Hashing

Hashing: Large collections of data, for example customer records in a database, need to be accessible very quickly without having to look through all the records. This can be done by holding an index of the physical address on the file where the data is held.

Hashing algorithm is applied to the value in the key field of each record to transform it into an address. One common hashing algorithm is to divide the key by the number of available addresses and take the remainder as the address.

Hash table: is a collection of items stored in such a way that they can quickly be located.

The hash table could be implemented as an array or list of a given size with a number of empty spaces.

Example

An empty hash table that can store a maximum of 11 items is shown below, with spaces labelled 0,1, 2,…10.

0	1	2	3	4	5	6	7	8	9	10
Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty	Empty

Store items 78, 55, 34, 19 and 29 in the table using division by 11 and taking the remainder.

Item	Hash value
78	1
55	0
34	1
19	8
29	7

Each of these items can now be inserted into their location in the hash table.

0	1	2	3	4	5	6	7	8	9	10
55	78	34	Empty	Empty	Empty	Empty	29	19	Empty	Empty

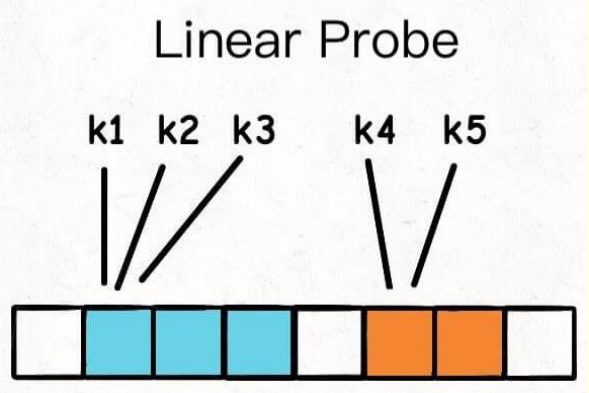
Hashing a string
A hash function can be created for alphanumeric strings by using the ASCII code for each character.

To hash the word CAB, we could add up the ASCII values for each letter and, if there are 11 spaces in the hash table, for example, divide by 11 and take the remainder as its hash value.

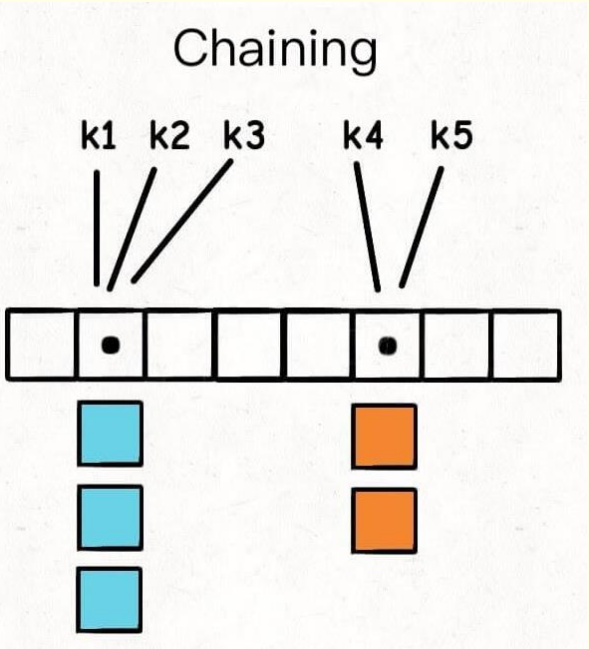
Collisions

A collision can happen when two keys generate the same hash address.

Linear probing could be used. Move through the structure one space at a time to find the next free space



Chaining could be used. Each location points (to the start of) a linked list. The new item is added to the end of the linked list / free.



Searching for an item

When searching for an item, these steps are followed:

1. apply the hashing algorithm to the key field of the item
2. examine the resulting cell in the list
3. if the item is there, return the item
4. if the cell is empty, the item is not in the table
5. if there is another item in that spot, keep moving forward until either the item is found or a blank cell is encountered, when it is apparent that the item is not in the table.

Comparison of linked lists and Hash tables

Linked List	Hash Tables
Searching of a Linked list involves starting at the first node and following the pointers until either the desired value is found, or the end of the list is reached, meaning the item isn't in the list. The bigger the linked list grows, the longer it takes to search. If a linked list doubles in size it will, on average, take twice as long to search. A list of size n takes on average n/2 checks. In Big O this is O(n), or linear complexity. If items are added to the end of the linked list then if the location of the last node is stored, that location can be ready made to point at the new item. The time to add items is constant. If they are added in some sort of order then the time to add items grows linearly due to the time spent searching for the right position. (Storing in order has the advantage that it is if an item isn't in the list this can be deduced once its location is passed, rather than waiting until the end.)	Searching of a hash table requires the key to be hashed and the correct location accessed. The time this takes is largely dependent on the time to create the hash. If we ignore collisions, the time to find an item will stay the same regardless of the size of the list. In other words it has O(1) or constant complexity. As the list grows collisions become more likely. Linear probing and chaining means that once a location has been found the time taken grows linearly with the number of collisions that have occurred for that location. However is still going to perform significantly better than a linked list. Adding items to a hash table involves hashing the key and placing it in the correct location. This takes a constant amount of time unless there are collisions then there is an overhead which grows with the number of collisions for that location.

Overall a hash table is likely to be the best option (assuming it has enough space and a good hashing algorithm which produces a hash quickly and with few collisions). It will give very consistent performance even as the list grows.

Implementation

Hash tables are used in the implementation of the data structure called a dictionary.

Dictionaries

A dictionary is an abstract data type consisting of associated pairs of items, where each pair consists of a key and a value. In Python, dictionaries are written as comma-delimited pairs in the format key:value and enclosed in curly braces. For example:

IDs = {342:'Harry', 634:'Jasmine', 885:'Max',571:'Sheila'}

Operations on dictionaries

It is possible to implement a dictionary using either a static or a dynamic data structure. The implementation needs to include the following operations:

```
# Create a new empty dictionary
my_dict = {}

# Add a new key:value pair to the dictionary
my_dict["name"] = "Alice"
print("After adding:", my_dict)

# Return a value associated with key 'name'
value = my_dict.get("name")
print("Value for 'name':", value)

# Return True or False depending on whether key is in the dictionary
has_age = "age" in my_dict
print("Is 'age' a key?", has_age)

# Return the length of the dictionary
length = len(my_dict)
print("Length of dictionary:", length)
```