BLG335E - Analysis of Algorithms

Mareva Zenelaj

150150906

Project 3 – Report

In this project we were asked to read words from *vocab.txt* and save them in a hash table by generating a key in three different ways: linear, double and universal. After inputting the words, *search.txt* was opened and each word is read and then searched in the hash table. The search function returns the place where the word is located in the hash table.

Linear probing:

```
int HashTable::linear_probing(int key){
    int i = 0;
    int hash_key;

    hash_key = (key+i) % capacity;
    while(hashTable[hash_key] != NULL){
        i++;
        collisions++;
        hash_key = (key+i) % capacity;
    }
    return hash_key;
}
```

The hashing function: $h(k,i) = (k+i) \ mod \ m$ and $i \in [0, m-1]$;

Here for every number of times the hash_key is occupied there's a collision. The total number of collisions during insertion is saved in *collisions*.

This function is called in *insert_set* and then used to input the new value and key to the hash table.

```
hashTable[hash_key] = new HashNode(key, word);
```

In order to write insert_set only once, I used a switch key to determine which type of hashing function would be used to insert the set. The latter is determined by the parameter that is passed in the main function where l stands for linear, d for double and u for universal.

Double Probing:

```
int HashTable::double_hashing(int key){
    int i = 0;
    int p = 7;
    int hash_key;
    int h1, h2;
    h1 = key % capacity;
    h2 = p - (key % p);
    hash_key = (h1+i*h2) % capacity;
    while(hashTable[hash_key] != NULL){
        i++;
        collisions++;
        hash_key = (h1+i*h2) % capacity;
    }
    return hash_key;
}
```

The hashing function:

*h(k, i) = (h1(k) + i * h2(k)) mod m and*
$i \in [0, m-1]$;
h1(k) = k mod m;
h2(k) = p – (k mod p), p is a prime number and
$p \in [0, m-1]$;

Here p is chosen to be 7 arbitrarily.
Again the number of collisions is saved in *collisions*.

Universal probing:

```
int HashTable::universal_hashing(int key){
    int r = 3;
    int k_array[3] = {};
    divide_number(key, k_array);
    int hash_key = 0;
    fill_a();
    for(int j = 0; j < r; j++){
        hash_key += k_array[j]*a[j];
    }
    hash_key = hash_key % capacity;
    while(hashTable[hash_key] != NULL){
        hash_key++;
        collisions++;
    }
    return hash_key;
}
```

The hashing function:

$$h_a(k) = \sum_{i=0}^{r} a_i k_i \mod m$$

If there are collisions, then there's a linear approach followed to find the empty slot.

While inserting the words in the hashing table and generating keys from the line number of the words, I saved the words and the line numbers in a map in order to have access to the line numbers of the words lately while searching.

The latter was done as shown in the picture below where the cursor is.

```
if(file_vocab){
    while(counter < m){
    getline(file_vocab, str_);
    mymap[str_] = counter;
    table->insert_set(counter, str_, 'l');
    counter++; // counter is the key
    }
}
```

After all the words are inserted in the hash table in the preferred way, now searching should be done. *search.txt* is read and each word is firstly searched in the map declared as mymap and then the line number is retrieved and send as a parameter to one of the functions: *search_linear, search_double* and *search_universal* as preferred.

search_linear:

```
int HashTable::search_linear(string word, int line_number){
    int key;
    int k;
    if(strcmp(hashTable[line_number]->value.c_str(), word.c_str()) == 0){
        key = line_number;
    }
    else{
        k = line_number + 1;
        while(k < capacity){
            if(!hashTable[k])
                continue;
            if(strcmp(hashTable[k]->value.c_str(), word.c_str())==0){
                key = k;
                break;
            }
            k++;
            collisions_search++;
            }
    }
    return key;
```

First it's checked if the word is in the current line number and if not the required word is searched linearly until found.

The other search functions follow a similar pattern.

	Insertion		
	Linear	**Double**	**Universal**
M = 17863	**0**	**0**	**942811**
M = 21929	**0**	**0**	**2244675**

	Search		
	Linear	**Double**	**Universal**
M = 17863	**0**	**0**	**942811**
M = 21929	**0**	**0**	**2244675**

The tables above show the number of collisions that are seen during insertion and searching of the words. The reason why linear and double have 0 collisions both while searching and inserting is merely because the line number is in an increasing order and consequently so is the *hash_key*. Meanwhile in the universal probing there is definitely a lot of collisions since the insertions is done uniformly and random numbers are used to generate the hash key.