Project 4 Report

Nicole Ju

**Description:**

I used a hash table to store the data. I implemented my own hash table, with the identifier as the “key.” I created a struct called “decl” to represent each declaration; each decl stores the identifier string, line number, and scope. Each bucket in the hash table points to a list of declarations. My hash function uses the STL hash function for strings (in <function>). My hash table has an add function to add declarations, a find function to see if a certain identifier is in the hash table, and a remove function to remove the most recently added declaration with that identifier string. This structure allows for much faster search-ups and insertions (O(1)). For instance, the slow implementation of NameTable.cpp requires you to loop through the vector of identifiers until you reach the beginning of a scope, while with the faster version, checking to see if an identifier has already been declared in the same scope is close to constant time. The find function is also faster using a hash table because instead of looping through an entire vector of the identifiers, you can just plug the identifier into the hash function and check if it is in the list in that bucket (again, close to constant time).

In the NameTableImpl class, I added a member variable to keep track of the current scope number, a hash table, and a vector of the scope ids.

**Time Complexity:**

enterScope: constant time

exitScope: linear in the number of identifiers going out of scope (removing from the hash table is constant time)

declare: close to constant time (linear in the number of declarations in the bucket, but there should be a low load factor)

find: close to constant time (linear in the number of declarations in the bucket, but there should be a low load factor)

**Pseudocode:**

HashTable:

**bool** add(**const** string& id, **const** **int**& line, **const** **int**& scope)

find the bucket number using the hash function

create a new decl with the id, line, and scope

if the bucket points to a nullptr

have it point to a new list of declarations

add the new declaration to the list

return true

for each decl in the list at that bucket

if its id is the same as the id passed in

if its scope is not the same

break

return false

add the new declaration to the front of the list at that bucket

return true

**int** find(**const** string& id) **const** //if found, returns the line number of the most recently available decl

find the bucket number using the hash function

if the bucket points to a nullptr

return -1

for each decl in the list at that bucket

if its id is equal to the id passed in

return its line number

return -1

**void** remove(**const** string& id) //removes the most recently added item in that bucket

find the bucket number

if the list at that bucket has a size of 1

delete the list

set that bucket to nullptr

return

pop the first item off the list at that bucket

NameTableImpl

**void** NameTableImpl::enterScope()

increment the current scope

push empty string onto vector of ids

**bool** NameTableImpl::exitScope()

return false if the current scope is 0

while the last element in the vector of ids is not the empty string

remove the last element from the hash table

pop it off the vector

pop the empty string off the vector

decrement current scope

return true

**bool** NameTableImpl::declare(**const** string& id, **const** **int**& lineNum)

if id is empty string return false

call the hash table’s add function with the id, lineNum, and current scope

if false

return false

else

add the id to the vector of ids

return true

**int** NameTableImpl::find(**const** string& id) **const**

call the hash table’s find function

**Difficulties:**

I first implemented the name table with a binary tree; however, it was not fast enough since adding and searching are log time operations. A struggle I had with the hash table was figuring out how to exit a scope. I originally just decremented the current scope, but this leads to issues if you declare multiple scopes of the same level within another scope (there would seem to be multiple of the same identifiers in the same scope if their scopes are of the same level, even if they are in separate scopes). Thus, I had to remove the identifiers of a scope from the hash table when the scope was exited. Furthermore, I had a memory leak problem because I wasn’t removing the identifiers that were not in any scope (I called this scope 0), so I had to keep track of those identifiers and remove them separately in the destructor.