L05.01
Hashing II

50.004 Introduction to Algorithms

Ioannis Panageas (ioannis@sutd.edu.sg)

ISTD, SUTD

CLRS Ch 11.3-11.4

Slides by A.Binder and based on Dr. Simon LUI

Recap last lecture …

- Chained hash:
  - an array s.t. each entry is a linked list
  - a hash function that maps keys onto array indices
  - Map n keys from a very large key space onto hash table of size m
- may have hash collisions
  - Hash operations: insert,delete as O(1) worst case
  - Search O(n) worst, O(1+α) average case (simple uniform hashing assumption)
  - When combined with table doubling: O(1) average case
  - Amortized costs for table doubling+ insertions: also O(1)
- Good hash functions: close to simple uniform hashing assumption each key has equal chance to end up in any of the bins … ensures on average equal load across the whole table

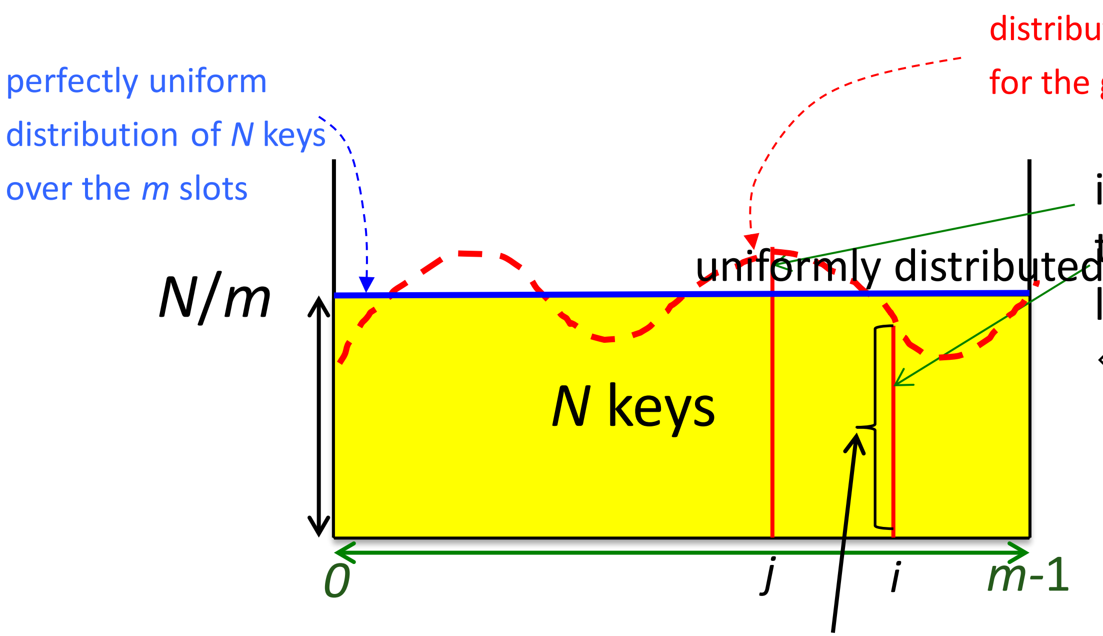
Today

- Open addressing – hashing without linked lists
  - Different probing strategies
- Cuckoo hashing O(1) search worst case
- Other uses of hashes / cryptographic hashes
  - File tampering
  - Digital message signing with hashed messages
- Universal hashing

Worse case discussion

Assume $|U|=N$, hash table size = l, $N > l$

<u>Lemma</u>: Then for any hash function h there exist at least N/l keys that hash on the same position (collide)



if there is a location i with $< N/l$ keys then there must be some other location j with $> N/l$ keys
⇔ there is some location where $\geq N/l$ keys hash

For any hash function we can find $\geq N/l$ keys that collide

- An alternative to deal with collisions than doubly linked lists?

Open addressing

- Do away with doubly linked lists, one array entry = one element
- Pay a price:
  - can fill table with at most n=l elements, then must do table doubling …
  - Need to search for free slots if A[h(x.key)] is used "Probing"

  - Replace hash function h(k) by a hash function

    h(k,i) with a parameter i that allows to search for the next slot

Example – linear probing

- k – key to be inserted
- g(k) = k mod 10
- h(k,i) = (g(k)+i) mod 10   i – index for probing, start with h(k,i=0)

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Open addressing algorithms

- Linear probing

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

- Quadratic probing

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

- Double hashing

$$h(k,i) = (h_1(k) + i\ h_2(k)) \bmod m$$

$$h_1(k) = k \bmod 13$$

$$h_2(k) = 1 + k \bmod 11$$

$$k = 14 : h_1(14) = 1, h_2(14) = 4$$

We probe positions 1, 1+4, 1+2x4, 1+3x4, ...

Example – linear probing

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

h(28)=28%11=6

h(47)=47%11=3

h(20)=20%11=9

h(36)=36%11=3

h(43)=43%11=10

h(23)=23%11=1

h(25)=25%11=3

h(54)=54%11=10

0	
1	
2	
3	47
4	
5	
6	28
7	
8	
9	20
10	

insert 28,47,20

0	
1	
2	
3	47
4	36
5	
6	28
7	
8	
9	20
10	

insert 36

0	
1	23
2	
3	47
4	36
5	
6	28
7	
8	
9	20
10	43

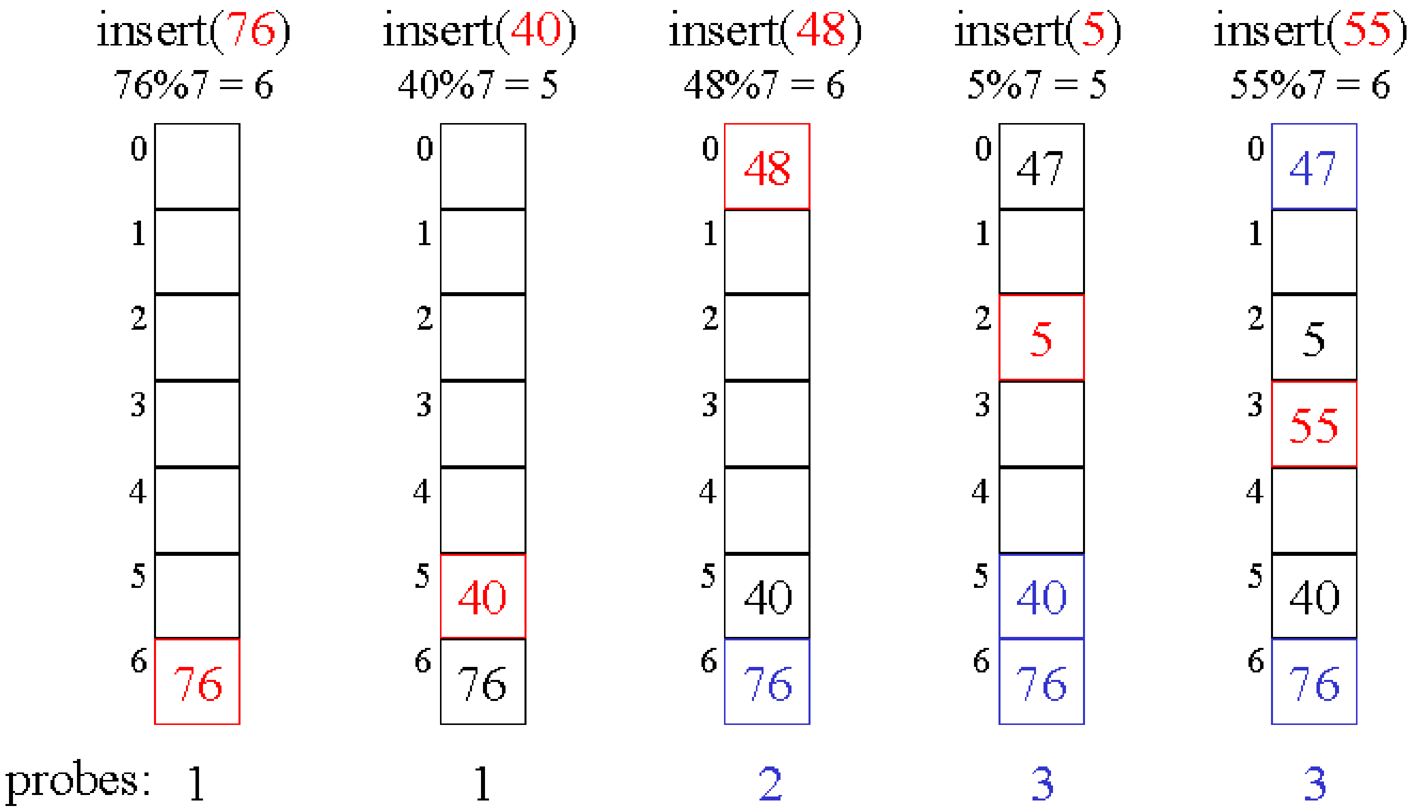
insert 43,23

0	54
1	23
2	
3	47
4	36
5	25
6	28
7	
8	
9	20
10	43

insert 25,54

Example – quadratic probing

- $h(k,i) = (h'(k)+c_1 i + c_2 i^2) \bmod 7 \qquad c_1=0\ ,\ \ c_2=1$
- $H'(k) = k \bmod 11$



Questions

A. Linear probing
Consider the hash table in the picture with some keys already inserted. Where will you insert k=17 when h'(k) = k mod 13?

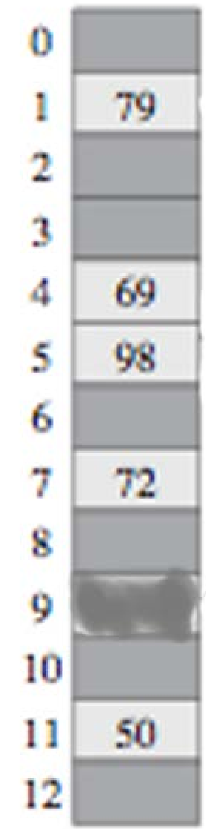
1. location 2
2. location 6
3. location 10

B. Quadratic probing
Where will you insert k=17 when h'(k) = k mod 13, c1 =2, c2 =1?

1. location 2
2. location 7
3. location 12

C. Double hashing
Where will you insert k=14 when h1(k) = k mod 13, h2(k) = 1+(k mod 11)?

1. location 2
2. location 9
3. location 10



Open Adressing performance

- Uniform hashing assumption: each key is equally likely to have any of m! permutations of {0,…,m-1} as probe sequence

- With above: Insertion: $1/(1-\alpha)$ probe steps on average

- Search:
  - $1/(1-\alpha)$ probe steps on average for unsuccessful search
  - $1/\alpha \ \log(1/(1-\alpha))$ probe steps on average for sucessful search

Open addressing vs chaining

- Open addressing:
  - Better cache performance (no pointers to off regions needed when objects are "small", e.g. integers, floats)
- Chaining:
  - Less sensitive to hash function choices
    - When keys are clustering in parts of the table, then linear/quadratic probing will have many steps
  - Less sensitive to high load factors
    - In practice open adressing needs alpha to be kept small, rule of thumb: like 50-70%, otherwise $1/(1-\alpha)$ becomes a nightmare

Cuckoo Hashing

- Uses 2 hash functions $h_1(k), h_2(k)$, where $h_1(k) \neq h_2(k)$
- Key k stored either in $T[h_1(k)]$ or in $T[h_2(k)]$
- Lookup: Just look at at most 2 places! $O(1)$
- Insertion:

If $T[h_1(k)]$ empty, store the key there
else if $T[h_2(k)]$ empty, store the key there
If both full, store key in $T[h_1(k)]$ and move key that was there to its other location (bumping out the key that might be there, etc)

- If α<1 insertion succeeds with high probability
- If insertion loops: rehash the entire table (or double table size)
- Insertion takes constant time on average

Example of cuckoo hashing

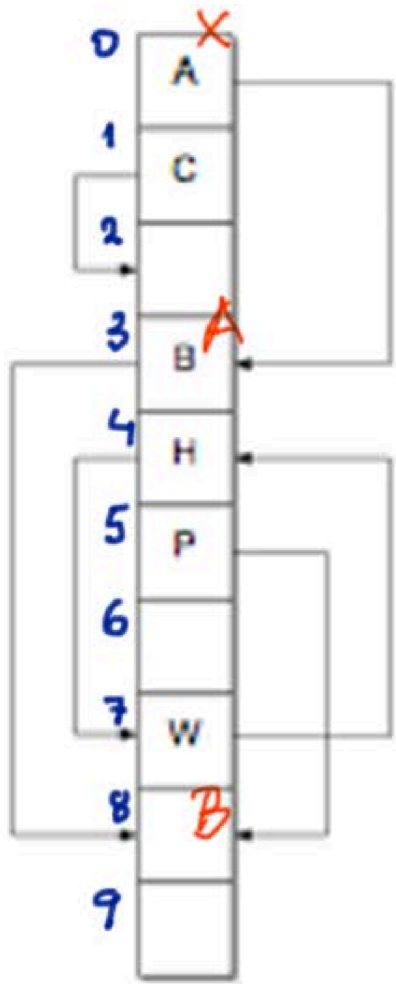
insert x

$h_1(x) = 0$

$h_2(x) = 4$

$h_1(A) = 0, \; h_2(A) = 3$

$h_1(B) = 3, \; h_2(B) = 8$



x bumps A, A bumps B, B finds an empty location

Bumping H creates a cycle → rehash everything

Cuckoo Hashing

- Search is O(1) worst-case, not average
- Insertion is O(n) worst-case (average performance is better)

Other hashing use cases

- File modification check
  - Was your data x tampered that is stored somewhere?
  - Compute hash before you upload,
  - Check by rehashing

  - Problem: when an attacker succeeds to fool you ?

Other hashing use cases

- File modification check
  - Was your data x tampered that is stored somewhere?
  - Compute hash before you upload,
  - Check by rehashing

  - Problem: when an attacker succeeds to fool you ?
  - if he finds an x’ such that $h(x)=h(x')$ for the given x

- Digital signatures
  - A has public key PK_A, private key SK_A. A can sign a message M by private key to obtain a signature s:
  - For large messages a hash h(M) instead of M is signed.

    $$s = sign(h(M), SK_A) .$$
  - recipient can verify that M was signed by A.
    - B runs a function verify(h(M),s, PK_A)
  - Problem: attacker wants to pretend that Alice signed document D2, that the attacker owns. what can he try to do?
  - Side info: attacker does not want to show D2 to Alice, no way to let alice sign D2 directly.

- Digital signatures
  - A has public key PK_A, private key SK_A. A can sign a message M by private key to obtain a signature s:
  - For large messages a hash h(M) instead of M is signed.

    $$s = sign(h(M), SK_A) .$$
  - recipient can verify that M was signed by A.
    - B runs a function verify(h(M),s, PK_A)

  - Problem: attacker can try to find a document D1 such that h(D1)=h(D2), then ask Alice to sign D1 to obtain s
  - Then reuse s with D2

- Commitment check:
  - I want to assure somebody that I committed a sum x of SGD for some project, and that I did not change that sum afterwards on the bank account.
  - I do not want to disclose the sum.
  - Give the person the right to ask the bank to see a hash z=h(x) of the account balance instead

  - Problem: don't want that my ominous partner can reverse the hash, e.g. find that x that created the hash value z.

- Commitment check:
  - I want to assure somebody that I committed a sum x of SGD for some project, and that I did not change that sum afterwards on the bank account.
  - Problem: don't want that partner can reverse the hash, e.g. find the sum x that created the hash value.
  - One solution: I do not hash x but I hash x+c, where c is a large random number that I will remember
    c is a so called salt → salted hashing (off lecture)

Cryptographic hashing functions

- What properties hash functions are desirable for such applications?
- One-way: given a hash z, it should be infeasible to find the x that created this hash: $h(x)=z$
- Collision-resistance: infeasible to find any pair x,x' such that $h(x)=h(x')$
- Target-collision-resistance:
  - Given some x it is infeasible to find an x' such that $h(x')=h(x)$
- Desired property: hash maps 2 close keys x,x' to very different locations

Universal hashing

- Problem: If hash function is known, then a malicious creator of keys can force O(n) insertion behaviour

- Solution: choose a function h at random from a function class $\mathcal{H} = \{h_1, \dots, h_r\}$

- What properties should that class have?

Universal hashing

- $\mathcal{H}$ is universal collection of hash functions if for every fixed pair of keys $k_1 \neq k_2$ the number functions h causing a collision is bounded as:

$$|\{\, h \in \mathcal{H} \colon h(k_1) = h(k_2)\}| \leq \frac{|\mathcal{H}|}{m}$$

- Theorem: for h drawn randomly from a uniform distribution over a universal collection of hash functions we have O(1+alpha) insertion time

Importance of that theorem?

- Theorem: for h drawn randomly from a uniform distribution over a universal collection of hash functions we have O(1+alpha) insertion time

- What does that mean for our search = O(1) result obtained by table doubling and amortization analysis?

Importance of that theorem?

- Theorem: for h drawn randomly from a uniform distribution over a universal collection of hash functions we have O(1+alpha) search time

- What does that mean for our search = O(1) result obtained by table doubling and amortization analysis?

- Can exchange in last lecture simple uniform hashing assumption with universal collection of hash functions !!! O(1) still holds!

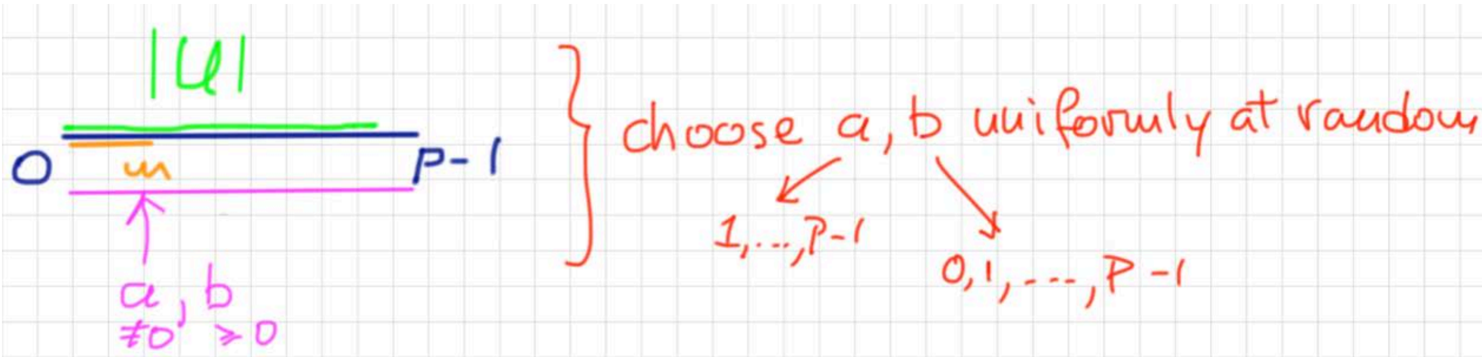
Extra (out of syllabus): example of universal hash function

Universal hash functions

$h(k) = ((ak + b) \bmod p) \bmod m$ m = size of hash table = arbitrary

p = large prime s.t. all keys are in range {0,...,p-1}

$a \in \{1, \ldots, p-1\},\ b \in \{0, \ldots, p-1\}$, are chosen uniformly at random



We can prove: $\text{Prob}_{a,b}\{h_{a,b}(k) = h_{a,b}(l)\} \leq \frac{1}{m},\ \forall k, l$

random choice of *a, b* for each experiment

$\Rightarrow$ less than $\frac{|H|}{m}$ hash functions are "bad" for a pair k, l.

$\rightarrow$ definition of universality!

(proof, out of curriculum)

- $r = (ak_1 + b) mod\ p, s = (ak_2 + b) mod\ p$
- Claim r!=s

By assumption: k1,k2 <=p-1, so k1-k2 in [-(p-1),p-1], so k1-k2 mod p != 0, then. Also a mod p !=0, therefore (p must be prime for that)

a (k1-k2) mod p !=0 … and this is r-s mod p.

(a,b) with a!=0 <-> r,s with r!=s (p(p-1) elements) because a= (r-s)(k1-k2)^{-1} (this inverse exists in Z/Zp)

… its in CLRS, you need to know about multiplication of equivalence classes in Rings Z/Zp and fields (if p is a prime number, then the ring Z/Zp is a field, i.e. every class has an inverse …. A bit mathy+technical stuff)

Example of usage

- Built into most modern programming languages (Python, Perl, Java, C++,…)
- Example:
  - English dictionary for spelling corrections, definitions
  - Compilers: symbol tables (list of names and related info)
  - Network routers: port number -> socket id
  - virtual memory: virtual address -> physical address

Conclusions

- Hashing is an efficient way to keep average cost of operations to $O(1)$
- Collisions are unavoidable in practice and are solved by chaining
- Worse case $\Theta(n)$
- We have some simple ways to construct “good” hash functions

- Hashes are not first choice if worst case behaviour is important
  - Better worst case: Van Emde Boas Trees O(log log n) [not covered here]
- “Hashes kill caches”