VirginiaTech

CS 4604: Introduction to Database Management Systems

B. Aditya Prakash

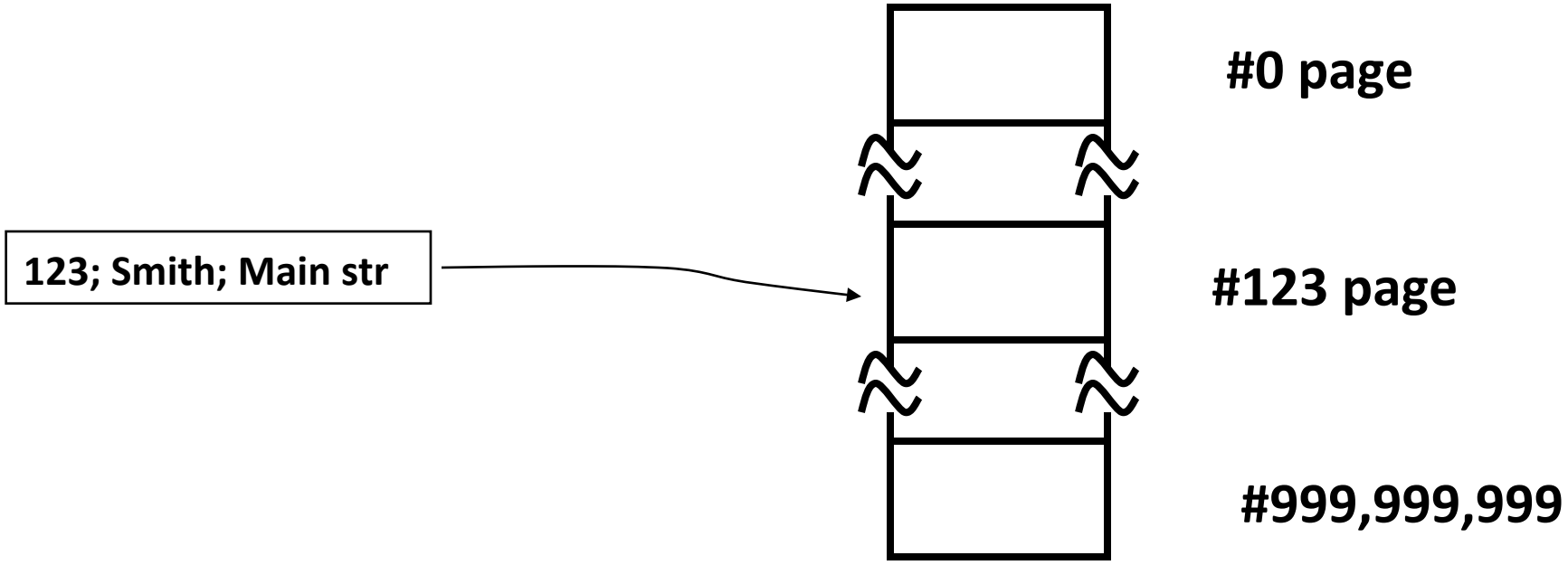
Lecture #9: Hashing and Sorting

(Static) Hashing

- Problem: "find EMP record with ssn=123"
- What if disk space was free, and time was at premium?

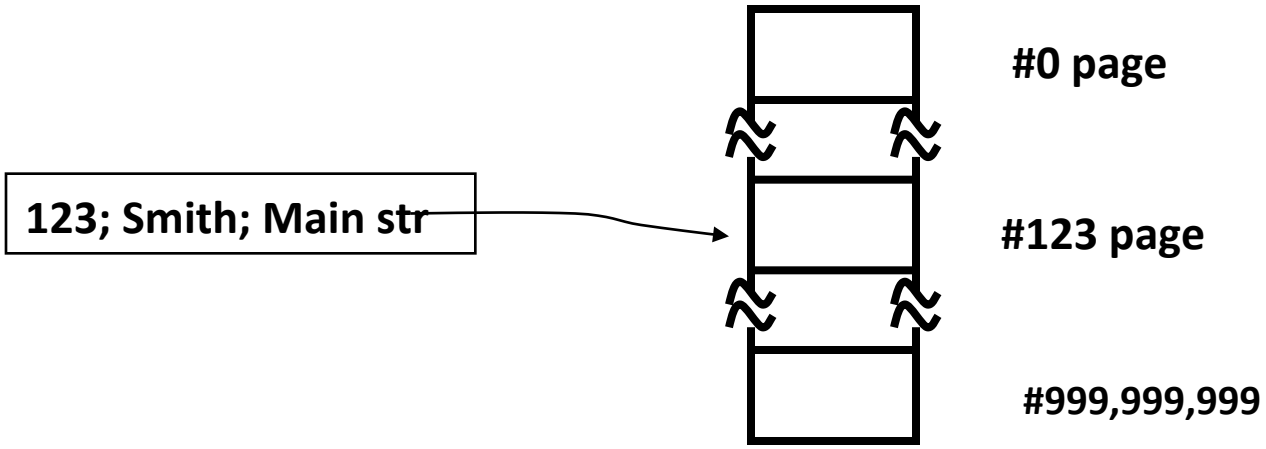
Hashing

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



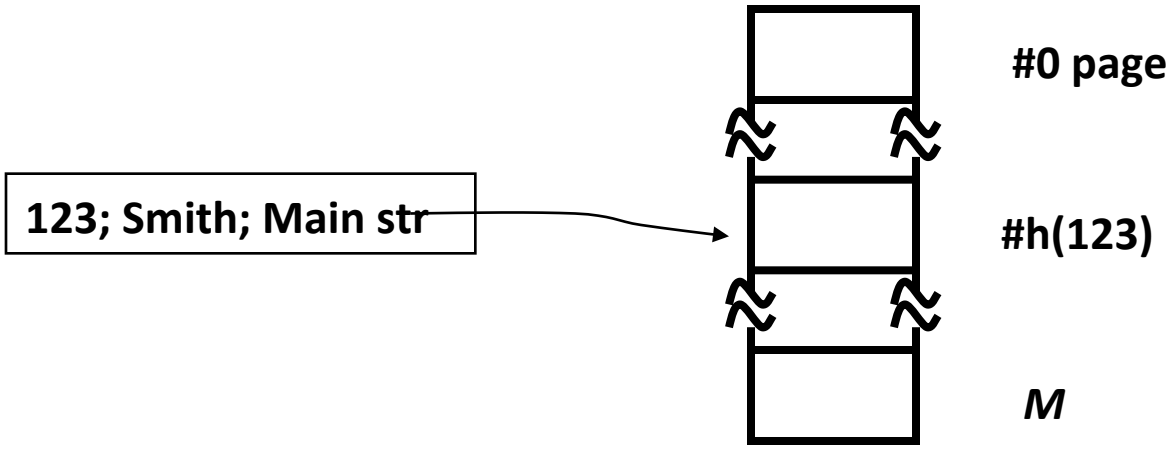
Hashing

- Since space is NOT free:
- use M, instead of 999,999,999 slots
- hash function: h(key) = slot-id



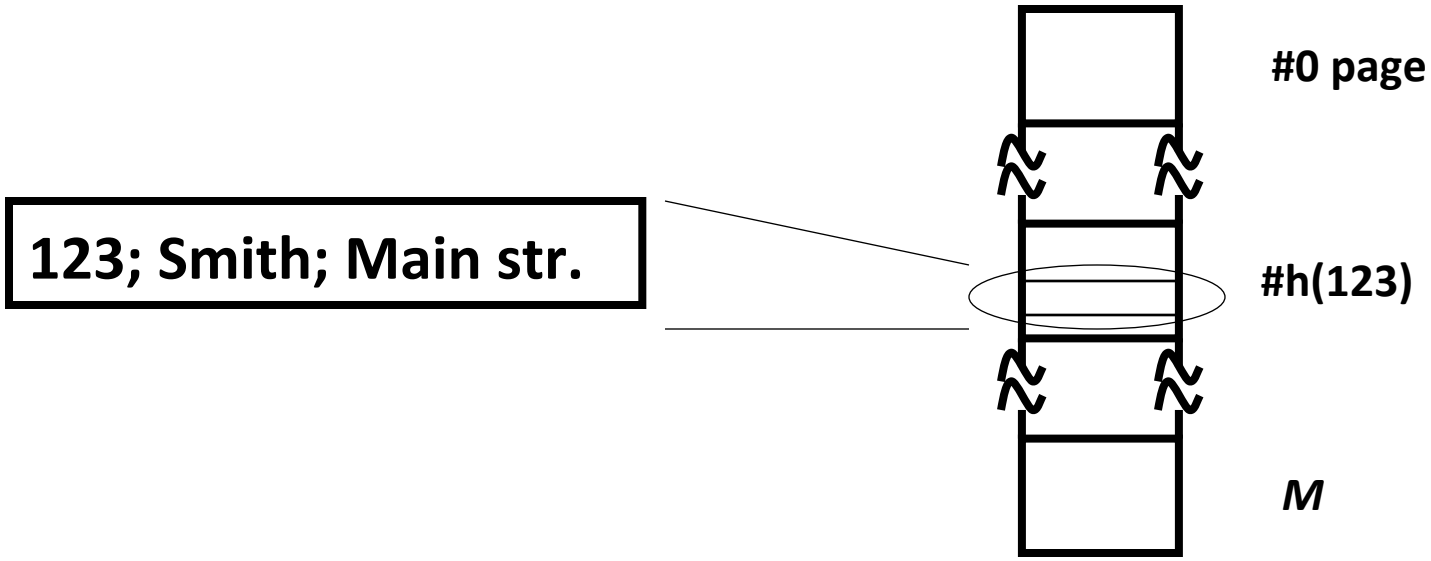
Hashing

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



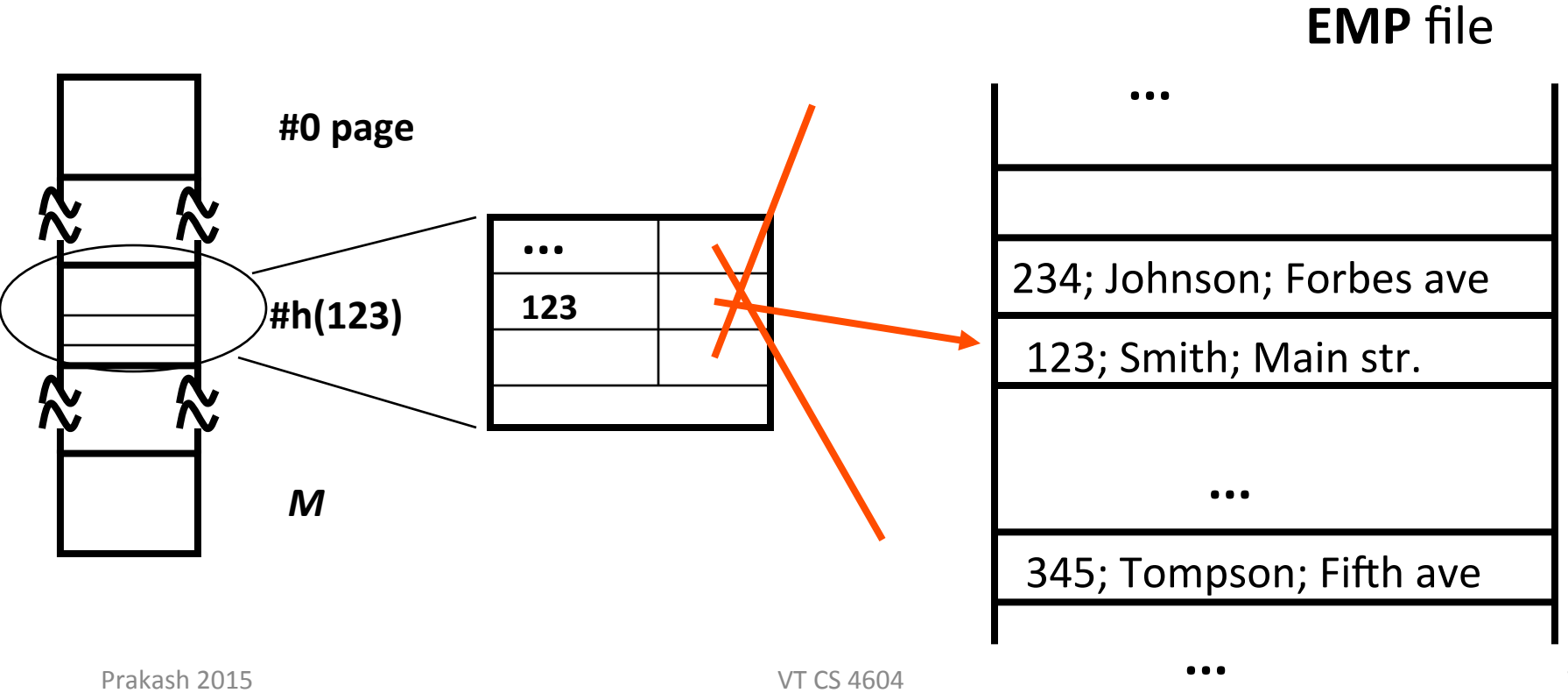
Hashing

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



Hashing

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



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) 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?)

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!

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

Other hashing functions

- quadratic hashing (bad)
- ...

Other hashing functions

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

Size of hash table

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

Size of hash table

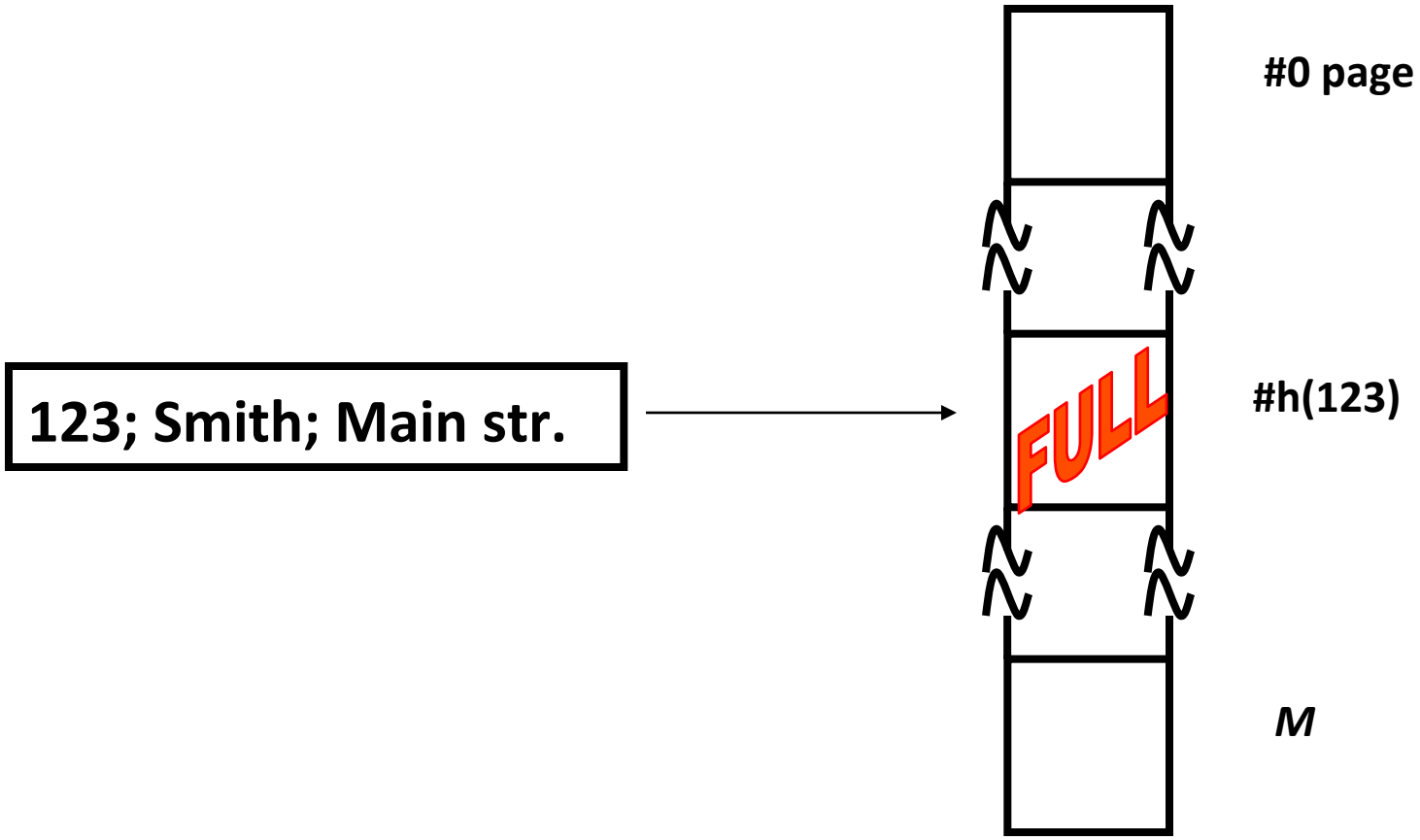
- eg., 50,000 employees, 10 employees/page
- Q: *M*=?? pages/buckets/slots
- A: utilization ~ 90% and
  - *M*: prime number

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

Collision resolution

- Q: what is a ‘collision’?
- A: ??

Collision resolution



Collision resolution

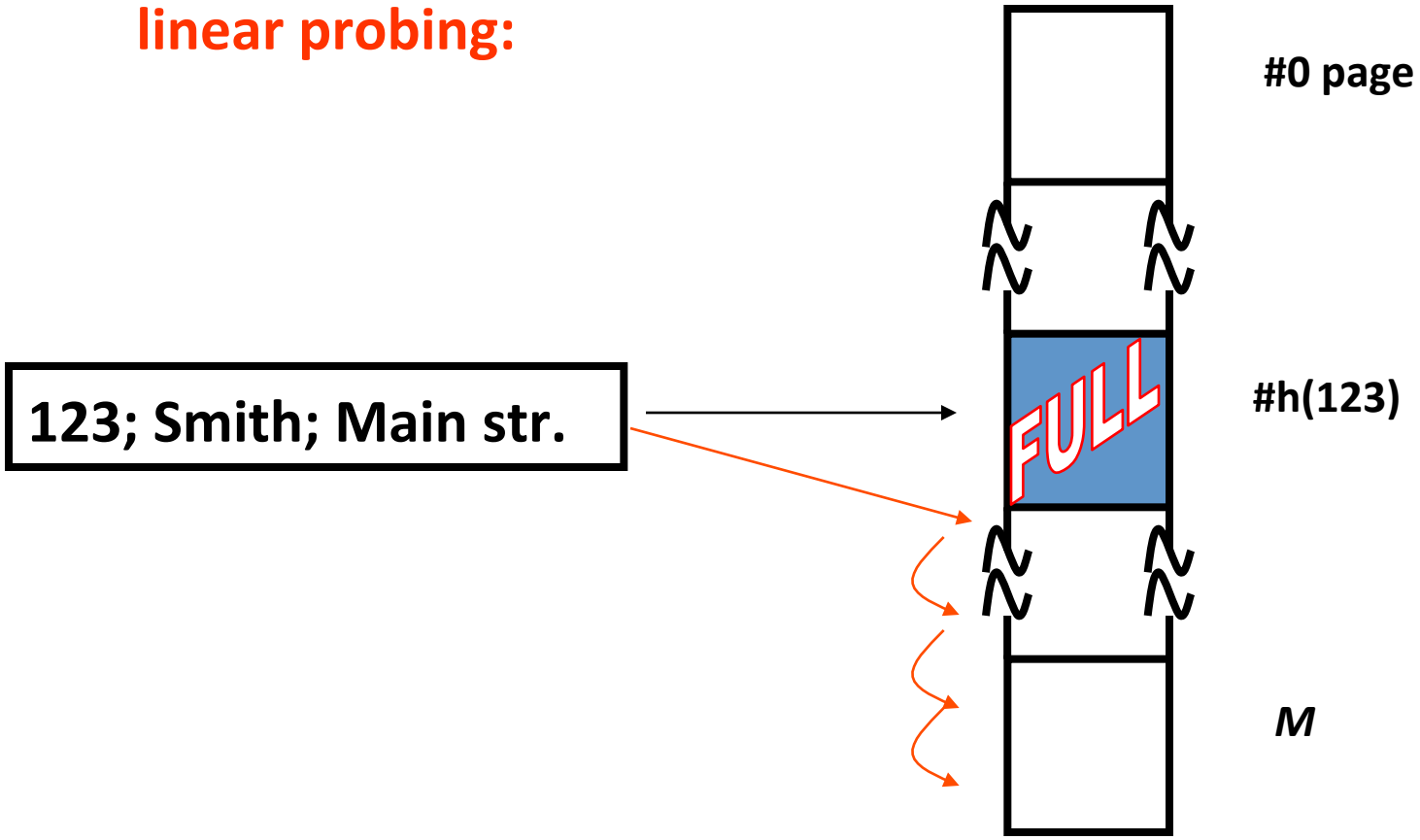
- Q: what is a 'collision'?
- A: ??
- Q: why worry about collisions/overflows? (recall that buckets are ~90% full)
- A: 'birthday paradox'

Collision resolution

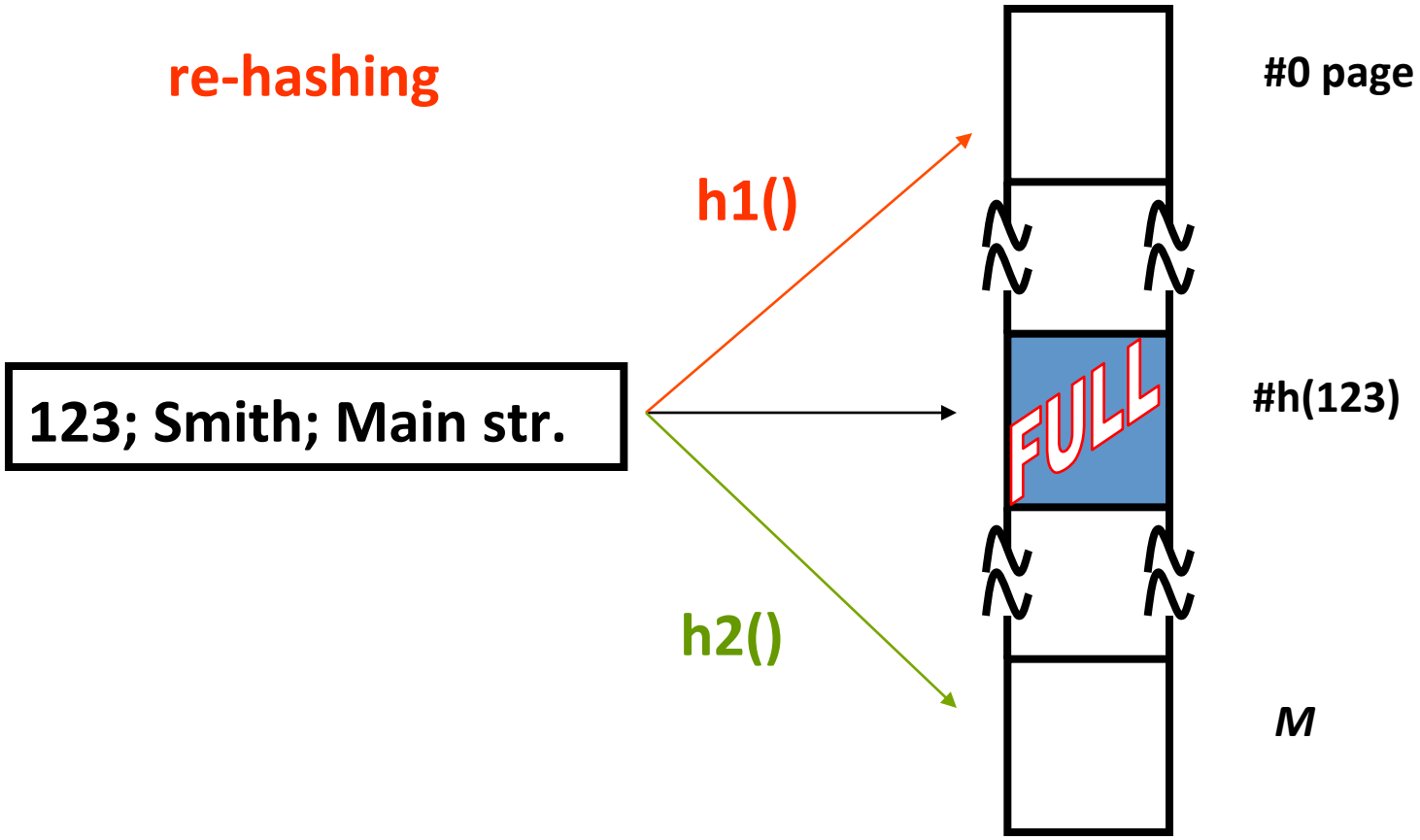
- open addressing
  - linear probing (ie., put to next slot/bucket)
  - re-hashing
- separate chaining (ie., put links to overflow pages)

Collision resolution

linear probing:

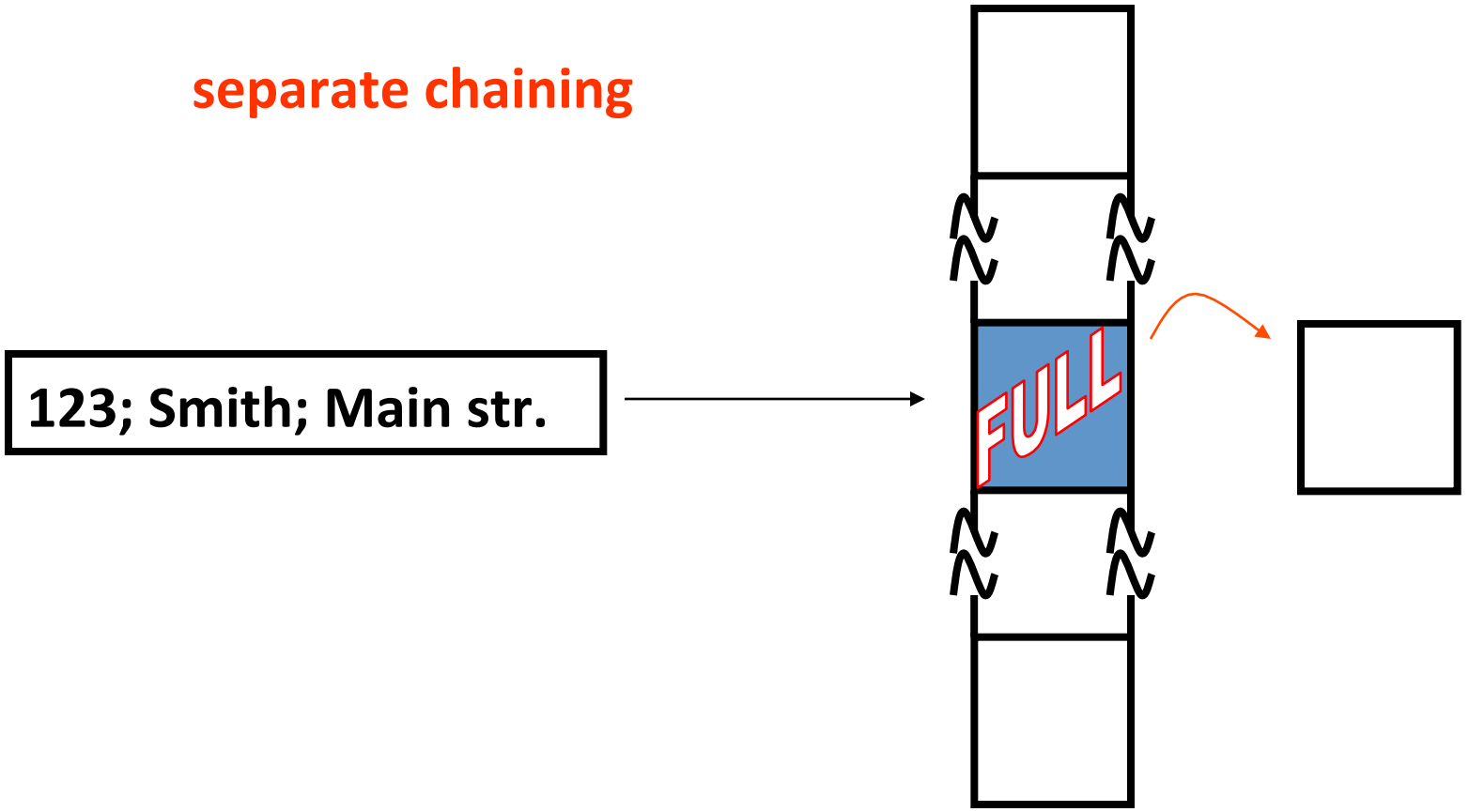


Collision resolution



Collision resolution

separate chaining



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

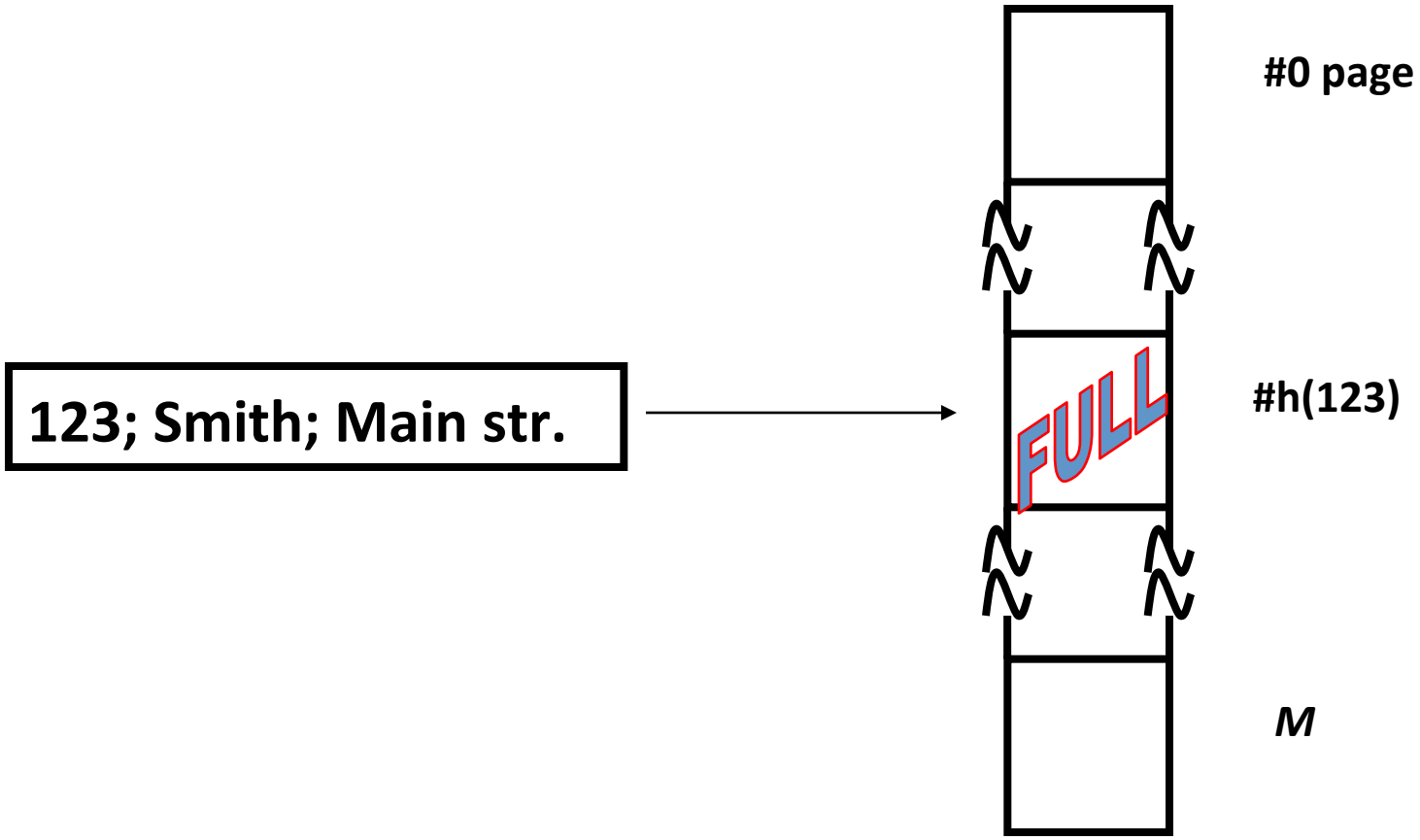
Problem with static hashing

- 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]

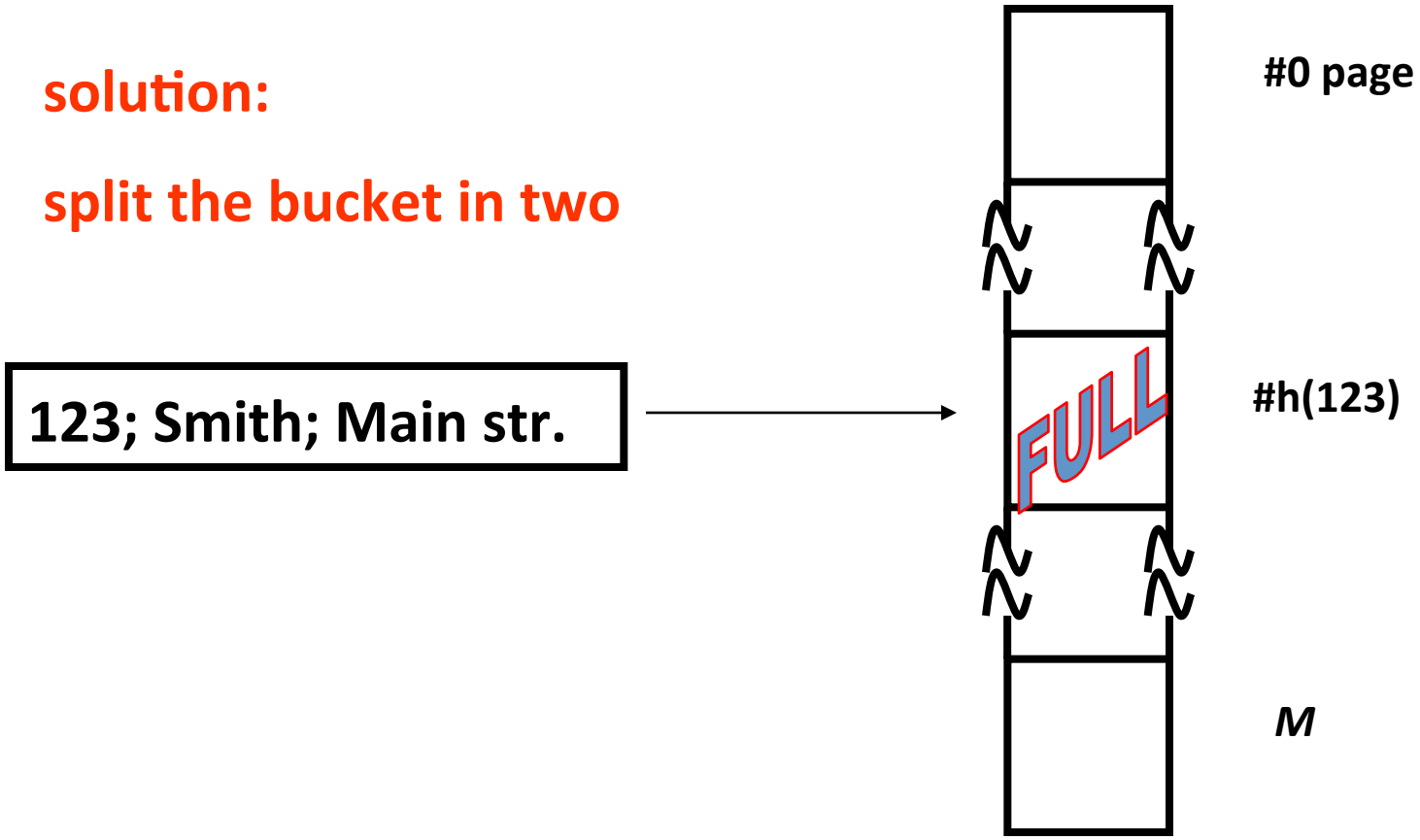
Extendible hashing



Extendible hashing

solution:

split the bucket in two



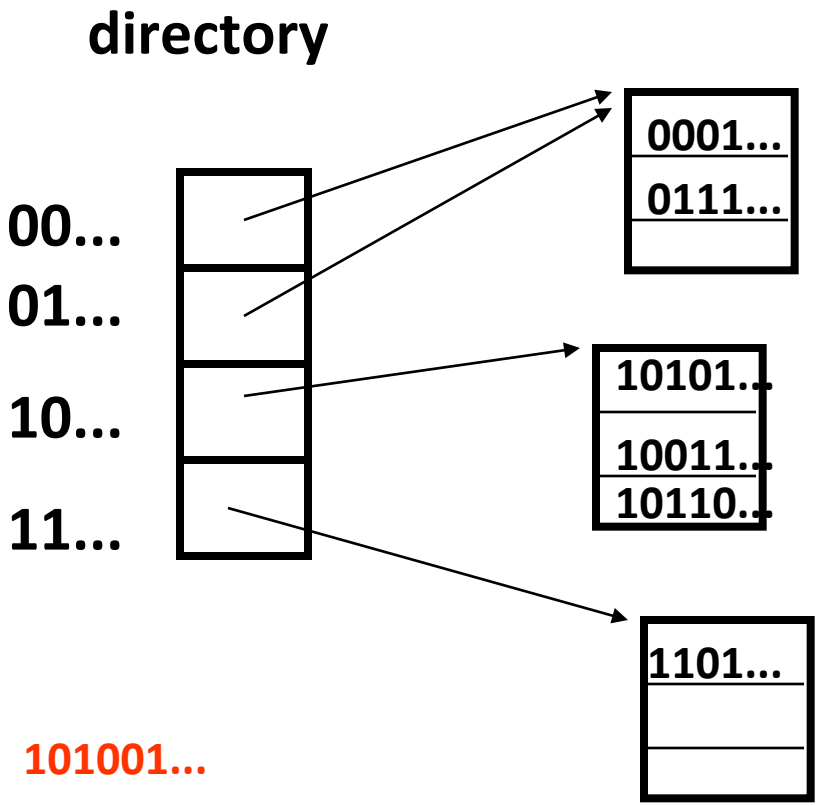
Extendible hashing

in detail:

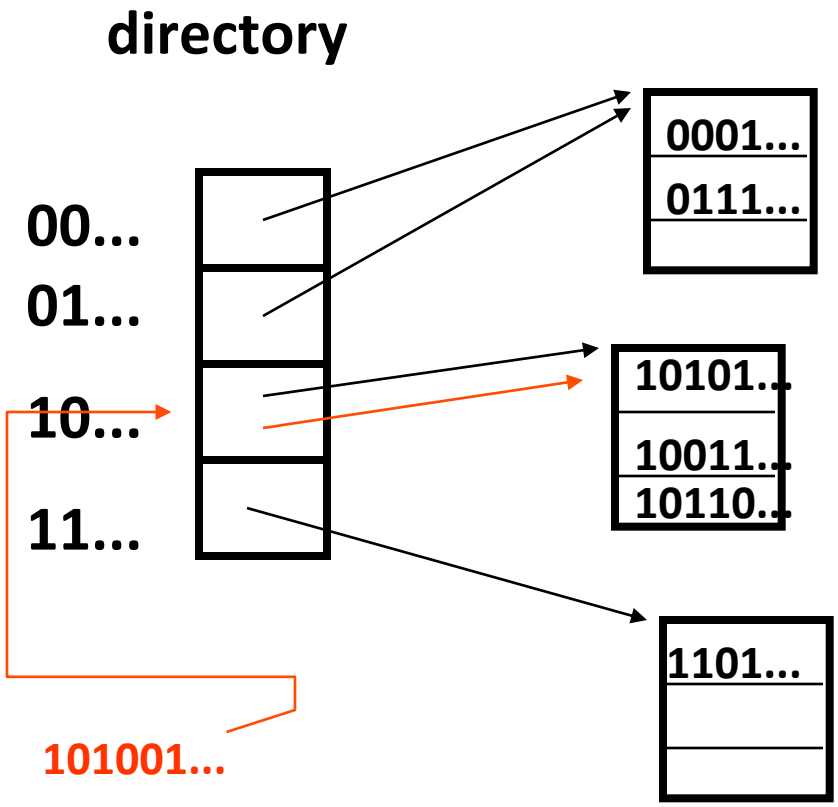
- keep a directory, with ptrs to hash-buckets
- Q: how to divide contents of bucket in two?
- A: hash each key into a very long bit string; keep only as many bits as needed

Eventually:

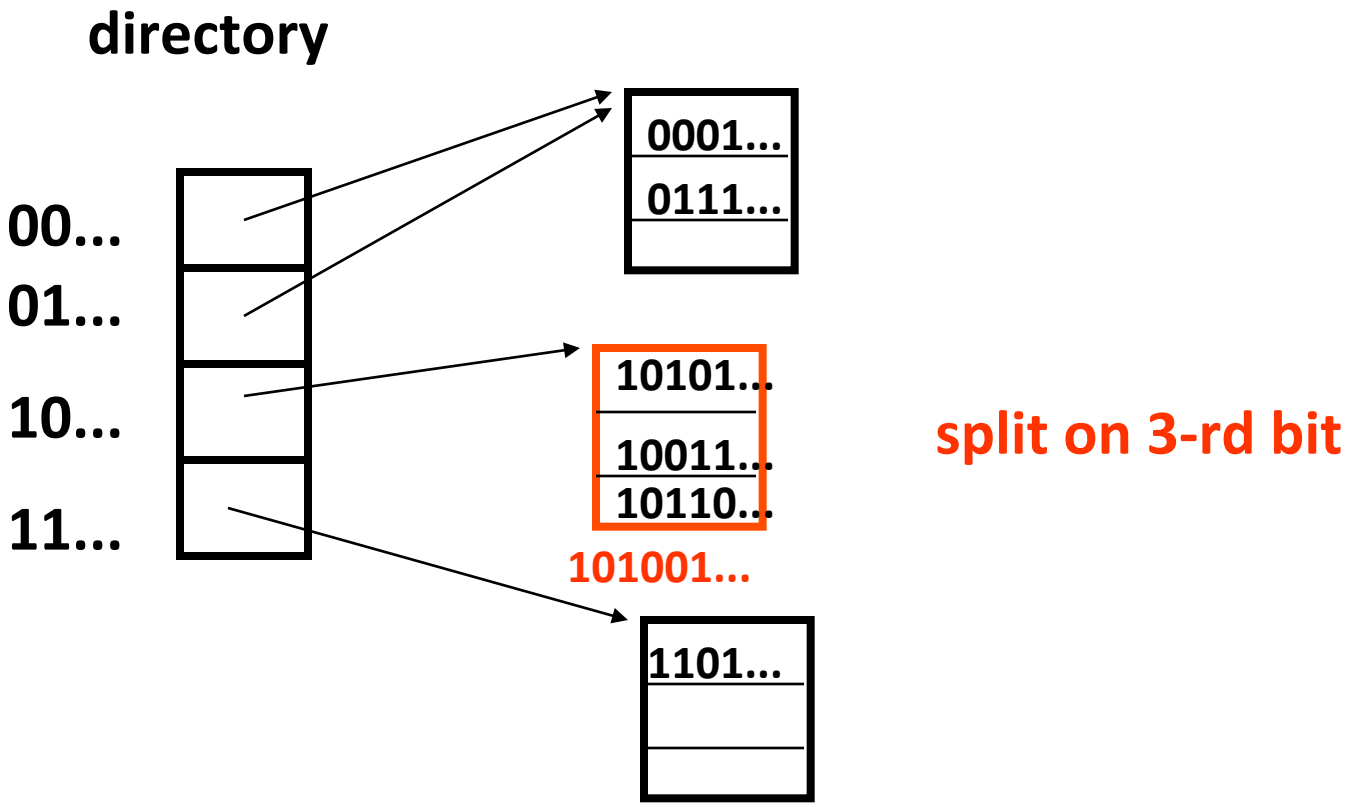
Extendible hashing



Extendible hashing

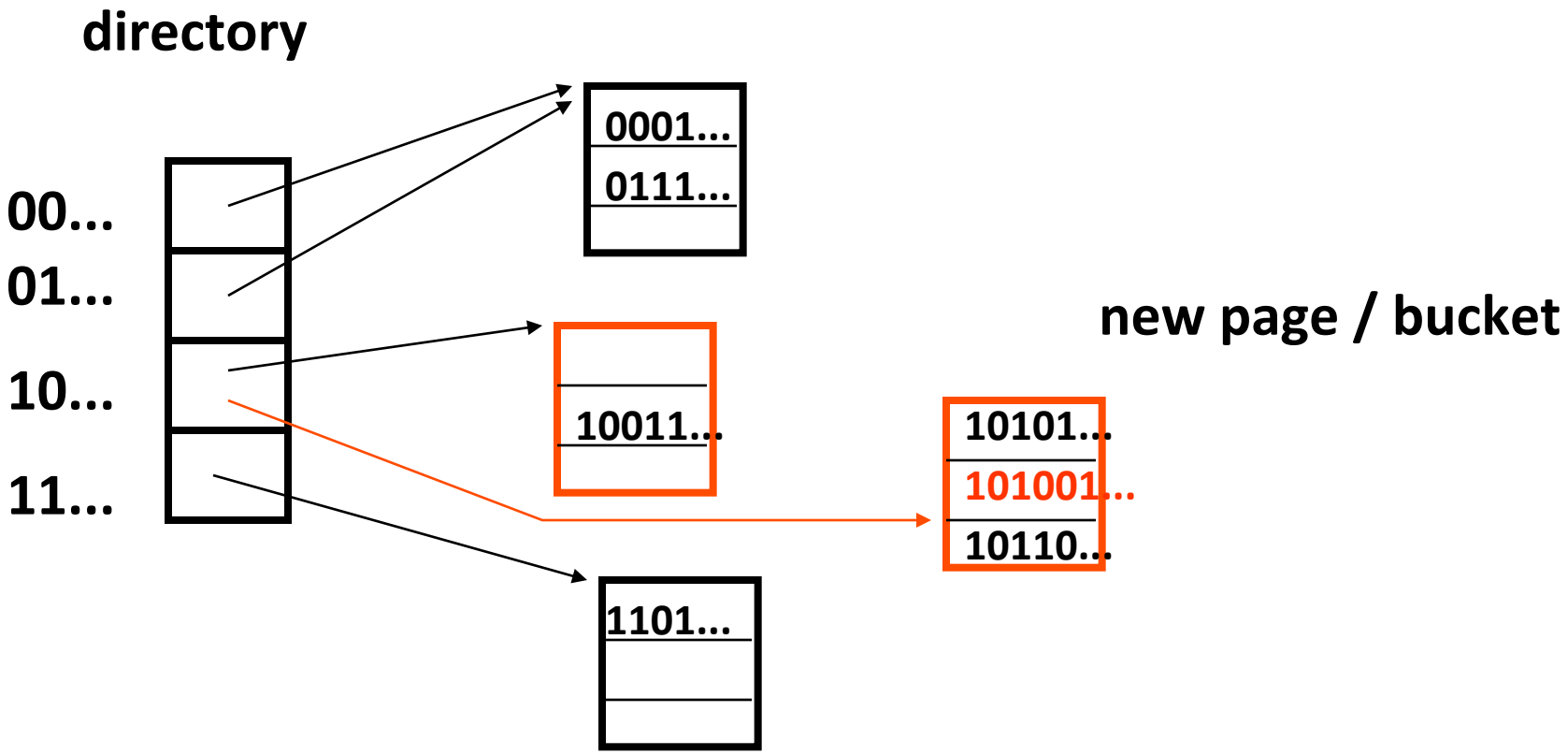


Extendible hashing

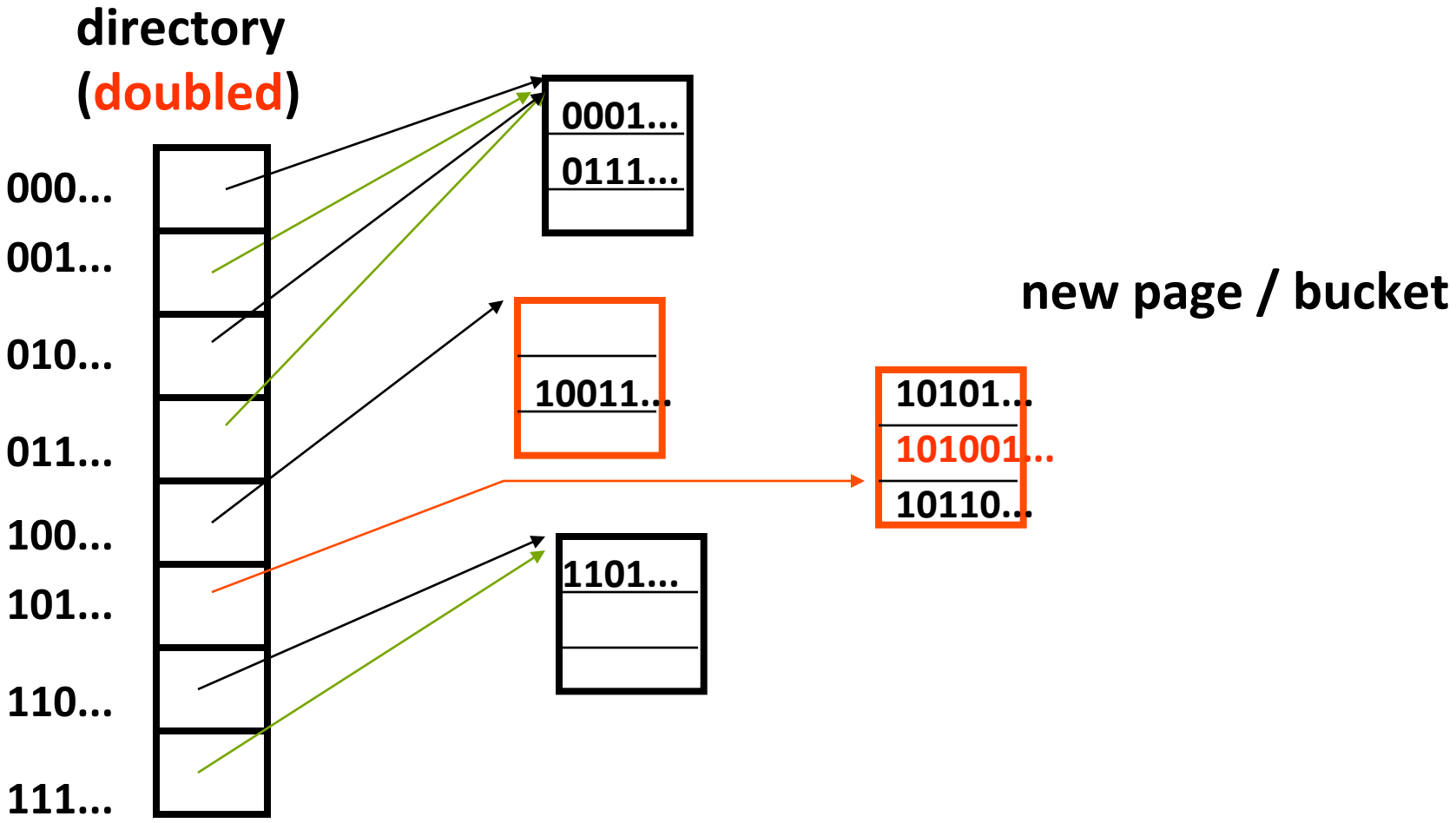


split on 3-rd bit

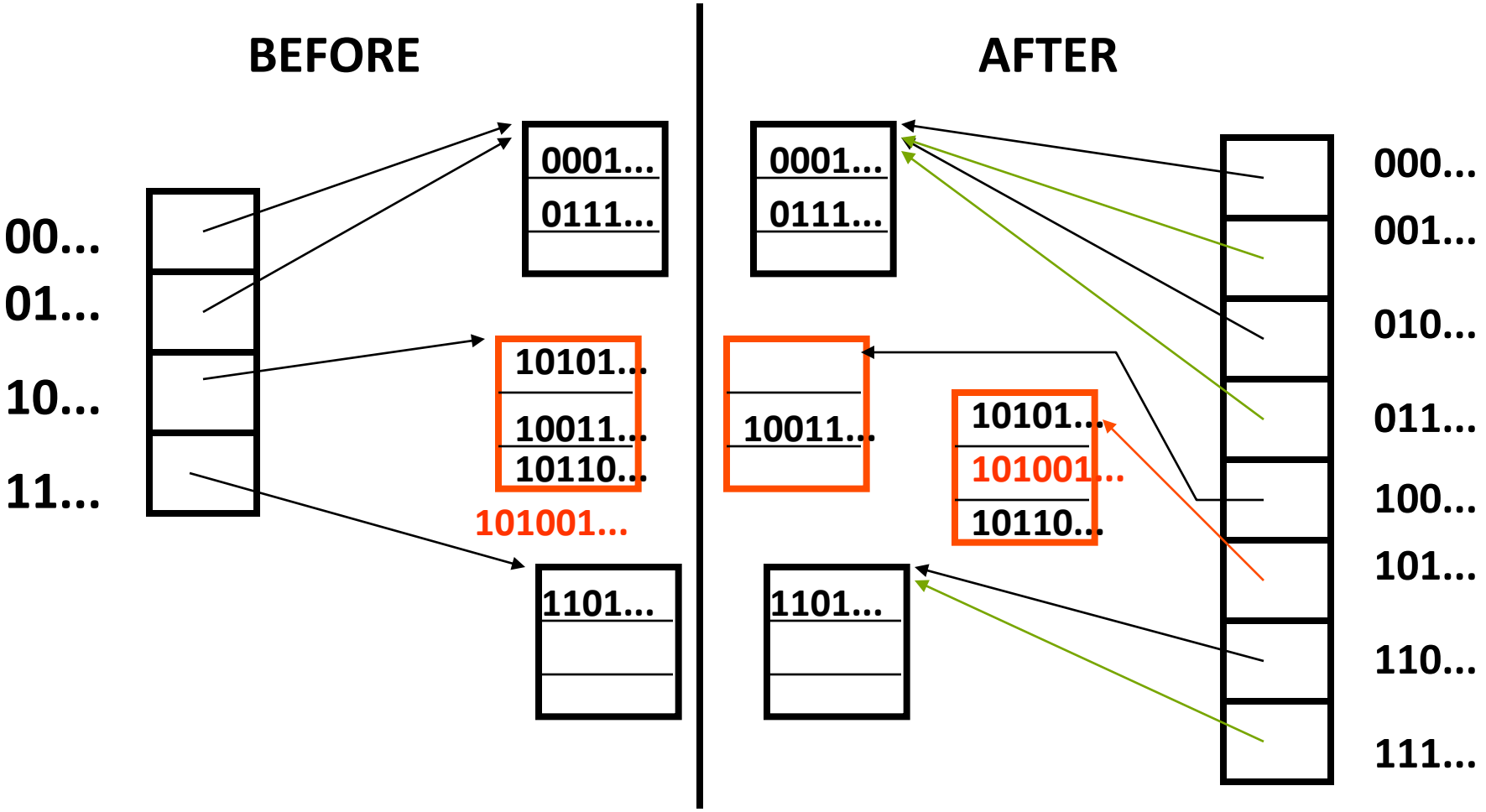
Extendible hashing



Extendible hashing



Extendible hashing



Extendible hashing

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

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

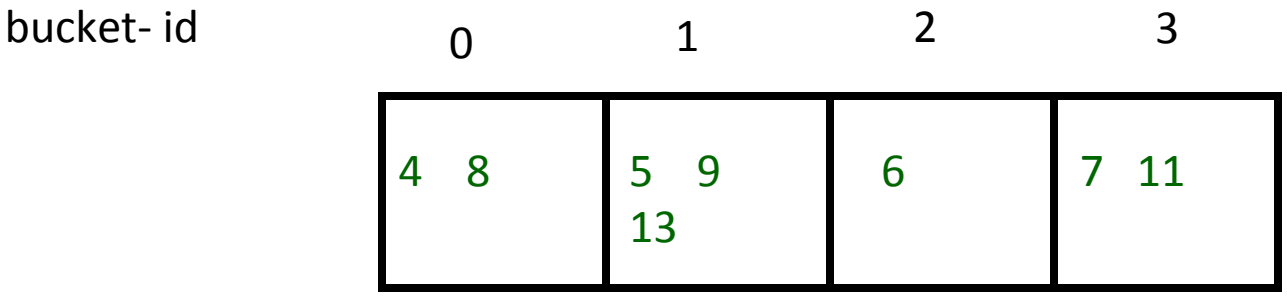
- Motivation: ext. hashing needs directory etc etc; which doubles (ouch!)
- Q: can we do something simpler, with smoother growth?
- A: split buckets from left to right, regardless of which one overflowed ('crazy' , but it works well!) - Eg.:

Linear hashing

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

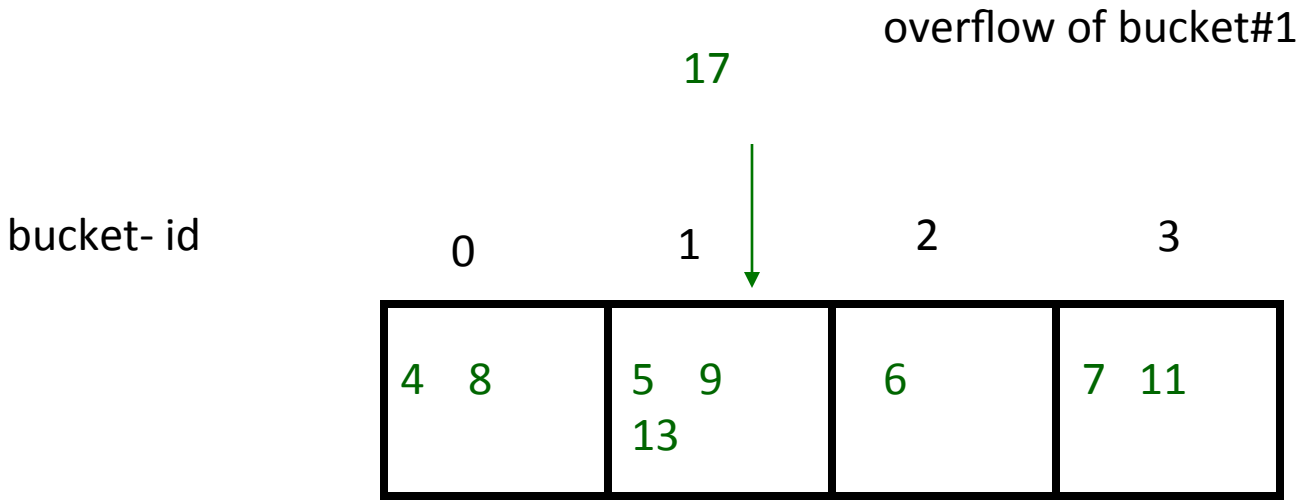
Assume capacity: 3 records / bucket

Insert key '17'



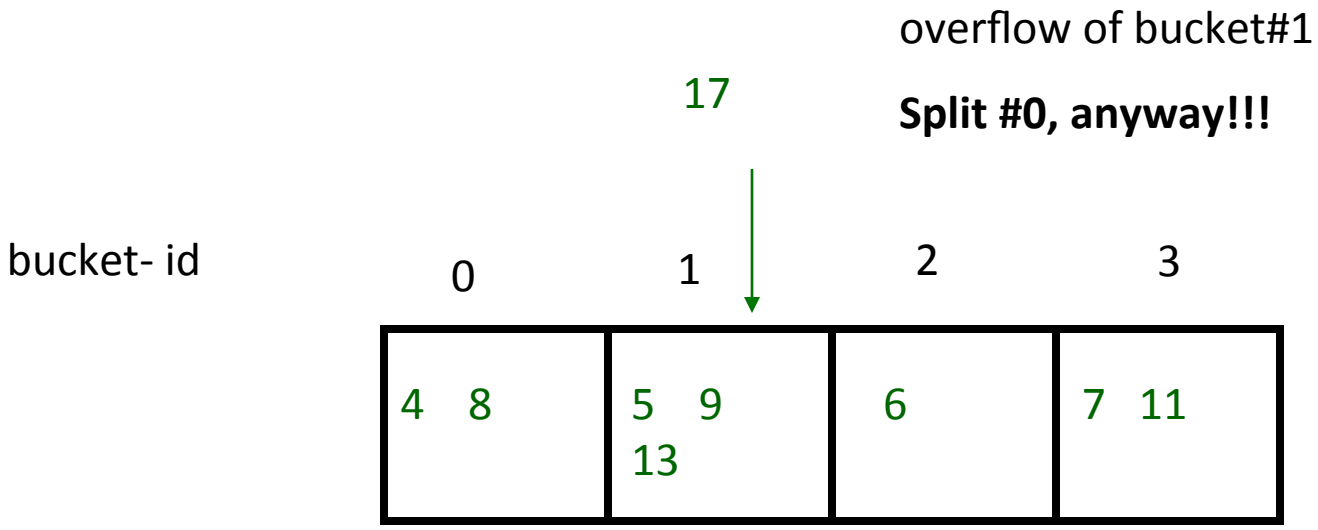
Linear hashing

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



Linear hashing

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

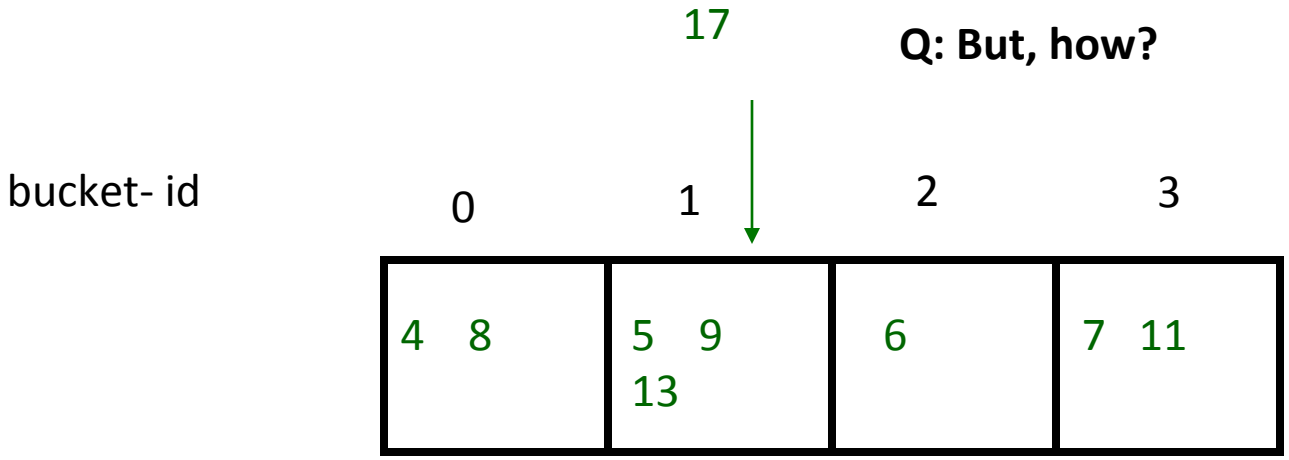


Linear hashing

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

Split #0, anyway!!!

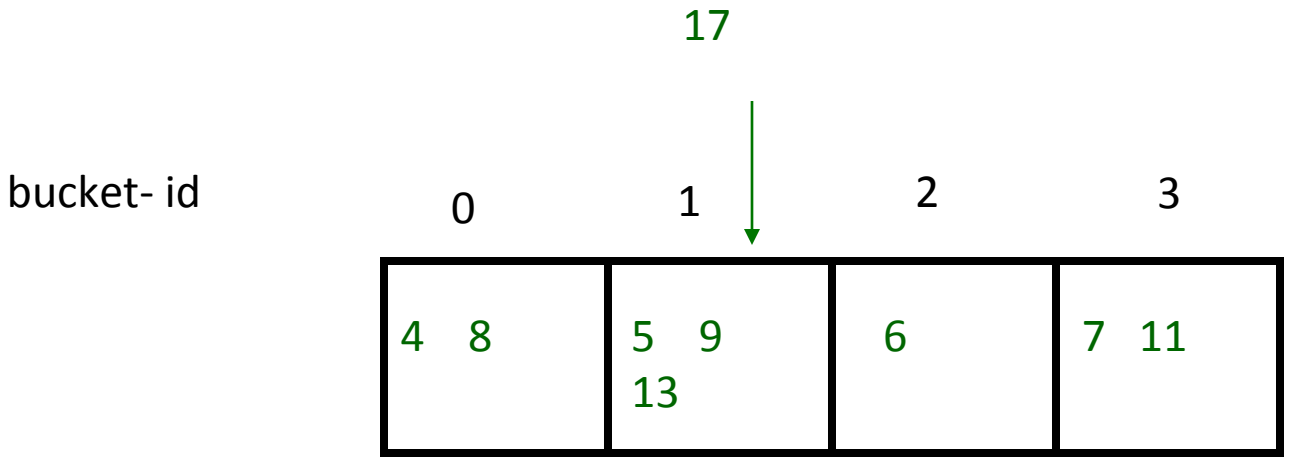
Q: But, how?



Linear hashing

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

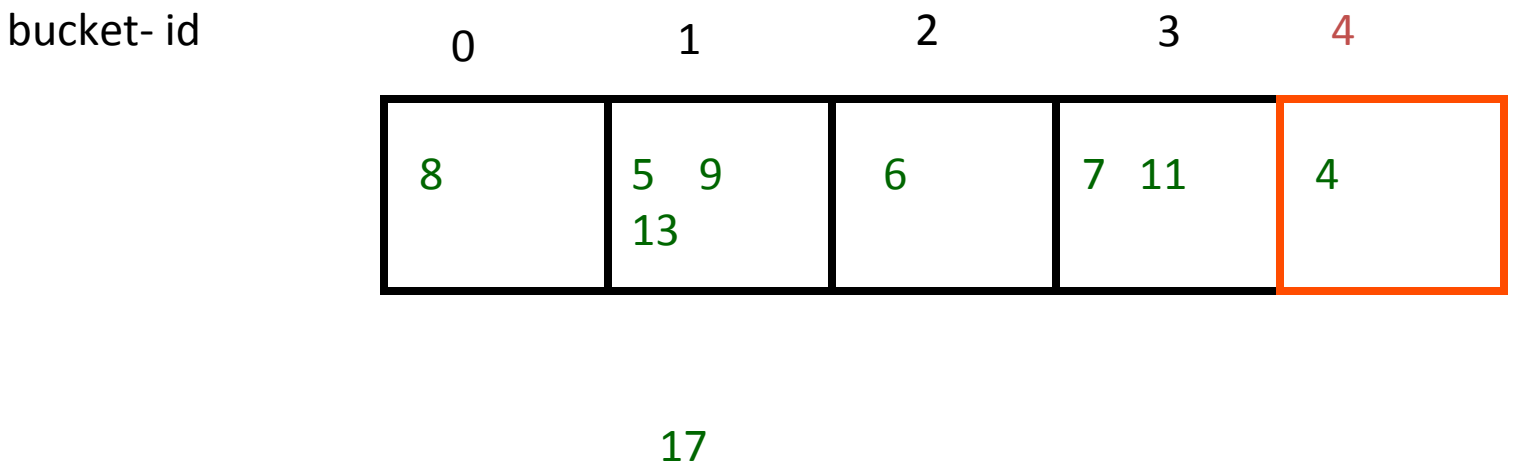
*h1(x) = x mod (2*N)*



Linear hashing - after split:

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

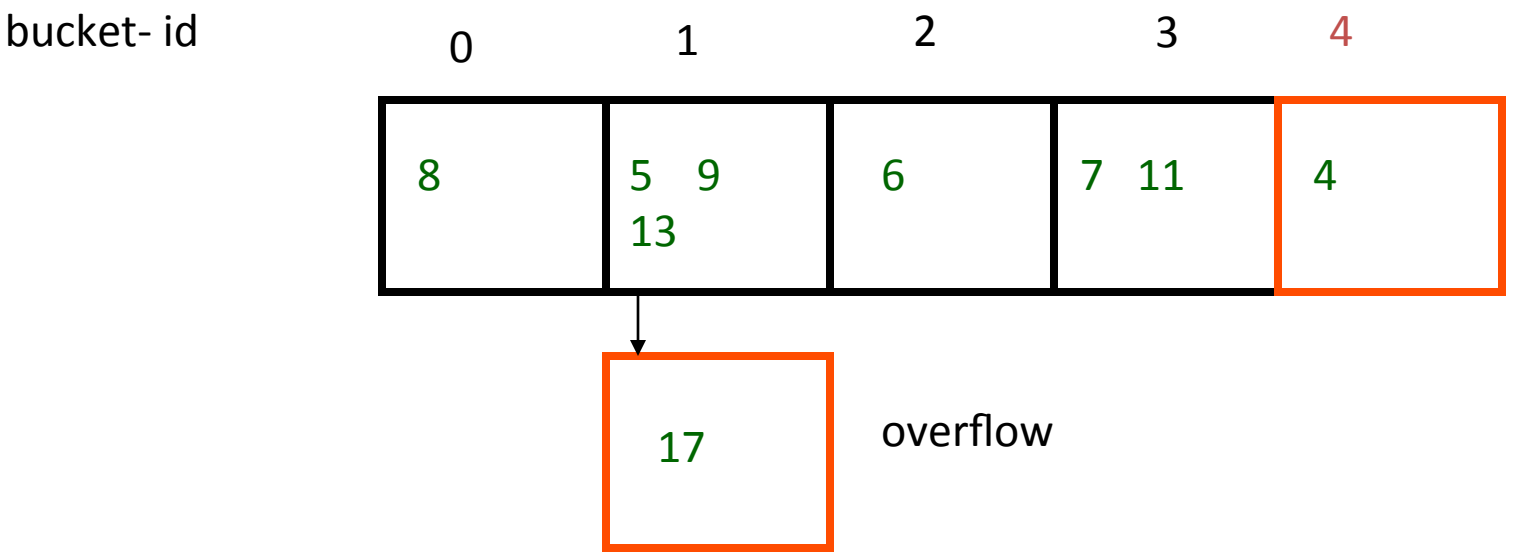
*h1(x) = x mod (2*N)*



Linear hashing - after split:

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

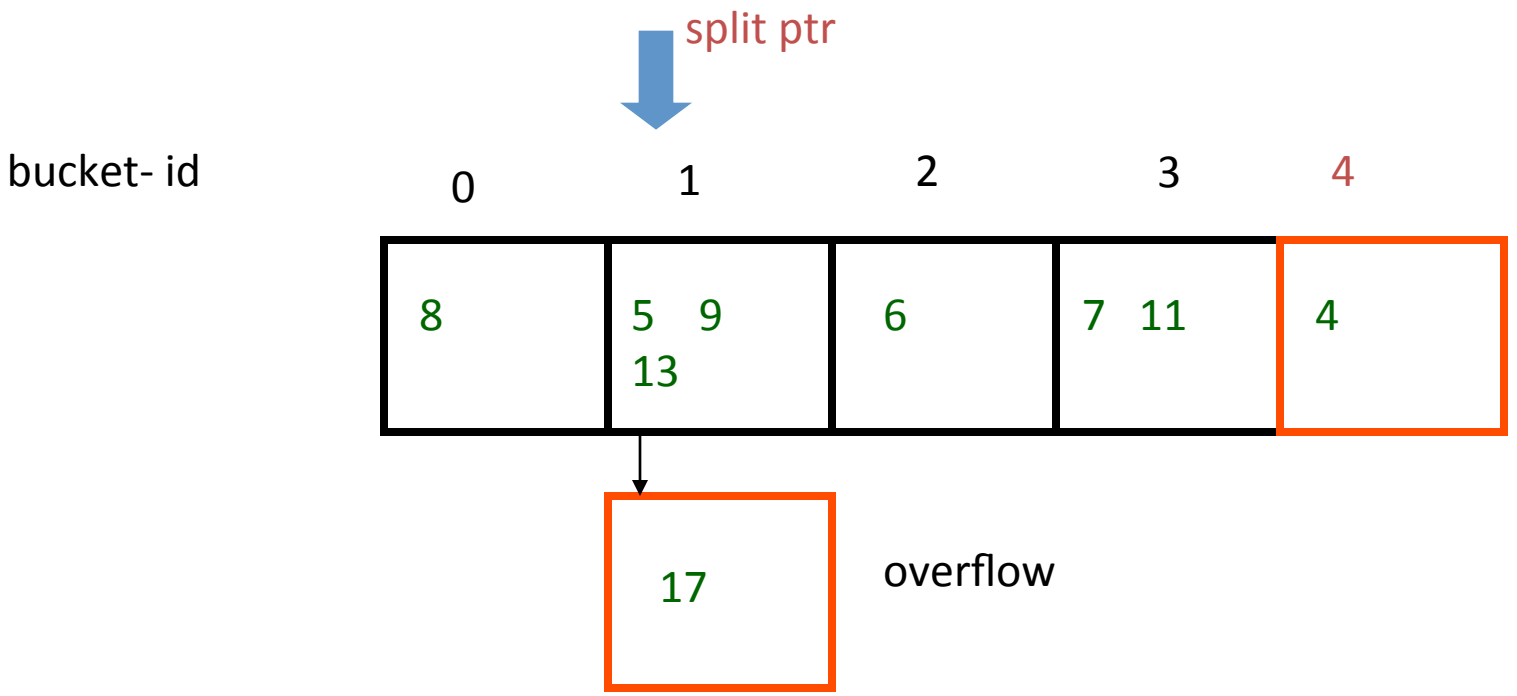
*h1(x) = x mod (2*N)*



Linear hashing - after split:

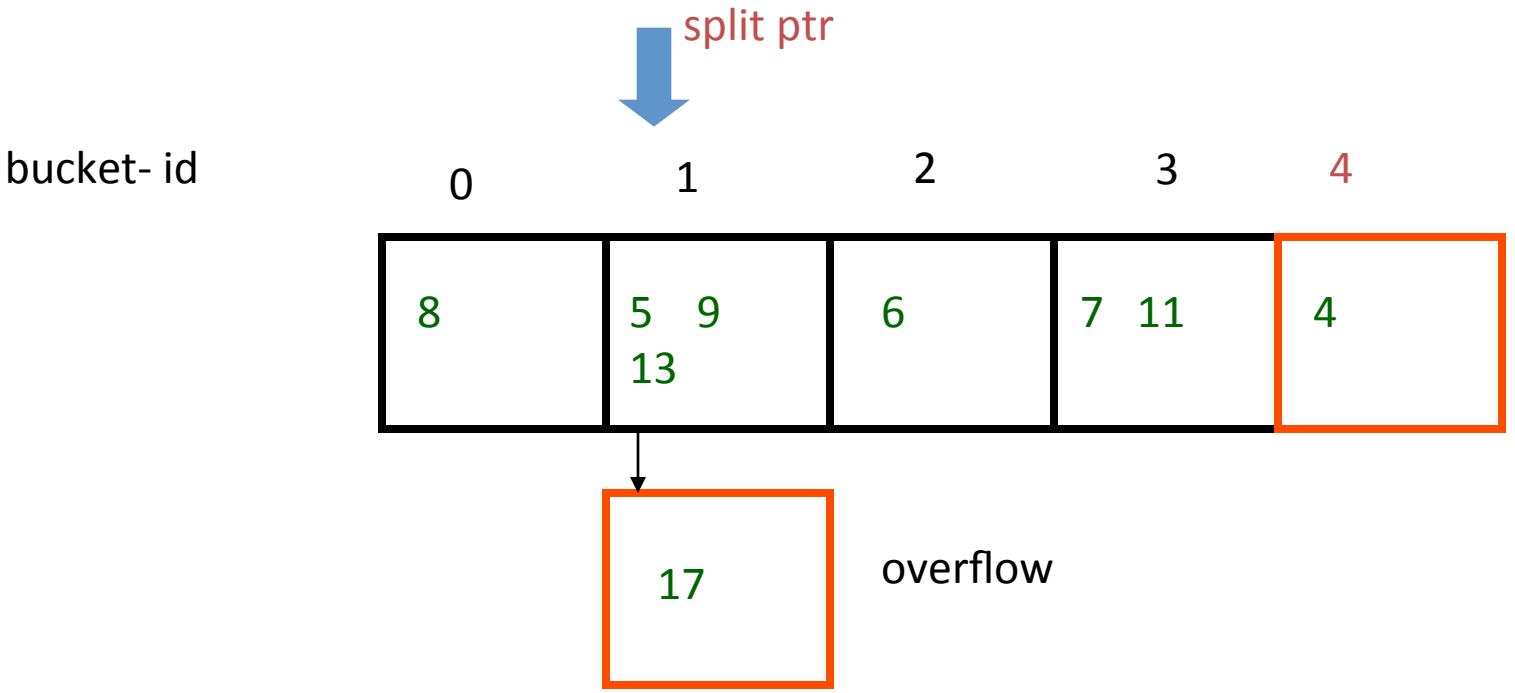
A: use two h.f.: *h0(x) = x mod N*

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



Linear hashing - searching?

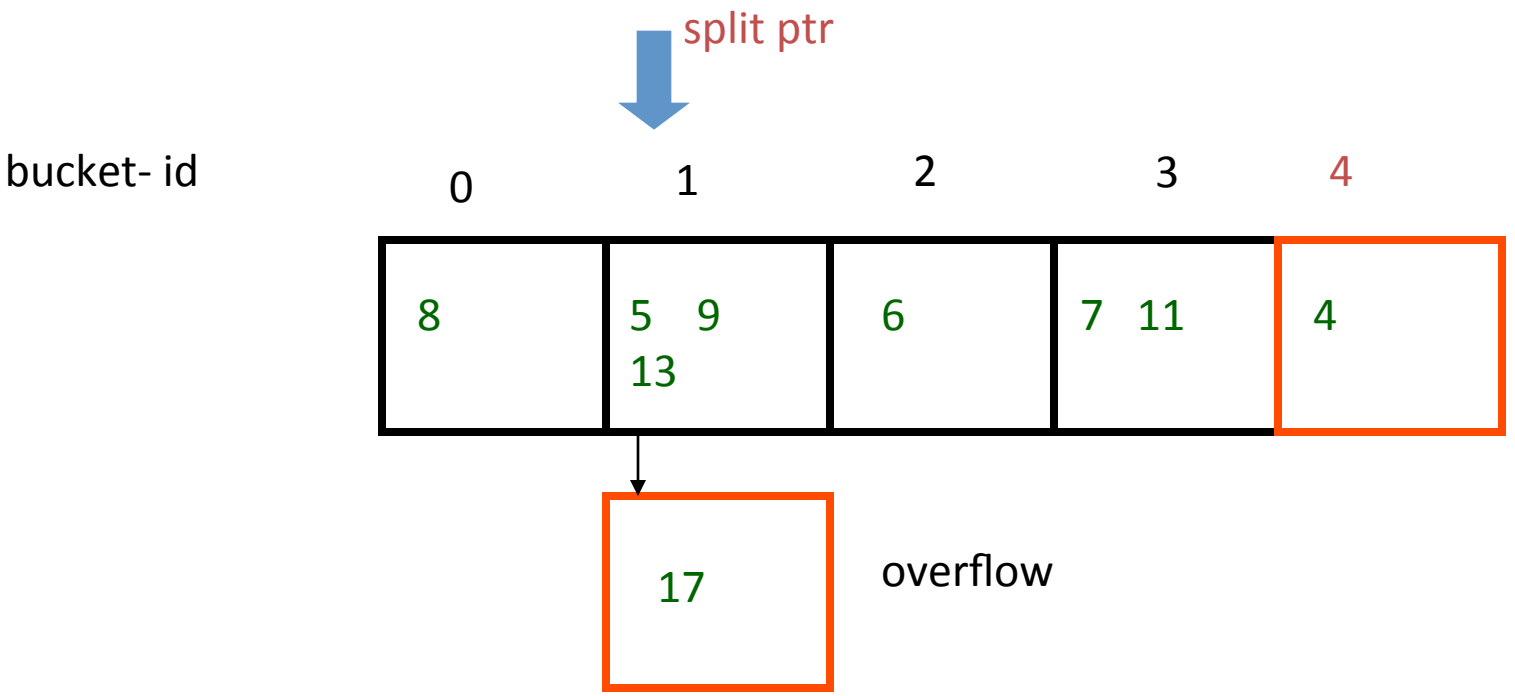
*h0(x) = x mod N (for the un-split buckets) h1(x) = x mod (2*N) (for the splitted ones)*



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-ptr*, compute *b=h1(k)*
- search bucket *b*

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?

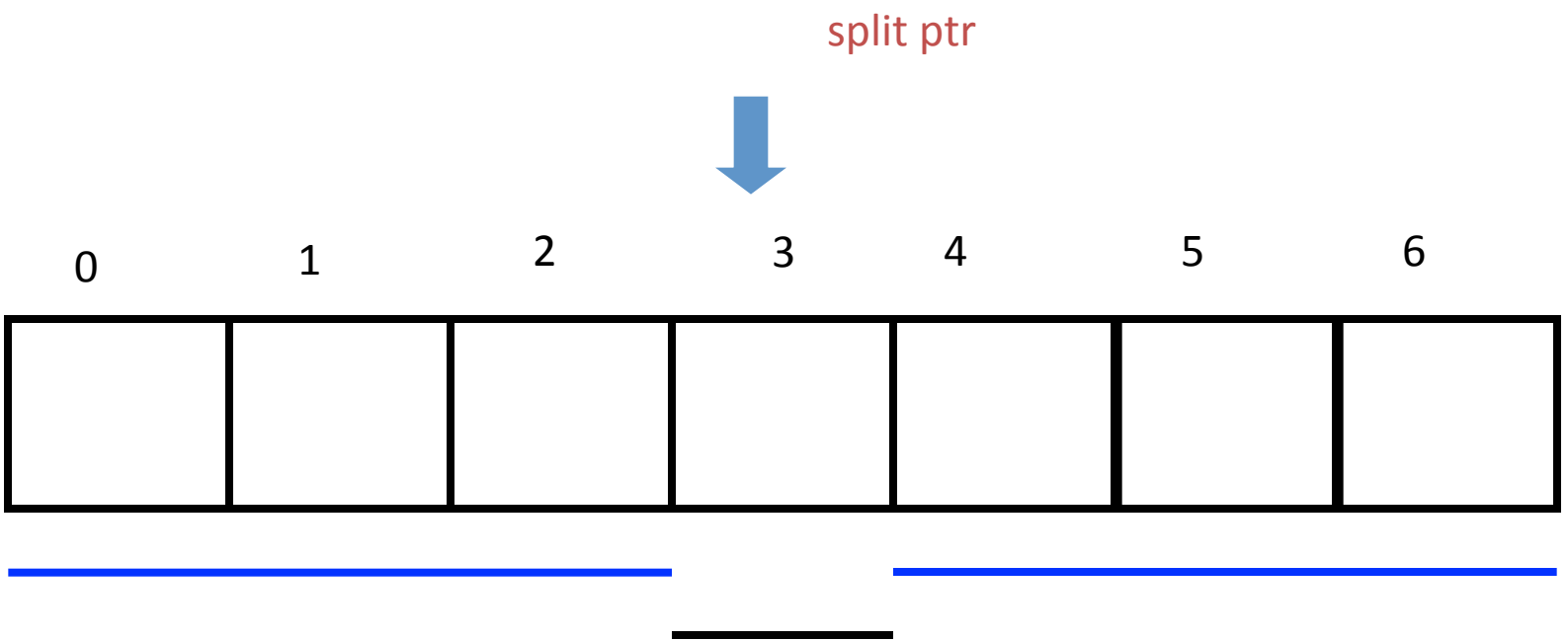
Algo: insert key '*k*'

- 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??

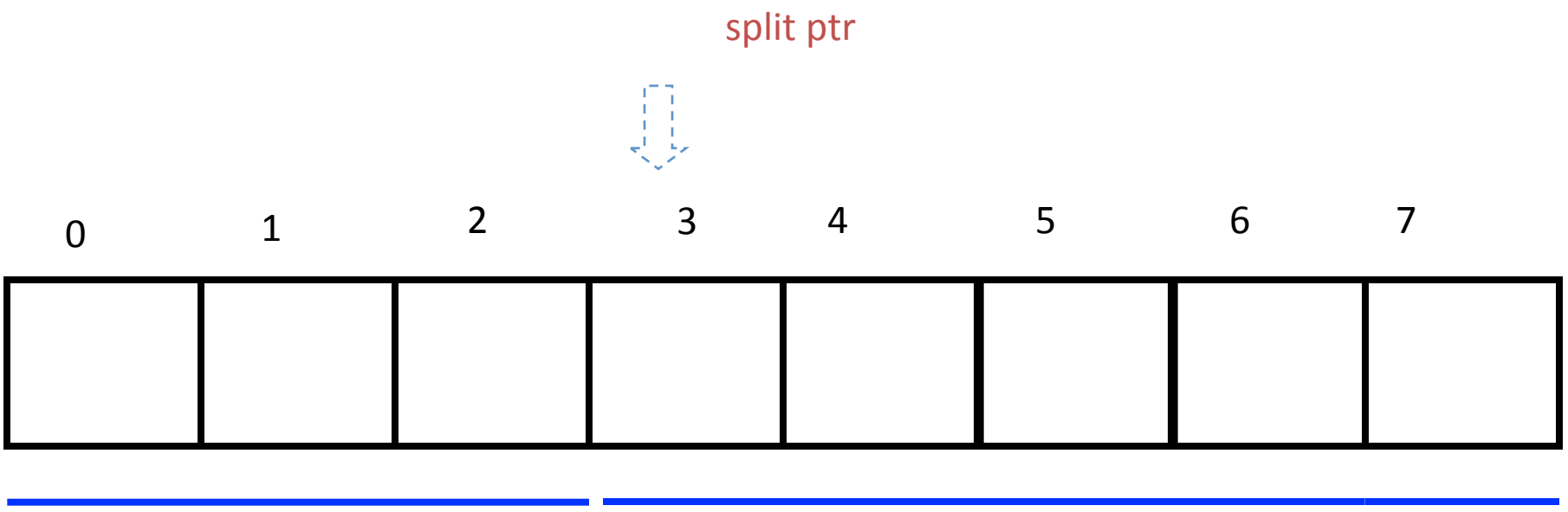
Linear hashing - split now?

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



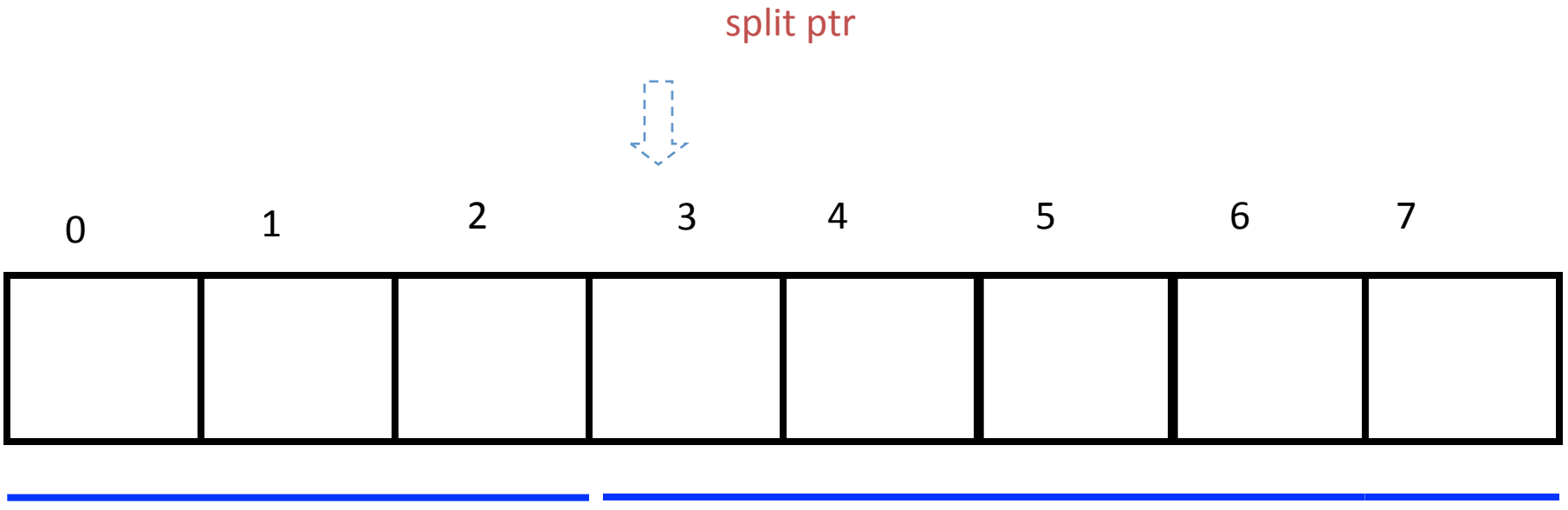
Linear hashing - split now?

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



Linear hashing - split now?

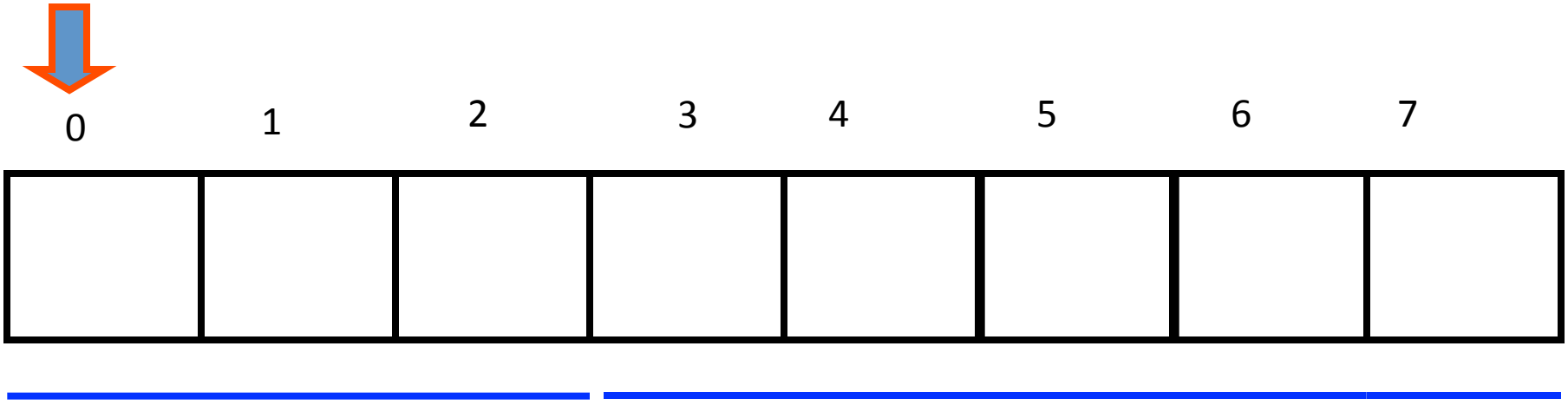
~~*h0(x) = x mod N (for the un-split buckets) h1(x) = x mod (2*N) (for the splitted ones)*~~



Linear hashing - split now?

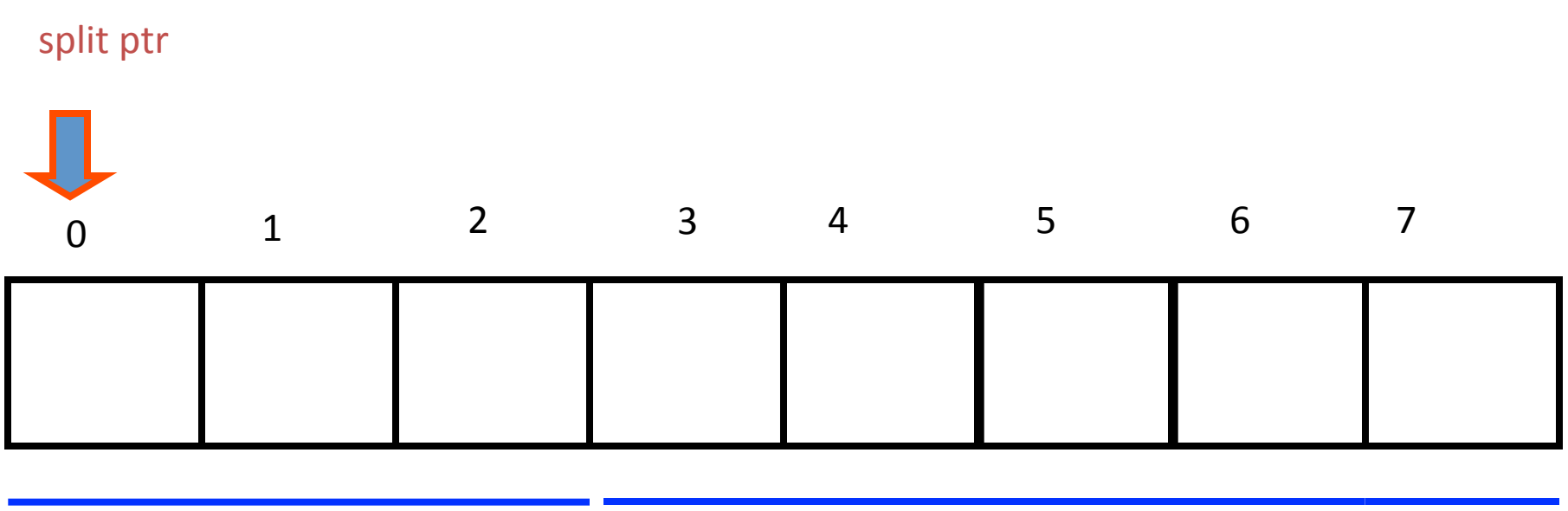
~~*h0(x) = x mod N (for the un-split buckets) h1(x) = x mod (2*N) (for the splitted ones)*~~

split ptr



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 - deletion?

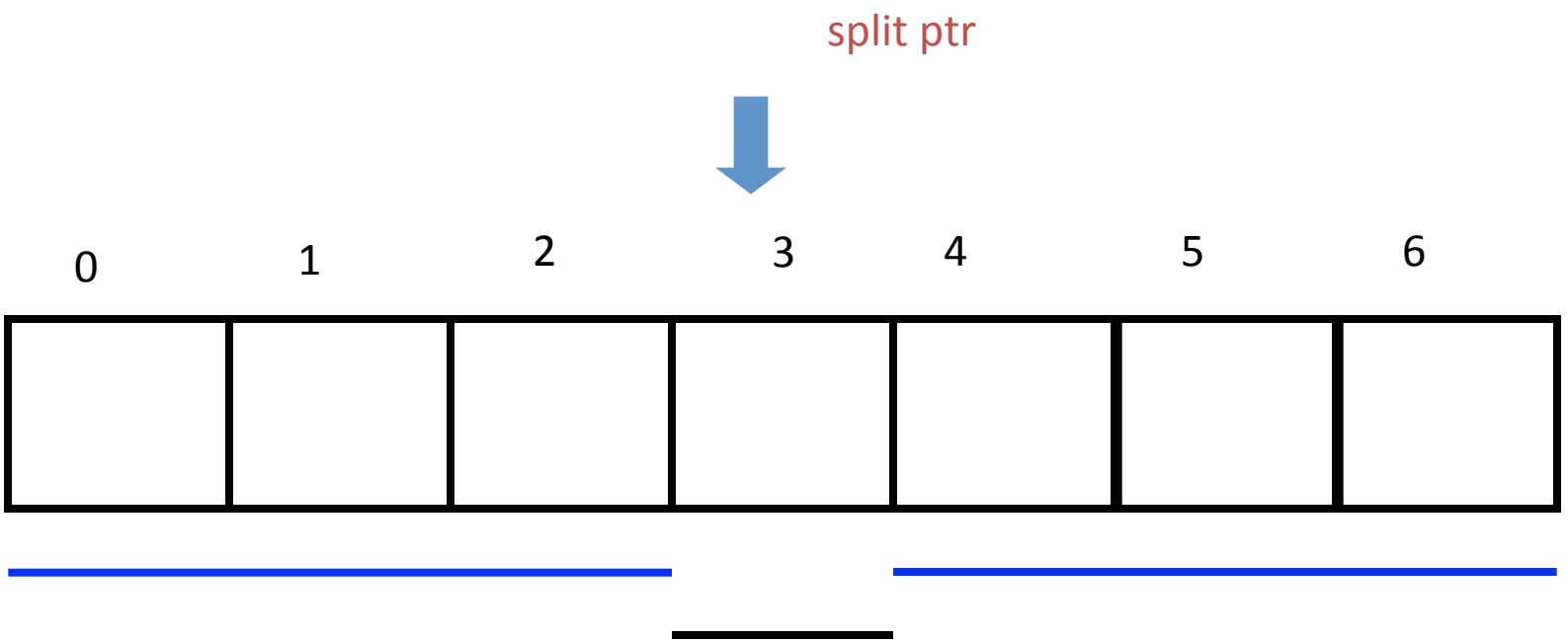
- reverse of insertion:

Linear hashing - deletion?

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

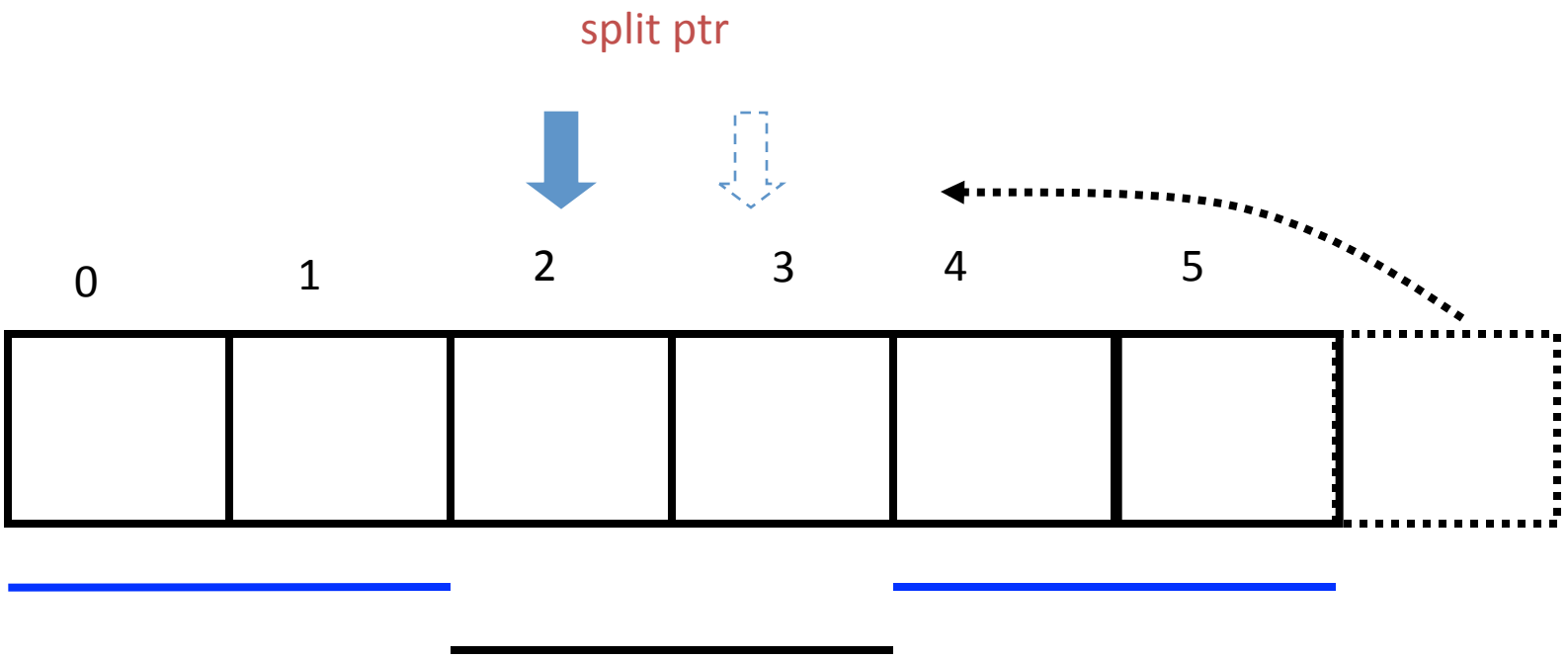
Linear hashing - how to contract?

*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)*



Hashing - pros?

Hashing - pros?

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

B(+)-trees - pros?

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)

Conclusions

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

SORTING

Why Sort?

Why Sort?

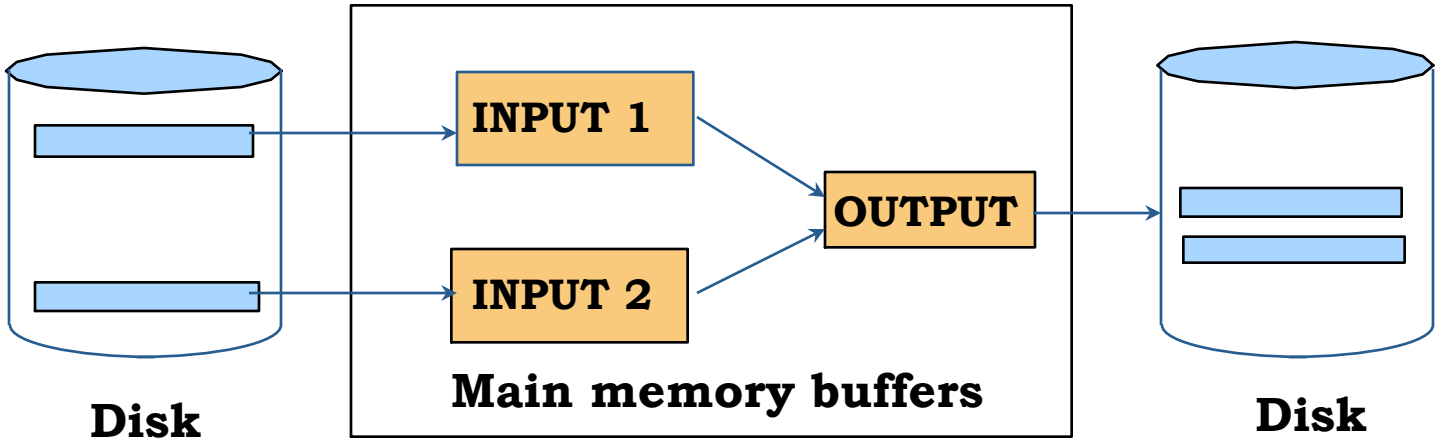
- select ... order by
  - e.g., find students in increasing *gpa* order
- *bulk loading* B+ tree index.
- *duplicate elimination* (select distinct)
- select ... group by
- *Sort-merge* join algorithm involves sorting.

Outline

- two-way merge sort
- external merge sort
- fine-tunings
- B+ trees for sorting

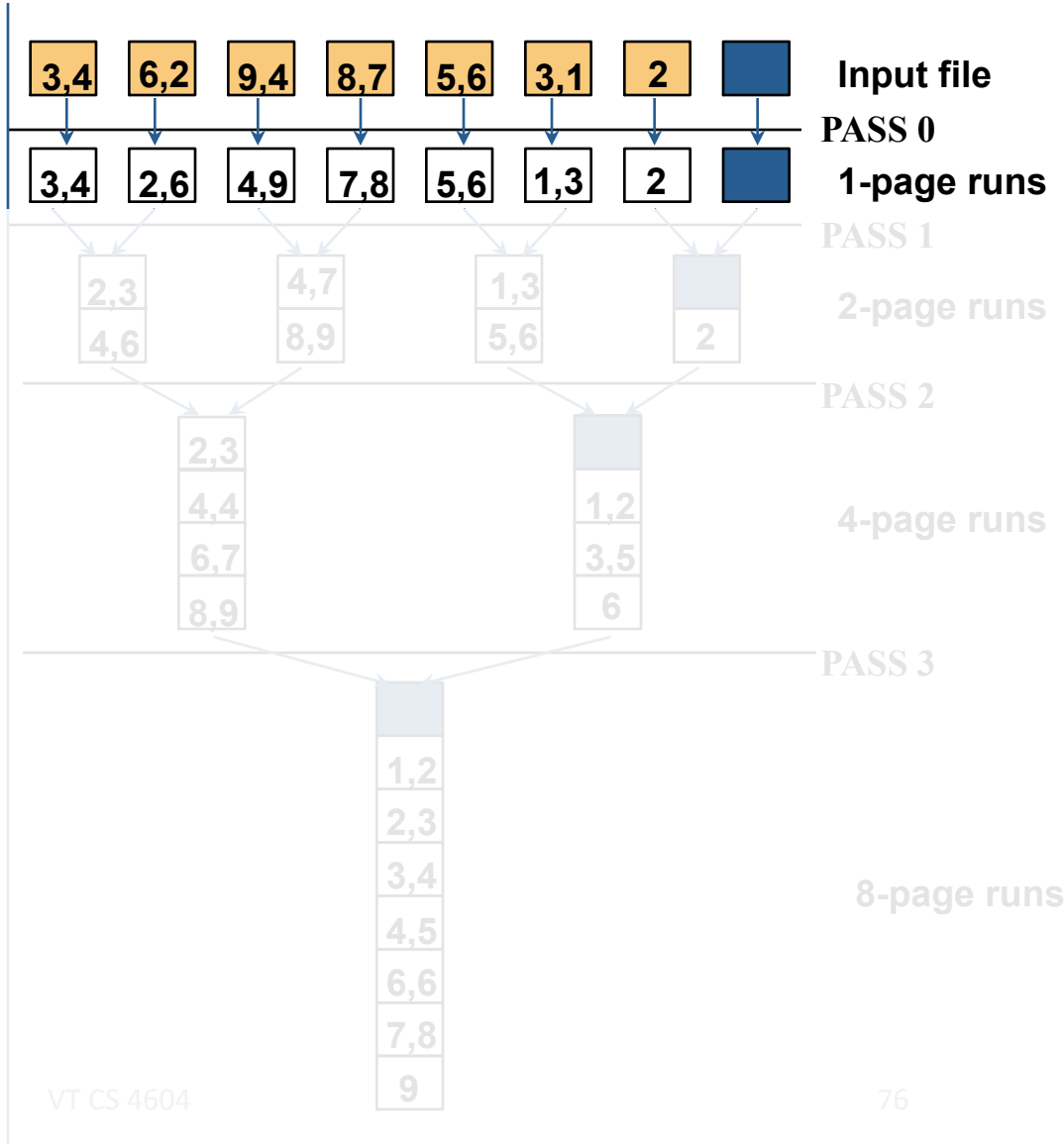
2-Way Sort: Requires 3 Buffers

- Pass 0: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 1, 2, 3, …, etc.: requires 3 buffer pages
  - merge pairs of **runs** into runs twice as long
  - three buffer pages used.



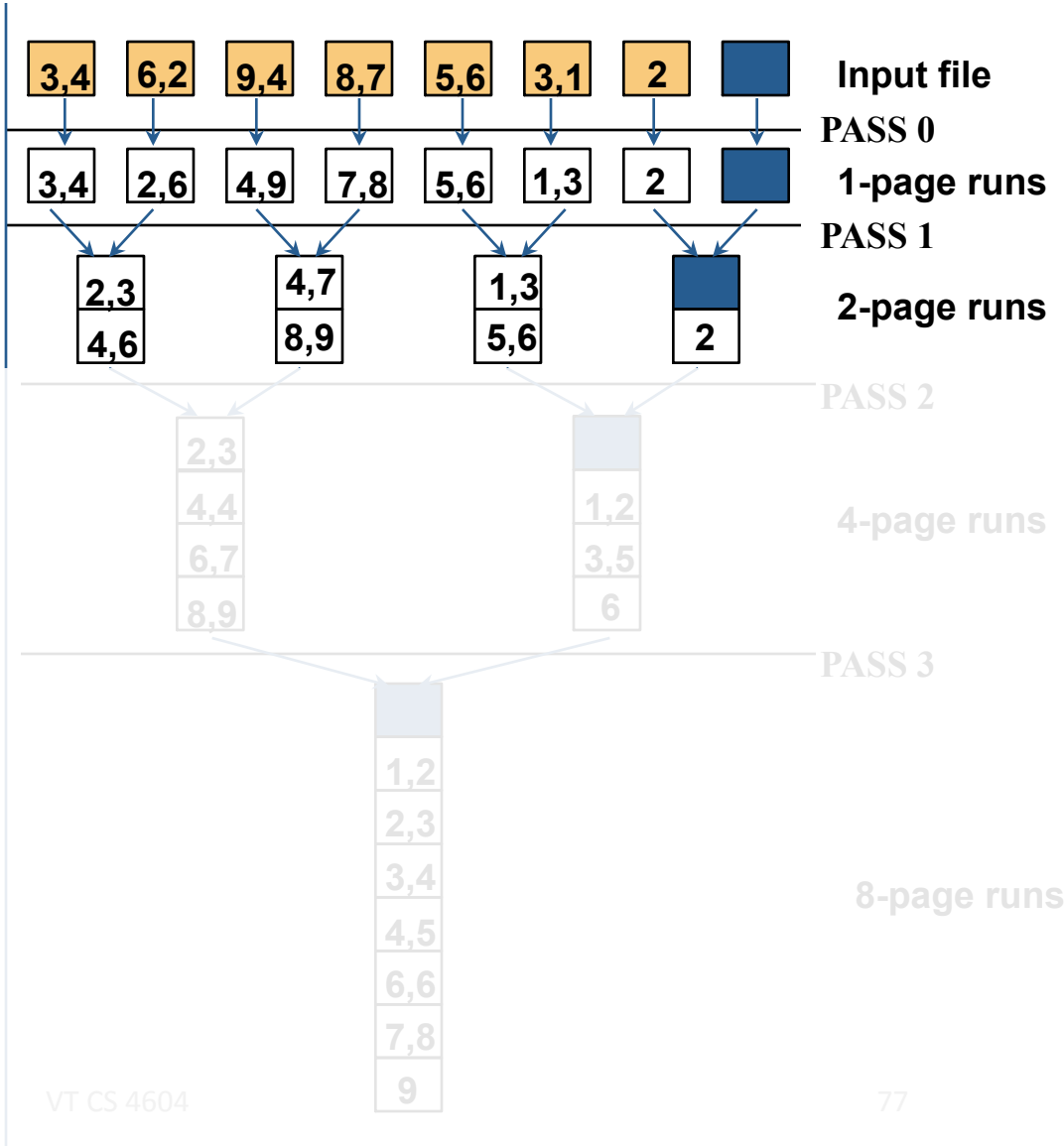
Two-Way External Merge Sort

- Each pass we read + write each page in file.



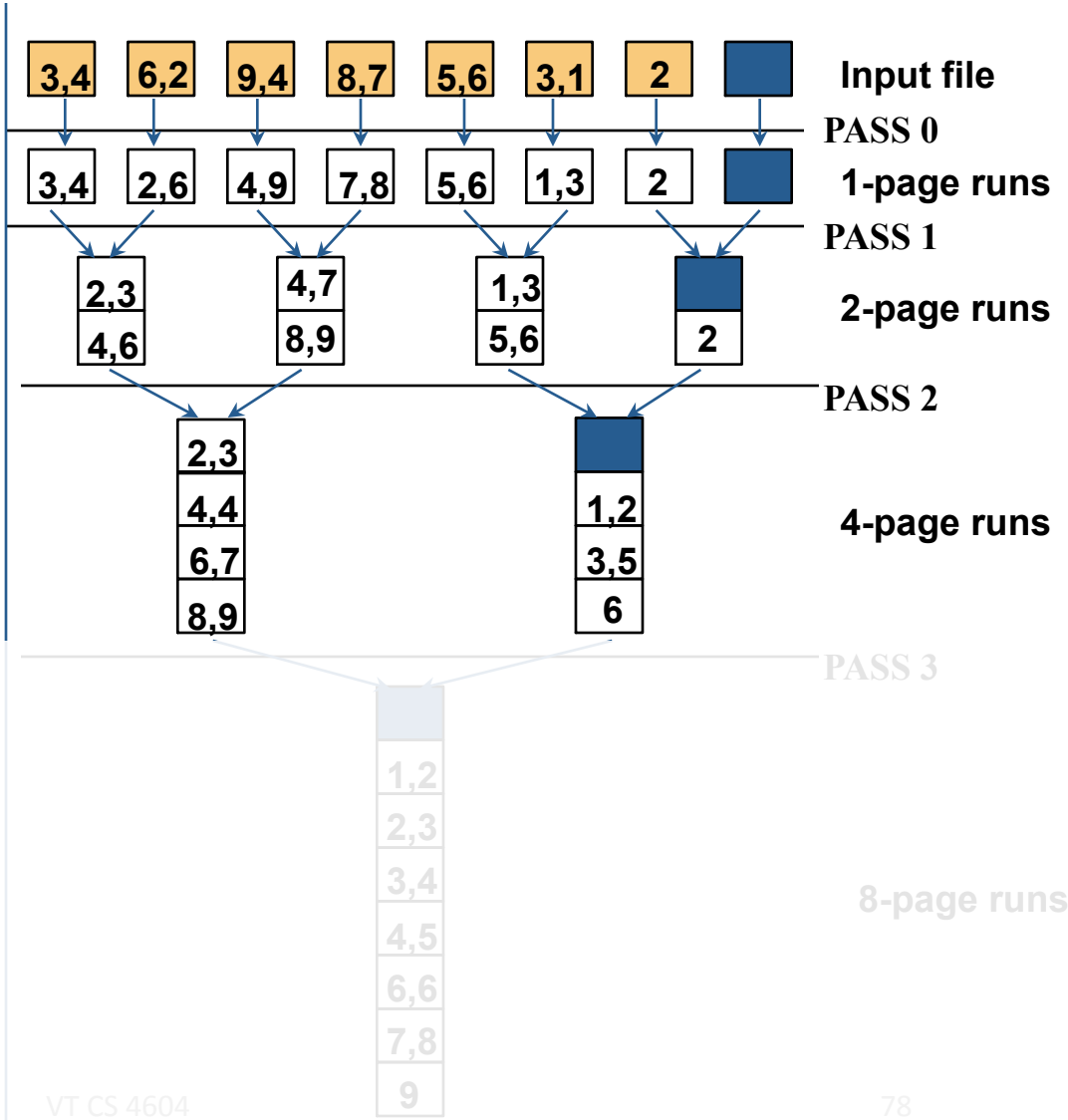
Two-Way External Merge Sort

- Each pass we read + write each page in file.



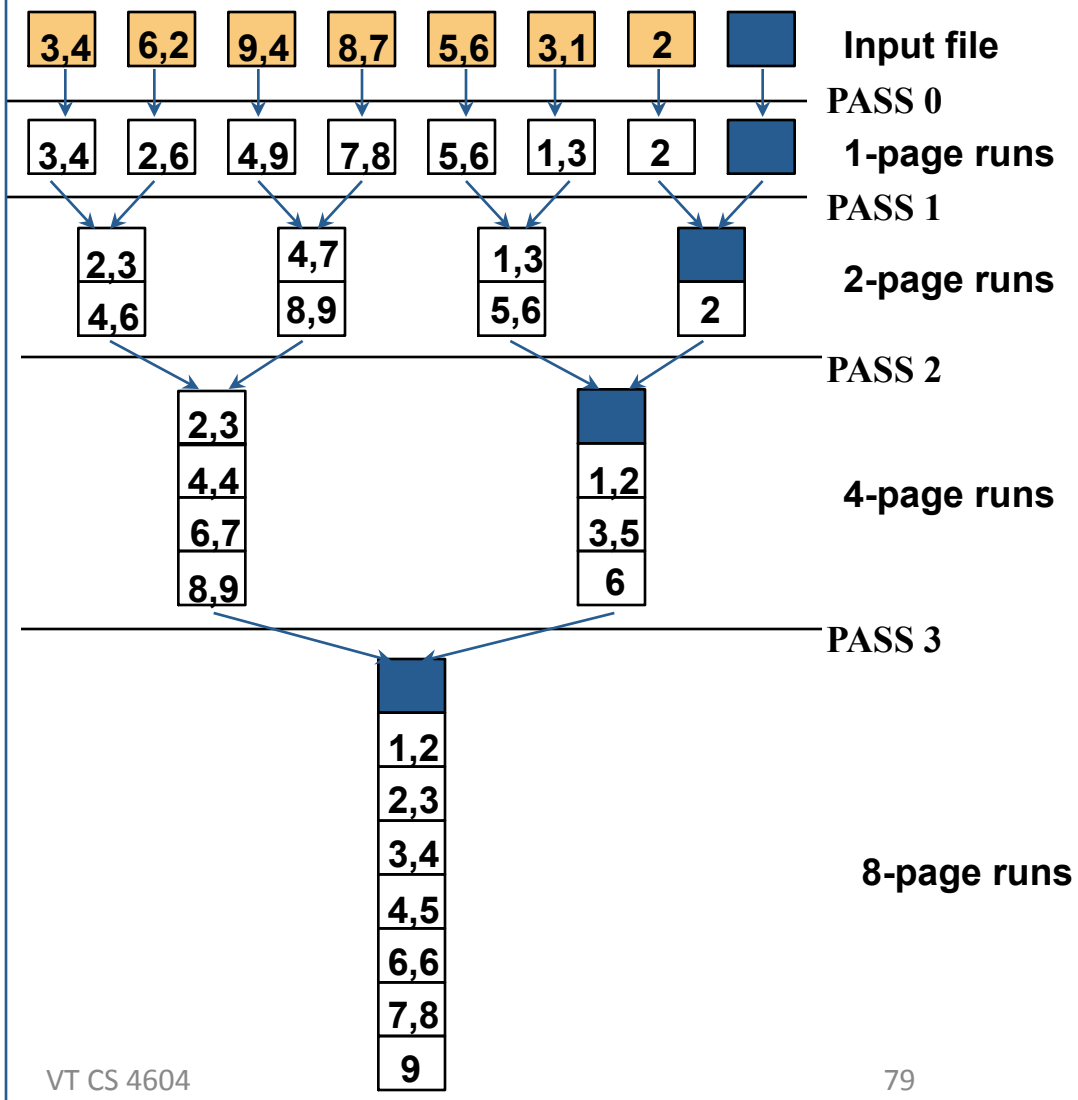
Two-Way External Merge Sort

- Each pass we read + write each page in file.



Two-Way External Merge Sort

- Each pass we read + write each page in file.



Two-Way External Merge Sort

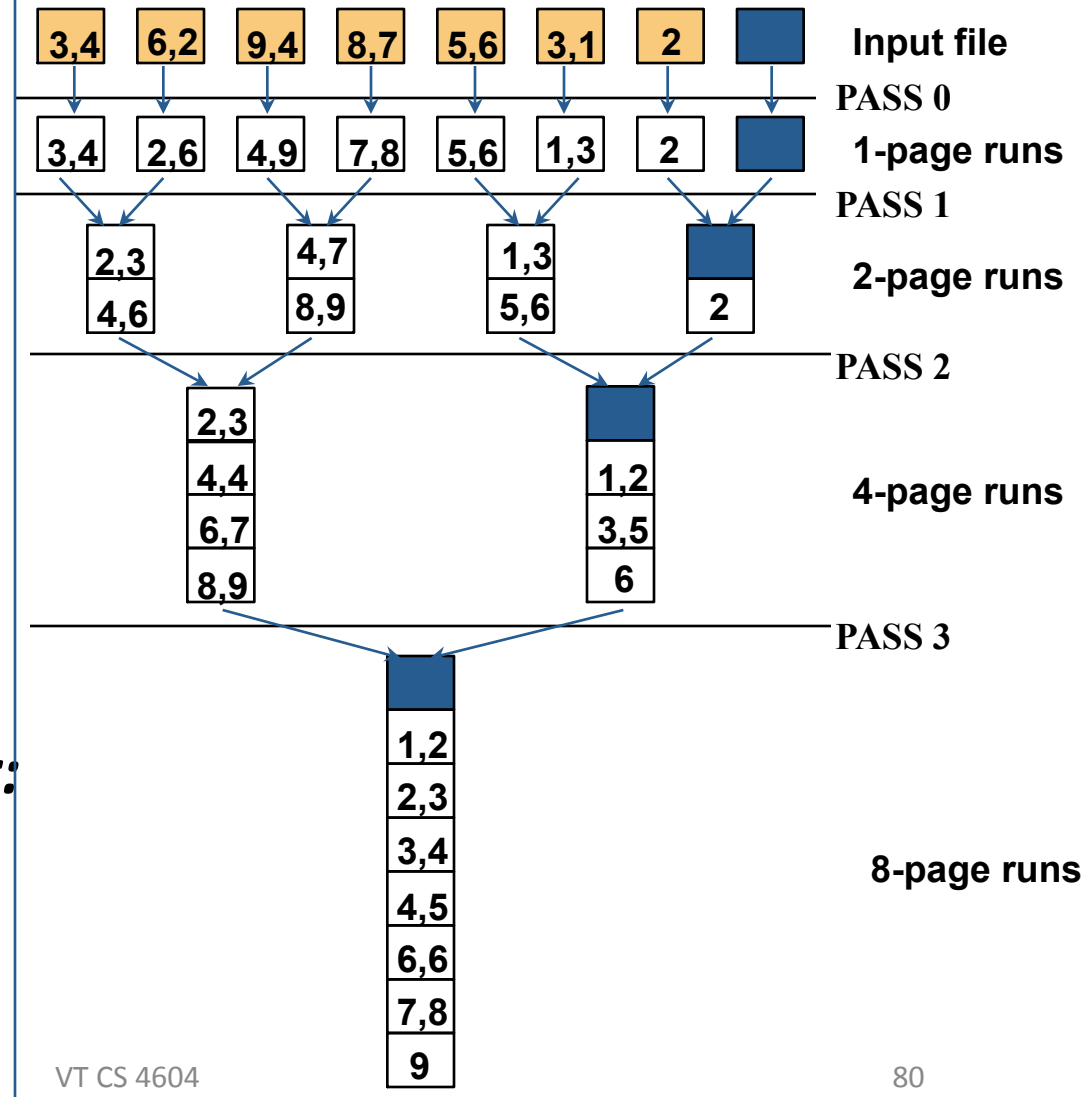
- Each pass we read + write each page in file.
- N pages in the file =>

$$= \lceil \log_2 N \rceil + 1$$

- So total cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

- *Idea:* ***Divide and conquer:*** sort subfiles and merge



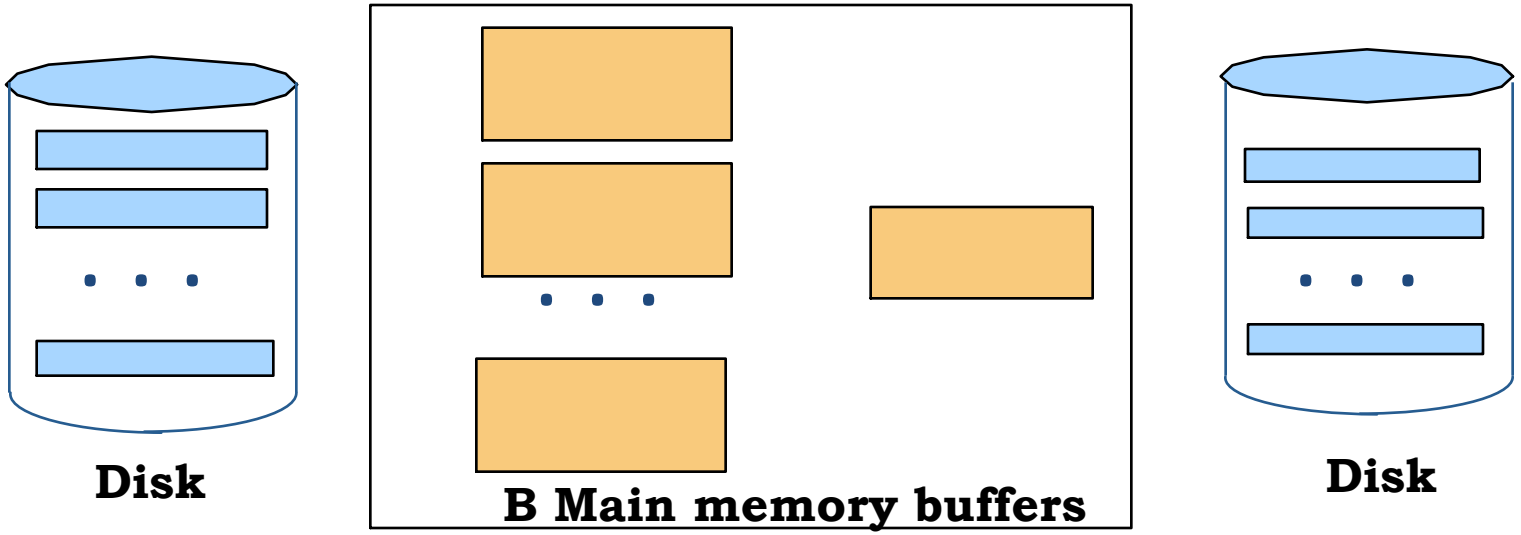
External merge sort

B > 3 buffers

- Q1: how to sort?
- Q2: cost?

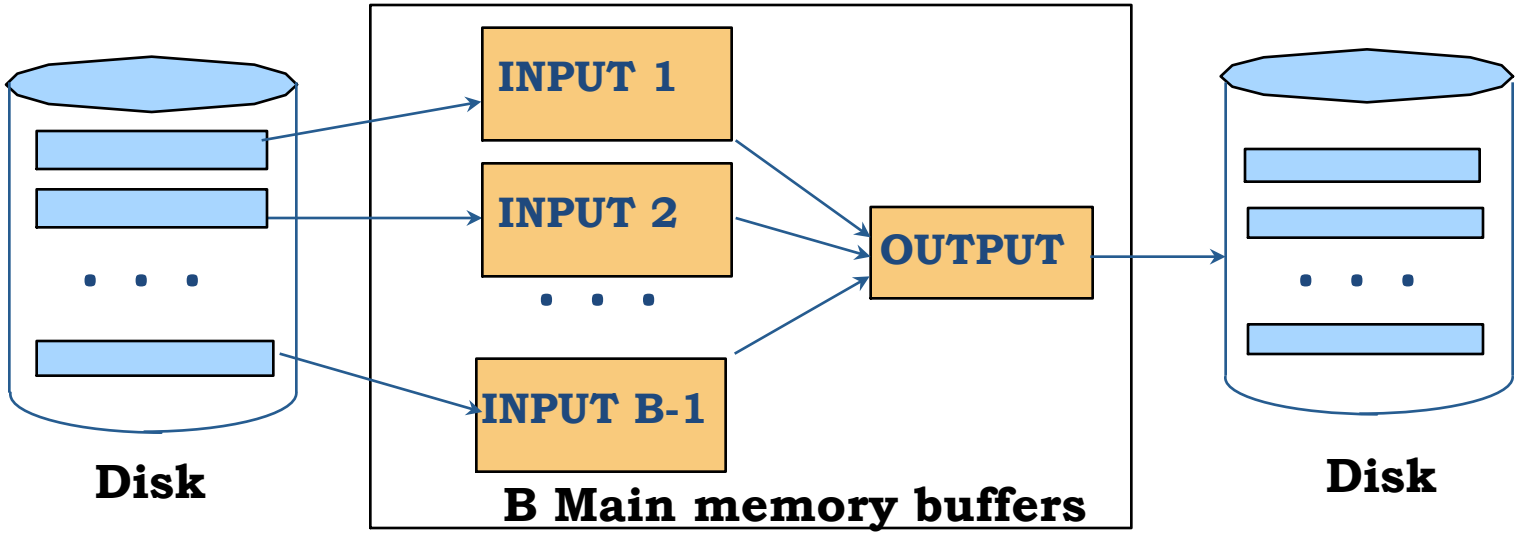
General External Merge Sort

B>3 buffer pages. How to sort a file with N pages?



General External Merge Sort

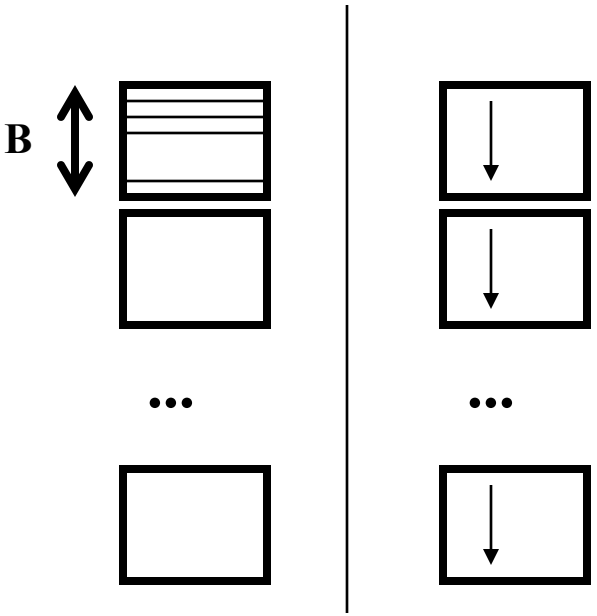
- Pass 0: use *B* buffer pages. Produce $\lceil N / B \rceil$ sorted runs of *B* pages each.
- Pass 1, 2, …, etc.: merge *B-1* runs.



Sorting

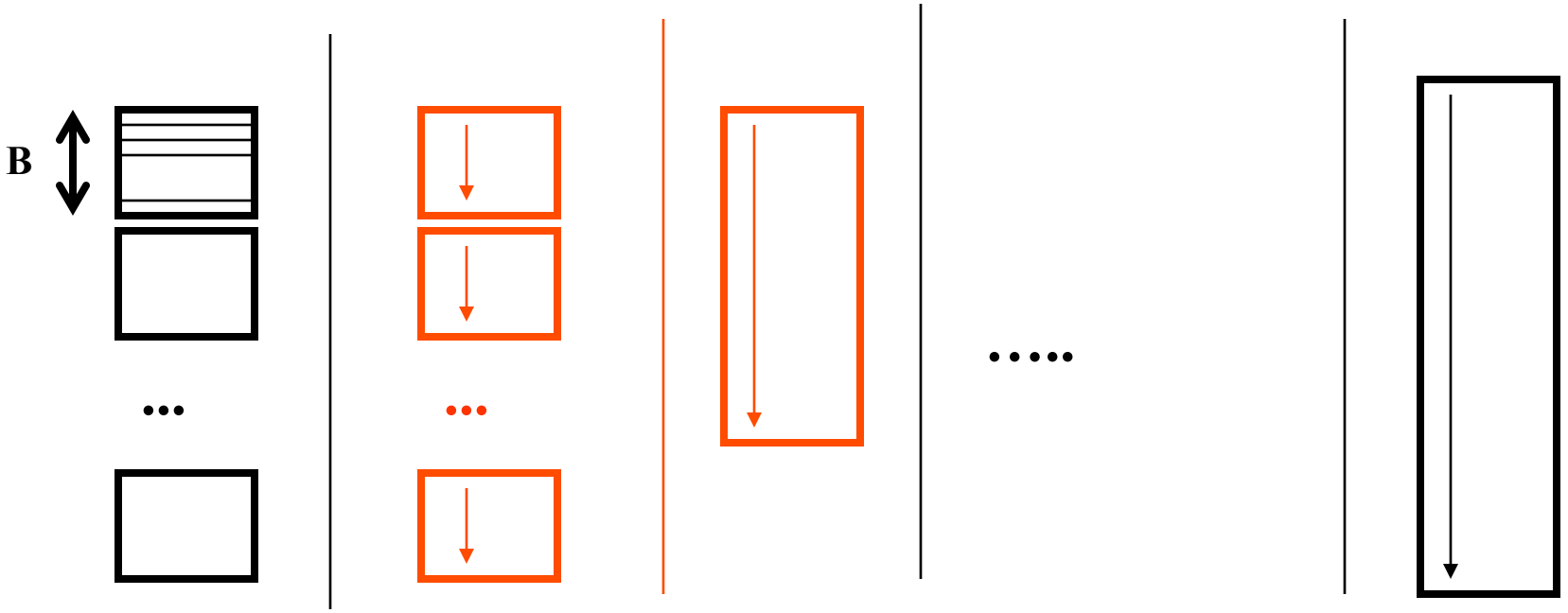
– create sorted runs of size B (how many?)

– merge them (how?)



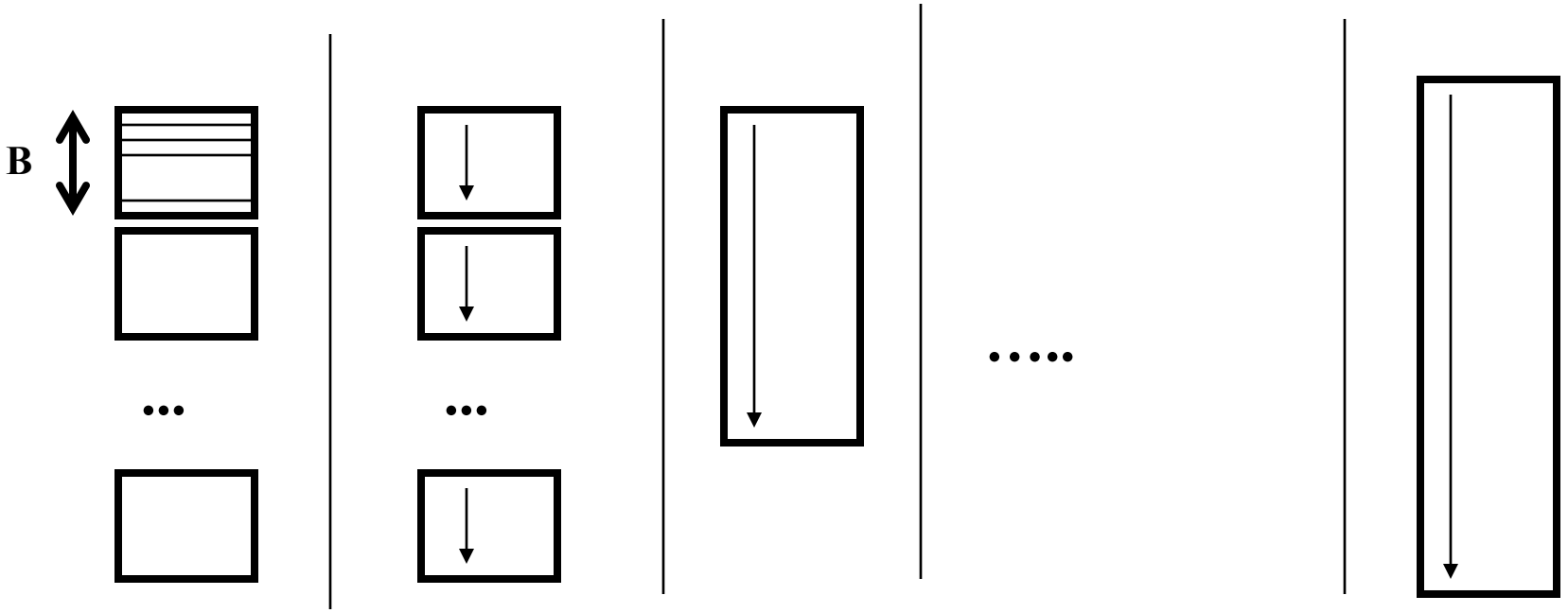
Sorting

- create sorted runs of size B
- merge first B-1 runs into a sorted run of (B-1) *B, ...



Sorting

- How many steps we need to do?
  'i', where B*(B-1)^i > N
- How many reads/writes per step? N+N



Cost of External Merge Sort

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Cost = 2N * (# of passes)

Cost of External Merge Sort

- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each (last run is only 3 pages)
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each (last run is only 8 pages)
  - Pass 2: 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Formula check: $\lceil \log_4 22 \rceil = 3 \ldots + 1 \rightarrow$ 4 passes √

Number of Passes of External Sort

(I/O cost is 2N times number of passes)

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Internal Sort Algorithm

- Quicksort is a fast way to sort in memory.

Blocked I/O & double-buffering

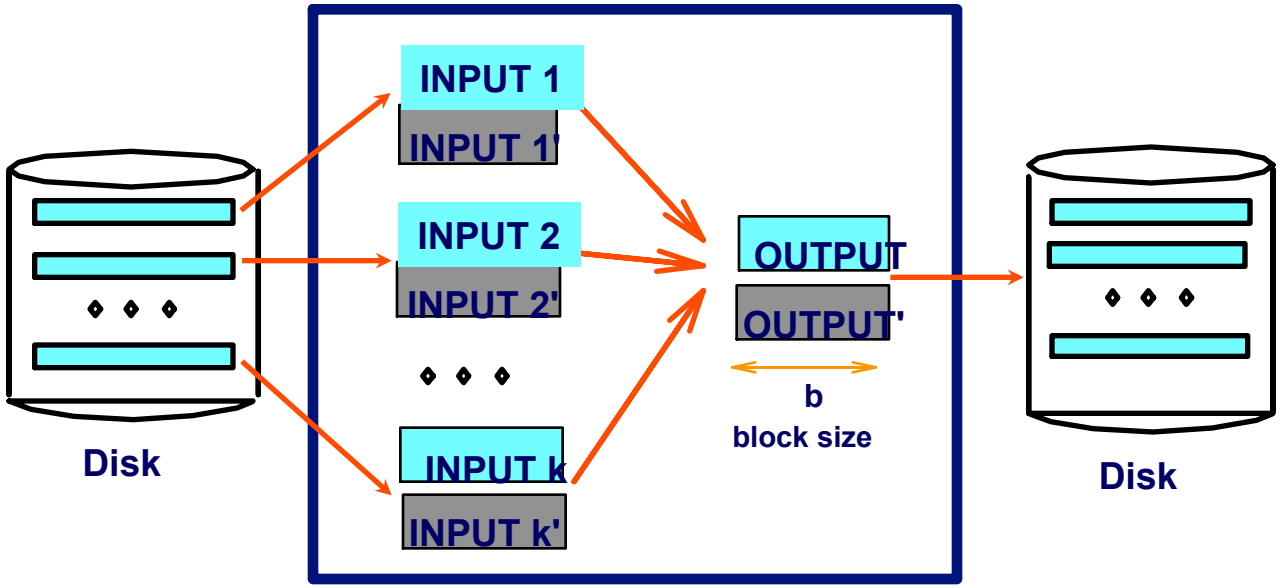
- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?

Blocked I/O & double-buffering

- So far, we assumed random disk access
- Cost changes, if we consider that runs are written (and read) sequentially
- What could we do to exploit it?
- A1: Blocked I/O (exchange a few r.d.a for several sequential ones)
- A2: double-buffering

Double Buffering

- To reduce wait time for I/O request to complete, can *prefetch* into `*shadow block*' .
  - Potentially, more passes; in practice, most files still sorted in 2-3 passes.



Using B+ Trees for Sorting

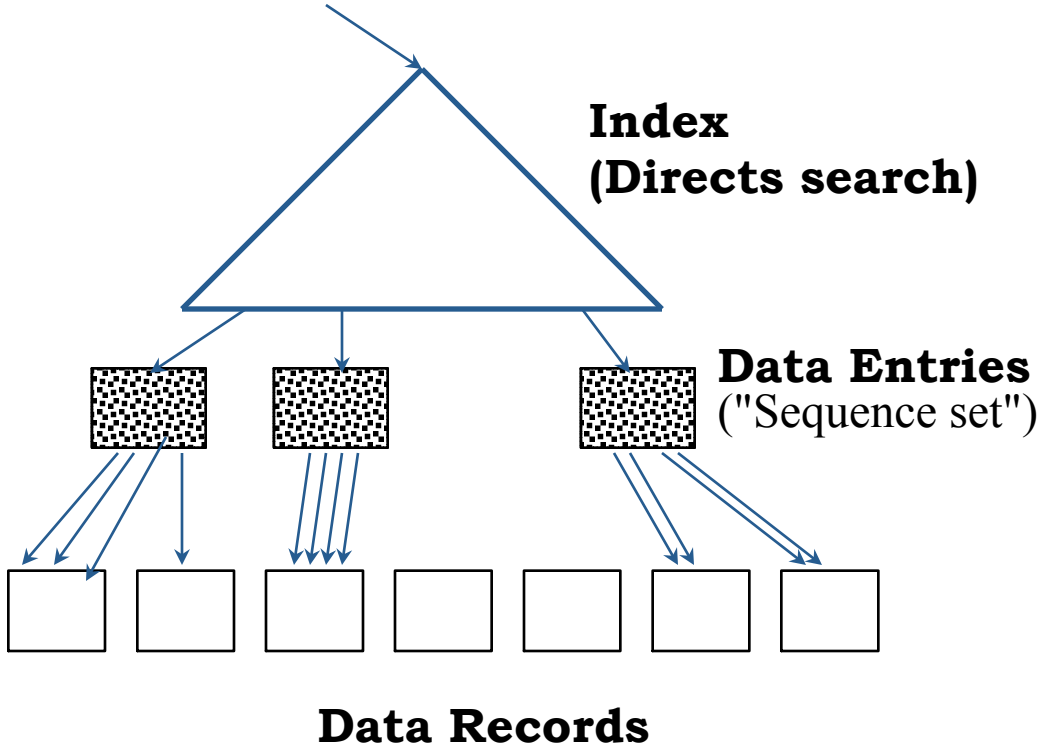
- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- *Idea*: Can retrieve records in order by traversing leaf pages.
- *Is this a good idea?*
- Cases to consider:
  - B+ tree is clustered
  - B+ tree is not clustered

Using B+ Trees for Sorting

- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- *Idea*: Can retrieve records in order by traversing leaf pages.
- *Is this a good idea?*
- Cases to consider:
  - B+ tree is clustered Good idea!
  - B+ tree is not clustered Could be a very bad idea!

Clustered B+ Tree Used for Sorting

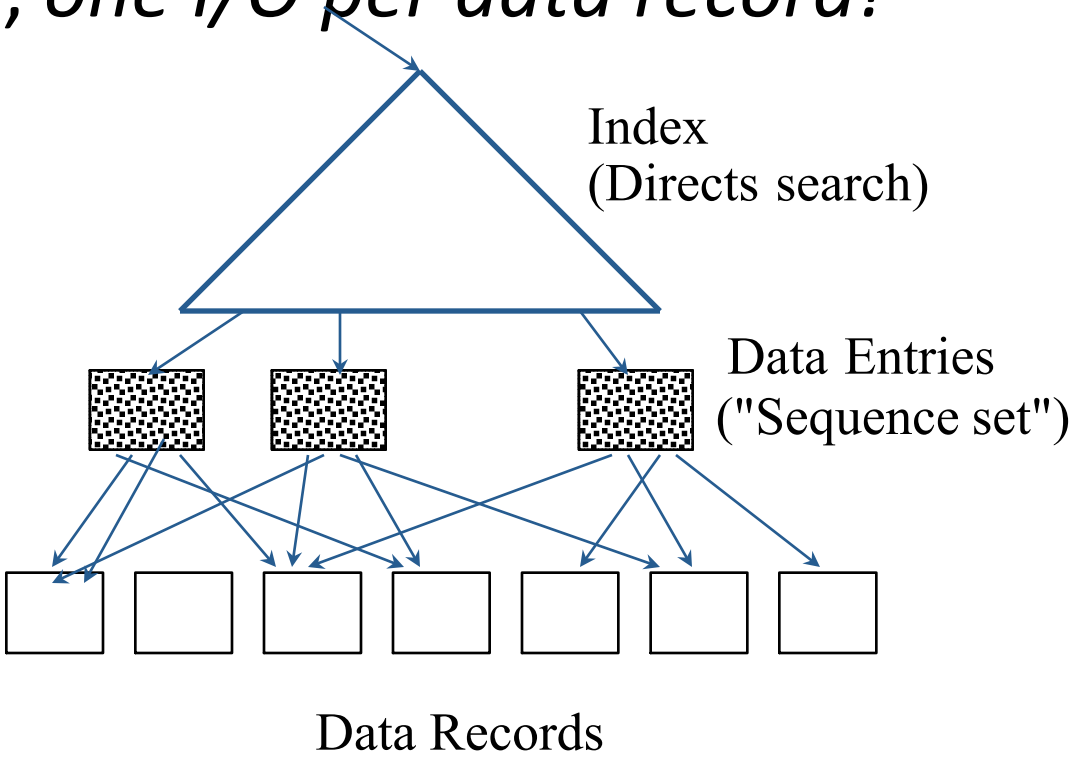
- Cost: root to the left-most leaf, then retrieve all leaf pages (Alternative 1)



Always better than external sorting!

Unclustered B+ Tree Used for Sorting

- Alternative (2) for data entries; each data entry contains *rid* of a data record. In general, *one I/O per data record!*



External Sorting vs. Unclustered Index

N	Sorting	p=1	p=10	p=100
100	200	100	1,000	10,000
1,000	2,000	1,000	10,000	100,000
10,000	40,000	10,000	100,000	1,000,000
100,000	600,000	100,000	1,000,000	10,000,000
1,000,000	8,000,000	1,000,000	10,000,000	100,000,000
10,000,000	80,000,000	10,000,000	100,000,000	1,000,000,000

***p*: # of records per page**

B=1,000 and block size=32 for sorting

p=100 is the more realistic value.

Summary

- External sorting is important
- External merge sort minimizes disk I/O cost:
  - Pass 0: Produces sorted ***runs*** of size ***B*** (# buffer pages).
  - Later passes: ***merge*** runs.
- Clustered B+ tree is good for sorting; unclustered tree is usually very bad.