Hash Tables & Hashing

Simon Powers

SET08122 – Algorithms & Data Structures

Overview

- Hash tables
- Hashing functions
- Applications to security

A basic problem in computer science

- Finding data that you have stored
- e.g. list of names and telephone numbers
- How to organise the list so you can find the phone number given the name?
- Any ideas about how to solve this?

Hash tables

- Hash tables support one of the most efficient forms of searching: hashing.
- A hash table consists of an array in which data is accessed by a special index called a key.
- Establish a mapping between the set of *all possible keys* and indices in the array.
  - a hash function, $h(k)$.
  - $h(name)$ -> *index* of phone number in array.
- Use this to perform constant time searches!

Good hashing

index:	0	1	2	3	4	5	6	7
	Simon W			Simon P		Emma		Neil

h("Simon P") = 3
h("Simon W") = 0
h("Emma") = 5
h("Neil") = 7

- Best case: direct hashing. Every key hashes to a unique index.
- Rarely possible. Why?
- Otherwise, we need to spread entries around the table and minimise collisions.

Good hashing

- In most applications, number of positions in hash table much less than *universe* of possible keys
- => unfortunately some keys will map to the same index = a collision ☹
- We want to minimise collisions
  - How can we do this?

Good hashing

- In most applications, number of elements in hash table much less than universe of possible keys
- => unfortunately some keys will map to the same index = a collision ☹
- We want to minimise collisions
  - How can we do this? (uniform hashing)
- But collisions will still occur ☹

The hard work with hash tables

- Mapping keys evenly around the table
  - approximating uniform hashing
- Dealing with collisions
  - chained hash tables
  - open-addressed hash tables

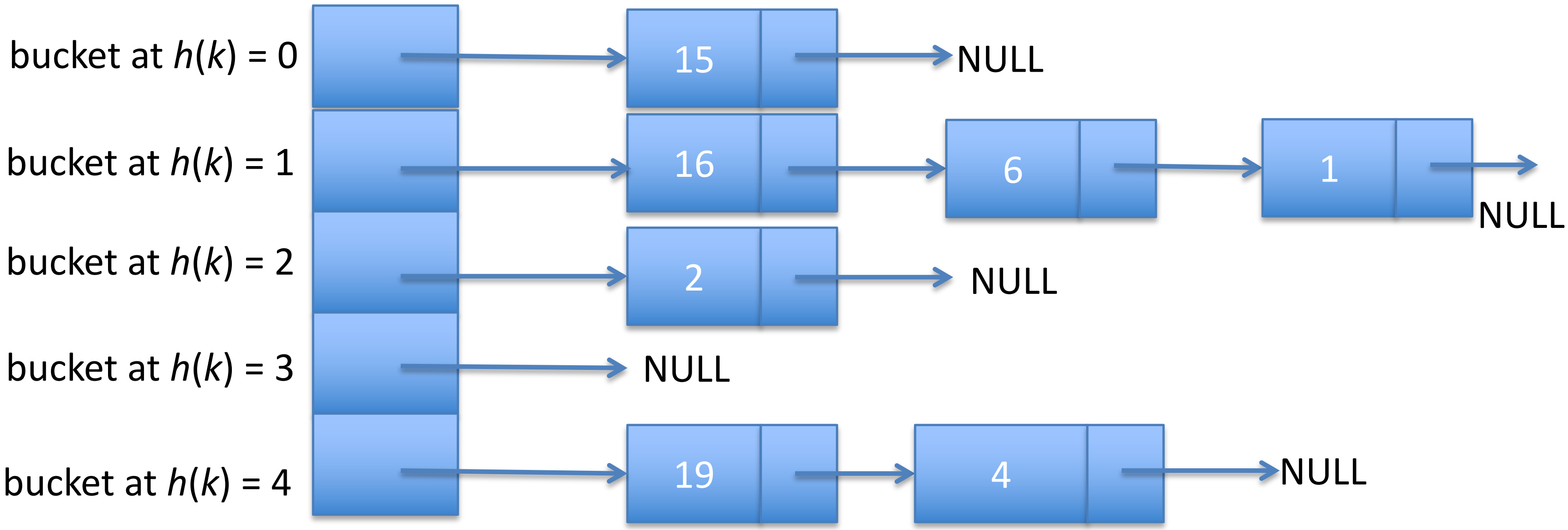
Applications of hash tables

- Database systems (random access)
- Compilers
- Data dictionaries
- Associative arrays

CHAINED HASH TABLES

Chained hash tables....

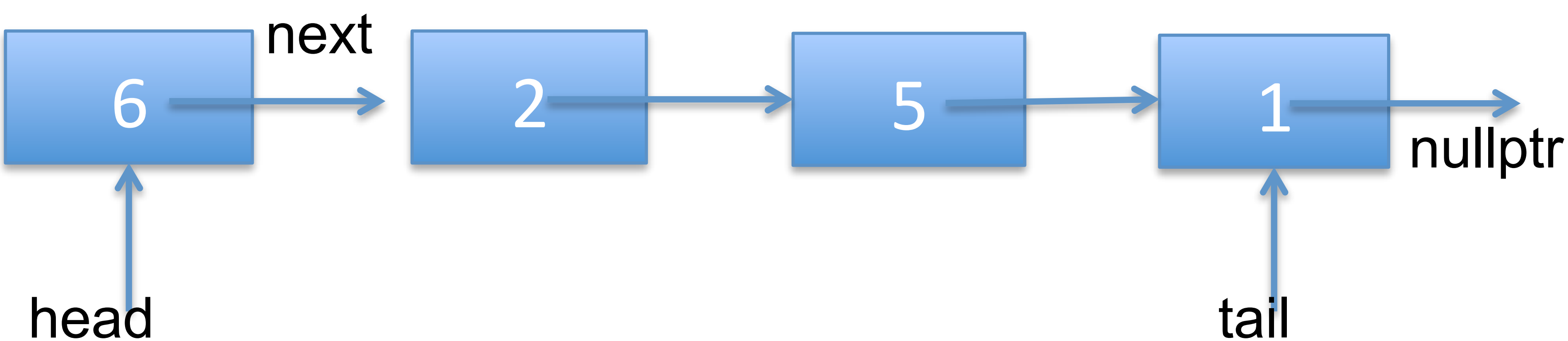
- .... are arrays of linked lists



- When a collision occurs, simply place the new item at the head of the corresponding linked list.

Linked Lists

- Definition
  - A data structure that makes it easy to rearrange data elements without having to move them in the memory
  - Consists of multiple **nodes**, each contains two subentries:
    - The actual data value
    - A pointer to the next node



Exercise

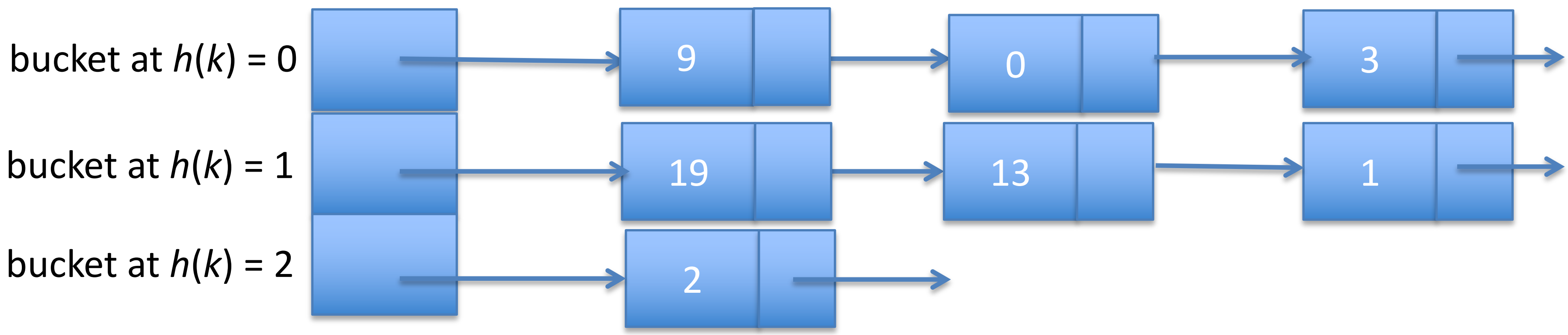
- Insert the following keys, *in order*, into a chained hash table of size 3:

1,3,0,2,13,19,9

Using the hash function:

$h(k) = k \bmod 3$

Exercise



Dealing with collisions...

- ...is easy
- But, if an excessive number of collisions occurs at a specific position, the bucket becomes longer and longer ☹
- Worst case we go from O(1) to O(n) for retrieving an item

Dealing with collisions...

- ...is easy
- But, if an excessive number of collisions occurs at a specific position, the bucket becomes longer and longer ☹
- Worst case we go from O(1) to O(n) for retrieving an item
  - e.g. with $h(k) = c$
- Ideally we want all buckets to grow at the same rate.

Keeping an eye on the load factor

- Load Factor = *number of elements* in table / *number of positions* into which elements may be hashed.
- In chained hash tables, this gives the maximum number of elements we can *expect* to encounter in a bucket, assuming uniform hashing.
  - e.g. with 1699 buckets and 3198 elements inserted, the load factor is....

Keeping an eye on the load factor

- load factor = number of elements in table / number of positions into which elements may be hashed
- In chained hash tables, this gives the maximum number of elements we can *expect* to encounter in a bucket, assuming uniform hashing.
  - e.g. with 1699 buckets and 3198 elements inserted, the load factor is....
  - 3198 / 1699 = 2
- But remember, in reality uniform hashing is only *approximated*...

SELECTING A HASH FUNCTION

Hash functions

- A good hash function should approximate uniform hashing.
- $h(k) = x$, where x is the hash value of k.
- Most methods assume k to be an integer (and x must be!)
- If k isn't an integer, we can coerce it to be one…

Coercing strings into integers before hashing

- Can convert some combination of the characters into their ASCII codes and sum them.
- We then feed this as our number *k* to the hash function.
- e.g. convert the first 3 characters…

Emma = E + m + m = 69 + 109 + 109 = 287

Neil = N + e + i = 78 + 101 + 105 = 284

Simon = S + i + m = 83 + 105 + 109 = 297

Coercing strings into integers before hashing

getName() = g + e + t = 103 + 101 + 116 = 320

getAge() = g + e + t = 103 + 101 + 116 = 320

getAddress() = g + e + t = 103 + 101 + 116 = 320

- A better way might be:
  1. select the 4^{th}, 5^{th}, 6^{th} and 7^{th} characters
  2. permutate them in a way that randomises them further
  3. stuff them into specific bytes of a 4-byte integer.

Division method hash functions

- Once we have our key k as an integer, an easy way to map it into one of m positions in the table is:

  $h(k) = k \bmod m$

- But, we need to avoid values of m that are powers of 2.

Division method hash functions

- Once we have our key k as an integer, an easy way to map it into one of m positions in the table is:

  $h(k) = k \bmod m$

- But, we need to avoid values of m that are powers of 2.
  – Because if $m = 2^p$, then $h(k)$ becomes just the p lowest-order bits of k.

Choosing our table size, *m*

- Typically we set *m* to be a prime number not too close to a power of 2.
- e.g. if we expect to insert around 4500 elements into a chained hash table, we might choose m=1699
  - why?

Choosing our table size, *m*

- Typically we set *m* to be a prime number not too close to a power of 2.
- e.g. if we expect to insert around 4500 elements into a chained hash table, we might choose m=1699
  - why?
  - it's a good prime between 2^10 and 2^11.
  - it results in a load factor of

Choosing our table size, *m*

- Typically we set *m* to be a prime number not too close to a power of 2.
- e.g. if we expect to insert around 4500 elements into a chained hash table, we might choose m=1699
  - why?
  - it’s a good prime between 2^10 and 2^11.
  - it results in a load factor of 4500/ 1699 (approx. 2.6)

OPEN-ADDRESSED HASH TABLES

Open-addressed hash tables

- All elements reside directly in the table itself (no linked lists).
- We need another way of resolving collisions, why?

Open-addressed hash tables

- All elements reside directly in the table itself (no linked lists).
- We need another way of resolving collisions, why?
- We resolve collisions by *probing* the table until we find an empty slot.
  - Go to index $h(k)$, then probe until we find a free slot (insertion) or we find the item (retrieval or deletion)

Performance

- The goal is to minimise the number of probes we have to do.
- This depends on
  - The load factor
  - How uniform $h(k)$ is
- The load factor of an open-addressed hash table cannot be > 1. Why?

Watch your load factor…

- Assuming uniform hashing, the number of positions we can expect to probe in an open-addressed hash table is

1 / (1 – load factor)

Load factor (%)	Expected probes
< 50	< 2
80	5
90	10
95	20

- How close we approximate uniform hashing also depends on how we probe the table when collisions occur….

Linear probing

- The simplest sort of probing you can think of…

Linear probing

- The simplest sort of probing you can think of…
- When a collision occurs store the item in the next empty slot in the table.
- More formally:

$h(k,i) = (h'(k) + i) \bmod m$

Linear probing

- The simplest sort of probing you can think of…
- When a collision occurs store the item in the next empty slot in the table.
- More formally:

$h(k,i) = (h'(k) + i) \bmod m$

where $h'(k)$ is our plain old hash function, i is the number of times the table has been probed, and m is the size of the table.

After inserting k = 37, 83, 97, 78, 14 with no collisions

$h(k,i) = (k \bmod 11 + i) \bmod 11$

After inserting k = 37, 83, 97, 78, 14 with no collisions

0										10
	78		14	37		83			97	

Inserting an element with k = 59; i = 0, 1

				59						
	78		14	37	59	83			97	

$h(k,i) = (k \bmod 11 + i) \bmod 11$

After inserting k = 37, 83, 97, 78, 14 with no collisions

0										10
	78		14	37		83			97	

Inserting an element with k = 59; i = 0, 1

59

	78		14	37	59	83			97	

Inserting an element with k = 25; i = 0, 1, 2, 3, 4

25

	78		14	37	59	83	25		97	

$h(k,i) = (k \bmod 11 + i) \bmod 11$

After inserting k = 37, 83, 97, 78, 14 with no collisions

0 10

	78		14	37		83			97	

Inserting an element with k = 59; i = 0, 1

59

	78		14	37	59	83			97	

Inserting an element with k = 25; i = 0, 1, 2, 3, 4

25

	78		14	37	59	83	25		97	

Inserting an element with k = 72; i = 0, 1, 2

72

	78		14	37	59	83	25	72	97	

$h(k,i) = (k \bmod 11 + i) \bmod 11$

The disadvantage of linear probing

- Suffers from *primary clustering*
  - large chains of occupied positions develop as the table becomes more and more full ☹

Quadratic probing

- Reduce linear clustering by going up in increments of i^2 instead of increments of i.
  - i.e. probe positions 1 long, then 4 along, then 9 along…
- Performs better than linear probing as clustering is less severe
  - technically the clustering is known as secondary clustering

Universal hashing

- Generates hashing functions randomly at run time. Why?

Universal hashing

- Generates hashing functions randomly at run time. Why?
  - So that no particular set of keys is likely to produce a bad distribution of elements in the hash table.
- Even hashing the same set of keys during different executions may lead to different numbers of collisions.

Review questions

1. Why are hash tables good for random access but not sequential access (e.g. accessing records sequentially in a database)?
2. What is the worst-case performance of searching for an element in a chained hash table? How can we avoid this?
3. What is the worst-case performance of searching for an element in an open-addressed hash table? How can we avoid this?

Review questions answers

1. After hashing to sum position, we have no way to determine where the next smallest or largest key resides.

2. A chained hash table performs worst when all elements hash into the same bucket. Searching is then O(n). If the hash function approximates uniform hashing, this will not occur.

3. Worst-case in open-addressed hash table is when the table is completely full and the key is not in the table, leading to O(m), where m is number of positions in table. This can occur with any hash function, but to ensure reasonable performance we should not let the load factor go above 0.8.

APPLICATIONS TO SECURITY

Cryptographic hash functions

- In general, hash functions map data of arbitrary size down to data of a fixed size.
- In security applications, we want to map data to a *bit string* (hash value) of a fixed size…
- ...and make it a one-way function so it's very difficult to go back from the hash value to the original data.
- Applications?

Password verification

- We don't want to store passwords on the server as plain text. Why?
- Instead we store the hash of the password.

Cryptographic hash functions

- Something like $h(k) = k \bmod 11$ would not be a good choice.
- Given a hash value v, it should be difficult to find a k such that $h(k) = v$.
  - Known as pre-image resistance
- It should be difficult to find two keys such that $h(k_1) = h(k_2)$
  - Collision resistance

Rainbow tables for cracking password hashes

- Dedicate huge amounts of computer time to hashing a huge database of possible passwords.
  - e.g. use an Nvida GPU cluster
- This database is a rainbow table.
- Reversing a password hash is now as simple as looking up the hash in the rainbow table.

Salting

- Add a random value to the password
  - Store this value with the hash
- Means that rainbow tables have to be computed for many more combinations of possible passwords.
- Leads to an arm race (but far better than not salting).

MD5 checksums

- Hash a block of data, e.g. a file.
- Send the hash to another person.
- Then when they receive a copy of the file, they can hash the copy and check the hashes match.
- Allows us to detect tampering or errors in transmission.

We've covered…

- Hash tables can perform very efficient searching and insertion/deletion.
- Need to choose a good hash function.
- Need to deal with collisions.
- Applications of hashing to security.

Acknowledgements

- Much of the content on hash tables is based on Chapter 8 of Loudon, K. (1999). *Mastering Algorithms with C*. O'Reilly & Associates, Inc.