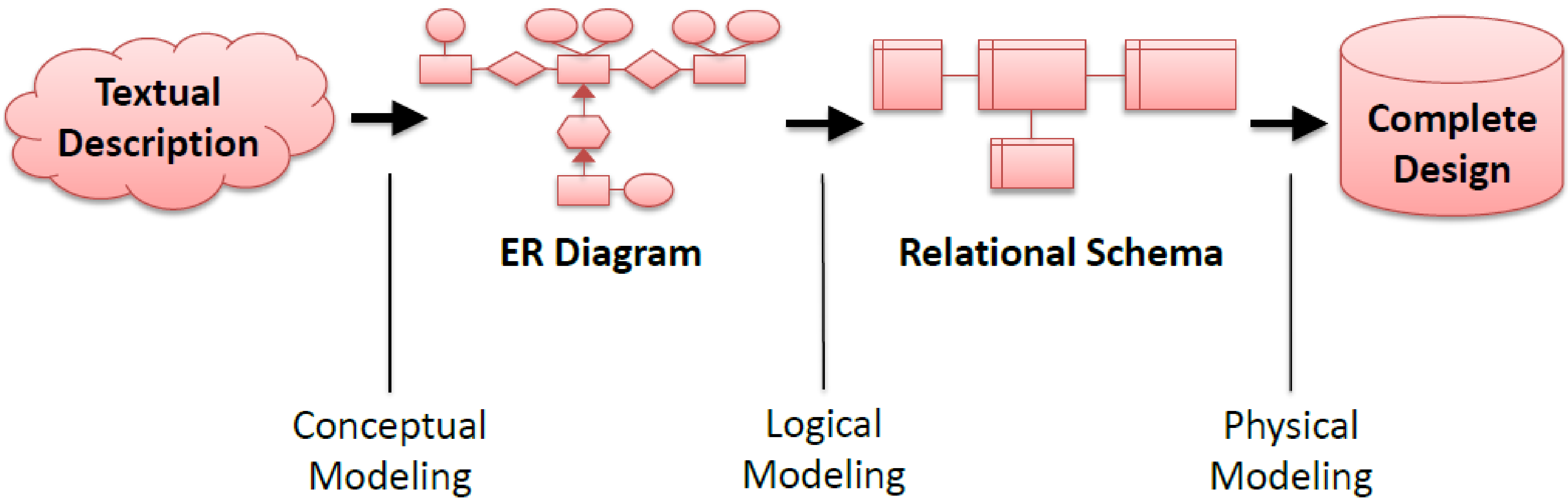
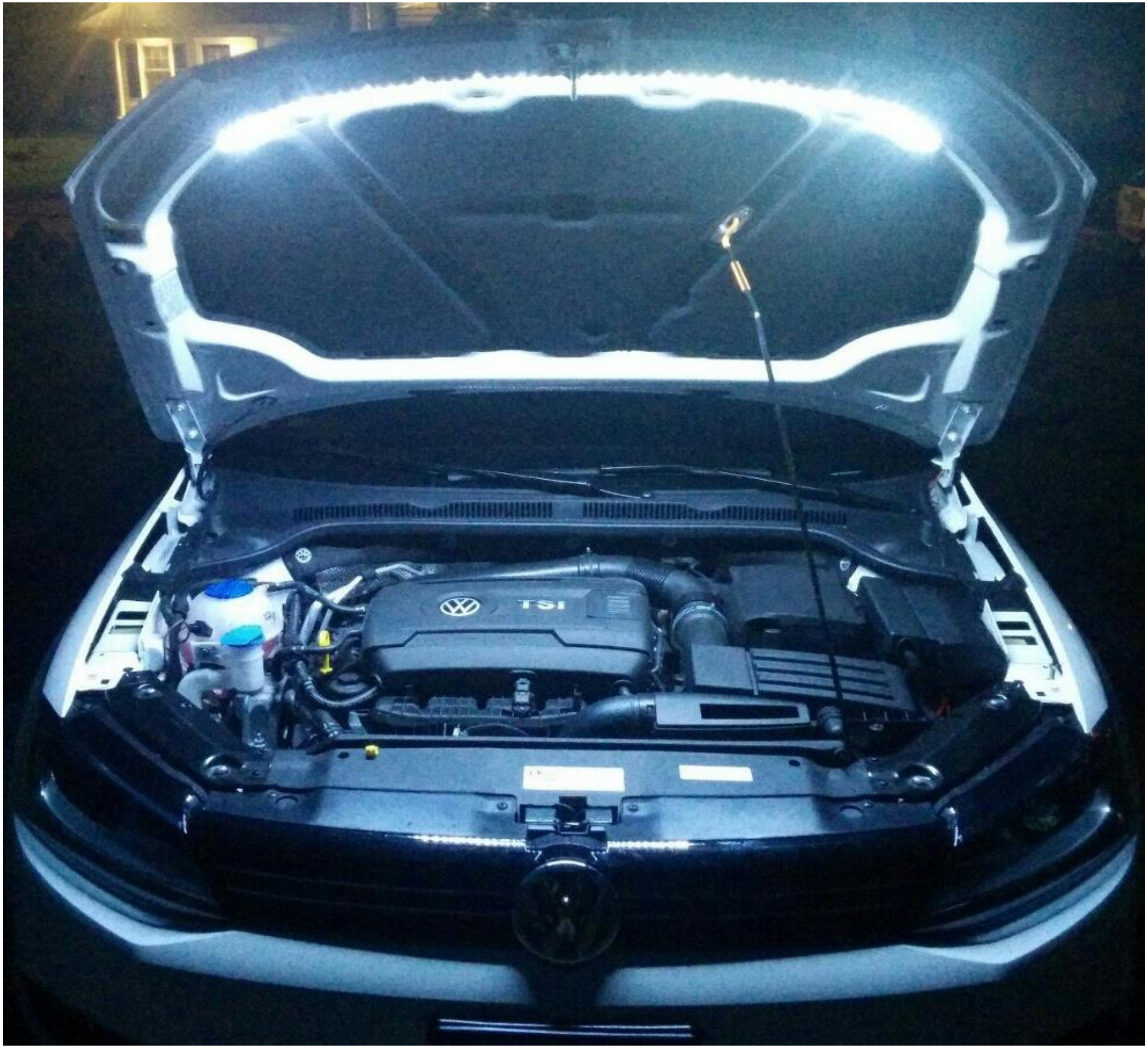
File management, Hash & External sorting

Monday, Oct. 12, 2015

Database Design



Under the hood of DBMS



Leverage OS for disk/file management?

- Layers of abstraction are good … but:
  - Unfortunately, OS often gets in the way of DBMS
- DBMS wants/needs to do things "its own way"
  - Specialized prefetching
  - Control over buffer replacement policy
  - Control over thread/process scheduling
    - Arises when OS scheduling conflicts with DBMS locking
  - Control over flushing data to disk

Disks and Files

- DBMS stores information on disks. (Really?)
  - but: disks are (relatively) VERY slow!
- Major implications for DBMS design:
  - READ: disk -> main memory (RAM).
  - WRITE: reverse
  - Both are high-cost operations, relative to in-memory operations, so must be planned carefully!

The Storage Hierarchy

Smaller, Faster

Registers

↕

L1 Cache

↕

Main Memory 

↕

SSD 

↕

Magnetic Disk ?

↕

Magnetic Tape ???

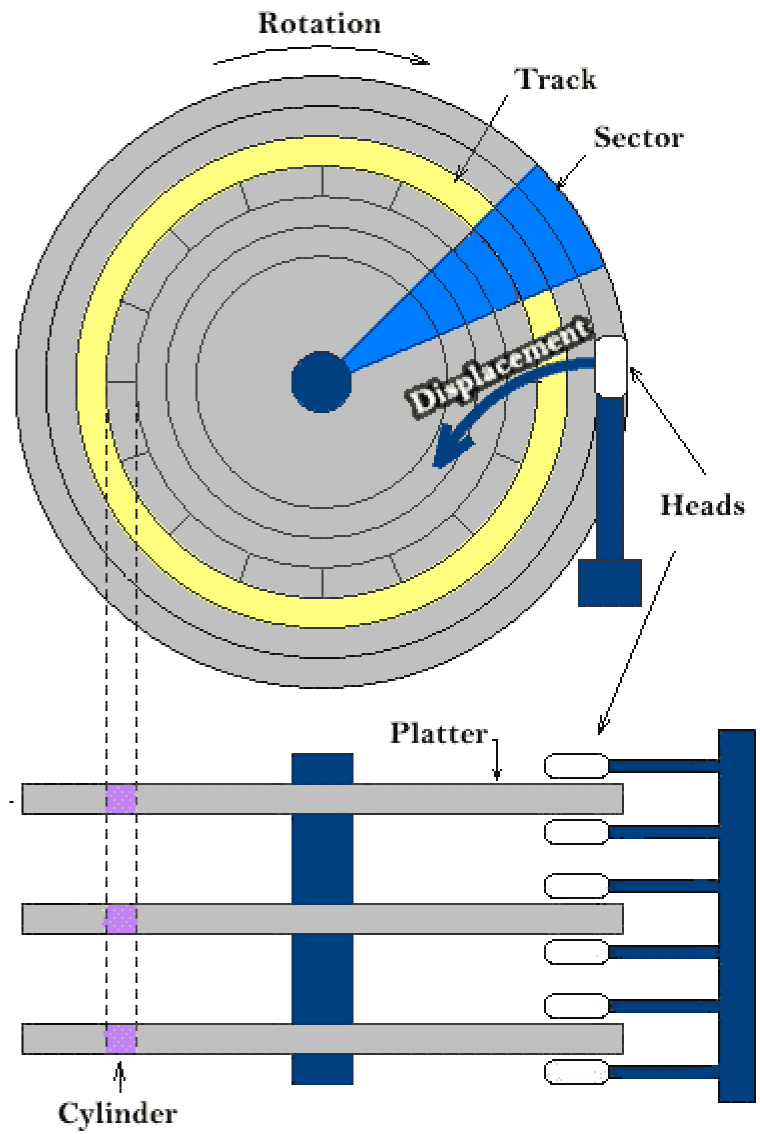
Bigger, Slower

Anatomy of a Disk

Unlike RAM, time to retrieve a disk page varies depending upon location on disk.

relative placement of pages on disk is important!

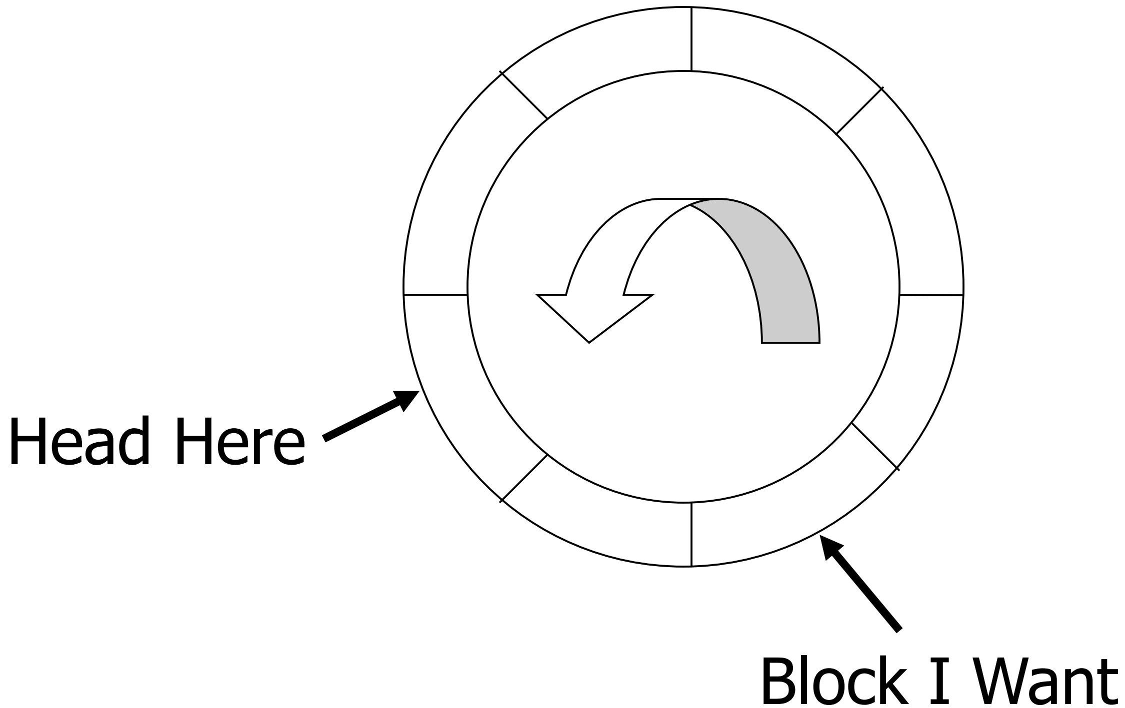
- Sector
- Track
- Cylinder
- Platter
- Block size = multiple of sector size (which is fixed)



Accessing a Disk Page

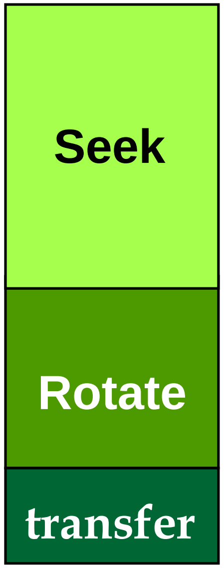
- Time to access (read/write) a disk block:
  - *seek time:* moving arms to position disk head on track
  - *rotational delay:* waiting for block to rotate under head
  - *transfer time:* actually moving data to/from disk surface

Rotational Delay



Accessing a Disk Page

- Relative times?
  - *seek time:* about 1 to 20msec
  - *rotational delay:* 0 to 10msec
  - *transfer time:* < 1msec per 4KB page

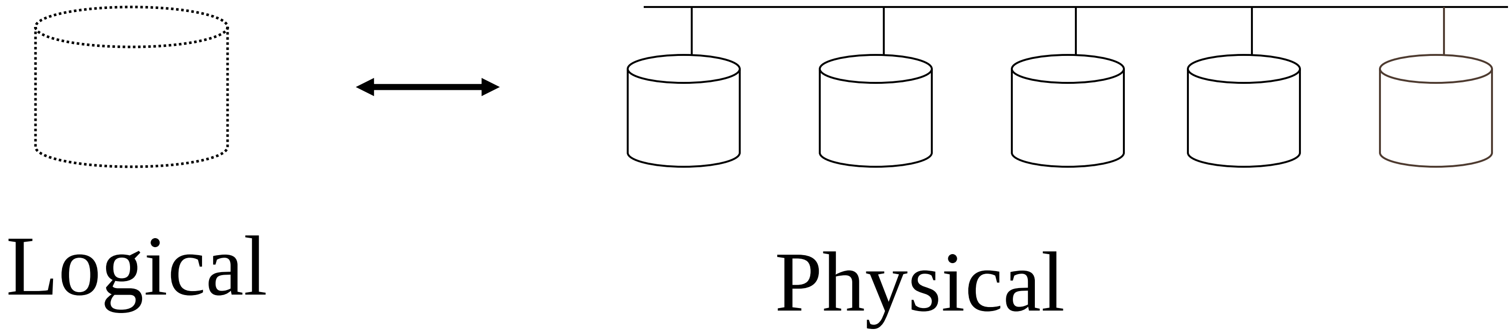


Rules of thumb…

Memory access <u>much</u> faster than disk I/O (~ 1000x)

"Sequential" I/O faster than "random" I/O (~ 10x)

Disk Arrays: RAID

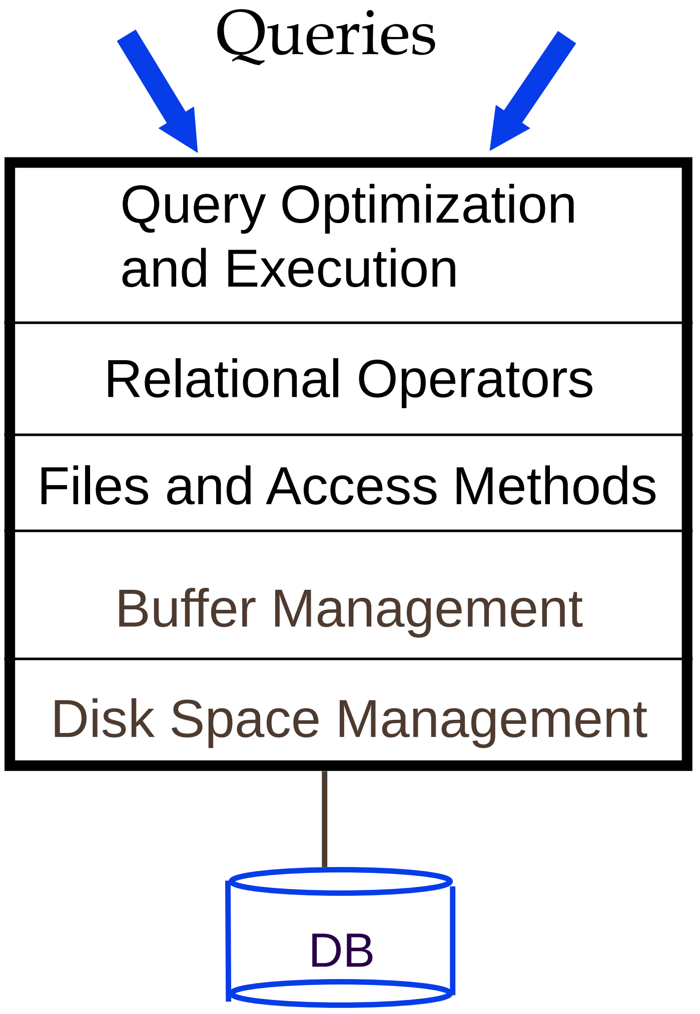


- Benefits:
  - Higher throughput (via data "striping")
  - Longer MTTF (via redundancy)

Disk Space Management

- Lowest layer of DBMS software manages space on disk
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Best if requested pages are stored sequentially on disk! Higher levels don't need to know if/how this is done, nor how free space is managed.

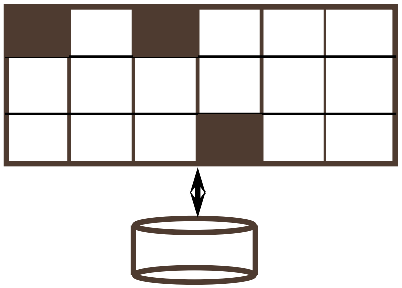
DBMS Layers



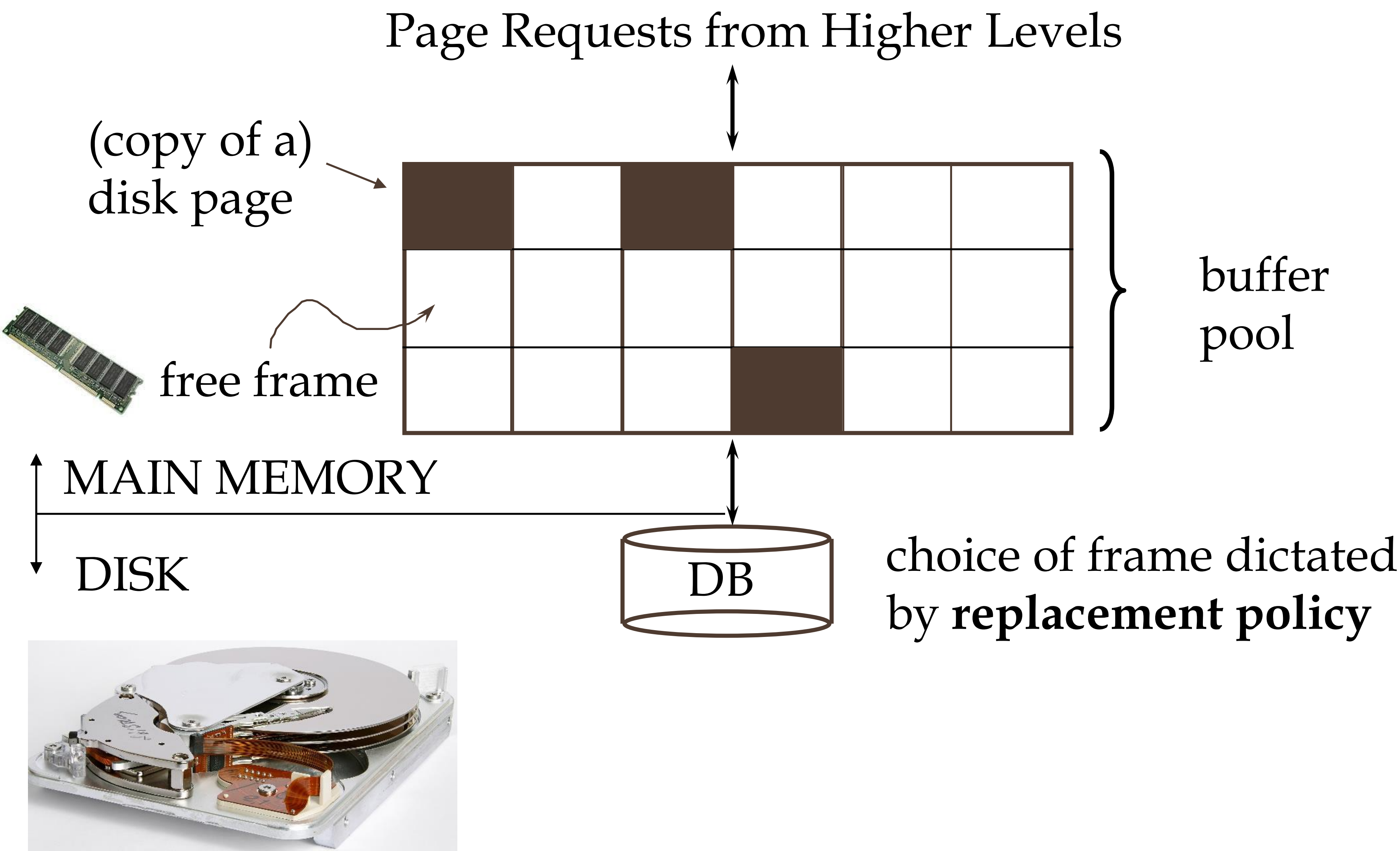
Queries
Query Optimization
and Execution
Relational Operators
Files and Access Methods
Buffer Management
Disk Space Management
DB

Buffer Management in a DBMS

- Data must be in RAM for DBMS to operate on it!
- Buffer Mgr hides the fact that not all data is in RAM



Buffer Management in a DBMS



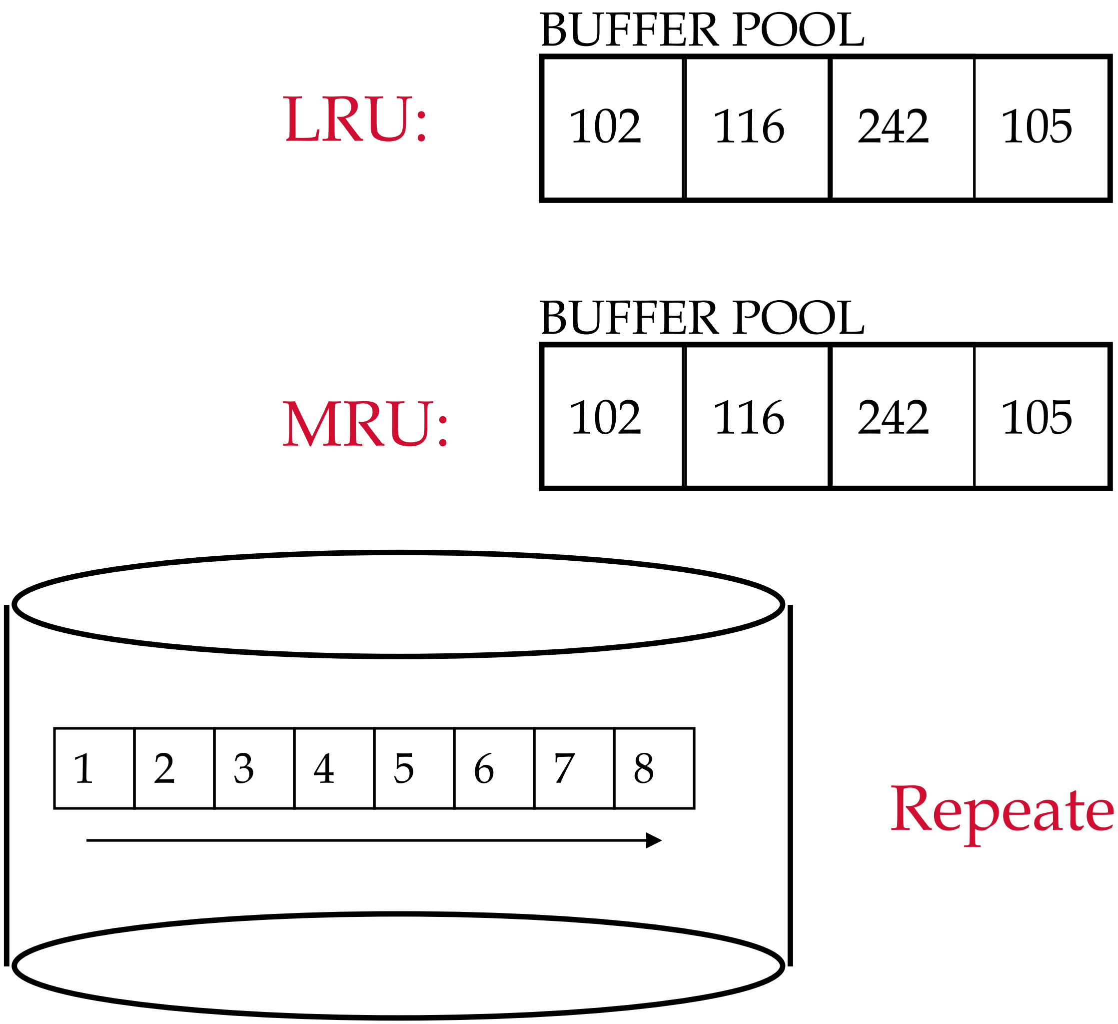
Buffer Replacement Policy

- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), MRU, Clock, etc.
- Policy -> big impact on # of I/O ’ s; depends on the *access pattern*.

LRU Replacement Policy

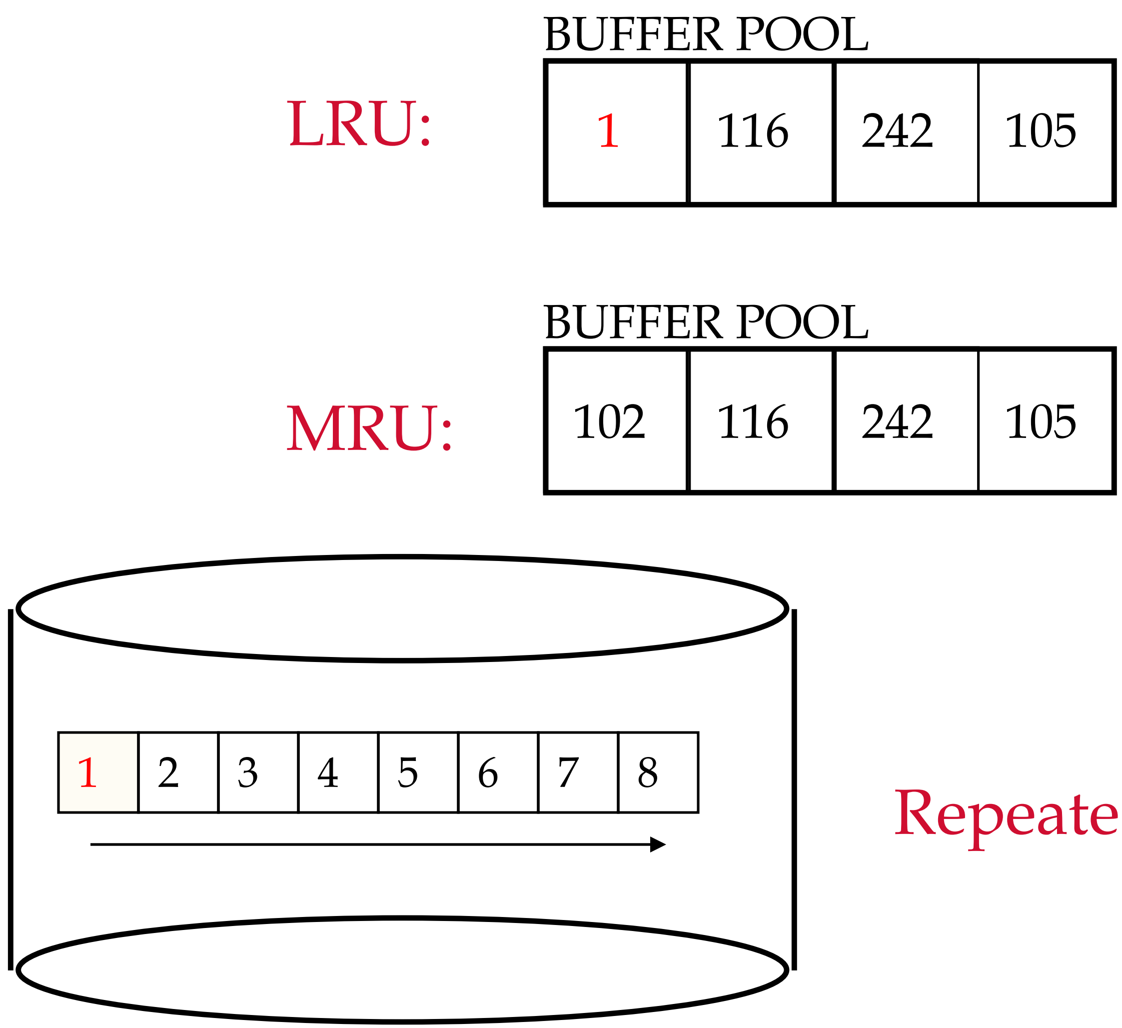
- *Least Recently Used (LRU)*
  - replace the frame which has the oldest (earliest) time
  - very common policy: intuitive and simple
- Problems?
- *# buffer frames < # pages in file* means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

Sequential Flooding – Illustration



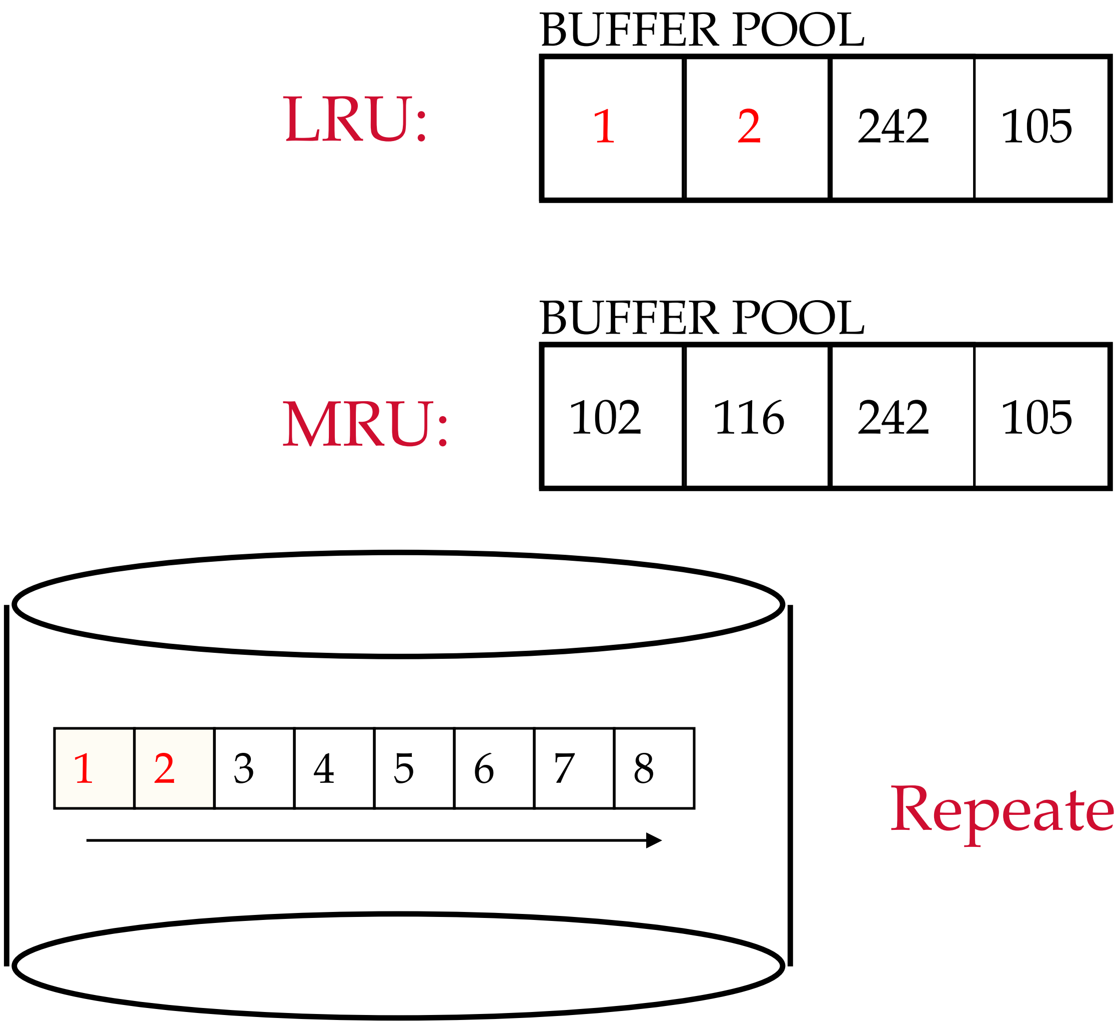
Repeated scan of file …

Sequential Flooding – Illustration



Repeated scan of file …

Sequential Flooding – Illustration



Repeated scan of file …

Sequential Flooding – Illustration

BUFFER POOL

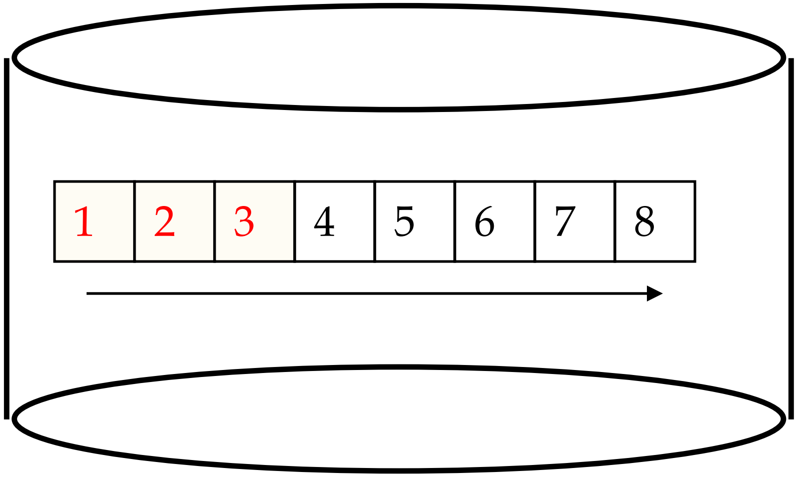
LRU:

1	2	3	105

BUFFER POOL

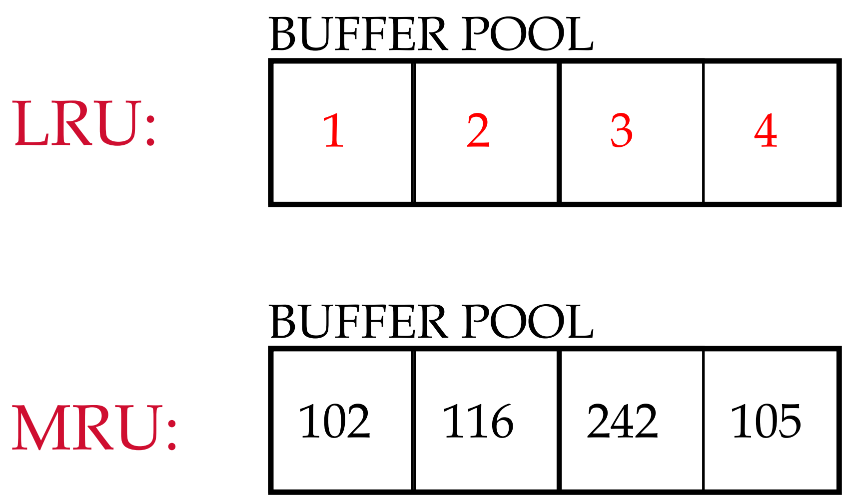
MRU:

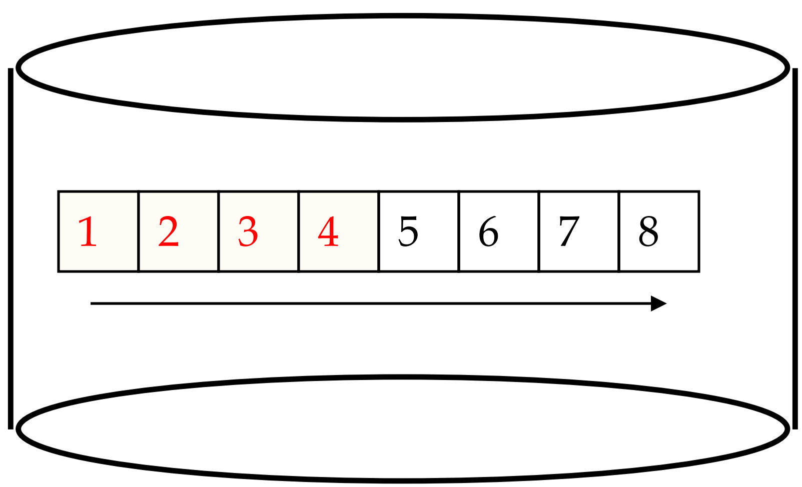
102	116	242	105



Repeated scan of file …

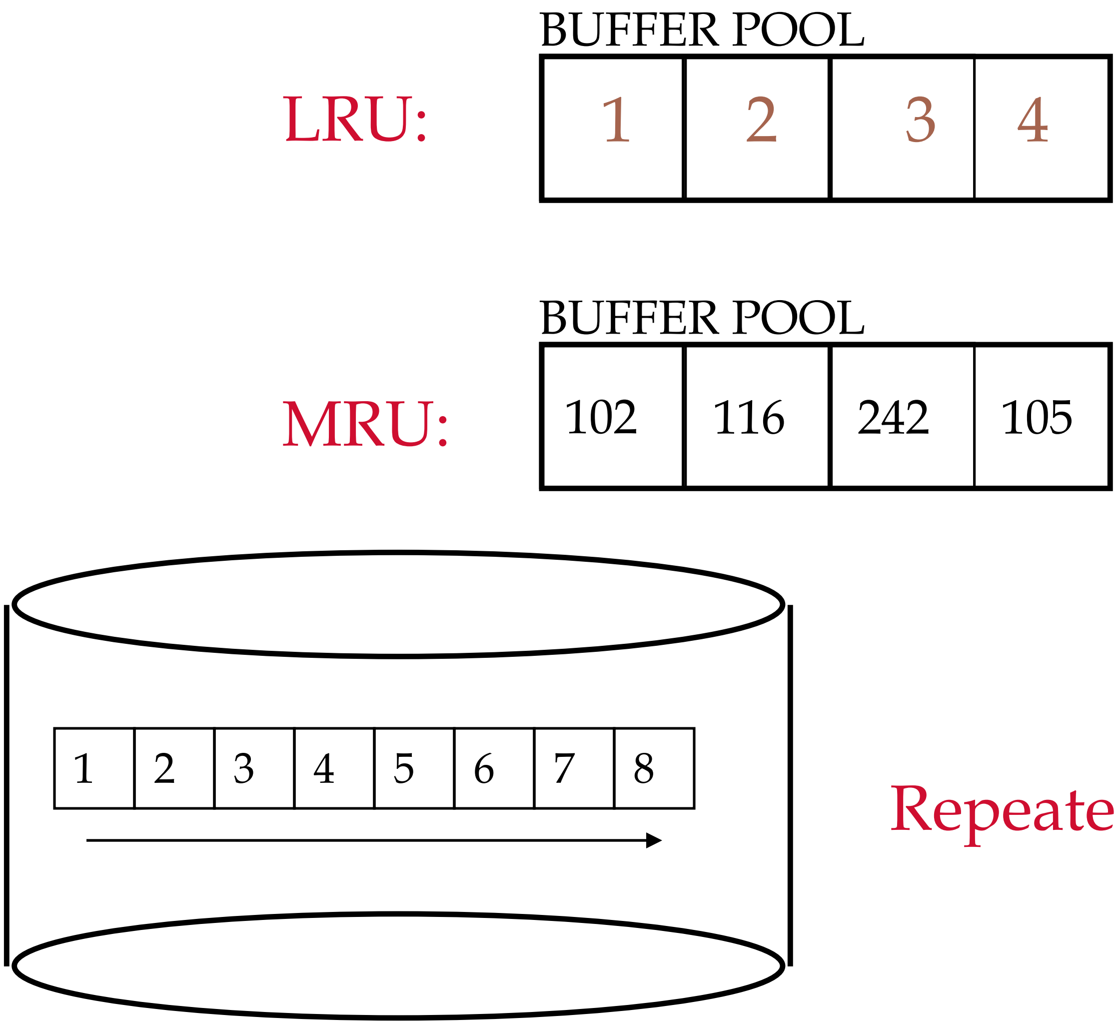
Sequential Flooding – Illustration





Repeated scan of file …

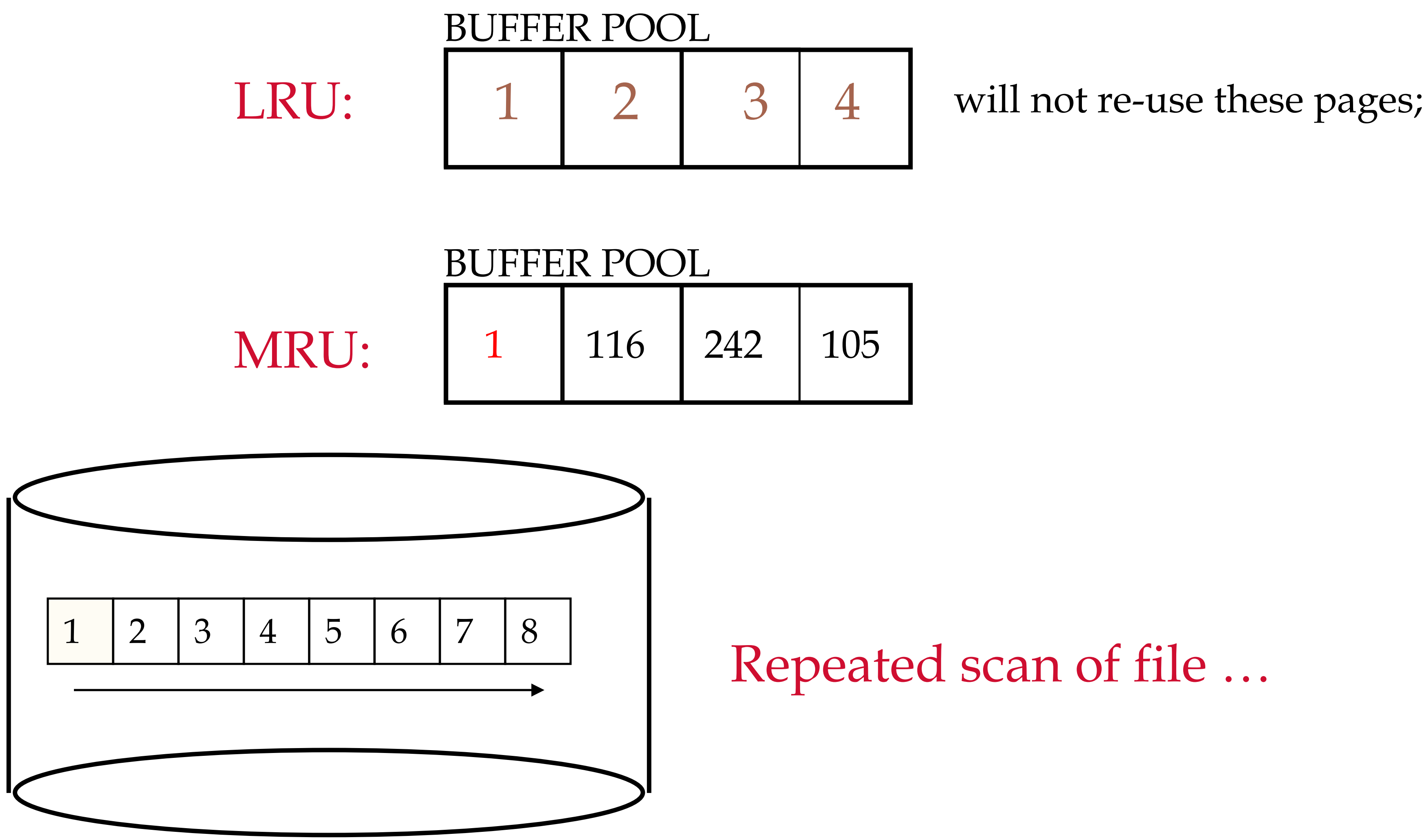
Sequential Flooding – Illustration



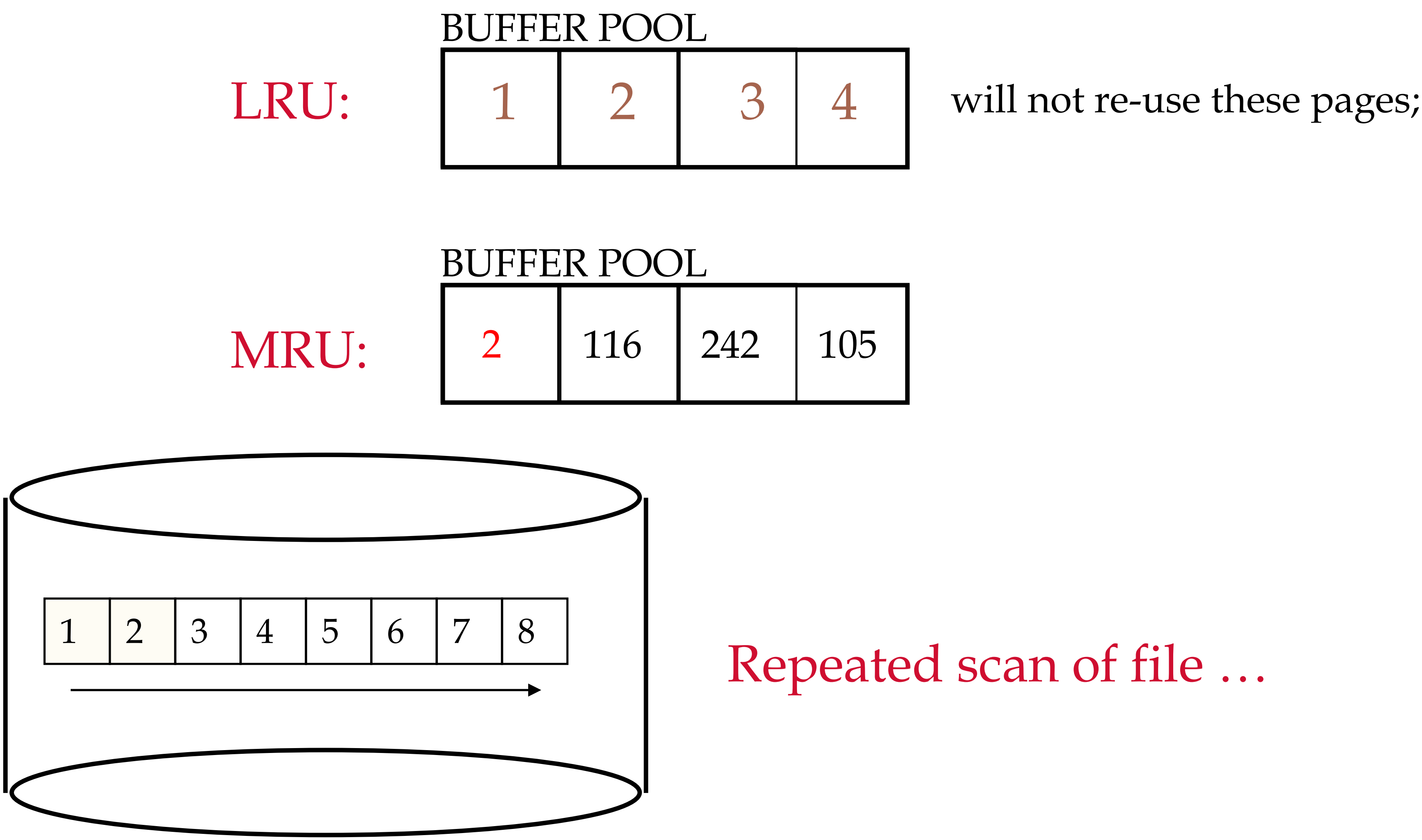
will not re-use these pages;

Repeated scan of file …

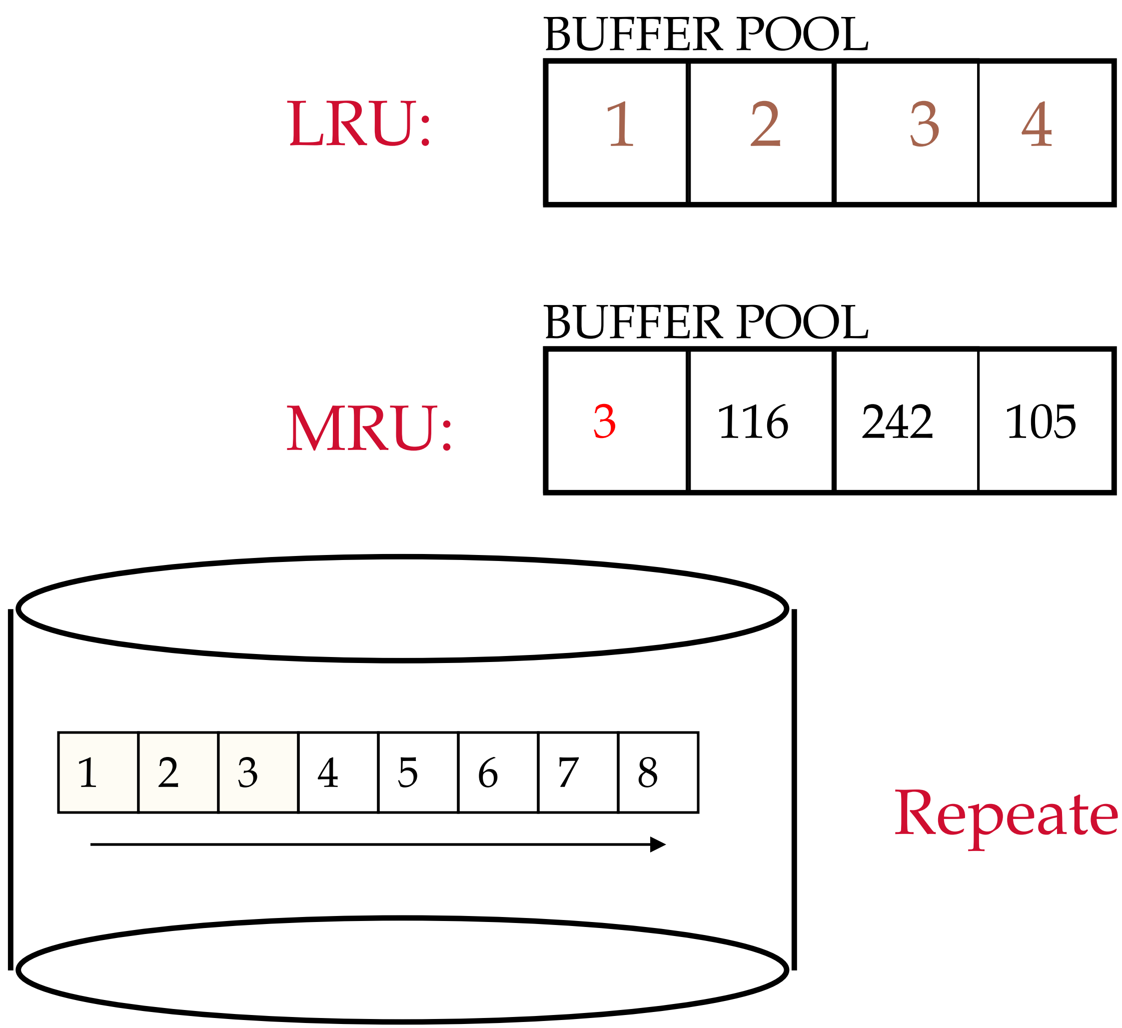
Sequential Flooding – Illustration



Sequential Flooding – Illustration



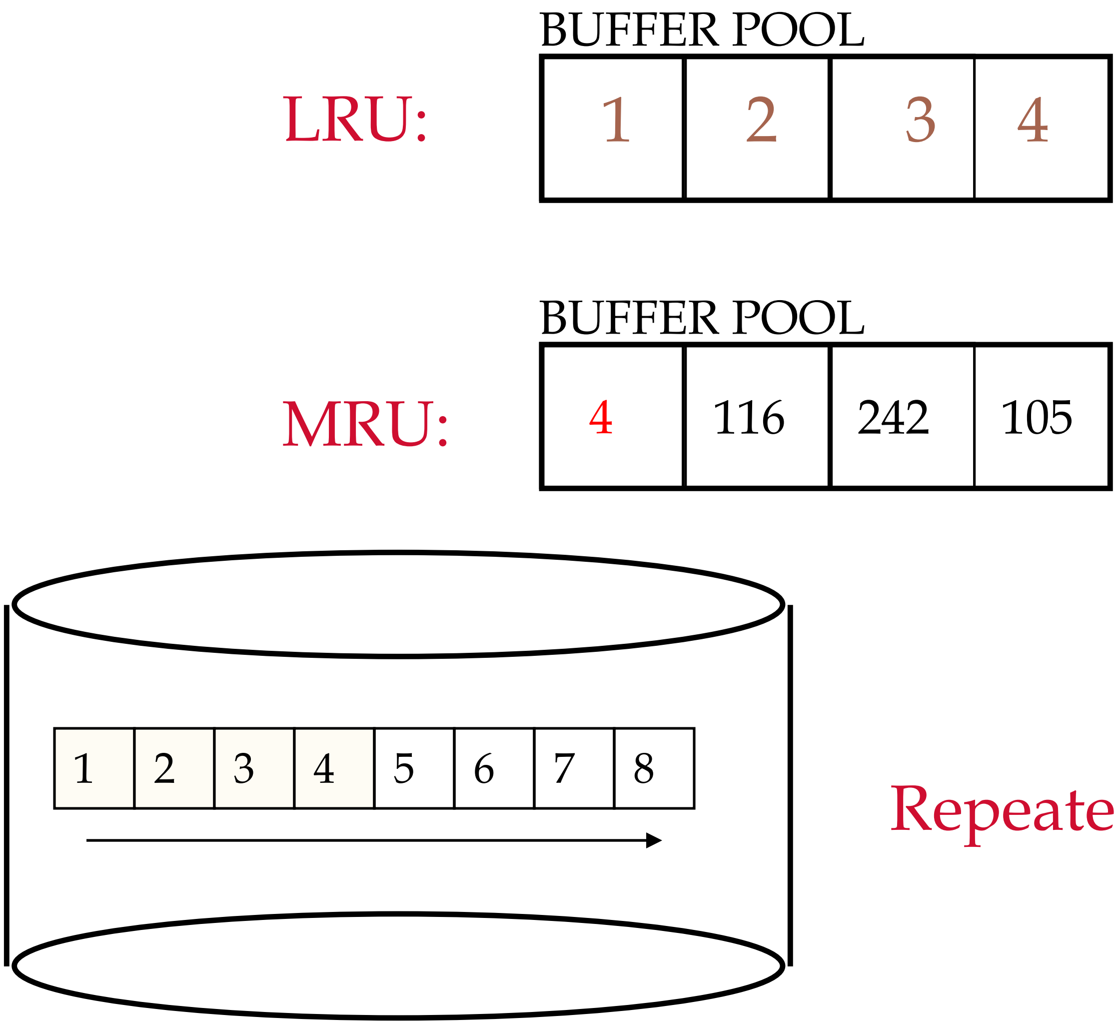
Sequential Flooding – Illustration



will not re-use these pages;

Repeated scan of file …

Sequential Flooding – Illustration



will not re-use these pages;

Repeated scan of file …

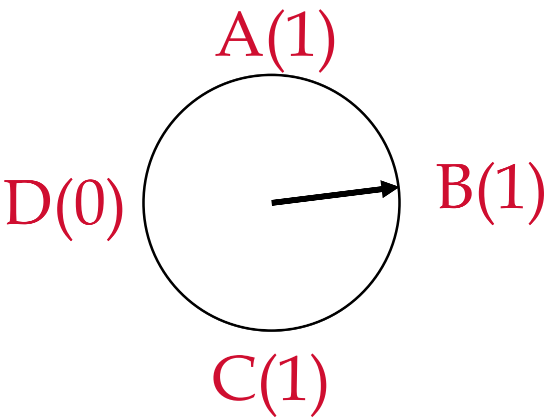
Other policies?

- LRU is often good - but needs timestamps and sorting on them
- something easier to maintain?

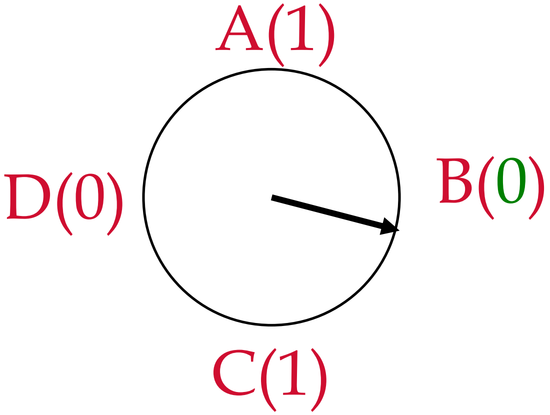
"Clock" Replacement Policy

Main ideas:

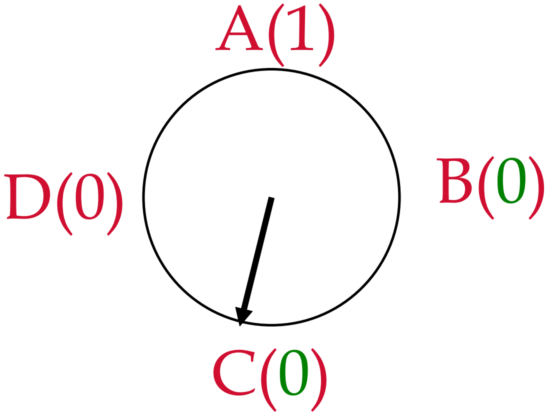
- Approximation of LRU.
- Instead of maintaining & sorting time-stamps, find a 'reasonably old' frame to evict.
- How? by round-robin, and marking each frame - frames are evicted the second time they are visited.



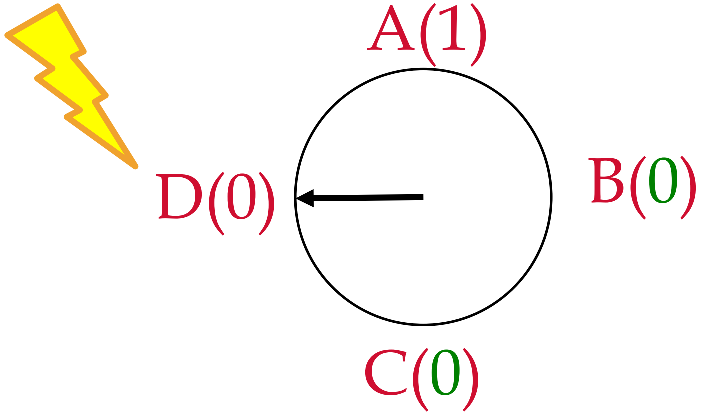
"Clock" Replacement Policy



“Clock” Replacement Policy



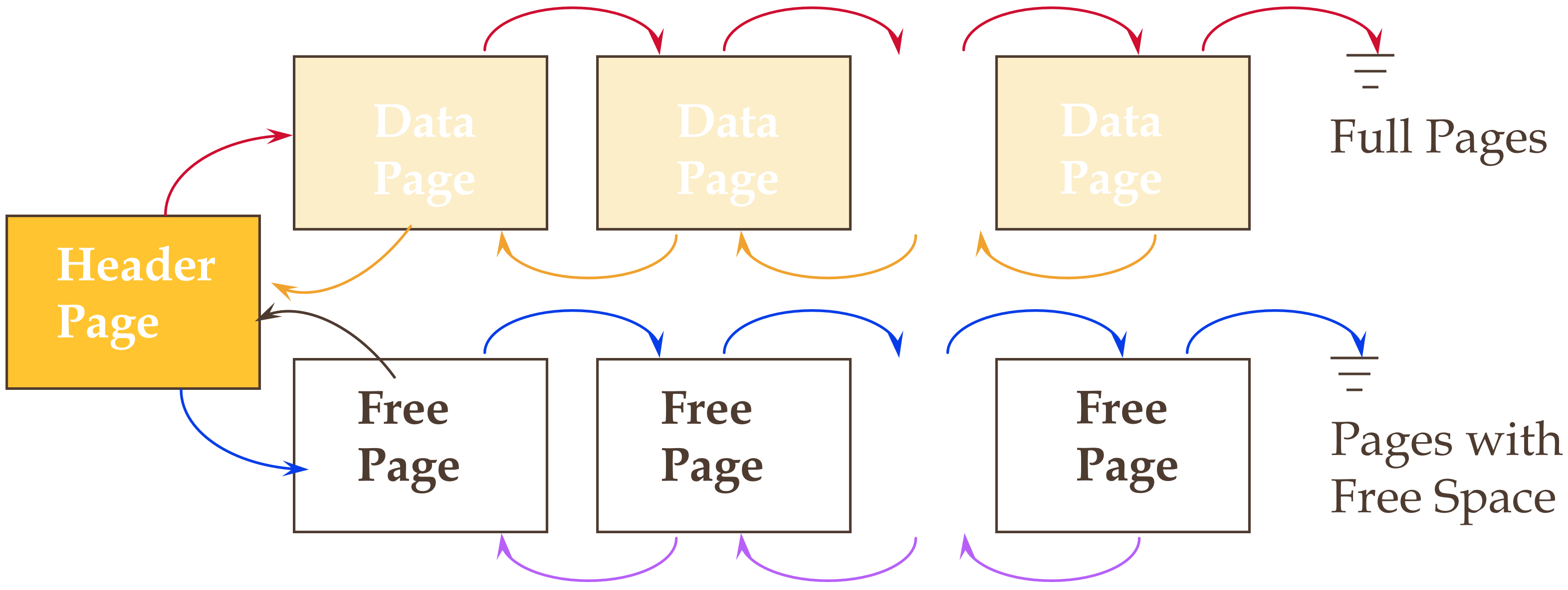
“Clock” Replacement Policy



Files

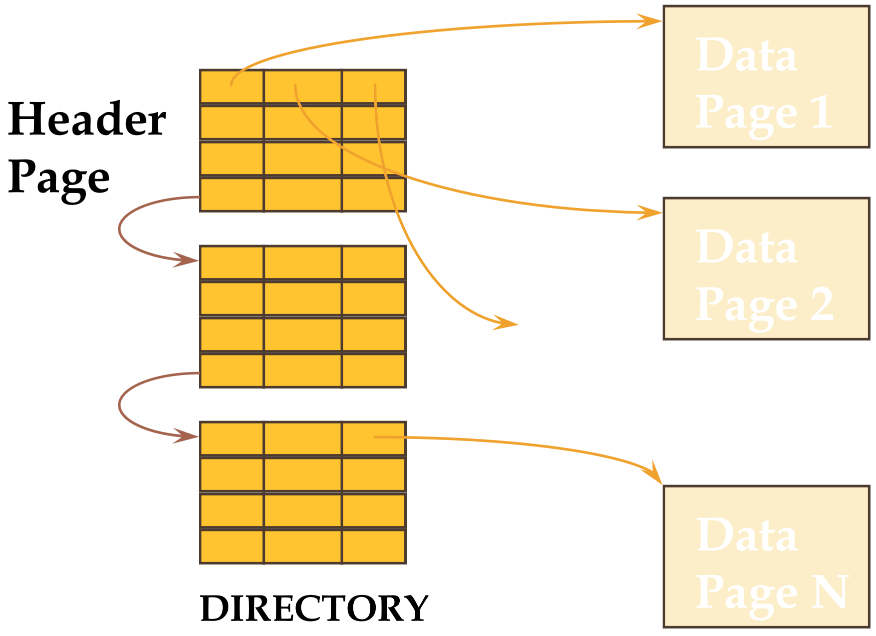
- FILE: A collection of pages, each containing a collection of records.
- Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

Heap File Using Lists



- Any problems?

Heap File Using a Page Directory



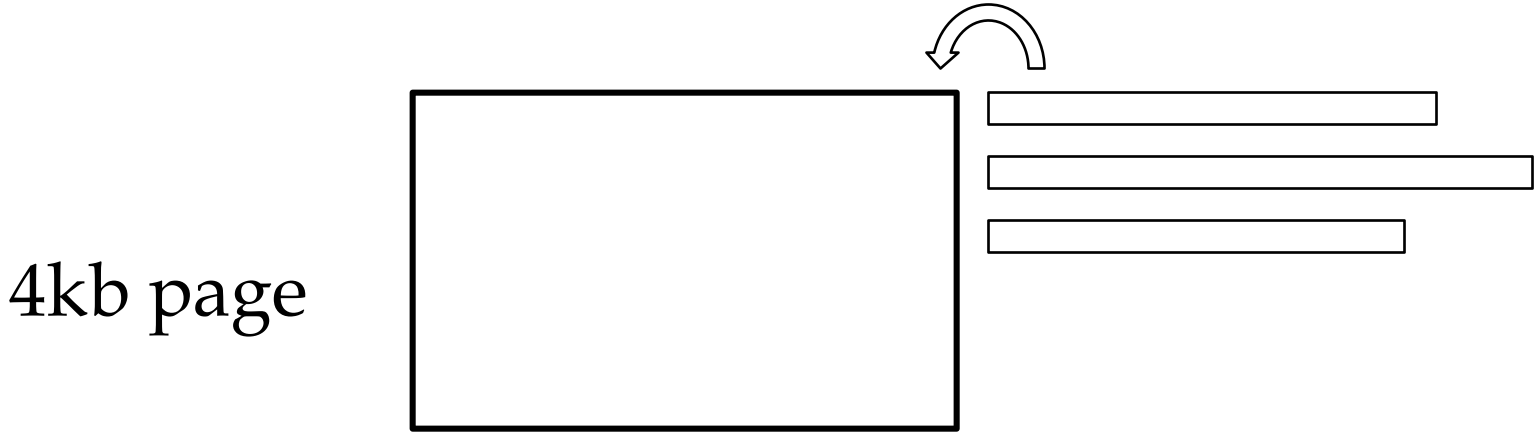
Page Formats

- fixed length records
- variable length records

Problem definition

Q: How would you store records on a page/file, such that

1. you can point to them
2. you can insert/delete records with few disk accesses



Page Formats

Important concept: *rid* == record id

Q0: why do we need it?

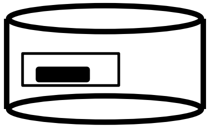
A0: eg., for indexing

Q1: How to mark the location of a record?

A1: <u>rid = record id = page-id & slot-id</u>

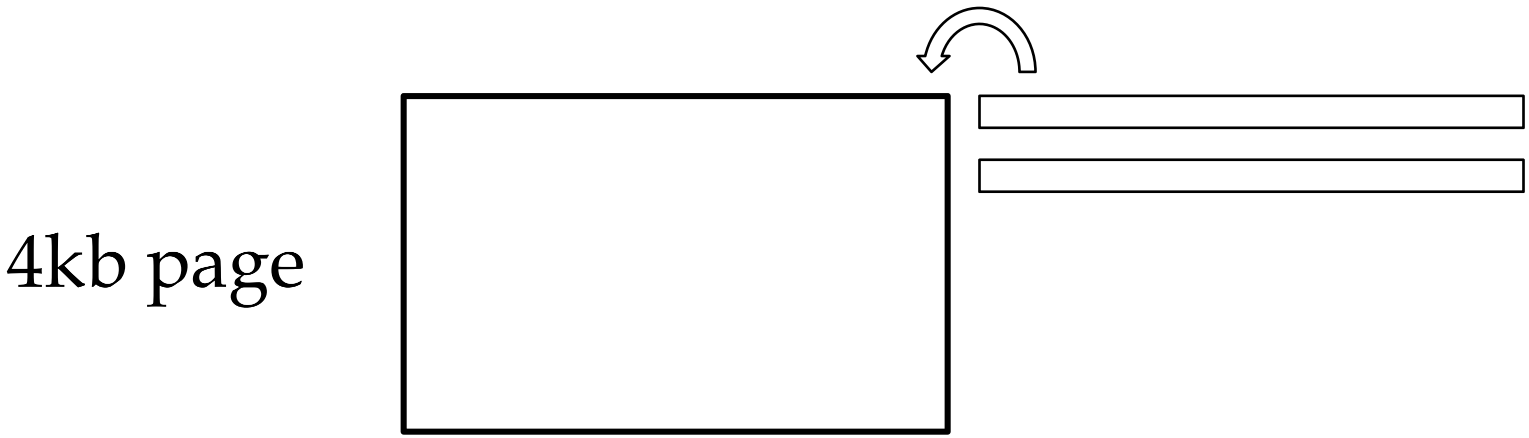
Q2: Why not its byte offset in the file?

A2: too much re-organization on ins/del.



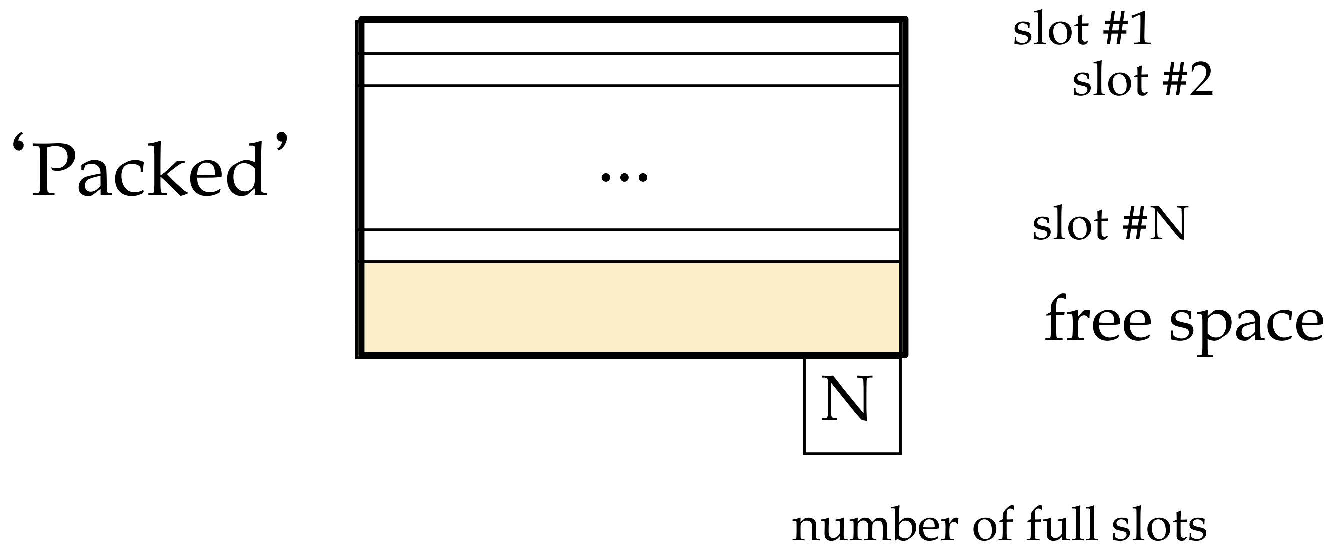
Fixed length records

- Q: How would you store them on a page/file?



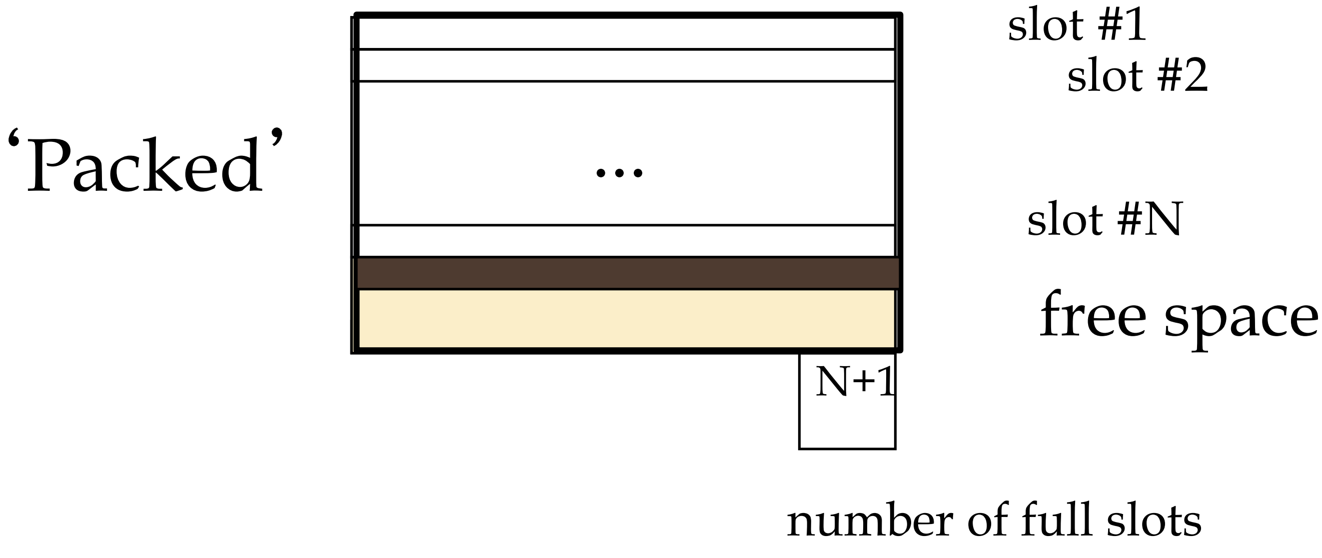
Fixed length records

- OK – how about insertion?



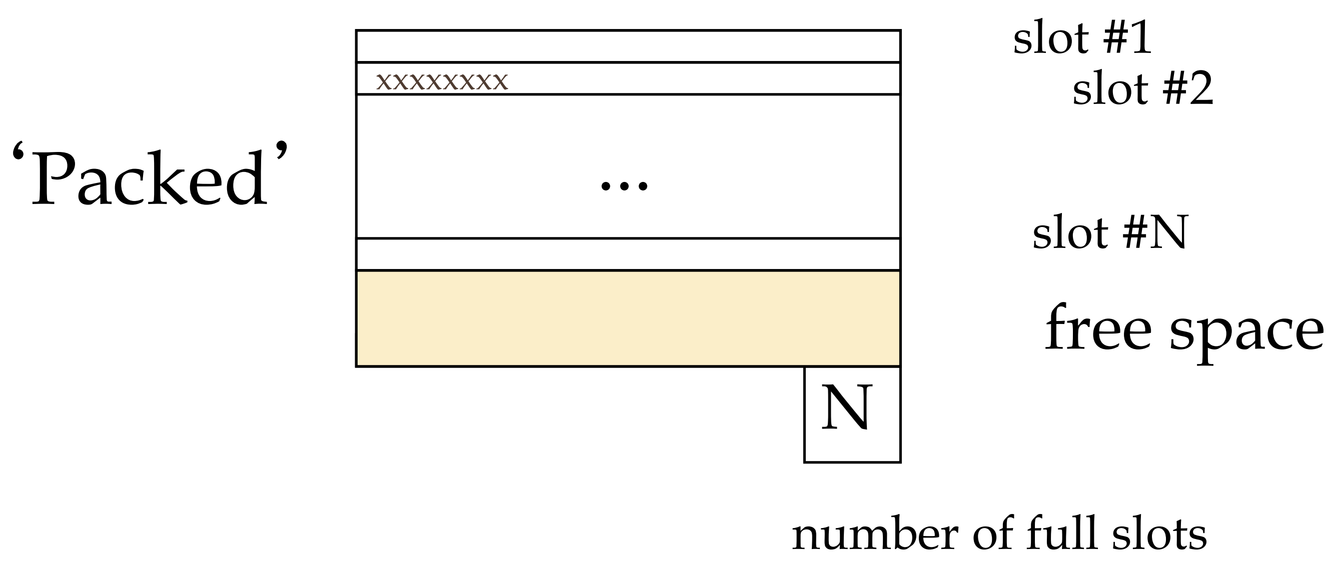
Fixed length records

- OK – how about insertion?



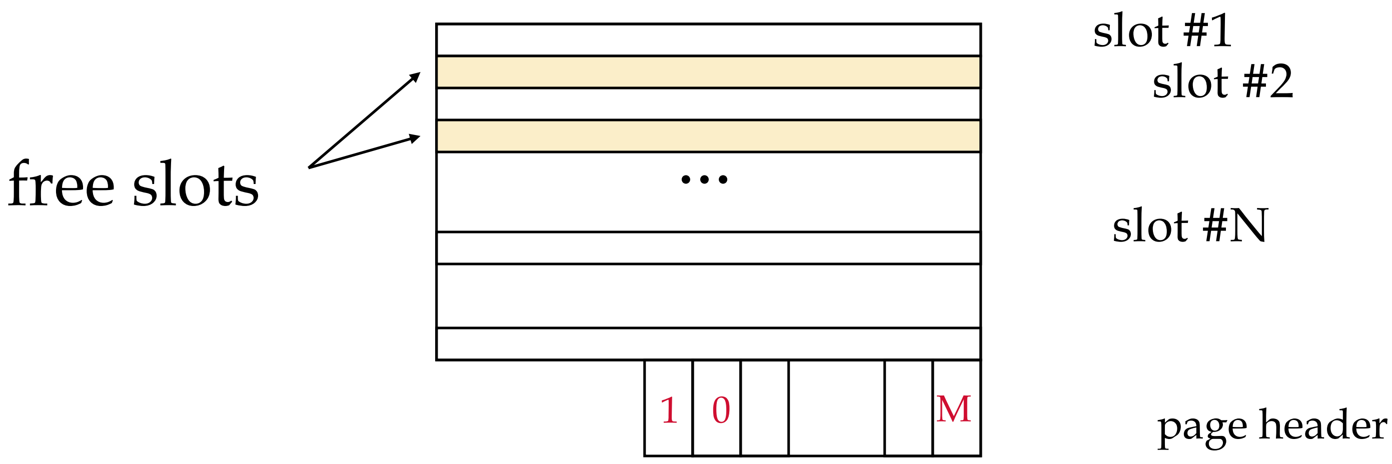
Fixed length records

- How about deletion?



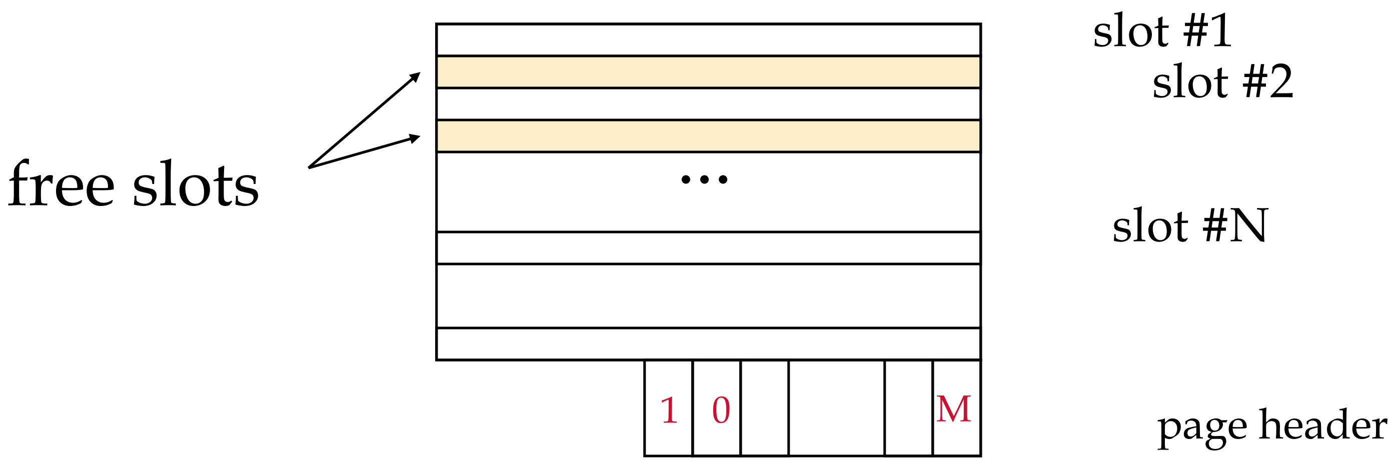
Fixed length records

- Q: How would you store them on a page/file?
- A2: Bitmaps



Fixed length records

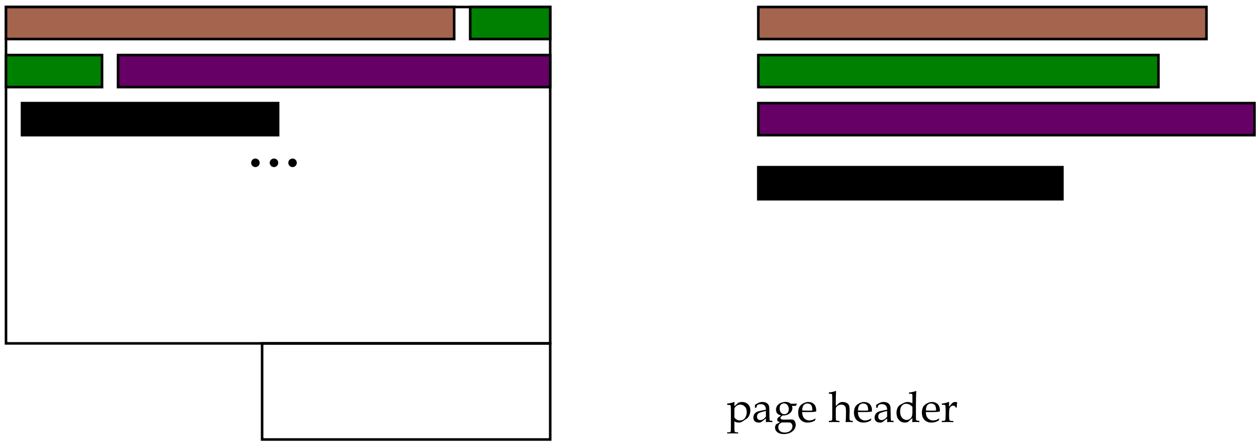
- Q: How would you store them on a page/file?
- A2: Bitmaps : ✓ insertions, ✓ deletions



Variable length records

- Q: How would you store them on a page/file?

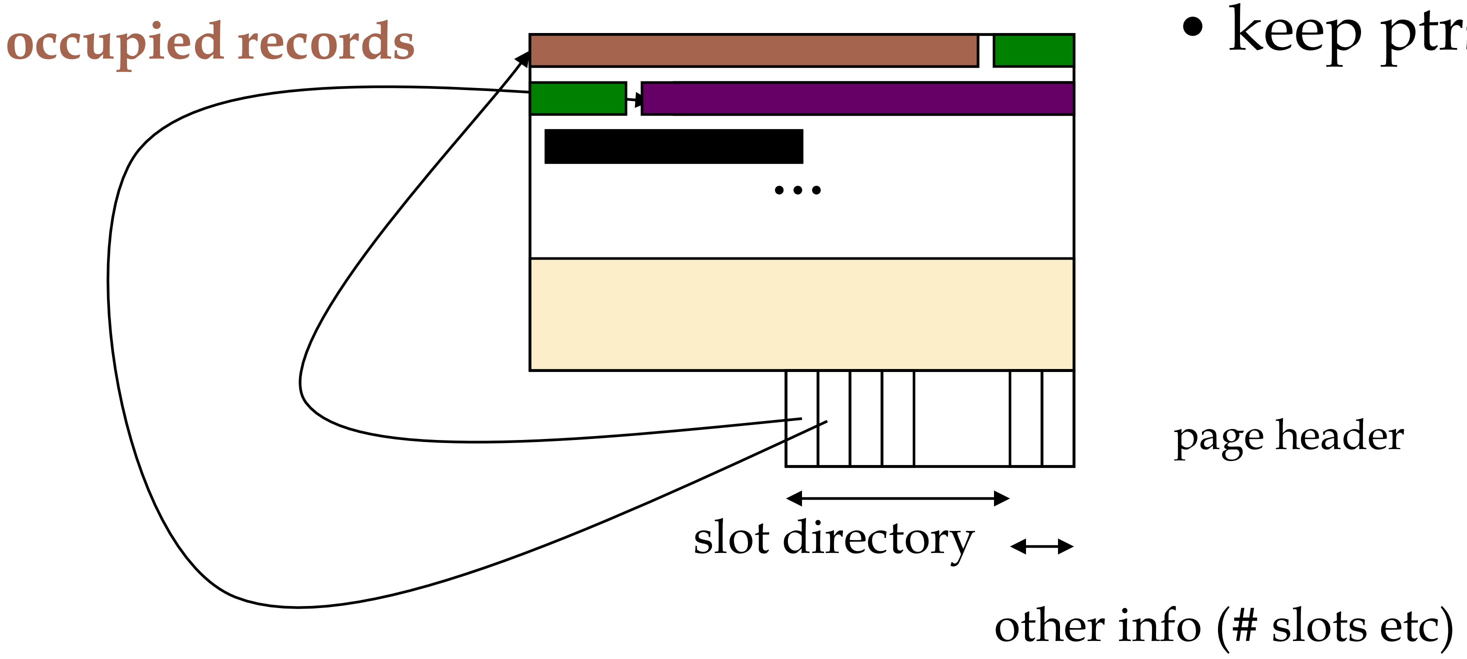
occupied records



page header

Variable length records

- Q: How would you store them on a page/file?



- pack them
- keep ptrs to them

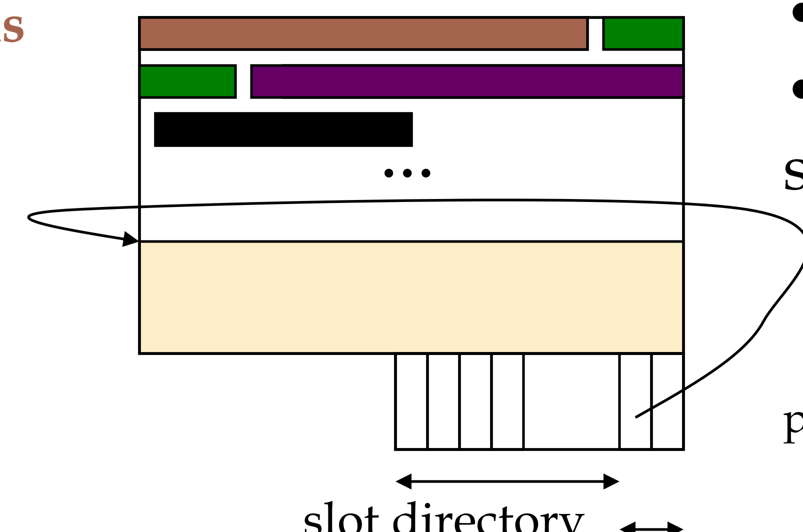
SLOTTED PAGE

Variable length records

- Q: How would you store them on a page/file?

- pack them
- keep ptrs to them
- mark start of free space

occupied records



page header

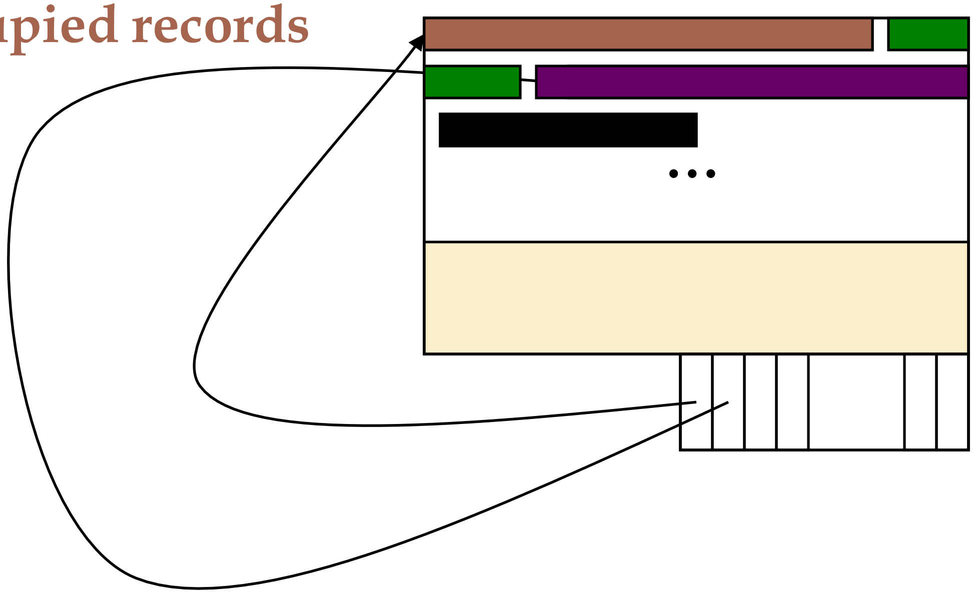
slot directory

other info (# slots etc)

Variable length records

- Q: How would you store them on a page/file?

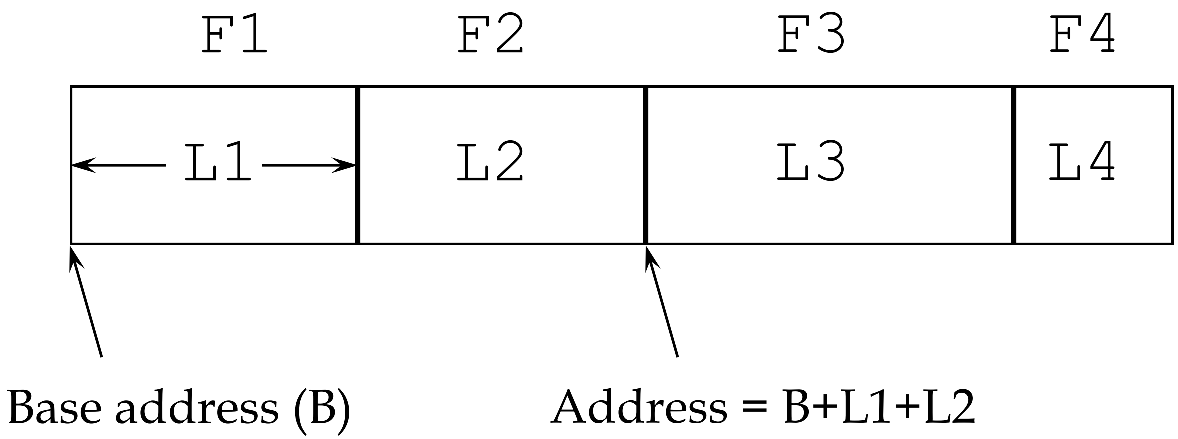
occupied records



- how many disk accesses to insert a record?
- to delete one?

page header

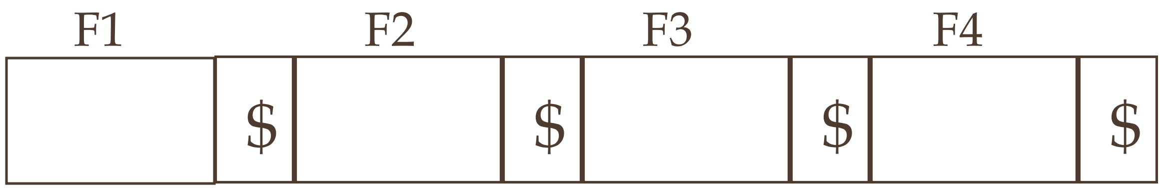
Record Formats: Fixed Length



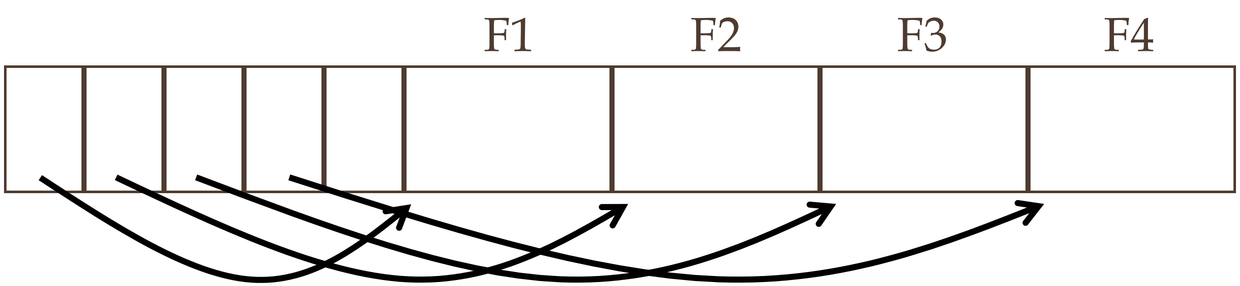
- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field done via arithmetic.

Variable Length records

- Two alternative formats (# fields is fixed):



Fields Delimited by Special Symbols

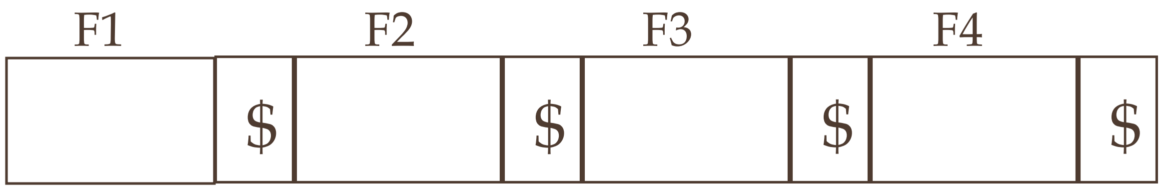


Array of Field Offsets

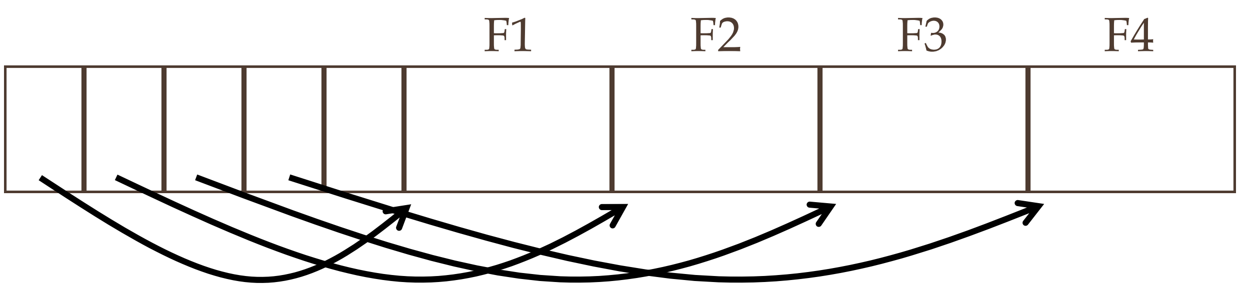
Pros and cons?

Variable Length records

- Two alternative formats (# fields is fixed):



Fields Delimited by Special Symbols

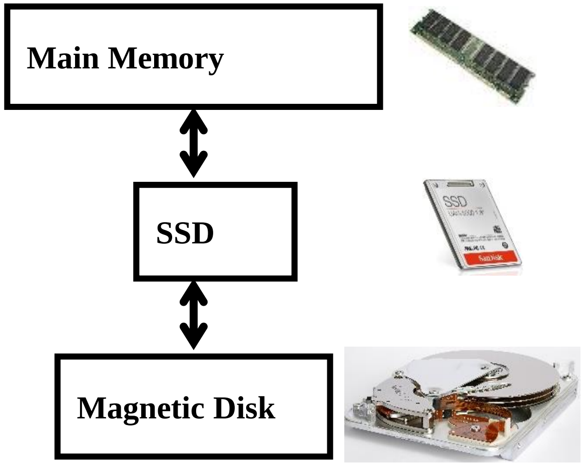


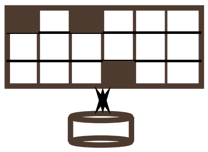
Array of Field Offsets

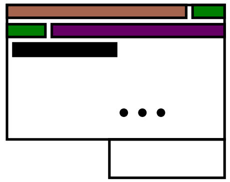
Offset approach: usually superior (direct access to i-th field)

Till now

- Memory hierarchy
- Disks: (>1000x slower) - thus
  - pack info in blocks
  - try to fetch nearby blocks (sequentially)
- Buffer management: very important
  - LRU, MRU, Clock, etc
- Record organization: Slotted page







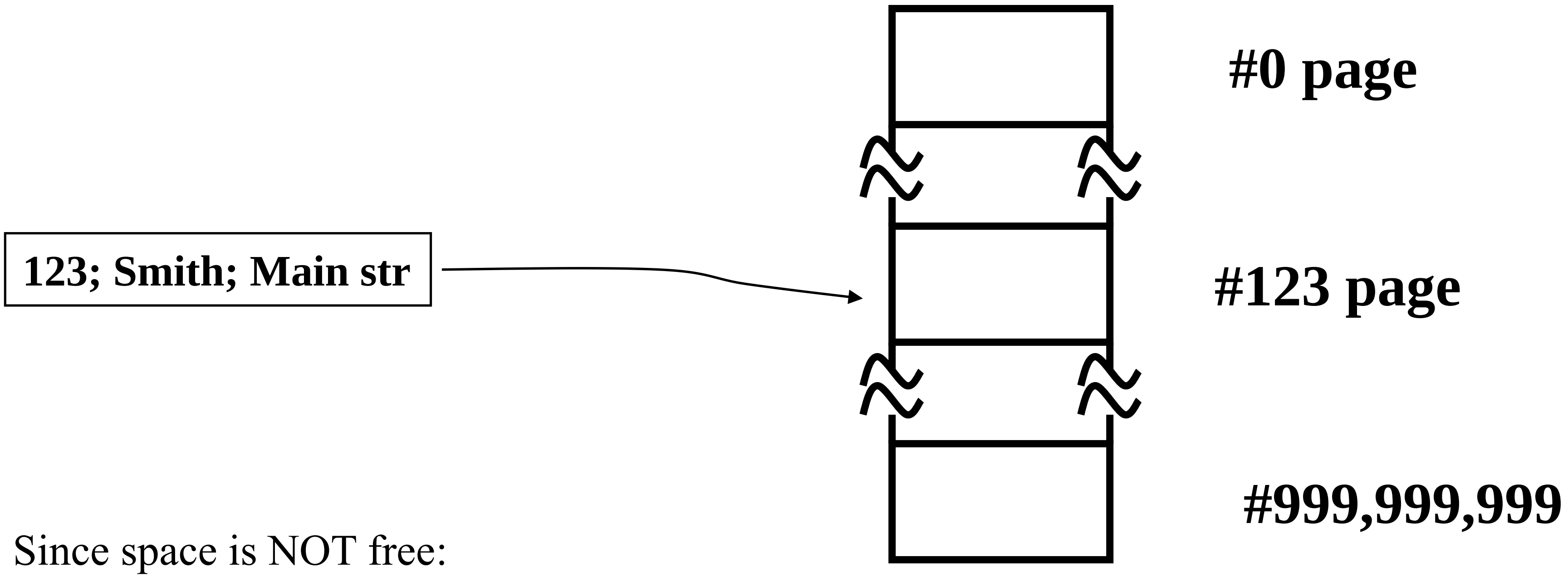
Hashing

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

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

Hashing

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

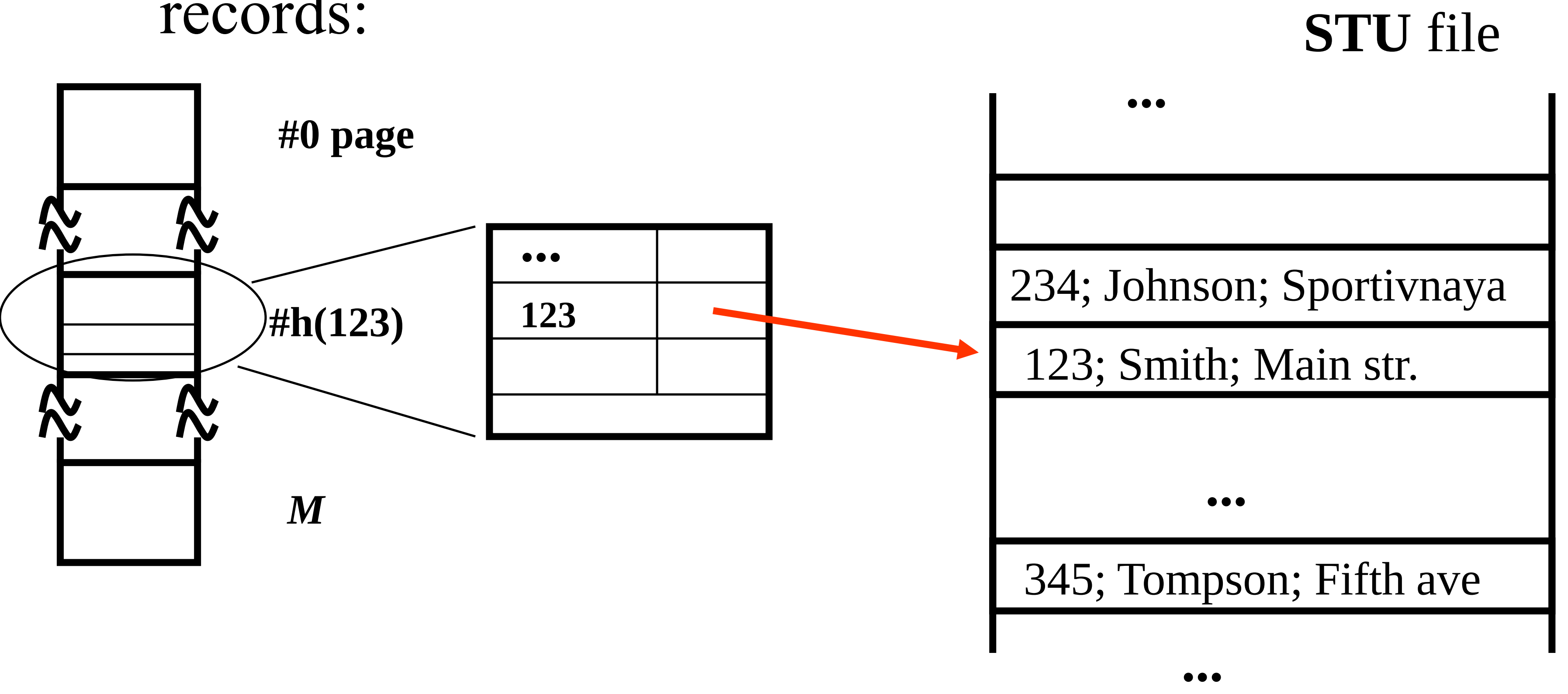


Since space is NOT free:

Hashing

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

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



Design decisions - functions

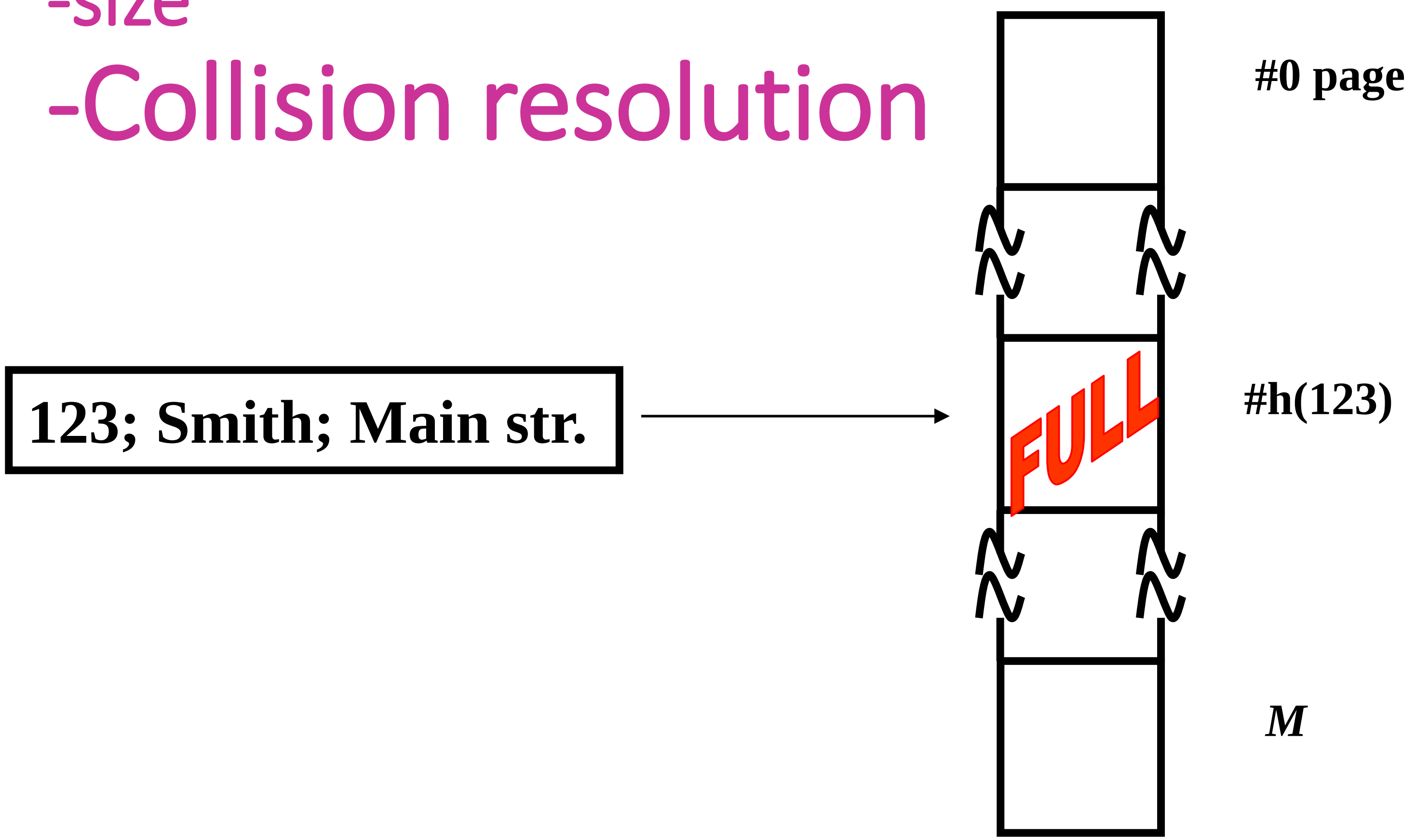
- Goal: uniform spread of keys over hash buckets
- Popular choices:
  - Division hashing
    - $h(x) = (a*x+b) \mod M$
  - Multiplication hashing
    - $h(x) = [\, \text{fractional-part-of}\, (x * \varphi)\,] * M$

Hash Design decisions

-functions

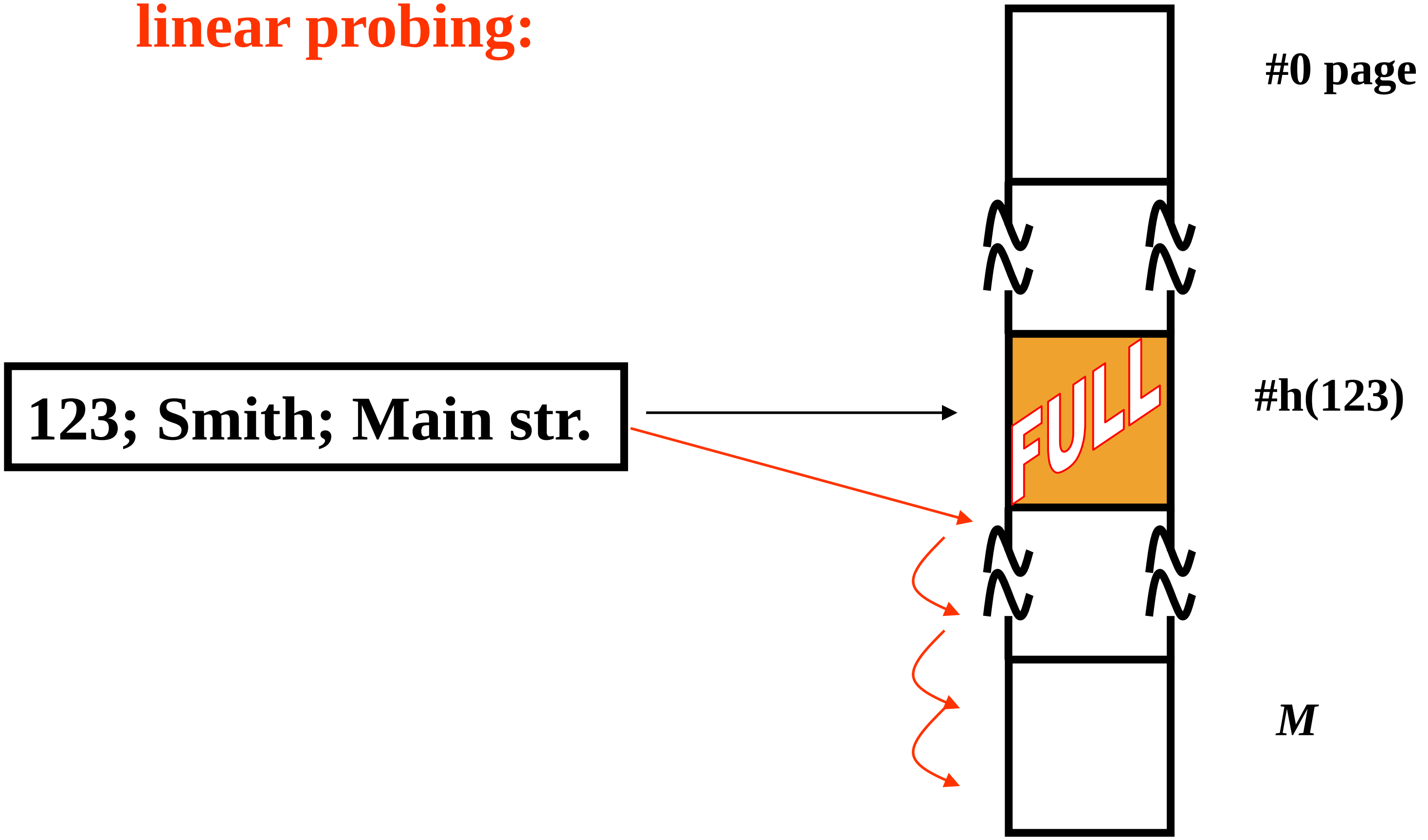
-size

-Collision resolution

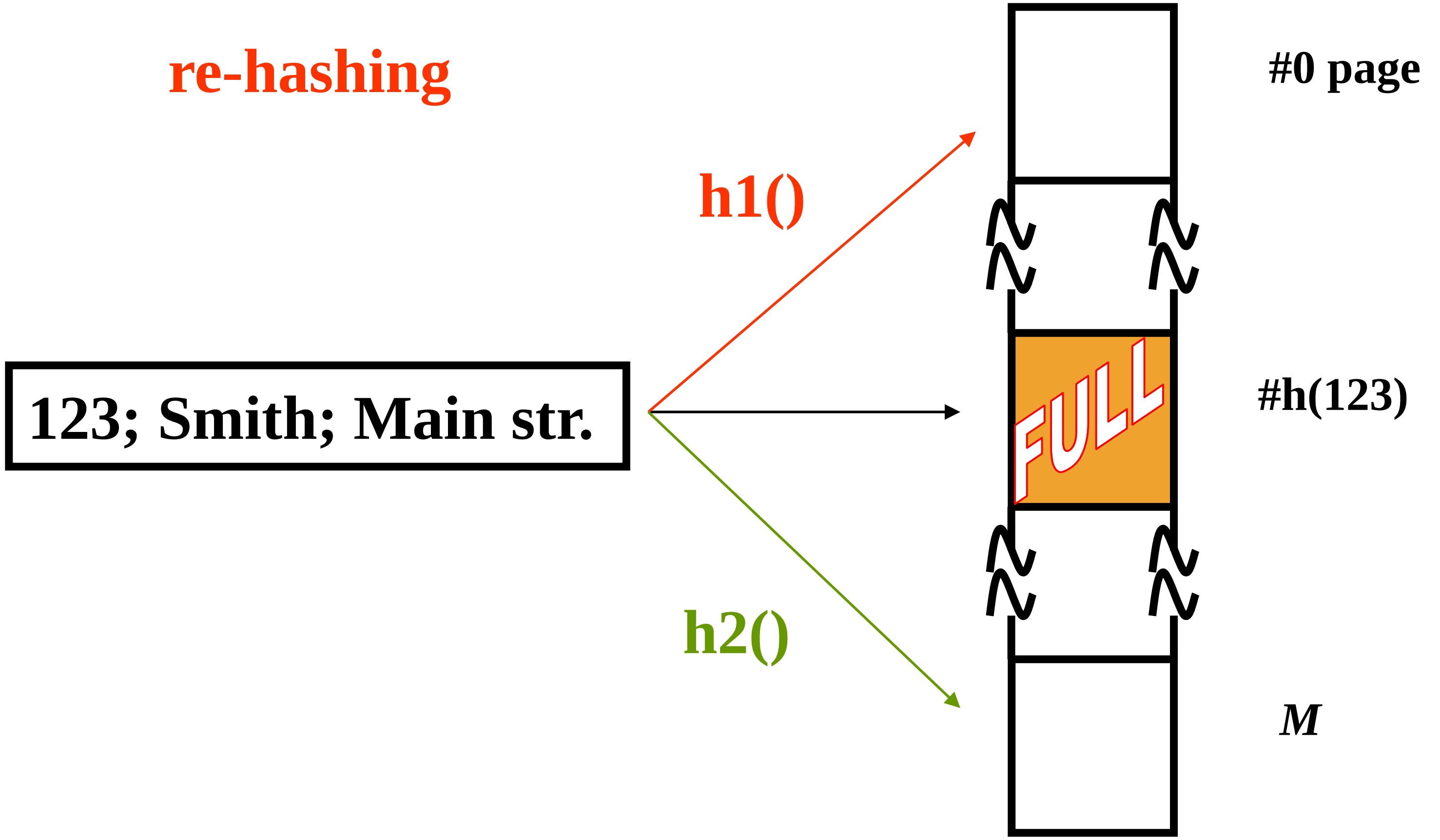


Collision resolution

linear probing:

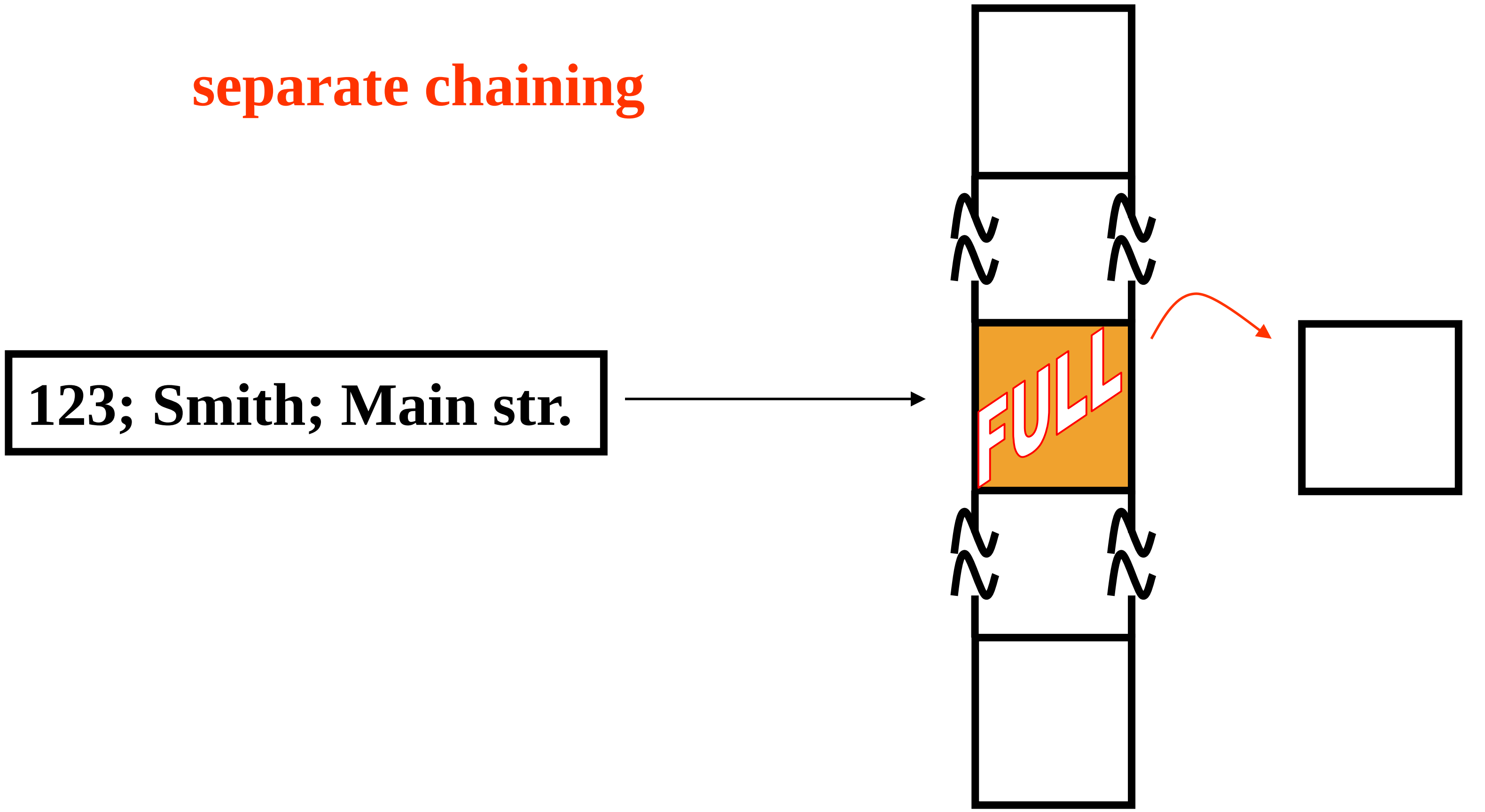


Collision resolution



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

Problem with static hashing

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

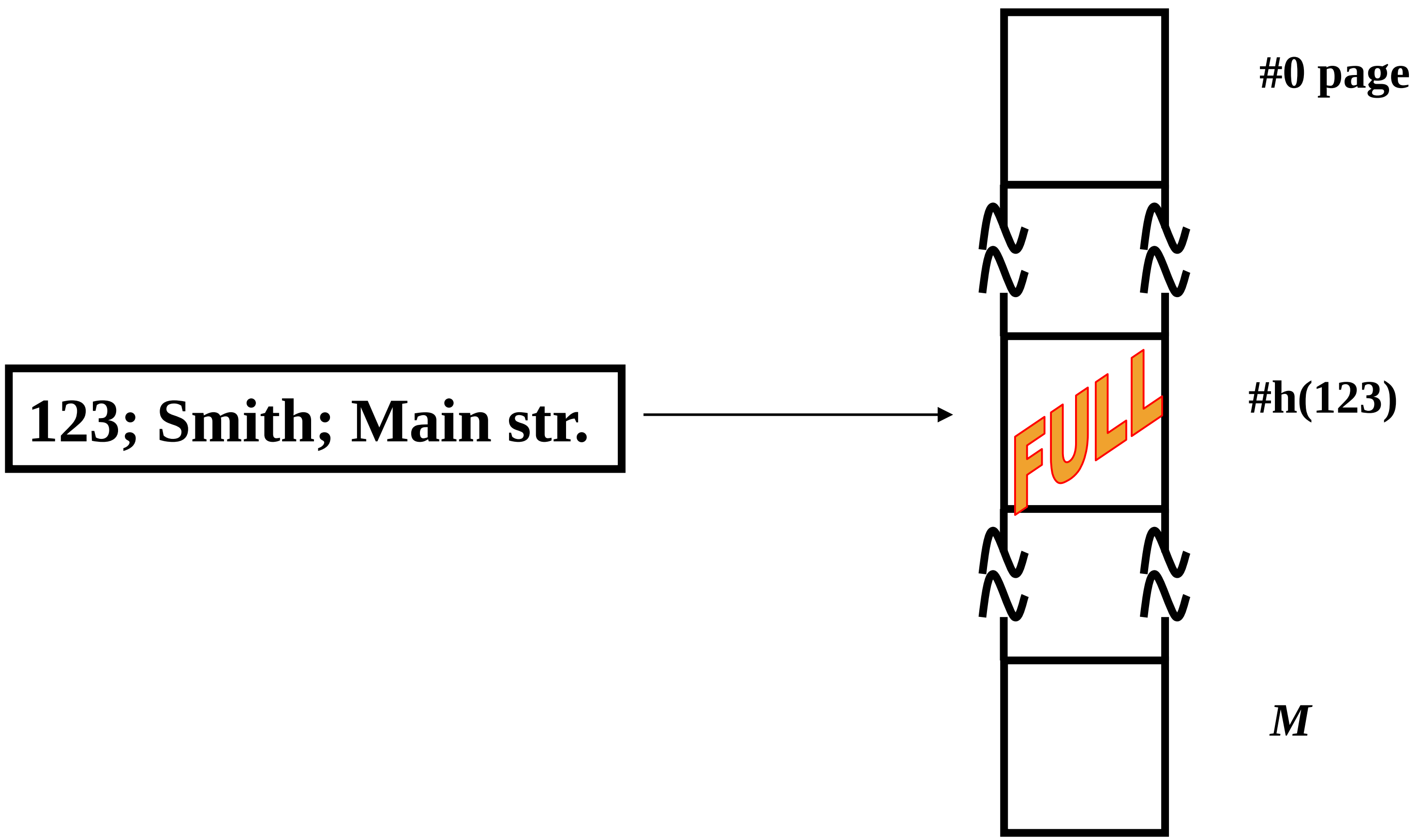
Solution: Dynamic/extendible hashing

- 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

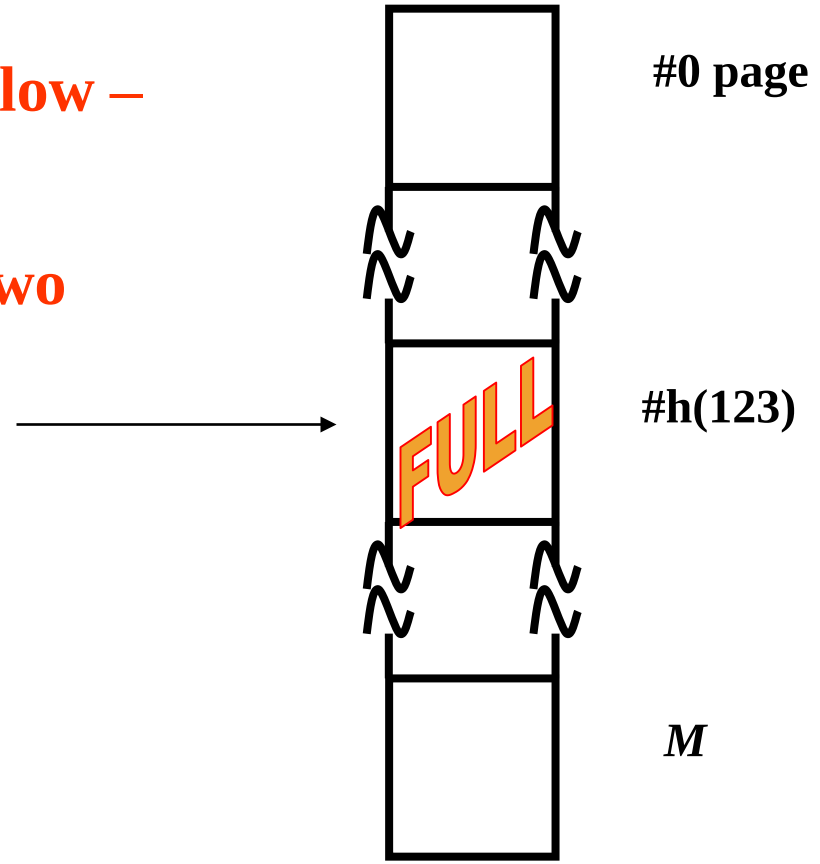


Extendible hashing

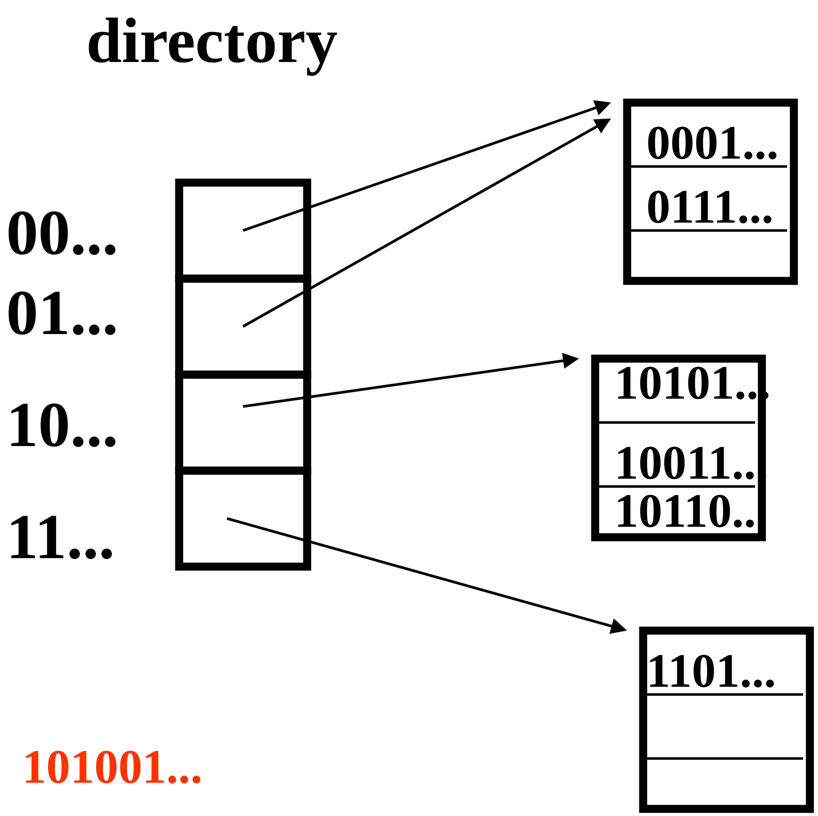
solution: don' t overflow – instead:

<u>SPLIT</u> the bucket in two

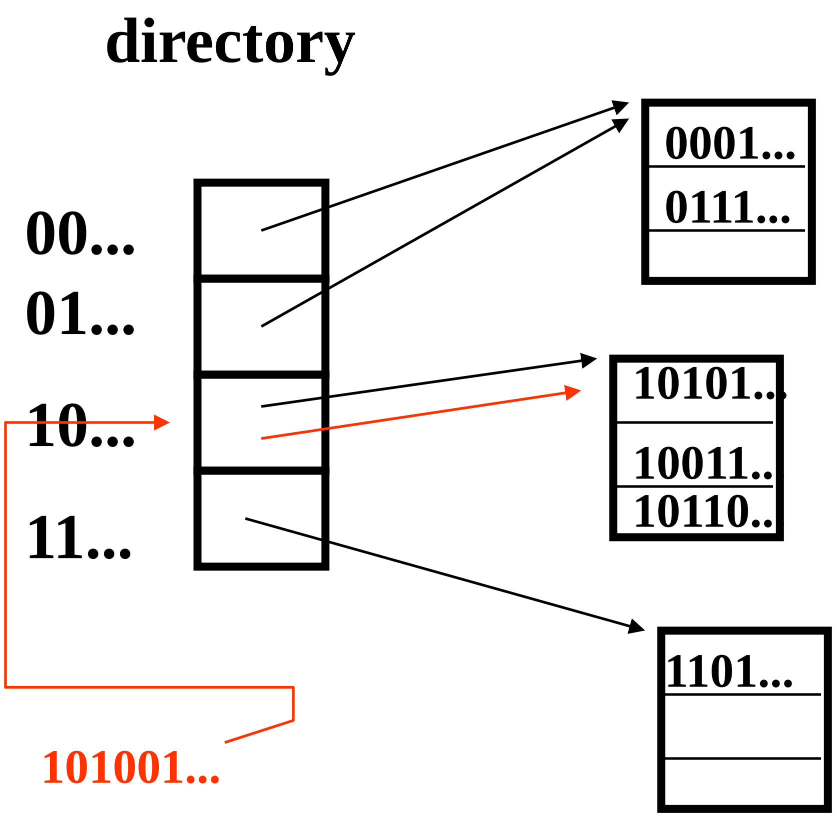
123; Smith; Main str.



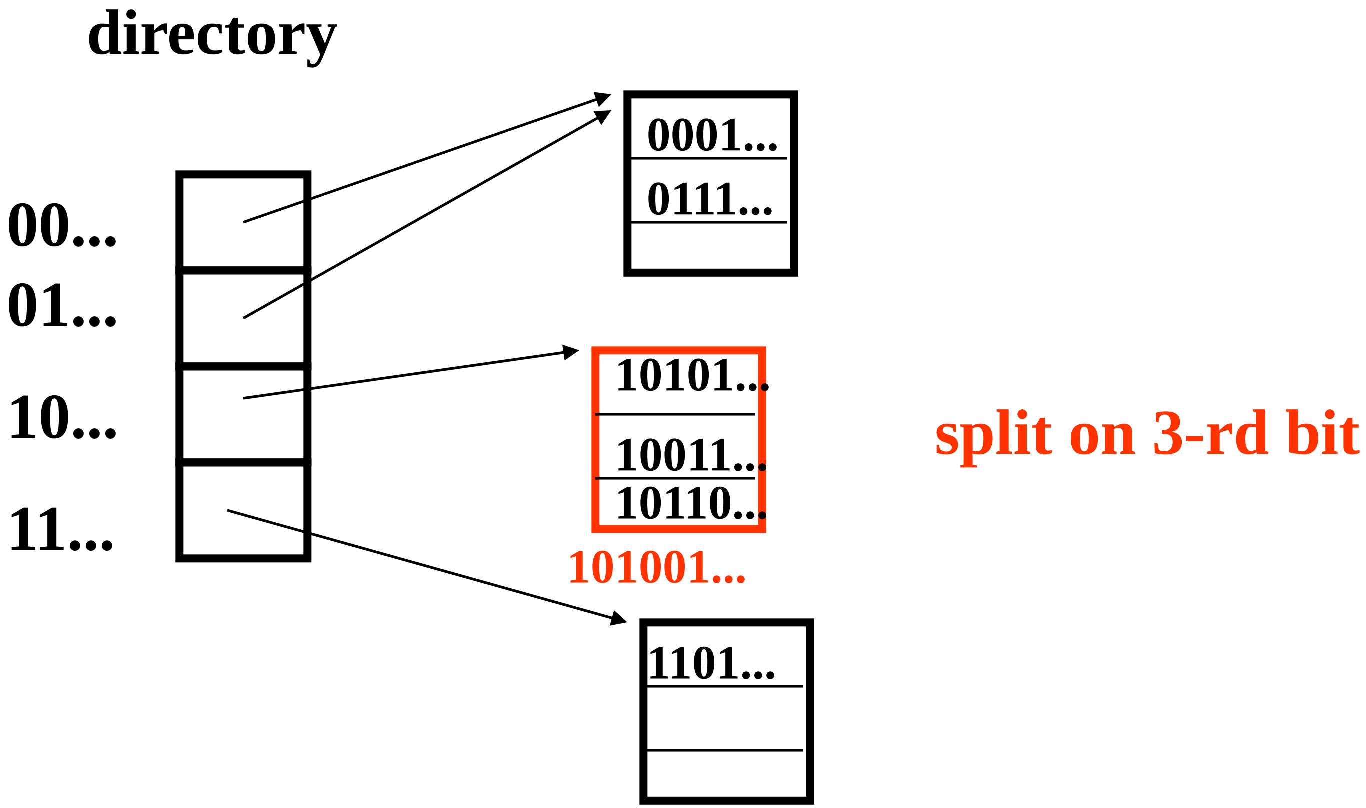
Extendible hashing



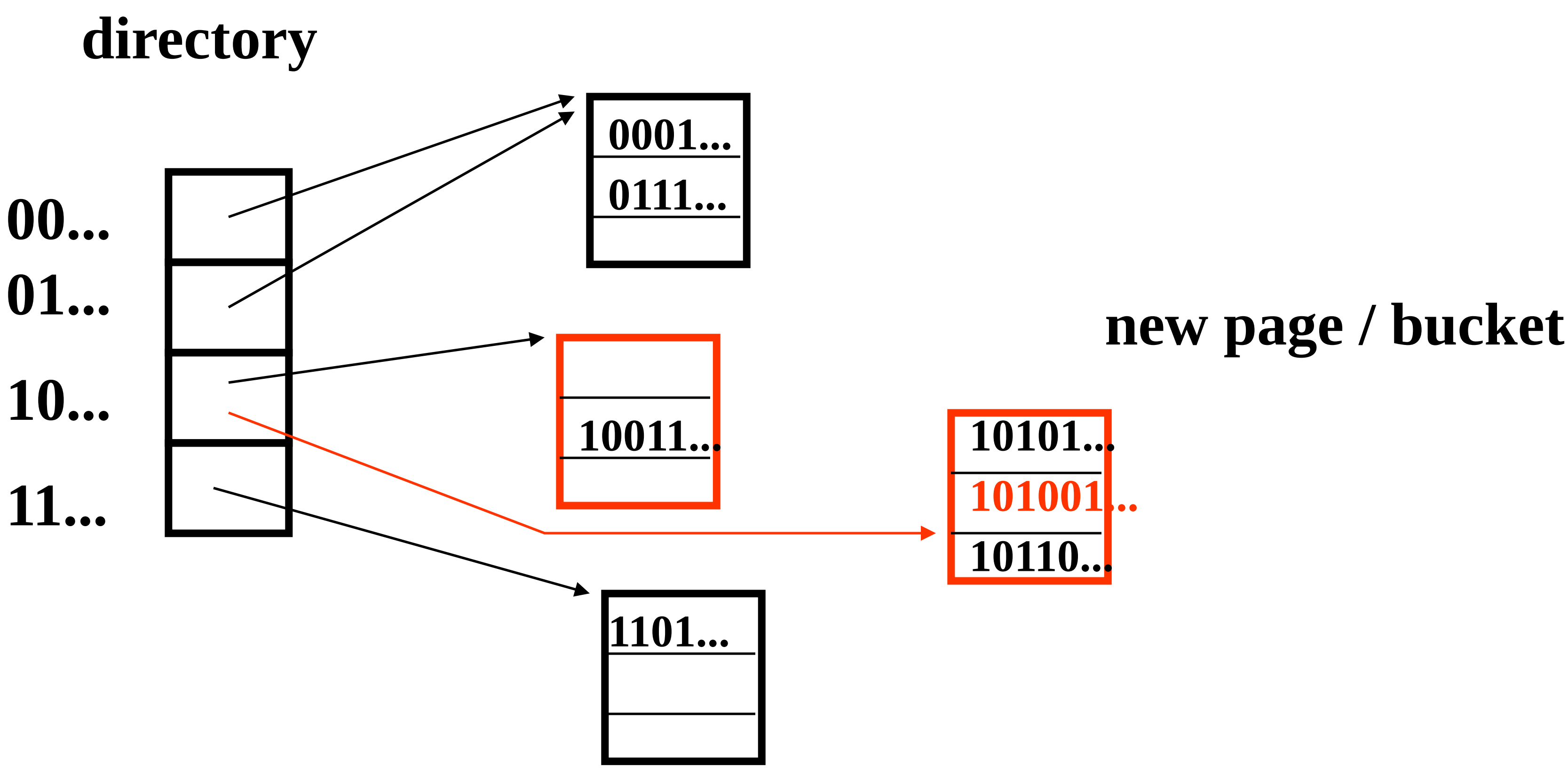
Extendible hashing



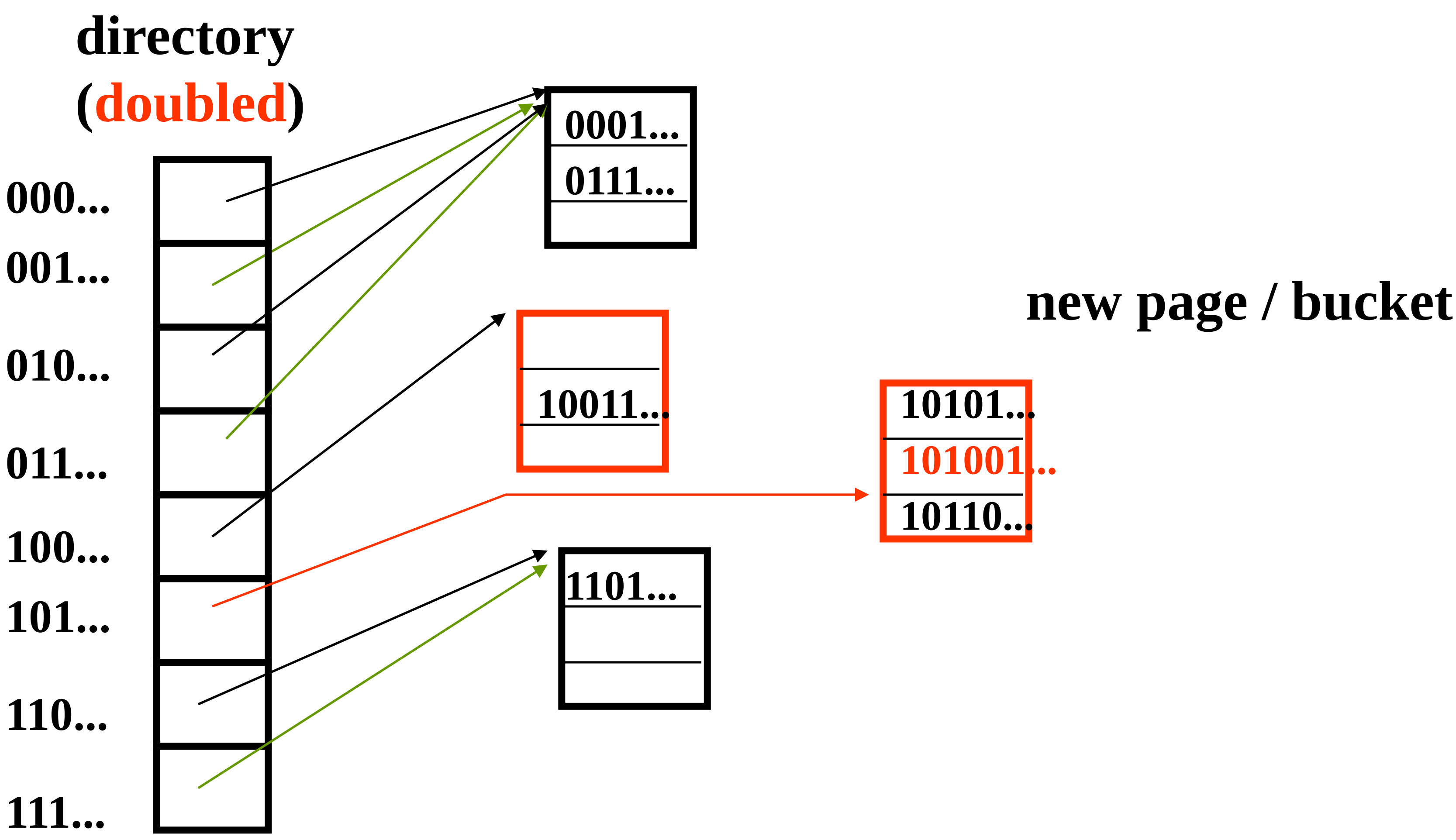
Extendible hashing



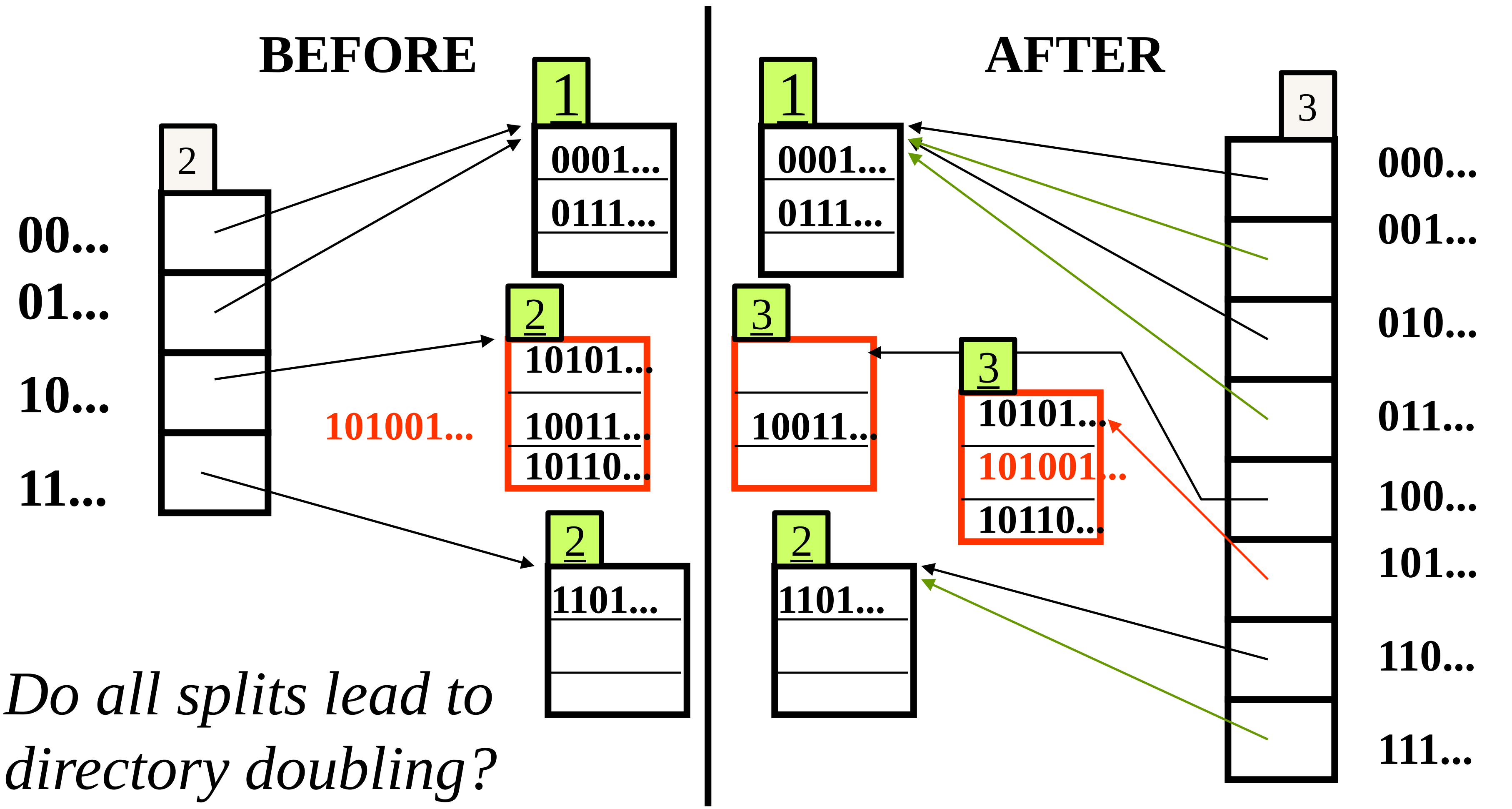
Extendible hashing



Extendible hashing

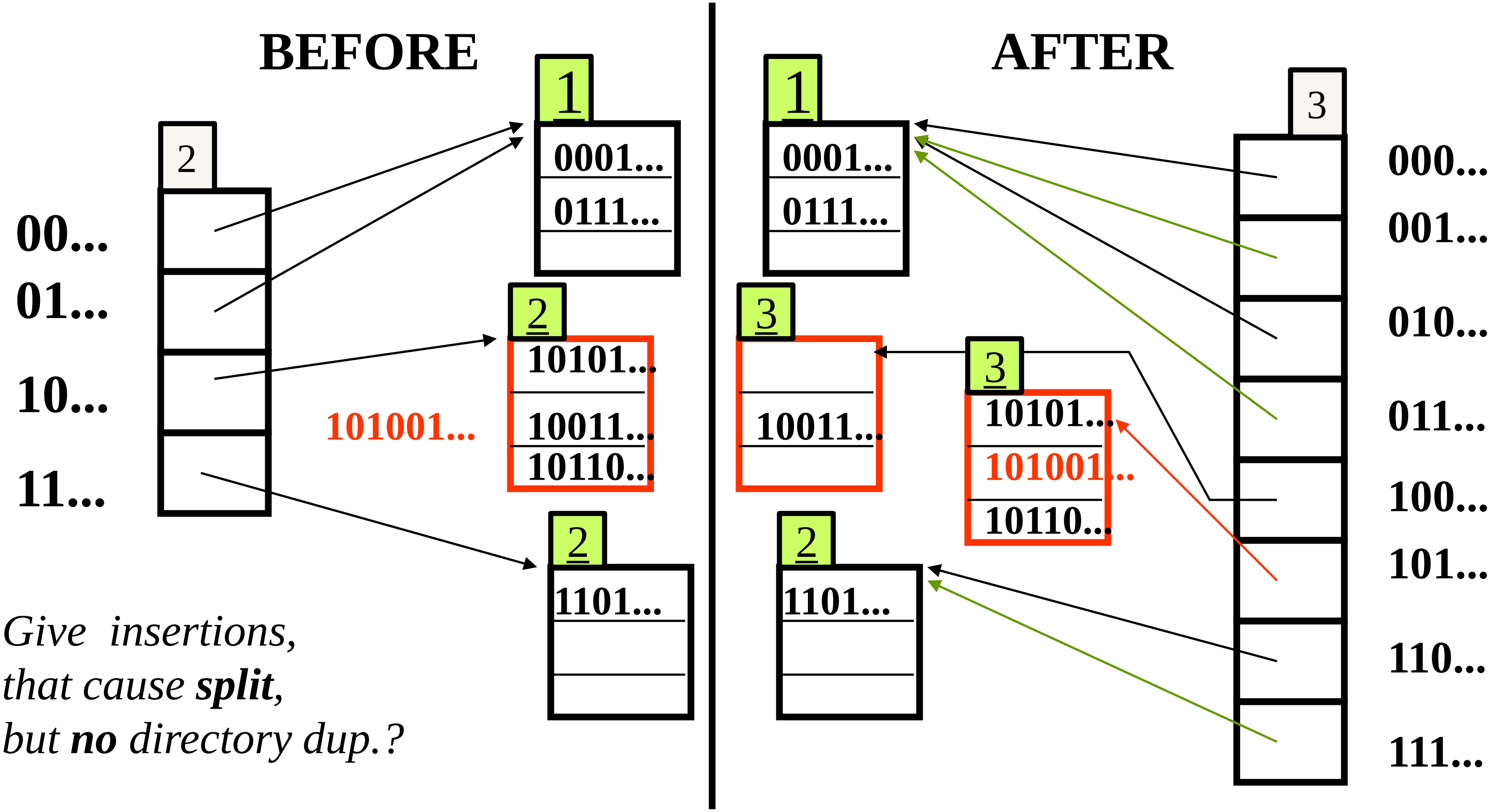


Extendible hashing



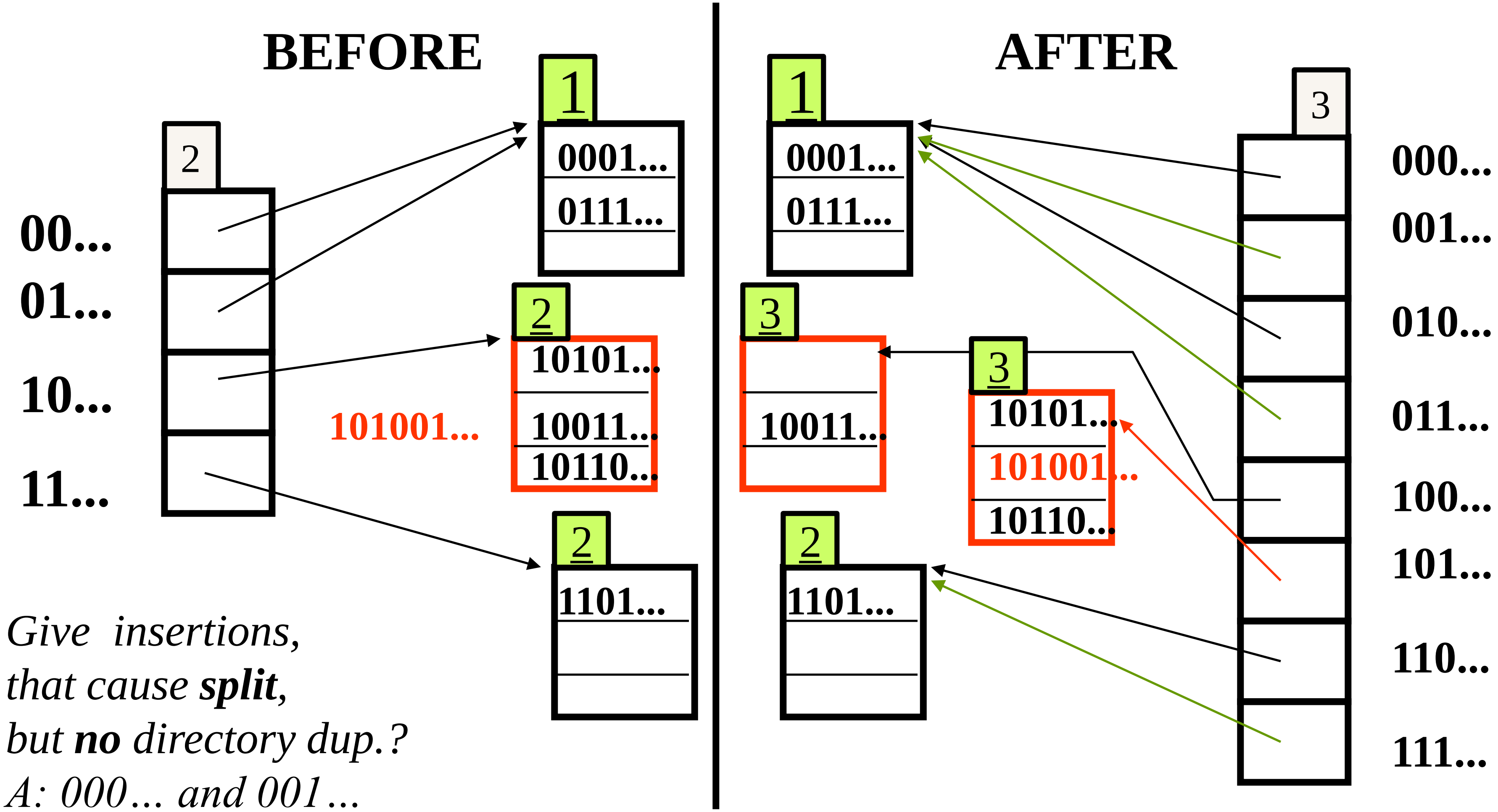
Do all splits lead to directory doubling?
A:NO! (most don' t)

Extendible hashing



Give insertions,
*that cause **split**,*
*but **no** directory dup.?*

Extendible hashing



Give insertions, that cause ***split****, but* ***no*** *directory dup.?*
A: 000… and 001…

Linear hashing

Motivation: 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

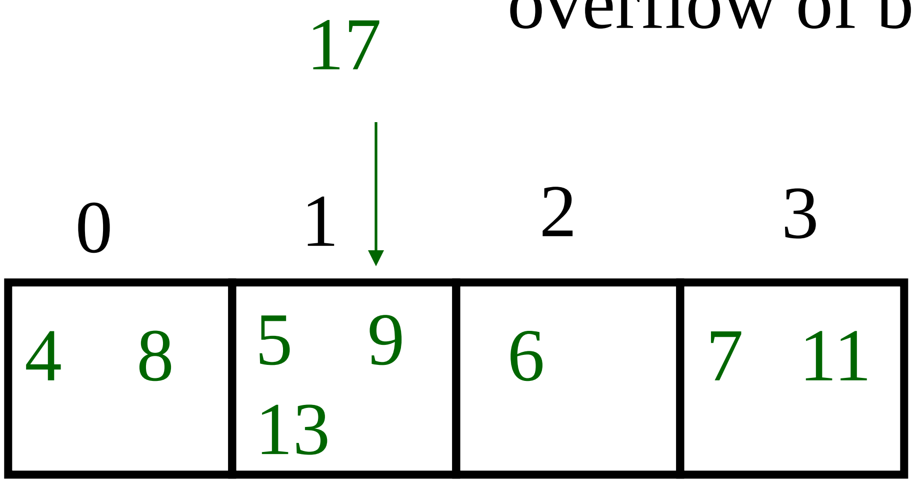
0	1	2	3
4 8	5 9 13	6	7 11

Linear hashing

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

overflow of bucket#1

bucket- id



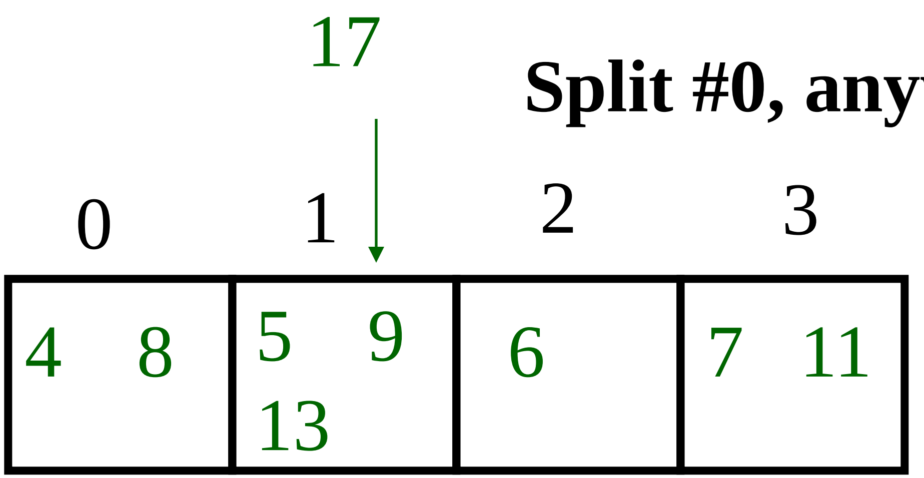
Linear hashing

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

overflow of bucket#1

Split #0, anyway!!!

bucket- id



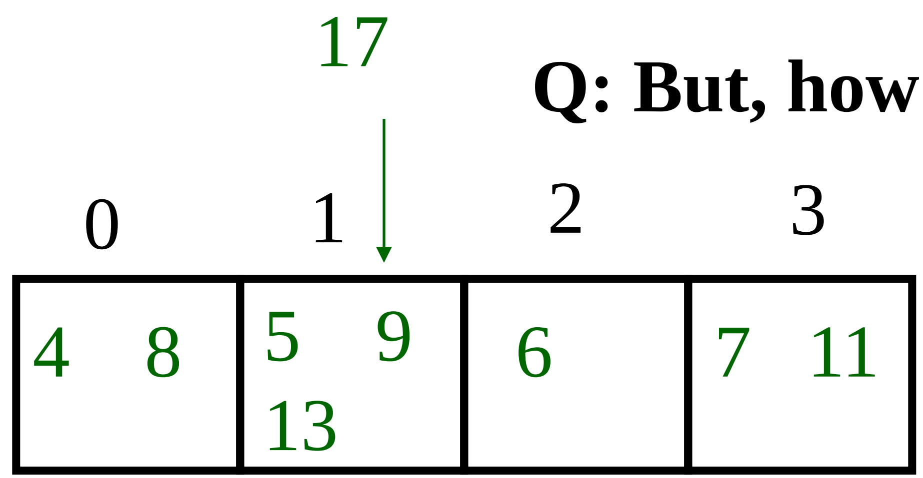
Linear hashing

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

Split #0, anyway!!!

Q: But, how?

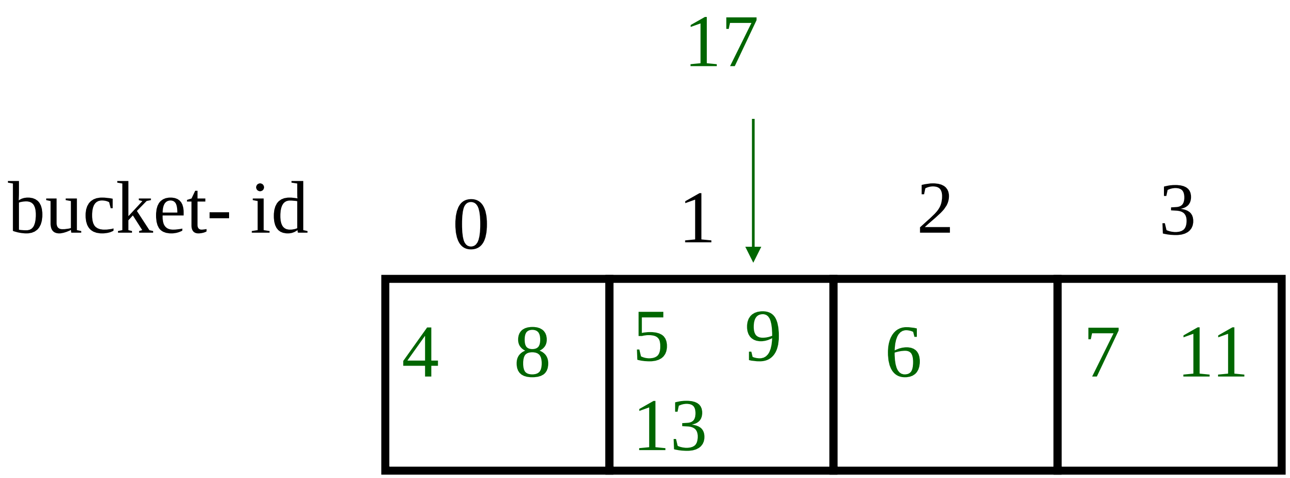
bucket- id



Linear hashing

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

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



Linear hashing - after split:

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

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

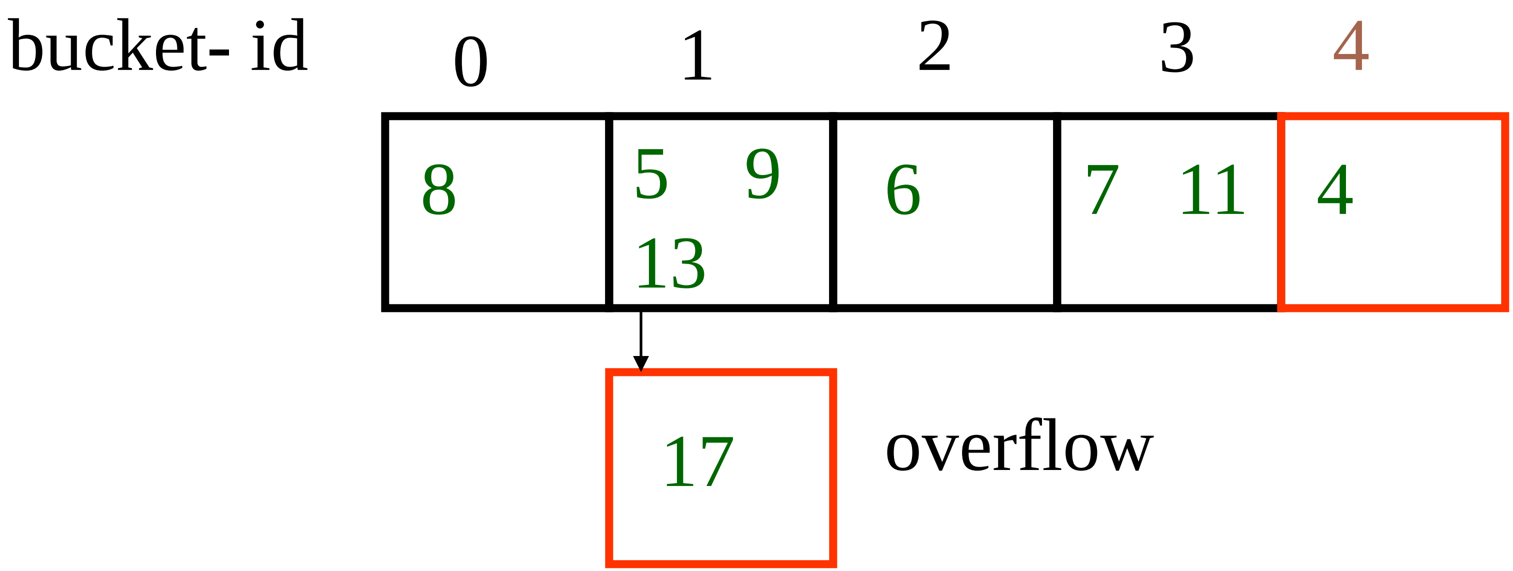
bucket- id	0	1	2	3	4
	8	5 9 13	6	7 11	4

17

Linear hashing - after split:

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

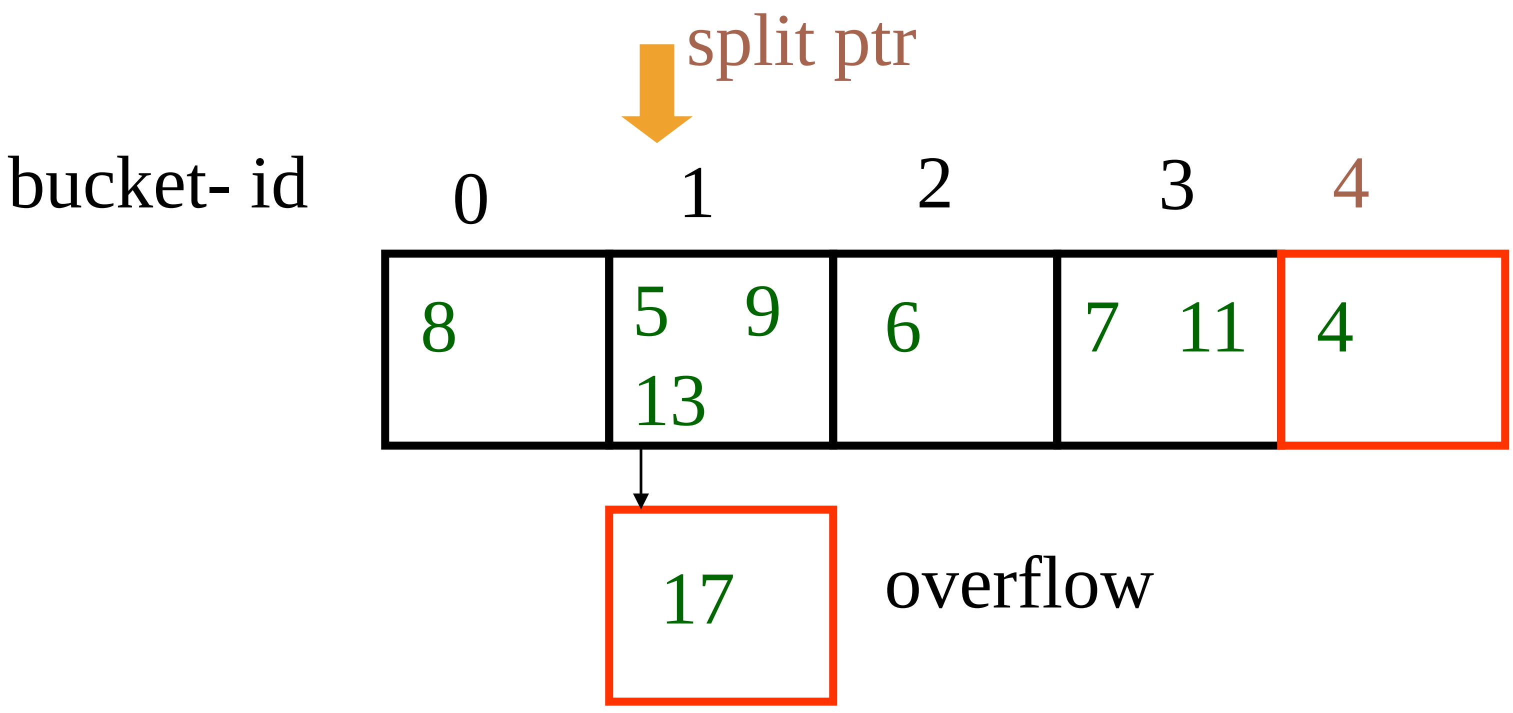
$$h1(x) = x \bmod (2*N)$$



Linear hashing - after split:

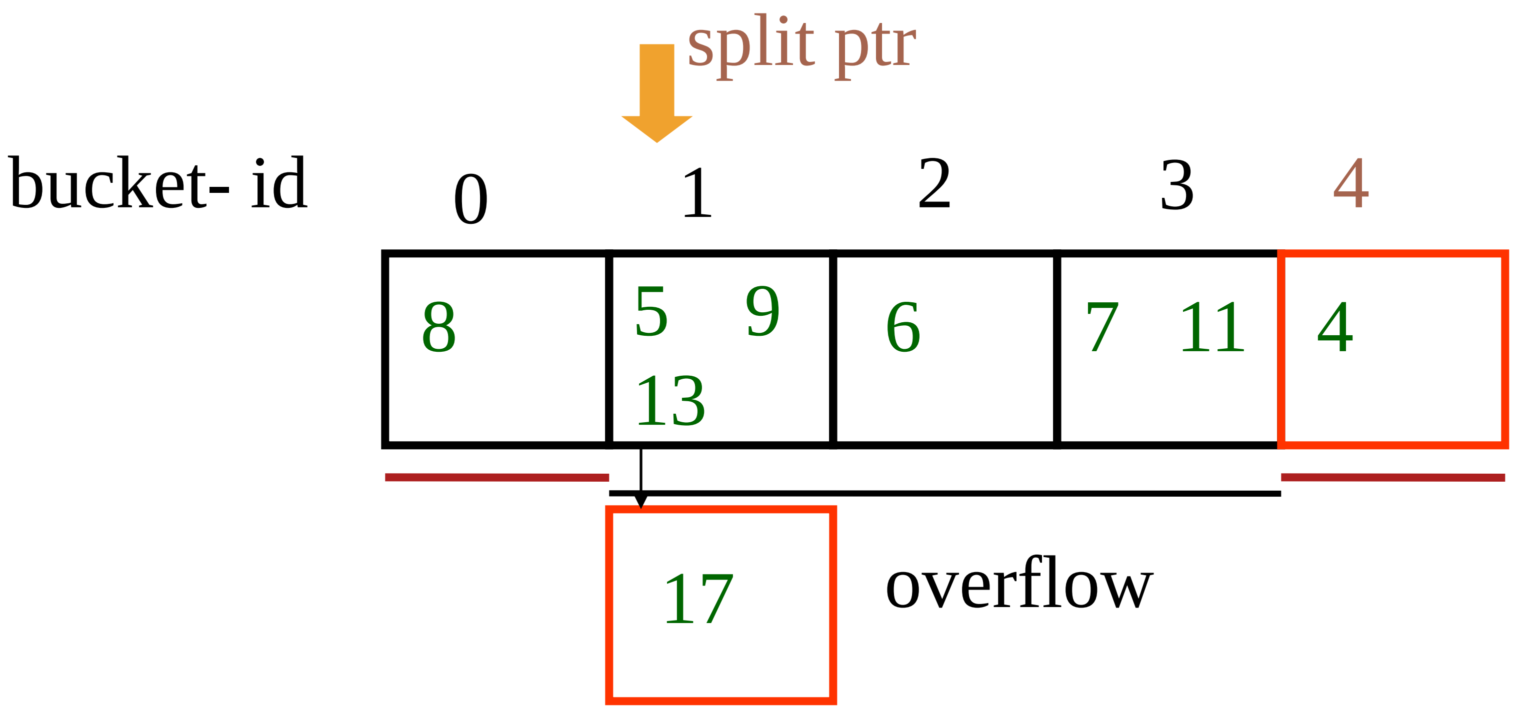
A: use two h.f.: $h0(x) = x \bmod N$

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



Linear hashing - searching?

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

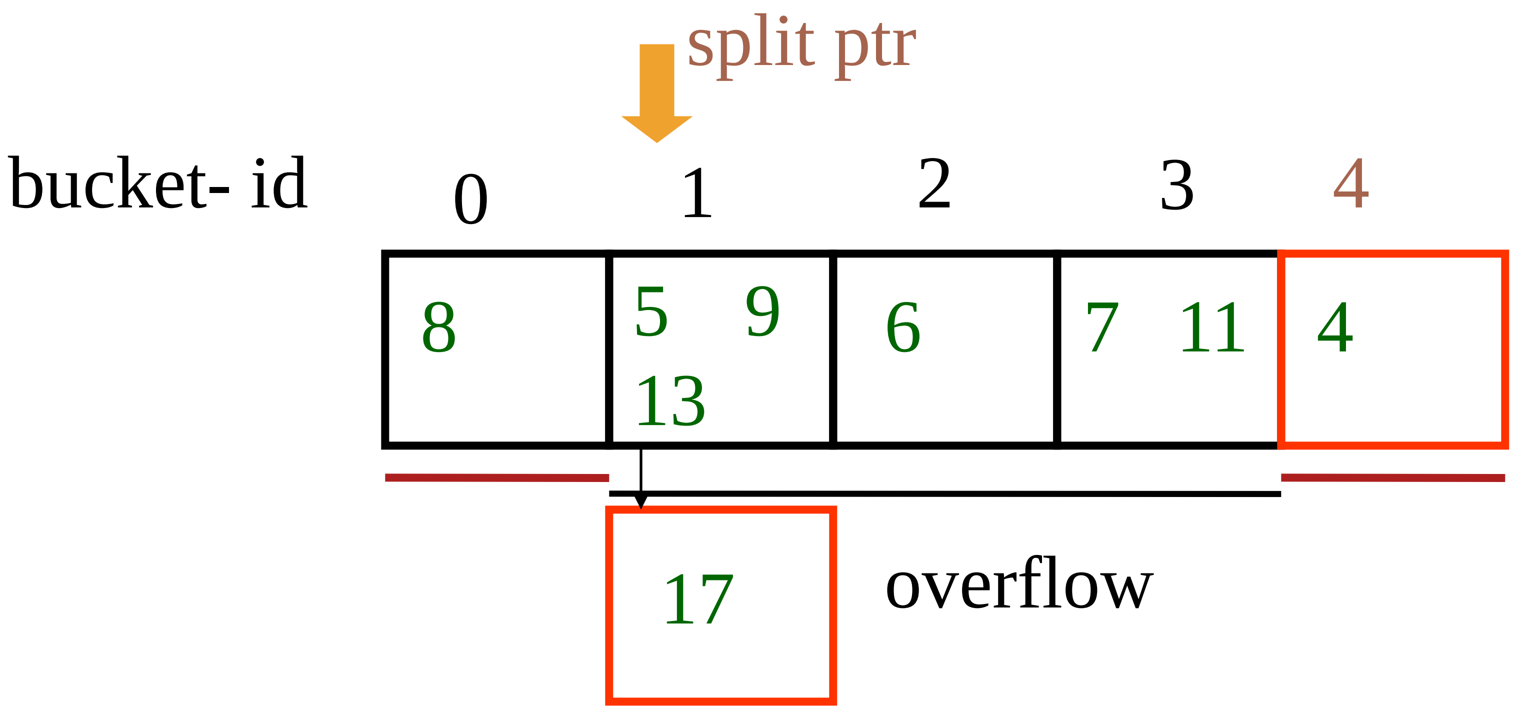


Linear hashing - searching?

Q1: find key ‘6’ ?

Q2: find key ‘4’ ?

Q3: key ‘8’ ?



Linear hashing - insertion?

Algo: insert key '*k*'

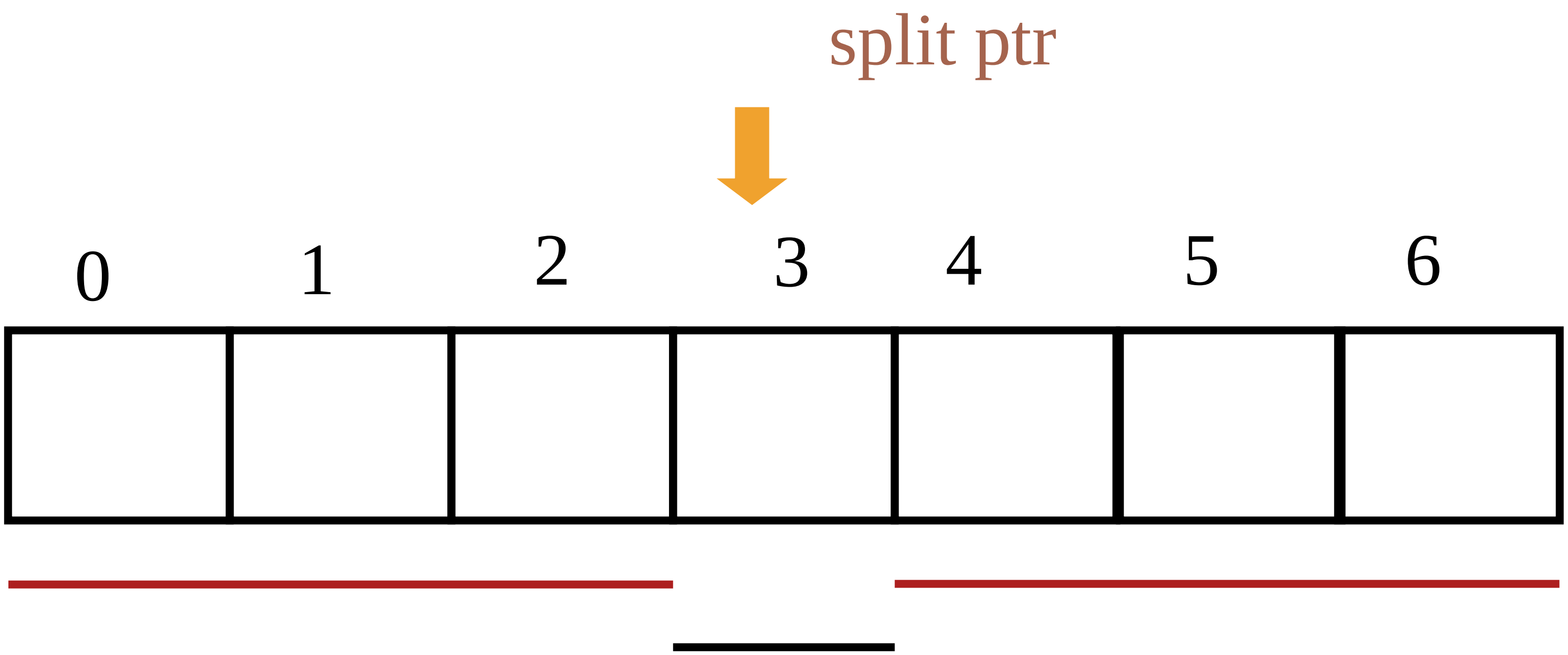
- compute appropriate bucket '*b*'
- if the **overflow criterion** is true
    - split the bucket of 'split-ptr'
    - split-ptr ++ **(*)**

what if we reach the right edge??

Linear hashing - split now?

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

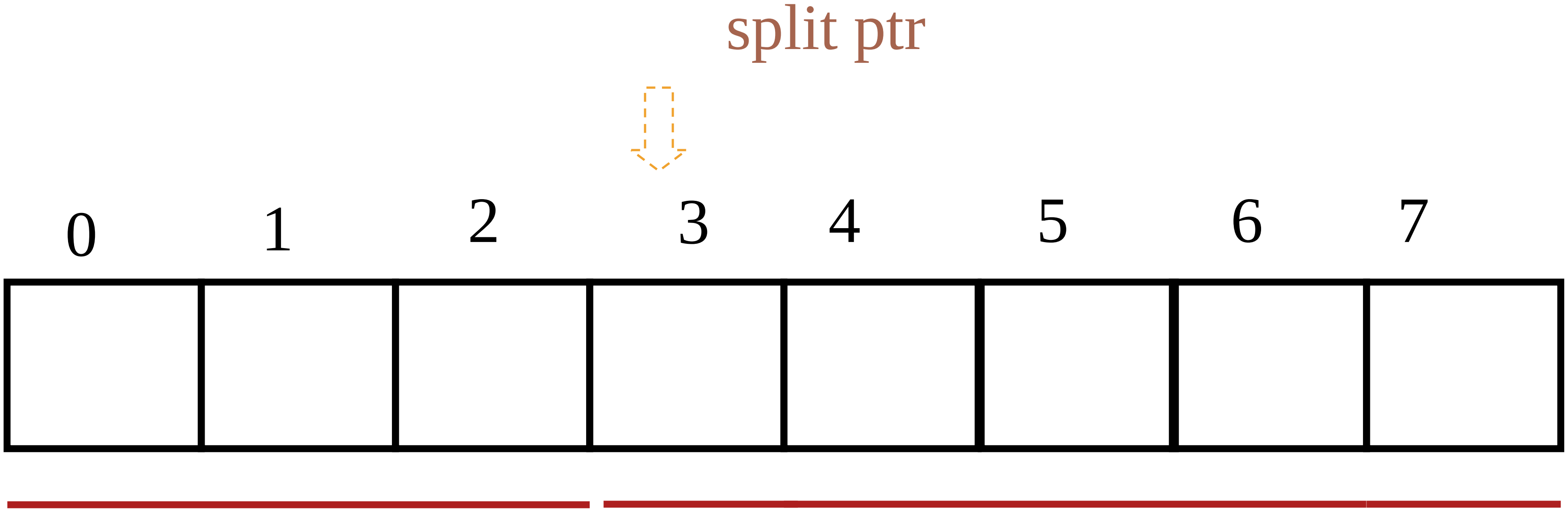
*h1(x) = x mod (2*N) for the splitted ones)*



Linear hashing - split now?

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

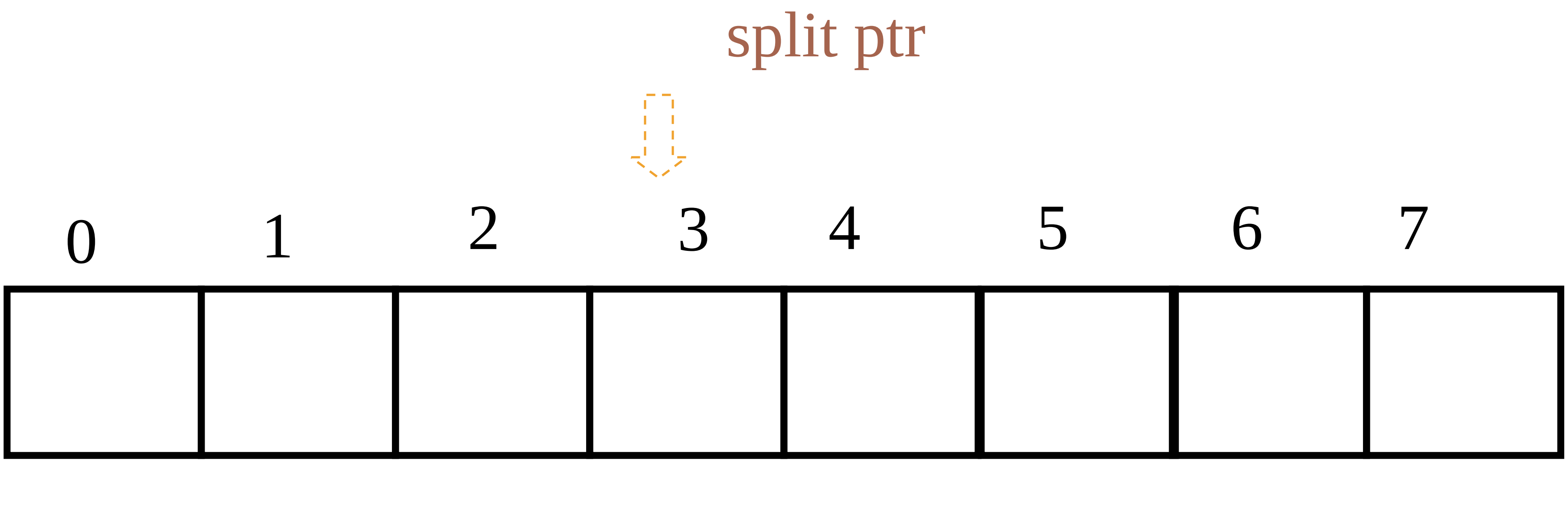
$h1(x) = x \bmod (2*N)$ (for the splitted ones)



Linear hashing - split now?

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

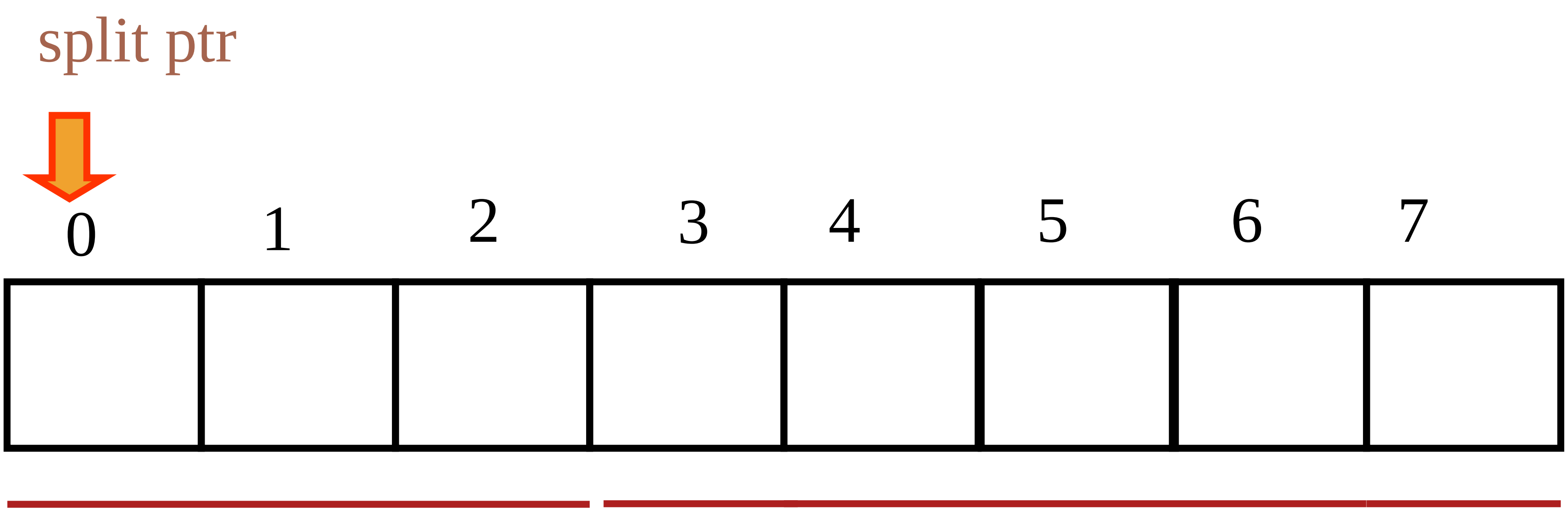
*h1(x) = x mod (2*N)* *(for the splitted ones)*



Linear hashing - split now?

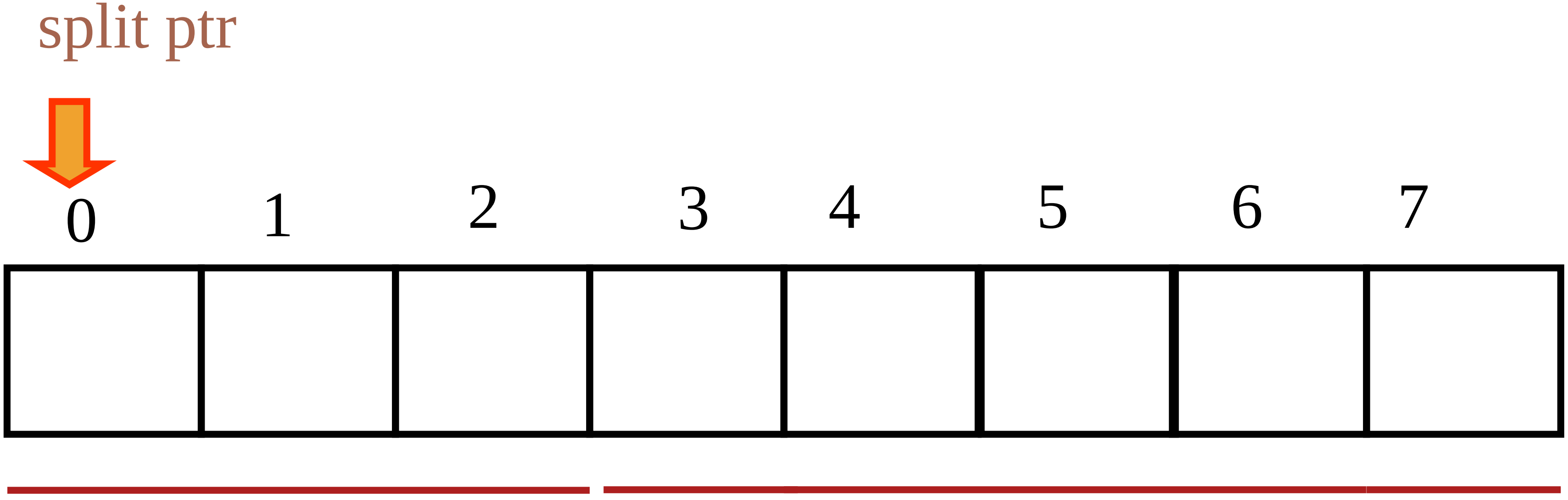
~~$h0(x) = x \bmod N$ (for the un-split buckets)~~

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

Hashing - pros?