**Izak Bunda – UID 305783387 – Due Tuesday, June 7, 2022**

A small aside about prime numbers. I learned that multiplying two prime numbers produce a number whose factors are only those two prime numbers and 1. This is an important fact I had to consider for creating my functions because in order to reduce collisions as much as possible in the hash table, I had to make sure that the hash keys were unique. Cryptography.

I struggled a lot in creating my hash function because there were random times where the whole program wouldn’t build just because of a single change in my hash function. Sometimes the issue would be bad access, but most of the time, according to XCode, I was violating some rule instated in the string class. I still don’t really understand what was causing that error, but I eventually debugged and found that my hash function sometimes returned an insanely high integer value (greater than the upper limit of an unsigned long). This made sense because my original hash function originated as a big prime number like 1471 (picked randomly) and multiplied each ASCII value of a char by 33 (from djb2), then multiplied the total again by 33. You can see how quickly this value can grow with a large string. So, in order to fix this, I simply reduced it incrementally until I reached my final prime numbers which are listed below.

**Time complexity is marked red**

**Structures**:

**struct dataBucket –** This struct holds the name, line number, scope number, and a pointer which is used to sort of make the *dataBucket* into a singly-linked list.

**class NameTableImpl –** This class has the *enterScope*, *exitScope*, *declare*, *find*, *insert*, and *hashFunction*. I essentially merged the hash table into this class which is why *find*, *insert*, and *hashFunction* are its member functions. For its member variables, I implemented an integer *scopeNum* which keeps track of the scope number, a vector of integers *hashNum* which keeps track of the hash keys, and an array of pointers of dataBuckets called *m\_buckets*. This array is initialized to 19997 because it is the biggest prime number before 20000, the max amount of buckets we can implement for this project.

**Non-trivial functions:**

// my personal hashfunction:

// comments: this is a franken-function because I pulled from many different published hash functions online (ex. djb2 algorithm by Dan Bernstein) and the “standard” hash function. Essentially, I start off at some prime number (147) and then multiply the ASCII value of the first three characters of the id by another prime number (7) and add it to the total. I made sure that if the id is less than three characters, then it just goes through the whole string. This is the part that I adapted from the djb2 algorithm is adding to and multiplying by prime numbers. The final part of this which divides by the size of the hash table is adapted from the “standard” hash function.

**unsigned int NameTableImpl::hashFunction(const string& id) const; O(1)**

initialize an auto data type hash to 147

initialize an auto data type idSize to the size of id

if idSize is greater than 3

for the first three characters

add the ASCII value of the character to the hash

multiply hash by 7

otherwise

for the entire length of the string

add the ASCII value of the character to the hash

multiply hash by 7

divide the final value of hash by the size of the hash table

return the final hash value

// this is the constructor

**NameTableImpl::NameTableImpl(); O(1)**

Loop through the entire m\_buckets array and initialize all pointers to nullptr

**void NameTableImpl::insert(const string& id, int num, int scope, dataBucket\* &record); O(1)**

initialize an integer to the hash key of the id using the hashFunction

initialize a pointer to dataBucket and point it to the corresponding bucket to the hash key from above

create a new dataBucket with the listed parameters

point the next to nullptr

**void NameTableImpl::enterScope(); O(1)**

push back -1 into the hashNum vector as it represents entering a scope

increment scopeNum

**bool NameTableImpl::exitScope(); O(N) – with respect to the number of items in the scope**

if scopeNum is 0, return false

initialize an integer to the back value of the hasnNum vector

while this integer is not -1 // which means that it hasn’t reached an enterScope

if the m\_bucket at the current hash has something in it

point m\_bucket to the next

delete m\_bucket

remove this hash key

then assign the integer to the next hash key

Then, one more time remove the last hash key

And decrement the scopeNum // to indicate that we left a scope

Return true

**bool NameTableImpl::declare(const string& id, int num); O(1)**

If id is “”, return false

Initialize an integer to the current amount of scopeNum

Initialize a pointer to a dataBucket to the corresponding bucket of the id

// this means it is already in the hashTable

If this dataBucket scope equal the scopeNum and dataBucket id equal the id

Return false

Otherwise

Create a new pointer to a dataBucket

Use the insert function with the parameters, the current scopeNum, and the pointer above

Pushback the corresponding hash key of the id to hashNum

Return true

**int NameTableImpl::find(const string& id) const; O(N) - with respect to the number of collisions in the bucket**

if the id is empty, return false

initialize integer scope to zero, num to -1, and x to the corresponding hash key for the id

initialize a pointer to dataBucket to the dataBucket corresponding to x

loop through the list

if the id corresponding to the pointer equals the id in the parameter

set the declared num to the pointer num

set the declared scope to the pointer scope

break the loop

otherwise, move to the next

when the loop ends, that means it has been found or not been found

So return num // it will be either -1 or the proper value