# Index Structures

Chapter 4

# Outline

- Indexes on sequential files
- Secondary indexes
- B-Trees
- Hash Tables

# Introduction

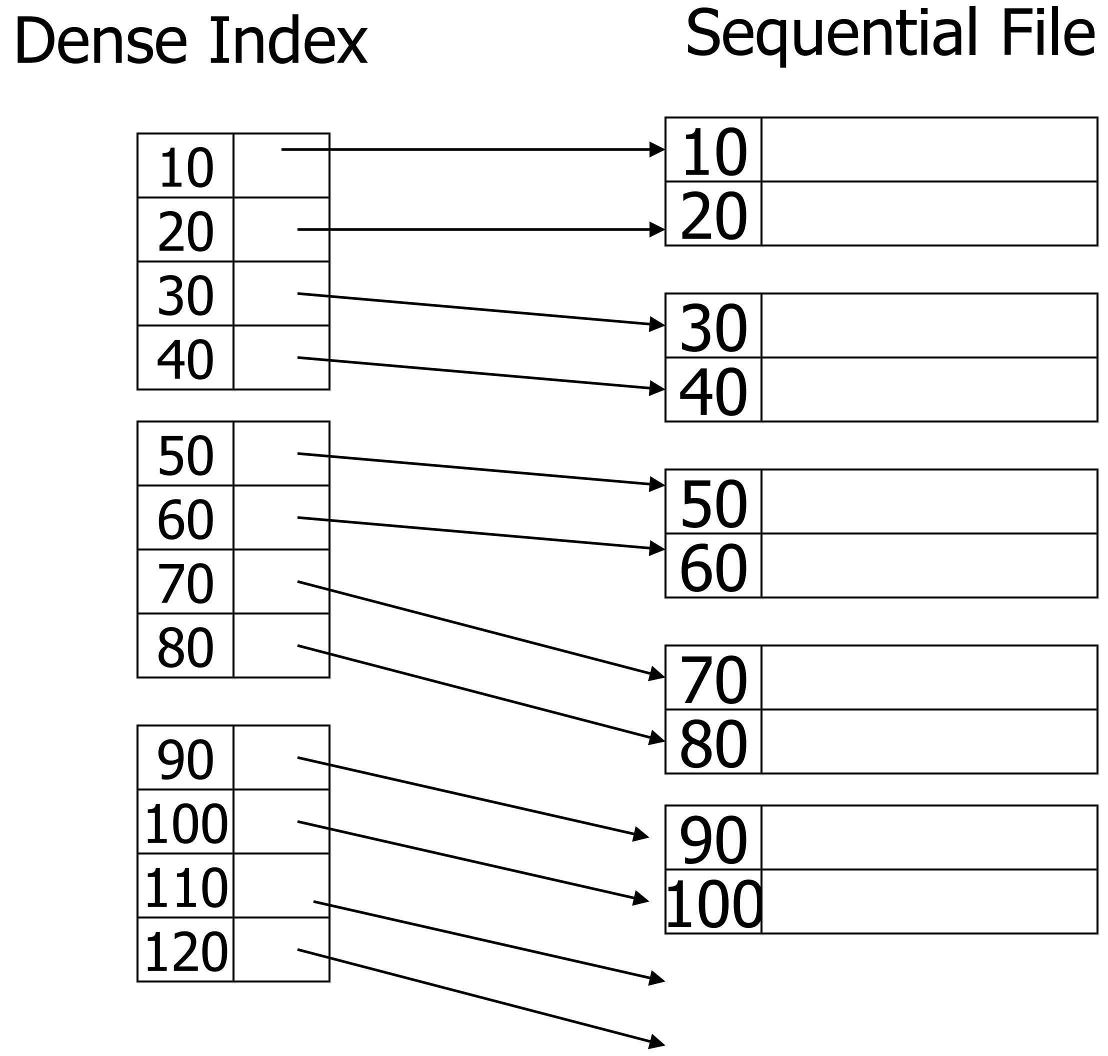
- SELECT * FROM R
  - Examine every block in the storage system
  - Enough information on block headers to identify where in the block records begin
  - Enough information in record headers to tell what relation the record belongs to
- Better organization
  - Reserve some blocks, several cylinders for a given relation
- Suppose we want to answer the following query
  - SELECT* FROM MOvieStar WHERE name=`Jim Carrey’
    - Scan all the blocks on which MovieStar tuples could be found.
- To execute the query quickly, we create one or more indexes.

# Indexes on Sequential Files

- A sorted file is called the index file
- A search key K in the index file is associated with a pointer to a data-file record that has search key K.
- Dense index
  - There is an entry in the idex file for every record in the data file.
- Sparse index
  - Only some of the data records are represented in the data file.

Sequential File

| | |
|---|---|
| 10 | |
| 20 | |
| 30 | |
| 40 | |
| 50 | |
| 60 | |
| 70 | |
| 80 | |
| 90 | |
| 100 | |

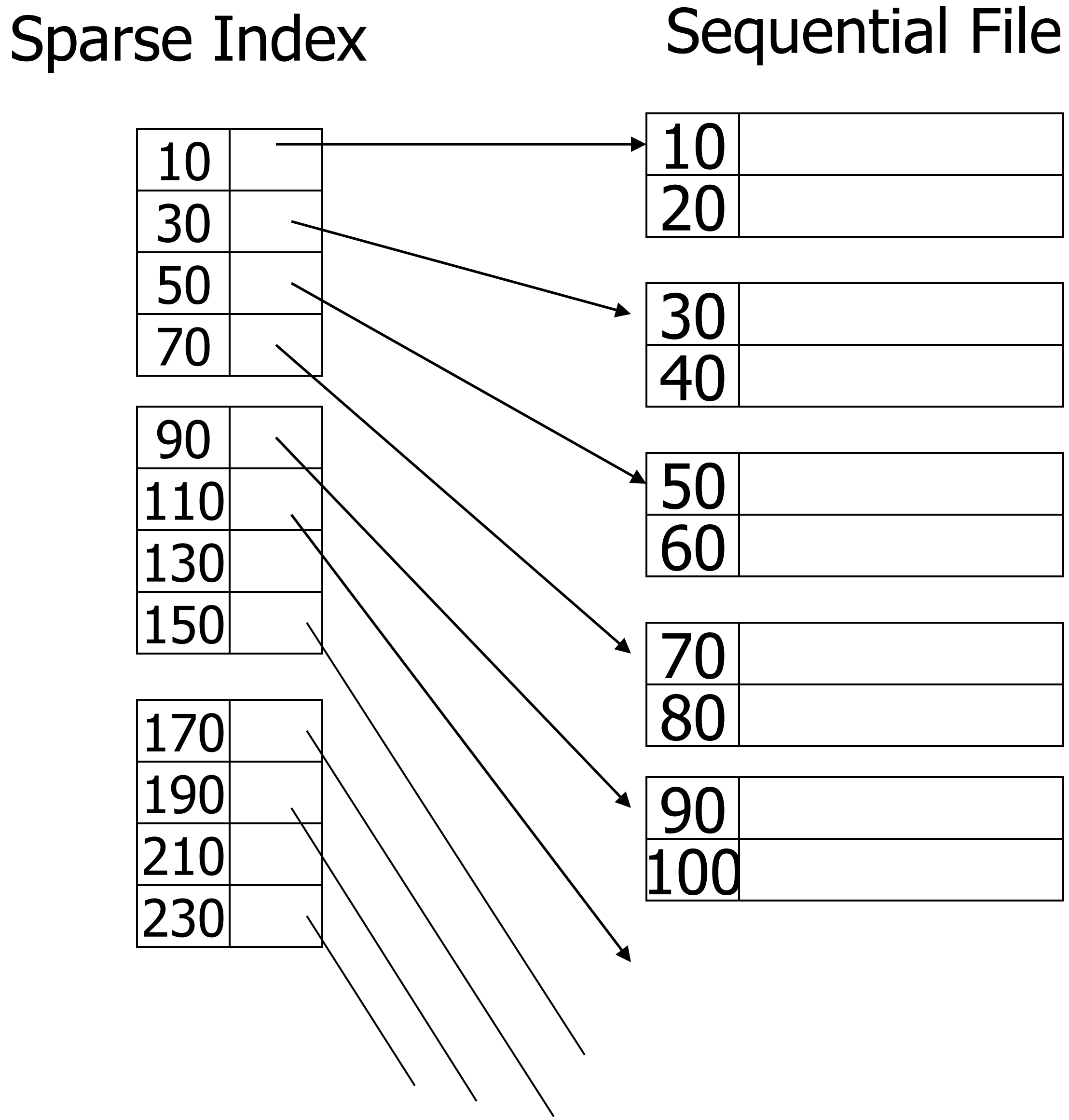

Dense Index
Sequential File
10
20
30
40
50
60
70
80
90
100
110
120
10
20
30
40
50
60
70
80
90
100

# About Dense Index

- The dense index supports queries that ask for records with a given search key value.
- The index-based search is efficient
  - Number of index blocks is usually small compared with the number of data blocks
  - We can use binary search $\log_2 n$ of them
  - The index is small enough to be kept permanently in main memory buffers.
    - So requires only main memory accesses

# About Sparse Indexes

- If a dense index is too large, we can use similar structure called a sparse index
  - Uses less space but requires some more time to find a record.
  - A sparse index holds only one key-pointer to the first record on the block

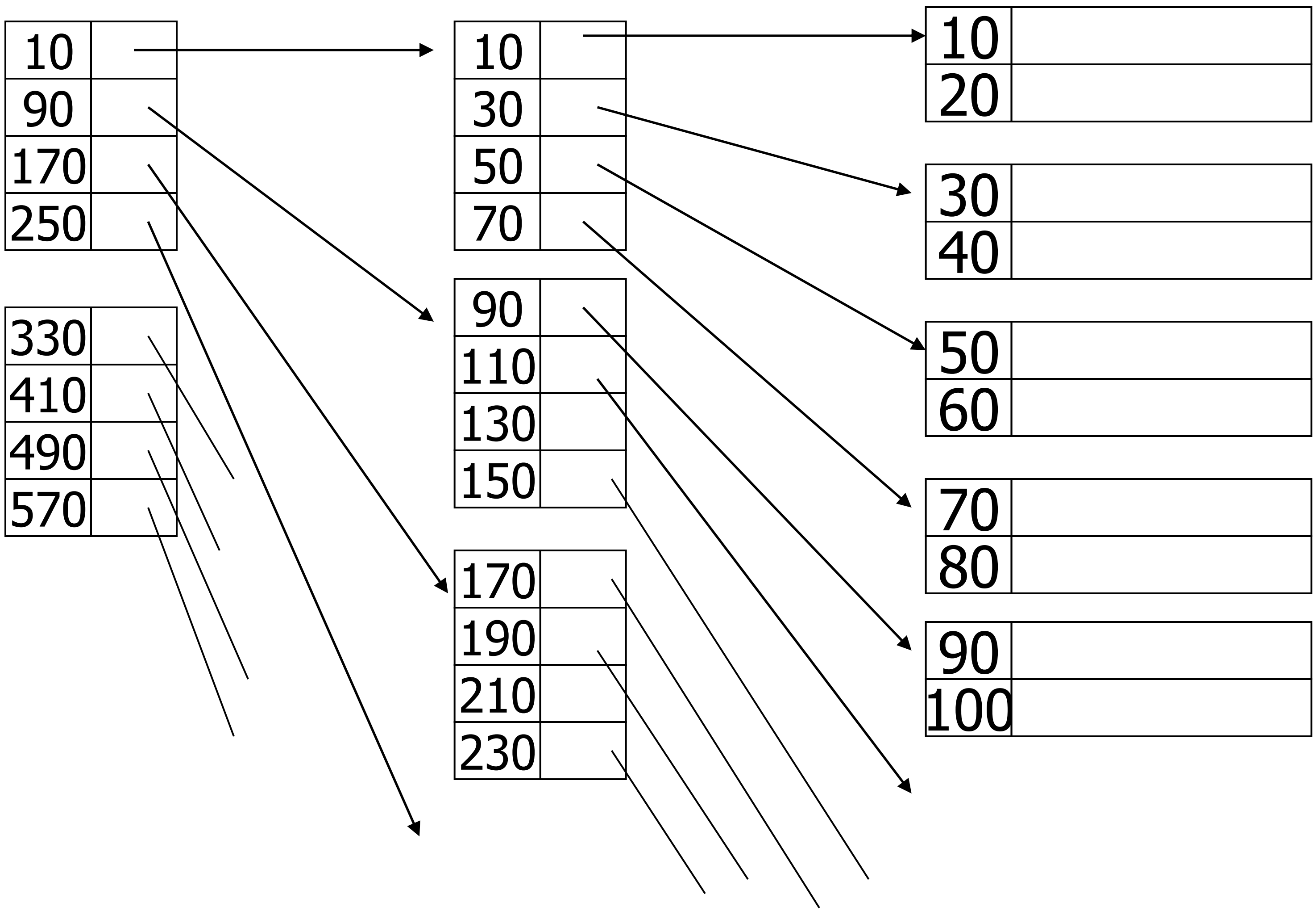

Sparse Index
Sequential File
10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100

# Multiple-levels  of index

- As index can cover several blocks, search is required
- By putting index on index we can make first level index more efficient.

Sparse 2nd level

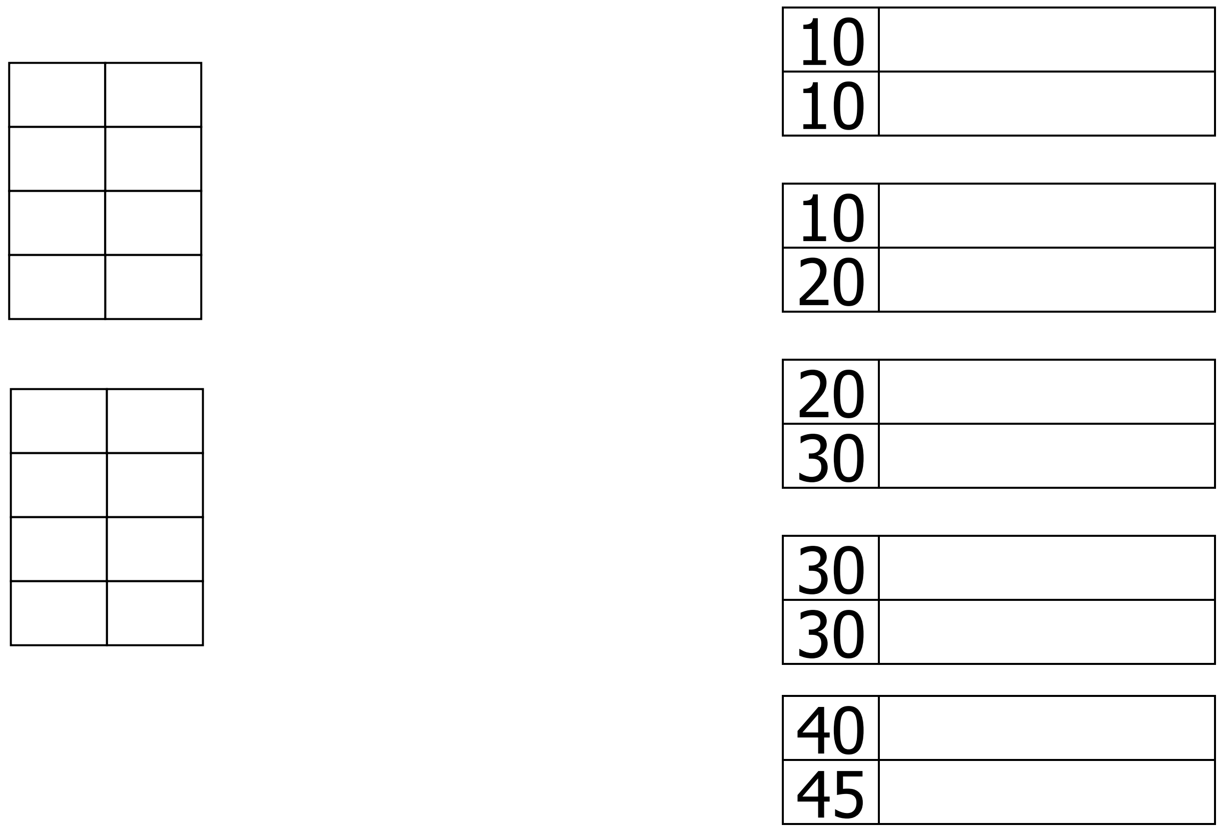
Sequential File



# Indexes with duplicate search keys

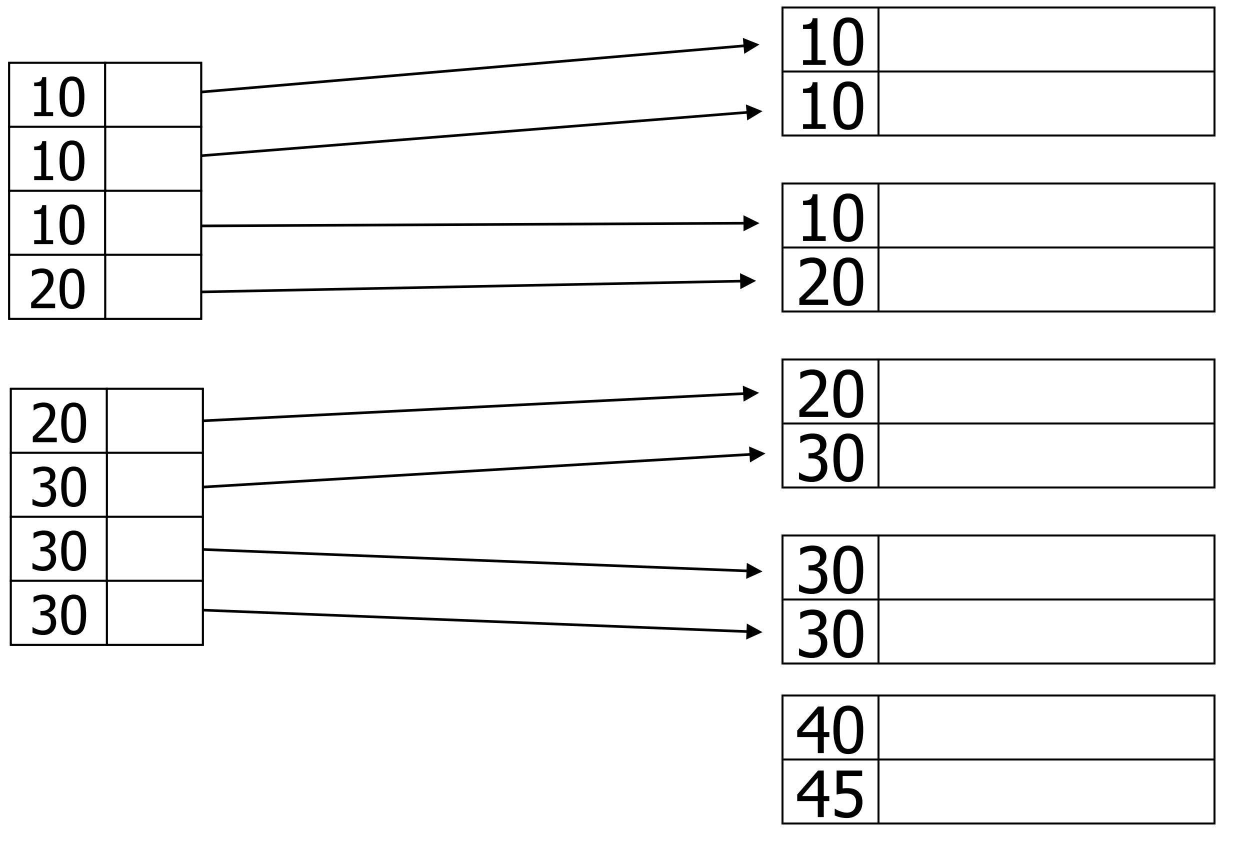
- So far, search key is the key of the relation.
- Indexes can be used on non-key attributes.
  - More than one record has the same key value
- Have a dense index with one entry with key K for each record of the data file that has search key K.

# Duplicate keys



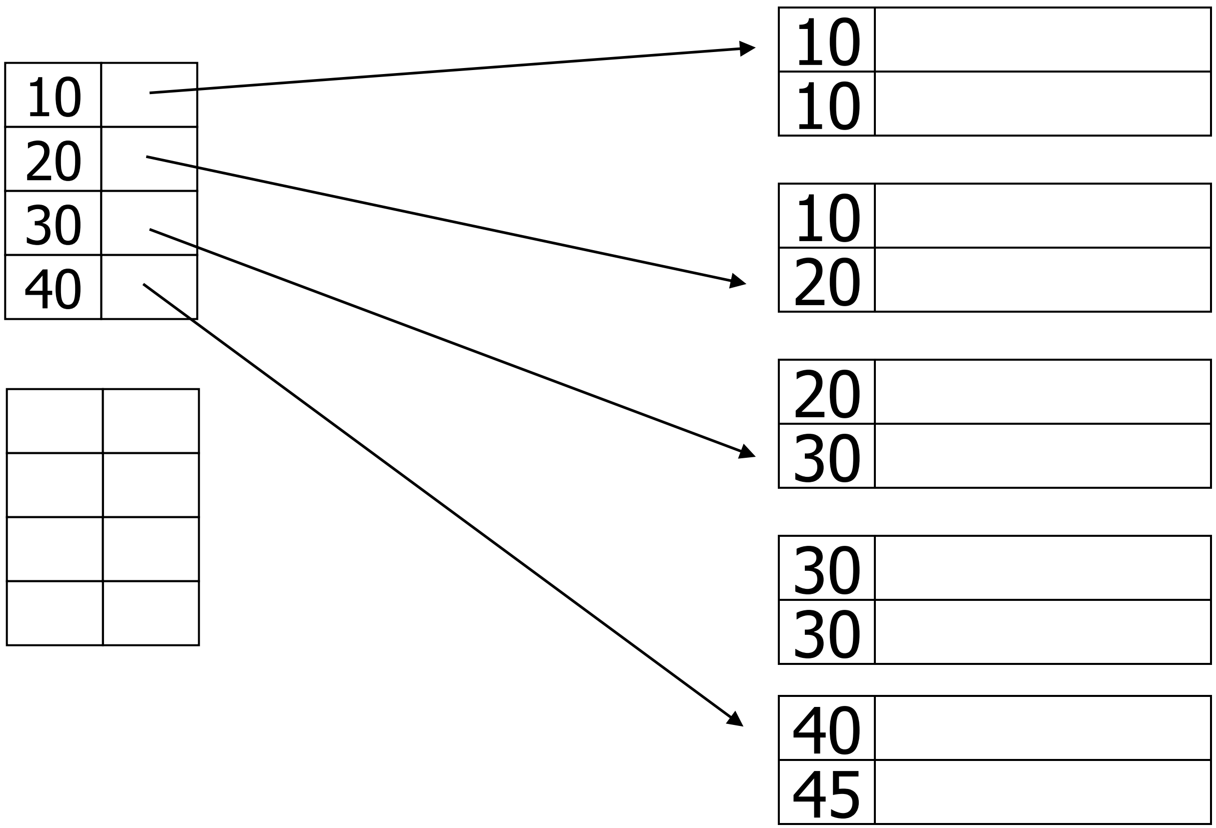
# Duplicate keys

Dense index, one way to implement?



# Duplicate keys

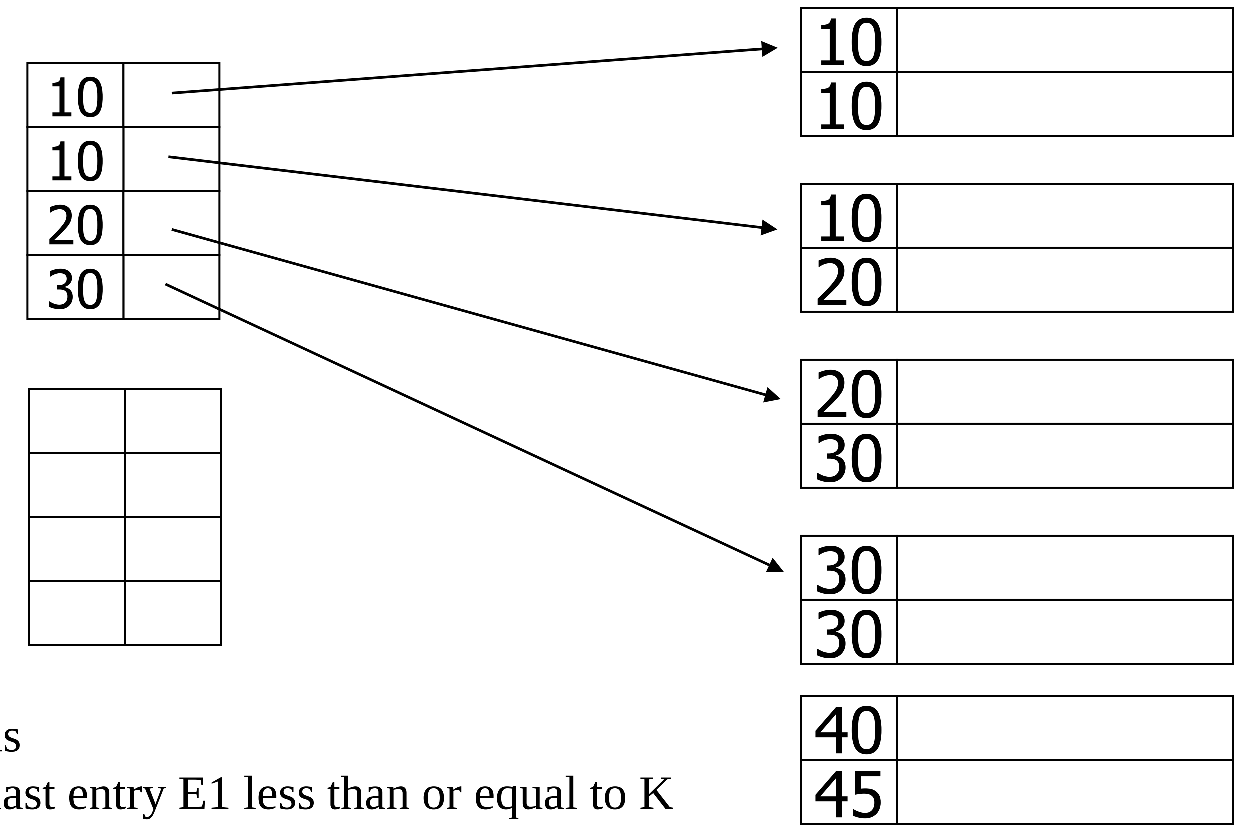
## Dense index, better way?



# Duplicate keys

* First search key on the each block of data file

## Sparse index, one way?




Finding the records
•We first find the last entry E1 less than or equal to K
• Move to the front
    •Either find the first
    •Find E2 with a key strictly less than K

# Managing Indexes During Data Modifications

- Records will be inserted, deleted and sometimes updated.
  - As a result the sequential file will evolve.
- Solution:
  - Create overflow blocks
    - Do not have entries in sparse index
  - Insert new blocks in the sequential order
    - New block needs an entry in the sparse index
  - If there is no space, slide tuples to adjacent blocks.
    - There will be changes into index

# Index modifications

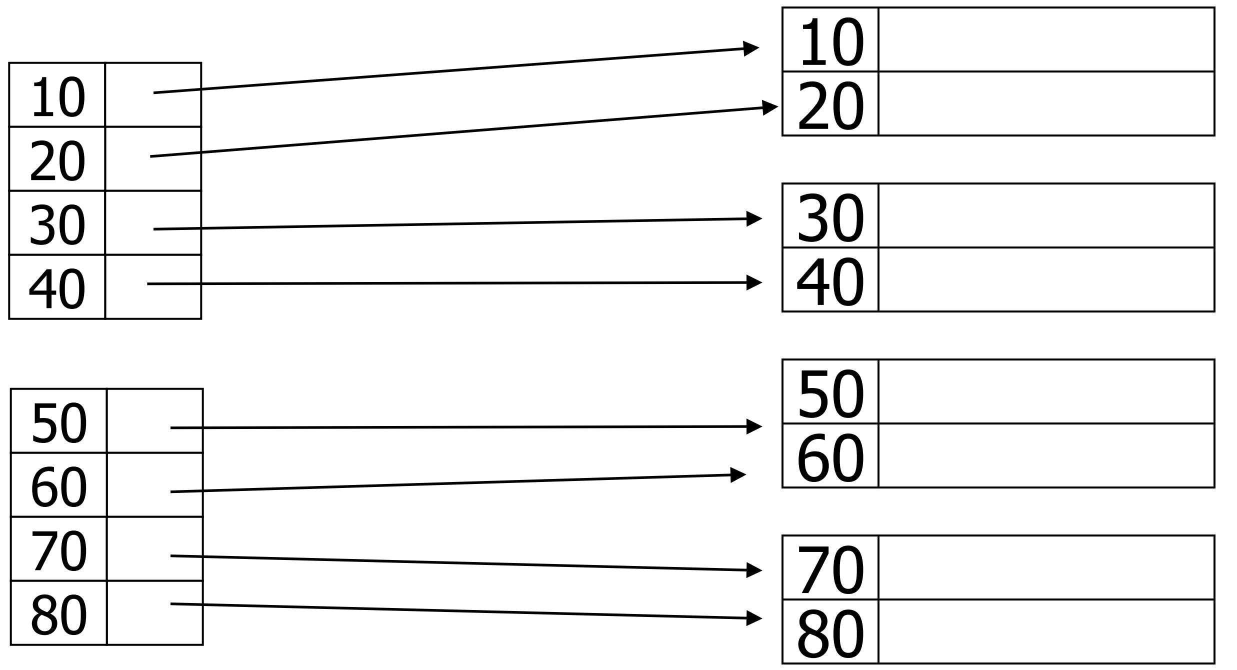
- Creating or deleting overflow block
  - has no effect on dense index or sparse index as index is created on primary blocks
- Creating or destroying blocks of the sequential file
  - has no effect on dense index because the index refers to records not blocks. It does effect a sparse index, since we must insert or delete an index entry for the block created or destroyed.
- Inserting or deleting records
  - has the same action on a dense index as a key-pointer pair for that record is inserted or deleted. But there is no effect of sparse index; if it is the first record of the block then the corresponding key value in the sparse index must be updated.
- Sliding a record
  - Within a block or among the blocks, results in updating to the corresponding entry of dense index.
  - Sparse index is updated if the moved record is the first record.

# Index Modifications

- When changes occur to data file, we must often change the index to adapt.
- Strategy
  - An index file is an example of a sequential file; the key-pointer pairs can be treated as records sorted by the value of search key.
    - Same strategies adapted for data files can be used

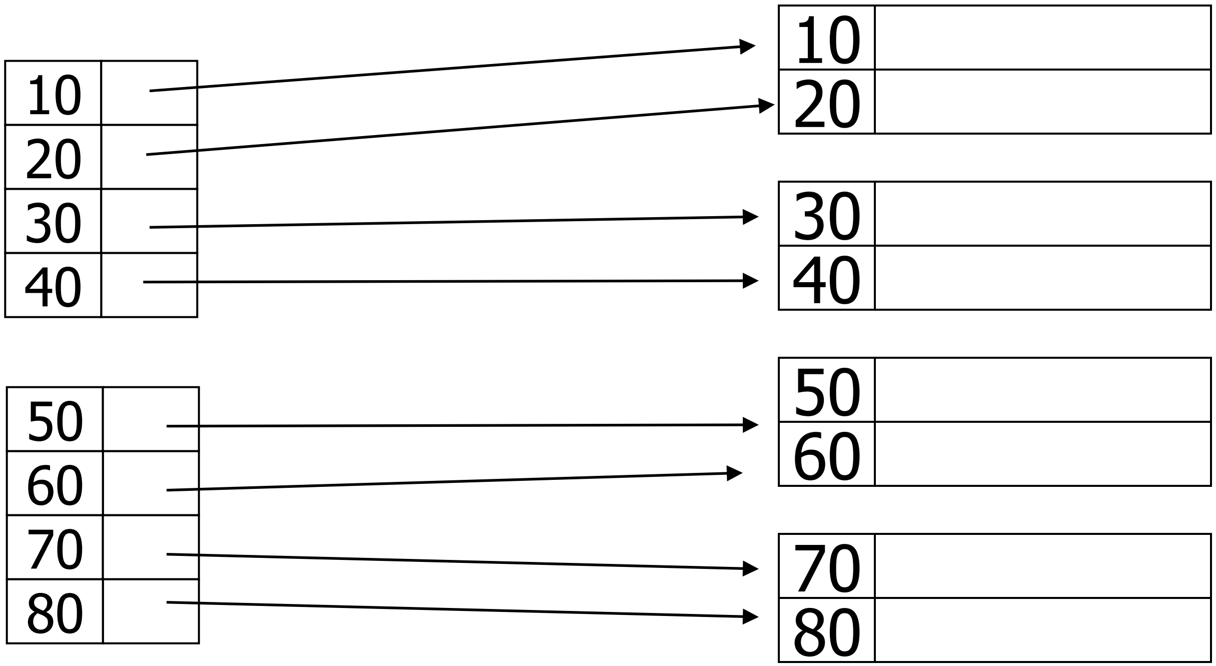
| **Action** | **Dense Index** | **Sparse Index** |
|---|---|---|
| Create empty overflow block | none | None |
| Delete empty overflow block | None | None |
| Create empty sequential block | None | Insert |
| Delete empty sequential block | None | Delete |
| Insert record | Insert | Update (?) |
| Delete record | Delete | Update (?) |
| Slide record | update | Update (?) |

# Deletion from dense index



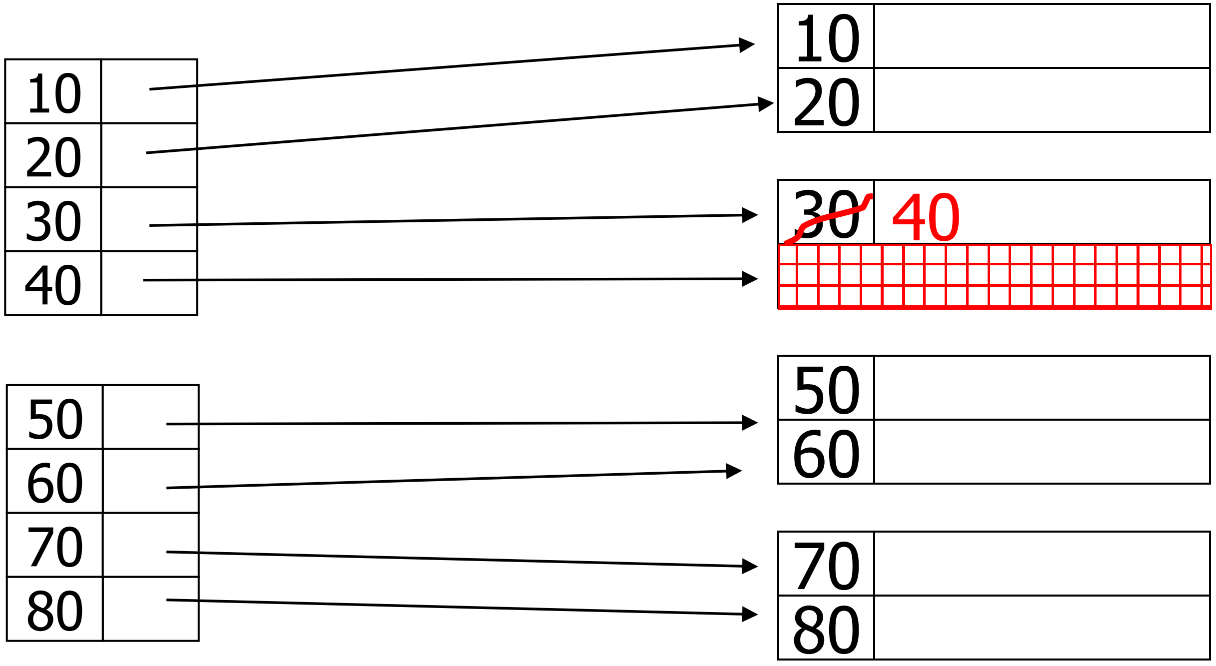
# Deletion from dense index

– delete record 30



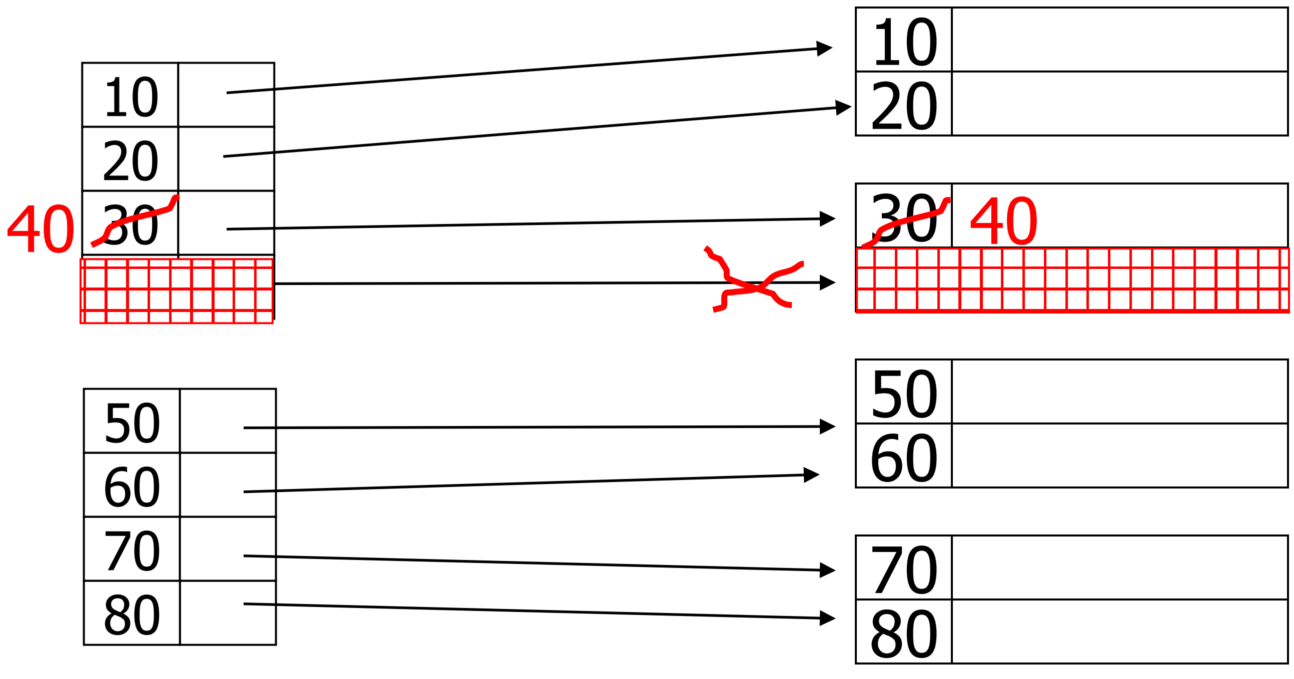
# Deletion from dense index

– delete record 30

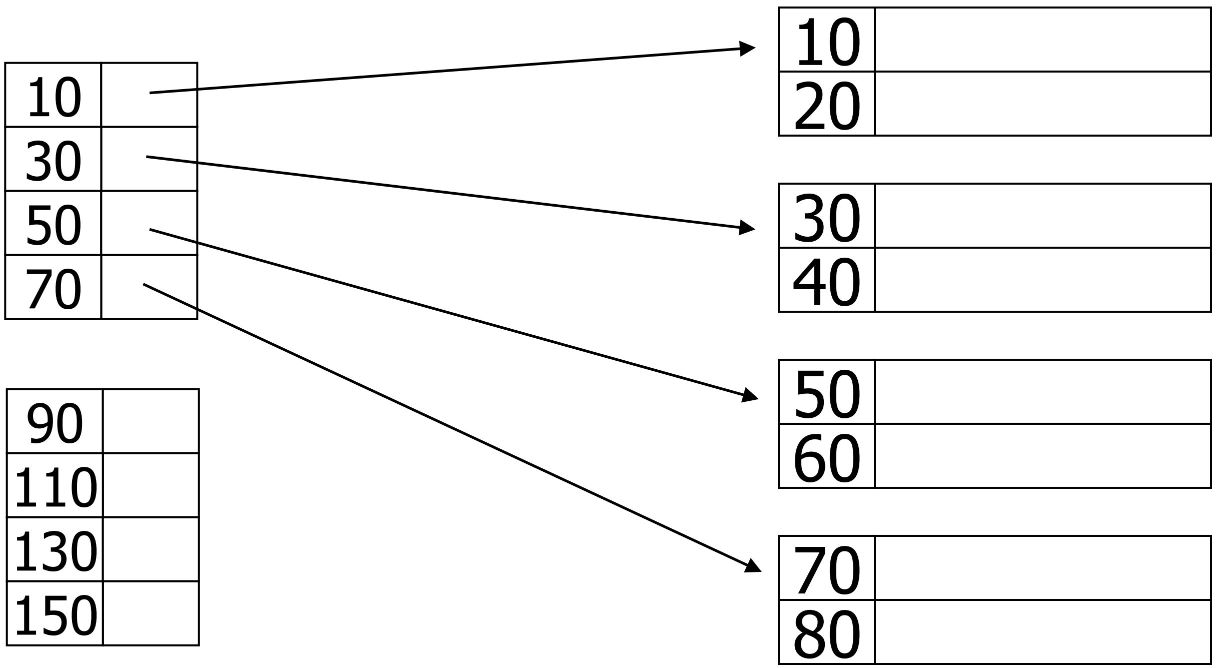


# Deletion from dense index

– delete record 30

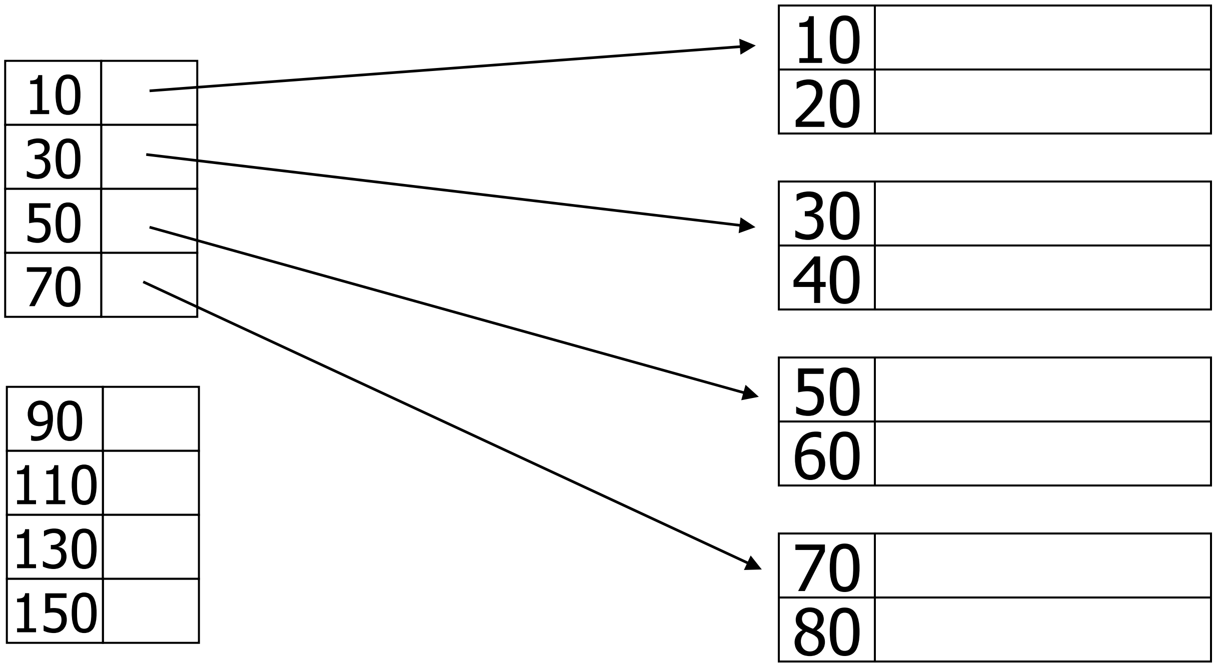


# Deletion from sparse index



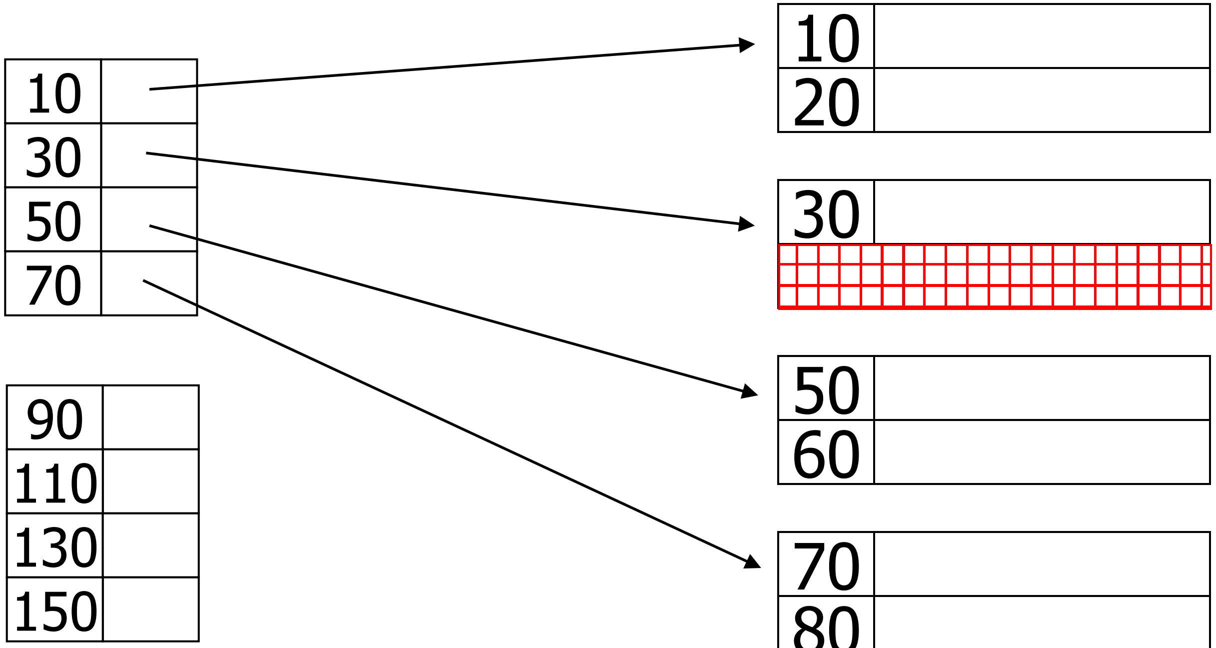
# Deletion from sparse index

– delete record 40



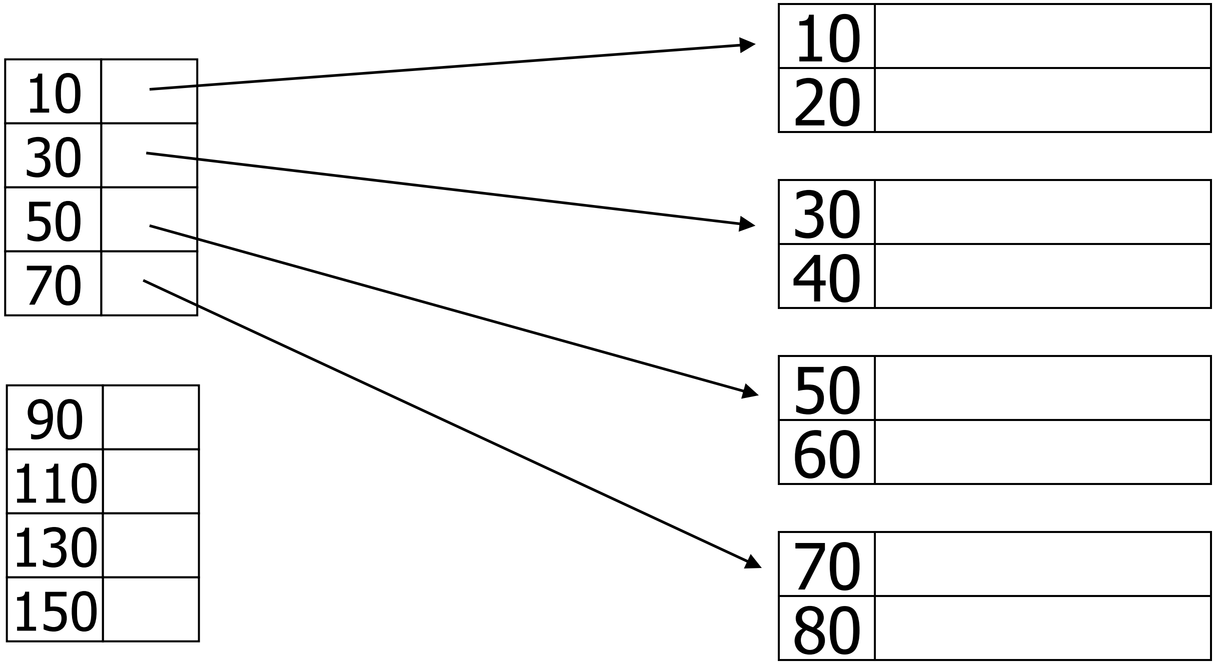
# Deletion from sparse index

– delete record 40



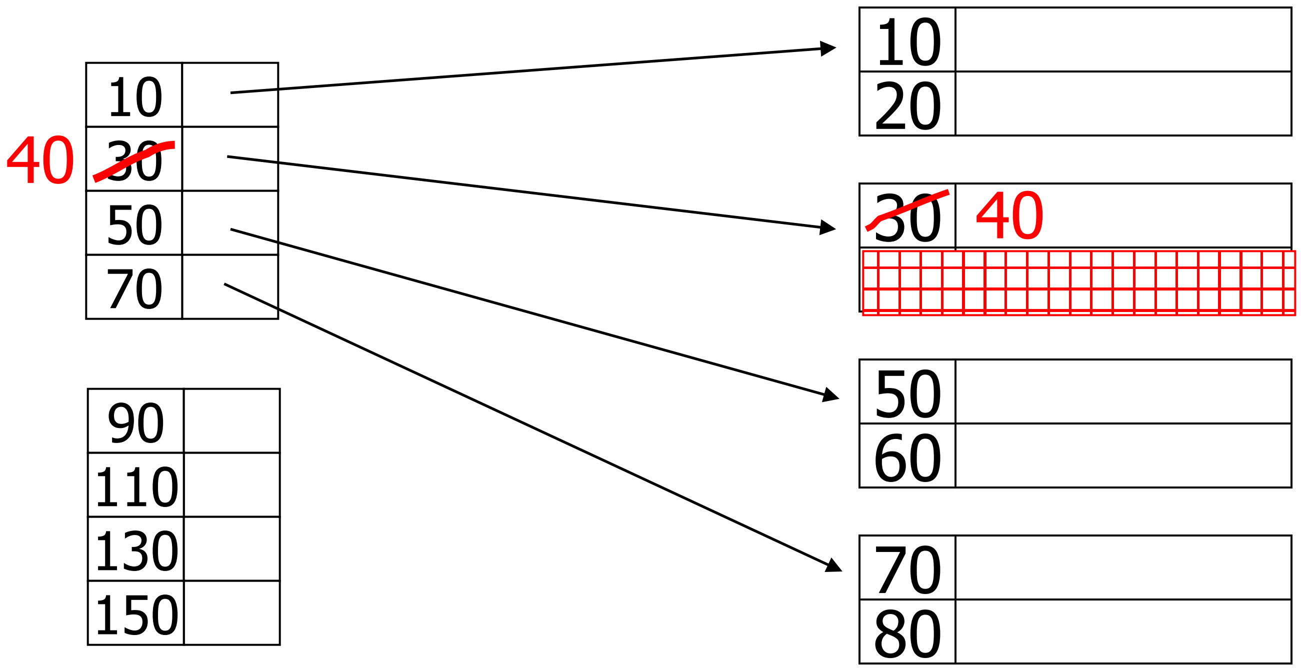
# Deletion from sparse index

– delete record 30



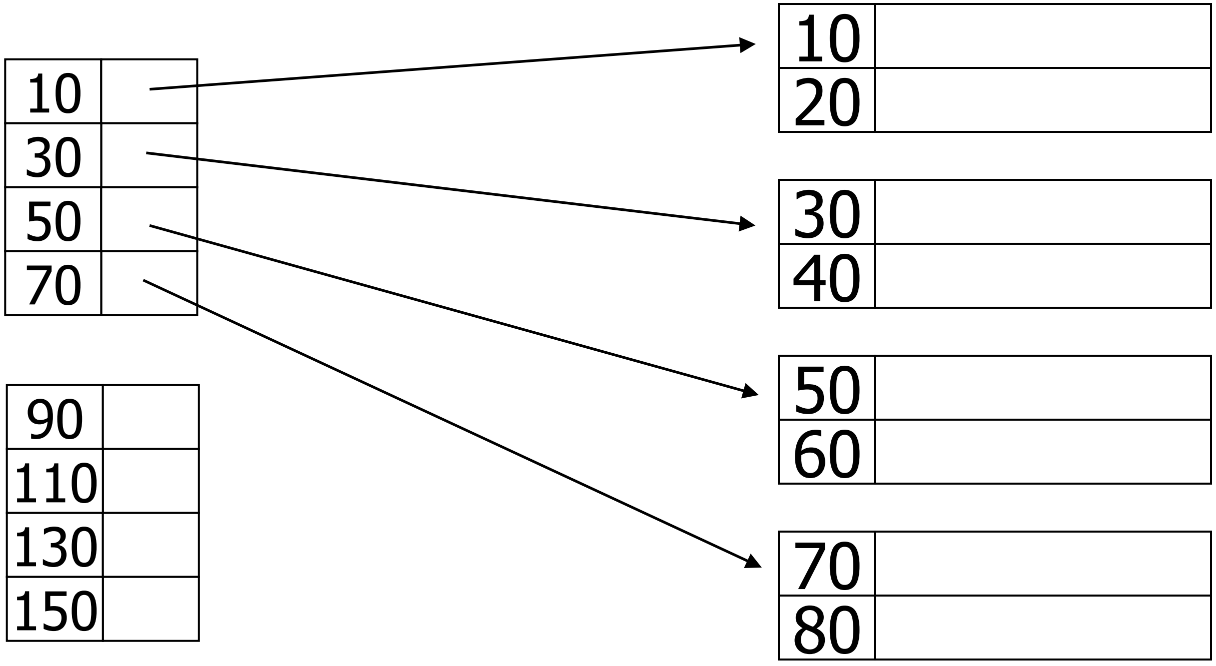
# Deletion from sparse index

– delete record 30



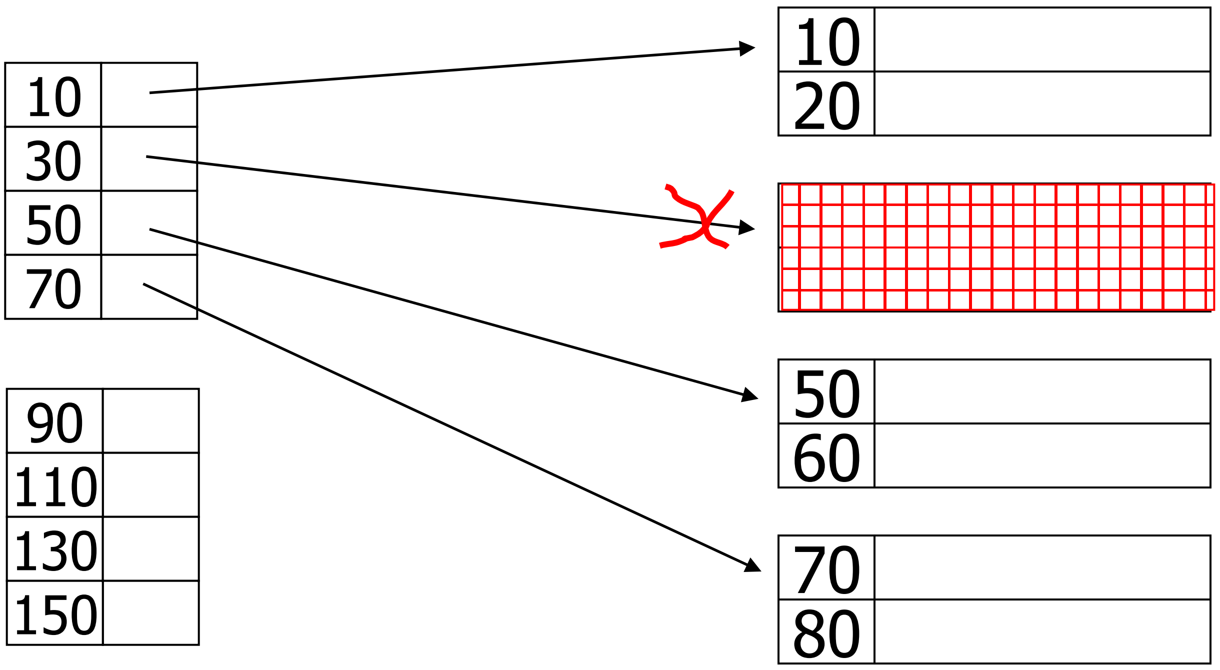
# Deletion from sparse index

– delete records 30 & 40



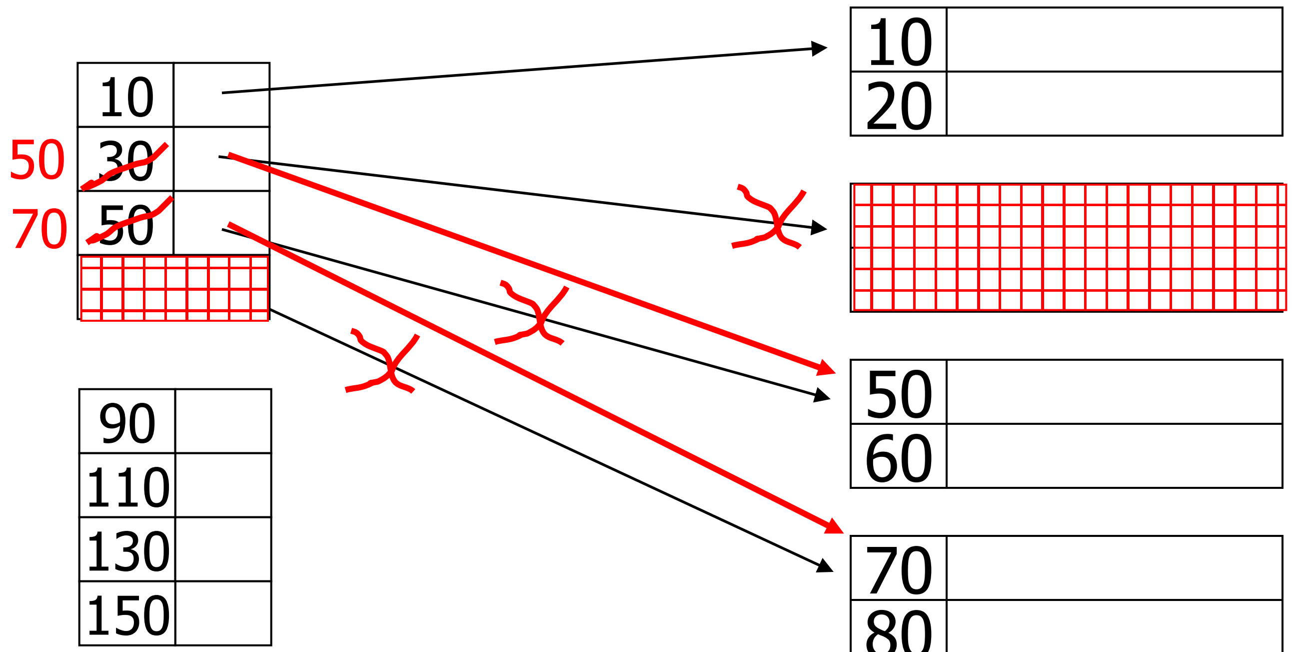
# Deletion from sparse index

– delete records 30 & 40

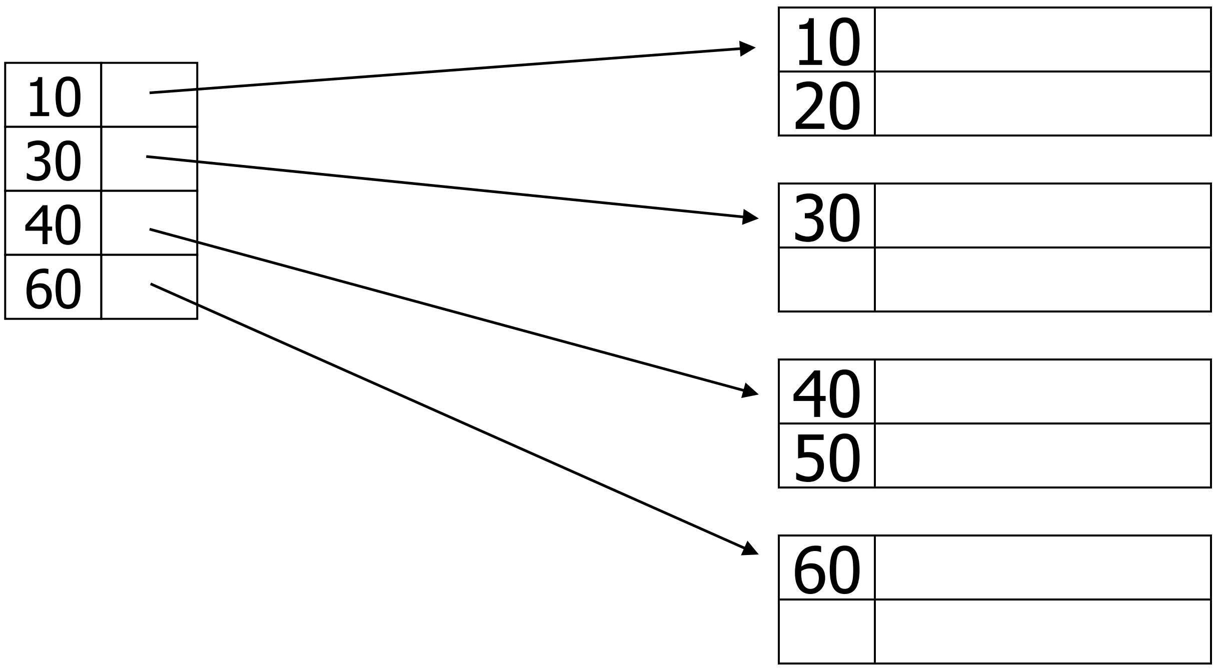


# Deletion from sparse index

– delete records 30 & 40

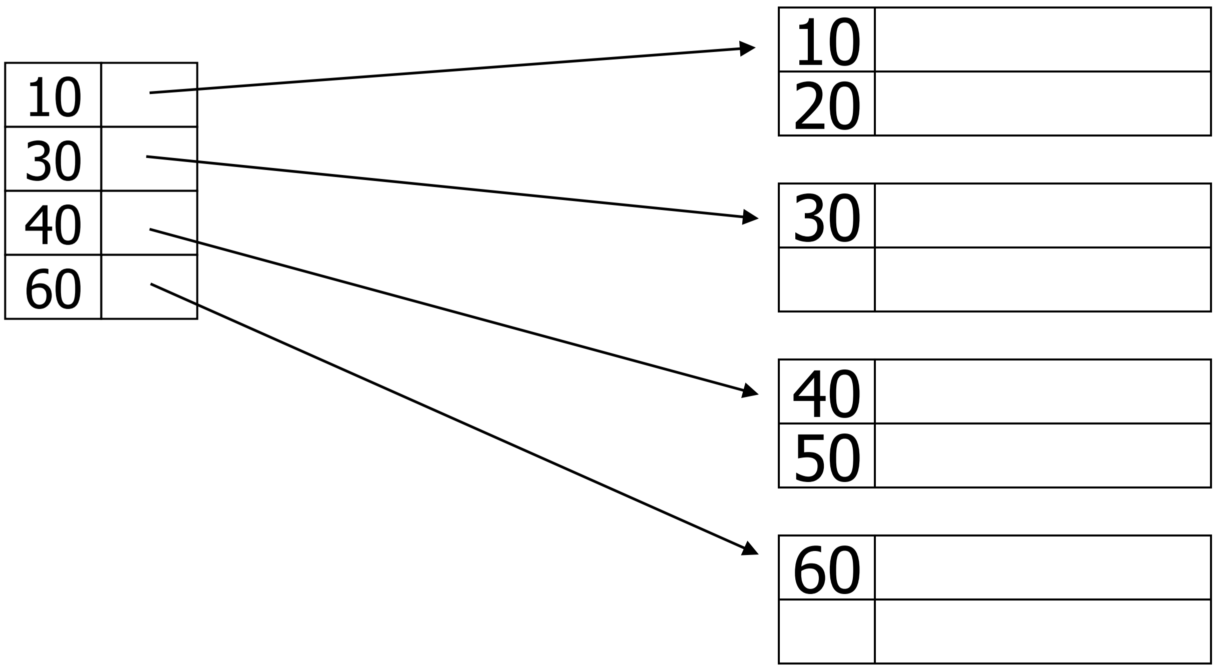


# Insertion, sparse index case



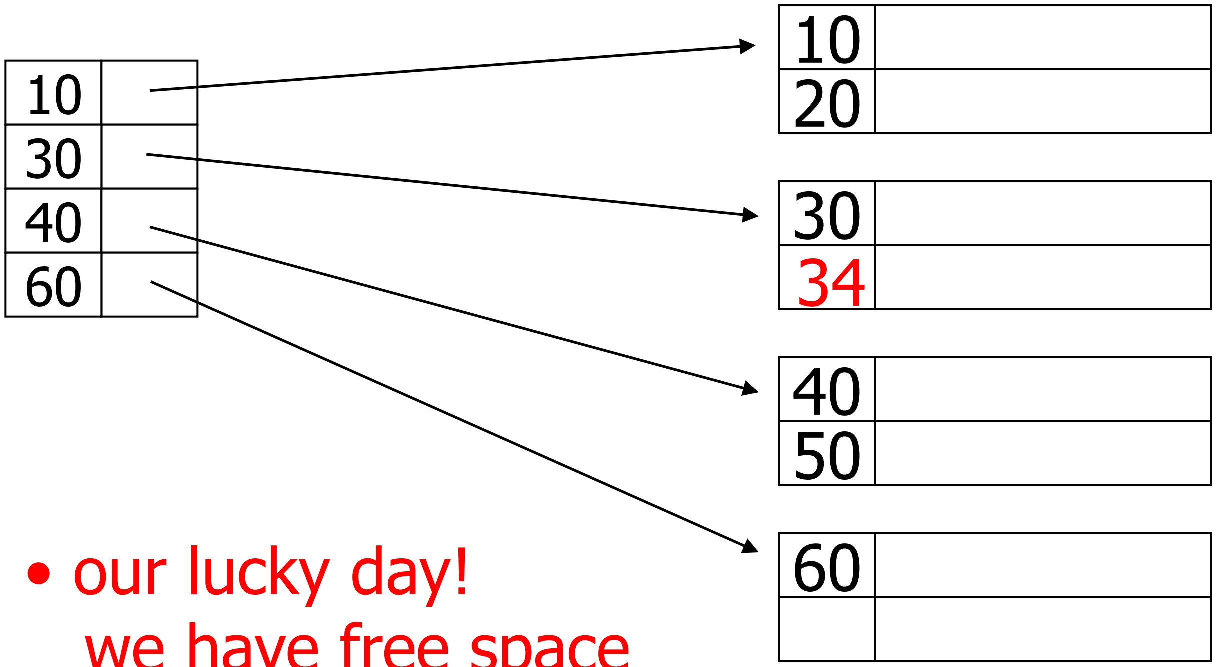
# Insertion, sparse index case

– insert record 34



# Insertion, sparse index case

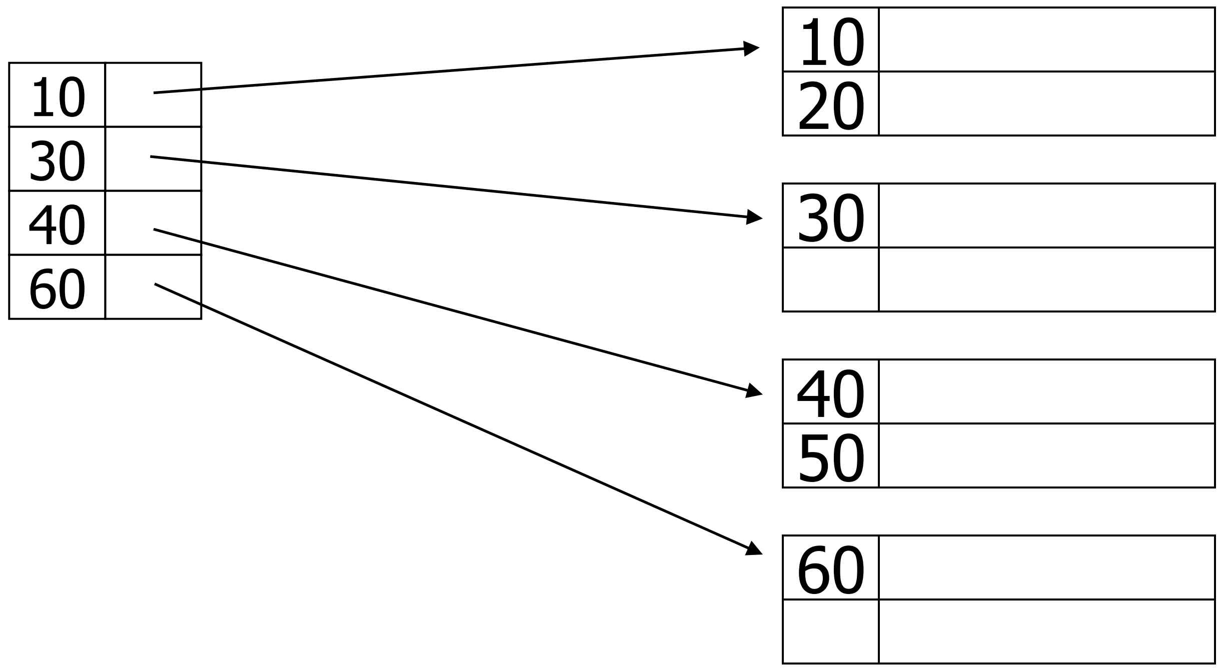
– insert record 34




- our lucky day!
  we have free space
  where we need it!

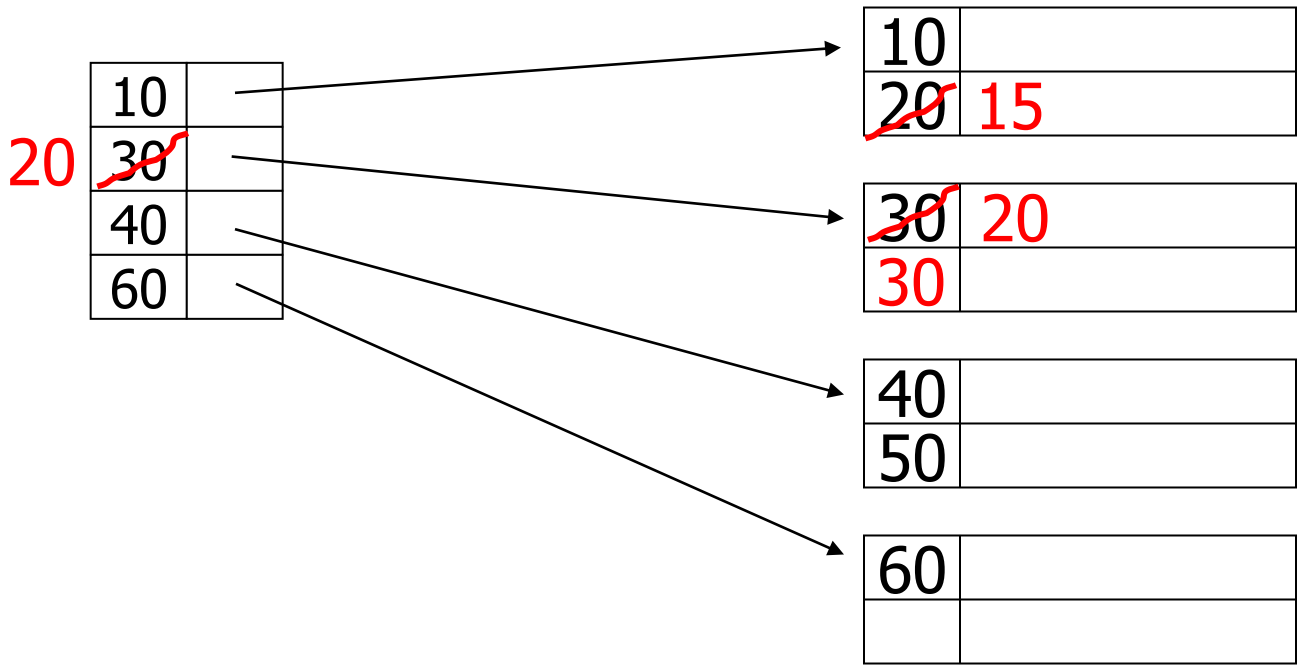
# Insertion, sparse index case

– insert record 15



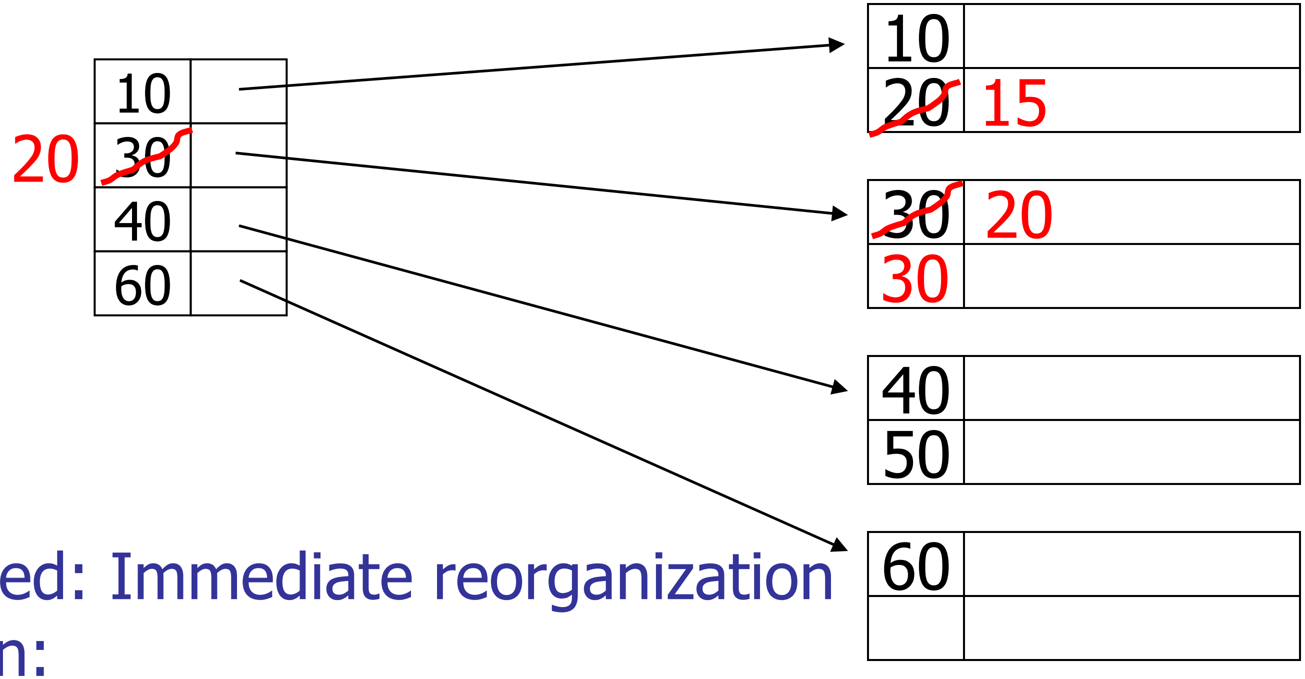
# Insertion, sparse index case

– insert record 15



# Insertion, sparse index case

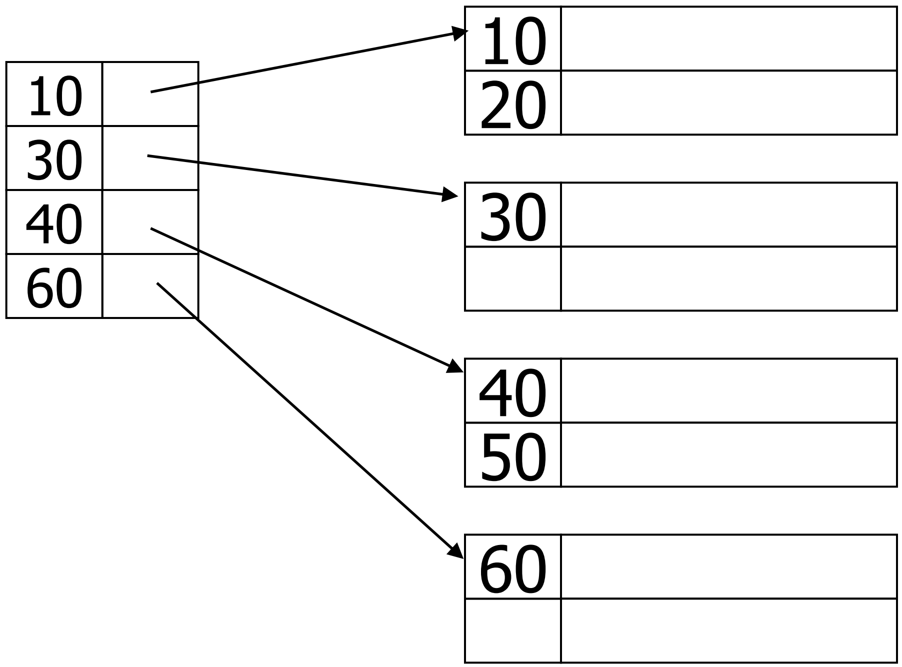
– insert record 15




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

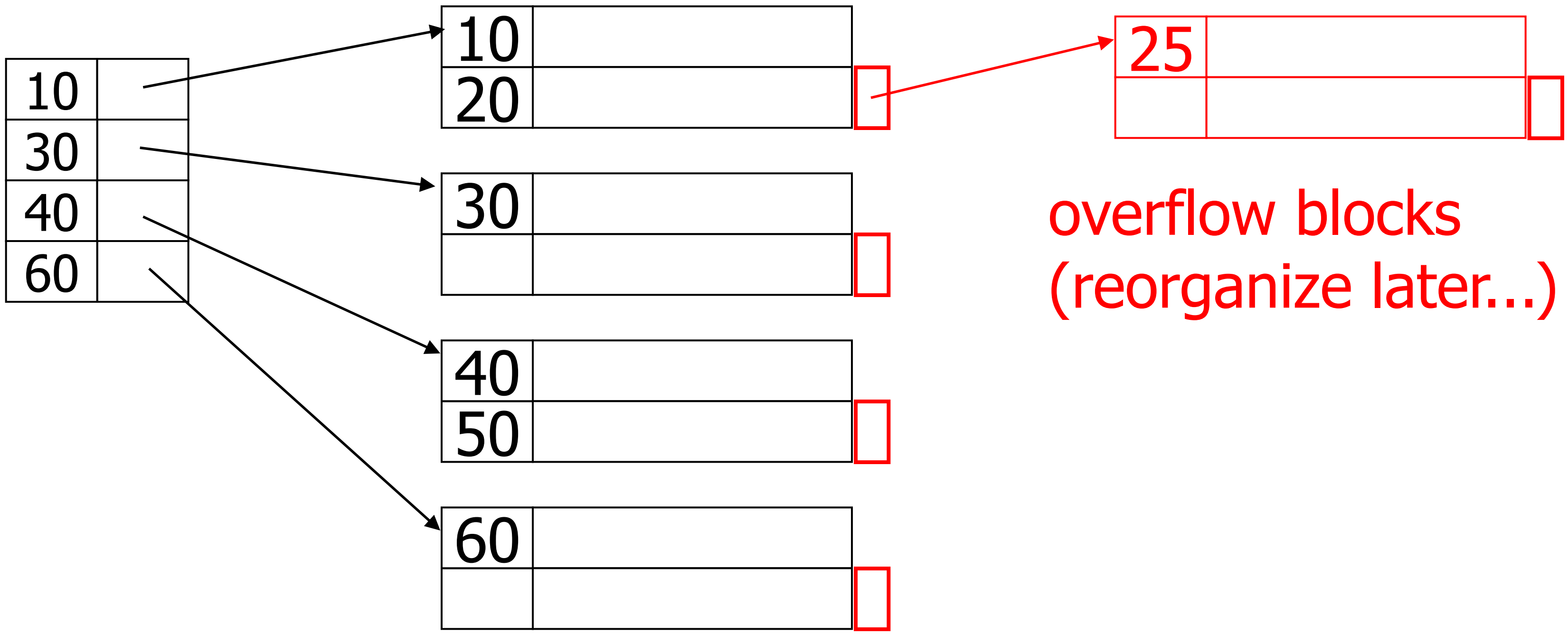
# Insertion, sparse index case

– insert record 25



# Insertion, sparse index case

– insert record 25




overflow blocks
(reorganize later...)

# Insertion, dense index case

- Similar
- Often more expensive . . .

# Secondary Indexes

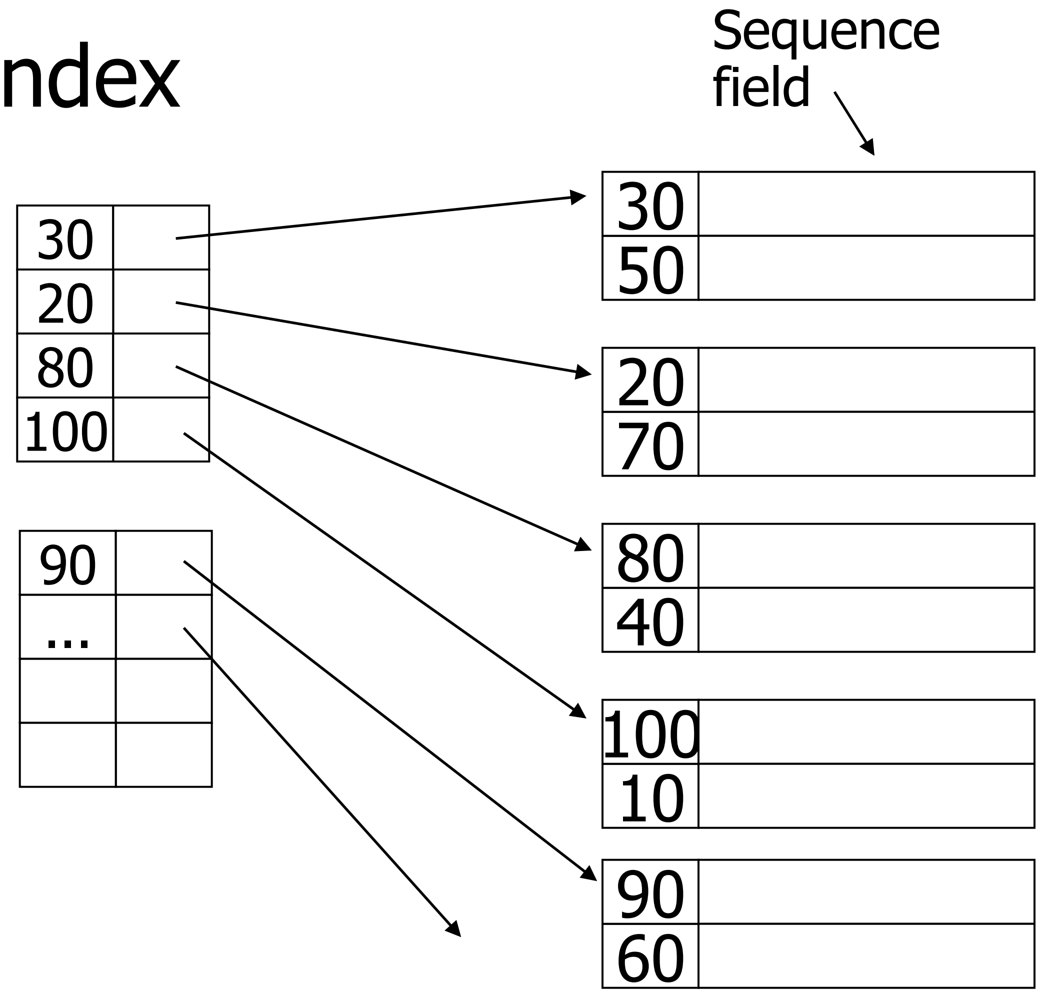
- Primary indexes
  - Underlying file was sorted on the search key.
- But we need a several indexes in the relation
- In a MovieStar relation we declare name as a primary key
- We may want to search on birth dates
  - We need a secondary index on birth date.

# Secondary Indexes

- A secondary index facilitates finding records given a value for one or more fields.
- The secondary index does not determine the placement of records in a data file.
- So, there is no sense to talk of sparse, secondary index
  - Since secondary index does not influence location, we could not use it to predict the location of any record whose key was not mentioned in the index file explicitely.
  - So secondary index is always dense.

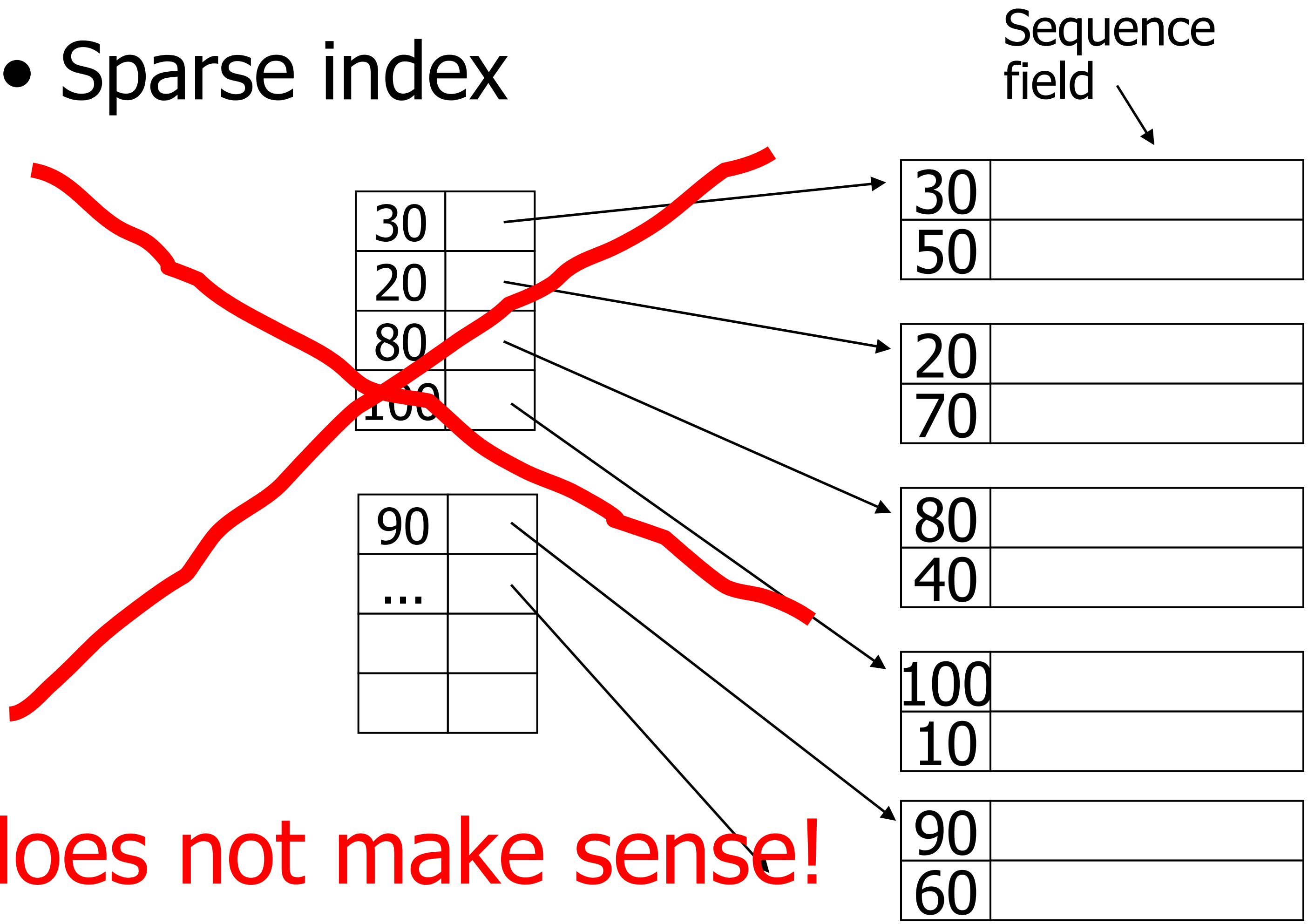
# Secondary indexes

- Sparse index



# Secondary indexes

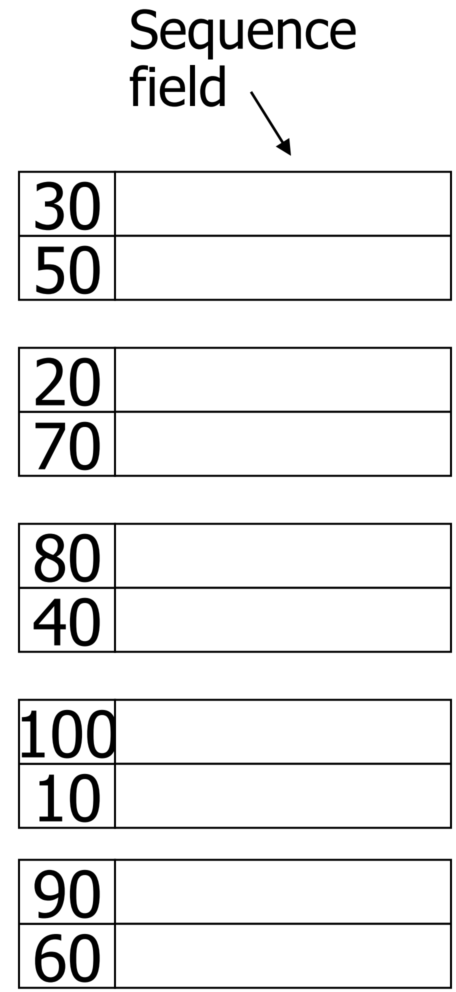
- Sparse index




does not make sense!

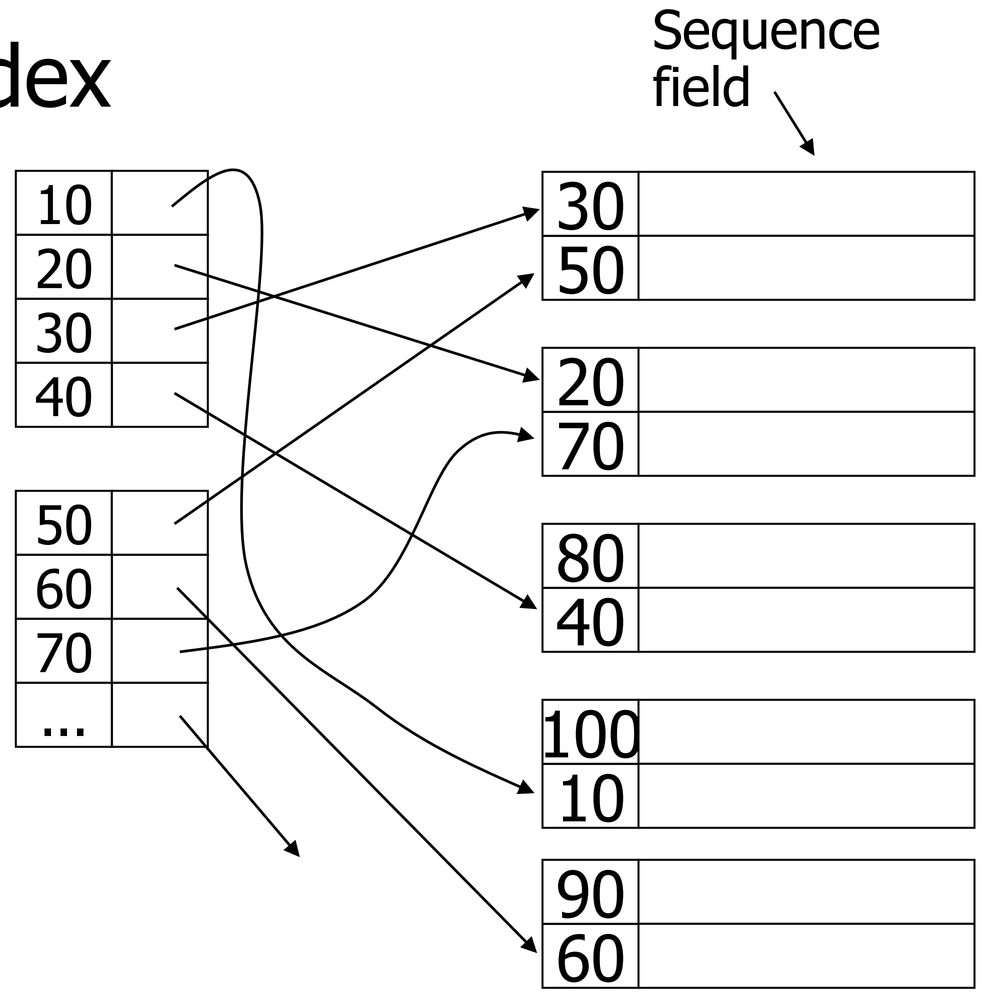
# Secondary indexes

- Dense index



# Secondary indexes

- Dense index

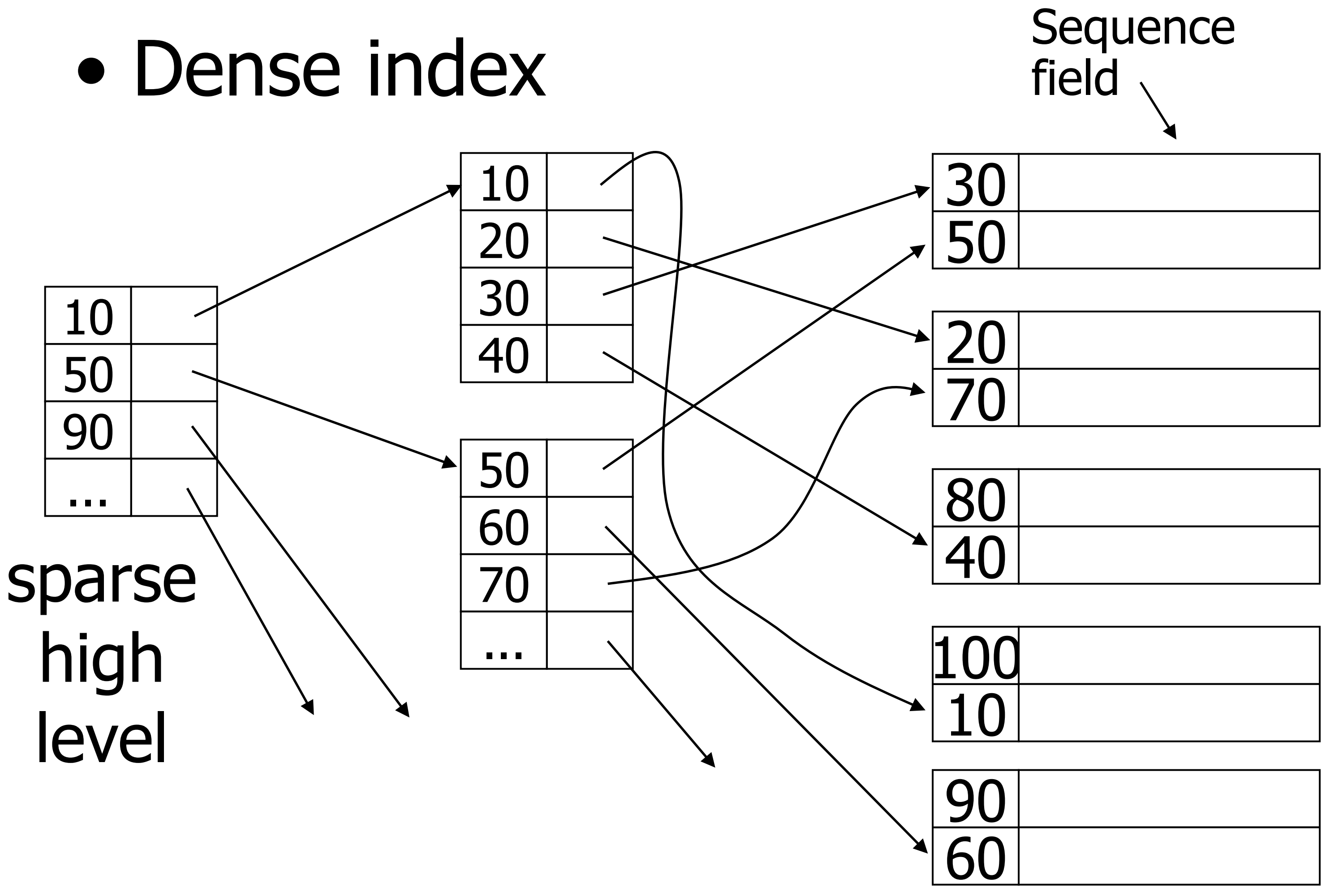


# Design of Secondary Indexes

- The keys in the index file are sorted.
- The points in one index block can go to many different data blocks.
  - Using secondary index may result in many more disk I/Os.
- It is possible to add a second level of index

# Secondary indexes

- Dense index

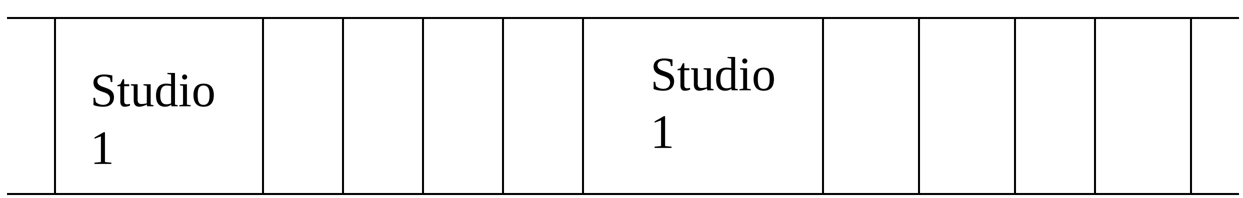


# Clustered File Structure

- Two or more relations are stored with their records intermixed.
- Consider the following two relations
  - Movie(title, year, length, studioName)
  - Studio(name, address, president)
- Consider the following query
  - SELECT title, year
  - FROM Movie
  - WHERE studioName='zzz';
- If the above is typical query we can order the tuples with studioName
- We can put a primary index on studioName

# Clustered File Structure

- Consider the following query
  - SELECT president
  - FROM Movie, Studio
  - WHERE title=‘Star Wars’ AND Movie.studioName=Studio.name;
- If we are sure that joins are common, we can make those joints efficient by chossing a clustered file structure.
  - Movie tuples are placed with Studio tuples in the same sequence of blocks.
    - We place for each studio tuple, all the movie tuples for the movies made by that studio.



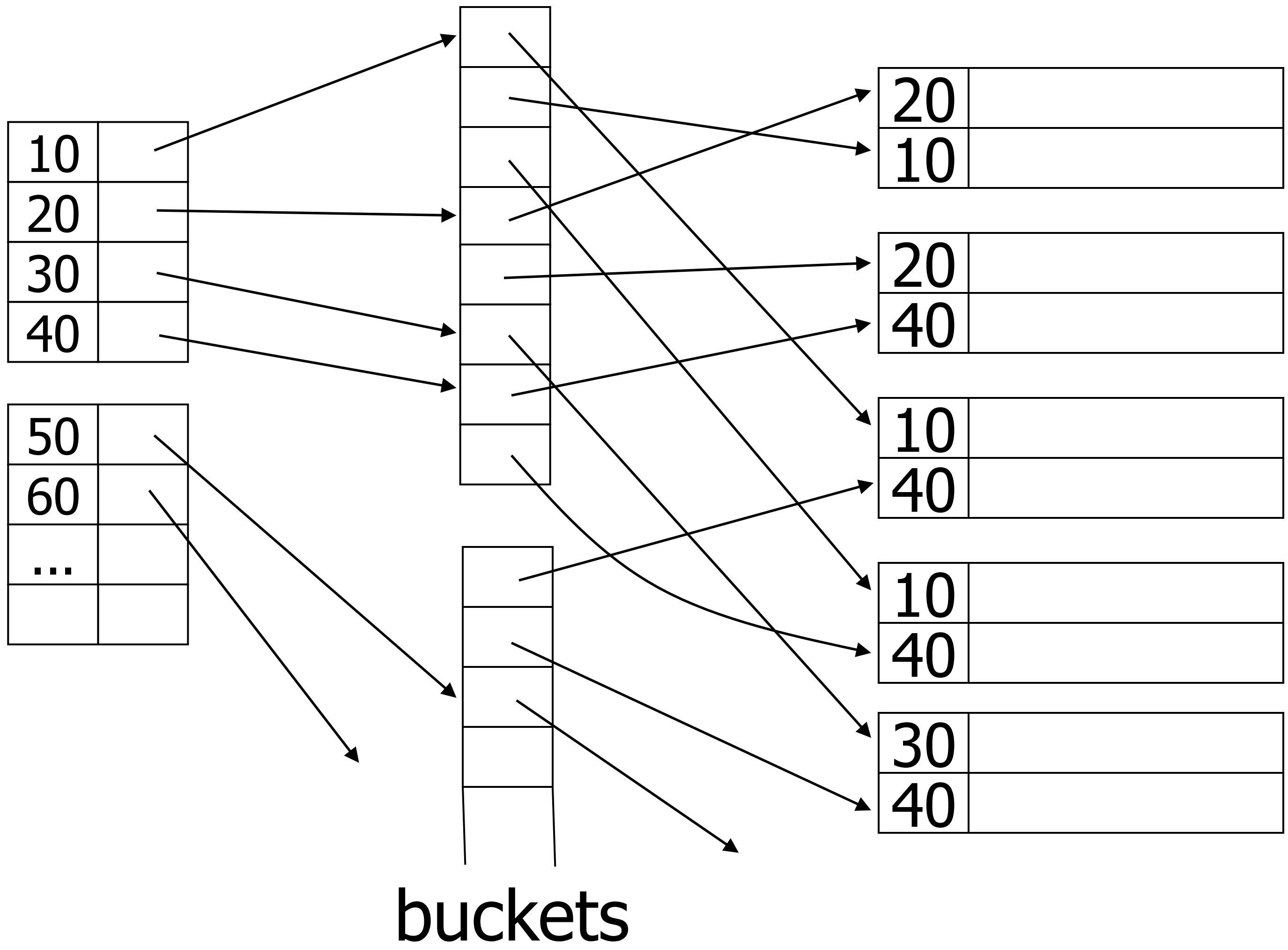

Movies by studio1

Movies by studio1

# Indirection in Secondary Indexes

- Significant amount of space is wasted
  - If the search key value appears n times in the data file, the value is written n times  in the index file.
- Solution
  - Use the indirection  called buckets between secondary index file and the data file

# Duplicate values & secondary indexes



# Advantages of bucket idea

- We can use the pointers in the buckets to answer the queries

# Why “bucket” idea is useful

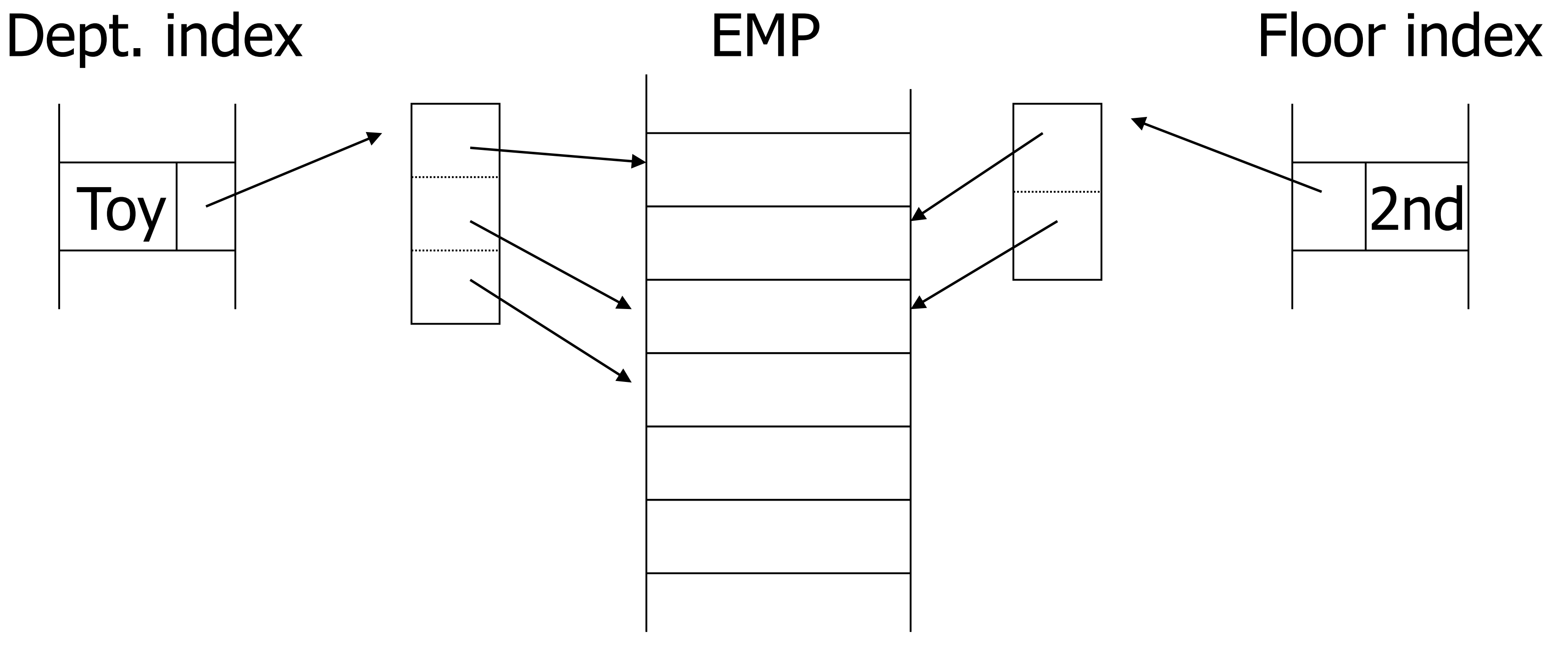
Indexes Records

Name: primary EMP (name,dept,floor,...)

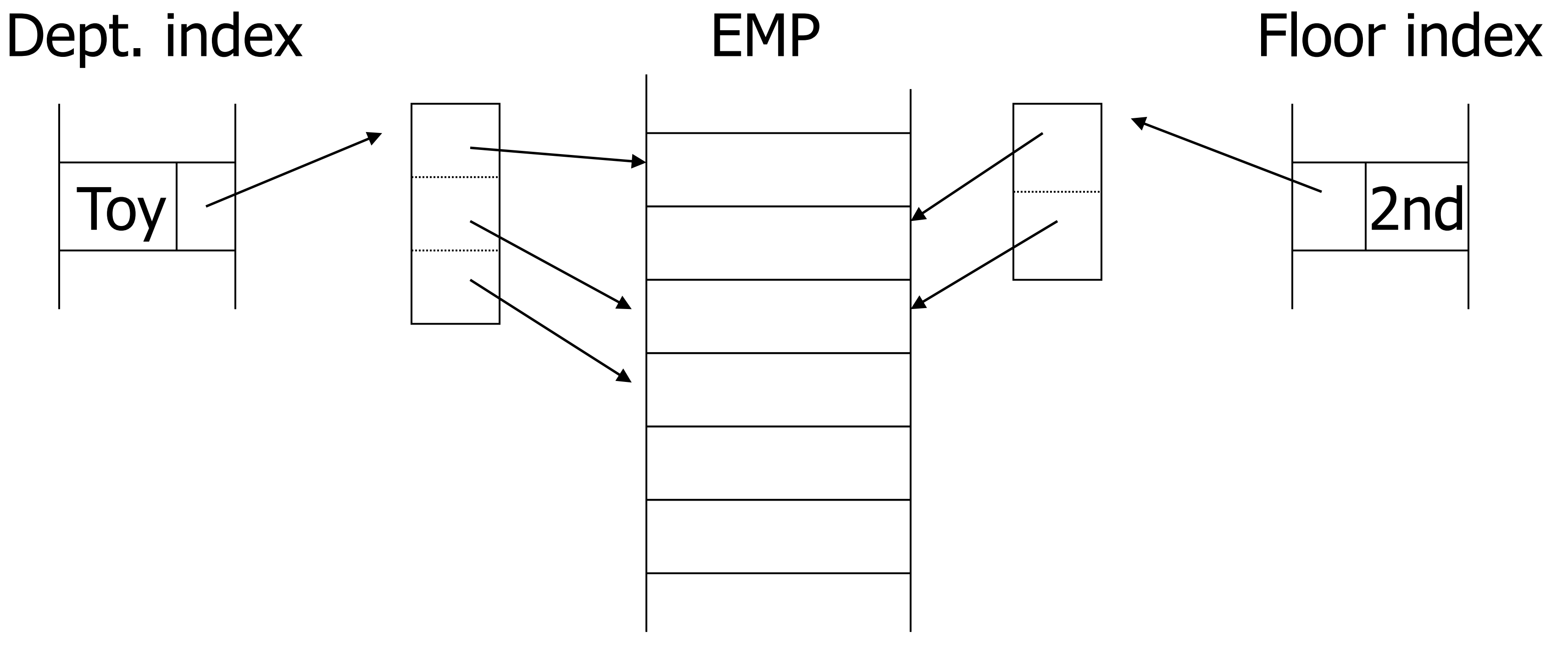
Dept: secondary

Floor: secondary

# Query: Get employees in
# (Toy Dept) $\wedge$ (2nd floor)



Query: Get employees in
(Toy Dept) $\wedge$ (2nd floor)




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

# IR QUERIES

- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “dog”

# IR QUERIES

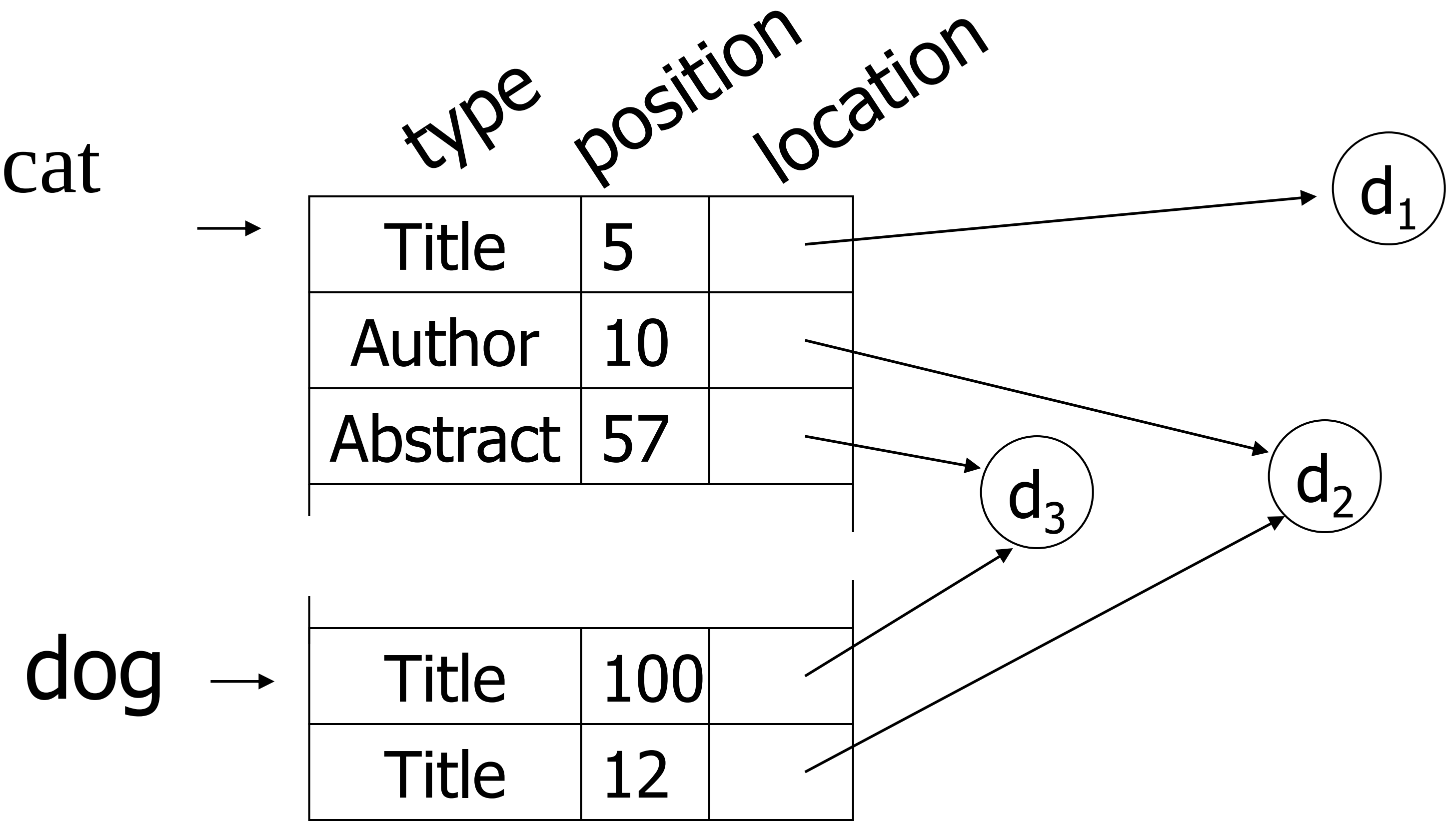
- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “dog”

- Find articles with “cat” in title
- Find articles with “cat” and “dog” within 5 words

# Inverted Index for documents

<u>Common technique</u>:

more info in inverted list



Posting: an entry in inverted list.
Represents occurrence of
term in article

Size of a list: 1 Rare words or
(in postings) miss-spellings

$10^6$ Common words

Size of a posting: 10-15 bits (compressed)

# IR DISCUSSION

- Stop words
- Truncation
- Thesaurus
- Full text vs. Abstracts
- Vector model

# Vector space model

| | w1 | w2 | w3 | w4 | w5 | w6 | w7 … |
|---|---|---|---|---|---|---|---|
| DOC = | <1 | 0 | 0 | 1 | 1 | 0 | 0 …> |
| Query= | <0 | 0 | 1 | 1 | 0 | 0 | 0 …> |

# Vector space model

| | w1 | w2 | w3 | w4 | w5 | w6 | w7 | … |
|---|---|---|---|---|---|---|---|---|
| DOC = | <1 | 0 | 0 | 1 | 1 | 0 | 0 | …> |
| Query= | <0 | 0 | 1 | 1 | 0 | 0 | 0 | …> |

PRODUCT = 1 + ……. = score

- Tricks to weigh scores + normalize

e.g.: Match on common word not as
useful as match on rare words...

- How to process V.S. Queries?

| | w1 | w2 | w3 | w4 | w5 | w6 | … | |
|---|---|---|---|---|---|---|---|---|
| Q = < | 0 | 0 | 0 | 1 | 1 | 0 | … | > |

# Conventional indexes

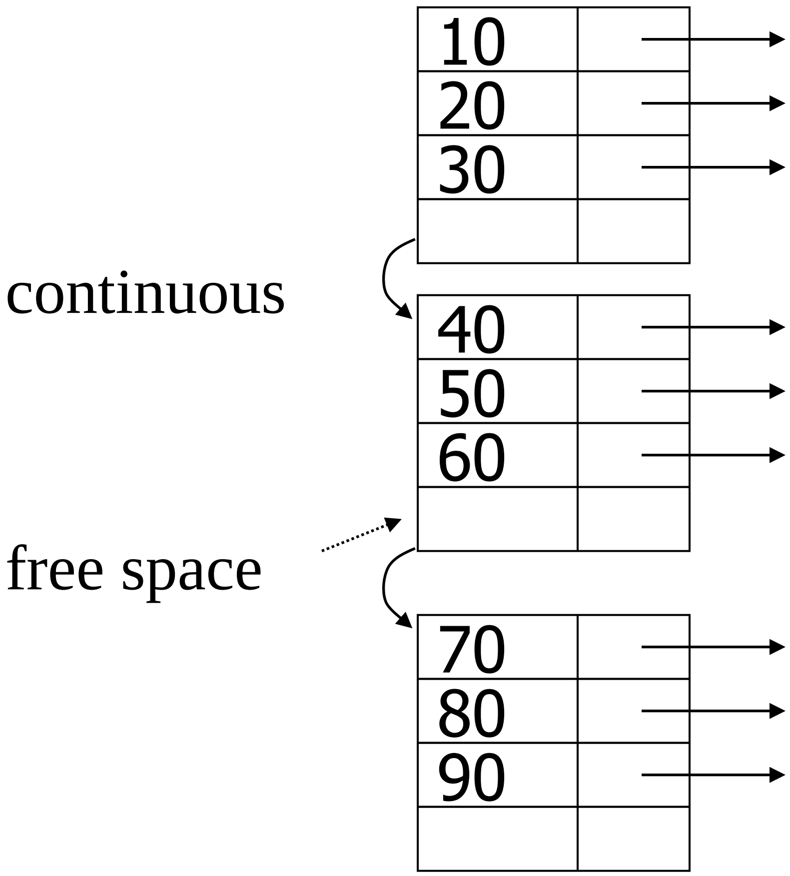
Advantage:

- Simple
- Index is sequential file
  good for scans

Disadvantage:

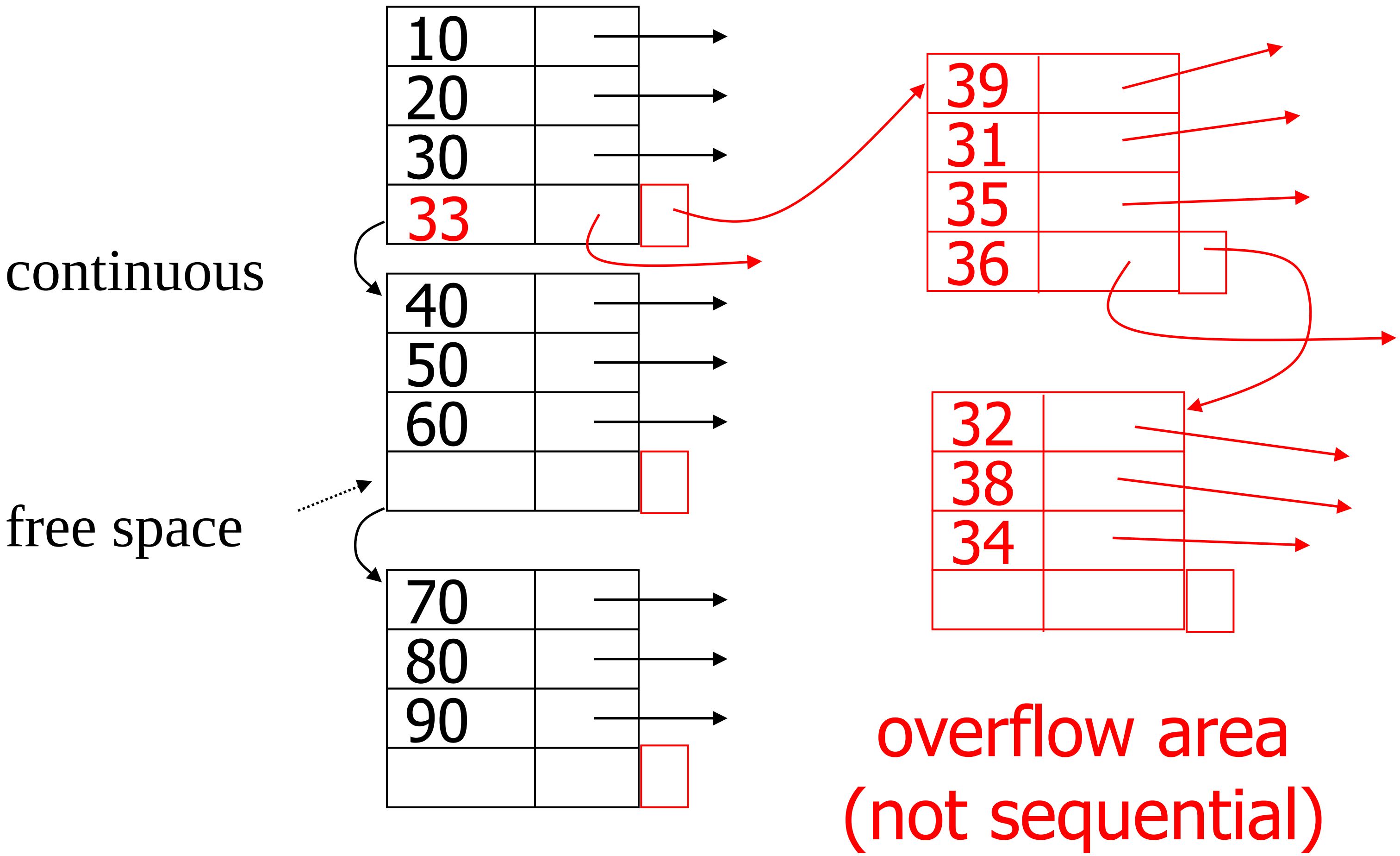
- Inserts expensive, and/or
- Lose sequentiality & balance

# Example Index (sequential)



# Example

Index (sequential)



# B-trees

- It is a general multi-level structure
- B-tree  is a family of data structures
  - B+-tree is often used
- B-trees automatically maintain as many levels of index as in appropriate for a file.
- Every block is used between half and completely full.
  - No overflow blocks are needed

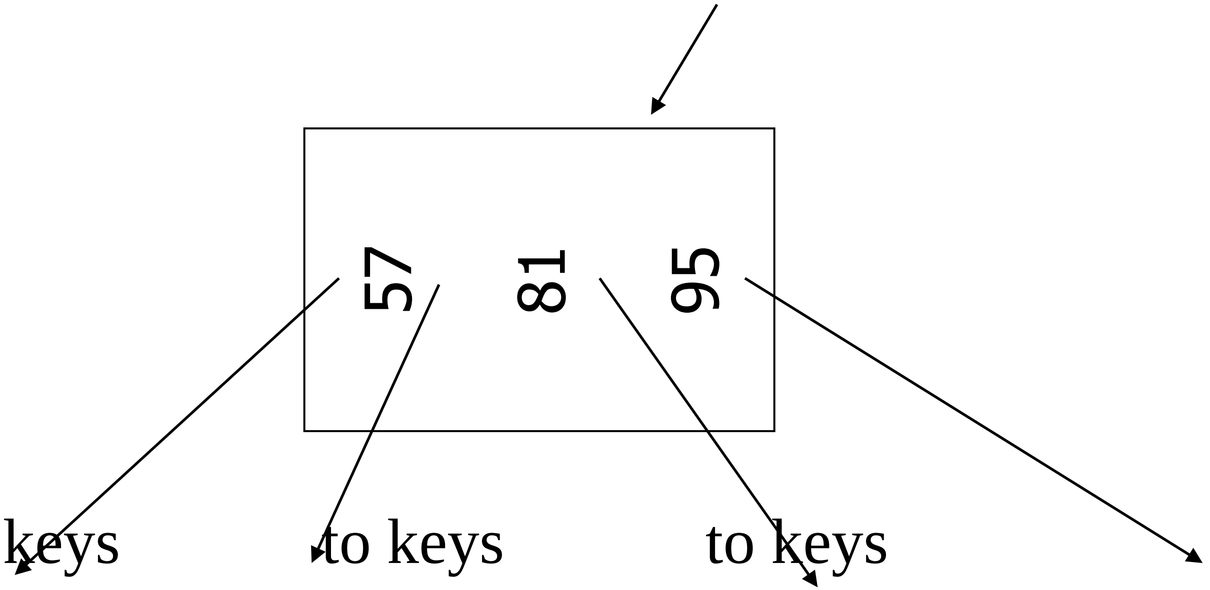
# Structure of Btree

- The tree is balanced
  - All paths from root to a leaf have the same length
- Three layers
  - The root
  - Intermediate layer
  - and leaves
- The parameter “n” is associated with each B-tree index
  - Each block has a layout for “n” keys and “n+1” pointers

# Rules of Btree

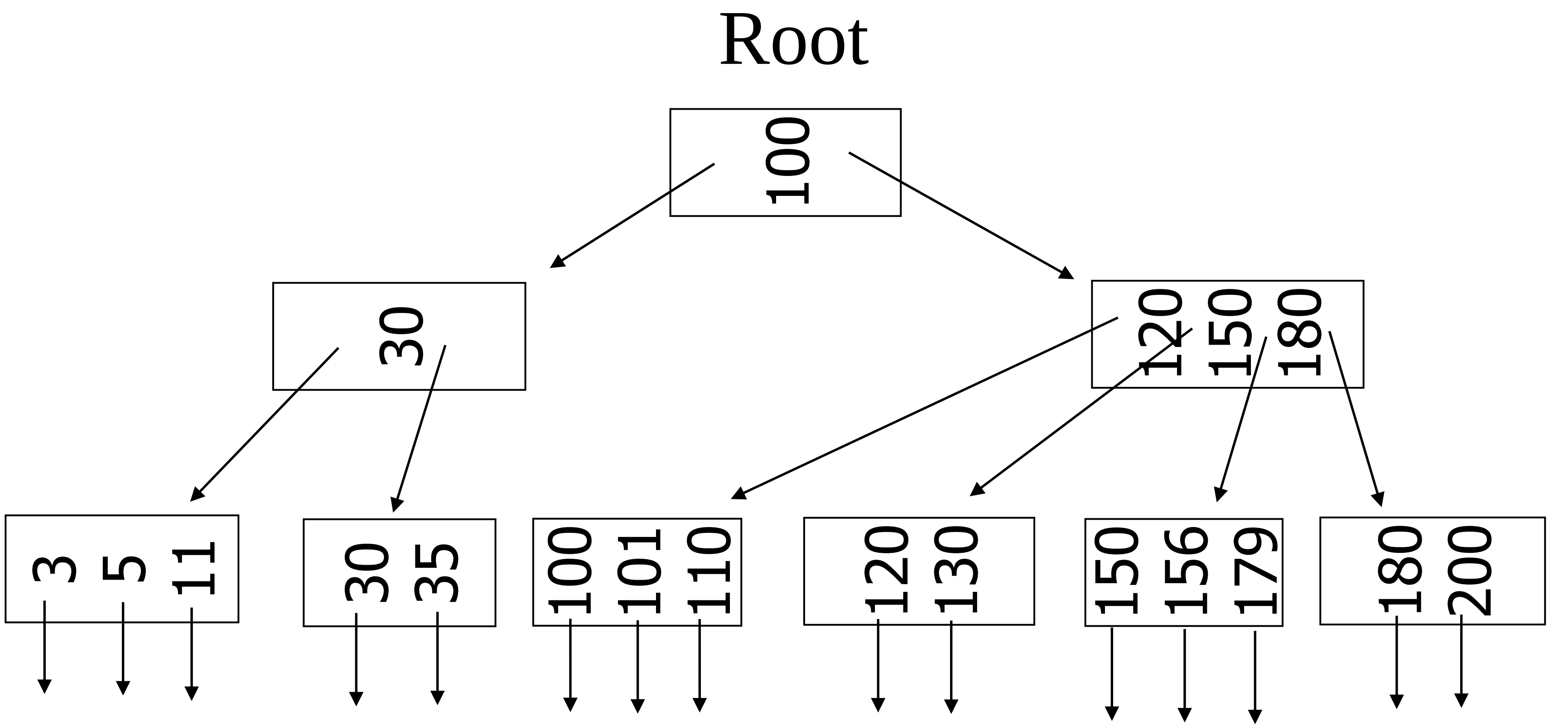
- At the root there are at least two pointers
  - All pointers point to B-tree blocks at the level below
- At the leaf, one last pointer points to the next leaf block to the right (to the block with next higher keys)
  - At least $\lfloor n+1/2 \rfloor$ pointers are used and point to data records.
  - Unused pointers may be thought of null and do not point any where.
- At an interior node, all n+1 pointers can be used to point to B-tree blocks at the next lower level.
  - At least $\lceil n+1/2 \rceil$ of them are actually used (for the root only two pointers are used)
  - If j pointers are used then there will be j-1 keys.
    - The first pointer points to part of Btree where some of the records less than k1 will be found.
    - The second pointer goes to that part of the tree where all records with keys that are at least K1, but less than K2 will be found.
    - The jth pointer points to the part of BTree where some of the records greater than Kj-1 are found.

# Sample non-leaf

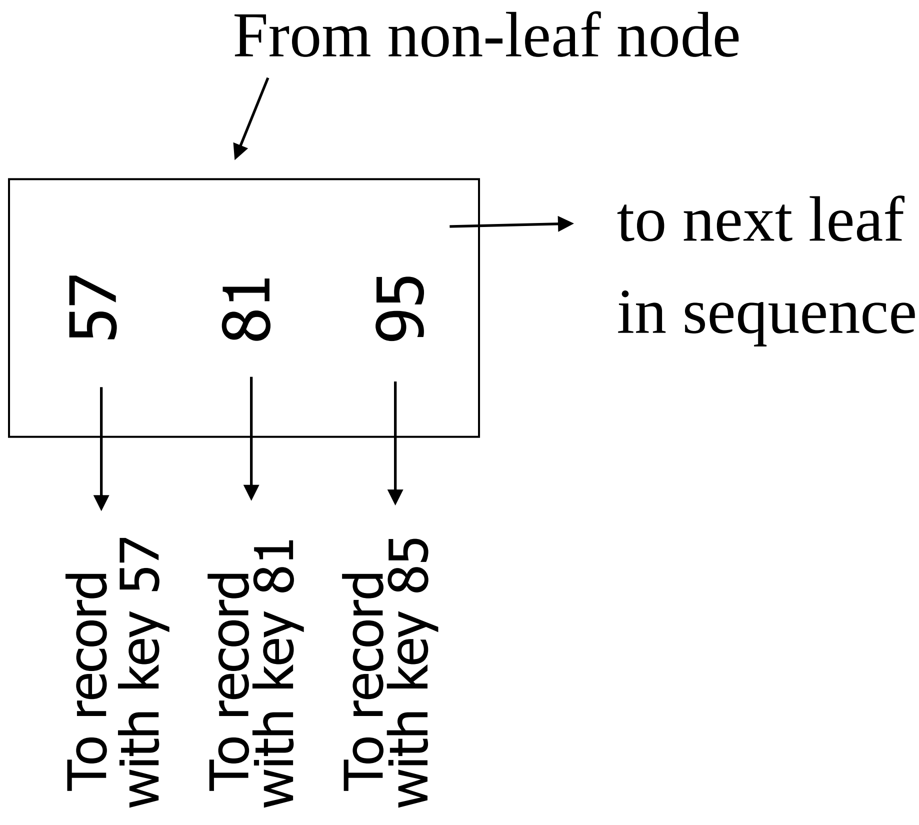



| | | | |
|---|---|---|---|
| to keys | to keys | to keys | to keys |
| $< 57$ | $57 \leq k < 81$ | $81 \leq k < 95$ | $\geq 95$ |

# B+Tree Example n=3

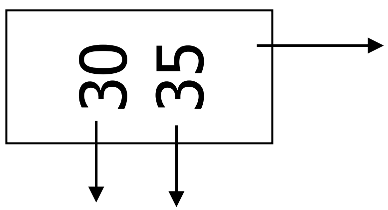


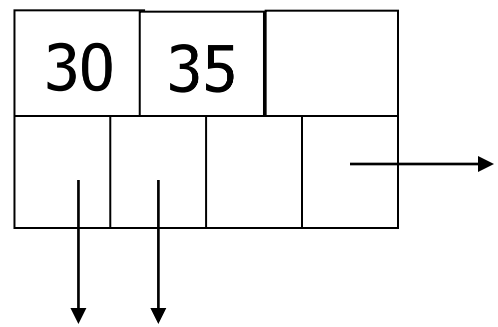
# Sample leaf node:



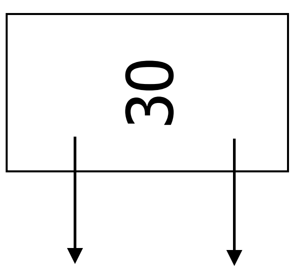
# In textbook's notation n=3

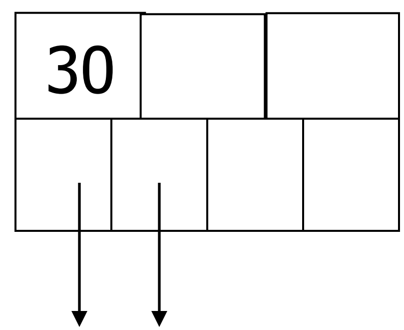
Leaf:




Non-leaf:



Size of nodes: n+1 pointers

n keys

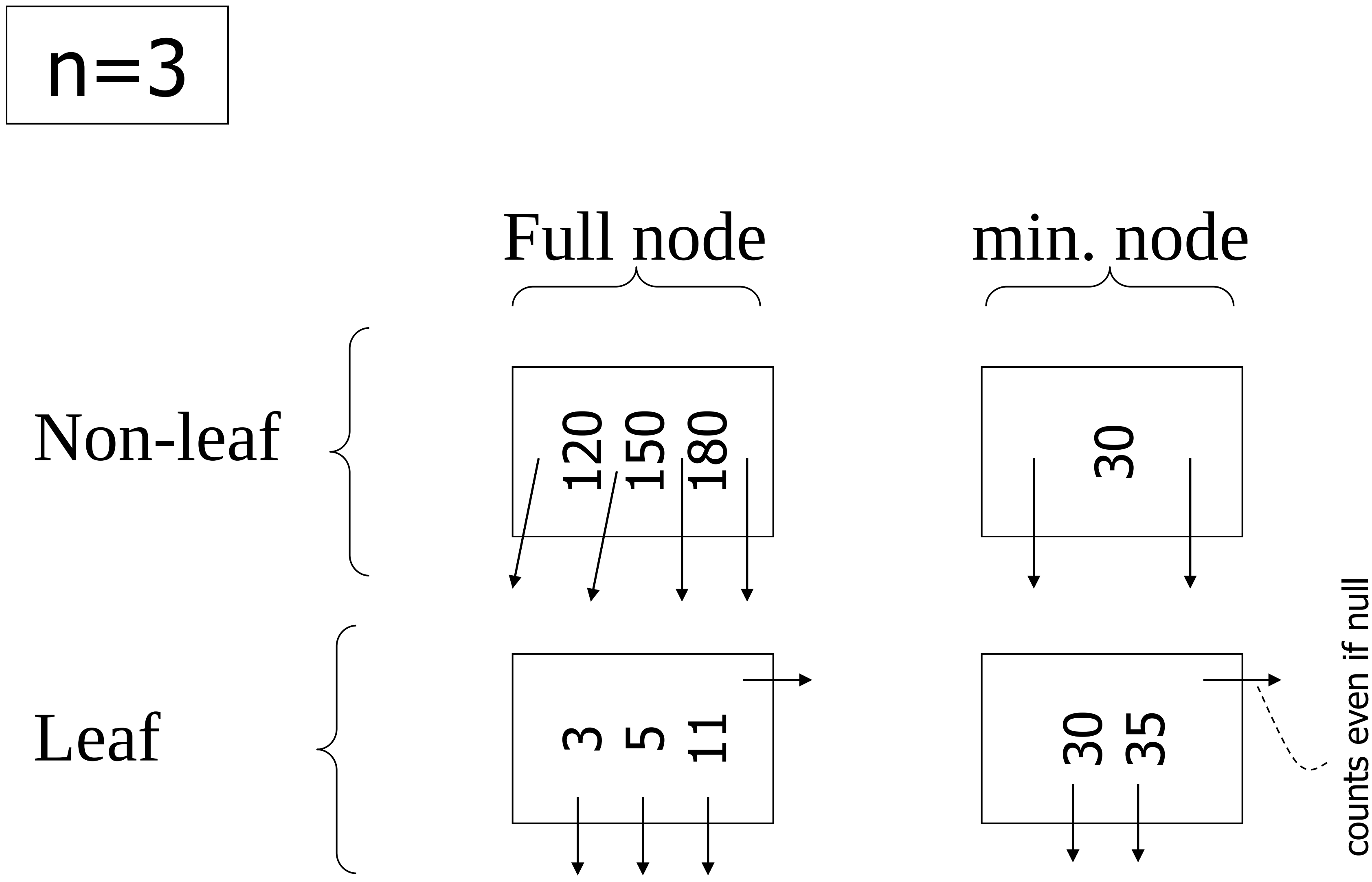
(fixed)

# 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
120
150
180
30
3
5
11
30
35
counts even if null

# B+tree rules tree of order *n*

(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 |

# Applications of B-trees

- B-tree is a powerful tool for building indexes
- The search key is the primary key  for the data file and index is dense, there is a leaf pointer for every entry. The data file may not be sorted by primary key.
- If the data file is sorted by primary key, the B+tree is a sparse index with key-pointer pair at the leaf for each block of the data file.
- There is a variant of Btree for multiple occurrences of search key.
  - Refer text book.

# Lookup in Btree

- Find a record with search-key value K
- If we are at the leaf, look among the keys. If the ith leaf is K, ith pointer will take us to the desired record.
- Follow the rules of Btree and reach the leaf node.

# Range queries

- SELECT *
- FROM R
- WHERE R.k >40
- Or
- SELECT *
- FROM R
- WHERE R.k >= 10 and R.k <= 25;

# Insertion into Btree

- We try to find a place
- If there is no room in the leaf, we split the leaf into two and divide the keys into two nodes, so each is half full or just over half full.
- The splitting of nodes lead to inserting a new key-pointer pair  at a higher level.
  - Apply this strategy recursively at higher levels.
- We try to insert at the root. If there is no room we try to split the root and create a new root.

# Insert

- Suppose N is a leaf node with n keys
  - We trying to insert “n+1” key and a pointer
  - Create a new node M, which will be a sibling of N, immediately to its right
  - The first (n+1)/2 (high) key-pointer pairs in the sorted order will be with N. The other key pointer pairs move to M
  - Both will have at least (n+1)/2 (low) key pointer pairs.
- Suppose N is an interior node with n keys and n+1 pointers.
  - Create a new node M which will be a sibling of N, immediately to its right.
  - The first (n+2)/2 (high) pointers pairs in the sorted order will be with N and move M other (n+2)/2(low) pointers.
  - The n/2 (high) keys stay with N, and last n/2 (low) keys move to M.
  - There is a one key left over, it is the smallest key reachable via M children.
    - K will be used as a parent of N and M to divide the searches between those two nodes.

# Insert into B+tree

(a) simple case
  – space available in leaf

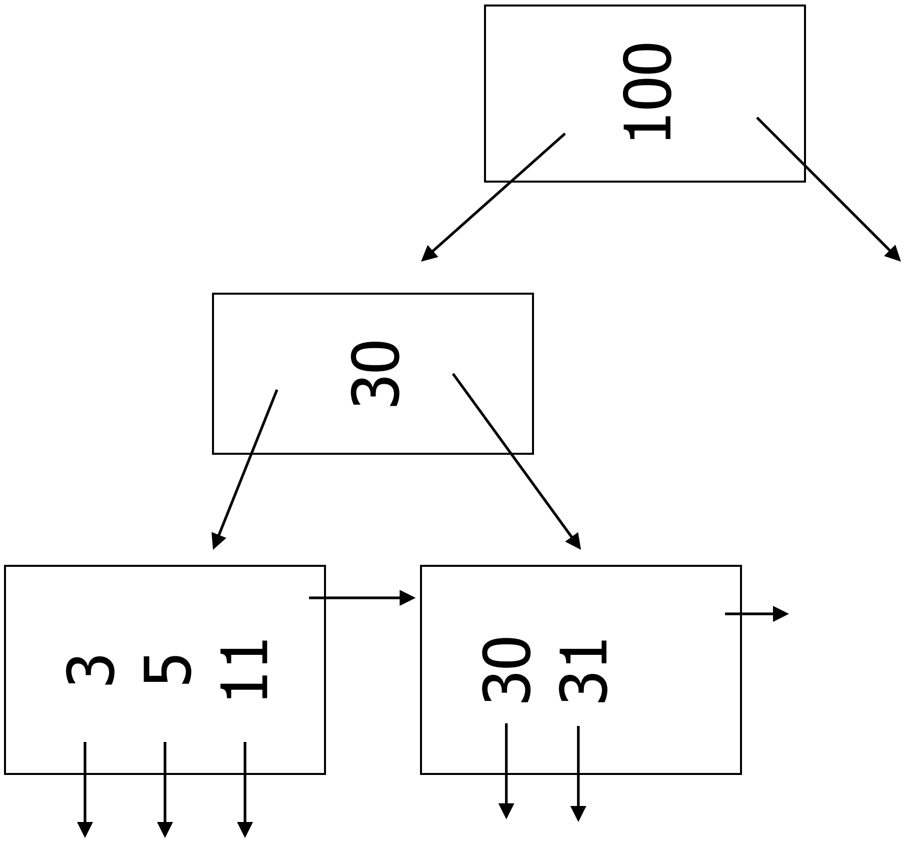
(b) leaf overflow

(c) non-leaf overflow

(d) new root

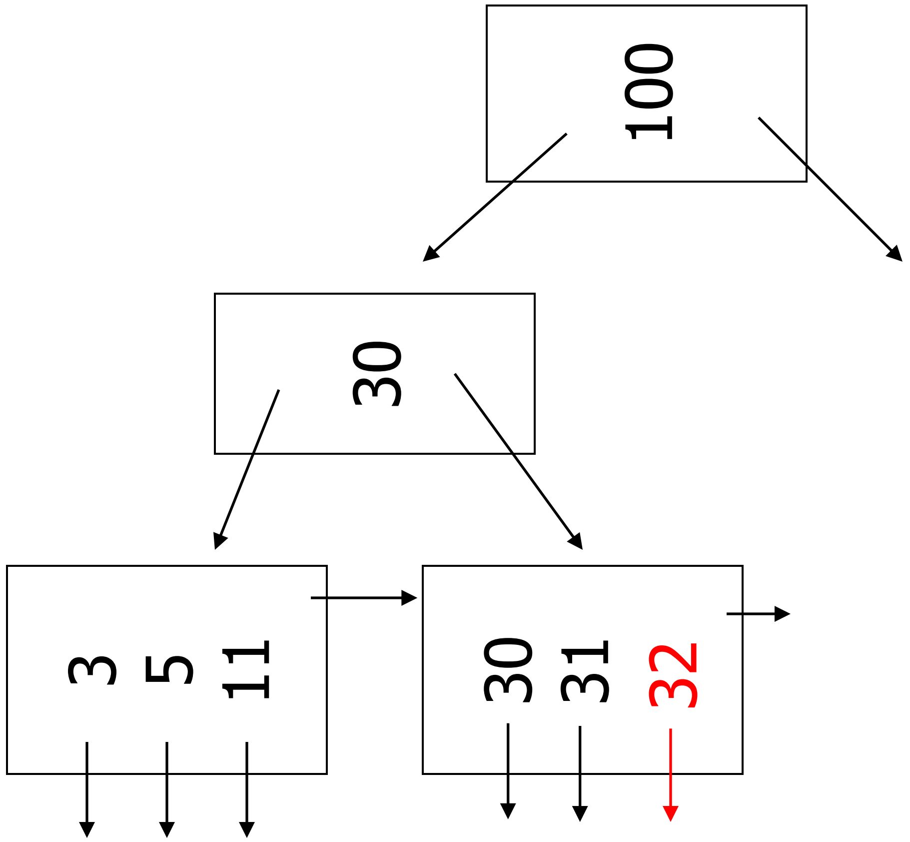
# (a) Insert key = 32

n=3



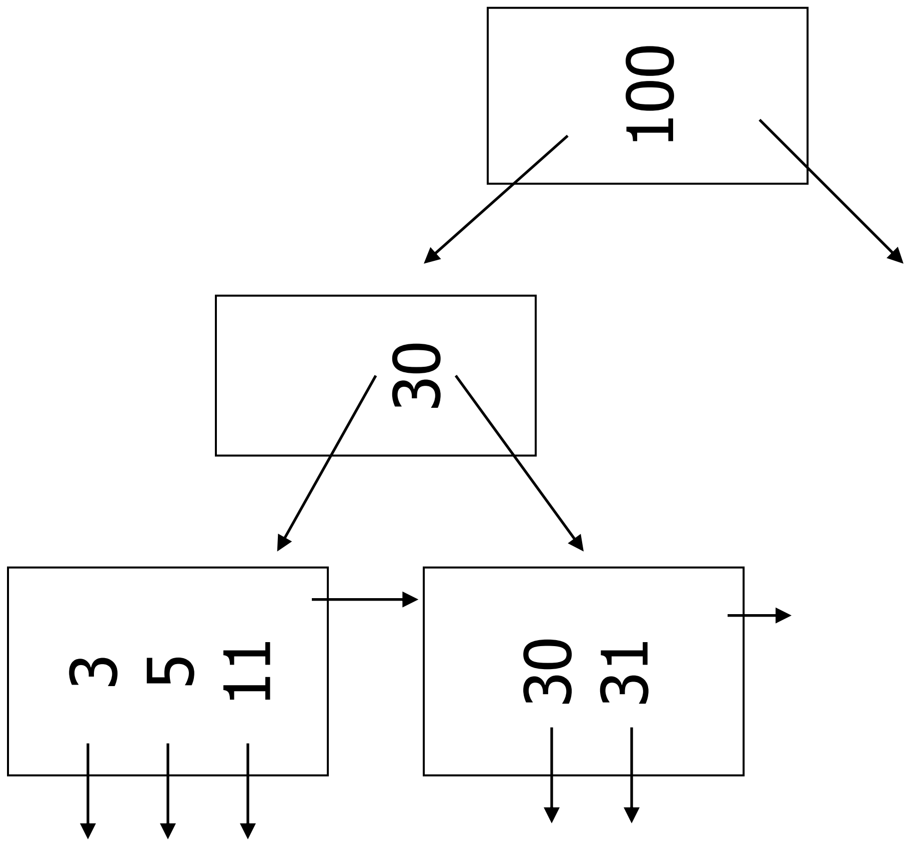
# (a) Insert key = 32

n=3



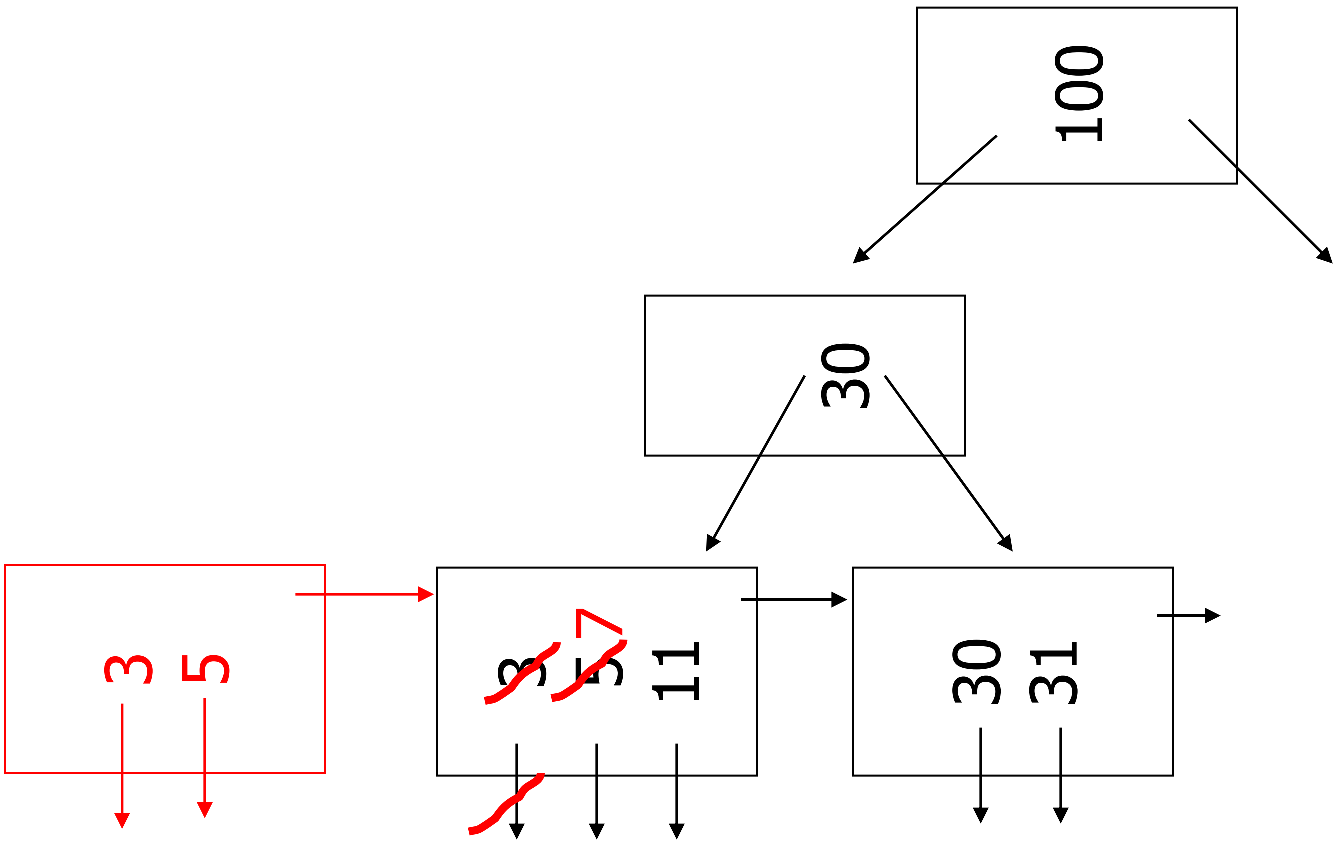
# (a) Insert key = 7

n=3



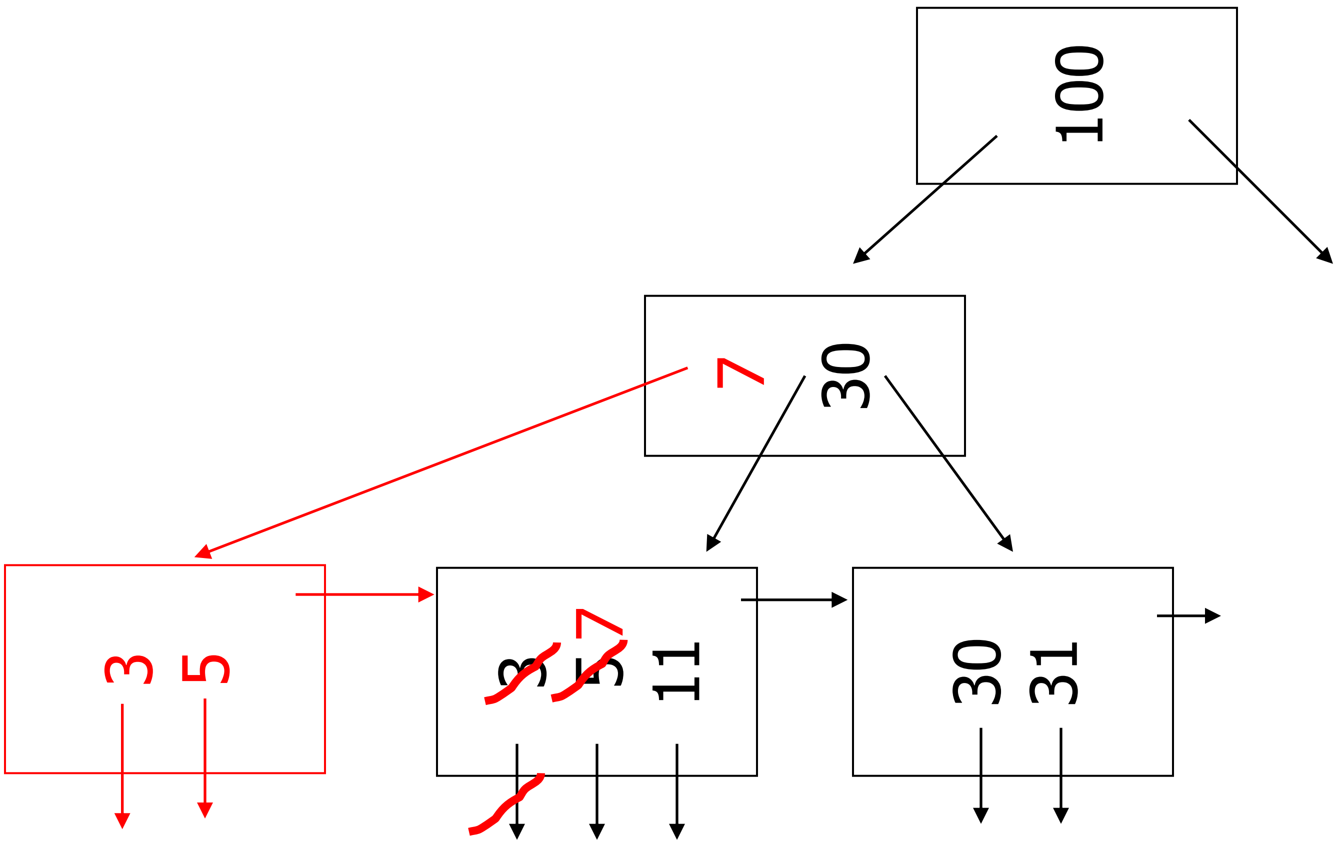
# (a) Insert key = 7

n=3



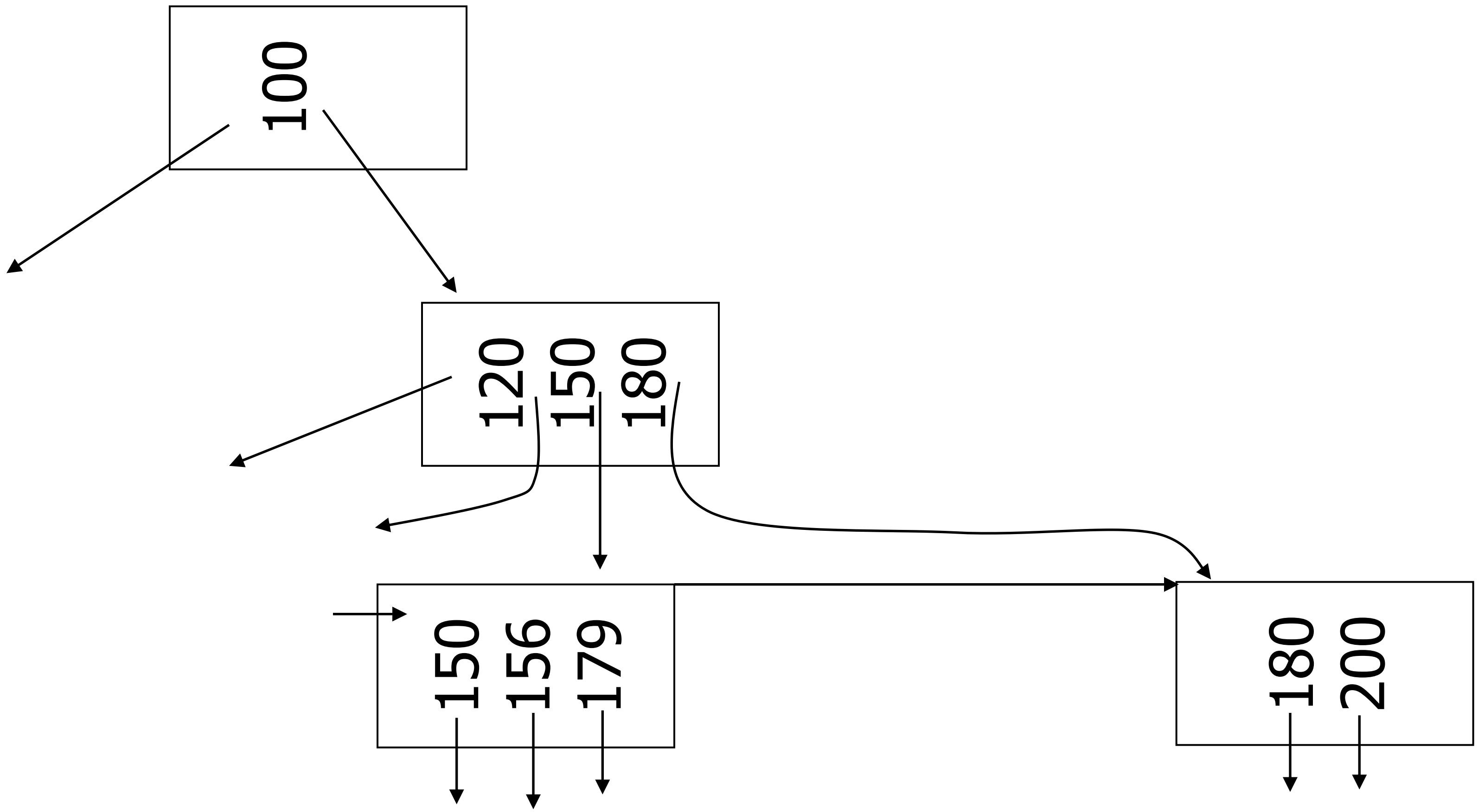
# (a) Insert key = 7

n=3



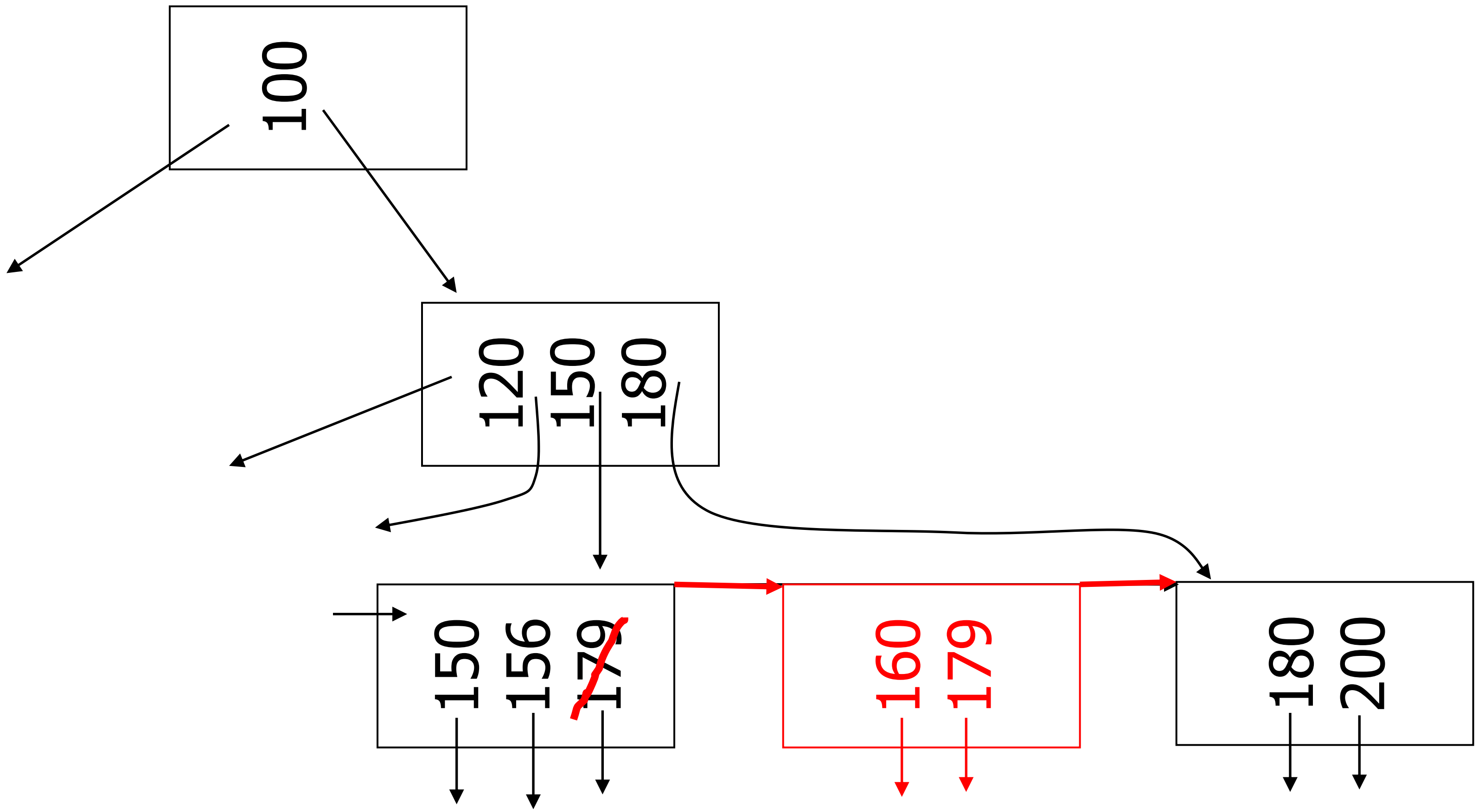
# (c) Insert key = 160

n=3



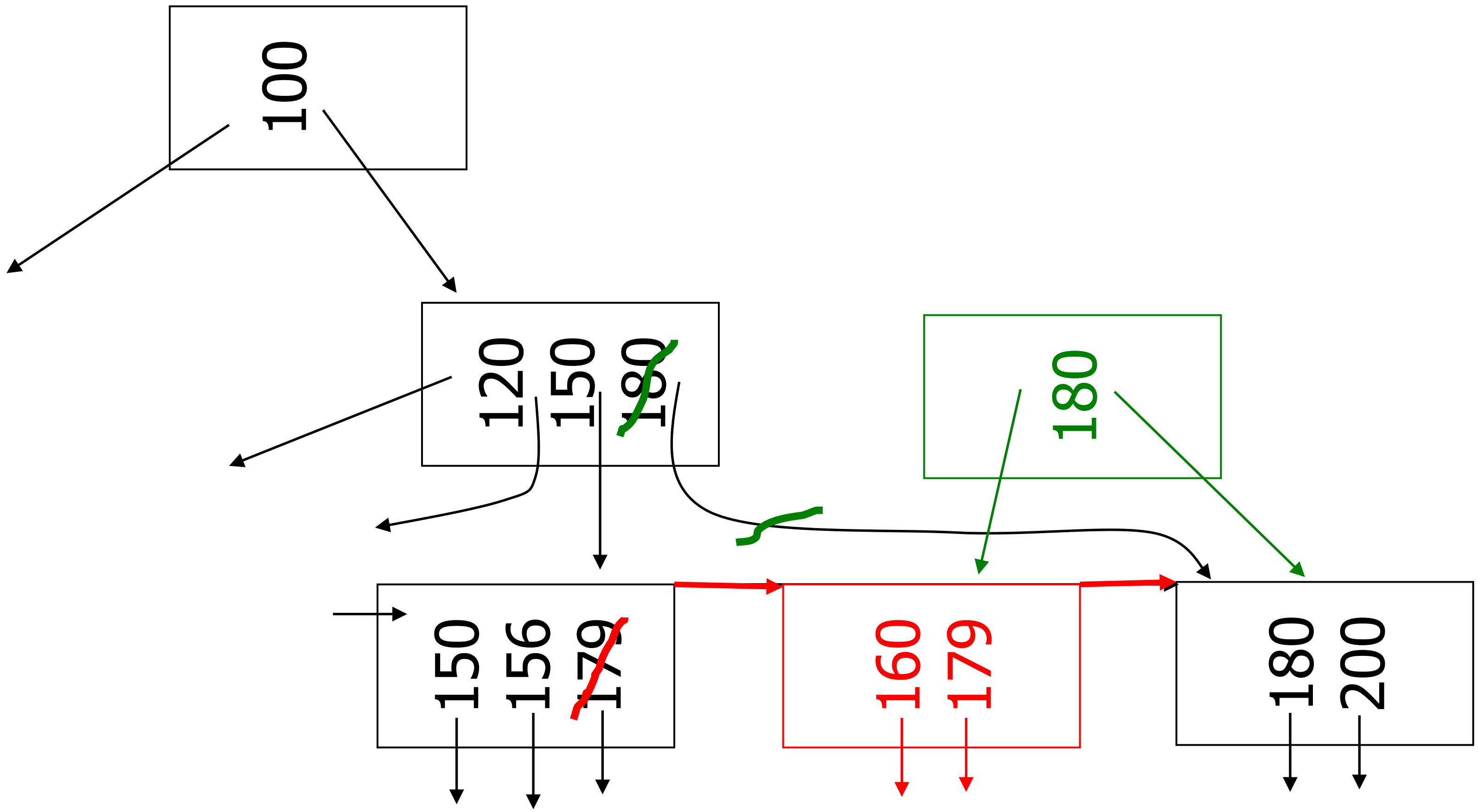
# (c) Insert key = 160

n=3



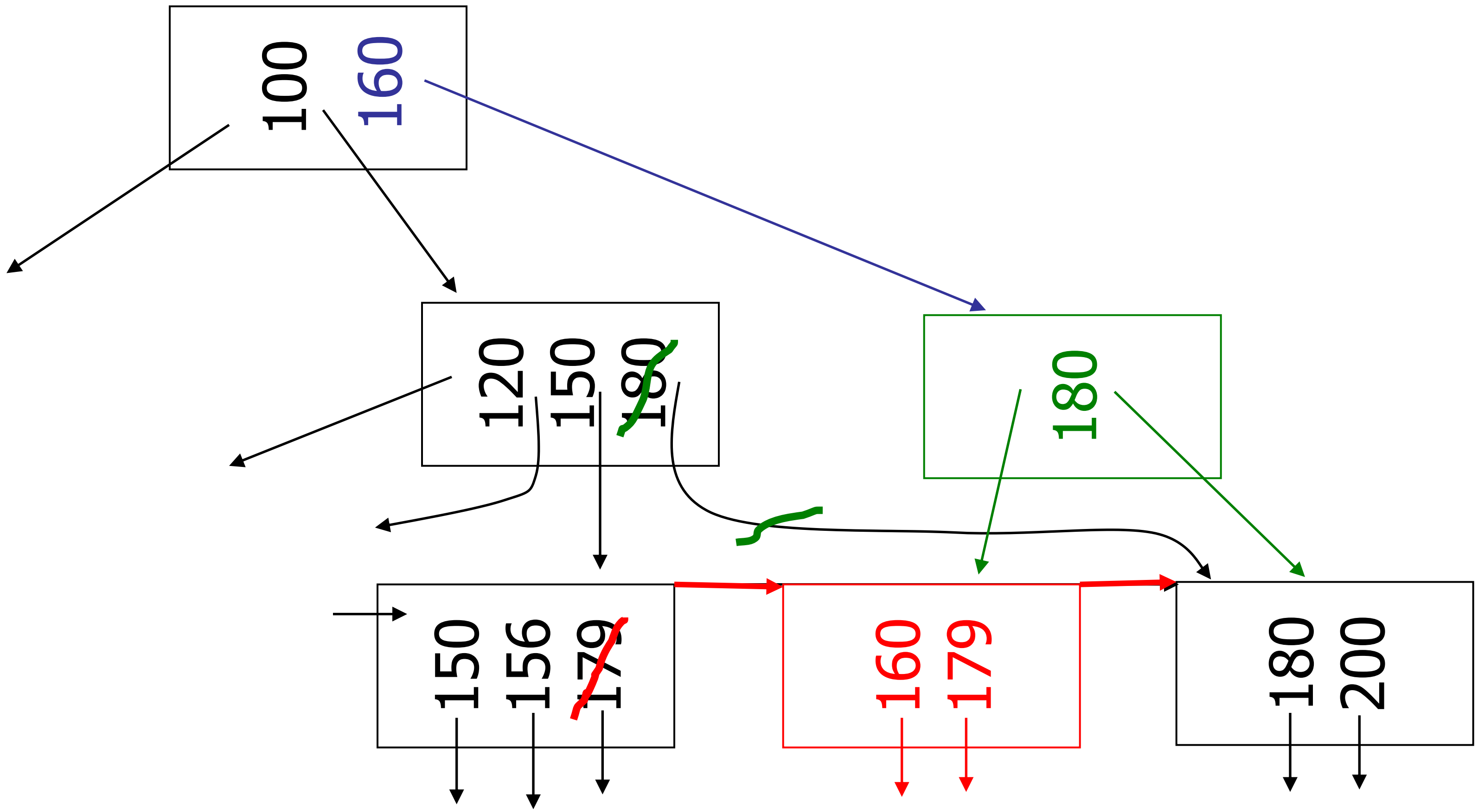
# (c) Insert key = 160

n=3



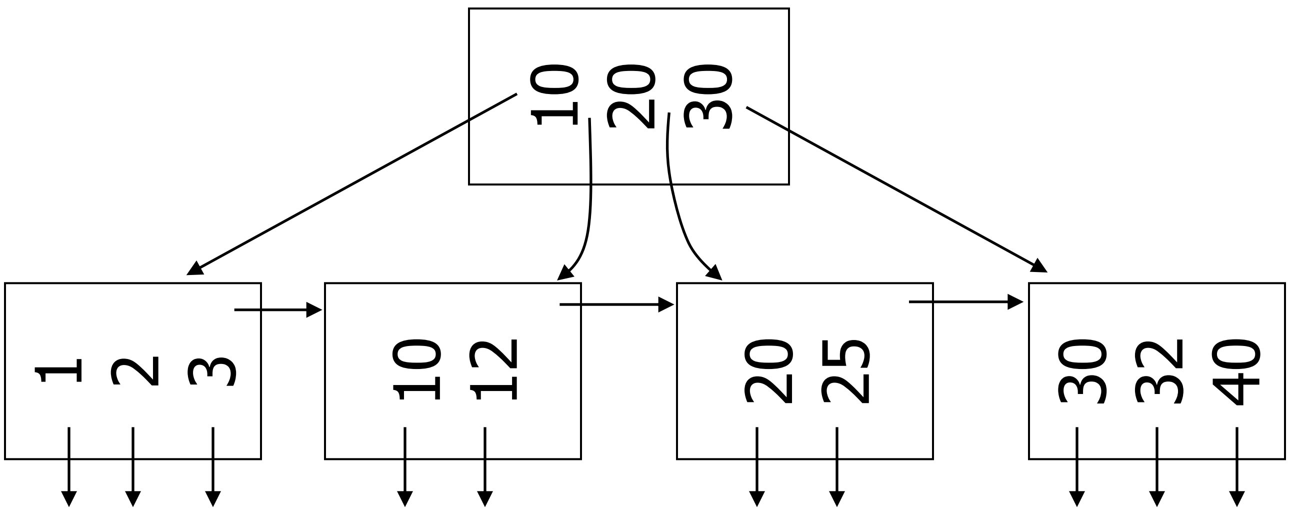
# (c) Insert key = 160

n=3



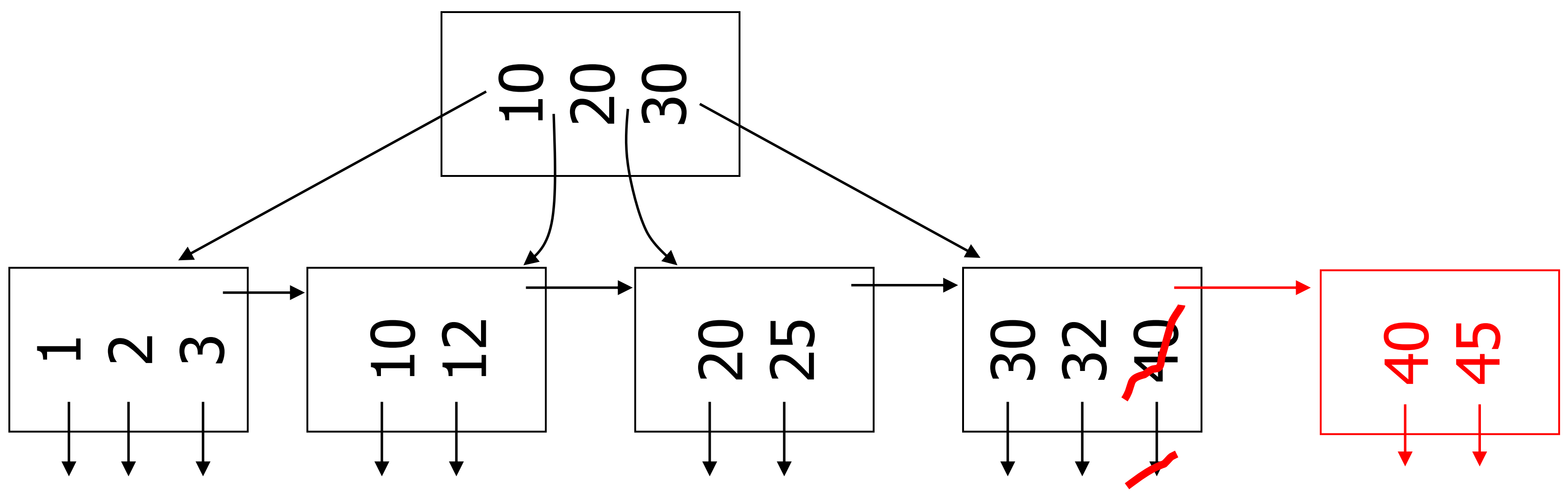
# (d) New root, insert 45

n=3



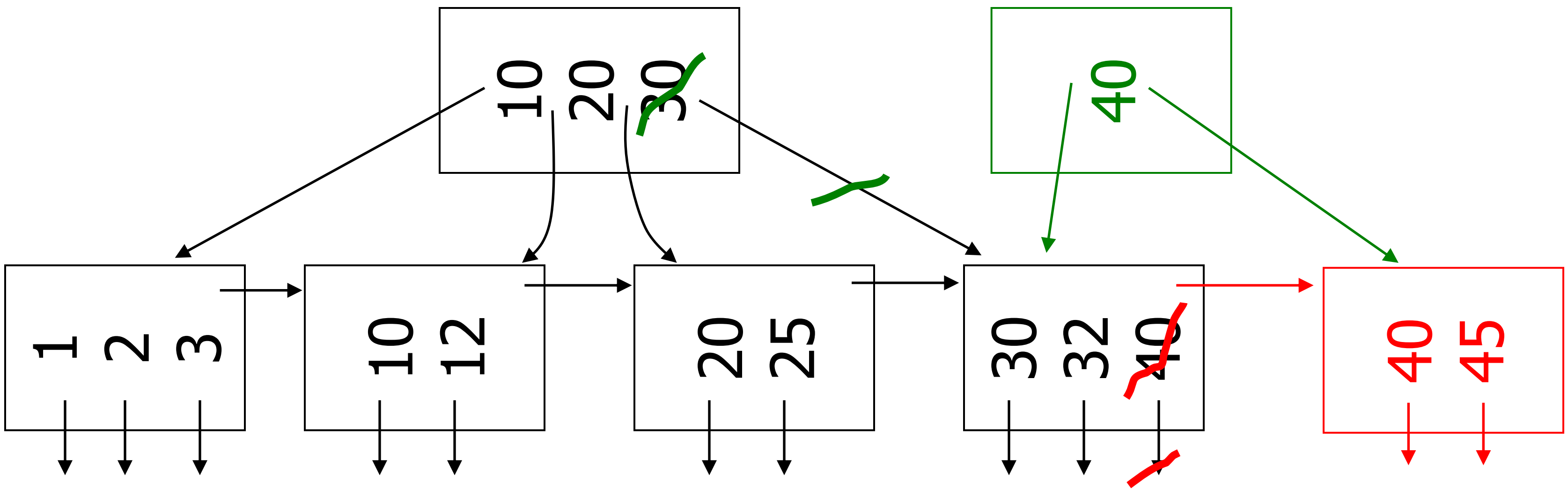
# (d) New root, insert 45

n=3



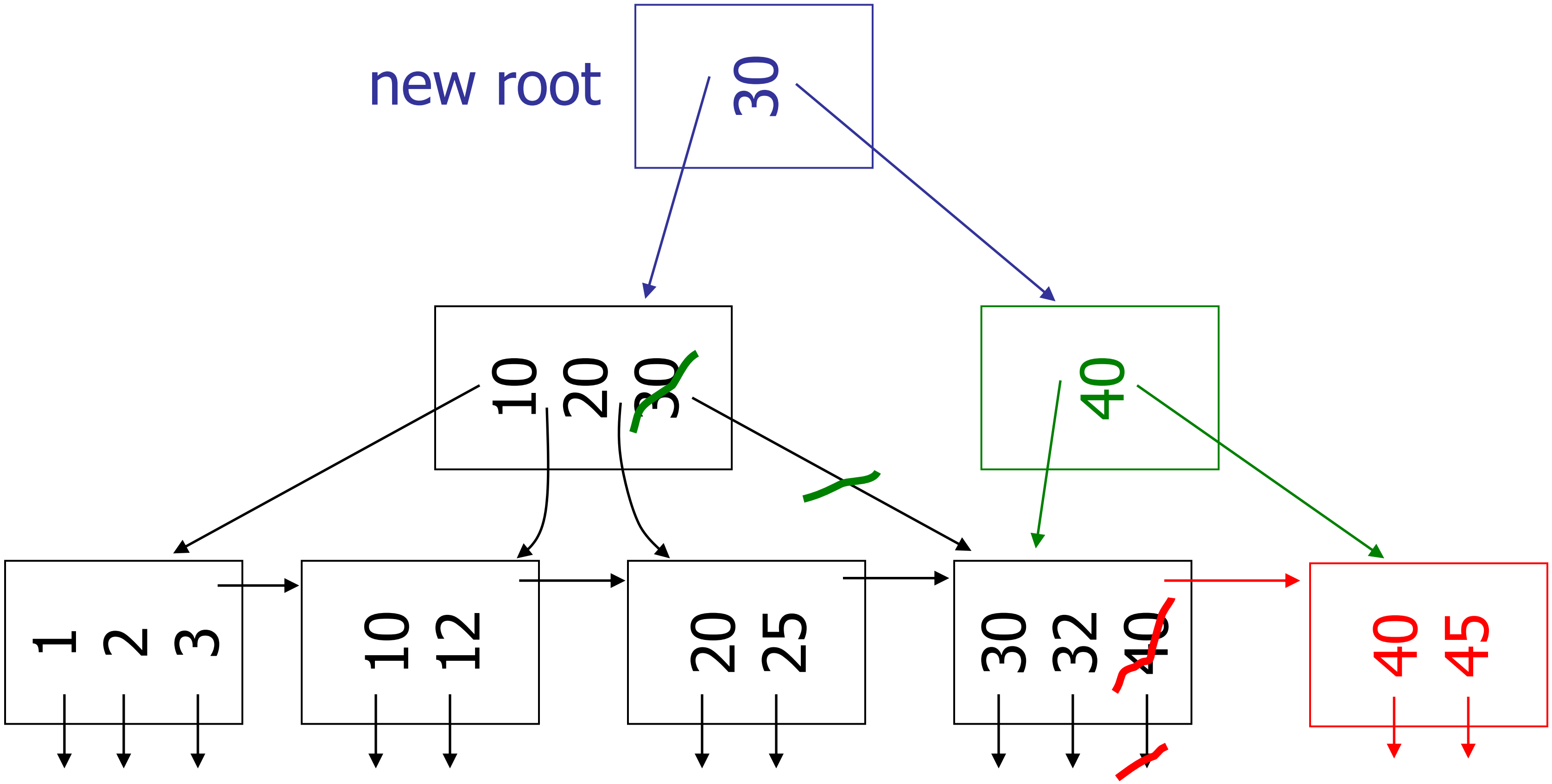
# (d) New root, insert 45

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

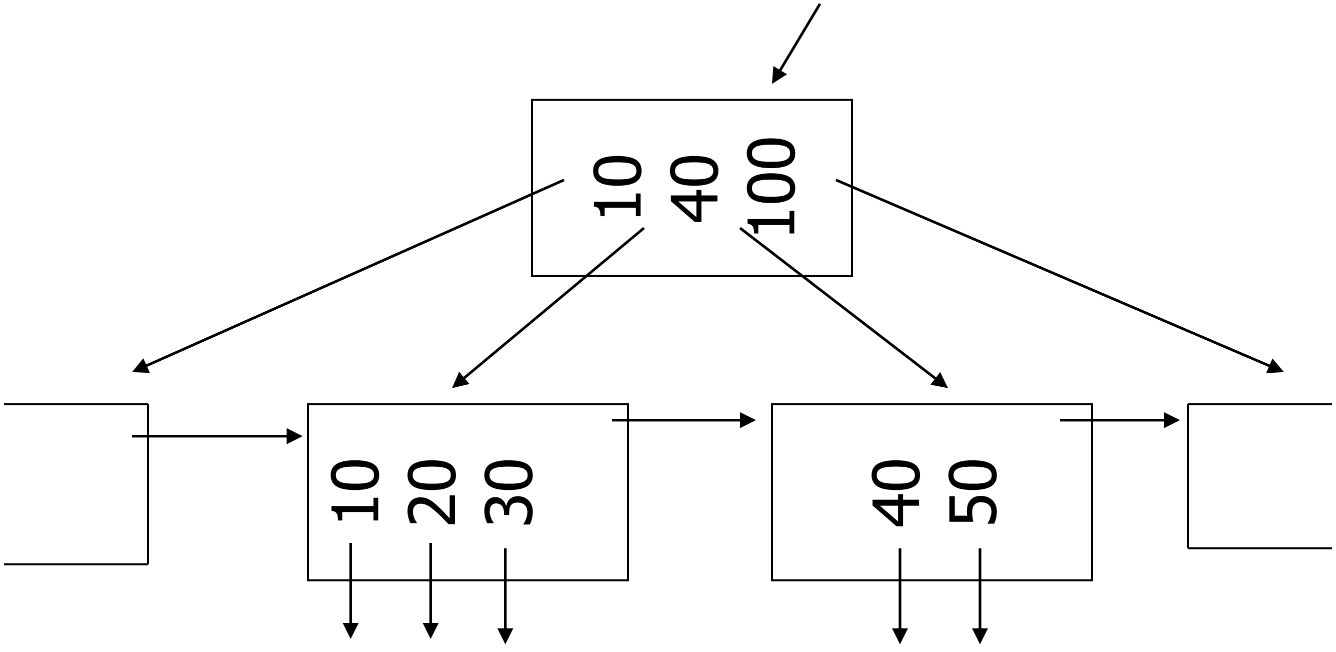
# Delete from BTree

- Locate that record (lookup)
- If after the deletion, the tree has minimum number of records, nothing to be done.
- Move the key from one of the sibling and adjust the parents.
- Hard case: no sibling has extra key
  - Merge the nodes by deleting one of them.
  - Adjust the keys at the parent.
  - If the parent is not full, we recursively continue the deletion process.

# (b) Coalesce with sibling

– Delete 50

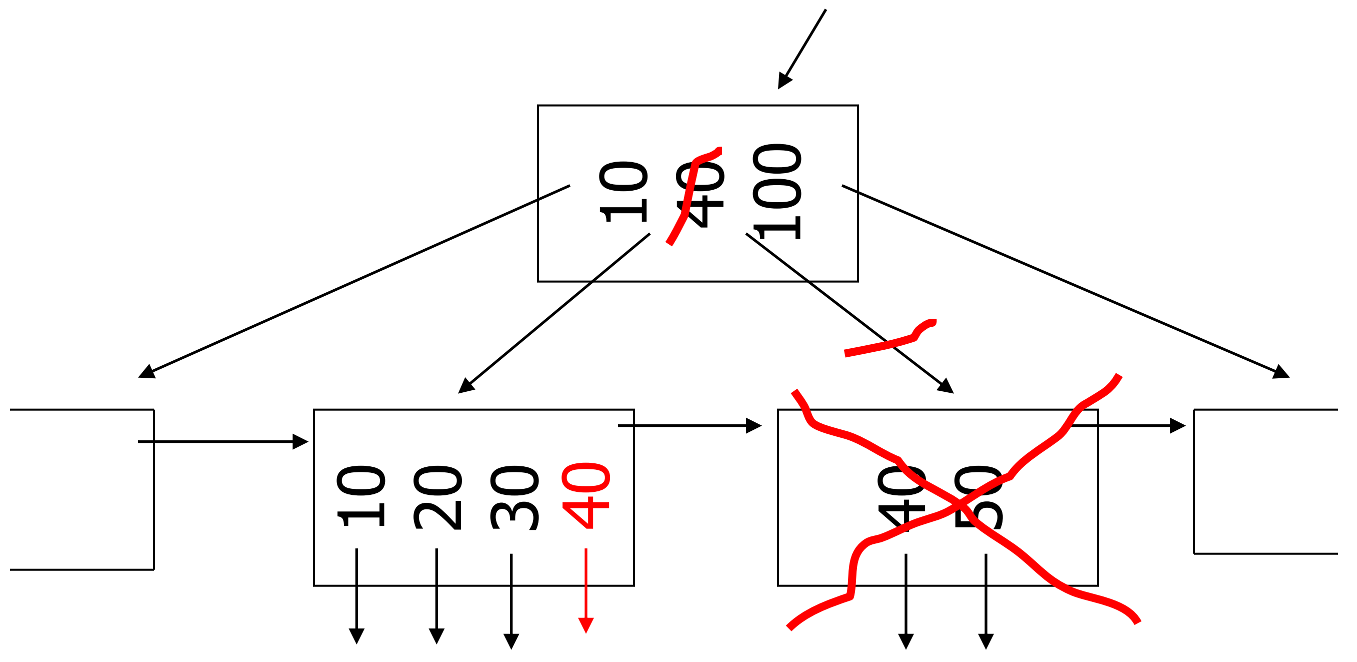
n=4



# (b) Coalesce with sibling

– Delete 50

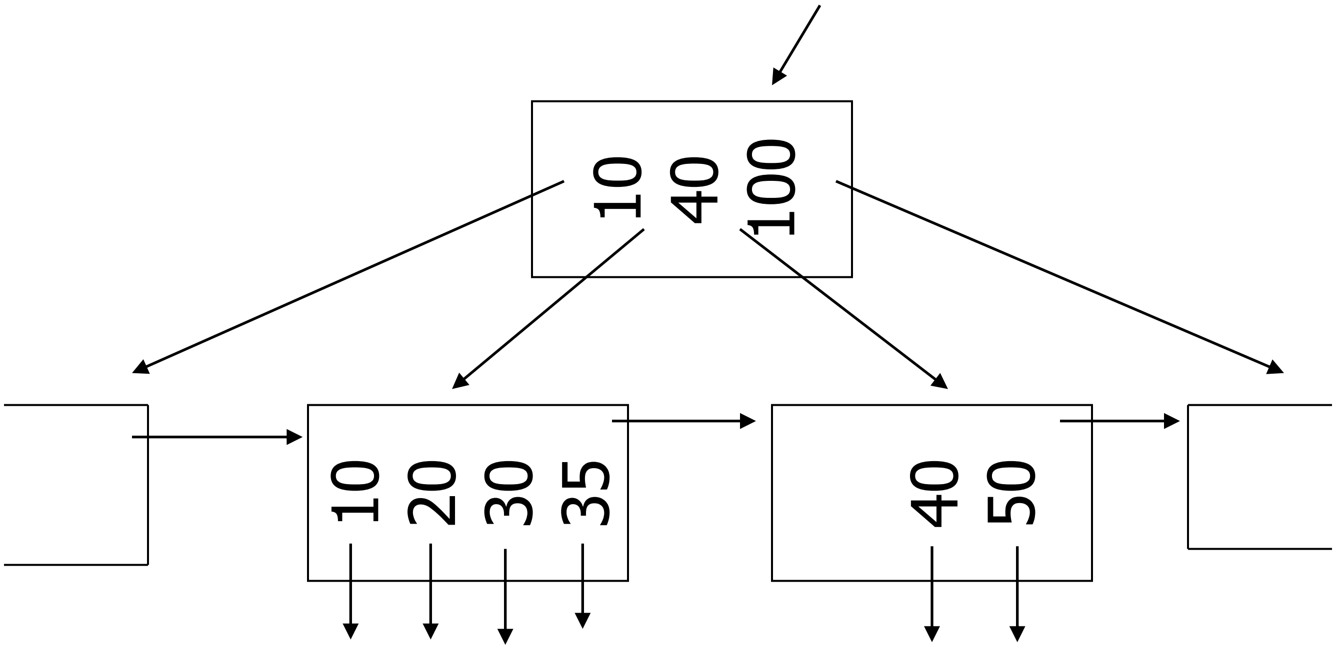
n=4



# (c) Redistribute keys

– Delete 50

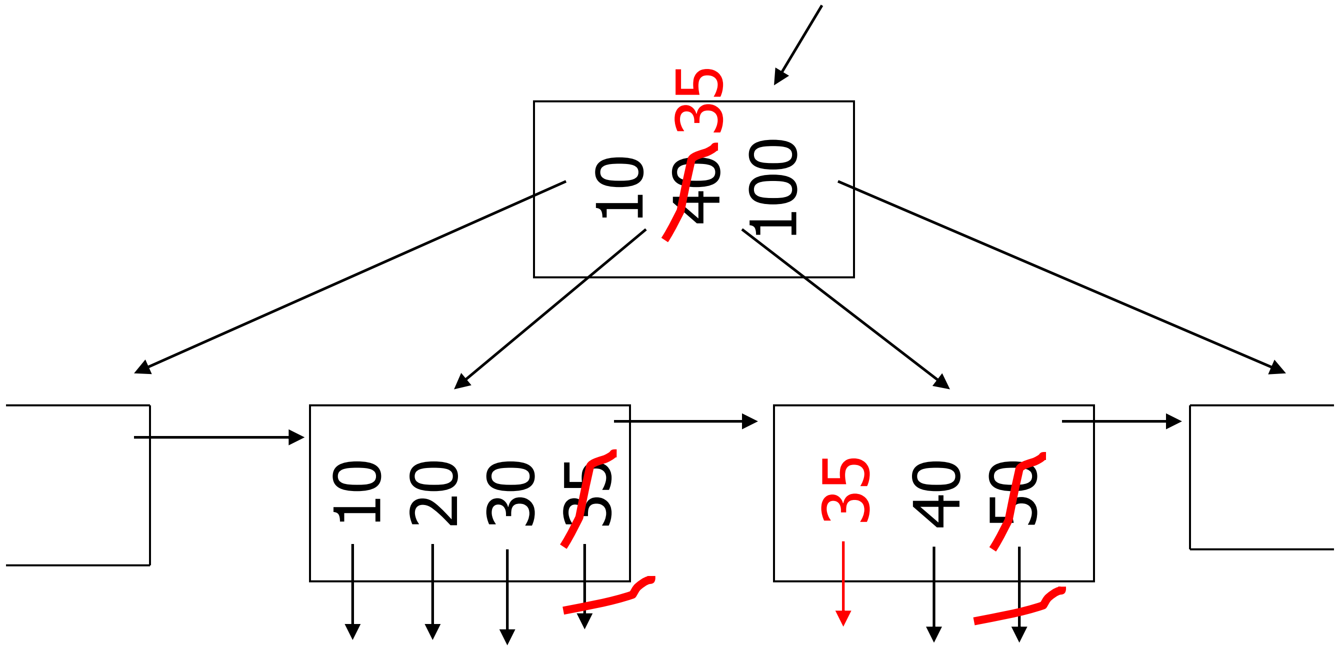
n=4



# (c) Redistribute keys

– Delete 50

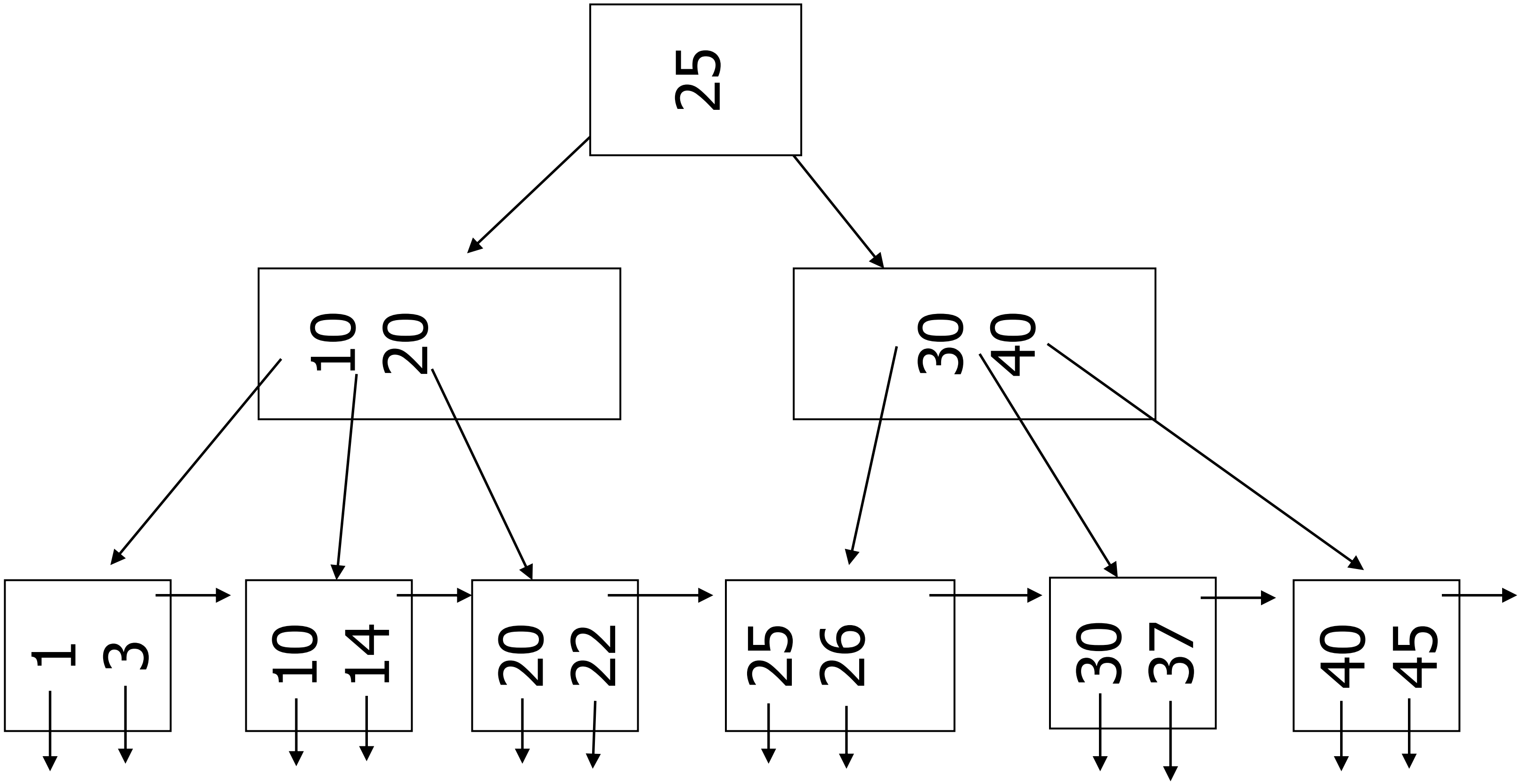
n=4



# (d) Non-leaf coalese

– Delete 37

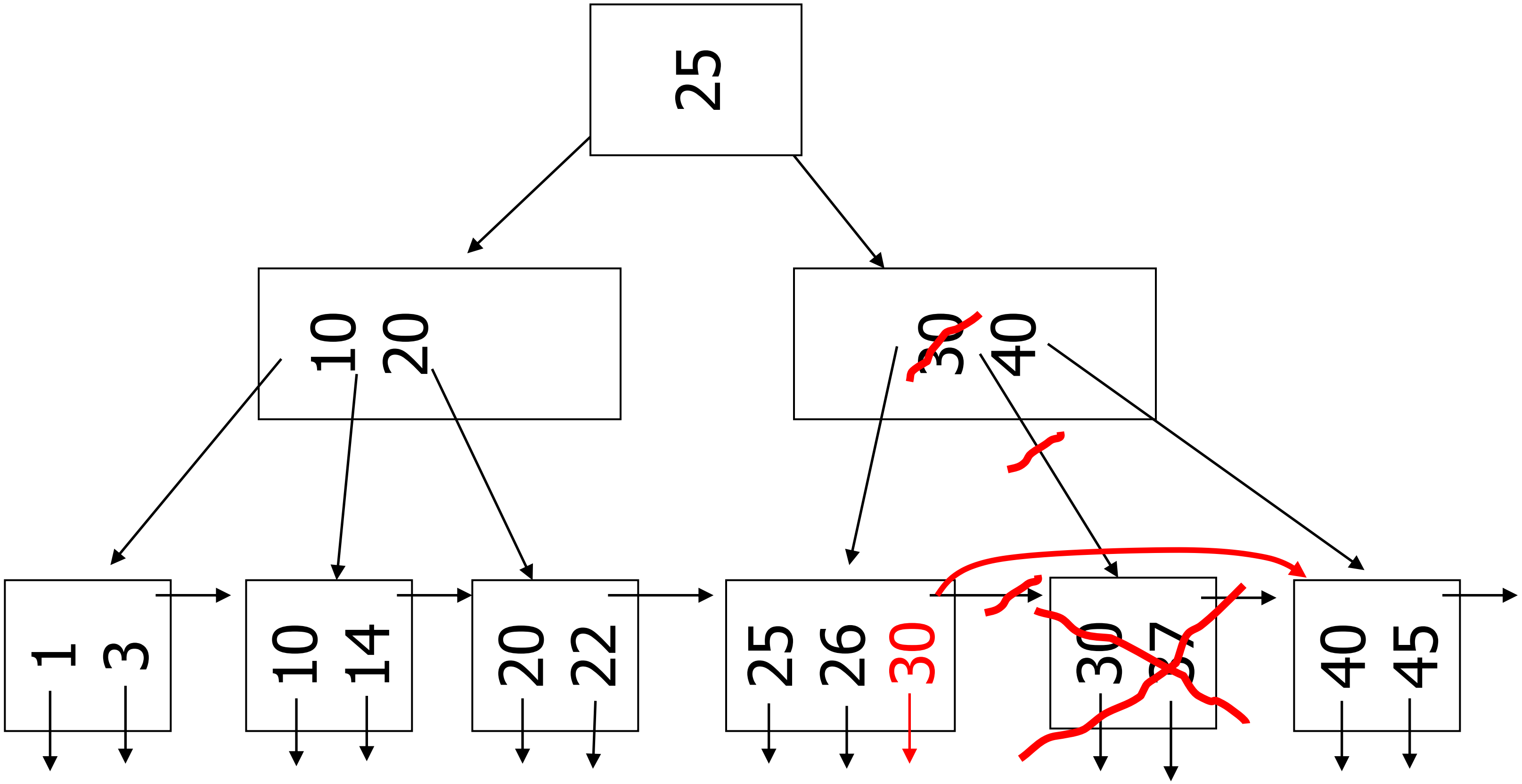
n=4



# (d) Non-leaf coalese

– Delete 37

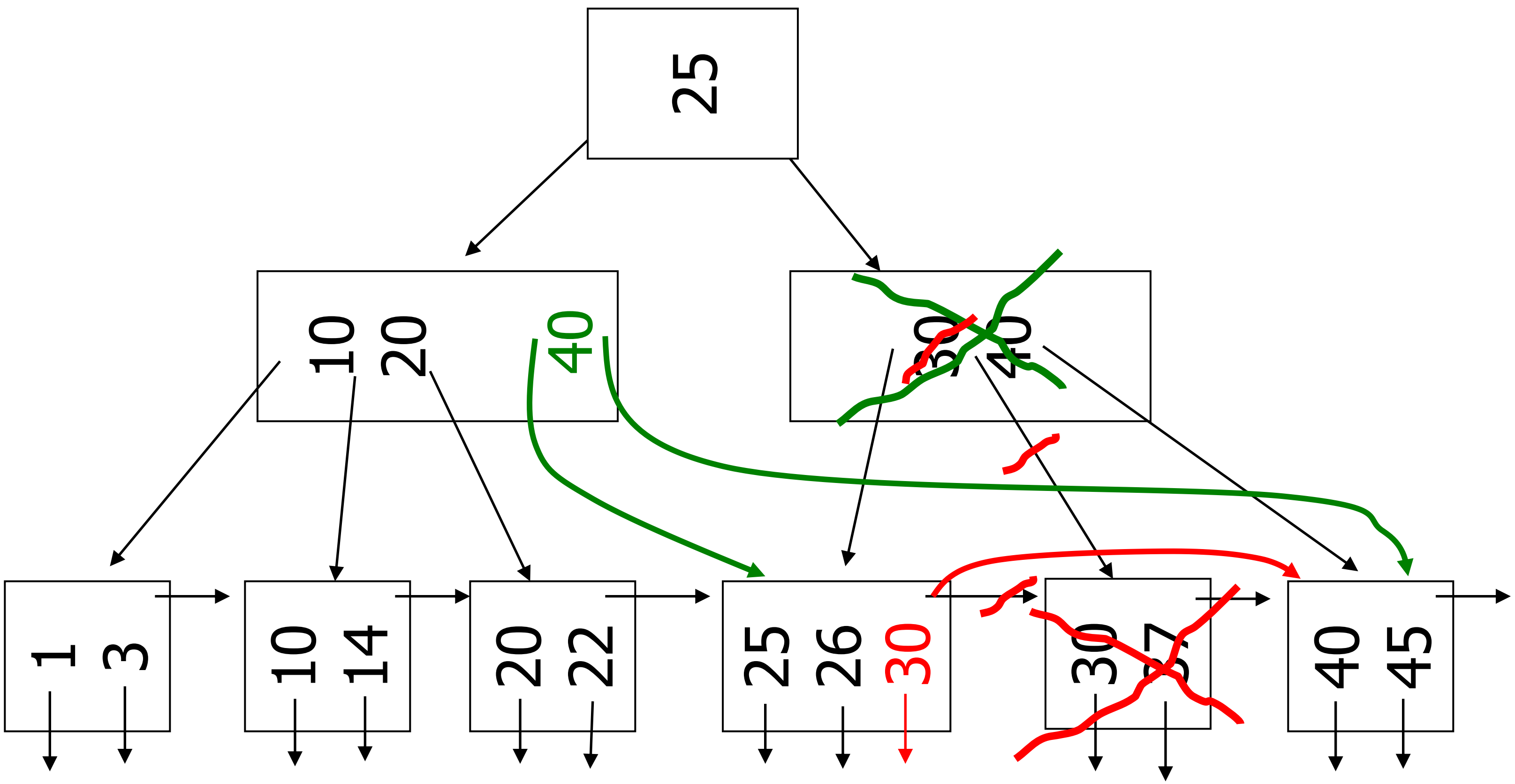
n=4



# (d) Non-leaf coalese

– Delete 37

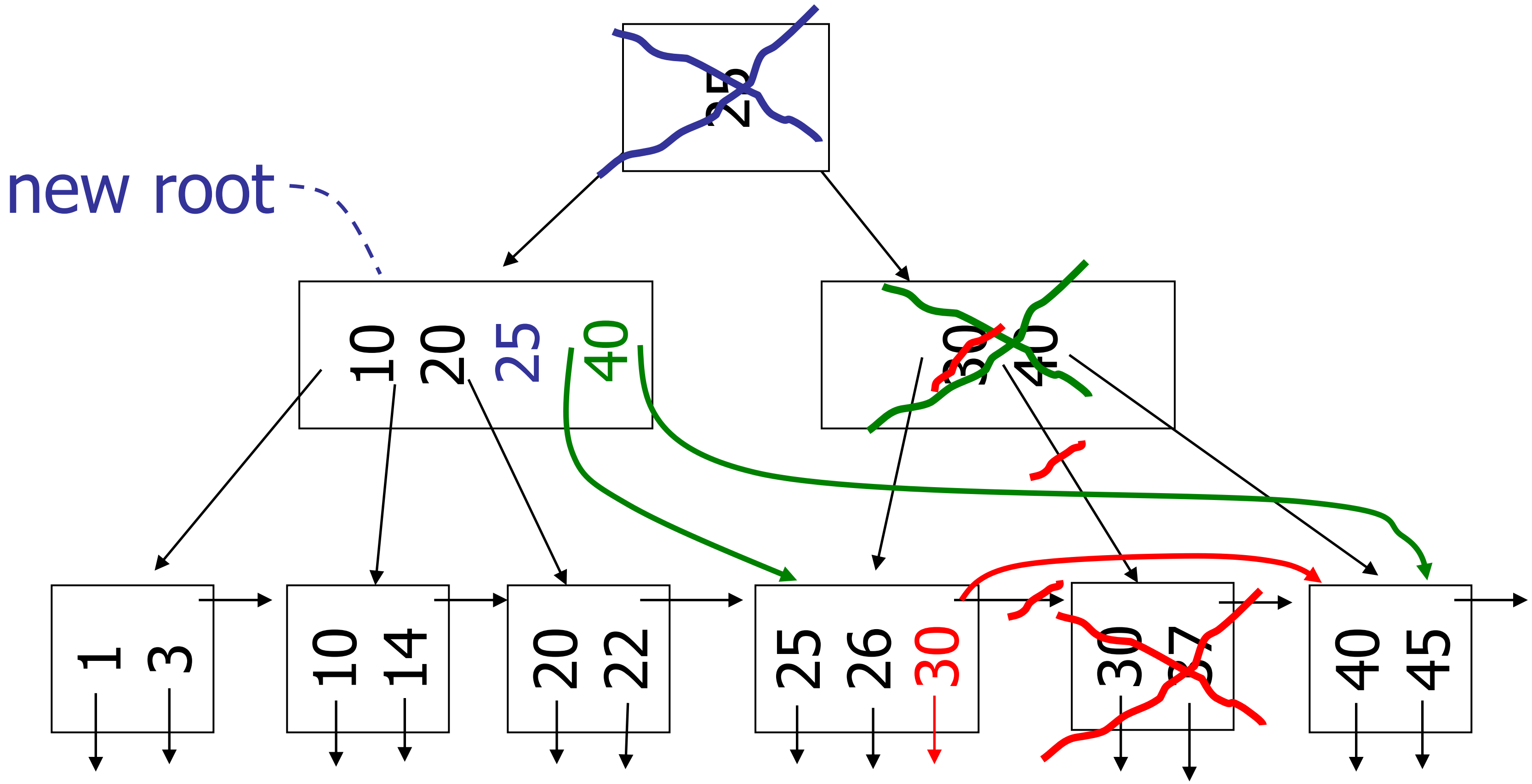
n=4



# (d) Non-leaf coalese

– Delete 37

n=4



# Efficiency of Btrees

- Very few disk I/Os per file operation
- Splitting and merging blocks occur rarely
- Keep the root in the main memory

# B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: $133^4$ = 312,900,700 records
  - Height 3: $133^3$ = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

# Hash Tables

- A hash function takes a search key as an argument and computes an integer range 0 to B-1, where B is a number of buckets.
  - A bucket array, which is an array indexed from 0 to B-1, holds headers of B linked lists, one for each bucket of the array.
  - If the record has a search key K, we store the record by linking it to the bucket list for the bucket numbered h(K), where h is the hash function.
- Common hash function
  - Remainder of K/B

# Secondary Storage Hash Tables

- Bucket contains blocks
- If a bucket overflows, a chain of overflow blocks are added.

# Insertion into a Hash Table

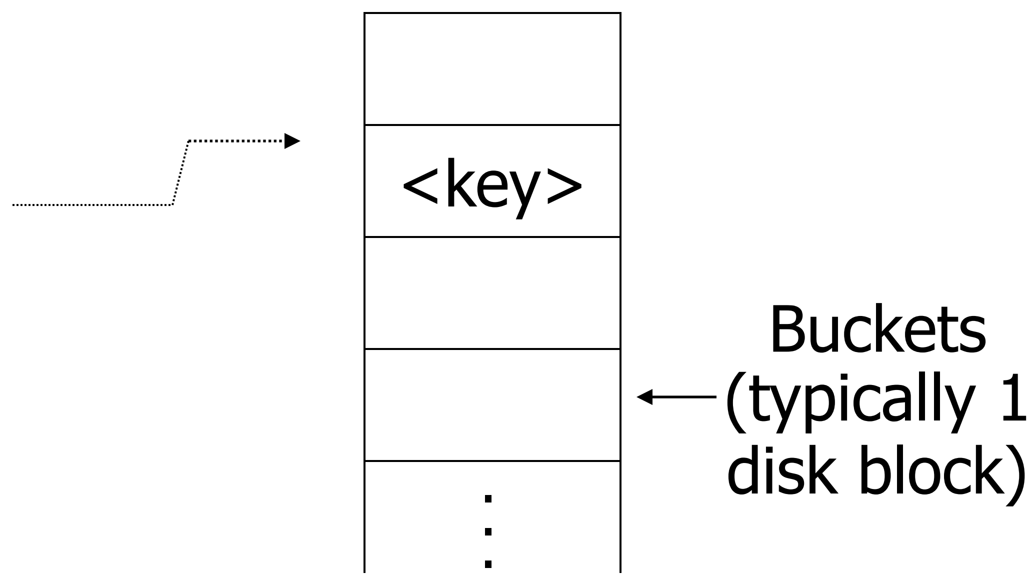
- When a new record with search key K must be inserted, we compute h(K).
- If the bucket number h(K) has the space, we insert the record into the block for this bucket or into one of the chain of blocks.
- If there is no space, we add extra block.

# Hash-table insertion

- Go to the bucket number h(K) and search for records with that search key. Delete if we find any data.
  - Consolidate (optional)

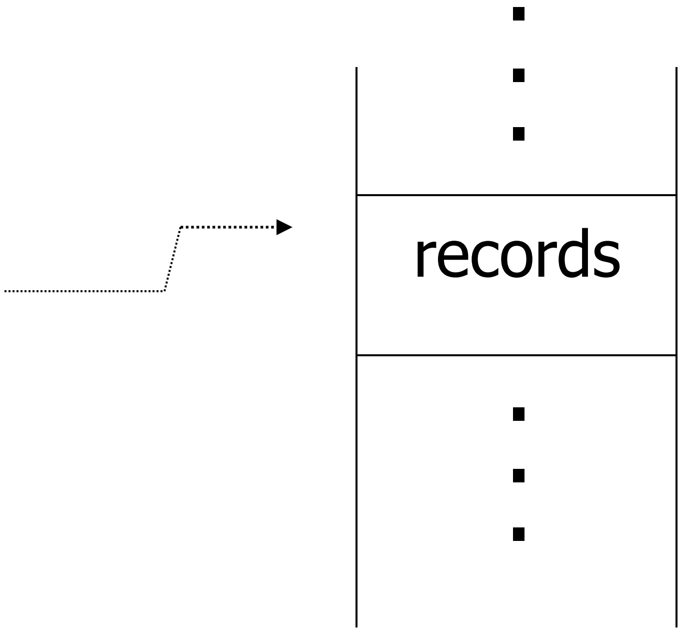
# Hashing

$$\text{key} \rightarrow h(\text{key})$$



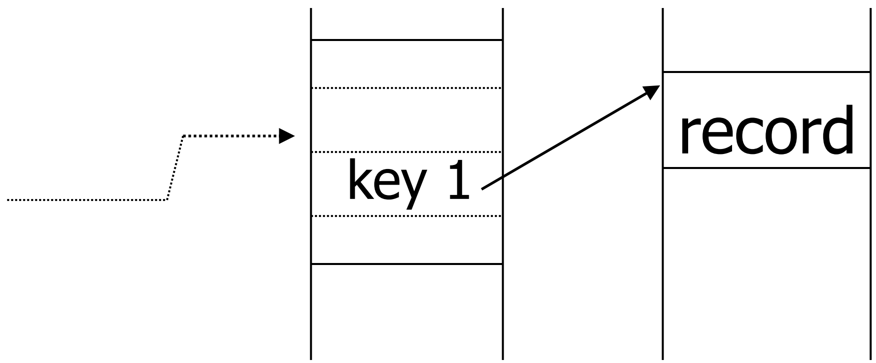
# Two alternatives

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



# Two alternatives

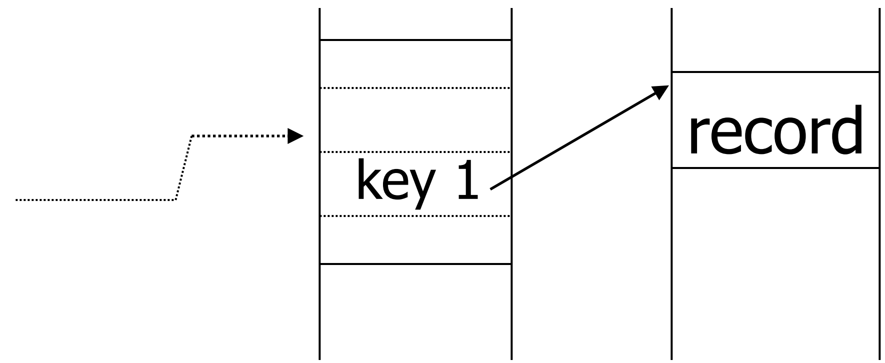
(2) key $\rightarrow$ h(key)




Index

# Two alternatives

(2) key $\rightarrow$ h(key)




Index

- Alt (2) for "secondary" search key

# Example hash function

- Key = '$x_1\ x_2 \ldots x_n$'   *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.

☒ This may not be best function …

☒ Read Knuth Vol. 3 if you really need to select a good function.

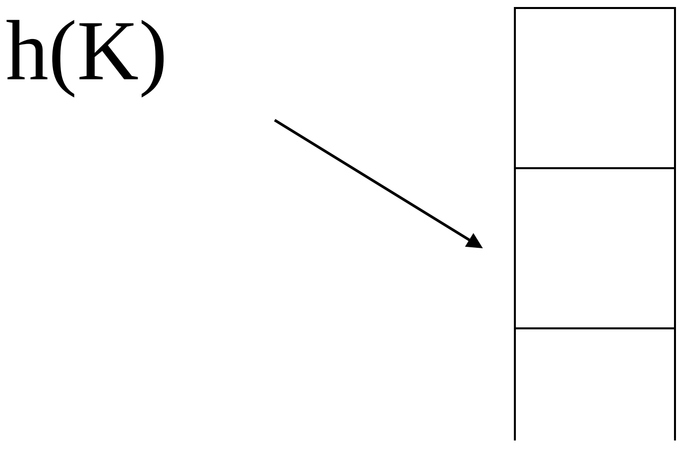
Good hash function: ☞ Expected number of keys/bucket is the same for all buckets

# 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

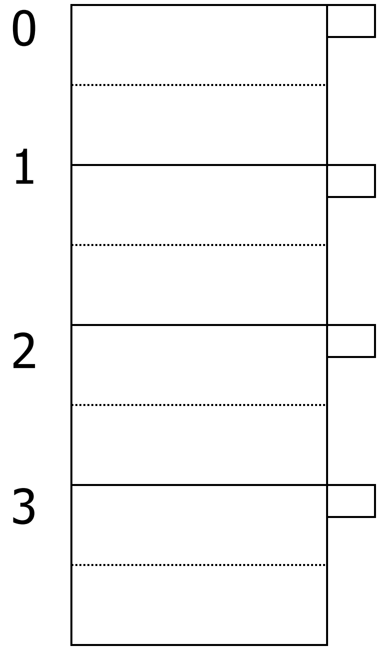
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0



# EXAMPLE 2 records/bucket

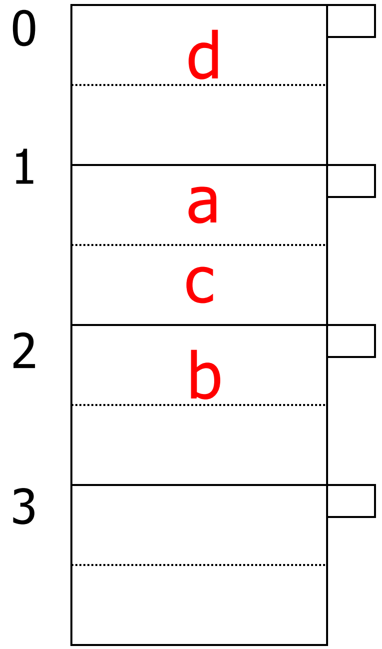
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0




h(e) = 1

# EXAMPLE 2 records/bucket

INSERT:

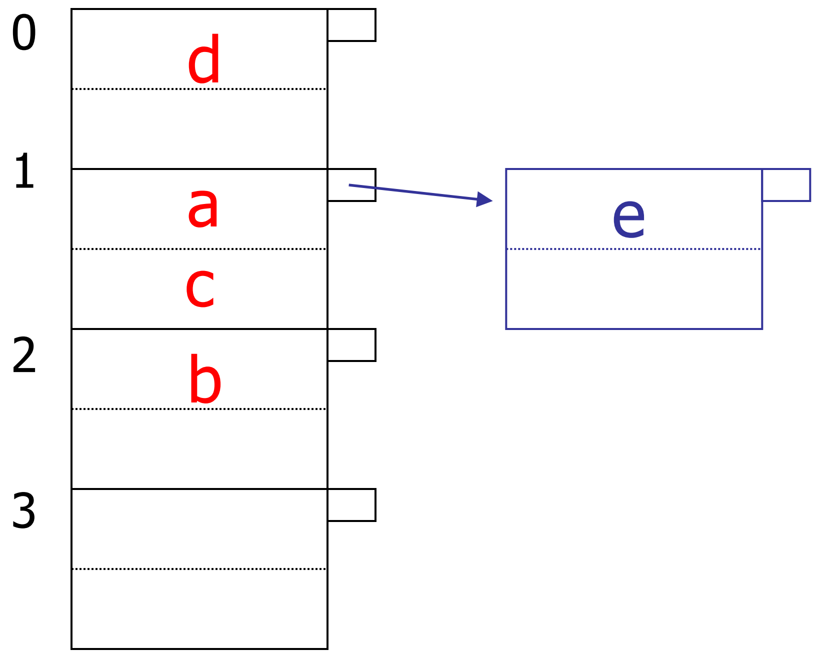
h(a) = 1

h(b) = 2

h(c) = 1

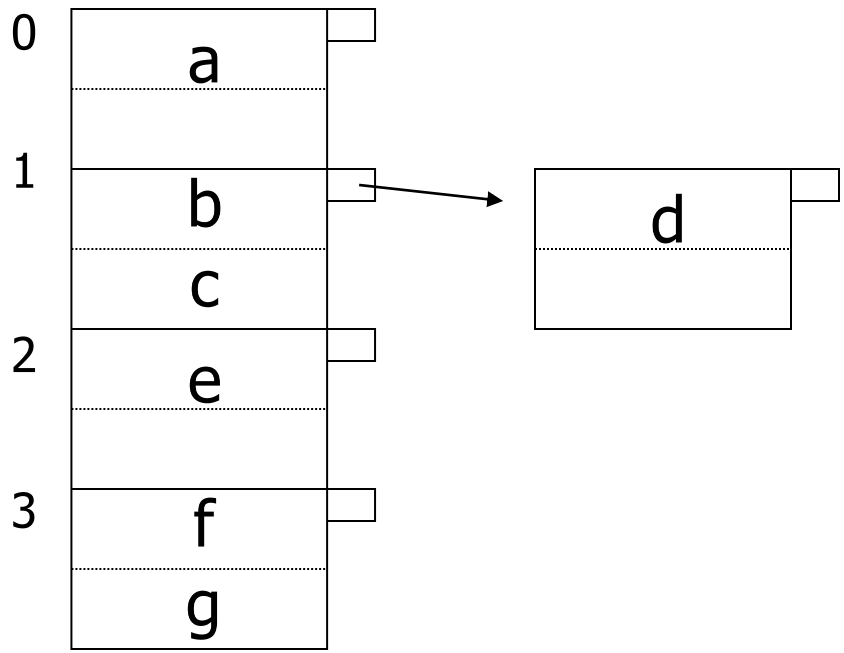
h(d) = 0

h(e) = 1



# EXAMPLE: deletion

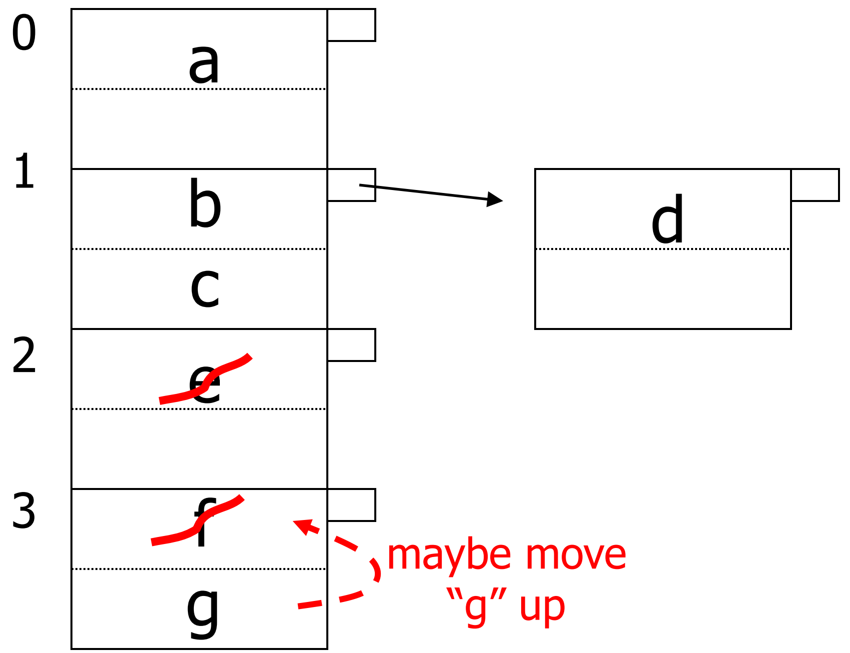
Delete:
- e
- f



# EXAMPLE: deletion

Delete:

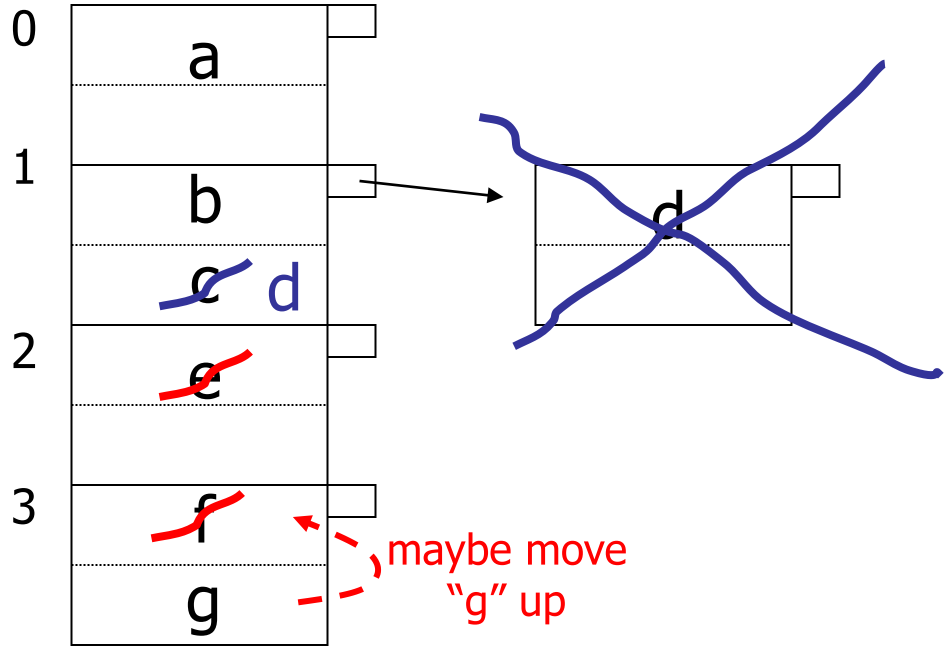
- e
- f
- c



# EXAMPLE: deletion

Delete:

- e
- f
- c



# Rule of thumb:

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

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

# Rule of thumb:

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

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

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

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

# Efficiency of hash table indexes

- Typical lookup takes only one disk I/O
- Insertion/deletion takes only two disk I/Os.
  – Better than sparse or dense indexes
  – Btree
- If the file grows, there will be many blocks in the chain for a typical bucket and taking one disk I/O per block
  – So it is better to keep number of blocks per bucket low.

# Dynamic hash tables

- So far we have discussed static hash tables
  - The number of buckets never changes.
  - Overflow problems
- There are several kinds of dynamic hash tables, where B is allowed to vary.
  - Extensible hashing
  - Linear hashing

# Extensible Hash Tables

- There is a level of indirection introduced for the buckets
- An array of pointers to blocks represents the buckets instead of the array consisting of the data blocks themselves.
- The array of pointers can grow. The length is always a power of two
  - The number of blocks doubles
- There need not be data block for each bucket; certain buckets can share a block if the total number of records in those buckets can fit in the block
- The hash function computes a sequence of k-bits for some large k,
  - However, the bucket numbers will at all times use some smaller number of bits, say "i" bits from the beginning of sequence.
  - The bucket array will have $2^i$ entries when i is the number of bits used.

# Insertion into Extensible Hash Tables

- To insert a record with search key K, we compute h(K), take the first i bit sequence and go to the entry of the bucket array indexed by these i bits.
- Follow the pointer
  - If there is a room insert
  - If there is no room, determine the number of bits to determine used to determine membership in block B.
    - If j<i nothing needs to be done to bucket array
      - Split block B into two
      - Distribute records in B to the new blocks based on j+1 bit.
      - Put j+1 in each blocks nub to indicate the number of bits used to determine the membership
      - Adjust the pointers in the bucket array so entries that formerly pointed to B now point either to B or the new block depending on the j+1 bit.

# Insertion into Extensible Hash Tables

- If j=i
  - Increment i by 1.
  - Double the length of the bucket array, so it now has $2^{i+1}$ entries.

# Extensible hashing: two ideas

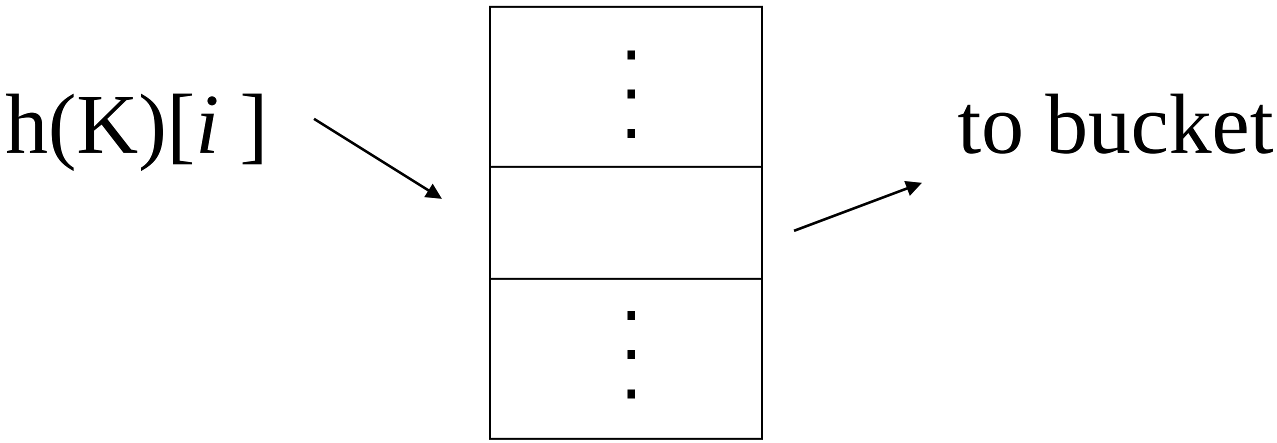
(a) Use *i* of *b* bits output by hash function

$$\longleftarrow \quad b \quad \longrightarrow$$

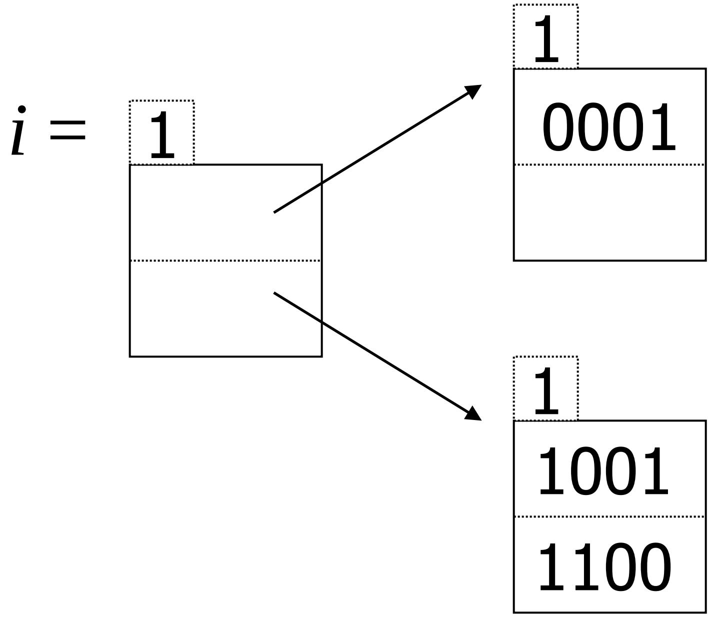
$$h(K) \rightarrow \boxed{\underbrace{0011}_{}0101}$$

use *i* → grows over time….

(b) Use directory

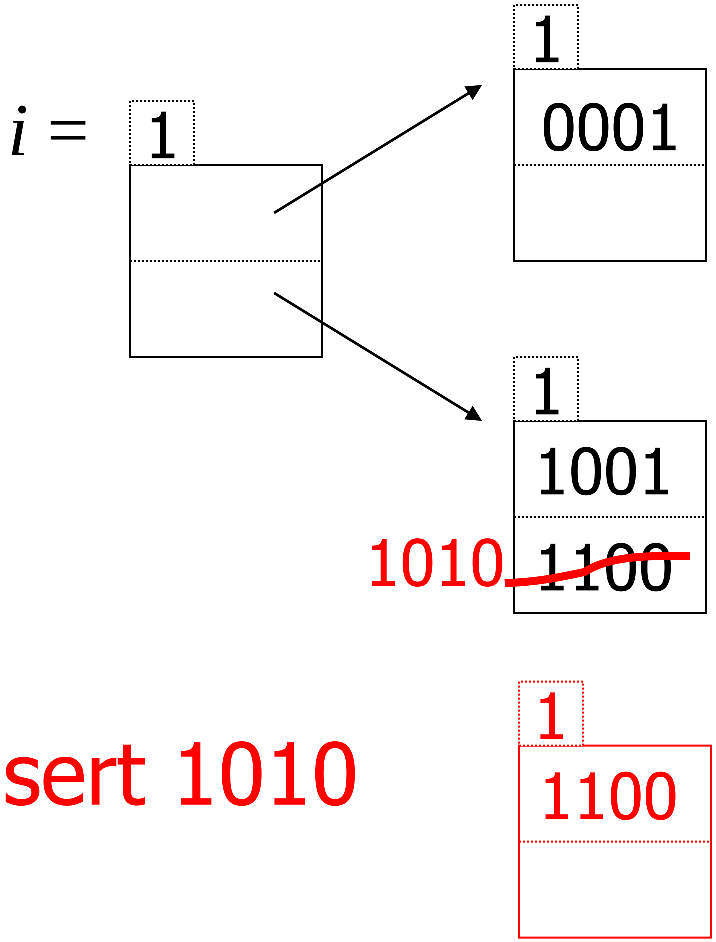


# Example: h(k) is 4 bits; 2 keys/bucket



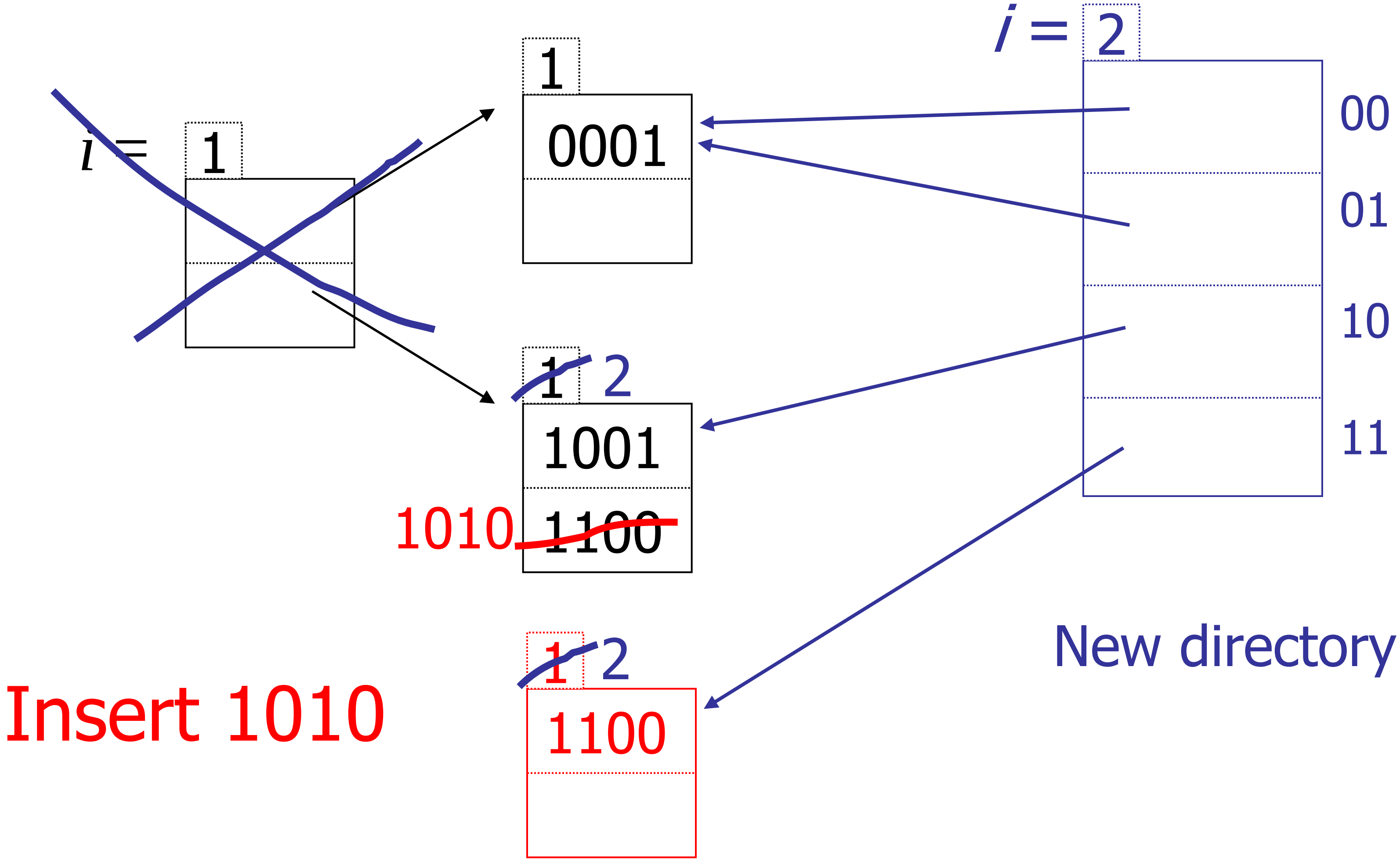

Insert 1010

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

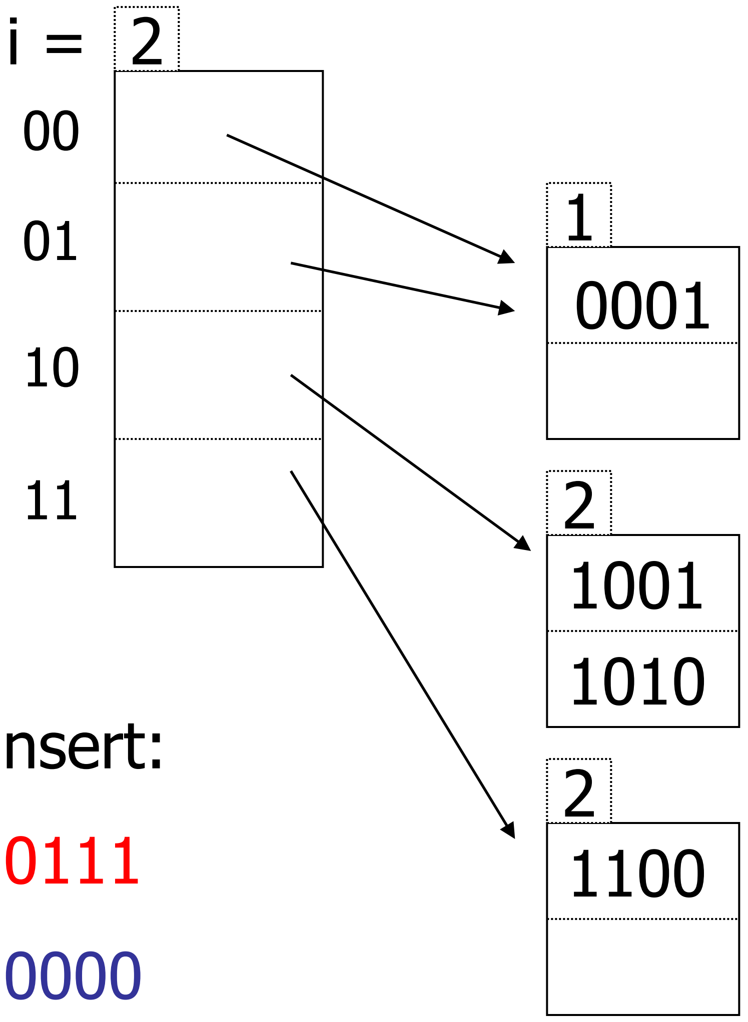



Insert 1010

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



# Example continued

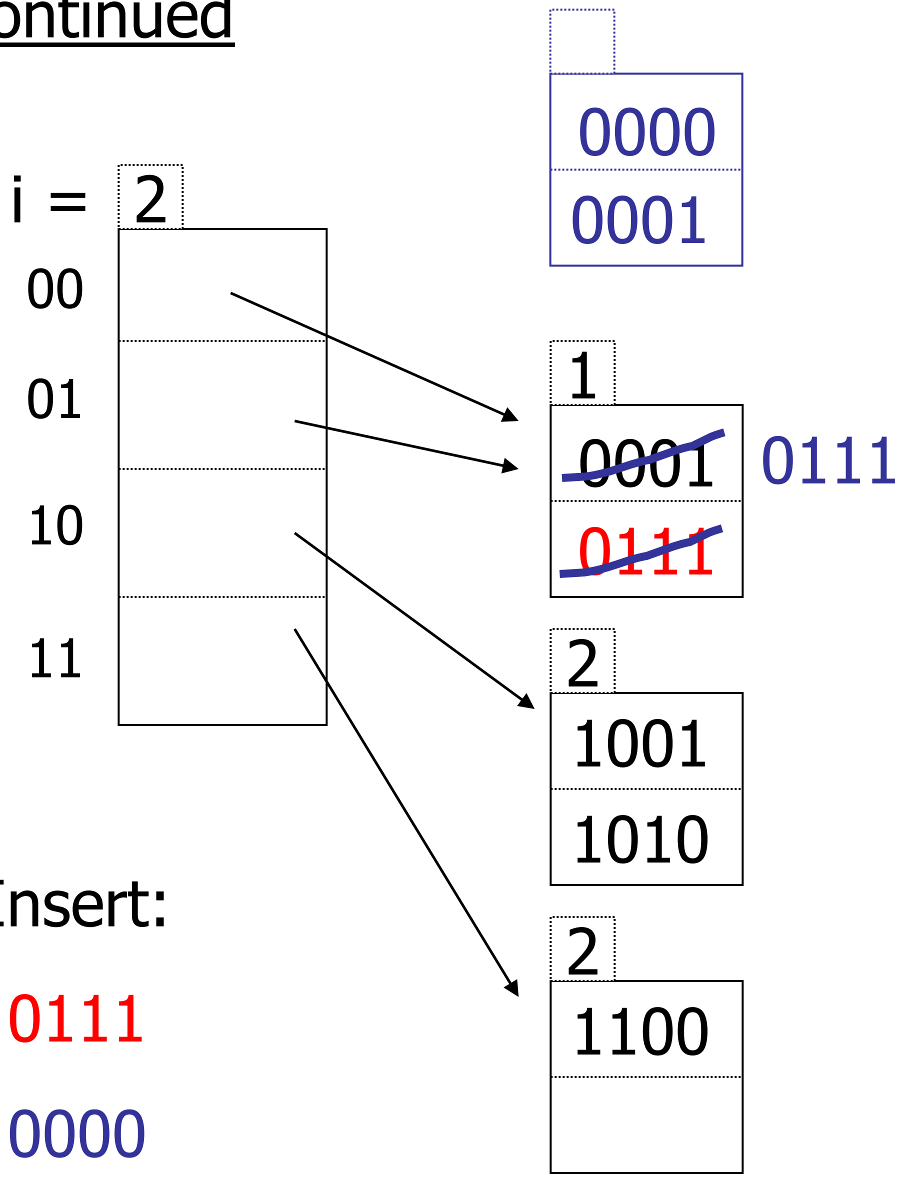



Insert:

0111

0000

# Example continued

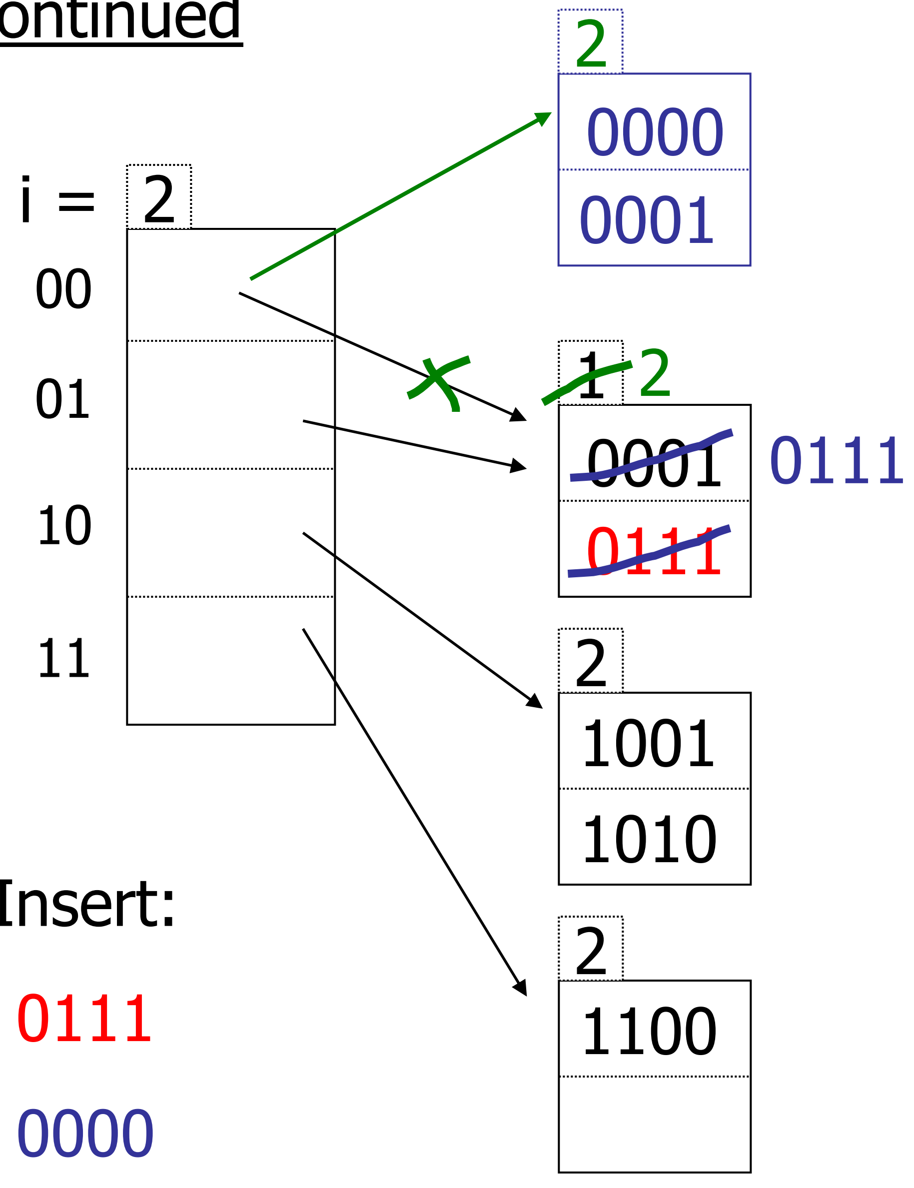



Insert:

0111

0000

# Example continued

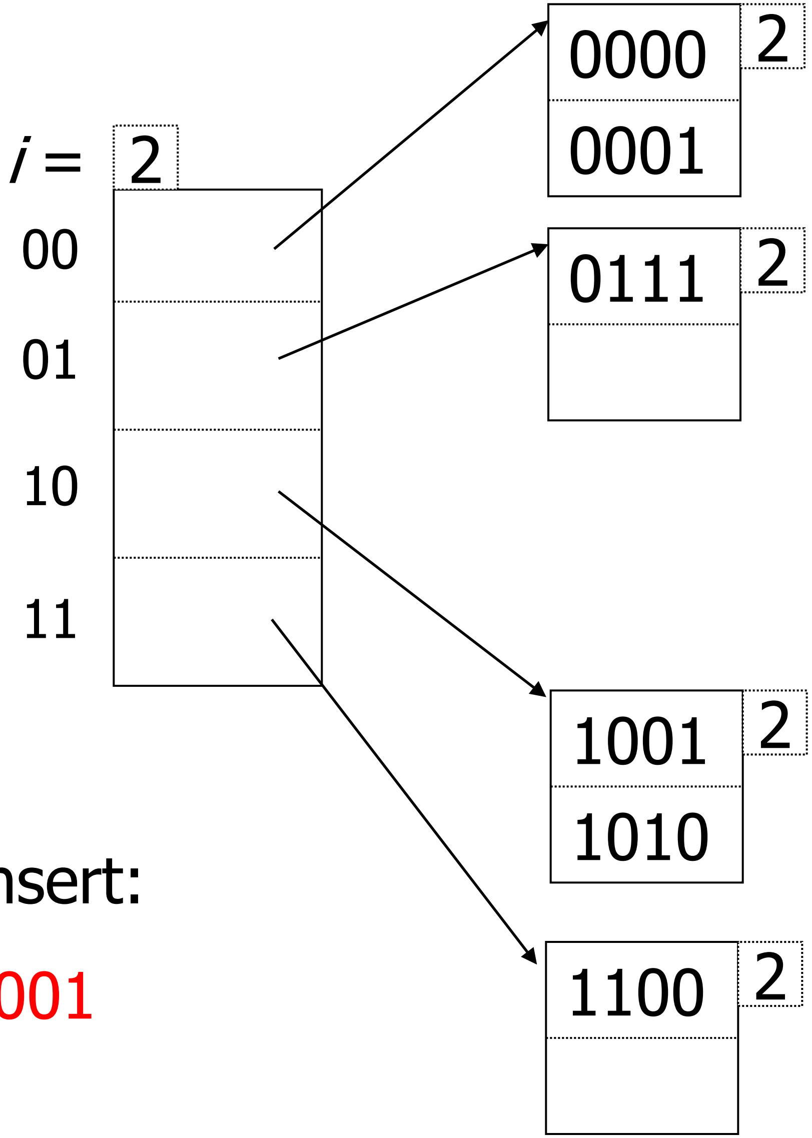



Insert:

0111

0000

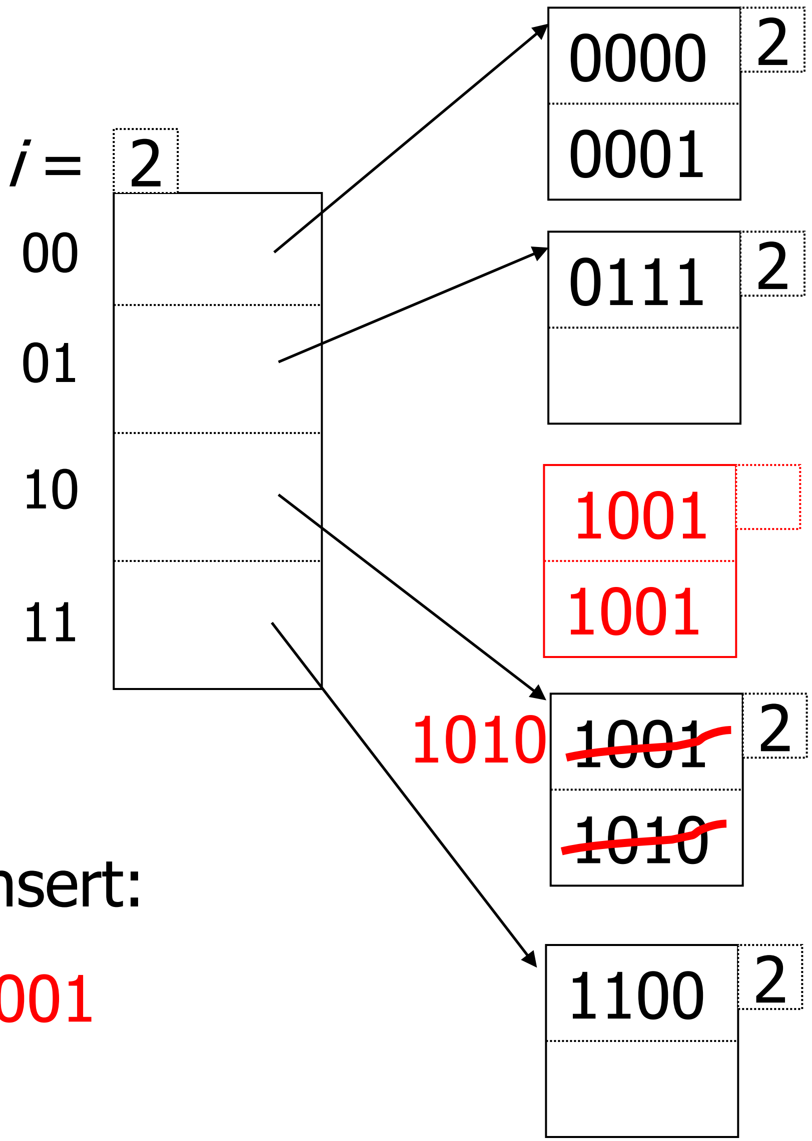
# Example continued




Insert:

1001

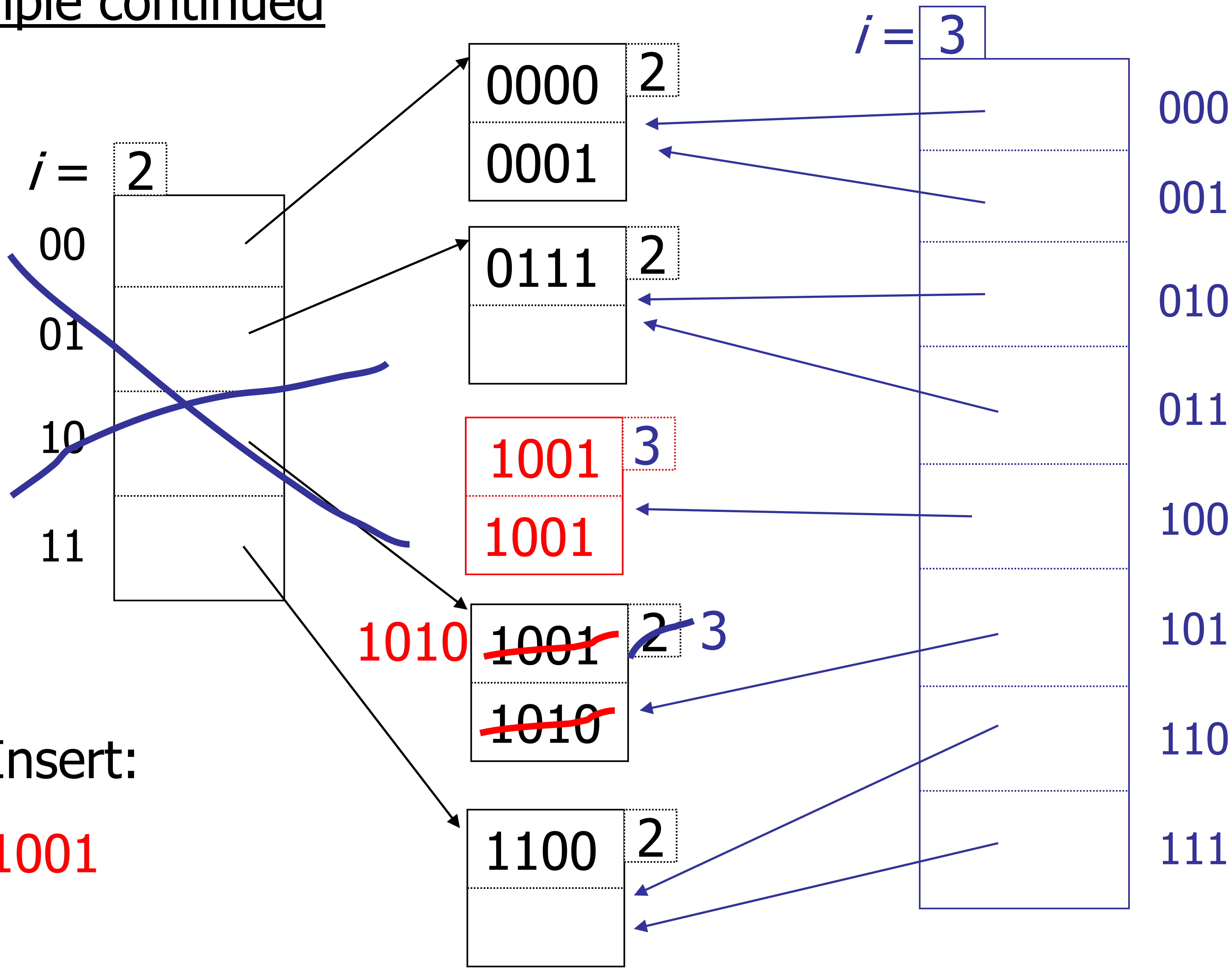
# Example continued




Insert:

1001

# Example continued




Insert:

1001

# Extensible hashing: deletion

- No merging of blocks
- Merge blocks
  and cut directory if possible

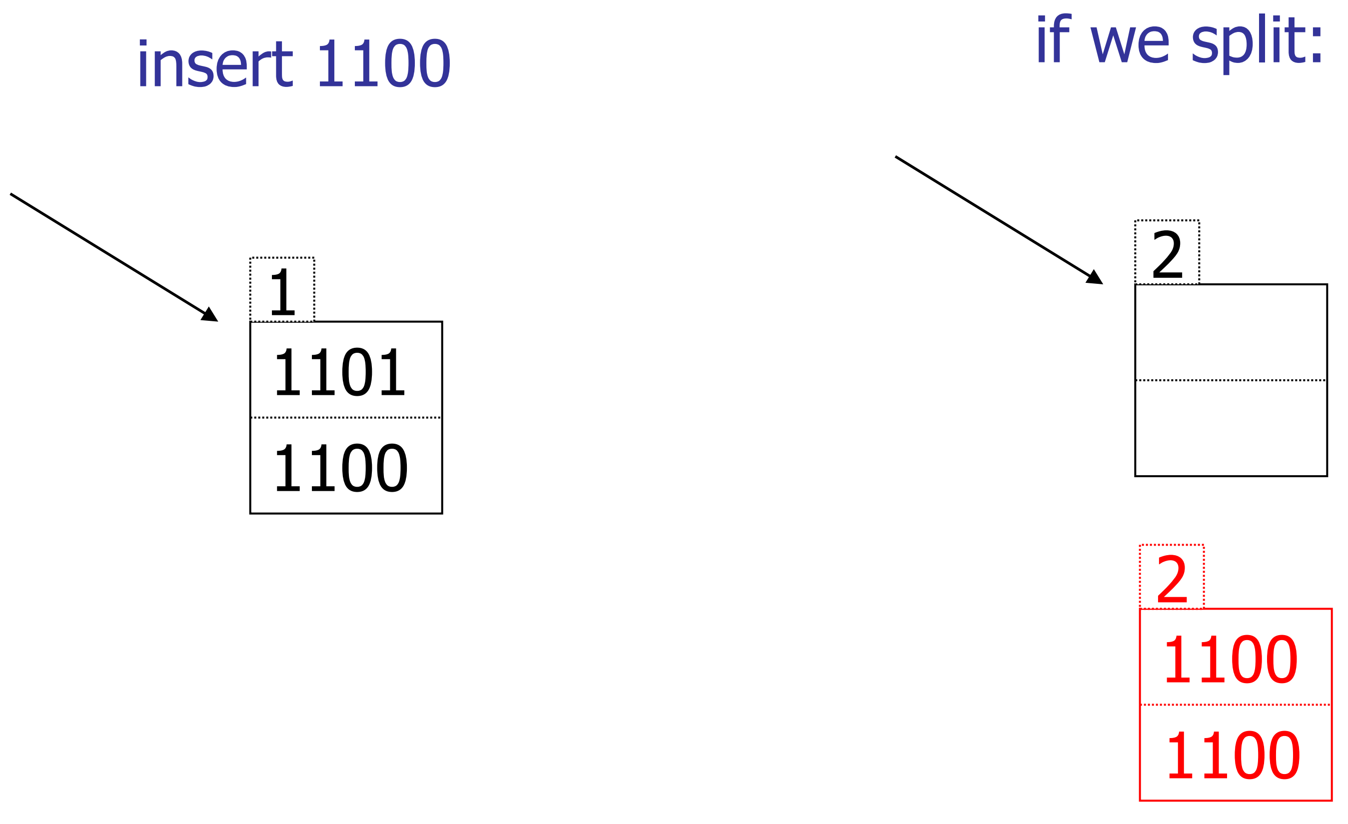
  (Reverse insert procedure)

# Deletion example:

- Run thru insert example in reverse!

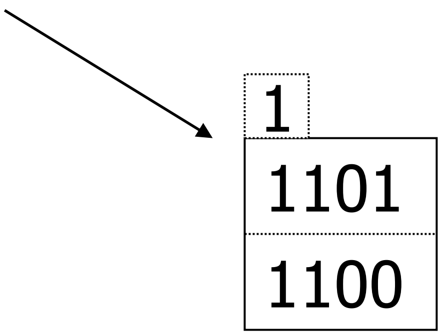
# Note: Still need overflow chains

- Example: many records with duplicate keys

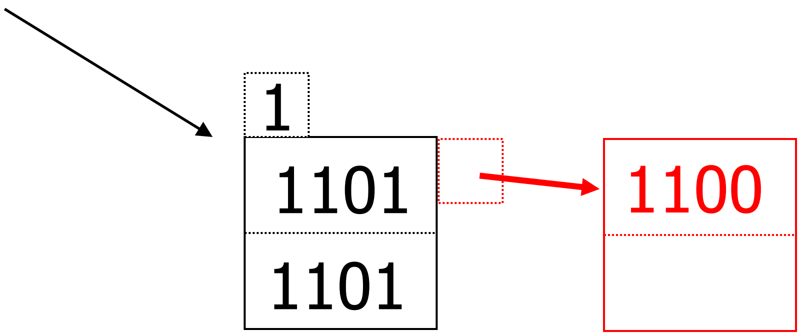


# Solution: overflow chains

insert 1100




add overflow block:



# 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

- Advantages of extensible hashing
  - we need to search only one data block
  - We have to examine the bucket array
    - But if the bucket is small to fit in main memory, there is no disk I/O.
- Problems with extensible hash
  - Doubling requires substantial work when “i” is large.
    - Interrupts access to data file
  - If it does not fit in main memory, may push other data that may be needed.
    - As a result the system may perform many disk I/Os
  - If the number of records per block is small, one block may be split multiple times.
    - There might be million bucket entries and the small number of data entries.
- LINEAR HASHING allows growth of number of blocks very slowly.

# Linear Hashing: Strategy

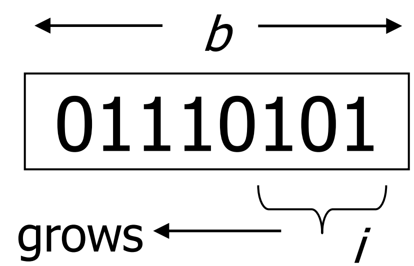
- The number of buckets n is equal to a fraction of number average number of records per block.
- Since blocks can not always split, overflow blocks are allowed
- The number of bits used to number of entries of bucket array is $\log_2 n$, n is the number of buckets. The bits are always taken from lower order side.
- Suppose with “i” bits of the hash function, record with K is intended for bucket a1, a2,…,ai (last i bits of k).

# Linear hashing

- Another dynamic hashing scheme

Two ideas:

(a) Use *i* low order bits of hash

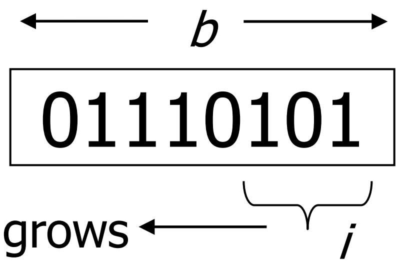


# Linear hashing

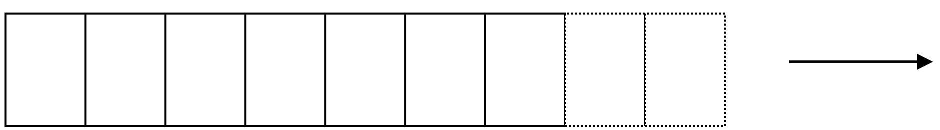
- Another dynamic hashing scheme

Two ideas:

(a) Use *i* low order bits of hash




(b) File grows linearly

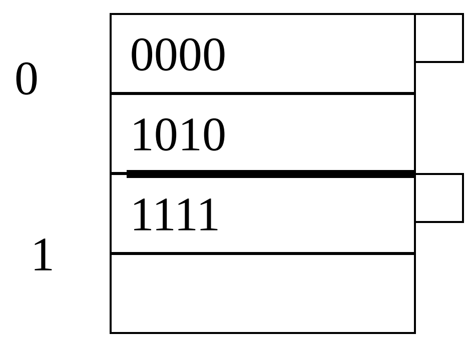


# Example

n is number of blocks= 2
Hash function h produces four bits
`i' is number of bits in hash function
n is number of current buckets
r is number of records in a table

The ratio of r/n will be limited to number of records in a block
In this case we ensure r <=1.7*n.
So average occupancy of a bucket can not exceed 85%.

| |
|---|
| i=1 |
| n=2 |
| r=3 |



# Insertion

$\boxed{\text{Rule}}$ If $h(k)[i\,] \le n$, then

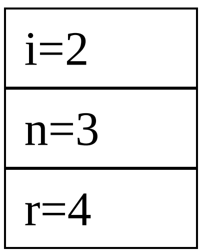
look at bucket h(k)[i ]

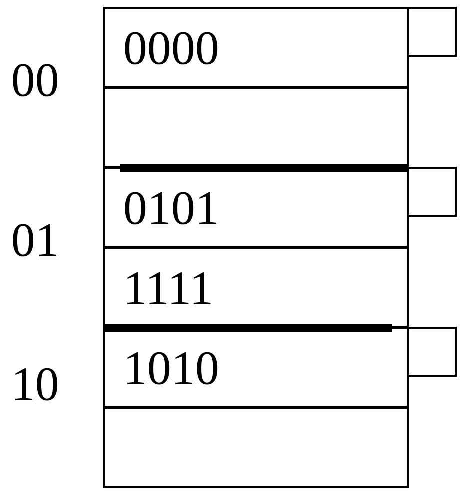
else, look at bucket $h(k)[i\,] - 2^{i-1}$

- Compute h(k), where k is the key of the record
- Determine the correct number of bits at the end of bit sequence to use as the bucket number
  - We put the record either in that bucket or in the bucket with the leading bit changed from 1 to 0.
  - If there is no room in the bucket we create a overflow block.
- Each time we insert
  - Compute ratio r/n
  - If it exceeds predetermined ratio (1.7 in this case), add a new bucket.
- If n exceeds $2^i$, "i" is incremented by one. All existing blocks get "0" in front of their bit sequences.

# Example (contd.)

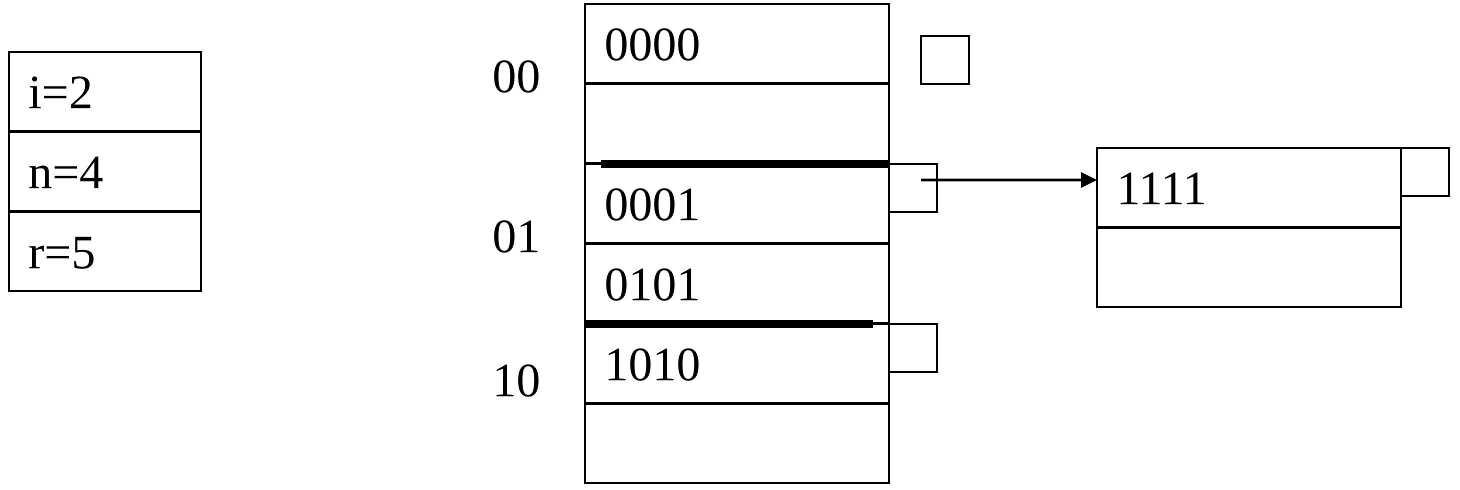
- Add 0101
  - It goes to “1” bucket.
  - However the ration r/n exceeds 1.7, so add another bucket. Increment “i”.



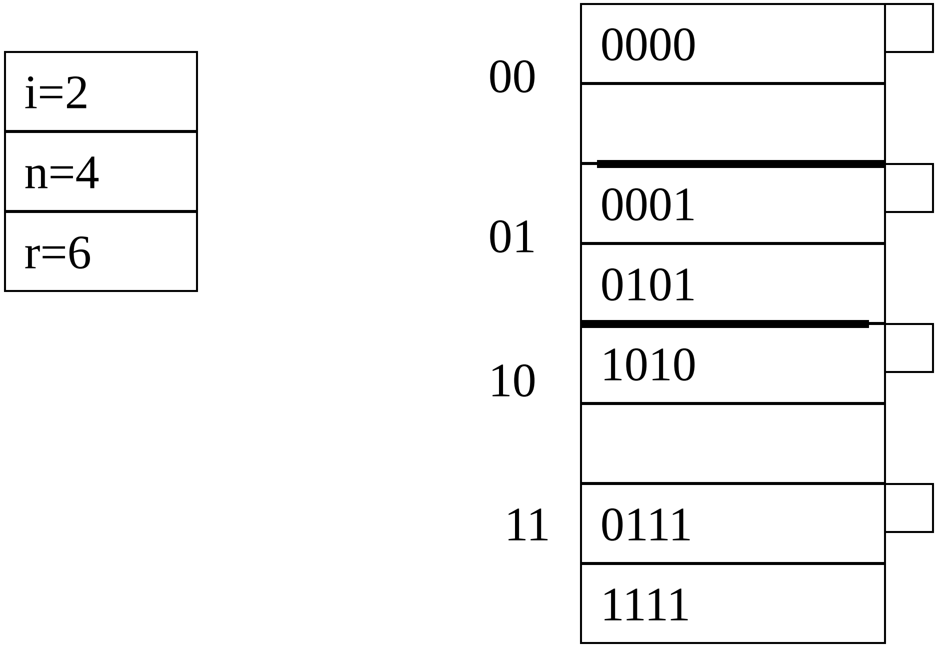
# Example (contd.)

- Add 0001
  - Last two bits are 01
  - However the 01 bucket is full, so add an over flow block.
  - Since ration of r/n is less than 1.7 it is OK



# Example (contd.)

- Add 0111
  - Last two bits are 11
  - The bucket does not yet exist. So redirect this record to 01, whose number differs having a 0 in the first bit. The new record fits in the overflow bucket.
  - Since ration of r/n exceeded 1.7, we have to create a new bucket.




Next time, we insert
a new record, we exceed
The ratio, so "i" should be
Incremented.

## ✉ When do we expand file?

- Keep track of: $\dfrac{\text{\# used slots}}{\text{total \# of slots}} = U$

## ✉ When do we expand file?

- Keep track of: $\frac{\text{\# used slots}}{\text{total \# of slots}} = U$

- If U > threshold then increase *m*
  (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