Database System Architecture

Index Structures

Hector Garcia-Molina
Stijn Vansummeren

Index structure

- Any data structure that takes as input a search key and ***efficiently*** returns the collection of matching records

Sequential File

10	
20	

30	
40	

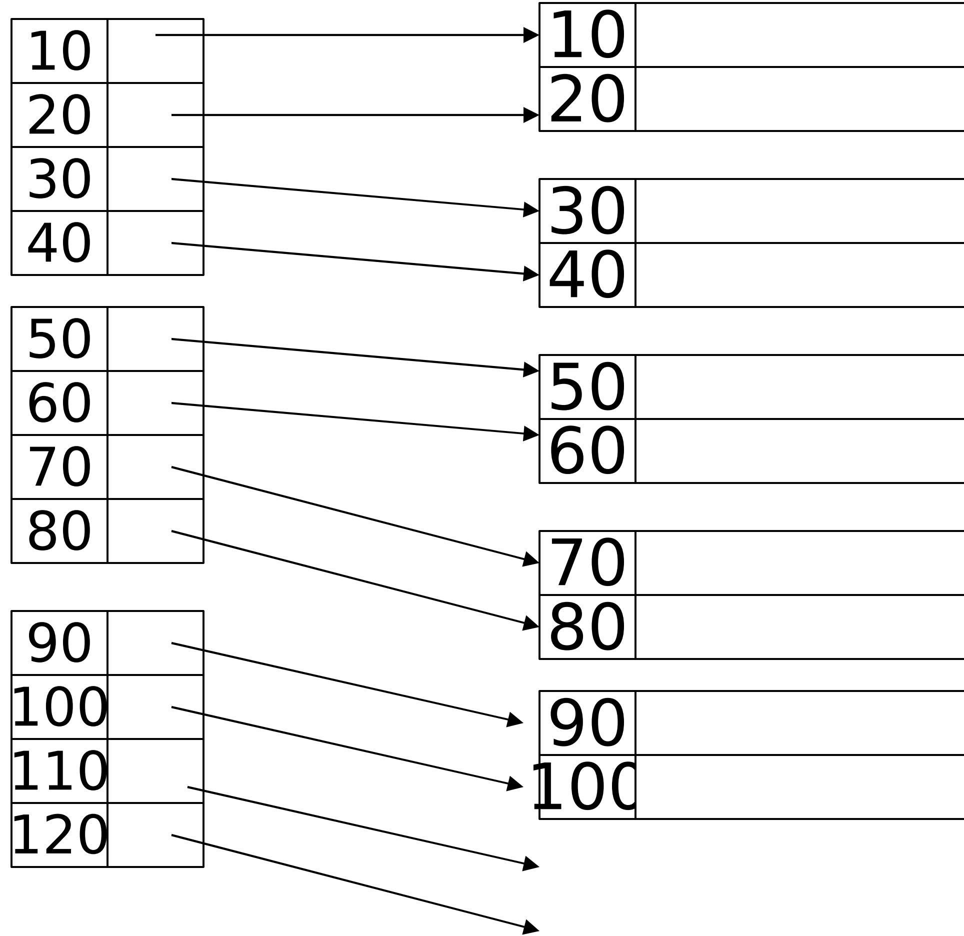
50	
60	

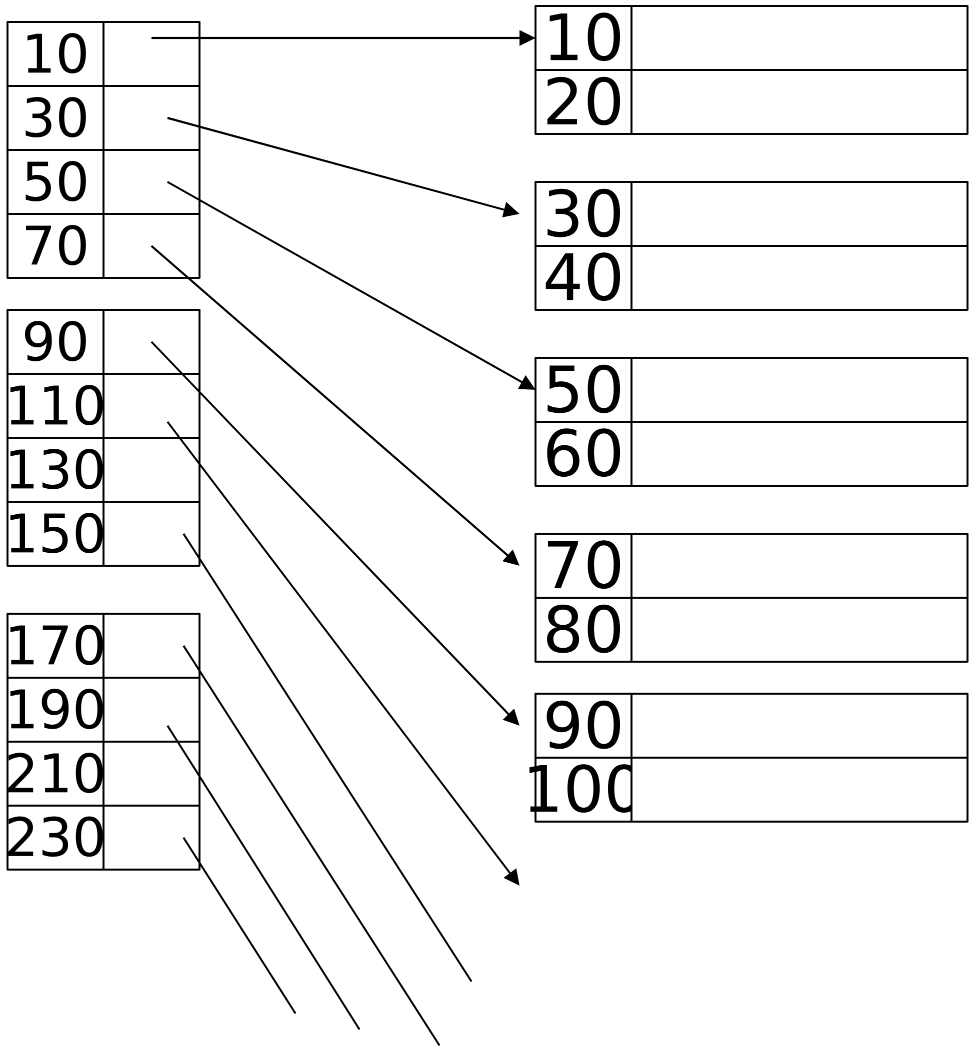
70	
80	

90	
100	

Dense Index

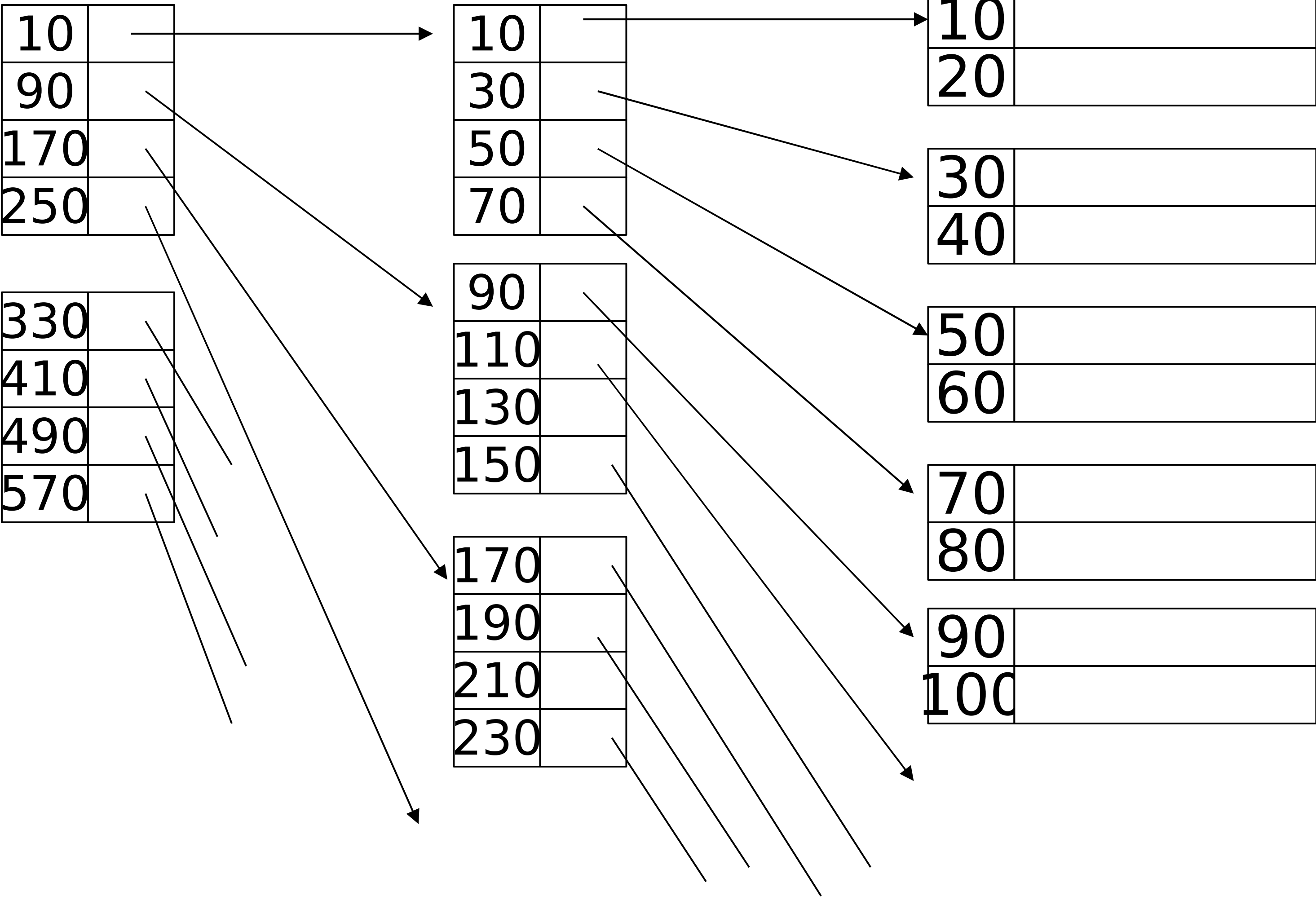
Sequential File




Sparse Index
Sequential File
10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100

Sparse 2nd level

Sequential File



Question:

- Can we build a dense, 2nd level index for a dense index?

Sparse vs. Dense Tradeoff

- Sparse: Less index space per record
  can keep more of index in
  memory
- Dense: Can tell if any record exists
  without accessing file

(Later:

- sparse better for insertions
- dense needed for secondary indexes)

Next:

- Duplicate keys
- Deletion/Insertion
- Secondary indexes

Duplicate keys

10	
10	

10	
20	

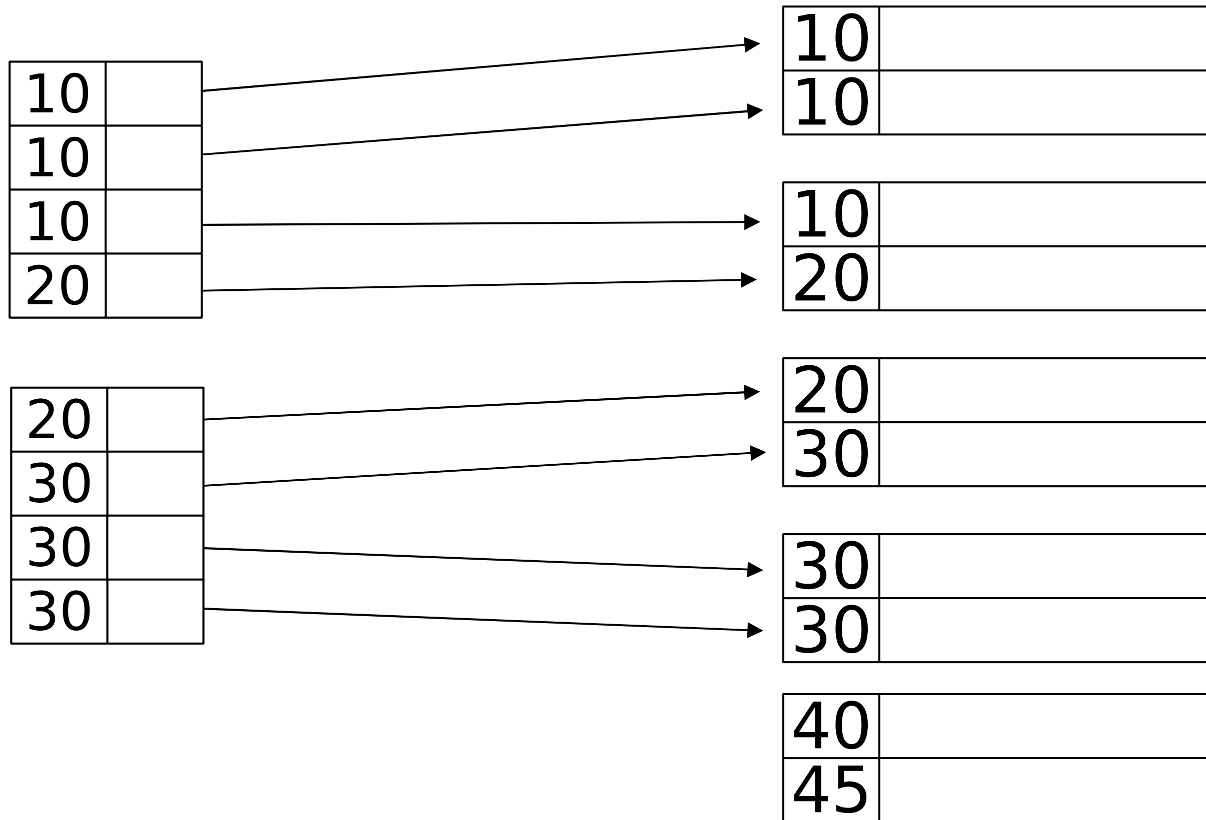
20	
30	

30	
30	

40	
45	

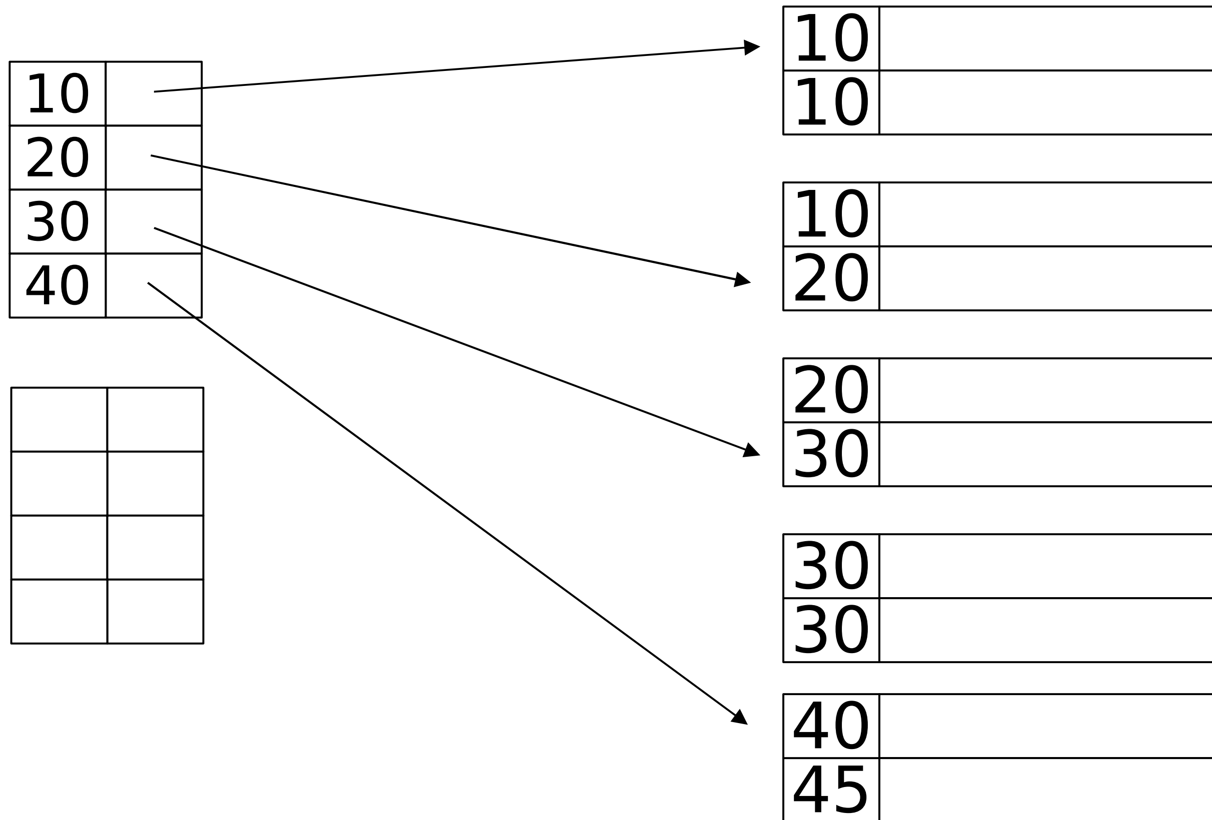
Duplicate keys

Dense index, one way to implement?



Duplicate keys

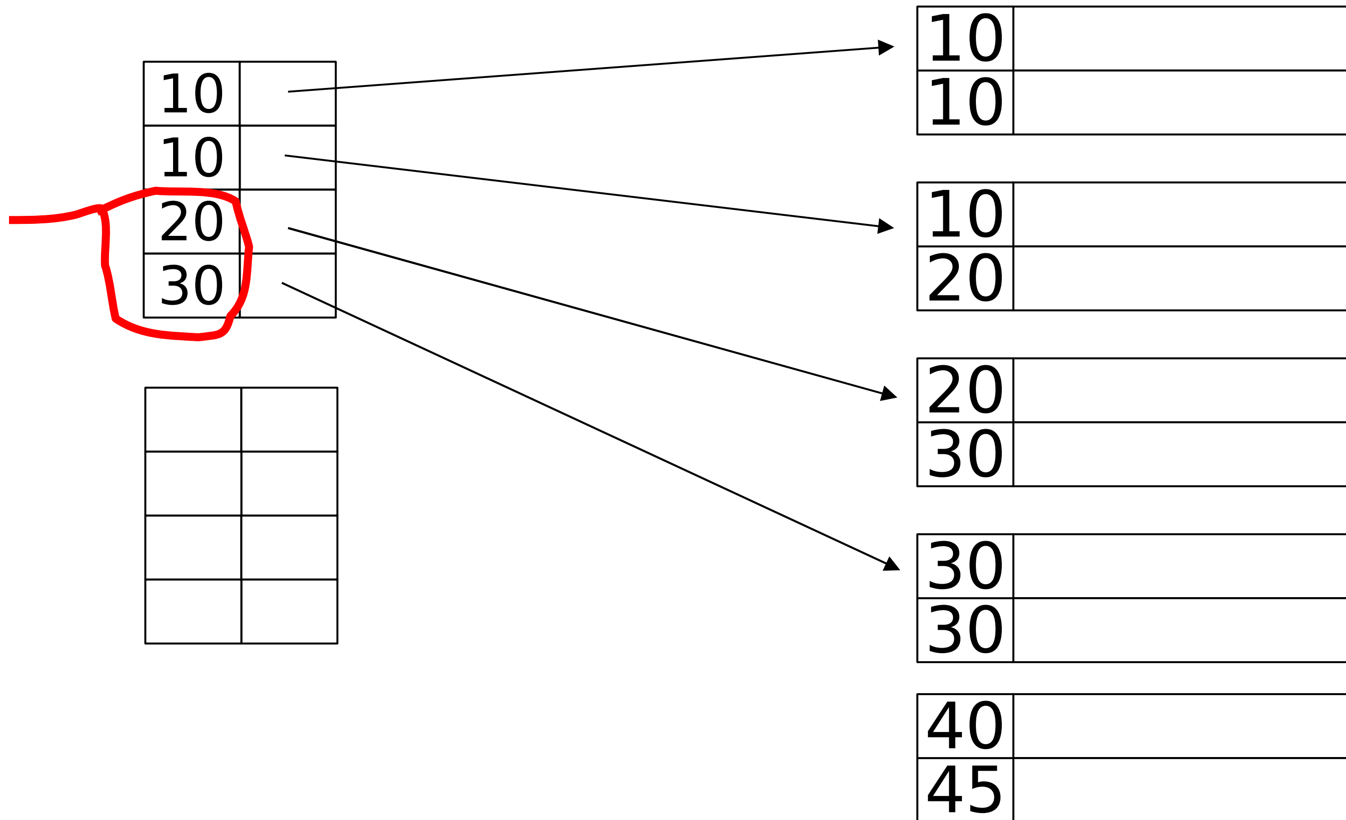
Dense index, better way?



Duplicate keys

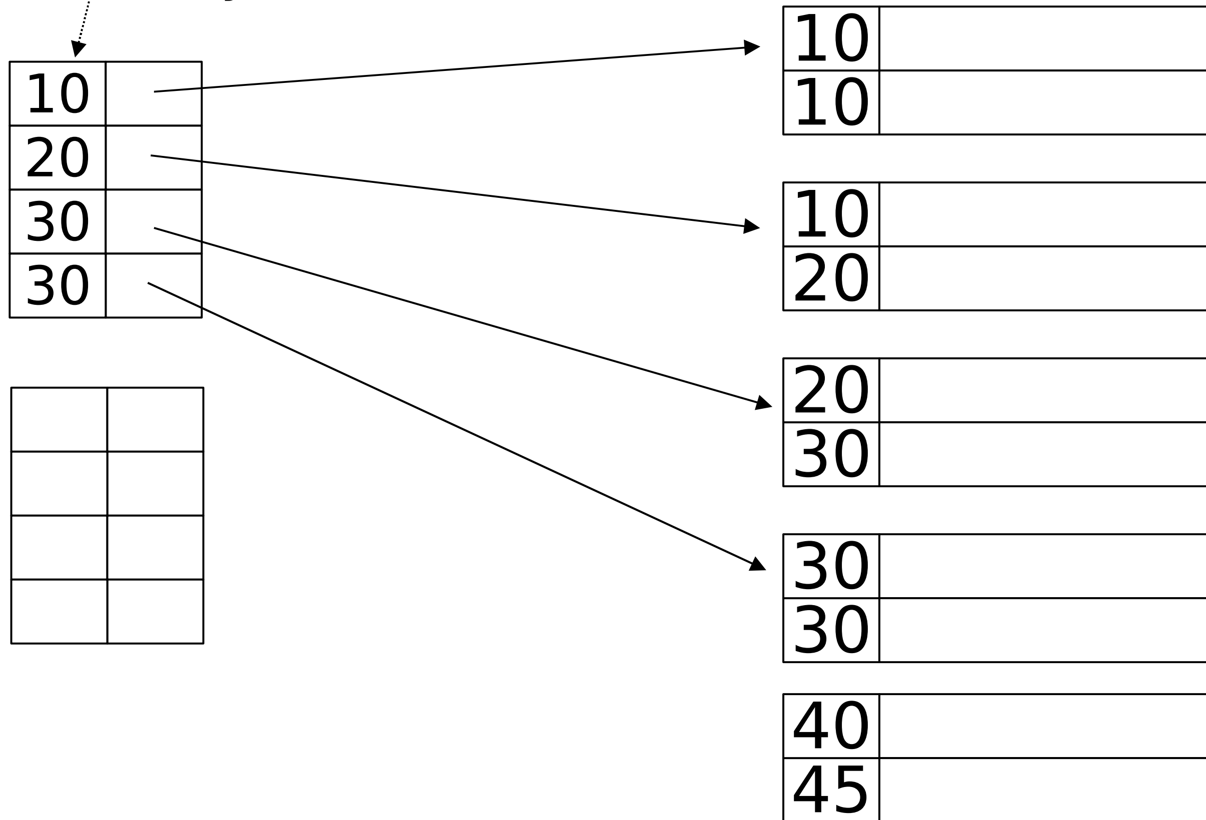
Sparse index, one way?

careful if looking for 20 or 30!

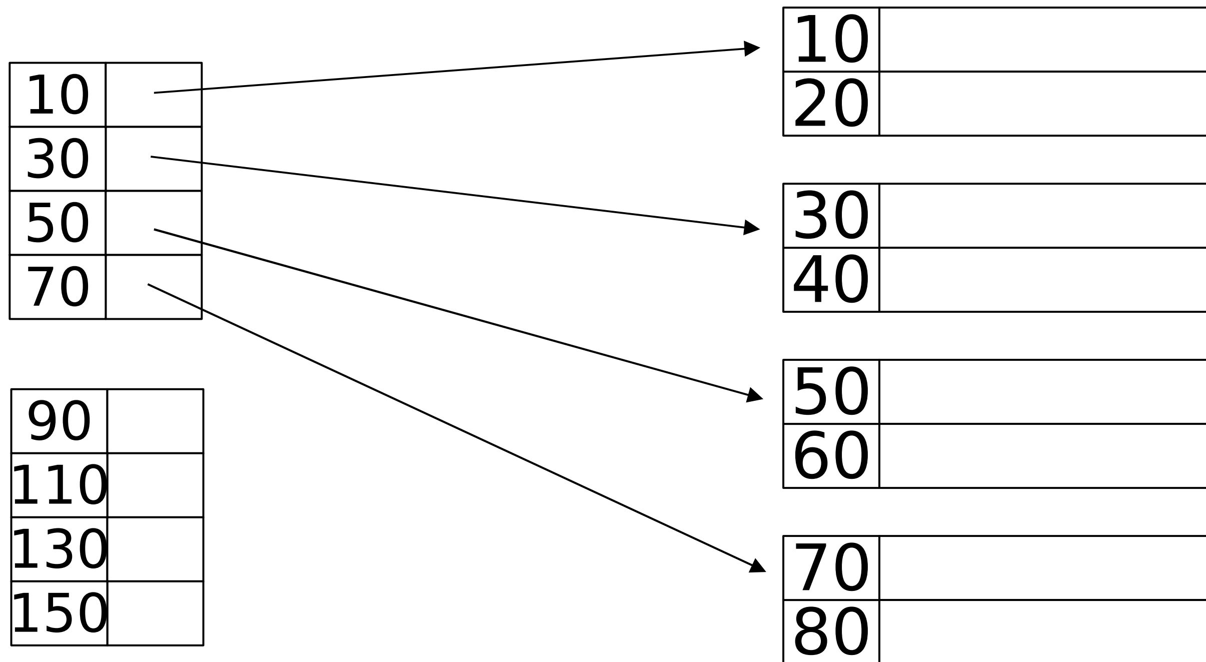


Duplicate keys

Sparse index, another way?

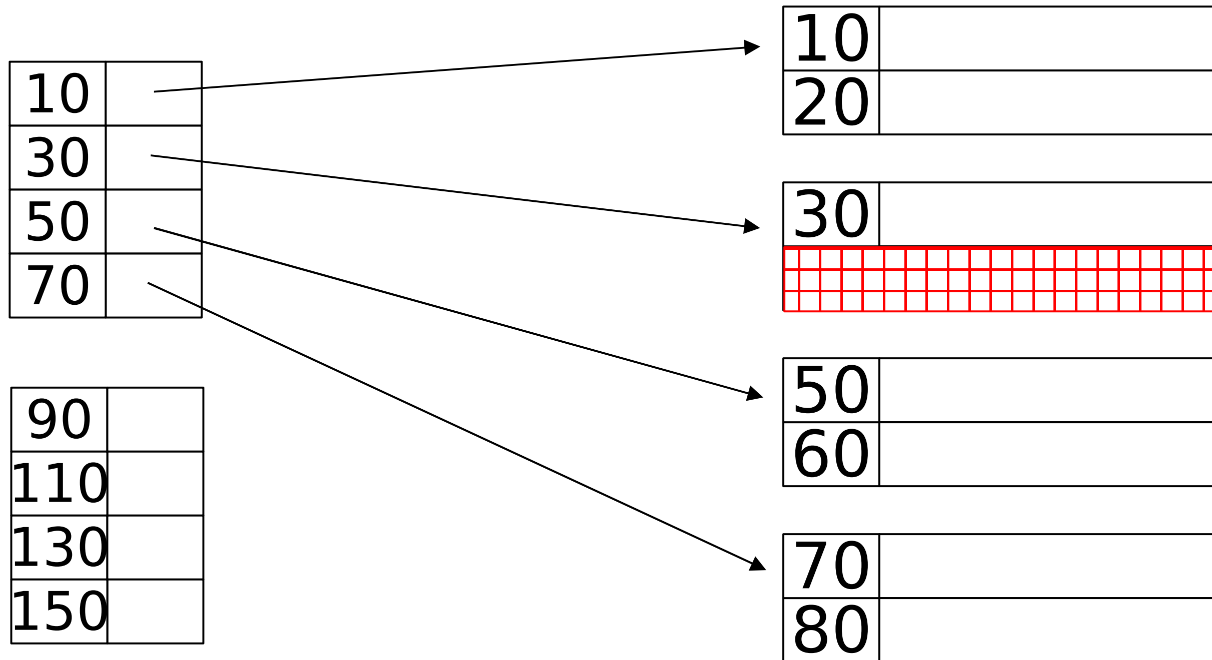


Deletion from sparse index



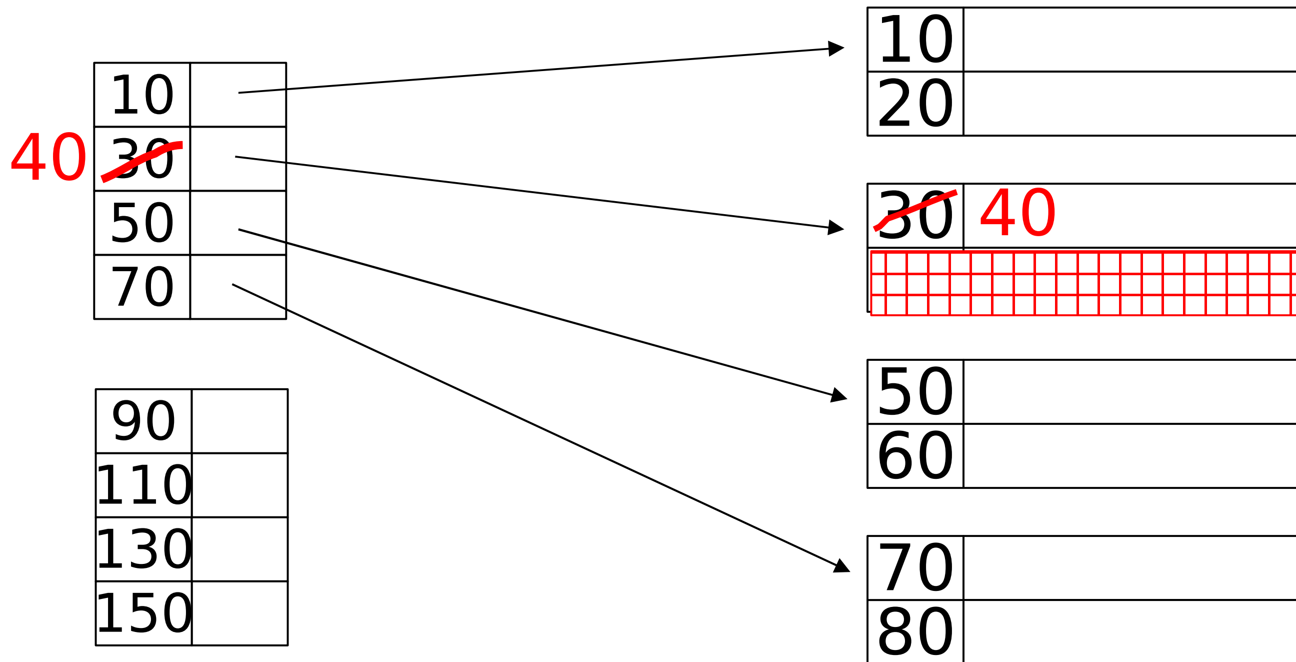
Deletion from sparse index

- delete record 40



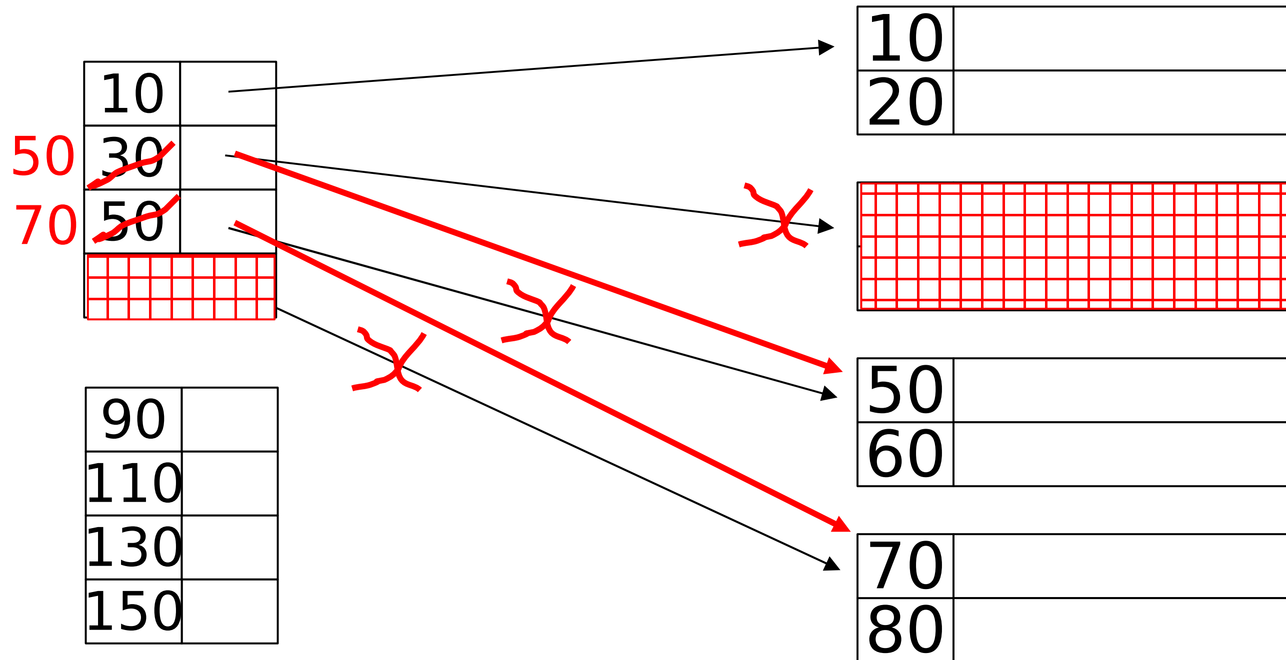
Deletion from sparse index

- delete record 30

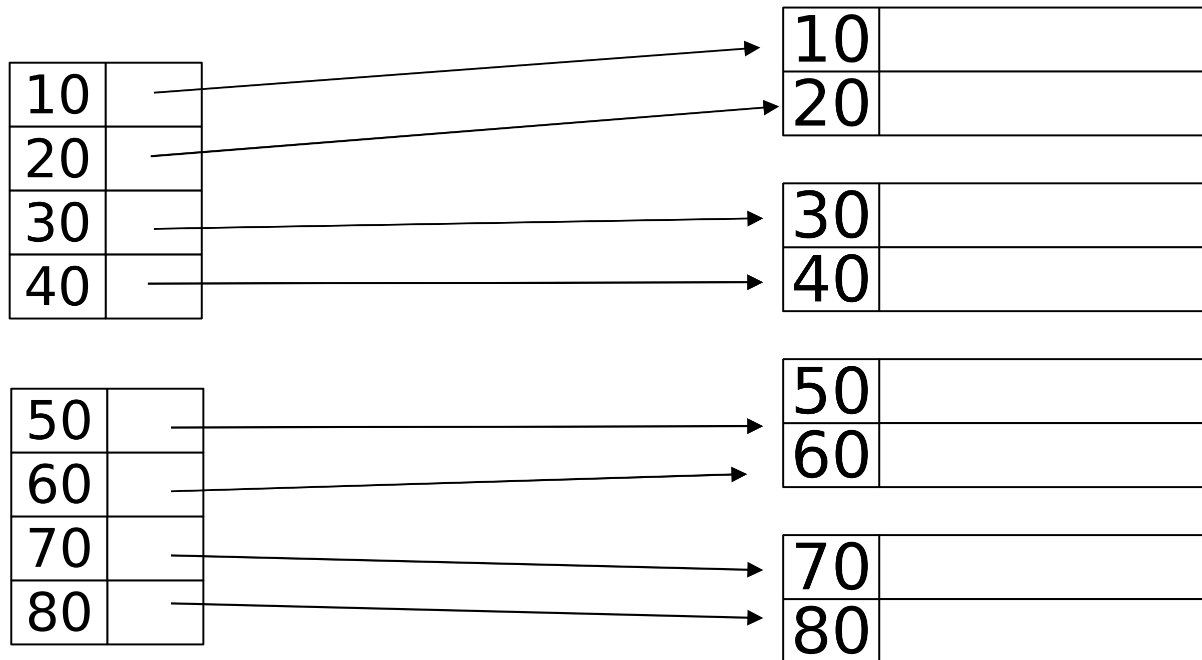


Deletion from sparse index

- delete records 30 & 40

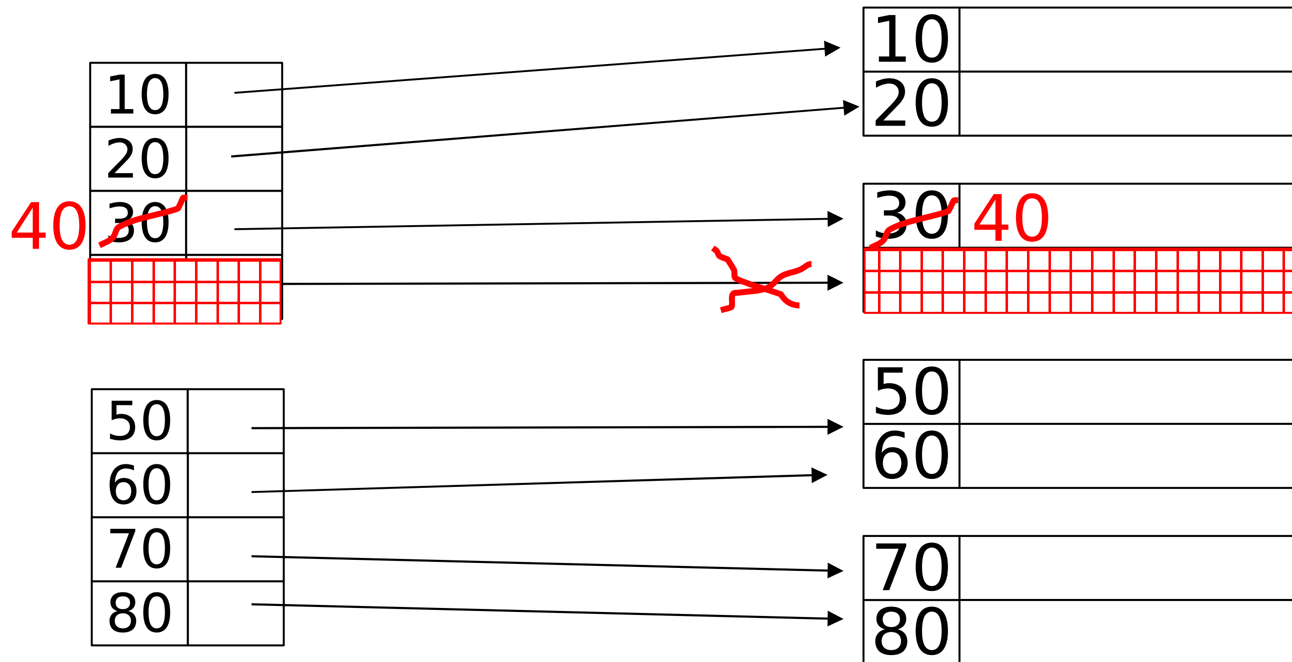


Deletion from dense index

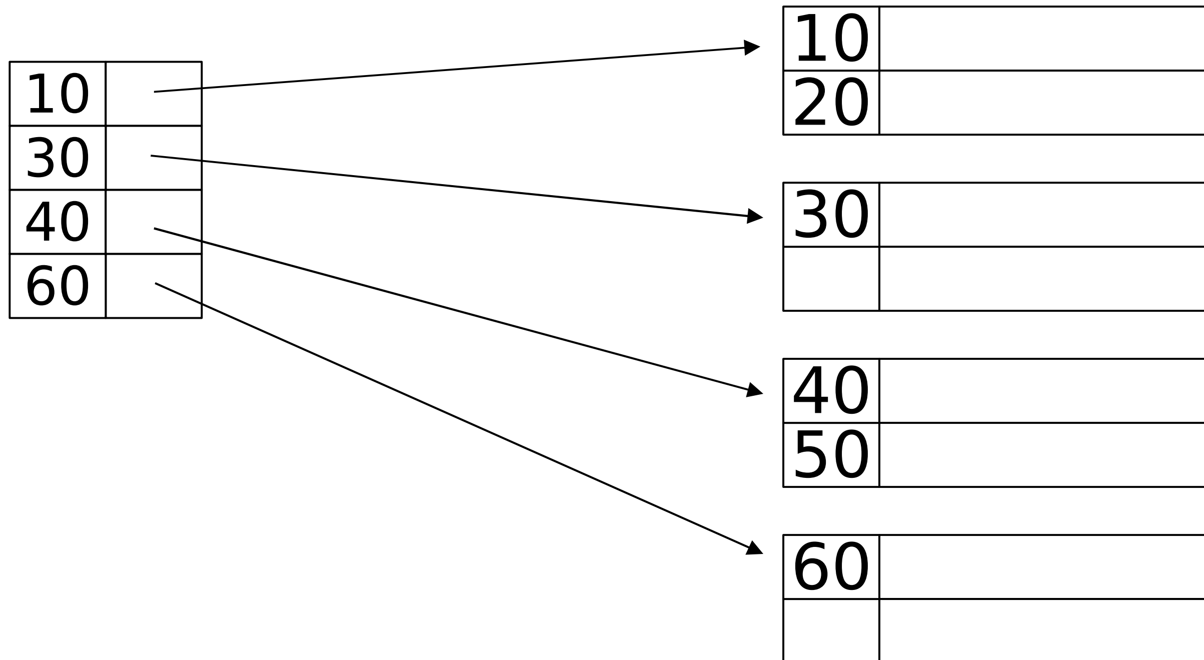


Deletion from dense index

- delete record 30

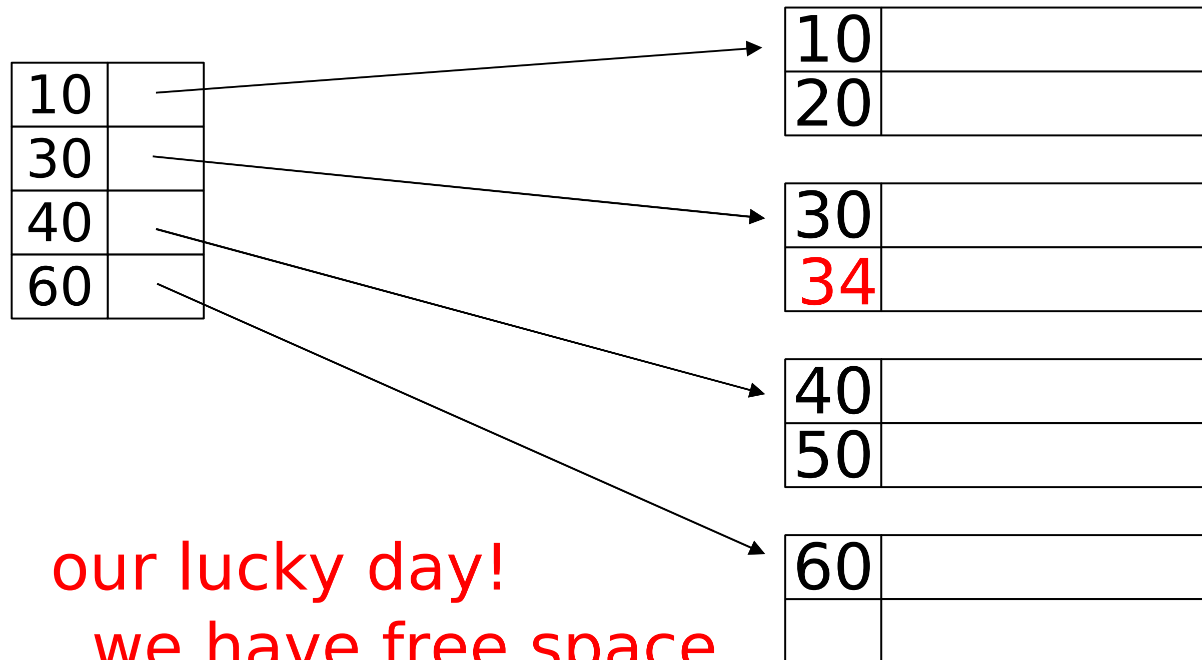


Insertion, sparse index case



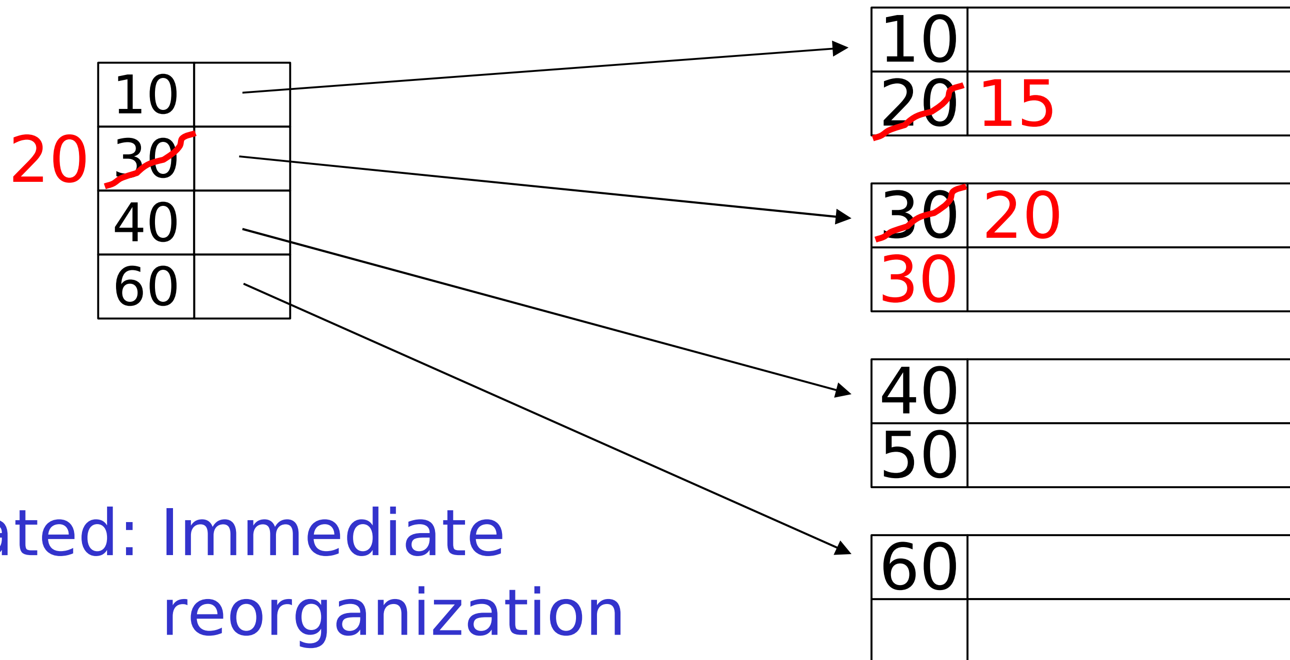
Insertion, sparse index case

- insert record 34



Insertion, sparse index case

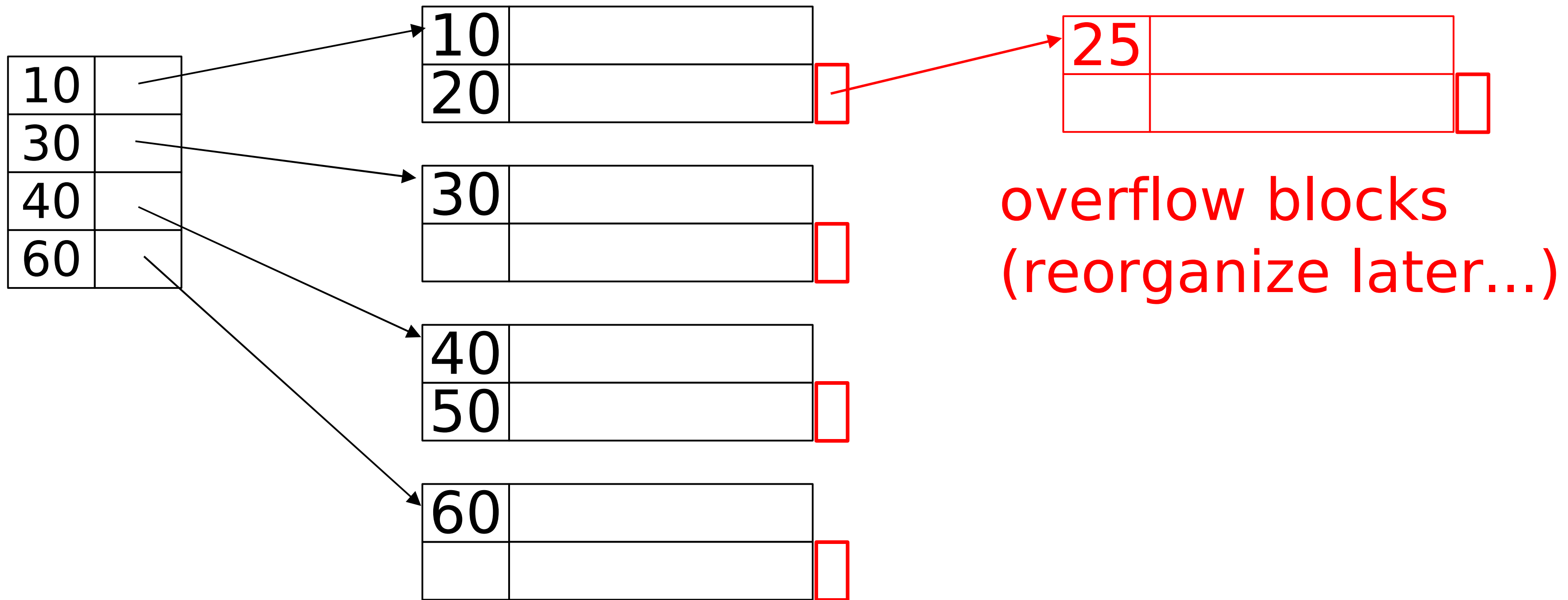
- insert record 15



- Illustrated: Immediate reorganization
- Variation:
  - insert new block (chained file)
  - update index

Insertion, sparse index case

- insert record 25



Insertion, dense index case

- Similar
- Often more expensive . . .

Secondary indexes

Sequence field

30	
50	

20	
70	

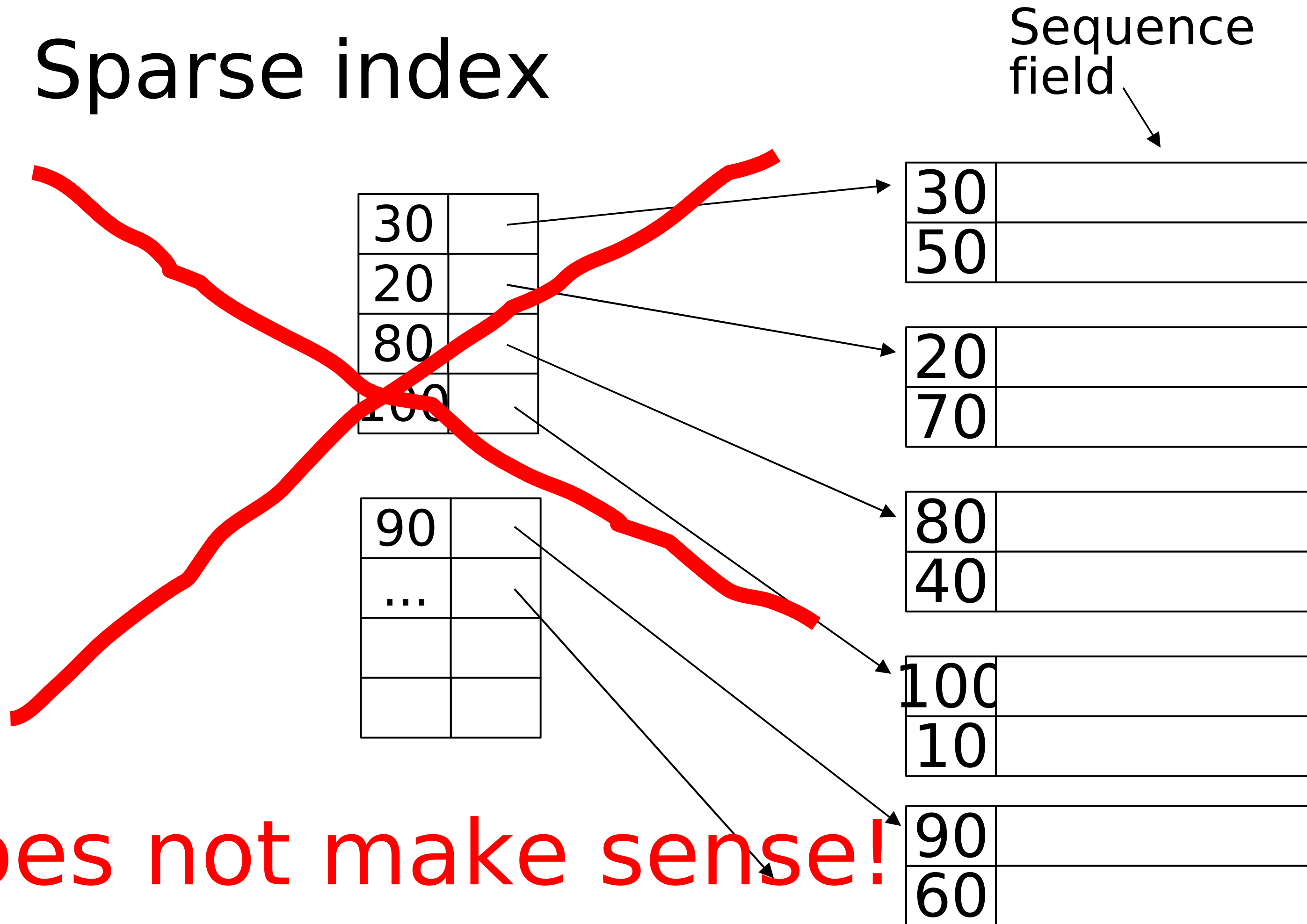
80	
40	

100	
10	

90	
60	

Secondary indexes

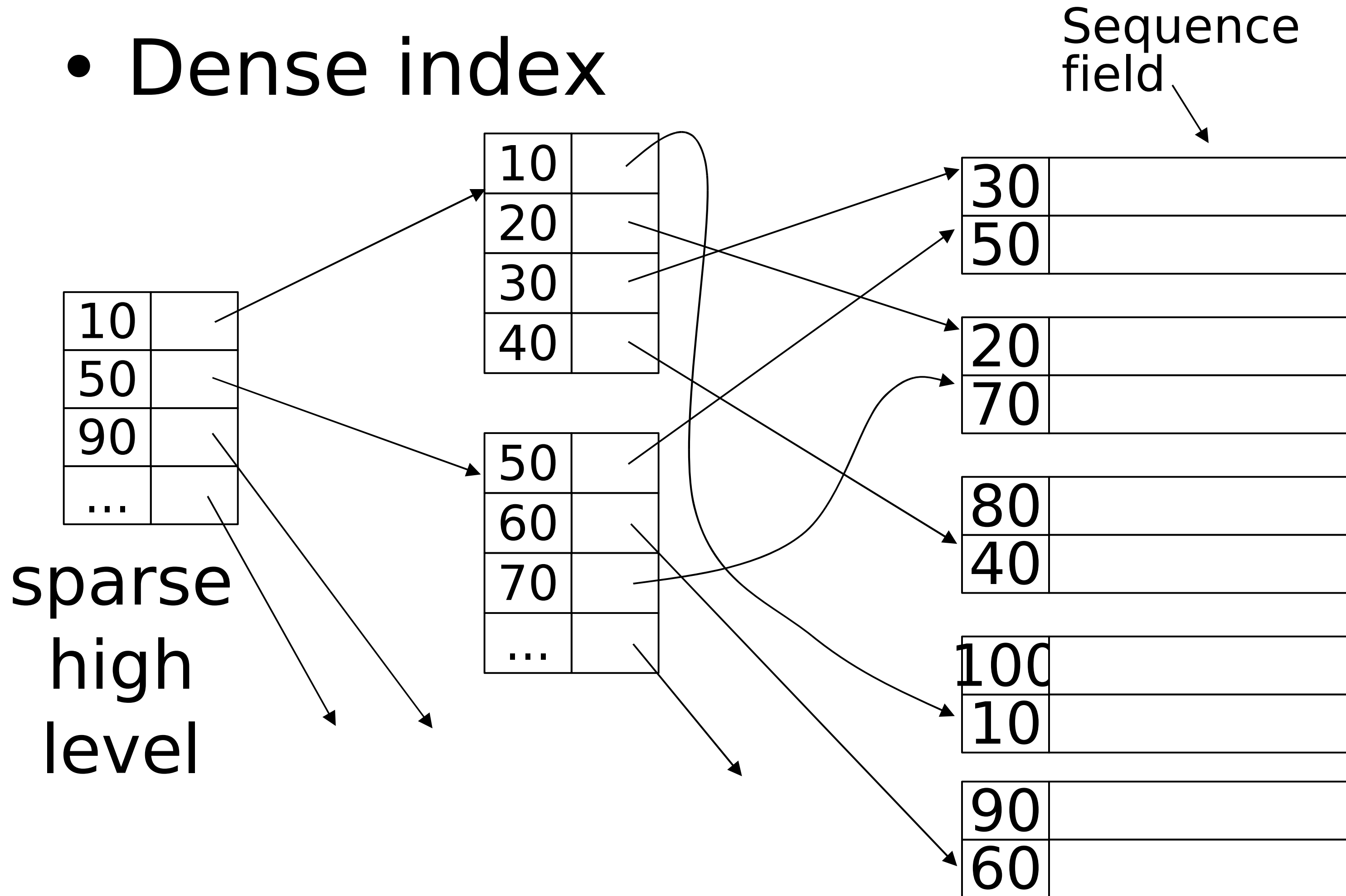
- Sparse index



does not make sense!

Secondary indexes

- Dense index



With secondary indexes:

- Lowest level is dense
- Other levels are sparse

Also: Pointers are record pointers

(not block pointers; not computed)

Duplicate values & secondary indexes

20	
10	

20	
40	

10	
40	

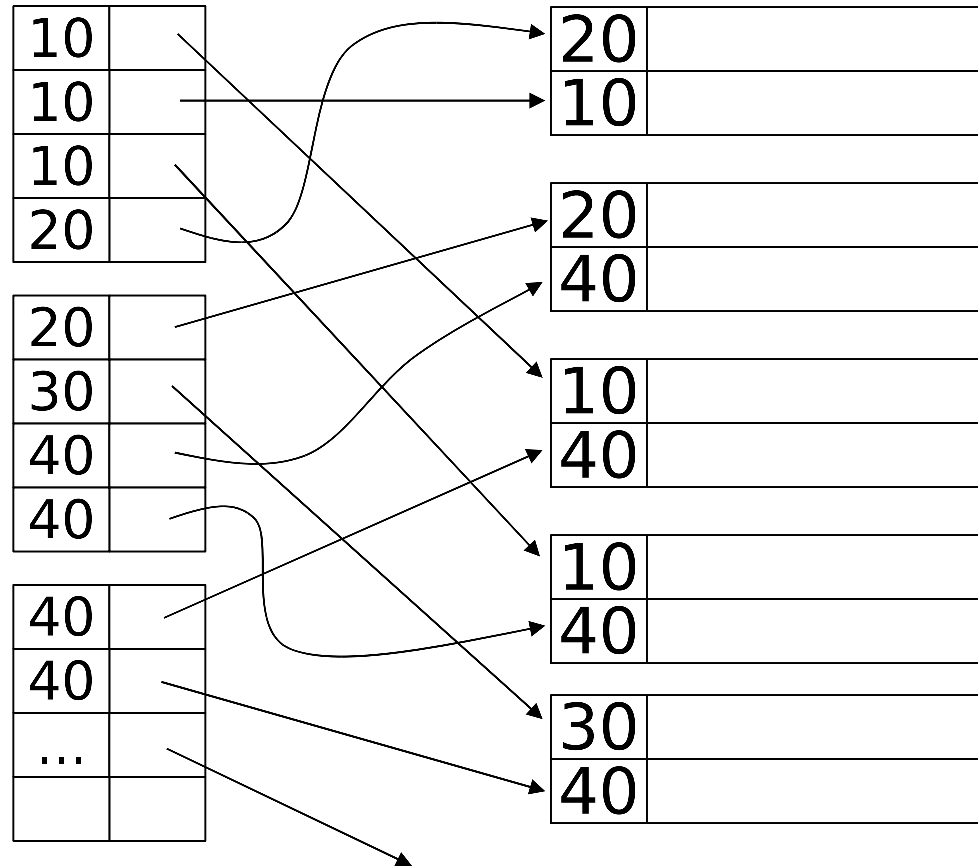
10	
40	

30	
40	

Duplicate values & secondary indexes

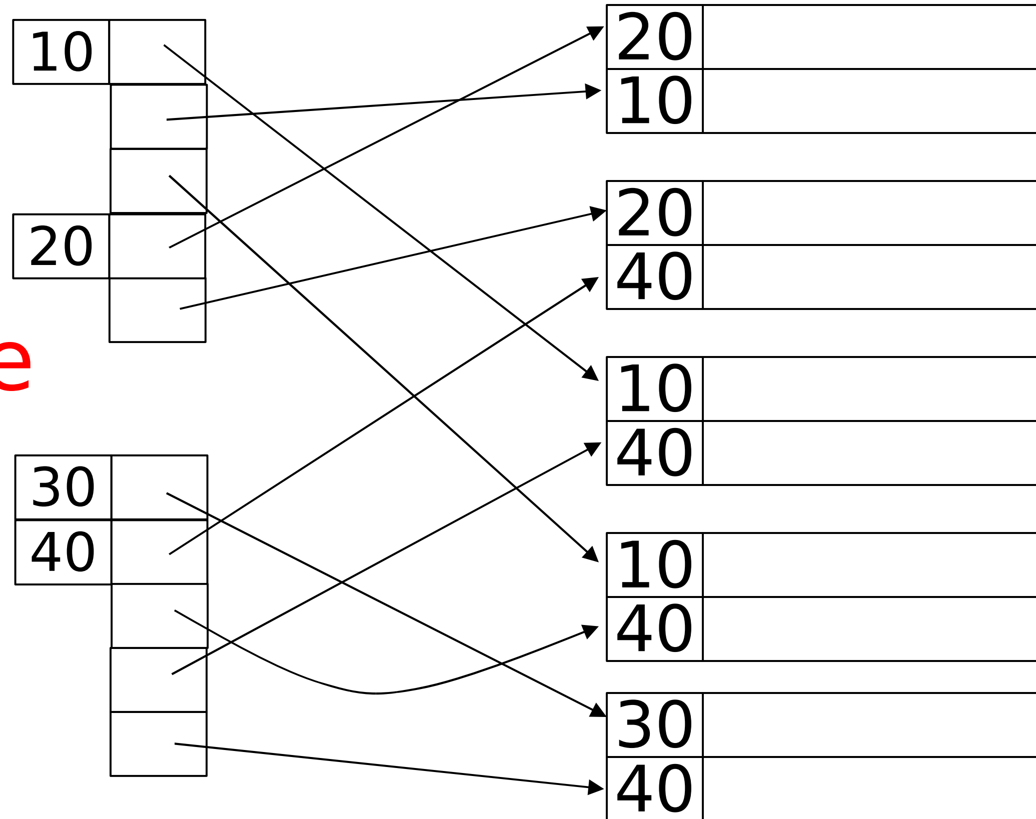
one option...

Problem:
excess overhead!
- disk space
- search time



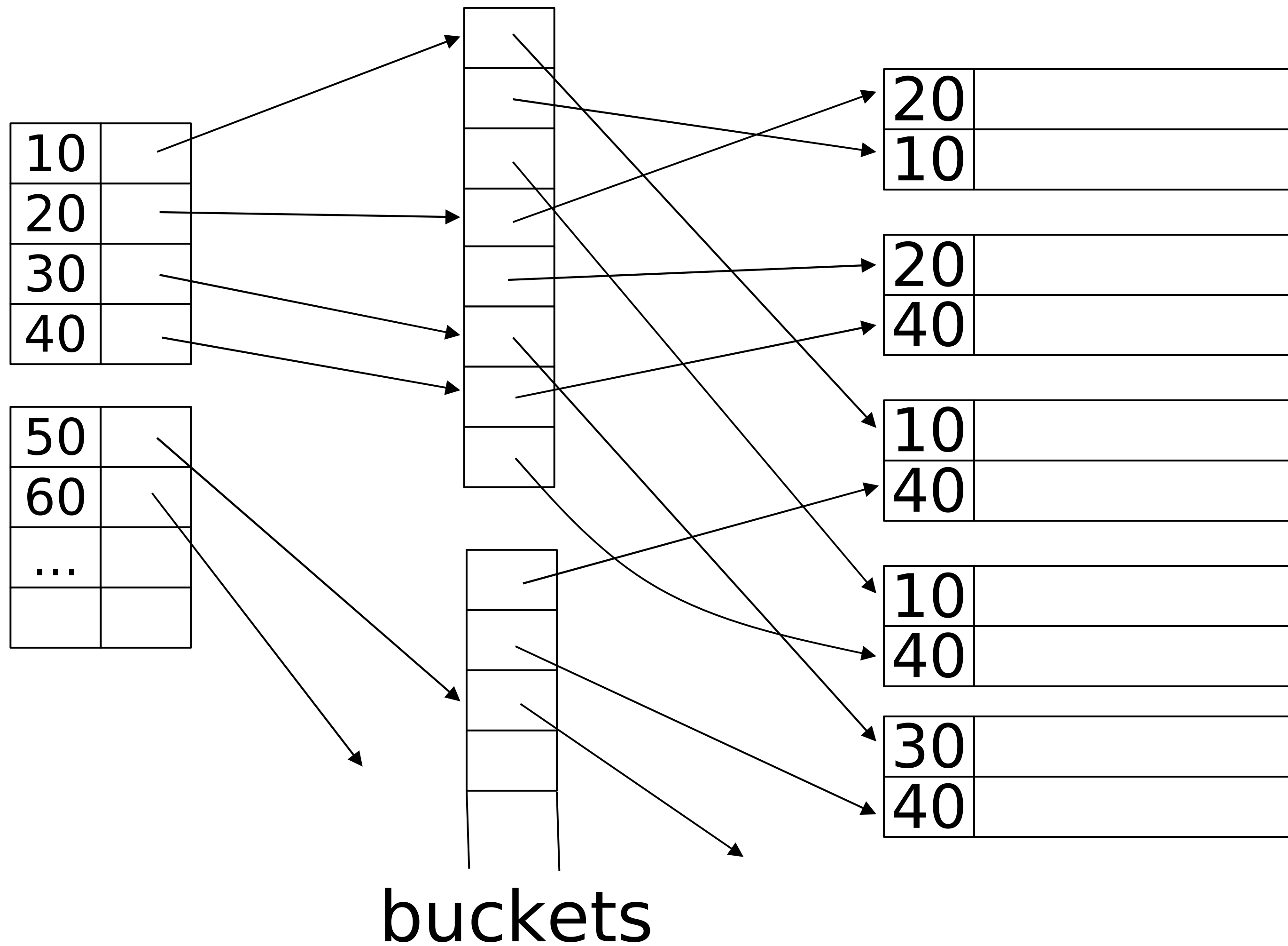
Duplicate values & secondary indexes

another option...



Problem: variable size records in index!

Duplicate values & secondary indexes

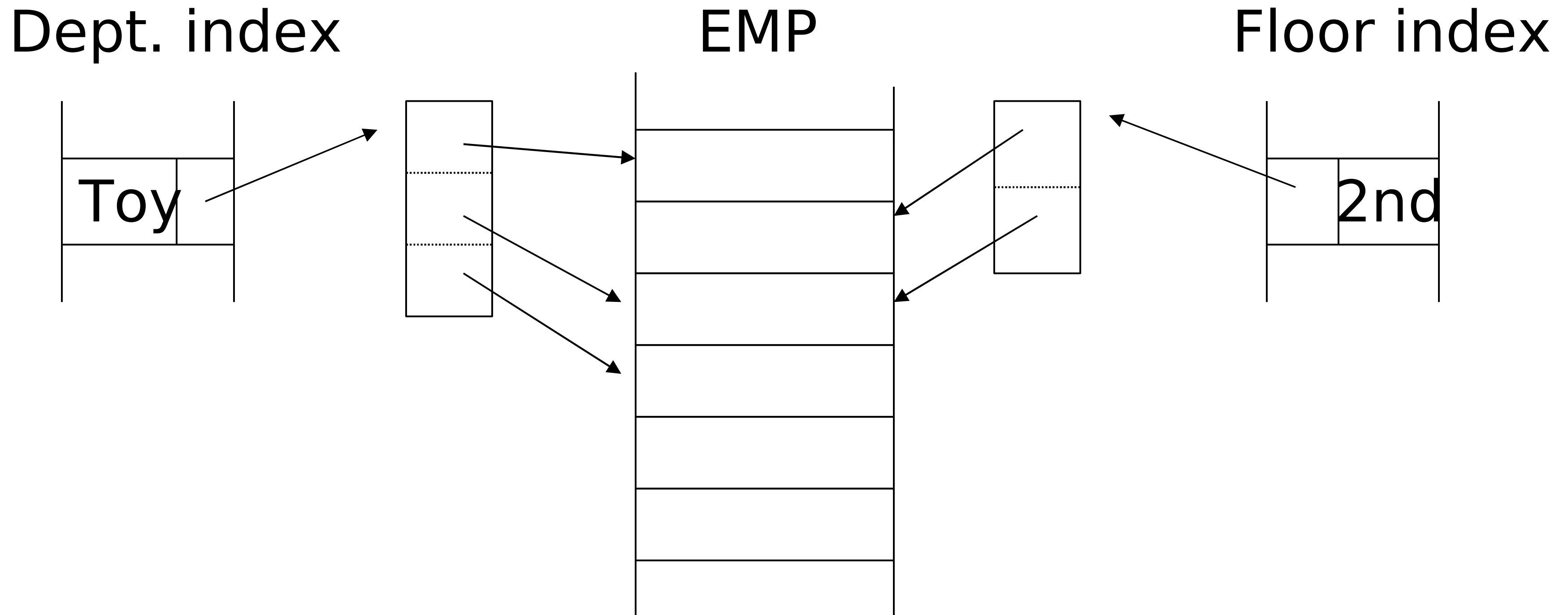


Why "bucket" idea is useful

Indexes	Records
Name: primary	EMP
(name,dept,floor,...)	
Dept: secondary	
Floor: secondary	

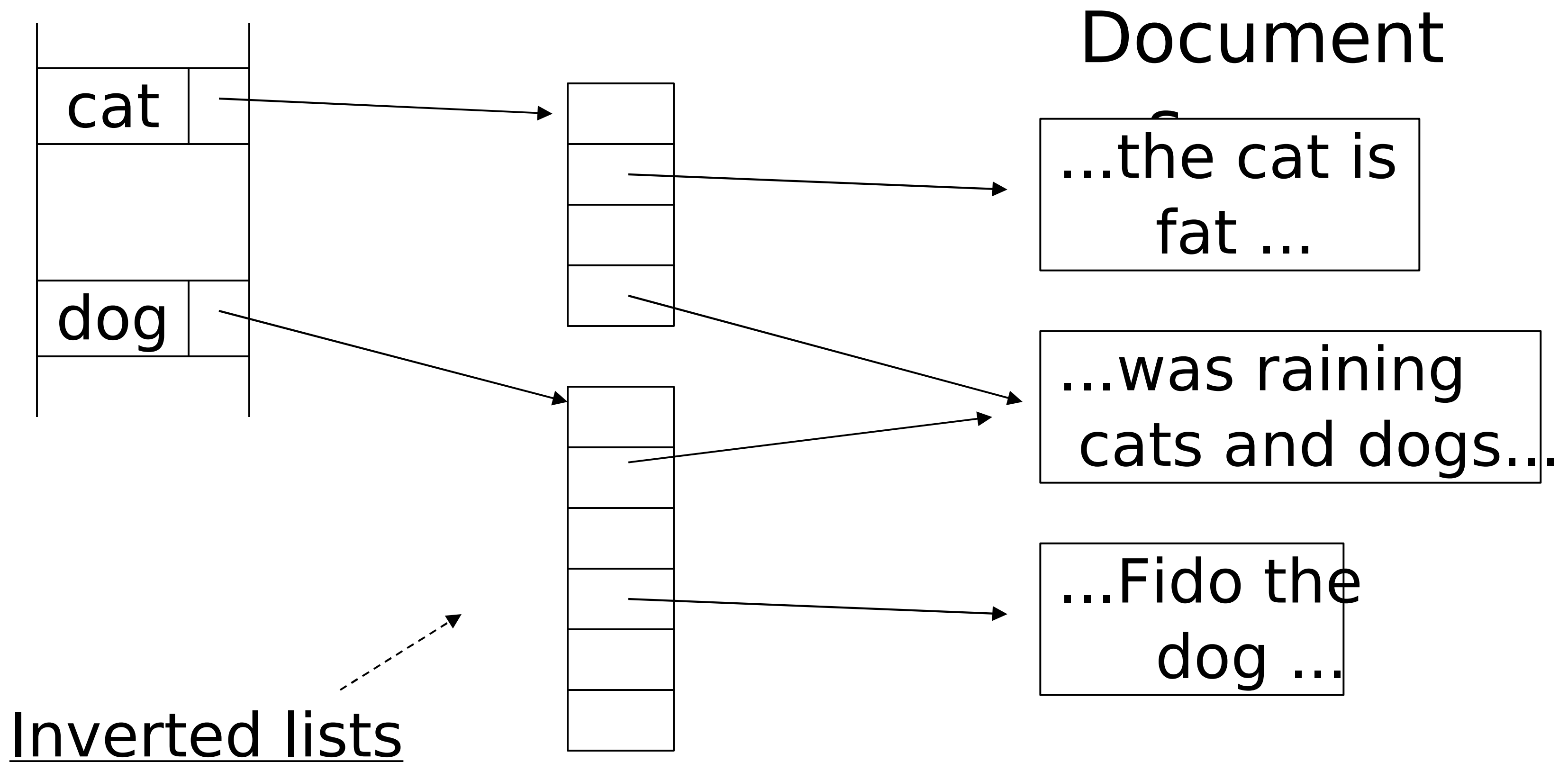
Query: Get employees in

(Toy Dept) ^ (2nd floor)



→ Intersect toy bucket and 2nd Floor bucket to get set of matching EMP’s

This idea used in text information retrieval



IR QUERIES

- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “dog”
- Find articles with “cat” in title
- Find articles with “cat” and “dog” within 5 words

Summary so far

- Conventional index
  - Basic Ideas: sparse, dense, multi-level…
  - Duplicate Keys
  - Deletion/Insertion
  - Secondary indexes
    - Buckets of Postings List

Outline/summary

- Conventional Indexes
    - Sparse vs. dense
    - Primary vs. secondary
- B trees --> Next
    - B+trees vs. indexed sequential
- Hashing schemes

Conventional indexes

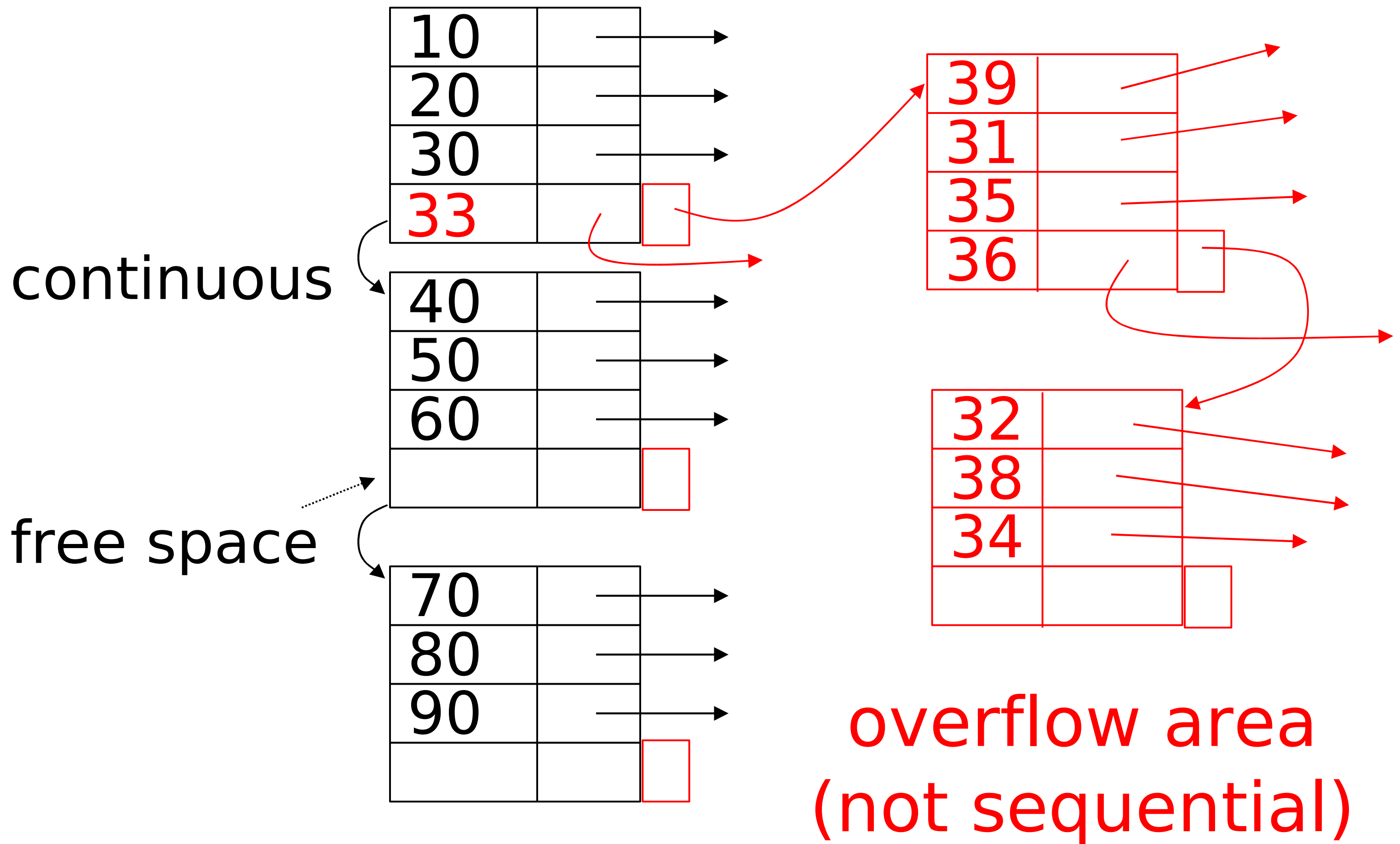
Advantage:

- Simple
- Index is sequential file
  good for scans

Disadvantage:

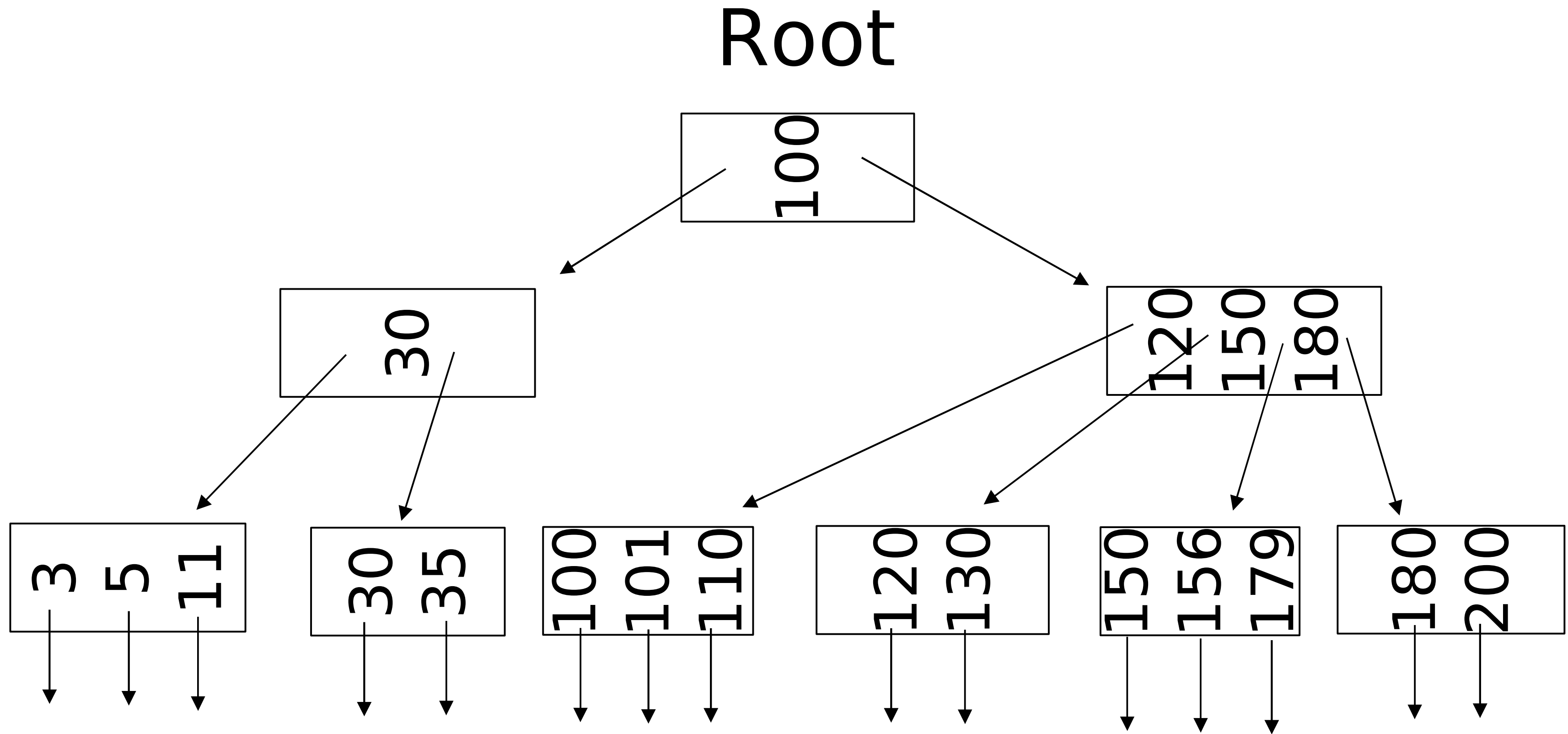
- Inserts expensive, and/or
- Lose sequentiality & balance

Example

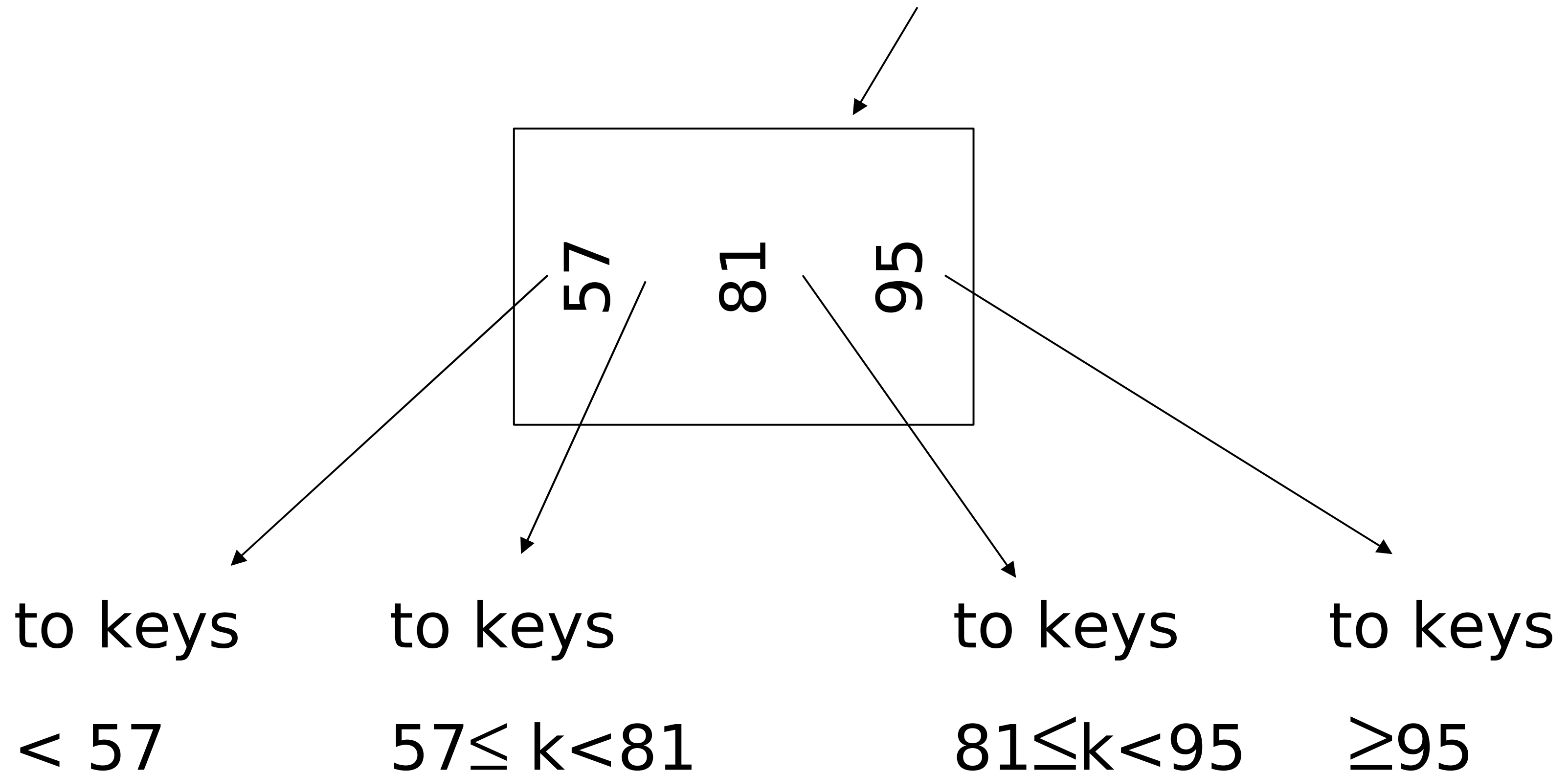


- NEXT: Another type of index
  - Give up on sequentiality of index
  - Try to get “balance”

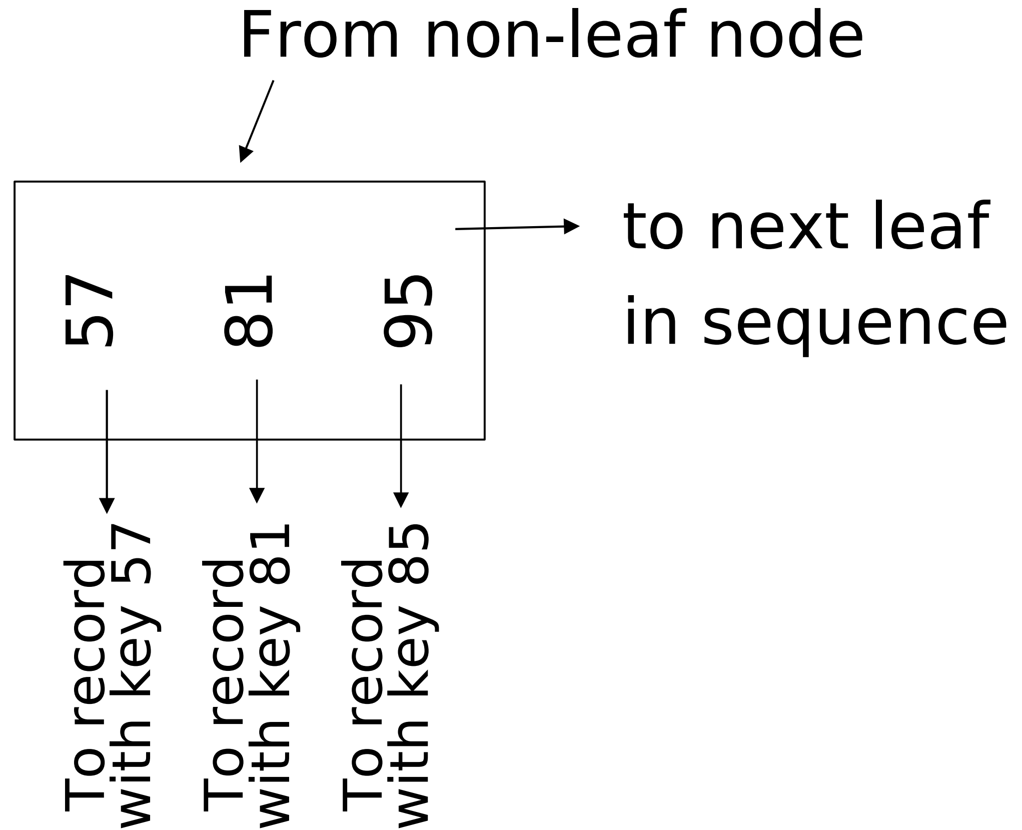
B+Tree Example n=3



Sample non-leaf



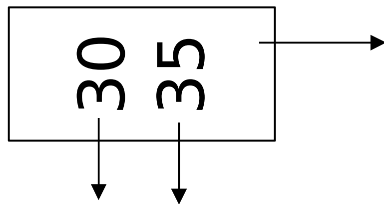
Sample leaf node:

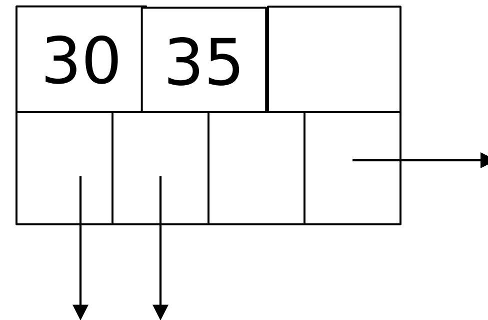


In textbook's notation

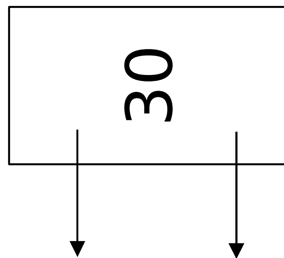
n=3

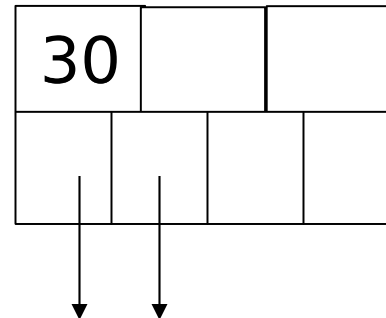
Leaf:





Non-leaf:





Size of nodes: { n+1 pointers, n keys } (fixed)

Don’t want nodes to be too empty

- Use at least

  Non-leaf: $\lceil (n+1)/2 \rceil$ pointers

  Leaf : $\lfloor (n+1)/2 \rfloor$ pointers to data

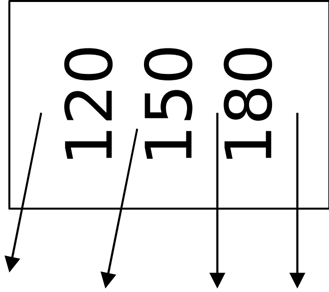
n=3

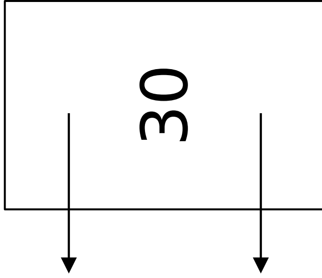
Full node

min.

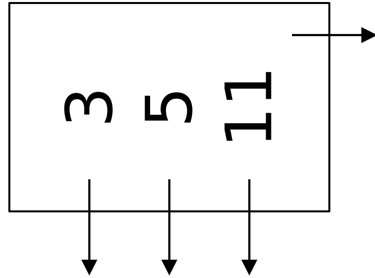
node

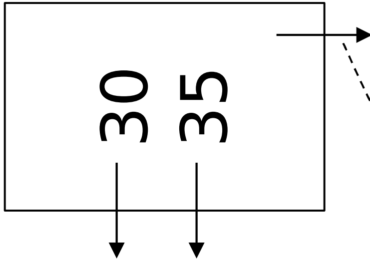
Non-leaf





Leaf





counts even if null

B+tree rules — tree of order *d*

(1) All leaves at same lowest level (balanced tree)

(2) Pointers in leaves point to records except for “sequence pointer”

(3) Number of pointers/keys for B+tree

	Max ptrs	Max keys	Min ptrs→data	Min keys
Non-leaf (non-root)	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf (non-root)	n+1	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	1	1

Insert into B+tree

(a) simple case

- space available in leaf

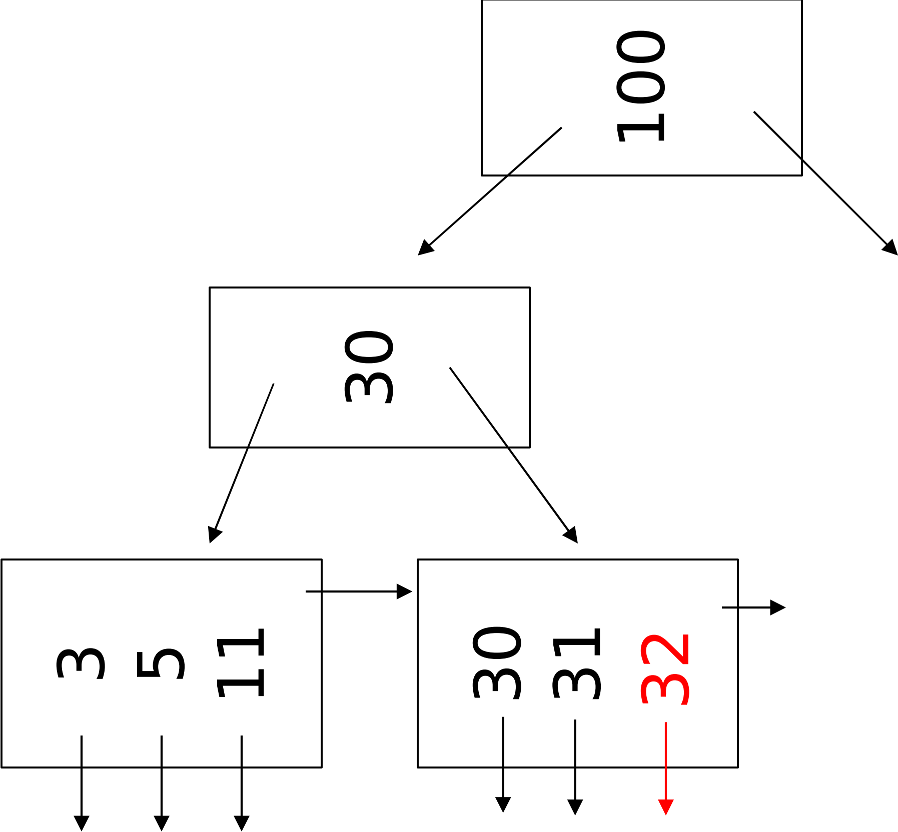
(b) leaf overflow

(c) non-leaf overflow

(d) new root

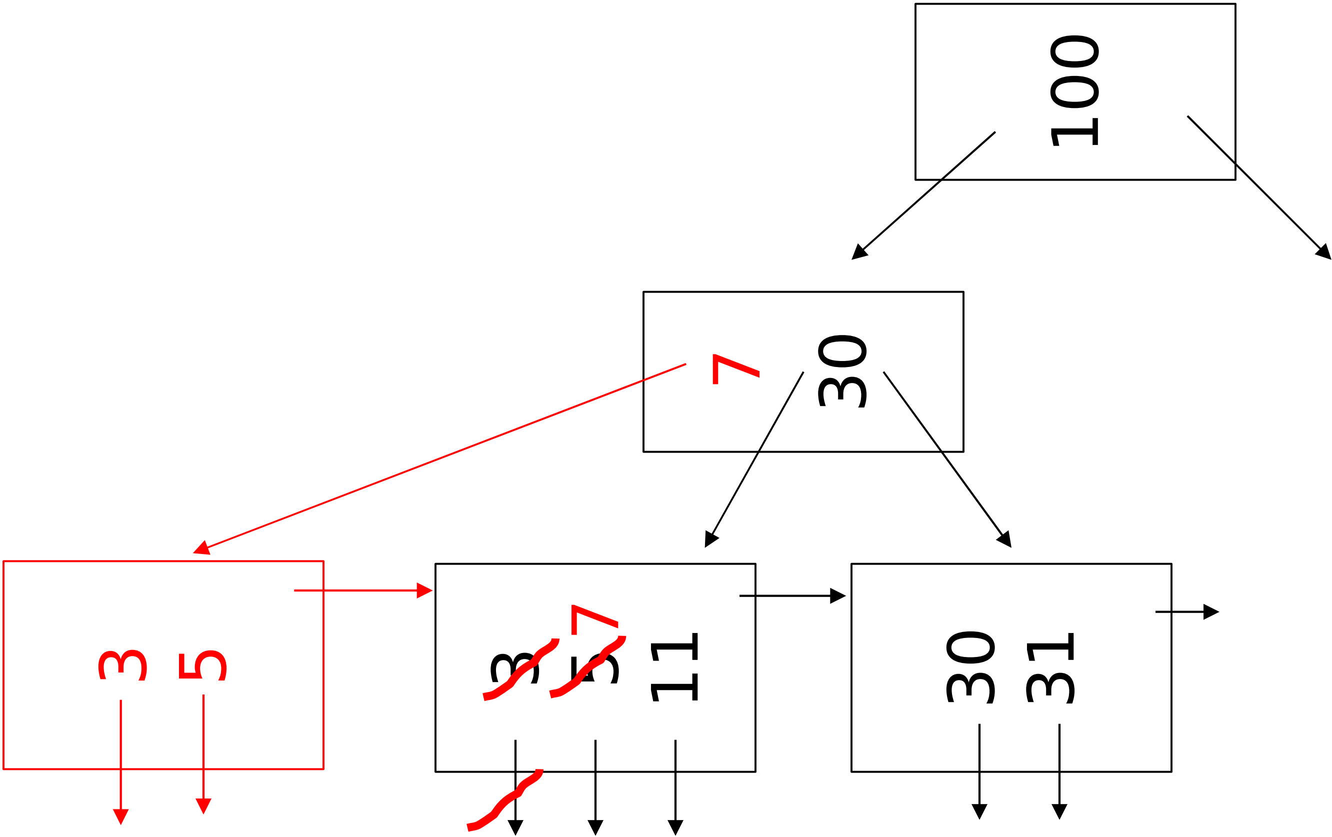
(a) Insert key = 32

n=3



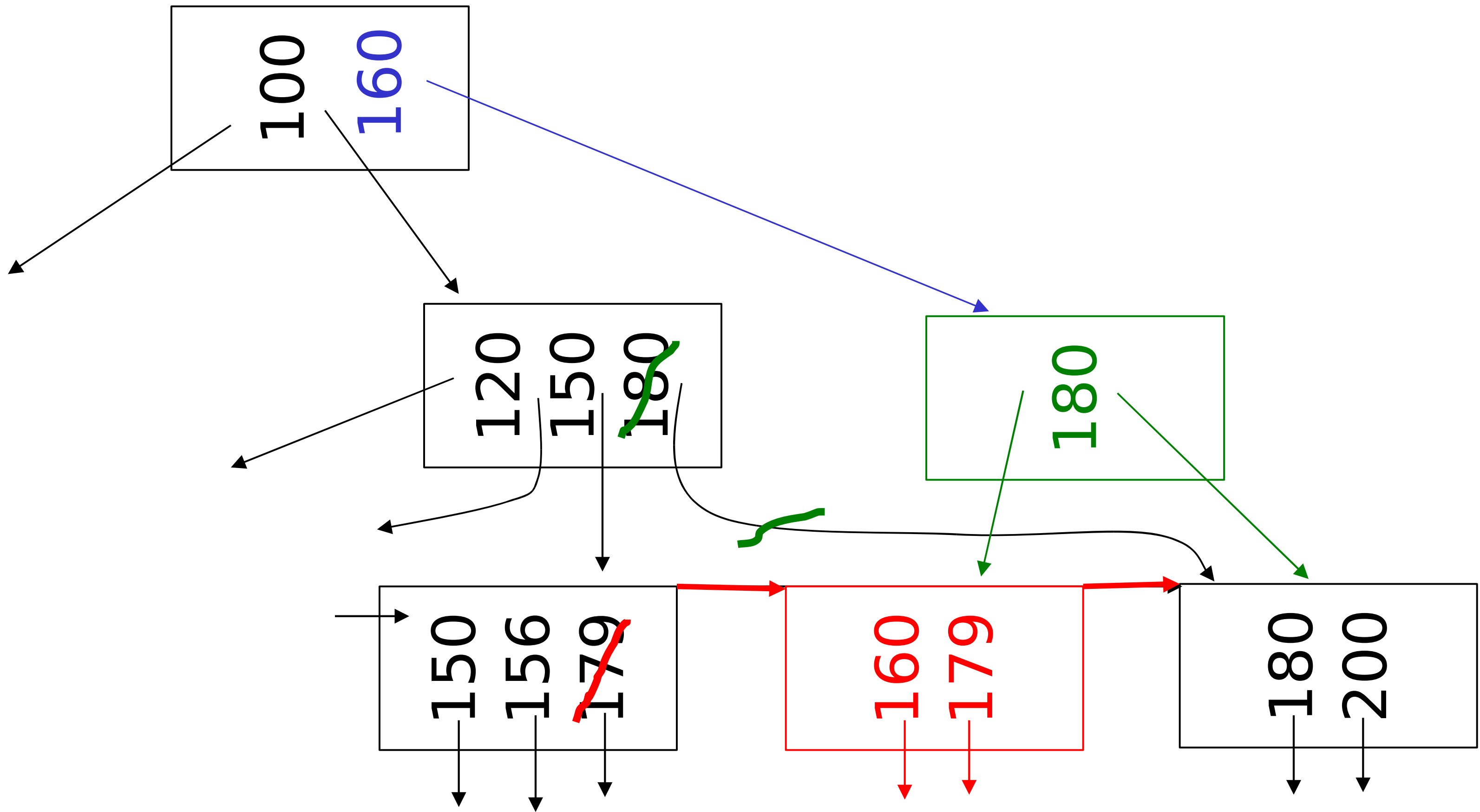
(a) Insert key = 7

n=3



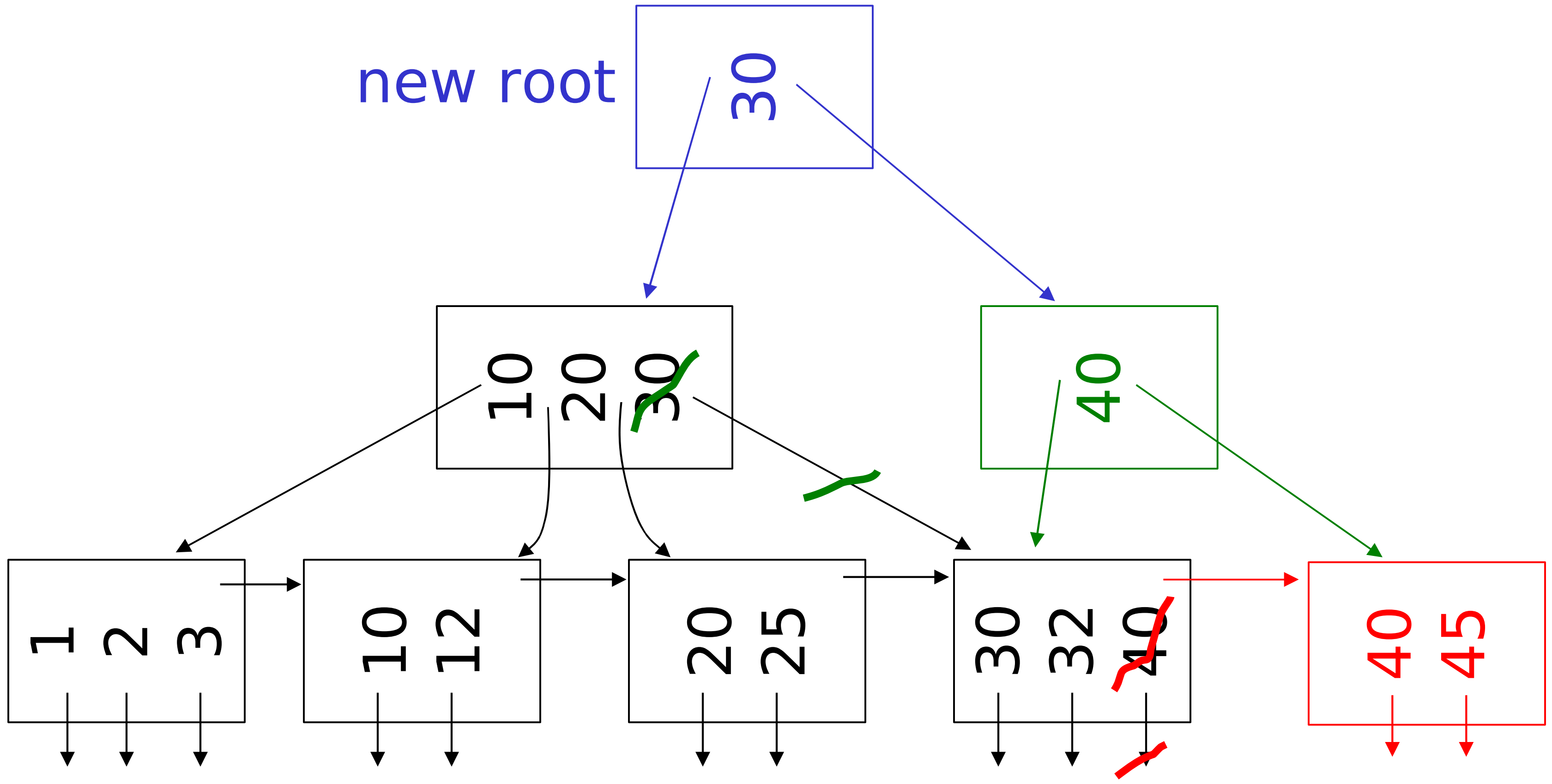
(c) Insert key = 160

n=3



(d) New root, insert 45

n=3



Deletion from B+tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

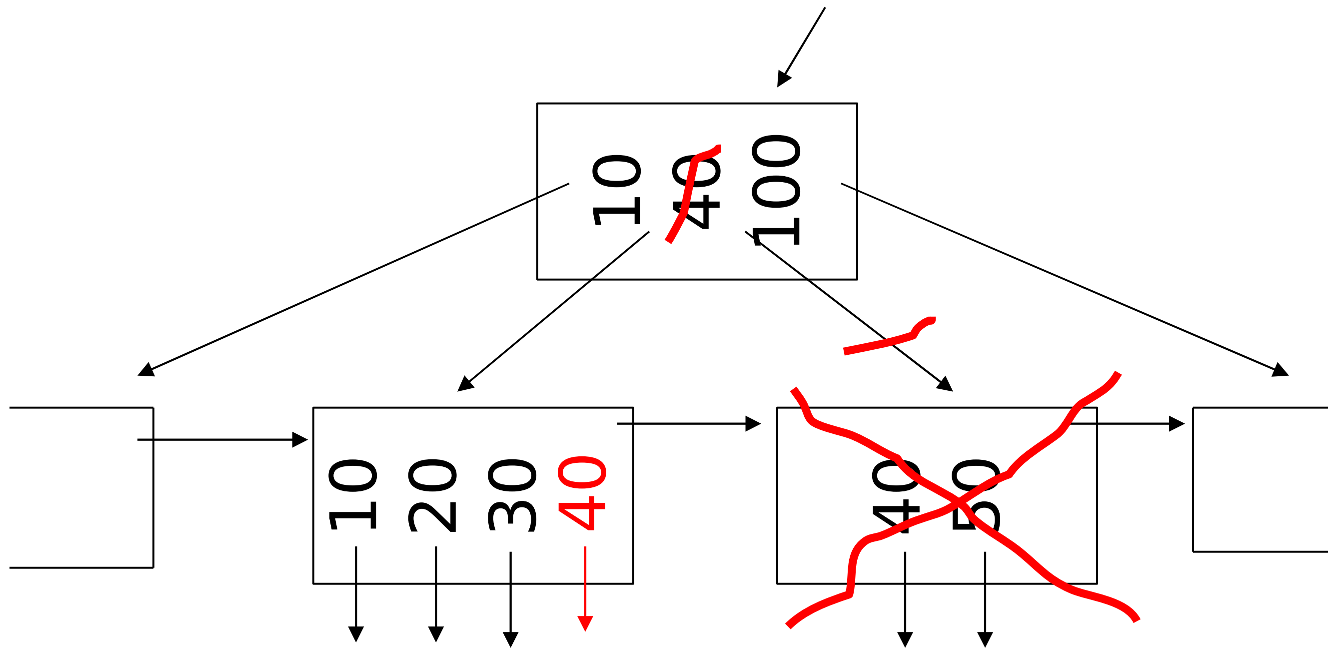
(c) Re-distribute keys

(d) Cases (b) or (c) at non-leaf

(b) Coalesce with sibling

– Delete 50

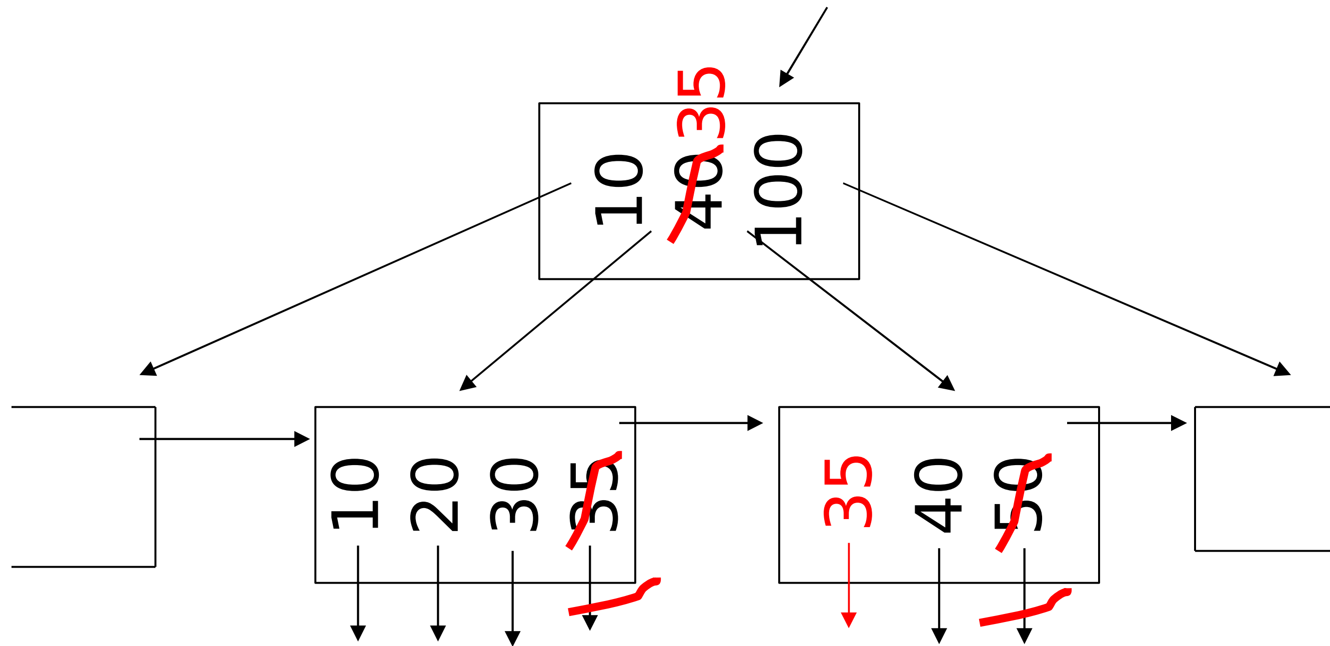
n=4



(c) Redistribute keys

– Delete 50

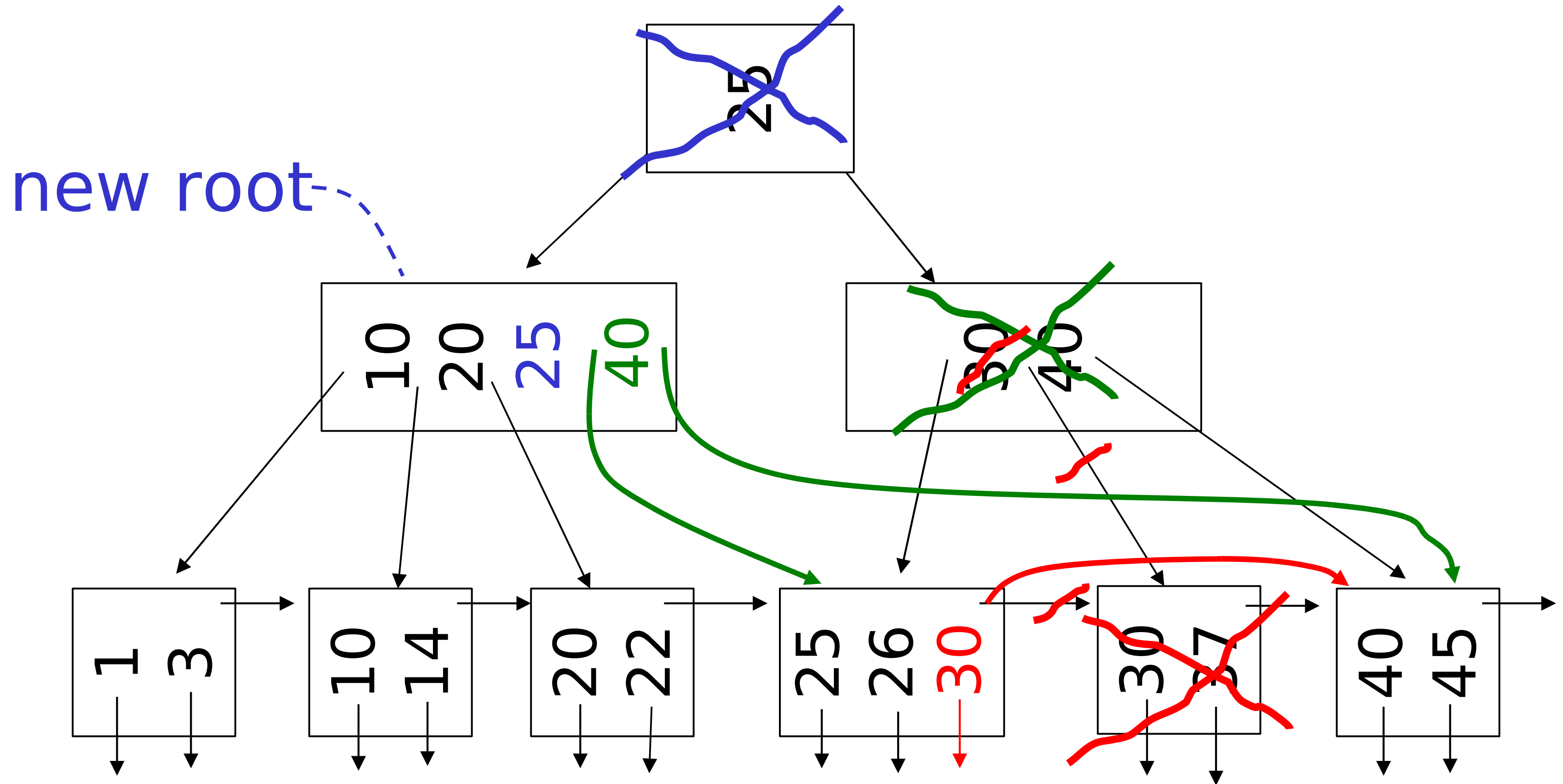
n=4



(d) Non-leaf coalese

– Delete 37

n=4

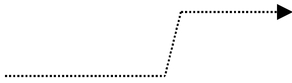


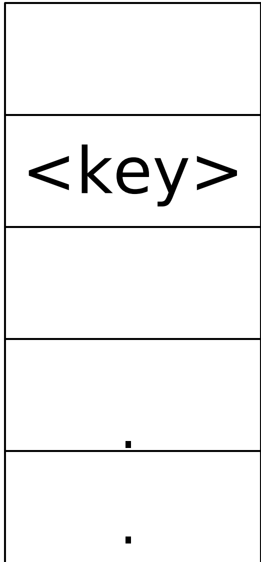
Outline/summary

- Conventional Indexes
  - Sparse vs. dense
  - Primary vs. secondary
- B trees
  - B+trees vs. indexed sequential
- Hashing schemes --> Next

Hashing

key $\rightarrow$ h(key)

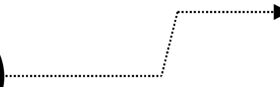




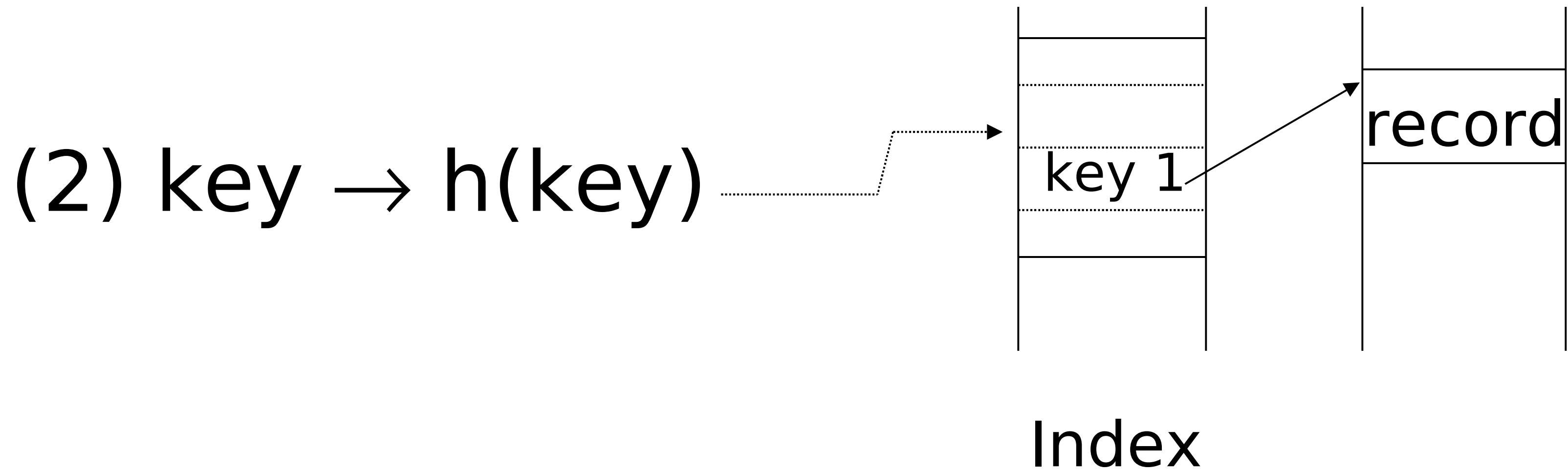
Buckets (typically 1 disk block)

Two alternatives

(1) key $\rightarrow$ h(key)



Two alternatives



- Alt (2) for “secondary” search key

Example hash function

- Key = 'x_1 x_2 … x_n'   *n* byte character string
- Have *b* buckets
- h: add $x_1 + x_2 + \ldots\ldots x_n$
  - compute sum modulo *b*

- This may not be best function …
- Read Knuth Vol. 3 if you really need to select a good function.

Good hash function:

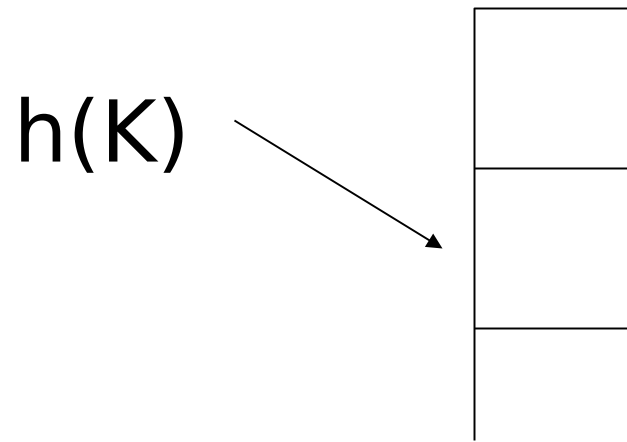
☞ Expected number of keys/bucket is the same for all buckets

Within a bucket:

- Do we keep keys sorted?

- Yes, if CPU time critical
  & Inserts/Deletes not too frequent

Next: example to illustrate inserts, overflows, deletes



EXAMPLE 2 records/bucket

INSERT:

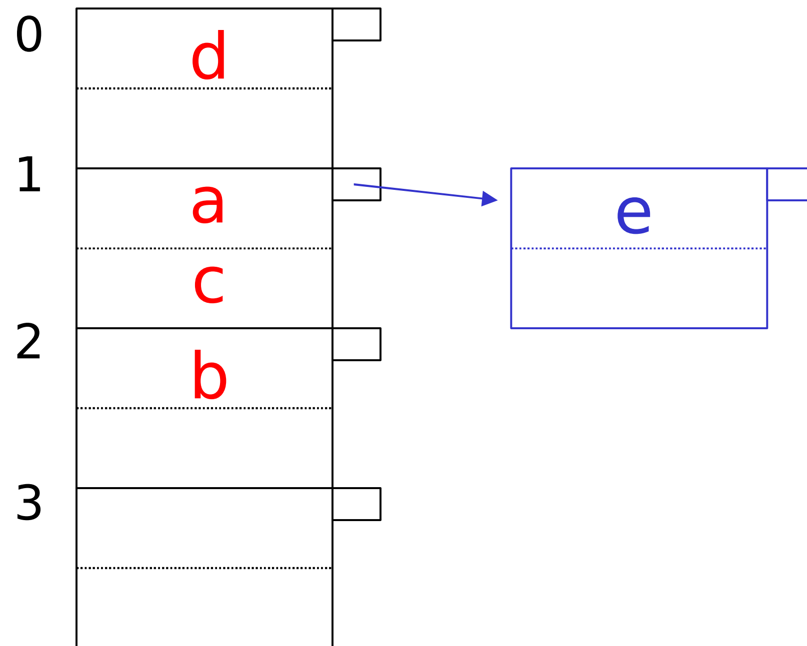
h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0

h(e) = 1



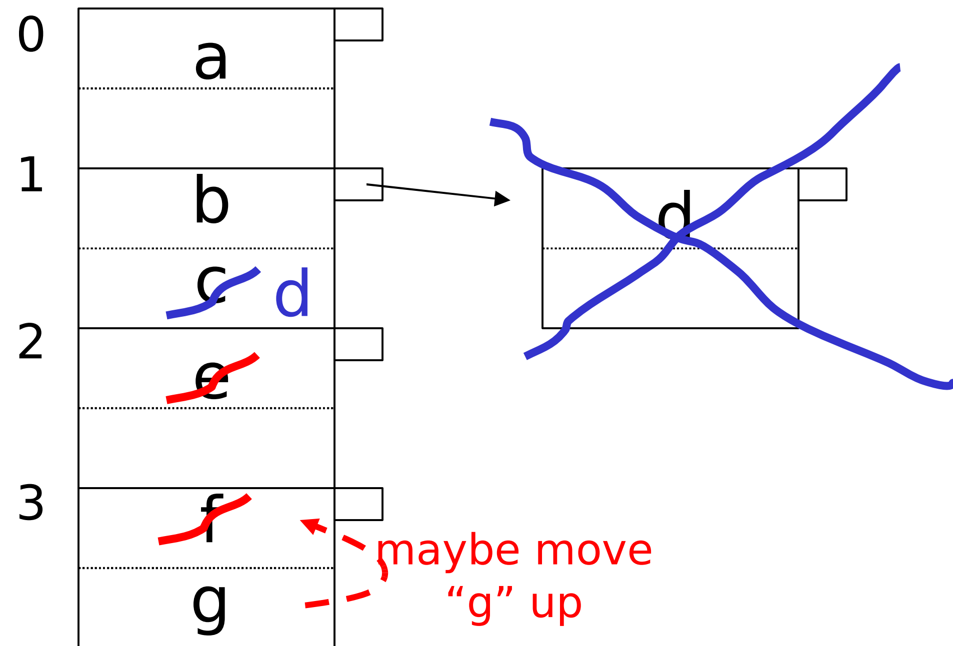
EXAMPLE: deletion

Delete:

e

f

c



Rule of thumb:

- Try to keep space utilization between 50% and 80%

$$\text{Utilization} = \frac{\text{\# keys used}}{\text{total \# keys that fit}}$$

- If < 50%, wasting space
- If > 80%, overflows significant

depends on how good hash function is & on # keys/bucket

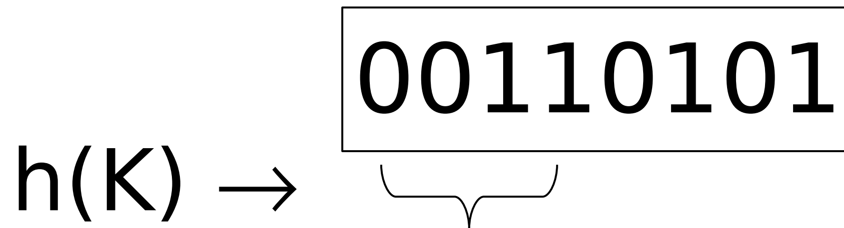
How do we cope with growth?

- Overflows and reorganizations
- Dynamic hashing

- Extensible
- Linear

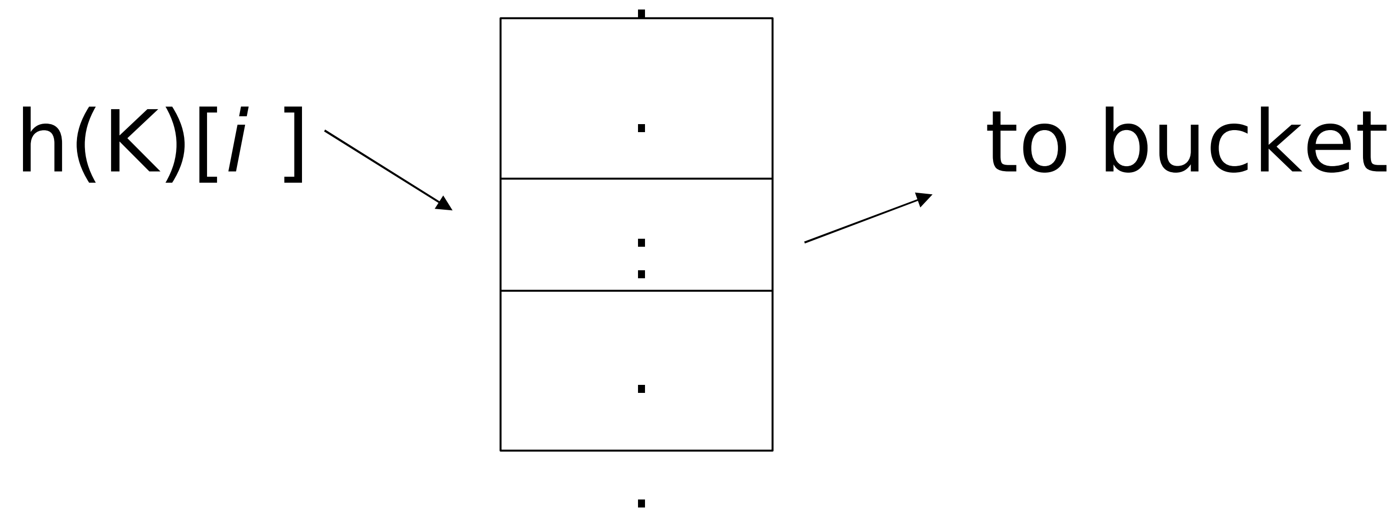
<u>Extensible hashing:</u> two ideas

(a) Use i of b bits output by hash function

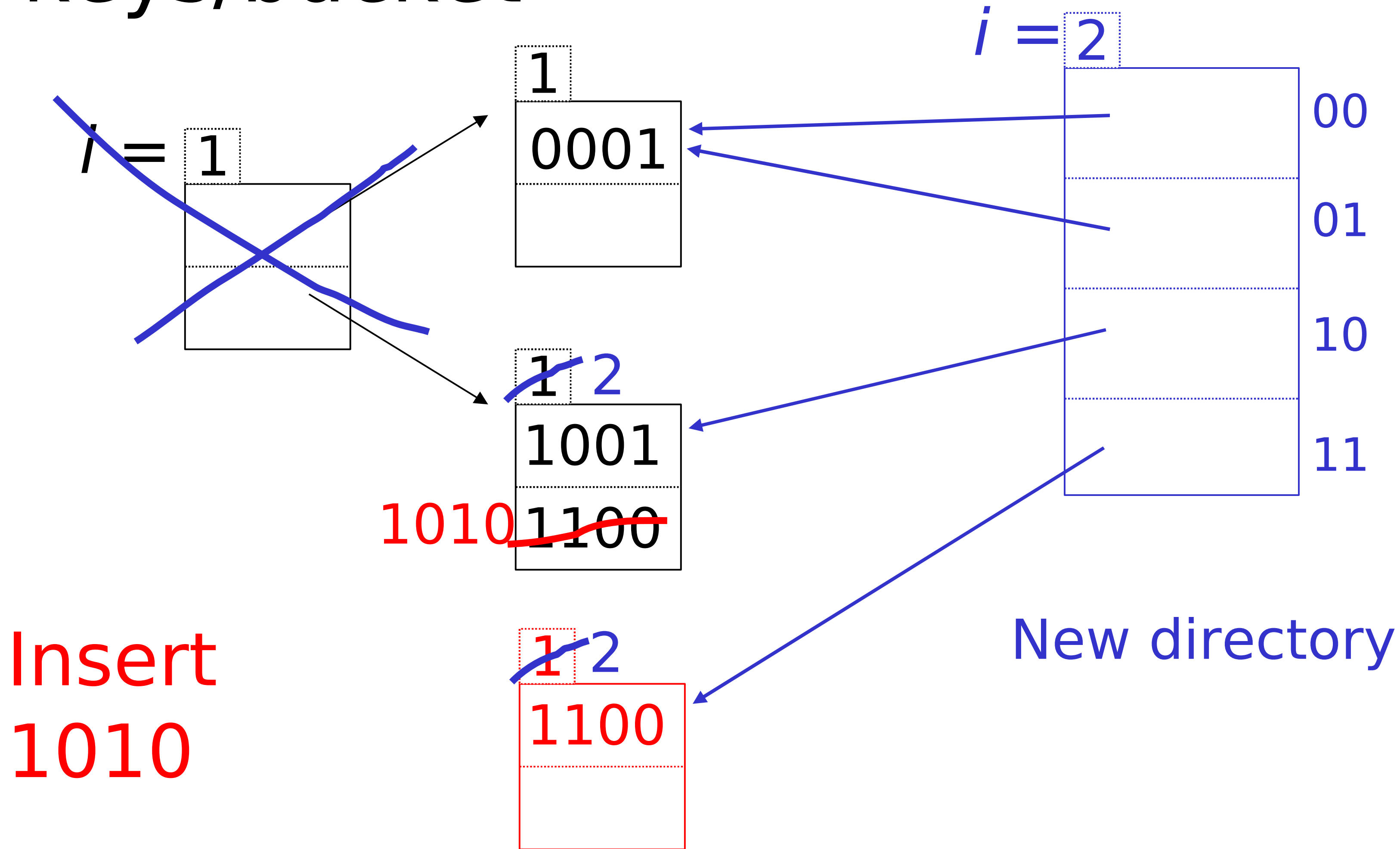


use i → grows over time….

(b) Use directory

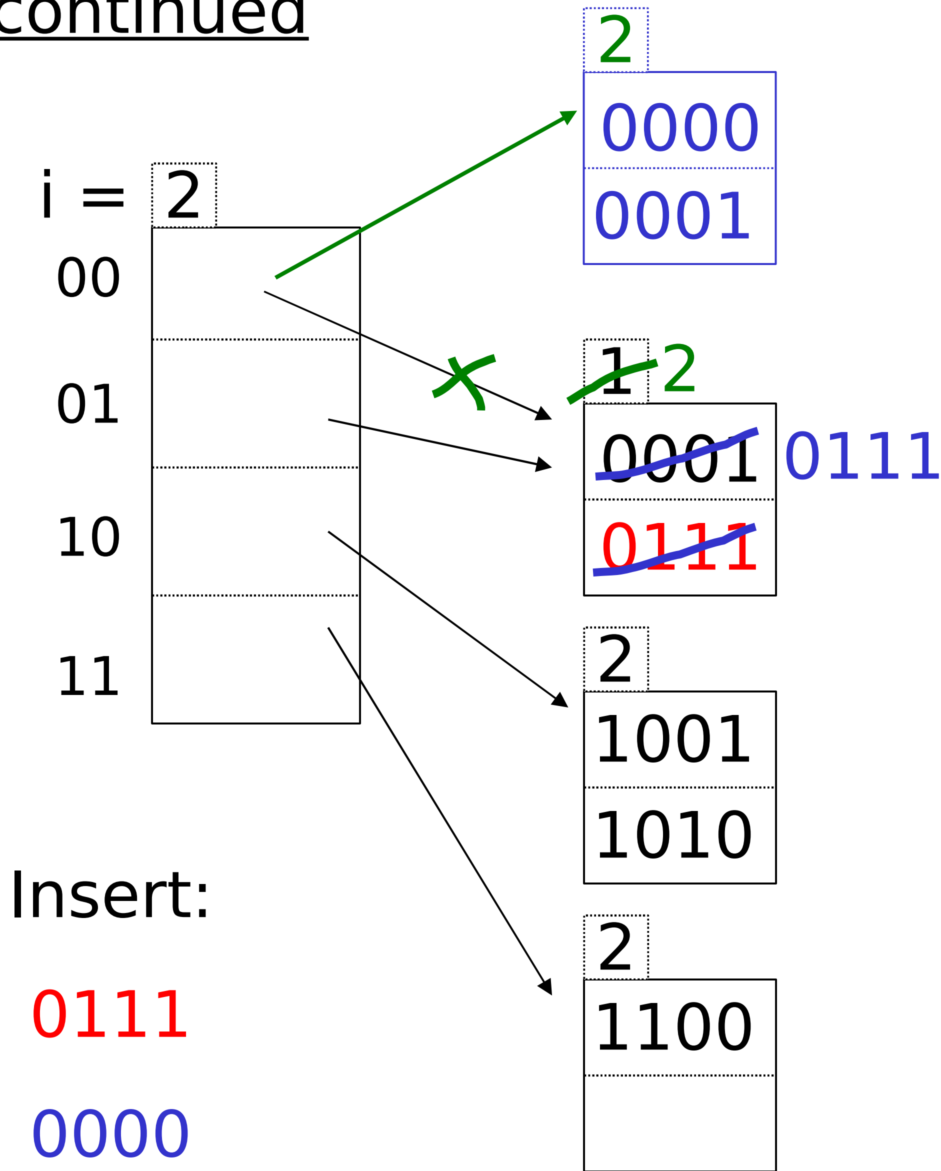


Example: h(k) is 4 bits; 2 keys/bucket

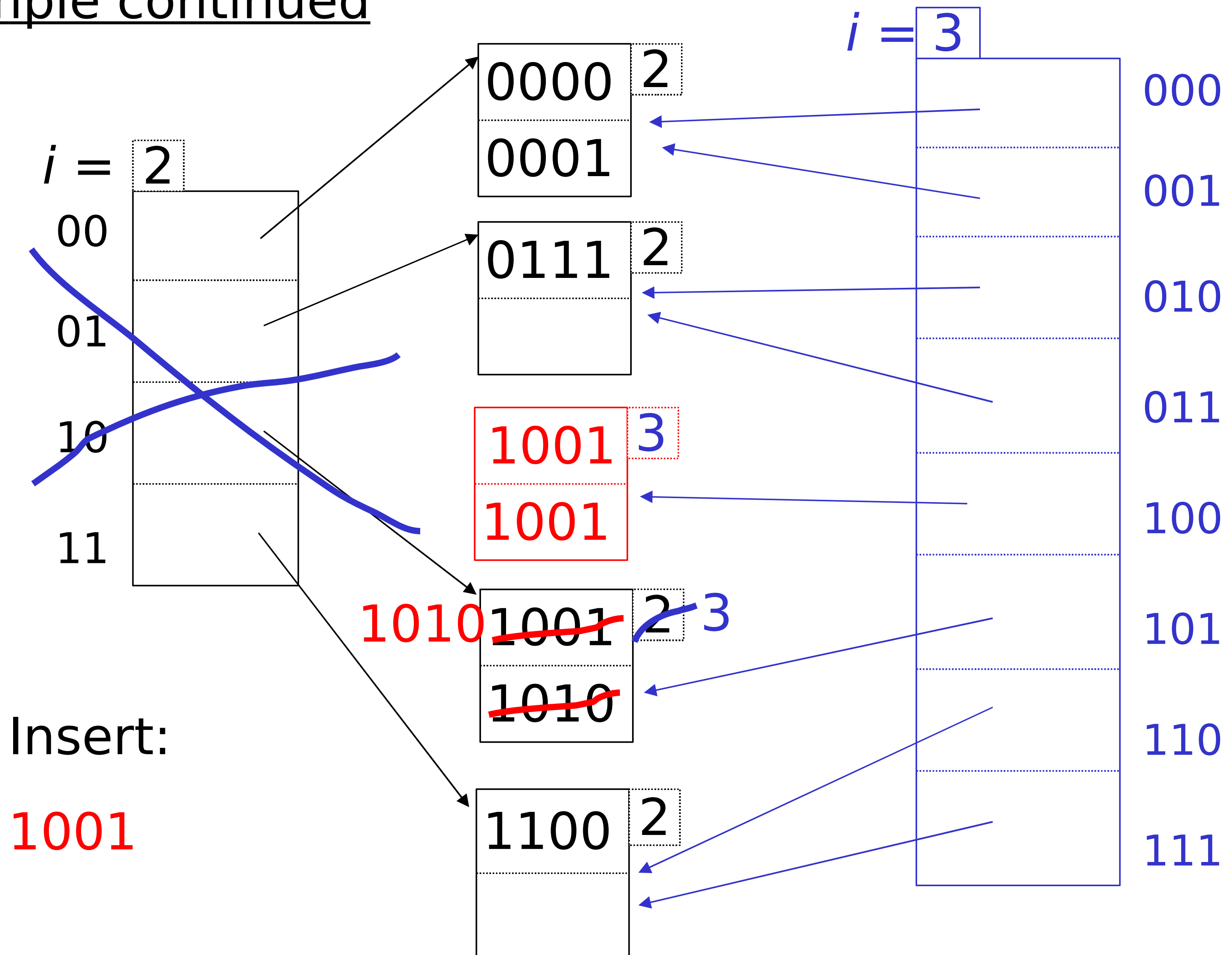


Insert 1010

Example continued



Example continued



Insert:

1001

Extensible hashing: deletion

- No merging of blocks
- Merge blocks
  and cut directory if possible
  (Reverse insert procedure)

Deletion example:

- Run thru insert example in reverse!

Summary Extensible hashing

(+) Can handle growing files

- with less wasted space
- with no full reorganizations

(-) Indirection

(Not bad if directory in memory)

(-) Directory doubles in size

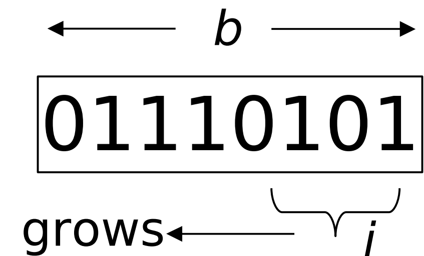
(Now it fits, now it does not)

Linear hashing

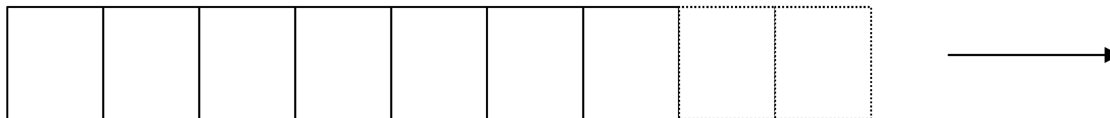
- Another dynamic hashing scheme

Two ideas:

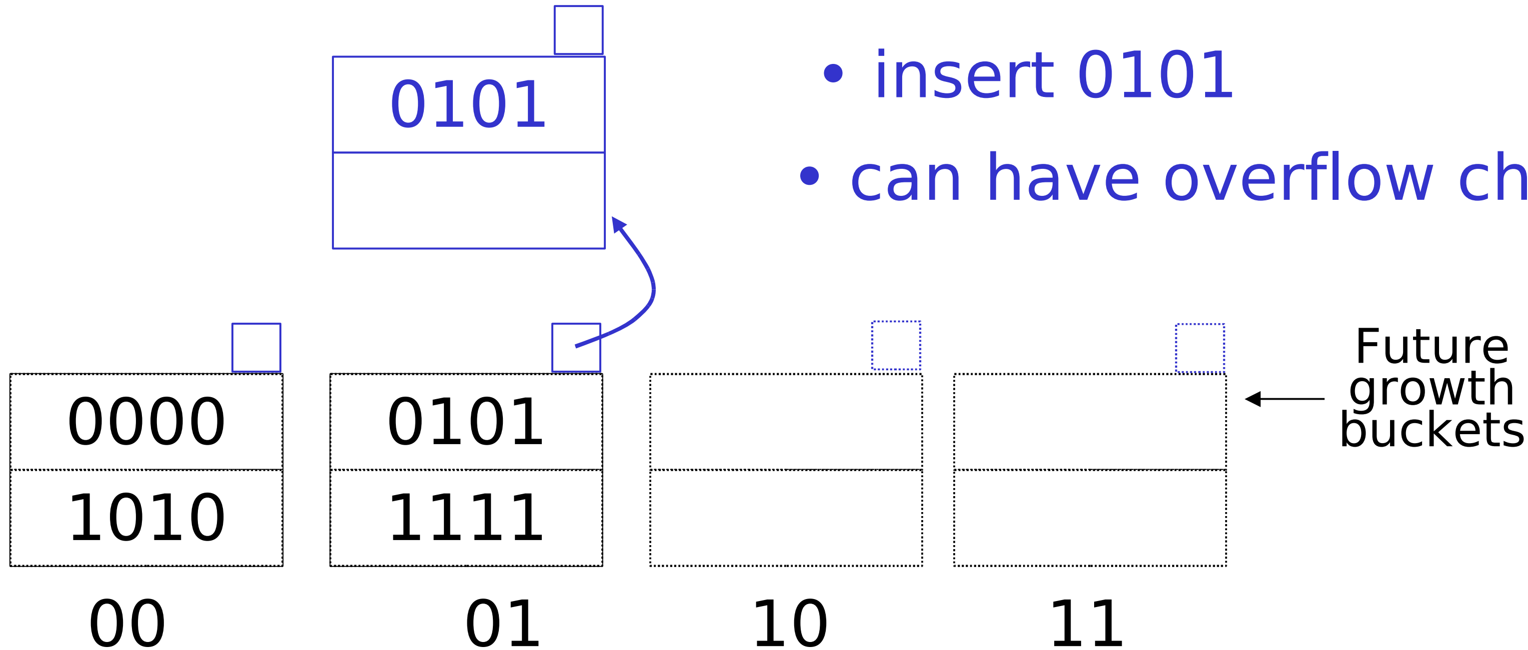
(a) Use i low order bits of hash



(b) Number n of buckets in use grows linearly



Example b=4 bits, i =2, 2 keys/bucket



- insert 0101
- can have overflow chains!

n = 01 (number of buckets in use)

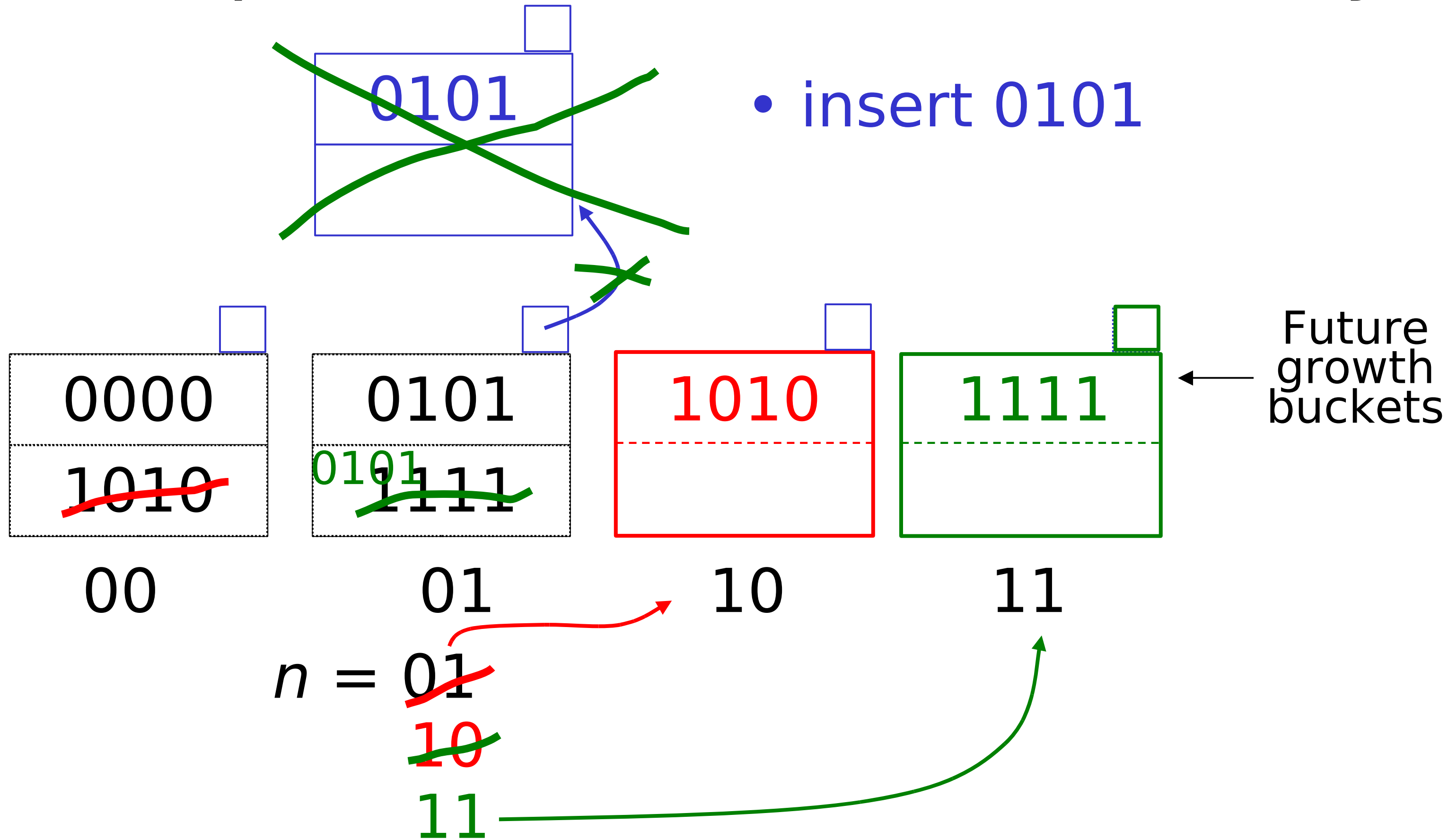
Rule

If h(k)[i] ≤ n, then

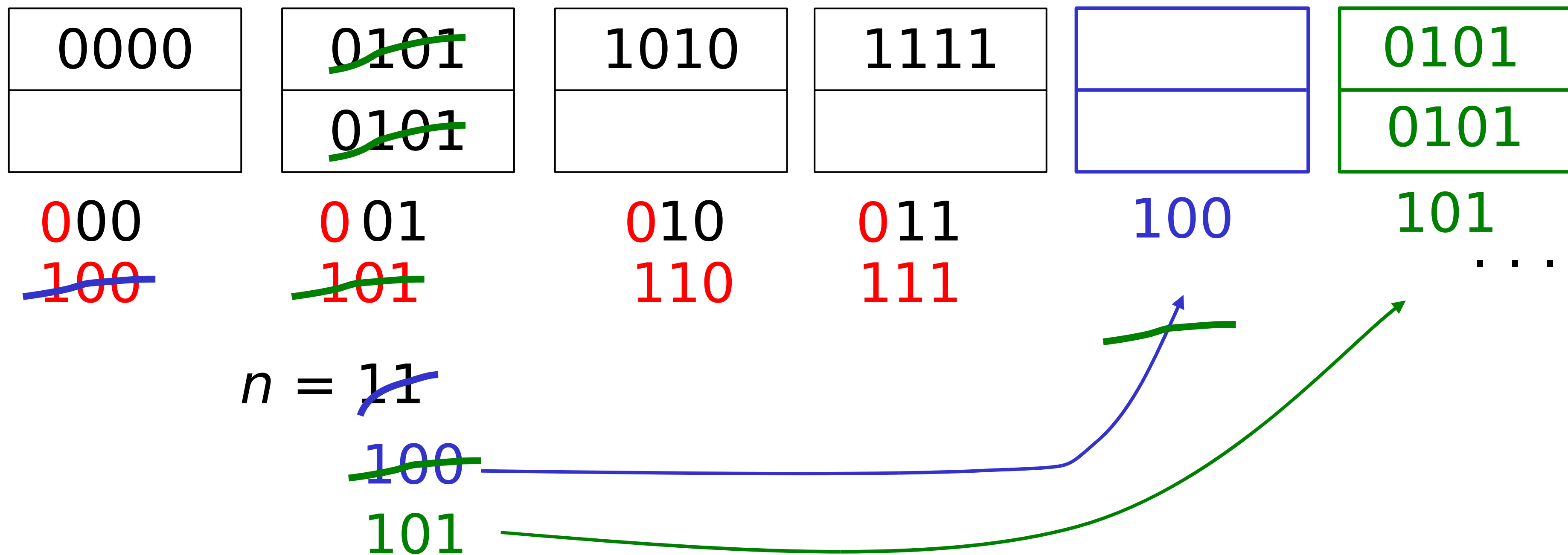
look at bucket h(k)[i]

else, look at bucket h(k)[i] - 2^{i-1}

Example b=4 bits, i =2, 2 keys/bucket



Example Continued: How to grow beyond this?



☛ When do we expand file?

- Keep track of: $\frac{\text{\# records}}{\text{\# buckets}} = U$

- If U > threshold then increase *n* (and maybe *i*)

Summary Linear Hashing

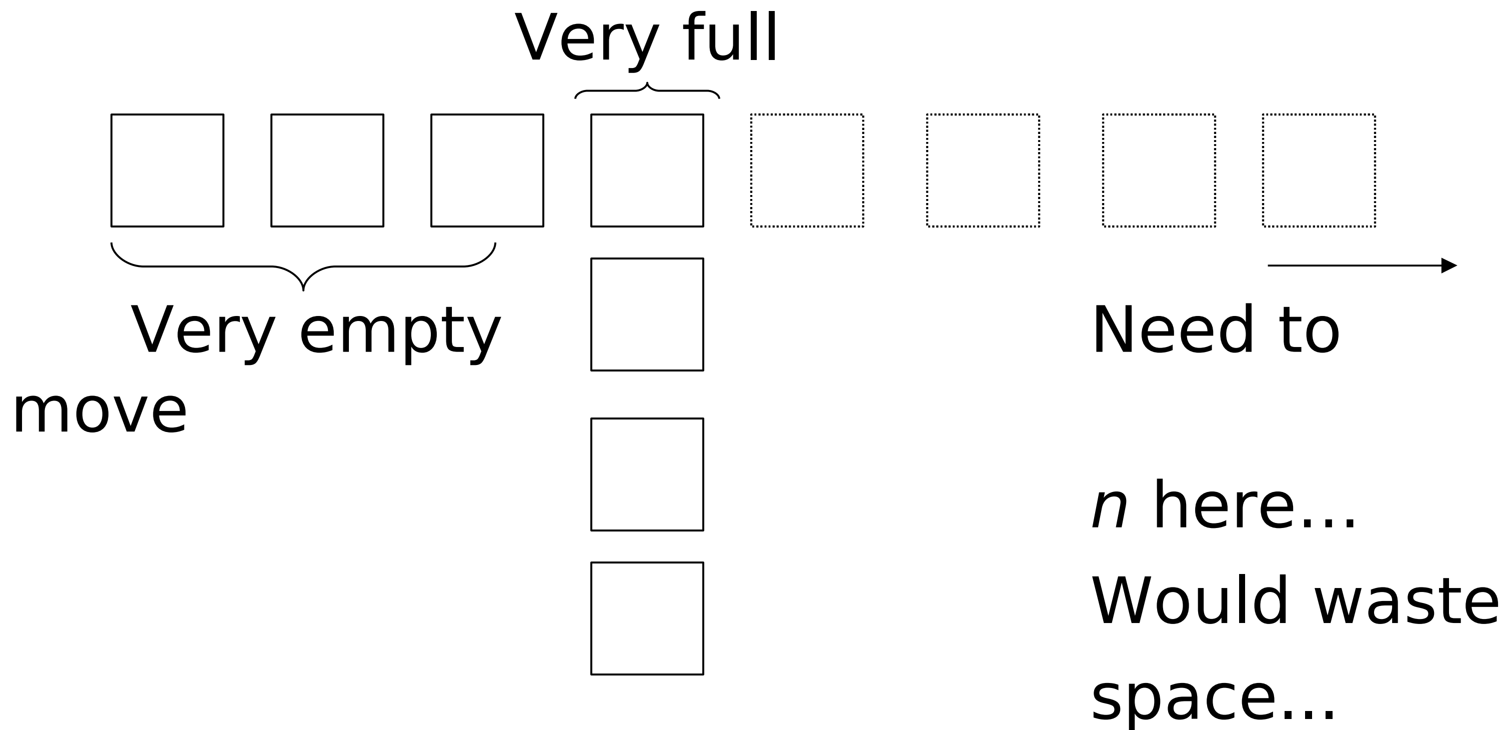
(+) Can handle growing files

- with less wasted space
- with no full reorganizations

(+) No indirection like extensible hashing

(-) Can still have overflow chains

Example: BAD CASE



Summary

<u>Hashing</u>

- How it works
- Dynamic hashing
  - Extensible
  - Linear

B+trees vs Hashing

- Hashing good for probes given key

e.g.,

```
SELECT ...
FROM R
WHERE R.A = 5
```

B+Trees vs Hashing

- INDEXING (Including B Trees) good for

  Range Searches:

  e.g., SELECT
  FROM R
  WHERE R.A > 5