

ANALYSIS OF ALGORITHMS

Project 3 Report

Ahmed Burak Gulhan
150160903

Compiling and Running

To compile use the code: `g++ ./main.cpp ./hash_table.cpp -o output.o`

To run the program use: `./output.o vocab.txt search.txt`

Explanation of the code

In this program a class called hash_table is used.

```
4  class Hash_table{
5      string *table;
6      int hash_size = 0;
7      int array_size = 0;
8      int collisions = 0; ///total collisions when inserting
9      int search_collisions = 0; /// total collisions when searching
10     int p; ///random prime
11     int a[4]; ///used for universal hashing
12 public:
13     Hash_table(int size); ///read from file and initialize hash table
14     ~Hash_table();
15     //int get_array_size();
16     int get_array_size();
17     int get_hash_size();
18     int get_collisions();
19     int get_p();
20     int get_a(int);
21     int get_search_collisions();
22     void linear_probe(int line_number, string word);
23     void double_hash(int line_number, string word);
24     void universal_hash(int line_number, string word);
25     void print(int = -1);
26     int search_linear_probe(int line_number, string word);
27     int search_double_hash(int line_number, string word);
28     int search_universal_hash(int line_number, string word);
29
30 };
```

This class contains the variables:

- `*string table` , stores hash table
- `int hash_size` , keeps track of how many items are added to hash table
- `int array_size`, the total size of the hash table (how many items it can hold)
- `int collisions` , keeps track of collisions during insertion
- `int search_collisions`, keeps track of collisions during insertion
- `int p` , stores a random prime number smaller than `array_size`, this is different for every `hash_table` instance. Used for double hashing
- `int a[4]`, each item of this list contains a random number smaller than `array_size`. This is used for universal hashing.

Important methods in this class:

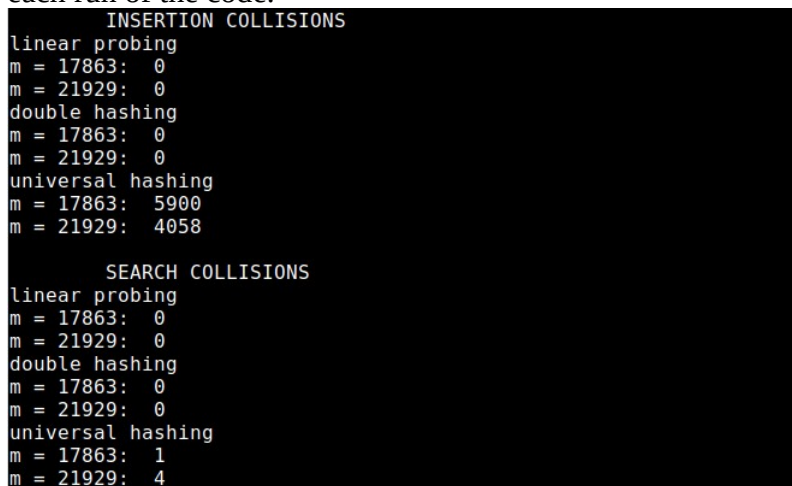
- `linear_probe()`, `double_hash()`, `universal_hash()` these methods use the hashing methods, linear probing, double hashing and universal hashing respectively to add an item to the stack. These methods require a line number (from `vocab.txt`) which is used in the hashing function, and a string which is added to the respective hash table index. These methods are implemented in the way shown in the homework instructions. Universal hashing uses double hashing as it's open addressing strategy.
- `search_linear_hash()`, `search_double_hash()`, `search_universal_hash()` these methods are used for finding the location in the hash table. These functions require a `line_number` and a string of the word we are searching as inputs. The way these methods work is that first using the key value they calculate the hash index. If this hash index contains the word then the search is finished. If not there must have been a collision when adding this word to the hash table, therefore we increment 'i' and calculate the hash index until the word is found.

In the main function we initialize 6 different hash tables. With these hash tables we insert all items in “vocab.txt” using different hashing methods. Then using the words in “search.txt” we search these words in each hash table. Then we print the results to the command line. While searching for the words in “search.txt” we do not know the line numbers for these words. So using the function find_line_number(), we search for the word in “vocab.txt” and find the line number. This searching operation has $O(n)$ complexity. So if we do not know the line_number for the word we are searching, the search (in the hash table) takes $O(n)$ time. However if we do know the line_number, then the search (in the hash table) takes $O(1)$ time.

```
int find_line_number(string search_word, string vocab_file){
    ifstream infile(vocab_file);
    string word;
    int i=0; //line counter
    while (infile >> word){ //read file
        if(word == search_word){
            return i;}
        i++;
    }
    cout << "word: " << search_word << " not found" << endl;
    return -1;
}
```

Collisions during insertion and search operations:

The output of this program displays the insertion and search collisions for array sizes, 17863 and 21929, as shown below. Since some values are generated randomly, the output will be different for each run of the code.



```
INSERTION COLLISIONS
linear probing
m = 17863: 0
m = 21929: 0
double hashing
m = 17863: 0
m = 21929: 0
universal hashing
m = 17863: 5900
m = 21929: 4058

SEARCH COLLISIONS
linear probing
m = 17863: 0
m = 21929: 0
double hashing
m = 17863: 0
m = 21929: 0
universal hashing
m = 17863: 1
m = 21929: 4
```

The reason for linear probing and double hashing having 0 collisions is because we use line numbers to hash our words. Since each line number is unique, sequential and starts from 0, the only way to get a collision is by having the line number bigger than our array size. However since line numbers start from 0, when the line number is larger than the array size, our hash table will already be full. Therefore having collisions is impossible when hashing with line numbers in linear probing and double hashing.

The reason for universal hashing having collisions is that universal hashing is random (the array is randomly set). Therefore even though line numbers are sequential and start from 0, there can still be collisions depending on our random variables. On average the table with the smaller size will have more collisions.

For searching collisions, linear probing and double hashing have 0 collisions, since there are no collisions when adding. For universal hashing, there will be search collisions if the words we are searching have collisions when being added.