Hashing and Sorting

Carsten Schürmann

Example

Student

cpr	name	address
140298-1234	Jesper	Copenhagen
041297-5367	Nikoline	Aarhus
151197-2352	Claus	Dragør
050596-1142	Martin	Copenhagen

- Find the a student with CPR number 151197-2352

Idea

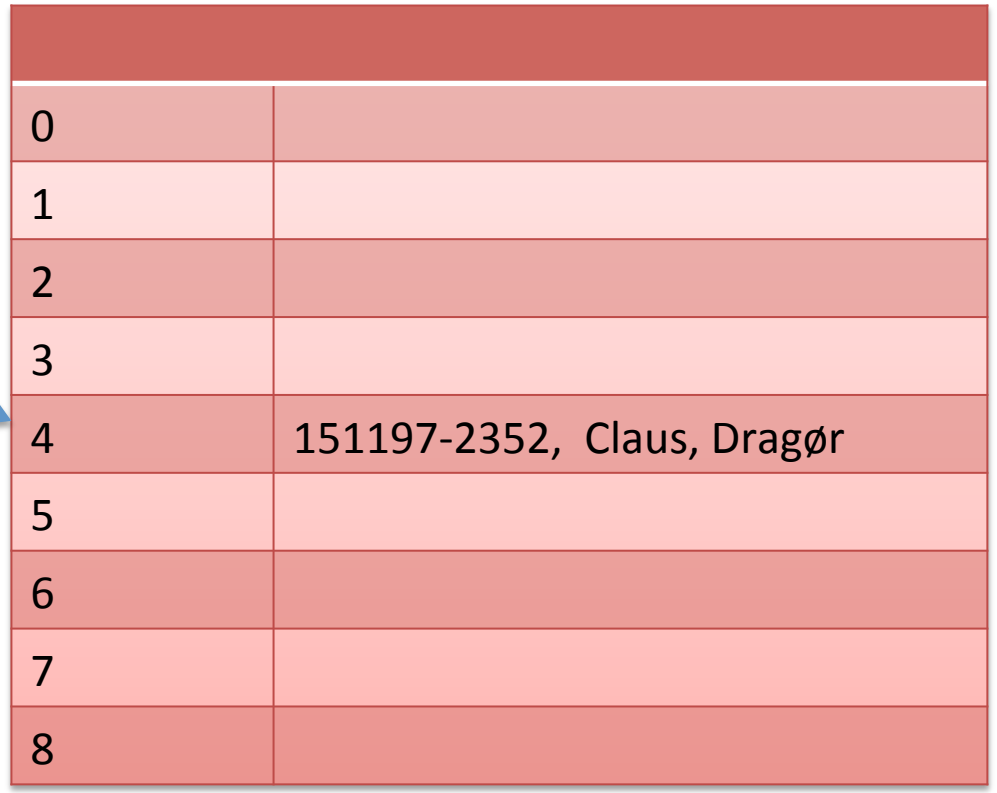
Compute the location from the key!!!

Collision resolution method

151197-2352

Hash function

Utilization



0	
1	
2	
3	
4	151197-2352, Claus, Dragør
5	
6	
7	
8	

Hash table size

Hash bucket

Hash Functions

Notation: $h(x)$

Goal: Uniform distribution over buckets

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

Example: $h(cpr) = cpr \bmod 1000$

Choice of Parameters

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

- *a* abritrary
- *b* abritrary
- *M* prime number

Concern: *M = 2*, last digit of cpr reveals sex

Size of the Hash Table

Example:

50000 students, 10 students per page

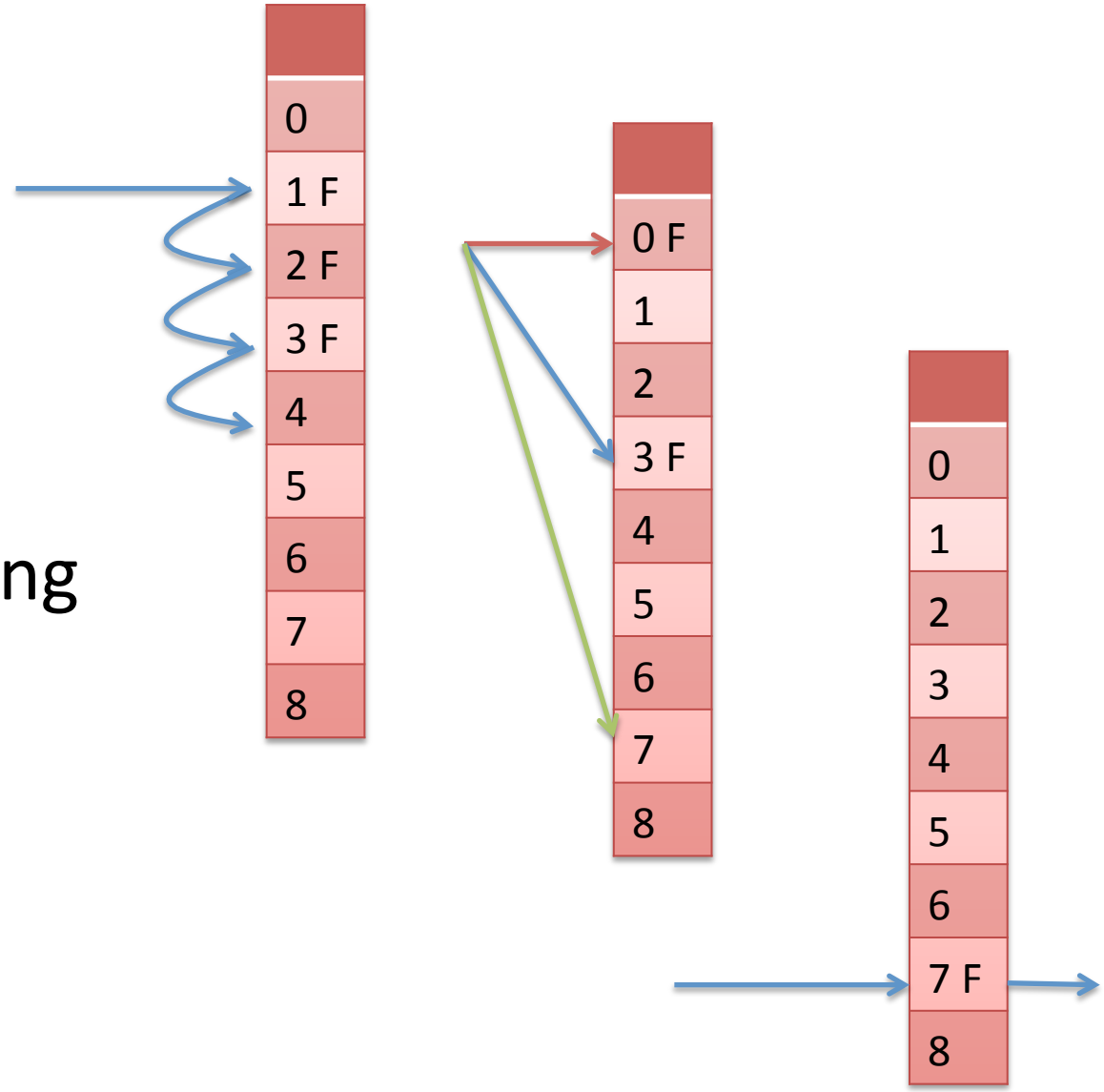
Utilization: 90%

In average 50000/10/0.9 = 5555

Pick *M* as closest prime to 5555.

Collision Resolution

- Linear probing
- Re-hashing
- Separate chaining



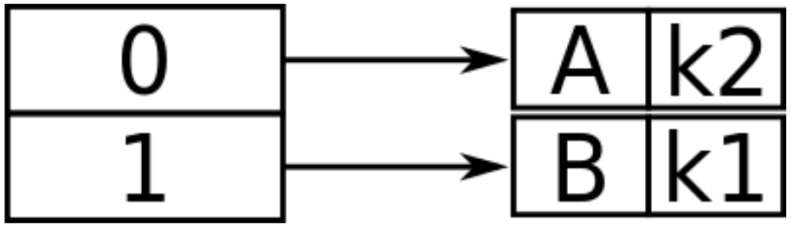
Extendible Hashing

Keys are turned into bitstrings

*h(*k1*)* = 100100

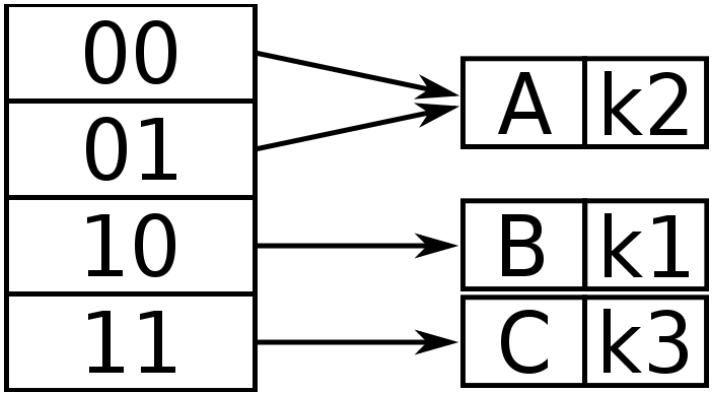
*h(*k2*)* = 010110

*h(*k3*)* = 110110



Directory

Inserting k3, doubling the buckets



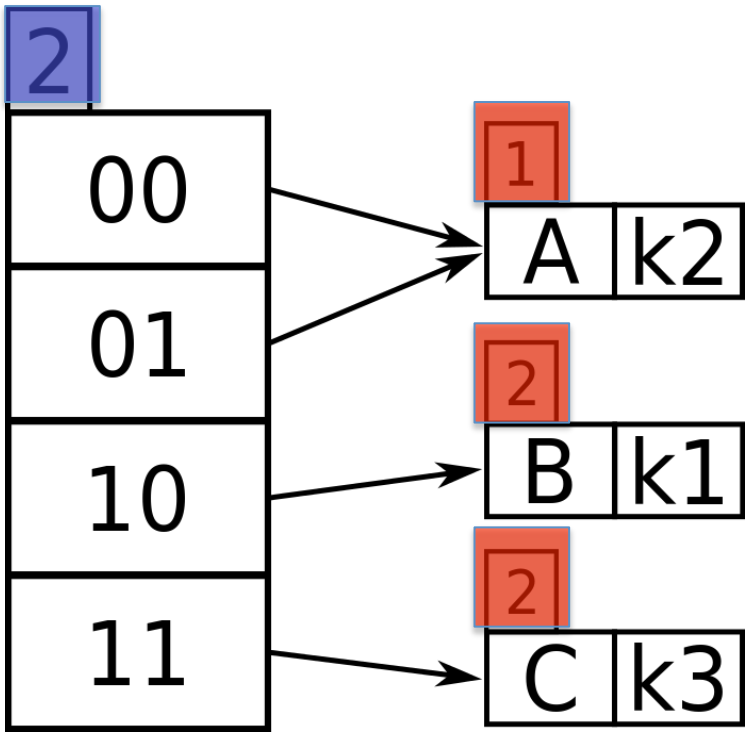
Next, insert $h(k4) = 011110$

Idea of Depth

*h(*k4*)* = 011110

Global depth

Local depth

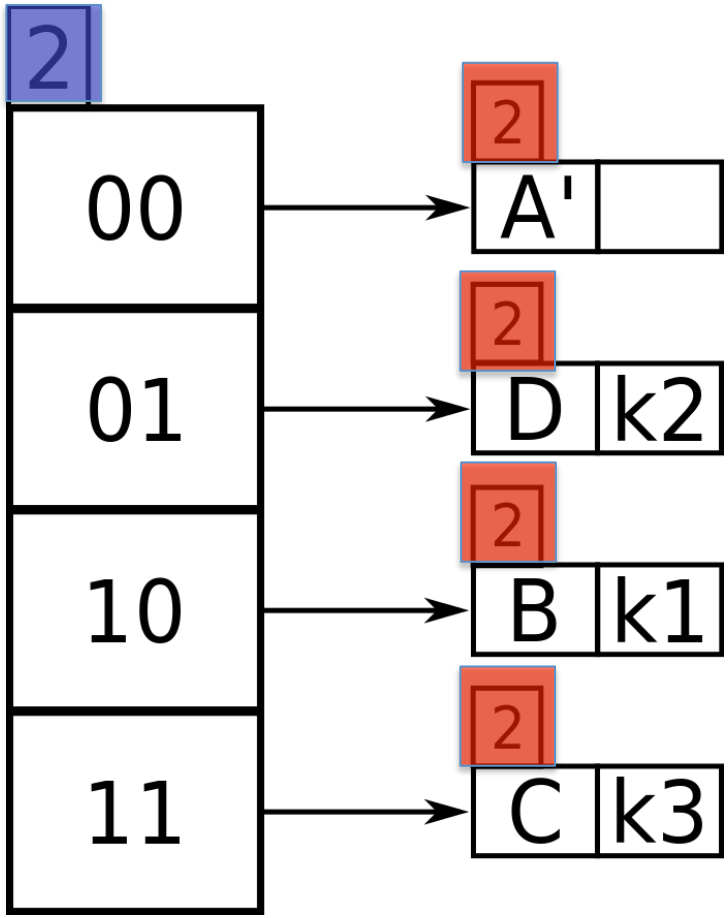


Overfull 01 bucket

*h(*k4*)* = 011110

Global depth

Local depth

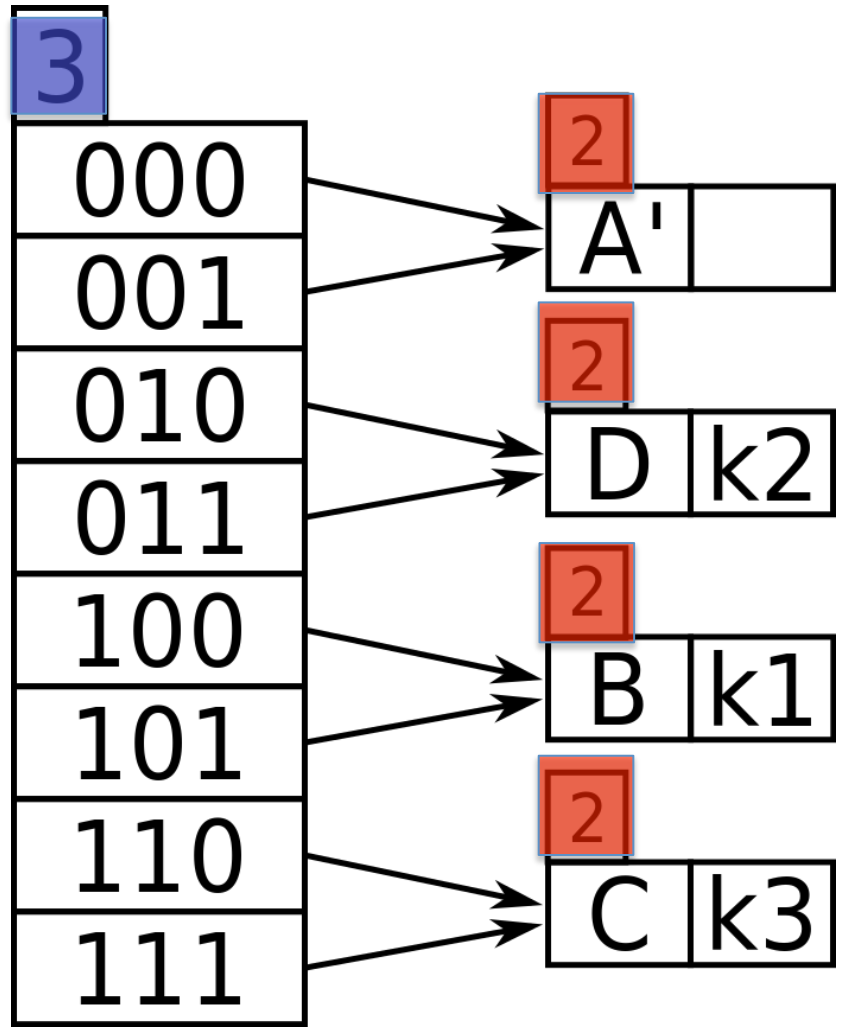


Splitting

*h(*k4*)* = 011110

Global depth

Local depth

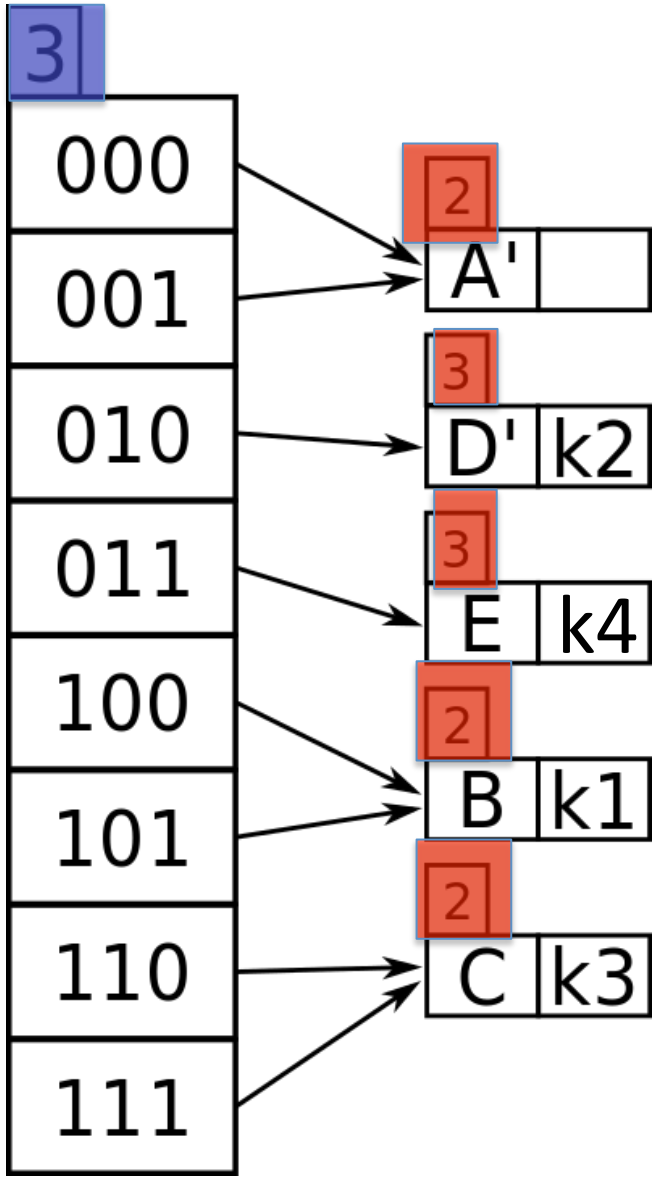


Insertion

*h(*k4*)* = 011110

Global depth

Local depth



Summary

Extendible hashing

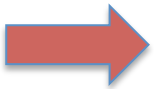
- Directory doubles on demand
- Shrink on demand
- Local and global depth as indicators
  - If both are the same, double

Linear Hashing

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content
0	8
1	13

Utilization: 50%

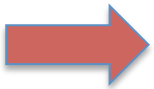
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content
0	8
1	13

Utilization: 50%

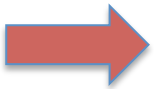
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content
0	8, 10
1	13

Utilization: 75%

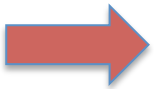
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content
0	8, 10
1	13, 15

Utilization: 100%

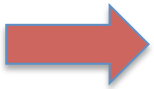
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content	
0	8, 10	mod 4
1	13, 15	mod 2
2		mod 4

Utilization: 100%

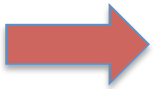
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content	
0	8	mod 4
1	13, 15	mod 2
2	10	mod 4

Utilization: 100%

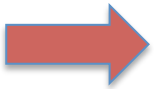
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content	
0	8	mod 4
1	13, 15	mod 2
2	10	mod 4

Utilization: 4/6

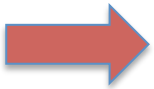
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content	
0	8	mod 4
1	13, 15	mod 2
2	10	mod 4

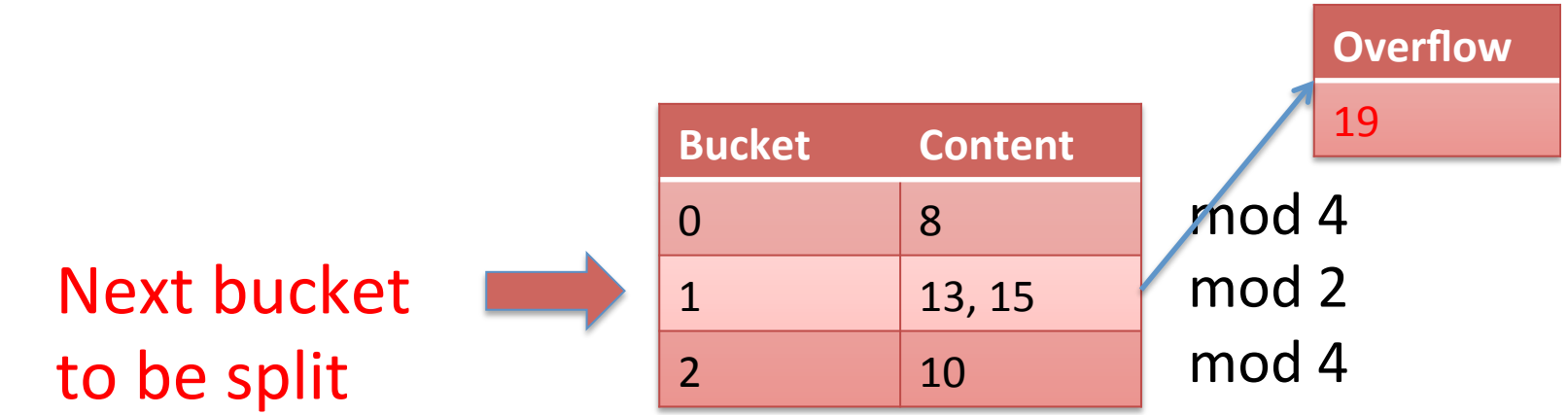
Utilization: 4/6

Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2



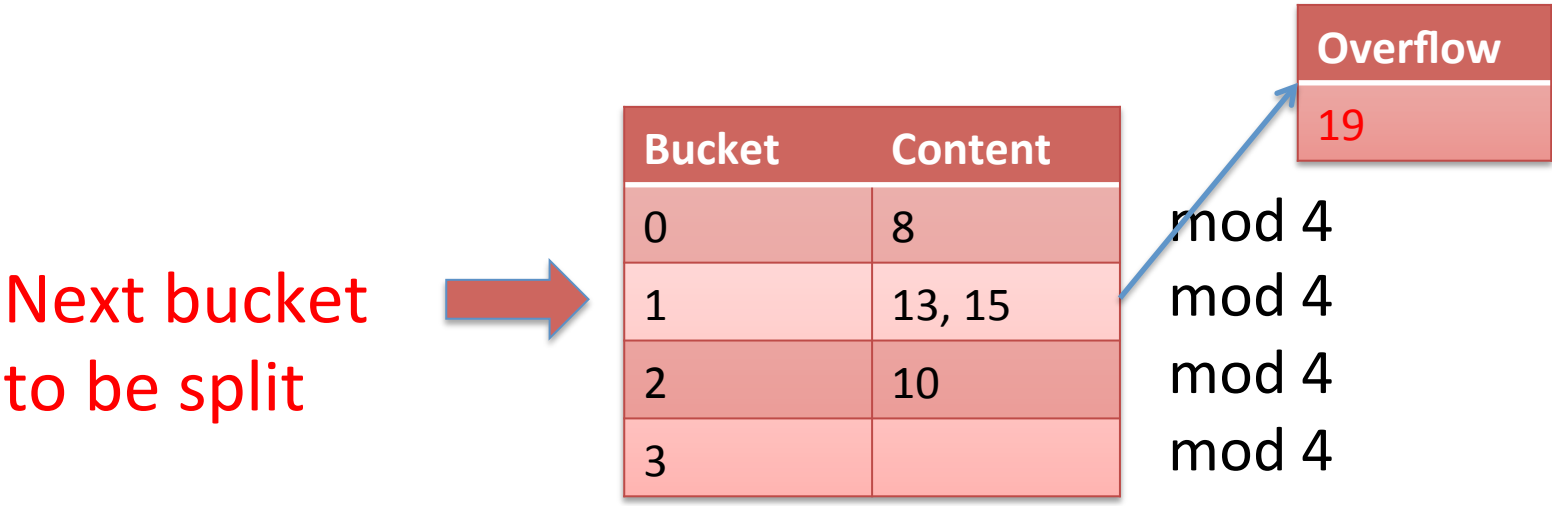
Utilization: 5/6

Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2



Utilization: 5/6

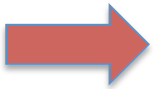
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content	
0	8	mod 4
1	13	mod 4
2	10	mod 4
3	15, 19	mod 4

Utilization: 5/8

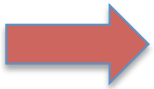
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content	
0	8	mod 4
1	13	mod 4
2	10	mod 4
3	15, 19	mod 4

Utilization: 5/8

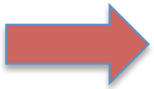
Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2

Next bucket to be split



Bucket	Content	
0	8	mod 4
1	13	mod 4
2	10, 22	mod 4
3	15, 19	mod 4

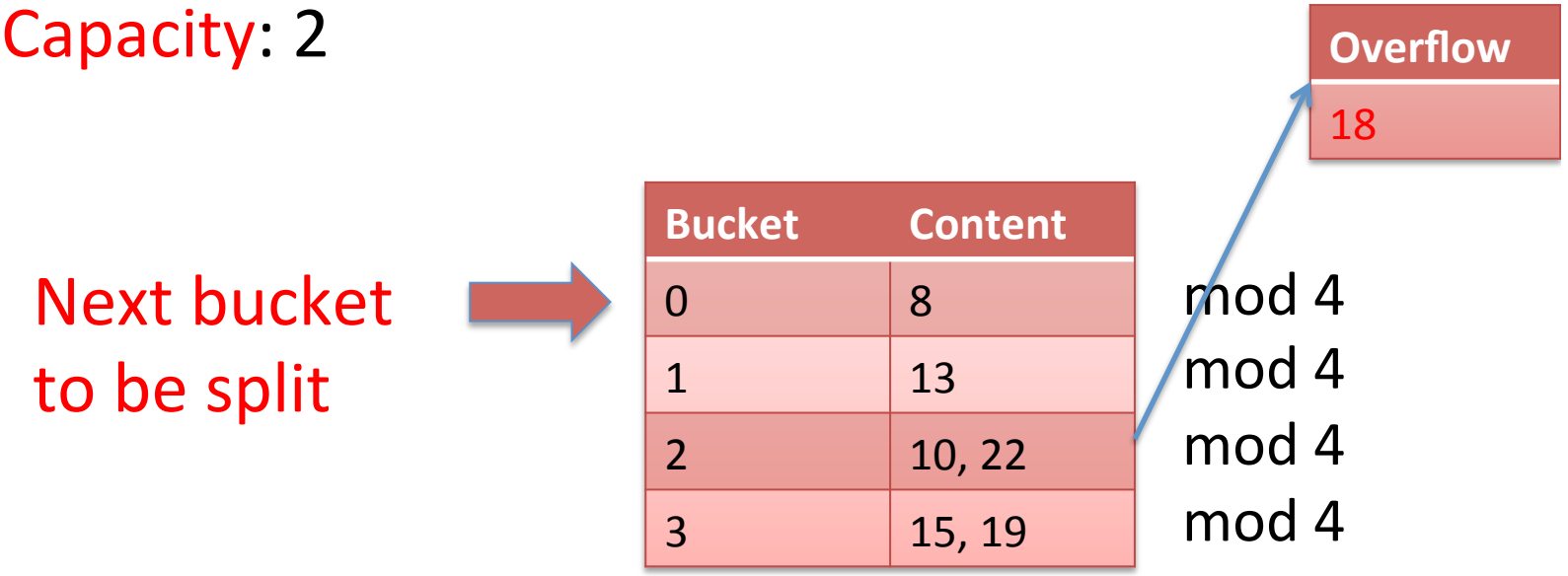
Utilization: 6/8 = 75%

Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2



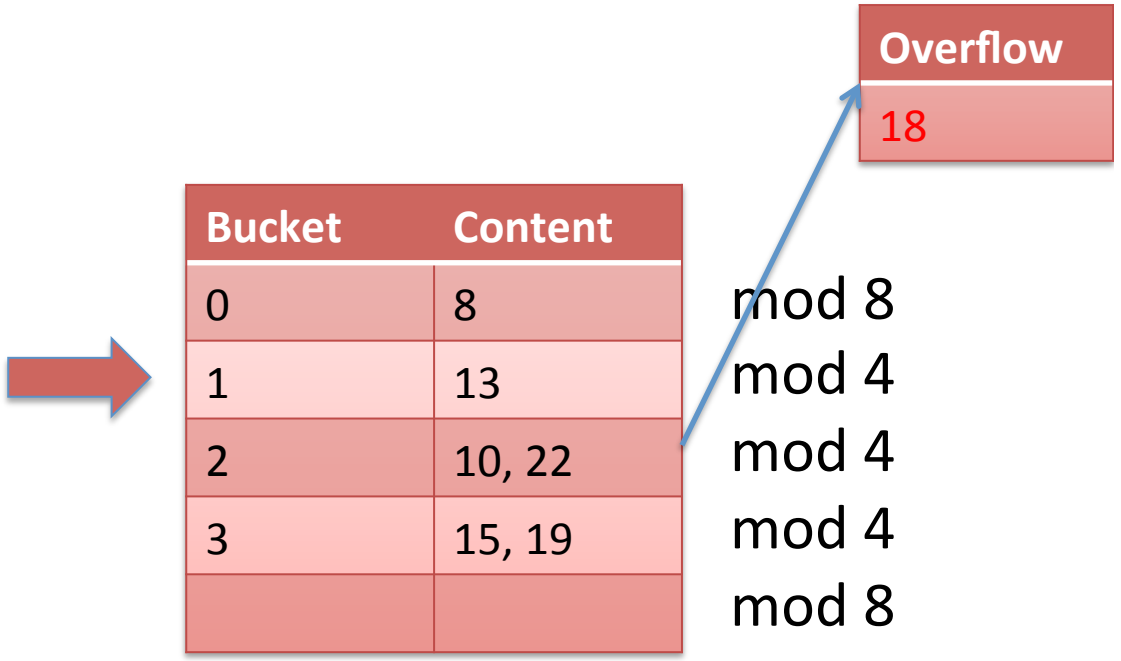
Utilization: 7/8

Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2



Next bucket to be split

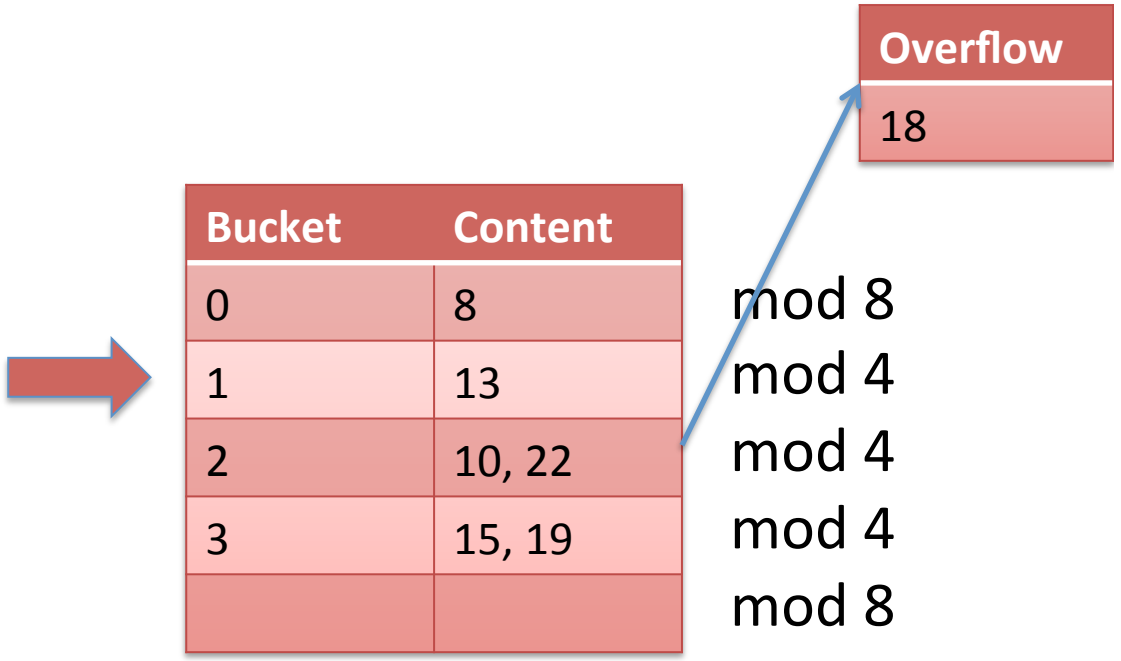
Utilization: 7/16

Splitting policy: Hash table > 75% full

Insert 10, 15, 19, 22, 18

Number of Buckets: 2

Bucket Capacity: 2



Next bucket to be split

Utilization: 7/16

Splitting policy: Hash table > 75% full

Summary

- Linear hashing
  - Algorithms for search, insertion and deletion
  - Growth and contracts one bucket at the time

B+ trees versus Hashing

- Speed search
  - Exact match queries
  - Range queries
- No reorganization

- Speed
  - Very fast on exact match queries
- No good behavior on
  - Range queries
  - Ordered by queries

Hash Index Support

- Some DBMS' support hash indexes

```
create index test3
on Movie(id)
using hash;
```

- But not mysql.

Sorting

When To Sort

- Duplicate elimination
- `select … order by`
- `select … group by`
- Sort-merge join from last lecture involves sorting

What to do when the data does not fit into memory?

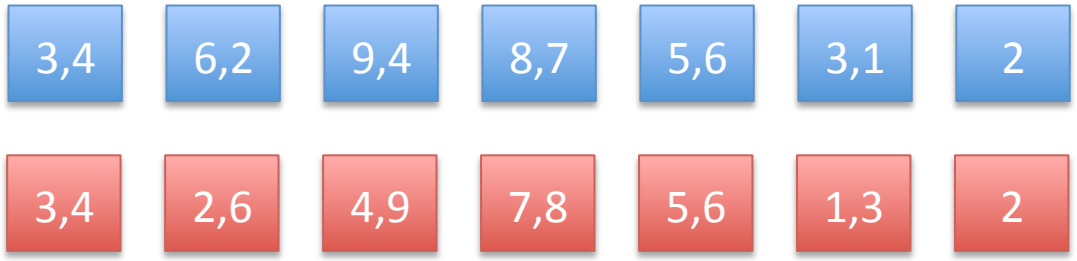
Two Way External Sort

- Files consist of *n* pages
- Buffer size *b*

Algorithm

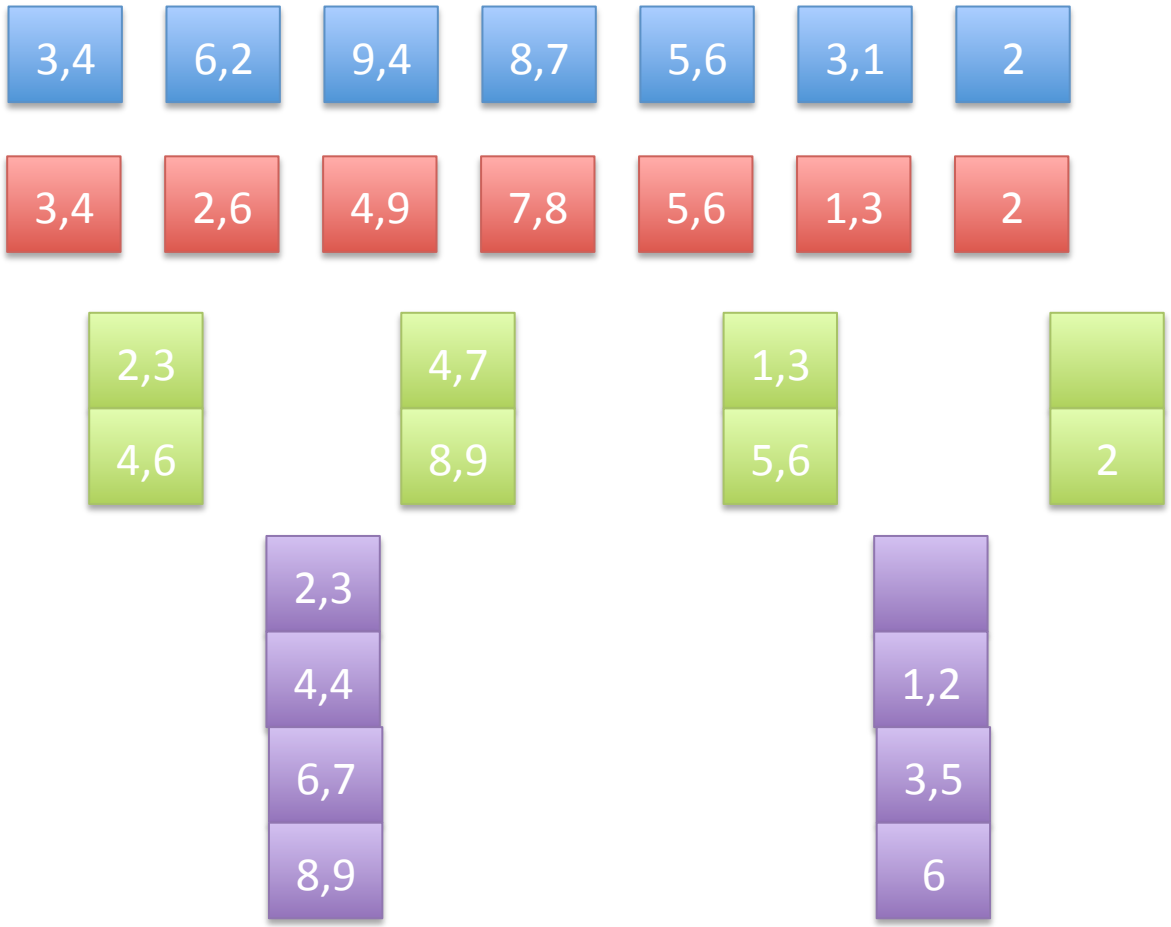
- Pass 0 : Read a page, sort it, Write it
- Pass *i* :
  - Read two input pages (in one time step)
  - Write one output page (in two time steps)

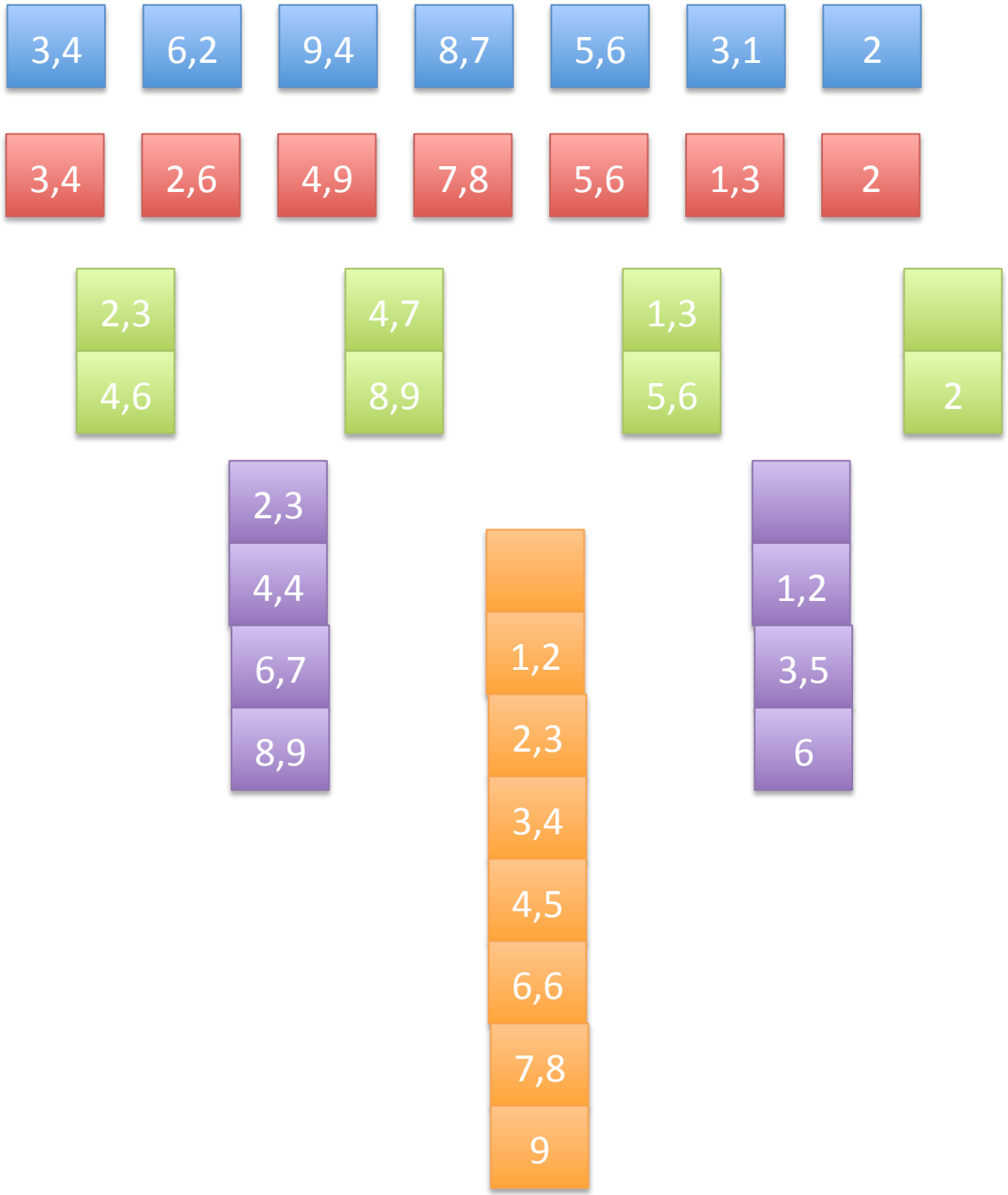
Step 0



Step 1

Step 2


3,4
6,2
9,4
8,7
5,6
3,1
2
3,4
2,6
4,9
7,8
5,6
1,3
2
2,3
4,6
4,7
8,9
1,3
5,6
2
2,3
4,4
6,7
8,9
1,2
3,5
6


3,4
6,2
9,4
8,7
5,6
3,1
2
3,4
2,6
4,9
7,8
5,6
1,3
2
2,3
4,6
4,7
8,9
1,3
5,6
2
2,3
4,4
6,7
8,9
1,2
2,3
3,4
4,5
6,6
7,8
9
1,2
3,5
6

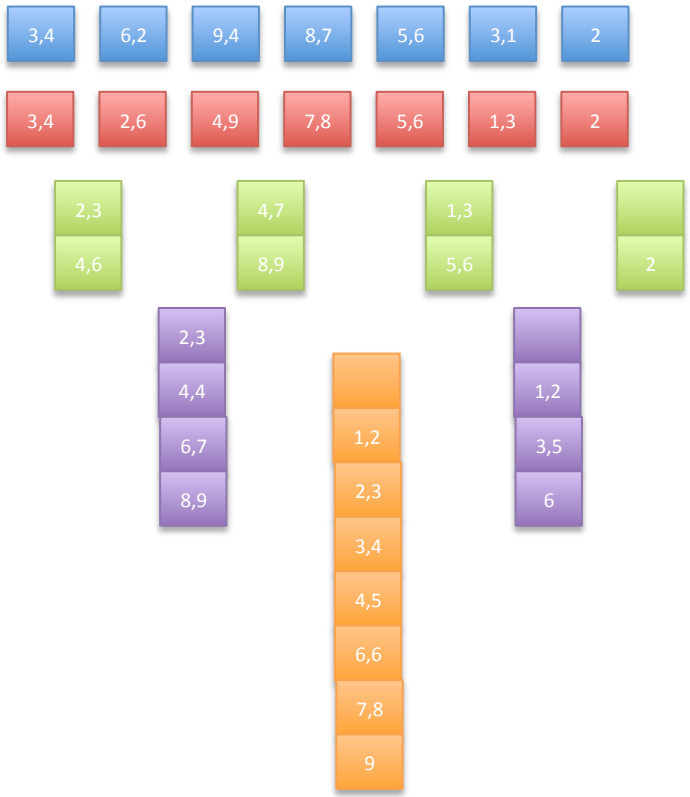
Step 3

- Each pass:
  - Read each page
  - write each page
- Number of passes

  $[\log_2 n] + 1$

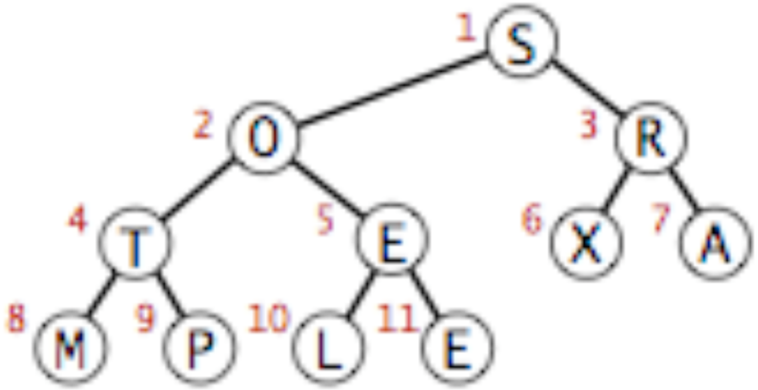
- I/O cost

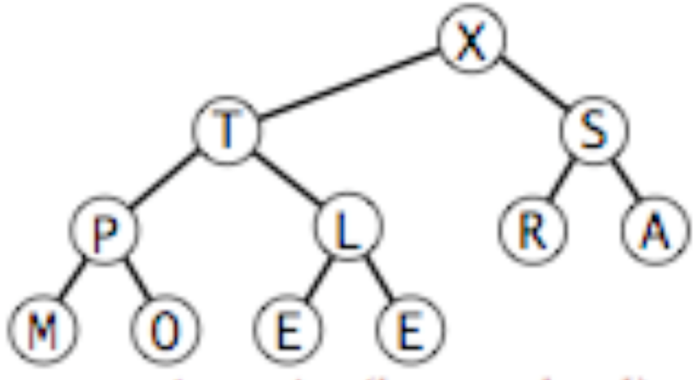
  $2\,n\,([\log_2 n] + 1)$
- Note: This scales also to combining n pages



Heap Order

A heap is a data structure where left and right subtree contain only smaller elements than the root.

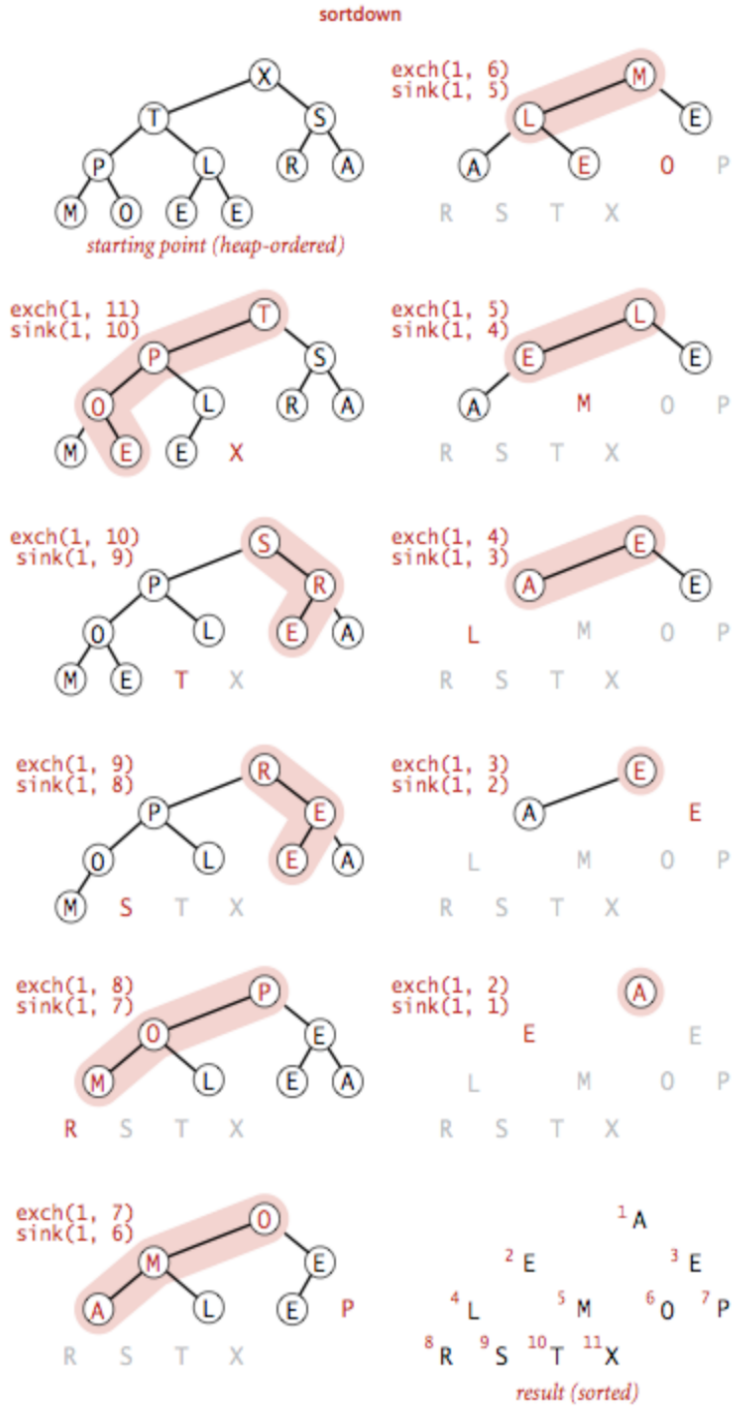




Heap Sort

- Alternative way to sort
- Used in databases
- Guarantees a more continuous use of b buffers
- Supports also double buffering

More about this in RG Chapter 13



State of the Art in Sorting

Current Champions 2014 (tie):

- TritonSort (UCSD)
  - 100 TB in 1,378 seconds (4.3 TB/min)
  - 186 Amazon EC2 i2.8xlarge nodes
- Apache Spark
  - 100 TB in 1,406 seconds (4.27 TB/min)
  - 207 Amazon EC2 i2.8xlarge nodes

More Info: http://sortbenchmark.org