CSCE 221 Cover Page
Programming Assignment #5
Bonus Due Date: November 27th, 11:59pm
Final Due Date: December 2nd, 11:59pm

First Name: Arvind Last Name: Venkatesan UIN: 827004034

Any assignment turned in without a fully completed cover page will receive ZERO POINTS.
Please list all below all sources (people, books, webpages, etc) consulted regarding this assignment:

CSCE 221 Students	Other People	Printed Material	Web Material (URL)	Other
1.	1. Lab TAs	1.	1.	1. Tyagi Notes
2.	2.	2.	2.	2.
3.	3.	3.	3.	3.
4.	4.	4.	4.	4.
5.	5.	5.	5.	5.

Recall that University Regulations, Section 42, define scholastic dishonesty to include acquiring answers from any unauthorized source, working with another person when not specifically permitted, observing the work of other students during any exam, providing answers when not specifically authorized to do so, informing any person of the contents of an exam prior to the exam, and failing to credit sources used. Disciplinary actions range from grade penalties to expulsion. Please consult the Aggie Honor System Office for additional information regarding academic misconduct – it is your responsibility to understand what constitutes academic misconduct and to ensure that you do not commit it.

I certify that I have listed above all the sources that I consulted regarding this assignment, and that I have not received nor given any assistance that is contrary to the letter or the spirit of the collaboration guidelines for this assignment.

Today's Date: 11/24/19

Printed Name (in lieu of a signature): Arvind Venkatesan

Hash Table Performance Test

Description:
For this programming assignment, you will implement **three different versions of hash table** and study their performance. You will implement (1) Chaining, (2) Linear Probing and (3) Double Hashing.

Your program will be a simple **word-counting application**. It reads a text file and creates a hash table, mapping each word to its number of occurrences. Your program should take in and process a .txt file as input (example dictionary.txt). The file **dictionary.txt** has one million words in which there are about 58,000 unique words, each occurring a different number of times. Once you have inserted all the words into a hash table, your program should output the number of occurrences for all given strings in the .txt file.

Your program should use an array-based implementation (vector in C++) for the hash table. The program should also implement a hash function that takes the ASCII values of each character within a word, multiplies them by a positive constant, adds them together, then compresses them with the modulo function (pseudocode below).

```
int hash = 0
int n = s.length     //s is the given word
for (int i = 0; i < n; i++) {
   hash = g * hash + ASCII value of char at position i in the word  //g is a positive constant chosen by you
}
index = hash % capacity // capacity is total memory size allocated for hash table
```

For the Double Hashing portion you should review and consider course materials and literature regarding double hashing and implement an efficient secondary hash function. Be sure to give this some critical thought.

Skeleton code is provided, except for a main.cpp which you will have to create yourself to assure all test cases pass. Please do not change function names , class names, or file names. You need to add variables to the classes in the .h files and fill in ALL the functions in ALL the .cpp files. For functions that already have return values in the .cpp files, delete them before filling them out - they are only there so that your code can successfully compile when those functions are not complete. Also, additional classes/structs can be added; just be sure to #include classes created in new files to appropriate .h files.

Coding Portion (80 Points):

- Create the three implementations of the different hash tables.

- Be sure to test the correctness of your algorithms and implementations (we will).
- Submit all your .h and .cpp files. We do not need the dictionary.txt.
- Your code will be graded based on whether or not it compiles, runs, produces the expected output, produces correct output, whether or not your experimental setup is correct, and your coding style (does the code follow proper indentation/style and comments).
- main() functionality:
    - Your main function should take in a file name as input either as command-line arguments or using cin.
    - It should then populate the three hash-tables appropriately.
    - Finally, it should print the contents of the hash tables into three separate files titled "Probing.txt", "DoubleHashing.txt", "Chaining.txt",
    - For grading purposes, we will be comparing the generated files against files containing the expected output.

Report(20 Points**)**

The purpose of this report is to motivate us to learn more about the performance of the hash map implementations.

What is expected:

- **Theoretical Statement**: State the time complexities for insert/removal operations (amortized and total).
- **Experimental Analysis**: Create graph(s) displaying the runtimes of **insert()** operations for all three implementations. Briefly compare the performances of the implementations based on your results.
- **Discussion**: Write a few sentences explaining your experiment and outcomes. For example:
    - Were your results what you expected? Did your experimental data deviate from theoretical runtimes, or were they similar? If they deviated, why?
    - Which implementation performs the best? Is there an implementation that performs better for all capacities, or are certain implementations more efficient at different capacities?

Bonus GitHub Opportunity (5 Points):

- Utilize GitHub to host your solution!
- Requirements:
    - Host a repository of your solution with your TAMU GitHub account.
    - Make sure your repository is **private**, as opposed to public!
    - Be sure to add your TAs as collaborators
        - Section 505: danwgun & kevinkuriachan
        - Section 506: foufou1560 & maitreyiramaswamy
        - Section 507: Josegerag & edgarmtz97
        - Section 508: theinen & shravankumaran
        - Other sections: yahuis & peixin94
- If you plan to host your code on GitHub, submit a link to your repository in eCampus instead of submitting a zip file.
- We will check your commit history. Do not just push all your code into the repository with one commit; commit your code semi-regularly.

Github

For github I realized that I didn't use my tamu github, but below is the link to my git repository with the lab data!

https://github.com/varvind/Project-5-CSCE-221

Theoretical Statement

In this experiment, we are testing the functionality of the hash table, and specifically three different implementations of an array-based hash table. The primary way a hash table is written is by using what is called a hash function, which maps a specific key to a bucket in the hash table. And of course, a hashing function could literally be anything. With, there are no perfect hashing functions that map everything to a unique key, which means there will inevitably be collisions which is handled by three main methods which we test in this lab, this being chaining, linear probing, and double hashing. Chaining handles collisions by basically keeping a list for a bucket as opposed to only having one item per bucket. Linear probing is another way which basically checks for collision, and if there is, it linearly moves through the indexes until there is an open space and places the item there. Finally, double hashing handles collisions by having a second hashing function which pushes the key to a different index.

Chaining

Chaining in the first of the three methods I implemented. For insert, it has a runtime complexity of O(1) for best, worst and average. This is because the mapping is always to a singular index, and it gets added to a list at that specific bucket. For removal the worst-case runtime is O(n), where little n is the number of keys. In theory, if the number of keys equals the capacity, then the complexity is O(N = n). The amortized complexity for removal is O(1) because it is relatively rare for something to be hashed to the same key, so removal is a simple O(1) operation.

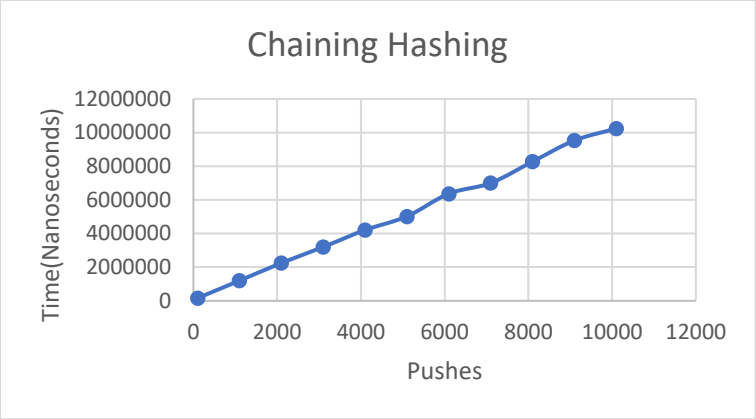
Linear Probing

Linear Probing is the first open addressing method implemented, which is slightly different than separate chaining. In open addressing, insert in the worst case is O(N), big N being the capacity (number of buckets) in the array. This is because if there is a collision, the linear probing algorithm could theoretically search the entire hash table to find an open space to fill because that is how the algorithm works. Although this is not realistic, and that is why the average runtime complexity is O(1) because the number of checks if there is a collision is low on average, on top of the fact that collisions are rare. For removal it is much of the same, being O(N), and O(1) on average, and this is because removal performs the same as insertion in terms of searching for the key using the hashing function.

Double Hashing

Double Hashing is the second open addressing method implemented and works like linear probing but not quite. Insertion and removal have the same runtime complexity as linear probing, this being O(N) for the worst case, and O(1) on average. Insertion is O(N) on average because the double hashing function could go through all index values until the capacity, N. This is the same case for removal, as the function could search for the value until the capacity, N. Although like mentioned before, this is not very realistic, and this leads to the complexity being O(1). It is not realistic because the hashing function for double hashing prevents N times until an open spot is available, which is the entire premise behind it.

Insert Function Analysis



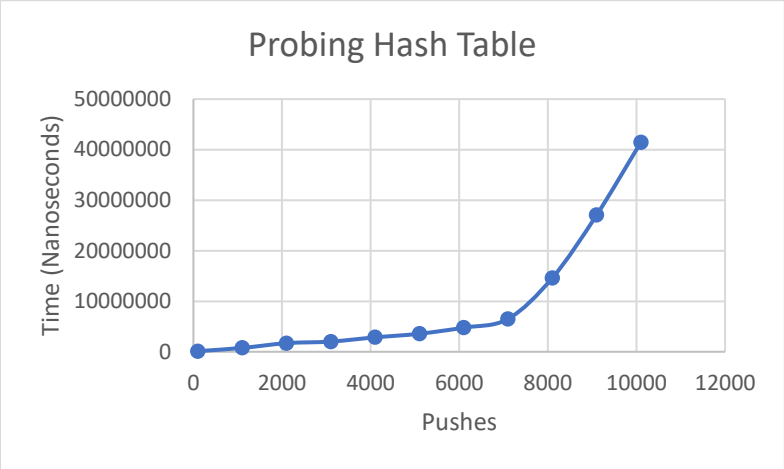
Based on the graph, it looks to be linear, which is not constant as mentioned above in the theoretical analysis. This is because there is a for loop that pushes n values into the heap, and that is why the graph is linear, although the true complexity is O(1). This coincides with the theoretical analysis above as pushing is a constant function in the case of chaining.

```
void ChainingHashTable::insert(std::string key, int val) {
    if (table[hash(key)].size() == 0) {
        pair tempPair;
        tempPair.key = key;
        tempPair.val = 1;

        table[hash(key)].push_back(tempPair);
    }
    else  {
        int index = hash(key);
        std::list<pair>::iterator it;
        for(it = table[index].begin(); it != table[index].end(); it++) { //loops
through to check and see if the key is already in the linked list
            if(it->key == key) {
                it->val = it->val + 1;
                return;
            }
        }
        pair tempPair;
        tempPair.key = key;
        tempPair.val = 1;
        table[index].push_back(tempPair);
    }
```

```
}
```

Based on the code segment, it is evident that only O(1) functions are used in inserting a value into the hash table using chaining.

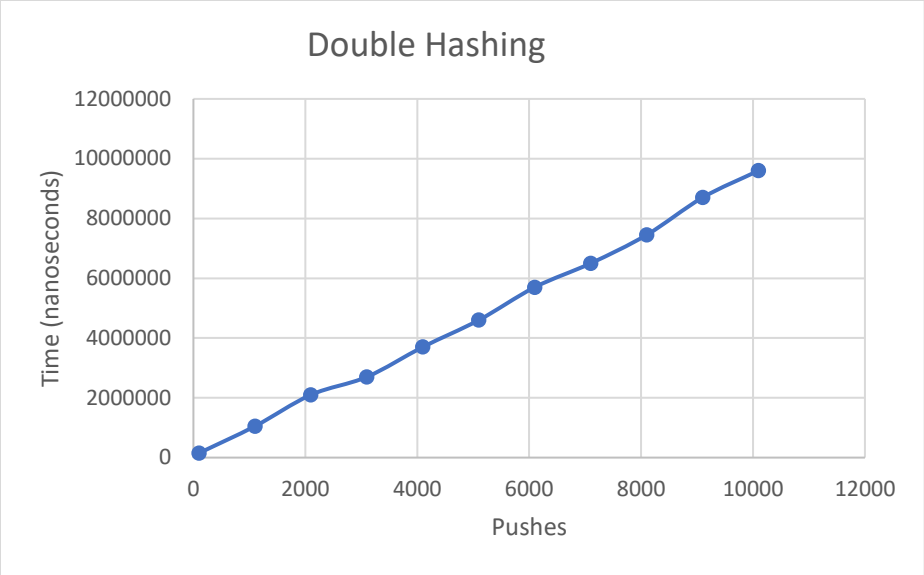


For linear probing the graph is very interesting, and honestly, it's a good thing how variable it is. For about the first 7000 pushes, it seems to be linear, which indicates O(1) complexity because as mentioned before, the for loop that is used to push a certain number of values adds to the complexity. However, from 7000+ there seems to be exponential growth. This can be explained because as the number of keys increases, it leads to more collisions based on how the linear probing algorithm works, which leads to a runtime complexity of O(N). The graph shows both instances, this being the average case from zero to 7000, and the worst case from 7000 to 10000. Therefore, I believe it is safe to say that the graph coincides with the theoretical analysis.

```
void ProbingHashTable::insert(std::string key, int val) {

    for(int i = 0; i < capacity; i++) {
        int j = (hashProbing(key) + i) % capacity;
        if(table[j].key == "") {
            table[j].key = key;
            table[j].val = 1;
            break;
        }
        if(table[j].key == key) {
            table[j].val = table[j].val + 1;
            break;
        }
    }
}
```

Based on the code above it is clear to see that for the worst case the complexity is in fact O(N), N being the capacity of the hash table (number of buckets). However for average cases where there are fewer collisions, the runtime complexity tends to be O(1).



According to the graph, it seems to be linear, representing a complexity of O(1) for double hashing, which we determined to be the average case complexity. This can be explained by the fact that unlike linear probing(the other open addressing function), collisions happen less often because the second hashing function makes the possibility less likely to happen. This explains why the graph shows the average case O(1).

```
void DoubleHashTable::insert(std::string key, int val) {
    for(int i = 0; i < capacity; i++) {
        int index = hash(key) + (i * (secondHash(key)));

        if(table[index].key == "") {
            pair temp;
            temp.key = key;
            temp.val = 1;
            table[index] = temp;
            break;
        }
        if(table[index].key == key) {
            table[index].val = table[index].val + 1;
            break;
        }
    }
}
```

Based on the code above, it shows a worst-case complexity to be O(N) based on how the algorithm is written, but in reality this isn't the case, and the graph proves it. Often a few runs of the for loop are required to handle collisions because of the nature of the hashing function, and this is why the graph represents O(1).

Results

Overall the results are accurate to the theoretical analysis made prior to testing. Each graph represented the function accurately, and it also displayed which functions performed better or worse. According to the runtime analysis, the worst algorithm seems to be linear probing, and this is because it runs into collisions more often, which is less than ideal as the data sets get bigger and bigger, and this is not just for insert, but also for removal. The best in terms of insertion according to the graphs is a tie between the double hashing and chaining, but from a practicality perspective, it makes more sense to use chaining which always runs insert in a constant time. Removal is the same way, both being very similar in theory, but again I believe the edge goes to chaining, and overall the best algorithm according to the data and analysis is chaining, although double hashing is a good choice as well.