Handling Collisions

Recap of Map ADT
Recap of Hashing and Compression

Handling collisions with Bucket Lists

Map ADT

Store elements (or (key,value) pairs)

getitem(key)	return the element with given key, or None if not there
setitem(key,value)	assign value to element with key; add new if needed
contains(key)	return True if map has some an element with key
delitem(key)	remove element with key, or return None if not there
length()	return the number of elements in the map

Complexity of implementations

	getitem(k)	contains()	setitem(k,v)	delitem()	build full map
unsorted list	O(n)	O(n)	O(n)	O(n)	$O(n^2)$
sorted list	O(log n)	O(log n)	O(n)	O(n)	$O(n^2)$
AVL tree	O(log n)	O(log n)	O(log n)	O(log n)	O(n log n)

Can we get search time closer to O(1)?

Initial Hash table

Use an array of known size N for storage

Generate an integer for each key, using a hash function.

Compress (i.e. reduce) the hash number to an integer in [0,N-1] by taking the hash number modulo N

index = hash(key) mod N

Store the item in cell *index*.

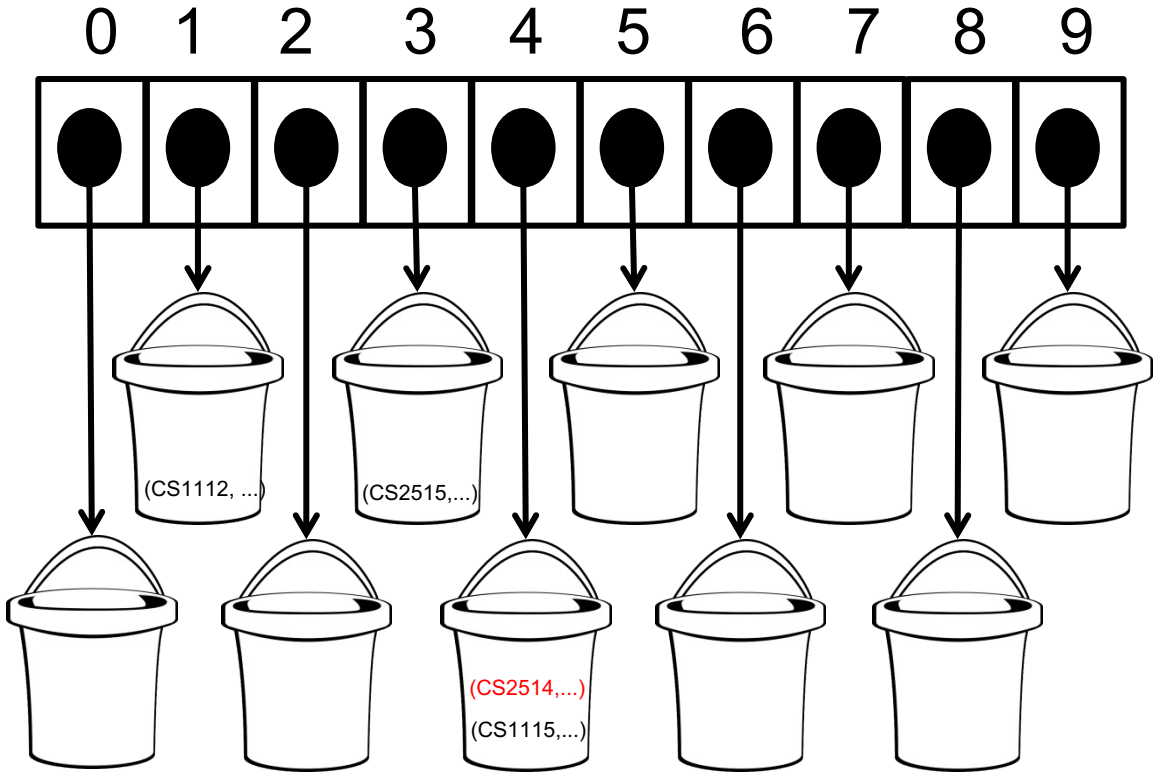
Separate Chaining

Each cell in the list will maintain its own list of values

- known as a *bucket* array

```
map.setitem('CS2514','Introduction to Java')
```

hash('CS2514') % 10 evaluates to 4



search complexity in bucket arrays

To find an item in a bucket array, we have to compute the hash, then the compressed location, and then search the bucket (i.e. the list).

If we are unlucky, all of our items could end up being sent to the same bucket, and so search is O(n).

If we have *n* items to insert into a bucket array of size *N* then we can expect *n/N* items in any given bucket, and so the *expected* complexity is O(*n/N)*.

But we were aiming for O(1) ...

How can we bring O(n/N) closer to O(1)?

If we have *n* items to insert into a bucket array of size *N* then we can expect *n/N* items in any given bucket, and so the *expected* complexity is O(*n/N)*.

dynamic bucket arrays

We have already seen dynamic arrays

- when we need more space, create bigger list storage and copy elements across

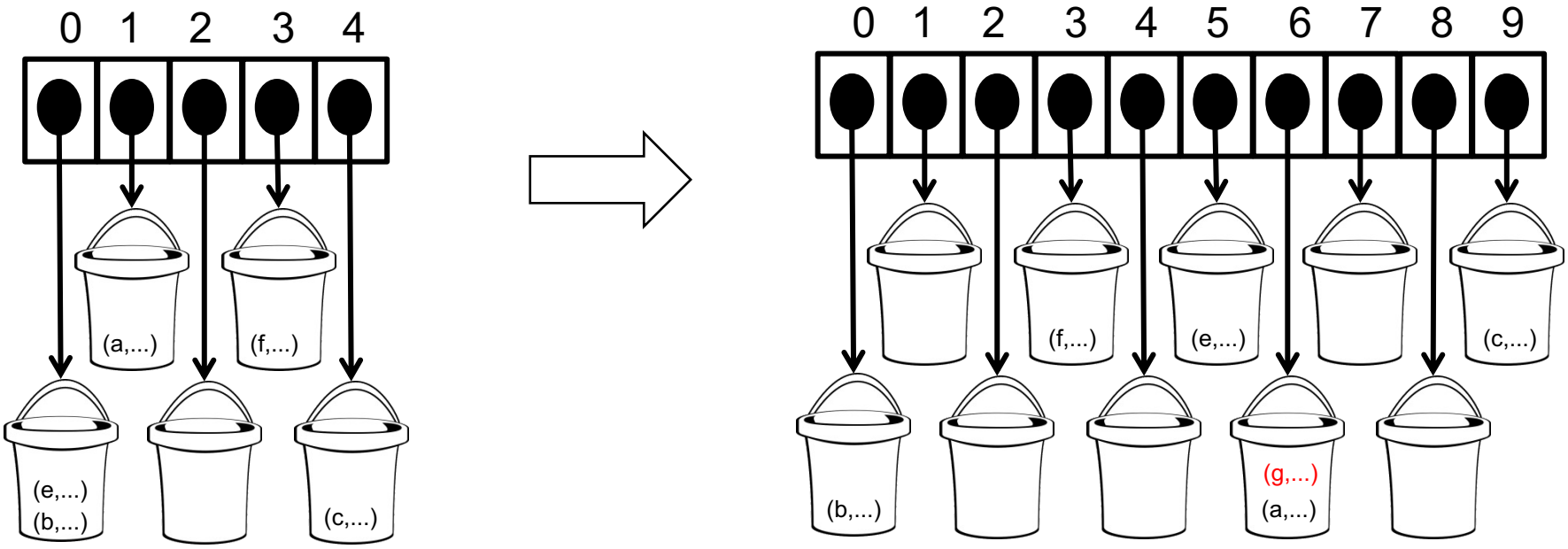
We can do the same for hash tables, and thus increase the value of N

If we can guarantee that *n/N* will always be less than some constant *c*, then we have an expected search time of O(*c*), which is O(1).

But each time we resize, we must compute new locations, since the original locations were based on the old N

- so that if we now add an item equivalent to one already in the map, it will be assigned the same bucket

map.setitem('g', '...')



How do we obtain the expected complexity?

By using bucket arrays with dynamic list sizes, we get a complexity of *expected* O(1) for lookup, insertion and deletion.

"Expected" in this case does **not** mean "average", and it is **not** the same as the term in probability / statistics

- on most occasions, for most buckets, it is O(1)

It assumes an even spread of items across the buckets.

To get close to an even spread, we need to pay attention to

- the hash function
- the compression function

Hash function properties

We would like to avoid collisions as far as possible

- distinct items should get distinct hashes, so we should use as much of the content as possible
- with a fixed size output, some collisions are inevitable

Simple hash function:

- integers that fit into 32 bit hash to their own value
  - integers in larger bit size hash to exclusive-or of first half $\oplus$ second half
- don't hash floats
- hash strings by polynomial sum of the characters, using a fixed constant integer *c*
  - $x_0c^{n-1} + x_1 c^{n-2} + x_2c^{n-3} + \ldots + x_{n-2}c + x_{n-1}$

Compression

Main aim is to distribute the compressed hash values as evenly as possible across the range.

- we used h % N, where N is the size of the list.

This works, but if a sequence of values has a pattern related to N, many will end up colliding

e.g. compressing the numbers in the sequence

10,14,18,22,26,30,34,38,42,46,50

try to choose prime numbers for N

mod 12 gives buckets: [10,22,34,46], [14,26,38,50], [18,30,42]
(the numbers increase by 4, which is a factor of 12)

mod 13 gives: [10],[14],[18],[22],[26],[30],[34],[38],[42],[46],[50]

Multiply, Add and Divide

MAD_hash(key) = ((a*hash(key) + b) % p) % N

where

p is a prime number > N

$0 < a < p$

$0 \leq b < p$

can be shown (using Group Theory) that this gives a better distribution of compressed values than simple %N

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item

SHA-1

From Wikipedia, the free encyclopedia

In cryptography, **SHA-1** (**Secure Hash Algorithm 1**) is a cryptographic hash function designed by the United States National Security Agency and is a U.S. Federal Information Processing Standard published by the United States NIST.[2] SHA-1 is considered insecure against well-funded opponents, and it is recommended to use SHA-2 or SHA-3 instead.[3][4]

SHA-1 produces a 160-bit (20-byte) hash value known as a message digest. A SHA-1 hash value is typically rendered as a hexadecimal number, 40 digits long.

SHA-1 is a member of the Secure Hash Algorithm family. The four SHA algorithms are structured differently and are named SHA-0, SHA-1, SHA-2, and SHA-3. SHA-0 is the original version of the 160-bit hash function published in 1993 under the name SHA: it was not adopted by many applications. Published in 1995, SHA-1 is very similar to SHA-0, but alters the original SHA hash specification to correct weaknesses that were unknown to the public at that time. SHA-2, published in 2001, is significantly different from the SHA-1 hash function.

SHA-1

General	
Designers	National Security Agency
First published	1993 (SHA-0), 1995 (SHA-1)
Series	(SHA-0), SHA-1, SHA-2, SHA-3
Certification	FIPS PUB 180-4, CRYPTREC (Monitored)
Detail	
Digest sizes	160 bits
Structure	Merkle–Damgård construction
Rounds	80
Best public cryptanalysis	

Best public cryptanalysis

A 2011 attack by Marc Stevens can produce hash collisions with a complexity between $2^{60.3}$ and $2^{65.3}$ operations.[1] The first public collision was published on 23 February 2017.[2] SHA-1 is prone to length extension attacks.

Next lecture

Other solutions to collisions
Implementing a dictionary class

Remaining Schedule:

Wed 22nd Lecture:	revision
Thurs 23rd Lab:	binary heap implementation
Fri 24th Lecture:	revision
Wed 29th 2pm:	**2nd class test**
Thurs 30th Lab:	drop-in revision, Q&A
Fri 1st Lecture:	revision.