cs280s21-b.sg

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

```
/******************************************************************************/
/*!
\file:      ChHashTable.h
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      April 10, 2021
\brief      This file contains the declarations needed to construct a templated
            Hash Table

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/******************************************************************************/

//---------------------------------------------------------------------------
#ifndef CHHASHTABLEH
#define CHHASHTABLEH
//---------------------------------------------------------------------------

#include <cstring>
#include <cmath>
#include "ObjectAllocator.h"
#include "support.h"

// client-provided hash function: takes a key and table size,
// returns an index in the table.
typedef unsigned (*HASHFUNC)(const char *, unsigned);

// Max length of our "string" keys
const unsigned MAX_KEYLEN = 10;

class HashTableException
{
  private:
    int error_code_;
    std::string message_;

  public:
    HashTableException(int ErrCode, const std::string& Message) :
        error_code_(ErrCode), message_(Message) {};

    virtual ~HashTableException() {
    }

    virtual int code() const {
      return error_code_;
    }

    virtual const char *what() const {
      return message_.c_str();
    }
    enum HASHTABLE_EXCEPTION {E_ITEM_NOT_FOUND, E_DUPLICATE, E_NO_MEMORY};
};


// HashTable statistical info
struct HTStats
{
  HTStats(void) : Count_(0), TableSize_(0), Probes_(0), Expansions_(0),
                  HashFunc_(0) {};
  unsigned Count_;      // Number of elements in the table
  unsigned TableSize_;  // Size of the table (total slots)
  unsigned Probes_;     // Number of probes performed
  unsigned Expansions_; // Number of times the table grew
  HASHFUNC HashFunc_;   // Pointer to primary hash function
  ObjectAllocator *Allocator_; // The allocator in use (may be 0)
};

template <typename T>
class ChHashTable
{
  public:

    typedef void (*FREEPROC)(T); // client-provided free proc (we own the data)

    struct HTConfig
    {
        HTConfig(unsigned InitialTableSize,
                 HASHFUNC HashFunc,
                 double MaxLoadFactor = 3.0,
                 double GrowthFactor = 2.0,
                 FREEPROC FreeProc = 0) :

        // The number of slots in the table initially.
        InitialTableSize_(InitialTableSize),
        // The hash function used in all cases.
        HashFunc_(HashFunc),
        // The maximum "fullness" of the table.
        MaxLoadFactor_(MaxLoadFactor),
        // The factor by which the table grows.
        GrowthFactor_(GrowthFactor),
        // The method provided by the client that may need to be called when
        // data in the table is removed.
        FreeProc_(FreeProc) {}

      unsigned InitialTableSize_;
      HASHFUNC HashFunc_;
      double MaxLoadFactor_;
      double GrowthFactor_;
      FREEPROC FreeProc_;
    };

      // Nodes that will hold the key/data pairs
    struct ChHTNode
    {
      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

```
/*****************************************************************************/
/*!
\file:      ChHashTable.cpp
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      April 10, 2021
\brief      This file contains the definitions needed to construct a templated
            Hash Table

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/*****************************************************************************/
#include "ChHashTable.h"

/*****************************************************************************/
/*!
\fn     template<typename T>
        ChHashTable<T>::ChHashTable(const HTConfig& Config,
        ObjectAllocator* allocator)

\brief  Constructor for ChHashTable

\param  Config - Configuration struct for this object

\param  allocator - Object allocator to allocate memory, if provided

*/
/*****************************************************************************/
template<typename T>
ChHashTable<T>::ChHashTable(const HTConfig& Config, ObjectAllocator* allocator)
: _config{Config}, _oa{allocator}
{
    //Initilise stats
    _stats.TableSize_ = _config.InitialTableSize_;
    _stats.HashFunc_ = _config.HashFunc_;

    //Initilise table array
    _table = new ChHTHeadNode[_stats.TableSize_];
}

/*****************************************************************************/
/*!
\fn     template<typename T>
        ChHashTable<T>::~ChHashTable()

\brief  Destructor for ChHashTable
*/
/*****************************************************************************/
template<typename T>
ChHashTable<T>::~ChHashTable()
{
    clear();
    delete[] _table;
}

/*****************************************************************************/
/*!
\fn     template<typename T>
        void ChHashTable<T>::insert(const char* Key, const T& Data)

\brief  Insert a key / data pair into the hash table

\param  Key - Key to determine place in the hash table

\param  Data - value to be stored in the hash table

*/
/*****************************************************************************/
template<typename T>
void ChHashTable<T>::insert(const char* Key, const T& Data)
{
    //Calculate load factor
    double loadFactor =(_stats.Count_+1)/static_cast<double>(_stats.TableSize_);

    //if loadFactor > max load factor, expand table
    if(loadFactor > _config.MaxLoadFactor_)
    {
        unsigned oldTableSize = _stats.TableSize_;
        double factor = std::ceil(_stats.TableSize_ * _config.GrowthFactor_);

        _stats.TableSize_ = GetClosestPrime(static_cast<unsigned>(factor));
        ChHTHeadNode* oldTable = _table;

        //reinsert items from old to new table
        _table = new ChHTHeadNode[_stats.TableSize_];

        for(unsigned i = 0; i < oldTableSize; ++i)
        {
            ChHTHeadNode headNode = oldTable[i];
            ChHTNode* node = headNode.Nodes;

            while(node)
            {
                ChHTNode* prev_node;

                ChHTNode* temp = node;
                node = node->Next;

                //Get index in table
                unsigned i = _stats.HashFunc_(temp->Key, _stats.TableSize_);

                //check if node exist in new table
                ChHTNode* NodeFound = SearchNode(i, Key, prev_node);

                //If node found is not first node, have to check for duplicate
```

```
                //If there is duplicate, will throw exeception
                if(NodeFound != nullptr)
                    throw HashTableException(HashTableException::E_DUPLICATE,
                    "Key inserted is already in Hash Table");

                temp->Next = _table[i].Nodes;
                _table[i].Nodes = temp;
                ++_table[i].Count;
            }

            oldTable[i].Nodes = nullptr;
        }

        delete[] oldTable;
        ++_stats.Expansions_;
    }

    //Get index in table
    unsigned i = _stats.HashFunc_(Key, _stats.TableSize_);

    ChHTNode* prev_node;

    ChHTNode* node = SearchNode(i, Key, prev_node);

    if(node != nullptr)
        throw HashTableException(HashTableException::E_DUPLICATE,
        "Key inserted is already in the Hash Table");

    //Create and insert node
    node = CreateNode(Key, Data);
    node->Next = _table[i].Nodes;
    _table[i].Nodes = node;

    ++_table[i].Count;
    ++_stats.Count_;
}

/******************************************************************************/
/*!
\fn     template<typename T>
        void ChHashTable<T>::remove(const char* Key)

\brief  Remove a node in Hash Table

\param  Key - ID of the data to be remove

*/
/******************************************************************************/
template<typename T>
void ChHashTable<T>::remove(const char* Key)
{
    unsigned i = _stats.HashFunc_(Key, _stats.TableSize_);

    ChHTNode* prev_node;
    ChHTNode* node = SearchNode(i, Key, prev_node);

    if(node == nullptr)
        throw HashTableException(HashTableException::E_ITEM_NOT_FOUND,
        "Key does not exist in table");

    //Remove node
    ChHTNode* temp = node;

    if(prev_node == node)
        _table[i].Nodes = node->Next;
    else
        prev_node->Next = node->Next;

    DeleteNode(temp);

    --_table[i].Count;
    --_stats.Count_;
}

/******************************************************************************/
/*!
\fn     template<typename T>
        const T& ChHashTable<T>::find(const char* Key) const

\brief  Find a node in Hash Table

\param  Key - ID of the node to be found

\return Returns referenced to the node requested
*/
/******************************************************************************/
template<typename T>
const T& ChHashTable<T>::find(const char* Key) const
{
    //Get index from table
    unsigned i = _stats.HashFunc_(Key, _stats.TableSize_);

    //Search nodes
    ChHTNode* prev_node;
    ChHTNode* node = SearchNode(i, Key, prev_node);

     if(node == nullptr)
        throw HashTableException(HashTableException::E_ITEM_NOT_FOUND,
        "Key does not exist in table");

    return node->Data;
}

/******************************************************************************/
/*!
\fn     template<typename T>
        void ChHashTable<T>::clear()
```

```
\brief  Delete all nodes in Hash Table, does not delete Hash Table itself

*/
/*****************************************************************************/
template<typename T>
void ChHashTable<T>::clear()
{
    for(unsigned i = 0; i < _stats.TableSize_; ++i)
    {
        ChHTHeadNode headNode = _table[i];
        ChHTNode* node = headNode.Nodes;

        while(node)
        {
            ChHTNode* temp = node;
            node = node->Next;

            DeleteNode(temp);

            --_table[i].Count;
            --_stats.Count_;
        }

        _table[i].Nodes = nullptr;
    }
}

/*****************************************************************************/
/*!
\fn     template<typename T>
        HTStats ChHashTable<T>::GetStats() const

\brief  Get Statistics of the Hash Table

\return Returns Statistics of the Hash Table
*/
/*****************************************************************************/
template<typename T>
HTStats ChHashTable<T>::GetStats() const
{
    return _stats;
}

/*****************************************************************************/
/*!
\fn     template<typename T>
        const typename ChHashTable<T>::ChHTHeadNode* ChHashTable<T>::GetTable()
        const

\brief  Get reference to the Hash Table

\return Returns reference to the Hash Table
*/
/*****************************************************************************/
template<typename T>
const typename ChHashTable<T>::ChHTHeadNode* ChHashTable<T>::GetTable() const
{
    return _table;
}

/*****************************************************************************/
/*!
\fn     template<typename T>
        typename ChHashTable<T>::ChHTNode* ChHashTable<T>::SearchNode
        (const unsigned& i, const char* Key, ChHTNode*& prev_node) const

\brief  Search for specific node and increment stats

\param  i- index of the node to be searched

\param  Key - ID of the node to be searched

\param  prev_node - Previous node

\return Returns referenced to the node searched
*/
/*****************************************************************************/
template<typename T>
typename ChHashTable<T>::ChHTNode* ChHashTable<T>::SearchNode
(const unsigned& i, const char* Key, ChHTNode*& prev_node) const
{
    //Loop through linked list to find key

    if(_table[i].Count == 0)
    {
        ++_stats.Probes_;
        return nullptr;
    }
    else
    {
        ChHTNode* node = _table[i].Nodes;
        prev_node = node;

        while(node)
        {
            ++_stats.Probes_;

            if(strcmp(node->Key, Key) == 0)
                return node;

            prev_node = node;
            node = node->Next;
        }

        ++_stats.Probes_;
        return node;
    }
```

```
}

/*****************************************************************************/
/*!
\fn     template<typename T>
        typename ChHashTable<T>::ChHTNode* ChHashTable<T>::
        CreateNode(const char* Key, const T& Data)

\brief  Create a Node with Key and Data

\param  Key - ID of the new node

\param  Data - Value of the new node

\return Returns the new node
*/
/*****************************************************************************/
template<typename T>
typename ChHashTable<T>::ChHTNode* ChHashTable<T>::CreateNode(const char* Key,
const T& Data)
{
    ChHTNode* newNode;

    if(_oa)
        newNode = new (_oa->Allocate()) ChHTNode(Data);
    else
        newNode = new ChHTNode(Data);

    std::strcpy(newNode->Key, Key);
    return newNode;
}

/*****************************************************************************/
/*!
\fn     template<typename T>
        void ChHashTable<T>::DeleteNode(ChHTNode* node)

\brief  Delete node in Hash Table

\param  node - Node to be deleted
*/
/*****************************************************************************/
template<typename T>
void ChHashTable<T>::DeleteNode(ChHTNode* node)
{
    if(_oa)
        node->~ChHTNode();

    delete node;
```

VPL

◄ Assignment 4: Graphs

Jump to...

Quiz: Leet Code ►