Modern Database Systems
Lecture 2

Aristides Gionis
Michael Mathioudakis

T.A.: Orestis Kostakis

Spring 2016

logistics

- **assignment 1** is up
- **cowbook** available at learning center beta, otakaari 1 x
- if you do not have **access to the lab**, provide Aalto username or email **today!!**
- if you do not have **access at mycourses**, i will post material (slides and assignments) also at http://michalis.co/moderndb/

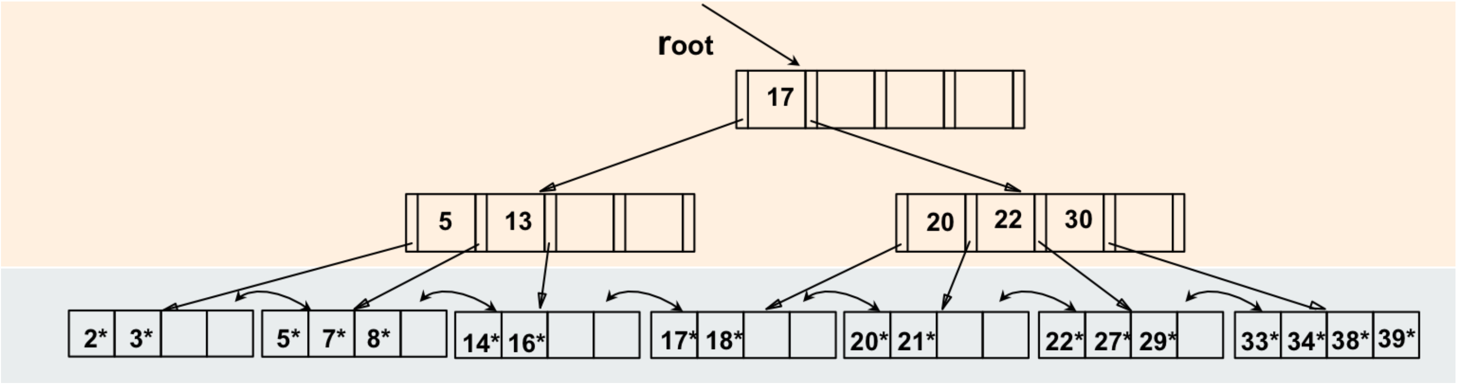
in this lecture...

b+ trees and hash-based indexing
external sorting
join algorithms
query optimization

b+ trees

b+ trees

non-leaf nodes
index entries
used to direct search



leaf nodes
contain **data-entries**
sequentially linked

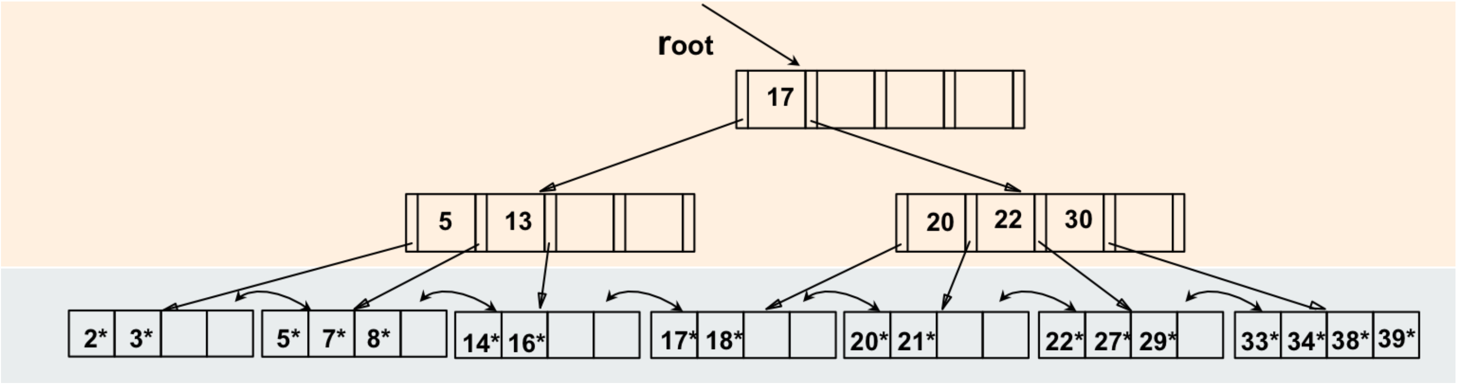
each node stored in one page
data entries can be *any* one of the three *alternative* types
type 1: data records; type 2: (k, rid); type 3: (k, rid*s*)
at least 50% capacity - except for root!

in the examples that follow...
alternative 2 is used
all nodes have between **d** and **2d** key entries
d is the **order** of the tree

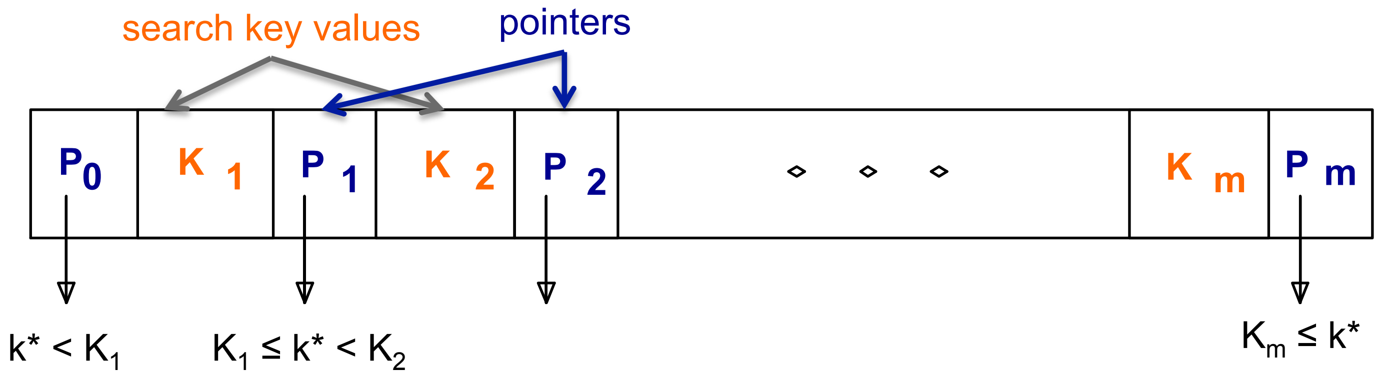
b+ trees

non-leaf nodes
index entries
used to direct search

leaf nodes
contain **data-entries**
sequentially linked



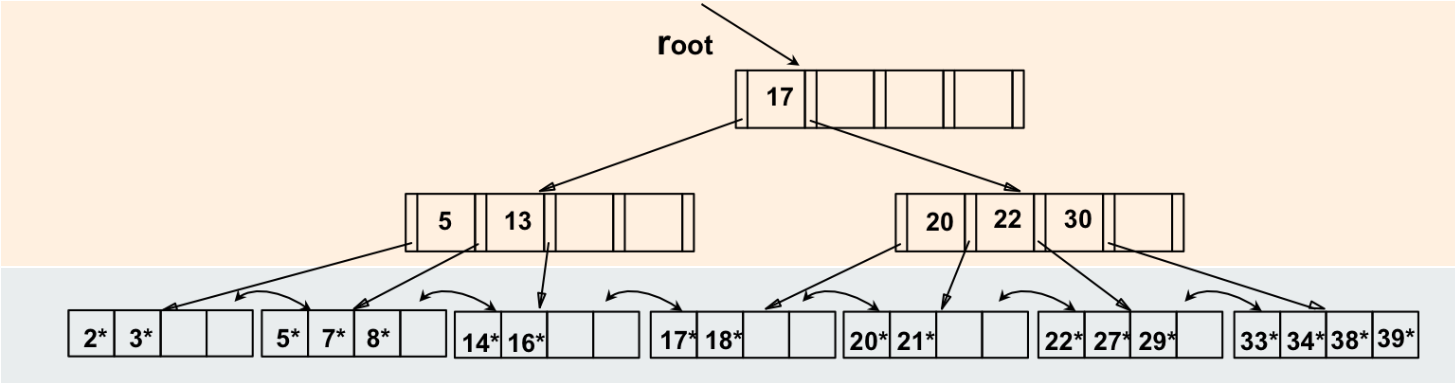
closer look at non-leaf nodes



b+ trees

non-leaf nodes
index entries
used to direct search

leaf nodes
contain **data-entries**
sequentially linked



most widely used index

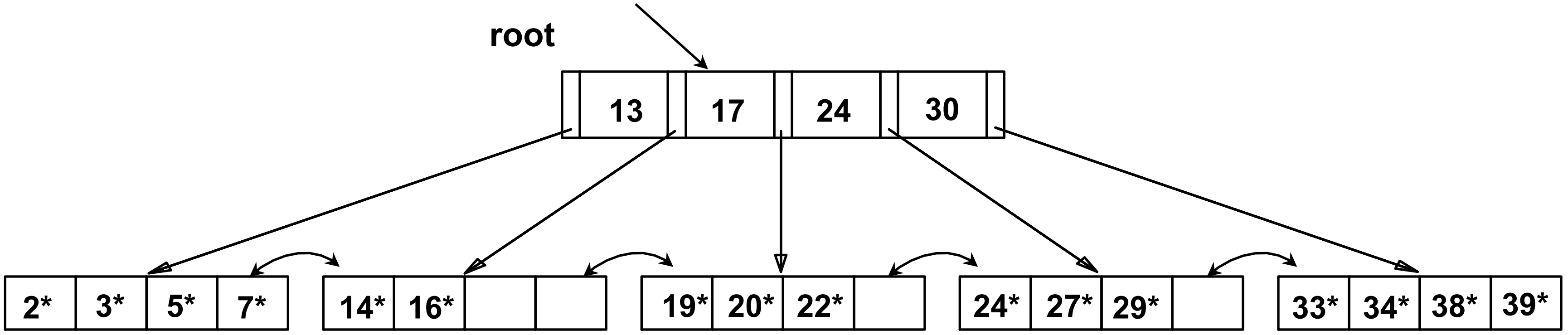
search and updates at $\log_F N$ cost (cost = pages I/O)
F = fanout (num of pointers per index node); N = num of leaf pages

efficient equality **and** range queries

example b+ tree - search

search begins at root, and key comparisons direct it to a leaf

search for 5*; search for all data entries >= 24*



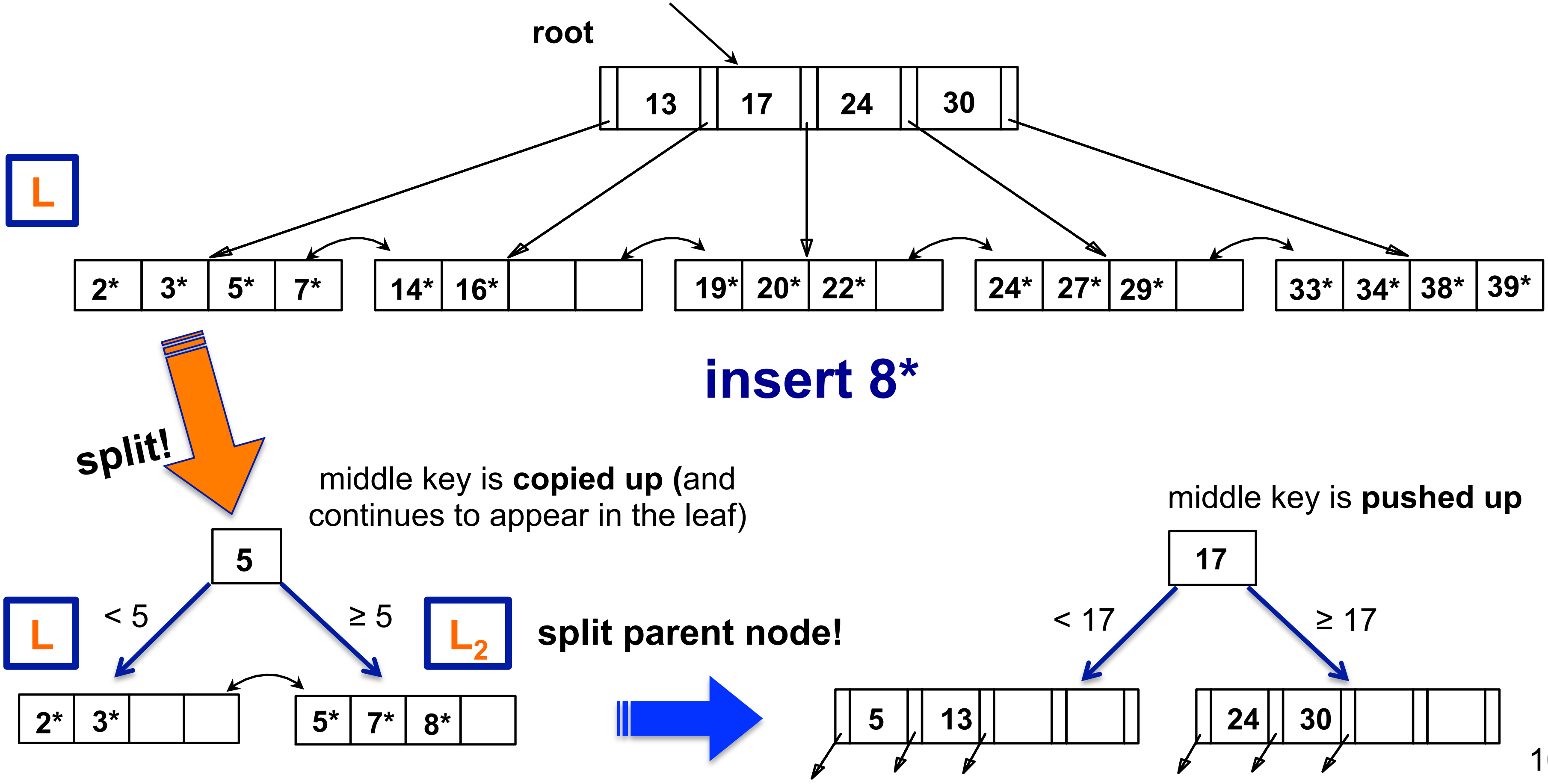
inserting a data entry

1. find correct leaf **L**
2. place data entry onto **L**
   a. if L has enough space, done!
   b. else must split **L** into **L** and $\mathbf{L_2}$
      - redistribute entries evenly
      - **copy up** the middle key to parent of **L**
      - insert entry pointing to $\mathbf{L_2}$ to parent of **L**

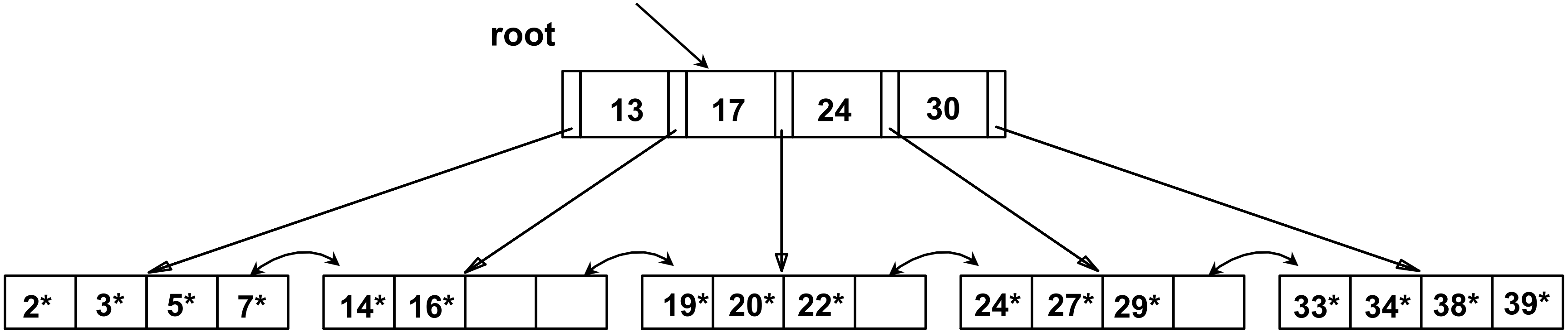
the above happens recursively
when **index nodes** are split, **push up** middle key

splits grow the tree
root split increases height

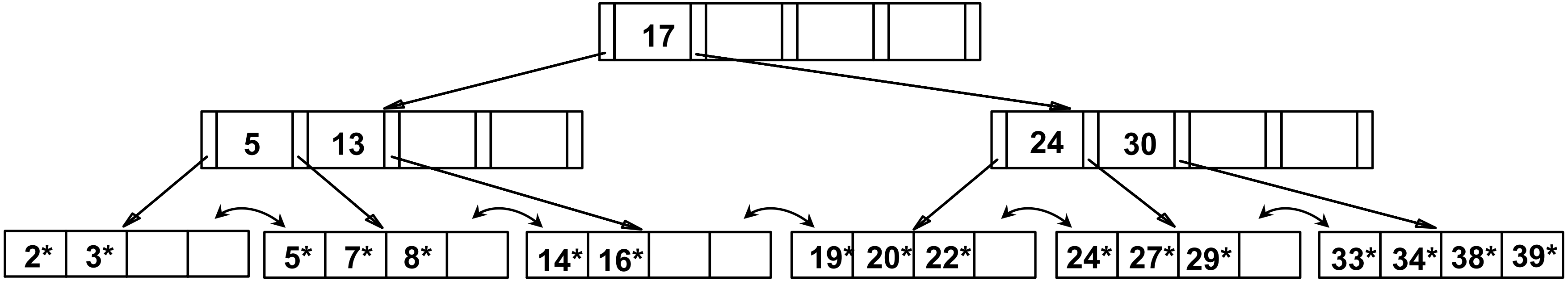
example b+ tree



example b+ tree



insert 8*



deleting a data entry

inverse of insertion

deletion		insertion
remove data entry	vs	add data entry
when nodes are less than half-full		when nodes overflow
re-distribute entries & (maybe) merge nodes	vs	split nodes & re-distribute entries

not always used!
data tend to grow

b+ trees in practice

typical order d = 100, fill-factor = 67%

average fan-out 133

typical capacities:

for height 4: 133^4 = 312,900,700 records

for height 3: 133^3 = 2,352,637 records

can often hold top levels in main memory

level 1: 1 page = 8KBytes

level 2: 133 pages = 1MByte

level 3: 17,689 pages = 133 MBytes

hash-based indexes

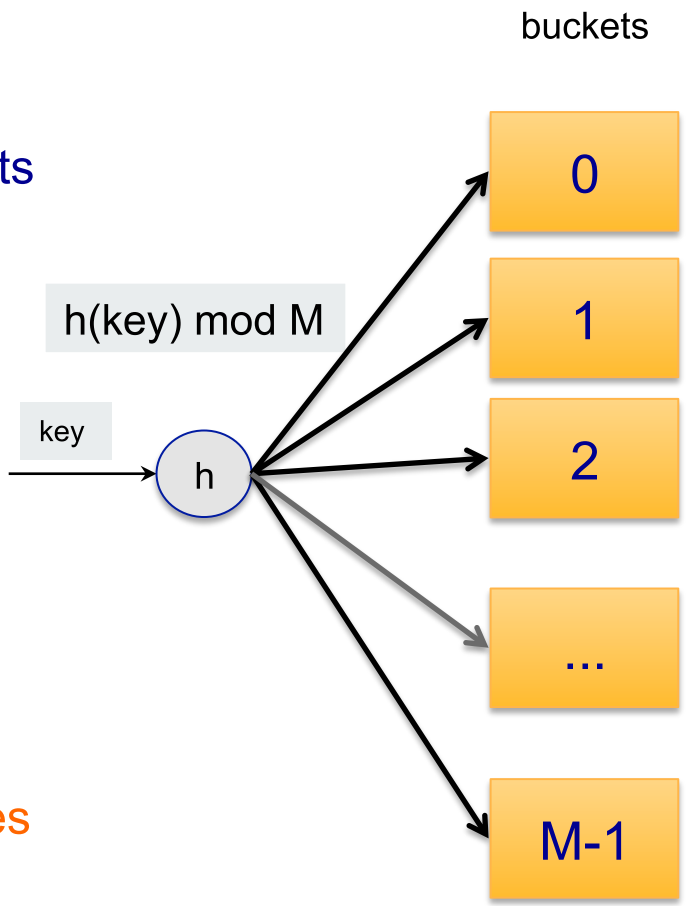
hash-based index

data entries organized in M buckets
bucket = a collection of pages

the data entry for record
with search key value key
is assigned to bucket
h(key) mod M

hash function h(key)
e.g., h(key) = α key + β

the index supports equality queries
does ***not*** support range queries
static and dynamic variants exist

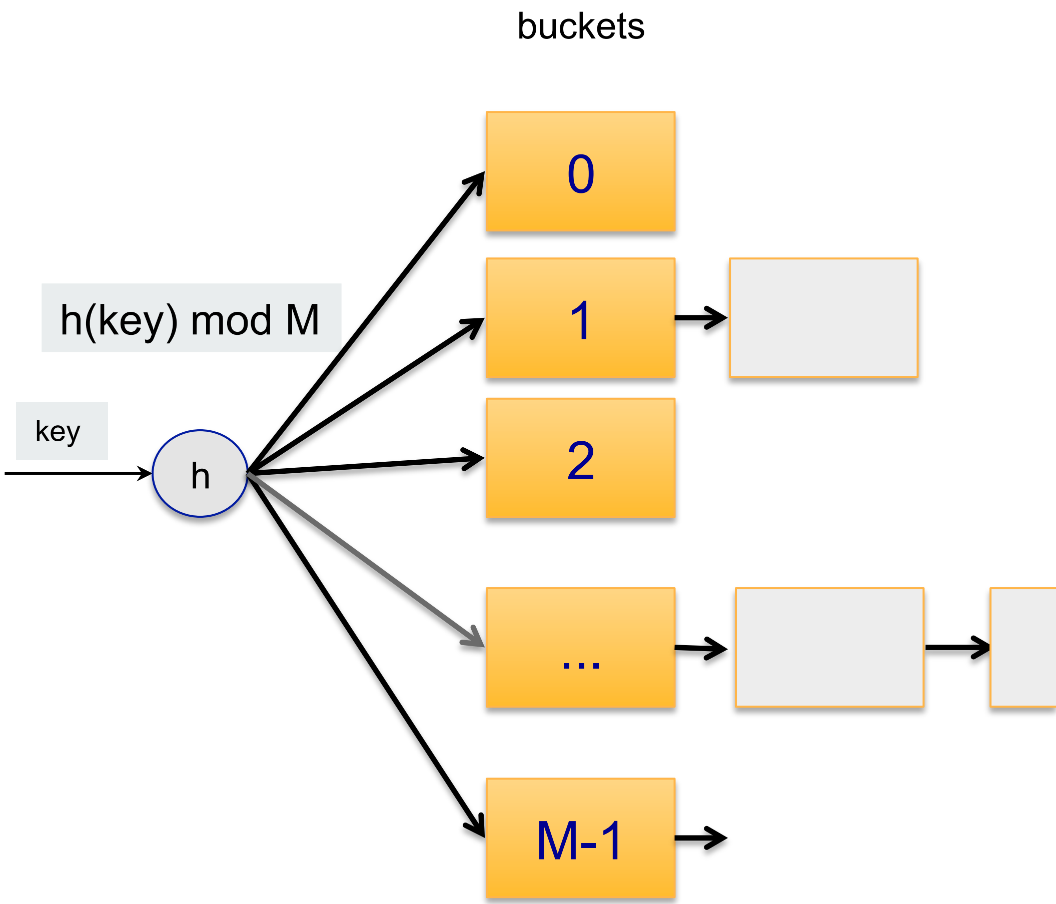


static hashing

number of buckets is fixed

start with one page per bucket

allocated sequentially, never de-allocated
can use overflow pages



static hashing

drawback

long overflow chains can degrade performance

dynamic hashing techniques

adapt index to data size

extendible and *linear hashing*

extendible hashing

problem: bucket becomes full

one solution

double the number of buckets...

...and redistribute data entries

however

reading and re-writing all buckets is expensive

better idea:

use **directory of pointers** to buckets

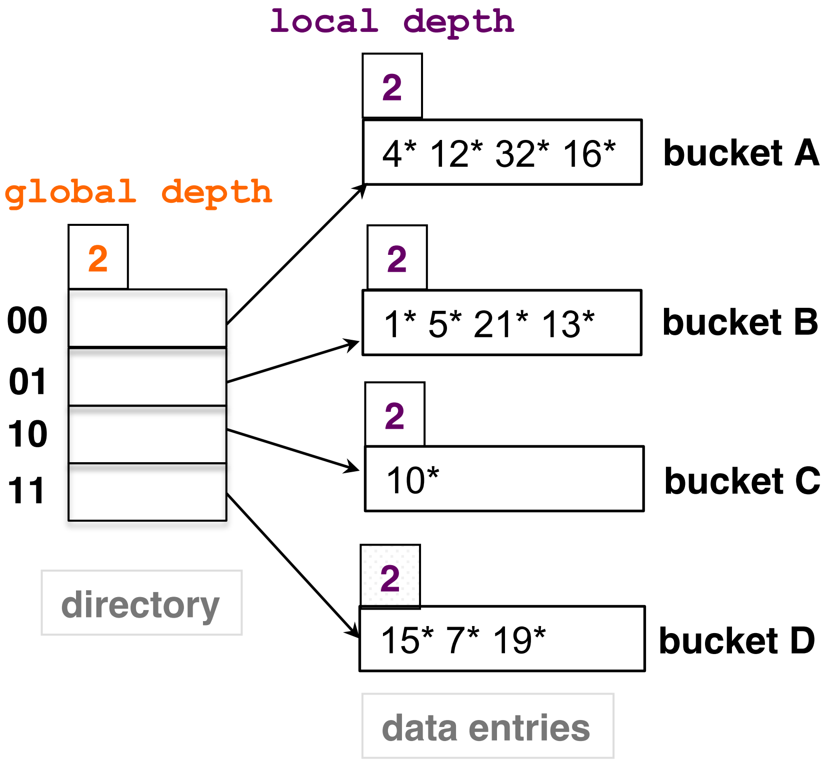
double number of ‘logical’ buckets…

but split ***‘physically’*** only the ***overflown*** bucket

directory much smaller than data entry pages - good!

no overflow pages - good!

example



directory is array of size $M = 4 = 2^2$

to find bucket for *r*, take last 2 # bits of **h**(*r*)

h(***r***) = key

e.g., if **h**(*r*) = 5 = binary 101

it is in bucket pointed to by 01

global depth = 2

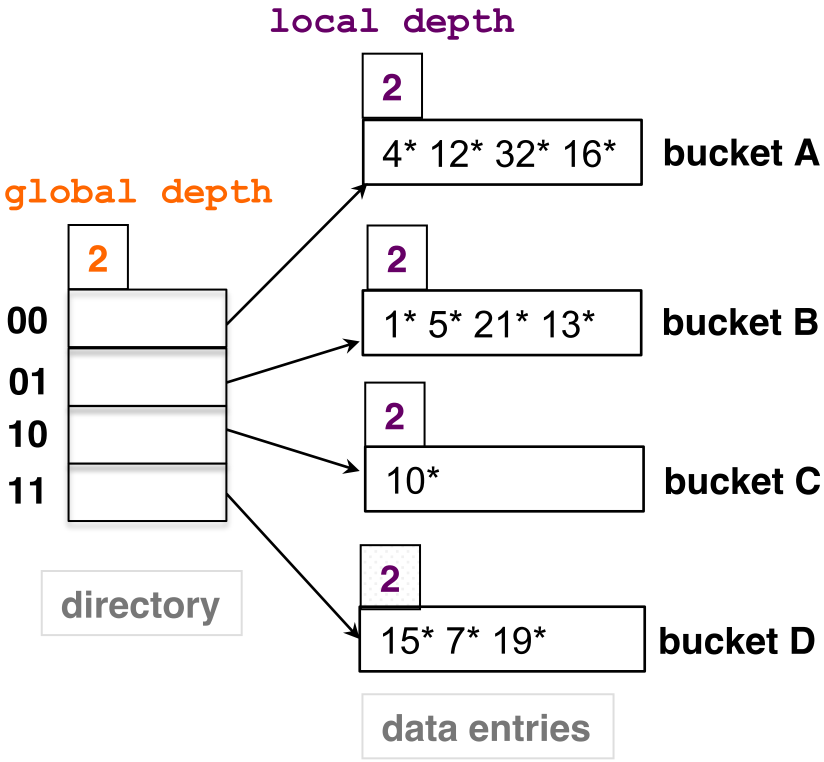
= min. bits enough to enumerate buckets

local depth

= min bits to identify individual bucket

= 2

insertion



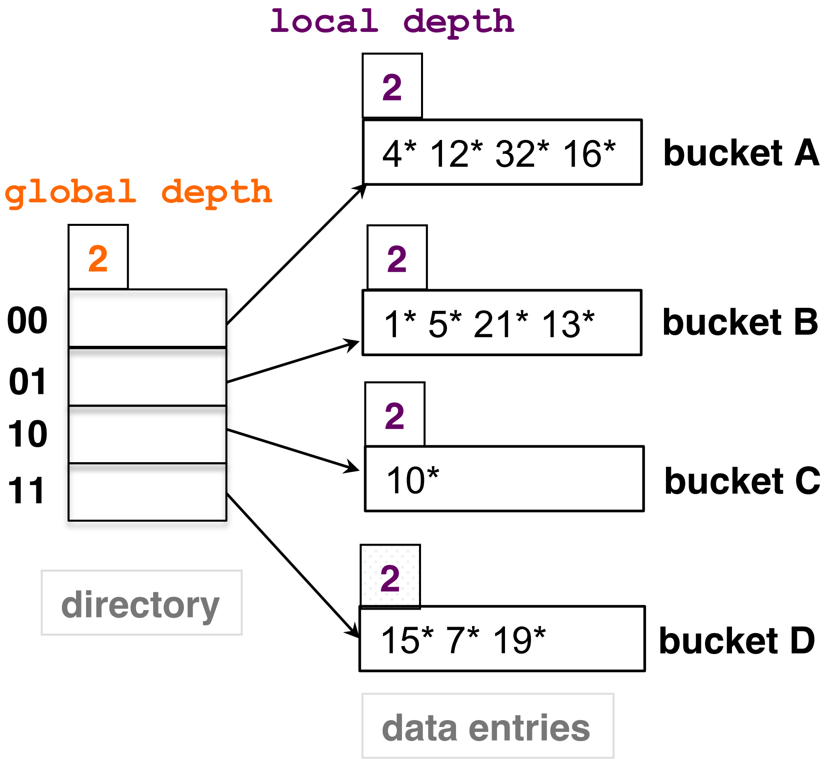
try to insert entry to corresponding bucket

if bucket is full, increase +1 local depth and *split* bucket (*allocate new bucket, re-distribute*)

if necessary, double the directory i.e., when for split bucket local depth > global depth

when directory doubles, increase global depth +1

example



insert record with h(r) = 20
= binary 10100 ➔ bucket A

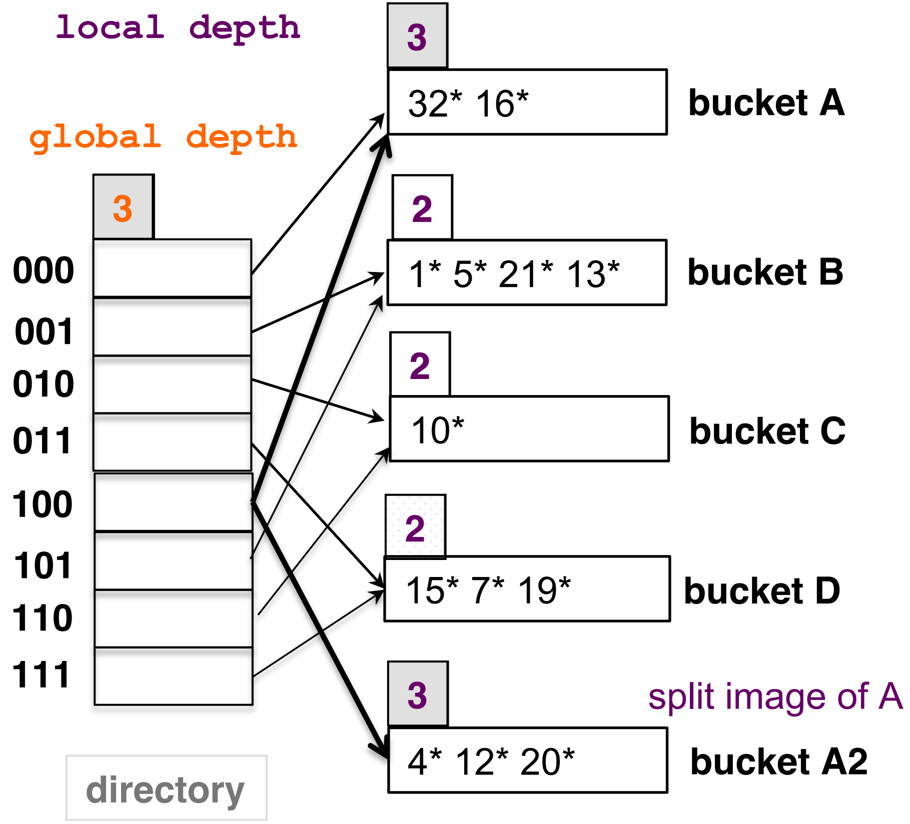
split bucket A !
allocate new page,
redistribute according to
modulo 2M = 8
3 least significant bits

we'll have more than 4
buckets now,
so double the directory!

example

split bucket A and
redistribute entries

update local depth
double the directory
update global depth
update pointers



notes

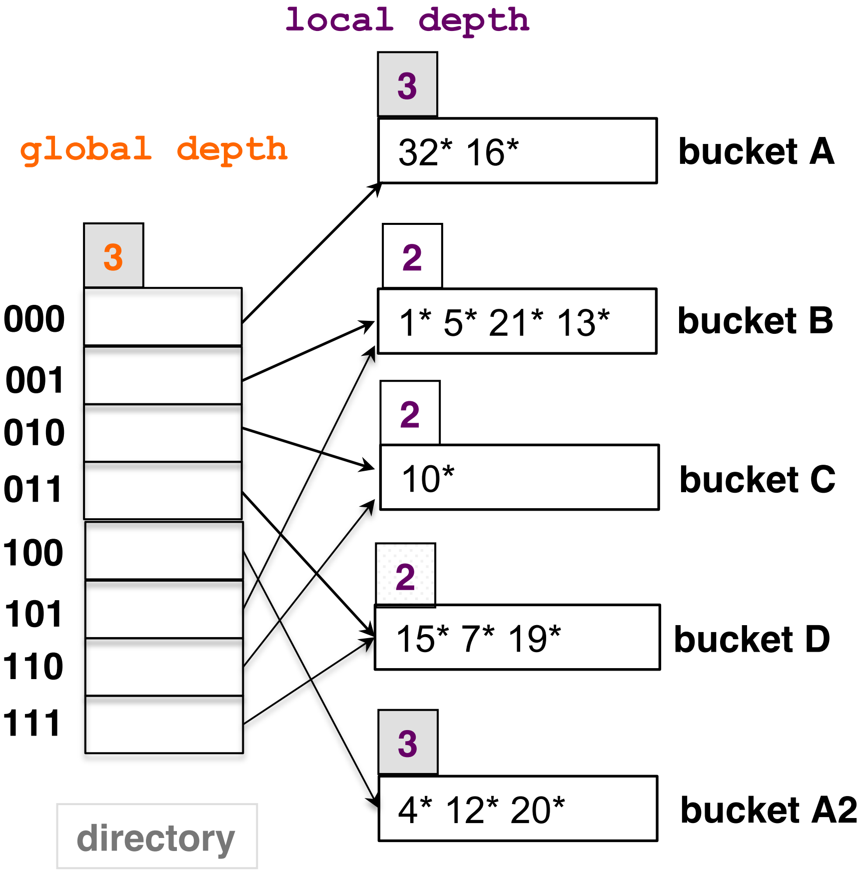
20 = binary 10100
last **2** bits (00) tell us *r* belongs in A or A2
last **3** bits needed to tell which

global depth of directory
number of bits enough to determine which bucket any entry belongs to
local depth of a bucket
number of bits enough to determine if an entry belongs to this bucket

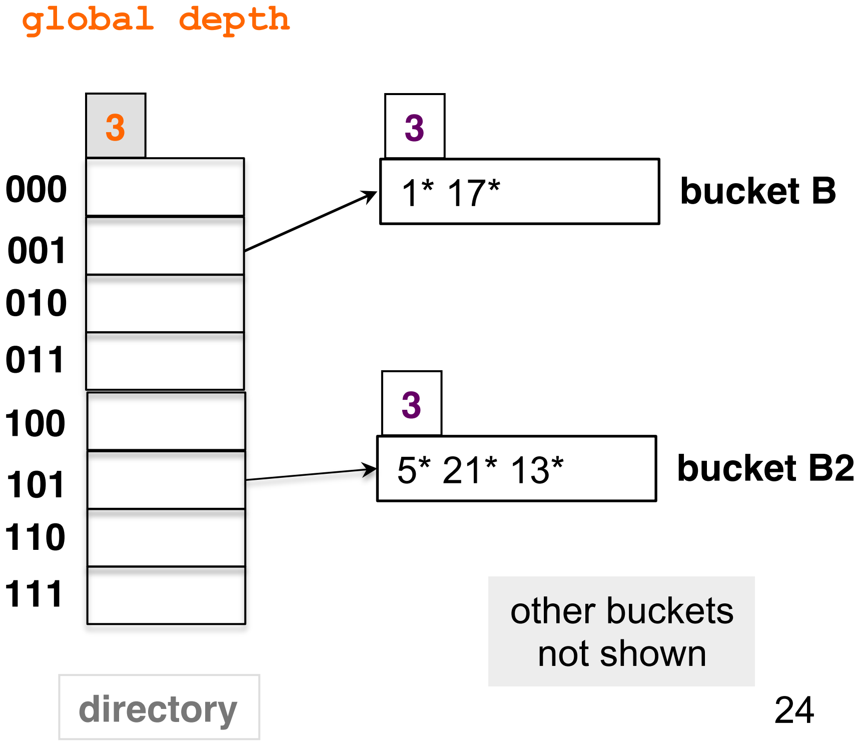
when does bucket split cause directory doubling?
before insert, *local depth* of bucket = *global depth*

example

insert h(r) = 17



split bucket B



comments on extendible hashing

if directory fits in memory,
equality query answered in one disk access

answered = retrieve rid

directory grows in spurts

if hash values are skewed, it might grow large

delete: reverse algorithm

empty bucket can be merged with its ‘split image’

when can the directory be halved?

when all directory elements point to same bucket as their ‘split image’

indexes in SQL

create index

```
CREATE INDEX indexb
ON students (age, grade)
USING BTREE;

CREATE INDEX indexh
ON students (age, grade)
USING HASH;

DROP INDEX indexh
ON student;
```

external sorting

the sorting problem

setting

a relation R, stored over N disk pages

$3 \le B < N$ pages available in memory (buffer pages)

task

sort records of R and store result on disk

sort by a function of record field values *f(r)*

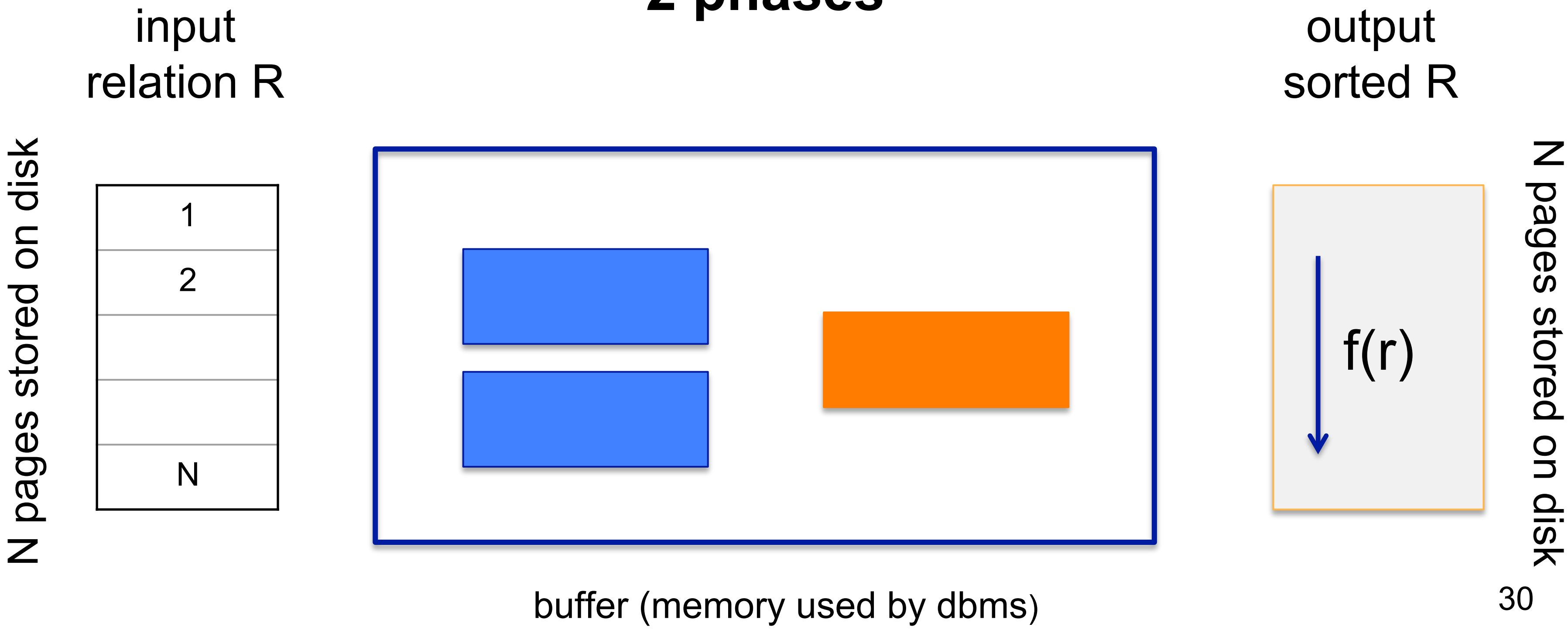
why

application need records ordered

part of join implementation (*soon...*)

sorting with 3 buffer pages

2 phases



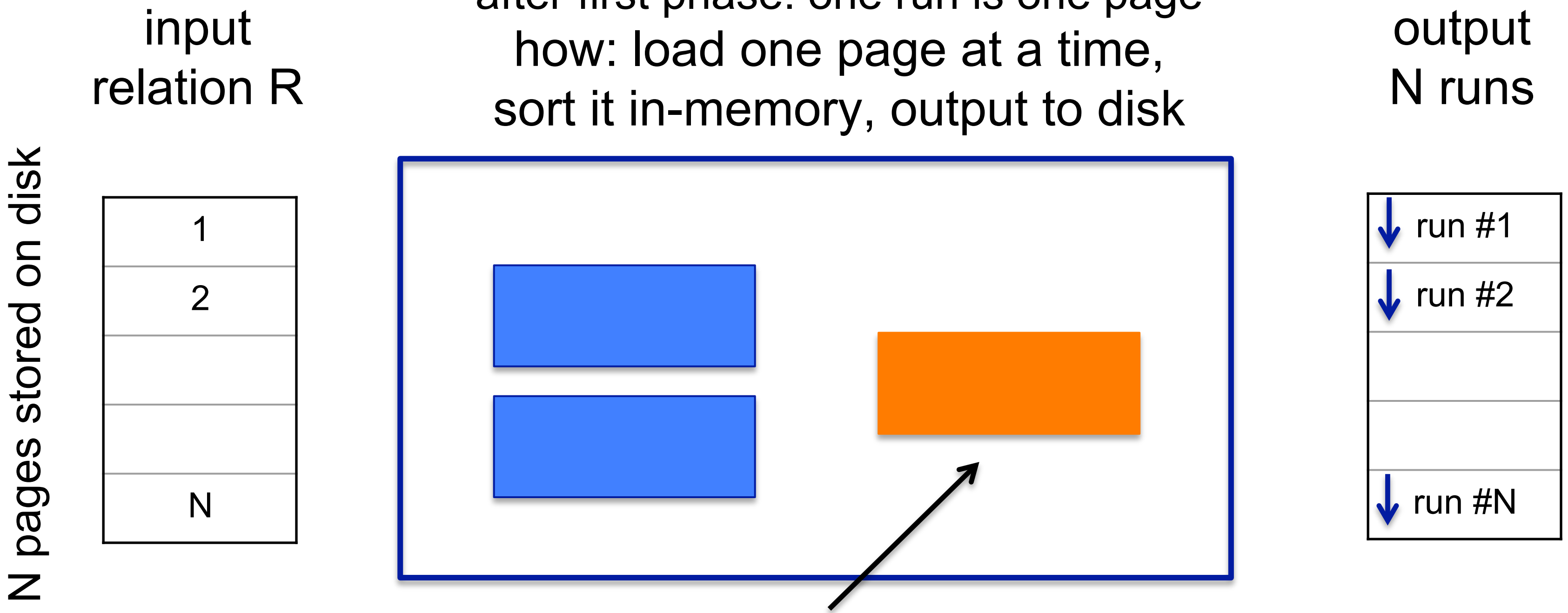
sorting with 3 buffer pages - first phase

pass 0: output N runs

run: sorted sub-file

after first phase: one run is one page

how: load one page at a time,
sort it in-memory, output to disk

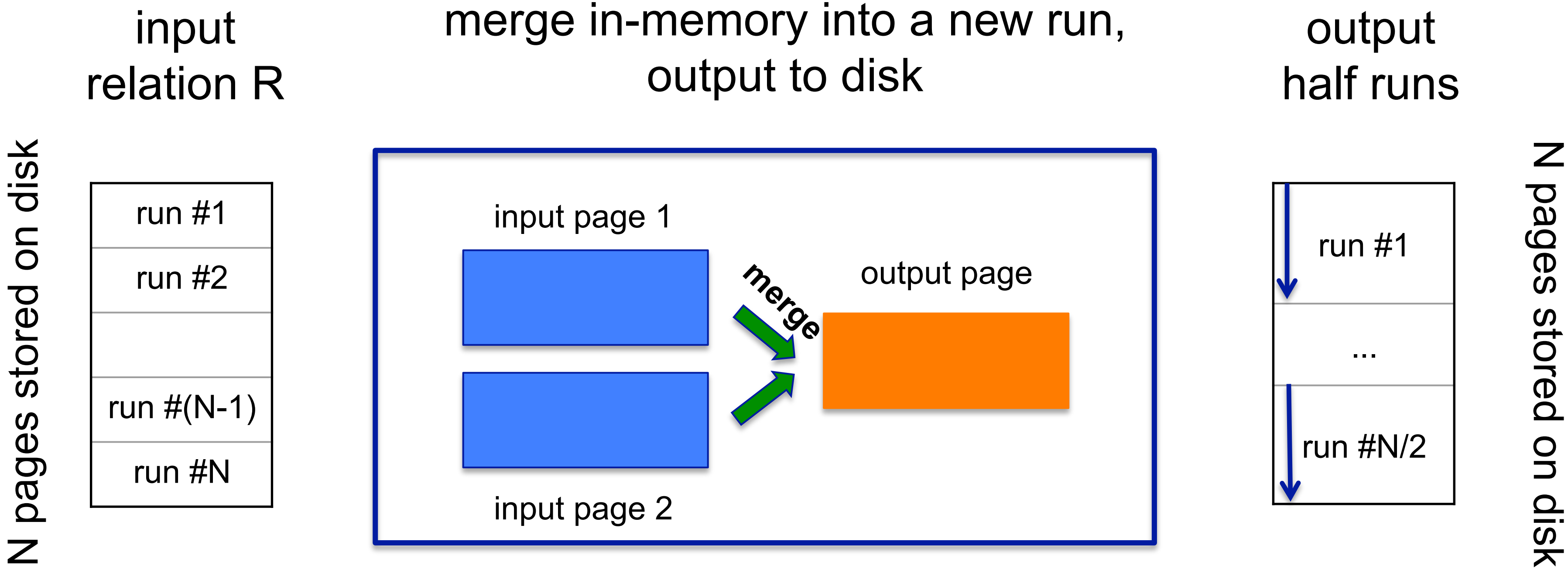


only 1 buffer page needed for first phase

sorting with 3 buffer pages - second phase

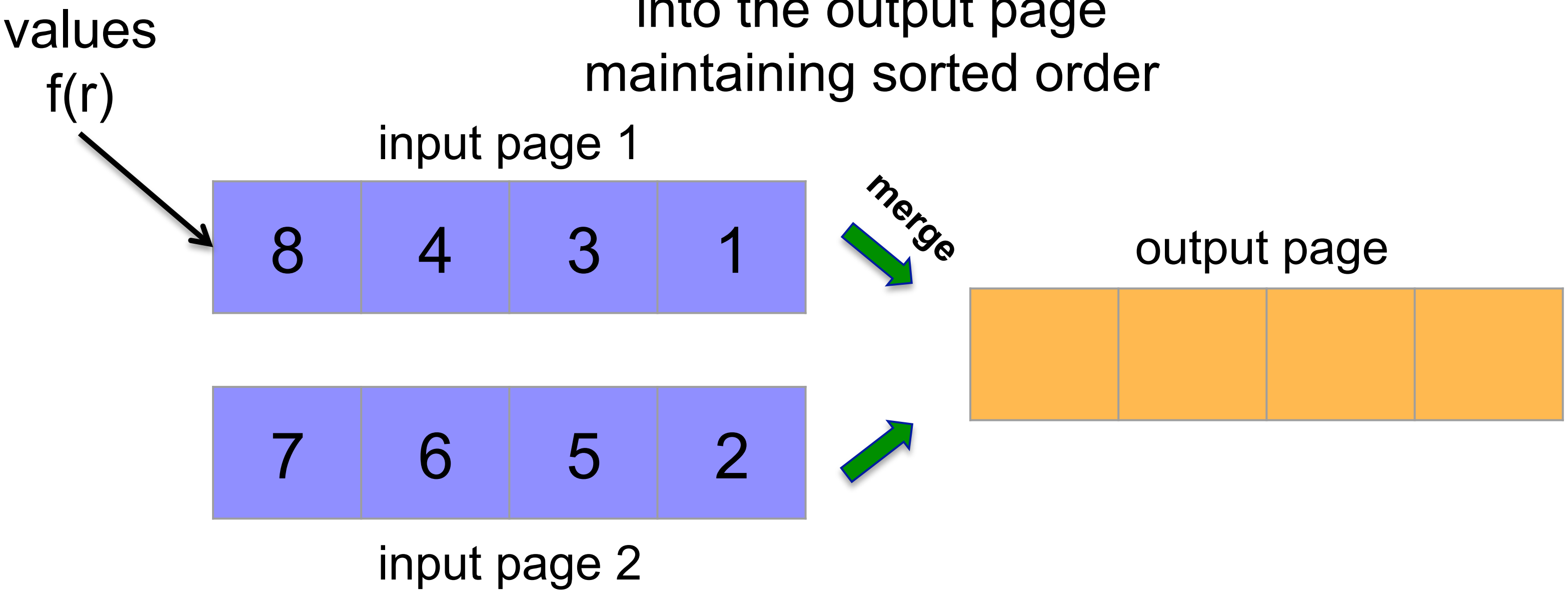
pass 1,2,...: halve the runs

how: scan pairs of runs, each in own page,
merge in-memory into a new run,
output to disk



merge?

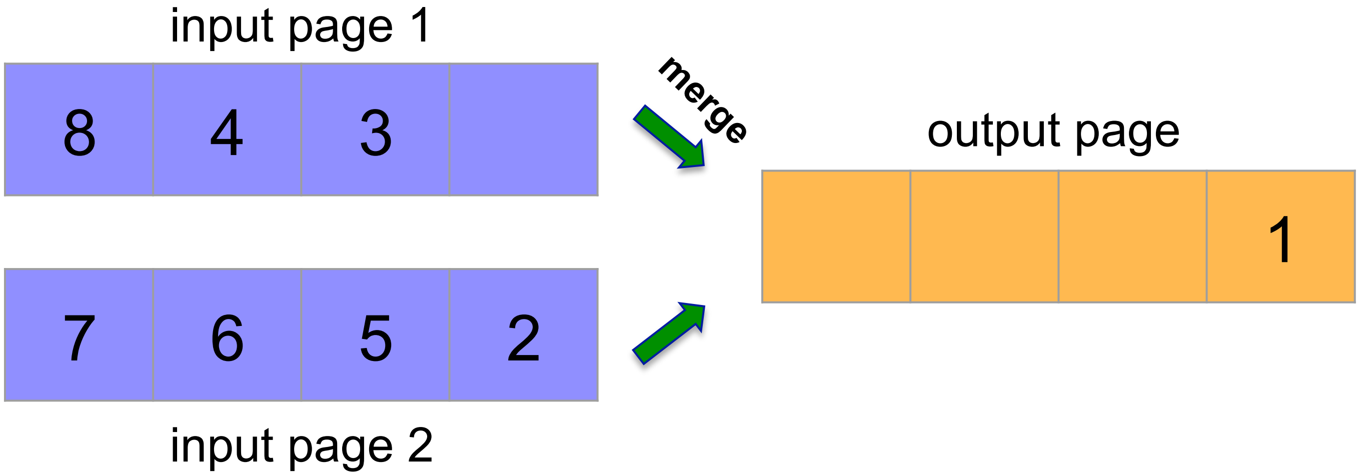
merge the two sorted input pages
into the output page
maintaining sorted order



compare the next smallest value from each page
move smallest to output page

merge?

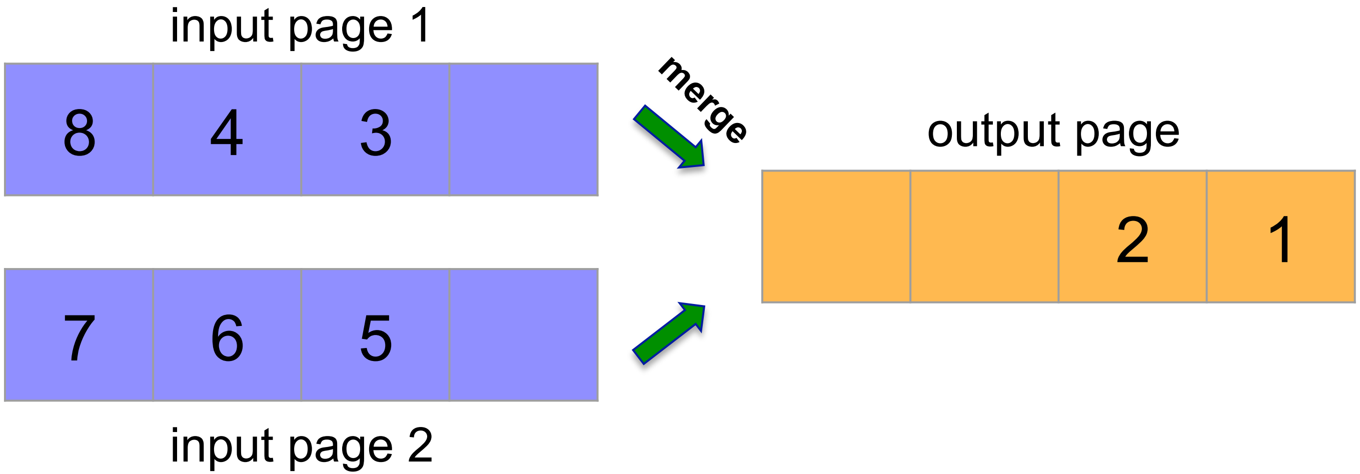
merge the two input pages into the output page
maintaining sorted order



compare the next smallest value from each page
move smallest to output page

merge?

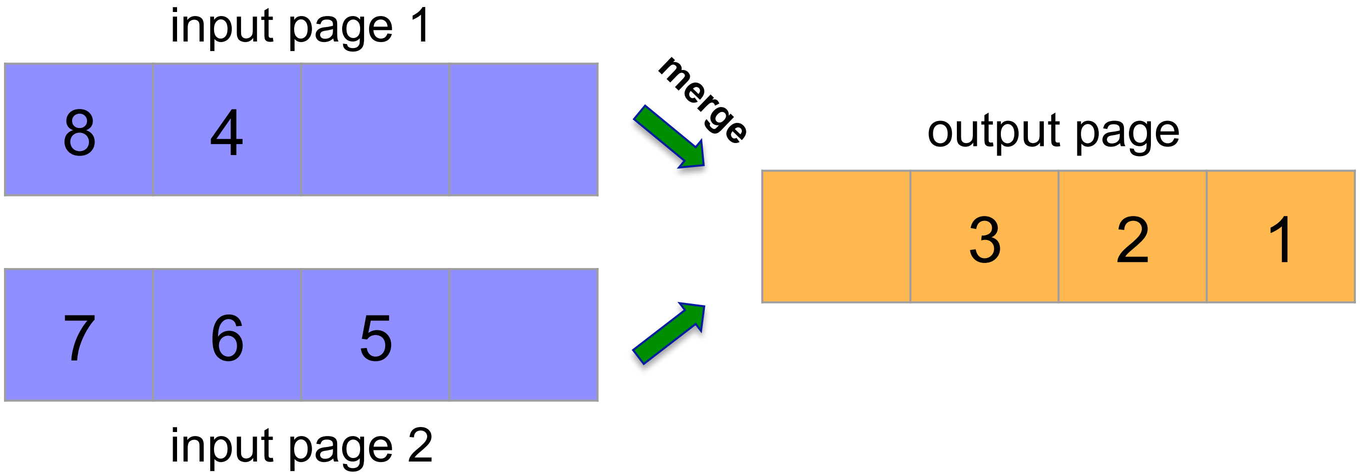
merge the two input pages into the output page
maintaining sorted order



compare the next smallest value from each page
move smallest to output page

merge?

merge the two input pages into the output page
maintaining sorted order

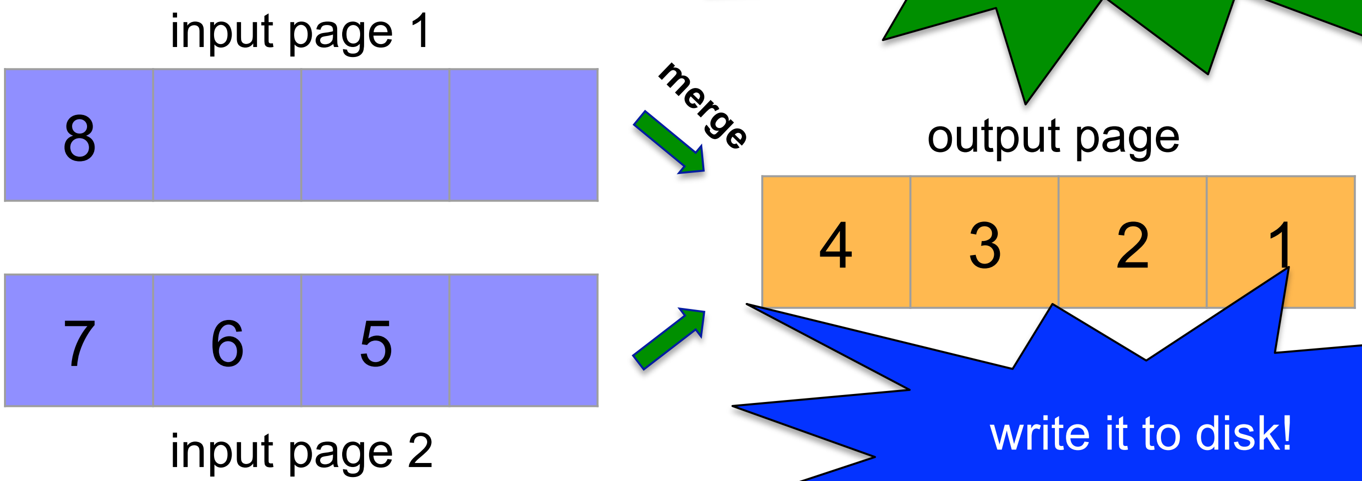


compare the next smallest value from each page
move smallest to output page

merge?

merge the two input pages int the ou t pa
maintaining sort

output page is full,
what do we do?

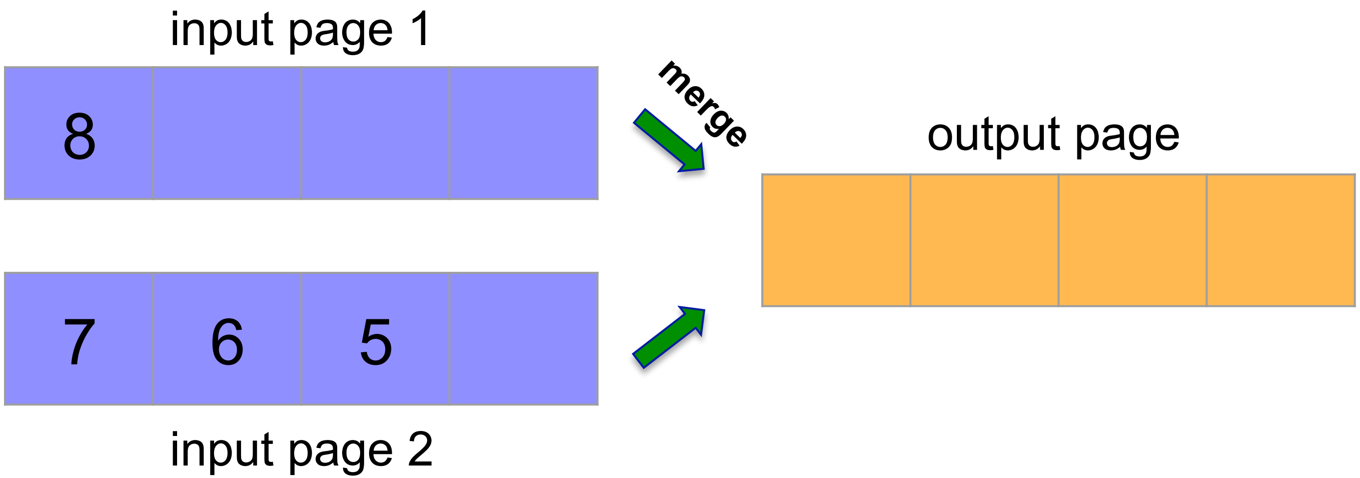


write it to disk!

compare the next smallest value from each page
move smallest to output page

merge?

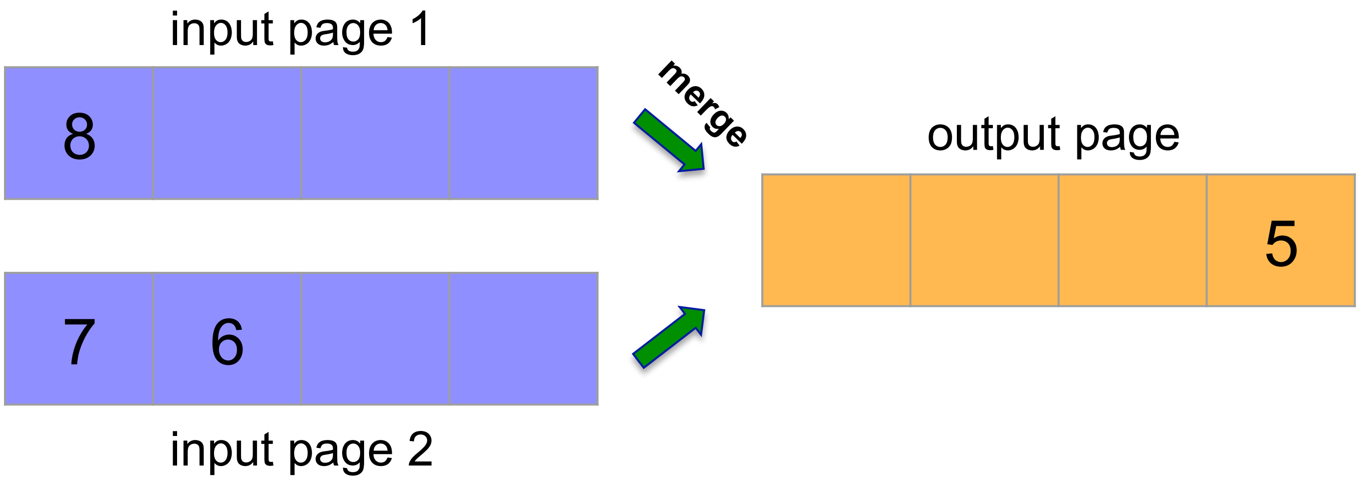
merge the two input pages into the output page
maintaining sorted order



compare the next smallest value from each page
move smallest to output page

merge?

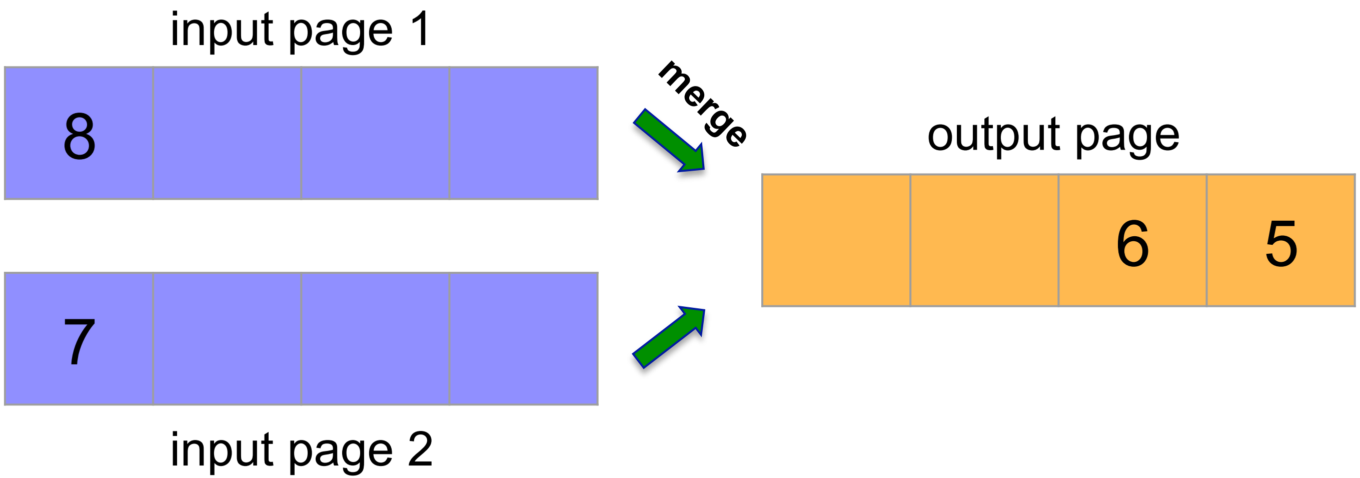
merge the two input pages into the output page
maintaining sorted order



compare the next smallest value from each page
move smallest to output page

merge?

merge the two input pages into the output page
maintaining sorted order



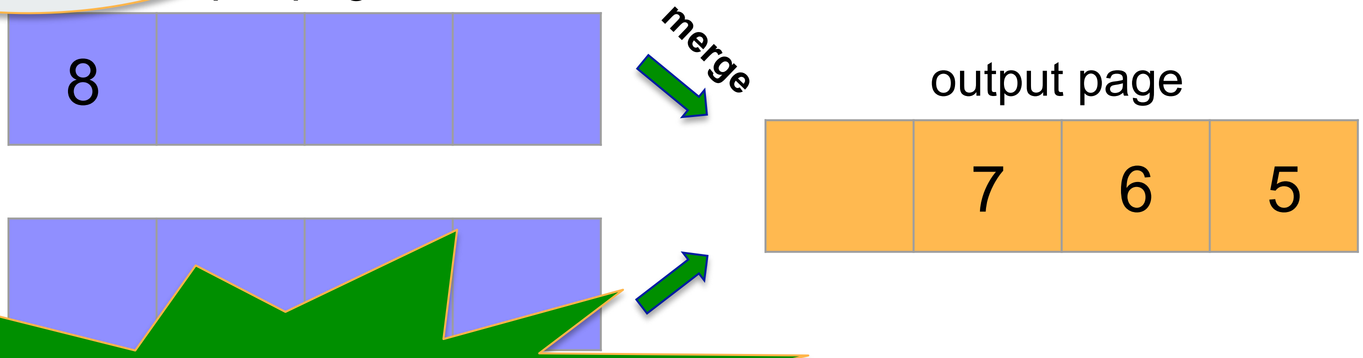
compare the next smallest value from each page
move smallest to output page

merge?

if the input run has more pages, load next one

the two input pages into the output page maintaining sorted order

put page 1

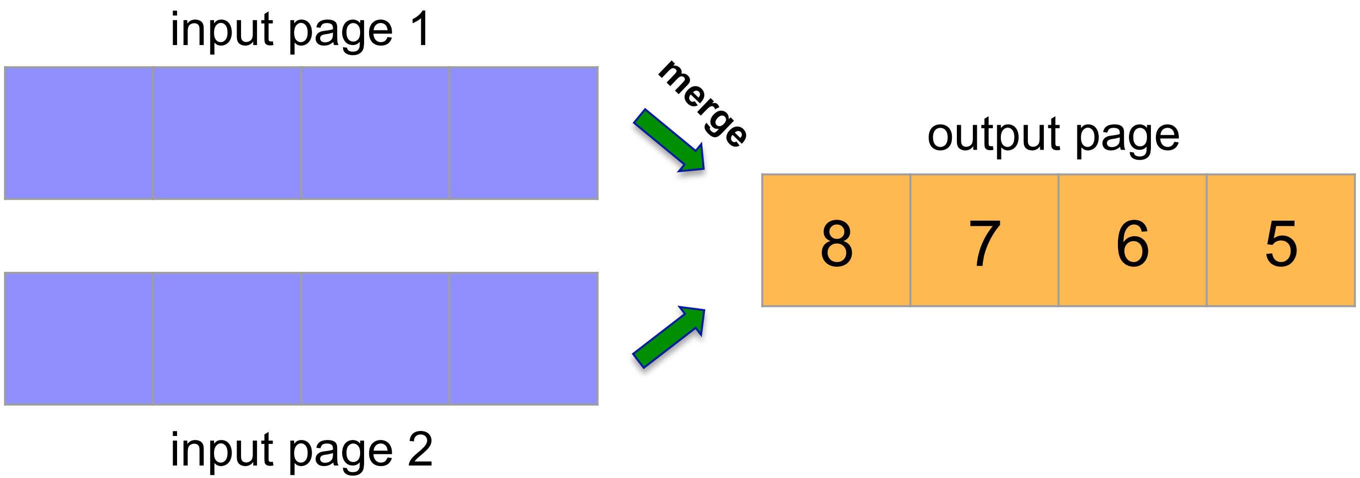


input page is empty!
what do we do?

ext smallest value from each page
move smallest to output page

merge?

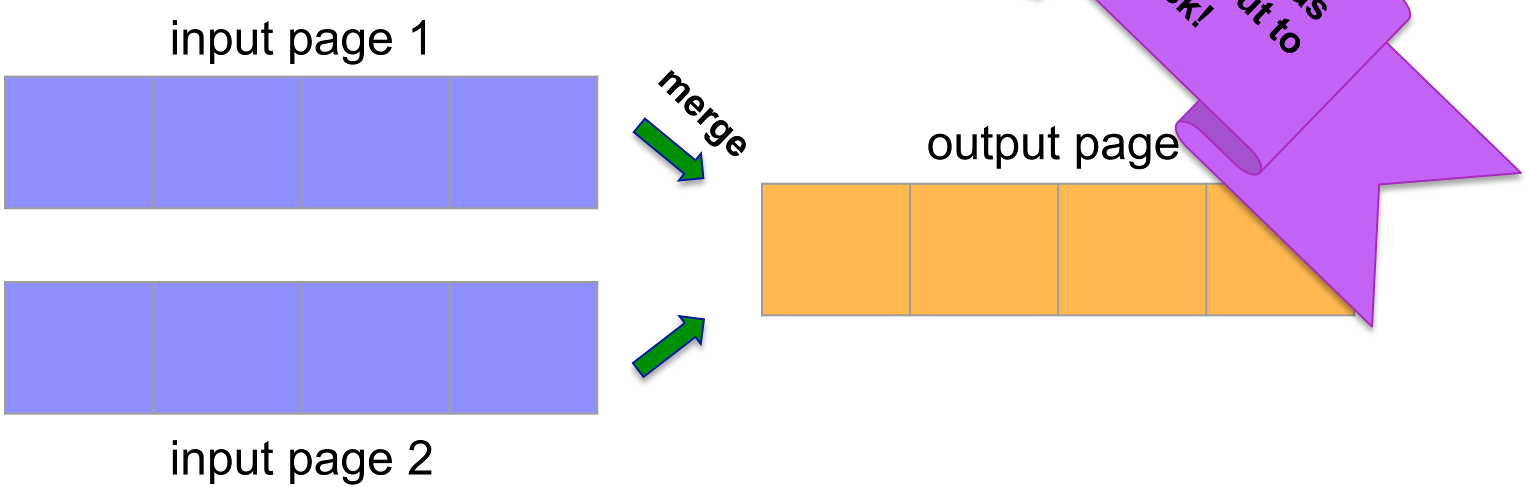
merge the two input pages into the output page
maintaining sorted order



compare the next smallest value from each page
move smallest to output page

merge?

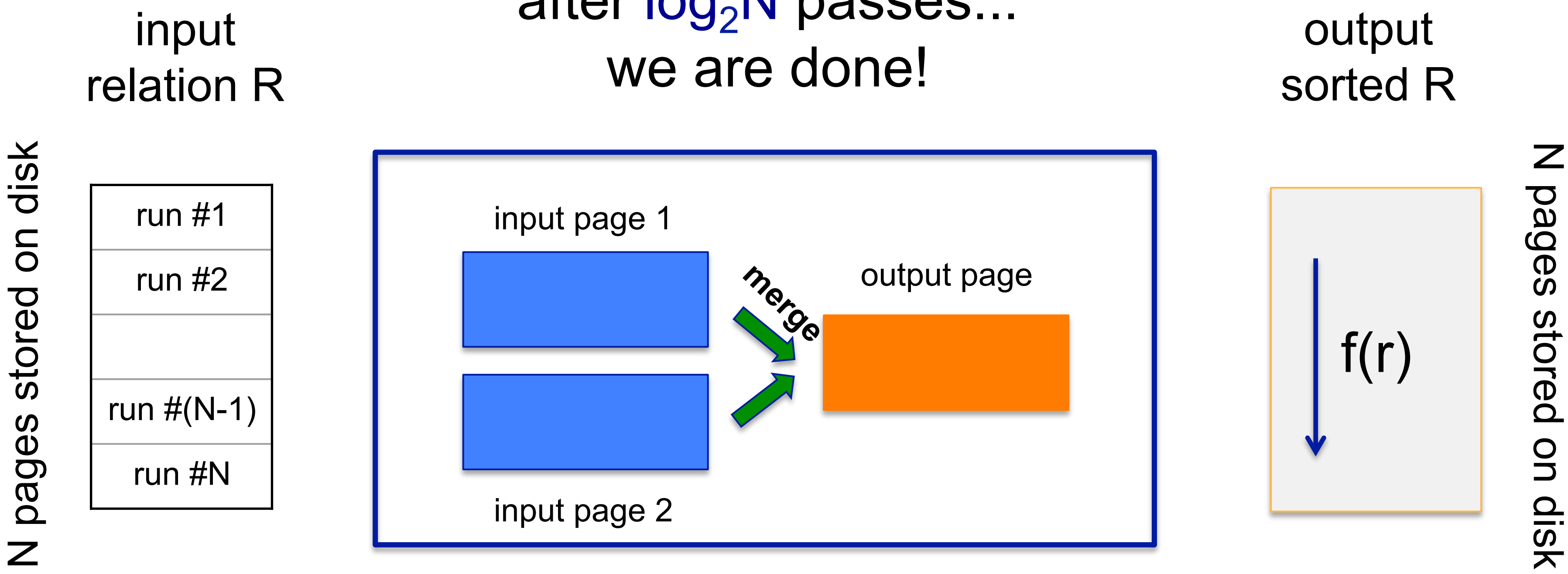
merge the two input pages into t
maintaining sorted orde



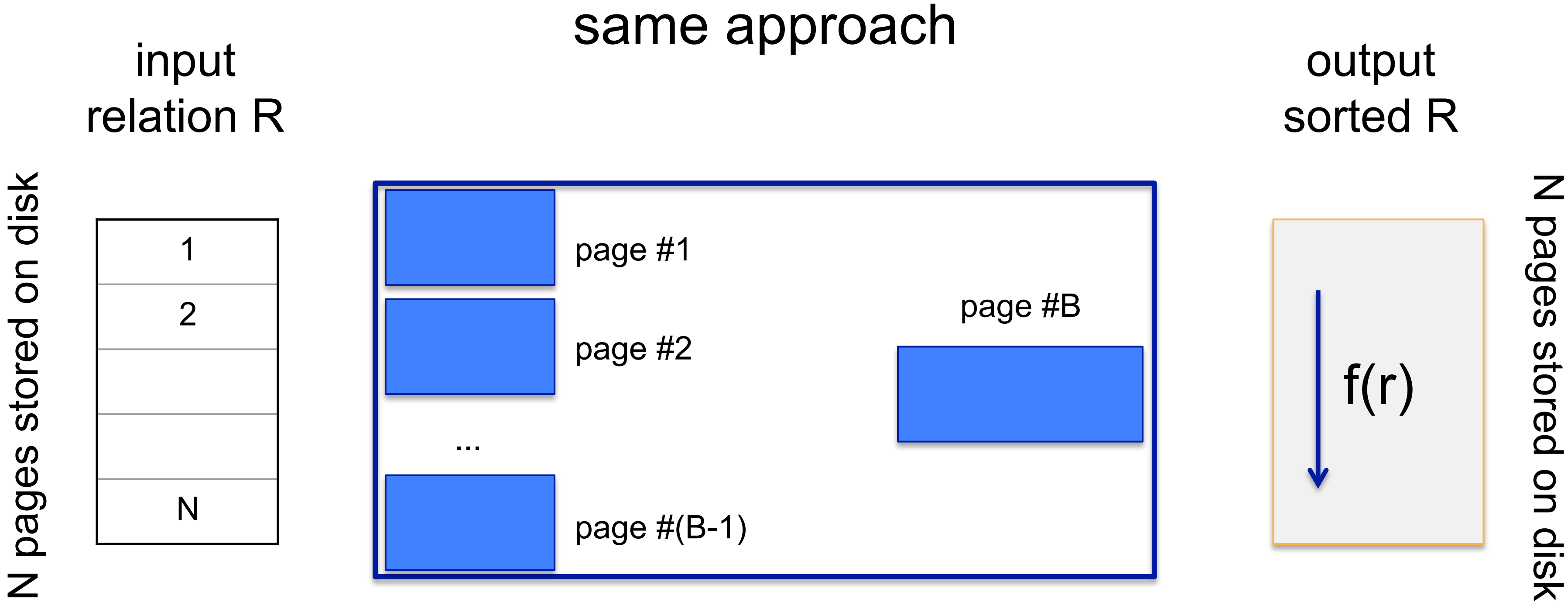
compare the next smallest value from each page
move smallest to output page

sorting with 3 buffer pages - second phase

pass 1,2,...: halve the runs
after $\log_2 N$ passes...
we are done!

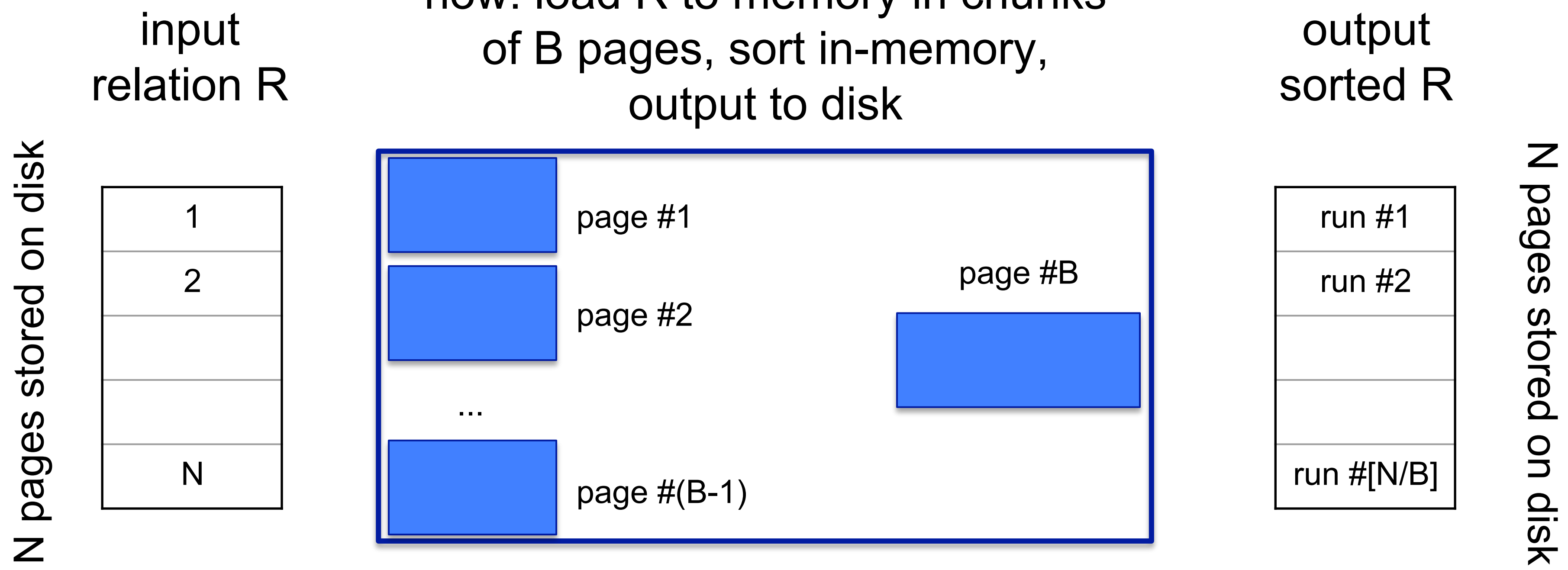


sorting with B buffer pages

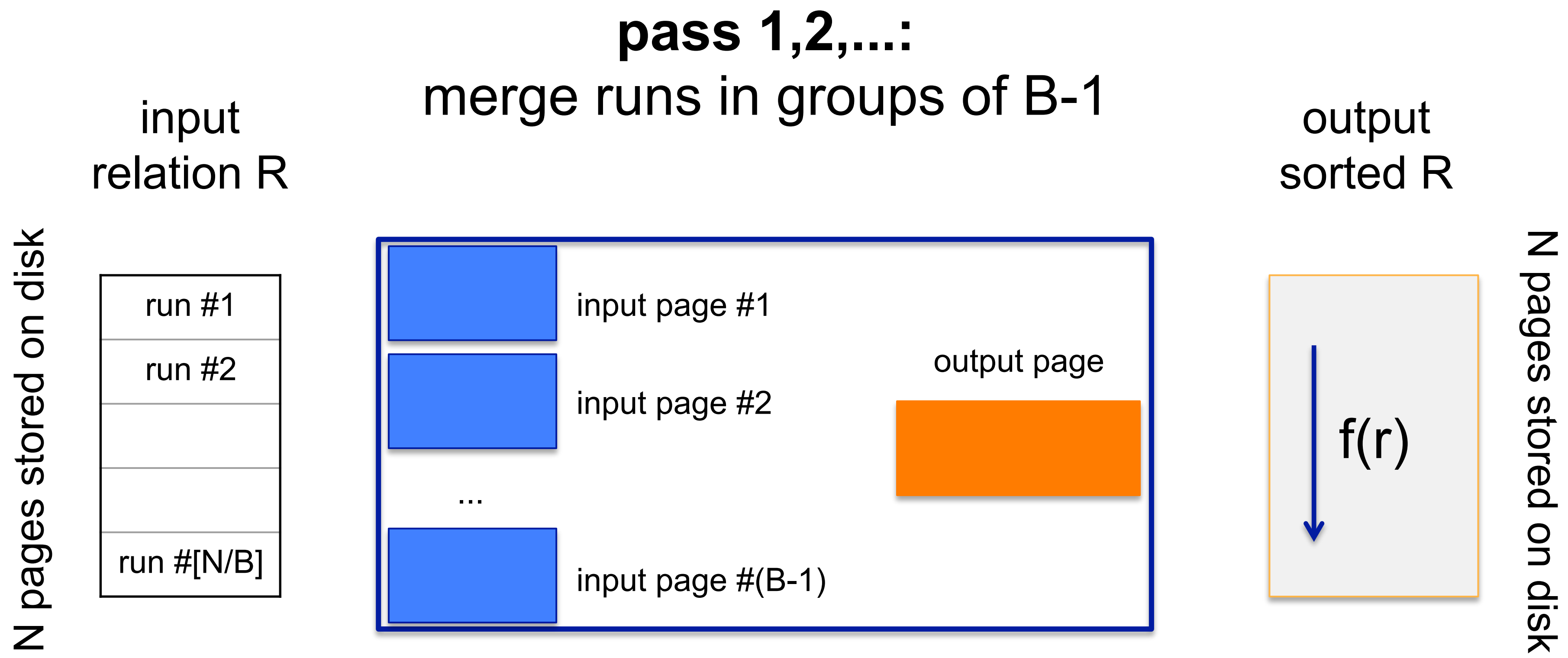


sorting with B buffer pages - first phase

pass 0: output [N/B] runs
how: load R to memory in chunks of B pages, sort in-memory, output to disk



sorting with B buffer pages - second phase



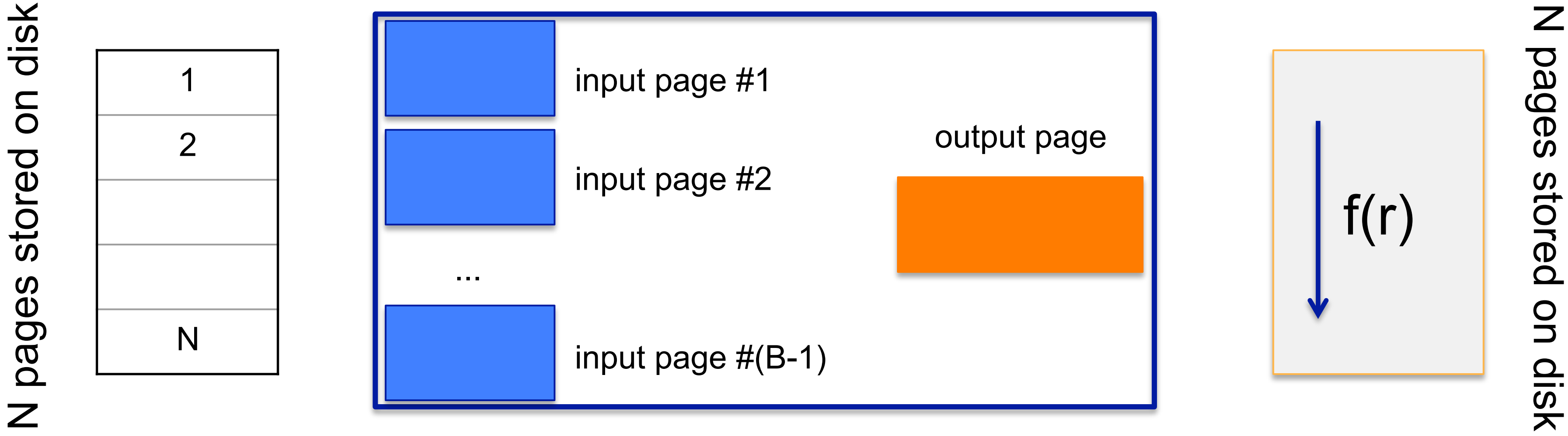
sorting with B buffer pages

how many passes in total?

let $N_1 = [N/B]$

total number of passes =

input relation R — phase 1 → $1 + [\log_{B-1}(N_1)]$ ← phase 2 — output sorted R



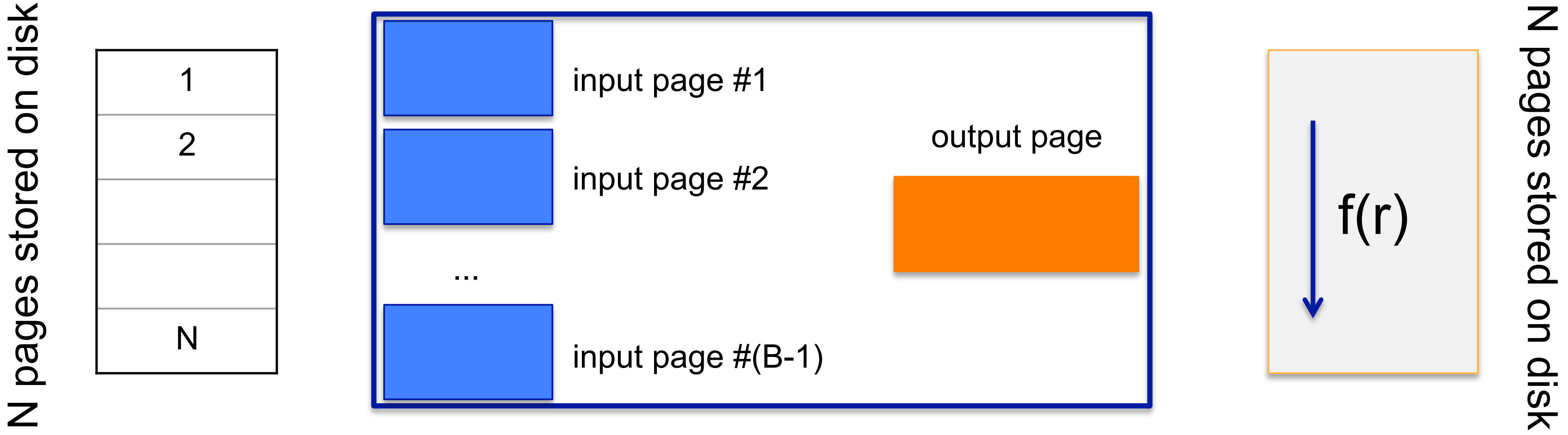
sorting with B buffer

how many pages I/O per pass?

2N: N input, N output

input relation R

output sorted R



sql joins

joins

so far, we have seen queries that
operate on a single relation

but we can also have queries that
combine information
from two or more relations

joins

students

sid	name	username	age
53666	Sam Jones	jones	22
53688	Alice Smith	smith	22
53650	Jon Edwards	jon	23

dbcourse

sid	points	grade
53666	92	A
53650	65	C

what does this compute?

```
SELECT *
FROM students S, dbcourse C
WHERE S.sid = C.sid
```

S.sid	S.name	S.username	C.age	C.sid	C.points	C.grade
53666	Sam Jones	jones	22	53666	92	A
53650	Jon Edwards	jon	23	53650	65	C

joins

```
SELECT *
FROM students S, dbcourse C
WHERE S.sid = C.sid
```

intuitively...
take all pairs of records from S and C
(the “cross product” S x C)

keep only records that
satisfy `WHERE` condition

S record #1	C record #1
S record #1	C record #2
S record #1	C record #3
...	...
S record #2	C record #1
S record #2	C record #2
S record #2	C record #3
...	...

joins

```
SELECT *
FROM students S, dbcourse C
WHERE S.sid = C.sid
```

intuitively...
take all pairs of records from S and C
(the "cross product" S x C)

keep only records that
satisfy `WHERE` condition

output join result

$S \bowtie_{S.sid=C.sid} C$

S record #1	C record #1
S record #1	C record #2
S record #1	C record #3
...	...
S record #2	C record #1
S record #2	C record #2
S record #2	C record #3
...	...

joins

```
SELECT *
FROM students S, dbcourse C
WHERE S.sid = C.sid
```

expensive to materialize!

intuitively...

take all pairs of records from S and C
(the “cross product” S x C)

keep only records that
satisfy `WHERE` condition

output join result

$S \bowtie_{S.sid=C.sid} C$

S record #1	C record #2
S record #2	C record #1

in what follows...

```
SELECT *
FROM students S, dbcourse C
WHERE S.sid = C.sid
```

algorithms to compute joins
without materializing cross product

assuming WHERE condition is equality condition
as in the example
assumption is not essential, though

join algorithms

the join problem

input

relation **R**: M pages on disk, p_R records per page

relation **S**: N pages on disk, p_S records per page

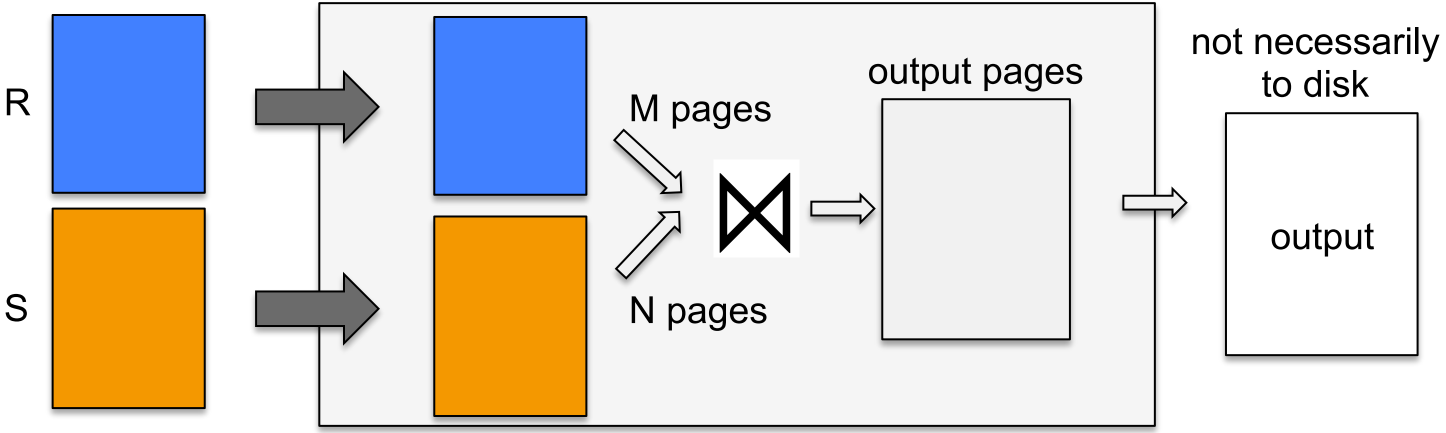
$M \leq N$

output

$$R \bowtie_{R.a=S.b} S$$

if there is enough memory...

load both relations in memory

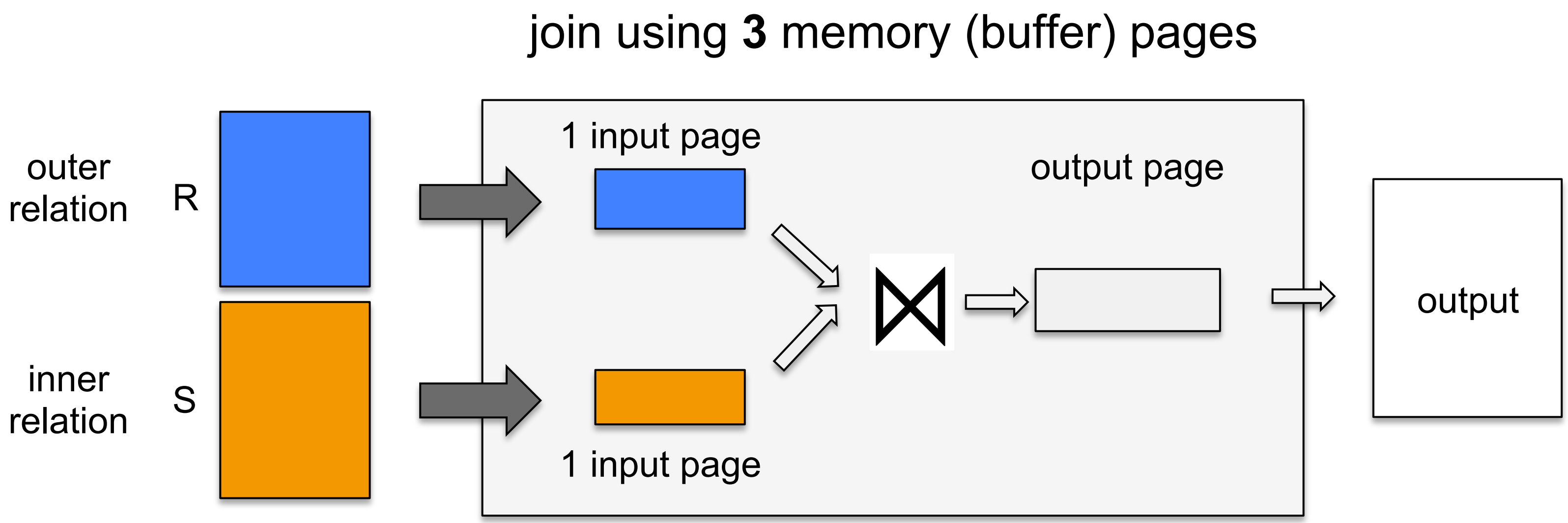


in-memory
for each record **r** in R
for each record **s** in S
if **r**.a = **s**.b:
store (**r, s**) in output pages
output

we only have to scan
each relation once

I/O cost
(ignoring final output cost)
M + N pages

page-oriented simple nested loops join



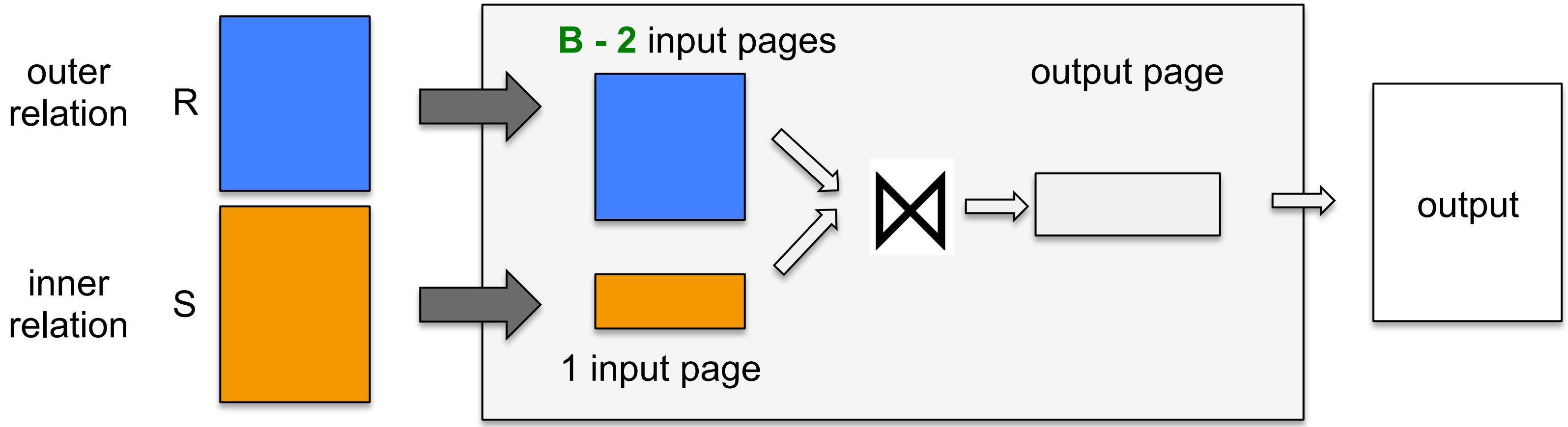
R is scanned once
S is scanned M times

```
for each page P of R
  for each page Q of S
    compute P join Q; store in output page
```

I/O cost (pages)
M + M * N

block nested loops join

join using **B** memory (buffer) pages



R is scanned once
S is scanned M/(B-2) times

```
for each block P of (B-2) pages of R
  for each page Q of S
    compute P join Q; store in output page
```

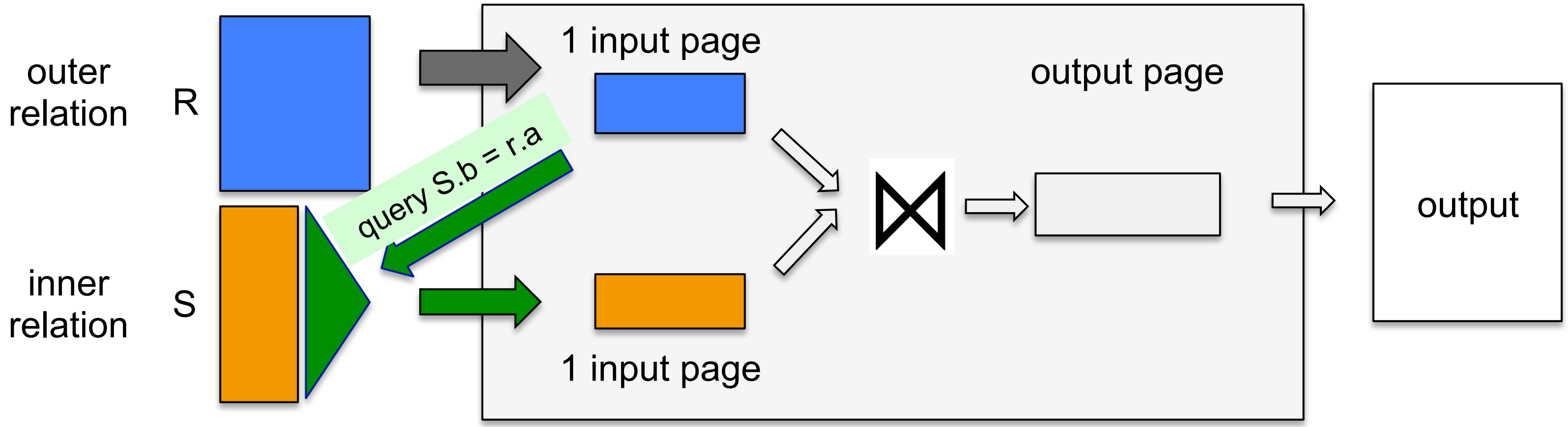
I/O cost (pages)
M + [M/(B-2)] * N

index nested loop join

relation S has an index on the join attribute

use one page to make a pass over R

use index to retrieve only matching records of S




```
for each record r of R
  for each record s of S with s.b = r.a // query index
    add (r,s) to output
```

index nested loop join

cost

M
to scan R

(Mp_R) x (cost of query on S)

number of records of R

covered in previous lectures

total

$M + Mp_R \times$ (cost of query on S)

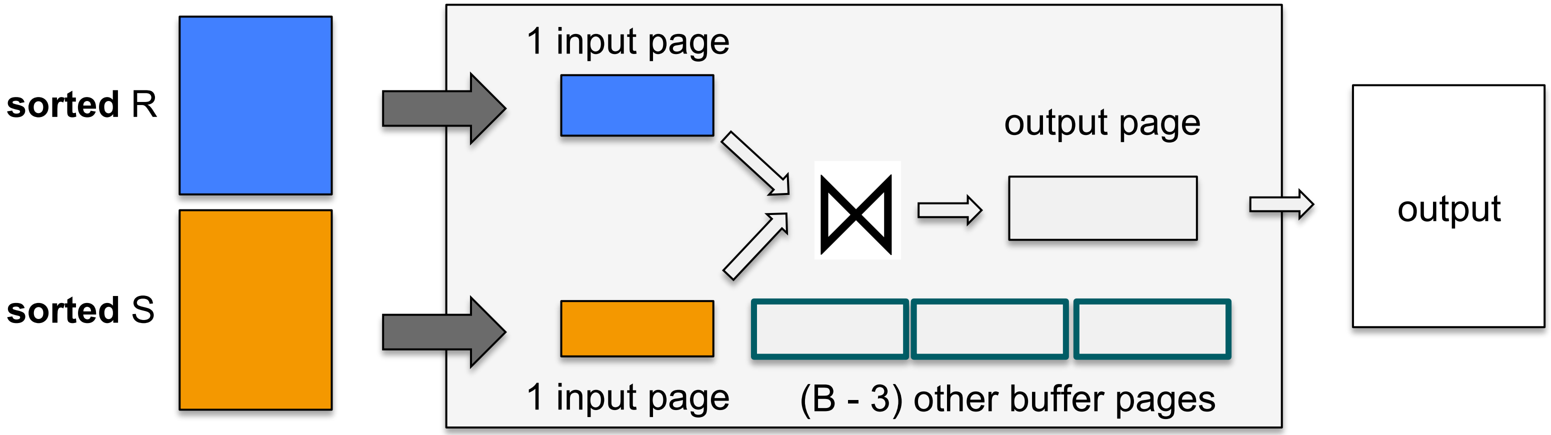
sort-merge join

two phases
sort and merge

sort
R and S on the join attribute
using external sort algorithm
cost O(nlogn), n: number of relation pages

merge sorted R and S

sort-merge join: merge



sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)

	R.a	S.b	
r	1	3	s
	2	4	
	6	5	
	7	6	
	8	9	
	9	10	

main loop

```
repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s
```

sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)

	R.a	S.b	
	1	3	← s
r →	2	4	
	6	5	
	7	6	
	8	9	
	9	10	

main loop

repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s

sort-merge join: merge

current pages
in memory
(only join attributes
are shown)

assumption
b is a key for S

R.a	S.b
1	3
2	4
6	5
7	6
8	9
9	10

r

s

main loop

```
repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s
```

sort-merge join: merge

assumption
b is a key for S

current pages in memory (only join attributes are shown)

R.a	S.b
1	3
2	4
6	5
7	6
8	9
9	10

r

s

main loop

repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s

sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)

R.a	S.b
1	3
2	4
6	5
7	6
8	9
9	10

r

s

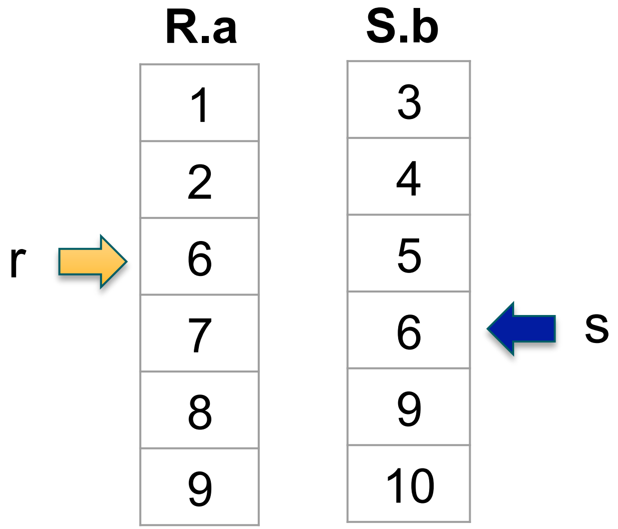
main loop

```
repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s
```

sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)



main loop

repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s

sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)

	R.a	S.b	
	1	3	
	2	4	
r	6	5	
	7	6	s
	8	9	
	9	10	

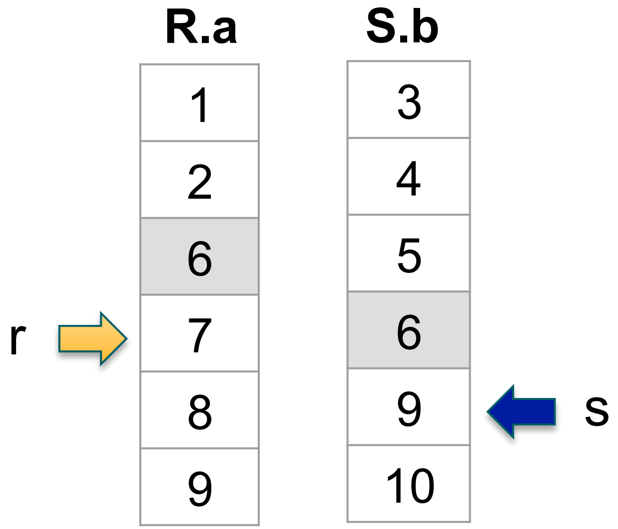
main loop

```
repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s
```

sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)



main loop

repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s

sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)

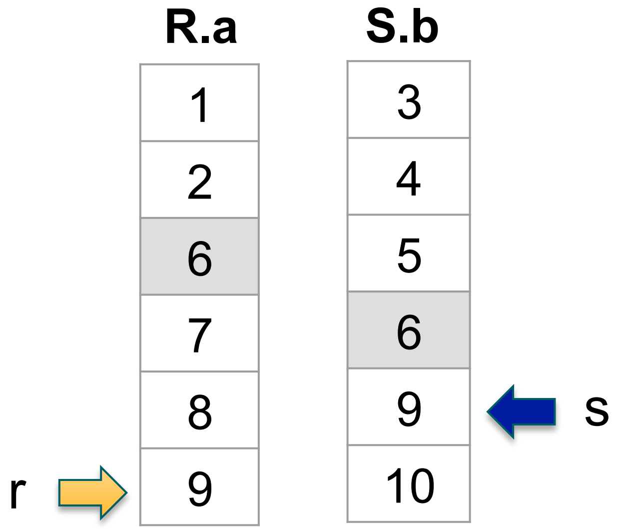
	R.a	S.b	
	1	3	
	2	4	
	6	5	
	7	6	
r	8	9	s
	9	10	

main loop

```
repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s
```

sort-merge join: merge

assumption
b is a key for S



current pages
in memory
(only join attributes
are shown)

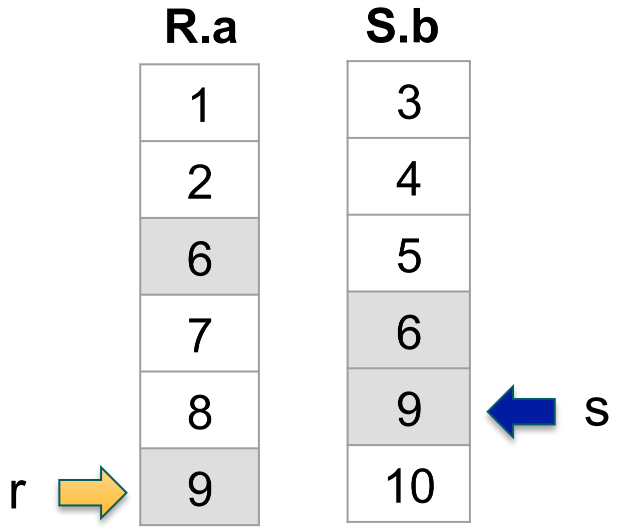
main loop

```
repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s
```

sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)



main loop

repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s

sort-merge join: merge

new page for R!

assumption
b is a key for S

current pages in memory (only join attributes are shown)

r

R.a	S.b
13	3
14	4
15	5
17	6
19	9
23	10

s

main loop

repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s

sort-merge join: merge

assumption
b is a key for S

current pages
in memory
(only join attributes
are shown)

	R.a	**S.b**	
r	13	3	
	14	4	
	15	5	
	17	6	
	19	9	
	23	10	s

main loop

```
repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s
```

cost for merge
M + N

sort-merge join: merge

what if this assumption does not hold?

assumption
b is a key for S

	R.a	S.b	
r	13	3	
	14	4	
	15	5	
	17	6	
	19	9	
	23	10	s

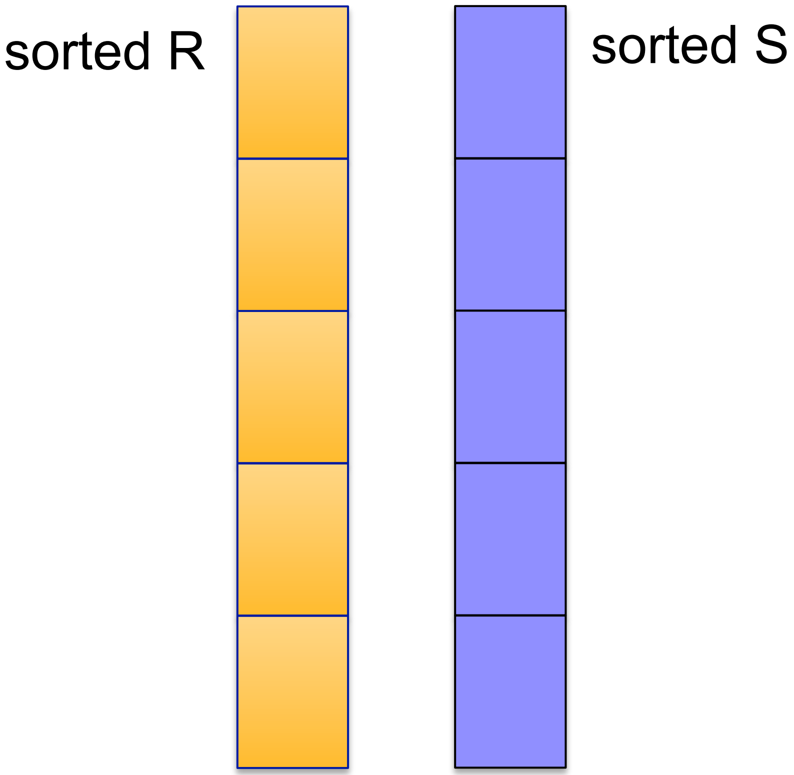
current pages
in memory
(only join attributes
are shown)

main loop

```
repeat while r != s:
  advance r until r >= s
  advance s until s >= r
output (r, s)
advance r and s
```

cost for merge
M + N

sort-merge join: merge



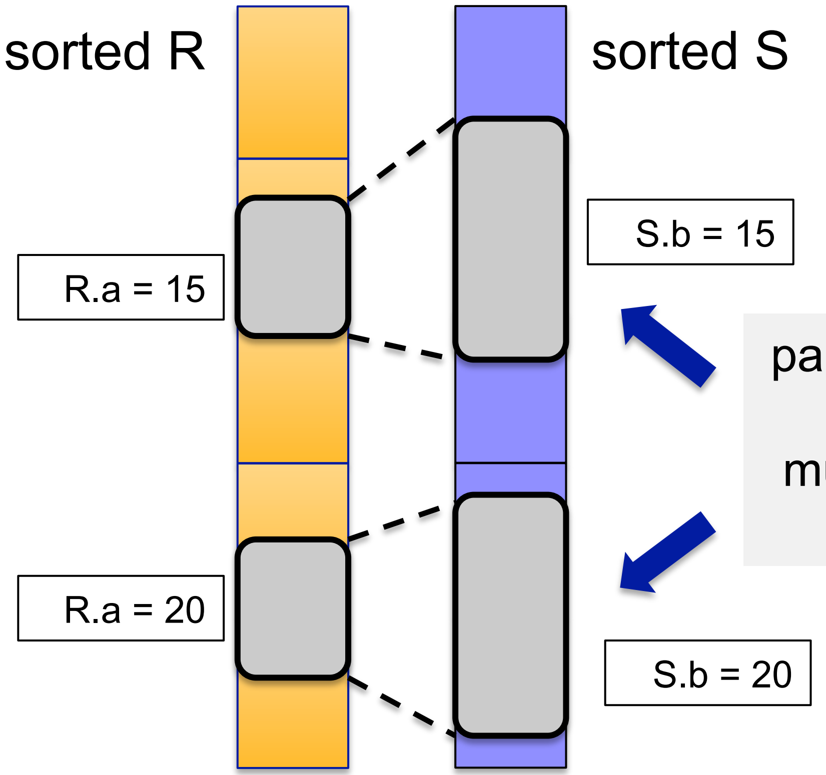
the pages of
sorted R and S

join attribute is not key
on either relation

sort-merge join: merge

the pages of
sorted R and S

join attribute is not key
on either relation

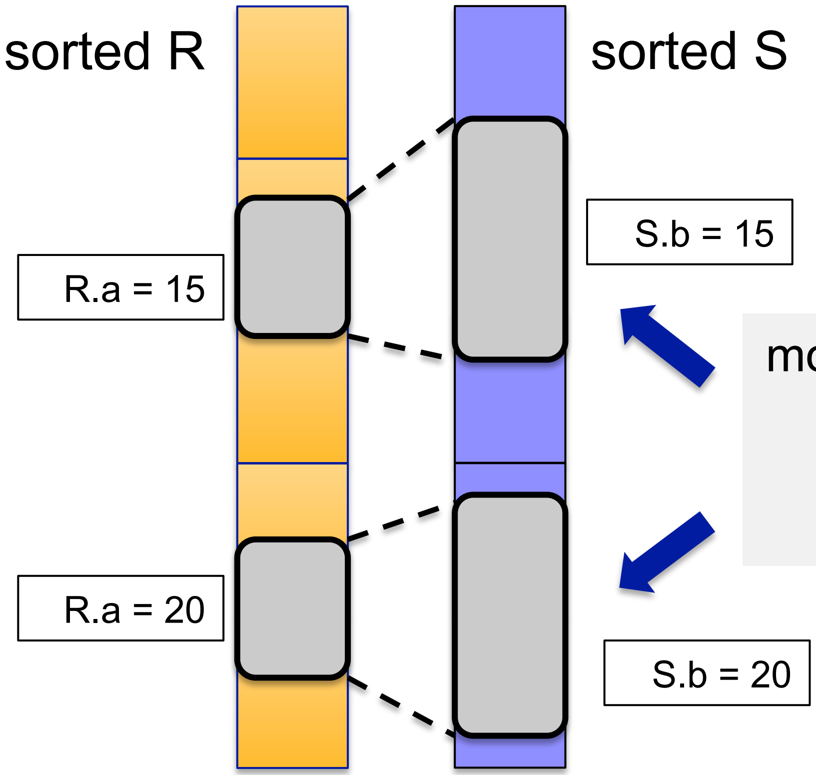


parts of R and S with same
join attribute value
must output cross product
of these parts

sort-merge join: merge

the pages of
sorted R and S

join attribute is not key
on either relation



modify algorithm to perform join over these areas e.g., page-oriented simple nested loops

for that algorithm, worst case cost is M + MN

hash join

two phases
partition and probe

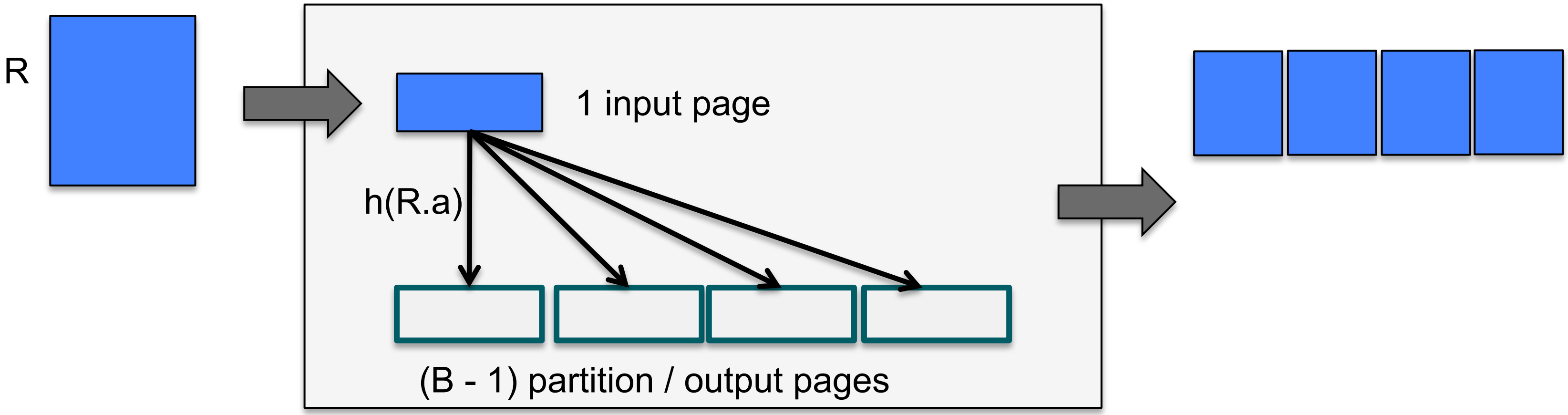
partition
each relation into partitions
using the same hash function
on the join attribute

probe
join the corresponding partitions

hash join - partition

B buffer pages available
scan R with 1 buffer page
hash into B-1 partitions
on **join attribute**

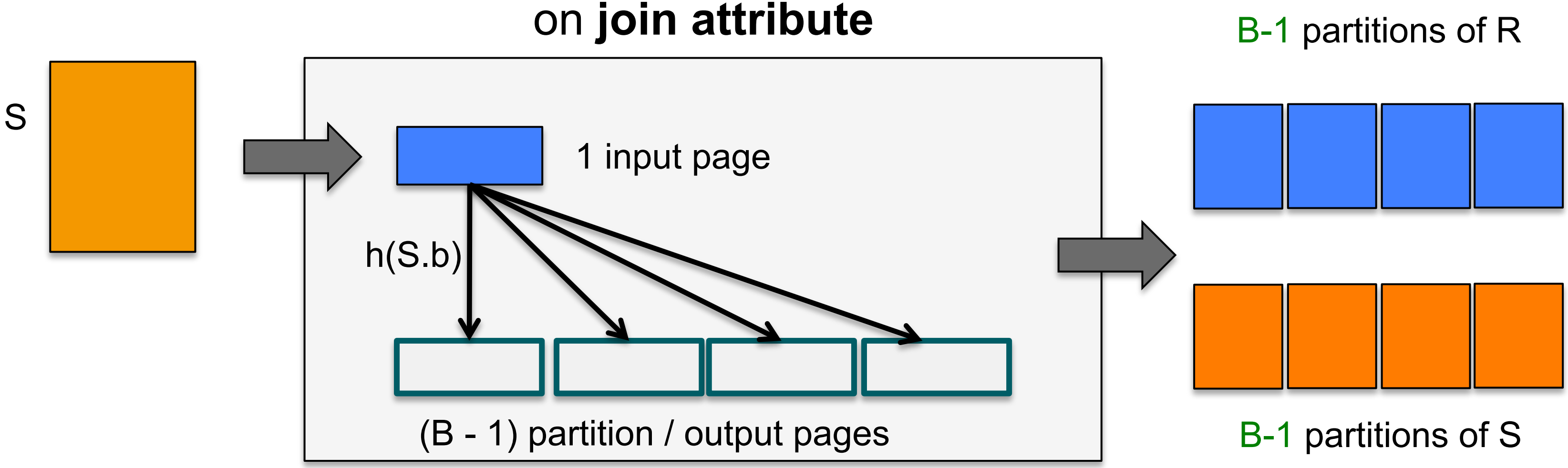
B-1 partitions of R



use B-1 pages to hold the partitions,
flush when full or scan of R ends

hash join - partition

B buffer pages available
scan S with 1 buffer page
hash into B-1 partitions
on **join attribute**



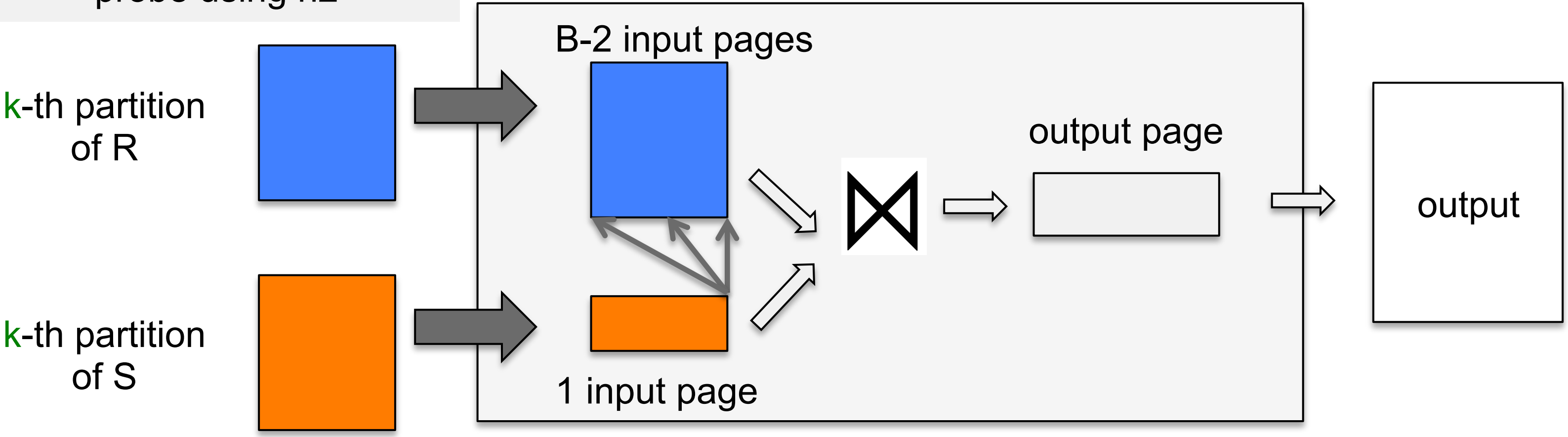
use B-1 pages to hold the partitions,
flush when full or scan of S ends

hash join - probe

variant
re-partition the partition of R in-memory with hash function h2,
probe using h2

B buffer pages available
load k-th partition of R into memory
(assuming it fits in B-2 pages)

holds when size of each partition fits in B-2 pages
approximately
B-2 > M / (B - 1)
B > √M



scan k-th partition of S one page at a time;
for each record of S, probe the partition of R for matching records;
store matches in output page;
flush when full or done

hash join

cost

partition phase
read and write each relation once
2M + 2N

probing phase
read each partition once
(final output cost ignored)
M + N

total
3M + 3N

a few words on

query optimization

query optimization

once we submit a query
the dbms is responsible for
efficient computation

the same query can be
executed in many ways
each is an ‘**execution plan’**
or ‘query evaluation plan’

example

```
select *
from students
where sid = 100
```

execution plan

annotated relational algebra tree

$\sigma_{sid = 100}$ (on-the-fly) ← algorithm used by operator

students (scan) ← **access path** how we retrieve data from the relation: scan or index?

another execution plan

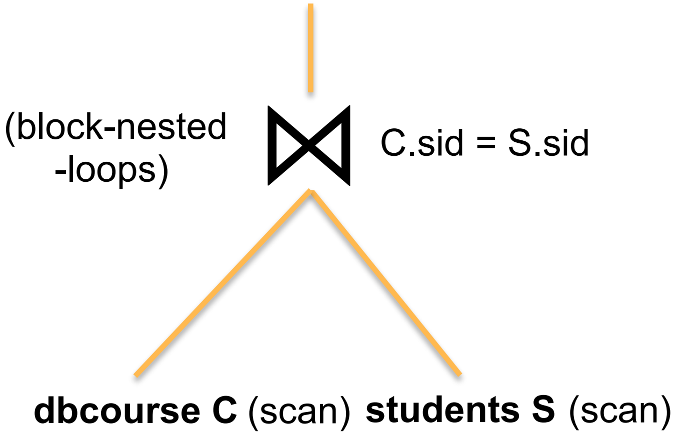
$\sigma_{sid = 100}$ (query index stud_btree)

students (b+ tree index stud_btree on sid)

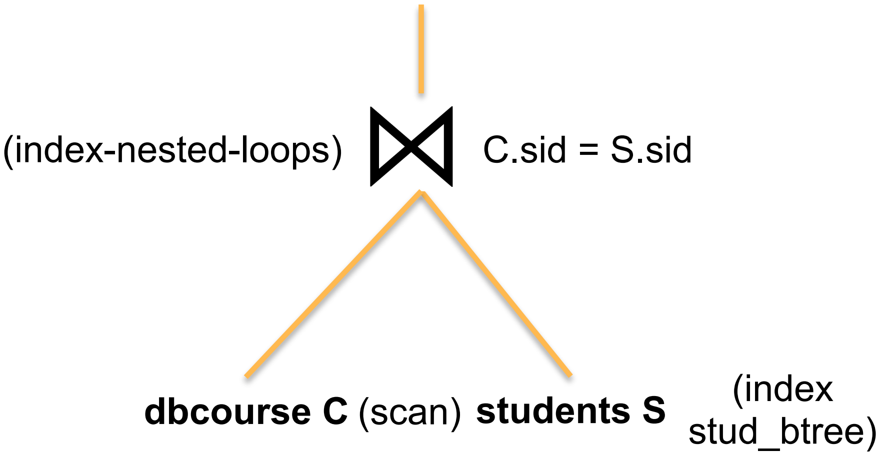
example

```
select *
from students S, dbcourse C
where S.sid = C.sid
```

execution plan



another execution plan



which plan to choose?

dbms estimates cost for a number of execution plans
(not all possible plans, necessarily!)

the estimates follow the cost analysis
we presented earlier

dbms picks the execution plan with
minimum estimated cost

summary

summary

- commonly used indexes
- B+ tree
  - most commonly used
  - supports efficient equation and range queries
- hash-based indexes
  - extendible hashing uses directory, not overflow pages
- external sorting
- joins
- query optimization

tutorial

next week

references

- “cowbook”, database management systems, by ramakrishnan and gehrke
- “elmasri”, fundamentals of database systems, elmasri and navathe
- other database textbooks

credits

some slides based on material from
database management systems, by ramakrishnan and gehrke

backup slides

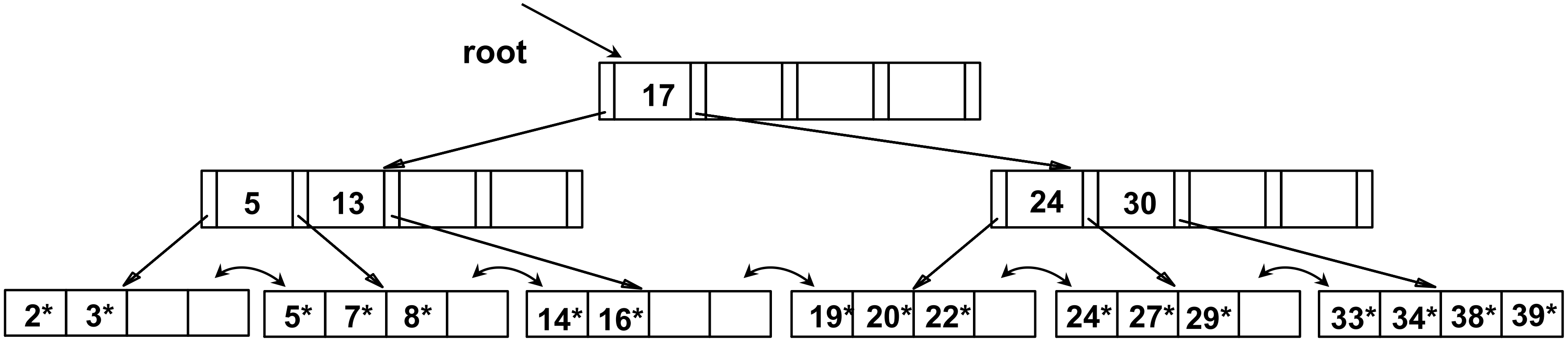
b+ tree - deletion

deleting a data entry

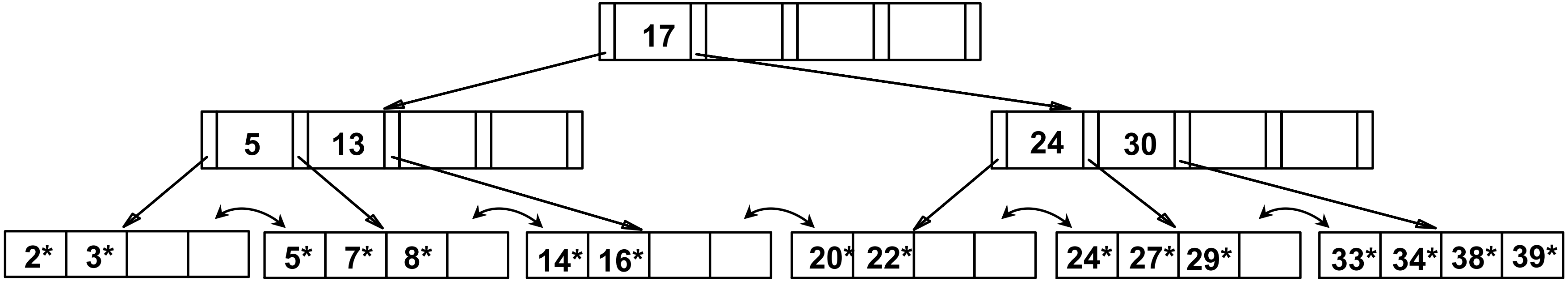
1. start at root, find leaf **L** of entry
2. remove the entry, if it exists
   - if **L** is at least half-full, done!
   - else
     - try to re-distribute, borrowing from sibling
       - adjacent node with same parent as L
     - if that fails, merge L into sibling
   - if merge occured,
     must delete L from parent of L

merge could propagate to root

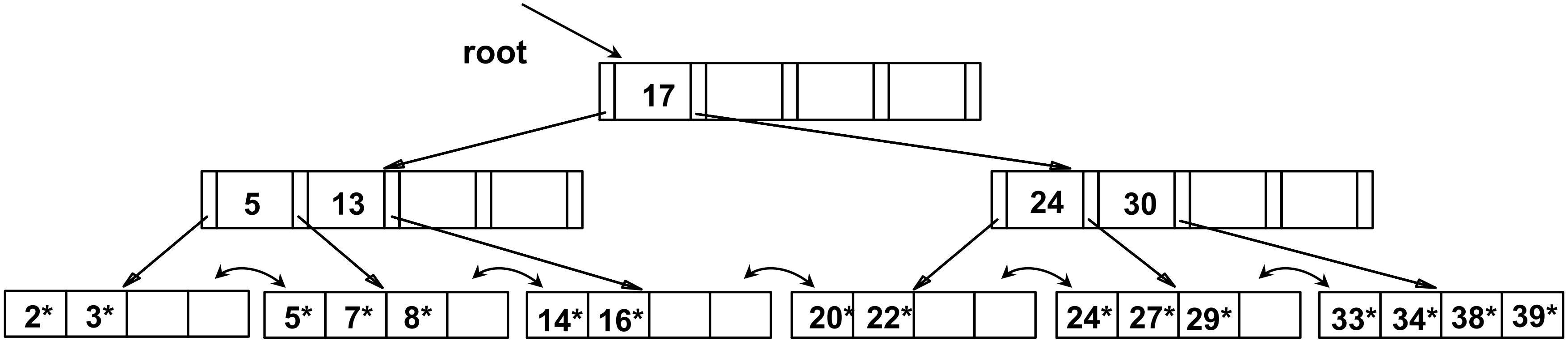
example b+ tree



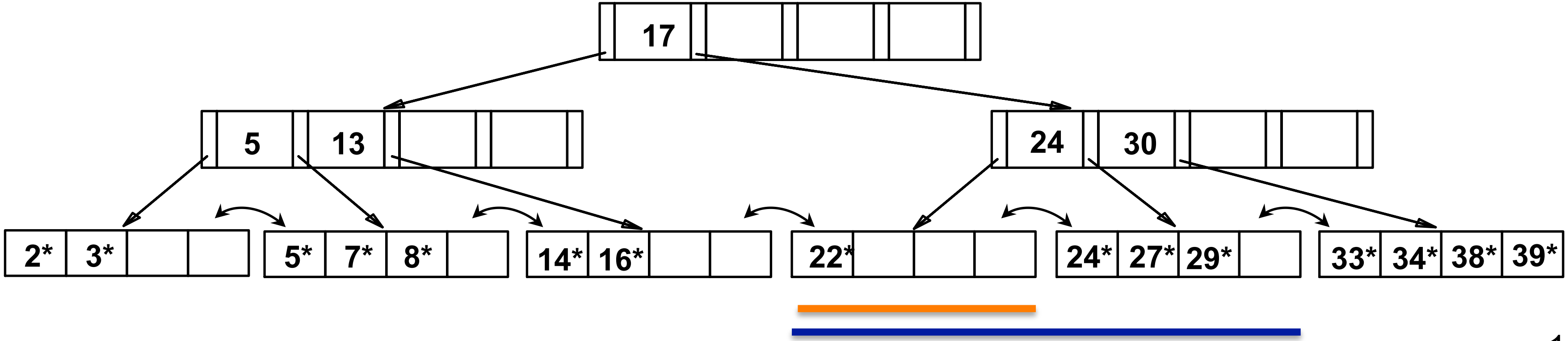
delete 19*



example b+ tree

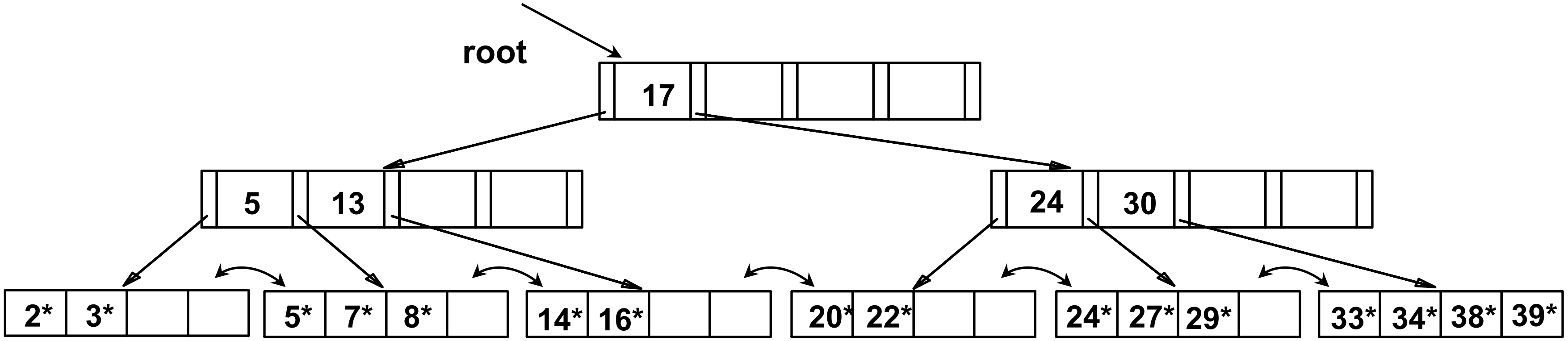


delete 20*

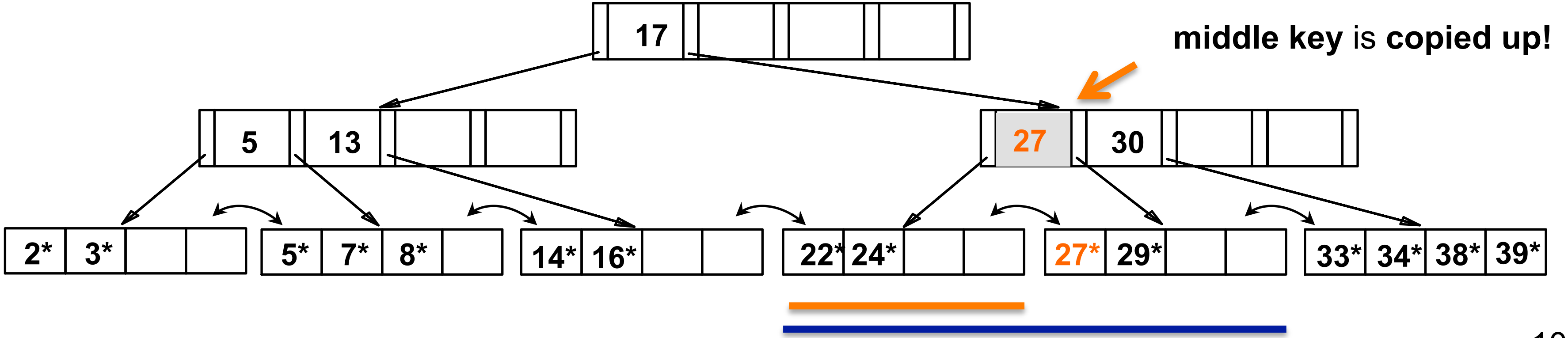


occupancy below 50%, redistribute!

example b+ tree

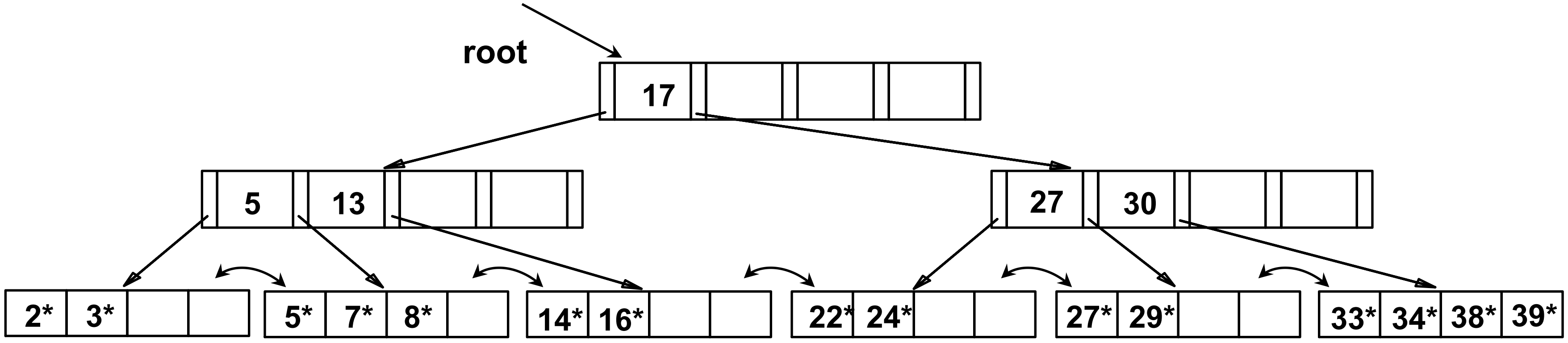


delete 20*

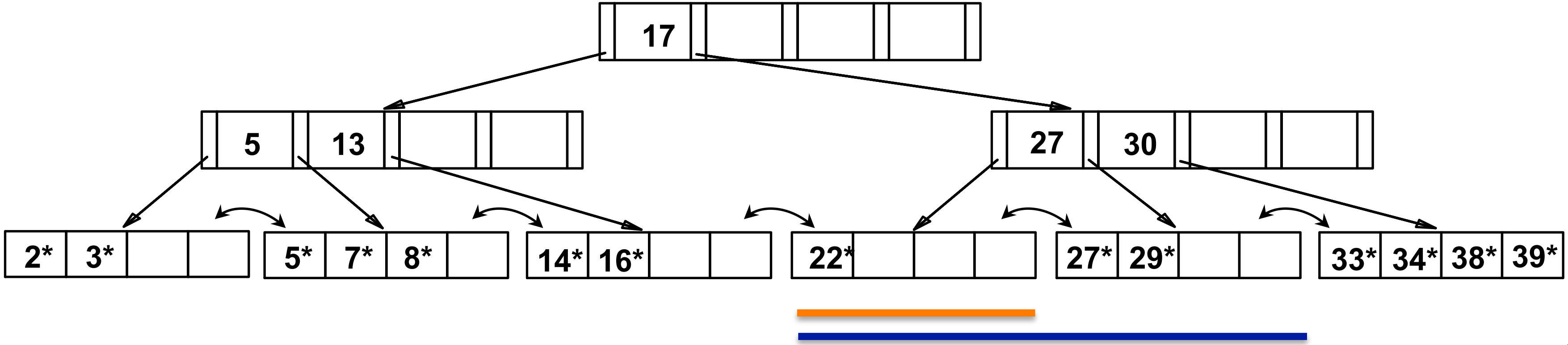


occupancy below 50%, redistribute!

example b+ tree

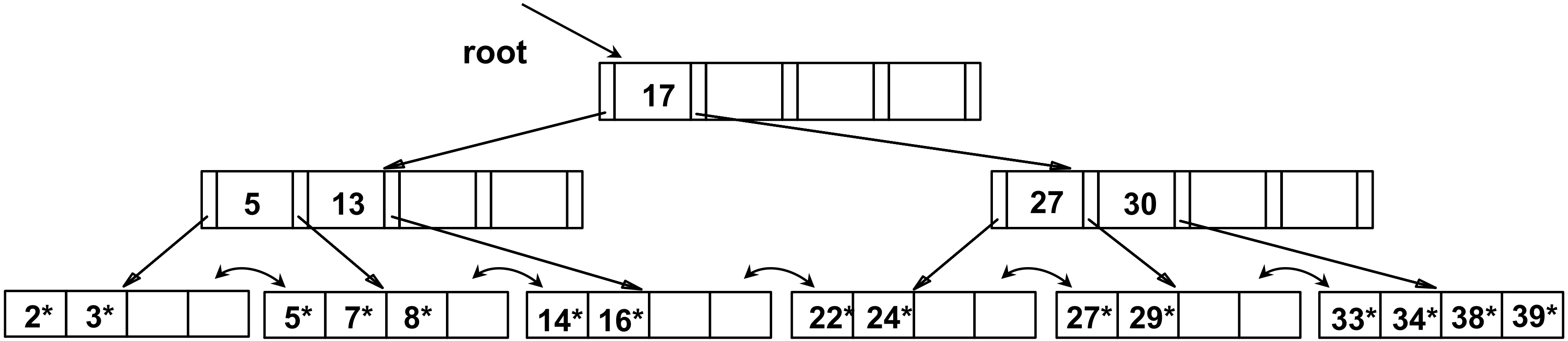


delete 24*



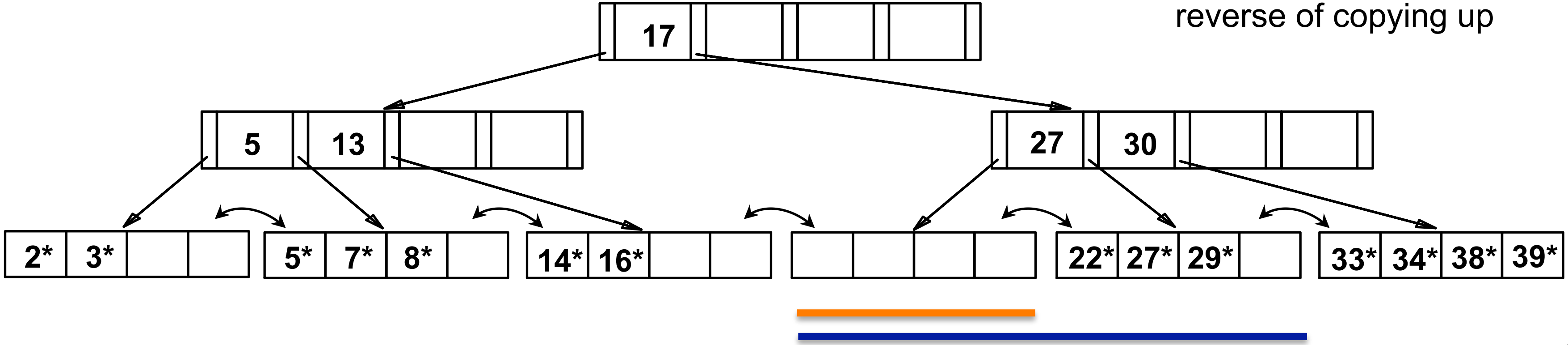
occupancy below 50%, merge!

example b+ tree



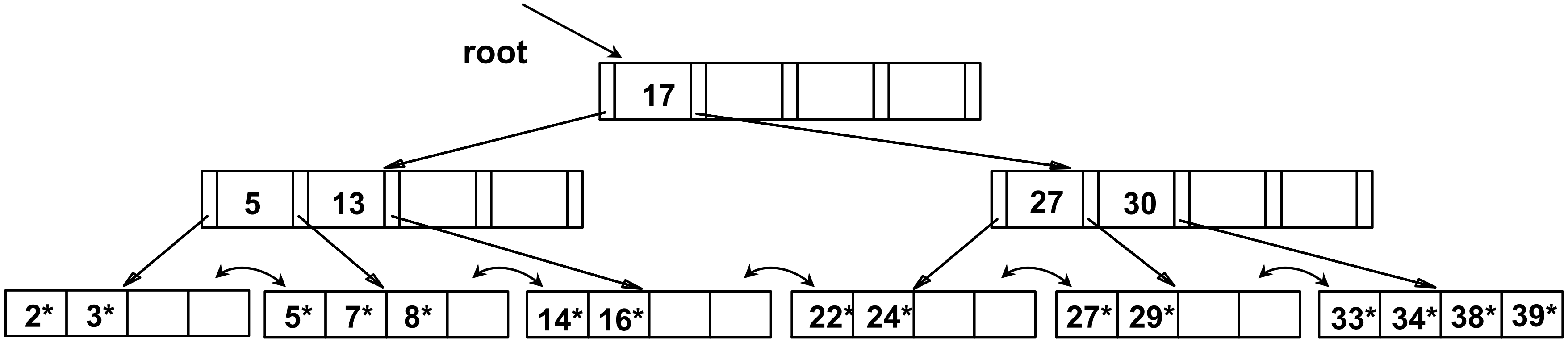
delete 24*

delete from parent!
reverse of copying up



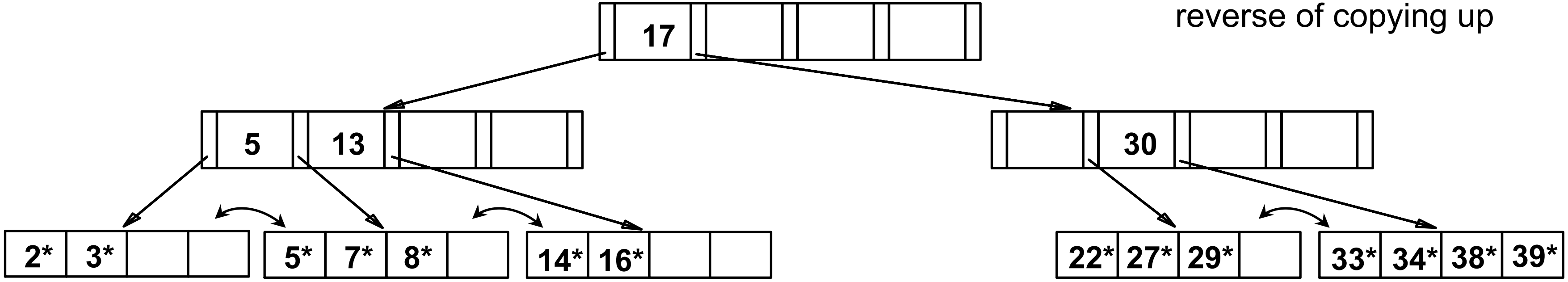
occupancy below 50%, merge!

example b+ tree

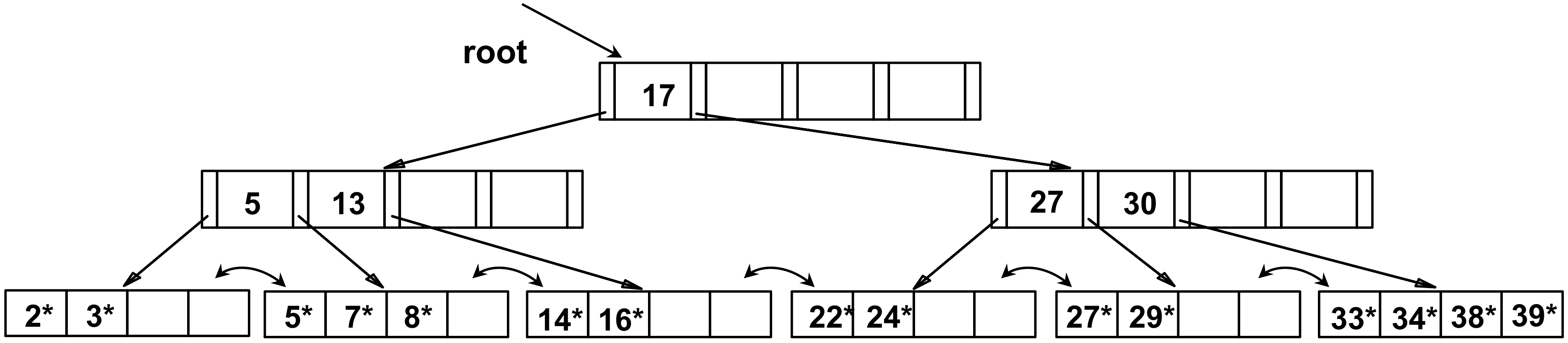


delete 24*

delete from parent!
reverse of copying up

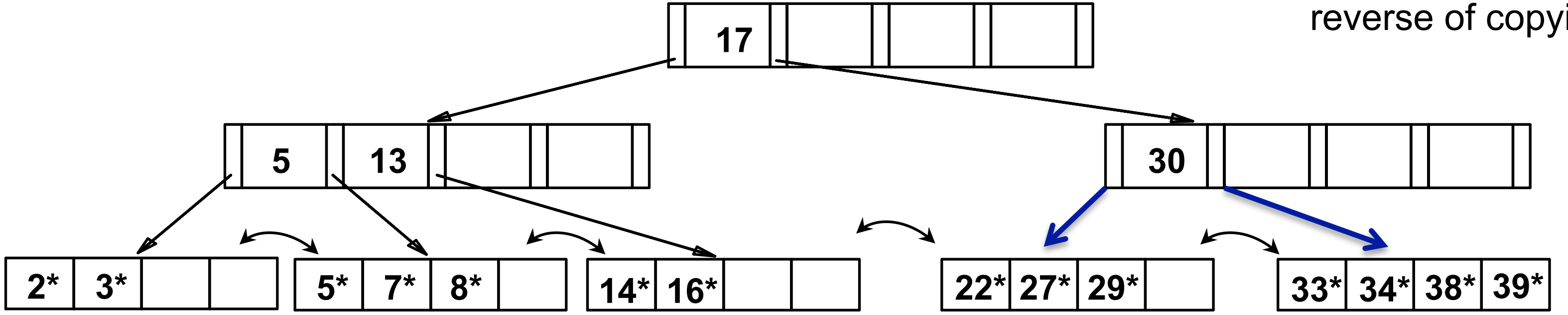


example b+ tree

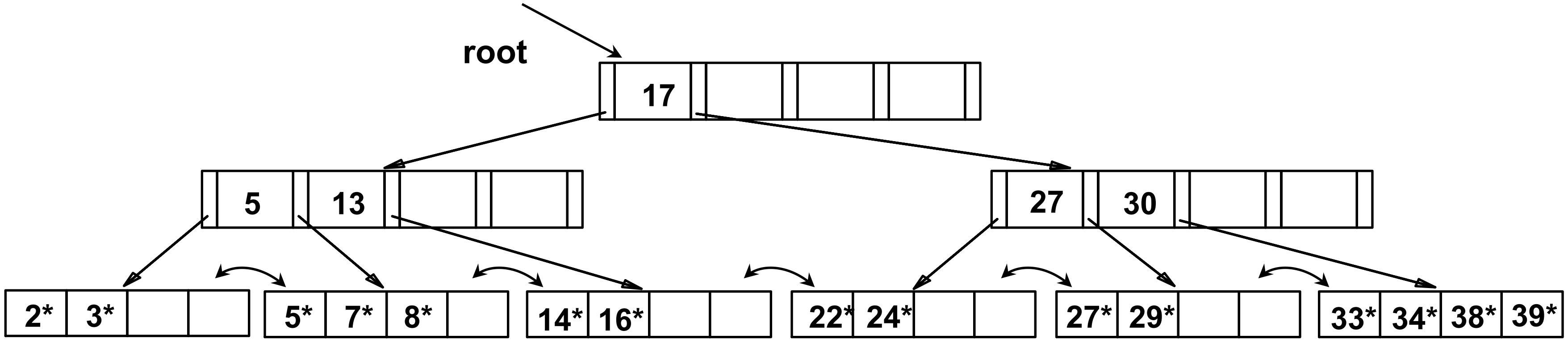


delete 24*

delete from parent!
reverse of copying up

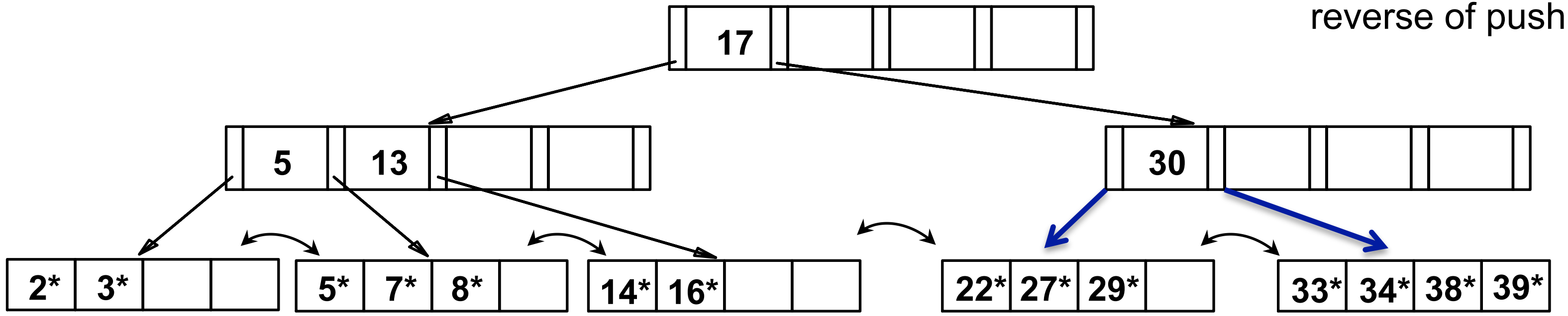


example b+ tree

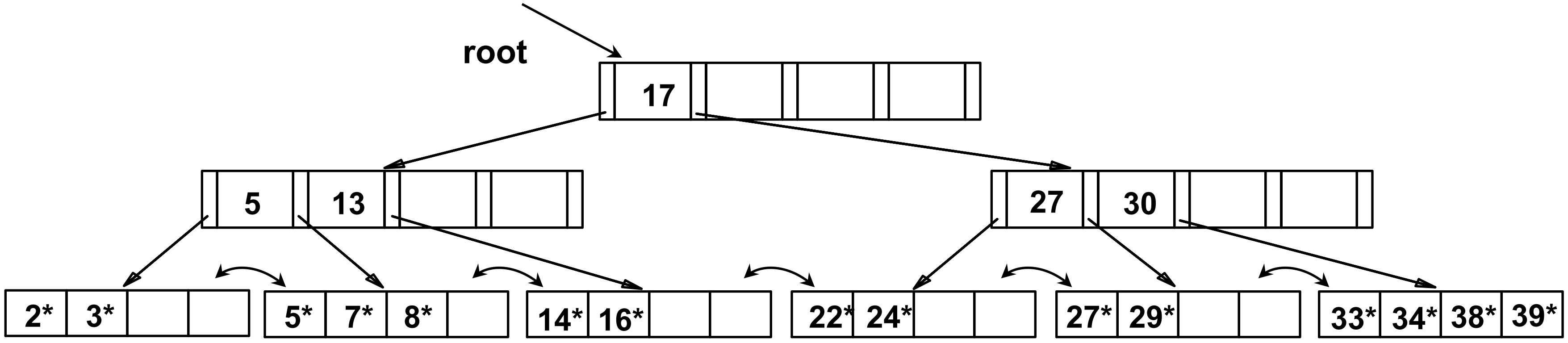


delete 24*

merge children of root!
reverse of pushing up

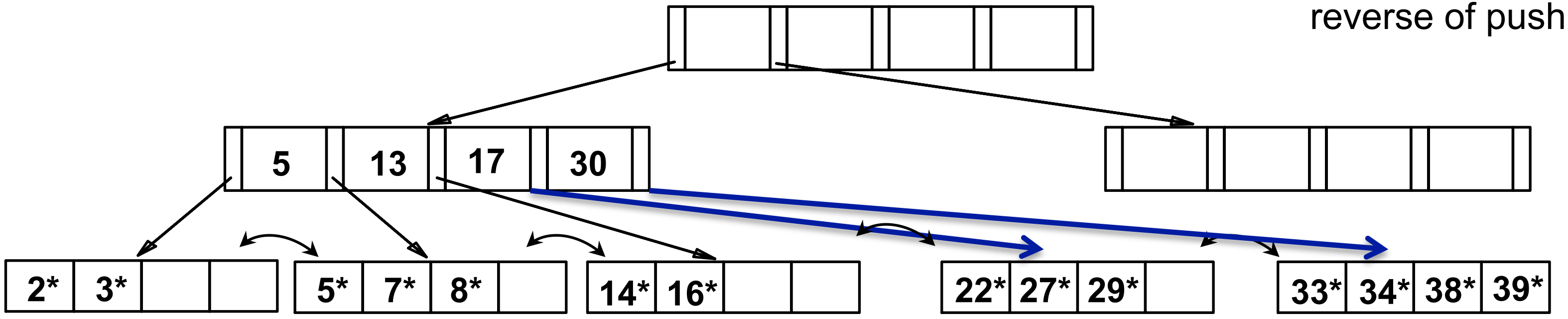


example b+ tree

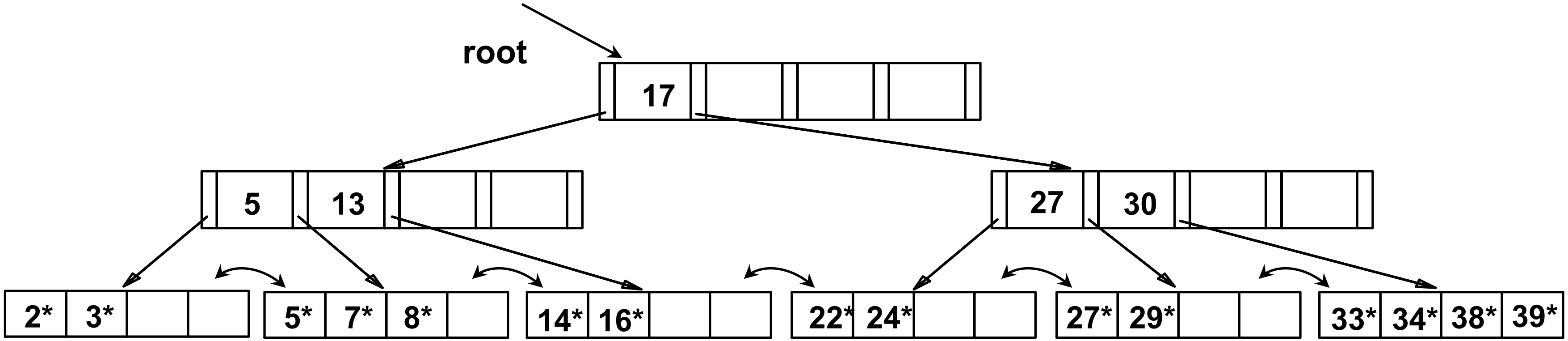


delete 24*

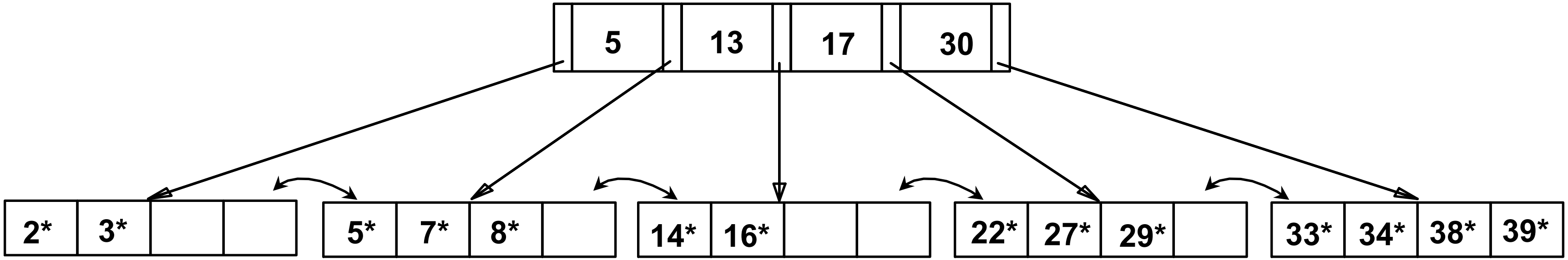
merge children of root!
reverse of pushing up



example b+ tree



delete 24*

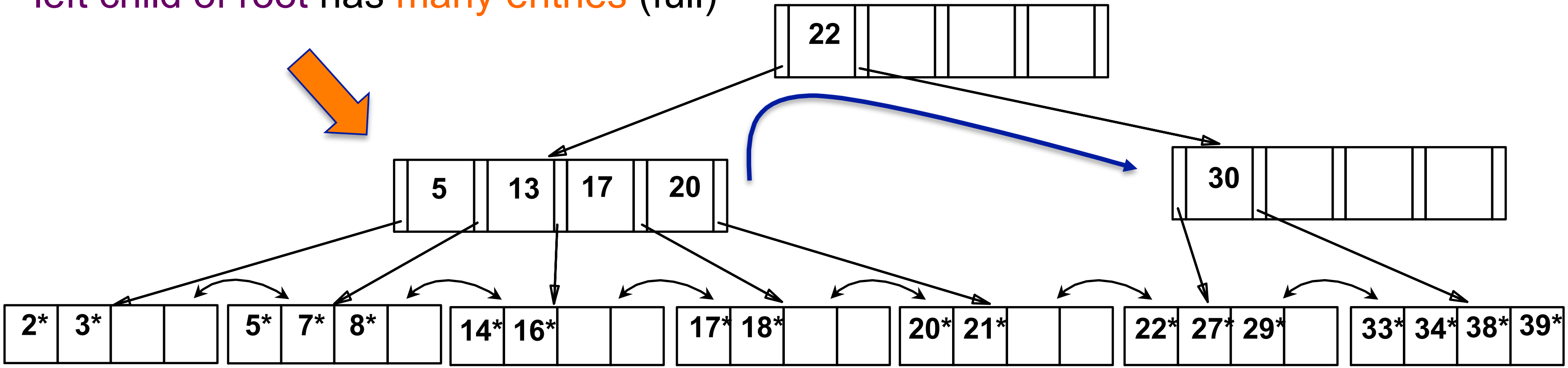


example b+ tree

during deletion of 24* -- different example

can redistribute entries of index nodes
pushing through root splitting entry

left child of root has many entries (full)

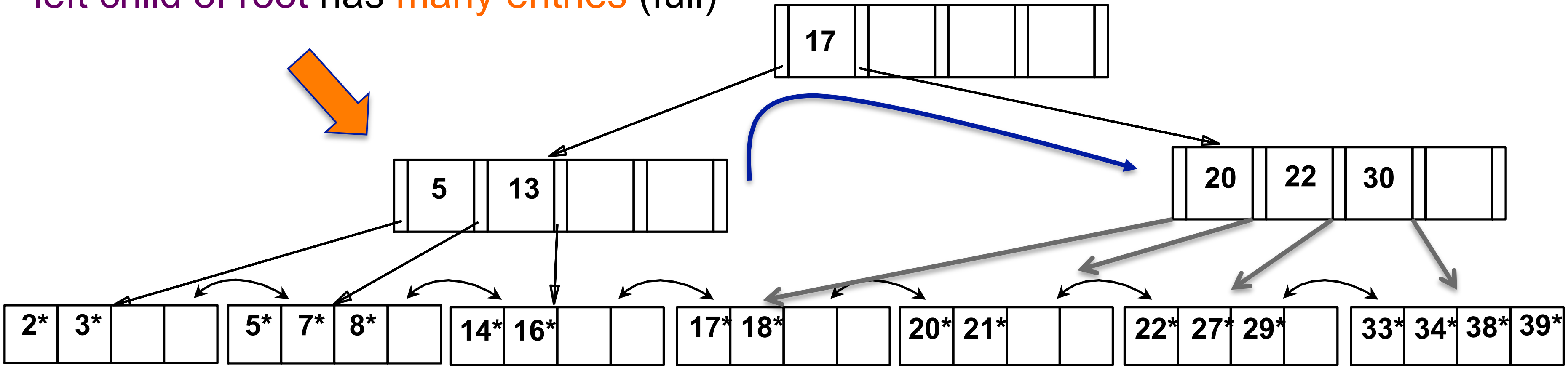


example b+ tree

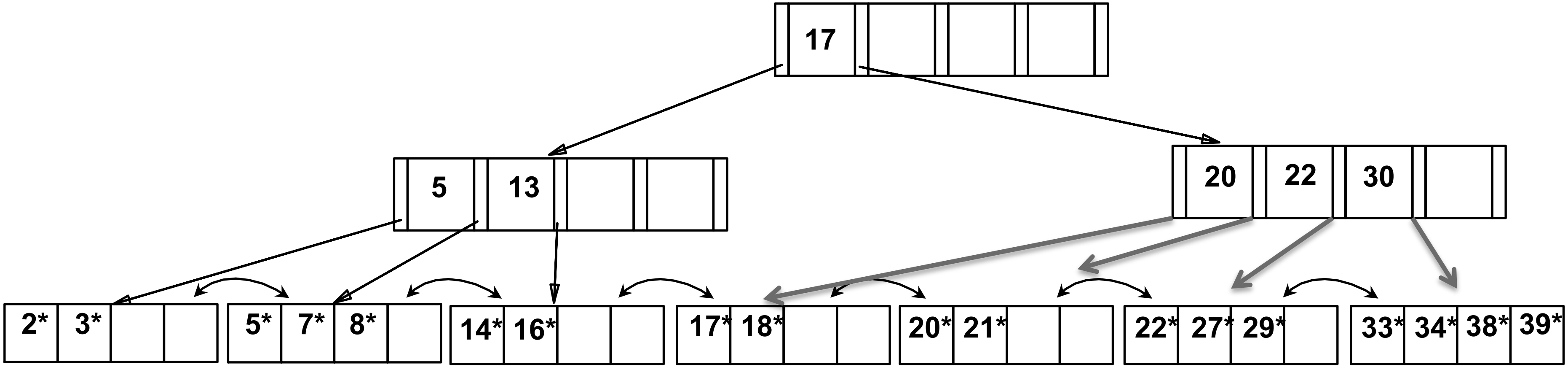
during deletion of 24* -- different example

can redistribute entries of index nodes
pushing through root splitting entry

left child of root has many entries (full)



example b+ tree



linear hashing

linear hashing

dynamic hashing

uses overflow pages; **no directory**

splits a bucket in round-robin fashion

when an overflow occurs

$M = 2^{level}$: number of buckets at beginning of round

pointer next $\in$ [0, M) points at next bucket to split

already next - 1 ‘split-image’ buckets appended to original M

linear hashing

to allocate entries, use

$\mathbf{H_0(key)}$ = h(key) mod M, or

$\mathbf{H_1(key)}$ = h(key) mod 2M

i.e., level or level+1 least significant bits of h(key)

to allocate bucket for key

first use $\mathbf{H_0(key)}$

if $\mathbf{H_0(key)}$ is less than next

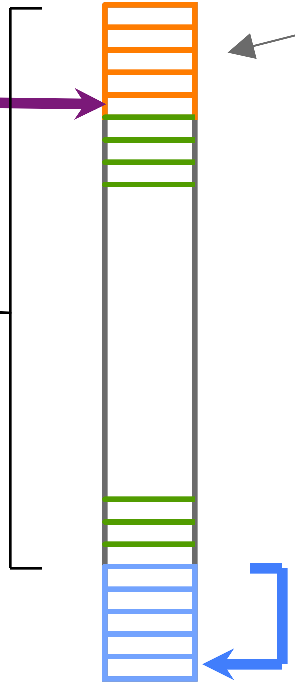
then it refers to a split bucket

use $\mathbf{H_1(key)}$ to determine if it refers

to original or its split image

linear hashing

in the middle of a round...



bucket to be split **next**

buckets already split in this round

M buckets that existed at the beginning of this round; this is the range of $\mathbf{H_0}$

if H_0(key) is in this range, then must use H_1(key) to decide if entry is in split image bucket

split image buckets created through splitting of other buckets in this round

> is a directory necessary?

linear hashing inserts

insert

find bucket by applying $\mathbf{H_0}$ / $\mathbf{H_1}$ and insert if there is space

if bucket to insert into is **full**:

add overflow page, insert data entry,

split next bucket and increment next

since buckets are split round-robin,

long overflow chains don't develop!

example - insert h(r) = 43

on split, **H_1** is used to redistribute entries