CMSC 123: Data Structures

1st Semester AY 2020-2021

Prepared by: CE Poserio & KBP Pelaez. Updated by: ZO Arnejo

Lab 11: HashTable ADT

HashTable ADT is an unordered collection of data items, which can be used to implement *insertion*, *deletion*, and *find* operations in constant average time. Since it is *unordered*, *find_min*, *find_max*, *sort*, and other operations that need ordering information are not efficiently implemented. Hash tables are similar to `Python`'s *dictionary*. In this discussion, we delve deeper into its data structure and implementation.

Hashing

Hashing is the use of a data structure called a **hash table**[1], simply, an array of some fixed size, to store data identified by some *key*. The *key*[2] is then mapped to an integer (in the range of valid array indices), which serves as the *index* or position of the data in the array. By having a key assigned to each data in a hash table, hashing allows to find an item, on the average, in *O(1)* time.

In arrays, we use the array index to access a certain element; in hash tables, we use the key of the data we want to store *to compute* for the index. This mapping from key to index is done using a **hash function**. A hash function should be easy to compute. We want to implement abovementioned operations in $O(1)$ time, thus, the index, the most important value needed to access a position in the array, must be easy to find. Moreover, a hash function must be ensured to *hash* or map each key to distinct keys so that different data items are placed in different positions. However, this cannot be done since in most cases, the number of keys (or data items) are very much greater than the number of valid indices (or positions) in the array. Thus, we need to find a hash function that maps keys evenly among valid positions. The process of hashing described earlier is shown in Figure 1.

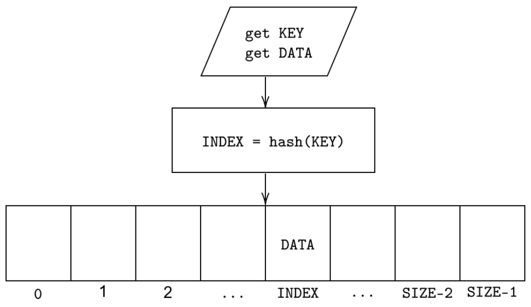

Figure 1: The hashing process, where `DATA` is the data to be stored/found, `KEY` is the key associated with the data, and `SIZE` is the size of the hash table.

Furthermore, to create the `HashTable ADT`, the following must be considered:

- creating a hash function that is easy to compute yet evenly distributes keys among cells; and
- deciding what to do when two keys are mapped in the same index (called *collision*).

[1] *HashTable* is the name of the ADT and *hash table* is the data structure used to implement it. Do not be confused.

[2] The *key* is usually a string derived from the data itself so that the key need not be stored *e.g.* if the data contains name and student number, the key could be the concatenation of both fields as in DATA $= [Juan, 201912345]$; KEY $= juan201912345$.

Hash Function

A hash function "converts" a *key* to an integer. Note that the *key* could be of any type. The hash function must be selected carefully considering the properties (*e.g.* type, values, etc.) of the key. Here are some known hashing methods:

Modulo

In the case that the key is an integer[3], a fairly simple hash function could be `hash(key) = key % SIZE`. This could be a good hash function since the range of this function is $[0, \text{SIZE} - 1)$, which is the range also of valid indices. However, if SIZE = 10 and every key ends with a zero, then every data is mapped to index 0. Even though the function is easy to compute, it fails to distribute the keys to other available positions in the hash table. Many studies suggest that to avoid this, `SIZE` must be prime. This method is usually preferred when not much is known about the key.

Folding

This method divides the key into several parts, combines or folds the pieces together using a simple operation *e.g.* addition, and then, finally transforms the result to an integer. For example, if the key is your student number, it can be split into two parts: the first four digits and the last five digits, then add these parts to get the result integer. If this integer is larger than the table size, *modulo size* can be applied.

Radix Transformation

Radix transformation converts the key to another number system and uses the result for the position (*modulo size* is also applied to ensure a valid result). For example, given the student number 202012345_{10}, it can be converted using 9-system which results to 87164627_9. Then use `87164627 % SIZE` as the table index.

Hashing methods are not limited to what was described above.

Collision Resolution

Collision happens when two keys are hashed into same values. The two simplest collision resolution techniques are discussed in the following sections.

Open Hashing

The first commonly used resolution method is open hashing also known as *chaining*. In this method, a list of all data elements that are hashed in the same position is maintained. In other words, an array of lists[4] is prepared and each data element is inserted to the list which corresponds to the index produced by the hash function.

To find a data element, the key is hashed to identify which list contains the data element. Any search algorithm can be used to search the data element from this list. An illustration of a sample hash table is shown in Figure 2.

[3]Keys are commonly strings. Some preprocessing can be done to transform a string to an integer. Furthermore, if keys are neither an integer nor a string, it's trivial to convert any object to a string, then continue as how strings are processed.

[4]can be implemented using an array of linked-lists

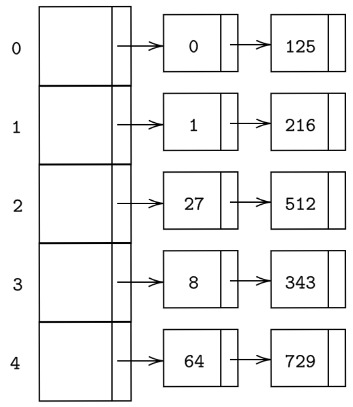

Figure 2: Open hashing demo: keys used are the first ten cubes, hash table size is five, and hash function is `key%5`

Closed Hashing

Instead of maintaining an array of lists, closed hashing, also known as *open addressing*, resolves collisions by finding the next empty cell (called probing) in the hash table. If the position $h(k)$ is occupied, then $(h(k) + p(1))\%\text{SIZE}$ is tried. If it is occupied, $(h(k) + p(2))\%\text{SIZE}$ is tried, then $(h(k) + p(3))\%\text{SIZE}$, and so on, where `h` is the hashing function, `k` is the key, and `p` is the probing function. The probing function `p` is the collision resolution strategy. If no available cell can be found, then the table is full. There are two common probing functions: linear and quadratic.

Linear Probing

In linear probing, `p` is linear and prominently, $p(i) = i$. That is, after a collision, the next adjacent cell is tried until an empty cell is found; after `i` collisions, the position $h(k) + i$ is tried.

Quadratic Probing

Quadratic probing, obviously has a quadratic probing function, usually $p(i) = i^2$. In this strategy, the next cell being tried is a factor of two cells away from the previous tested cell.

Double Hashing

Another strategy involves using another hashing function, h_2; that is, $p(i) = i \cdot h_2(k)$.

To summarize, closed hashing uses the hashing function $\text{hash}(k) = (h(k) + p(i))\%\text{SIZE}$. The advantages and disadvantages of the probing functions above are left as a reading assignment.

Rehashing

As keys are added onto the hash table, the frequency of collision increases. In order to solve that, *load factor* and *rehashing* is employed.

Load factor is a measure that limits the capacity of a hash table. Example, our load factor is 0.7 and the table size of the hash table is 10. Given the load factor, it emplies that the total amount of keys that can be stored in the hash table before it is treated FULL is $capacity = ceiling(tableSize * loadFactor)$.

The idea is, once the load factor is reached, the hash table is rehashed first before insert operation is done on the hash table. Rehashing means creating a new table with size larger than the current hash table. One good size is the smallest prime number given twice the size of the current hash table size. Example is, given the current hash table size = 7, the next hash table size is 17. Once the new hash table is created, all the stored keys in the old hash table is hashed to the new hash table one by one. Once all the keys are already hashed, the old hash table is disregarded and the new one is used for all succeeding operations.

References

Cormen, Thomas H., Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.

Weiss, Mark Allen. *Data Structures and Algorithms*. Benjamin/Cummings.