CS 4604: Introduction to Database Management Systems

Hashing and Sorting

Virginia Tech CS 4604 Sprint 2021

Instructor: Yinlin Chen

Today's Topics

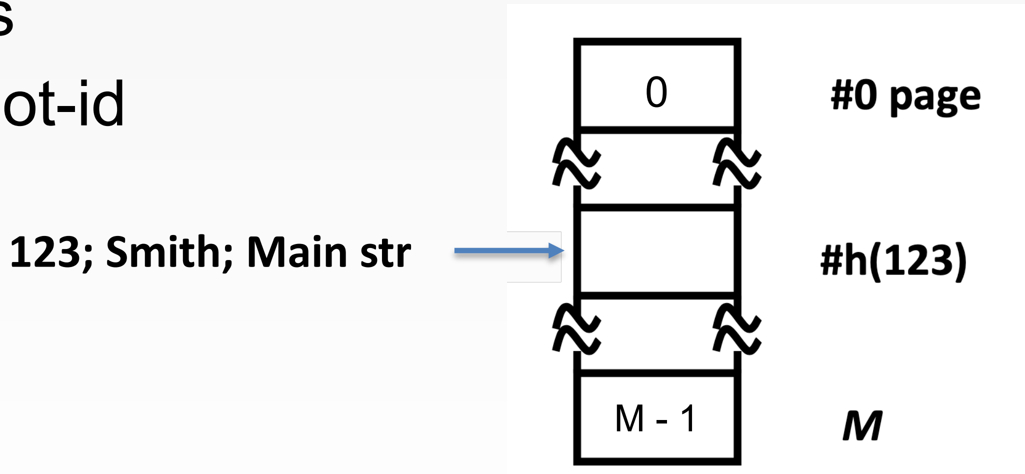
- Hashing
  - Static Hashing
  - Extendible Hashing
  - Linear Hashing
- Sorting
  - Two-way merge sort
  - External merge sort
  - Fine-tunings
  - B+ trees for sorting

Hashing

- Many times, we don’t require order
  - Problem: “find EMP record with ssn=123”
- Static Hashing
- Dynamic hashing techniques:
  - Extendible Hashing
  - Linear Hashing

(Static) Hashing

- Each hash bucket has one **primary** page and possibly additional **overflow** pages
- Page holds many records
- hash function: h(key) = slot-id



(Static) Hashing

- Insert:
  1. hash function: h(key) find the correct bucket

  2.1 There is a space, insert a data there

  2.2 There is no space

     step 1. allocate a new **overflow** page and then insert a data there

     step 2. add that page to the **overflow chain** of the bucket

(Static) Hashing

- Delete:
  1. hash function: h(key) find the correct bucket
  2. Locate the data then remove it

  2.1 Last item in an overflow page? ~~overflow page~~

Cost of (Static) Hashing

- Search: One disk I/O
- Insert and Delete: Two disk I/O
- Many overflow pages → poor performance

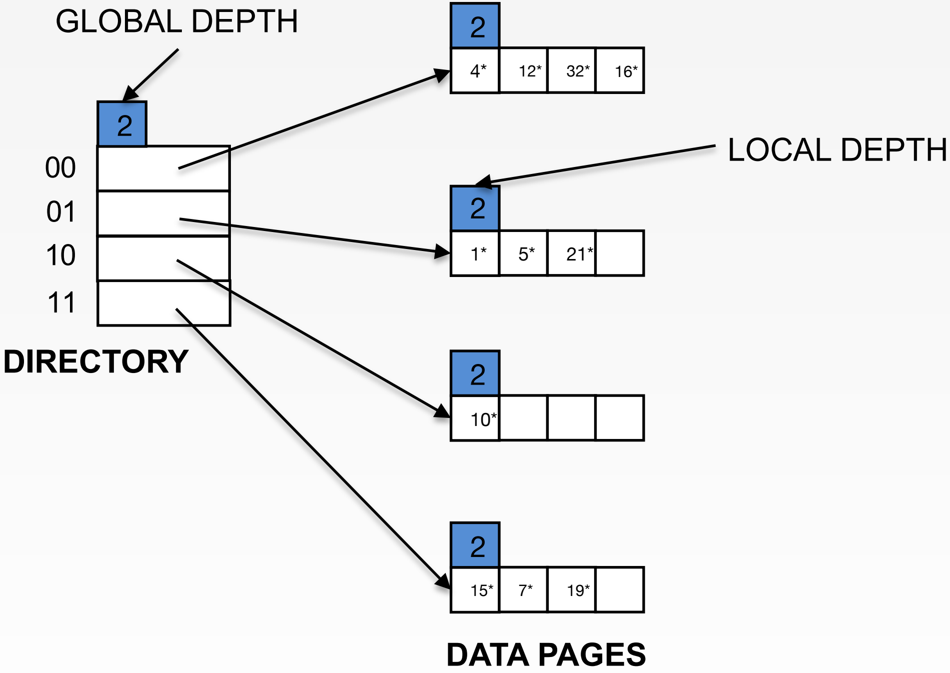
Problem with static hashing

- The number of bucket is fixed
- Underflow:
  - A lot of space is wasted (underutilization)
- Overflow:
  - Poor performance
- Better solution: **Dynamic hashing**

Extendible hashing

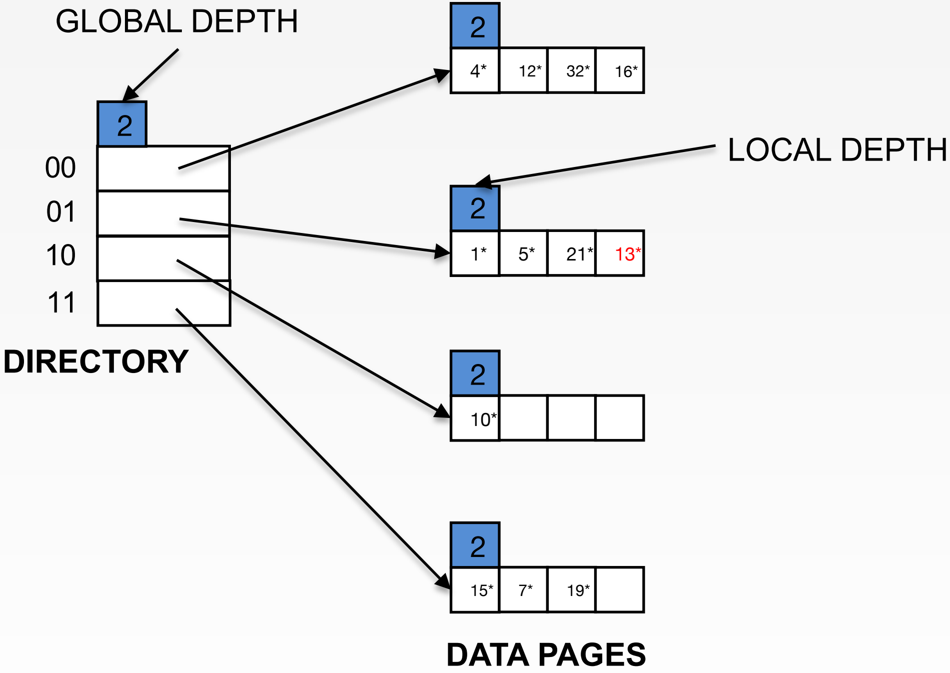
- Idea:
  - Use a directory of pointers to buckets
  - Double the directory
  - Double the size of the number of buckets
  - Splitting the bucket that overflowed

Extendible hashing



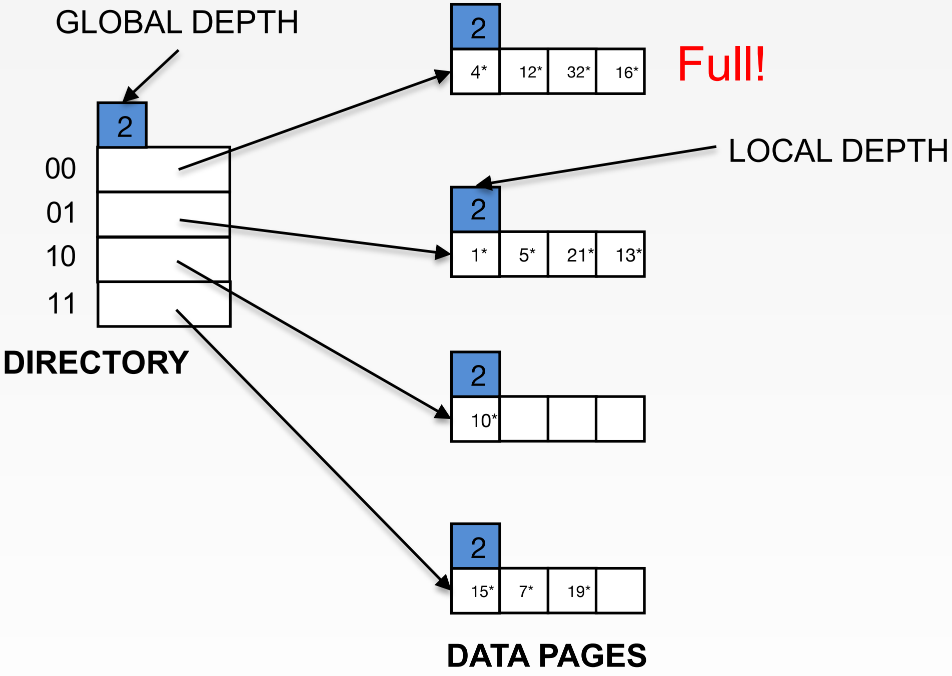
Search 5: 101

Extendible hashing



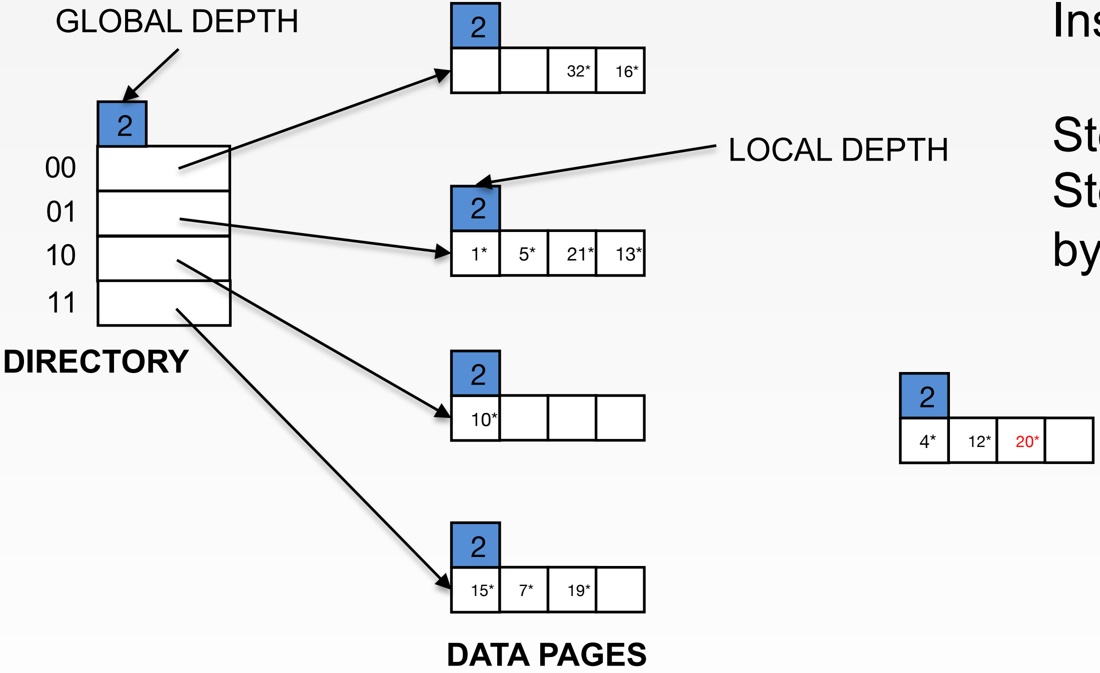
Insert 13: 1101

Extendible hashing



Insert 20: 10100

Extendible hashing

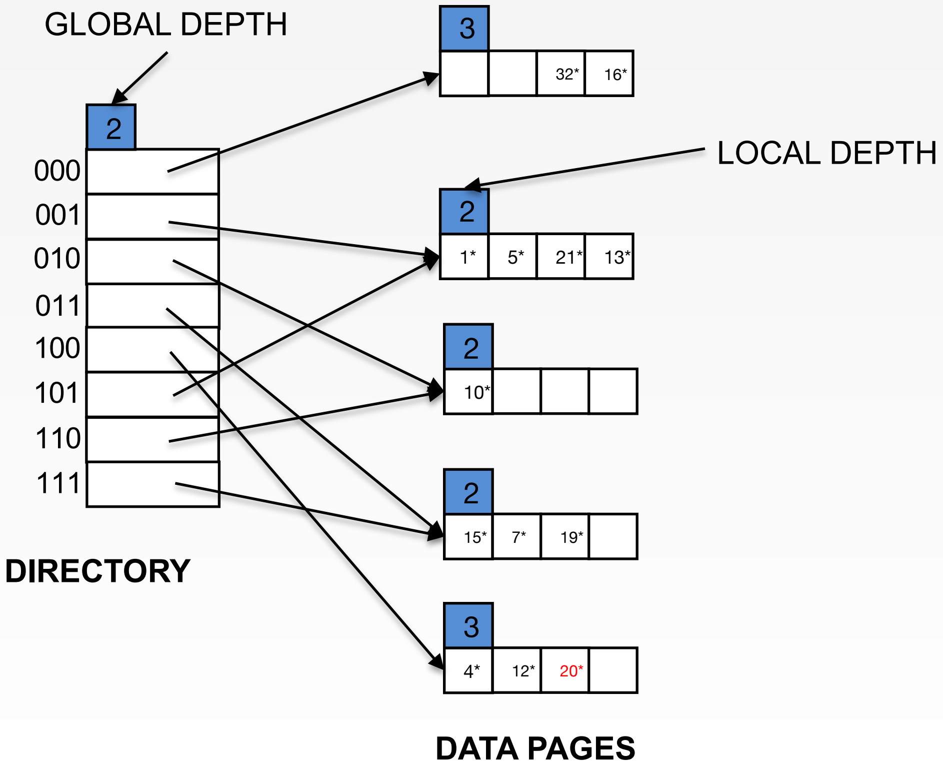


Insert 20: 10100

Step 1: **split** the bucket
Step 2: redistribute the contents by last *three* bits of h(r)

4	100
12	1100
32	100000
16	10000
20	10100

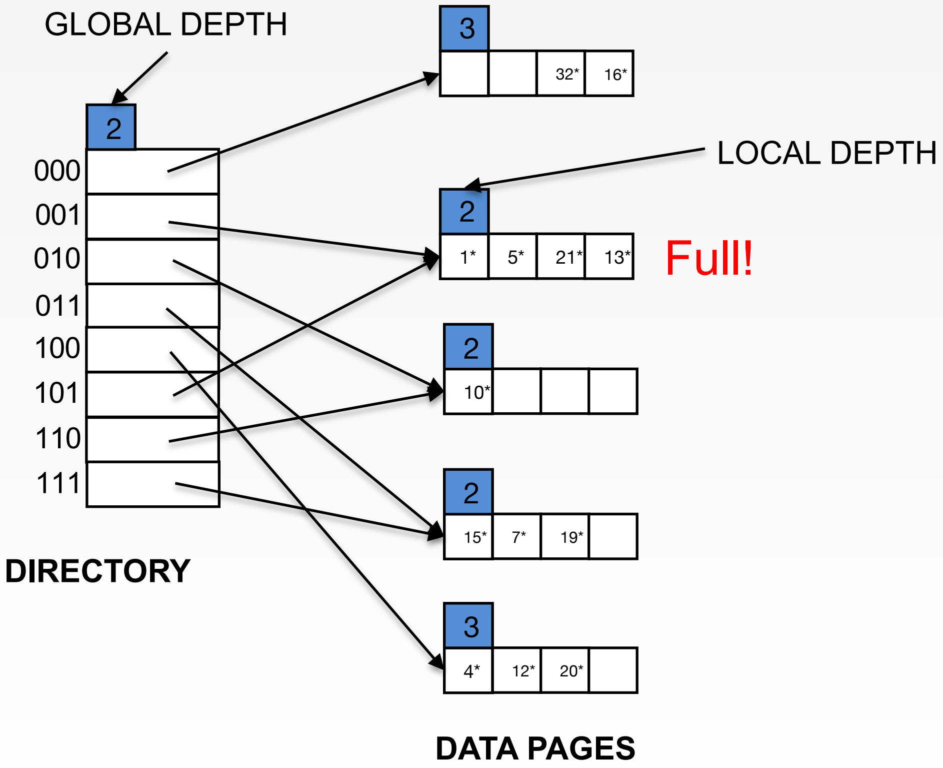
Extendible hashing



Insert 20: 10100

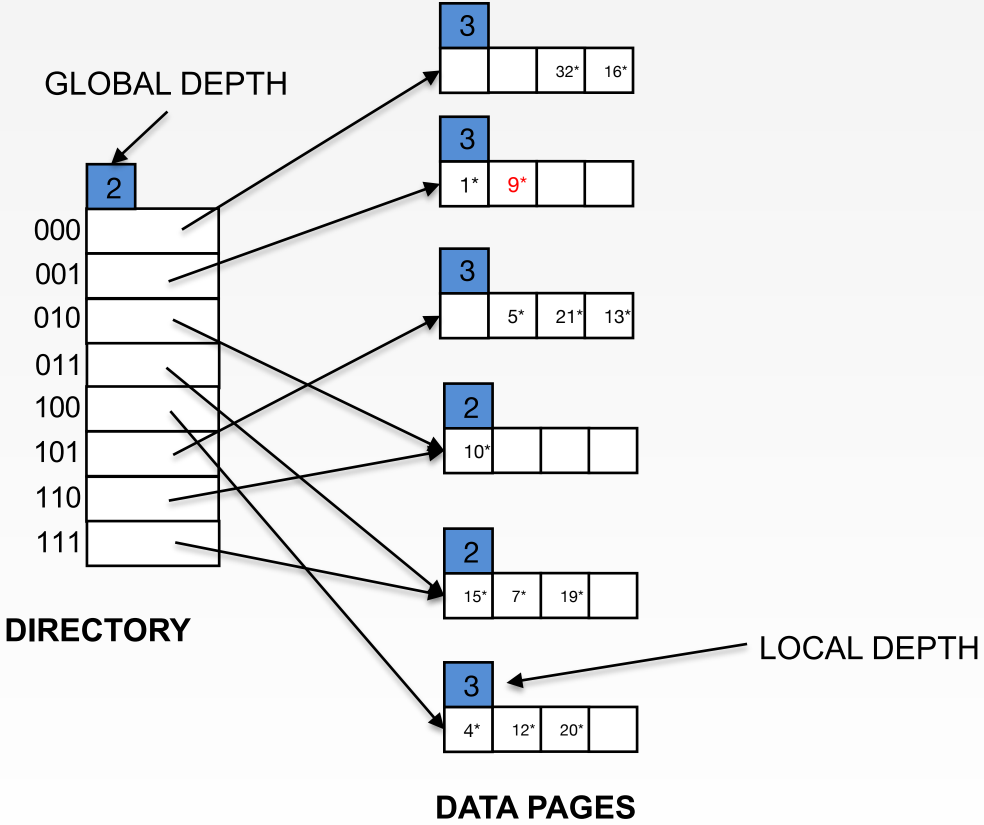
Step 3: **double** the directory

Extendible hashing



Insert 9: 1001

Extendible hashing



Insert 9: 1001

Step 1: **split** the bucket
Step 2: redistribute the contents by last *three* bits of h(r)
Step 3: no need to **double** the directory

How to know if we need to double a directory?

- Global and local depth are the same (Double!)

Cost of Extendible Hashing

- Search: One disk I/O or (worse) two I/Os (and rare)
- Insert and Delete: Two disk I/O
- Better performance
- Special case: collisions, or data entries with the same hash value.
  - Need overflow pages

Linear hashing

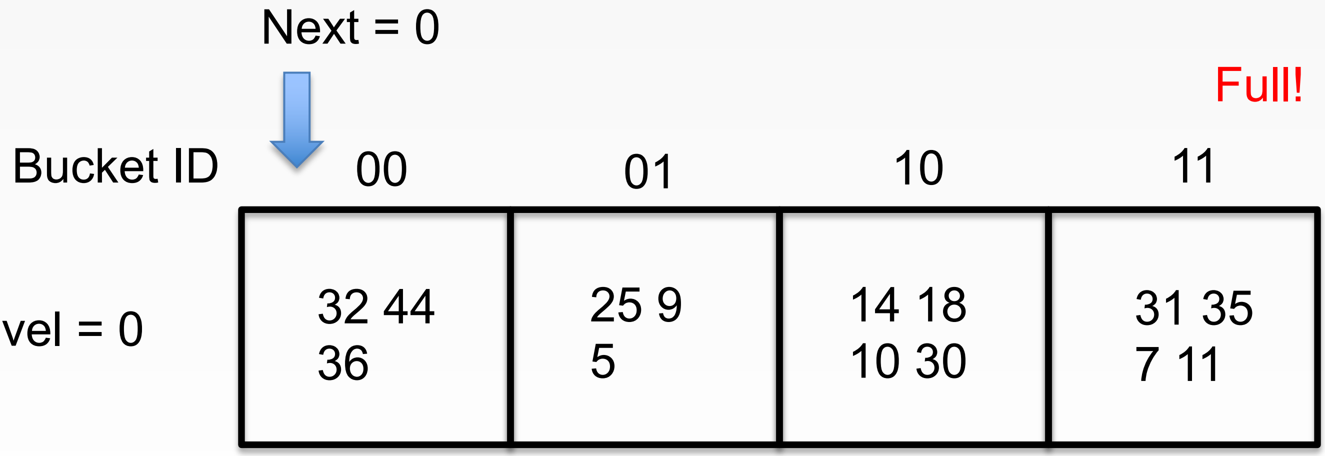
- It does not require a directory
- Deal naturally with collisions
- Still need overflow pages and chains
- Utilizes a *family* of hash functions: h_0, h_1, h_2, ….
  - h_0: M buckets
  - h_1: 2M buckets
  - h_2: 4M buckets
  - …

Linear hashing

- Number of N buckets (N = 4)
- d_0 is the number of bits needed to represent N ($d_0 = 2$)
- h_0 is h mod 4: 0 to 3
- $d_1 = d_0 + 1 = 3$
- h_1 is h mod (2 * 4): 0 to 7

Linear hashing

- h(x) = x mod N (N = 4)
- Assume capacity: 4 records per bucket
- Insert key 43 (101011)



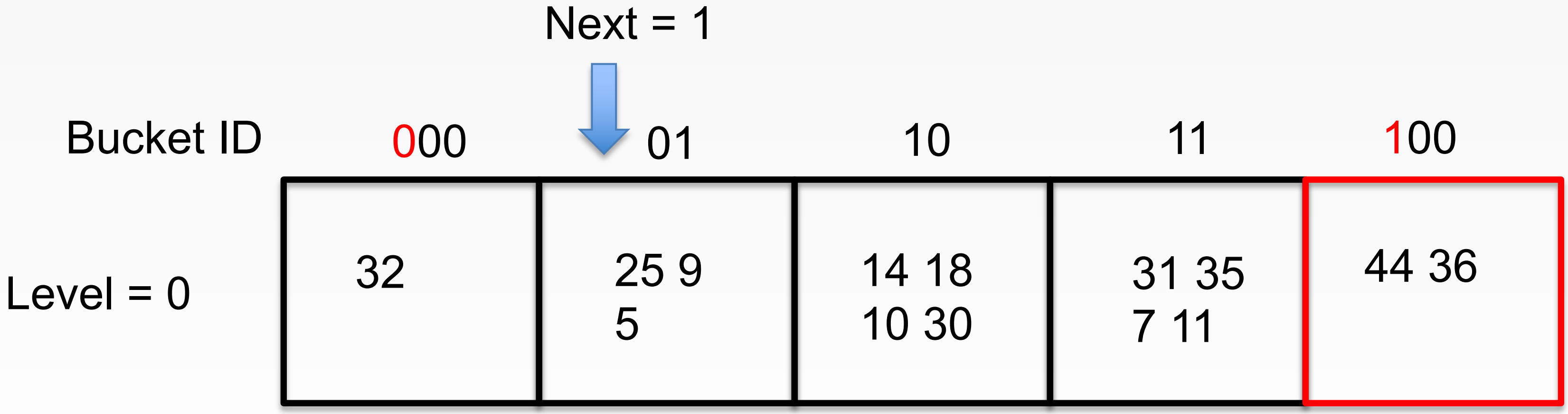
Next = 0

Full!

Bucket ID	00	01	10	11
Level = 0	32 44 36	25 9 5	14 18 10 30	31 35 7 11

Linear hashing – split first

- h(x) = x mod N (N = 4)
- Assume capacity: 4 records per bucket
- Insert key 43 (101011)

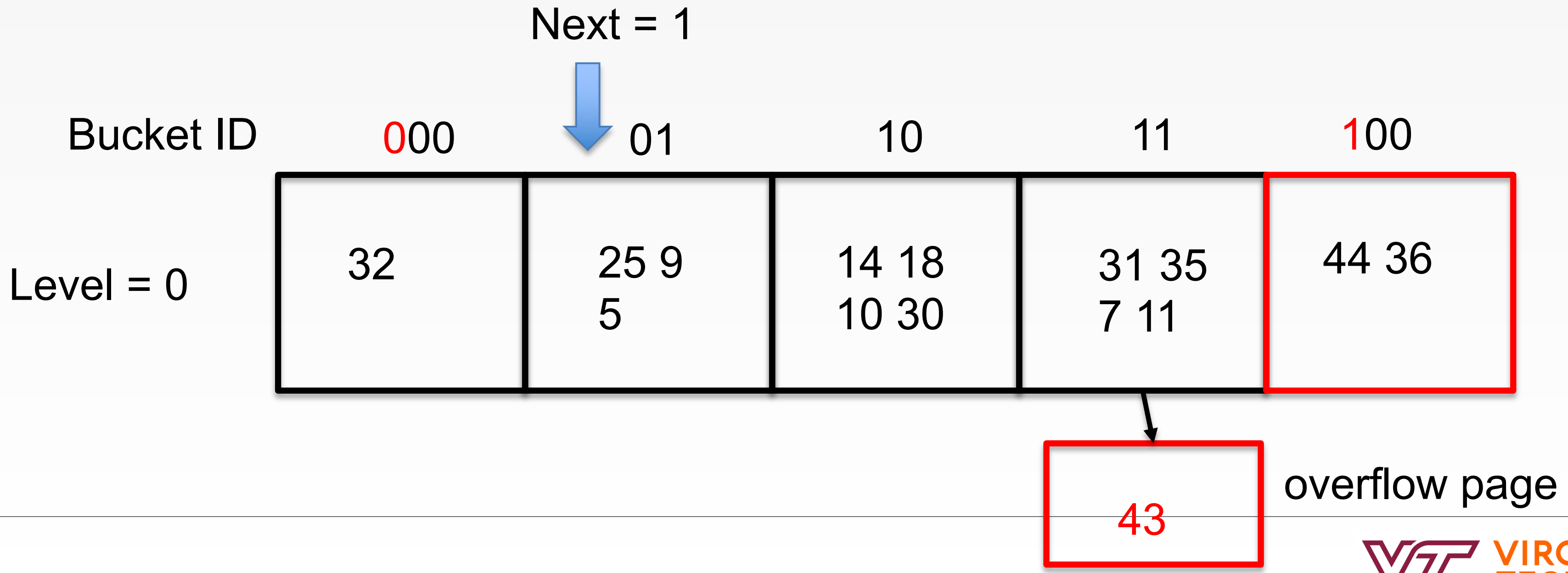


Next = 1

Bucket ID	000	01	10	11	100
Level = 0	32	25 9 5	14 18 10 30	31 35 7 11	44 36

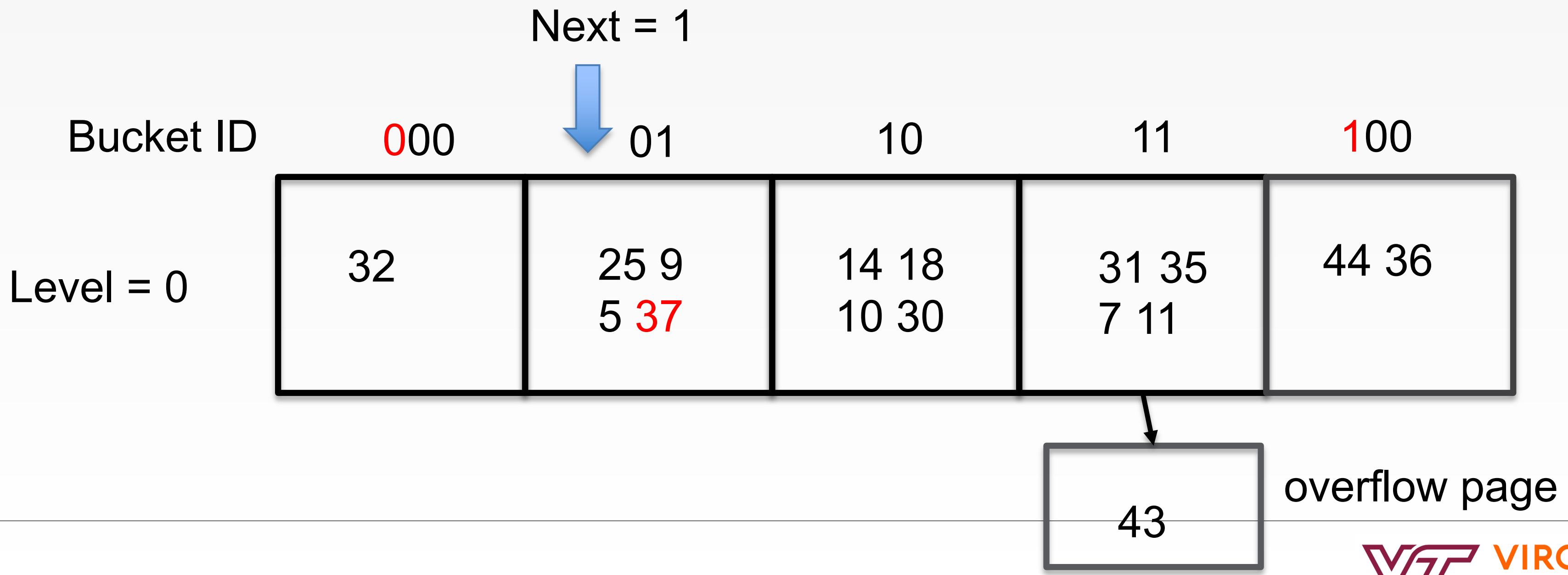
Linear hashing – after split

- $h_0(x)$ = x mod N (N = 4)
- $h_1(x)$ = x mod (2*N)
- Insert key 43 (101011)



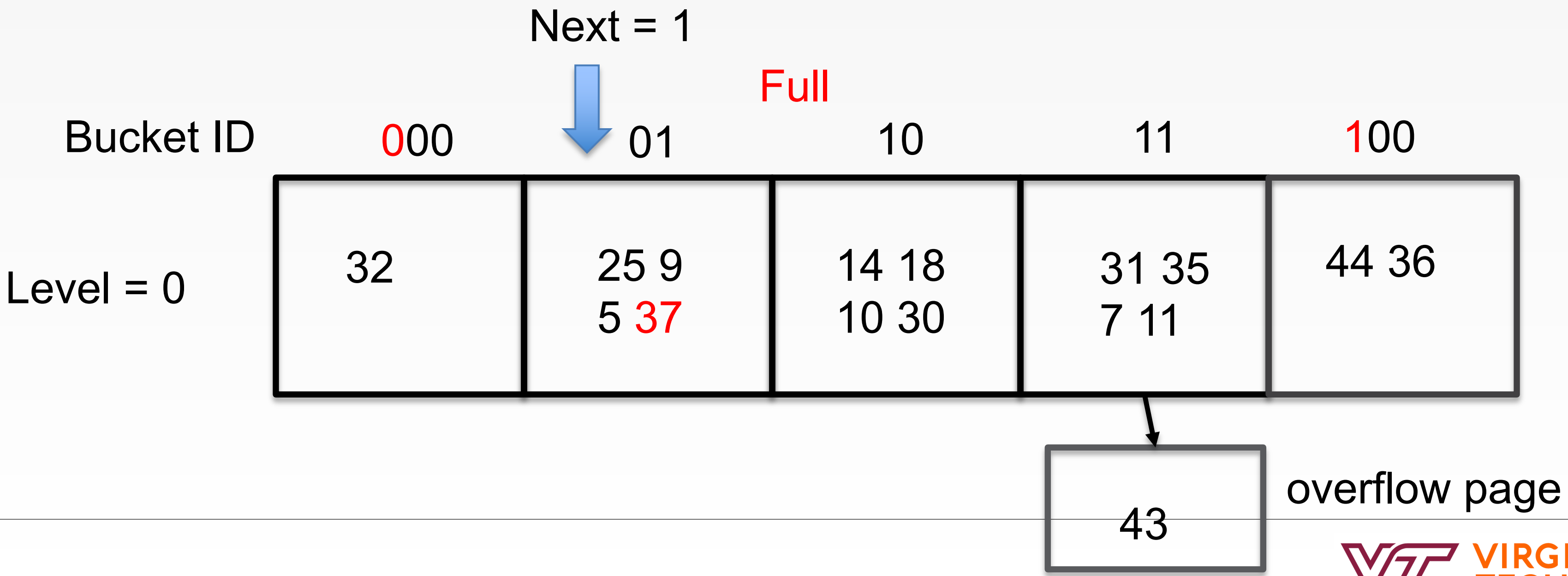
Linear hashing

- $h_0(x) = x \bmod N \ (N = 4)$
- $h_1(x) = x \bmod (2*N)$
- Insert key 37 (100101)



Linear hashing

- $h_0(x) = x \bmod N \ (N = 4)$
- $h_1(x) = x \bmod (2*N)$
- Insert key 29 (11101)



Next = 1

Full

Bucket ID

000

01

10

11

100

Level = 0

32

25 9
5 37

14 18
10 30

31 35
7 11

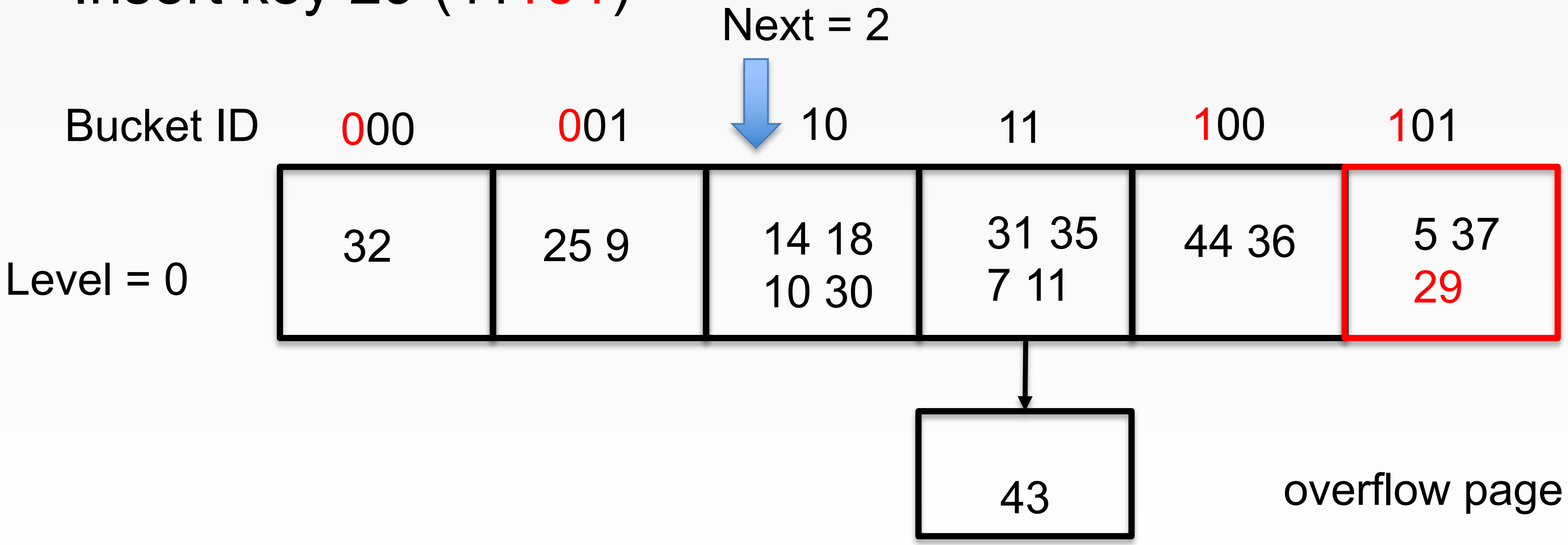
44 36

43

overflow page

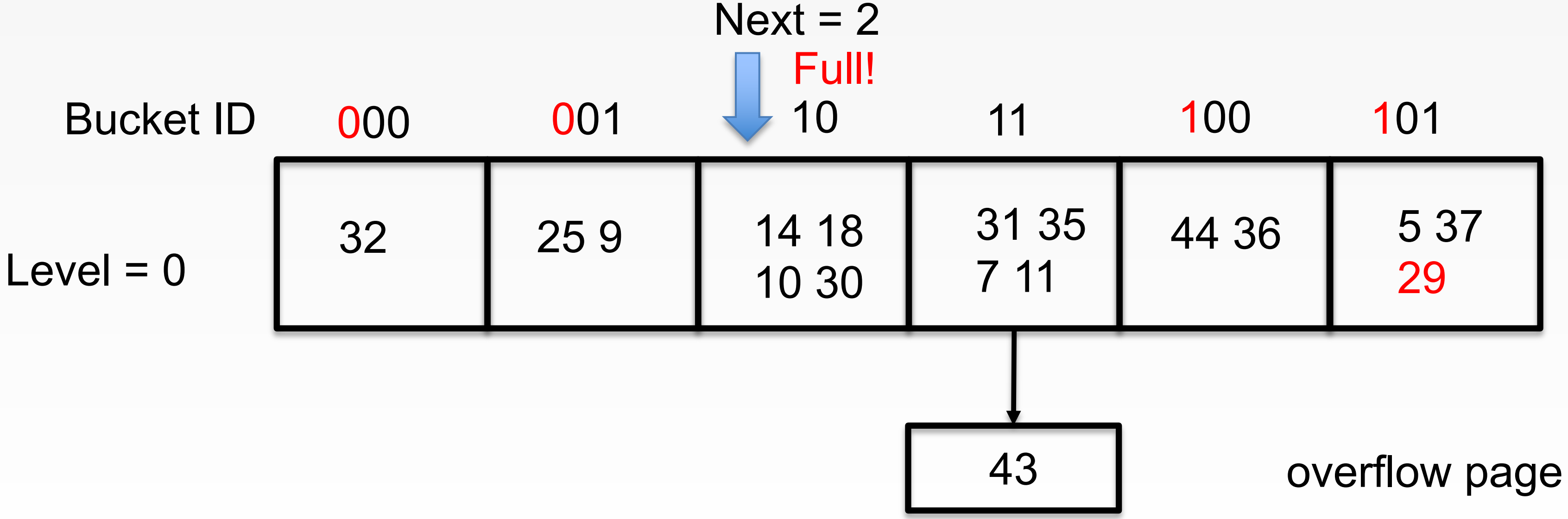
Linear hashing

- $h_0(x)$ = x mod N (N = 4)
- $h_1(x)$ = x mod (2*N)
- Insert key 29 (11101)



Linear hashing

- $h_0(x) = x \bmod N \ (N = 4)$
- $h_1(x) = x \bmod (2*N)$
- Insert key 22 (10110)



Next = 2

Full!

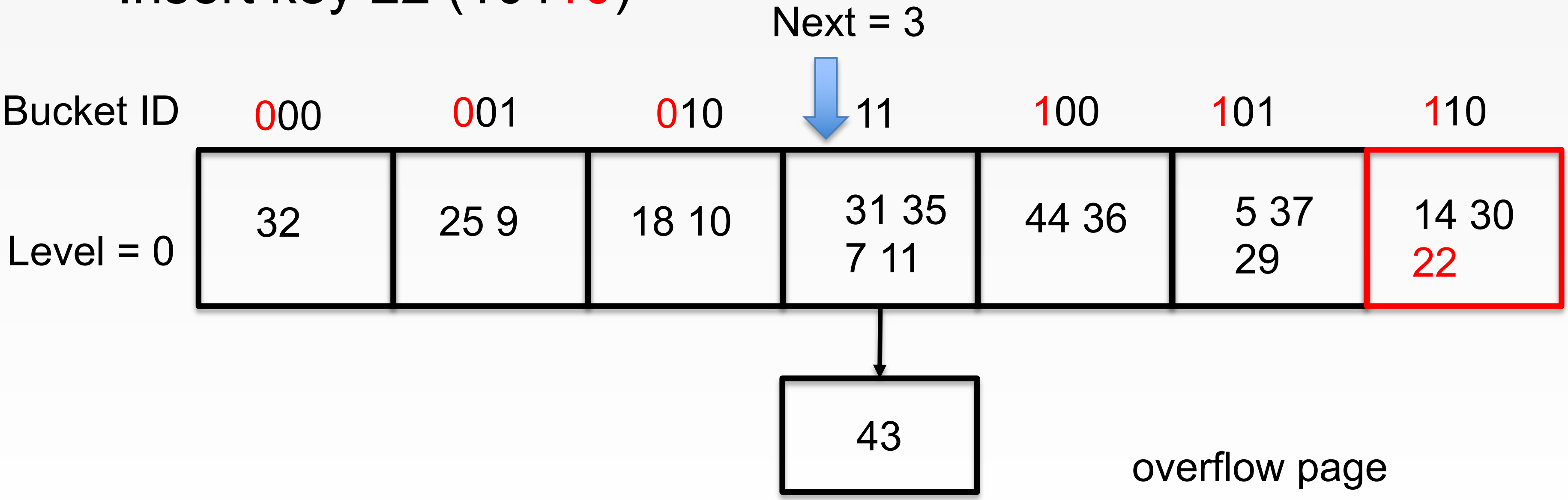
Bucket ID	000	001	10	11	100	101
Level = 0	32	25 9	14 18 10 30	31 35 7 11	44 36	5 37 29

43

overflow page

Linear hashing

- $h_0(x) = x \bmod N \ (N = 4)$
- $h_1(x) = x \bmod (2*N)$
- Insert key 22 (10110)



Next = 3

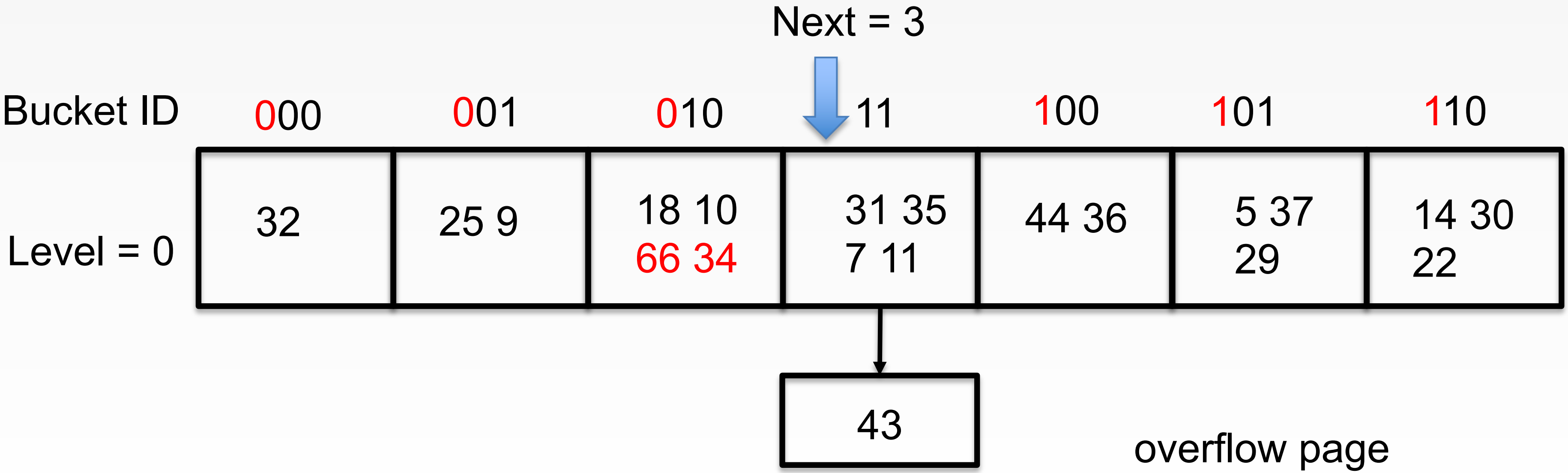
Bucket ID	000	001	010	11	100	101	110
Level = 0	32	25 9	18 10	31 35 7 11	44 36	5 37 29	14 30 22

43

overflow page

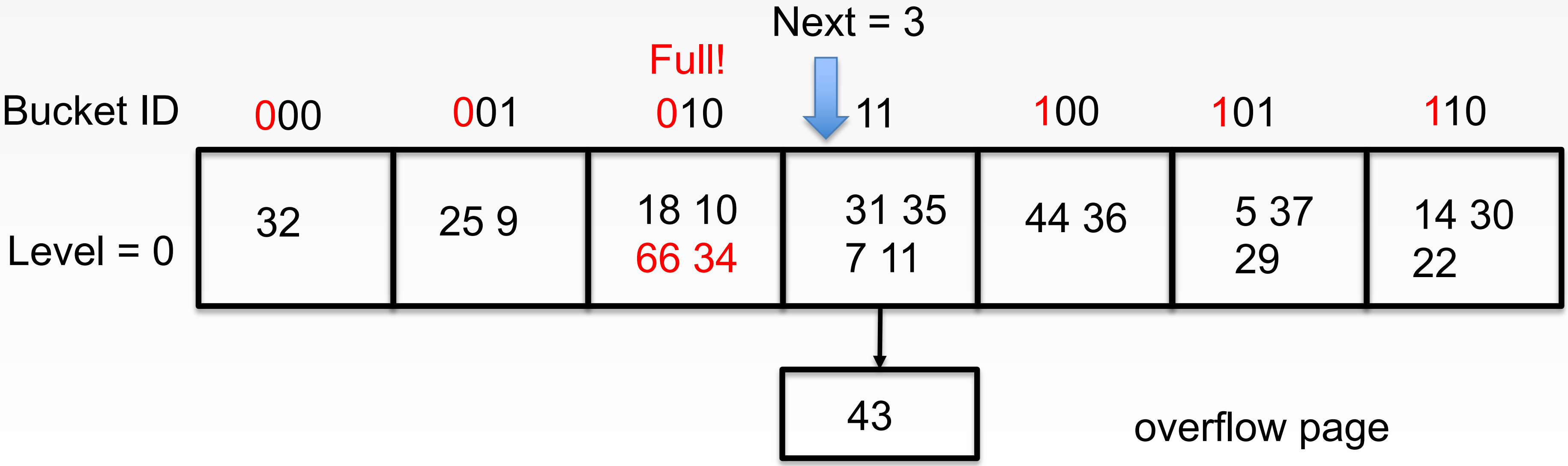
Linear hashing

- $h_0(x)$ = x mod N (N = 4)
- $h_1(x)$ = x mod (2*N)
- Insert key 66 (1000010) and 34 (100010)



Linear hashing

- $h_0(x) = x \bmod N \ (N = 4)$
- $h_1(x) = x \bmod (2*N)$
- Insert key 50 (110010)



Next = 3

Full!

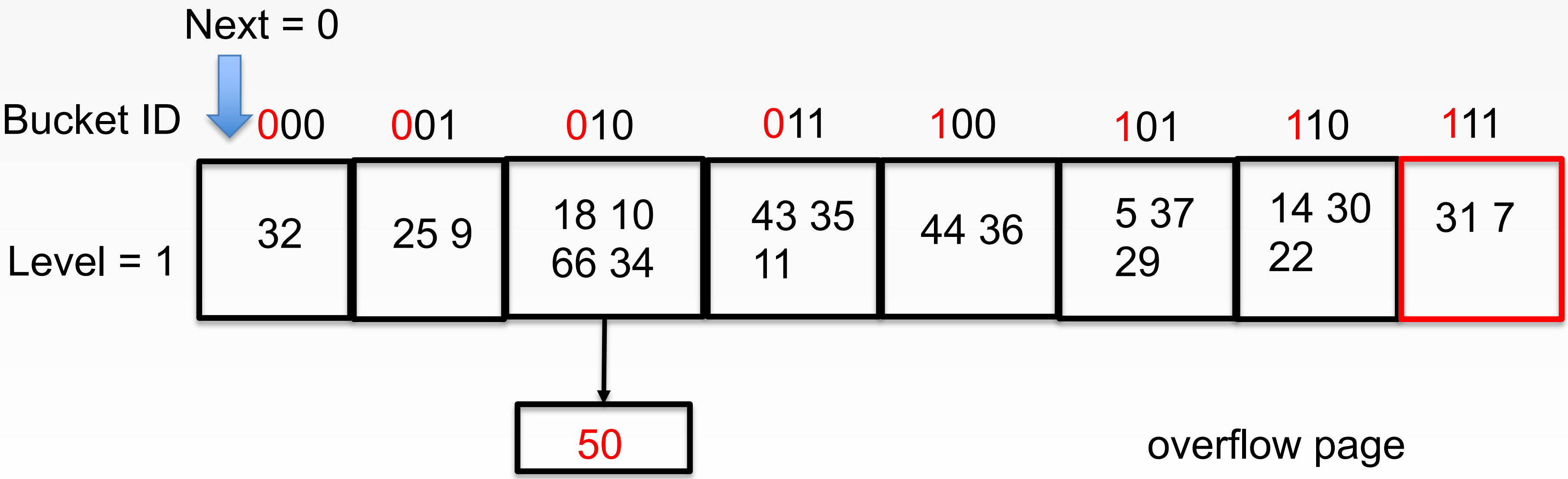
Bucket ID	000	001	010	11	100	101	110
Level = 0	32	25 9	18 10 66 34	31 35 7 11	44 36	5 37 29	14 30 22

43

overflow page

Linear hashing

- $h_0(x) = x \bmod N$ $(N = 4)$
- $h_1(x) = x \bmod (2*N)$
- Insert key 50 (110010)



Next = 0

Bucket ID	000	001	010	011	100	101	110	111
Level = 1	32	25 9	18 10 66 34	43 35 11	44 36	5 37 29	14 30 22	31 7

50

overflow page

Cost of Linear Hashing

- Search: One disk I/O or more when having overflow pages (average 1.2 I/Os)
- Insert and Delete: Two disk I/O (unless a split is triggered)
- Better performance

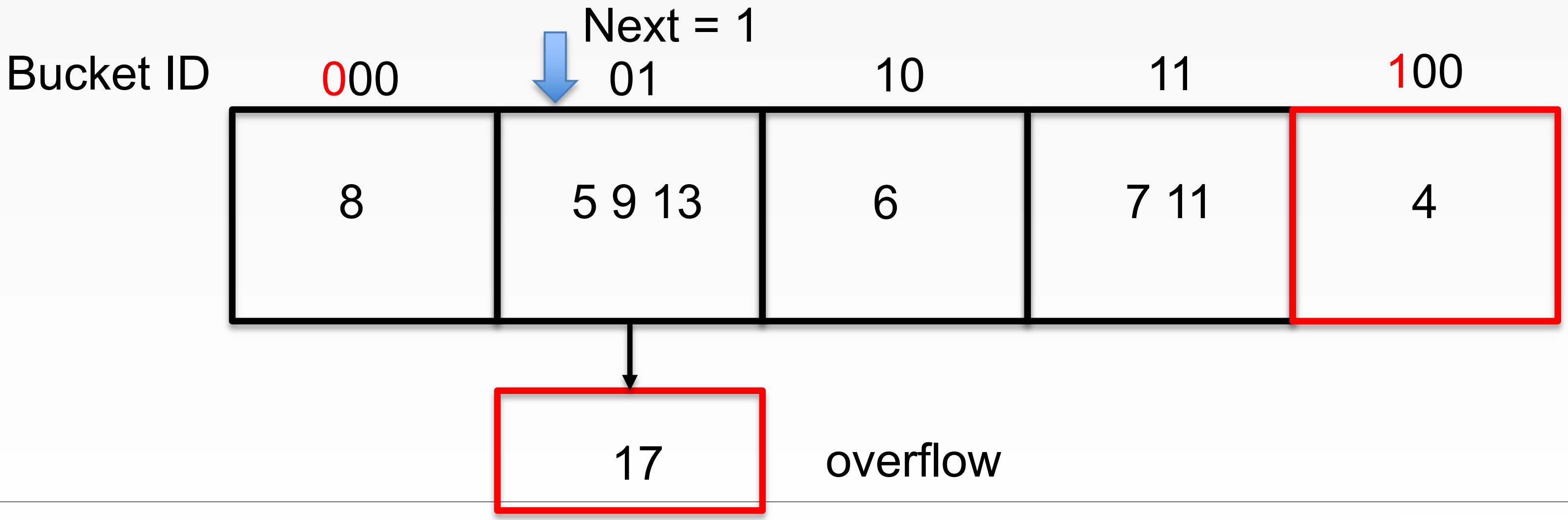
Example: Linear hashing

- h(x) = x mod N (N = 4)
- Assume capacity: 3 records per bucket
- Insert key 17 (10001)

Bucket ID	00	01	10	11
	4 8	5 9 13	6	7 11

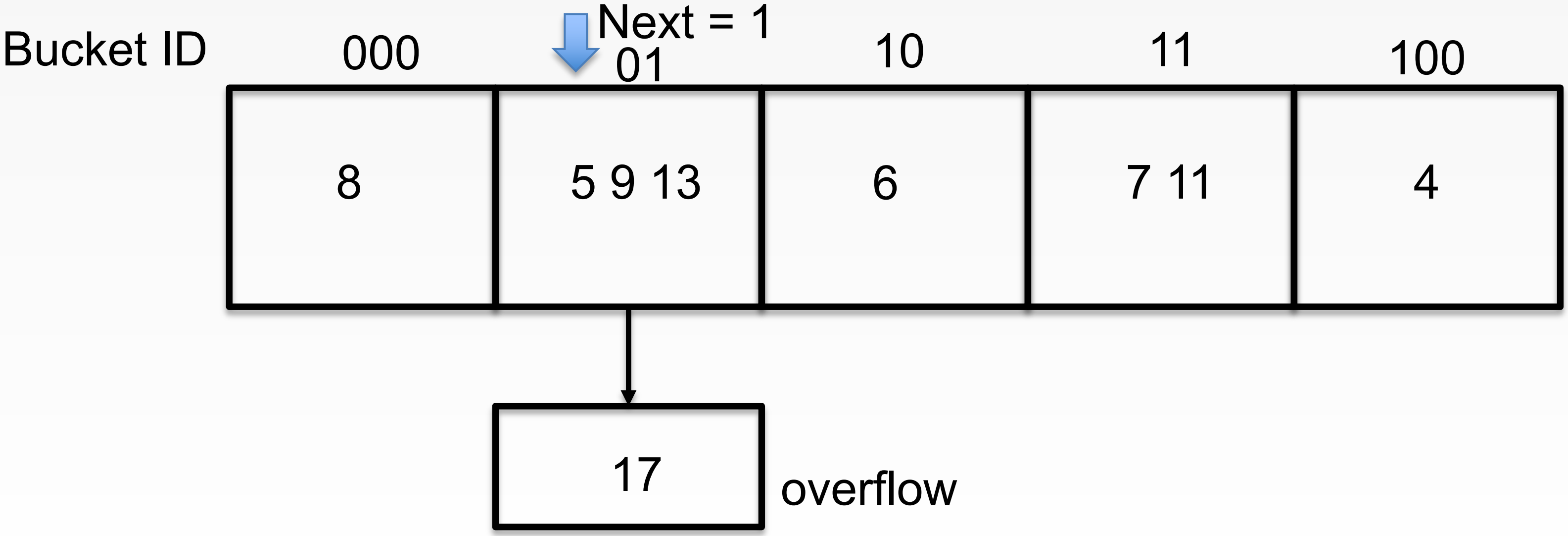
Example: Linear hashing – after split

- $h_0(x) = x \bmod N \ (N = 4)$
- $h_1(x) = x \bmod (2*N)$
- Insert key 17 (10001)



Linear hashing – searching

- $h_0(x)$ = x mod N (*for the un-split buckets*)
- $h_1(x)$ = x mod (2*N) *(for the split ones)*
- Q1: find key '6'? Q2: find key '4'? Q3: key '8'?



Hashing Summary

- B-trees and variants: in all DBMSs
- Hash indices: in some DBMSs
  - Hashing is useful for joins
- Hashing performs well on exact match queries
- B+ tree performs well on:
  - Search:
    - exact match queries
    - range queries
    - nearest-neighbor queries
  - Insertion and deletion
  - Smooth growing and shrinking

Sorting

- Two-way merge sort
- External merge sort
- Fine-tunings
- B+ trees for sorting

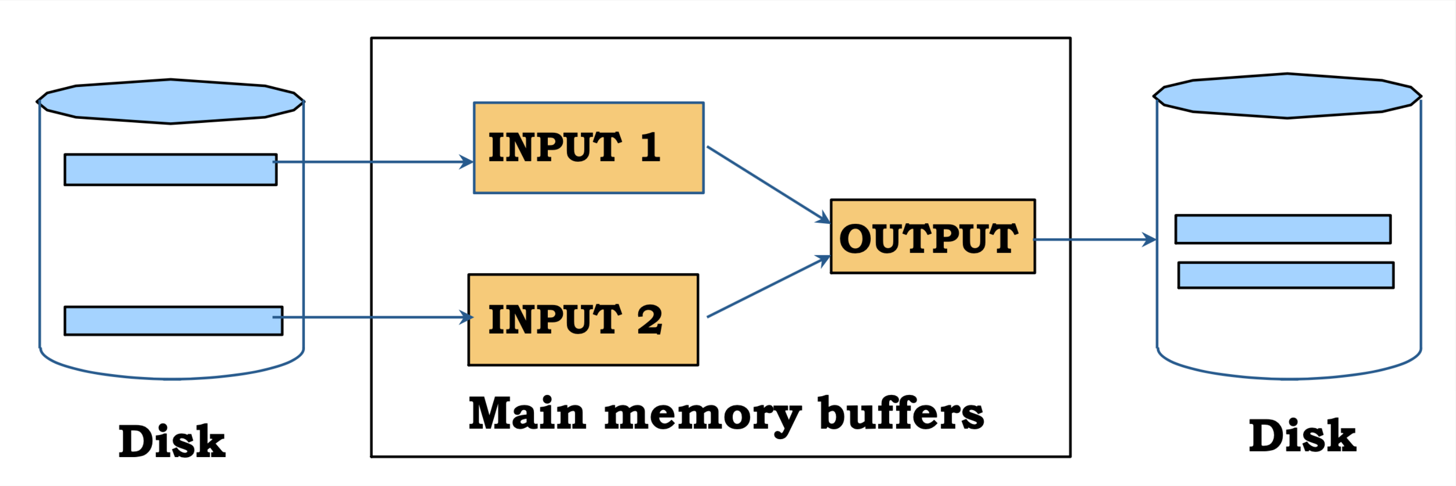
Why Sort?

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

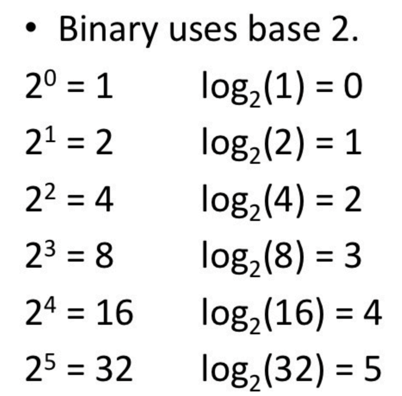
Two-Way Merge Sort

- Overview: break file into smaller subfiles, sort each subfile, and merge
- Utilizes only three (buffer) pages of main memory
- Pass 0: Read a page, sort it, write out
  - only one buffer page is used (a sorted **run**)
  - In-memory sorting technique. E.g., Quicksort
- Pass 1, 2, 3, ...k: Requires 3 buffer pages
  - merge pairs of **runs** into runs twice as long
  - three buffer pages used.
- Cost: $2N(\lceil log_2 N \rceil + 1)$ I/Os

Two-Way Merge Sort



- Cost: $2N(\lceil log_2 N \rceil + 1)$ I/Os
- N = 8, 2 * 8 * (3+1) = 64 I/Os

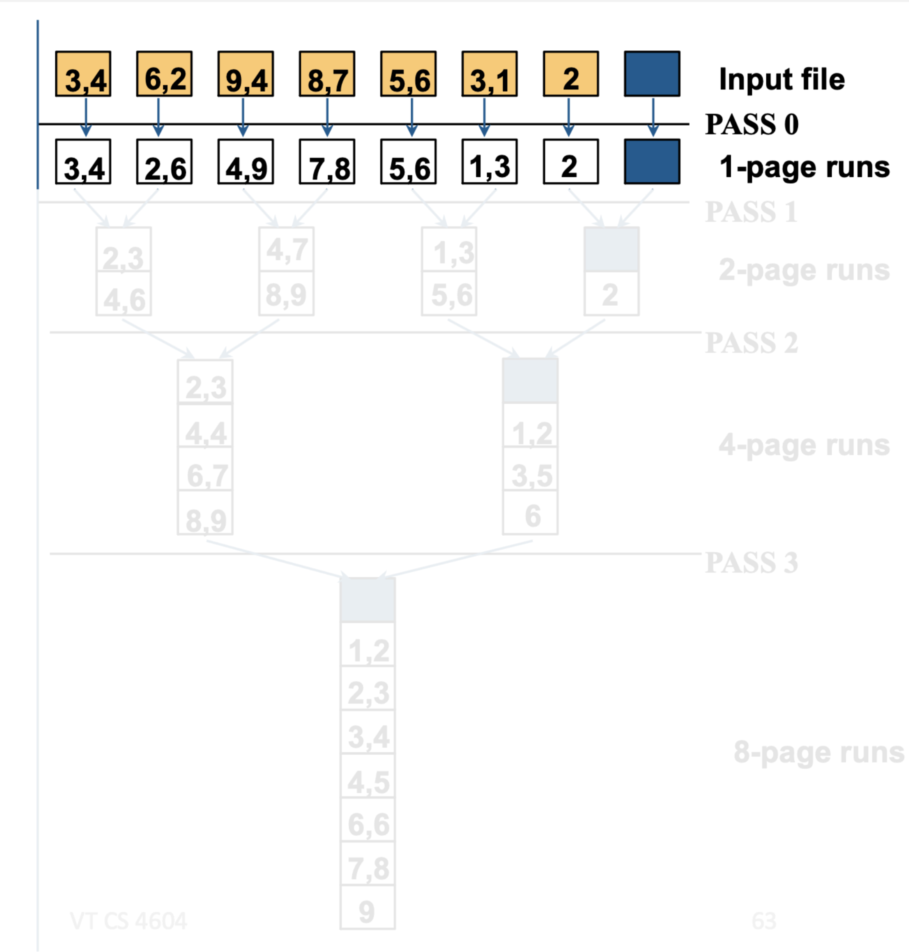


- Binary uses base 2.

$2^0 = 1$ $\log_2(1) = 0$
$2^1 = 2$ $\log_2(2) = 1$
$2^2 = 4$ $\log_2(4) = 2$
$2^3 = 8$ $\log_2(8) = 3$
$2^4 = 16$ $\log_2(16) = 4$
$2^5 = 32$ $\log_2(32) = 5$

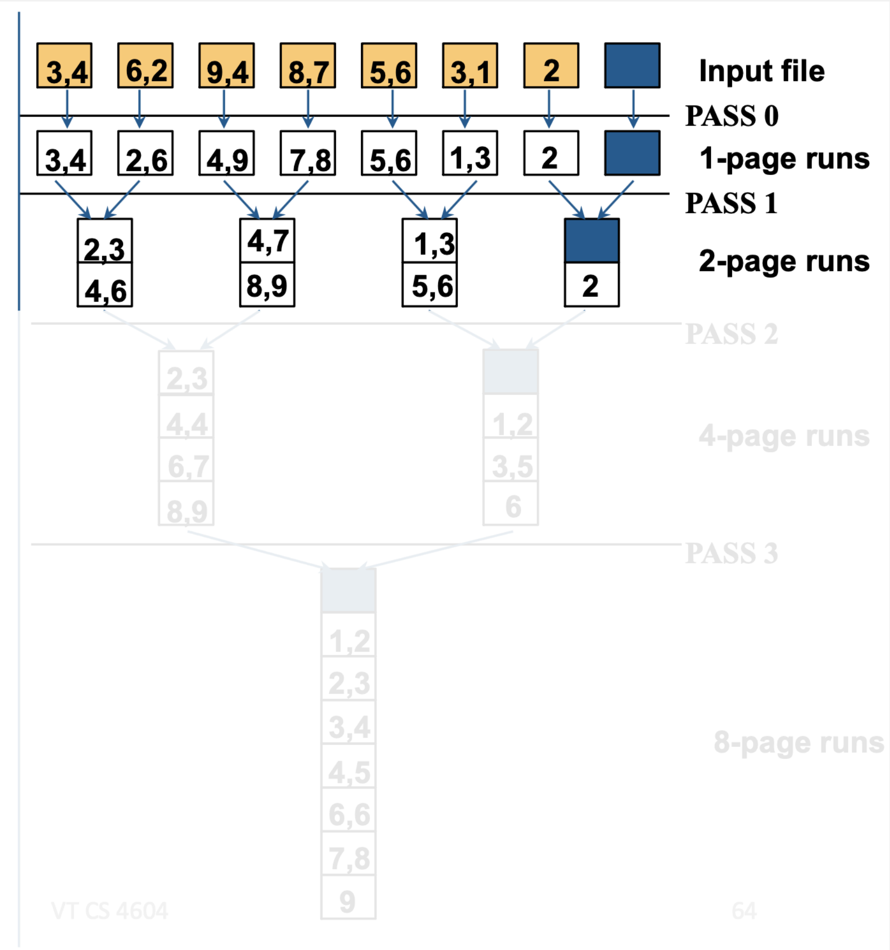
Two-Way Merge Sort

- Each pass we read and write each page in file



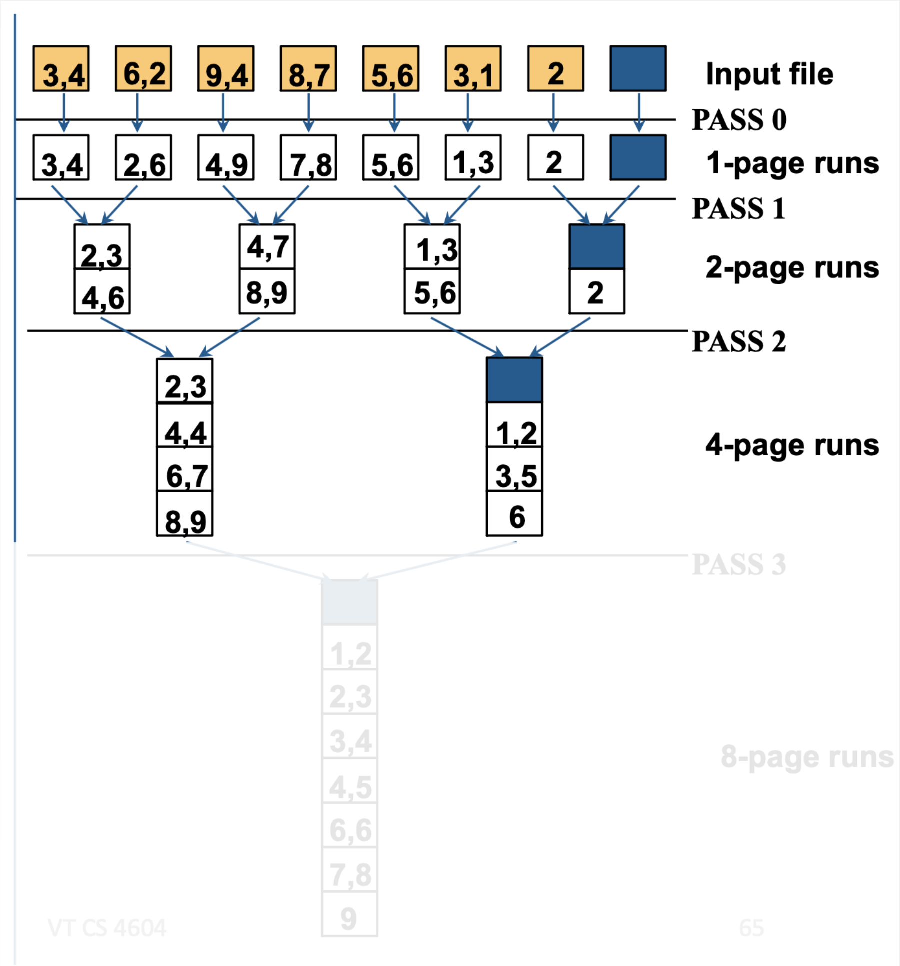
Two-Way Merge Sort

- Each pass we read and write each page in file



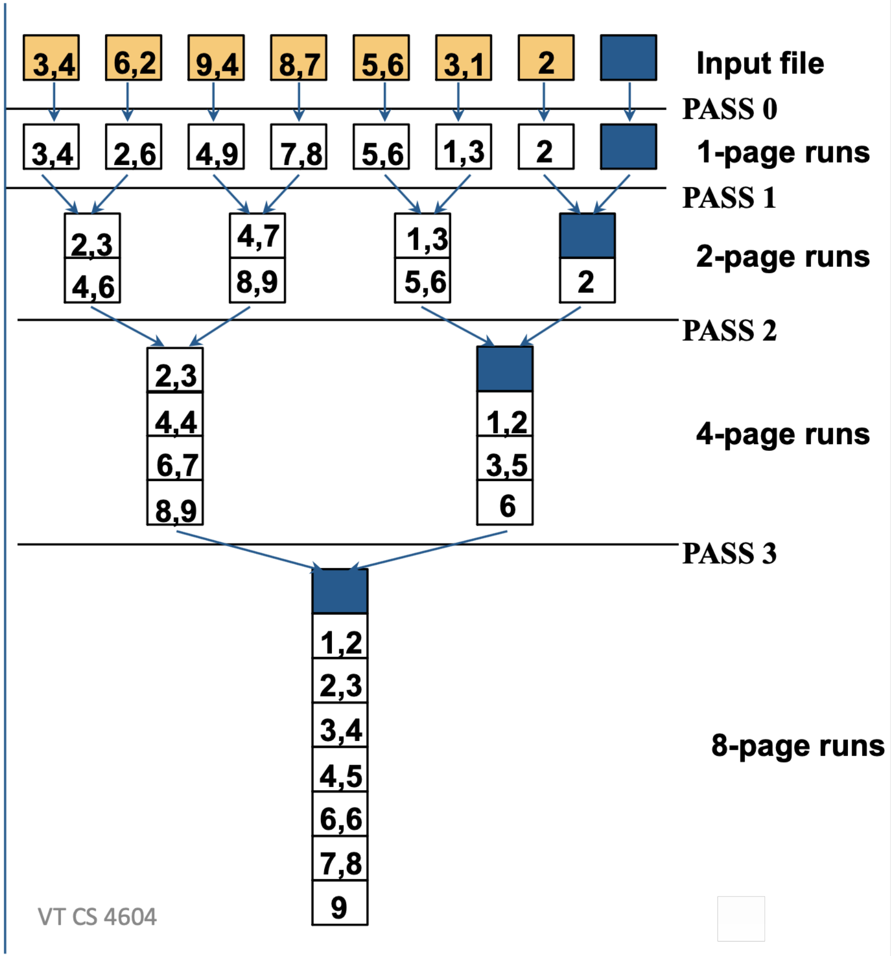
Two-Way Merge Sort

- Each pass we read and write each page in file



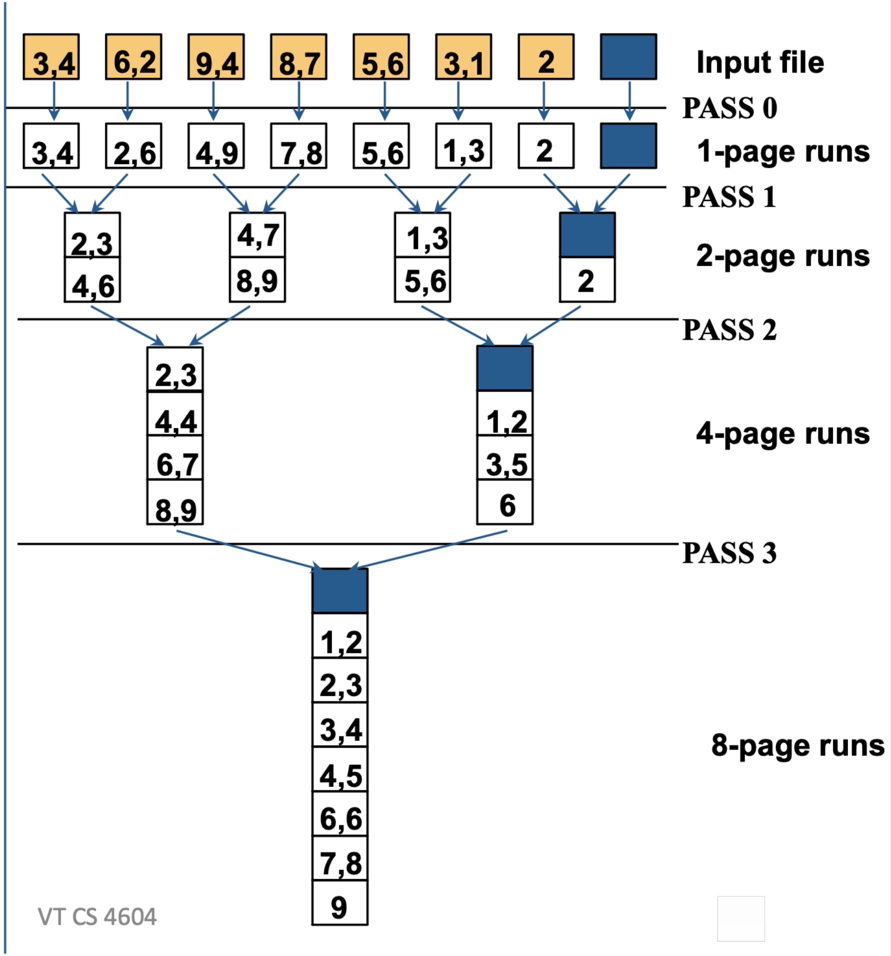
Two-Way Merge Sort

- Each pass we read and write each page in file



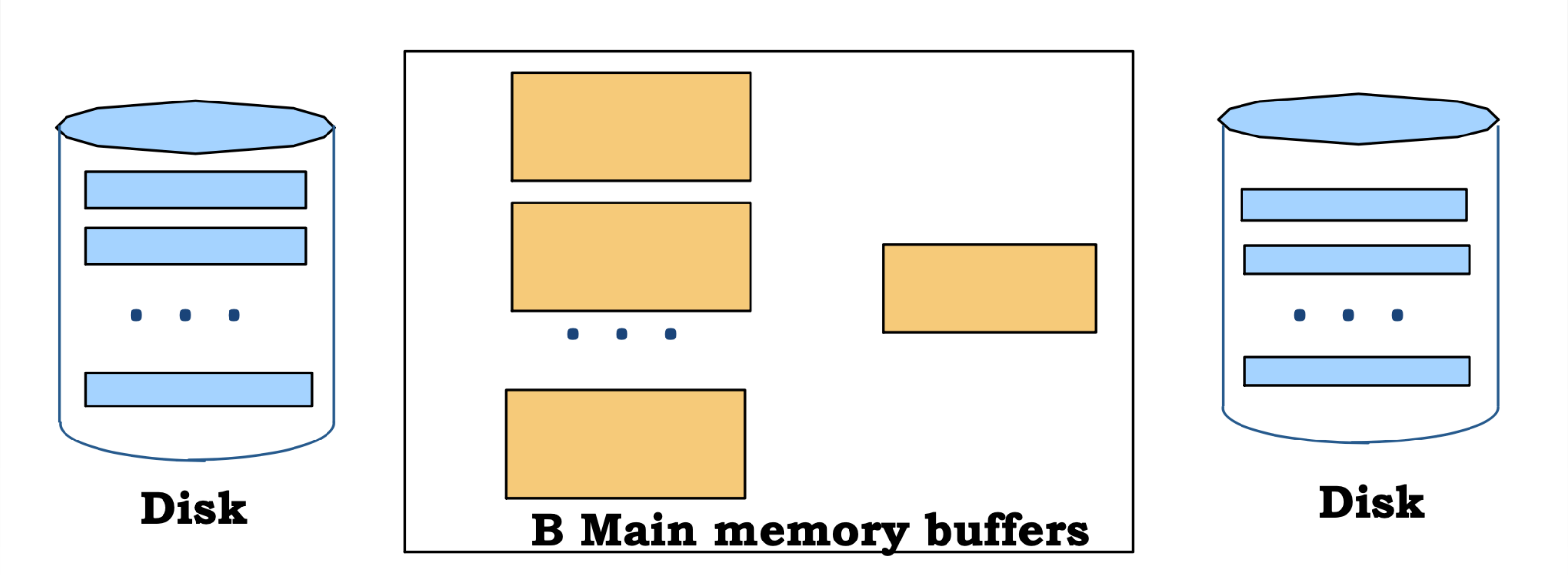
Two-Way Merge Sort

- Each pass we read and write each page in file
- N pages in the file: $\lceil log_2 N \rceil + 1$
- Total cost: $2N(\lceil log_2 N \rceil + 1)$ I/Os
- ***Divide and conquer:*** sort subfiles and merge



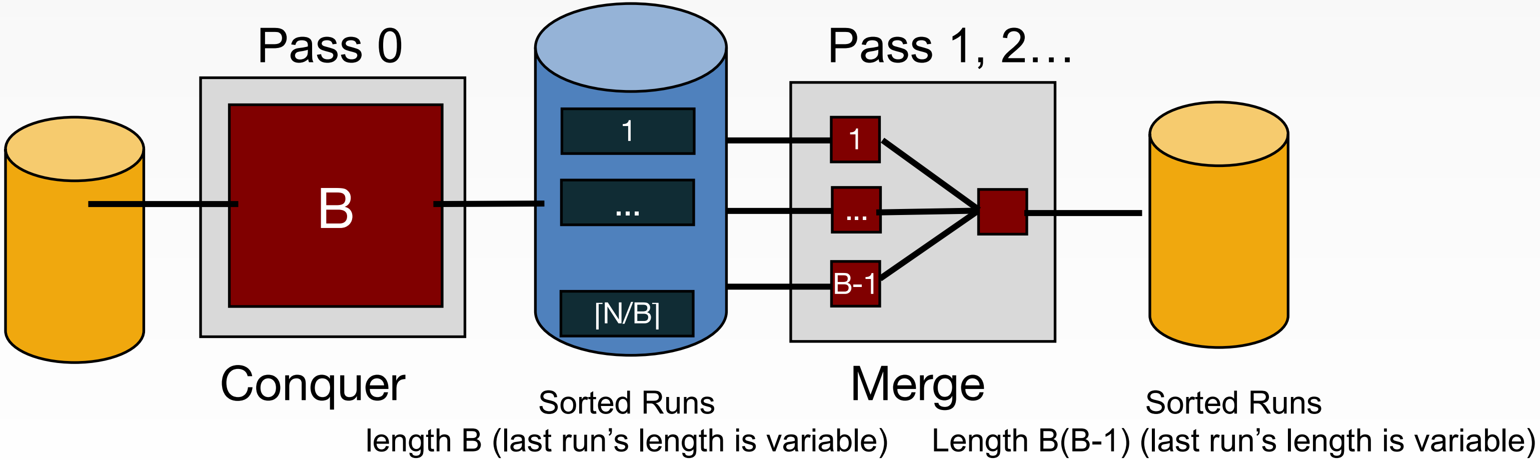
External Merge Sort

- Two-Way Merge Sort: We have more then three buffer pages available in main memory, we just use three. (underutilize)



External Merge Sort

- A large file with N pages needs to be sorted
- B buffer pages in memory
- 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

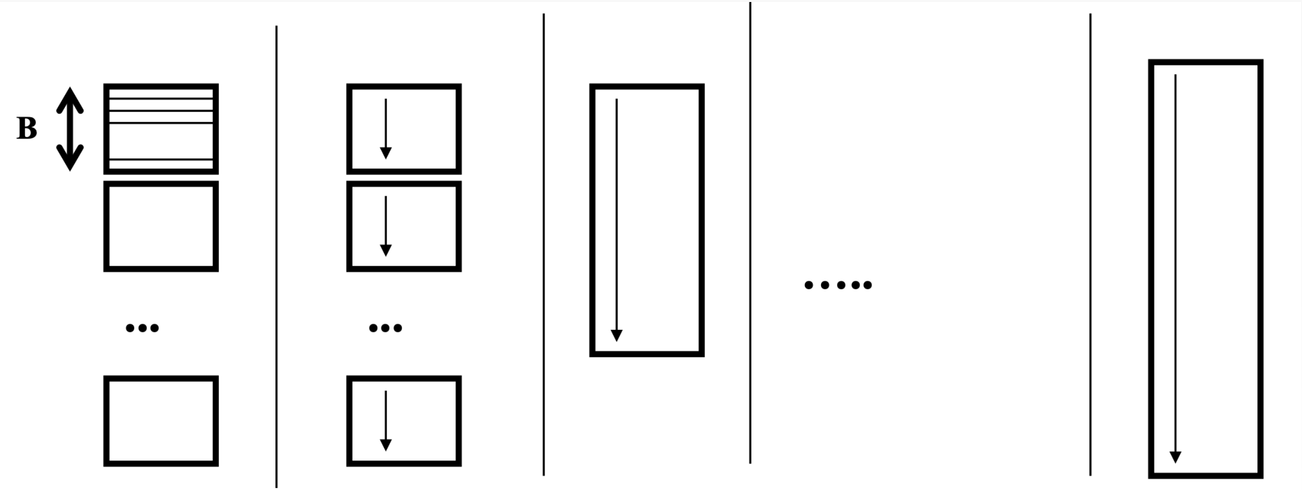


External merge sort

- Number of passes:

$$1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil = 1 + \lceil \log_{B-1} N1 \rceil, \; N1 = \lceil N / B \rceil$$

- Cost = 2N * (# of passes)



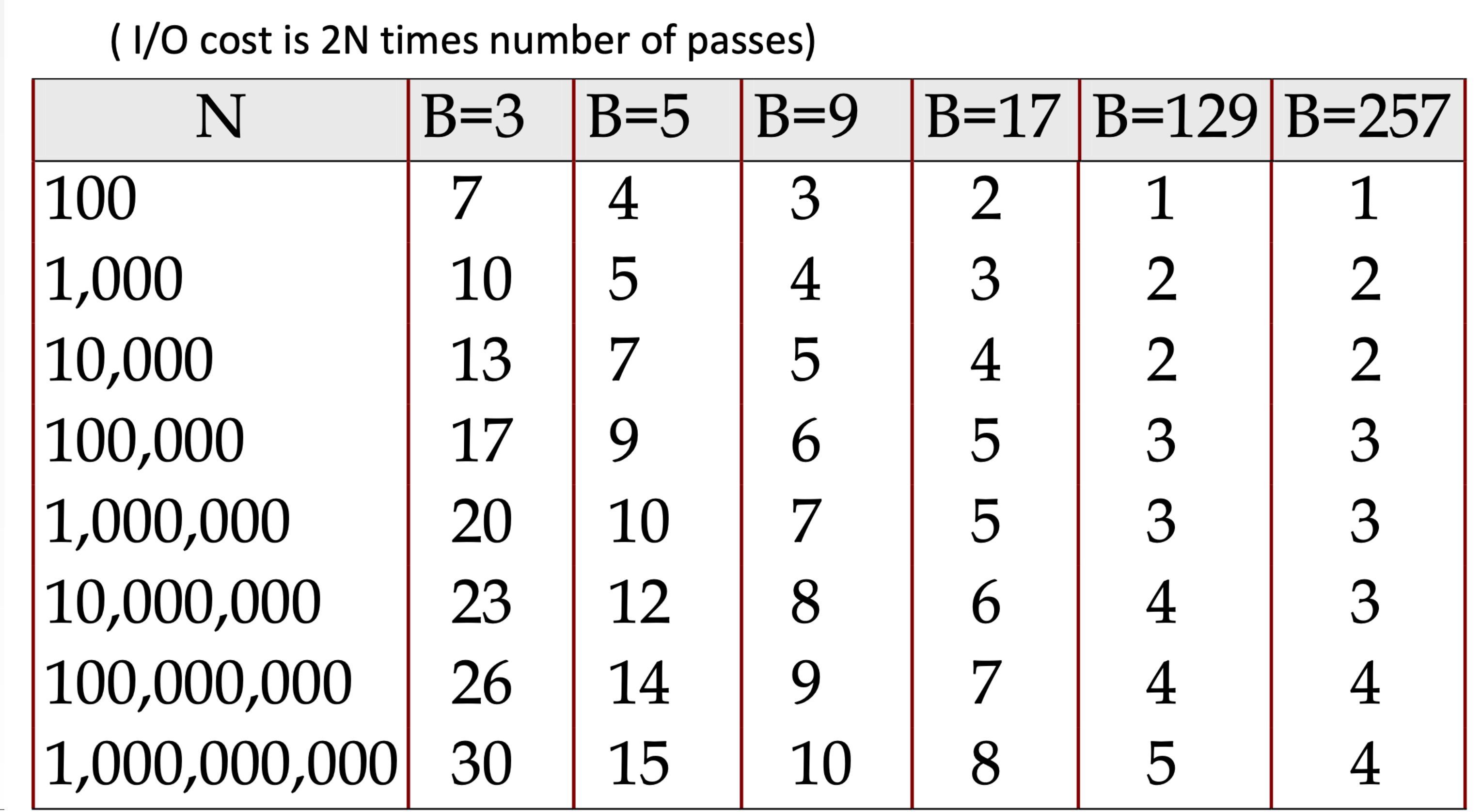
Cost of External Merge Sort

- Example: we have 5 buffer pages and want to sort a file with 108 pages
- Pass 0: $\lceil 108/5 \rceil$ = 22 sorted runs of 5 pages each
- Pass 1: $\lceil 22/4 \rceil$ = 6 sorted runs of 20 pages
- Pass 2: $\lceil 6/4 \rceil$ = 2 sorted runs, one run with 80 pages and one run with 28 pages
- Pass 3: Sorted file of 108 pages
- Formula check: $\lceil \log_4 22 \rceil$ = 3 ... + 1 → 4 passes

Cost of External Merge Sort

- Each pass we read and write 108 pages
- Total cost: 2 * 108 * 4 = 864 I/Os
- $N1 = \lceil N / B \rceil = \lceil 108/5 \rceil = 22$
- $B = 5$
- $2N * (1 + \lceil \log_{B-1} N1 \rceil) = 2 * 108 * 4$

Number of Passes of External Sort

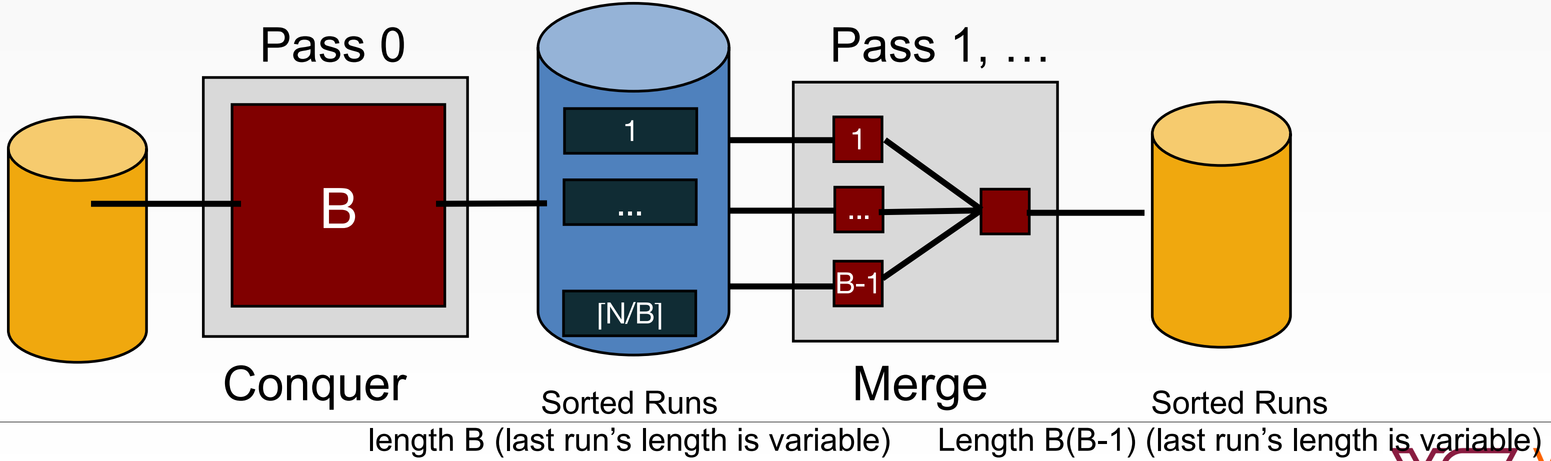


(I/O cost is 2N times number of passes)

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

Memory Requirement for External Sorting

- How big of a table can we sort in two passes?
  - Each “sorted run” after Phase 0 is of size B
  - Can merge up to B-1 sorted runs in Phase 1
- Answer: B(B-1).
  - Sort N pages of data in about $B = \sqrt{N}$ space

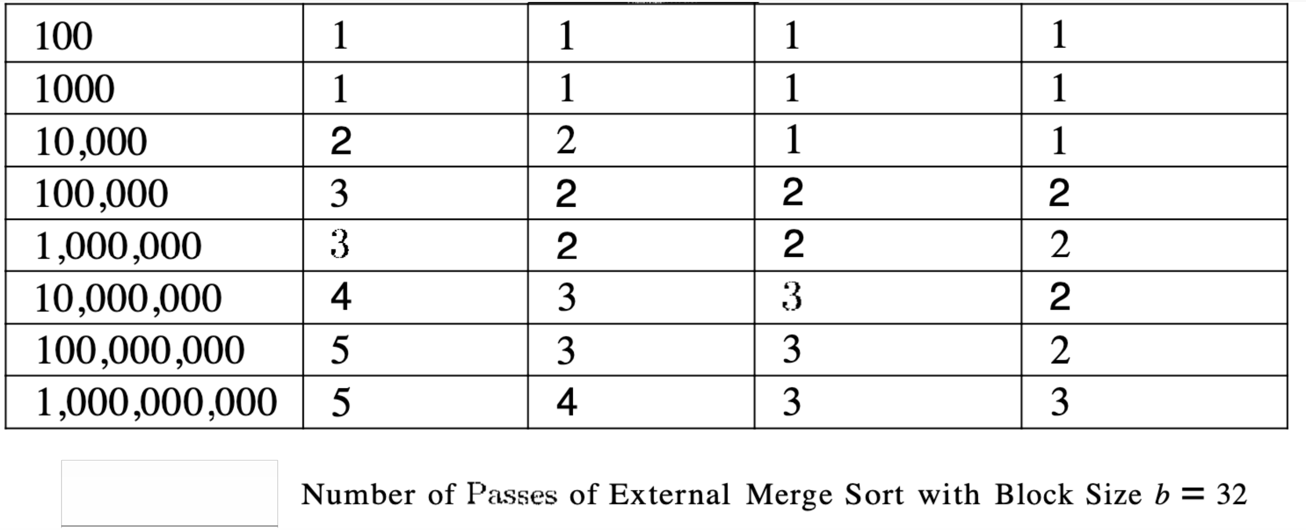


Cost Metric

- We assumed random disk access (# of page I/Os)
- Blocked I/O: a single request to read(or write) sequentially
- Also, double buffering: Keep the CPU busy while an I/O op is in progress

Blocked I/O

- $\left\lfloor \frac{B-b}{b} \right\rfloor$ runs
- 10 buffer pages:
  - 9 runs (one buffer blocks)
  - 4 runs (two buffer blocks)

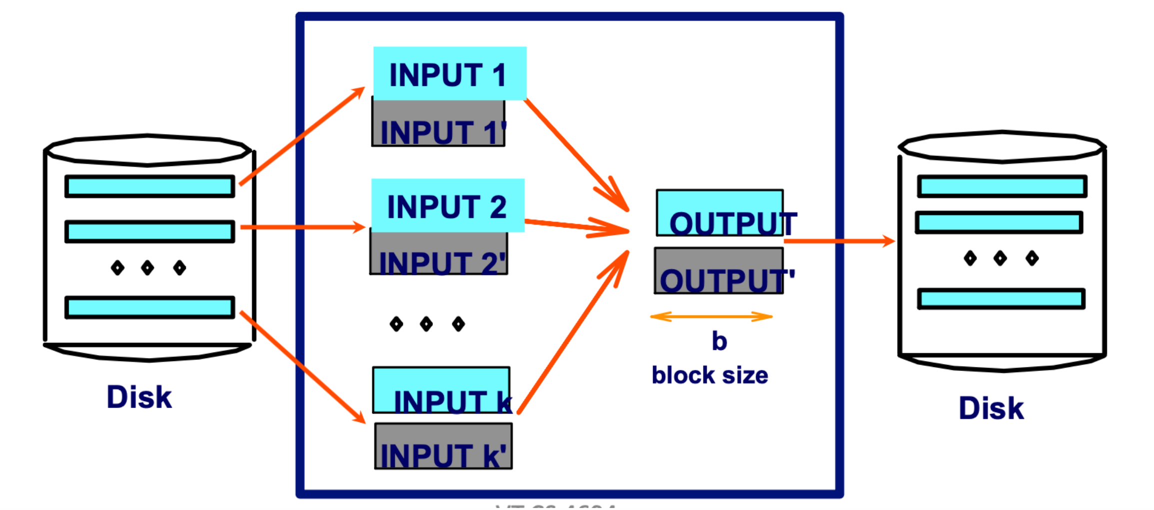


N	B = 1000	B = 5000	B = 10,000	B = 50,000
100	1	1	1	1
1000	1	1	1	1
10,000	2	2	1	1
100,000	3	2	2	2
1,000,000	3	2	2	2
10,000,000	4	3	3	2
100,000,000	5	3	3	2
1,000,000,000	5	4	3	3

Number of Passes of External Merge Sort with Block Size $b = 32$

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

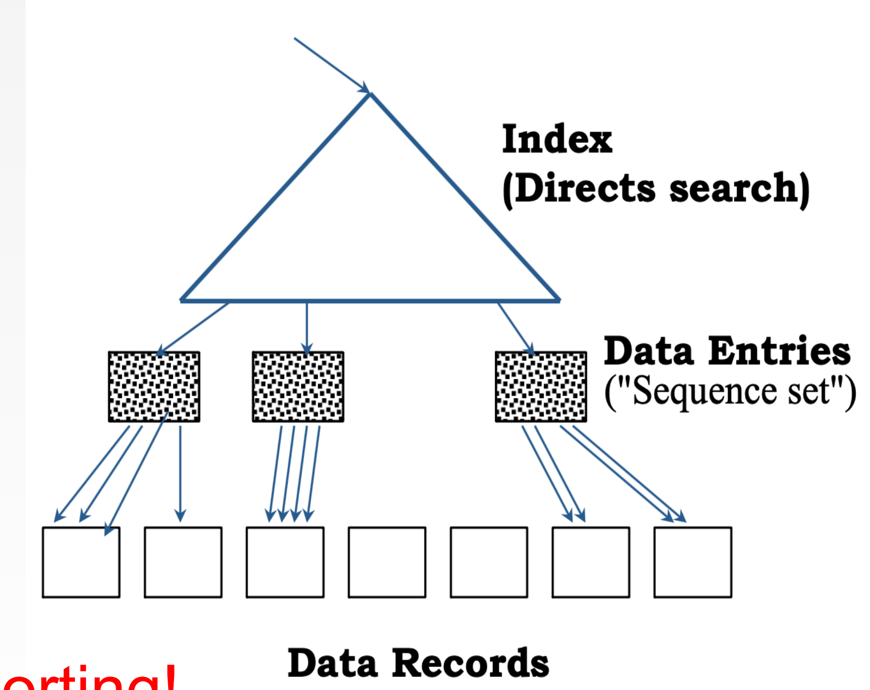
- Quicksort is a fast way to sort in memory
- Scenario: Table to be sorted has B+ tree index on sorting column(s).
- Idea: Can retrieve records in order by traversing leaf pages.
- Is this a good idea?
- Cases to consider:
  - B+ tree is clustered
  - B+ tree is not clustered

Good idea!

Could be a very bad idea!

Clustered B+ Tree Used for Sorting

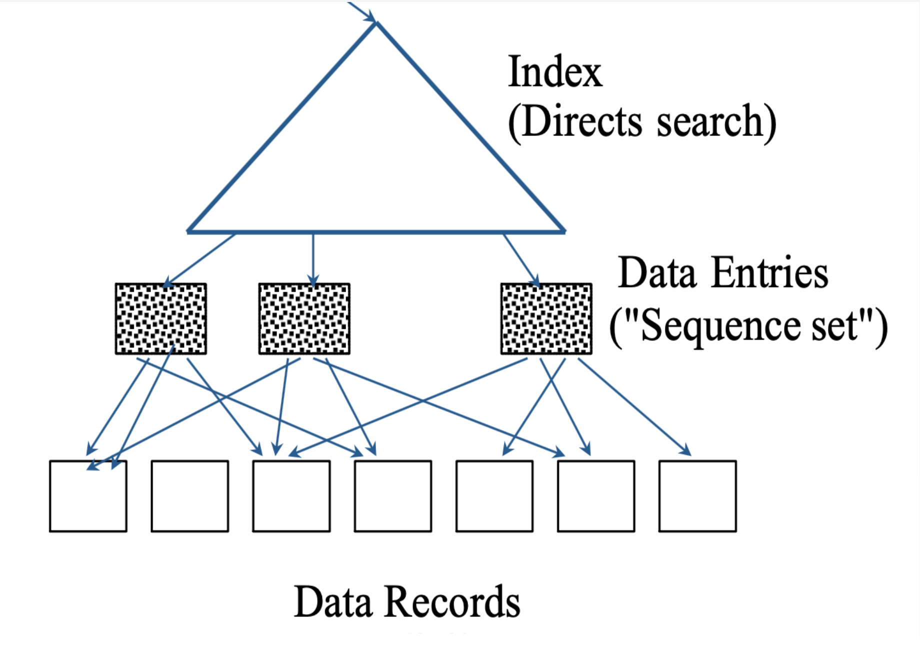
- Cost: root to the left-most leaf, then retrieve all leaf page
  - Use alternative 1: Actual data record (with key value **k**)



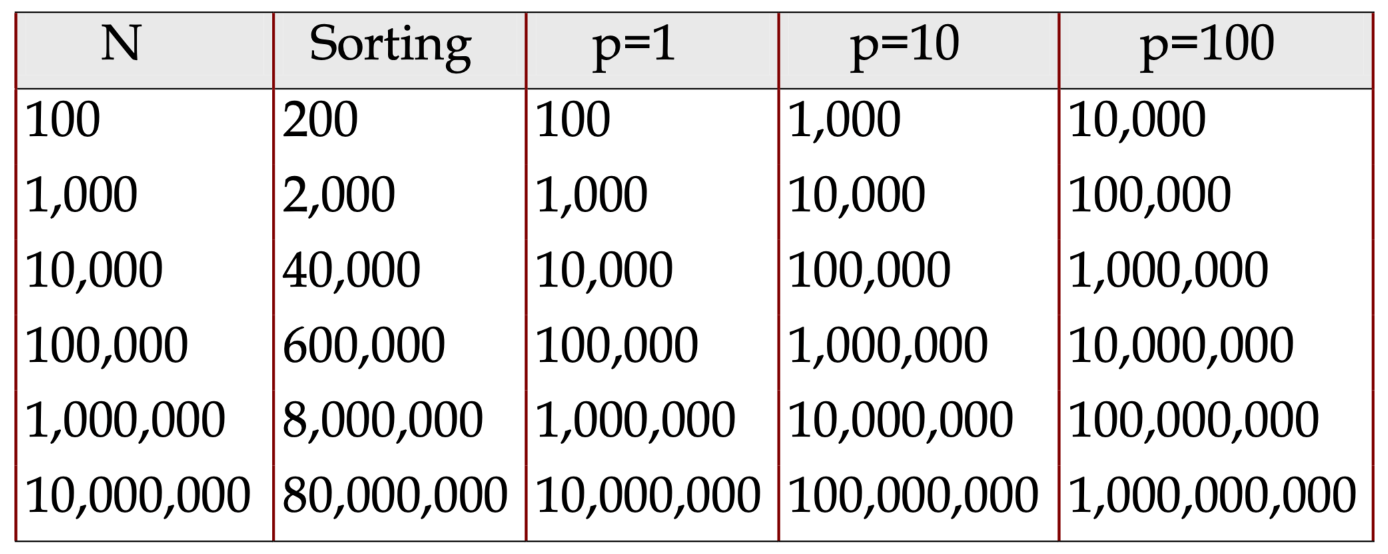
Always better than external sorting!

Unclustered B+ Tree Used for Sorting

- Use alternative (2) for data entries <**k**, rid of matching data record>
- 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.

Sorting 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 B+ tree is usually very bad

Reading and Next Class

- Hashing and Sorting: Ch 11, Ch 13
- Next: Query Processing: Ch 12, Ch 14