

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\n'

- 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

```
1  #ifndef BASECLASS_H
2  #define BASECLASS_H
3
4  #include <string>
5  #include <vector>
6  struct Account {
7      std::string id;
8      int balance;
9  };
10
11 class BaseClass {
12 public:
13     virtual void createAccount(std::string id, int count) = 0;
14     virtual std::vector<int> getTopK(int k) = 0;
15     virtual int getBalance(std::string id) = 0;
16     virtual void addTransaction(std::string id, int count) = 0;
17     virtual bool doesExist(std::string id) = 0;
18     virtual bool deleteAccount(std::string id) = 0;
19     virtual int databaseSize() = 0;
20     virtual int hash(std::string id) = 0;
21
22
23
24     std::vector<Account> bankStorage1d;
25     std::vector<std::vector<Account>> bankStorage2d;
26
27 };
28
29 #endif // BASECLASS_H
```

```
1  #include "Chaining.h"
2  using namespace std;
3
4
5  Chaining::Chaining(){
6      bankStorage2d.resize(100001);
7  }
8
9
10 int Chaining::hash(std::string id) {
11     int s=0;
12     for(char c: id){
13         s= s+ int(c);
14     }
15
16 }
17
18 // IMPLEMENT YOUR CODE HERE
19 return s% bankStorage2d.size(); // Placeholder return value
20 }
21
22
23
24 std::vector<int> merge(const std::vector<int>& left, const std::vector<int>& right) {
25     std::vector<int> merged;
26
27     size_t leftIdx = 0, rightIdx = 0;
28
29     while (leftIdx < left.size() && rightIdx < right.size()) {
30         if (left[leftIdx] >= right[rightIdx]) {
31
32             merged.push_back(left[leftIdx]);
33
34             leftIdx++;
35         } else {
36             merged.push_back(right[rightIdx]);
37
38             rightIdx++;
39         }
40     }
41 }
42
43
44 while (leftIdx < left.size()) {
45     merged.push_back(left[leftIdx]);
46
47     leftIdx++;
48 }
49
```

```
50     while (rightIdx < right.size()) {
51
52         merged.push_back(right[rightIdx]);
53
54         rightIdx++;
55     }
56
57     return merged;
58 }
59
60 // Merge sort function
61 std::vector<int> mergeSort(const std::vector<int>& arr) {
62     if (arr.size() <= 1) {
63
64         return arr;
65     }
66
67     size_t mid = arr.size() / 2;
68
69     std::vector<int> left(arr.begin(), arr.begin() + mid);
70
71     std::vector<int> right(arr.begin() + mid, arr.end());
72
73
74     left = mergeSort(left);
75     right = mergeSort(right);
76
77     return merge(left, right);
78 }
79
80
81 std::vector<int> Chaining::getTopK(int k) {
82
83     std::vector<int> balances;
84
85     for (const std::vector<Account>& bucket : bankStorage2d) {
86
87         for (const Account& account : bucket) {
88
89             balances.push_back(account.balance);
90         }
91     }
92
93
94     balances = mergeSort(balances);
95
96
97     if (k < balances.size()) {
98
99         balances.resize(k);
100     }
101 }
```

```

102     return balances;
103 }
104
105
106 void Chaining::createAccount(std::string id, int count) {
107
108     int m = hash(id);
109
110     for (Account &x : bankStorage2d[m]){
111
112         if(x.id== id){
113
114             return ;
115         }
116     }
117
118     bankStorage2d[m].push_back({id, count});
119
120
121
122 }
123
124
125
126
127 void Chaining::addTransaction(std::string id, int count) {
128     // IMPLEMENT YOUR CODE HERE
129     int index = hash(id);
130
131
132     for (Account& account : bankStorage2d[index]) {
133         if (account.id == id) {
134
135             account.balance += count;
136             return;
137         }
138     }
139
140     bankStorage2d[index].push_back({id, count});
141 }
142
143
144
145 bool Chaining::deleteAccount(std::string id) {
146     // IMPLEMENT YOUR CODE HERE
147     int index = hash(id);
148
149
150     for (auto x = bankStorage2d[index].begin(); x != bankStorage2d[index].end(); ++x) {
151
152         if (x->id == id) {
153

```




```
154         bankStorage2d[index].erase(x);
155
156         return true;
157     }
158 }
159
160
161 return false; // Placeholder return value
162 }
163
164
165
166 bool Chaining::doesExist(std::string id) {
167     // IMPLEMENT YOUR CODE HERE
168     int index = hash(id);
169
170
171     for (const Account& account : bankStorage2d[index]) {
172
173         if (account.id == id) {
174
175             return true;
176
177         }
178     }
179
180
181
182     return false; // Placeholder return value
183 }
184
185
186
187 int Chaining::getBalance(std::string id) {
188     // IMPLEMENT YOUR CODE HERE
189
190
191     int index = hash(id);
192
193
194     for (const Account& account : bankStorage2d[index]) {
195         if (account.id == id) {
196             return account.balance;
197         }
198     }
199
200
201     return -1;
202
203     // Placeholder return value
204 }
205
```

```

206
207
208
209 int Chaining::databaseSize() {
210     int s= 0;
211
212
213     for (const std::vector<Account>& bucket : bankStorage2d) {
214
215         s += bucket.size();
216     }
217
218     return s;
219
220     // Placeholder return value
221 }
222
223

```

▼ Chaining.h

 Download

```

1  #ifndef CHAINING_H
2  #define CHAINING_H
3
4  #include "BaseClass.h"
5  #include <iostream>
6  #include <vector>
7
8  class Chaining : public BaseClass {
9  public:
10     Chaining();
11     void createAccount(std::string id, int count) override;
12     std::vector<int> getTopK(int k) override;
13     int getBalance(std::string id) override;
14     void addTransaction(std::string id, int count) override;
15     bool doesExist(std::string id) override;
16     bool deleteAccount(std::string id) override;
17     int databaseSize() override;
18     int hash(std::string id) override;
19
20 private:
21
22     // Other data members and functions specific to Chaining
23 };
24
25 #endif // CHAINING_H

```

```
1  #include "Comp.h"
2
3  void Comp::createAccount(std::string id, int count) {
4
5      // IMPLEMENT YOUR CODE HERE
6  }
7
8  std::vector<int> Comp::getTopK(int k) {
9      // IMPLEMENT YOUR CODE HERE
10     return std::vector<int>(); // Placeholder return value
11 }
12
13 int Comp::getBalance(std::string id) {
14     // IMPLEMENT YOUR CODE HERE
15     return 0; // Placeholder return value
16 }
17
18 void Comp::addTransaction(std::string id, int count) {
19     // IMPLEMENT YOUR CODE HERE
20 }
21
22 bool Comp::doesExist(std::string id) {
23     // IMPLEMENT YOUR CODE HERE
24     return false; // Placeholder return value
25 }
26
27 bool Comp::deleteAccount(std::string id) {
28     // IMPLEMENT YOUR CODE HERE
29     return false; // Placeholder return value
30 }
31 int Comp::databaseSize() {
32     // IMPLEMENT YOUR CODE HERE
33     return 0; // Placeholder return value
34 }
35
36 int Comp::hash(std::string id) {
37     return 0;
38     // IMPLEMENT YOUR CODE HERE
39
40 }
41
42
43 // Feel free to add any other helper functions you need
44 // Good Luck!
```

```
1  #ifndef COMP_H
2  #define COMP_H
3
4  #include "BaseClass.h"
5  #include <iostream>
6  #include <vector>
7
8  class Comp : public BaseClass {
9  public:
10     void createAccount(std::string id, int count) override;
11     std::vector<int> getTopK(int k) override;
12     int getBalance(std::string id) override;
13     void addTransaction(std::string id, int count) override;
14     bool doesExist(std::string id) override;
15     bool deleteAccount(std::string id) override;
16     int databaseSize() override;
17     int hash(std::string id) override;
18
19 private:
20
21     // Other data members and functions specific to Your implementation
22 };
23
24 #endif // COMP_H
```

```
1
2  #include "CubicProbing.h"
3  #include <iostream>
4  #include <vector>
5  using namespace std;
6
7
8
9  CubicProbing::CubicProbing () {
10     int size=0;
11     m=300007;
12     bankStorage1d.resize(m);
13     for (int i=0;i<m;i++){
14         bankStorage1d[i].id="";
15         bankStorage1d[i].balance=-1;
16     }
17
18     bankStorage1d.resize(m);
19
20 }
21
22
23
24 int CubicProbing::hash(std::string id) {
25
26     // int sum = 0;
27
28
29     // for (char c : id) {
30     //     sum += int(c);
31     // }
32
33     // return sum % m;
34     int prod=0;
35     int n=id.length();
36     for (int i=0;i<n;i++){
37         prod+=(int)id[i];
38     }
39     return prod;
40 }
41
42 int CubicProbing::databaseSize() {
43     // int size = 0;
44
45
46     // for (const Account& account : bankStorage1d) {
47     //     if (!account.id.empty()) {
48     //         size++;
49     //     }
```

```

50     // }
51
52     return size;
53
54     // int size = 0;
55
56     // for (int i=0 ;i<100000;i++) {
57
58     //     if (!bankStorage1d[i].id.empty()) {
59     //         size++;
60     //     }
61     // }
62
63
64     // return size;
65 }
66
67
68
69
70
71
72 std::vector<int> merge(const std::vector<int>& left, const std::vector<int>& right) {
73
74
75     std::vector<int> merged;
76
77
78     size_t leftIdx = 0, rightIdx = 0;
79
80     while (leftIdx < left.size() && rightIdx < right.size()) {
81
82         if (left[leftIdx] >= right[rightIdx]) {
83
84             merged.push_back(left[leftIdx]);
85
86             leftIdx++;
87         } else {
88
89             merged.push_back(right[rightIdx]);
90
91             rightIdx++;
92         }
93     }
94
95
96     while (leftIdx < left.size()) {
97         merged.push_back(left[leftIdx]);
98         leftIdx++;
99     }
100
101     while (rightIdx < right.size()) {

```

```
102     merged.push_back(right[rightIdx]);
103     rightIdx++;
104 }
105
106 return merged;
107 }
108
109 // Merge sort function
110 std::vector<int> mergeSort(const std::vector<int>& arr) {
111     if (arr.size() <= 1) {
112         return arr;
113     }
114
115     size_t mid = arr.size() / 2;
116     std::vector<int> left(arr.begin(), arr.begin() + mid);
117     std::vector<int> right(arr.begin() + mid, arr.end());
118
119     left = mergeSort(left);
120     right = mergeSort(right);
121
122     return merge(left, right);
123 }
124
125
126 std::vector<int> CubicProbing::getTopK(int k) {
127
128     std::vector<int> balances;
129     for (const Account& account : bankStorage1d) {
130         if (!account.id.empty()) {
131             balances.push_back(account.balance);
132         }
133     }
134
135
136     balances = mergeSort(balances);
137
138
139     if (k < balances.size()) {
140         balances.resize(k);
141     }
142
143     return balances;
144 }
145
146
147
148
149
150 void CubicProbing::createAccount(std::string id, int count) {
151
152     // int index = hash(id);
153     // size++;
```

```

154
155
156 // int originalIndex = index;
157
158 // int i = 1;
159 // while (!bankStorage1d[index].id.empty() && bankStorage1d[index].id != id) {
160
161
162
163 // index = (originalIndex + (i * i * i)) % m;
164
165 // i++;
166 // }
167
168 // if (bankStorage1d[index].id.empty() || bankStorage1d[index].id == id) {
169
170 // bankStorage1d[index] = {id, count};
171 // }
172 Account new_acc;
173 new_acc.id=id;
174 new_acc.balance=count;
175 int ind=hash(id);
176 if (bankStorage1d[ind].id==" " || bankStorage1d[ind].id=="del"){
177     bankStorage1d[ind].balance=count;
178     bankStorage1d[ind].id=id;
179     size+=1;
180 } else{
181     int j=1;
182     while (bankStorage1d[ind].balance!=-1){
183         ind=(ind+j*j*j)%m;
184         j++;
185     } bankStorage1d[ind].balance=count;
186     bankStorage1d[ind].id=id;
187     size+=1;
188 }
189 }
190
191
192 int CubicProbing::getBalance(std::string id) {
193
194     if (doesExist(id)==false){
195         return -1;
196     } else{
197         int ind=hash(id);
198         int j=1;
199         while (j>0){
200             if (bankStorage1d[ind].id==id){
201                 return bankStorage1d[ind].balance;
202             } ind=(ind+j*j*j)%m;
203             j++;
204         }
205     } return -1;

```



```

206 // int index = hash(id);
207 // int oriindx = index;
208 // int i=1;
209 // while (bankStorage1d[index].id!= id){
210 //     index= (index +(i*i*i))%m;
211 //     i++;
212 //     if (index== oriindx){
213 //         return -1;
214
215 //     }
216 //     return bankStorage1d[index].balance;
217
218
219
220 // }
221
222 }
223
224
225
226
227
228
229 void CubicProbing::addTransaction(std::string id, int count) {
230
231     // int index = hash(id);
232     // int originalIndex = index;
233
234     // int i = 1;
235     // while (!bankStorage1d[index].id.empty()) {
236
237
238     //     if (bankStorage1d[index].id == id) {
239
240
241
242     //         bankStorage1d[index].balance += count;
243
244     //         return;
245     //     }
246
247     //     index = (originalIndex + (i * i * i)) % m;
248     //     i++;
249     // }
250
251
252     // Account newAccount = {id, count};
253     // bankStorage1d[index] = newAccount;
254     int ind=hash(id);
255     if (doesExist(id)==true){
256         int j=1;
257         while (bankStorage1d[ind].id!=""){

```

```

258         if (bankStorage1d[ind].id==id){
259             bankStorage1d[ind].balance+=count;
260             break;
261         } ind=(ind+j*j*j)%m;
262         j++;
263     }
264 } else{
265     createAccount(id,count);
266 }
267
268 }
269
270
271 bool CubicProbing::doesExist(std::string id) {
272
273
274
275     // int index = hash(id);
276
277
278     // int originalIndex = index;
279
280     // int i = 1;
281     // while (!bankStorage1d[index].id.empty()) {
282
283
284         // if (bankStorage1d[index].id == id) {
285
286         //     return true;
287         // }
288
289
290         // index = (originalIndex + i * i * i) % m;
291
292
293         // i++;
294     // }
295
296
297     // return false;
298     int ind=hash(id);
299     int j=1;
300     while (bankStorage1d[ind].id!=""){
301         if (bankStorage1d[ind].id==id){
302             return true;
303         } ind=(ind+j*j*j)%m;
304         j++;
305     }
306     return false;
307 }
308
309

```

```
310
311 bool CubicProbing::deleteAccount(std::string id) {
312
313
314     // int index = hash(id);
315
316     // int originalIndex = index;
317
318     // int i = 1;
319     // while (!bankStorage1d[index].id.empty()) {
320
321         // if (bankStorage1d[index].id == id) {
322
323
324         //     bankStorage1d[index].id.clear();
325
326         //     bankStorage1d[index].balance = 0;
327         //     size--;
328
329         //     return true;
330         // }
331
332
333         // index = (originalIndex + i * i * i) % m;
334         // i++;
335         // }
336
337
338     // return false;
339     if (doesExist(id)==false){
340         return false;
341     } else{
342         int ind=hash(id);
343         int j=1;
344         while(bankStorage1d[ind].id!=""){
345             if (bankStorage1d[ind].id==id){
346                 bankStorage1d[ind].id="del";
347                 bankStorage1d[ind].balance=-1;
348                 size-=1;
349                 return true;
350             } ind=(ind+j*j*j)%m;
351             j++;
352         }
353
354     } return false;
355 }
356
```

```
1  #ifndef CUBICPROBING_H
2  #define CUBICPROBING_H
3
4  #include "BaseClass.h"
5  #include <iostream>
6
7  class CubicProbing : public BaseClass {
8  public:
9      CubicProbing();
10     void createAccount(std::string id, int count) override;
11     std::vector<int> getTopK(int k) override;
12     int getBalance(std::string id) override;
13     void addTransaction(std::string id, int count) override;
14     bool doesExist(std::string id) override;
15     bool deleteAccount(std::string id) override;
16     int databaseSize() override;
17     int hash(std::string id) override;
18
19
20 private:
21     // Other data members and functions specific to Quadratic Probing
22     int size=0;
23     int m=300007;
24
25 };
26
27 #endif // CUBICPROBING_H
```

```
1  #include "LinearProbing.h"
2
3  LinearProbing:: LinearProbing(){
4      int size=0;
5      int m = 300007;
6      bankStorage1d.resize(300007);
7      for (int i=0;i<m;i++){
8          bankStorage1d[i].id="";
9          bankStorage1d[i].balance=-1;
10     }
11
12 }
13
14
15
16 int LinearProbing::hash(std::string id) {
17     int sum=1;
18     int k=id.length();
19
20     for (int i=0;i<k;i++){
21         int c= (int)id[i];
22         sum*=c;
23         sum=sum%m;
24     }
25     return sum;
26 }
27
28
29
30 int LinearProbing::databaseSize() {
31
32     // int count= 0;
33
34     // for (auto account : bankStorage1d) {
35
36     //     if (!account.id.empty()) {
37
38     //         count++;
39     //     }
40     // }
41
42     return size;
43     // int size = 0;
44
45     // for (int i=0 ;i<100000;i++) {
46
47     //     if (!bankStorage1d[i].id.empty()) {
48     //         size++;
49     //     }
```

```

50     // }
51
52
53     // return size;
54 }
55
56
57
58 std::vector<int> merge(const std::vector<int>& left, const std::vector<int>& right) {
59     std::vector<int> merged;
60     size_t leftIdx = 0, rightIdx = 0;
61
62     while (leftIdx < left.size() && rightIdx < right.size()) {
63
64         if (left[leftIdx] >= right[rightIdx]) {
65
66             merged.push_back(left[leftIdx]);
67
68             leftIdx++;
69
70         } else {
71             merged.push_back(right[rightIdx]);
72
73             rightIdx++;
74
75         }
76     }
77
78
79     while (leftIdx < left.size()) {
80
81         merged.push_back(left[leftIdx]);
82
83         leftIdx++;
84     }
85
86     while (rightIdx < right.size()) {
87
88         merged.push_back(right[rightIdx]);
89
90         rightIdx++;
91     }
92
93     return merged;
94 }
95
96
97 std::vector<int> mergeSort(const std::vector<int>& arr) {
98     if (arr.size() <= 1) {
99
100         return arr;
101     }

```

```
102
103     size_t mid = arr.size() / 2;
104
105     std::vector<int> left(arr.begin(), arr.begin() + mid);
106
107
108     std::vector<int> right(arr.begin() + mid, arr.end());
109
110     left = mergeSort(left);
111
112     right = mergeSort(right);
113
114     return merge(left, right);
115 }
116
117
118
119
120
121
122 std::vector<int> LinearProbing::getTopK(int k) {
123
124     std::vector<int> vec;
125     for (const Account& account : bankStorage1d) {
126         if (!account.id.empty()) {
127             vec.push_back(account.balance);
128         }
129     }
130
131
132     vec = mergeSort(vec);
133
134
135     if (k < vec.size()) {
136         vec.resize(k);
137     }
138
139     return vec;
140 }
141
142
143 void LinearProbing::createAccount(std::string id, int count) {
144
145     // int ind= hash(id);
146     // size++;
147
148
149     // while (!bankStorage1d[ind].id.empty() && bankStorage1d[ind].id != id) {
150
151     //     ind = (ind + 1) % m;
152
153     // }
```

```

154
155
156
157 // if (bankStorage1d[ind].id.empty() || bankStorage1d[ind].id == id) {
158
159 //   bankStorage1d[ind] = {id, count};
160 // }
161 Account new_acc;
162 new_acc.id=id;
163 new_acc.balance=count;
164 int ind=hash(id);
165 if (bankStorage1d[ind].id==" " || bankStorage1d[ind].id=="del"){
166     bankStorage1d[ind].balance=count;
167     bankStorage1d[ind].id=id;
168     size+=1;
169 } else{
170     int j=1;
171     while (bankStorage1d[ind].balance!=-1){
172         ind=(ind+1)%m;
173         j++;
174     } bankStorage1d[ind].balance=count;
175     bankStorage1d[ind].id=id;
176     size+=1;
177 }
178
179 }
180
181 int LinearProbing::getBalance(std::string id) {
182
183     if (doesExist(id)==false){
184         return -1;
185     } else{
186         int ind=hash(id);
187         int j=1;
188         while (j>0){
189             if (bankStorage1d[ind].id==id){
190                 return bankStorage1d[ind].balance;
191             } ind=(ind+1)% m;
192             j++;
193         }
194     } return -1;
195 }
196
197 // int index = hash(id);
198
199 // while (bankStorage1d[index].id != id) {
200 //     if (bankStorage1d[index].id == " " ) {
201 //         return -1;
202 //     }
203 //     index = (index + 1) % bankStorage1d.size();
204 // }
205

```



```

206 // return bankStorage1d[index].balance;
207
208
209
210
211
212
213 void LinearProbing::addTransaction(std::string id, int count) {
214
215     // int index = hash(id);
216
217
218     // while (!bankStorage1d[index].id.empty()) {
219
220     //     if (bankStorage1d[index].id == id) {
221     //         bankStorage1d[index].balance += count;
222     //         return;
223     //     }
224     //     index = (index + 1) % m;
225     // }
226     // Account newAccount = {id, count};
227     // bankStorage1d[index] = newAccount;
228     // // createAccount(id,count);
229     int ind=hash(id);
230     if (doesExist(id)==true){
231         int j=1;
232         while (bankStorage1d[ind].id!=""){
233             if (bankStorage1d[ind].id==id){
234                 bankStorage1d[ind].balance+=count;
235                 break;
236             } ind=(ind+1)%m;
237             j++;
238         }
239     } else{
240         createAccount(id,count);
241     }
242 }
243
244
245
246
247
248
249
250 bool LinearProbing::deleteAccount(std::string id) {
251
252     // int index = hash(id);
253
254
255     // while (!bankStorage1d[index].id.empty()) {
256
257     //     if (bankStorage1d[index].id == id) {

```

```

258
259
260 //    bankStorage1d[index].id.clear();
261
262 //    bankStorage1d[index].balance = 0;
263 //    size--;
264
265 //    return true;
266 // }
267 //    index = (index + 1) % m;
268
269 // }
270
271
272 // return false;
273
274 if (doesExist(id)==false){
275     return false;
276 } else{
277     int ind=hash(id);
278     int j=1;
279     while(bankStorage1d[ind].id!=""){
280         if (bankStorage1d[ind].id==id){
281
282             bankStorage1d[ind].id="del";
283
284             bankStorage1d[ind].balance=-1;
285
286             size-=1;
287             return true;
288         } ind=(ind+1)%m;
289         j++;
290     }
291
292 } return false;
293 }
294
295
296
297
298 bool LinearProbing::doesExist(std::string id) {
299
300     // int index = hash(id);
301
302
303     // while (!bankStorage1d[index].id.empty()) {
304
305     //     if (bankStorage1d[index].id == id) {
306
307         //         return true;
308         //     }
309     //     index = (index + 1) % m;

```

```

310
311 // }
312
313
314 // return false;
315 int ind=hash(id);
316 int j=1;
317 while (bankStorage1d[ind].id!=""){
318     if (bankStorage1d[ind].id==id){
319         return true;
320     } ind=(ind+1)%m;
321     j++;
322 }
323 return false;
324
325 }

```

▼ LinearProbing.h

 Download

```

1  #ifndef LINEARPROBING_H
2  #define LINEARPROBING_H
3
4  #include "BaseClass.h"
5  #include <iostream>
6
7  class LinearProbing : public BaseClass {
8  public:
9      LinearProbing();
10     void createAccount(std::string id, int count) override;
11     std::vector<int> getTopK(int k) override;
12     int getBalance(std::string id) override;
13     void addTransaction(std::string id, int count) override;
14     bool doesExist(std::string id) override;
15     bool deleteAccount(std::string id) override;
16     int databaseSize() override;
17     int hash(std::string id) override;
18
19
20 private:
21     // Other data members and functions specific to Linear Probing
22     int size=0;
23     int m =300007;
24 };
25
26 #endif // LINEARPROBING_H

```

```
1
2  #include "QuadraticProbing.h"
3  #include <iostream>
4  #include <vector>
5  using namespace std;
6
7
8  QuadraticProbing:: QuadraticProbing(){
9      int size=0;
10     m=300007;
11     bankStorage1d.resize(m);
12     for (int i=0;i<m;i++){
13         bankStorage1d[i].id="";
14         bankStorage1d[i].balance=-1;
15     }
16
17     bankStorage1d.resize(m);
18
19 }
20
21
22 // int QuadraticProbing::databaseSize() {
23 //     int size = 0;
24
25
26 //     for (const Account& account : bankStorage1d) {
27 //         if (!account.id.empty()) {
28 //             size++;
29 //         }
30 //     }
31
32 //     return size;
33
34
35 // }
36
37 int QuadraticProbing::databaseSize() {
38
39     return size;
40 }
41
42
43 int QuadraticProbing::hash(std::string id) {
44
45     // int sum = 0;
46     // for (char c : id) {
47     //     sum += int (c);
48     // }
49
```

```

50
51 // return sum % m;
52 int prod=1;
53 int n=id.length();
54 for (int i=0;i<n;i++){
55     int c= (int)id[i];
56     prod*=c;
57     prod=prod%m;
58 }
59 return prod;
60 }
61
62
63
64 std::vector<int> merge(const std::vector<int>& left, const std::vector<int>& right) {
65     std::vector<int> merged;
66     size_t leftIdx = 0, rightIdx = 0;
67
68     while (leftIdx < left.size() && rightIdx < right.size()) {
69         if (left[leftIdx] >= right[rightIdx]) {
70             merged.push_back(left[leftIdx]);
71             leftIdx++;
72         } else {
73             merged.push_back(right[rightIdx]);
74             rightIdx++;
75         }
76     }
77
78
79     while (leftIdx < left.size()) {
80         merged.push_back(left[leftIdx]);
81         leftIdx++;
82     }
83
84     while (rightIdx < right.size()) {
85         merged.push_back(right[rightIdx]);
86         rightIdx++;
87     }
88
89     return merged;
90 }
91
92
93 std::vector<int> mergeSort(const std::vector<int>& arr) {
94     if (arr.size() <= 1) {
95         return arr;
96     }
97
98     size_t mid = arr.size() / 2;
99     std::vector<int> left(arr.begin(), arr.begin() + mid);
100    std::vector<int> right(arr.begin() + mid, arr.end());
101

```

```

102     left = mergeSort(left);
103     right = mergeSort(right);
104
105     return merge(left, right);
106 }
107
108
109
110
111 std::vector<int> QuadraticProbing::getTopK(int k) {
112
113     std::vector<int> bal;
114     for (const Account& account : bankStorage1d) {
115         if (!account.id.empty()) {
116             bal.push_back(account.balance);
117         }
118     }
119
120
121     bal = mergeSort(bal);
122
123
124     if (k < bal.size()) {
125         bal.resize(k);
126     }
127
128     return bal;
129
130
131
132
133 }
134
135
136
137
138 void QuadraticProbing::createAccount(std::string id, int count) {
139
140     // int index = hash(id);
141     // int originalIndex = index;
142     // size++;
143
144     // int i = 1;
145     // while (!bankStorage1d[index].id.empty() && bankStorage1d[index].id != id) {
146
147         // index = (originalIndex + i * i) % bankStorage1d.size();
148         // i++;
149     // }
150
151     // if (bankStorage1d[index].id.empty() || bankStorage1d[index].id == id) {
152
153         // bankStorage1d[index] = {id, count};

```

```

154 // }
155 Account new_acc;
156 new_acc.id=id;
157 new_acc.balance=count;
158 int ind=hash(id);
159 if (bankStorage1d[ind].id=="" || bankStorage1d[ind].id=="del"){
160     bankStorage1d[ind].balance=count;
161     bankStorage1d[ind].id=id;
162     size+=1;
163 } else{
164     int j=1;
165     while (bankStorage1d[ind].balance!=-1){
166         ind=(ind+j*j)%m;
167         j++;
168     } bankStorage1d[ind].balance=count;
169     bankStorage1d[ind].id=id;
170     size+=1;
171 }
172 }
173
174
175 int QuadraticProbing::getBalance(std::string id) {
176
177     // int index = hash(id);
178     // int oriindx = index;
179     // int i=1;
180     // while (bankStorage1d[index].id!= id){
181     //     index= (index +(i*i))%m;
182     //     i++;
183     //     if (index== oriindx){
184     //         return -1;
185
186     //     }
187     //     return bankStorage1d[index].balance;
188
189
190
191 // }
192 if (doesExist(id)==false){
193     return -1;
194 } else{
195     int ind=hash(id);
196     int j=1;
197     while (j>0){
198         if (bankStorage1d[ind].id==id){
199             return bankStorage1d[ind].balance;
200         } ind=(ind+j*j)% m;
201         j++;
202     }
203 } return -1;
204
205

```

```

206
207 }
208
209
210 void QuadraticProbing::addTransaction(std::string id, int count) {
211
212     // int index = hash(id);
213     // int originalIndex = index;
214
215     // int i = 1;
216     // while (!bankStorage1d[index].id.empty()) {
217     //     if (bankStorage1d[index].id == id) {
218
219         //     bankStorage1d[index].balance += count;
220         //     return;
221         //     }
222
223
224     //     index = (originalIndex + (i * i)) % m;
225     //     i++;
226     // }
227
228
229     // Account newAccount = {id, count};
230     // bankStorage1d[index] = newAccount;
231
232     int ind=hash(id);
233     if (doesExist(id)==true){
234         int j=1;
235         while (bankStorage1d[ind].id!=""){
236             if (bankStorage1d[ind].id==id){
237                 bankStorage1d[ind].balance+=count;
238                 break;
239             } ind=(ind+j*j)%m;
240             j++;
241         }
242     } else{
243         createAccount(id,count);
244     }
245 }
246
247
248
249
250 bool QuadraticProbing::doesExist(std::string id) {
251
252     // int index = hash(id);
253
254     // int originalIndex = index;
255
256     // int i = 1;
257     // while (!bankStorage1d[index].id.empty()) {

```



```

258
259 // if (bankStorage1d[index].id == id) {
260
261 //     return true;
262 // }
263
264
265 // index = (originalIndex + i * i) % m;
266 // i++;
267 // }
268
269
270 // return false;
271
272 int ind=hash(id);
273 int j=1;
274 while (bankStorage1d[ind].id!=""){
275     if (bankStorage1d[ind].id==id){
276         return true;
277     } ind=(ind+j*j)%m;
278     j++;
279 }
280 return false;
281 }
282
283
284 bool QuadraticProbing::deleteAccount(std::string id) {
285
286     // int index = hash(id);
287
288     // int originalIndex = index;
289
290     // int i = 1;
291
292     // while (!bankStorage1d[index].id.empty()) {
293     //     if (bankStorage1d[index].id == id) {
294
295         //         bankStorage1d[index].id.clear();
296
297         //         bankStorage1d[index].balance = 0;
298         //         size--;
299         //         return true;
300     //     }
301
302
303     //     index = (originalIndex + i * i) % m;
304
305     //     i++;
306
307     // }
308
309

```

```
310 // return false;
311 if (doesExist(id)==false){
312     return false;
313 } else{
314     int ind=hash(id);
315     int j=1;
316     while(bankStorage1d[ind].id!=""){
317         if (bankStorage1d[ind].id==id){
318             bankStorage1d[ind].id="del";
319             bankStorage1d[ind].balance=-1;
320             size-=1;
321             return true;
322         } ind=(ind+j*j)%m;
323         j++;
324     }
325
326 } return false;
327 }
328
329
330
```

```
1  #ifndef QUADRATICPROBING_H
2  #define QUADRATICPROBING_H
3
4  #include "BaseClass.h"
5  #include <iostream>
6
7  class QuadraticProbing : public BaseClass {
8  public:
9      QuadraticProbing();
10     void createAccount(std::string id, int count) override;
11     std::vector<int> getTopK(int k) override;
12     int getBalance(std::string id) override;
13     void addTransaction(std::string id, int count) override;
14     bool doesExist(std::string id) override;
15     bool deleteAccount(std::string id) override;
16     int databaseSize() override;
17     int hash(std::string id) override;
18
19
20 private:
21     // Other data members and functions specific to Quadratic Probing
22     int size=0;
23     int m =300007;
24
25
26
27 };
28
29 #endif // QUADRATICPROBING_H
```