https://github.com/the-coding-cloud/FLCD/tree/main/Lab2 - GitHub repository

Lab 2 – Documentation

Purpose

The purpose of this lab was to implement a data structure that can be used to represent a Symbol Table (ST). I decided to implement a hashtable, with open addressing (using double hashing) as a collision resolution.

Implementation

I used Python during this lab and used basic Python lists in order to implement a hashtable.

My representation consists of lists of lists initialized with a single element ("*empty" is an invalid identifier or constant). I assumed that this hashtable will NOT need to be resized.

The hashcode function computes the hashcode based on the sum of the ASCII codes of each character from the identifier/constant.

Collision resolution

One important aspect to be taken into consideration when implementing a hashtable is the situation of collision, that occurs when two or more values are mapped by the hash function to the same location. Collision resolution can be obtained in two ways, through chaining or open addressing.

Separate chaining involves additional data structures and possibly more linear searching; each slot from the hash table contains a linked list, with the elements that hash to that slot, therefore we may have an efficiency issue when multiple elements hash to the same slot. *This is one of the reasons I chose not to use a separate chaining hashtable.* Its main advantage, though, would be that we can insert more elements than the number of slots in the hash table itself.

Open addressing uses all the space provided by the hashtable and every element of the hash table is inside the table, there are no links, no external structures (therefore, no extra memory). **The main idea of the open addressing technique is to successively generate positions for the element, check (probe) the generated position and add it to the first empty one.** There are several ways to implement this:

- Linear probing:
  - The probing sequence (= the sequence of potential positions) for an element x would be generated by the formula: $(h(x) + i)\%t$, where h(x) is a simple hash function, t is the hashtable capacity and $i \in \{0, ..., t-1\}$
  - Disadvantage: there are only t distinct position sequences; therefore, 2 elements that are hashed to the same initial position will have the same sequence of positions
- Quadratic probing:
  - The probing sequence for x would be generated by the formula: $(h(k) + c_1 * i + c_2 * i^2)\%t$, where h(x) is a simple hash function, t is the hashtable capacity, c_1 and c_2 are constants initialized when the hash function is initialized (c_2 should not be 0) and $i \in \{0, ..., t-1\}$
  - Disadvantage: there are, again, only t distinct position sequences; therefore, 2 elements that are hashed to the same initial position will have the same position sequence
- Double hashing:
  - As the name suggests, the probing sequence for an element x would be generated by a formula that uses ***two hash functions***: $(h'(x) + i*h''(x))\%t$, where h'(x) and h"(x) are simple hash functions, t is the hashtable capacity and $i \in \{0, ..., t-1\}$
  - The first position for an element x corresponds to (h'(x) + 0*h"(x))%t = h'(x)%t (only the first simple hash function is used here); the rest of the positions in the probing sequence will be computed based on the second hash function, h"(x).
  - *What is the main advantage?* Even if h'(x1)=h'(x2), the probing sequence will be different if x1≠x2

- In order to produce a permutation of all the possible positions in the hashtable, the hashtable capacity and all the values of h”(x) have to be relatively primes. This can be achieved by choosing a prime number for the hashtable capacity and design h” in such a way that it always returns a value from the {0, t-1} set

I decided to implement the hashtable using the ***open addressing with double hashing technique.***

I have chosen the following functions:

- h’(x) = x%t, where t is the hashtable capacity (division hashing)
- h’’(x) = 1 + (x%(t-1))

So, the probing sequence can be generated in a for loop, as it follows:

```
for i in range(hashtable_capacity):
        position = (h1(x) + i*h2(x))%hashtable_capacity
        if hashtable[position] == “empty”:
                # add element
```

However, in my implementation, I have only one *extended* hash function, that receives as parameters both the element and the index in the probing sequence (so i=0 means that we are generating the first postion in the probing sequence) and directly maps the element by using the two simple hash functions:

```
h(x,i) = (h'(x) + i*h''(x)) % hashtable_capacity,      where i = 0, ..., hashtable_capacity-1
```

and the code above becomes:

```
for i in range(hashtable_capacity):
        position = hashFunction(x, i)
        if hashtable[position] == “empty”:
                # add element
```

The principle stays the same: we will successively examine the positions h(x,0), h(x,1), ..., h(x, t-1). An element will be added on the first empty position in the position sequence that can be generated based on the hashing function above. The default hash table capacity was chosen to be a prime number.

For clarification, I have added another file on GitHub that implements the double hashing strategy using 2 separate hashing functions (instead of an extended hash function).

To sum it up, open addressing with double hashing:

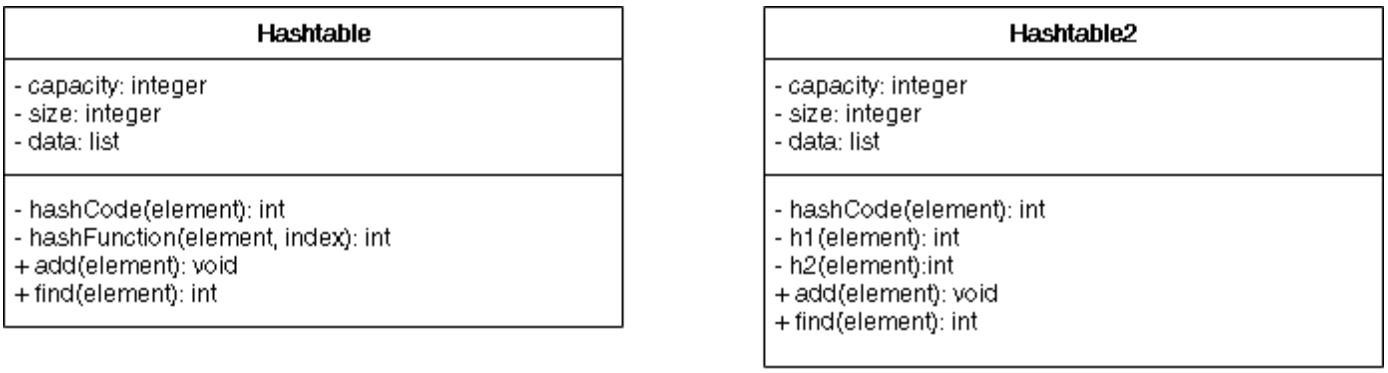
- does not require additional space besides the hash table
- collisions are solved by placing the element in the first available possible in the hashtable
- the possible positions (probing sequence) are generated successively by using a second hash function
- the main advantage of double hashing is that if h(k1)=h(k2), but k1 and k2 are different, their probing sequences will be different, unlike in linear probing or quadratic probing

Interface

The operations that can be performed and are needed for this hashtable are:

- hashCode(element):
    - Input: *element*
    - Output: *hashcode*
    - Preconditions: *element* – string
    - Postconditions: *hashcode* – integer representing the sum of ASCII codes for each character in *element*
- hashFunction(element, index):
    - Input: *element, index*
    - Output: *position*
    - Preconditions: *element* – string, *index* – integer
    - Postconditions: *position* – integer
- add(element):
    - Input: *element*
    - Output: no output
    - Preconditions: *element* - string
    - Postconditions: the element will be added to the hashtable after the execution
    - Takes the element we want to add to the ST and adds it to the table by successively generating positions for this element (based on the hash function) and looking for the first empty position to place the element in
    - Complexities:
        - Best Case: theta(1)
        - Average Case: theta(1)
        - Worst Case: theta(m)
        - Overall: O(m)
- find(element):
    - Input: *element*
    - Output: *position*
    - Preconditions: *element* - string
    - Postconditions: *position* – integer representing the index of the element in the hashtable, can take values from {-1}U{0,1,…,hashtable_capacity-1}
    - If the elements exists, it returns the position in the hashtable of the element we are searching for; otherwise it returns -1
    - The search is based on a similar strategy with the one from the add function
    - Complexities:
        - Best Case: theta(1)
        - Average Case: theta(1)
        - Worst Case: theta(m)
        - Overall: O(m)

UML Diagram



Hashtable is the class implemented in *main.py*. It uses the "extended" hash function, that takes as parameter the index as well.

Hashtable2 is the class implemented in *hashtable2.py*. It is implemented in a manner that uses two separate hash functions.

References:

Data Structures and Algorithms Course, Year I

https://en.wikipedia.org/wiki/Double_hashing