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.

```
// hash magic
int simplehash::hash(const String& key) {
    // starting hash value
    int hash_value = 0;
    // loops through every char in key
    for (char c : key) {
        // each loop it multipies the starting hash value  by 11 and adds the value of the char itself.
        //then it finishs off by using  modulo  to keep it in the table size.
        hash_value = (hash_value * 11 + c) % table.size();
    }
    return hash_value; // returned value
}
```

//contains function

```
bool simplehash::contains(const String& key) {
    // calls hash function to generate location
    int index = hash(key);
    // returns the value at the postion.
    return table[index].has_value();
}
```

// insert function // auto overwrite any collisions

```
void simplehash::insert(const String& key, const String& value) {
    // calls hash function to generate location
    int index = hash(key);
    // adds to the collisionCount if it already has a value
    if (table[index].has_value()) {
        //notice
        std::cout << "collision at index " << index << " for " << key << std::endl;
        collisionCount++;
    }
    //place value into table at location
    //by default collison not an issue, will override
    table[index] = value;
}
```

// get function retrieves data for each key

```
std::optional<String> simplehash::get(const String& key) {
    // calls hash function to generate location
    int index = hash(key);
    // pops it into table to return
    return table[index];
}
```

//Collison tracker function

```
// keeps track of collisons, just an int
int simplehash::collisionTracker()const{
    return collisionCount;
}
```

// 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

```
// has magic 2 repeat of 1
int fancyhash::secondHash(const String& key) {
    int hash_value = 0;
    // a different prime number for the second hash function
    for (char c : key) {
        hash_value = (hash_value * 7 + c) % table.size();
    }
    // extra step to make sure step size is not zero
    return hash_value == 0 ? 1 : hash_value;
}
```

// couple of changes to the insert function

```
void fancyhash::insert(const String& key, const String& value) {
    // calls hash functions to generate location
    int index = hash(key);
    int stepSize = secondHash(key);
    // adds to the collisionCount if it already has a value
    while (table[index].has_value() && table[index]->first != key) {
        std::cout << "collision at index " << index << " for " << key << std::endl;
        collisionCount++;
        //new index that generated from by adding both hashes and making sure its not bigger than table
        index = (index + stepSize) % table.size();
    }
    //place value into table at location
    //by default collison not an issue, will override
    table[index] = std::make_pair(key, value);
}
```

// contains functions

```
bool fancyhash::contains(const String& key) {
    // calls hash functions to generate location
    int index = hash(key);
    int stepSize = secondHash(key);
    //saves the orginal index
    int initialIndex = index;
    // loops for key using double hash
    while (table[index].has_value()) {
        // checks with first hash
        if (table[index]->first == key) {
            return true;
        }
        //  using the second hash fuction
        index = (index + stepSize) % table.size();
        if (index == initialIndex) {
            break;
        }
    }
    return false;
}
```

//get functions

```
std::optional<String> fancyhash::get(const String& key) {
    // calls hash functions to generate location
    int index = hash(key);
    int stepSize = secondHash(key);
    //saves the orginal index
    int initialIndex = index;
    // loops for key using double hash
    while (table[index].has_value()) {
        if (table[index]->first == key) {
            //returns value when found
            return table[index]->second;
        }
        // second hash
        index = (index + stepSize) % table.size();
        //checks for change in index
        if (index == initialIndex) {
            break;
        }
    }
    return std::nullopt;
}
```

//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

```cpp
//  simple hash testing
void testInsertGet(simplehash& simple) {
    std::cout << std::endl;
    std::cout << "Testing insert and get" << std::endl;
    std::cout << std::endl;
    //test duck data
    std::cout << "data inserted and retretived" << std::endl;
    simple.insert("Mallard", "A common duck breed.");
    simple.insert("Pekin", "A popular breed for meat.");
    simple.insert("yellow","A rubber ducky.");
    simple.insert("10", "A number of ducks.");

    // checks if in table

    std::cout << "checking if key is there " << std::endl;
    std::cout << "return of 1 if true" << std::endl;

    std::cout << "Mallard: " << simple.contains("Mallard") << std::endl;
    std::cout << "yellow: " << simple.contains("yellow") << std::endl;
    std::cout << "Pekin: " << simple.contains("Pekin") << std::endl;
    std::cout << "10: " << simple.contains("10") << std::endl;
    std::cout << std::endl;
    //test data retrival
    std::cout << "Grabs information for each key" << std::endl;
    std::cout << "yellow: " << simple.get("yellow").value() << std::endl;
    std::cout << "10: " << simple.get("10").value() << std::endl;
    std::cout << "Mallard: " << simple.get("Mallard").value() << std::endl;
    std::cout << std::endl;
}

void testReplacement(simplehash& simple) {
    //Test replaceing key , using information with key to show it works
    std::cout << "Testing Replacing key" << std::endl;
    simple.insert("Mallard", "A different description for Mallard.");
    std::cout << "Mallard Info after replacement: " << simple.get("Mallard").value() << std::end
    std::cout << std::endl;
}
        // test the collsion count by causing a collision
void testCollisionCount(simplehash& simple) {
    std::cout << "Testing Collision" << std::endl;
    // adds another pekin
    simple.insert("Pekin", "Another description for Pekin.");
    std::cout << "Collision Count: " << simple.collisionTracker() << std::endl;
    std::cout << std::endl;
}
```

//results for simple testing

```
terpreter=mi'
Simple

Testing insert and get

data inserted and retretived
checking if key is there
return of 1 if true
Mallard: 1
yellow: 1
Pekin: 1
10: 1

Grabs information for each key
yellow: A rubber ducky.
10: A number of ducks.
Mallard: A common duck breed.

Testing Replacing key
collision at index 1 for Mallard
Mallard Info after replacement: A different description for Mallard.

Testing Collision
collision at index 3 for Pekin
Collision Count: 2
```

//Fancy

```cpp
// fancy hash testing
// first 2 test are a rehash of simple test, to make sure it still works like expected
void testInsertGetFancy(fancyhash& fancy) {
    std::cout << std::endl;
    std::cout << "Testing insert and get for fancyhash" << std::endl;
    std::cout << std::endl;
    fancy.insert("Mallard", "A common duck breed.");
    fancy.insert("Pekin", "A popular breed for meat.");
    fancy.insert("yellow", "A rubber ducky.");
    fancy.insert("10", "A number of ducks.");

    std::cout << "Mallard: " << fancy.contains("Mallard") << std::endl;
    std::cout << "yellow: " << fancy.contains("yellow") << std::endl;
    std::cout << "Pekin: " << fancy.contains("Pekin") << std::endl;
    std::cout << "10: " << fancy.contains("10") << std::endl;
    std::cout << std::endl;

    std::cout << "yellow: " << fancy.get("yellow").value() << std::endl;
    std::cout << "10: " << fancy.get("10").value() << std::endl;
    std::cout << "Mallard: " << fancy.get("Mallard").value() << std::endl;
    std::cout << std::endl;
}

void testReplacementFancy(fancyhash& fancy) {
    std::cout << "Testing Replacing key in fancyhash" << std::endl;
    fancy.insert("Mallard", "A different description for Mallard.");
    std::cout << "Mallard Info after replacement: " << fancy.get("Mallard").value() << std::endl;
    std::cout << std::endl;
}
// added a little more to collision test, I made sure I could retrive the data after it was placed the new position after coll
void testCollisionCountFancy(fancyhash& fancy) {
    std::cout << "Testing Collision in fancyhash" << std::endl;
    fancy.insert("Pekin", "Another description for Pekin.");
    fancy.insert("test1", "This should cause a collision");
    fancy.insert("test2", "This should cause another collision");
    std::cout << "Collision Count: " << fancy.collisionTracker() << std::endl;
    std::cout << std::endl;
    std::cout << "Grabs information for each key" << std::endl;
    std::cout << "test1: " << fancy.get("test1").value() << std::endl;
    std::cout << "test2: " << fancy.get("test2").value() << std::endl;
}
```

//Results

```
Fancy

Testing insert and get for fancyhash

Mallard: 1
yellow: 1
Pekin: 1
10: 1

yellow: A rubber ducky.
10: A number of ducks.
Mallard: A common duck breed.

Testing Replacing key in fancyhash
Mallard Info after replacement: A different description for Mallard.

Testing Collision in fancyhash
collision at index 7 for test1
collision at index 8 for test2
Collision Count: 2

Grabs information for each key
test1: This should cause a collision
test2: This should cause another collision
PS C:\Users\ferna\Documents\CS-260\Assignment7>
```