James Fu

105532210

7 June 2022

Project 4 Report - What’s in a Name?

(1) Algorithm and Data Structures

For the NameTableImpl, I wanted to create a closed hash table with a maximum number of 19997 buckets (a prime number was chosen to have a more even distribution and avoid collisions).

The closed hash table searches for duplicate or empty spots using an insert helper function, where it determines if there is a duplicate and if not, adds it to the hash table by pushing it to the front where there is an empty spot. There is a structure idNode that contains the id, line number, and scope of each node. The structure also contains a structure which allows us to easily add new nodes using an insert function. The hashTable was created using a vector of lists containing idNode objects. Additionally, a stack was used to store the scopes (in the original inefficient implementation, a vector was used but a stack was used to improve efficiency as you can easily add objects by pushing them to the stack or access/remove elements). A m\_scope function was added to keep track of the current scope and make updates within NameTableImpl.

As for algorithms, a mapFunc function converts the id into the actual value by using modulus. To code the mapFunc, I used the mapFunc discussed in Nachenberg’s lecture 14. For other algorithms, I added insert which gets the hash id using the mapping function and then determines if the item being implemented is a duplicate, and if it is not a duplicate, adds it to the front of the hashTable using push\_front (the constructor in the idNode struct allows us to add idNode objects easily). The declare uses the insert function recursively and adds the idNode to the hashTable, which is similar to the previous code but it saves the id and lineNum by creating an idNode object instead of adding to a list twice, which should be more efficient.

For enterScope(), the ids get pushed back to the m\_scp stack while the NameTable scope m\_scope is incremented by 1. This essentially does the same thing, but we don’t need a m\_lines list to keep track of the line number. For exitScope(), first we check if the scope is valid, and if not then we return false. Next, we get the id and pop the id after we access it, and determine if they have the same scope as the current scope, and if they do then we remove it from the Hashtable and decrement m\_scope.

Lastly, the find function has a rv which is set to -1 by default (which is the value that should be returned if no greater scope is found) and then a for loop is used to check if a scope greater than the current greatest scope is found, in which the max value is changed and the return value is set to where that max value is found.

(2) Time complexity of the NameTable functions

NameTableImpl::NameTableImpl() O(N)

enterScope O(1)

exitScope O(N)

declare O(1)

find O(N)

insert (helper function) O(1)

(3) Pseudocode for non-trivial algorithms

unsigned int mapFunc(const string& id) const {

store the string id as an integer (size\_t)

divide it by the hash size to get the actual id

insert

store the starting bucket number by calling mapFunc on the id provided

create an iterator

if slope or id same // check if the item being inserted is a duplicate

return false

add element if it is not a duplicate

return true

enterScope

push an empty string to the list of ids

increment scope

exitScope

if the scope is less than 0

return false // we know it is invalid

access the top element

remove the top element

while there are still ids available in the stack

check if it has the current scope and if it does

remove

exit

decrement the scope

declare

if the id provided is empty

return false

add the id to the list of ids (m\_scp)

insert the given id, linenum, and current scope into the hashTable

find

if the id provided is empty

return false

create a return value variable set to -1

create a temporary max variable

for each item in the hashtable

if greater scope is found with the same id

change max to the current scope

change return value to the current line number

return return value (where the greatest number was found)

(4) Known Bugs

There are no known bugs that I had. I had a lot of trouble getting the exitScope function to work and I ended up using a for and while loop, which would seem fairly inefficient but the time complexity was still not affected significantly since the while loop was based on if the top element was empty. I still believe that I could’ve improved the efficiency of this function, however. I also had a lot of trouble with the declare function and tried to use recursion, but I was able to get it to work and add non-duplicate elements to the hash table by using push\_front, all in constant time.

Another inefficiency was the constructor for the NameTableImpl, which initialized everything in the hashtable to 0 using a for loop. While this was technically still in O(N) time (the algorithm seemed to run fast enough most of the time) I couldn’t figure out a better way to initialize 19997 items.