

# Contents

<a href="#">2</a>	<a href="#">ex6/README.md</a>	<a href="#">3</a>
<a href="#">3</a>	<a href="#">ex6/Dictionary.hpp</a>	<a href="#">4</a>
<a href="#">4</a>	<a href="#">ex6/HashMap.hpp</a>	<a href="#">6</a>



## 2 ex6/README.md

1 # ex6-michaelmerzin

## 3 ex6/Dictionary.hpp

```
1
2 // Created by yurim on 6/12/2022.
3 //
4 #include "HashMap.hpp"
5 #ifndef _DICTIONARY_HPP_
6 #define _DICTIONARY_HPP_
7 class InvalidKey:public std::invalid_argument
8 {
9     public:
10     InvalidKey():std::invalid_argument("")
11     {}
12     InvalidKey(const std::string &a):std::invalid_argument(a)
13     {}
14 };
15 class Dictionary: public HashMap<std::string ,std::string>
16 {
17     private:
18     public :
19     Dictionary()= default;
20     Dictionary(std::vector<std::string> first, std::vector<std::string> second)
21     {
22         if(first.size()!=second.size())
23         {
24             throw std::out_of_range ("Error size");
25         }
26         for (unsigned i = 0; i < first.size(); ++i)
27         {
28             this->insert (first[i],second[i]);
29         }
30     }
31 }
32 bool erase(std::string key) override
33 {
34     bool check = HashMap<std::string, std::string>::erase(key);
35     if(!check)
36     {
37         throw InvalidKey("error");
38     }
39     return true;
40 }
41 template<class Itertaor>
42 void update(Itertaor a
43             ,Itertaor b)
44 {
45     for (auto i = a; i !=b; ++i)
46     {
47         std::hash<std::string> hash_func;
48         int check = hash_func (i->first) & (capacity1-1);
49         bool check2 =this->contains_key (i->first);
50         if(check2)
51         {
52
53             for (unsigned j = 0; j < this->table[check].size(); ++j)
54             {
55                 if( this->table[check][j].first==i->first)
56                 {
57                     this->table[check][j].second=i->second;
58                 }
59             }
60         }
61     }
62 }
```

```
60
61     }
62     else
63     {
64         table[check].push_back ({i->first,i->second});
65         M++;
66     }
67 }
68
69 }
70 };
71
72 #endif //_DICTIONARY_HPP_
```

## 4 ex6/HashMap.hpp

```
1  #ifndef _HASHMAP_HPP_
2  #define _HASHMAP_HPP_
3  #include <algorithm>
4  #include <utility>
5  #include <iostream>
6  #include <vector>
7  #define DEFSIZE 16
8  #define LOWER 0.25
9  #define UPPER 0.75
10 #define A12 12
11 template<typename KeyT, typename ValT>
12 class HashMap
13 {
14     protected:
15         int capacity1;
16         int M;
17         std::vector<std::pair<KeyT, ValT >> *table;
18     public :
19         friend class Iterator;
20         friend class ConstIterator;
21         HashMap ()
22         {
23             table = new std::vector<std::pair<KeyT, ValT>>[DEFSIZE];
24             capacity1 = DEFSIZE;
25             M = 0;
26         }
27         HashMap (HashMap<KeyT, ValT> const &a)
28         {
29             capacity1 = a.capacity1;
30             M = a.M;
31             this->table = new std::vector<std::pair<KeyT, ValT>>[a.capacity1];
32             for (int i = 0; i < a.capacity1; ++i)
33             {
34                 table[i] = a.table[i];
35             }
36         }
37     }
38     void help_when_if(std::vector<KeyT> first, std::vector<ValT> second)
39     {
40         table = new std::vector<std::pair<KeyT, ValT>>[DEFSIZE];
41         capacity1 = DEFSIZE;
42         M = 0;
43         for (unsigned int i = 0; i < first.size (); i++)
44         {
45             int check = 0;
46             std::hash<KeyT> hash_func;
47             int check2 = hash_func (first[i]) & (capacity1-1);
48
49             int size_help = table[check2].size ();
50             for (int j = 0; j < size_help; ++j)
51             {
52                 KeyT helper = table[check2][j].first;
53                 if (helper == first[i])
54                 {
55                     if (table[check2][j].second != second[i])
56                     {
57                         table[check2][j].second = second[i];
58                     }
59                 }
60             }
61         }
62     }
63 }
```

```

60         check++;
61     }
62
63 }
64 if (check == 0)
65 {
66     int index = hash_func (first[i] & (capacity1-1));
67
68     table[index].push_back ({first[i], second[i]});
69     M++;
70 }
71 }
72 if (M > A12)
73 {
74     make_new_hash ();
75 }
76 }
77 void help_when_else(std::vector<KeyT> first, std::vector<ValT> second)
78 {
79     capacity1 = new_size (first.size ());
80     table = new std::vector<std::pair<KeyT, ValT>>[capacity1];
81     M = 0;
82     for (int i = 0; i < DEFSIZE; i++)
83     {
84         for (unsigned int i = 0; i < first.size (); i++)
85         {
86             int check = 0;
87             std::hash<KeyT> hash_func;
88             int size_help = table[hash_func (first[i] & (capacity1-1))].size ();
89             for (int j = 0; j < size_help; ++j)
90             {
91                 KeyT helper = table[hash_func (first[i] & (capacity1-1))][j].first;
92                 if (helper == first[i])
93                 {
94                     if (table[hash_func (first[i] & (capacity1-1))][j].second
95                         != second[i])
96                     {
97                         table[hash_func (first[i] &
98                             (capacity1-1))][j].second = second[i];
99                     }
100                     check++;
101                 }
102             }
103             if (check == 0)
104             {
105                 table[hash_func (first[i] & (capacity1-1))].push_back ({first[i],
106                                         second[i]});
107                 M++;
108             }
109         }
110     }
111 }
112 int find_first_size (int size)
113 {
114     int x = 2;
115     while (x < size || x <= capacity1)
116     {
117         x = x * 2;
118     }
119     return x;
120 }
121 HashMap (std::vector<KeyT> first, std::vector<ValT> second)
122 {
123     if (first.size () != second.size ())
124     {
125         throw std::out_of_range ("Error size");
126     }
127     if (first.size () <= DEFSIZE)

```

```

128     {
129         help_when_if (first,second);
130     }
131     else
132     {
133         help_when_else( first,second);
134     }
135 }
136 int new_size (int vec_size)
137 {
138     vec_size++;
139     int x = 2;
140     if (get_load_factor () > UPPER)
141     {
142         return capacity1 * 2;
143     }
144     if (get_load_factor () < LOWER)
145     {
146         if (capacity1 == 1)
147         { return 1; }
148         return capacity1 / 2;
149     }
150 }
151 return x;
152 }
153 virtual ~HashMap ()/// check if it must
154 /// be array or it can be vector
155 {
156     if(table!=NULL){delete[] table;}
157 }
158
159
160 int size () const
161 {
162     return M;
163 }
164
165 int capacity () const
166 {
167     return capacity1;
168 }
169
170 bool empty () const
171 {
172     return M == 0;
173 }
174
175 bool contains_key (const KeyT& key2) const
176 {
177     std::hash<KeyT> hash_func;
178     int check = hash_func (key2) & (capacity1-1);
179
180     for (unsigned int j = 0; j < table[check].size (); ++j)
181     {
182         if (table[check][j].first == key2)
183         {
184             return true;
185         }
186     }
187 }
188
189 return false;
190 }
191 void make_new_hash ()
192 {
193     int new_size1 = new_size (M);
194     std::vector<std::pair<KeyT, ValT >> *new_table =
195         new std::vector<std::pair<KeyT, ValT>>[new_size1];

```



```

196     for (int i = 0; i < capacity1; ++i)
197     {
198         for (unsigned int j = 0; j < table[i].size (); ++j)
199         {
200             if (table[i].size () != 0)
201             {
202                 KeyT thekey = table[i][j].first;
203                 std::hash<KeyT> hash_func;
204                 int check = hash_func (thekey) & (new_size1-1);
205
206                 new_table[check].push_back
207                     ({table[i][j].first, table[i][j].second});
208             }
209         }
210     }
211
212     delete[] table;
213     this->capacity1 = new_size1;
214     this->table = new_table;
215 }
216
217
218 ValT &at (const KeyT &key)
219 {
220     std::hash<KeyT> hash_func;
221     int check = hash_func (key) & (capacity1-1);
222
223
224
225     for (std::pair<KeyT, ValT>& elem : table[check])
226     {
227         if (elem.first == key)
228         {
229             return elem.second;
230         }
231     }
232
233     throw std::out_of_range ("Error size");
234
235 }
236 ValT at (const KeyT& key) const
237 {
238     std::hash<KeyT> hash_func;
239     int check = hash_func (key) & (capacity1-1);
240
241     int vecsize = table[check].size ();
242     for (int i = 0; i < vecsize; i++)
243     {
244         if (table[check][i].first == key)
245         {
246             return table[check][i].second;
247         }
248     }
249
250     throw std::out_of_range ("Error size");
251
252 }
253 bool insert (const KeyT& key, const ValT& value)
254 {
255     if (!contains_key(key))
256     {
257         std::hash<KeyT> hash_func;
258         int check2 = hash_func (key) & (capacity1-1);
259
260         std::pair<KeyT, ValT> pair;
261         pair.first=key;
262         pair.second=value;
263         if(value==ValT())

```

```

264         {table[check2].push_back (std::make_pair(key,ValT()));}
265     else{table[check2].push_back (std::make_pair(key,value));}
266
267     M++;
268     if (get_load_factor () > UPPER)
269     {
270         make_new_hash ();
271     }
272     return true;
273 }
274 else
275 {
276
277     return false;
278 }
279 }
280
281 virtual bool erase (KeyT key)
282 {
283     std::hash<KeyT> hash_func;
284     int check = hash_func (key) & (capacity1-1);
285     typename std::vector<std::pair<KeyT, ValT>>::
286     iterator del = this->table[check].end ();
287     int check_if_find = 0;
288     for (auto i = table[check].begin (); i != table[check].end (); i++)
289     {
290         if (key == i->first)
291         {
292             del = i;
293             check_if_find++;
294         }
295     }
296     if (check_if_find != 0)
297     {
298         table[check].erase (del);
299         M--;
300
301         double fac = get_load_factor ();
302         if (fac < LOWER)
303         {
304             make_new_hash ();
305         }
306         return true;
307     }
308     return false;
309 }
310
311 double get_load_factor ()const
312 {
313     return (double) M / capacity1;
314 }
315 void clear ()
316 {
317     for (int i = 0; i < capacity1; ++i)
318     {
319         M = M - table[i].size ();
320         table[i].clear ();
321     }
322 }
323
324 int bucket_size (const KeyT& key)const
325 {
326     std::hash<KeyT> hash_func;
327     int check = hash_func (key) & (capacity1-1);
328     return table[check].size ();
329 }
330
331 int bucket_index (const KeyT &key)const

```

```

332 {
333     std::hash<KeyT> hash_func;
334     return hash_func (key) & (capacity1-1);
335 }
336
337 ValT operator[] (const KeyT &key) const
338 {
339     std::hash<KeyT> hash_func;
340     int check_if_we_found = 0;
341     int check = hash_func (key) & (capacity1-1);
342
343     for (int i = 0; i < table[check].size (); ++i)
344     {
345         if (table[check][i].first == key)
346         {
347             check_if_we_found++;
348             return table[check][i].second;
349         }
350     }
351     return ValT();
352 }
353
354 ValT &operator[] (const KeyT &key)
355 {
356     if(!contains_key(key))
357     {
358         insert (key, ValT());
359     }
360
361     ValT &ret = this->at (key);
362     return ret;
363 }
364
365
366
367 HashMap<KeyT, ValT> &operator= (const HashMap a)
368 {
369     if(a==*this)
370     {return *this;}
371     this->capacity1 = a.capacity1;
372     this->M = a.M;
373     delete[] this->table;
374     table = new std::vector<std::pair<KeyT, ValT>>[capacity1];
375     for (int i = 0; i < a.capacity1; ++i)
376     {
377         table[i] = a.table[i];
378     }
379     return *this;
380 }
381
382 // bool equalhelp(const HashMap a)
383 // {}
384 bool operator== (const HashMap a) const
385 {
386     std::hash<KeyT> hash_func;
387     if (this->M != a.M)
388     {return false;}
389     for( int i = 0; i<capacity1; i++)
390     {
391         for (unsigned int j = 0; j < table[i].size(); ++j)
392         {
393             int check_if_found=0;
394             int check = hash_func (table[i][j].first) & (a.capacity1-1);
395             for (unsigned int k = 0; k < a.table[check].size(); ++k)
396             {
397                 if(table[i][j].first==a.table[check][k].first&&
398                    table[i][j].second==a.table[check][k].second)
399                 {check_if_found++;}

```

```

400     }
401     if(check_if_found==0)
402     {return false;}
403 }
404 }
405 for( int i = 0;i<a.capacity1;i++)
406 {
407     for (unsigned int j = 0; j < a.table[i].size(); ++j)
408     {
409         int check_if_found=0;
410         int check = hash_func (a.table[i][j].first) & (a.capacity1-1);
411
412         for (unsigned int k = 0; k < table[check].size(); ++k)
413         {
414             if(a.table[i][j].first==table[check][k].first&&a.
415                 table[i][j].second==table[check][k].second)
416             {
417                 check_if_found++;
418             }
419         }
420         if(check_if_found==0)
421         {return false;}
422     }
423 }
424 return true;
425 }
426 bool operator!= (const HashMap a) const
427 {
428     if (this->capacity1 != a.capacity1)
429     {
430         return true;
431     }
432     for (int i = 0; i < capacity1; ++i)
433     {
434         if (table[i].size () != a.table[i].size ())
435         { return true; }
436         if (table[i] != a.table[i])
437         { return true; }
438     }
439     return false;
440 }
441 }
442 }
443
444
445 class ConstIterator
446 {
447
448     friend class Hashmap;
449     int index_for_vec1;
450     int index_in_vec1;
451     const HashMap &a;
452
453 public:
454
455     typedef std::pair<KeyT, ValT> value_type;
456     typedef std::pair<KeyT, ValT> &reference;
457     typedef std::pair<KeyT, ValT> *pointer;
458     typedef std::ptrdiff_t difference_type;
459     typedef std::forward_iterator_tag iterator_category;
460
461     // Constructor
462     ConstIterator (int index_for_vec, int index_in_vec, const HashMap &Ha)
463         : index_for_vec1 (index_for_vec), index_in_vec1 (index_in_vec),a (Ha)
464     {
465     }
466
467     ConstIterator &operator++ ()

```

```

468 {
469     if (index_for_vec1 == a.capacity1)
470     {
471         int check2 = -1;
472         for (int i = 0; i < a.size (); ++i)
473         {
474             if (a.size () != 0 && check2 == -1)
475             {
476                 check2++;
477                 index_for_vec1 = i;
478             }
479         }
480     }
481 }
482 if (a.capacity1 == index_for_vec1
483     && index_in_vec1 == a.table[index_for_vec1].size () - 1)
484 {
485     std::cout << "error";
486 }
487 if (index_in_vec1 < a.table[index_for_vec1].size () - 1)
488 {
489     index_in_vec1++;
490     return *this;
491 }
492 if (index_in_vec1 == a.table[index_for_vec1].size ()
493     - 1)/// check if we in the end of the vector
494 {
495     int check = -1;
496     /// check if we find new vector
497     for (int i = index_for_vec1 + 1; i < a.capacity1; ++i)
498     {
499         if (check == -1 && a.table[i].size () != 0)
500         {
501             check = i;
502             index_in_vec1 = 0;
503             index_for_vec1 = i;
504         }
505     }
506     if (check == -1)
507     {
508         this->index_for_vec1= a.capacity1 + 1;
509         this->index_in_vec1 = 0;
510         return *this;
511     }
512 }
513 return *this;
514 }
515
516 ConstIterator operator++ (int)
517 {
518     if (index_for_vec1 == a.capacity1)
519     {
520         int check2 = -1;
521         for (int i = 0; i < a.size (); ++i)
522         {
523             if (a.size () != 0 && check2 == -1)
524             {
525                 check2++;
526                 index_for_vec1 = i;
527             }
528         }
529     }
530 }
531 if (a.capacity1 == index_for_vec1
532     && index_in_vec1 == a.table[index_for_vec1].size () - 1)
533 {
534     std::cout << "error";
535 }

```

```

536     if (index_in_vec1 < a.table[index_for_vec1].size () - 1)
537     {
538         index_in_vec1++;
539         return *this;
540     }
541     if (index_in_vec1 == a.table[index_for_vec1].size ()
542         - 1)/// check if we in the end of the vector
543     {
544         int check = -1;
545         /// check if we find new vector
546         for (int i = index_for_vec1 + 1; i < a.capacity1; ++i)
547         {
548             if (check == -1 && a.table[i].size () != 0)
549             {
550                 check = i;
551                 index_in_vec1 = 0;
552                 index_for_vec1 = i;
553             }
554         }
555         if (check == -1)
556         {
557             this->index_for_vec1= a.capacity1 + 1;
558             this->index_in_vec1 = 0;
559             return *this;
560         }
561     }
562     return *this;
563 }
564
565 bool operator== (const ConstIterator &rhs) const
566 {
567     return index_for_vec1 == rhs.index_for_vec1
568         && index_in_vec1 == rhs.index_in_vec1;
569 }
570
571 bool operator!= (const ConstIterator &rhs) const
572 {
573     return index_for_vec1 != rhs.index_for_vec1
574         || index_in_vec1 != rhs.index_in_vec1 || rhs.a!=this->a;
575 }
576
577 reference operator* ()
578 {
579     return a.table[index_for_vec1][index_in_vec1];
580 }
581
582 pointer operator-> ()
583 { return &(operator* ()); }
584
585 };
586
587 // using iterator = Iterator;
588 using const_iterator = ConstIterator;
589
590 // Iterator begin ()
591 // {
592 //     if (M == 0)
593 //     {
594 //         return Iterator (capacity1 + 1, 0, *this);
595 //     }
596 //     for (unsigned int i = 0; i < table->size (); ++i)
597 //     {
598 //         if (table[i].size () != 0)
599 //         {
600 //             return Iterator (i, 0, *this);
601 //         }
602 //     }
603 //     return Iterator (capacity1 + 1, 0, *this);

```

```

604 //
605 //
606 // }
607
608 const_iterator begin () const
609 {
610
611     if (M == 0)
612     {
613         return const_iterator (capacity1 + 1, 0, *this);
614     }
615     for ( int i = 0; i < this->capacity1; ++i)
616     {
617         if (table[i].size () != 0)
618         {
619             return const_iterator (i, 0, *this);
620         }
621     }
622     return const_iterator (capacity1 + 1, 0, *this);
623
624 }
625
626 const_iterator cbegin () const
627 {
628     if (M == 0)
629     {
630         return const_iterator (capacity1 + 1, 0, *this);
631     }
632     for ( long unsigned int i = 0; i < table->capacity(); ++i)
633     {
634         if (table[i].size () != 0)
635         {
636             return const_iterator (i, 0, *this);
637         }
638     }
639     return const_iterator (capacity1 + 1, 0, *this);
640 }
641
642 // Iterator end ()
643 // { return Iterator (capacity1 + 1, 0, *this); }
644 const_iterator end () const
645 { return const_iterator (this->capacity1 + 1, 0, *this); }
646 const_iterator cend () const
647 { return const_iterator (this->capacity1 + 1, 0, *this); }
648
649
650
651 };
652
653
654 #endif // _HASHMAP_HPP_

```