**Project 4 Report Justin Chang**

**504732893**

**1. Algorithms and Data Structures Description**

The algorithms and data structures primarily in use for this project was using a closed Hash Table (data structure), without linear probing and a bucket number of 10007 (closest prime number to 10000). The hash function I used was from the #include<functional> library or basically one of the c++ std library string hashing functions. Additionally, each bucket was contained in a linked list (originally considered using a binary search tree such as AVL or Red Black Tree, but sets/unordered maps were unallowed containers, and debugging the data structure was too tedious, as well as not that significantly better than relying on a linked list which defaults worst case O(N) with an extremely small constant of proportionality). Another Data structure considered was the Huffman’s encoding binary search tree (inspired by professor Carey Nachenberg’s slides), but also mentioned in class, relying on a O(logN) v.s. an extremely small constant of proportionality closed Hash Table O(N) is not too far off. Especially since searching, deletion, and inserting by key in a hash table with a good hash function usually defaults to O(1), I decided to resort to using a Hash Table data structures.

Next, in terms of the program holistically, the only addition to my SymbolTableImpl class was a 1. myHashTable object, 2. A levelCounter, and 3. A stack of IDs, which holds a struct that contains a string id and int level. Keeping track of levels is important in knowing when to insert a particular item, or search for a particular item, and especially important in passing in the right strings to delete (which is kept in a stack for every successful declaration). Most of the work is done in a class I created in the same cpp file called HashTableSymbol.

In HashTableSymbol, it has a Boolean addValue(int line, string id) function, void deleteValue(string id) function, int searchValue(string id) function, bool isEmpty function, int hash(string id) function, and an int getLevel(string id) function. In it’s private data fields, it contains a struct that holds 1. String id. 2. Int level. 3. LineNumber, a fixed bucket array of linked list of type above struct, and a size counter. More details of how this was implemented will be in part 2 and 3 of the report when describing complexity of the functions and pseudocode, if it doesn’t make enough sense here.

deleteValue takes a string, hashes it, and uses a list iterator at the index hashed in the bucket array and searches for the same id to erase. Once it is found, it is erased, or if not found, returns and nothing happens (making sure level is maintained and passed into this function is taken care of in the implementation of SymbolTableImpl exitScope function).

addValue takes a line number and string id to consideration, but also does a search on the current level and makes sure no other declaration matches the same id in the bucket, when hashed and checked in the bucket list. It usually doesn’t and a object representing the struct described above is inserted into the table. However, in the case that there is a level falling out of order or a redeclaration on the same scope, this function returns false and doesn’t update the bucket list. This also reduces the work necessary on the SymbolTableImpl declare function.

searchValue takes a string id, hashes it, and runs a linked list bucket iterator through the list until it either finds the first declaration and returns it’s line number, or -1 if not found. Since it does all of these already as well, it reduces the work necessary on the SymbolTableImpl find function.

isEmpty just returns if the size is 0, helpful for edge cases and debugging.

Hash in my program utilizes the std library functional implementation (which is O(1) and effective in distributing strings), and also does a check to make sure the value to hash is a positive array. This could have been done by just declaring unsigned\_int but I decided to stay in more familiar territory by letting it be possibly negative at first, but does a check to make sure if its negative, that it gets its value flipped into positive.

Lastly, getLevel just takes a string id and hashes it, making sure to return the highest level usually on the hashed index. It follows a pattern much similar to searchValue when traversing the bucket list, but doesn’t return line number, and returns the level instead or -1 if not found.

**2. Complexity of Symbol Table Functions**

The complexity of each symbol table function in the implementation is all actually really efficient. Here is a breakdown of the details for each function.

**enterScope()** This function is O(1) and only does two things. It increments a private variable level count and a private object’s public variable level count.

**exitScope()** This function is O(1), or on worst case, O(1+1+N (super low, almost not noticeable constant of proportionality) +1)=O(N) with a very low constant of proportionality. This is because aside from the declarations and small function checks for size requirements, it does a pop of successfully declared IDs and attempts to delete these IDs by using the hashtable object function calls to delete those values by level. Once a level falls out of scope, the levelCount is dropped. Putting this all together, in a worst case scenario, it could be O(N) with a low constant of proportionality, when deleting values in bucket collisions in a linked list, so just have a big bucket size and good hash function to lower that value more. Additionally, in terms of AVERAGE case, this should still be O(N) with an even LOWER constant of proportionality assuming huge N for the same reason, with best case scenario of O(1).

**declare(string id, int lineNum)**  This function is O(1), or on worst case, O(N) with an extremely low constant of proportionality. This is primarily due to the fact that declare only calls on the hashtable’s addValue function which does a few O(1) boundary checks, then a O(N) search on a linked list bucket for verifying the level doesn’t overlap anything in the case of a redeclaration in a new scope. This O(N) is a lot more efficient and on average, though Is still O(N) in a much less noticeable constant of proportionality, does have a pretty high chance of a best case of O(1). If the sample size isn’t extremely big and the buckets are well utilized (spread out declarations well enough), then the O(N) is less noticeable. Additionally, the value is added to the front (which is O(1) for linked list).

**find(string id, int lineNum)** This function is O(1), or on worst case, O(N) with an extremely low constant of proportionality. This basically does what declare is doing (literally word for word since declare relies on find prior to an insert) and returns either true or false if found. Basically just omits the O(1) adding to the front of linked list step which means the find function still on average is O(N) with super small constant of proportionality, and a O(1) case if buckets are well spread and utilized.

**3. Pseudocode for Nontrivial Algorithms**

**Add Value**

If linecount is = line

Increment linecount

Else

While linecount is not line

Increment linecount

If searching for value is found,

If value’s level number matches greater or equal to current level

Return false.

Create a new bucket with current hashed id, marking line number and level

Append to bucket array the hashed index id to the front of the linked list.

Increment sizecounter for the hash table.

**Delete Value**

Hash the passed in string.

While the string in the bucket linked list doesn’t match the passed in string

Advance until it is matching, when it matches,

Erase the value from the list, decrement the size of the table,

Return.

**Search**

Hash passed in string

While the string in the bucket linked list doesn’t match the passed in string

Advance until it is matching, when it matches,

Return the line number of the matching string.

If string doesn’t exist in the list,

Return -1

**Get Level**

Hash passed in string

While the string in the bucket linked list doesn’t match the passed in string

Advance until it is matching, when it matches,

Return the level of the matching string.

If string doesn’t exist in the list,

Return -1

**Hash**

Use std::str\_hash on the string passed in (a c++ standard library hash function)

Index to be inserted in the bucket is set to that value, and modulo the size of the bucket array (10007)

Return that value.

**4. Known inefficiencies**

As of now, I’m pretty satisfied with the overall flow of the program. It does many things well correctly and handles very large sample sizes well and meets the spec requirements. However, there probably is still even a way to get it to improve just a tad more, since the spec also points out that their efficient implementation does the same job in 10-18 milliseconds (while mine on the best runs still get capped at around 22-27 milliseconds). Although not obvious, I think there are still a few search and insert checks I might be able to eliminate if I were able to take advantage of a few facts in each of those functions since I do a search on the declaration existing, and then search the level to make sure its higher, before making an insertion, which might be condensable to one step.