Overview

A strongly-written hashtable that uses a string as a key and can hold any object.

Introduction

This document holds the technical design of the CSCI231\_Assignment7\_Schmidt program and serves as a pseudo lessons learned for the changes made to the initial design. The program is intended to serve as a hashtable with a hashing algorithm for storing any object and using a string as a key. The user picks the size of the Hashtable so the user has full control over its performance, a larger sized hashtable will have more buckets and less collisions and a smaller sized hashtable will have less buckets and more collisions.

## Scope

Create a hashtable that uses a string as a key.

### PROCESSING

The user creates a SchmidtsHashTable by giving it an integer, this integer sets the size of the array and plays into the hash function. Prime numbers are suggested to be used as that integer, they seem to work better.

Then the user sets an object into the array using the void set() and giving it a string key and an object to hold.

The hashing algorithm:

int largeInt = 0;

For every byte c in the key,

largeInt = largeInt + largeInt + (int)c

largeInt = largeInt \* (largeInt/(int)c)

if largeInt is less than 0

then largeInt = largeInt \* -1

int hashKey is equal to the remainder of largeInt / (hashTableSize-1)

end process;

If a collision happens, then the collision is added to the LinkedList in that array spot. That hashKey would then hold two objects. When the get() function is called, it returns an array of objects that is all the objects in that selection. Since my HashTable is strongly written, it is up to the user to check the key against the objects in that bucket because I cannot touch the objects and see what their actual key is.

For testing purposes, ten student class instances are added to the Hashtable, then pulled from the hashtable and prints when it pulls and if a collision existed.

### DATA

The logical and physical data structure of files should be defined in detail.

Data structure definitions must include the:

description of each element, e.g. name, type, dimension;

relationships between the elements, i.e. the structure;

range of possible values of each element;

initial values of each element.

SchmidtsHashTable

The Hashtable uses an array that holds LinkedLists of type <object> based off of the array size given when constructed. It only has a getter and setter that holds the hash function and returns/sets the objects from/to the array.

### COMPONENTS

HashTableTester

* Holds the static main and initiates the simulator using student testing data.

SchmidtsHashTable

* Holds the HashTable and HashTable functions.I

|  |
| --- |
| **ProcessHeap** |
| -Int: m\_hashTableSize;  -LinkedLIst<object>[]: m\_hashTable |
| +SchmidtsHashTable(int): constructor  +set(string, object): void  +get(string): object[] |

### TESTING

Present one or more named scenarios that will be utilized to test the application.

The testing plan should be repeatable.

Describe the scenario in detail, the steps required to execute the test, the input data, the output data, and the success criteria.

Present a summary of the testing scenarios before the details of each scenario.

|  |  |  |
| --- | --- | --- |
| Scenario | Description | Pass/Fail |
| 1st student data run | Run simulation based off input file | Fail |
| 2nd student data run | Run simulation based off input file | Fail |
| 3rd student data run | Run simulation based off input file | Fail |
| 4th student data run | Run simulation based off input file | Fail |

Student data:

students.AddLast(new student("10056486"));

students.AddLast(new student("50098451"));

students.AddLast(new student("70056165"));

students.AddLast(new student("80856495"));

students.AddLast(new student("50056168"));

students.AddLast(new student("10056498"));

students.AddLast(new student("05686492"));

students.AddLast(new student("60058945"));

students.AddLast(new student("80058794"));

students.AddLast(new student("87948148"));

Scenario #1- student data test

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1. | Run program | Input: Student Data + 64 |
| 2. |  | Output: StudentIds pulled from Hashtable |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| EXPECTED OUTPUT | |  |
| ACTUAL OUTPUT | | My hashing function gave a hash of -8, so it broke when it tried to pull the location -8. |
| RESULTS – | | Fail, I added an if statement to check if the number is negative, and turn it into a positive. |

Scenario #2- student data test

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1. | Run program | Input: student data + 64 |
| 2. |  | Output: studentIds pulled from hashtable |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| EXPECTED OUTPUT | | To process in order of 4,1,2,3 |
| ACTUAL OUTPUT | | My hashing program worked and did its job, but two collisions were found. I only made the hash table a size of 64 though. |
| RESULTS – | | Fail, too many collisions |

Scenario #3 – student data test

##### 

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1. | Run program | Input: Student data + 124 (array size) |
|  |  | Output: studentIds pulled from hashtable |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| EXPECTED OUTPUT | | To process in order of 4,1,2,3 |
| ACTUAL OUTPUT | | It got worse. I had more collisions than with 64, like 6 collisions out of the 8 students. |
| RESULTS – | | Fail, way too many collisions |

##### *Scenario #4 – student data test*

|  |  |  |
| --- | --- | --- |
| Step | Description | Input/Output |
| 1. | Enter input address | Input: student data + 199 |
| 2. | Enter desired file name | Output: studentId pulled from hashtable. |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
| EXPECTED OUTPUT | | studentIds pulled from hashtable. |
| ACTUAL OUTPUT | | It worked! No collisions! |
| RESULTS – | | Pass: By the magic of using a prime number it mystically worked better. |