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 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/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.

The hash function is based on open addressing with double hashing strategy:

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

where for an element x, we will successively examine the positions h(x,0), h(x,1), ..., h(x, hashtable_capacity-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. 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, hashtable_capacity} set. In my case, h'(x) = x%hashtable_capacity and h"(x) = 1 + (x%(hashtable_capacity - 1)).

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)

(References: Data Structures and Algorithms Course, Year I)