Maps, Hash tables, Sets

Python's dict class is arguably the most significant data structure in the language.

dictionaries are commonly known as associative arrays or maps

Maps use an array-like syntax for indexing

The Map ADT

Five most significant methods of a Map

M[k]: Return the value v associated with key k in map M, if one exists; otherwise raise a KeyError. In Python, this is implemented with the special method __getitem__.

M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k. In Python, this is implemented with the special method __setitem__.

del M[k]: Remove from map M the item with key equal to k; if M has no such item, then raise a KeyError. In Python, this is implemented with the special method __delitem__.

len(M): Return the number of items in map M. In Python, this is implemented with the special method __len__.

iter(M): The default iteration for a map generates a sequence of *keys* in the map. In Python, this is implemented with the special method __iter__, and it allows loops of the form, **for** k **in** M.

Other methods

k in M: Return True if the map contains an item with key k. In Python, this is implemented with the special __contains__ method.

M.get(k, d=None): Return M[k] if key k exists in the map; otherwise return default value d. This provides a form to query M[k] without risk of a KeyError.

M.setdefault(k, d): If key k exists in the map, simply return M[k]; if key k does not exist, set M[k] = d and return that value.

M.pop(k, d=None): Remove the item associated with key k from the map and return its associated value v. If key k is not in the map, return default value d (or raise KeyError if parameter d is None).

Other methods

M.popitem( ): Remove an arbitrary key-value pair from the map, and return a (k,v) tuple representing the removed pair. If map is empty, raise a KeyError.

M.clear( ): Remove all key-value pairs from the map.

M.keys( ): Return a set-like view of all keys of M.

M.values( ): Return a set-like view of all values of M.

M.items( ): Return a set-like view of (k,v) tuples for all entries of M.

M.update(M2): Assign M[k] = v for every (k,v) pair in map M2.

M == M2: Return True if maps M and M2 have identical key-value associations.

M != M2: Return True if maps M and M2 do not have identical key-value associations.

A lookup table to begin with

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

A lookup table with length 11 for a map containing items (1,D), (3,Z), (6,C), and (7,Q).

Limitations: keys confined to integers, poor memory utilisation

Hash function

A hash function is a special algorithm that takes an arbitrary input (data of any size) and converts it into a fixed-size output (hash value).

Key characteristics of a good hash function are:

Uniform Distribution: Ideally, the hash function should distribute the hash values uniformly across the available output range to avoid clustering.

Deterministic: The same input should always produce the same hash value (assuming the function hasn't changed).

Avalanche Effect: Small changes to the input should result in significant changes to the hash value. This helps to minimize collisions (when two different inputs produce the same hash value).

Hashing

Hashing is the process of applying a hash function to an input value to generate a hash code. It's a way to create a concise and unique (ideally) identifier for the data.

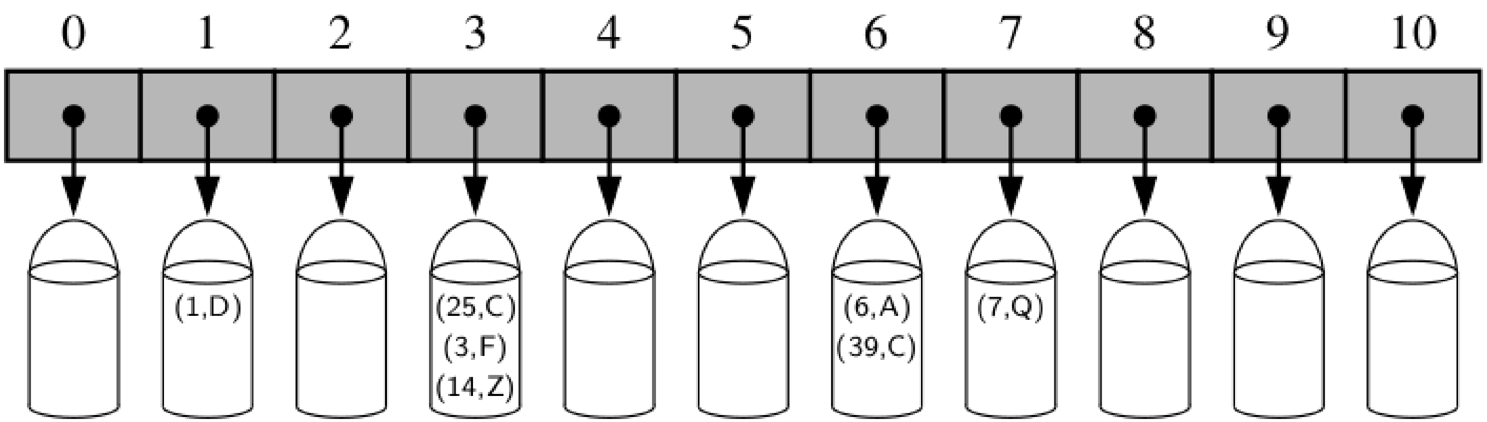
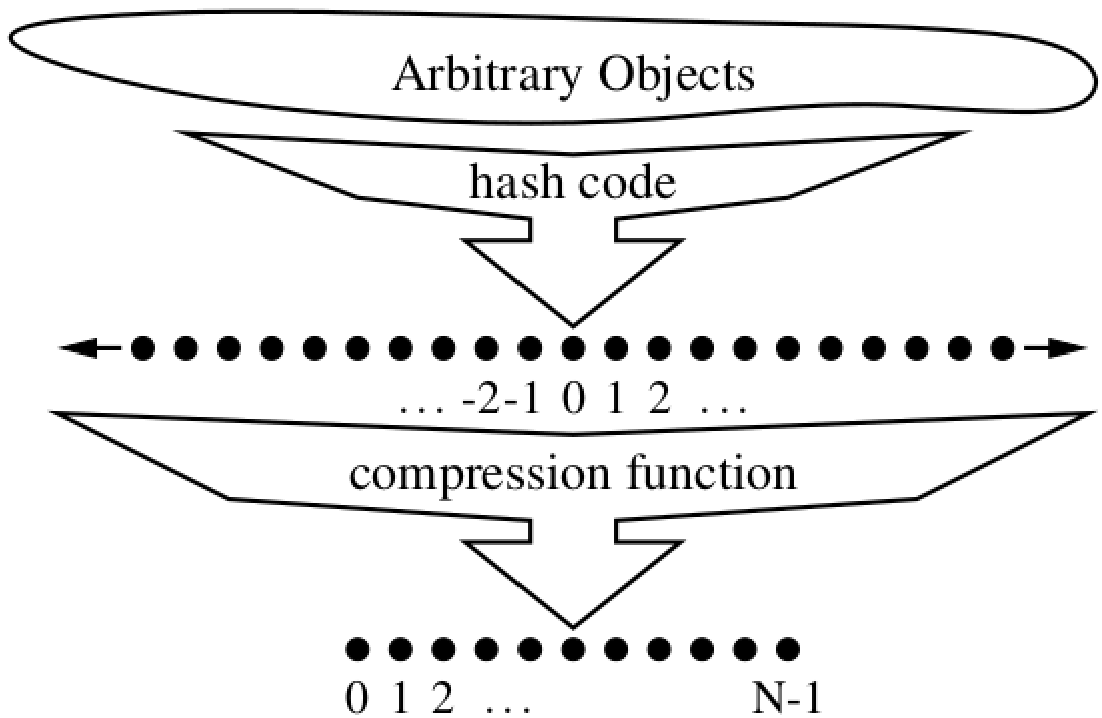


Figure 10.4: A bucket array of capacity 11 with items (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function.

hash function, h, is to map each key k to an integer in the range [0, N-1], where N is the capacity of the bucket array for a hash table.

Two components of a hash function: hash code and compression function



Hash code

Hash function performs is to take an arbitrary key k in our map and compute an integer that is called the hash code for k; this integer need not be in the range [0, N − 1], and may even be negative.

Common approaches to generate hash codes

- Bit Representation as an Integer
- Polynomial Hash Codes

$$x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}.$$

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \cdots + a(x_2 + a(x_1 + a x_0)) \cdots))$$

- Cyclic-Shift Hash Codes

Cyclic-Shift Hash Codes

```python
def cyclic_hash(s):
 mask = (1 << 32) - 1
 h=0
 for ch in s:
  h = (h << 5 & mask) | (h >> 27)
  h += ord(ch)
 return h
```

Cyclic hashing of 'hello' gives hash code: 112475631

Choice of shift matters

Shift	Collisions	
	Total	Max
0	234735	623
1	165076	43
2	38471	13
3	7174	5
4	1379	3
5	190	3
6	502	2
7	560	2
8	5546	4
9	393	3
10	5194	5
11	11559	5
12	822	2
13	900	4
14	2001	4
15	19251	8
16	211781	37

Comparison of collision behavior for the cyclic-shift hash code as applied to a list of 230,000 English words. The "Total" column records the total number of words that collide with at least one other, and the "Max" column records the maximum number of words colliding at any one hash code. Note that with a cyclic shif of 0, this hash code reverts to the one that simply sums all the characters.

Compression Functions

1) The Division Method

A simple compression function is the division method, which maps an integer i to

$$\mathbf{i \bmod N}$$

where N, the size of the bucket array, is a fixed positive integer.

if N is not prime, then it may lead to more collisions

E.g. - if we insert keys with hash codes $\{200, 205, 210, 215, 220, \ldots, 600\}$ into a bucket array of size 100, then each hash code will collide with at least three others. But if we use a bucket array of size 101, then there will be no collisions.

Compression Functions

2)Multiply-Add-and-Divide (or "MAD") method

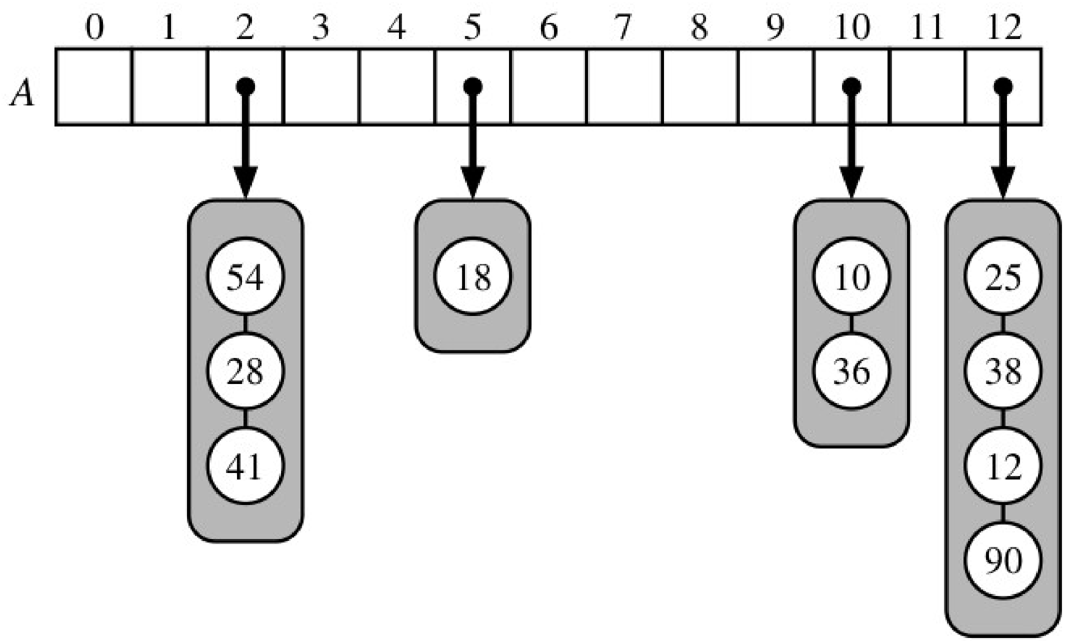
This method maps an integer i to

$$[(\mathbf{ai} + \mathbf{b}) \ \mathbf{mod}\ \mathbf{p}]\ \mathbf{mod}\ \mathbf{N},$$

where N is the size of the bucket array, p is a prime number larger than N, and a and b are integers chosen at random from the interval $[0, p - 1]$, with $a > 0$.

Collision-Handling Schemes

Separate Chaining



A hash table of size 13, storing 10 items with integer keys, with collisions resolved by separate chaining. The compression function is h(k) = k mod 13.For simplicity, we do not show the values associated with the keys.

Separate Chaining

Assuming we use a good hash function to index the n items of our map in a bucket array of capacity N, the expected size of a bucket is n/N.

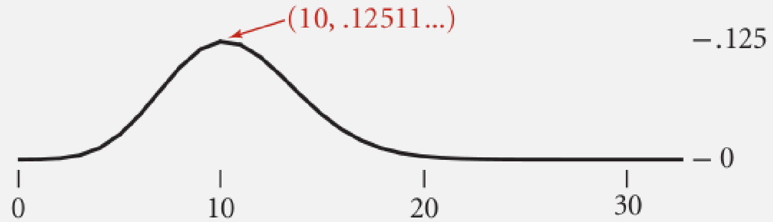
The ratio $\lambda = n/N$, called the load factor of the hash table, should be bounded by a small constant, preferably below 1.

As long as λ is O(1), the core operations on the hash table run in O(1) expected time.

Analysis of separate chaining

Proposition. Under uniform hashing assumption, prob. that the number of keys in a list is within a constant factor of N / M is extremely close to 1.

Pf sketch. Distribution of list size obeys a binomial distribution.



Binomial distribution $(N = 10^4, M = 10^3, \alpha = 10)$

equals() and hashCode()

Consequence. Number of probes for search/insert is proportional to N / M.

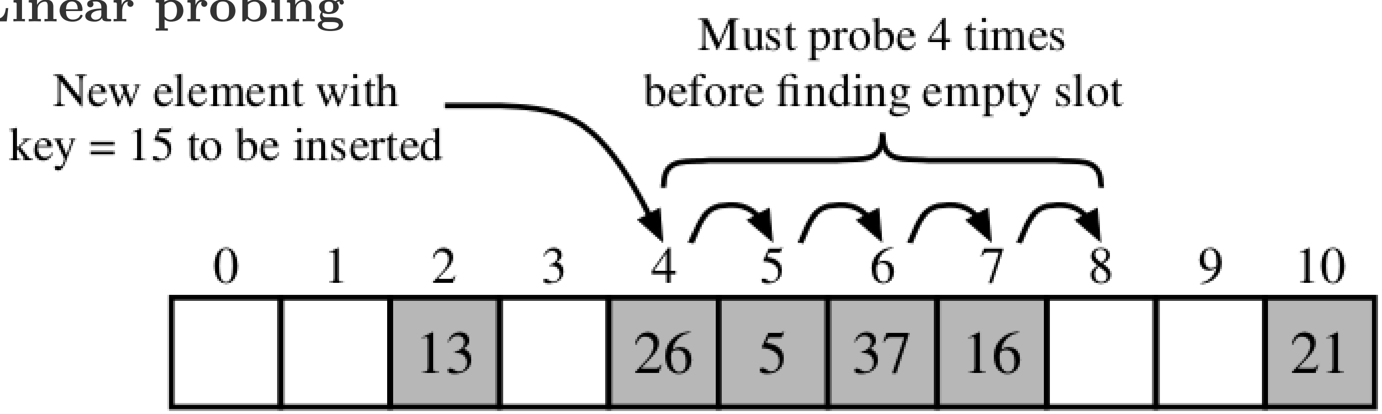
- M too large $\Rightarrow$ too many empty chains.
- M too small $\Rightarrow$ chains too long.
- Typical choice: $M \sim N / 4 \Rightarrow$ constant-time ops.

M times faster than sequential search

Open Addressing

Linear Probing and Its Variants

1) Linear probing



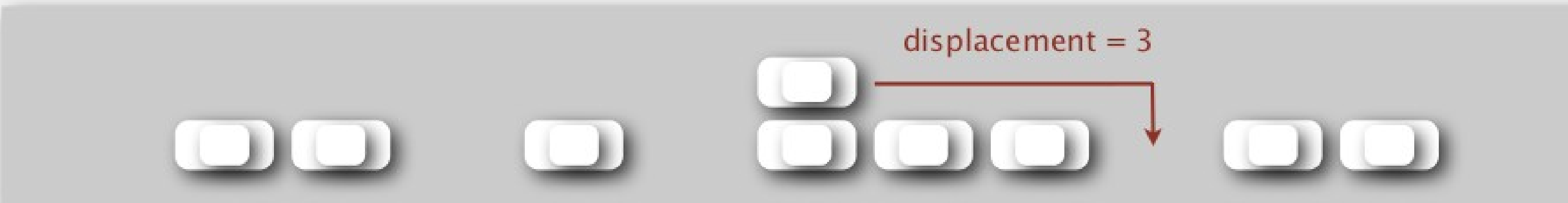
Insertion into a hash table with integer keys using linear probing. The hash function is h(k) = k mod 11. Values associated with keys are not shown.

Linear probing can save space but may lead to clustering problem (particularly if more than half of the cells in the hash table are occupied). Such contiguous runs of occupied hash cells cause searches to slow down considerably

Knuth's parking problem

Model. Cars arrive at one-way street with M parking spaces.
Each desires a random space i : if space i is taken, try $i + 1$, $i + 2$, etc.

Q. What is mean displacement of a car?



Half-full. With $M / 2$ cars, mean displacement is $\sim 3 / 2$.
Full. With M cars, mean displacement is $\sim \sqrt{\pi M / 8}$.

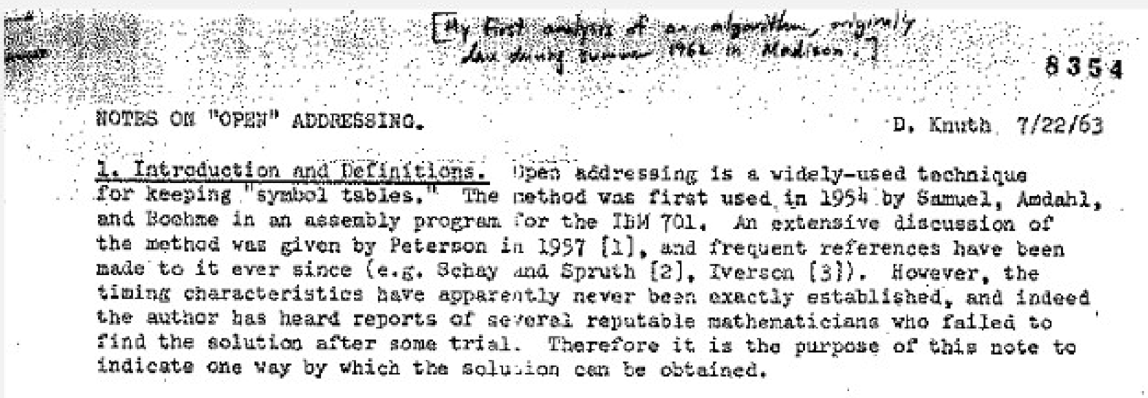
Analysis of linear probing

Proposition. Under uniform hashing assumption, the average # of probes in a linear probing hash table of size M that contains $N = \alpha M$ keys is:

$$\sim \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \qquad \sim \frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$$

search hit | **search miss / insert**

Pf.

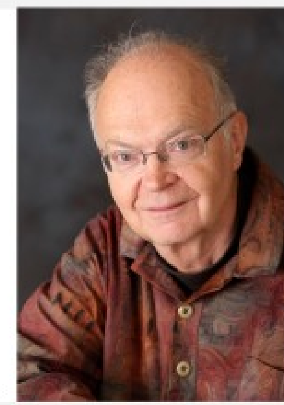


[My first analysis of an algorithm, originally done during Summer 1962 in Madison.]

8354

NOTES ON "OPEN" ADDRESSING. D. Knuth 7/22/63

1. Introduction and Definitions. Open addressing is a widely-used technique for keeping "symbol tables." The method was first used in 1954 by Samuel, Amdahl, and Boehme in an assembly program for the IBM 701. An extensive discussion of the method was given by Peterson in 1957 [1], and frequent references have been made to it ever since (e.g. Schay and Spruth [2], Iverson [3]). However, the timing characteristics have apparently never been exactly established, and indeed the author has heard reports of several reputable mathematicians who failed to find the solution after some trial. Therefore it is the purpose of this note to indicate one way by which the solution can be obtained.



Parameters.

- M too large $\Rightarrow$ too many empty array entries.
- M too small $\Rightarrow$ search time blows up.
- Typical choice: $\alpha = N / M \sim ½$.

probes for search hit is about 3/2
probes for search miss is about 5/2

Open Addressing

2) quadratic probing:

$$A[(h(k) + f(i)) \bmod N], \text{ for } i = 0, 1, 2, \ldots, \text{ where } f(i) = i^2$$

3) double hashing:

if h maps some key k to a bucket A[h(k)] that is already occupied, then we iteratively try the buckets

$A[(h(k) + f(i)) \bmod N]$ next, for $i = 1, 2, 3, \ldots,$

where $f(i) = i \cdot h^{\#}(k)$

Open Addressing: double hashing

In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is $h^{\#}(k) = q - (k \bmod q)$, for some prime number $q < N$. Also, N should be a prime.

Efficiency of Hash Tables

Operation	List	Hash Table	
		expected	worst case
__getitem__	$O(n)$	$O(1)$	$O(n)$
__setitem__	$O(n)$	$O(1)$	$O(n)$
__delitem__	$O(n)$	$O(1)$	$O(n)$
__len__	$O(1)$	$O(1)$	$O(1)$
__iter__	$O(n)$	$O(n)$	$O(n)$