Carnegie Mellon

15-826: Multimedia (Databases) and Data Mining

Lecture#3: Primary key indexing – hashing

C. Faloutsos

Reading Material

- [Litwin] Litwin, W., (1980), *Linear Hashing: A New Tool for File and Table Addressing*, VLDB, Montreal, Canada, 1980
- textbook, Chapter 3
- Ramakrinshan+Gehrke, Chapter 11

Outline

Goal: ‘Find similar / interesting things’

- Intro to DB
- Indexing - similarity search
- Data Mining

Indexing - Detailed outline

- primary key indexing
  - B-trees and variants
  - (static) hashing
  - extendible hashing
- secondary key indexing
- spatial access methods
- text
- ...

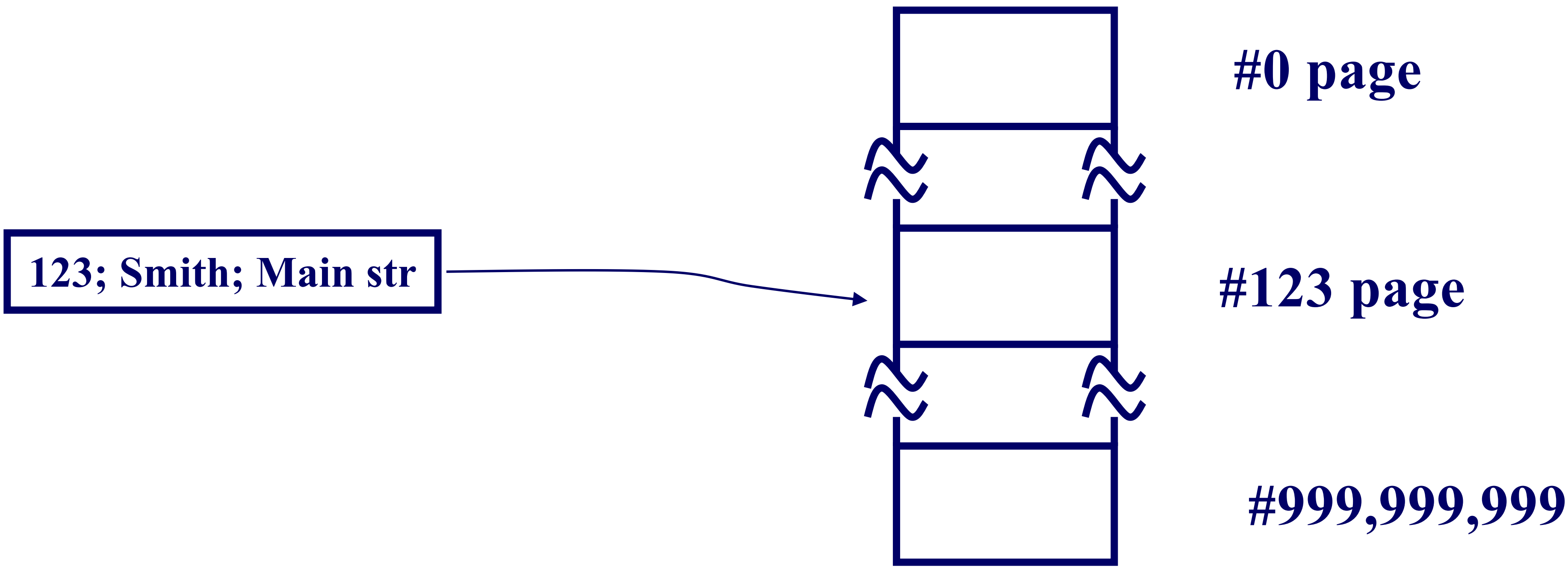
(Static) Hashing

Problem: "*find EMP record with ssn=123*"

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

Hashing

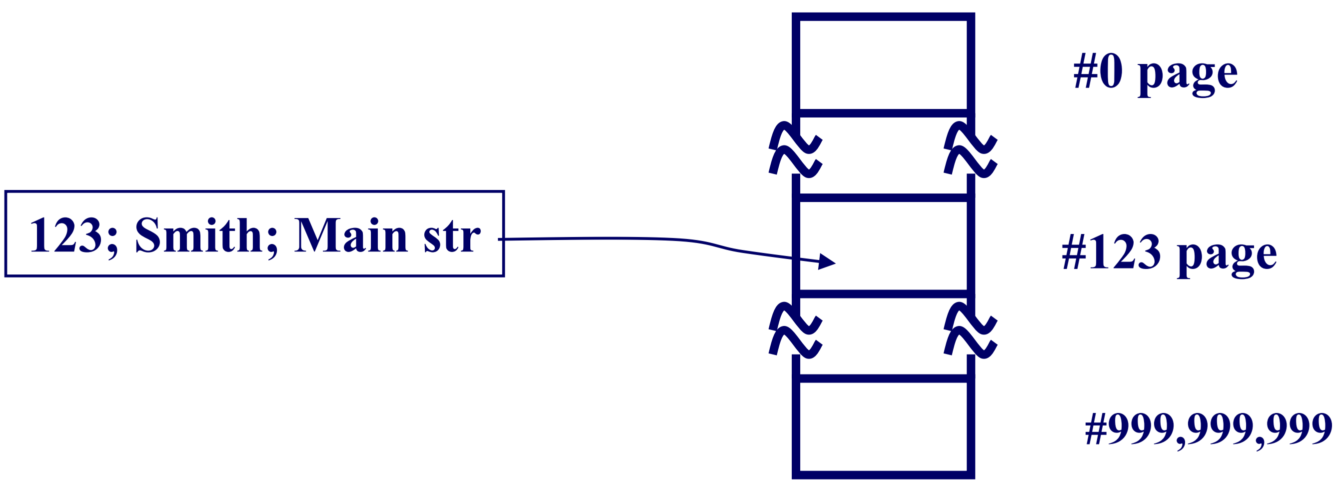
A: Brilliant idea: key-to-address transformation:



Hashing

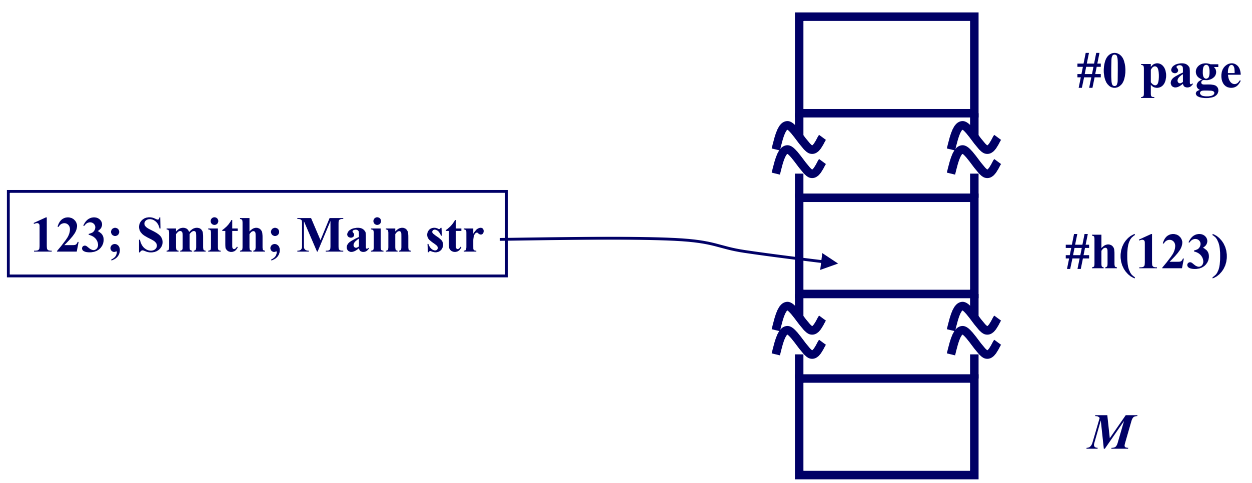
Since space is NOT free:

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



Hashing

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



Hashing - design decisions?

- eg., IRS, 200M tax returns, by SSN

Indexing- overview

- B-trees
- (static) hashing
  - hashing functions
  - size of hash table
  - collision resolution
  - Hashing vs B-trees
  - Indices in SQL
- Extendible hashing

Carnegie Mellon

Design decisions

1) formula *h()* for hashing function
2) size of hash table *M*
3) collision resolution method

Design decisions

1) formula *h()* for hashing function	Division hashing
2) size of hash table *M*	90% utilization
3) collision resolution method	Separate chaining

SKIP

Design decisions - functions

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

SKIP

Division hashing

$$h(x) = (a*x+b) \bmod M$$

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

SKIP

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!

SKIP

Design decisions

1) formula *h()* for hashing function
2) size of hash table *M*
3) collision resolution method

SKIP

Size of hash table

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

SKIP

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

SKIP

Design decisions

1) formula *h()* for hashing function
2) size of hash table *M*
3) collision resolution method

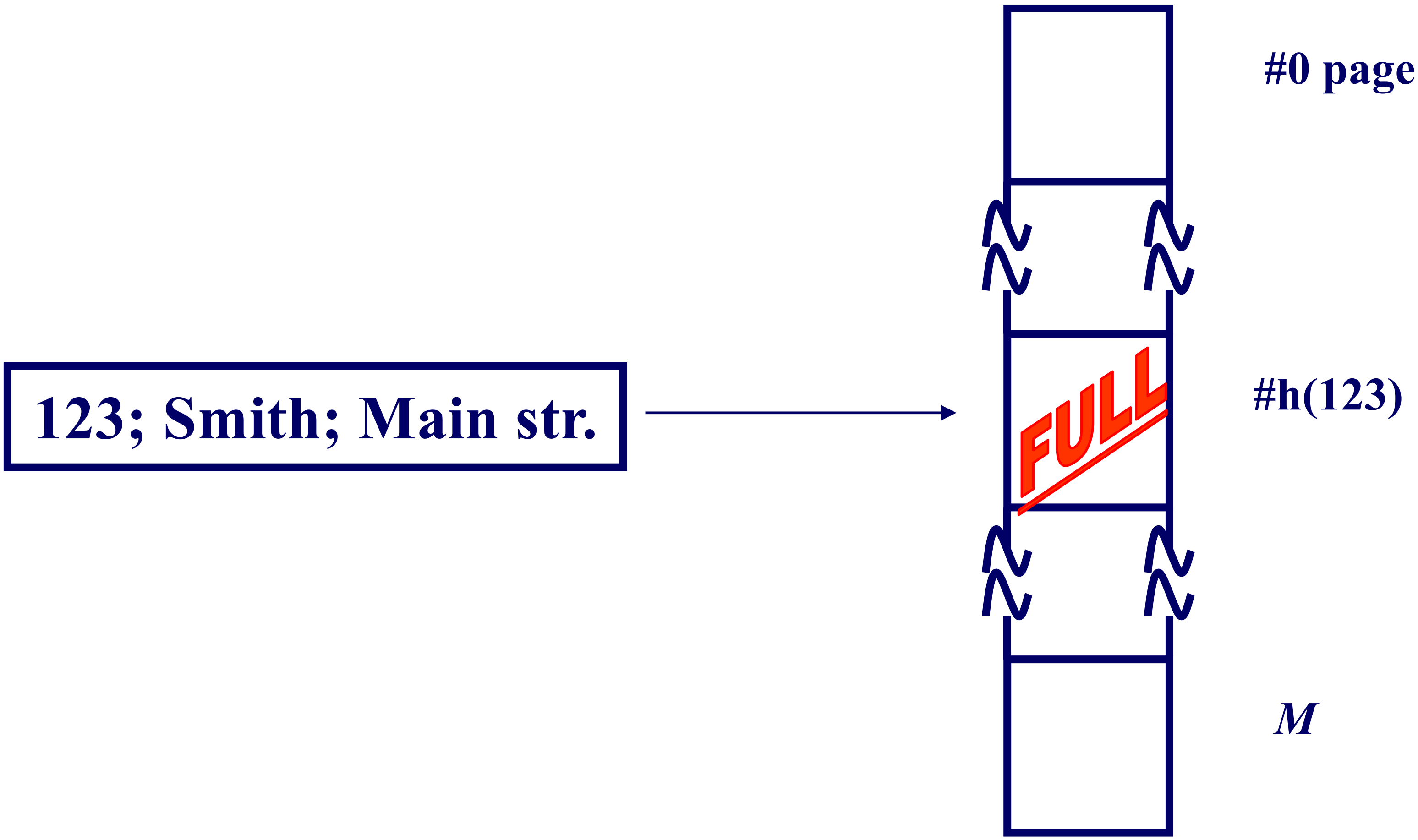
SKIP

Collision resolution

- Q: what is a 'collision'?
- A: ??

SKIP

Collision resolution



SKIP

Collision resolution

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

SKIP

Collision resolution

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

SKIP

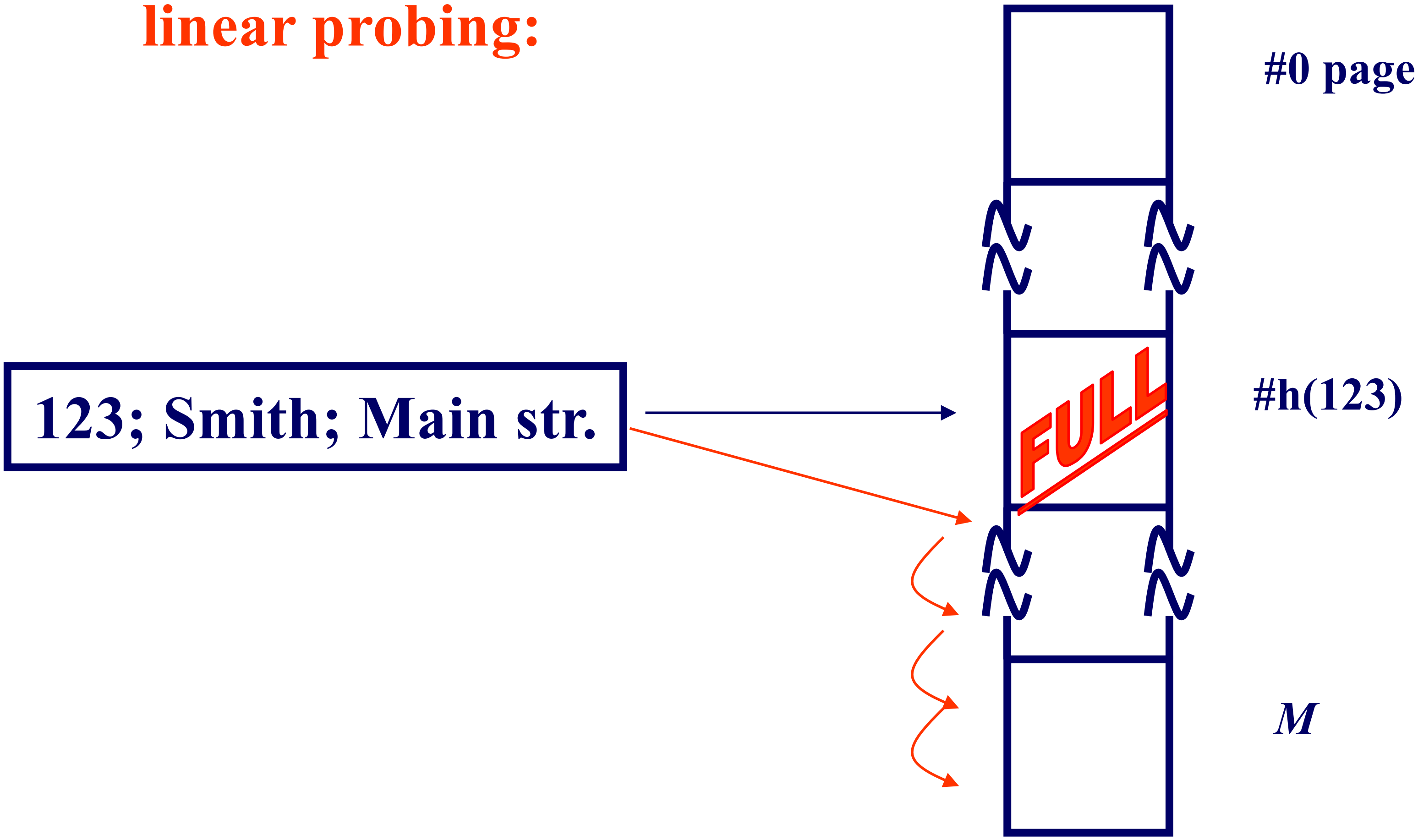
Collision resolution

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

SKIP

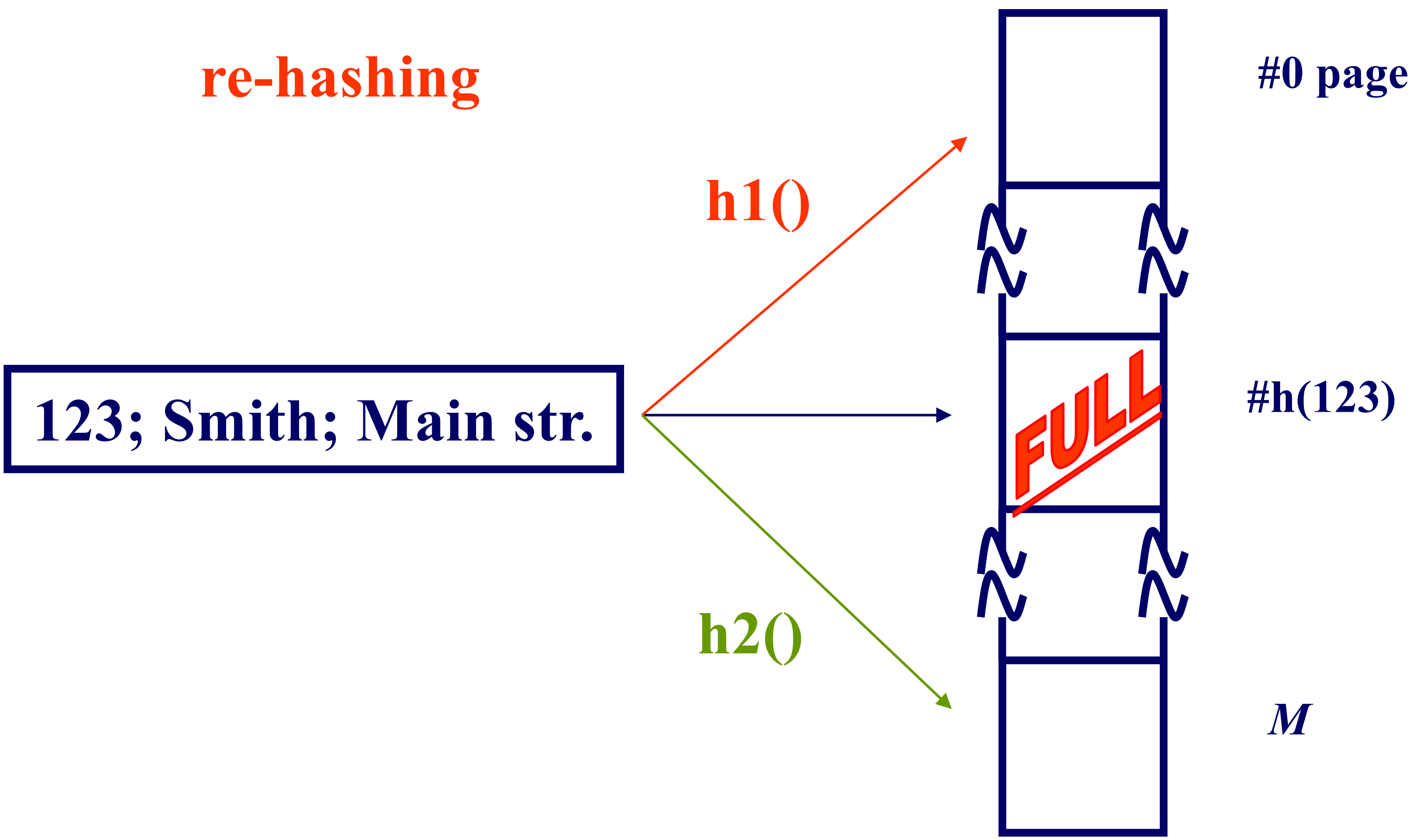
Collision resolution

linear probing:



SKIP

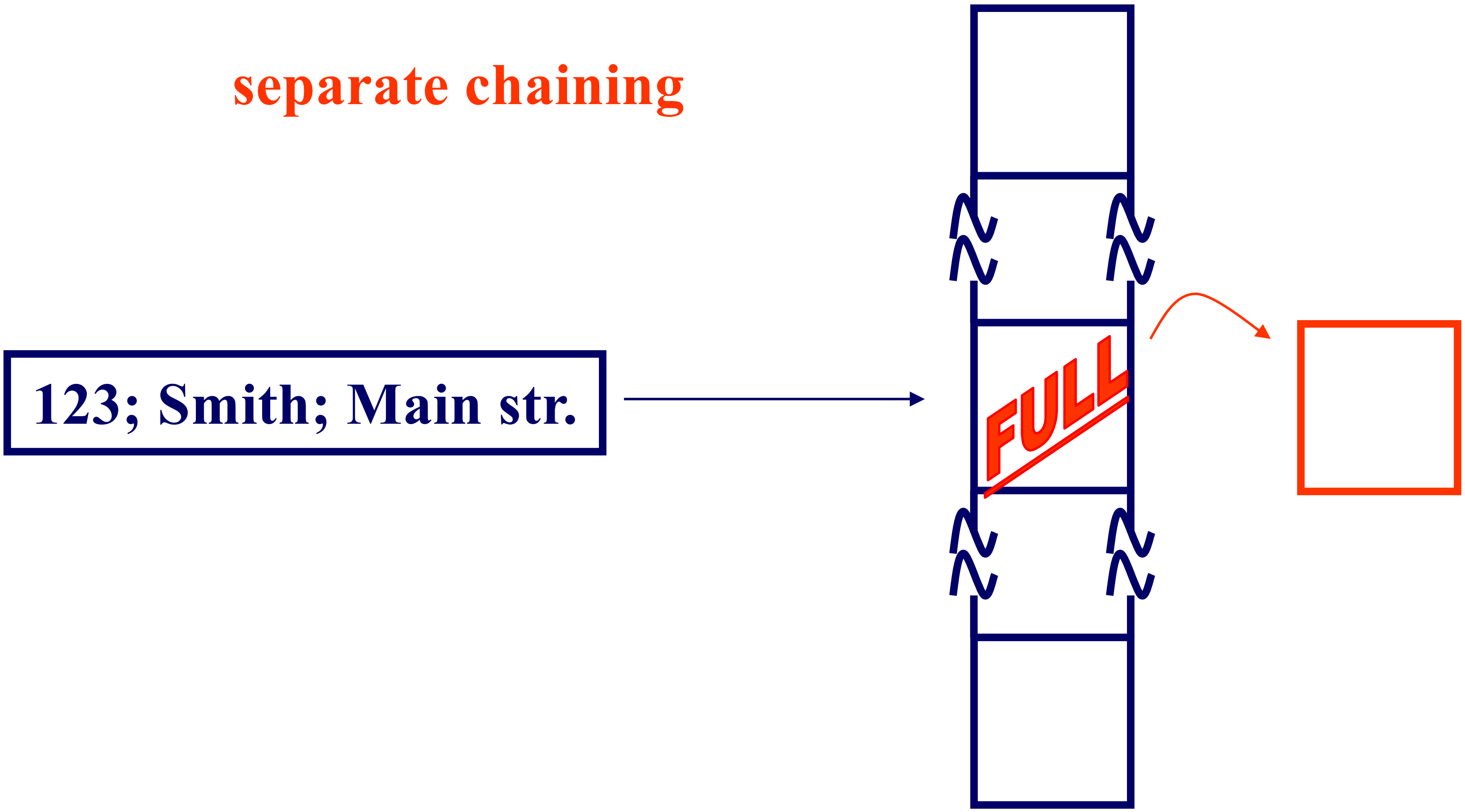
Collision resolution



SKIP

Collision resolution

separate chaining

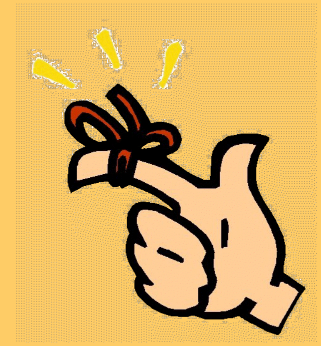


Design decisions - conclusions

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

Indexing- overview

- B-trees
- (static) hashing
  - hashing functions
  - size of hash table
  - collision resolution
  - Hashing vs B-trees
  - Indices in SQL
- Extendible hashing



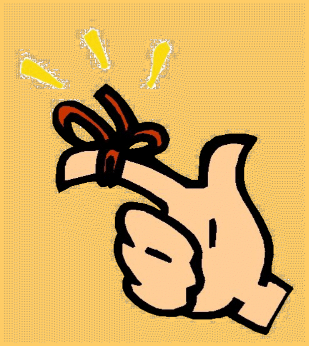
Hashing vs B-trees:

Hashing offers

- speed ! (O(1) **avg**. search time)

..but:

Hashing vs B-trees:



..but B-trees give:

- key ordering:
  - **range queries**
  - **proximity queries**
  - **sequential scan**
- O(log(N)) guarantees for search, ins./del.
- graceful growing/shrinking

Indexing- overview

- B-trees
- (static) Hashing
- extensible hashing
  - **'linear' hashing** [Litwin]

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 – simplest: Linear hashing [Litwin]

Indexing- overview

- B-trees
- Static hashing
- extendible hashing
  - 'extensible' hashing [Fagin, Pipenger +]
  - **'linear' hashing** [Litwin]

Linear hashing - Detailed overview

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
- performance analysis
- variations

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.:

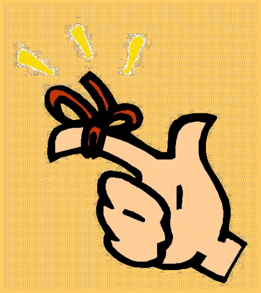
Linear hashing

Initially: $h(x) = x \bmod N$ (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



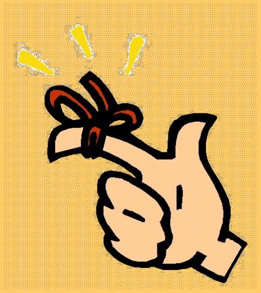
Linear hashing

Initially: $h(x) = x \bmod N$ (N=4 here)

overflow of bucket#1

17 → bucket 1

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



Linear hashing

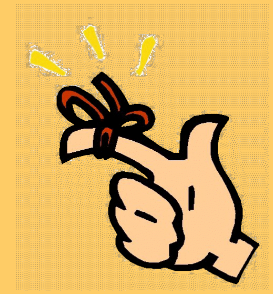
Initially: *h(x) = x mod N* (N=4 here)

overflow of bucket#1

17

Split #0, anyway!!!

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



Linear hashing

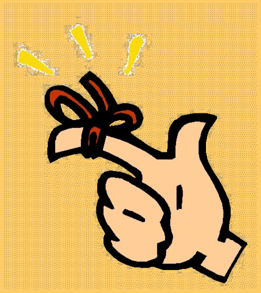
Initially: $h(x) = x \bmod N$ (N=4 here)

Split #0, anyway!!!

17

Q: But, how?

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

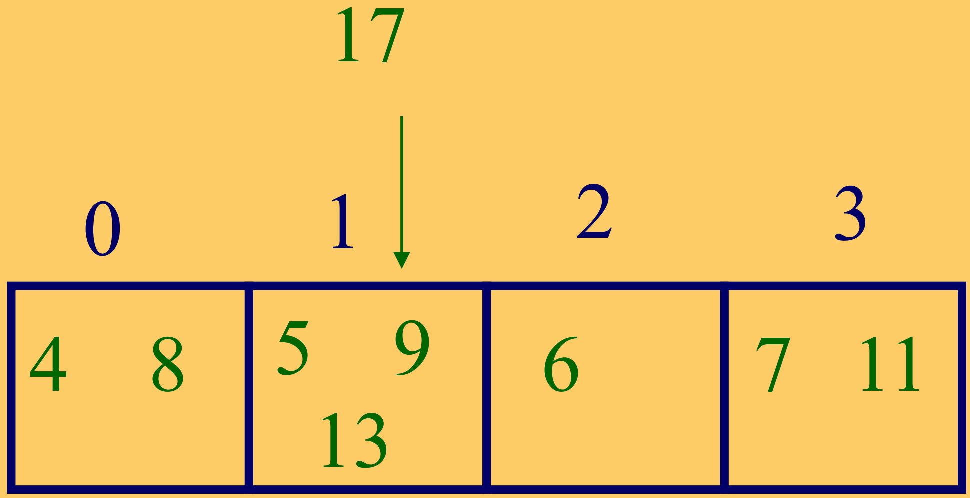


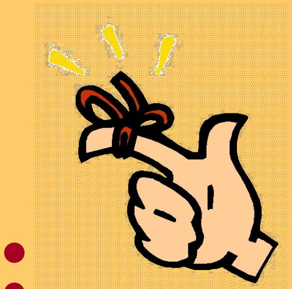
Linear hashing

A: use two h.f.: $h0(x) = x \bmod N$

$h1(x) = x \bmod (2*N)$

bucket- id





Linear hashing - after split:

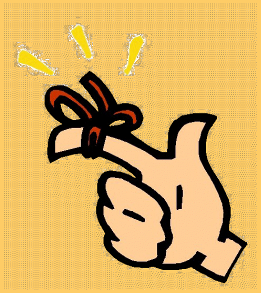
A: use two h.f.: $h0(x) = x \bmod N$

$h1(x) = x \bmod (2*N)$

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

17

Linear hashing - after split:

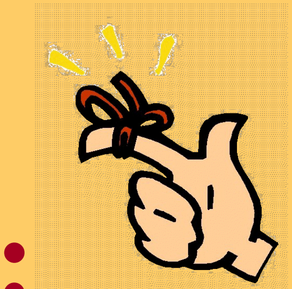


A: use two h.f.: $h0(x) = x \bmod N$

$h1(x) = x \bmod (2*N)$

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

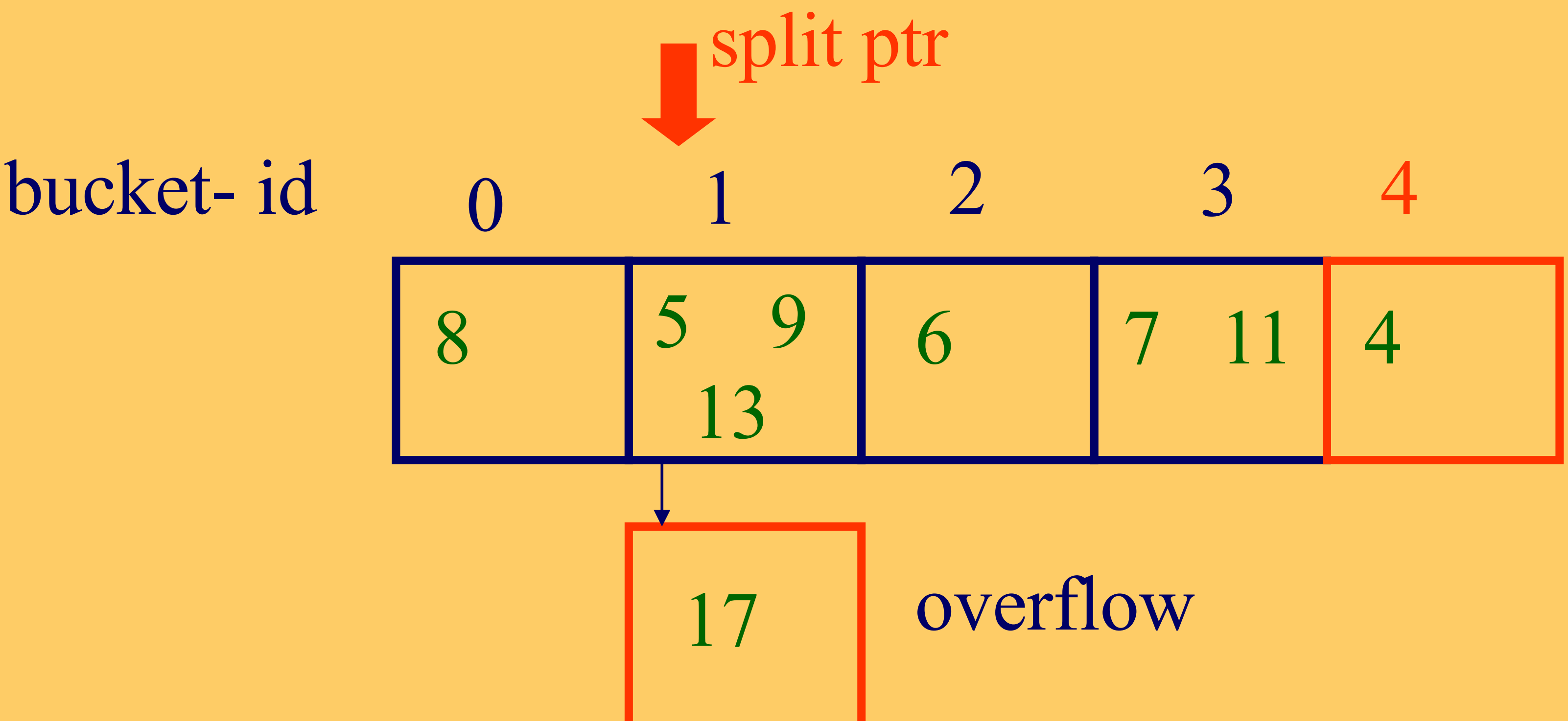
overflow (bucket 1): 17



Linear hashing - after split:

A: use two h.f.: $h0(x) = x \bmod N$

$$h1(x) = x \bmod (2*N)$$

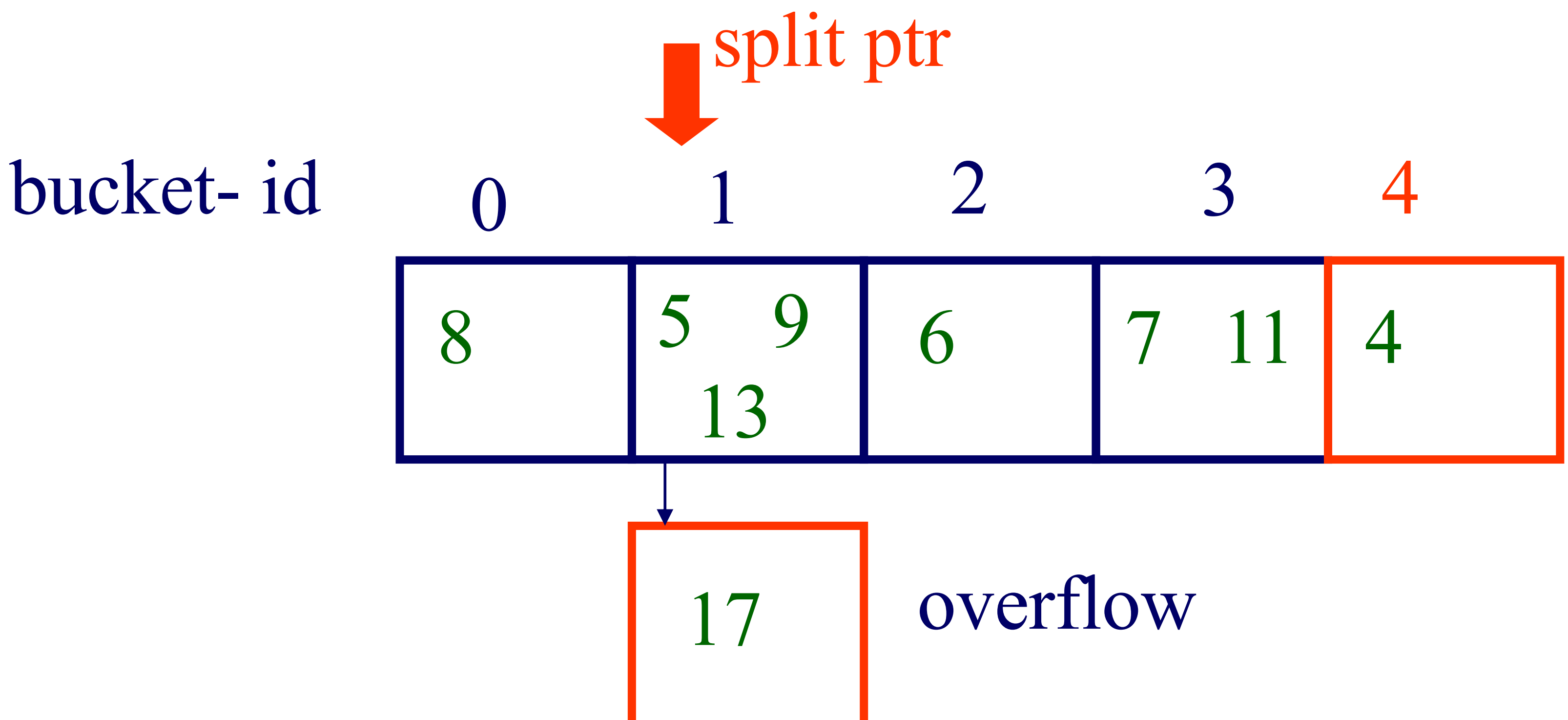


Linear hashing - overview

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
- performance analysis
- variations

Linear hashing - searching?

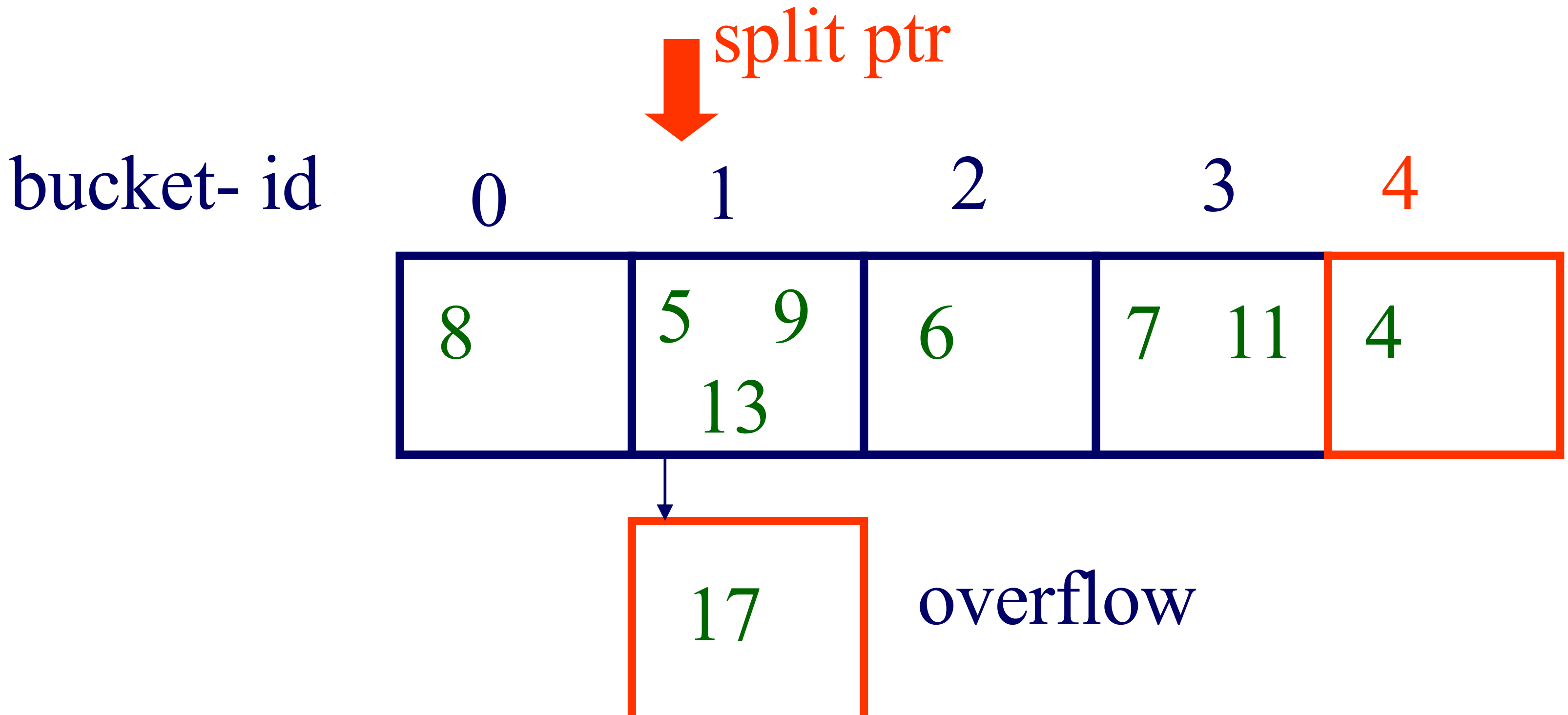
h0(x) = x mod N (for the un-split buckets)
*h1(x) = x mod (2*N) (for the splitted ones)*



Linear hashing - searching?

Q1: find key '6'? Q2: find key '4'?

Q3: key '8'?



Linear hashing - searching?

Algo to find key 'k' :

- compute $b = h0(k)$;
  - if $b < split\text{-}ptr$, compute $b = h1(k)$
- search bucket b

Linear hashing - overview

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
- performance analysis
- variations

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?

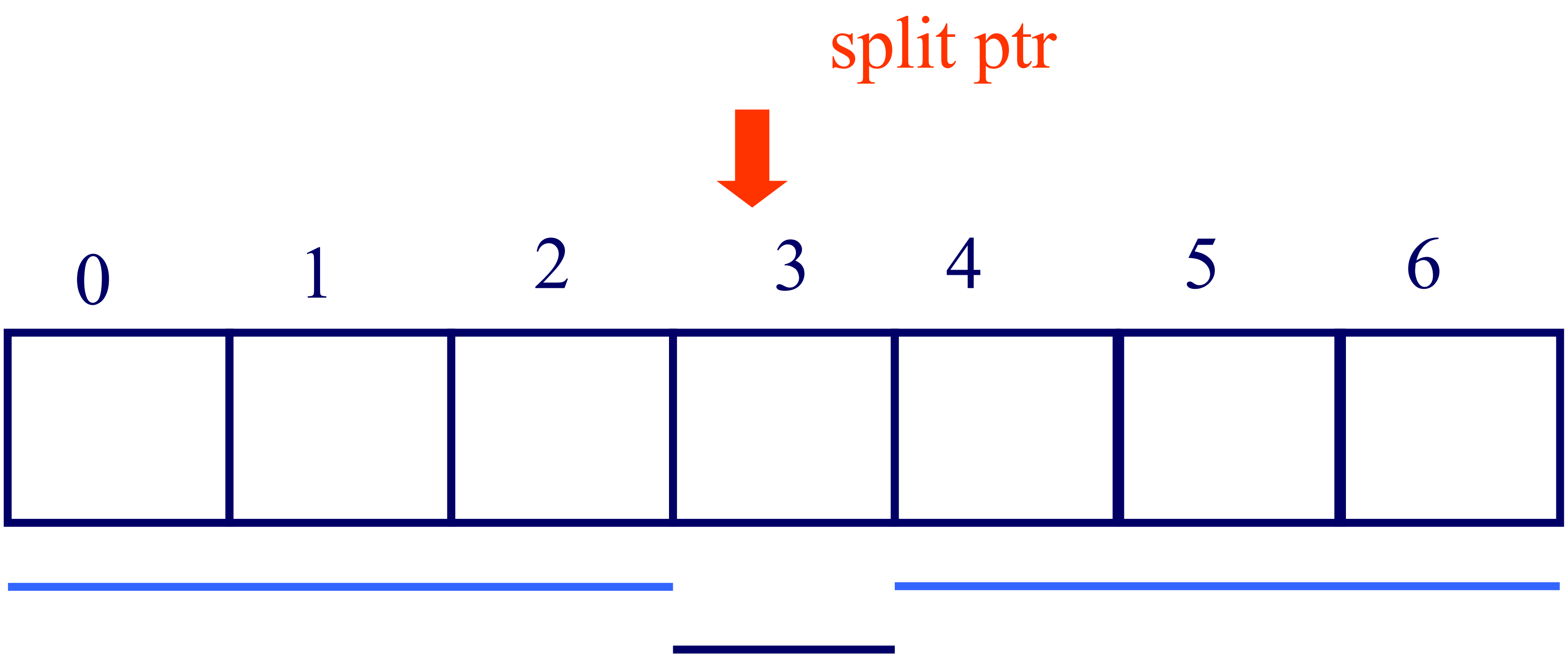
Algo: insert key '*k*'

- 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?

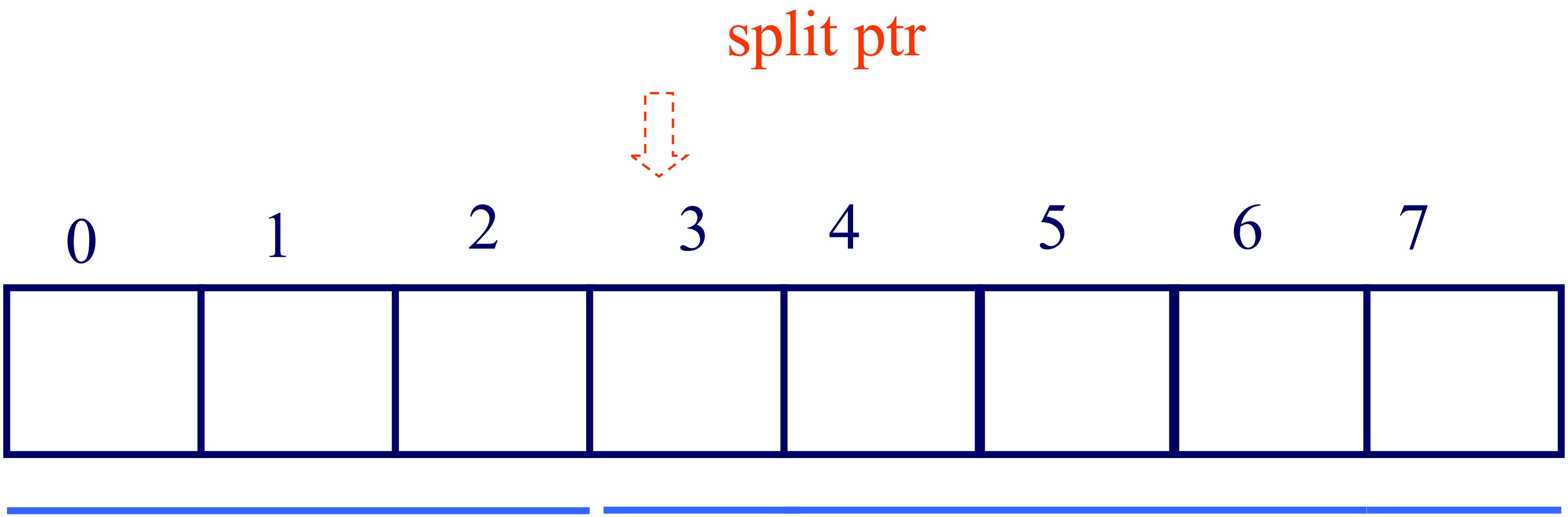
h0(x) = x mod N *(for the un-split buckets)*
*h1(x) = x mod (2*N)* *for the splitted ones)*



Linear hashing - split now?

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

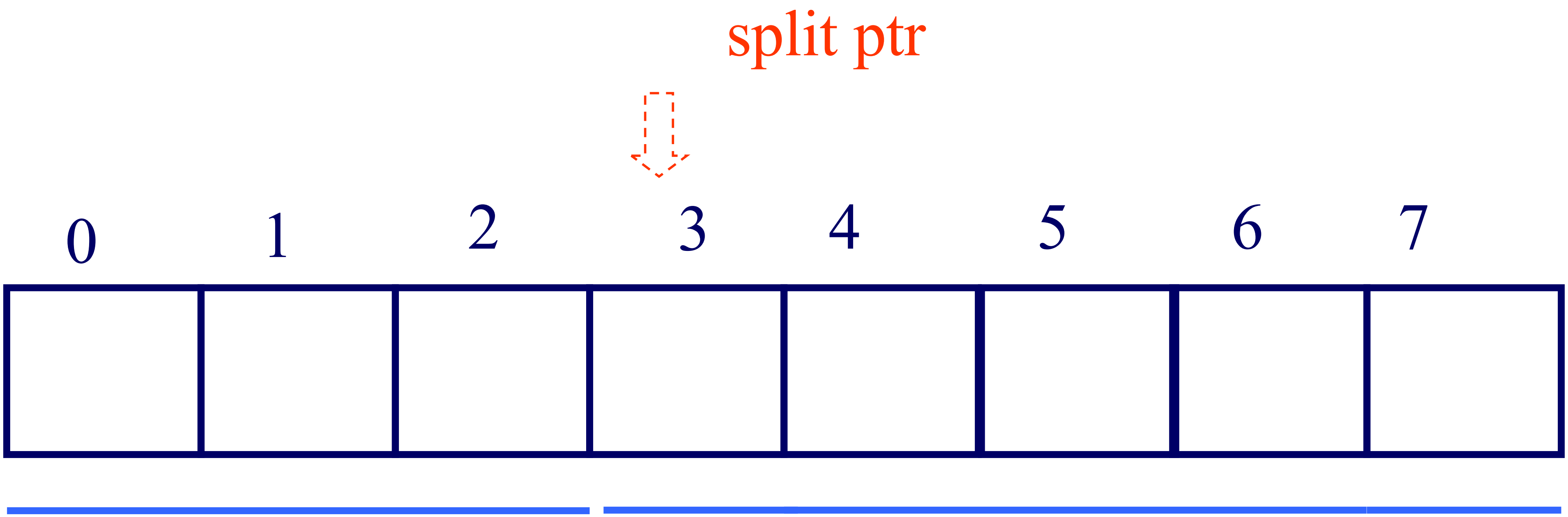
*h1(x) = x mod (2*N)* *(for the splitted ones)*



Linear hashing - split now?

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

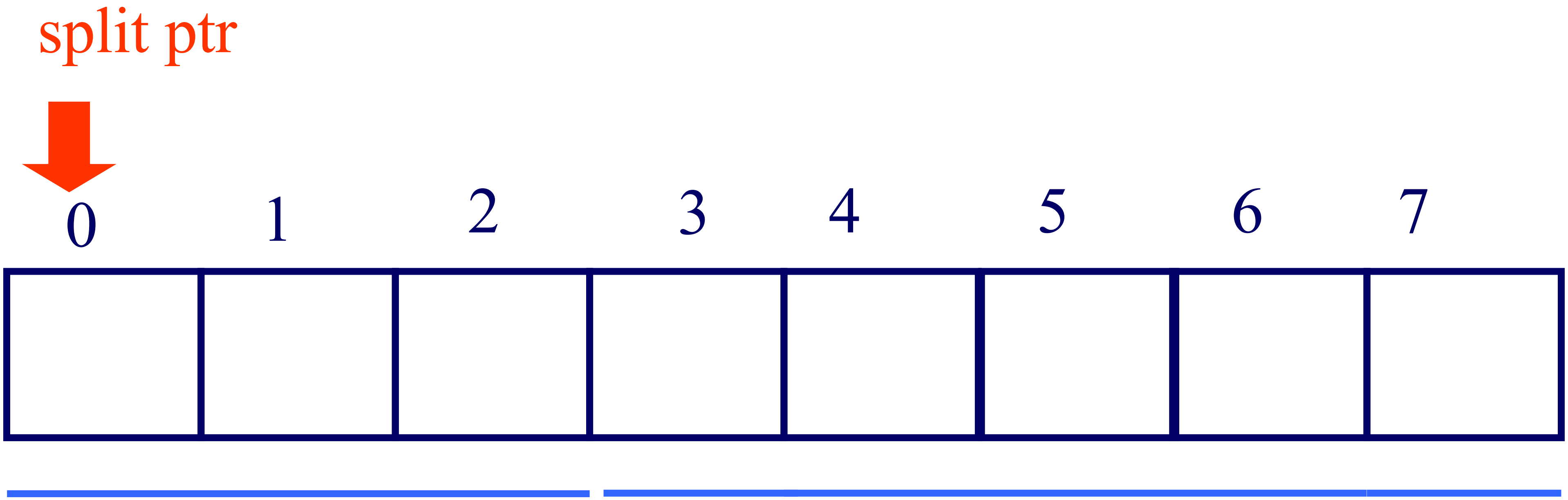
*h1(x) = x mod (2*N) (for the splitted ones)*



Linear hashing - split now?

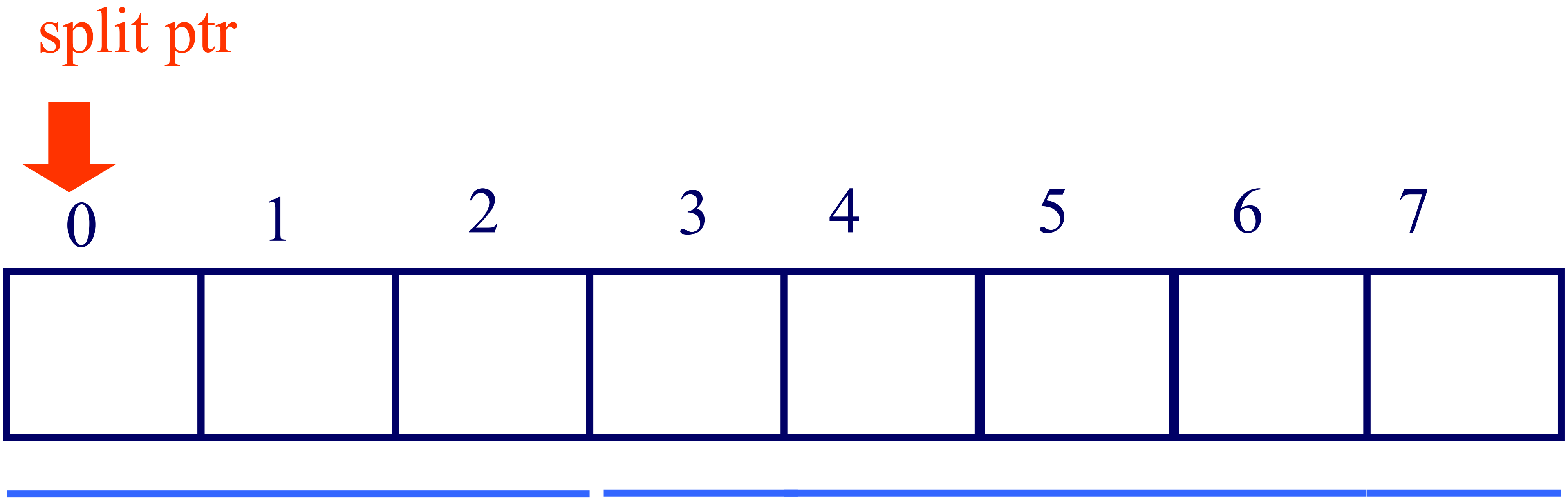
~~*h0(x) = x mod N* (for the un-split buckets)~~

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



Linear hashing - split now?

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



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.)

Linear hashing - overview

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
- performance analysis
- variations

Carnegie Mellon

Linear hashing - deletion?

- reverse of insertion:

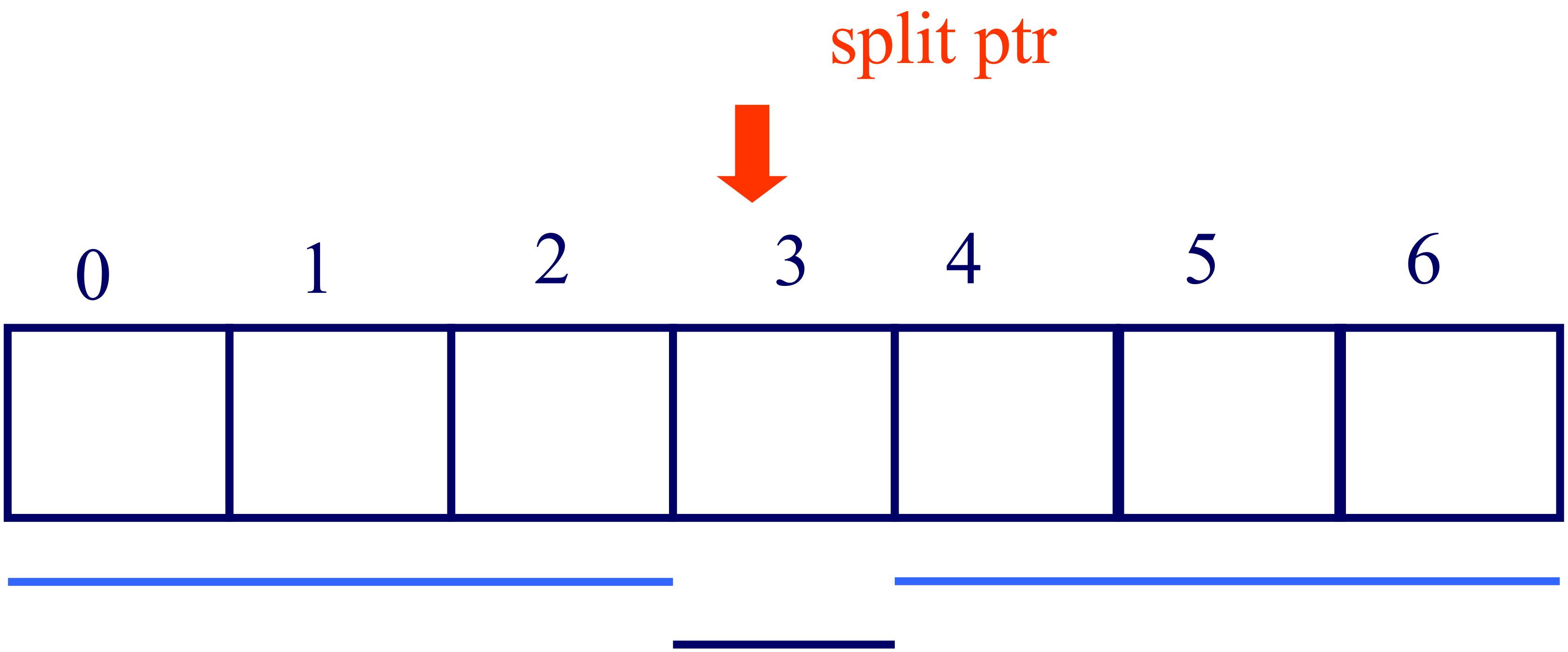
Linear hashing - deletion?

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

Linear hashing - how to contract?

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

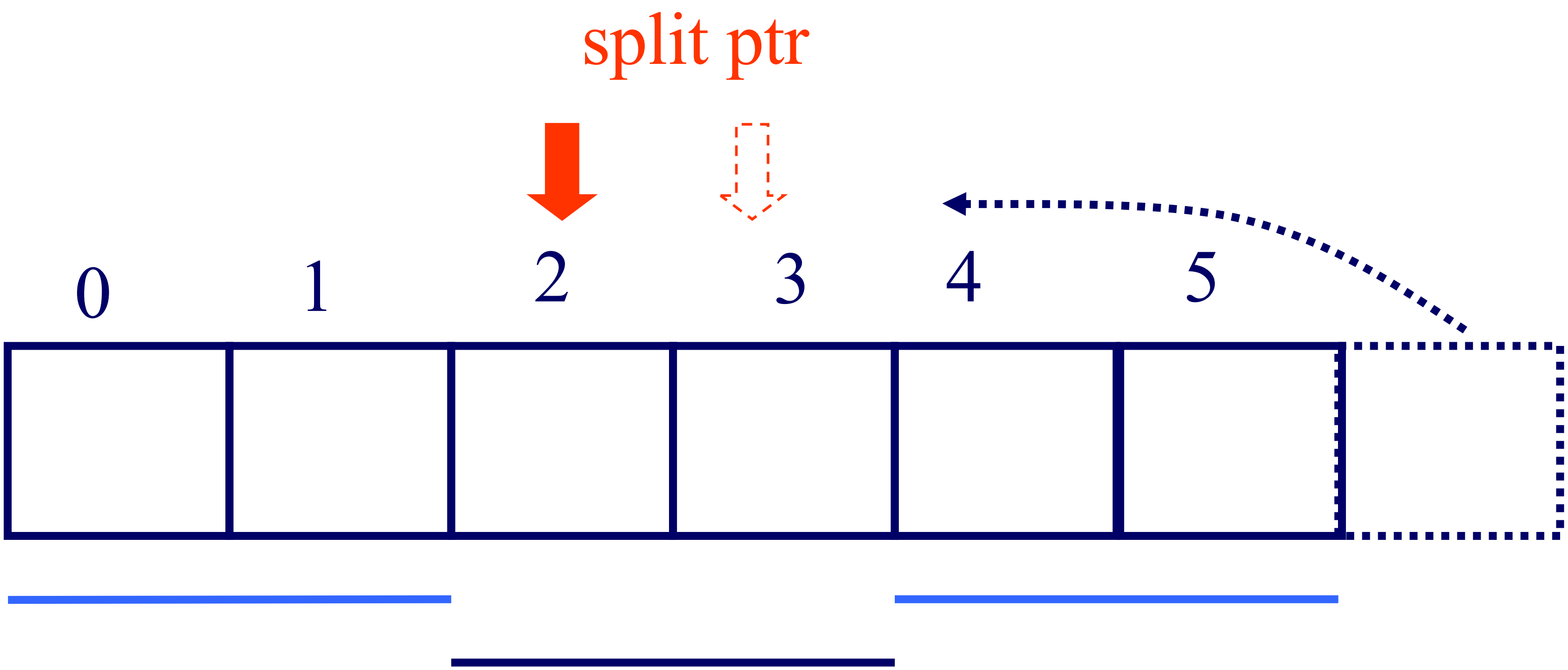
*h1(x) = mod (2*N)* (for the splitted ones)



Linear hashing - how to contract?

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

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



Carnegie Mellon

Linear hashing - overview

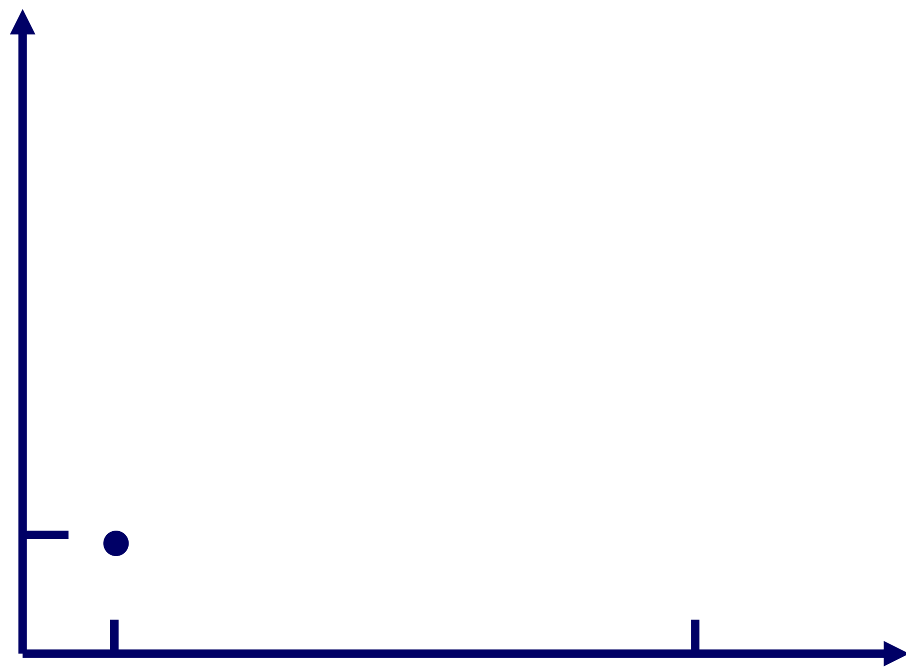
- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
- performance analysis
- variations

Linear hashing - performance

- [Larson, TODS 1982]

split: if $u>u_0$

(say u_0=.85)

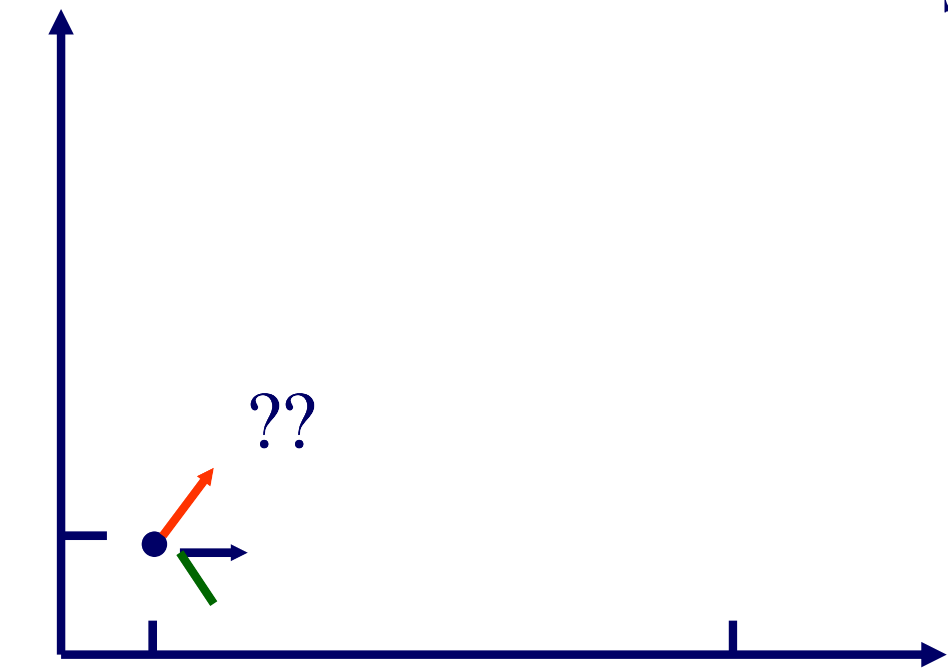


Linear hashing - performance

- [Larson, TODS 1982]

split: if $u>u_0$

(say u_0=.85)

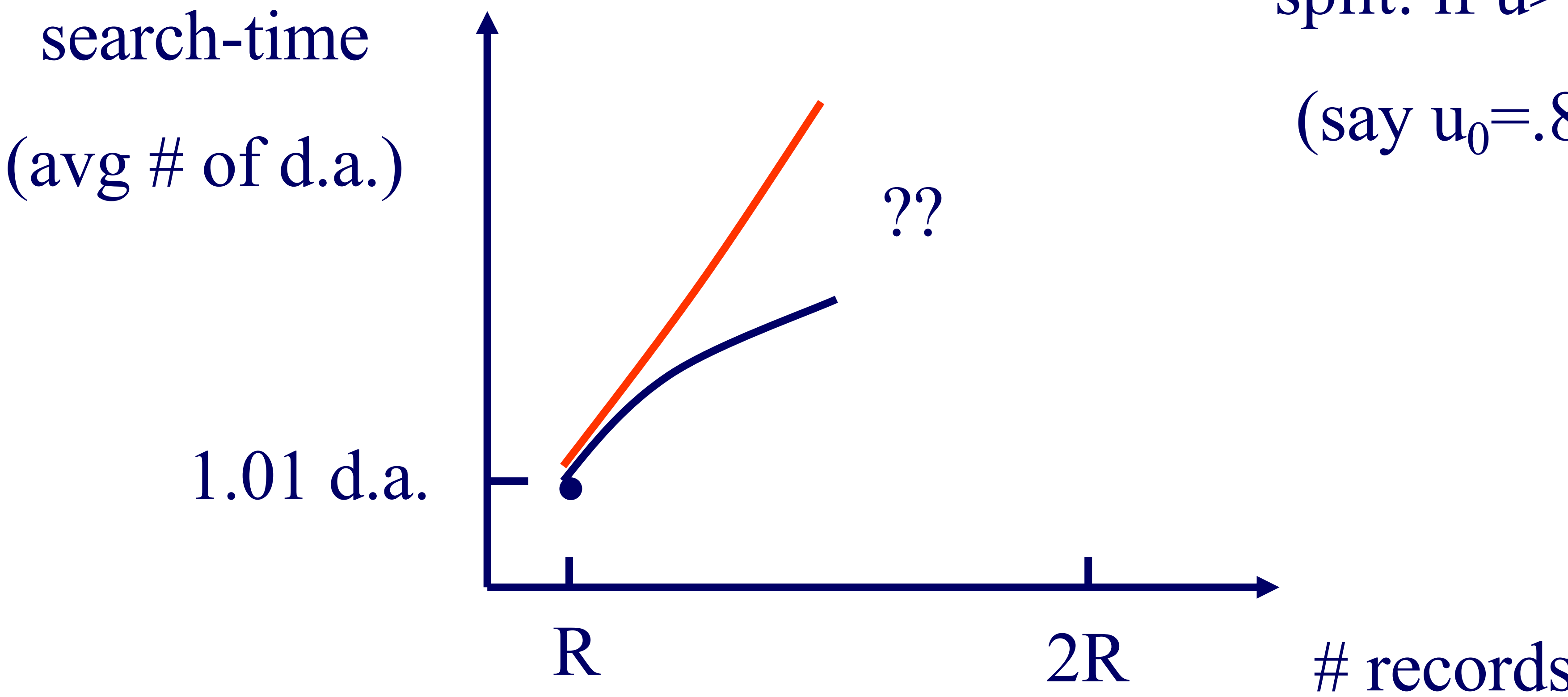


Linear hashing - performance

- [Larson, TODS 1982]

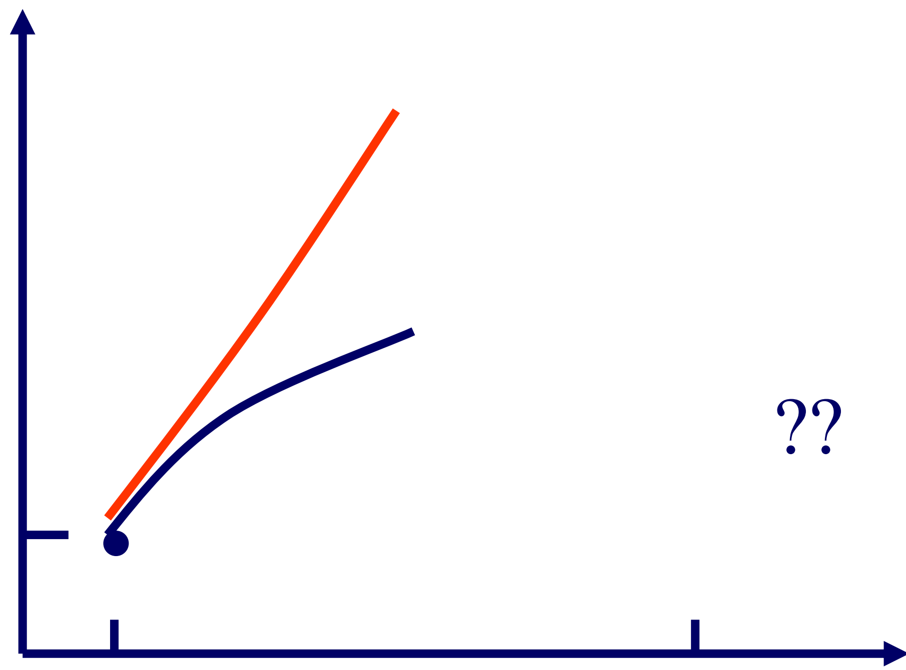
split: if $u > u_0$

(say $u_0 = .85$)



Linear hashing - performance

- [Larson, TODS 1982]



Linear hashing - performance

- [Larson, TODS 1982]

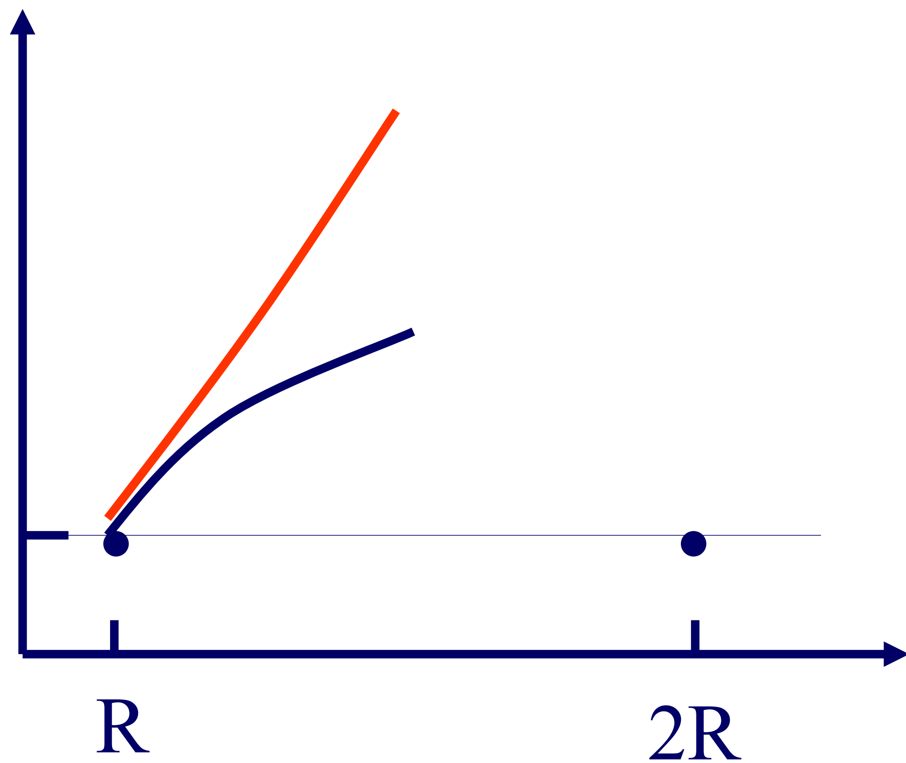
split: if $u>u_0$

(say u_0=.85)

search-time

(avg # of d.a.)

1.01 d.a.



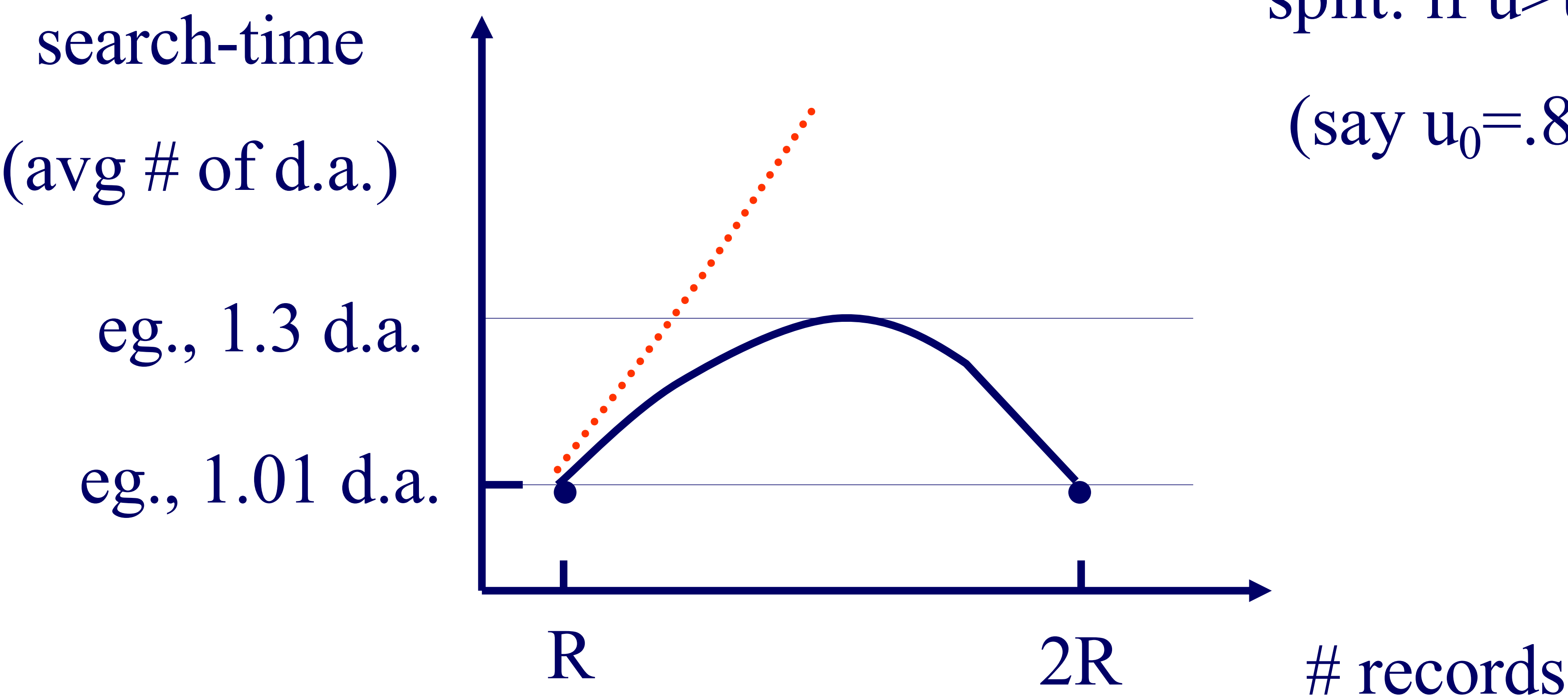
records

Linear hashing - performance

- [Larson, TODS 1982]

split: if $u > u_0$

(say $u_0 = .85$)



Linear hashing - overview

- Motivation
- main idea
- search algo
- insertion/split algo
- deletion
- performance analysis
- variations

Other hashing variations

- ‘order preserving’
- ‘perfect hashing’ (no collisions!) [Ed. Fox, et al]

Primary key indexing - conclusions

- hashing is O(1) on the average for search
- linear hashing: elegant way to grow a hash table
- B-trees: industry work-horse for primary-key indexing (O(log(N) w.c.!)

References for primary key indexing

- [Fagin+] Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, H. Raymond Strong: Extendible Hashing - A Fast Access Method for Dynamic Files. TODS 4(3): 315-344(1979)
- [Fox] Fox, E. A., L. S. Heath, Q.-F. Chen, and A. M. Daoud. "Practical Minimal Perfect Hash Functions for Large Databases." Communications of the ACM 35.1 (1992): 105-21.

Carnegie Mellon

References, cont’d

- [Knuth] D.E. Knuth. The Art Of Computer Programming, Vol. 3, Sorting and Searching, Addison Wesley
- [Larson] Per-Ake Larson Performance Analysis of Linear Hashing with Partial Expansions ACM TODS, 7,4, Dec. 1982, pp 566--587
- [Litwin] Litwin, W., (1980), Linear Hashing: A New Tool for File and Table Addressing, VLDB, Montreal, Canada, 1980

