**Sabastian Fasano**

**February 2, 2025**

**CS-300**

**Module 4 HashTable**

The code is broken down into the following functions/structs/classes:

* Class HashTable
  + Class definition of the HashTable objects that hold
    - struct Node
      * Consists of a Bid struct
      * An integer key
      * Pointer to the next node
    - Hash, Constructors/destructors, Insert, PrintAll, Remove, and Search methods
    - Private members: vector node, tableSize
* HashTable()
  + Default constructor that sets the nodes vector’s size
* HashTable::hash(key)
  + The bidID is serving as the key
  + The key value is hashed using a modulo hash
* HashTable::Insert(Bid)
  + A Bid is passed
  + The function then adds the passed Bid as a new Node in the hash table (nodes)
  + The new Node is placed at the end of the chained nodes if there is a collision
* HashTable::PrintAll()
  + Function loops through the HashTable starting at the hash table head and outputs, to the console, 4 values from the Bid struct
* HashTable::Remove(String)
  + Function starts at the hashed location and searches for the passed String
  + Upon finding the String the Node containing it is freed from memory
  + The chain is updated accordingly if either is the Node containing String
* HashTable::Search(String)
  + Function starts at the head and searches for the passed String
  + Upon finding the String the Node containing it is returned
* HashTable::Size()
  + A get method used to access the private member, size
* strToDouble
  + Used to convert the CSV file data into useable value
* Bid
  + Struct containing the data
  + Used with the vector that will be sorted
* displayBid
  + Used to send the values contained in the vector to the console
* loadBids
  + Function used to read in the csv data
  + Can read the csv path in from arguments or use a default path
  + Add parts of the data into the Bid structure and then adds that Bid to the unsorted HashTable
* main
  + *main* is the primary driver for the application
  + *main* has a menu to allow a user to enter a bid, load the data, view the data, and delete a node and then exit the application
  + *main* also reports the timing each algorithm takes to perform the sort using the *time.h* library

The code implementation was fairly straightforward, particularly since the parser was provided to me. However, I did encounter some unexpected challenges when adding nodes to the hash table. Initially, despite using what appeared to be correct syntax, the IDE consistently flagged errors. After troubleshooting and reviewing my approach multiple times, I couldn’t pinpoint the exact issue. Strangely enough, the problem resolved itself after restarting the virtual machine. This experience highlighted the occasional quirks of development environments, where sometimes a simple restart can resolve seemingly inexplicable errors. It also reinforced the importance of maintaining patience and systematically debugging issues rather than immediately assuming there is a flaw in the logic.

Pseudocode:

**Main** Function()

**Read** cmd arguments

**Store** argument as CSV file path

**If** no cmd arguments load default CSV file path

**Loop** while choice is not equal to ‘9’

**Output** menu

**Get** user input; Store in choice

**Validate** user input

**If** choice is not 1-4 or 9 throw an error

**If** choice equals ‘1’

**Start** the clock and **store** in ticks

**Call** loadBids and store CSV data in HashTable *bidTable*

**Output** number of records in the CSV file

**Stop** the clock

**Output** the elapsed time needed to read in the CSV file

**If** choice equals ‘2’

**Call** PintAll() with *bidTable*

**If** choice equals ‘4’

**Start** the clock and store in ticks

**Call** Search() passing a *bidKey* to search for

**Stop** the clock

**Output** the elapsed time needed to find the *bidKey*

**If** choice equals ‘4’

**Call** Remove() passing *bidKey*

**If** Choice equals ‘9’

**Exit** the application

**Output** ‘Good bye’

**End**

**HashTable::hash(int)**

**Return** *key* modulo *tableSize*

**End**

**HashTable::Insert(*Bid*)**

**Call** hash() with *Bid* member *bidId* and **Store** in *tempKey*

**Check** if the hash location within nodes if the node is empty since start

**If** empty since start **Create** *newNode* with *Bid* and *tempKey*

**Set** hash location equal to *newNode*

**If** not empty since start

**Create** a *Node* pointer and set to the address of the hash table index

**Create** a *newNode* pointer with *Bid* and *tempKey*

**Loop** until the end of the linked chain is found and add *newNode*

**End**

**HashTable::PrintAll()**

**Create** a new Node pointer and **Set** to the address of the nodes beginning

**Loop** through the list; starting at the beginning

**If** key value at iteration is not equal to UINT\_MAX

**Output** to console: *bidId, title, amount, fund*

**End**

**HashTable::Search(*String*)**

**Create** a new *Node* pointer called *cursor*

**Set** *cursor* to the bucket at the hash location

**Loop** until *cursor* is NULL (end of the list)

**If** the Node at cursor contains a *bidId* equal to *String*

**Return** cursor

**Set** *cursor* equal to the next Node

**End**

**HashTable::Remove(String)**

**Create** a new *Node* pointer called *cursor*

**Set** *cursor* to the bucket at the hash location

**Create** a new *Node* pointer called *tempNode*

**Check** if *curor* is pointing at a chain of collisions or a single bucket

**If** a chain

**Check** if *cursor* *Bid* member, *bidId*, matches *String*

**If** match **Set** *tempNode* to the next node

**Set** *cursor* equal to *tempNode*

**Delete** *tempNode*

**If** not a match at start of chain **Scan** the list for a match

**If** a single bucket

**Set** the bucket *Bid* members equal to the default constructor

**End**

//============================================================================

// Name : HashTable.cpp

// Author : Sabastian Fasano Feb 2, 2025

// Version : 1.0

// Copyright : Copyright © 2021 SNHU COCE

// Description : Hello World in C++, Ansi-style

//============================================================================

#include <algorithm>

#include <climits>

#include <iostream>

#include <string> // atoi

#include <time.h>

#include <Windows.h>

#include "CSVparser.hpp"

using namespace std;

//============================================================================

// Global definitions visible to all methods and classes

//============================================================================

const unsigned int DEFAULT\_SIZE = 179;

const int GLOBAL\_SLEEP\_TIME = 5000;

// forward declarations

double strToDouble(string str, char ch);

// define a structure to hold bid information

struct Bid {

string bidId; // unique identifier

string title;

string fund;

double amount;

Bid() {

amount = 0.0;

}

};

//============================================================================

// Hash Table class definition

//============================================================================

/\*\*

\* Define a class containing data members and methods to

\* implement a hash table with chaining.

\*/

class HashTable {

private:

// Define structures to hold bids

struct Node {

Bid bid;

unsigned int key;

Node \*next;

// default constructor

Node() {

key = UINT\_MAX;

next = nullptr;

}

// initialize with a bid

Node(Bid aBid) : Node() {

bid = aBid;

}

// initialize with a bid and a key

Node(Bid aBid, unsigned int aKey) : Node(aBid) {

key = aKey;

}

};

vector<Node> nodes;

unsigned int tableSize = DEFAULT\_SIZE;

unsigned int hash(int key);

public:

HashTable();

HashTable(unsigned int size);

virtual ~HashTable();

void Insert(Bid bid);

void PrintAll();

void Remove(string bidId);

Bid Search(string bidId);

};

/\*\*

\* Default constructor

\*/

HashTable::HashTable() {

// FIXME (1): Initialize the structures used to hold bids

// Initalize node structure by resizing tableSize

nodes.resize(tableSize);

}

/\*\*

\* Constructor for specifying size of the table

\* Use to improve efficiency of hashing algorithm

\* by reducing collisions without wasting memory.

\*/

HashTable::HashTable(unsigned int size) {

// invoke local tableSize to size with this->

// resize nodes size

this->tableSize = size;

nodes.resize(size);

}

/\*\*

\* Destructor

\*/

HashTable::~HashTable() {

// FIXME (2): Implement logic to free storage when class is destroyed

// erase nodes beginning

nodes.erase(nodes.begin());

}

/\*\*

\* Calculate the hash value of a given key.

\* Note that key is specifically defined as

\* unsigned int to prevent undefined results

\* of a negative list index.

\*

\* @param key The key to hash

\* @return The calculated hash

\*/

unsigned int HashTable::hash(int key) {

// FIXME (3): Implement logic to calculate a hash value

// return key tableSize

//use the modulo hash from Zybooks 5.7

return key % tableSize;

}

/\*\*

\* Insert a bid

\*

\* @param bid The bid to insert

\*/

void HashTable::Insert(Bid bid) {

// FIXME (5): Implement logic to insert a bid

// create the key for the given bid

// retrieve node using key

// if no entry found for the key

// assign this node to the key position

// else if node is not used

// assing old node key to UNIT\_MAX, set to key, set old node to bid and old node next to null pointer

// else find the next open node

// add new newNode to end

unsigned int tempKey = hash(stoi(bid.bidId));

//look to see if the node exists

//If the key is set to the default constructor overwrite

if (nodes.at(tempKey).key == UINT\_MAX) {

//make the new Node

Node newNode = Node(bid, tempKey);

//add the item

nodes.at(tempKey) = newNode;

}

else{//now we need to get to the bottom of the linked list

//make a cursor node as an iterator and point at the hash location

Node\* currNode = &nodes.at(tempKey);

//make the new node pointer

Node\* newNode = new Node(bid, tempKey);

// scan the linked list to find the last position

while (currNode->next != NULL) {

currNode = currNode->next;

}

//now append the new node

currNode->next = newNode;

}

}

/\*\*

\* Print all bids

\*/

void HashTable::PrintAll() {

// FIXME (6): Implement logic to print all bids

// for node begin to end iterate

// if key not equal to UINT\_MAx

// output key, bidID, title, amount and fund

// node is equal to next iter

// while node not equal to nullptr

// output key, bidID, title, amount and fund

// node is equal to next node

int j = 0;//small logical test

for (unsigned int i = 0; i < tableSize; i++) {

Node\* currNode = &nodes.at(i);

if (currNode->key != UINT\_MAX) {

j++;

cout << currNode->key << ": " << currNode->bid.bidId << " | " << currNode->bid.amount

<< " | " << currNode->bid.fund << endl;

while (currNode->next != nullptr) {

j++;

currNode = currNode->next;

cout << currNode->key << ": " << currNode->bid.bidId << " | " << currNode->bid.amount

<< " | " << currNode->bid.fund << endl;

}

}

}

//cout << j << endl;;

return;

}

/\*\*

\* Remove a bid

\*

\* @param bidId The bid id to search for

\*/

void HashTable::Remove(string bidId) {

// FIXME (7): Implement logic to remove a bid

// set key equal to hash atoi bidID cstring

// erase node begin and key

//see currNode to the head node

Node\* cursor = &nodes.at(hash(stoi(bidId)));

Node\* tempNode = NULL;

//see if there is a chain; probably will be given 12000+ receords

//if there is a chain adopt that methodology of scan and shift

//if there is no chain just check the head node

if (cursor->next != nullptr) { //if the bucket head is not null there is a chain

//special case; matches the head of the chain

if (cursor->bid.bidId == bidId) {

//point tempNode at the next node

//copy +1 node to headnode

//delete +1 node

tempNode = cursor->next;

cursor->key = tempNode->key;

cursor->bid = tempNode->bid;

cursor->next = tempNode->next;

free(tempNode);

return;

}

else { //we now need to scan the chain for a match

while (cursor != NULL && cursor->bid.bidId != bidId) {

tempNode = cursor; //set temp to the current currsor

cursor = cursor->next; //move cursor ahead

}

if (cursor == NULL) {//scanned tjhe whole list and didn't find the key

cout << " Key not found." << endl;

return;

}

tempNode->next = cursor->next;

delete cursor;

}

}

else {

if (cursor->bid.bidId == bidId) { //match the head node with no chaining

//set the bucket back to uninitiated

Node newNode = Node();

cursor->key = newNode.key;

cursor->bid = newNode.bid;

cursor->next = nullptr;

return;

}

else { //no chain and no match at the head; maybe previously deleted or never existed

cout << "No Bid exists." << endl;

}

}

}

/\*\*

\* Search for the specified bidId

\*

\* @param bidId The bid id to search for

\*/

Bid HashTable::Search(string bidId) {

Bid bid;

// FIXME (8): Implement logic to search for and return a bid

// create the key for the given bid

// if entry found for the key

//return node bid

// if no entry found for the key

// return bid

// while node not equal to nullptr

// if the current node matches, return it

//node is equal to next node

//see currNode to the head node

Node\* cursor = &nodes.at(hash(stoi(bidId)));

while (cursor != NULL) {

if (cursor->bid.bidId == bidId) {

return cursor->bid;

}

cursor = cursor->next;

}

return bid;

}

//============================================================================

// Static methods used for testing

//============================================================================

/\*\*

\* Display the bid information to the console (std::out)

\*

\* @param bid struct containing the bid info

\*/

void displayBid(Bid bid) {

cout << bid.bidId << ": " << bid.title << " | " << bid.amount << " | "

<< bid.fund << endl;

return;

}

/\*\*

\* Load a CSV file containing bids into a container

\*

\* @param csvPath the path to the CSV file to load

\* @return a container holding all the bids read

\*/

void loadBids(string csvPath, HashTable\* hashTable) {

cout << "Loading CSV file " << csvPath << endl;

// initialize the CSV Parser using the given path

csv::Parser file = csv::Parser(csvPath);

// read and display header row - optional

vector<string> header = file.getHeader();

//for (auto const& c : header) {

// cout << c << " | ";

//}

//cout << "" << endl;

try {

// loop to read rows of a CSV file

for (unsigned int i = 0; i < file.rowCount(); i++) {

// Create a data structure and add to the collection of bids

Bid bid;

bid.bidId = file[i][1];

bid.title = file[i][0];

bid.fund = file[i][8];

bid.amount = strToDouble(file[i][4], '$');

//cout << "Item: " << bid.title << ", Fund: " << bid.fund << ", Amount: " << bid.amount << endl;

// push this bid to the end

hashTable->Insert(bid);

}

} catch (csv::Error &e) {

std::cerr << e.what() << std::endl;

}

}

/\*\*

\* Simple C function to convert a string to a double

\* after stripping out unwanted char

\*

\* credit: http://stackoverflow.com/a/24875936

\*

\* @param ch The character to strip out

\*/

double strToDouble(string str, char ch) {

str.erase(remove(str.begin(), str.end(), ch), str.end());

return atof(str.c\_str());

}

/\*\*

\* The one and only main() method

\*/

int main(int argc, char\* argv[]) {

// process command line arguments

string csvPath, bidKey;

switch (argc) {

case 2:

csvPath = argv[1];

bidKey = "98109";

break;

case 3:

csvPath = argv[1];

bidKey = argv[2];

break;

default:

csvPath = "U:/CS 300 Hash Table Assignment Student Files/eBid\_Monthly\_Sales\_Dec\_2016.csv";

bidKey = "98010";//no collision

//bidKey = "98011";//collision head

//bidKey = "98190";//collision tail

}

// Define a timer variable

clock\_t ticks;

// Define a hash table to hold all the bids

HashTable\* bidTable = new HashTable();

Bid bid;

int choice = 0;

string anyKey = " ";

bool goodInput;

while (choice != 9) {

cout << "Menu:" << endl;

cout << " 1. Load Bids" << endl;

cout << " 2. Display All Bids" << endl;

cout << " 3. Find Bid" << endl;

cout << " 4. Remove Bid" << endl;

cout << " 9. Exit" << endl;

cout << "Enter choice: ";

try {

cin >> choice;

if ((choice > 0 && choice < 5) || (choice == 9)) {// limit the user menu inputs to good values

goodInput = true;

}

else {//throw error for catch

goodInput = false;

throw 1;

}

switch (choice) {

case 1:

//bidTable = new HashTable(); //comment out to have many many collision upon reload

// Initialize a timer variable before loading bids

ticks = clock();

// Complete the method call to load the bids

loadBids(csvPath, bidTable);

// Calculate elapsed time and display result

ticks = clock() - ticks; // current clock ticks minus starting clock ticks

cout << "time: " << ticks << " clock ticks" << endl;

cout << "time: " << ticks \* 1.0 / CLOCKS\_PER\_SEC << " seconds" << endl;

Sleep(GLOBAL\_SLEEP\_TIME);

break;

case 2:

bidTable->PrintAll();

cout << "\n Enter \'y\' to continue..." << endl;

cin >> anyKey;

break;

case 3:

ticks = clock();

bid = bidTable->Search(bidKey);

ticks = clock() - ticks; // current clock ticks minus starting clock ticks

if (!bid.bidId.empty()) {

displayBid(bid);

}

else {

cout << "Bid Id " << bidKey << " not found." << endl;

}

cout << "time: " << ticks << " clock ticks" << endl;

cout << "time: " << ticks \* 1.0 / CLOCKS\_PER\_SEC << " seconds" << endl;

Sleep(GLOBAL\_SLEEP\_TIME);

break;

case 4:

bidTable->Remove(bidKey);

Sleep(GLOBAL\_SLEEP\_TIME);

break;

case 9:

break;

default:

throw 2;

}

}

catch (int err) {

std::cout << "\nPlease check your input." << endl;

Sleep(GLOBAL\_SLEEP\_TIME);

}

//need to clear the cin operator of extra input, e.g., 9 9, or any errors generated by bad input, e.g., 'a'

cin.clear();

cin.ignore();

//clear the consolse to redraw a fresh menu

system("cls");

}

cout << "Good bye." << endl;

Sleep(GLOBAL\_SLEEP\_TIME);

return 0;

}