.id.empty()) {
            vec.push_back(account.balance);
        }
    }


    vec = mergeSort(vec);


    if (k < vec.size()) {
        vec.resize(k);
    }

    return vec;
}


void LinearProbing::createAccount(std::string id, int count) {

    // int ind= hash(id);
    // size++;


    // while (!bankStorage1d[ind].id.empty() && bankStorage1d[ind].id != id) {

    //     ind = (ind + 1) % m;

    // }
```

```
    // if (bankStorage1d[ind].id.empty() || bankStorage1d[ind].id == id) {

    //     bankStorage1d[ind] = {id, count};
    // }
    Account new_acc;
    new_acc.id=id;
    new_acc.balance=count;
    int ind=hash(id);
    if (bankStorage1d[ind].id=="" || bankStorage1d[ind].id=="del"){
        bankStorage1d[ind].balance=count;
        bankStorage1d[ind].id=id;
        size+=1;
    } else{
        int j=1;
        while (bankStorage1d[ind].balance!=-1){
            ind=(ind+1)%m;
            j++;
        }  bankStorage1d[ind].balance=count;
        bankStorage1d[ind].id=id;
        size+=1;
}

}

int LinearProbing::getBalance(std::string id) {

     if (doesExist(id)==false){
        return -1;
    } else{
        int ind=hash(id);
        int j=1;
        while (j>0){
            if (bankStorage1d[ind].id==id){
                return bankStorage1d[ind].balance;
            } ind=(ind+1)% m;
            j++;
        }
} return -1;
}

    // int index = hash(id);

    // while (bankStorage1d[index].id != id) {
    //     if (bankStorage1d[index].id == "" ) {
    //         return -1;
    //     }
    //     index = (index + 1) % bankStorage1d.size();
    // }

```

```
    // return bankStorage1d[index].balance;




void LinearProbing::addTransaction(std::string id, int count) {

    // int index = hash(id);


    // while (!bankStorage1d[index].id.empty()) {

    //     if (bankStorage1d[index].id == id) {
    //         bankStorage1d[index].balance += count;
    //         return;
    //     }
    //     index = (index + 1) % m;
    // }
    // Account newAccount = {id, count};
    // bankStorage1d[index] = newAccount;
    // // createAccount(id,count);
    int ind=hash(id);
    if (doesExist(id)==true){
        int j=1;
        while (bankStorage1d[ind].id!=""){
            if (bankStorage1d[ind].id==id){
                bankStorage1d[ind].balance+=count;
                break;
            } ind=(ind+1)%m;
            j++;
        }
} else{
    createAccount(id,count);
}
}




bool LinearProbing::deleteAccount(std::string id) {

    // int index = hash(id);


    // while (!bankStorage1d[index].id.empty()) {

    //     if (bankStorage1d[index].id == id) {
```

```
    //        bankStorage1d[index].id.clear();

    //        bankStorage1d[index].balance = 0;
    //        size--;

    //        return true;
    //    }
    //    index = (index + 1) % m;

    // }


    // return false;

     if (doesExist(id)==false){
        return false;
    } else{
        int ind=hash(id);
        int j=1;
        while(bankStorage1d[ind].id!=""){
            if (bankStorage1d[ind].id==id){

                bankStorage1d[ind].id="del";

                bankStorage1d[ind].balance=-1;

                size-=1;
                return true;
            } ind=(ind+1)%m;
            j++;
        }

    } return false;
}




bool LinearProbing::doesExist(std::string id) {

    // int index = hash(id);


    // while (!bankStorage1d[index].id.empty()) {

    //    if (bankStorage1d[index].id == id) {

    //        return true;
    //    }
    //    index = (index + 1) % m;
```

```

    // }


    // return false;
    int ind=hash(id);
    int j=1;
    while (bankStorage1d[ind].id!=""){
        if (bankStorage1d[ind].id==id){
            return true;
        } ind=(ind+1)%m;
        j++;
    }
    return false;

}
```

LinearProbing.h Download

```
#ifndef LINEARPROBING_H
#define LINEARPROBING_H

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

class LinearProbing : public BaseClass {
public:
    LinearProbing();
    void createAccount(std::string id, int count) override;
    std::vector<int> getTopK(int k) override;
    int getBalance(std::string id) override;
    void addTransaction(std::string id, int count) override;
    bool doesExist(std::string id) override;
    bool deleteAccount(std::string id) override;
    int databaseSize() override;
    int hash(std::string id) override;


private:
    // Other data members and functions specific to Linear Probing
    int size=0;
    int m =300007;
};

#endif // LINEARPROBING_H
```

QuadraticProbing.cpp

Download

```cpp

#include "QuadraticProbing.h"
#include <iostream>
#include <vector>
using namespace std;


QuadraticProbing:: QuadraticProbing(){
    int size=0;
    m=300007;
    bankStorage1d.resize(m);
    for (int i=0;i<m;i++){
        bankStorage1d[i].id="";
        bankStorage1d[i].balance=-1;
    }

    bankStorage1d.resize(m);

}


// int QuadraticProbing::databaseSize() {
//     int size = 0;


//     for (const Account& account : bankStorage1d) {
//         if (!account.id.empty()) {
//             size++;
//         }
//     }

//     return size;


// }

int QuadraticProbing::databaseSize() {

    return size;
}


int QuadraticProbing::hash(std::string id) {

    // int sum = 0;
    // for (char c : id) {
    //     sum += int (c);
    // }

```

```
    // return sum % m;
    int prod=1;
    int n=id.length();
    for (int i=0;i<n;i++){
        int c= (int)id[i];
        prod*=c;
        prod=prod%m;
    }
    return prod;
}



std::vector<int> merge(const std::vector<int>& left, const std::vector<int>& right) {
    std::vector<int> merged;
    size_t leftIdx = 0, rightIdx = 0;

    while (leftIdx < left.size() && rightIdx < right.size()) {
        if (left[leftIdx] >= right[rightIdx]) {
            merged.push_back(left[leftIdx]);
            leftIdx++;
        } else {
            merged.push_back(right[rightIdx]);
            rightIdx++;
        }
    }


    while (leftIdx < left.size()) {
        merged.push_back(left[leftIdx]);
        leftIdx++;
    }

    while (rightIdx < right.size()) {
        merged.push_back(right[rightIdx]);
        rightIdx++;
    }

    return merged;
}


std::vector<int> mergeSort(const std::vector<int>& arr) {
    if (arr.size() <= 1) {
        return arr;
    }

    size_t mid = arr.size() / 2;
    std::vector<int> left(arr.begin(), arr.begin() + mid);
    std::vector<int> right(arr.begin() + mid, arr.end());

```

```
    left = mergeSort(left);
    right = mergeSort(right);

    return merge(left, right);
}




std::vector<int> QuadraticProbing::getTopK(int k) {

    std::vector<int> bal;
    for (const Account& account : bankStorage1d) {
        if (!account.id.empty()) {
            bal.push_back(account.balance);
        }
    }


    bal = mergeSort(bal);


    if (k < bal.size()) {
        bal.resize(k);
    }

    return bal;




}




void QuadraticProbing::createAccount(std::string id, int count) {

    // int index = hash(id);
    // int originalIndex = index;
    // size++;

    // int i = 1;
    // while (!bankStorage1d[index].id.empty() && bankStorage1d[index].id != id) {

    //     index = (originalIndex + i * i) % bankStorage1d.size();
    //     i++;
    // }

    // if (bankStorage1d[index].id.empty() || bankStorage1d[index].id == id) {

    //     bankStorage1d[index] = {id, count};
```

```
    // }
    Account new_acc;
    new_acc.id=id;
    new_acc.balance=count;
    int ind=hash(id);
    if (bankStorage1d[ind].id=="" || bankStorage1d[ind].id=="del"){
        bankStorage1d[ind].balance=count;
        bankStorage1d[ind].id=id;
        size+=1;
    } else{
        int j=1;
        while (bankStorage1d[ind].balance!=-1){
            ind=(ind+j*j)%m;
            j++;
        }  bankStorage1d[ind].balance=count;
        bankStorage1d[ind].id=id;
        size+=1;
}
}


int QuadraticProbing::getBalance(std::string id) {

    // int index = hash(id);
    // int oriindx = index;
    // int i=1;
    // while (bankStorage1d[index].id!= id){
    //    index= (index +(i*i))%m;
    //    i++;
    //    if (index== oriindx){
    //     return -1;

    //    }
    //    return bankStorage1d[index].balance;



    // }
     if (doesExist(id)==false){
        return -1;
    } else{
        int ind=hash(id);
        int j=1;
        while (j>0){
            if (bankStorage1d[ind].id==id){
                return bankStorage1d[ind].balance;
            } ind=(ind+j*j)% m;
            j++;
        }
} return -1;


```

```

}


void QuadraticProbing::addTransaction(std::string id, int count) {

    // int index = hash(id);
    // int originalIndex = index;

    // int i = 1;
    // while (!bankStorage1d[index].id.empty()) {
    //     if (bankStorage1d[index].id == id) {

    //         bankStorage1d[index].balance += count;
    //         return;
    //     }


    //     index = (originalIndex + (i * i)) % m;
    //     i++;
    // }


    // Account newAccount = {id, count};
    // bankStorage1d[index] = newAccount;

    int ind=hash(id);
    if (doesExist(id)==true){
        int j=1;
        while (bankStorage1d[ind].id!=""){
            if (bankStorage1d[ind].id==id){
                bankStorage1d[ind].balance+=count;
                break;
            } ind=(ind+j*j)%m;
            j++;
        }
} else{
    createAccount(id,count);
}
}




bool QuadraticProbing::doesExist(std::string id) {

    // int index = hash(id);

    // int originalIndex = index;

    // int i = 1;
    // while (!bankStorage1d[index].id.empty()) {
```

```

    //     if (bankStorage1d[index].id == id) {

    //         return true;
    //     }


    //     index = (originalIndex + i * i) % m;
    //     i++;
    // }


    // return false;

    int ind=hash(id);
    int j=1;
    while (bankStorage1d[ind].id!=""){
        if (bankStorage1d[ind].id==id){
            return true;
        } ind=(ind+j*j)%m;
        j++;
    }
    return false;
}


bool QuadraticProbing::deleteAccount(std::string id) {

    // int index = hash(id);

    // int originalIndex = index;

    // int i = 1;

    // while (!bankStorage1d[index].id.empty()) {
    //     if (bankStorage1d[index].id == id) {

    //         bankStorage1d[index].id.clear();

    //         bankStorage1d[index].balance = 0;
    //         size--;
    //         return true;
    //     }


    //     index = (originalIndex + i * i) % m;

    //     i++;

    // }


```

```
    // return false;
     if (doesExist(id)==false){
        return false;
    } else{
        int ind=hash(id);
        int j=1;
        while(bankStorage1d[ind].id!=""){
            if (bankStorage1d[ind].id==id){
                bankStorage1d[ind].id="del";
                bankStorage1d[ind].balance=-1;
                size-=1;
                return true;
            } ind=(ind+j*j)%m;
            j++;
        }

    } return false;
}
```

QuadraticProbing.h

Download

```cpp
#ifndef QUADRATICPROBING_H
#define QUADRATICPROBING_H

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

class QuadraticProbing : public BaseClass {
public:
    QuadraticProbing();
    void createAccount(std::string id, int count) override;
    std::vector<int> getTopK(int k) override;
    int getBalance(std::string id) override;
    void addTransaction(std::string id, int count) override;
    bool doesExist(std::string id) override;
    bool deleteAccount(std::string id) override;
    int databaseSize() override;
    int hash(std::string id) override;


private:
    // Other data members and functions specific to Quadratic Probing
    int size=0;
    int m =300007;




};

#endif // QUADRATICPROBING_H
```