

cs280s21-b.sg

[Dashboard](#) / [My courses](#) / [cs280s21-b.sg](#) / [General](#) / [Assignment 5: Hashing](#)

- [Description](#)
- [Submission](#)
- [Edit](#)
- Submission view

Grade

Reviewed on Saturday, April 10, 2021, 9:35 PM by Automatic grade
grade: 100.00 / 100.00

Assessment report[\[-\]](#)
[\[+\]](#)**Summary of tests**

Submitted on Saturday, April 10, 2021, 9:35 PM ([Download](#))
ChHashTable.h

```

1  /*****
2  /*!
3  \file:      ChHashTable.h
4  \author:    Goh Wei Zhe, weizhe.goh, 44000119
5  \par email: weizhe.goh@digipen.edu
6  \date:      April 10, 2021
7  \brief      This file contains the declarations needed to construct a templated
8               Hash Table
9
10 Copyright (C) 2021 DigiPen Institute of Technology.
11 Reproduction or disclosure of this file or its contents without the
12 prior written consent of DigiPen Institute of Technology is prohibited.
13 */
14 /*****
15
16 //-----
17 #ifndef CHHASHTABLEH
18 #define CHHASHTABLEH
19 //-----
20
21 #include <cstring>
22 #include <cmath>
23 #include "ObjectAllocator.h"
24 #include "support.h"
25
26 // client-provided hash function: takes a key and table size,
27 // returns an index in the table.
28 typedef unsigned (*HASHFUNC)(const char *, unsigned);
29
30 // Max length of our "string" keys
31 const unsigned MAX_KEYLEN = 10;
32
33 class HashTableException
34 {
35 private:
36     int error_code_;
37     std::string message_;
38
39 public:
40     HashTableException(int ErrCode, const std::string& Message) :
41         error_code_(ErrCode), message_(Message) {};
42
43     virtual ~HashTableException() {
44     }
45
46     virtual int code() const {
47         return error_code_;
48     }
49
50     virtual const char *what() const {
51         return message_.c_str();
52     }
53     enum HASHTABLE_EXCEPTION {E_ITEM_NOT_FOUND, E_DUPLICATE, E_NO_MEMORY};
54 };
55
56
57 // HashTable statistical info
58 struct HTStats
59 {
60     HTStats(void) : Count_(0), TableSize_(0), Probes_(0), Expansions_(0),
61                   HashFunc_(0) {};
62     unsigned Count_; // Number of elements in the table
63     unsigned TableSize_; // Size of the table (total slots)
64     unsigned Probes_; // Number of probes performed
65     unsigned Expansions_; // Number of times the table grew
66     HASHFUNC HashFunc_; // Pointer to primary hash function
67     ObjectAllocator *Allocator_; // The allocator in use (may be 0)
68 };
69
70 template <typename T>
71 class ChHashTable
72 {
73 public:
74
75     typedef void (*FREEPROC)(T); // client-provided free proc (we own the data)
76
77     struct HTConfig
78     {
79         HTConfig(unsigned InitialTableSize,
80                 HASHFUNC HashFunc,
81                 double MaxLoadFactor = 3.0,
82                 double GrowthFactor = 2.0,
83                 FREEPROC FreeProc = 0) :
84
85             // The number of slots in the table initially.
86             InitialTableSize_(InitialTableSize),
87             // The hash function used in all cases.
88             HashFunc_(HashFunc),
89             // The maximum "fullness" of the table.
90             MaxLoadFactor_(MaxLoadFactor),
91             // The factor by which the table grows.
92             GrowthFactor_(GrowthFactor),
93             // The method provided by the client that may need to be called when
94             // data in the table is removed.
95             FreeProc_(FreeProc) {}
96
97     unsigned InitialTableSize_;
98     HASHFUNC HashFunc_;
99     double MaxLoadFactor_;
100    double GrowthFactor_;
101    FREEPROC FreeProc_;
102 };
103
104 // Nodes that will hold the key/data pairs
105 struct ChHTNode
106 {
107     char Key[MAX_KEYLEN]; // Key is a string

```

```
108     T Data;                // Client data
109     ChHTNode *Next;
110     ChHTNode(const T& data) : Data(data) {}; // constructor
111 };
112
113 // Each list has a special head pointer
114 struct ChHTHeadNode
115 {
116     ChHTNode *Nodes;
117     ChHTHeadNode() : Nodes(0), Count(0) {};
118     int Count; // For testing
119 };
120
121 // ObjectAllocator: the usual.
122 // Config: the configuration for the hash table.
123 ChHashTable(const HTConfig& Config, ObjectAllocator* allocator = 0);
124 ~ChHashTable();
125
126 // Insert a key/data pair into table. Throws an exception if the
127 // insertion is unsuccessful.(E_DUPLICATE, E_NO_MEMORY)
128 void insert(const char *Key, const T& Data);
129
130 // Delete an item by key. Throws an exception if the key doesn't exist.
131 // (E_ITEM_NOT_FOUND)
132 void remove(const char *Key);
133
134 // Find and return data by key. throws exception if key doesn't exist.
135 // (E_ITEM_NOT_FOUND)
136 const T& find(const char *Key) const;
137
138 // Removes all items from the table (Doesn't deallocate table)
139 void clear();
140
141 // Allow the client to peer into the data. Returns a struct that contains
142 // information on the status of the table for debugging and testing.
143 // The struct is defined in the header file.
144 HTStats GetStats() const;
145 const ChHTHeadNode *GetTable() const;
146
147 private:
148
149     // Private fields and methods...
150
151     HTConfig _config;
152     mutable HTStats _stats{};
153     ObjectAllocator* _oa;
154     ChHTHeadNode* _table;
155
156     ChHTNode* CreateNode(const char* Key, const T& Data);
157
158     ChHTNode* SearchNode(const unsigned& i, const char* Key,
159     ChHTNode*& prev_node) const;
160
161     void DeleteNode(ChHTNode* node);
162 };
163
164 #include "ChHashTable.cpp"
165
166 #endif
```

ChHashTable.cpp

```

1  /*****
2  /*!
3  \file:      ChHashTable.cpp
4  \author:    Goh Wei Zhe, weizhe.goh, 44000119
5  \par email: weizhe.goh@digipen.edu
6  \date:      April 10, 2021
7  \brief      This file contains the definitions needed to construct a templated
8               Hash Table
9
10 Copyright (C) 2021 DigiPen Institute of Technology.
11 Reproduction or disclosure of this file or its contents without the
12 prior written consent of DigiPen Institute of Technology is prohibited.
13 */
14 /*****
15 #include "ChHashTable.h"
16
17 /*****
18 /*!
19 \fn      template<typename T>
20           ChHashTable<T>::ChHashTable(const HTConfig& Config,
21           ObjectAllocator* allocator)
22
23 \brief    Constructor for ChHashTable
24
25 \param    Config - Configuration struct for this object
26
27 \param    allocator - Object allocator to allocate memory, if provided
28
29 */
30 /*****
31 template<typename T>
32 ChHashTable<T>::ChHashTable(const HTConfig& Config, ObjectAllocator* allocator)
33 : _config{Config}, _oa{allocator}
34 {
35     //Initilise stats
36     _stats.TableSize_ = _config.InitialTableSize_;
37     _stats.HashFunc_ = _config.HashFunc_;
38
39     //Initilise table array
40     _table = new ChHTHeadNode[_stats.TableSize_];
41 }
42
43 /*****
44 /*!
45 \fn      template<typename T>
46           ChHashTable<T>::~ChHashTable()
47
48 \brief    Destructor for ChHashTable
49 */
50 /*****
51 template<typename T>
52 ChHashTable<T>::~ChHashTable()
53 {
54     clear();
55     delete[] _table;
56 }
57
58 /*****
59 /*!
60 \fn      template<typename T>
61           void ChHashTable<T>::insert(const char* Key, const T& Data)
62
63 \brief    Insert a key / data pair into the hash table
64
65 \param    Key - Key to determine place in the hash table
66
67 \param    Data - value to be stored in the hash table
68
69 */
70 /*****
71 template<typename T>
72 void ChHashTable<T>::insert(const char* Key, const T& Data)
73 {
74     //Calculate load factor
75     double loadFactor = (_stats.Count+1)/static_cast<double>(_stats.TableSize_);
76
77     //if loadFactor > max load factor, expand table
78     if(loadFactor > _config.MaxLoadFactor_)
79     {
80         unsigned oldTableSize = _stats.TableSize_;
81         double factor = std::ceil(_stats.TableSize_ * _config.GrowthFactor_);
82
83         _stats.TableSize_ = GetClosestPrime(static_cast<unsigned>(factor));
84         ChHTHeadNode* oldTable = _table;
85
86         //reinsert items from old to new table
87         _table = new ChHTHeadNode[_stats.TableSize_];
88
89         for(unsigned i = 0; i < oldTableSize; ++i)
90         {
91             ChHTHeadNode headNode = oldTable[i];
92             ChHTNode* node = headNode.Nodes;
93
94             while(node)
95             {
96                 ChHTNode* prev_node;
97
98                 ChHTNode* temp = node;
99                 node = node->Next;
100
101                 //Get index in table
102                 unsigned i = _stats.HashFunc_(temp->Key, _stats.TableSize_);
103
104                 //check if node exist in new table
105                 ChHTNode* NodeFound = SearchNode(i, Key, prev_node);
106
107                 //If node found is not first node, have to check for duplicate

```

```

108 //If there is duplicate, will throw exception
109 if(NodeFound != nullptr)
110     throw HashTableException(HashTableException::E_DUPLICATE,
111         "Key inserted is already in Hash Table");
112
113     temp->Next = _table[i].Nodes;
114     _table[i].Nodes = temp;
115     ++_table[i].Count;
116 }
117
118     oldTable[i].Nodes = nullptr;
119 }
120
121     delete[] oldTable;
122     ++_stats.Expansions_;
123 }
124
125 //Get index in table
126 unsigned i = _stats.HashFunc_(Key, _stats.TableSize_);
127
128 ChHTNode* prev_node;
129
130 ChHTNode* node = SearchNode(i, Key, prev_node);
131
132 if(node != nullptr)
133     throw HashTableException(HashTableException::E_DUPLICATE,
134         "Key inserted is already in the Hash Table");
135
136 //Create and insert node
137 node = CreateNode(Key, Data);
138 node->Next = _table[i].Nodes;
139 _table[i].Nodes = node;
140
141 ++_table[i].Count;
142 ++_stats.Count_;
143 }
144
145 /*****
146  *!
147  \fn      template<typename T>
148           void ChHashTable<T>::remove(const char* Key)
149
150  \brief   Remove a node in Hash Table
151
152  \param   Key - ID of the data to be remove
153
154  */
155 /*****
156  template<typename T>
157  void ChHashTable<T>::remove(const char* Key)
158  {
159      unsigned i = _stats.HashFunc_(Key, _stats.TableSize_);
160
161      ChHTNode* prev_node;
162      ChHTNode* node = SearchNode(i, Key, prev_node);
163
164      if(node == nullptr)
165          throw HashTableException(HashTableException::E_ITEM_NOT_FOUND,
166              "Key does not exist in table");
167
168      //Remove node
169      ChHTNode* temp = node;
170
171      if(prev_node == node)
172          _table[i].Nodes = node->Next;
173      else
174          prev_node->Next = node->Next;
175
176      DeleteNode(temp);
177
178      --_table[i].Count;
179      --_stats.Count_;
180  }
181
182 /*****
183  *!
184  \fn      template<typename T>
185           const T& ChHashTable<T>::find(const char* Key) const
186
187  \brief   Find a node in Hash Table
188
189  \param   Key - ID of the node to be found
190
191  \return  Returns referenced to the node requested
192  */
193 /*****
194  template<typename T>
195  const T& ChHashTable<T>::find(const char* Key) const
196  {
197      //Get index from table
198      unsigned i = _stats.HashFunc_(Key, _stats.TableSize_);
199
200      //Search nodes
201      ChHTNode* prev_node;
202      ChHTNode* node = SearchNode(i, Key, prev_node);
203
204      if(node == nullptr)
205          throw HashTableException(HashTableException::E_ITEM_NOT_FOUND,
206              "Key does not exist in table");
207
208      return node->Data;
209  }
210
211 /*****
212  *!
213  \fn      template<typename T>
214           void ChHashTable<T>::clear()

```

```

215 \brief Delete all nodes in Hash Table, does not delete Hash Table itself
216
217 */
218 /*****
219 template<typename T>
220 void ChHashTable<T>::clear()
221 {
222     for(unsigned i = 0; i < _stats.TableSize_; ++i)
223     {
224         ChHTHeadNode headNode = _table[i];
225         ChHTNode* node = headNode.Nodes;
226
227         while(node)
228         {
229             ChHTNode* temp = node;
230             node = node->Next;
231
232             DeleteNode(temp);
233
234             --_table[i].Count;
235             --_stats.Count_;
236         }
237
238         _table[i].Nodes = nullptr;
239     }
240 }
241
242 /*****/
243 /*!
244 \fn     template<typename T>
245         HTStats ChHashTable<T>::GetStats() const
246
247 \brief  Get Statistics of the Hash Table
248
249 \return Returns Statistics of the Hash Table
250 */
251 /*****/
252 template<typename T>
253 HTStats ChHashTable<T>::GetStats() const
254 {
255     return _stats;
256 }
257
258 /*****/
259 /*!
260 \fn     template<typename T>
261         const typename ChHashTable<T>::ChHTHeadNode* ChHashTable<T>::GetTable()
262         const
263
264 \brief  Get reference to the Hash Table
265
266 \return Returns reference to the Hash Table
267 */
268 /*****/
269 template<typename T>
270 const typename ChHashTable<T>::ChHTHeadNode* ChHashTable<T>::GetTable() const
271 {
272     return _table;
273 }
274
275 /*****/
276 /*!
277 \fn     template<typename T>
278         typename ChHashTable<T>::ChHTNode* ChHashTable<T>::SearchNode
279         (const unsigned& i, const char* Key, ChHTNode*& prev_node) const
280
281 \brief  Search for specific node and increment stats
282
283 \param  i- index of the node to be searched
284
285 \param  Key - ID of the node to be searched
286
287 \param  prev_node - Previous node
288
289 \return Returns referenced to the node searched
290 */
291 /*****/
292 template<typename T>
293 typename ChHashTable<T>::ChHTNode* ChHashTable<T>::SearchNode
294 (const unsigned& i, const char* Key, ChHTNode*& prev_node) const
295 {
296     //Loop through linked list to find key
297
298     if(_table[i].Count == 0)
299     {
300         ++_stats.Probes_;
301         return nullptr;
302     }
303     else
304     {
305         ChHTNode* node = _table[i].Nodes;
306         prev_node = node;
307
308         while(node)
309         {
310             ++_stats.Probes_;
311
312             if(strcmp(node->Key, Key) == 0)
313                 return node;
314
315             prev_node = node;
316             node = node->Next;
317         }
318
319         ++_stats.Probes_;
320         return node;
321     }

```

```
322 }
323
324 /*****
325  *!
326  \fn      template<typename T>
327            typename ChHashTable<T>::ChHTNode* ChHashTable<T>::
328            CreateNode(const char* Key, const T& Data)
329
330  \brief   Create a Node with Key and Data
331
332  \param   Key - ID of the new node
333
334  \param   Data - Value of the new node
335
336  \return  Returns the new node
337  */
338 /*****
339  template<typename T>
340  typename ChHashTable<T>::ChHTNode* ChHashTable<T>::CreateNode(const char* Key,
341  const T& Data)
342  {
343      ChHTNode* newNode;
344
345      if(!_oa)
346          newNode = new (_oa->Allocate()) ChHTNode(Data);
347      else
348          newNode = new ChHTNode(Data);
349
350      std::strcpy(newNode->Key, Key);
351      return newNode;
352  }
353
354 /*****
355  *!
356  \fn      template<typename T>
357            void ChHashTable<T>::DeleteNode(ChHTNode* node)
358
359  \brief   Delete node in Hash Table
360
361  \param   node - Node to be deleted
362  */
363 /*****
364  template<typename T>
365  void ChHashTable<T>::DeleteNode(ChHTNode* node)
366  {
367      if(!_oa)
368          node->~ChHTNode();
369
370      delete node;
```

◀ Assignment 4: Graphs

Jump to...

⬆

Quiz: Leet Code ▶

[VPL](#)

You are logged in as [Wei Zhe GOH](#) ([Log out](#))
[cs280s21-b.sg](#)
[Data retention summary](#)
[Get the mobile app](#)