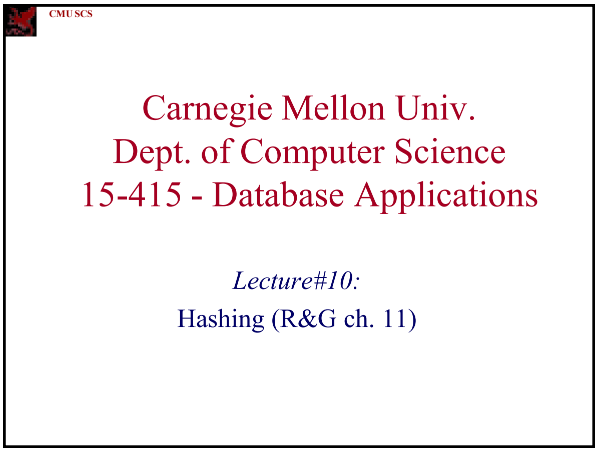
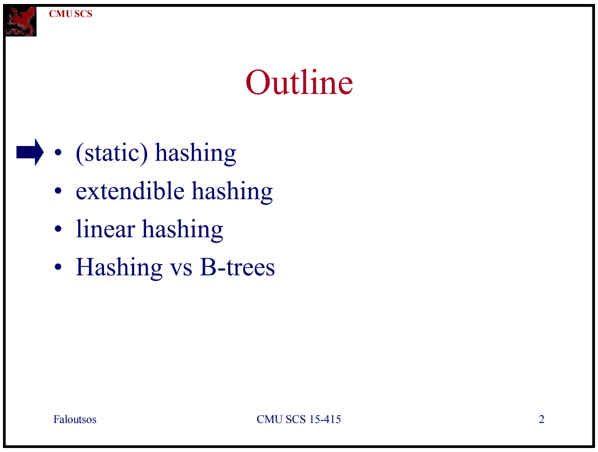


CMU SCS

Carnegie Mellon Univ. Dept. of Computer Science 15-415 - Database Applications

Lecture#10:

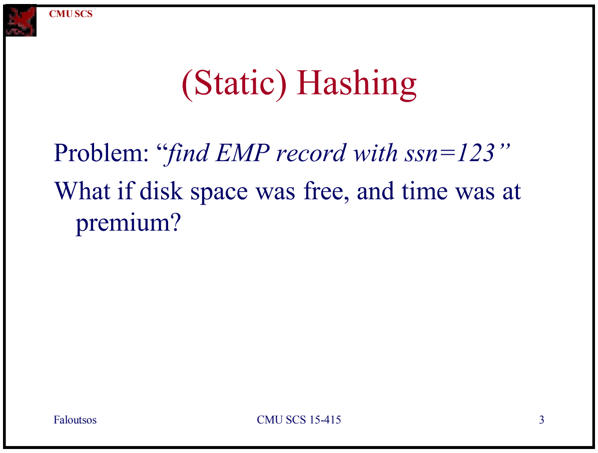
Hashing (R&G ch. 11)



CMU SCS

Outline

- (static) hashing
- extendible hashing
- linear hashing
- Hashing vs B-trees

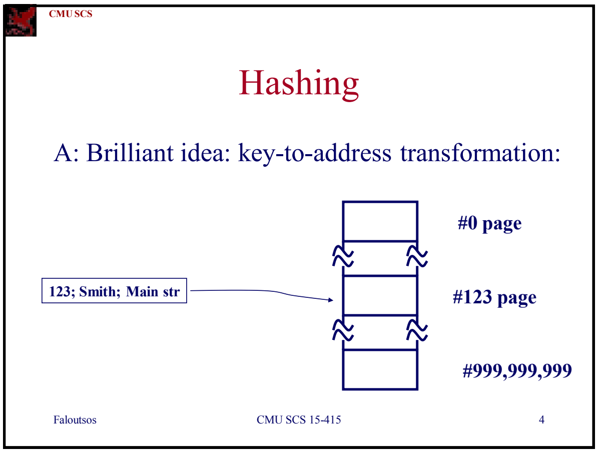


CMU SCS

(Static) Hashing

Problem: “*find EMP record with ssn=123*”

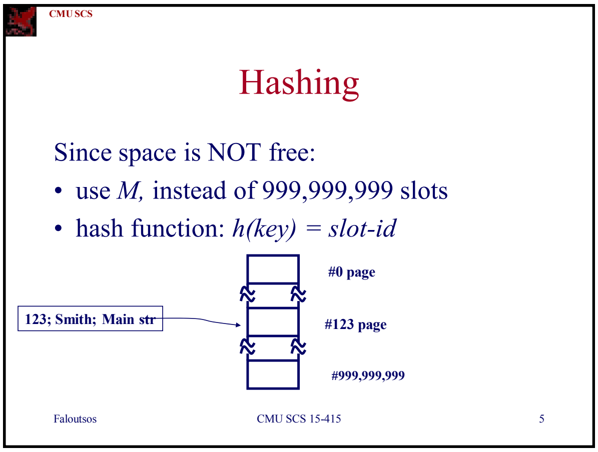
What if disk space was free, and time was at premium?



CMU SCS

Hashing

A: Brilliant idea: key-to-address transformation:

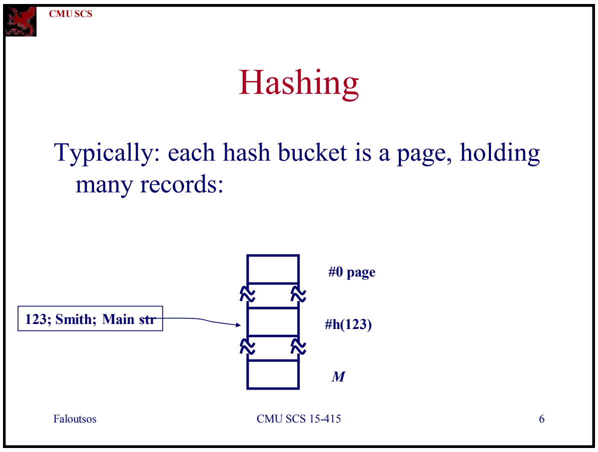


CMU SCS

Hashing

Since space is NOT free:

- use *M,* instead of 999,999,999 slots
- hash function: *h(key) = slot-id*



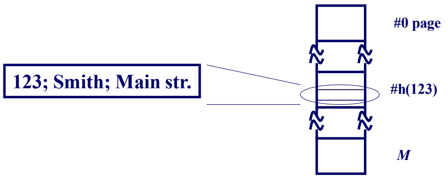
CMU SCS

Hashing

Typically: each hash bucket is a page, holding many records:

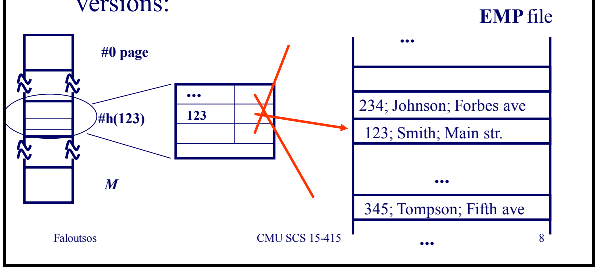
Hashing

Notice: could have **clustering**, or non-clustering versions:



Hashing

Notice: could have clustering, or **non-clustering** versions:



Indexing- overview

- hashing
  - hashing functions
  - size of hash table
  - collision resolution
- extendible hashing
- Hashing vs B-trees

Design decisions

1) formula *h()* for hashing function

2) size of hash table *M*

3) collision resolution method

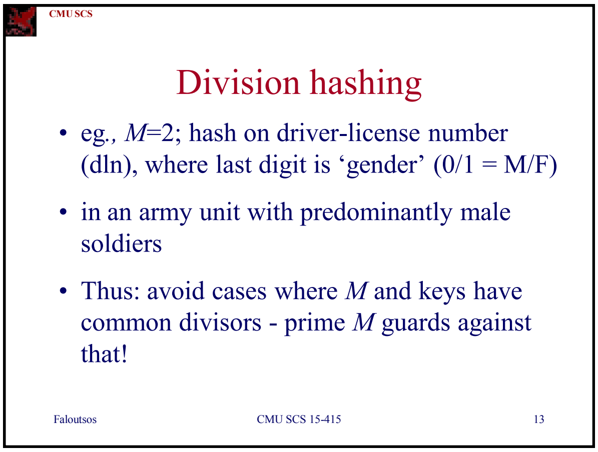
Design decisions - functions

- Goal: uniform spread of keys over hash buckets
- Popular choices:
  - Division hashing
  - Multiplication hashing

Division hashing

$h(x) = (a*x+b) \ \mathrm{mod} \ M$

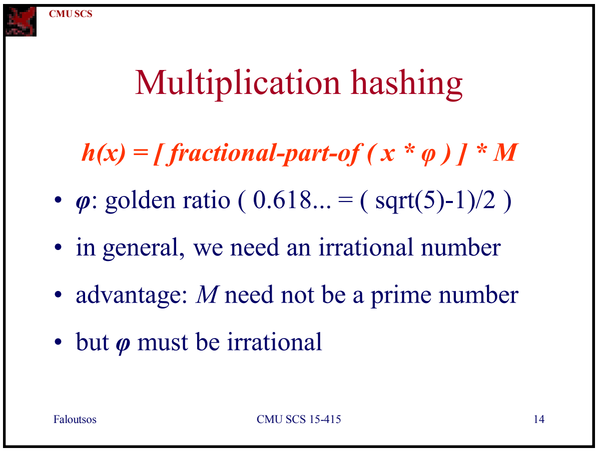
- eg., h(ssn) = (ssn) mod 1,000
  - gives the last three digits of ssn
- M: size of hash table - choose a prime number, defensively (why?)



CMU SCS

Division hashing

- eg., *M*=2; hash on driver-license number (dln), where last digit is ‘gender’ (0/1 = M/F)
- in an army unit with predominantly male soldiers
- Thus: avoid cases where *M* and keys have common divisors - prime *M* guards against that!

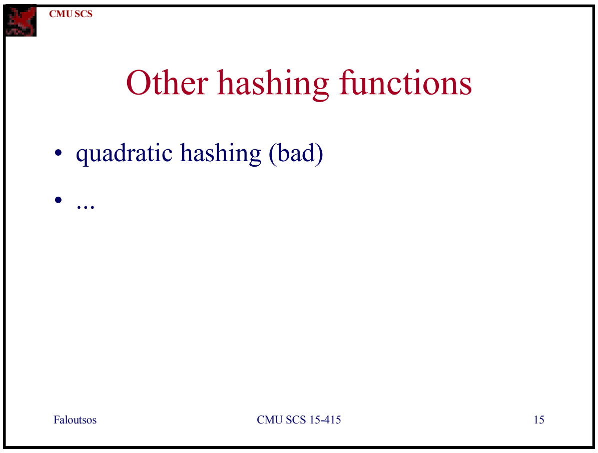


CMU SCS

Multiplication hashing

*h(x) = [fractional-part-of (x * φ)] * M*

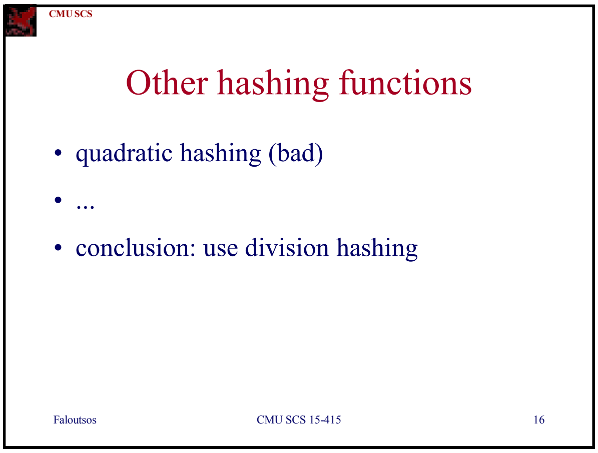
- ***φ***: golden ratio (0.618... = (sqrt(5)-1)/2)
- in general, we need an irrational number
- advantage: *M* need not be a prime number
- but ***φ*** must be irrational



CMU SCS

Other hashing functions

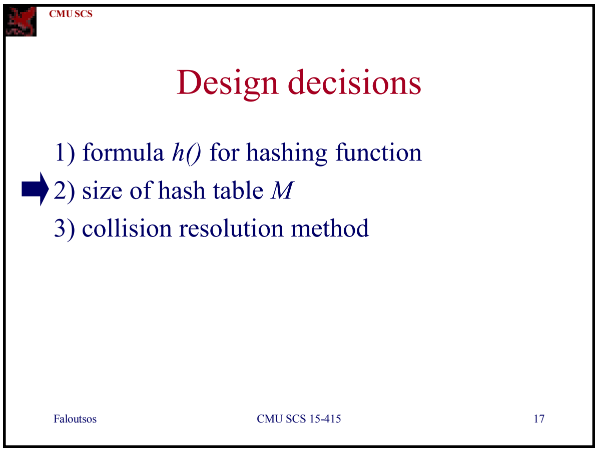
- quadratic hashing (bad)
- ...



CMU SCS

Other hashing functions

- quadratic hashing (bad)
- ...
- conclusion: use division hashing



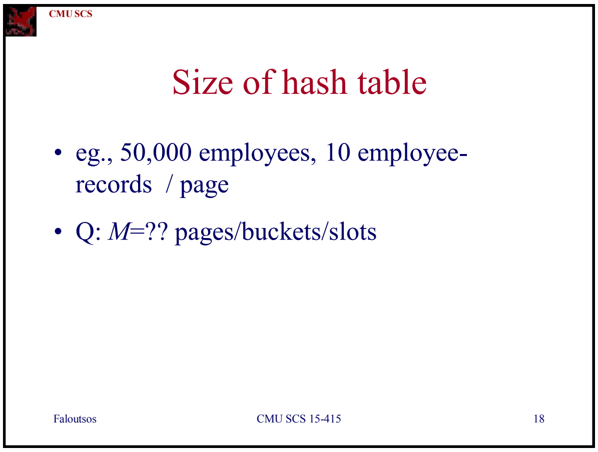
CMU SCS

Design decisions

1) formula *h()* for hashing function

➡ 2) size of hash table *M*

3) collision resolution method



CMU SCS

Size of hash table

- eg., 50,000 employees, 10 employee-records / page
- Q: *M*=?? pages/buckets/slots

Size of hash table

- eg., 50,000 employees, 10 employees/page
- Q: M=?? pages/buckets/slots
- A: utilization ~ 90% and
  - M: prime number

Eg., in our case: M= closest prime to 50,000/10 / 0.9 = 5,555

Design decisions

1) formula *h()* for hashing function

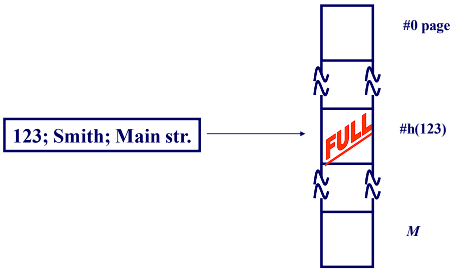
2) size of hash table M

➡ 3) collision resolution method

Collision resolution

- Q: what is a ‘collision’?
- A: ??

Collision resolution

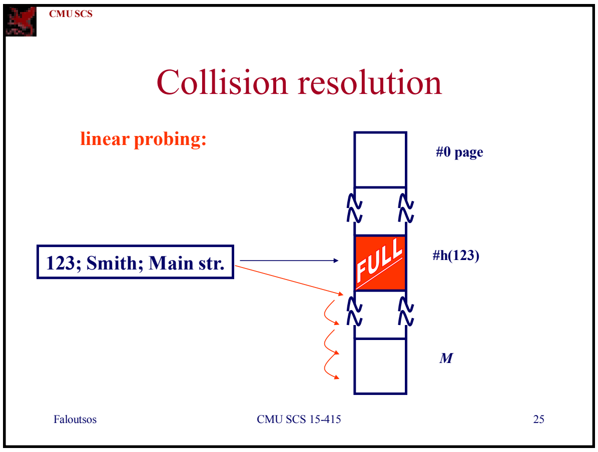


Collision resolution

- Q: what is a ‘collision’?
- A: ??
- Q: why worry about collisions/overflows? (recall that buckets are ~90% full)
- A: ‘birthday paradox’

Collision resolution

- open addressing
  - linear probing (ie., put to next slot/bucket)
  - re-hashing
- separate chaining (ie., put links to overflow pages)



Collision resolution

linear probing:

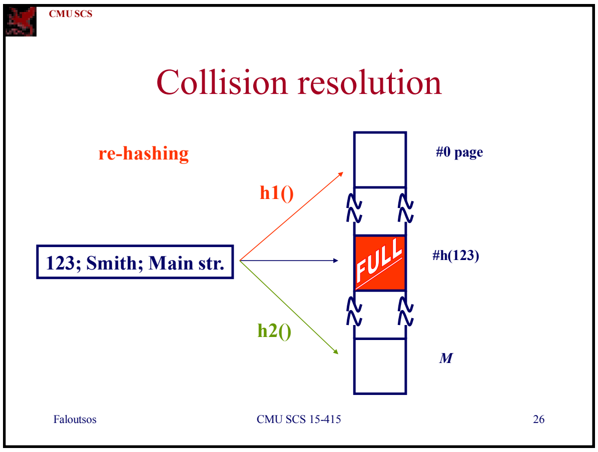
123; Smith; Main str.

FULL

#0 page

#h(123)

M



Collision resolution

re-hashing

123; Smith; Main str.

h1()

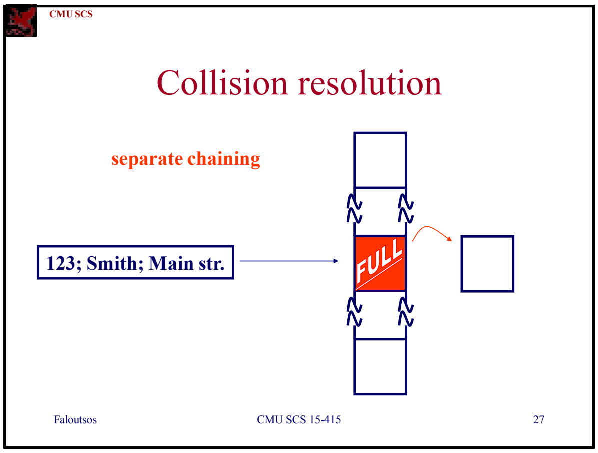
h2()

FULL

#0 page

#h(123)

M

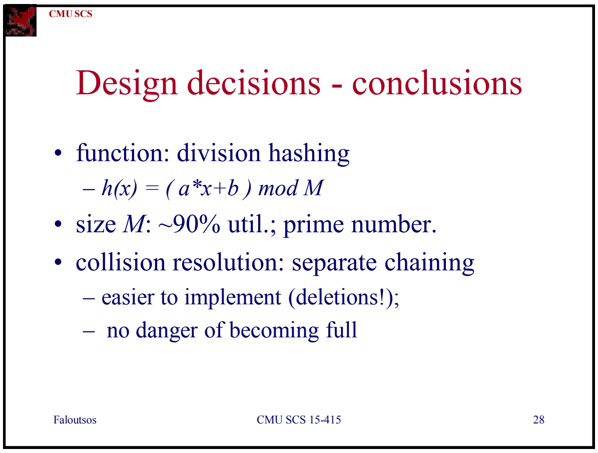


Collision resolution

separate chaining

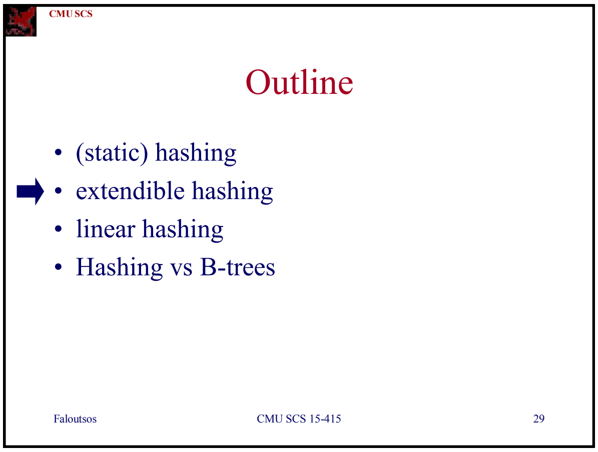
123; Smith; Main str.

FULL



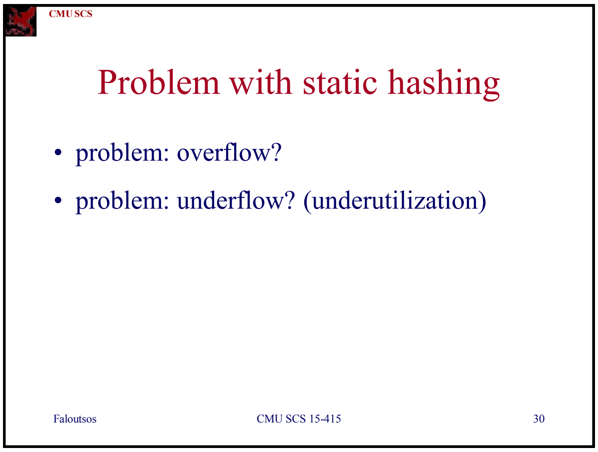
Design decisions - conclusions

- function: division hashing
  - *h(x) = (a*x+b) mod M*
- size *M*: ~90% util.; prime number.
- collision resolution: separate chaining
  - easier to implement (deletions!);
  - no danger of becoming full



Outline

- (static) hashing
- ➡ extendible hashing
- linear hashing
- Hashing vs B-trees



Problem with static hashing

- problem: overflow?
- problem: underflow? (underutilization)

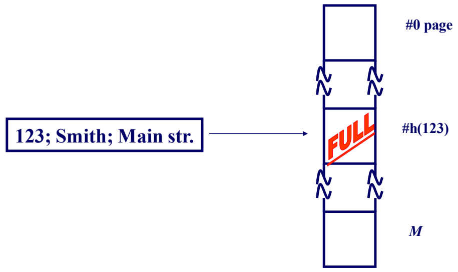
Solution: Dynamic/extendible hashing

- idea: shrink / expand hash table on demand..
- ..dynamic hashing

Details: how to grow gracefully, on overflow?

Many solutions - One of them: ‘extendible hashing’ [Fagin et al]

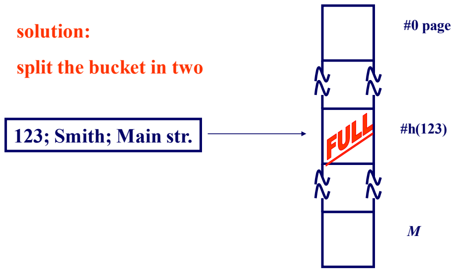
Extendible hashing



Extendible hashing

solution:

split the bucket in two



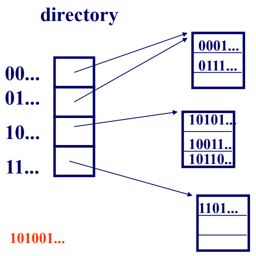
Extendible hashing

in detail:

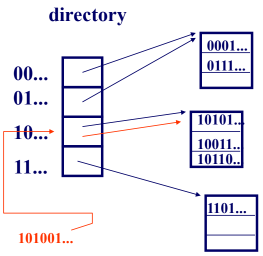
- keep a directory, with ptrs to hash-buckets
- Q: how to divide contents of bucket in two?
- A: hash each key into a very long bit string; keep only as many bits as needed

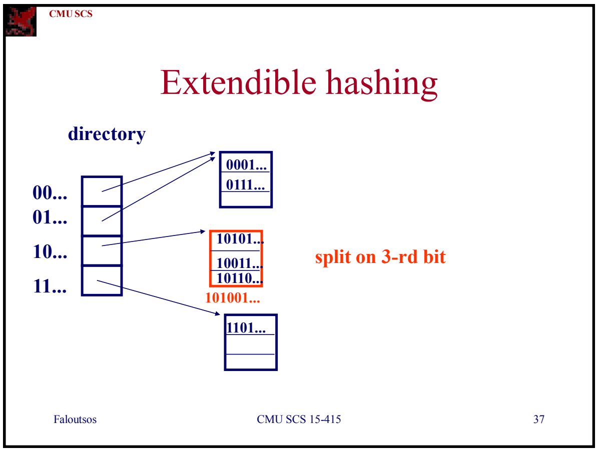
Eventually:

Extendible hashing



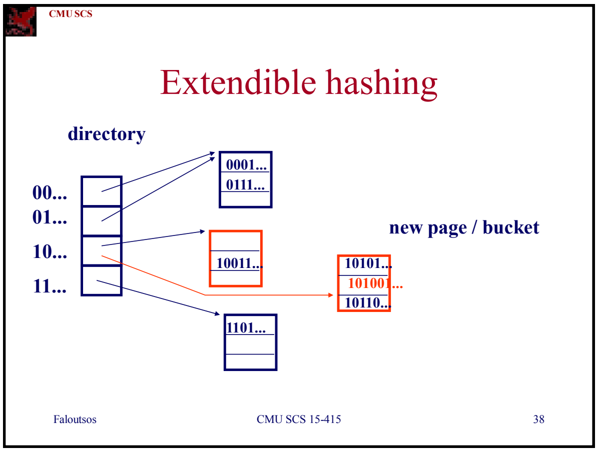
Extendible hashing





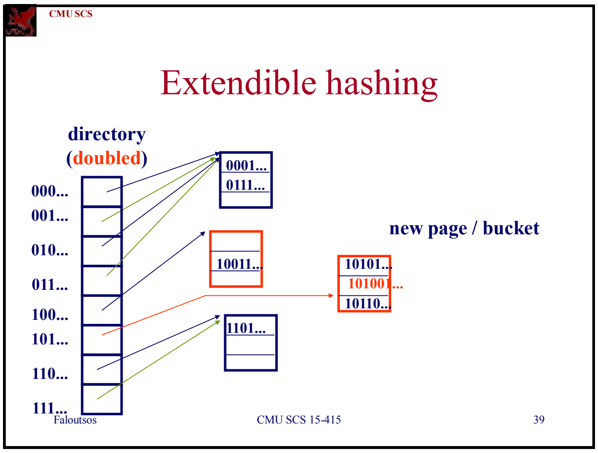
CMU SCS

Extendible hashing



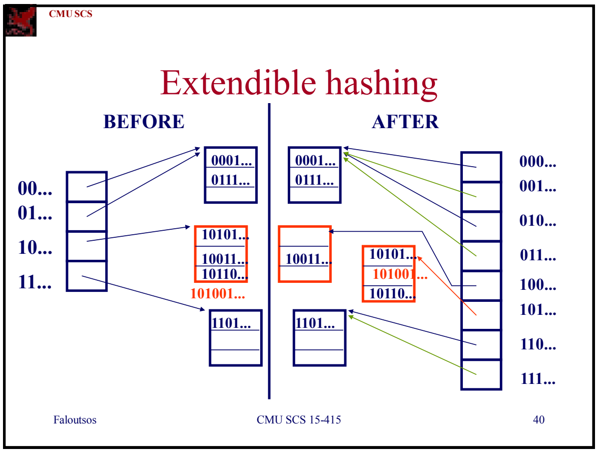
CMU SCS

Extendible hashing



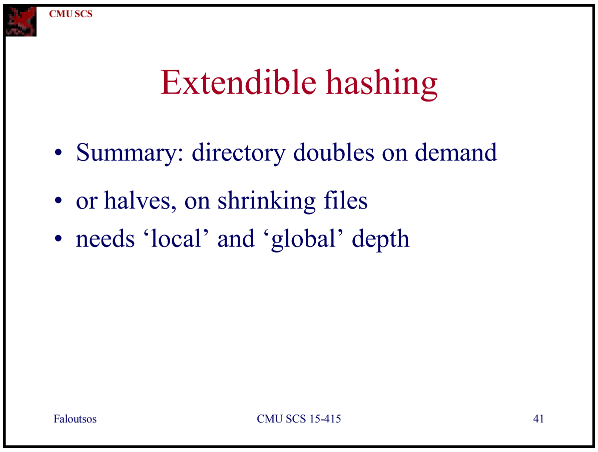
CMU SCS

Extendible hashing



CMU SCS

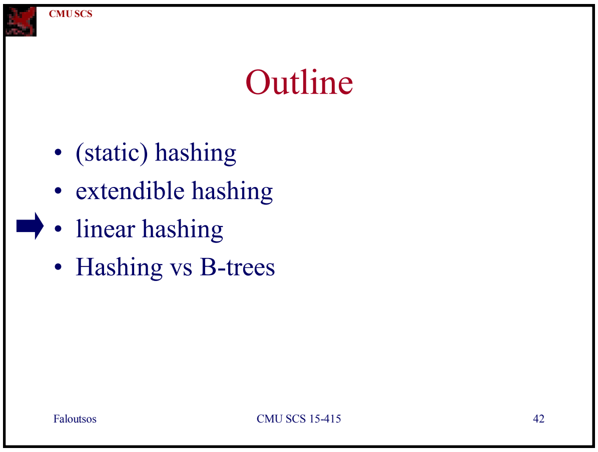
Extendible hashing



CMU SCS

Extendible hashing

- Summary: directory doubles on demand
- or halves, on shrinking files
- needs ‘local’ and ‘global’ depth



CMU SCS

Outline

- (static) hashing
- extendible hashing
- ➡ linear hashing
- Hashing vs B-trees

Linear hashing - overview

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion

Linear hashing

Motivation: ext. hashing needs directory etc etc; which doubles (ouch!)

Q: can we do something simpler, with smoother growth?

Linear hashing

Motivation: ext. hashing needs directory etc etc; which doubles (ouch!)

Q: can we do something simpler, with smoother growth?

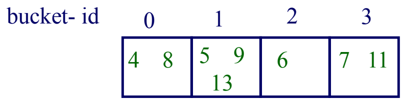
A: split buckets from left to right, **regardless** of which one overflowed (‘crazy’, but it works well!) - Eg.:

Linear hashing

Initially: *h(x) = x mod N* (N=4 here)

Assume capacity: 3 records / bucket

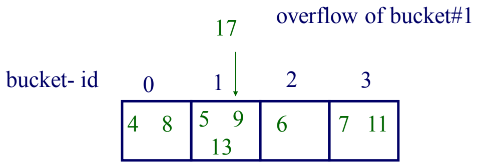
Insert key ‘17’



bucket- id	0	1	2	3
	4 8	5 9 13	6	7 11

Linear hashing

Initially: *h(x) = x mod N* (N=4 here)



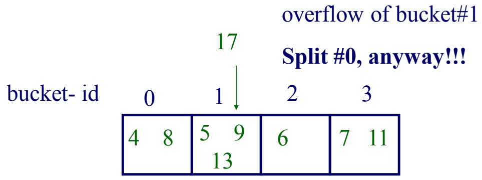
overflow of bucket#1

17

bucket- id	0	1	2	3
	4 8	5 9 13	6	7 11

Linear hashing

Initially: *h(x) = x mod N* (N=4 here)

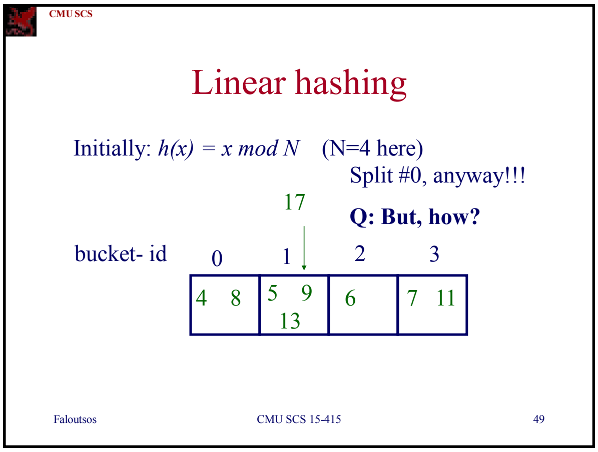


overflow of bucket#1

17

Split #0, anyway!!!

bucket- id	0	1	2	3
	4 8	5 9 13	6	7 11



CMU SCS

Linear hashing

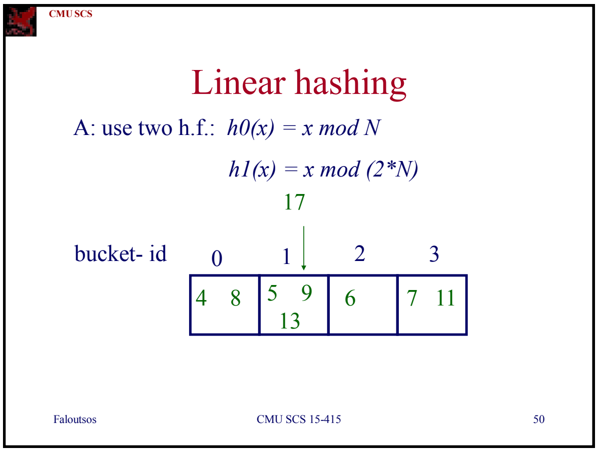
Initially: *h(x) = x mod N* (N=4 here)

Split #0, anyway!!!

Q: But, how?

17

bucket- id	0	1	2	3
	4 8	5 9 13	6	7 11



CMU SCS

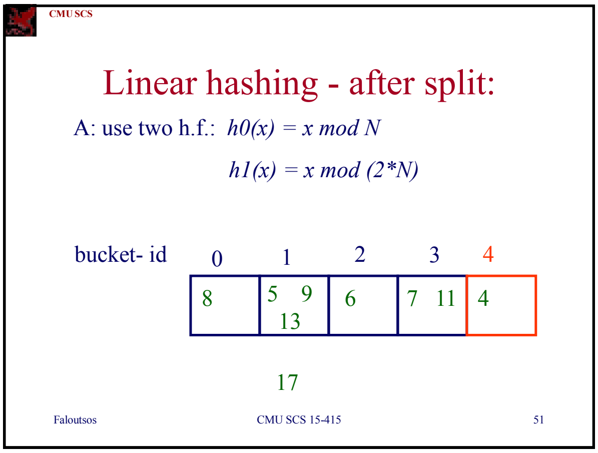
Linear hashing

A: use two h.f.: *h0(x) = x mod N*

*h1(x) = x mod (2*N)*

17

bucket- id	0	1	2	3
	4 8	5 9 13	6	7 11



CMU SCS

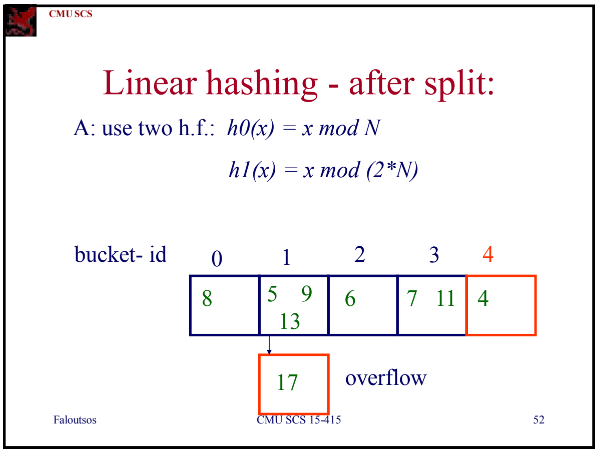
Linear hashing - after split:

A: use two h.f.: *h0(x) = x mod N*

*h1(x) = x mod (2*N)*

bucket- id	0	1	2	3	4
	8	5 9 13	6	7 11	4

17



CMU SCS

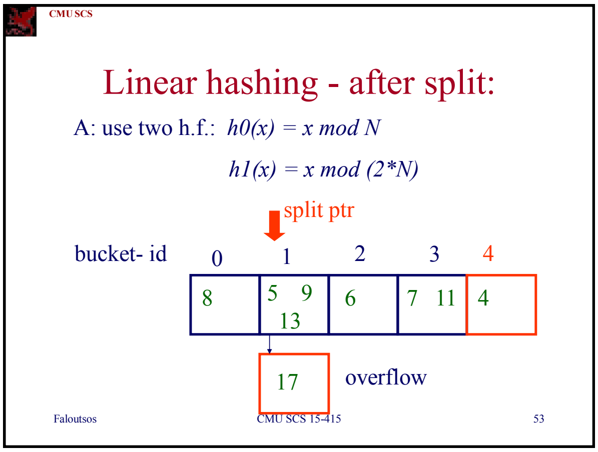
Linear hashing - after split:

A: use two h.f.: *h0(x) = x mod N*

*h1(x) = x mod (2*N)*

bucket- id	0	1	2	3	4
	8	5 9 13	6	7 11	4

17 overflow



CMU SCS

Linear hashing - after split:

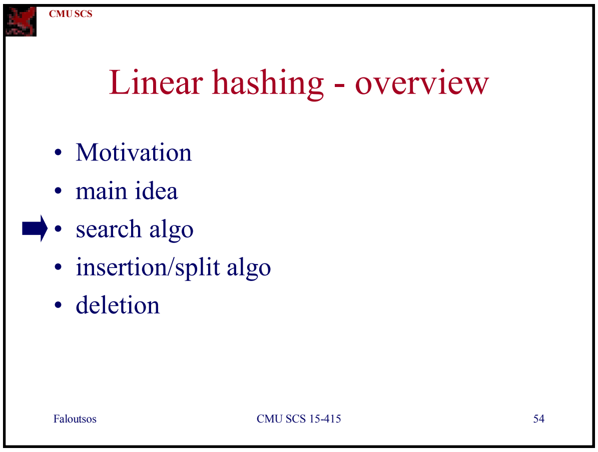
A: use two h.f.: *h0(x) = x mod N*

*h1(x) = x mod (2*N)*

split ptr

bucket- id	0	1	2	3	4
	8	5 9 13	6	7 11	4

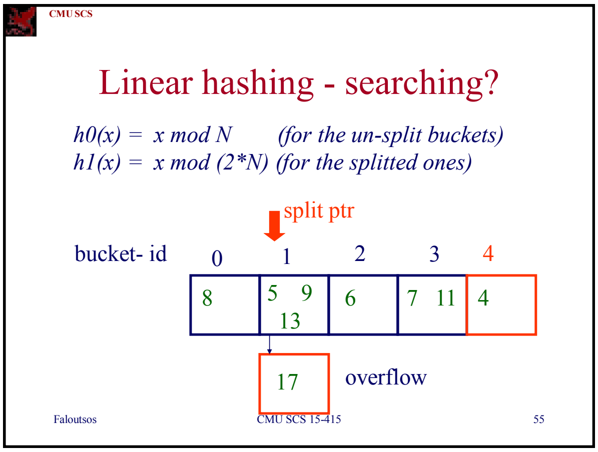
17 overflow



CMU SCS

Linear hashing - overview

- Motivation
- main idea
- ➡ search algo
- insertion/split algo
- deletion

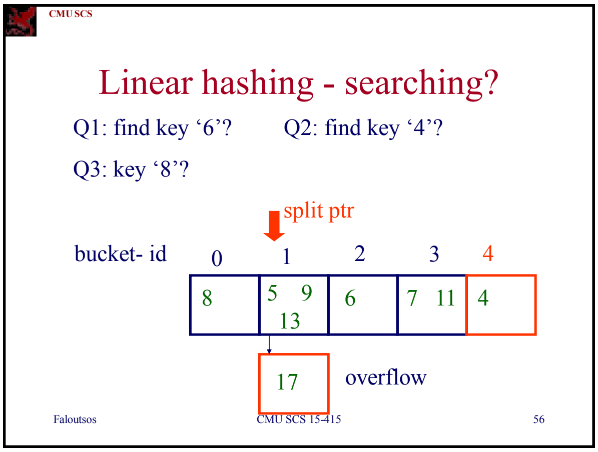


Linear hashing - searching?

h0(x) = x mod N (for the un-split buckets)
*h1(x) = x mod (2*N) (for the splitted ones)*

split ptr

bucket- id	0	1	2	3	4
	8	5 9 13	6	7 11	4
overflow		17			



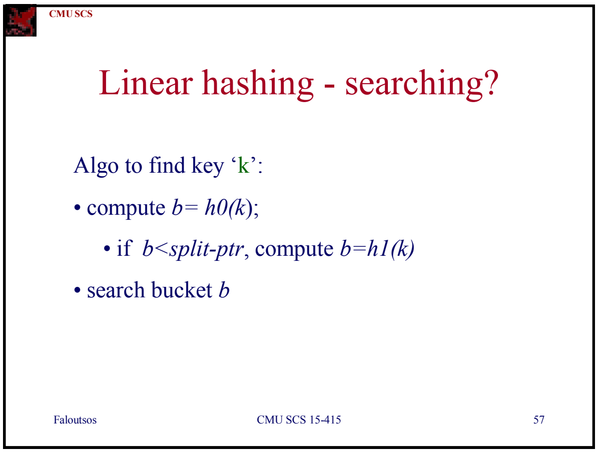
Linear hashing - searching?

Q1: find key ‘6’? Q2: find key ‘4’?

Q3: key ‘8’?

split ptr

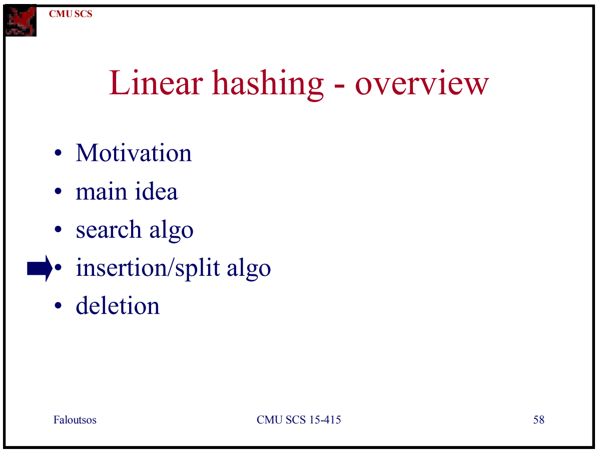
bucket- id	0	1	2	3	4
	8	5 9 13	6	7 11	4
overflow		17			



Linear hashing - searching?

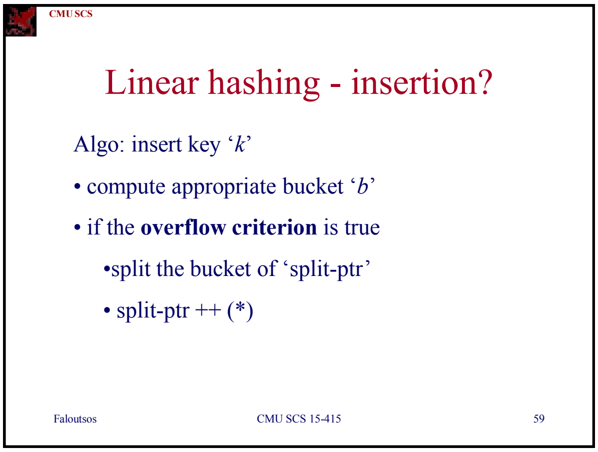
Algo to find key ‘k’:

- compute *b= h0(k)*;
  - if *b<split-ptr*, compute *b=h1(k)*
- search bucket *b*



Linear hashing - overview

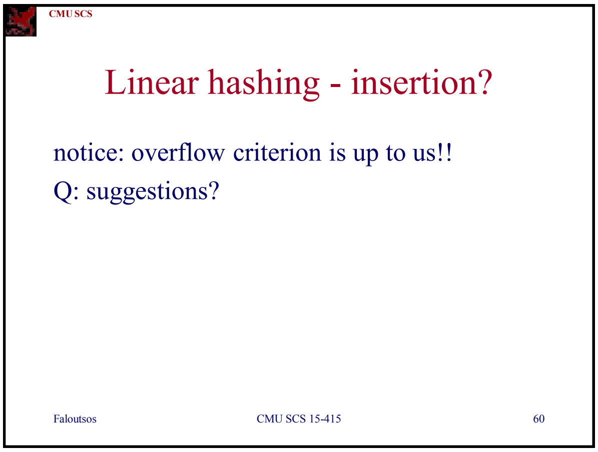
- Motivation
- main idea
- search algo
- ➡ insertion/split algo
- deletion



Linear hashing - insertion?

Algo: insert key ‘*k*’

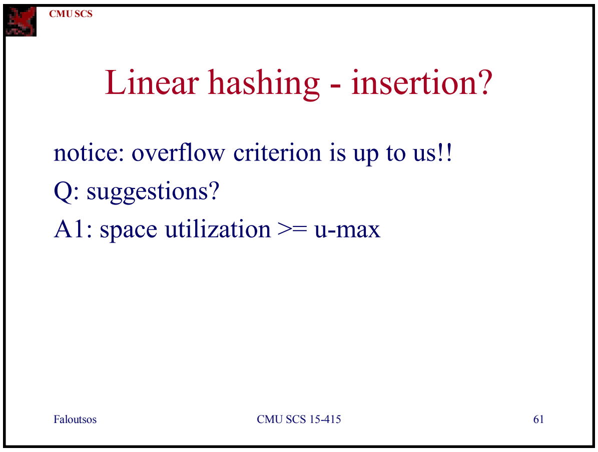
- compute appropriate bucket ‘*b*’
- if the **overflow criterion** is true
  - split the bucket of ‘split-ptr’
  - split-ptr ++ (*)



Linear hashing - insertion?

notice: overflow criterion is up to us!!

Q: suggestions?

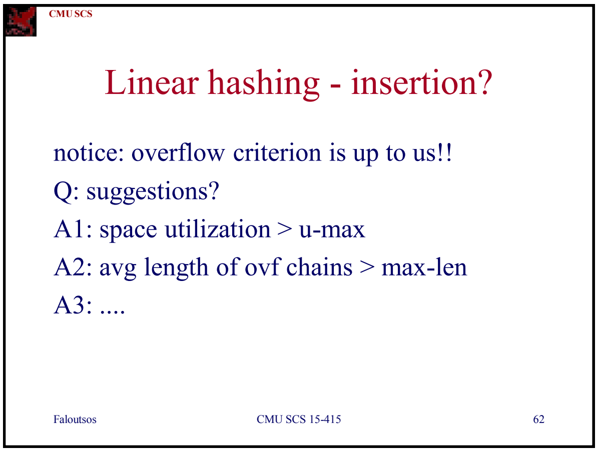


Linear hashing - insertion?

notice: overflow criterion is up to us!!

Q: suggestions?

A1: space utilization >= u-max



Linear hashing - insertion?

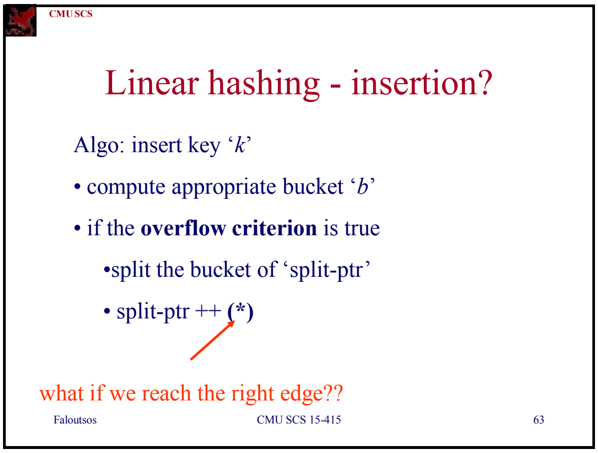
notice: overflow criterion is up to us!!

Q: suggestions?

A1: space utilization > u-max

A2: avg length of ovf chains > max-len

A3: ....

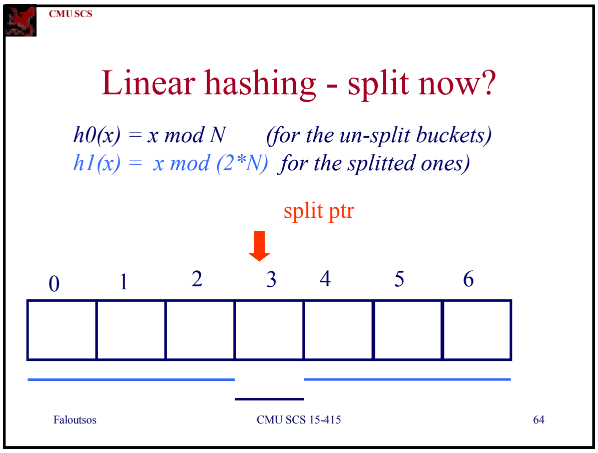


Linear hashing - insertion?

Algo: insert key ‘*k*’

- compute appropriate bucket ‘*b*’
- if the **overflow criterion** is true
  - split the bucket of ‘split-ptr’
  - split-ptr ++ **(*)**

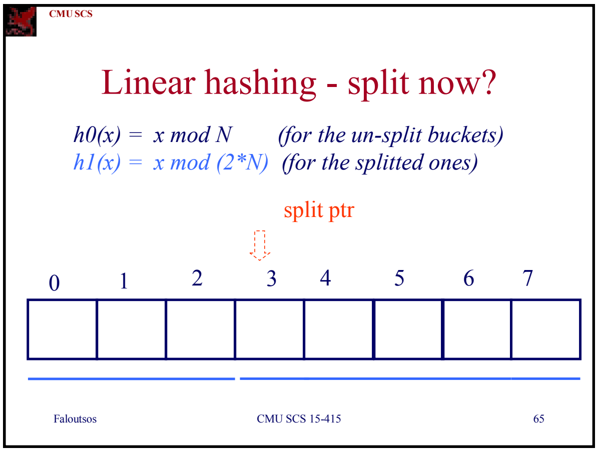
what if we reach the right edge??



Linear hashing - split now?

h0(x) = x mod N *(for the un-split buckets)*

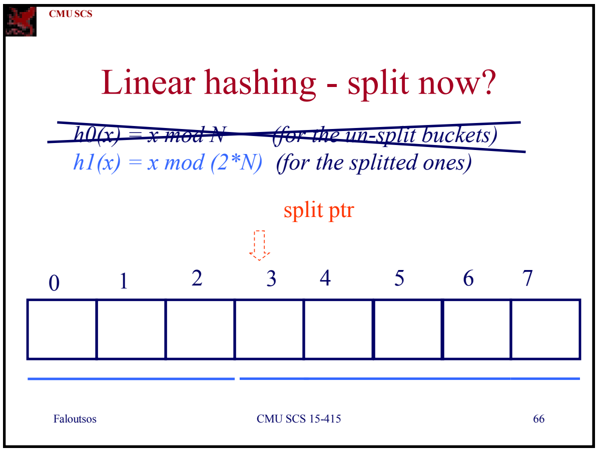
*h1(x) = x mod (2*N)* *for the splitted ones)*



Linear hashing - split now?

h0(x) = x mod N *(for the un-split buckets)*

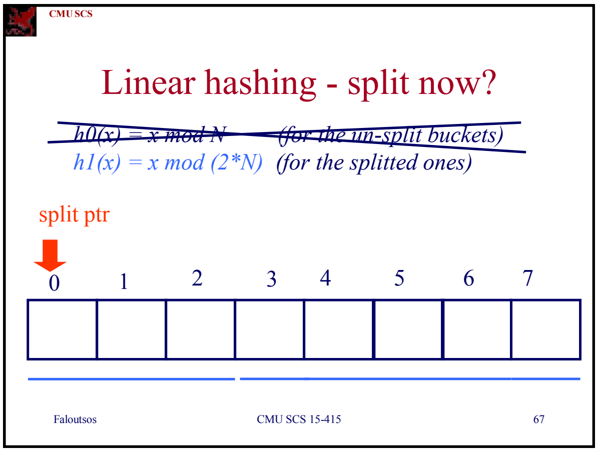
*h1(x) = x mod (2*N)* *(for the splitted ones)*



Linear hashing - split now?

~~*h0(x) = x mod N* *(for the un-split buckets)*~~

*h1(x) = x mod (2*N)* *(for the splitted ones)*

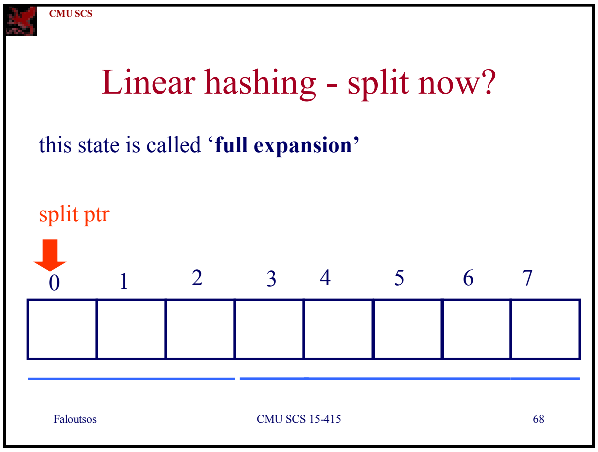


CMU SCS

Linear hashing - split now?

~~$h0(x) = x \bmod N$ (for the un-split buckets)~~

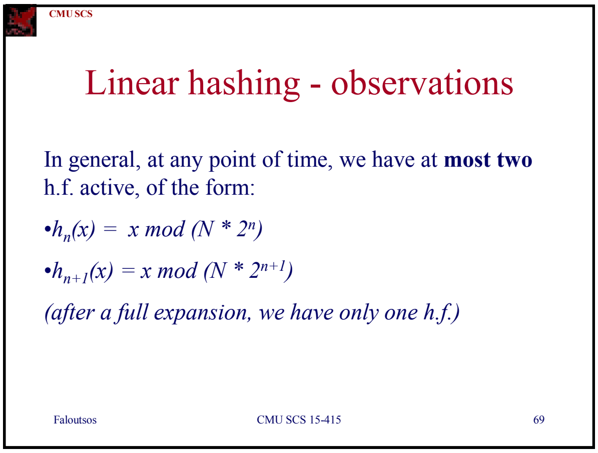
$h1(x) = x \bmod (2*N)$ (for the splitted ones)



CMU SCS

Linear hashing - split now?

this state is called '**full expansion**'



CMU SCS

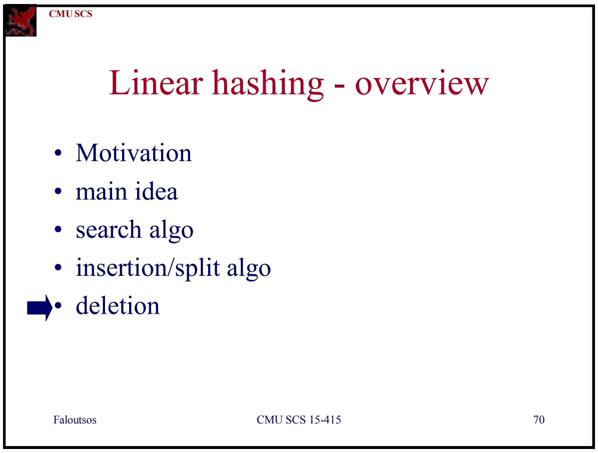
Linear hashing - observations

In general, at any point of time, we have at **most two** h.f. active, of the form:

•$h_n(x) = x \bmod (N * 2^n)$

•$h_{n+1}(x) = x \bmod (N * 2^{n+1})$

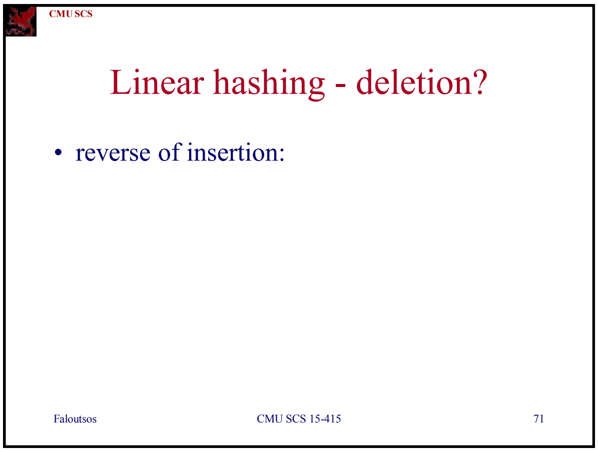
(after a full expansion, we have only one h.f.)



CMU SCS

Linear hashing - overview

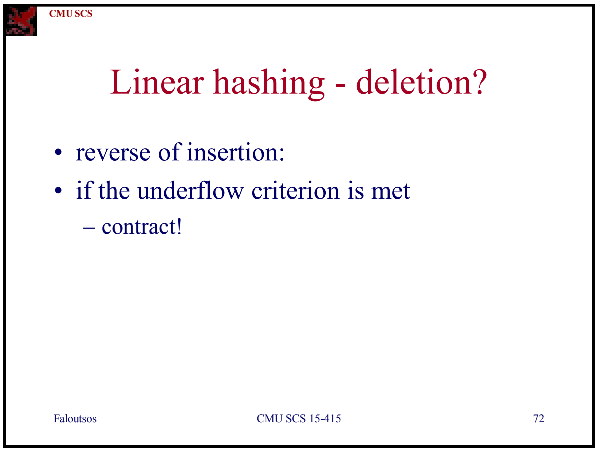
- Motivation
- main idea
- search algo
- insertion/split algo
- ➡ deletion



CMU SCS

Linear hashing - deletion?

- reverse of insertion:



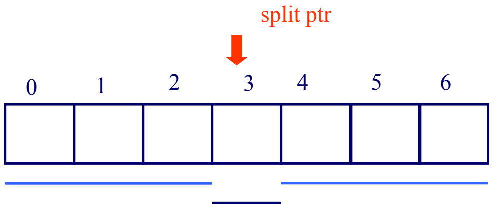
CMU SCS

Linear hashing - deletion?

- reverse of insertion:
- if the underflow criterion is met
  - contract!

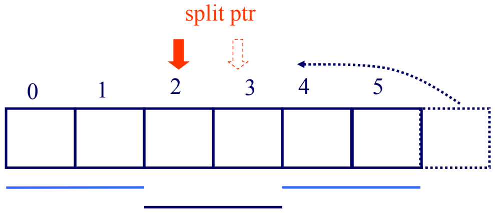
Linear hashing - how to contract?

h0(x) = mod N *(for the un-split buckets)*
*h1(x) = mod (2*N)* *(for the splitted ones)*



Linear hashing - how to contract?

h0(x) = mod N *(for the un-split buckets)*
*h1(x) = mod (2*N)* *(for the splitted ones)*



Outline

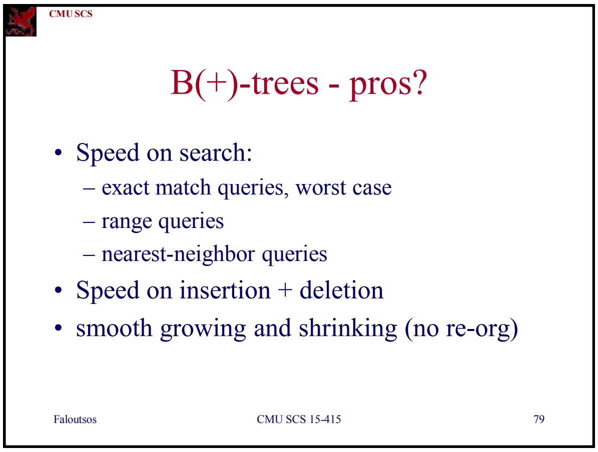
- (static) hashing
- extendible hashing
- linear hashing
- ➡ Hashing vs B-trees

Hashing - pros?

Hashing - pros?

- Speed,
  - on exact match queries
  - on the average

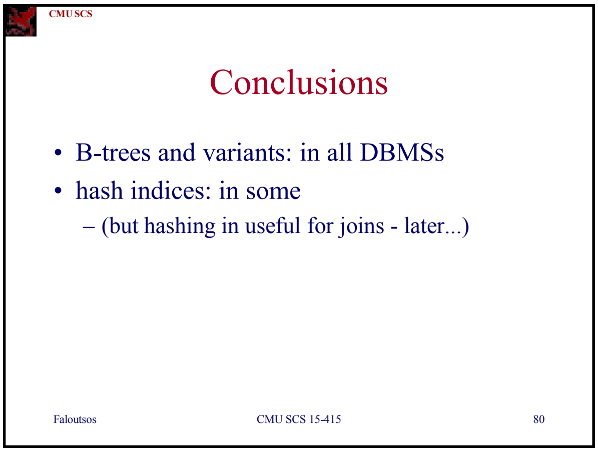
B(+)-trees - pros?



CMU SCS

B(+)-trees - pros?

- Speed on search:
  - exact match queries, worst case
  - range queries
  - nearest-neighbor queries
- Speed on insertion + deletion
- smooth growing and shrinking (no re-org)



CMU SCS

Conclusions

- B-trees and variants: in all DBMSs
- hash indices: in some
  - (but hashing in useful for joins - later...)