Fernando Salazar

Assignment 7

1. Create a design **before** you begin to code that describes or shows how we can store data in a hash table and what kind of problem we could solve with a hash table.
2. Create some tests (at least one per piece of functionality) **before** you begin coding that you want your hashtable to pass before you start coding.
3. Create a hashtable that resolves collisions by simply overwriting the old value with the new value, including at least:
   1. Describe the way that you decide on hashing a value  
      (this can be simple or complex based on how interesting you find the topic)
   2. An insert function that places the value at the appropriate location based on its hash value
   3. A contains function that returns whether the value is already in the hashtable
   4. (optional) A delete function that removes a value based on its hash and then returns that value…
4. Then create a smarter hashtable (double hashing or chaining) including at least the same functions as the simple hashtable
5. Compare some information relating to collisions (frequency) and their effect on complexity (of insert and contains methods)
6. Once you have implemented and tested your code, add to the README file what line(s) of code or inputs and outputs show your work meeting each of the above requirements (or better, include a small screen snip of where it meets the requirement!).

//Design

    // Duck Hash

        store duck breeds and blurb

// hash function,

    // int hash

// array to store hash key/ values

        // key and data for that key

// add function

Insert function

// search function

-contains functions

// overwrite for collessions

// insert/ add function will take of this because it will just overwrite

// get function

Wil grab the date for each key

//collision tracker added afterwards to keep track of collision and when they happened

/// second program with double hashing  // mattch functions to next

//requirements simple hash

Hash function

The hash function is pretty simple. I go through each character in the word that is being hashed.

I take the hash value and multiply it by 11, a prime number. For the first character, the hash value is 0. I take the value from this and add it to the character's value. I then run it through moduos to make sure it's not bigger than the table. The end result becomes the hash value for the next run. When it is done, this becomes the index location.

A computer screen with text and numbers

Description automatically generated

//contains function

A screen shot of a computer code

Description automatically generated

// insert function // auto overwrite any collisions

A screen shot of a computer code

Description automatically generated

// get function retrieves data for each key

A screen shot of a computer code

Description automatically generated

//Collison tracker function

A screen shot of a computer code

Description automatically generated

// smarthash aks fancyhash // used double hashing

// started with simple hash as basic, first hash is the same

// second hash is almost identical but with a different prime number

A computer code with numbers and symbols

Description automatically generated

// couple of changes to the insert function

A screen shot of a computer code

Description automatically generated

// contains functions

A computer screen shot of a program code

Description automatically generated

//get functions

A computer screen shot of a program code

Description automatically generated

//testing for fancy and simple hash is identical for the first two functions, only large difference is the collison check on the fancy hash.

//simple hash

A screen shot of a computer program

Description automatically generated

//results for simple testing

A screenshot of a computer

Description automatically generated

//Fancy

A screen shot of a computer program

Description automatically generated

//Results

A screenshot of a computer program

Description automatically generated