Contents

1 Basic Test Results

```
List of files submitted:
HashMapBinaryOperations.cpp
Makefile
MyHashMap.cpp
MyHashMap.h
MyLinkedList.cpp
MyLinkedList.h


Checking MyLinkedList.cpp
-------------------------
MyLinkedList.cpp file exists, checking compilation with our original example...
Pass compilation MyLinkedList.cpp with ListExample.cpp, check results...
Pass school example.


Checking MyHashMap.cpp
----------------------
MyHashMap.cpp file exists, checking compilation...
Pass compilation MyHashMap.cpp, checking file with test ~slabcpp/public/ex1/code_files/HashSimpleCheck.cpp.
Pass HashSimpleCheck.cpp


Checking Makefile
-----------------
Makefile exists


Checking HashMapBinaryOperations.cpp
------------------------------------
HashMapBinaryOperations.cpp file exists, trying to compile and run it...

---test start---

pass compare 2 HashMap test


pass after - operation test


pass after | operation test


pass after & operation test



Checking TestHashMapBinFuncs.cpp
--------------------------------
Pass basic binary MyHashMap tests.


-------------------------------------------------------------------------

Done running the tests.

NOTE: The presubmission script does not stop on error. You need to go over its output
      and check that there are no ERRORs lines.

```

```
=========================
= Checking coding style =
=========================
 ** Total Violated Rules      : 0
 ** Total Errors Occurs       : 0
 ** Total Violated Files Count: 0
```

2 HashMapBinaryOperations.cpp

```cpp
/**
* HashMapBinaryOperations.cpp
*
* demonstrate the operators
*/
#include <iostream>
#include <math.h>
#include <string>
#include "MyHashMap.h"

using namespace std;

#define EPS 0.000001
#define FAIL 1
#define SUCC 0

/**
* the main
* demonstrate the operators
*/
int main()
{
    MyHashMap h1;
    MyHashMap h2;

    cout << "\n---test start---\n" << endl;

    h1.add("a", 0.2);
    h1.add("aa", 0.3);
    h1.add("aaa", 0.5);
    h1.add("b", 2.2);
    h1.add("bb", 2.3);
    h1.add("bbb", 2.5);
    h2.add("c", 4.2);
    h2.add("cc", 4.3);
    h2.add("ccc", 4.5);
    h2.add("a", 1.2);
    h2.add("aa", 1.3);
    h2.add("aaa", 1.5);

    MyHashMap minus = h1 - h2;
    MyHashMap uniun = h1 | h2;
    MyHashMap inter = h2 & h1;

    //test > < ==
    if (h1 > h2)
    {
        cout << "ERROR: h1 weight is smaller then h2" << endl;
        return FAIL;
    }
    if (h1 == h2)
    {
        cout << "ERROR: h1 weight is smaller then h2" << endl;
        return FAIL;
    }
    if (h2 < h1)
    {
        cout << "ERROR: h1 weight is smaller then h2" << endl;
        return FAIL;
```

```
60      }
61      cout << "pass compare 2 HashMap test\n" << endl;
62
63      //test oprtator -
64      if (fabs(minus.totWeight() - 7) > EPS)
65      {
66          cout << "ERROR: minus weight expected to be: 7" << endl;
67          return FAIL;
68      }
69      if (minus.isIntersect(h2))
70      {
71          cout << "ERROR: fail - operator" << endl;
72          return FAIL;
73      }
74
75      if (!(minus.isIntersect(h1)))
76      {
77          cout << "ERROR: fail - operator" << endl;
78          return FAIL;
79      }
80      if (minus.isInHashMap("a"))
81      {
82          cout << "ERROR: fail - operator" << endl;
83          return FAIL;
84      }
85      if (minus.isInHashMap("c"))
86      {
87          cout << "ERROR: fail - operator" << endl;
88          return FAIL;
89      }
90      cout << "\npass after - operation test\n" << endl;
91
92      //test oprtator |
93      if (fabs(uniun.totWeight() - 21) > EPS)
94      {
95          cout << "ERROR: uniun weight expected to be: 21" << endl;
96          return FAIL;
97      }
98      if (!(uniun.isIntersect(h2)))
99      {
100         cout << "ERROR: fail | operator" << endl;
101         return FAIL;
102     }
103     if (!(uniun.isIntersect(h1)))
104     {
105         cout << "ERROR: fail | operator" << endl;
106         return FAIL;
107     }
108     if (!(uniun.isInHashMap("b")))
109     {
110         cout << "ERROR: fail | operator" << endl;
111         return FAIL;
112     }
113     if (!(uniun.isInHashMap("c")))
114     {
115         cout << "ERROR: fail | operator" << endl;
116         return FAIL;
117     }
118     cout << "\npass after | operation test\n" << endl;
119
120     //test oprtator &
121     if (fabs(inter.totWeight() - 4) > EPS)
122     {
123         cout << "ERROR: inter weight expected to be: 4" << endl;
124         return FAIL;
125     }
126     if (!(inter.isIntersect(h2)))
127     {
```

```
128             cout << "ERROR: fail | operator" << endl;
129             return FAIL;
130         }
131         if (!(inter.isIntersect(h1)))
132         {
133             cout << "ERROR: fail | operator" << endl;
134             return FAIL;
135         }
136         if (inter.isInHashMap("b"))
137         {
138             cout << "ERROR: fail & operator" << endl;
139             return FAIL;
140         }
141         if (inter.isInHashMap("c"))
142         {
143             cout << "ERROR: fail & operator" << endl;
144             return FAIL;
145         }
146         cout << "\npass after & operation test\n" << endl;
147 
148         return SUCC;
149     }
```

3 Makefile

MakeObjects

```
CC = g++ -Wall
H_FILE =  MyHashMap.h MyLinkedList.h
CPP_FILE = MyLinkedList.cpp MyHashMap.cpp

all: HashMapBinaryOperations

HashMapBinaryOperations: $(H_FILE) $(CPP_FILE) HashMapBinaryOperations.cpp
    $(CC) $(CPP_FILE) HashMapBinaryOperations.cpp -o HashMapBinaryOperations

test:
    $(CC) -DTEST MyHashMap.cpp MyLinkedList.cpp -o result
    result

tar:
    tar cvf ex1.tar HashMapBinaryOperations.cpp MyHashMap.h MyHashMap.cpp MyLinkedList.* Makefile

clean:
    rm -f *.o HashMapBinaryOperations result

.PHONY: all tar clean test
```

4 MyHashMap.h

```
/**
 * MyHashMap.h
 *
 * ----------------------------------------------------------------------------------------
 * General: This class represents a Hashmap, a data structure that provide fast accession to
 *          data stored in.
 *          Depend on MyLinkedList
 *
 *  Methods: MyHashMap() - Constructor
 *  ~MyHashMap()         - Destructor
 *
 *  add                  - Add a string to the HashMap. Locate the entry of the relevant linked list in
 *                         the HashMap using the function myHashFunction, and add the element to
 *                         the end of that list.
 *                         If the element already exists in the HashMap, change its data to the
 *                         input parameter. Important: unlike our MyLinkedList, this HashMap will
 *                         contain at most one copy of each std::string.
 *
 *  remove               - Remove a string from the HashMap. Locate the entry of the relevant linked
 *                         list in the HashMap using the function myHashFunction, and remove the element
 *                         from it.
 *                         Return true on success, or false if the element wasn't in the HashMap.
 *
 *  isInHashMap            - Return true if the element is in the HashMap, or false otherwise.
 *                         If the element exists in the HashMap, return in 'data' its appropriate data
 *                         value. Otherwise don't change the value of 'data'.
 *
 *  size                 - Return number of elements stored in the HashMap.
 *
 *  isIntersect            - Return true if and only if there exists a string that belongs both to the
 *                         HashMap h1 and to this HashMap
 *
 *  totWeight            - Return the total weight of the hash elements
 *
 *
 * note: for complexity -
 * n = hashmap size
 * k = relevent list length
 *
 * ----------------------------------------------------------------------------------------
 */
#ifndef MY_HASH_MAP_H
#define MY_HASH_MAP_H

#include <string>
#include "MyLinkedList.h"
#define HASH_SIZE 29     // The number of entrances of the hash table
                         // should be a prim number



/**
 * The definition of the HashMap
 */
class MyHashMap
{
public:
    /**
     * The hash function.
```

```
60          * Input parameter - any C++ string.
61          * Return value: the hash value - the sum of all the characters of the string
62          *   (according to their ASCII value) modulus HASH_SIZE. The hash value of the
63          *   empty string is 0 (since there are no characters, their sum according to
64          *   the ASCII value is 0).
65          * NOTE: In a better design the function would have belong to the string class
66          *     (or to a class that is derived from std::string). We implement it as a "stand
67          *     alone" function since you didn't learn inheritance yet. Keep the function
68          *     global (it's important to the automatic tests).
69          */
70         static int myHashFunction(const std::string &str);
71
72         /**
73          * Class constructor
74          * Complexity - O(1)
75          */
76         MyHashMap();
77
78         /**
79         * Copy constructor
80         * Complexity - O(n)
81         */
82         MyHashMap(const MyHashMap&);
83
84         /**
85          * Class destructor
86          * Complexity - O(n)
87          */
88         ~MyHashMap();
89
90         /**
91          * add a string to the HashMap. Locate the entry of the relevant linked list in
92          * the HashMap using the function myHashFunction, and add the element to the end of that list.
93          * If the element already exists in the HashMap, change its data to the
94          * input parameter. Important: unlike our MyLinkedList, that HashMap will contain at most
95          * one copy of each std::string.
96          * Complexity - O(k)
97          */
98         void add(const std::string &str, double data);
99
100        /**
101         * remove a string from the HashMap. Locate the entry of the relevant linked list in
102         * the HashMap using the function myHashFunction, and remove the element from it.
103         * Return one on success, or zero if the element wasn't in the HashMap.
104         * Complexity - O(k)
105         */
106        size_t remove(const std::string &str);
107
108        /**
109         * Return true if the element is in the HashMap, or false otherwise.
110         * If the element exists in the hash map, return in 'data' its appropriate data
111         * value. Otherwise don't change the value of 'data'.
112         * Complexity - O(k)
113         */
114        bool isInHashMap(const std::string &str, double &data = d) const;
115
116        /**
117         * Return number of elements stored in the HashMap
118         * Complexity - O(1)
119         */
120        int size() const;
121
122        /**
123         * Return true if and only if there exists a string that belong both to the
124         * HashMap h1 and to this HashMap
125         * Complexity - O(n)
126         */
127        bool isIntersect(const MyHashMap &h1) const;
```

```
128
129     /**
130      * Return the total wight of the hash elements
131      * Complexity - O(1)
132      */
133     double totWeight() const;
134
135     /**
136     * Return true is totWeight is equal to h1 totWeight
137     * Complexity - O(1)
138     */
139     bool operator==(const MyHashMap& h1) const;
140
141     /**
142     * Return true is totWeight is smaller then h1 totWeight
143     * Complexity - O(1)
144     */
145     bool operator<(const MyHashMap& h1) const;
146
147     /**
148     * Return true is totWeight is bigger then h1 totWeight
149     * Complexity - O(1)
150     */
151     bool operator>(const MyHashMap& h1) const;
152
153     /**
154     * Minus operation
155     * return new HashMAp contain the element from left HaspMap that doesn't appear in the right one
156     * Complexity - O(n)
157     */
158     MyHashMap operator-(const MyHashMap& h1);
159
160     /**
161     * Union operation
162     * return new HashMAp contain the element from both HashMaps.
163     * in case of duplicate only the element from the left HashMap will be saved.
164     * Complexity - O(n)
165     */
166     MyHashMap operator|(const MyHashMap& h1);
167
168     /**
169     * Intersection operation
170     * return new HashMAp contain only the element appear in both HashMaps.
171     * the data of the element from the left HashMap will be saved.
172     * Complexity - O(n)
173     */
174     MyHashMap operator&(const MyHashMap& h1);
175
176
177     /**
178     * print the hashmap
179     * **only to demonstration use**
180     */
181     void print() const;
182
183 public:
184     /**
185     * update the weight and size of a hashmap for list at the index
186     * complexity - O(1)
187     */
188     void _updateWeight(int index);
189
190     /**
191     * copy the right hashMap into the left one
192     * Complexity - O(n)
193     */
194     MyHashMap& operator=(const MyHashMap&s);
195 private:
```

```
196         // The lists of the hash table
197         MyLinkedList _hashT[HASH_SIZE];
198         unsigned int _size;
199         double _weight;
200     };
201
202     #endif
```

5 MyHashMap.cpp

```cpp
/**
* MyHashMap.h
*
* ----------------------------------------------------------------------------------------
* General: This class represents a Hashmap, a data structure that provide fast accession to
*          data stored in.
*          Depend on MyLinkedList
*/
#include <iostream>
#include <math.h>
#include <string>
#include "MyHashMap.h"


#define EPS 0.000001


using namespace std;

/**
* The hash function.
* Input parameter - any C++ string.
* Return value: the hash value - the sum of all the characters of the string
*   (according to their ASCII value) modulus HASH_SIZE. The hash value of the
*   empty string is 0 (since there are no characters, their sum according to
*   the ASCII value is 0).
* NOTE: In a better design the function would have belong to the string class
*     (or to a class that is derived from std::string). We implement it as a "stand
*     alone" function since you didn't learn inheritance yet. Keep the function
*     global (it's important to the automatic tests).
*/
int MyHashMap::myHashFunction(const std::string &str)
{
    int hash = 0;
    for (unsigned int i = 0; i < str.length(); i++)
    {
        hash += str[i];
    }
    return hash % HASH_SIZE;
}

/**
* Class constructor
*/
MyHashMap::MyHashMap()
{
    _weight = 0;
    _size = 0;
}

/**
* Copy constructor
*/
MyHashMap::MyHashMap(const MyHashMap& h1)
{
    *this = h1;
}

/**
```

```
60  * copy the right hashMap into the left one
61  */
62  MyHashMap& MyHashMap::operator=(const MyHashMap& h1)
63  {
64      for (int i = 0; i < HASH_SIZE; i++)
65      {
66          _hashT[i] = h1._hashT[i];
67      }
68      _weight = h1._weight;
69      _size = h1._size;
70      return *this;
71  }
72
73
74  /**
75  * Class destructor
76  */
77  MyHashMap::~MyHashMap(){ }
78
79  /**
80  * add a string to the HashMap. Locate the entry of the relevant linked list in
81  * the HashMap using the function myHashFunction, and add the element to the end of that list.
82  * If the element already exists in the HashMap, change its data to the
83  * input parameter. Important: unlike our MyLinkedList, that HashMap will contain at most
84  * one copy of each std::string.
85  */
86  void MyHashMap::add(const std::string &str, double data)
87  {
88      int hash = myHashFunction(str);
89      double oldWeight = _hashT[hash].sumList();
90      if (isInHashMap(str) == true)
91      {
92          _hashT[hash].remove(str);
93      }
94      else
95      {
96          _size++;
97      }
98      _hashT[hash].add(str, data);
99      _weight = _weight - oldWeight + _hashT[hash].sumList();
100 }
101
102 /**
103 * remove a string from the HashMap. Locate the entry of the relevant linked list in
104 * the HashMap using the function myHashFunction, and remove the element from it.
105 * Return one on success, or zero if the element wasn't in the HashMap.
106 */
107 size_t MyHashMap::remove(const std::string &str)
108 {
109     int hash = myHashFunction(str);
110     double oldWeight = _hashT[hash].sumList();
111     int result = _hashT[hash].remove(str);
112     if (result)
113     {
114         _weight = _weight - oldWeight + _hashT[hash].sumList();
115     }
116     _size -= result;
117     return result;
118 }
119
120 /**
121 * Return true if the element is in the HashMap, or false otherwise.
122 * If the element exists in the hash map, return in 'data' its appropriate data
123 * value. Otherwise don't change the value of 'data'.
124 */
125 bool MyHashMap::isInHashMap(const std::string &str, double &data) const
126 {
127     return _hashT[myHashFunction(str)].isInList(str, data);
```

```
128 }
129
130 /**
131 * Return number of elements stored in the HashMap
132 */
133 int MyHashMap::size() const
134 {
135     return _size;
136 }
137
138 /**
139 * Return true if and only if there exists a string that belong both to the
140 * HashMap h1 and to this HashMap
141 */
142 bool MyHashMap::isIntersect(const MyHashMap &h1) const
143 {
144     for (int i = 0; i < HASH_SIZE; i++)
145     {
146         if (_hashT[i].isIntersect(h1._hashT[i]))
147         {
148             return true;
149         }
150     }
151     return false;
152 }
153
154 /**
155 * Return the total wight of the hash elements
156 */
157 double MyHashMap::totWeight() const
158 {
159     return _weight;
160 }
161
162 /**
163 * Return true is totWeight is equal to h1 totWeight
164 */
165 bool MyHashMap::operator==(const MyHashMap& h1) const
166 {
167     return fabs(totWeight() - h1.totWeight()) <= EPS;
168 }
169
170 /**
171 * Return true is totWeight is smaller then h1 totWeight
172 */
173 bool MyHashMap::operator<(const MyHashMap& h1) const
174 {
175     return totWeight() - h1.totWeight() < -EPS;
176 }
177
178 /**
179 * Return true is totWeight is bigger then h1 totWeight
180 */
181 bool MyHashMap::operator>(const MyHashMap& h1) const
182 {
183     return totWeight() - h1.totWeight() > EPS;
184 }
185
186 /**
187 * Minus operation
188 * return new HashMAp contain the element from left HaspMap that doesn't appear in the right one
189 */
190 MyHashMap MyHashMap::operator-(const MyHashMap& h1)
191 {
192     MyHashMap minus;
193
194     for (int i = 0; i < HASH_SIZE; i++)
195     {
```

```
196         minus._hashT[i] = _hashT[i] - h1._hashT[i];
197         minus._updateWeight(i);
198     }
199     return minus;
200 }
201
202 /**
203 * Union operation
204 * return new HashMAp contain the element from both HashMaps.
205 * in case of duplicate only the element from the left HashMap will be saved.
206 */
207 MyHashMap MyHashMap::operator|(const MyHashMap& h1)
208 {
209     MyHashMap uniun;
210     for (int i = 0; i < HASH_SIZE; i++)
211     {
212         uniun._hashT[i] = _hashT[i] | h1._hashT[i];
213         uniun._updateWeight(i);
214     }
215
216     return uniun;
217 }
218
219 /**
220 * Intersection operation
221 * return new HashMAp contain only the element appear in both HashMaps.
222 * the data of the element from the left HashMap will be saved.
223 */
224 MyHashMap MyHashMap::operator&(const MyHashMap& h1)
225 {
226     return *this - (*this - h1);
227 }
228
229 void MyHashMap::_updateWeight(int index)
230 {
231     _weight += _hashT[index].sumList();
232     _size += _hashT[index].lengthList();
233 }
234
235 /**
236 * print the hashmap
237 * **only to demonstration use**
238 */
239 void MyHashMap::print() const
240 {
241     for (int i = 0; i < HASH_SIZE; i++)
242     {
243         if (_hashT[i].sumList() != 0)
244         {
245             cout << "  -" << i << " :";
246             _hashT[i].printList();
247         }
248     }
249 }
250
251 #ifdef TEST
252
253 #define FAIL 1
254 #define SUCC 0
255 /**
256 Dear tester.
257 this test has been writen for my own use.
258 since you mention on the ex description to support "test" in makefile
259 I decided to put them here.
260
261 Please do not examinate them since I don't think I should waste more time on it to make it proparly
262 */
263 #define SIZE  100
```

{Attaching your private testers is not a requirement. Please Make sure you do not attach them next time if you are not asked to}

```
264
265 /**
266 * some test of list
267 */
268 int testList()
269 {
270     cout << "\ntest list with duplicate + remove duplicate" << endl;
271     MyLinkedList l;
272     l.add("a", 10);
273     l.add("b", 11);
274     l.add("a", 12);
275     l.add("b", 13);
276     l.add("a", 14);
277     l.add("b", 15);
278     if (l.sumList() != 75)
279     {
280         cout << "list sum should be 75 not " << l.sumList() << endl;
281         return FAIL;
282     }
283     if (l.remove("a") != 3)
284     {
285         cout << "3 a elements should be remove" << endl;
286         return FAIL;
287     }
288     if (l.sumList() != 39)
289     {
290         cout << "list size should be 38 not " << l.sumList() << endl;
291         return FAIL;
292     }
293     if (l.remove("b") != 3)
294     {
295         cout << "3 b elements should be remove" << endl;
296         return FAIL;
297     }
298     if (l.sumList() != 0)
299     {
300         cout << "list sum should be 0 not " << l.sumList() << endl;
301         return FAIL;
302     }
303     cout << "    pass test" << endl;
304     return SUCC;
305 }
306
307 /**
308 * some test of hash function
309 */
310 int testHashFunc()
311 {
312     cout << "\ntests some hash code" << endl;
313     if (MyHashMap::myHashFunction("") != 0)
314     {
315         cout << "wrong hash number for empty string" << endl;
316         return FAIL;
317     }
318     if (MyHashMap::myHashFunction("f") != 15)
319     {
320         cout << "wrong hash number for 'f'" << endl;
321         return FAIL;
322     }
323     if (MyHashMap::myHashFunction("hh") != 5)
324     {
325         cout << "wrong hash number for 'hh'" << endl;
326         return FAIL;
327     }
328     if (MyHashMap::myHashFunction("aaa") != 1)
329     {
330         cout << "wrong hash number for 'aaa'" << endl;
331         return FAIL;
```

```
332     }
333     if (MyHashMap::myHashFunction("cccc") != 19)
334     {
335         cout << "wrong hash number for 'cccc'" << endl;
336         return FAIL;
337     }
338     if (MyHashMap::myHashFunction("bbbbb") != 26)
339     {
340         cout << "wrong hash number for 'bbbb'" << endl;
341         return FAIL;
342     }
343     cout << "    pass test" << endl;
344     return SUCC;
345 }
346 
347 /**
348 * create Big Hash Map
349 */
350 void createBigHashMap(MyHashMap& m)
351 {
352     string ars[SIZE] =
353     {
354         "a", "b", "c", "d", "e", "f", "g", "h", "i", "k",
355         "aa", "bb", "cc", "dd", "ee", "ff", "gg", "hh", "ii", "kk",
356         "aaa", "bbb", "ccc", "ddd", "eee", "fff", "ggg", "hhh", "iii", "kkk",
357         "aaaa", "bbbb", "cccc", "dddd", "eeee", "ffff", "gggg", "hhhh", "iiii", "kkkk",
358         "aaaaa", "bbbbb", "ccccc", "ddddd", "eeeee", "fffff", "ggggg", "hhhhh", "iiiii",
359         "kkkkk",
360         "aaaaaa", "bbbbbb", "cccccc", "dddddd", "eeeeee", "ffffff", "gggggg", "hhhhhh",
361         "iiiiii", "kkkkkk",
362         "aaaaaaa", "bbbbbbb", "ccccccc", "ddddddd", "eeeeeee", "fffffff", "ggggggg",
363         "hhhhhhh", "iiiiiii", "kkkkkkk",
364         "aaaaaaaa", "bbbbbbbb", "cccccccc", "dddddddd", "eeeeeeee", "ffffffff", "gggggggg",
365         "hhhhhhhh", "iiiiiiii", "kkkkkkkk",
366         "aaaaaaaaa", "bbbbbbbbb", "ccccccccc", "ddddddddd", "eeeeeeeee", "fffffffff",
367         "ggggggggg", "hhhhhhhhh", "iiiiiiiii", "kkkkkkkkk",
368         "aaaaaaaaaa", "bbbbbbbbbb", "cccccccccc", "dddddddddd", "eeeeeeeeee", "ffffffffff",
369         "gggggggggg", "hhhhhhhhhh", "iiiiiiiiii", "kkkkkkkkkk",
370     };
371     double ard[SIZE] =
372     {
373         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
374         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
375         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
376         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
377         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
378         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
379         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
380         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
381         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
382         0, 0.001, 0.01, 0.015, 0.02, 0.1, 1, 0.5, 0.12, 0.05,
383     };
384 
385     for (int i = 0; i < SIZE; i++)
386     {
387         m.add(ars[i], ard[i]);
388     }
389 }
390 
391 /**
392 * some test of hash map
393 */
394 int testHashMap(MyHashMap& m)
395 {
396     cout << "\ntest start size" << endl;
397     if (m.size() != 100)
398     {
399         cout << "size should be 100 not " << m.size() << endl;
```

```
400             return FAIL;
401         }
402         cout << "    pass test" << endl;
403 
404         cout << "\ntest start totWeight" << endl;
405         if (fabs(m.totWeight() - 18.16) > EPS)
406         {
407             cout << "m Weight should be 18.16 not " << m.totWeight() << endl;
408             return FAIL;
409         }
410         cout << "    pass test" << endl;
411 
412 
413         cout << "\ntest isInHashMap and remove + updated size and weight" << endl;
414         if (m.isInHashMap("c", d) != 1)
415         {
416             cout << "c should be in m" << endl;
417             return FAIL;
418             if (d != 0.01)
419             {
420                 cout << "wrong data recived" << endl;
421                 return FAIL;
422             }
423         }
424 
425         if (m.isInHashMap("C", d) != 0)
426         {
427             cout << "C shouldnt be in m" << endl;
428             return FAIL;
429         }
430 
431         if (m.remove("kkkkkkkkkk") != 1)
432         {
433             cout << "remove kkkkkkkkkk didnt succeed" << endl;
434             return FAIL;
435         }
436         if (m.remove("kkkkkkkkkk") != 0)
437         {
438             cout << "remove kkkkkkkkkk shouldnt succeed" << endl;
439             return FAIL;
440         }
441         if (m.isInHashMap("kkkkkkkkkk", d) != 0)
442         {
443             cout << "kkkkkkkkkk shouldnt be in m" << endl;
444             return FAIL;
445         }
446         if (fabs(m.totWeight() - 18.11) > EPS)
447         {
448             cout << "m Weight should be 18.11 not " << m.totWeight() << endl;
449             return FAIL;
450         }
451         if (m.size() != 99)
452         {
453             cout << "\nsize should be 99 not " << m.size() << endl;
454             return FAIL;
455         }
456         cout << "    pass test" << endl;
457 
458 
459         cout << "\ntest add exist element + updated size and weight" << endl;
460         m.add("c", 0.05);
461         m.add("kkk", 0.15);
462         if (m.isInHashMap("c", d) != 1)
463         {
464             cout << "c should be in m" << endl;
465             return FAIL;
466             if (d != 0.05)
467             {
```

```
468             cout << "wrong data recived" << endl;
469         }
470     }
471     if (fabs(m.totWeight() - 18.25) > EPS)
472     {
473         cout << "m Weight should be 18.25 not " << m.totWeight() << endl;
474         return FAIL;
475     }
476     if (m.size() != 99)
477     {
478         cout << "m size should be 99 not " << m.size() << endl;
479         return FAIL;
480     }
481     cout << "    pass test" << endl;
482     return SUCC;
483 }
484
485 int testOperators()
486 {
487     MyHashMap h1;
488     MyHashMap h2;
489
490     cout << "\ntest operators < > == | & -" << endl;
491
492     h1.add("a", 0.2);
493     h1.add("aa", 0.3);
494     h1.add("aaa", 0.5);
495     h1.add("b", 2.2);
496     h1.add("bb", 2.3);
497     h1.add("bbb", 2.5);
498     h2.add("c", 4.2);
499     h2.add("cc", 4.3);
500     h2.add("ccc", 4.5);
501     h2.add("a", 1.2);
502     h2.add("aa", 1.3);
503     h2.add("aaa", 1.5);
504
505     MyHashMap minus = h1 - h2;
506     MyHashMap uniun = h1 | h2;
507     MyHashMap inter = h2 & h1;
508
509     //test > < ==
510     if (h1 > h2)
511     {
512         cout << "ERROR: h1 weight is smaller then h2" << endl;
513         return FAIL;
514     }
515     if (h1 == h2)
516     {
517         cout << "ERROR: h1 weight is smaller then h2" << endl;
518         return FAIL;
519     }
520     if (h2 < h1)
521     {
522         cout << "ERROR: h1 weight is smaller then h2" << endl;
523         return FAIL;
524     }
525
526     //test oprtator -
527     if (fabs(minus.totWeight() - 7) > EPS)
528     {
529         cout << "ERROR: minus weight expected to be: 7" << endl;
530         return FAIL;
531     }
532     if (minus.isIntersect(h2))
533     {
534         cout << "ERROR: fail - operator" << endl;
535         return FAIL;
```

```
536     }
537
538     if (!(minus.isIntersect(h1)))
539     {
540         cout << "ERROR: fail - operator" << endl;
541         return FAIL;
542     }
543     if (minus.isInHashMap("a"))
544     {
545         cout << "ERROR: fail - operator" << endl;
546         return FAIL;
547     }
548     if (minus.isInHashMap("c"))
549     {
550         cout << "ERROR: fail - operator" << endl;
551         return FAIL;
552     }
553
554     //test oprtator |
555     if (fabs(uniun.totWeight() - 21) > EPS)
556     {
557         cout << "ERROR: uniun weight expected to be: 21" << endl;
558         return FAIL;
559     }
560     if (!(uniun.isIntersect(h2)))
561     {
562         cout << "ERROR: fail | operator" << endl;
563         return FAIL;
564     }
565     if (!(uniun.isIntersect(h1)))
566     {
567         cout << "ERROR: fail | operator" << endl;
568         return FAIL;
569     }
570     if (!(uniun.isInHashMap("b")))
571     {
572         cout << "ERROR: fail | operator" << endl;
573         return FAIL;
574     }
575     if (!(uniun.isInHashMap("c")))
576     {
577         cout << "ERROR: fail | operator" << endl;
578         return FAIL;
579     }
580
581     //test oprtator &
582     if (fabs(inter.totWeight() - 4) > EPS)
583     {
584         cout << "ERROR: inter weight expected to be: 4" << endl;
585         return FAIL;
586     }
587     if (!(inter.isIntersect(h2)))
588     {
589         cout << "ERROR: fail | operator" << endl;
590         return FAIL;
591     }
592     if (!(inter.isIntersect(h1)))
593     {
594         cout << "ERROR: fail | operator" << endl;
595         return FAIL;
596     }
597     if (inter.isInHashMap("b"))
598     {
599         cout << "ERROR: fail & operator" << endl;
600         return FAIL;
601     }
602     if (inter.isInHashMap("c"))
603     {
```

```
604             cout << "ERROR: fail & operator" << endl;
605             return FAIL;
606         }
607         cout << "    pass test" << endl;
608         return SUCC;
609     }
610 
611     /**
612     * the main test
613     */
614     int main()
615     {
616         double d;
617         MyHashMap m;
618         MyHashMap m2;
619         cout << "\n*******************************************" << endl;
620 
621         if (!(testList() == SUCC))
622         {
623             return FAIL;
624         }
625 
626         if (!(testHashFunc() == SUCC))
627         {
628             return FAIL;
629         }
630 
631         //test empty HashMap
632         cout << "\ntest initial size" << endl;
633         if (m.size() != 0)
634         {
635             cout << "size not 0" << endl;
636             return FAIL;
637         }
638         cout << "    pass test" << endl;
639 
640         createBigHashMap(m);
641 
642         MyHashMap m3(m);
643 
644         if (!(testHashMap(m) == SUCC))
645         {
646             return FAIL;
647         }
648         //test intersect
649         cout << "\ntest isIntersect" << endl;
650         m2.add("A", 2);
651         if (m.isIntersect(m2) != 0)
652         {
653             cout << "m & m2 shouldnt intersect" << endl;
654             return FAIL;
655         }
656         m2.add("a", 2);
657         if (m.isIntersect(m2) != 1)
658         {
659             cout << "m & m2 should intersect" << endl;
660             return FAIL;
661         }
662         cout << "    pass test" << endl;
663 
664         //test m2
665         cout << "\ntest emptying m2" << endl;
666         if (m2.remove("A") != 1)
667         {
668             cout << "remove A didnt succeed" << endl;
669             return FAIL;
670         }
671         if (m2.remove("a") != 1)
```

```
672     {
673         cout << "remove a didnt succeed" << endl;
674         return FAIL;
675     }
676     if (fabs(m2.totWeight() - 0) > EPS)
677     {
678         cout << "m2 Weight should be 0 not " << m2.totWeight() << endl;
679         return FAIL;
680     }
681     if (m2.size() != 0)
682     {
683         cout << "m2 size should be 0 not " << m2.size() << endl;
684         return FAIL;
685     }
686     cout << "    pass test" << endl;
687 
688     //test m3
689     cout << "\ntest m3 - copy constructor" << endl;
690     if (fabs(m3.totWeight() - 18.16) > EPS)
691     {
692         cout << "m3 Weight should be 18.16 not " << m3.totWeight() << endl;
693         return FAIL;
694     }
695     if (m3.size() != 100)
696     {
697         cout << "m3 size should be 100 not " << m3.size() << endl;
698         return FAIL;
699     }
700     if (m3.isInHashMap("c", d) != 1)
701     {
702         cout << "c should be in m3" << endl;
703         if (d != 1)
704         {
705             cout << "wrong data recived" << endl;
706             return FAIL;
707         }
708     }
709     if (m3.remove("kkkkkkkkkk") != 1)
710     {
711         cout << "remove kkkkkkkkkk didnt succeed" << endl;
712         return FAIL;
713     }
714     if (m3.remove("kkkkkkkkkk") != 0)
715     {
716         cout << "remove kkkkkkkkkk shouldnt succeed" << endl;
717         return FAIL;
718     }
719     if (m3.size() != 99)
720     {
721         cout << "m3 size should be 99 not " << m3.size() << endl;
722         return FAIL;
723     }
724     if (fabs(m3.totWeight() - 18.11) > EPS)
725     {
726         cout << "m3 Weight should be 18.11 not " << m3.totWeight() << endl;
727         return FAIL;
728     }
729     cout << "    pass test" << endl;
730 
731     if (!(testOperators() == SUCC))
732     {
733         return FAIL;
734     }
735     cout << "\n*****************************************\n" << endl;
736     return 0;
737 }
738 #endif
```

6 MyLinkedList.h

```
1  /**
2  * MyLinkedList.h
3  *
4  * ---------------------------------------------------------------------------------------------
5  * General: This class represents a Linkedlist, a data structure that every node point to the next
6  * and prev nodes
7  *
8  *  Methods:
9  *  MyLinkedList()    - Constructor
10 *  ~MyLinkedList()      - Destructor
11 *
12 *  add                  - Add a string to the LinkedList
13 *
14 *  remove               - Remove all element of string from the list
15 *
16 *  isInList        - Return true if the element is in the LinkedList, or false otherwise.
17 *                        If the element exists in the HashMap, return in 'data' its appropriate
18 *              data value. Otherwise don't change the value of 'data'.
19 *
20 *  printList        - print the list
21 *
22 *  sumList        - Return the total weight of the hash elements
23 *
24 *  getNext        - Return the node next
25 *
26 *  getKey        - Return the node key
27 *
28 *  getData        - Return the node data
29 *
30 *  operator=        - copy the right list into the left one
31 *
32 *
33 * note: for complexity - n=list length
34 *
35 * ---------------------------------------------------------------------------------------------
36 */
37 #include <string>
38
39 #include <iostream>
40
41 using namespace std;      using_namespace_header
42
43 #ifndef _MYLINKEDLIST_H_
44 #define _MYLINKEDLIST_H_
45
46 /**
47 * represent a Node of the LinkedList
48 *
49 * complexity: all method are work in O(1)
50 */
51 class MyLinkedListNode;
52
53 // static double for default parameter for isInList
54 static double d = 0;
55
56 /**
57 * The definition of the LinkedList
58 */
59 class MyLinkedList
```

```
60  {
61  public:
62      /**
63      * Class constructor
64      * Complexity - O(1)
65      */
66      MyLinkedList();
67
68      /**
69      * copy constructor
70      * Complexity O(n)
71      */
72      MyLinkedList(MyLinkedList& other);
73
74      /**
75      * Class disstructor
76      * Complexity - O(n)
77      */
78      ~MyLinkedList();
79
80      /**
81      * Add a string to the LinkedLis
82      * Complexity - O(1)
83      */
84      void add(const string key, const double data);
85
86      /**
87      * Remove all element with the same key from the list
88      * Complexity - O(n)
89      */
90      int remove(const string key);
91
92      /**
93      * Return true if the element is in the LinkedList, or false otherwise.
94      * If the element exists in the HashMap, return in 'data' its appropriate data
95      * value. Otherwise don't change the value of 'data'.
96      * Complexity - O(n)
97      */
98      bool isInList(const string key, double& data = d) const;
99
100     /**
101     * print the list
102     * Complexity - O(n)
103     */
104     void printList() const;
105
106     /**
107     * Return the total weight of the list
108     * Complexity - O(1)
109     */
110     double sumList() const;
111
112     /**
113     * Return the length of the list
114     * Complexity - O(1)
115     */
116     int lengthList() const;
117
118     /**
119     * Return true if and only if there exists a string that belong both lists
120     * Complexity - O(n*m)
121     */
122     bool isIntersect(const MyLinkedList &h1) const;
123
124     /**
125     * copy the right list into the left one
126     * Complexity - O(n)
127     */
```

```
128     MyLinkedList& operator=(const MyLinkedList&);
129
130     /**
131     * Minus operation
132     * return new LinkedList contain the element from left LinkedList that doesn't appear in the right one
133     * Complexity - O(n)
134     *
135     * note: the operator should use on set only!
136     */
137     MyLinkedList operator-(const MyLinkedList& l1);
138
139     /**
140     * Union operation
141     * return new LinkedList contain the element from both LinkedList.
142     * in case of duplicate only the element from the left LinkedList will be saved.
143     * Complexity - O(n)
144     *
145     * note: the operator should use on set only!
146     */
147     MyLinkedList operator|(const MyLinkedList& l1);
148
149
150 private:
151     /**
152     * helper method.
153     * remove the spesific node from the list
154     * Complexity - O(1)
155     */
156     void _removeNode(MyLinkedListNode* nodeToRemove);
157     double _sum;
158     int _length;
159     MyLinkedListNode* _head;
160     MyLinkedListNode* _tail;
161 };
162 #endif
```

7 MyLinkedList.cpp

```cpp
/**
* MyLinkedList.h
*
* ------------------------------------------------------------------------------------------------------------------
* General: This class represents a Linkedlist, a data structure that every node point to the next and prev nodes
*/
#include "MyLinkedList.h"
#include <iostream>
#include <string>


using namespace std;

/**
* represent a Node of the LinkedList
*
* complexity: all method are work in O(1)
*/
class MyLinkedListNode
{
public:
    MyLinkedListNode(const string key, const double data);
    ~MyLinkedListNode();
    MyLinkedListNode* getNext();
    MyLinkedListNode* getPrev();
    void setNext(MyLinkedListNode* next);
    void setPrev(MyLinkedListNode* prev);
    double& getData();
    string& getKey();

private:

    double _data;
    string _key;
    MyLinkedListNode* _next;
    MyLinkedListNode* _prev;

};

/**
* The constructor
*/
MyLinkedListNode::MyLinkedListNode(const string key, const double data)
{
    _key = key;
    _data = data;
    _next = NULL;
    _prev = NULL;
}


/**
* The distructor
*/
MyLinkedListNode::~MyLinkedListNode(){}
MyLinkedListNode* MyLinkedListNode::getNext()
{
    return _next;
}
```

```
60  MyLinkedListNode* MyLinkedListNode::getPrev()
61  {
62      return _prev;
63  }
64  void MyLinkedListNode::setNext(MyLinkedListNode* next)
65  {
66      _next = next;
67      if (_next != NULL)
68      {
69          next->_prev = this;
70      }
71  }
72  void MyLinkedListNode::setPrev(MyLinkedListNode* prev)
73  {
74      _prev = prev;
75      if (prev != NULL)
76      {
77          prev->_next = this;
78      }
79  }
80  double& MyLinkedListNode::getData()
81  {
82      return _data;
83  }
84  string& MyLinkedListNode::getKey()
85  {
86      return _key;
87  }
88
89  MyLinkedList::MyLinkedList()
90  {
91      _head = NULL;
92      _tail = NULL;
93      _length = 0;
94      _sum = 0;
95  }
96
97  MyLinkedList::MyLinkedList(MyLinkedList& other)
98  {
99          *this = other;
100 }
101
102 MyLinkedList::~MyLinkedList()
103 {
104     MyLinkedListNode* node = _head;
105     while (_head != NULL)
106     {
107         node = node->getNext();
108         _removeNode(_head);
109     }
110 }
111
112 void MyLinkedList::add(const string key, const double data)
113 {
114     MyLinkedListNode* newNode = new MyLinkedListNode(key, data);
115     if (_head == NULL)
116     {
117         _head = newNode;
118     }
119     else
120     {
121         _tail->setNext(newNode);
122     }
123     _tail = newNode;
124     _length++;
125     _sum += data;
126 }
127
```

```
128 void MyLinkedList::_removeNode(MyLinkedListNode* nodeToRemove)
129 {
130     MyLinkedListNode* node = nodeToRemove;
131     _sum -= node->getData();
132     _length--;
133     if (node == _head)
134     {
135         if (_head == _tail)
136         {
137             _head = NULL;
138             _tail = NULL;
139             delete(node);
140             return;
141         }
142         _head = node->getNext();
143     }
144     else
145     {
146         node->getPrev()->setNext(node->getNext());
147     }
148     if (node == _tail)
149     {
150         _tail = node->getPrev();
151     }
152     else
153     {
154         node->getNext()->setPrev(node->getPrev());
155     }
156     delete(node);
157 }
158
159 int MyLinkedList::remove(const string key)
160 {
161     MyLinkedListNode* node = _head;
162     MyLinkedListNode* nodeToDel;
163     int num = 0;
164     while (node != NULL)
165     {
166         nodeToDel = node;
167         node = node->getNext();
168         if (nodeToDel->getKey() == key)
169         {
170             _removeNode(nodeToDel);
171             num++;
172         }
173     }
174     return num;
175 }
176
177 bool MyLinkedList::isInList(const string key, double& data) const
178 {
179     MyLinkedListNode* node = _head;
180     while (node != NULL)
181     {
182
183         if (node->getKey() == key)
184         {
185             data = node->getData();
186             return true;
187         }
188         node = node->getNext();
189
190     }
191     return false;
192 }
193
194 void MyLinkedList::printList() const
195 {
```

```
196     if (_head == NULL)
197     {
198         cout << "Empty" << endl;
199         return;
200     }
201     MyLinkedListNode* node = _head;
202     while (node != NULL)
203     {
204         cout << node->getKey() << "," << node->getData() << endl;
205         node = node->getNext();
206     }
207 }
208 
209 double MyLinkedList::sumList() const
210 {
211     return _sum;
212 }
213 
214 /**
215 * Return the size of the list
216 */
217 int MyLinkedList::lengthList() const
218 {
219     return _length;
220 }
221 
222 /**
223 * Return true if and only if there exists a string that belong both to the
224 * HashMap h1 and to this HashMap
225 */
226 bool MyLinkedList::isIntersect(const MyLinkedList &l1) const
227 {
228     MyLinkedListNode* node;
229     node = _head;
230     while (node != NULL)
231     {
232         if (l1.isInList(node->getKey()))
233         {
234             return true;
235         }
236         node = node->getNext();
237     }
238     return false;
239 }
240 
241 MyLinkedList& MyLinkedList::operator=(const MyLinkedList& l1)
242 {
243     if (this == &l1)
244     {
245         return *this;
246     }
247     //reset the list
248     _head = NULL;
249     _tail = NULL;
250     _length = 0;
251     _sum = 0;
252     //copy the list
253     MyLinkedListNode* node = l1._head;
254     while (node != NULL)
255     {
256         add(node->getKey(), node->getData());
257         node = node->getNext();
258     }
259     return *this;
260 }
261 
262 /**
263 * Minus operation
```

```
264 * return new LinkedList contain the element from left LinkedList that doesn't appear in the right one
265 */
266 MyLinkedList MyLinkedList::operator-(const MyLinkedList& l1)
267 {
268     MyLinkedList list;
269     MyLinkedListNode* node;
270     node = _head;
271     while (node != NULL)
272     {
273         if (!l1.isInList(node->getKey()))
274         {
275             list.add(node->getKey(), node->getData());
276         }
277         node = node->getNext();
278     }
279     return list;
280 }
281
282 /**
283 * Union operation
284 * return new LinkedList contain the element from both LinkedList.
285 * in case of duplicate only the element from the left LinkedList will be saved.
286 */
287 MyLinkedList MyLinkedList::operator|(const MyLinkedList& l1)
288 {
289     MyLinkedList list;
290     list = *this;
291     MyLinkedListNode* node;
292     node = l1._head;
293     while (node != NULL)
294     {
295         if (!isInList(node->getKey()))
296         {
297             list.add(node->getKey(), node->getData());
298         }
299         node = node->getNext();
300     }
301     return list;
302 }
```