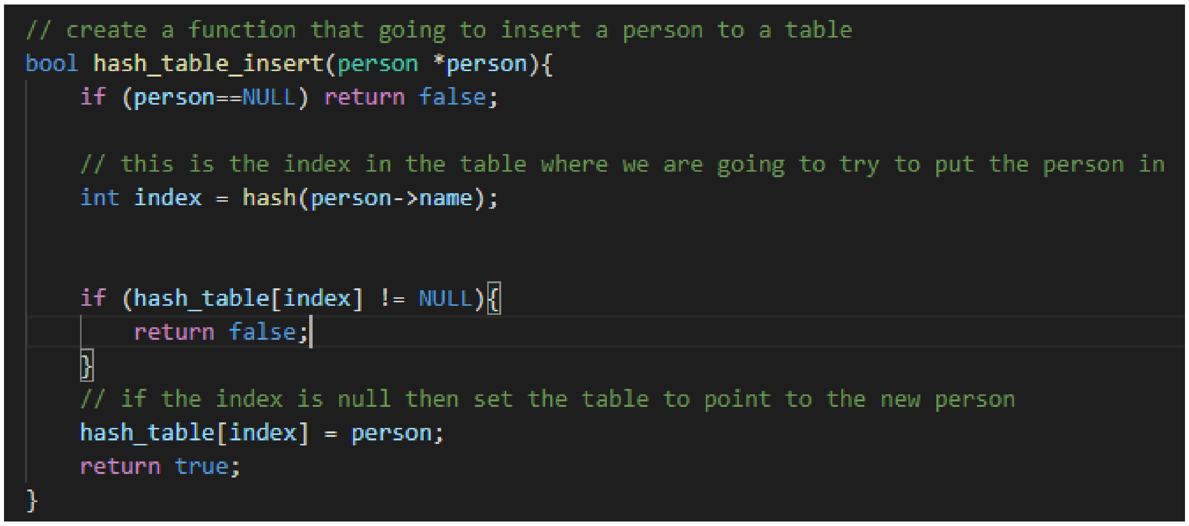
Part 4 – Written Exercises

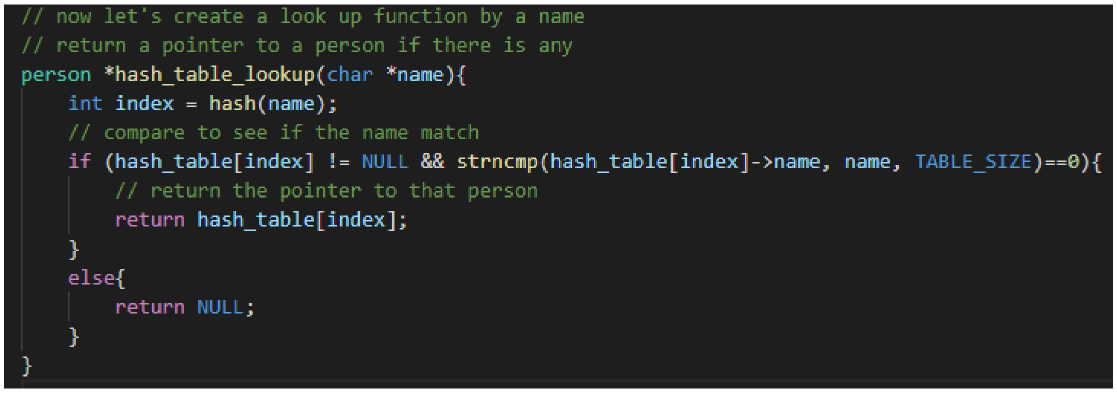
1. The provided hash function is not that good from my opinion. Even though it is good that we implement the chain linked list so that we don't have to worried about resizing the hash table which could lead to the time complexity of O(n), we still need to access the key-value inside of a node in a specific slot of the hash table. As we know, the cost of accessing the elements in the linked list is O(n) where n is the number of nodes. If we want to insert or removed the node in the end or insert or remove the node in the middle, the cost would also be O(n). The only thing that advantageous for us is if we insert or remove from the beginning of the node as the time complexity is O(1). Therefore, we can say that with this hash function, if the number of the nodes keeps growing, we still have a O(n) time complexity, and it is not as a good as if we are able to create algorithm that has the O(1) complexity.
2. Another way to implement the hash function would be to have a static hash table size and we don't have to do the chain linked list. We can set up the hash table by first set everything to null then we could insert the element in by checking if the hash_table index is null or not null. If it is null then insert the elements in the hash_table. We could also look up the elements by checking if the hash_table index is not null and if the strncmp comparing = 0. With this implementation, we don't have to loop through the nodes that is in one of the buckets. However, we will face the risk of resizing the hash table if the elements grow bigger and that would cause us the O(n).
   Example:



```
// create a function that going to insert a person to a table
bool hash_table_insert(person *person){
    if (person==NULL) return false;

    // this is the index in the table where we are going to try to put the person in
    int index = hash(person->name);


    if (hash_table[index] != NULL){
        return false;
    }
    // if the index is null then set the table to point to the new person
    hash_table[index] = person;
    return true;
}
```



```
// now let's create a look up function by a name
// return a pointer to a person if there is any
person *hash_table_lookup(char *name){
    int index = hash(name);
    // compare to see if the name match
    if (hash_table[index] != NULL && strncmp(hash_table[index]->name, name, TABLE_SIZE)==0){
        // return the pointer to that person
        return hash_table[index];
    }
    else{
        return NULL;
    }
}
```

3. If we resize the hashmap to add more buckets, the Big-Oh complexity is O(n) where n is the number of elements in the hash table. It is O(n) because we have to copy every element in the old hashmap and put it in the new table.
4. Open Addressing or close hashing is a method used to resolve the collision problems in hashing. With this method, all the keys are stored in table itself and the size of the table is greater than or equal to the number of keys that needs to be hashed. This collision resolution is resolved by probing or searching for the targeted key in the array until the key is found or an unused array slot is found which shows that there is no key the table. There are three types of probe sequences including linear probing, quadratic probing and double hashing. Open addressing is normally faster than chain hashing when the number of elements in the hash table is small compared to the length of the buckets (load factor). But once the load factor approaches 1, it gets slower.