COMP20230: Data Structures & Algorithms **2018-19 Semester 2**

Lecture 12: Hash Tables

Lecturer: Dr. Andrew Hines *Scribes: Karim Elgendy - Francesco Ensoli*

Note: *LaTeX template courtesy of UC Berkeley EECS dept.*

12.1 Outline

This session introduces Hash Tables. It shows the basic way of implementing this data structure, taking a look at it's inner working, while explaining the core concepts to use such method.
For anyone with a bit of programming experience, that doesn't know of hash tables yet, chances are that they are already using them without knowing. In other programming languagges they are commonly refered to associative arrays *PHP*, or hashes *Pearl*, or dictionaries *Phyton*.

12.1.1 What is a Hash Table?

In computing, a Hash table (hash map) is a data structures used to implement an associative array, which is a structure that allows the mapping of keys to values. A hash table uses a hash function to compute an index into an array of slots (or Buckets), from which the correct value can be found. Furthermore, it allows for the mapping of multiple values to a single key, strings are permitted as keys.

Type — Unordered associative array Invented — 1953

Time Complexity	Average	Worst case
Space	O(n)[1]	O(n)
Search	O(1)	O(n)
Insert	O(1)	(1)
Delete	(1)	(1)

In the majority of cases hash tables turn out to be more efficient than any other table lookup structure (e.g. search trees). As a result of their efficiency, they are widely used in many kinds of computer software, specially for associative arrays, database indexing, caches and sets.

To achieve a good hash table performance, generally the keys have to be well distributed in the range of the array *0* to *N 1*. In fact a basic requirement is that a function should provide a uniform distribution of hash values. A non-uniform distribution increases the number of collisions and with it the cost of resolving them.

What is a hash function and how does it work?

The hash function is a function which computes the index appropriate for the key based on two components:

- The hash code:
  - The portion of the function which translates the string representation of the key into an integer. There are various different implementations of how the hash code is generated. For simplicitys sake, the hash code is generated with the *hash()* function in python.

- The compression function:
  - The portion of the function which compresses the size of the integer using the size of the array so it is within range of the indices of the array.
  - One method of compressing the hash code is called *the division method*. Its formula is straightforward: $i \% N$

    Where i is your integer hash and N is the size of your array

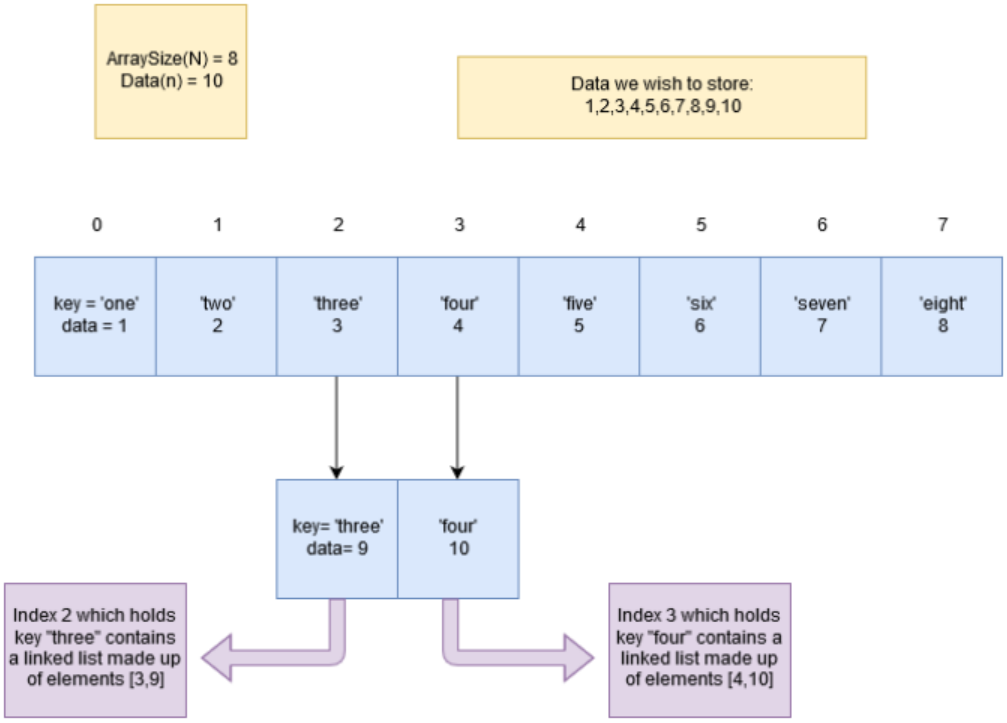
When hashing a random subset of a large set of possible keys, hash collisions are practically unavoidable. Approximately there is a 50% chance that at least two of the keys being hashed to the same slot according to the birthday paradox. Consequently a number of collision resolution strategies have been implemented to handle such occurrences. The number of entries *(K)* divided by the number of slots *(N)* is the load factor (K/N).

12.2 Collision resolution

Collision resolution is required because hash functions can provide two keys pointing to the same index. Collision resolution methods are ways to handle this.
Why do collisions happen in the first place?
If we provide dynamic hash tables, we could have infinite keys for infinite indexes. This would be considered memory hungry. Allowing two distinct keys to hash the same index saves us from this memory hungry scenario. However, we do claim that a hash function is good when it minimises the number of collisions that occur, is fast and easy to compute.

12.2.1 How do we manage collisions? Two primary methods.

Separate Chaining



- We have an array that is smaller than the amount of data that we have. For each node in the array, arr I is a linked list. This prevents the need for expanding the original size of the array to store additional elements. It also means that when performing a search, you only need to search the *ith* element.

- If we use a good hashing function, the expectation is that the size of each linked list would be n/N. The result of this is called the load factor of the hash table. We want this number bounded below 1 (meaning that the *O(n)* of the hash table would be *O(1)*).

- If N is too large, there are too many empty chains. If N is too small, the chains are too large. For constant time operations, N should be roughly about n/4.

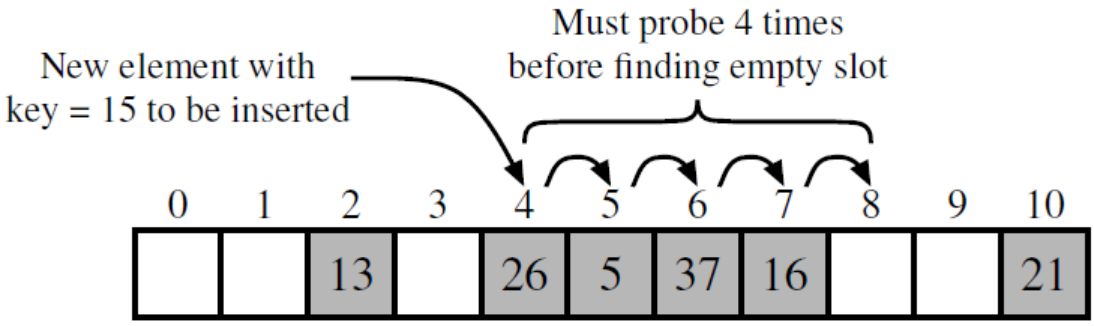
- If the array / number of linked lists ¿= 8, you should double the array size.

- If the array / number of linked lists ¡= 2, you should halve the array size.

- Keys need to be rehashed after performing such operations.

Open Addressing / Linear probing

- Linear probing is quite simple. It involves mapping a key to an index of integer in *0 - N-1*. If the index is already occupied, we continue probing i +1, i +2 etc until we find an available index.
- The array size N must be greater than the number of key-value pairs (n).
- If the operation continues to the end of the array and loops back, it means that the array is full and that the addition operation will fail
- If the the number of key:value pairs *(n)* / the size of the array *(N)* is ¡= , you should double the size of the array.
- If the the number of key:value pairs *(n)* / the size of the array *(N)* is ¿= , you should halve the size of the array.
- Re-hashing is required after the deletion of a key. For example (using diagram below), if we insert key 15 into an array and it takes 4 linear probes to find an open cell for our key -¿ we know that key 15 is at i+4. However, if we now delete the key found at i+2 and didnt re-hash everything to the right of the deletion, we would now have an empty cell before we find key 15. Our search for key 15 would fail (Goodrich, Tamassia and Goldwasser 2015).



12.3 Advantages

The main advantage of hash tables over different table data structures, is speed. This advantage becomes more apparent as the number of entries increases. Moreover when the number of entries can be predicted in advanced, meaning that the key value pairs is fixed and no insertion or deletion is allowed, that the bucket array can be allocated once for the optimum size, without the need to be ever resized.

12.4 Drawbacks

On average, operation on a hash table take constant time. If the table uses dynamic resizing, although the average cost per operation is small and constant, an insertion or deletion operation may occasionally take time proportionally to the number of entries, which can cause issues for real-time or interactive applications. Hash tables causes access pattern to jump around, being distributed seemingly at random in memory, which can trigger microprocessor cache misses, eventually generating extra delays. In case of many collisions, this method becomes quite inefficient *(e.g. a denial of service attack)*.

12.4.1 References

- http://netlibrary.net/articles/eng/Hash_table
- http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html
- MIT n.d. http://www.cs.rmit.edu.au/online/blackboard/chapter/05/documents/contribute/chapter/05/linear-probing.html
- Goodrich M., Tamassia R., Goldwasser M. Data Structures and Algorithms in Python (2015)