

```
1  /*****
2  * AUTHOR          : Nick Reardon
3  * Assignment #7   : Hashing Algorithms
4  * CLASS           : CS1D
5  * SECTION         : MW - 2:30p
6  * DUE DATE        : 03 / 04 / 20
7  *****/
8  #ifndef _DOUBLEHASH_H_
9  #define _DOUBLEHASH_H_
10 #include <string>
11 #include <iostream>
12 #include <iomanip>
13
14 namespace errorType
15 {
16     enum errors
17     {
18         DEFAULT,
19         EMPTY,
20         FULL
21     };
22
23     std::string errorString[]
24     {
25         "Error - An error occurred",
26         "Error - Map is empty",
27         "Error - Map is full"
28     };
29 }
30
31 enum indexLabel
32 {
33     EMPTY,
34     FULL,
35     AVAILABLE
36 };
37
38 template <class T_key, class T_value>
39 struct T_struct
40 {
41     T_key key;
42     T_value value;
43
44     enum indexLabel label = EMPTY;
45
46     T_struct<T_key, T_value>()
47     {
48         key = -1;
49         label = EMPTY;
50     }
51 }
52
```

```
53     T_struct<T_key, T_value>(const T_key& key, const T_value& value)
54     {
55         this->key = key;
56         this->value = value;
57
58         label = EMPTY;
59     }
60
61     T_struct<T_key, T_value>(const T_struct<T_key, T_value>& rhs)
62     {
63         this->key = rhs.key;
64         this->value = rhs.value;
65
66         this->label = rhs.label;
67     }
68
69     T_struct<T_key, T_value>& operator=(const T_struct<T_key, T_value>& rhs)
70     {
71         this->key = rhs.key;
72         this->value = rhs.value;
73
74         this->label = rhs.label;
75
76         return *this;
77     }
78 };
79
80 template <class T_key, class T_value>
81 T_struct<T_key, T_value> make_struct(T_key newKey, T_value newValue)
82 {
83     return T_struct<T_key, T_value>(newKey, newValue);
84 }
85
86
87
88 template <class T_key, class T_value>
89 class DoubleHashMap
90 {
91 private:
92
93     T_struct<T_key, T_value>* map;
94
95     int currentSize;
96     int capacity;
97
98     // ostream member? Assign it in constructor or method???
99     // set to NULL?
100
101 protected:
102
103     int DoubleHash(const int givenKey, const int collisionCount) const
104     {
```

```
105     int hashCode();
106
107     int j = collisionCount;
108     int k = givenKey;
109     int N = capacity;
110
111     /*
112     int hk;
113     int hk2;
114
115     hk = (k % N);
116     hk2 = (k % 13);
117     hk2 = 13 - hk2;
118     hk2 = j * hk2;
119
120     hashCode = hk + hk2;
121
122     hashCode = hashCode % N;
123     */
124
125     hashCode = ((( k % N ) + (j * (13 - (k % 13))) ) % N);
126
127     return hashCode;
128 }
129
130 int DoubleHash(const T_struct<T_key, T_value>& toInsert, const int collisionCount) const
131 {
132     DoubleHash(toInsert.key, collisionCount);
133 }
134
135 public:
136
137 DoubleHashMap(const int newCapacity)
138 {
139     map = new T_struct<T_key, T_value>[newCapacity];
140
141     currentSize = 0;
142
143     capacity = newCapacity;
144 }
145
146 ~DoubleHashMap()
147 {
148     delete[] map;
149 }
150
151 void insert(const T_struct<T_key, T_value>& toInsert)
152 {
153     if (full())
154     {
155         throw(errorType::FULL, errorType::errorString[FULL], 5);
```

```
156     }
157
158     int hashKey;
159     std::string output = std::to_string(toInsert.key);
160     int collisionCount = 0;
161     bool stopHash = false;
162     bool success = false;
163
164     while (stopHash == false)
165     {
166         hashKey = DoubleHash(toInsert.key, collisionCount);
167
168         if (map[hashKey].label == EMPTY ||
169             map[hashKey].label == AVAILABLE)
170         {
171             stopHash = true;
172             success = true;
173
174         }
175         else
176         {
177             if (map[hashKey].key == toInsert.key)
178             {
179                 stopHash = true;
180                 success = true;
181             }
182
183             collisionCount++;
184         }
185
186         output += "->" + std::to_string(hashKey);
187     }
188
189
190
191     if (success)
192     {
193         map[hashKey] = toInsert;
194         map[hashKey].label = FULL;
195
196         currentSize++;
197
198         std::cout << "Inserting: " << '(' << toInsert.key << ", " <<
199             toInsert.value << ')'
200             << '\n' << "Hashed Key: " << output << '\n' << '\n';
201     }
202
203     /**
204     * @fn void DoubleHashMap::remove(const T_key key)
205     * @brief Removes the given key
206     *
```

```
207     * @exception   errorType::FULL,    Thrown when a full, error condition occurs.
208     *
209     * @param   key The key to remove.
210     */
211 void remove(const T_key key)
212 {
213     if (empty())
214     {
215         throw(errorType::FULL, errorType::errorString[FULL], 5);
216     }
217
218     int hashKey;
219     std::string output = std::to_string(key);
220     int collisionCount = 0;
221     bool stopHash = false;
222     bool success = false;
223     while (stopHash == false)
224     {
225         hashKey = DoubleHash(key, collisionCount);
226
227         if (map[hashKey].label == FULL ||
228             map[hashKey].label == AVAILABLE)
229         {
230             if (map[hashKey].key == key)
231             {
232                 stopHash = true;
233                 success = true;
234             }
235             else
236             {
237                 collisionCount++;
238             }
239         }
240         else
241         {
242             stopHash = true;
243         }
244         output += "->" + std::to_string(hashKey);
245     }
246
247     if (success)
248     {
249         map[hashKey].key = -1;
250         map[hashKey].value = "";
251         map[hashKey].label = AVAILABLE;
252
253         currentSize--;
254
255         std::cout << "Removing key: " << key << '\n'
256                 << "Hashed Key: " << output << '\n' << '\n';
257     }
```

```
258
259
260     }
261
262
263     }
264
265     bool full()
266     {
267         return currentSize == capacity;
268     }
269
270     bool empty()
271     {
272         return currentSize == 0;
273     }
274
275     int size()
276     {
277         return currentSize;
278     }
279
280     void printAll(std::ostream& output)
281     {
282         if (empty())
283         {
284             throw(errorType::EMPTY, errorType::errorString[EMPTY], 5);
285         }
286
287         output << "  Index | LABEL | Key | Value" << '\n'
288             << "  _____|_____|_____|_____ "
289             << '\n';
290
291         for (int i = 0; i < capacity; i++)
292         {
293             output << std::right
294                 << " [" << std::setw(5) << i << "] | ";
295             switch (map[i].label)
296             {
297                 case EMPTY:
298                     output << "EMPTY | ";
299                     break;
300
301                 case FULL:
302                     output << "FULL | ";
303                     break;
304
305                 case AVAILABLE:
306                     output << "AVAIL | ";
307                     break;
308
309             }
```

```

310         output << ' ' << std::setw(4) << map[i].key << " |";
311
312         output << std::left
313             << ' ' << map[i].value
314             << '\n';
315     }
316     output << "\n\n";
317 }
318 };
319
320 //|||||||
321
322
323 template <class T_key, class T_value>
324 class QuadraticHashMap
325 {
326 private:
327
328     T_struct<T_key, T_value>* map;
329
330     int currentSize;
331     int capacity;
332
333     // ostream member? Assign it in constructor or method???
334     // set to NULL?
335
336 protected:
337
338     int QuadraticHash(const int givenKey, const int collisionCount) const
339     {
340         int hashKey;
341
342         int j = collisionCount;
343         int k = givenKey;
344         int N = capacity;
345
346         /*
347         int hk;
348         int hk2;
349
350         hk = (k % N);
351         hk2 = (k % 13);
352         hk2 = 13 - hk2;
353         hk2 = j * hk2;
354
355         hashKey = hk + hk2;
356
357         hashKey = hashKey % N;
358         */
359
360
361

```

```
362     if (j > 0)
363     {
364         hashKey = (((k % N) + (j * j)) % N);
365     }
366     else
367     {
368         hashKey = (k % N);
369     }
370
371     return hashKey;
372 }
373
374 int QuadraticHash(const T_struct<T_key, T_value>& toInsert, const int collisionCount) const
375 {
376     QuadraticHash(toInsert.key, collisionCount);
377 }
378
379 public:
380
381 QuadraticHashMap(const int newCapacity)
382 {
383     map = new T_struct<T_key, T_value>[newCapacity];
384
385     currentSize = 0;
386
387     capacity = newCapacity;
388 }
389
390 ~QuadraticHashMap()
391 {
392     delete[] map;
393 }
394
395 void insert(const T_struct<T_key, T_value>& toInsert)
396 {
397     if (full())
398     {
399         throw(errorType::FULL, errorType::errorString[FULL], 5);
400     }
401
402     int hashKey;
403     std::string output = std::to_string(toInsert.key);
404     int collisionCount = 0;
405     bool stopHash = false;
406     bool success = false;
407
408     while (stopHash == false)
409     {
410         hashKey = QuadraticHash(toInsert.key, collisionCount);
411
412         if (map[hashKey].label == EMPTY ||
```



```
413         map[hashKey].label == AVAILABLE)
414     {
415         stopHash = true;
416         success = true;
417     }
418     else
419     {
420         if (map[hashKey].key == toInsert.key)
421         {
422             stopHash = true;
423             success = true;
424         }
425         collisionCount++;
426     }
427     output += "->" + std::to_string(hashKey);
428 }
429
430 if (success)
431 {
432     map[hashKey] = toInsert;
433     map[hashKey].label = FULL;
434     currentSize++;
435     std::cout << "Inserting: " << '(' << toInsert.key << ", " <<
436         toInsert.value << ')'
437         << '\n' << "Hashed Key: " << output << '\n' << '\n';
438 }
439 }
440
441 void remove(const T_key key)
442 {
443     if (empty())
444     {
445         throw(errorType::FULL, errorType::errorString[FULL], 5);
446     }
447     int hashKey;
448     std::string output = std::to_string(key);
449     int collisionCount = 0;
450     bool stopHash = false;
451     bool success = false;
452     while (stopHash == false)
453     {
454         hashKey = QuadraticHash(key, collisionCount);
455         if (map[hashKey].label == FULL ||
```

```
464         map[hashKey].label == AVAILABLE)
465     {
466         if (map[hashKey].key == key)
467         {
468             stopHash = true;
469             success = true;
470         }
471         else
472         {
473             collisionCount++;
474         }
475     }
476 }
477 else
478 {
479     stopHash = true;
480 }
481 output += "->" + std::to_string(hashKey);
482 }
483
484 if (success)
485 {
486     map[hashKey].key = -1;
487     map[hashKey].value = "";
488     map[hashKey].label = AVAILABLE;
489
490     currentSize--;
491
492     std::cout << "Removing key: " << key << '\n'
493               << "Hashed Key: " << output << '\n' << '\n';
494
495 }
496 }
497
498 }
499 }
500
501 bool full()
502 {
503     return currentSize == capacity;
504 }
505
506 bool empty()
507 {
508     return currentSize == 0;
509 }
510
511 int size()
512 {
513     return currentSize;
514 }
515
```

```

516 void printAll(std::ostream& output)
517 {
518     if (empty())
519     {
520         throw(errorType::EMPTY, errorType::errorString[EMPTY], 5);
521     }
522
523     output << "  Index  | LABEL | Key   | Value" << '\n'
524     << "  _____|_____|_____|_____"
525     << '\n';
526
527     for (int i = 0; i < capacity; i++)
528     {
529         output << std::right
530             << " [" << std::setw(5) << i << "] | ";
531         switch (map[i].label)
532         {
533             case EMPTY:
534                 output << "EMPTY |";
535                 break;
536
537             case FULL:
538                 output << "FULL |";
539                 break;
540
541             case AVAILABLE:
542                 output << "AVAIL |";
543                 break;
544
545             }
546         output << ' ' << std::setw(4) << map[i].key << " |";
547
548         output << std::left
549             << ' ' << map[i].value
550             << '\n';
551     }
552     output << "\n\n";
553 }
554 };
555
556
557
558
559
560
561
562 #endif // !_DOUBLEHASH_H_
563
564
565
566
567

```