Contents

1 Basic Test Results

```
1
2  Running presubmission script...
3
4
5  Opening tar file
6  OK
7  Tar extracted O.K.
8  For your convenience, the MD5 checksum for your submission is e3d55f9a016f89699a93cae546bb2c89
9  Checking files...
10 OK
11 Making sure files are not empty...
12 OK
13 Checking CodingStyle...
14 Checking file HashMap.hpp...
15 Checking file Dictionary.hpp...
16 Passed codingStyle check.
17 Compilation check...
18 Compiling...
19
20 Compilation looks good!
21
22
23 ====================
24  Public test cases
25 ====================
26
27 Running test...
28 [#0][Presubmission] Test __presubmit_testCreateHashMaps... OK!
29 [#1][Presubmission] Test __presubmit_testInsert... OK!
30 [#2][Presubmission] Test __presubmit_testSize... OK!
31 [#3][Presubmission] Test __presubmit_testCapacity... OK!
32 [#4][Presubmission] Test __presubmit_testEmpty... OK!
33 [#5][Presubmission] Test __presubmit_testClear... OK!
34 [#6][Presubmission] Test __presubmit_testBucketSize... OK!
35 [#7][Presubmission] Test __presubmit_testGetElement... OK!
36 [#8][Presubmission] Test __presubmit_testContainsKey... OK!
37 [#9][Presubmission] Test __presubmit_testAssignment... OK!
38 [#10][Presubmission] Test __presubmit_testComparison... OK!
39 [#11][Presubmission] Test __presubmit_testIterator... OK!
40 [#12][Presubmission] Test __presubmit_testVectorsCtor... OK!
41 [#13][Presubmission] Test __presubmit_testCopyCtor... OK!
42
43
44 OK
45 ****************************************
46 *                  ***                 *
47 *          Passed all tests!!          *
48 *              Good Job!               *
49 *                  ***                 *
50 ****************************************
```

2 ex6/README.md

```
1  # ex6-ohadrav
```

3 ex6/Dictionary.hpp

```cpp
#ifndef _DICTIONARY_H_
#define _DICTIONARY_H_

#include "HashMap.hpp"
#include <string>

/*
 * Class Dictionary is a derived class from HashMap, and it contains the
 * specific implementation of HashMap when used with string as key and as
 * value.
 */

// Class InvalidKey is used for exceptions.
class InvalidKey : public std::invalid_argument
{
 public:
  InvalidKey() : std::invalid_argument ("Invalid key") {}
  InvalidKey(std::string msg) : std::invalid_argument (msg) {}
};

class Dictionary : public HashMap<std::string, std::string>
{
 public:
  Dictionary() {}
  Dictionary(std::vector<std::string> keys, std::vector<std::string> values)
  : HashMap<std::string, std::string>(keys, values) {}

  bool erase(const std::string &key)
  {
    if (!HashMap<std::string, std::string>::erase(key))
    {
     throw InvalidKey();
    }
    return true;
  }

  void update(std::vector<std::pair<std::string, std::string>>::iterator it,
              std::vector<std::pair<std::string, std::string>>::iterator
              end_it)
  {
    while (it != end_it)
    {
      (*this)[it->first] = it->second;
      ++it;
    }
  }
};

#endif //_DICTIONARY_H_
```

4 ex6/HashMap.hpp

```cpp
#ifndef _HASHMAP_H_
#define _HASHMAP_H_

#define INIT_TABLE_SIZE 16
#define MIN_LOAD_FACTOR 0.25
#define MAX_LOAD_FACTOR 0.75
#include <vector>
#include <stdexcept>

template <typename KeyT, typename ValueT>

/*
 * Class HashMap is the generic base class which is used for the data base.
 * It holds a vector of pointers where each one represents a bucket. The
 * init table size and the top and bottom load factors are set according to
 * the ex requirements. The class holds all the required methods and
 * operators, as well as a few extra methods I've added in the private
 * section in order to complete the task.
 */

class HashMap {
 public:
  HashMap() {
    _capacity = INIT_TABLE_SIZE;
    _size = 0;
    _arr = new std::vector<std::pair<KeyT, ValueT>> [_capacity];
  }

  HashMap(std::vector<KeyT> keys, std::vector<ValueT> values) : HashMap()
  {
    if (keys.size() != values.size())
    {
      throw std::length_error("The vectors have different lengths.");
    }
    _arr = new std::vector<std::pair<KeyT, ValueT>> [_capacity];
    for (size_t i = 0; i < keys.size(); ++i)
    {
      (*this)[keys[i]] = values[i];
    }
  }

  HashMap(const HashMap &hm) : _capacity(hm._capacity), _size(hm._size)
  {
    _arr = new std::vector<std::pair<KeyT, ValueT>> [_capacity];
    for (int i = 0; i < _capacity; ++i)
    {
      _arr[i] = hm._arr[i];
    }
  }

  virtual ~HashMap()
  {
    delete [] _arr;
  }

  int size() const
  {
    return _size;
  }
```

```
60
61      int capacity() const
62      {
63        return _capacity;
64      }
65
66      bool empty() const
67      {
68        return _size == 0;
69      }
70
71      bool insert(KeyT key, ValueT val)
72      {
73        int index = std::hash<KeyT>()(key) & (_capacity - 1);
74        for (std::pair<KeyT, ValueT> &i : _arr[index])
75        {
76          if (i.first == key)
77          {
78            return false;
79          }
80        }
81        _arr[index].push_back(std::make_pair (key, val));
82        _size++;
83        if (_size > _capacity * _max_lf)
84        {
85          rehash (_capacity * 2);
86        }
87        return true;
88      }
89
90      bool contains_key(KeyT key) const
91      {
92        int index = std::hash<KeyT>()(key) & (_capacity - 1);
93        for (const std::pair<KeyT, ValueT> &i : _arr[index])
94        {
95          if (i.first == key)
96          {
97            return true;
98          }
99        }
100       return false;
101     }
102
103     ValueT& at(KeyT key)
104     {
105       int index = std::hash<KeyT>()(key) & (_capacity - 1);
106       for (std::pair<KeyT, ValueT> &i : _arr[index])
107       {
108         if (i.first == key)
109         {
110           return i.second;
111         }
112       }
113       throw std::out_of_range("Index not found");
114     }
115
116     virtual bool erase(KeyT key)
117     {
118       int index = std::hash<KeyT>()(key) & (_capacity - 1);
119       for (typename std::vector<std::pair<KeyT, ValueT>>::iterator i =
120           _arr[index].begin(); i != _arr[index].end(); ++i)
121       {
122         if (i->first == key)
123         {
124           _arr[index].erase(i);
125           _size--;
126           while (_size < _capacity * _min_lf && _capacity > 1)
127           {
```

```
128             rehash (_capacity / 2);
129           }
130           return true;
131         }
132       }
133       return false;
134     }
135
136     double get_load_factor() const
137     {
138       return (double)_size / (double)_capacity;
139     }
140
141     int bucket_size(KeyT key) const
142     {
143       int index = std::hash<KeyT>()(key) & (_capacity - 1);
144       for (const std::pair<KeyT, ValueT> &i : _arr[index])
145       {
146         if (i.first == key)
147         {
148           return _arr[index].size();
149         }
150       }
151       throw std::range_error("Key does not exist.");
152     }
153
154     int bucket_index(KeyT key) const
155     {
156       int index = std::hash<KeyT>()(key) & (_capacity - 1);
157       for (const std::pair<KeyT, ValueT> &i : _arr[index])
158       {
159         if (i.first == key)
160         {
161           return index;
162         }
163       }
164       throw std::range_error("Key does not exist.");
165     }
166
167     void clear()
168     {
169       for (int i = 0; i < _capacity; ++i)
170       {
171         _arr[i].clear();
172       }
173       _size = 0;
174     }
175
176     HashMap &operator=(const HashMap &hm)
177     {
178       if (this == &hm)
179       {
180         return *this;
181       }
182       delete [] _arr;
183       _capacity = hm._capacity;
184       _size = hm._size;
185       _arr = new std::vector<std::pair<KeyT, ValueT>> [_capacity];
186       for (int i = 0; i < _capacity; ++i)
187       {
188         _arr[i] = hm._arr[i];
189       }
190       return *this;
191     }
192
193     ValueT &operator[](const KeyT &key)
194     {
195       int index = std::hash<KeyT>()(key);
```

```
196       index &= (_capacity - 1);
197       for (std::pair<KeyT, ValueT> &i : _arr[index])
198       {
199         if (i.first == key)
200         {
201           return i.second;
202         }
203       }
204       _arr[index].push_back(std::make_pair (key, ValueT()));
205       _size++;
206       if (_size > _capacity * _max_lf)
207       {
208         rehash (_capacity * 2);
209       }
210       return _arr[index].back().second;
211     }
212
213     bool operator==(const HashMap &rhs) const
214     {
215       if (_size != rhs._size)
216       {
217         return false;
218       }
219       for (auto &i : rhs)
220       {
221         if (!contains_pair(i))
222         {
223           return false;
224         }
225       }
226       return true;
227     }
228
229     bool operator!=(const HashMap &rhs) const
230     {
231       return !(*this == rhs);
232     }
233
234
235     /*
236      * This is a nested class to implement an iterator for the hash map to be
237      * used. The iterator class holds a constructor and all of the operators
238      * an iterator needs to have. The functions of begin and end are held in
239      * the parent HashMap class under the private section.
240      */
241     typedef class ConstIterator
242     {
243      public:
244       ConstIterator(const std::vector<std::pair<KeyT, ValueT>> *p,
245                 const HashMap<KeyT, ValueT> *base) : ptr(p), itr(p->cbegin()),
246                 base
247                 (base)
248                     {
249                        while (ptr != base->_arr + base->_capacity && itr ==
250                        ptr->cend())
251                        {
252                          ++ptr;
253                          itr = ptr->cbegin();
254                        }
255                      }
256
257       ConstIterator &operator++()
258       {
259         ++itr;
260         while (ptr != base->_arr + base->_capacity && itr == ptr->cend())
261         {
262           ++ptr;
263           itr = ptr->cbegin();
```

```
264         }
265         return *this;
266       }
267
268       ConstIterator &operator++(int)
269       {
270         auto save = *this;
271         ++*this;
272         return save;
273       }
274
275       bool operator==(const ConstIterator &rhs) const
276       {
277         return itr == rhs.itr;
278       }
279
280       bool operator!=(const ConstIterator &rhs) const
281       {
282         return itr != rhs.itr;
283       }
284
285       const std::pair<KeyT, ValueT> &operator*() const
286       {
287         return *itr;
288       }
289
290       const std::pair<KeyT, ValueT> *operator->() const
291       {
292         return &*itr;
293       }
294
295      private:
296       const std::vector<std::pair<KeyT, ValueT>> *ptr;
297       typename std::vector<std::pair<KeyT, ValueT>>::const_iterator itr;
298       const HashMap<KeyT, ValueT> *base;
299     } const_iterator;
300
301     // These are the begin functions used for the iterator.
302     ConstIterator begin() const
303     {
304       return ConstIterator(_arr, this);
305     }
306
307     ConstIterator cbegin() const
308     {
309       return ConstIterator(_arr, this);
310     }
311
312     // These are the end functions used for the iterator.
313     ConstIterator end() const
314     {
315       return ConstIterator(_arr + _capacity, this);
316     }
317
318     ConstIterator cend() const
319     {
320       return ConstIterator(_arr + _capacity, this);
321     }
322
323    private:
324     int _capacity;
325     int _size;
326     double _min_lf = MIN_LOAD_FACTOR;
327     double _max_lf = MAX_LOAD_FACTOR;
328     std::vector<std::pair<KeyT, ValueT>> *_arr;
329
330     // This is a function I've added which checks if a map contains a pair.
331     // It's working pretty similar to the contains_ket function but here we
```

```
332     // want to check if a pair exists in the map. It is used by the ==
333     // operator of the hash map.
334     bool contains_pair(const std::pair<KeyT, ValueT> &pair) const
335     {
336       int index = std::hash<KeyT>()(pair.first) & (_capacity - 1);
337       for (std::pair<KeyT, ValueT> &i : _arr[index])
338       {
339         if (i.first == pair.first)
340         {
341           return i.second == pair.second;
342         }
343       }
344       return false;
345     }
346
347     // This function is used to rehash and reorganize the map once we exceed
348     // the top or bottom load factor that was defined. It is used by the
349     // insert and erase functions as well as by the [] operator - all of which
350     // can enter or delete pairs from the map and change the current load
351     // factor.
352     void rehash(int new_capacity)
353     {
354       auto new_arr = new std::vector<std::pair<KeyT, ValueT>> [new_capacity];
355       for (int index = 0; index < _capacity; ++index)
356       {
357         for (const std::pair<KeyT, ValueT> &i : _arr[index])
358         {
359           int new_index = std::hash<KeyT>()(i.first) & (new_capacity - 1);
360           new_arr[new_index].push_back(i);
361         }
362       }
363       delete [] _arr;
364       _arr = new_arr;
365       _capacity = new_capacity;
366     }
367   };
368
369
370
371
372   #endif
```