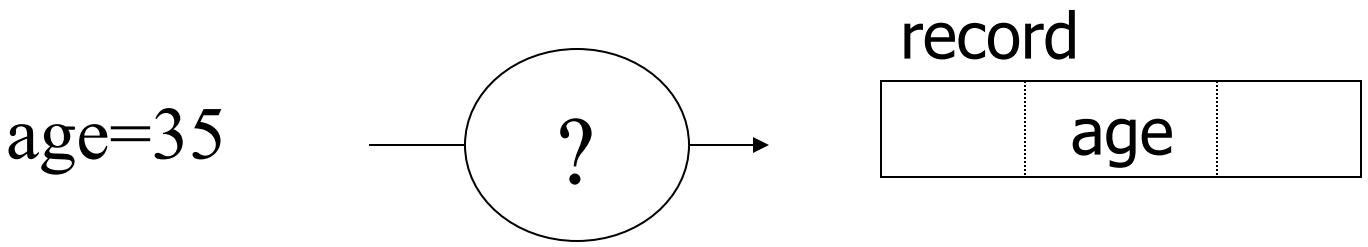
# Data Structures: **Indexing**

# Indexing and Hashing

People(name, age, address, title)

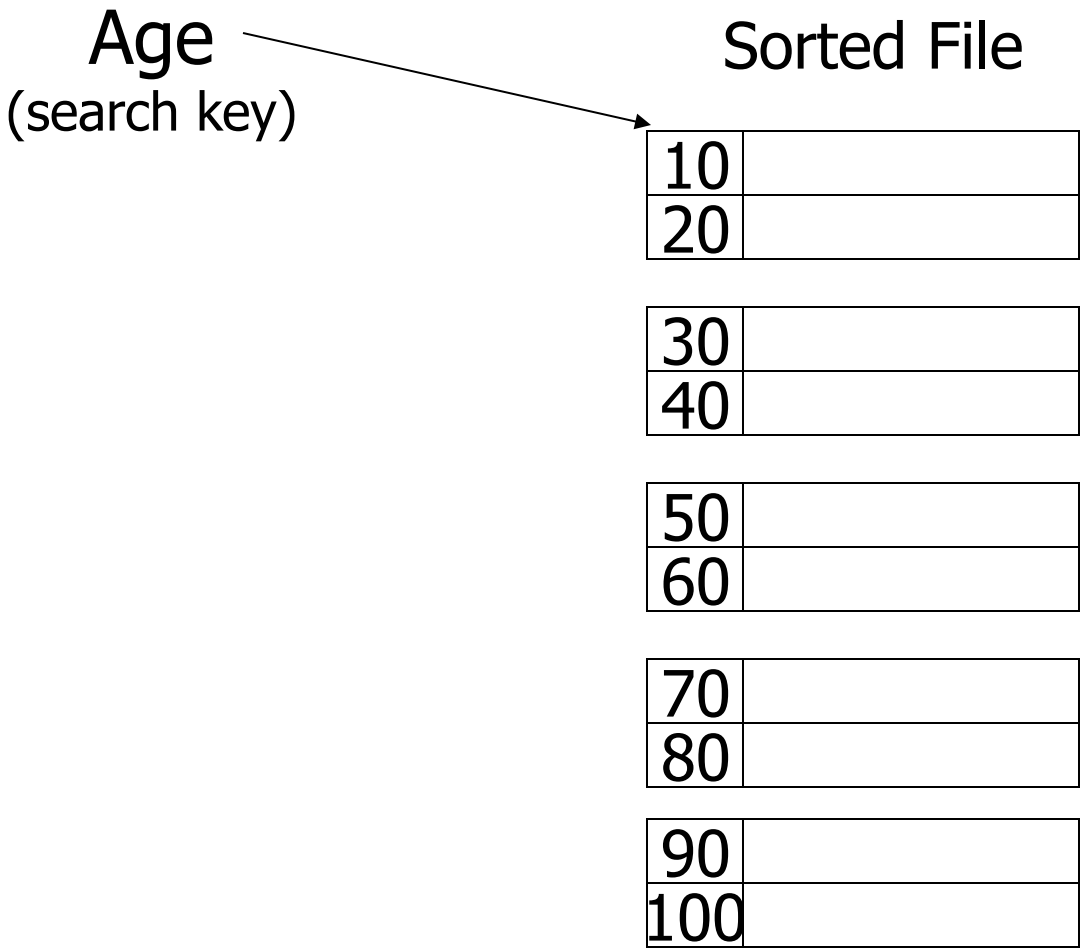
```
SELECT *
FROM   People
WHERE age = 35
```

How to answer the query fast?

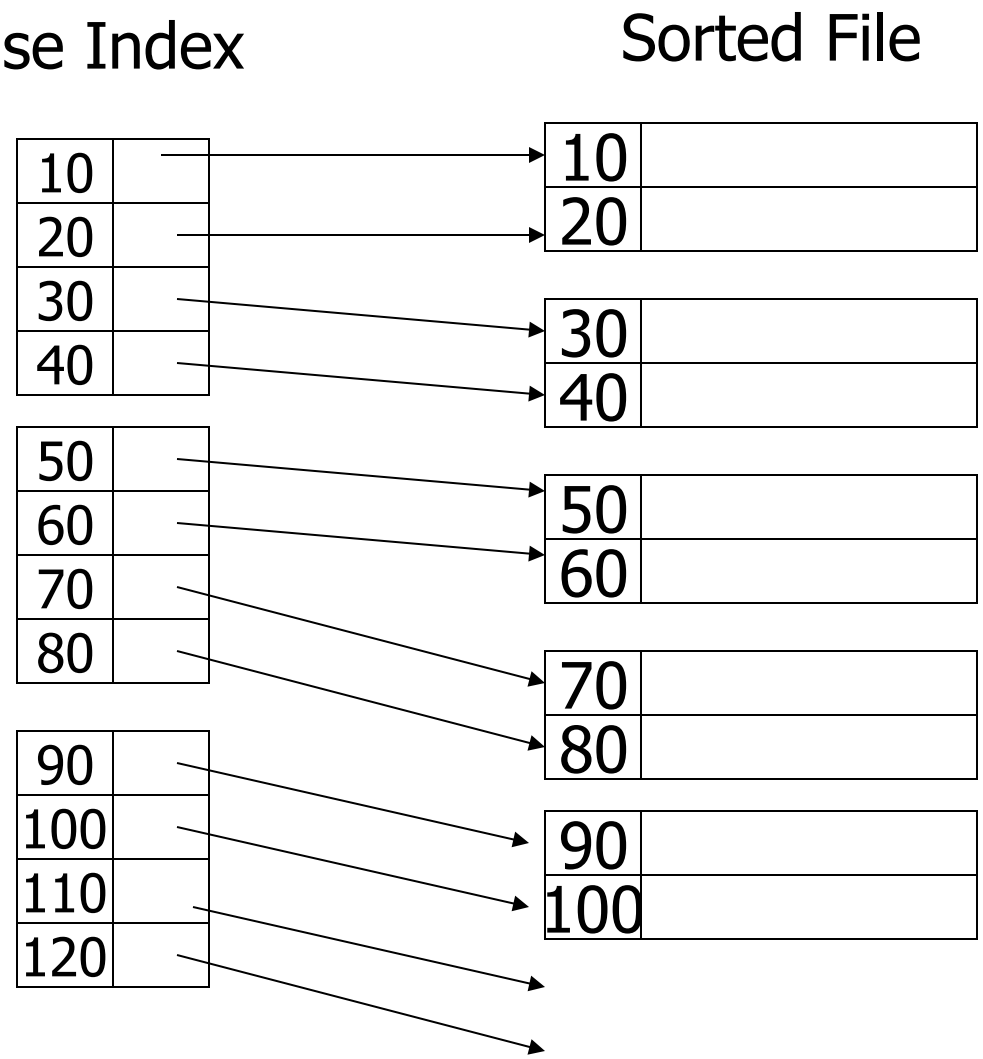


# Outline

- Dense/Sparse Indexes
  - Indexes on Sorted (Sequential) Files
  - Indexes on Unsorted Files (Secondary Indexes)
  - Applications of Secondary Indexes and Buckets
- B Trees
- Hashing
- Multi-dimensional Indexes

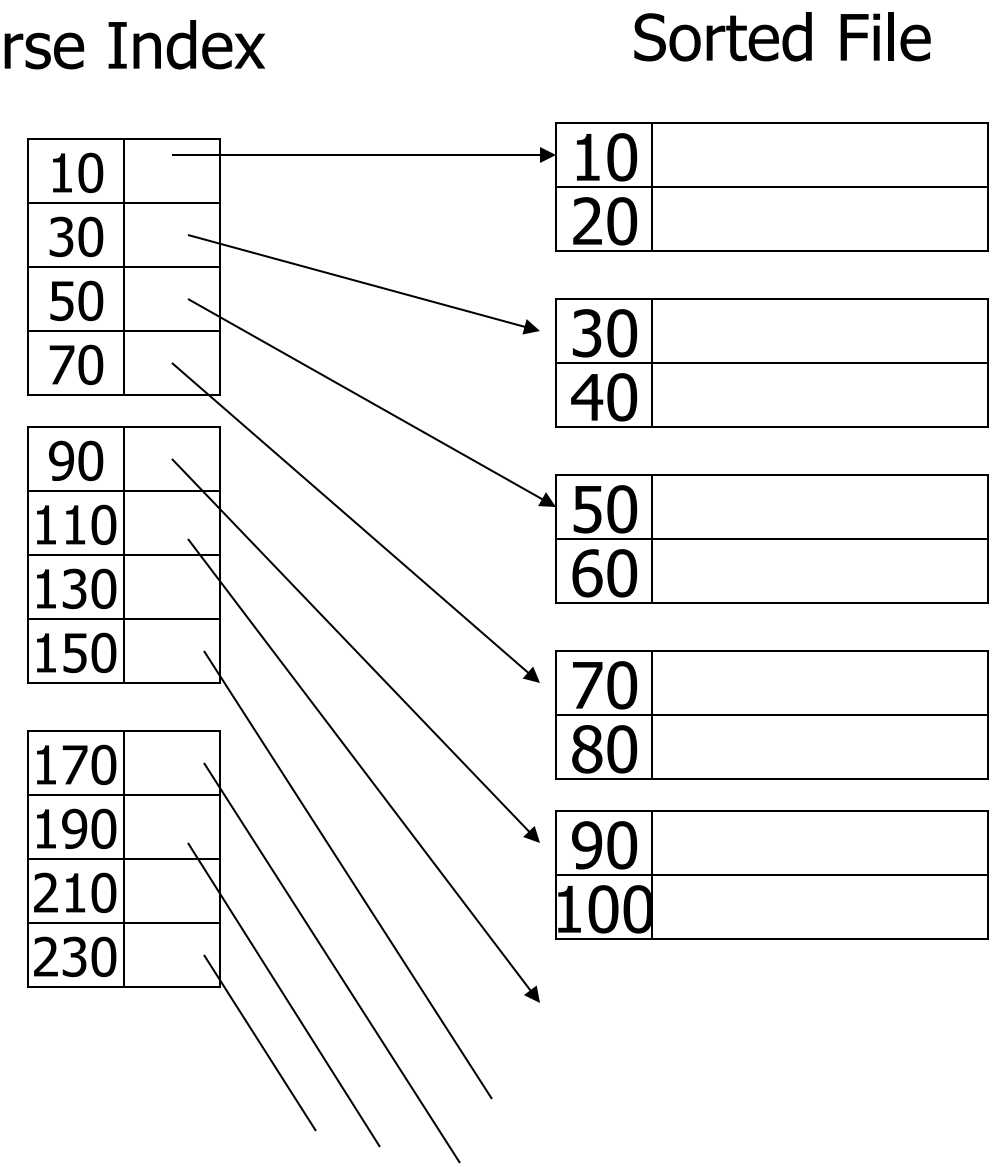

Age
(search key)
Sorted File
10
20
30
40
50
60
70
80
90
100

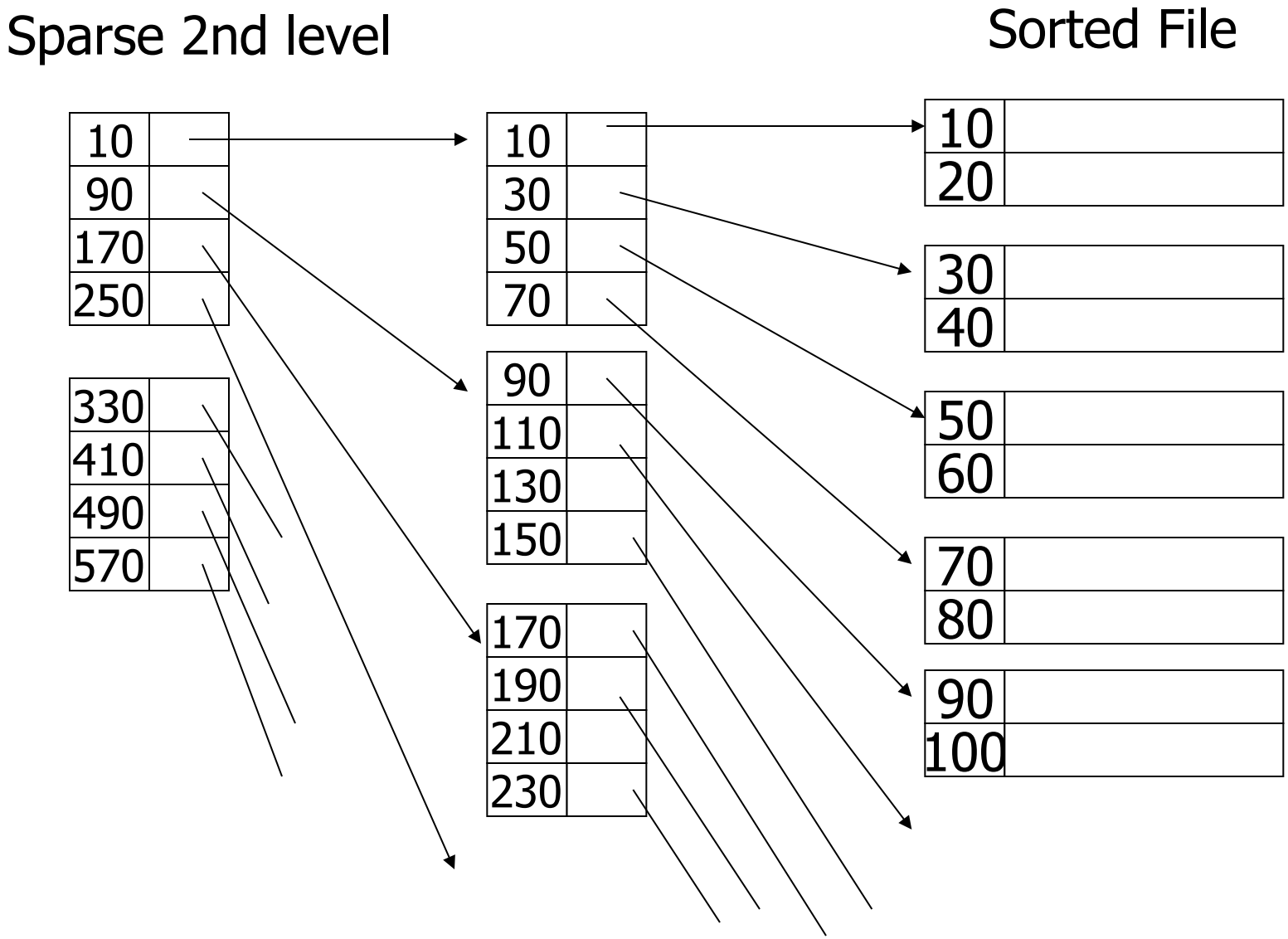
Pointer for each key value (age)




Can do binary search ..

Pointer for the lowest key in each block




Sparse 2nd level
Sorted File
10
90
170
250
330
410
490
570
10
30
50
70
90
110
130
150
170
190
210
230
10
20
30
40
50
60
70
80
90
100

# Dense/Sparse Index Summary

- Dense: Store pointer to each record/key value.
  - Can tell if a record exists without accessing the file
  - Efficient access
  - Needed for secondary indexes (as we'll see later)

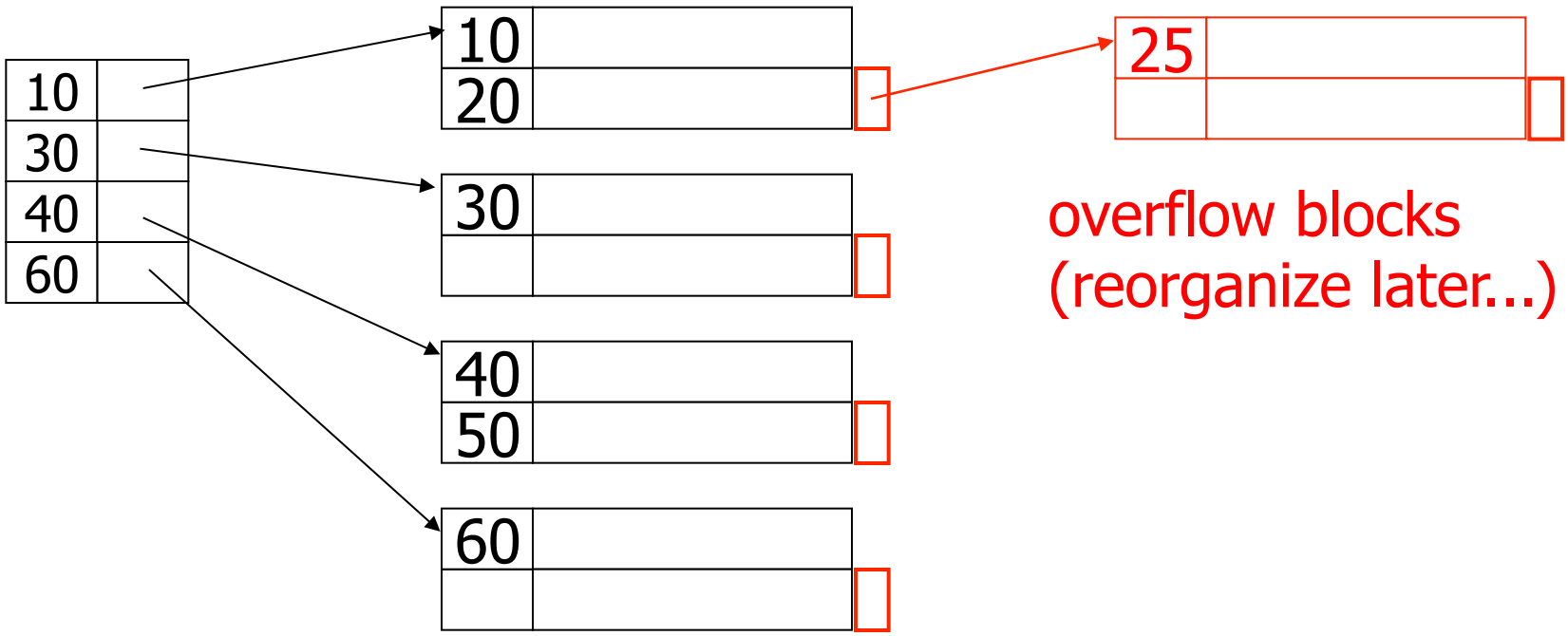
- Sparse: Store pointer to the lowest key in each block.
  - Less index space. May fit in memory.
  - Better for insertions/deletions

# Insertions/Deletions in Sorted Files

- Many options:
  - Leave extra space in data and/or index blocks to accommodate expansion
  - Overflow blocks (next 2 slides)
  - Localized reorganization (moving records within or among blocks)
  - Complete reorganization (done once in a while, in conjunction with above techniques)

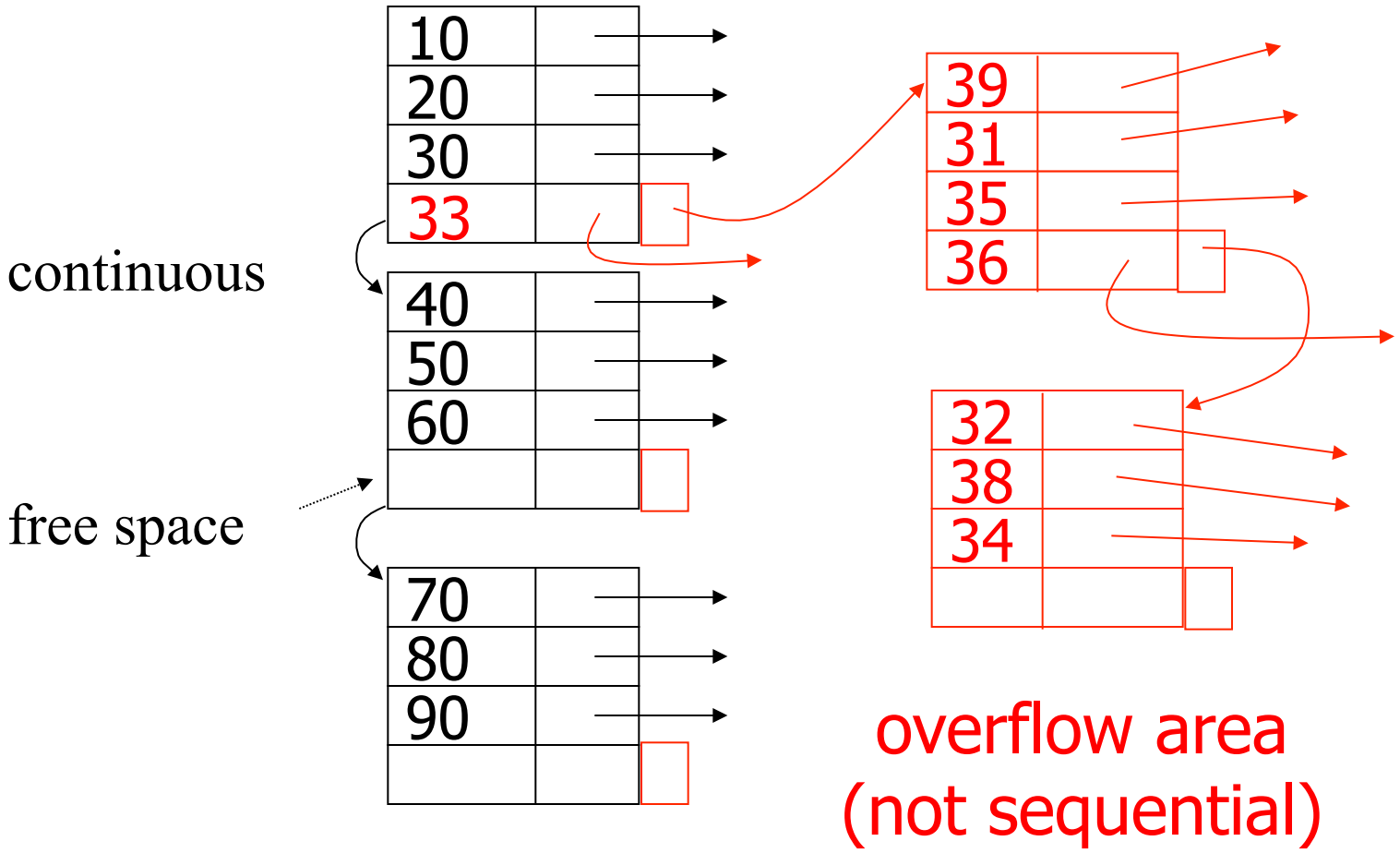
# Overflow Data Blocks

– insert record 25



# Overflow Index Blocks

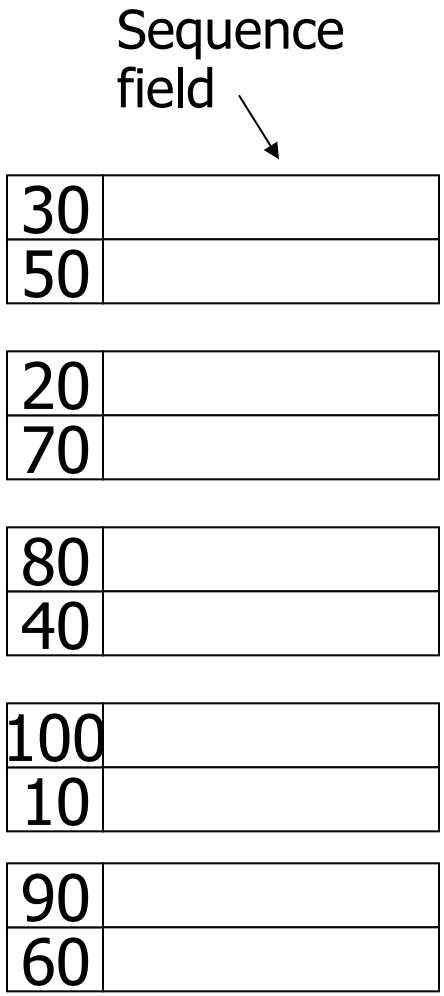
Index (sequential)



# Outline

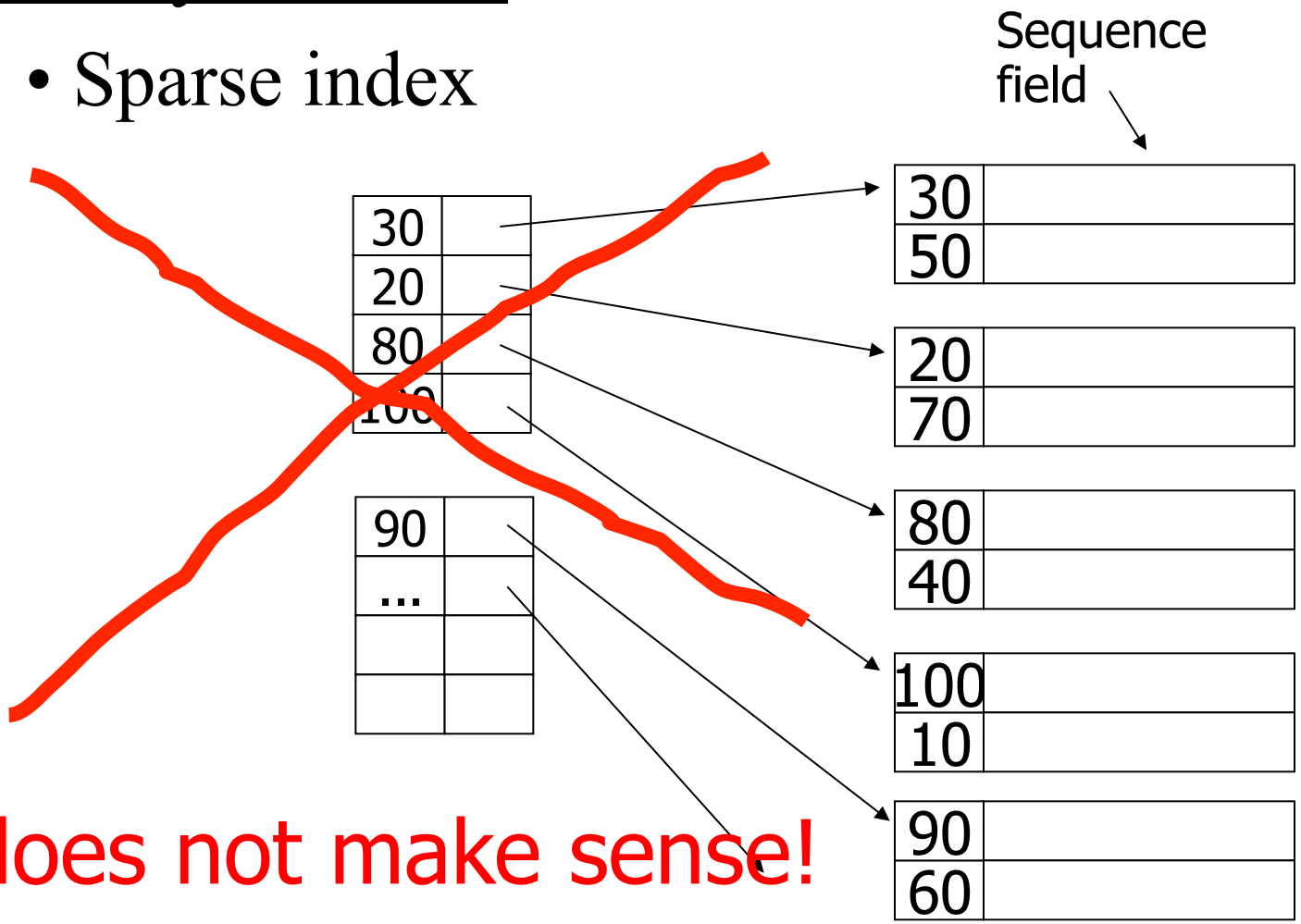
- Dense/Sparse Indexes
  - Indexes on Sorted (Sequential) Files
  - Indexes on Unsorted Files (Secondary Indexes)
  - Applications of Secondary Indexes and Buckets
- B Trees
- Hashing
- Multi-dimensional Indexes

# Secondary indexes



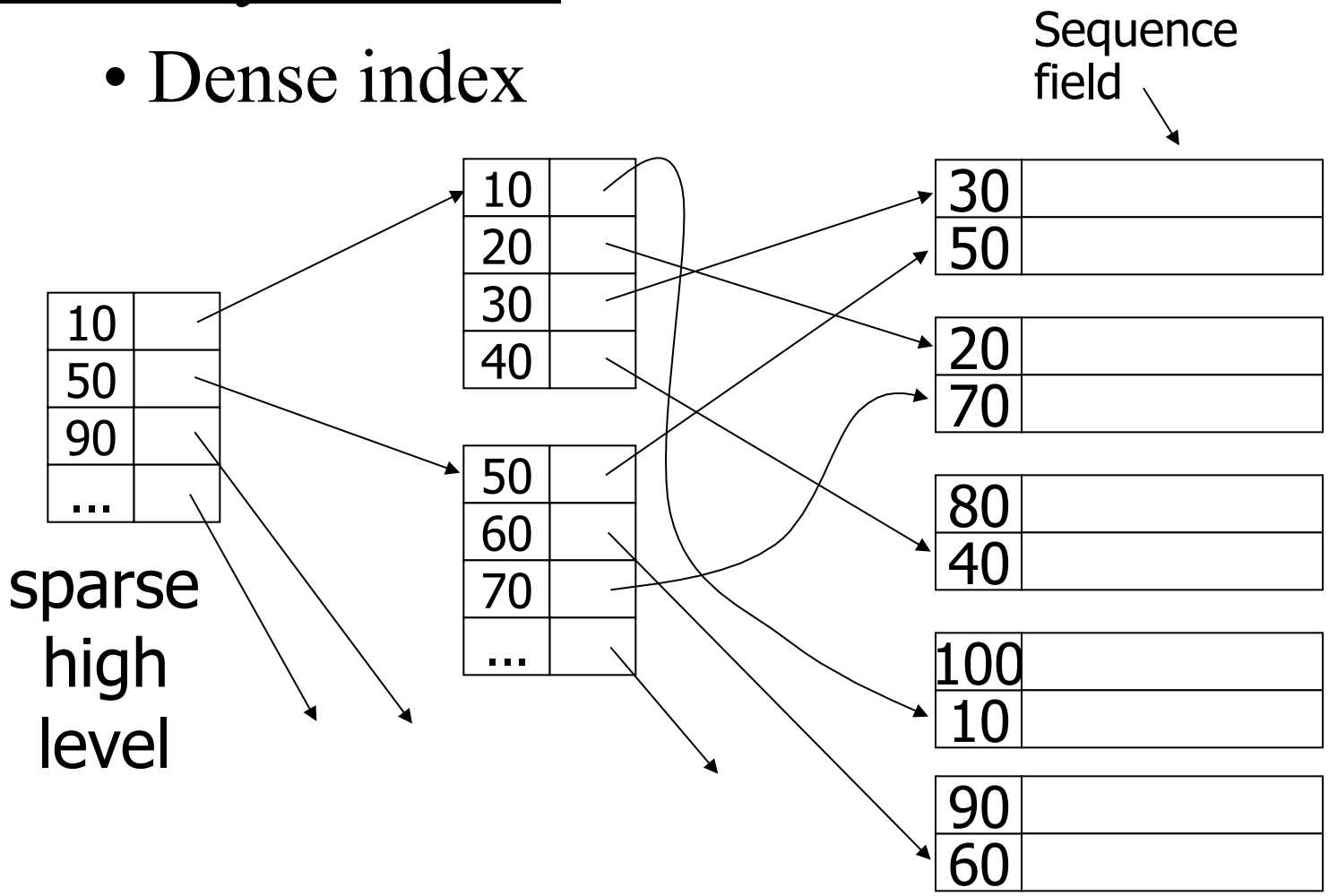
# Secondary indexes

- Sparse index



# Secondary indexes

- Dense index



# With secondary indexes:

- Lowest level has to be dense
- Other levels are sparse

# Duplicate values?

| | |
|---|---|
| 20 | |
| 10 | |

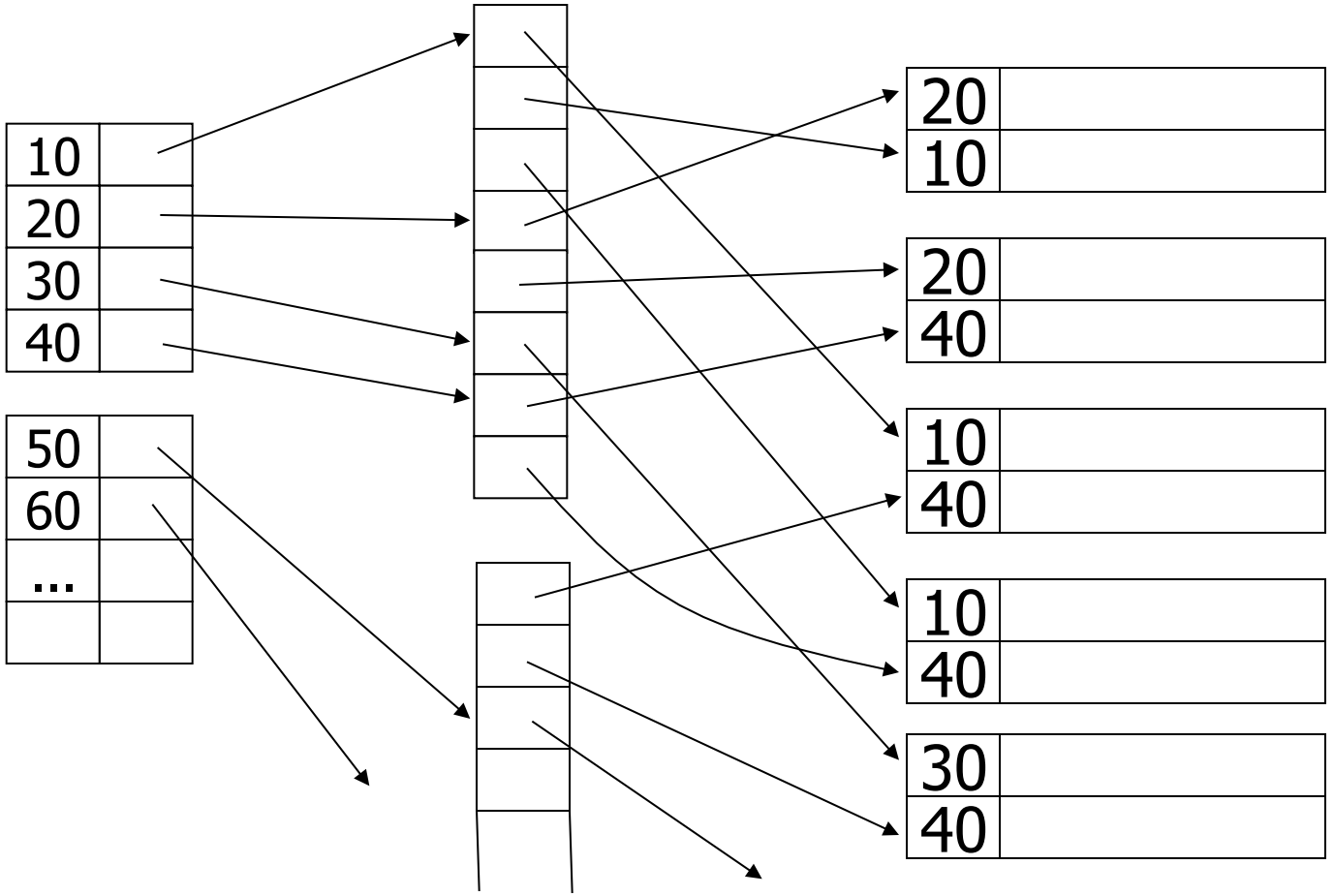
| | |
|---|---|
| 20 | |
| 40 | |

| | |
|---|---|
| 10 | |
| 40 | |

| | |
|---|---|
| 10 | |
| 40 | |

| | |
|---|---|
| 30 | |
| 40 | |

# Duplicate values: Use "buckets"




buckets

# Outline

- Dense/Sparse Indexes
  - Indexes on Sorted (Sequential) Files
  - Indexes on Unsorted Files (Secondary Indexes)
  - Applications of Secondary Indexes and Buckets
- B Trees
- Hashing
- Multi-dimensional Indexes

# Use-case for Secondary Indexes: Clustered Files

Movie(title, year, studioName, …, …)

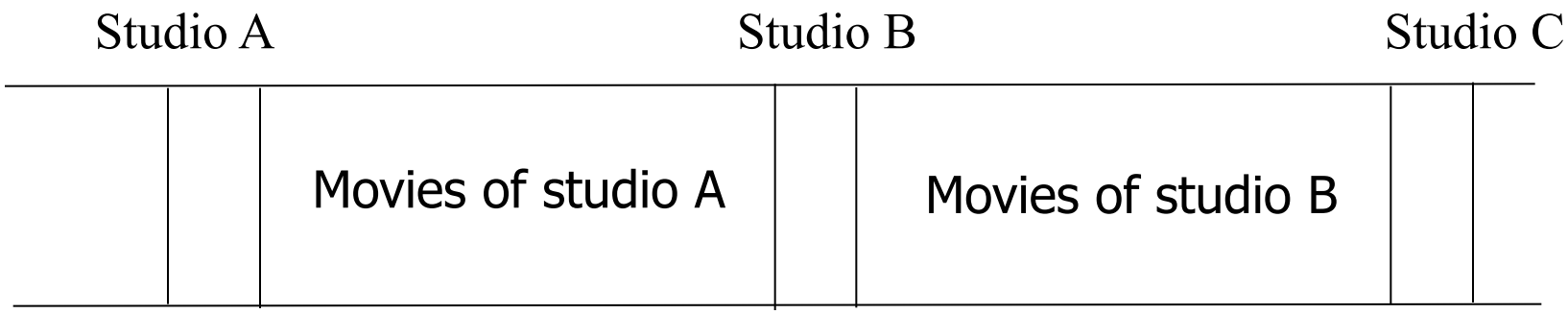
Studio(name, address, president)

Typical Query

```
Select    title, year
From      Movie JOIN Studio
Where     president = "xxx"
```

# Clustered Files

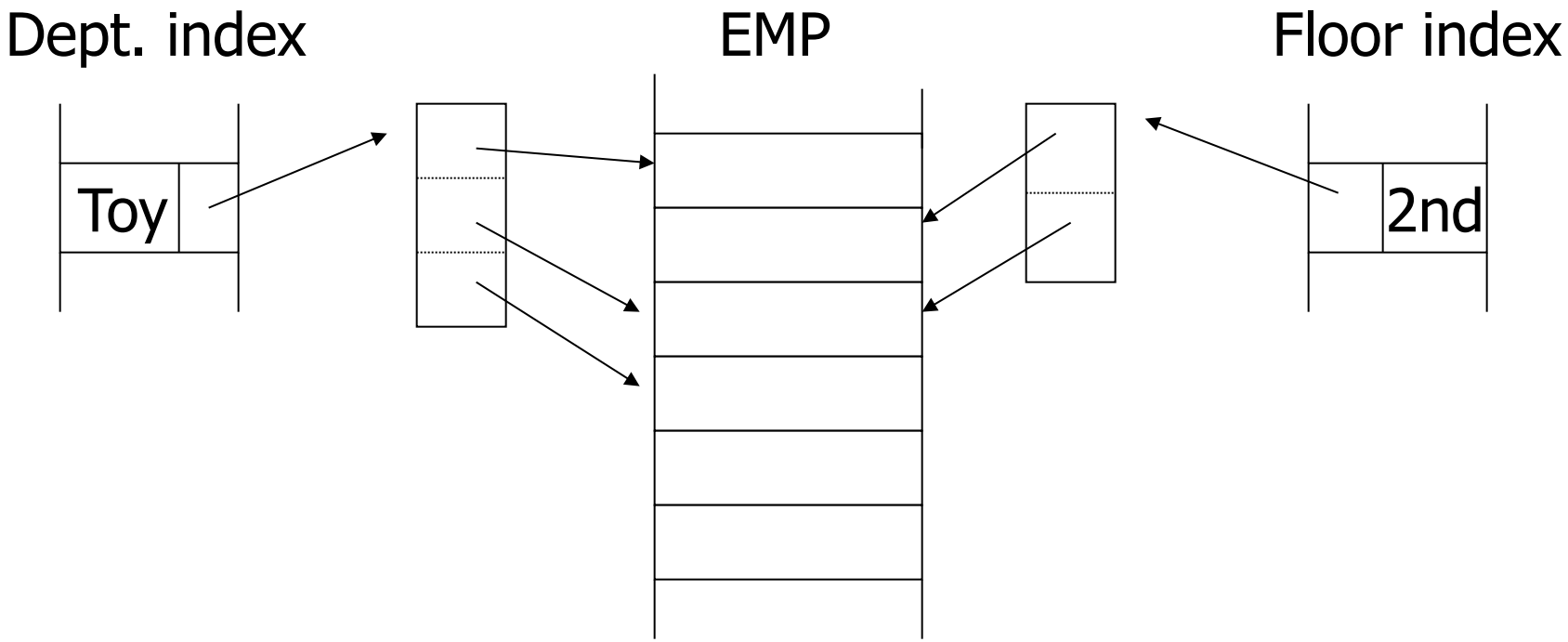
- Previous example:
  - Efficient to store movies with its studio
- Answering the Query:
  - Find studio record using an index on ‘president’
  - Find movies stored with the studio record found
  - However, index on movies must be secondary.



# Applications of Buckets

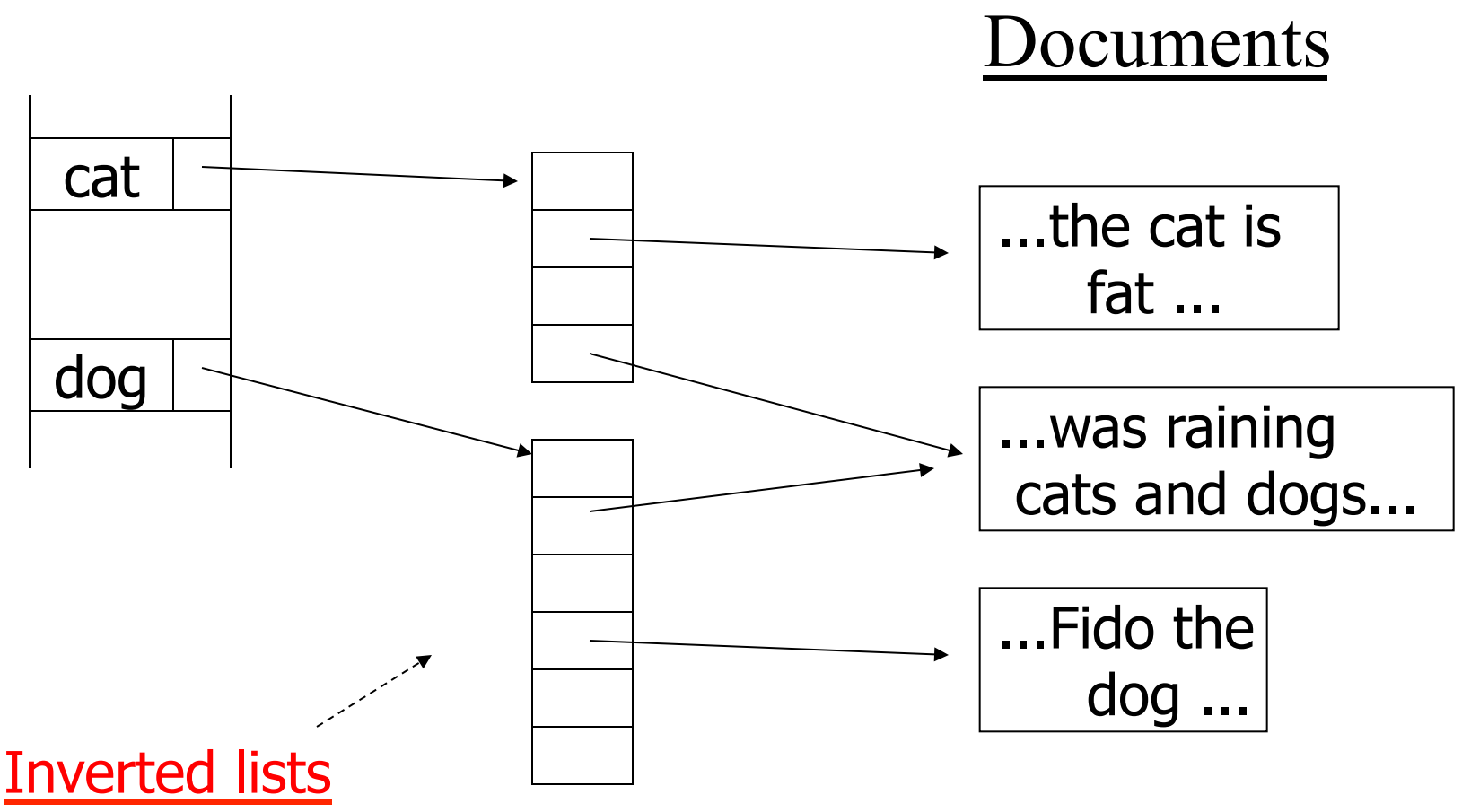
| Indexes | Table |
| --- | --- |
| Name: primary | EMP (name,dept,floor,...) |
| Dept: secondary | |
| Floor: secondary | |

# Query: Get employees in (Toy Dept) ∧ (2nd floor)




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

# Inverted Lists in Information Retrieval



# IR QUERIES

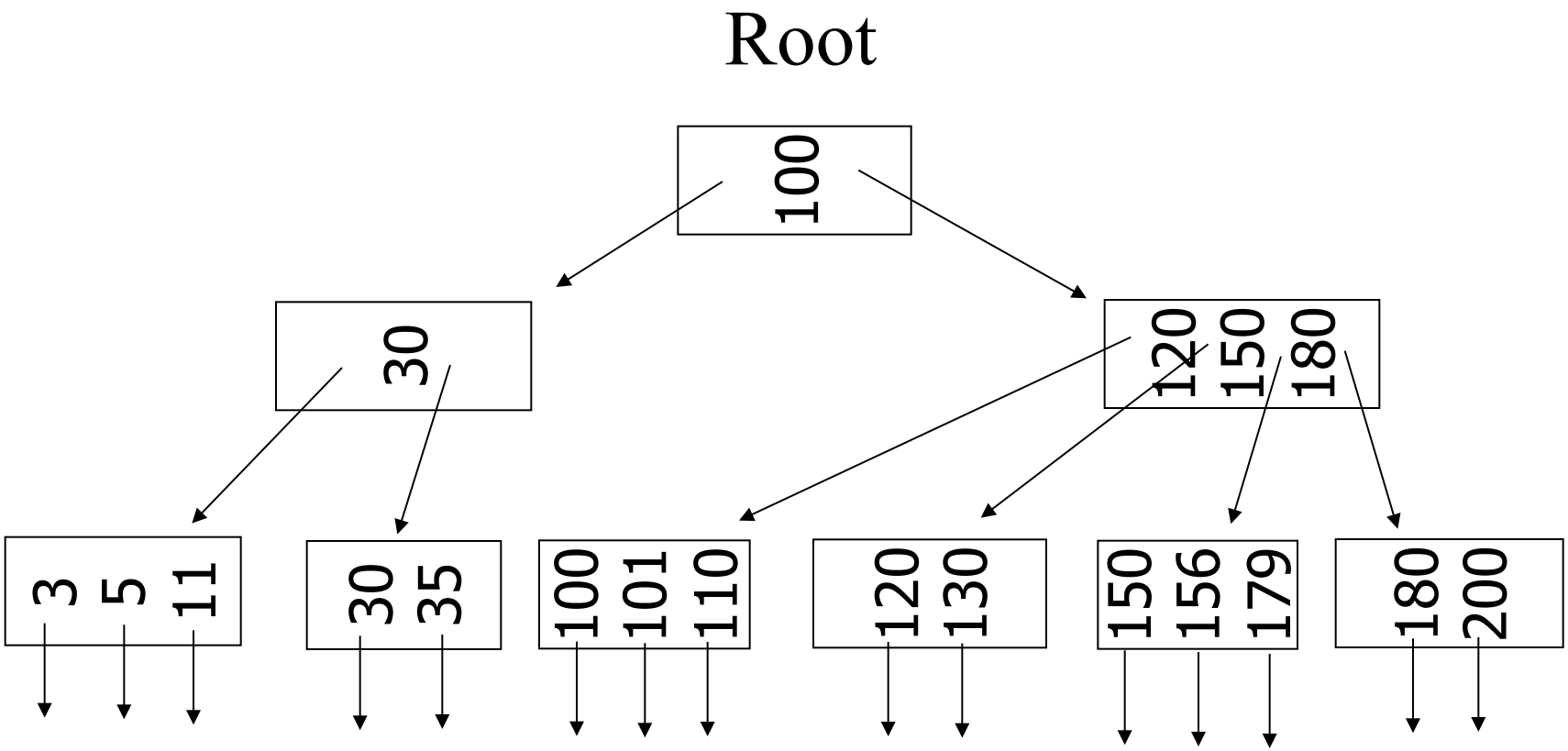
- Find articles with “cat” and “dog”
- Find articles with “cat” or “dog”
- Find articles with “cat” and not “dog”

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

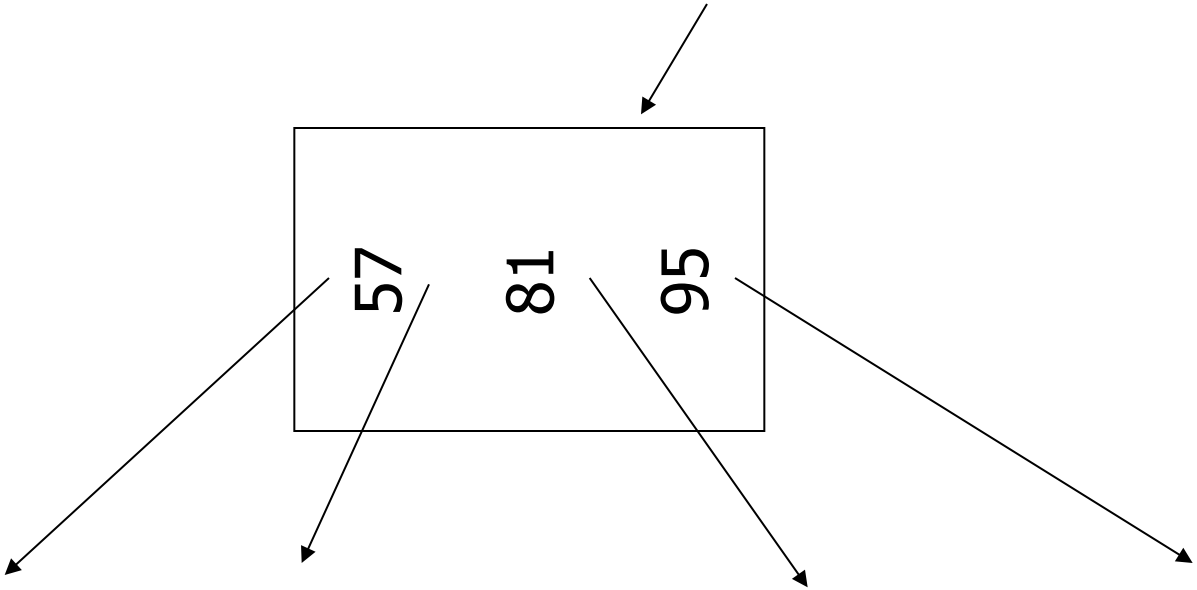
# Outline

- Dense/Sparse Indexes
  - Indexes on Sorted (Sequential) Files
  - Indexes on Unsorted Files (Secondary Indexes)
  - Applications of Secondary Indexes and Buckets
- B Trees
- Hashing
- Multi-dimensional Indexes

# B Tree Example

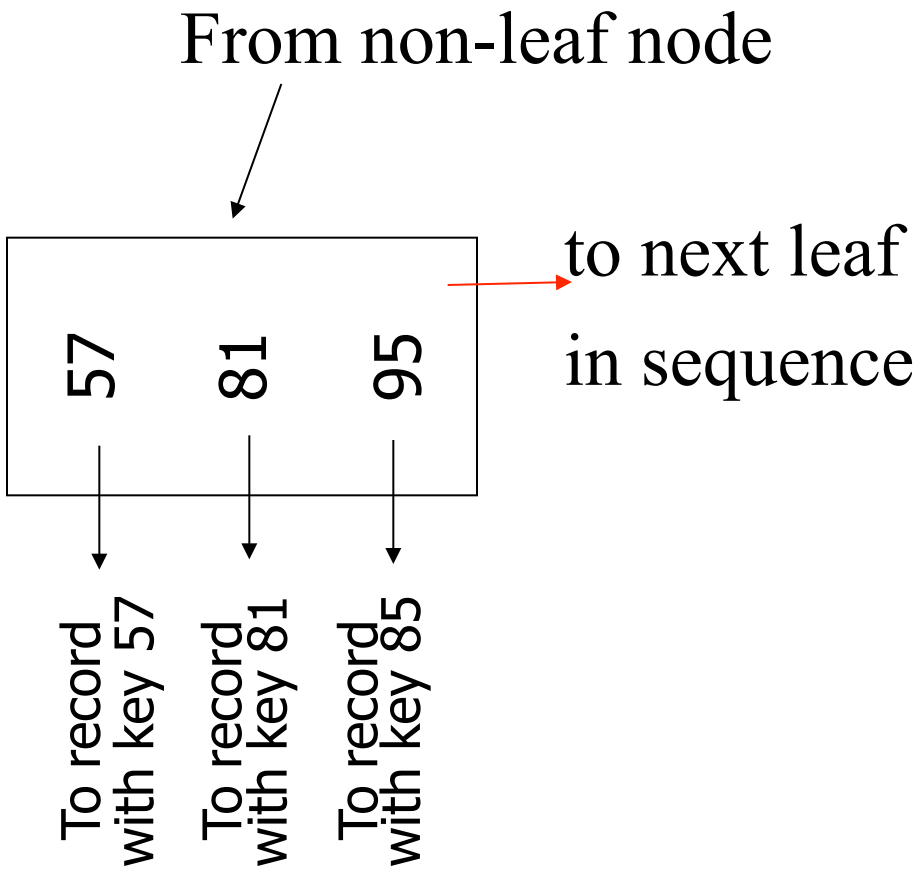


# Sample non-leaf




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

# Sample leaf node



## Main Challenge in B Trees

- To ensure good search cost, the tree should be kept of minimum and uniform height. I.e., keep the tree:
  - full (packed), and
  - balanced (almost uniform height).
- This can be difficult -- in face of insertions and deletions.

Solution: Keep the tree "semi-full" (i.e., each tree node half-full).

  - Makes it easy to keep the tree balanced and semi-optimal.

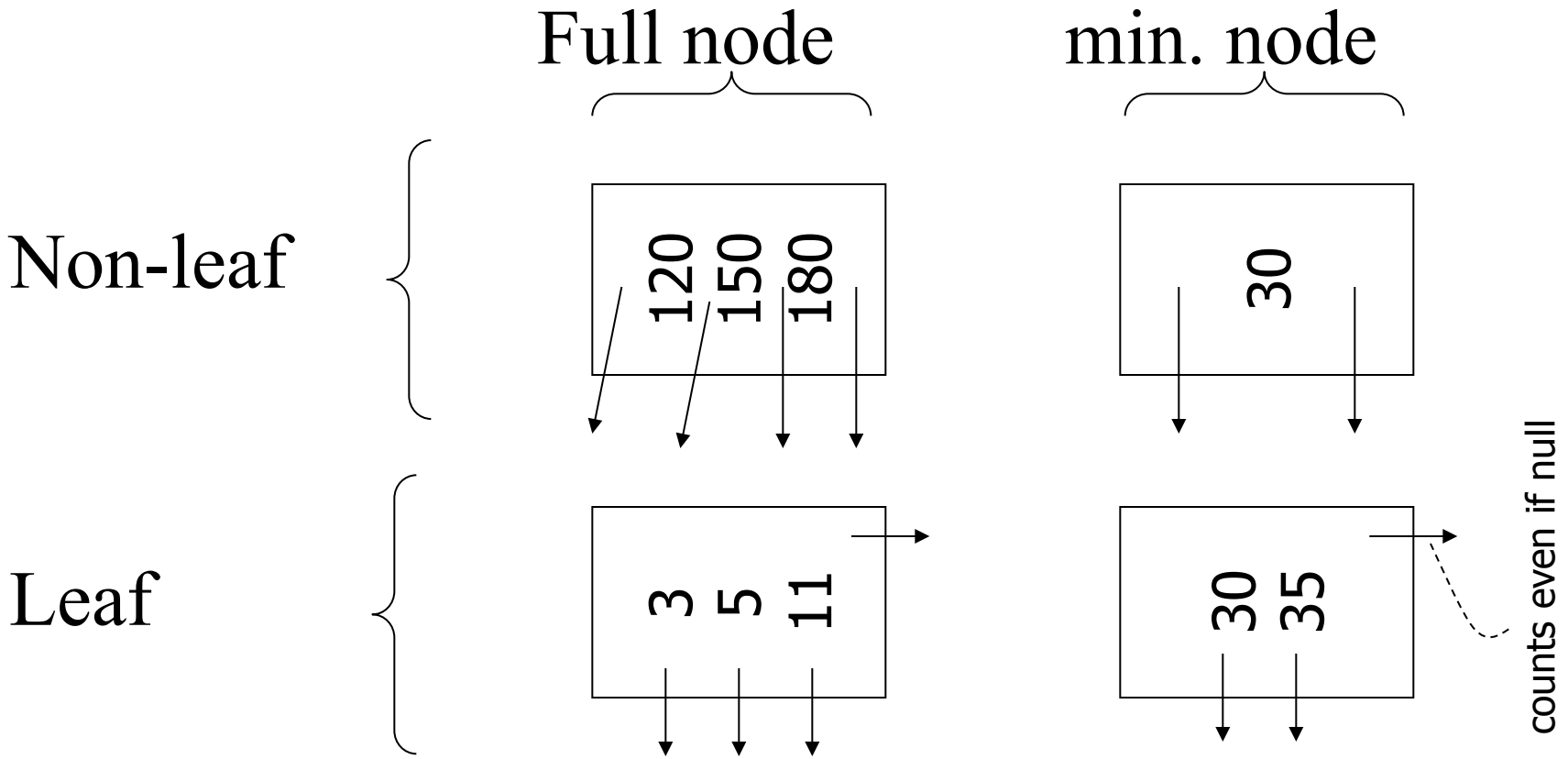
## Size of nodes

$$\begin{cases} n+1 & \text{pointers} \\ n & \text{keys} \end{cases}$$ (fixed)

## Use at least (to ensure a balanced tree)

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

# Insertions into B tree

(a) simple case
  - space available in leaf

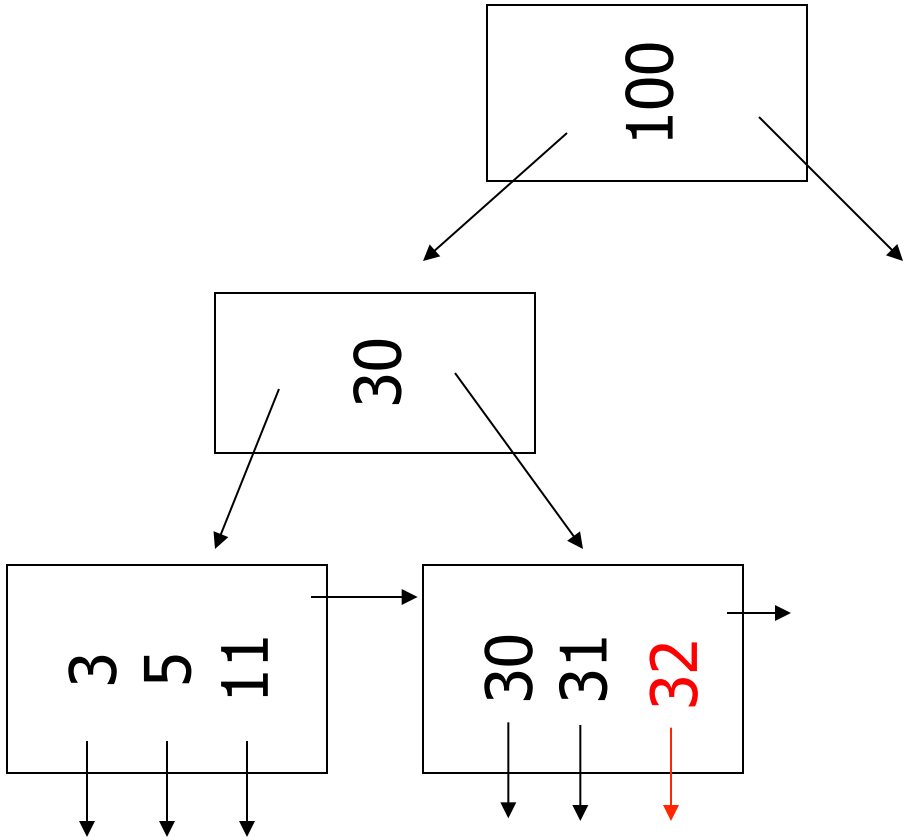
(b) leaf overflow

(c) non-leaf overflow

(d) new root

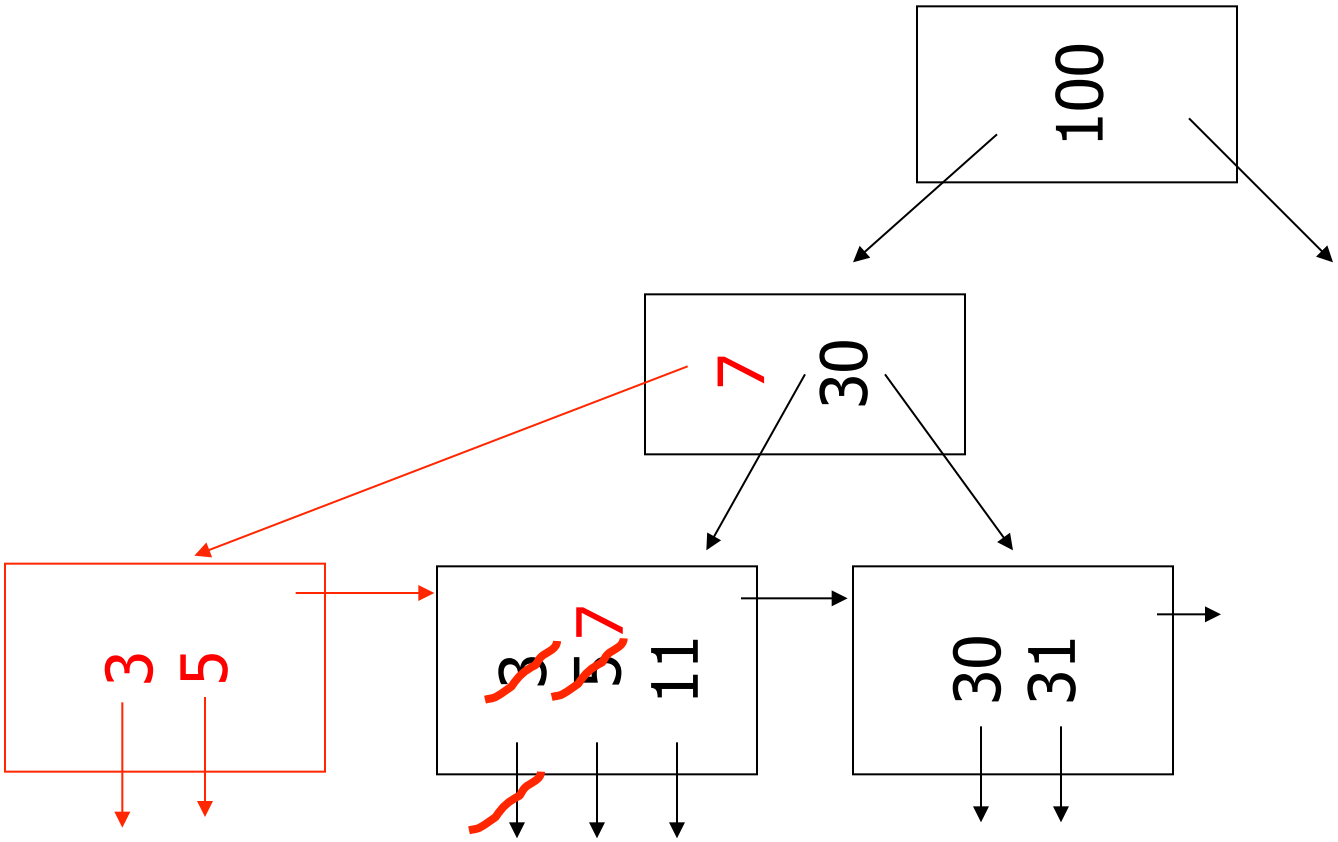
# (a) Insert key = 32

n=3



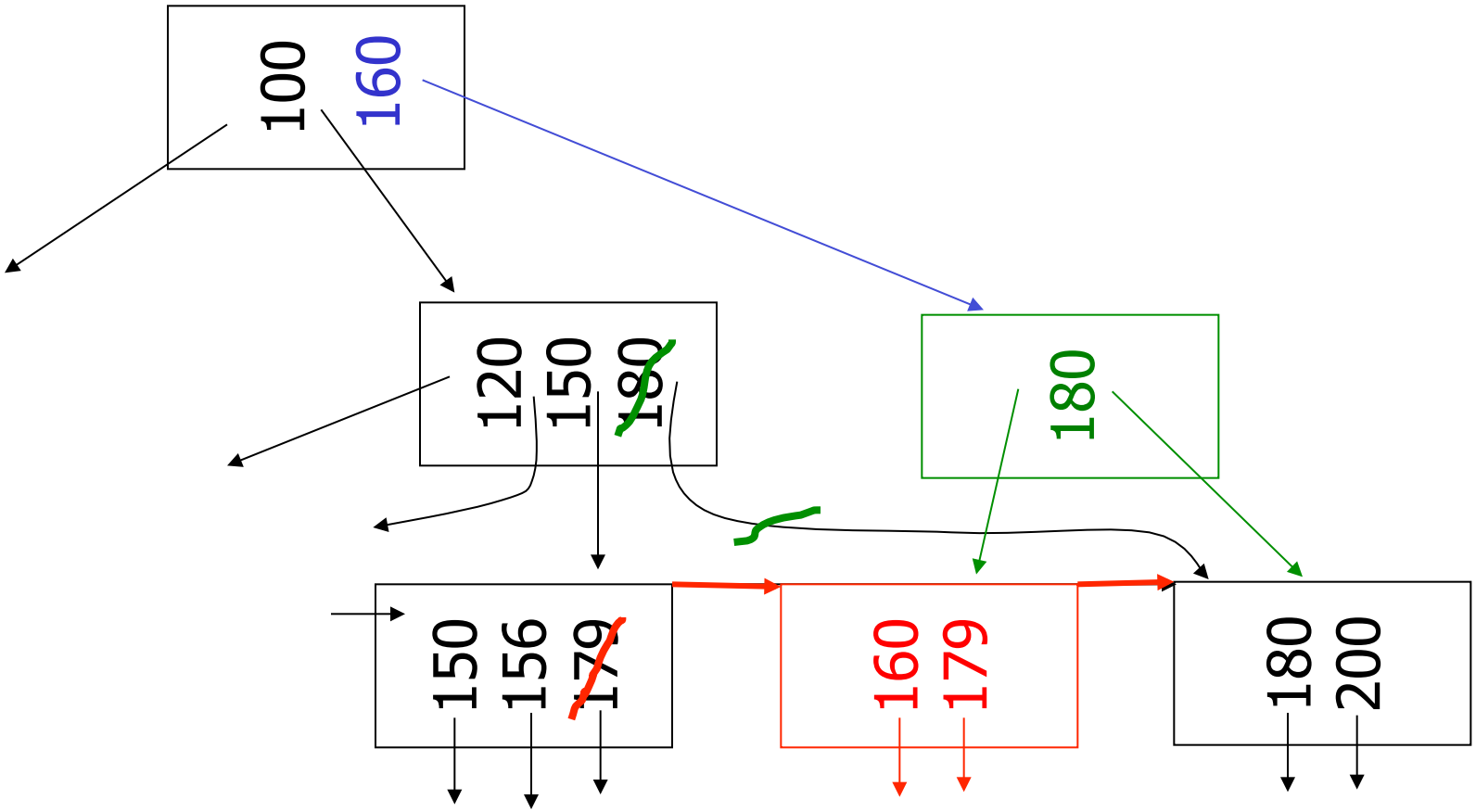
# (b) Leaf Overflow: Insert key = 7

n=3



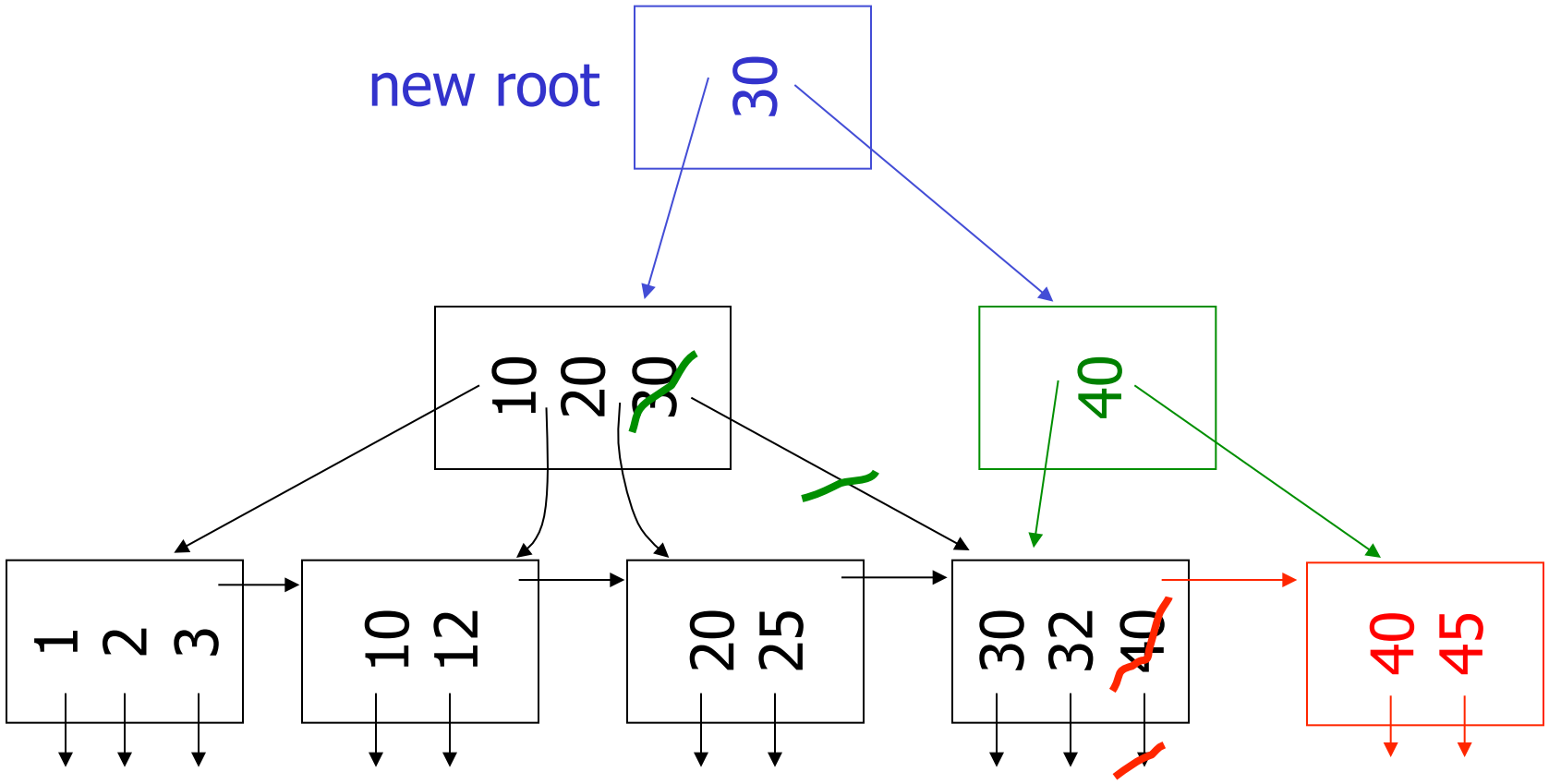
# (c) Non-leaf Overflow: Insert key = 160

n=3



# (d) New root: Insert 45

n=3



# Deletion from B tree

(a) Simple case - no example

(b) Coalesce with neighbor (sibling)

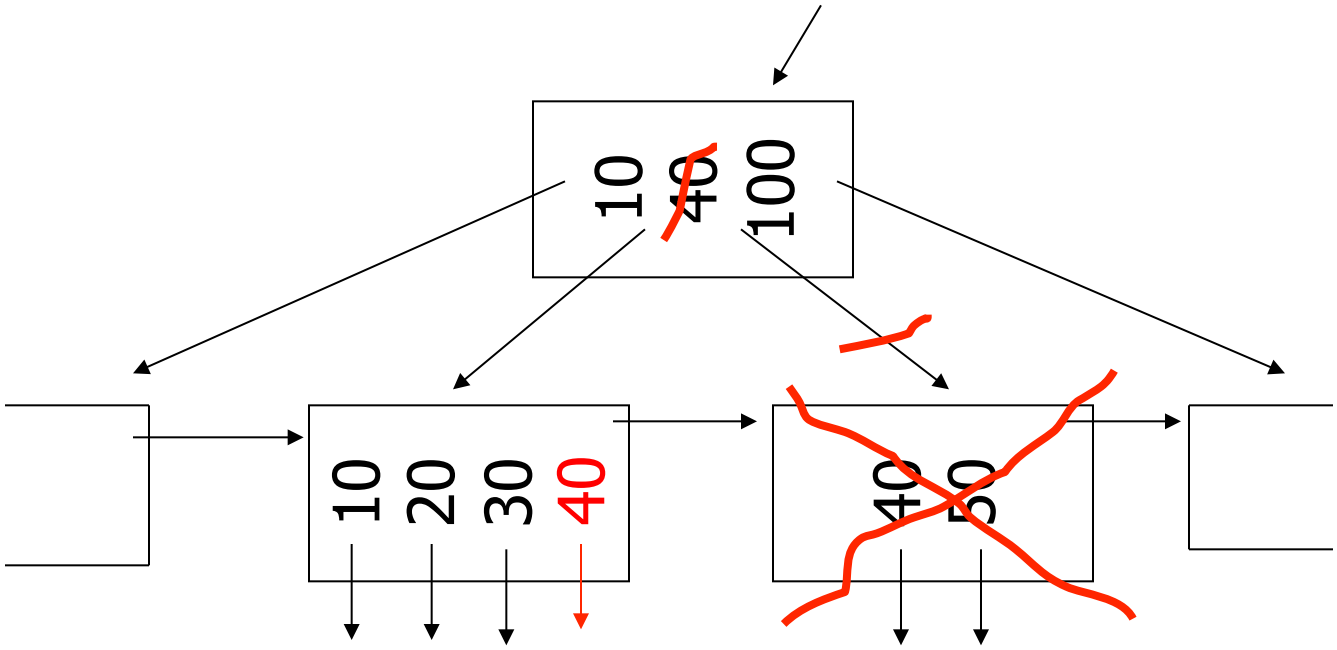
(c) Re-distribute keys

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

# (b) Coalesce with sibling

– Delete 50

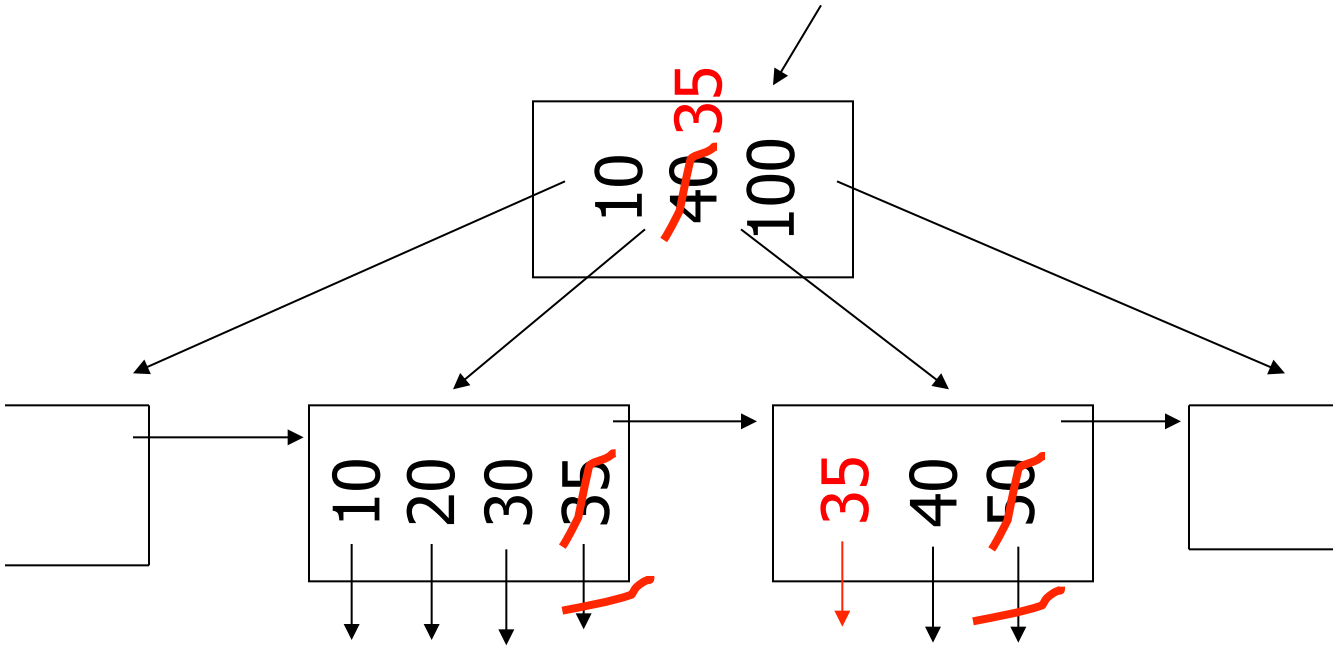
n=4



# (c) Redistribute keys

– Delete 50

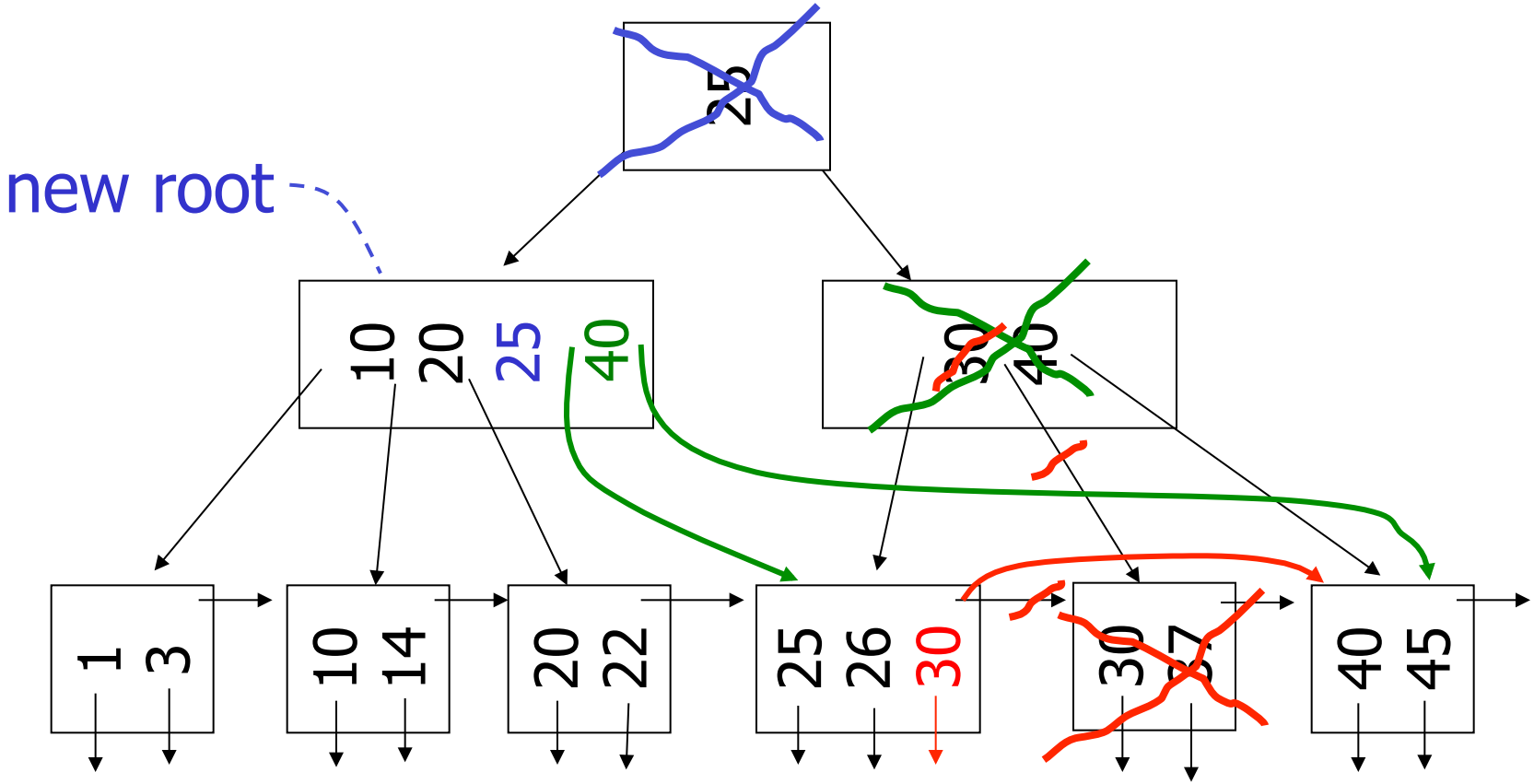
n=4



# (d) Non-leaf coalese

– Delete 37

n=4



# B-tree deletions in practice

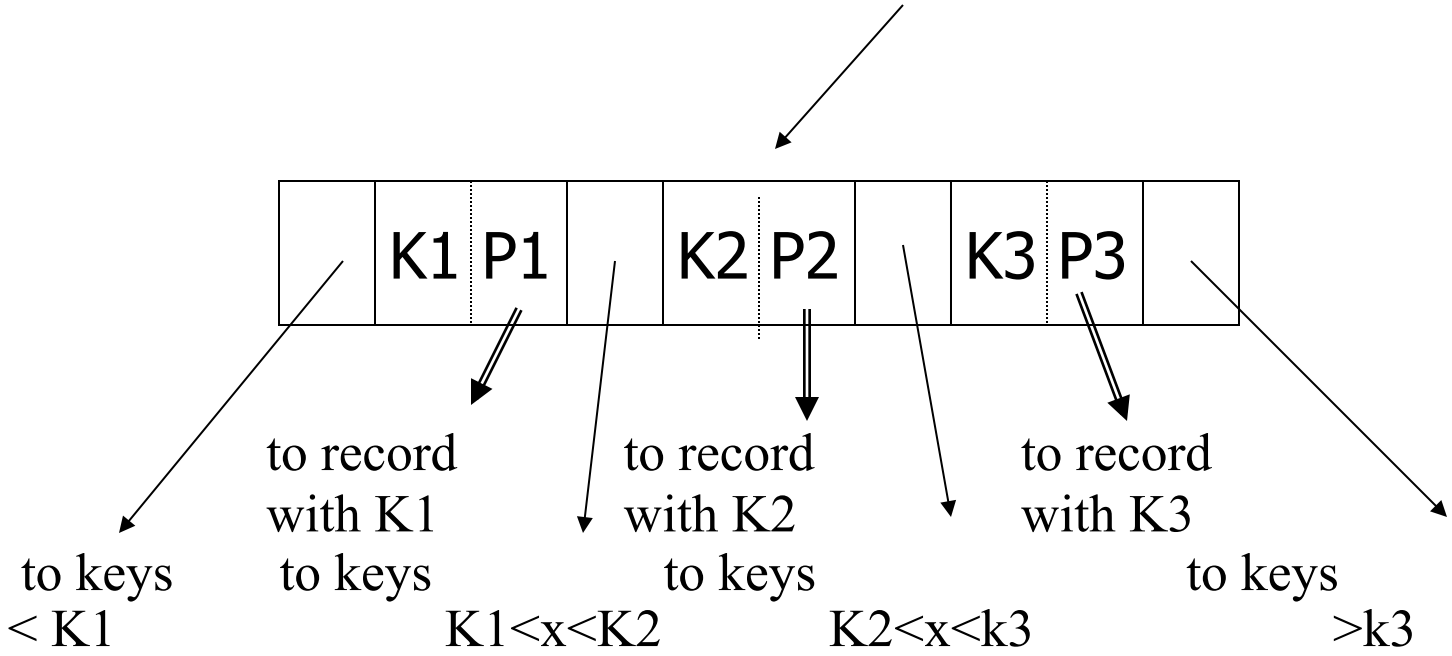
- Often, coalescing is not implemented
  - Too hard and not worth it!

# B-Trees vs Indexes on Sorted Files

- B Tree Advantages:
  - Fast inserts and deletes (splits/merges are rare) at the cost of sub-optimal number of levels.
  - But, in most practical cases – the number of levels is still only 3-4.

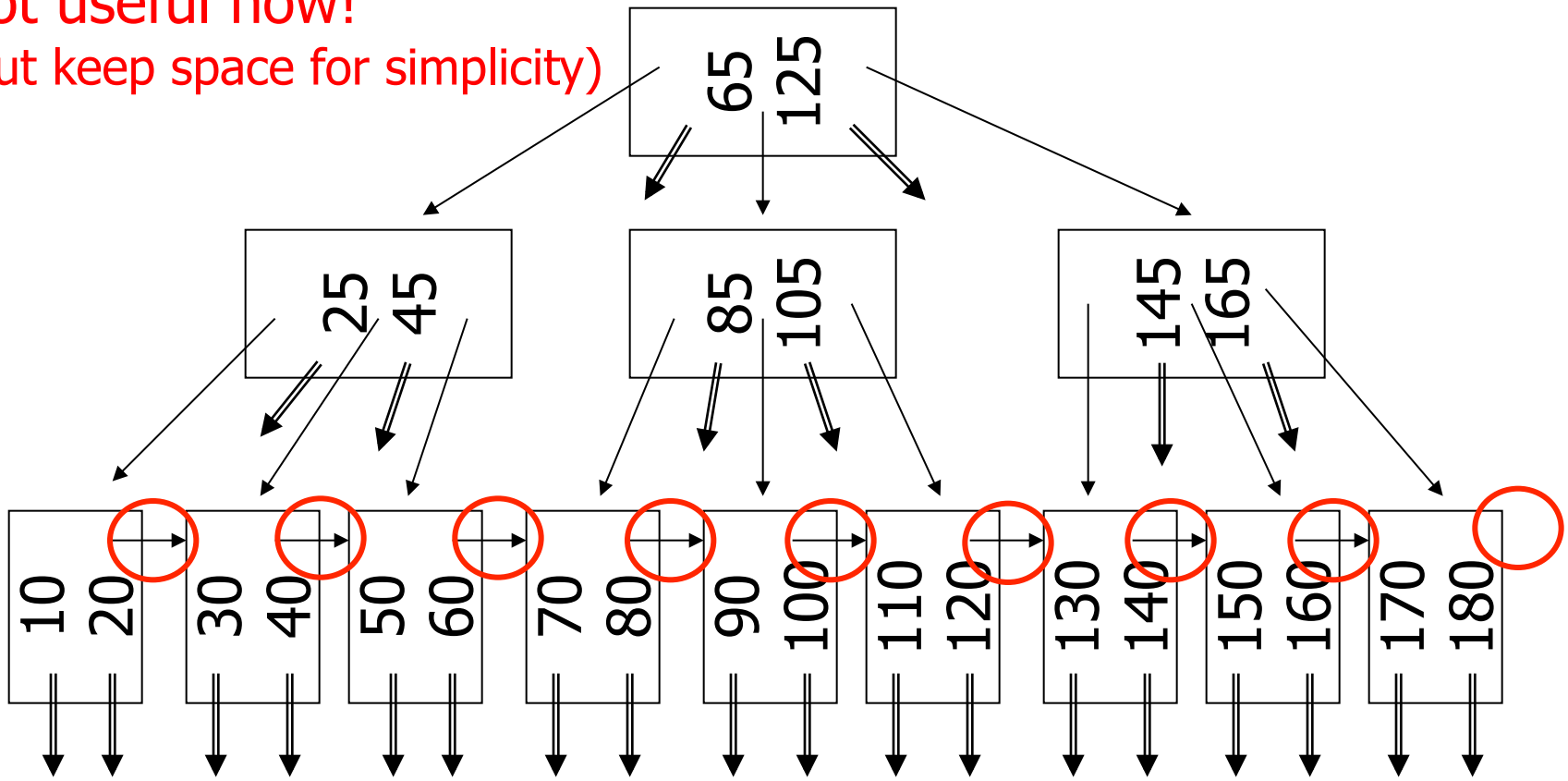
# Another Variant of B-Trees

- What we have discussed till now are actually called “B+ trees”.
- The original B-Tree (without the +):
  - Avoids key values being stored multiple times
  - Has pointers to records in non-leaf nodes also


K1
P1
K2
P2
K3
P3
to record with K1
to record with K2
to record with K3
to keys < K1
to keys K1<x<K2
to keys K2<x<k3
to keys >k3

# B-tree Variant Example (n=2)

- sequence pointers not useful now!
  (but keep space for simplicity)



# Pros and Cons of the Variant

Pros

- Has faster lookup

Cons

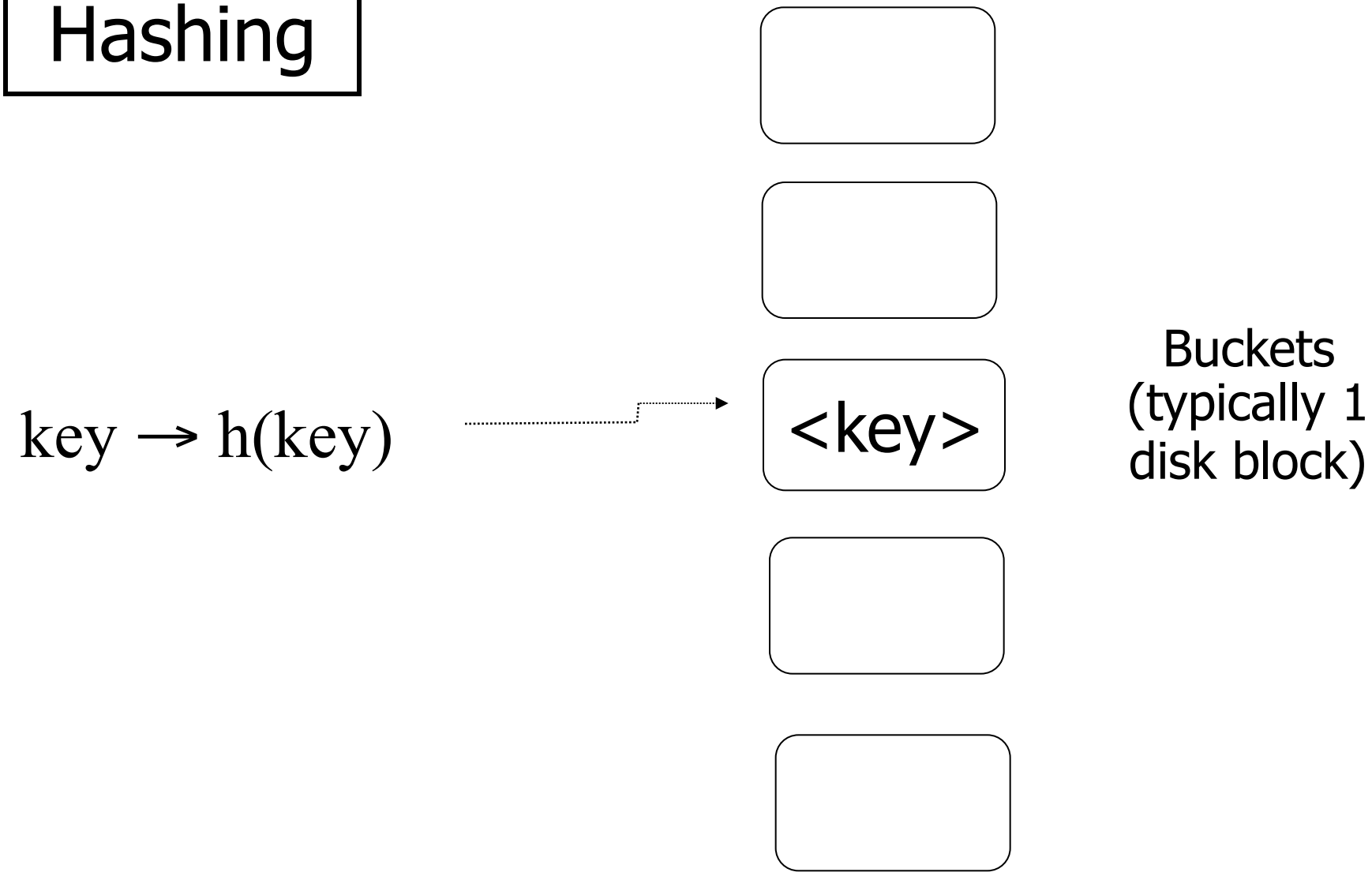
- Non-leaf and leaf has different sizes
- For fixed block size, less “bushier”

➡ Not preferred

# Outline

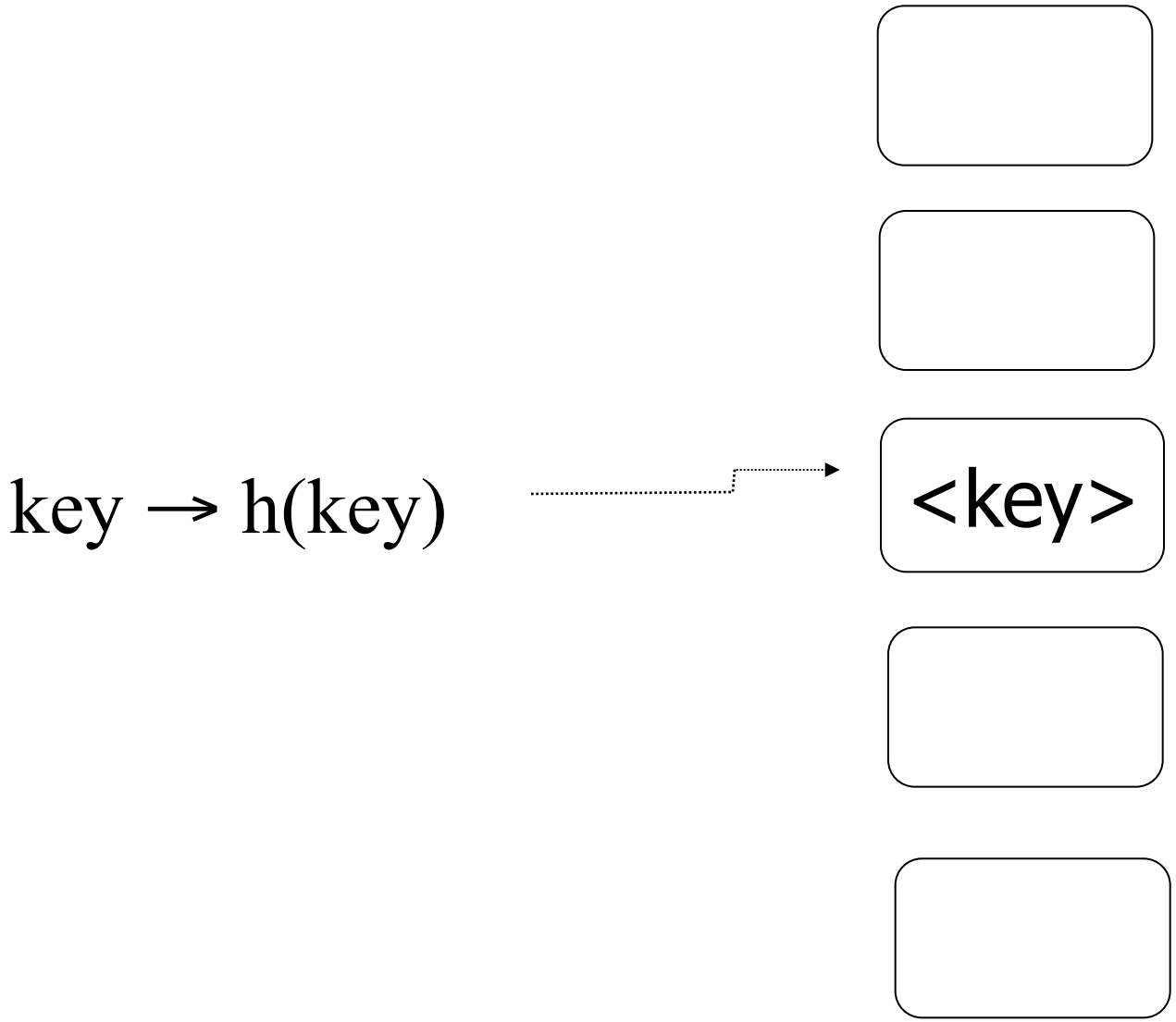
- Dense/Sparse Indexes
- B Trees
- Hashing
  - Overall Idea
  - Handling growth: (i) Extensible Hashing, (ii) Linear Hashing
- Multi-dimensional Indexes

# Hashing



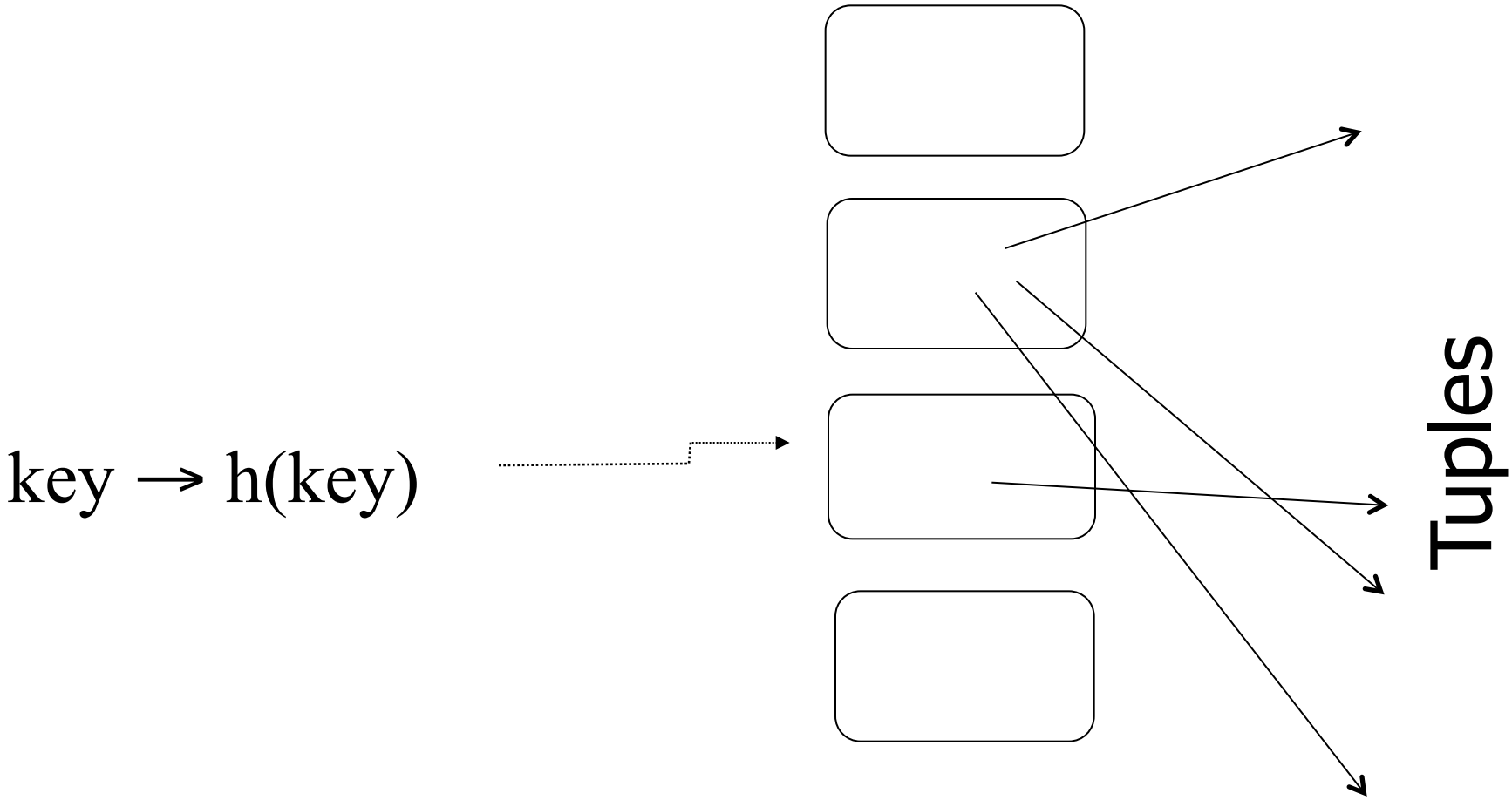
# Two Alternatives

(i) Buckets contain the tuples



# Two Alternatives

(ii) Buckets contain pointers to tuples




- Above needed if tuples are already stored (e.g., for secondary indexes)

# 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*

# EXAMPLE 2 records/bucket

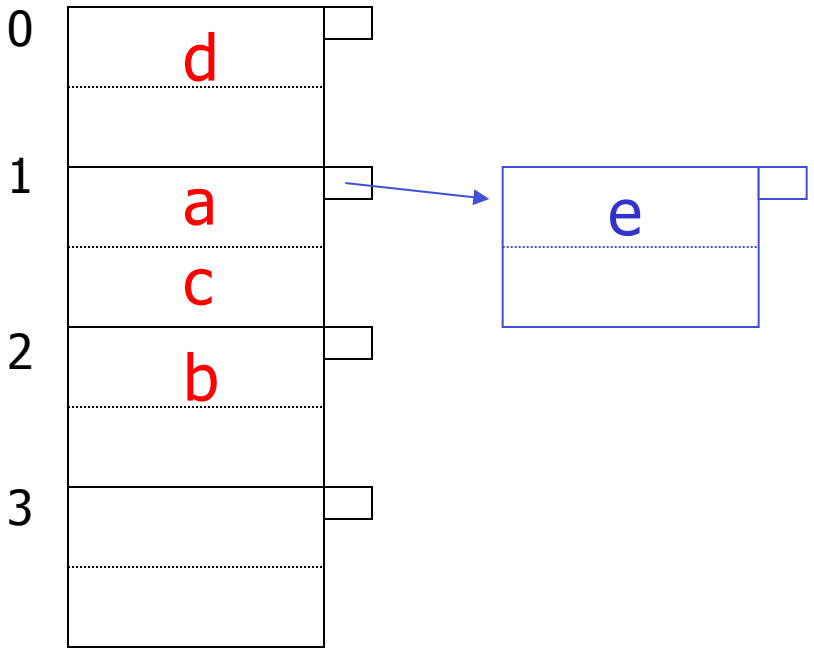
INSERT:

h(a) = 1

h(b) = 2

h(c) = 1

h(d) = 0




h(e) = 1

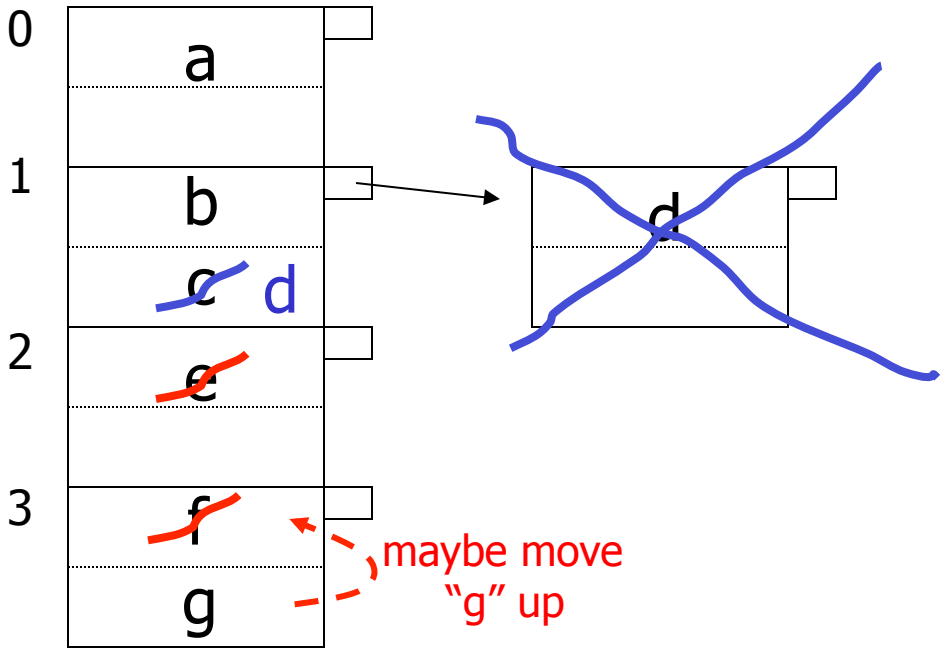
# EXAMPLE: deletion

Delete:

e

f

c



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

## Within a bucket:

- Do we keep keys sorted?
- Yes, if CPU time critical and inserts/deletes are not too frequent

# Rule of thumb:

$$\text{Space Utilization} = \frac{\text{\# keys in the system}}{\text{total \# keys that fit (without overflows)}}$$

- Keep space utilization between 50% to 80%
- If < 50%, wasting space
- If > 80%, overflows significant

# How do we cope with growth?

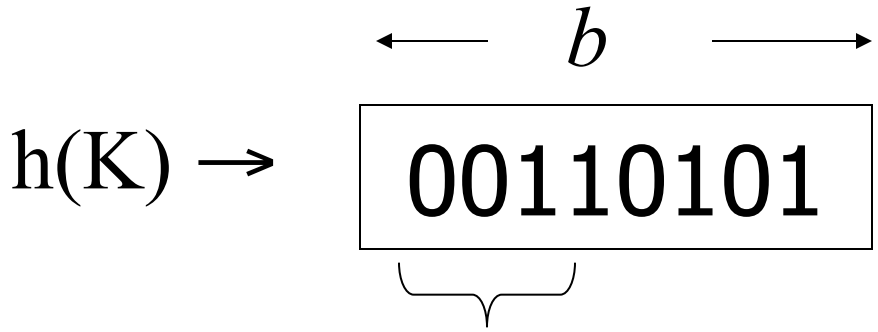
- Overflows and reorganizations
- Dynamic hashing
  - Extensible
  - Linear

# Outline

- Dense/Sparse Indexes
- B Trees
- Hashing
  - Overall Idea
  - Handling growth: (i) Extensible Hashing, (ii) Linear Hashing
- Multi-dimensional Indexes

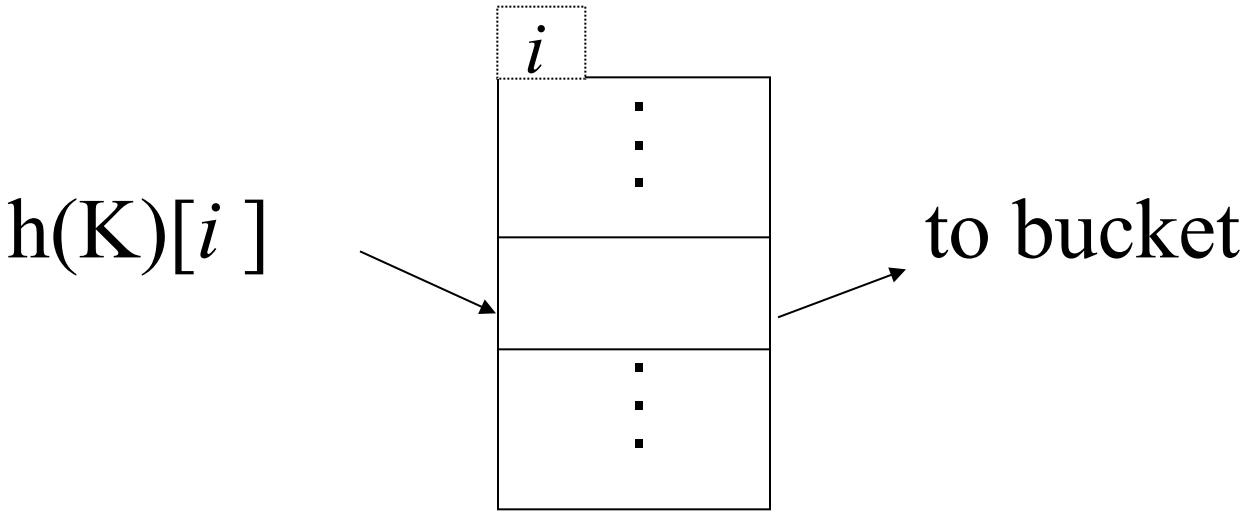
# Extensible Hashing: Two ideas

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

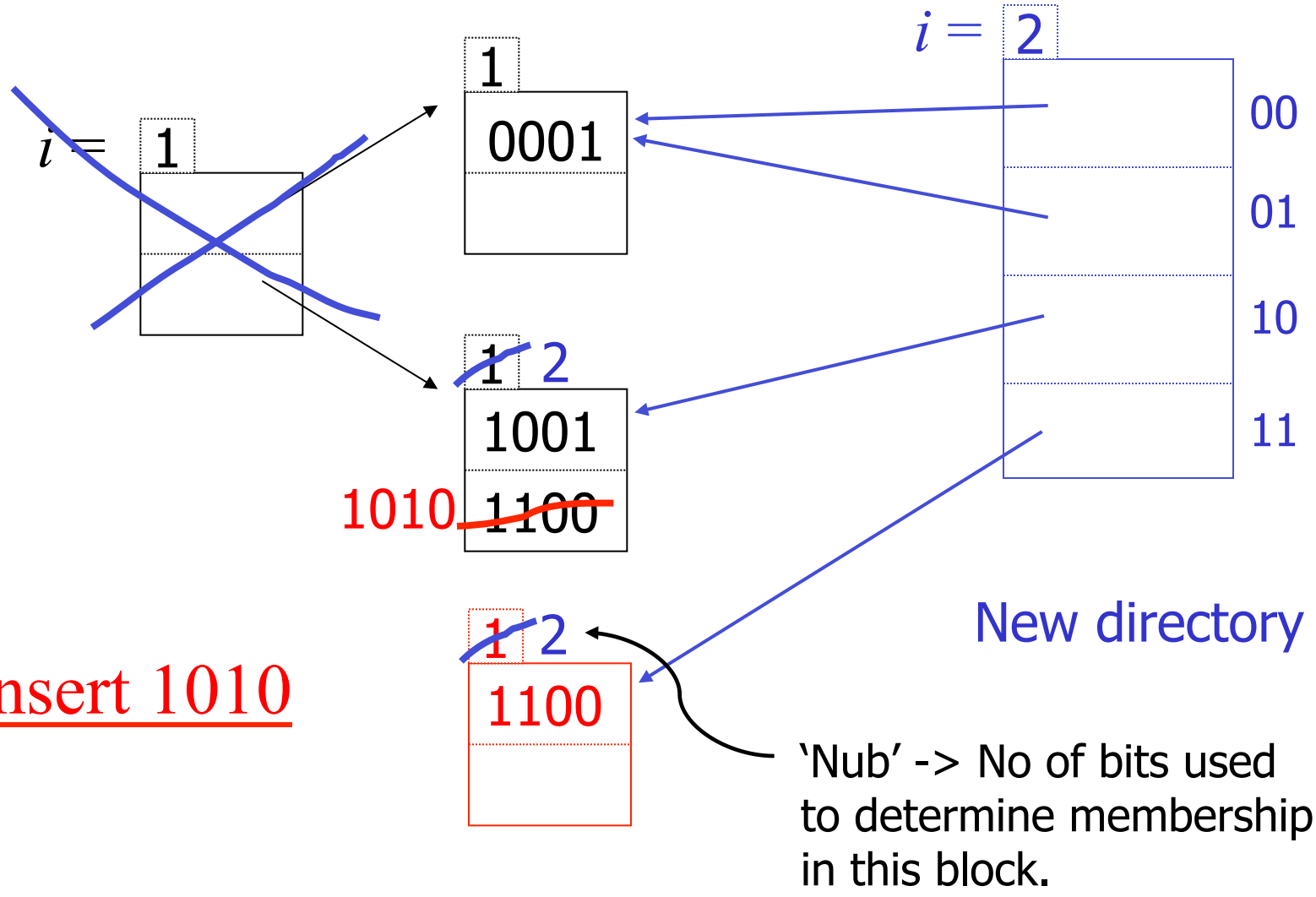



use $i$ → grows over time….

(b) Use directory

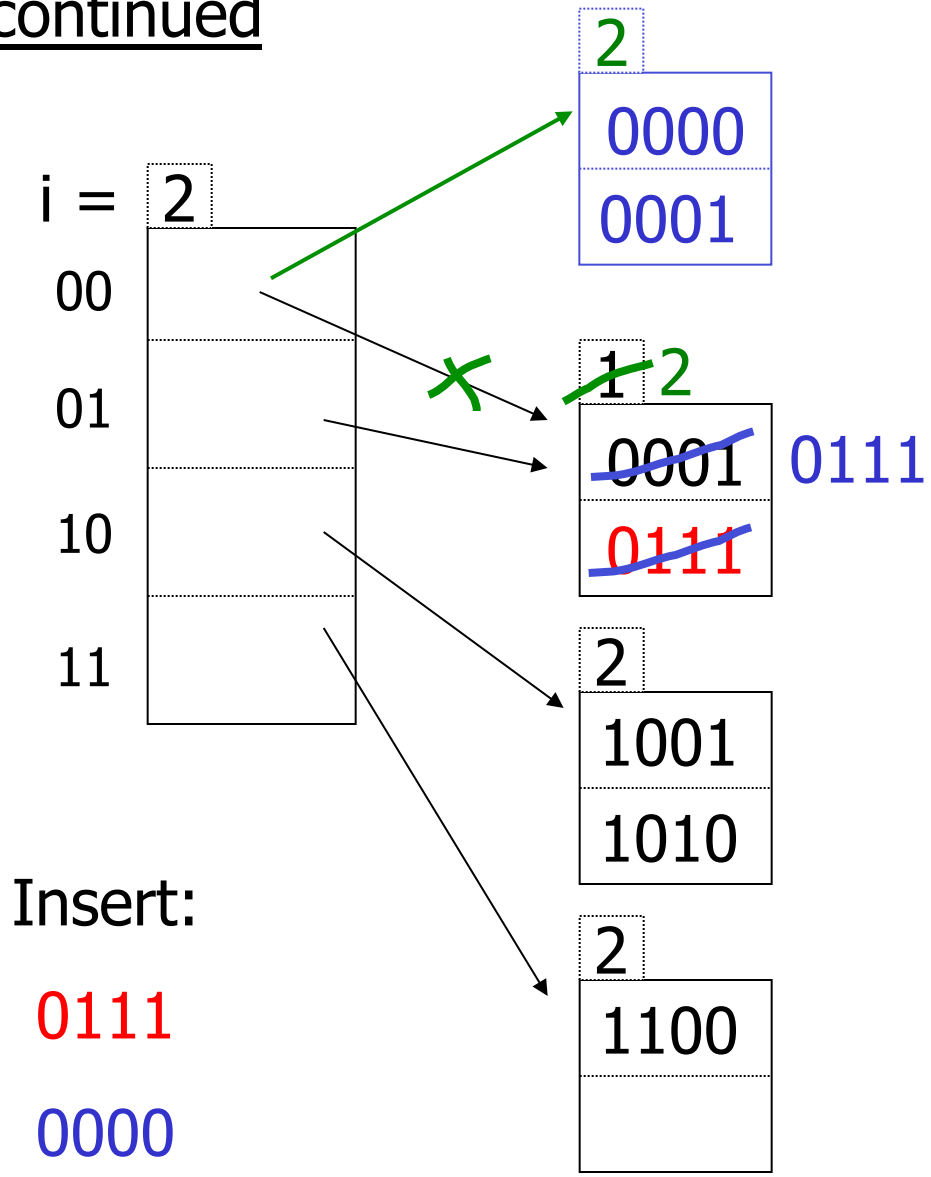


# Example: h(k) is 4 bits; 2 keys/bucket

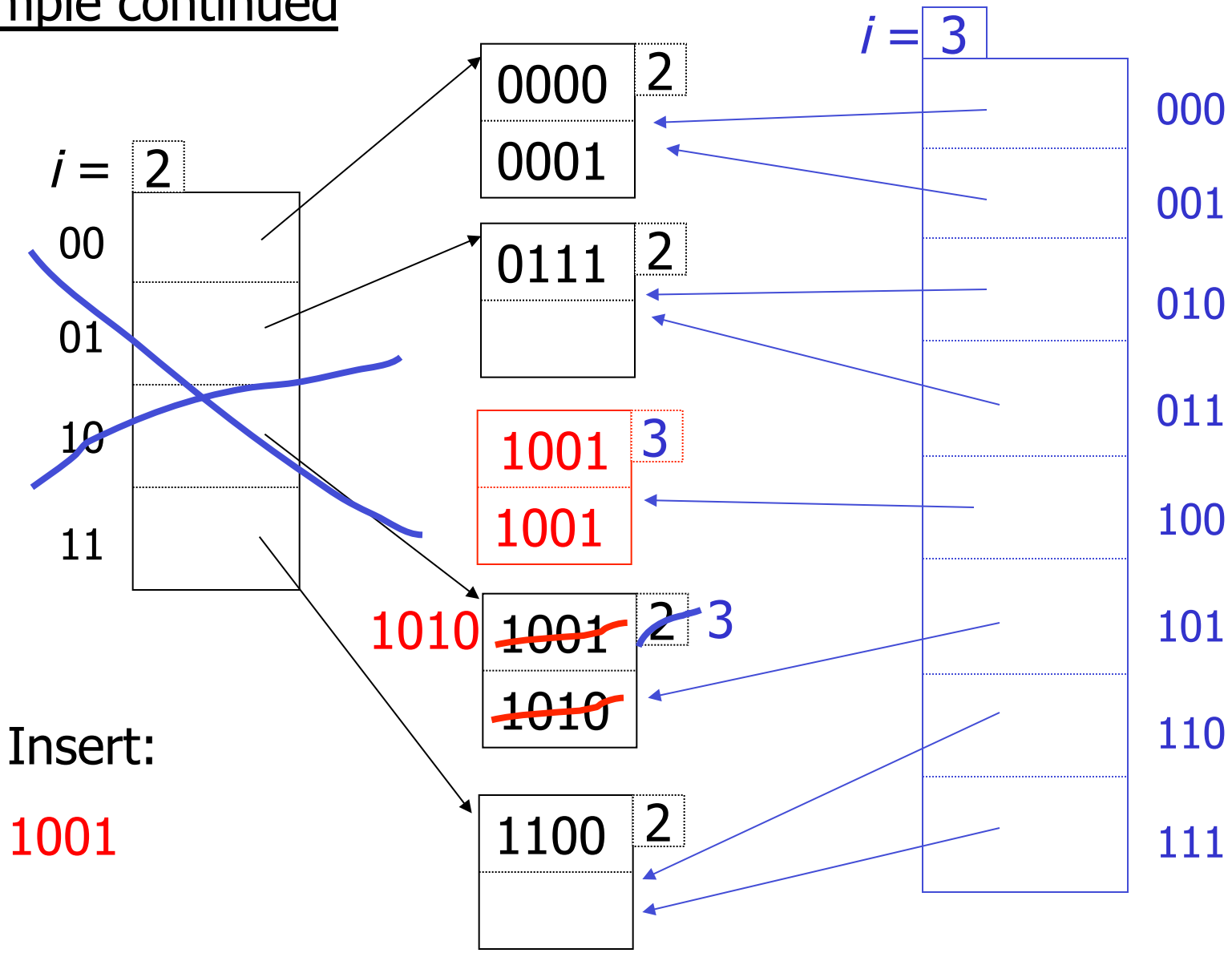



Insert 1010

# Example continued



# Example continued



# Extensible hashing: deletion

- Reverse insert procedure

# Extensible Hashing Summary

- If the directory fit in memory, then total I/Os is always ONE.

Negatives:

- Increase in i requires a lot of work.
- Directory doubles in size – so, sudden impact.
- Bad case: Assume blocks store 2 records. If there are 3 keys are "almost same", then i will end up being very large.

# Outline

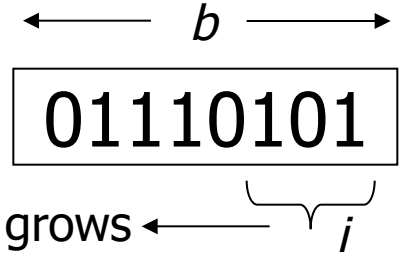
- Dense/Sparse Indexes
- B Trees
- Hashing
  - Overall Idea
  - Handling growth: (i) Extensible Hashing, (ii) Linear Hashing
- Multi-dimensional Indexes

# Linear hashing

- Another dynamic hashing scheme

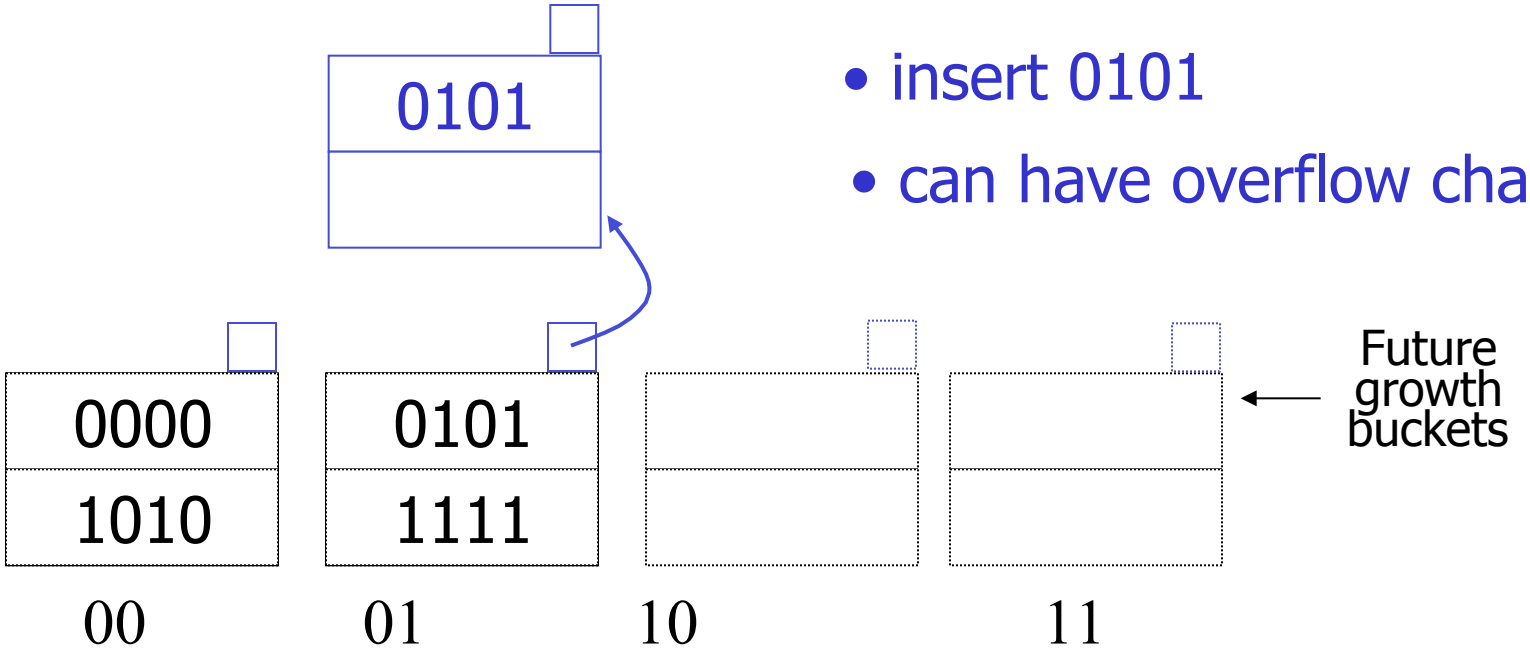
## Two ideas:

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




(b) File grows linearly

# Example: $b$=4 bits, $i$ =2, 2 keys/bucket




- insert 0101
- can have overflow chains!

$m$ = 01 (max used block)

**Rule** If h(k)[$i$ ] ≤ $m$, then

look at bucket h(k)[i ]

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

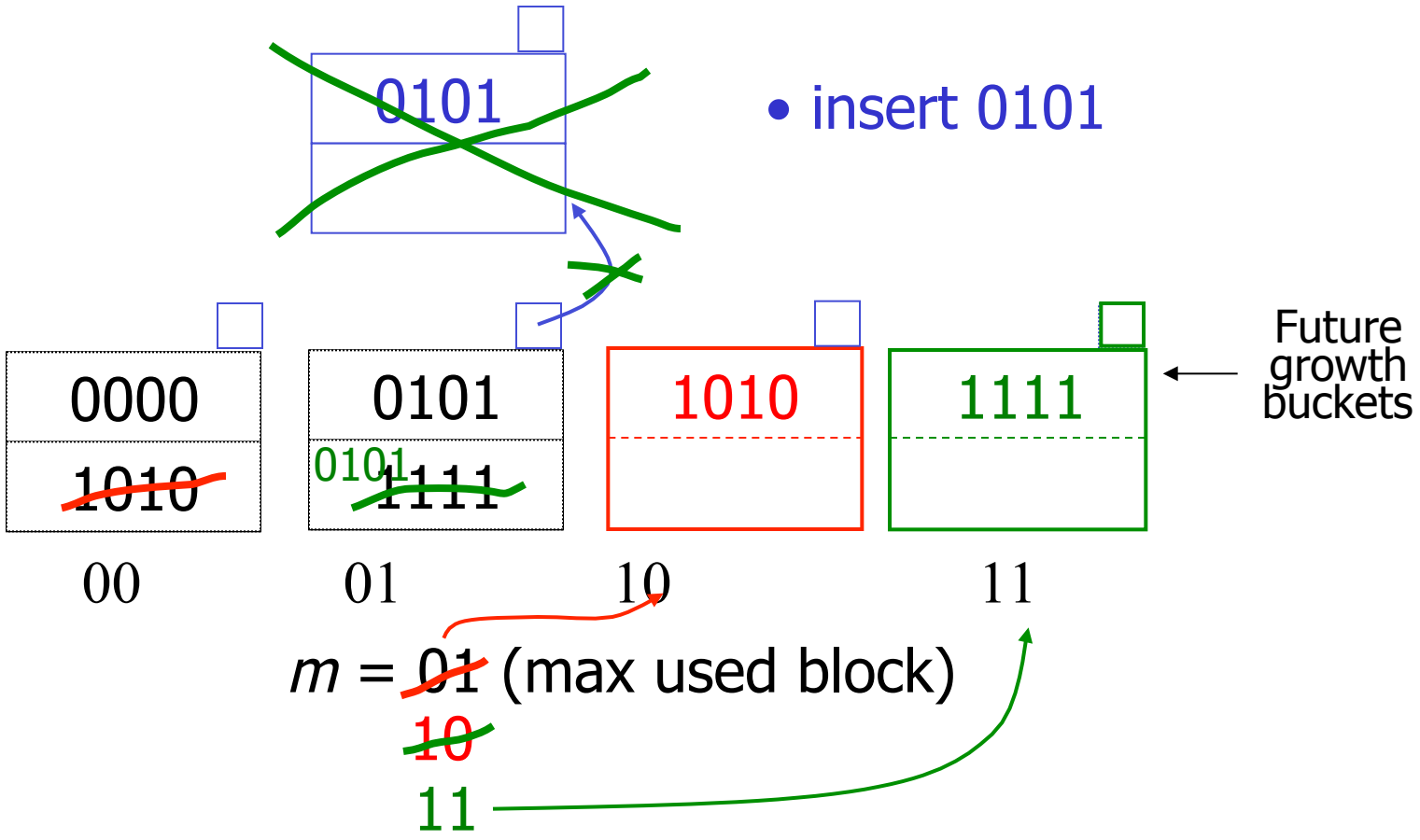
# When do we expand file?

Space Utilization =

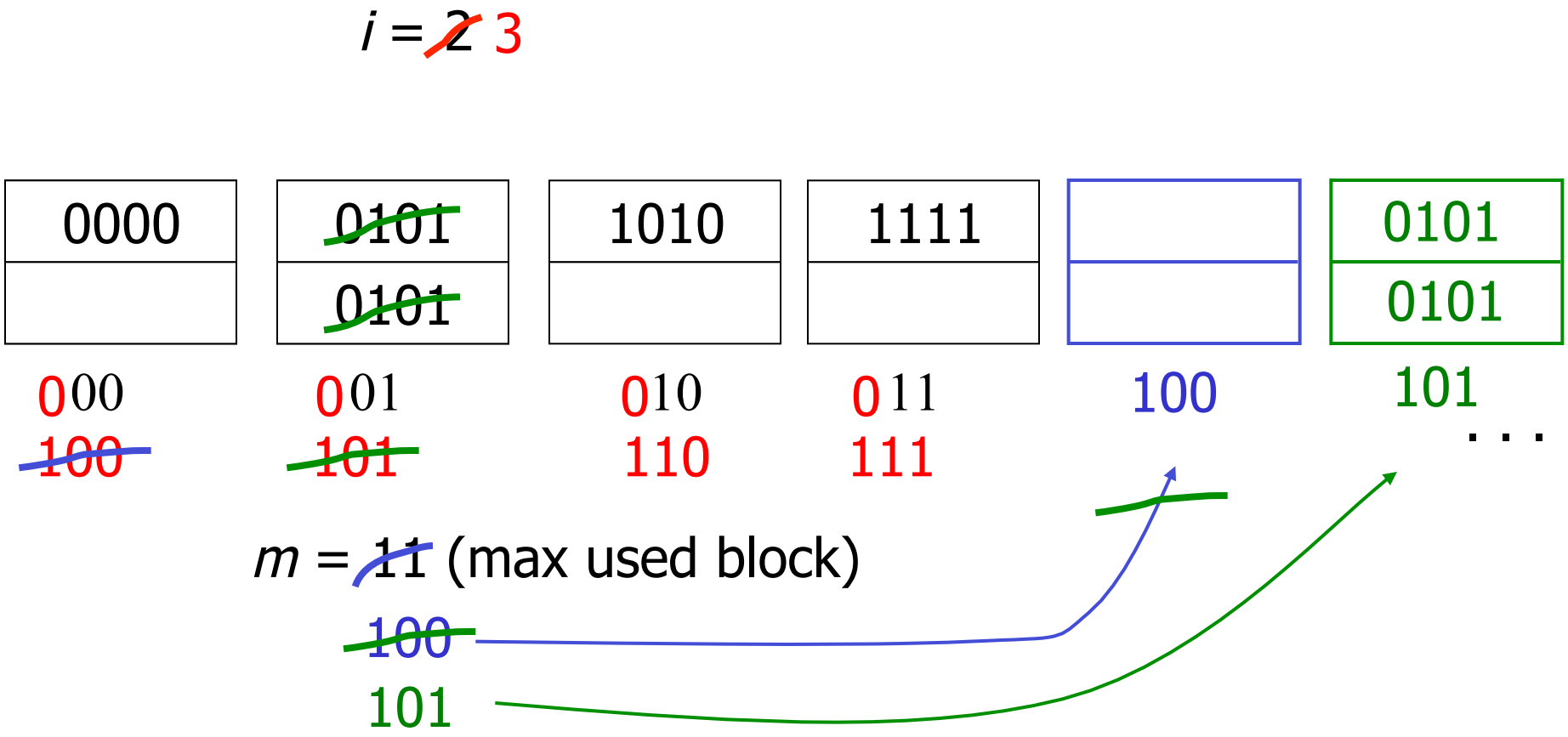
$$\frac{\text{\# keys in the system}}{\text{total \# keys that fit (without overflows)}}$$

- If $>$ threshold then increase $m$
  (and maybe $i$ )

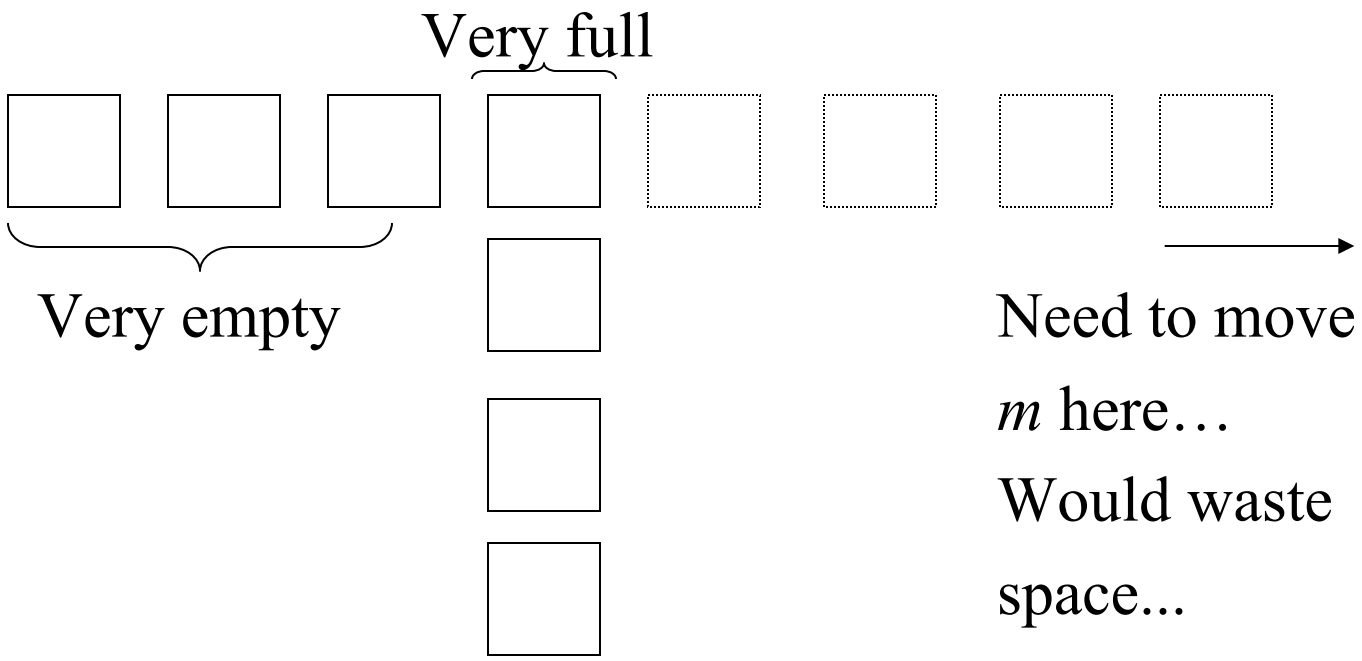
# Example: $b$=4 bits, $i$ =2, 2 keys/bucket



# Example Continued: How to grow beyond this?



# BAD CASE for Linear Hashing



# Indexing vs Hashing

- Hashing good for probes given key

  e.g., SELECT …
  FROM R
  WHERE R.A = 5

- INDEXING (Including B Trees) good for

  Range Searches:

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

# Outline

- Dense/Sparse Indexes
- B Trees
- Hashing (Extensible, Linear)
- Multi-dimensional Indexes

# Multi-dimensional Indexes

## Motivation

Find records where

DEPT = “Toy” AND SAL > 50k

# General Multi-dimensional Queries

Assume tuples with <w,x,y,z>

1. Partial-Match Queries:
   - Find tuples where "x=5 and y=6"
2. Range Queries
   - Find tuples where "5 < x < 10 and y >5"
3. Nearest Neighbor Queries
   - Find tuples "closest" to <4,3,5,7>.
4. Where-am-I Queries (over object shapes)
   - Find objects containing <2,3,6,7>

# Multi-dimensional Indexes

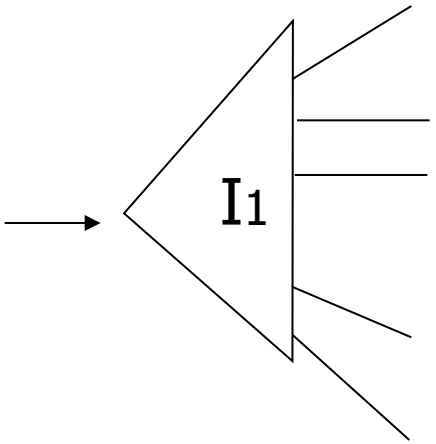
## Motivation

Find records where

DEPT = “Toy” AND SAL > 50k

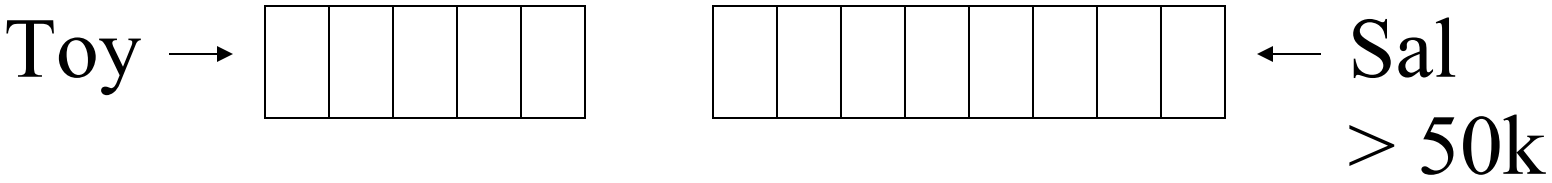
# Strategy I: Use one-dimensional index

- Use one index, say Dept.
- Get all Dept = "Toy" records and check their salary



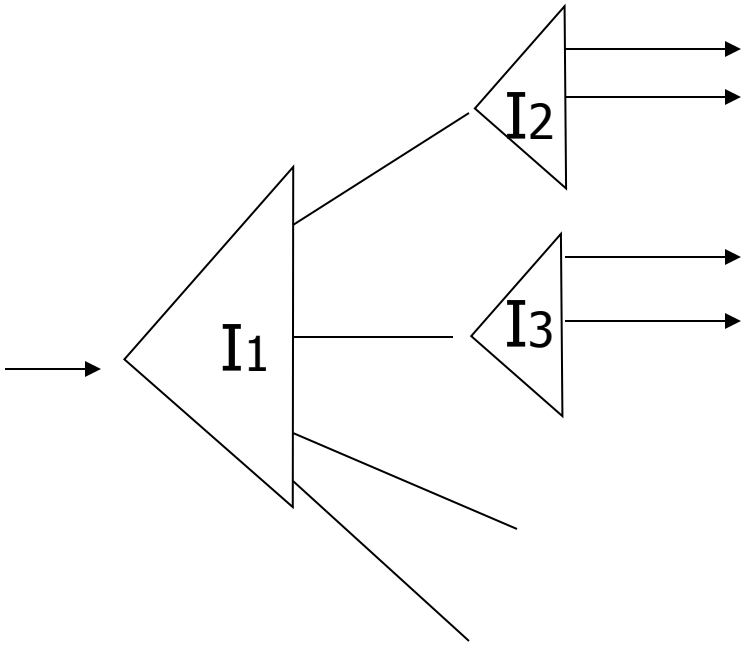
# Strategy II: Manipulate Pointers

- Use 2 Indexes; manipulate pointers

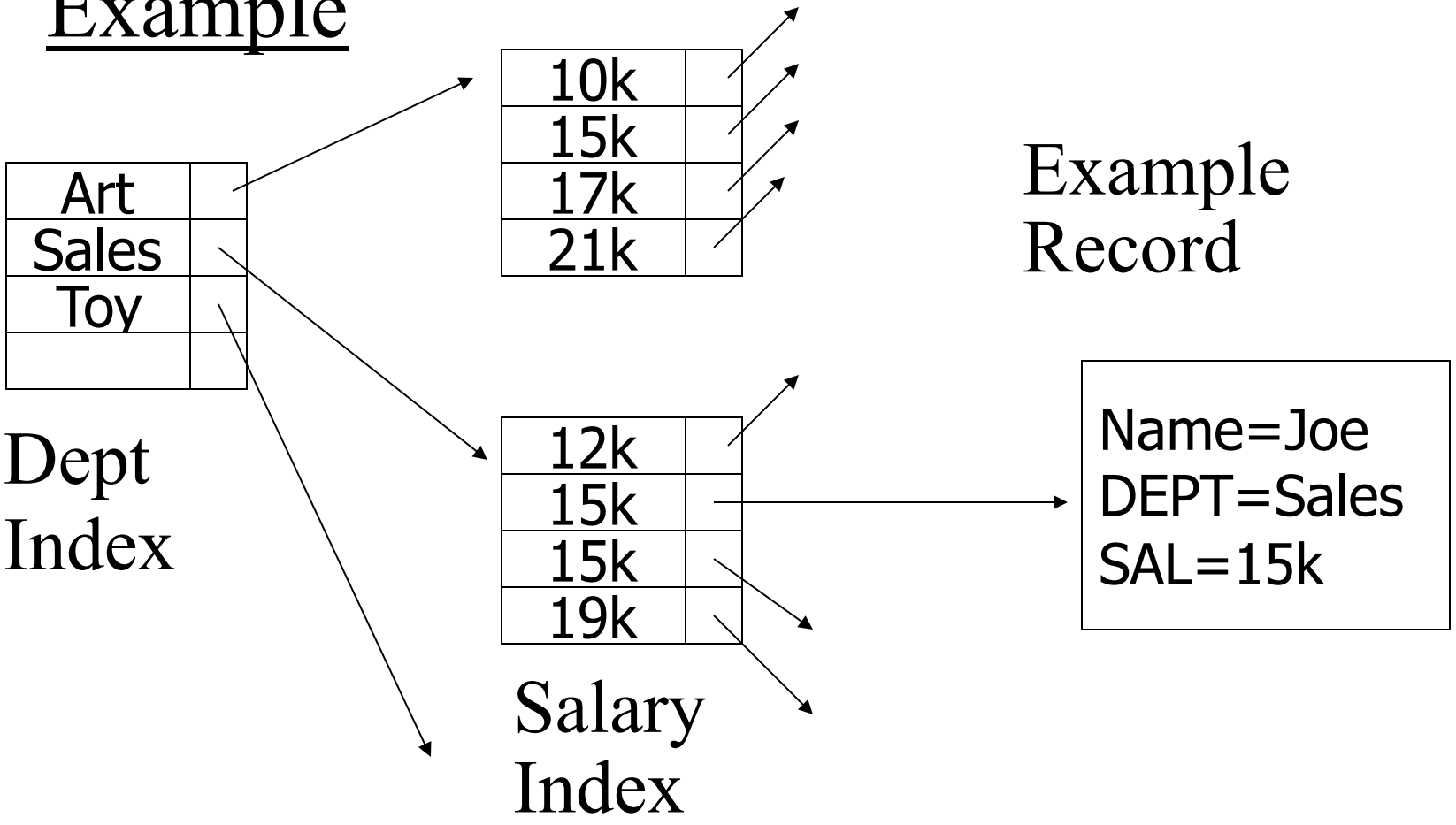


# Strategy III: Multi-dimensional Indexes

One idea:



# Example


Art
Sales
Toy
Dept Index
10k
15k
17k
21k
12k
15k
15k
19k
Salary Index
Example Record
Name=Joe
DEPT=Sales
SAL=15k

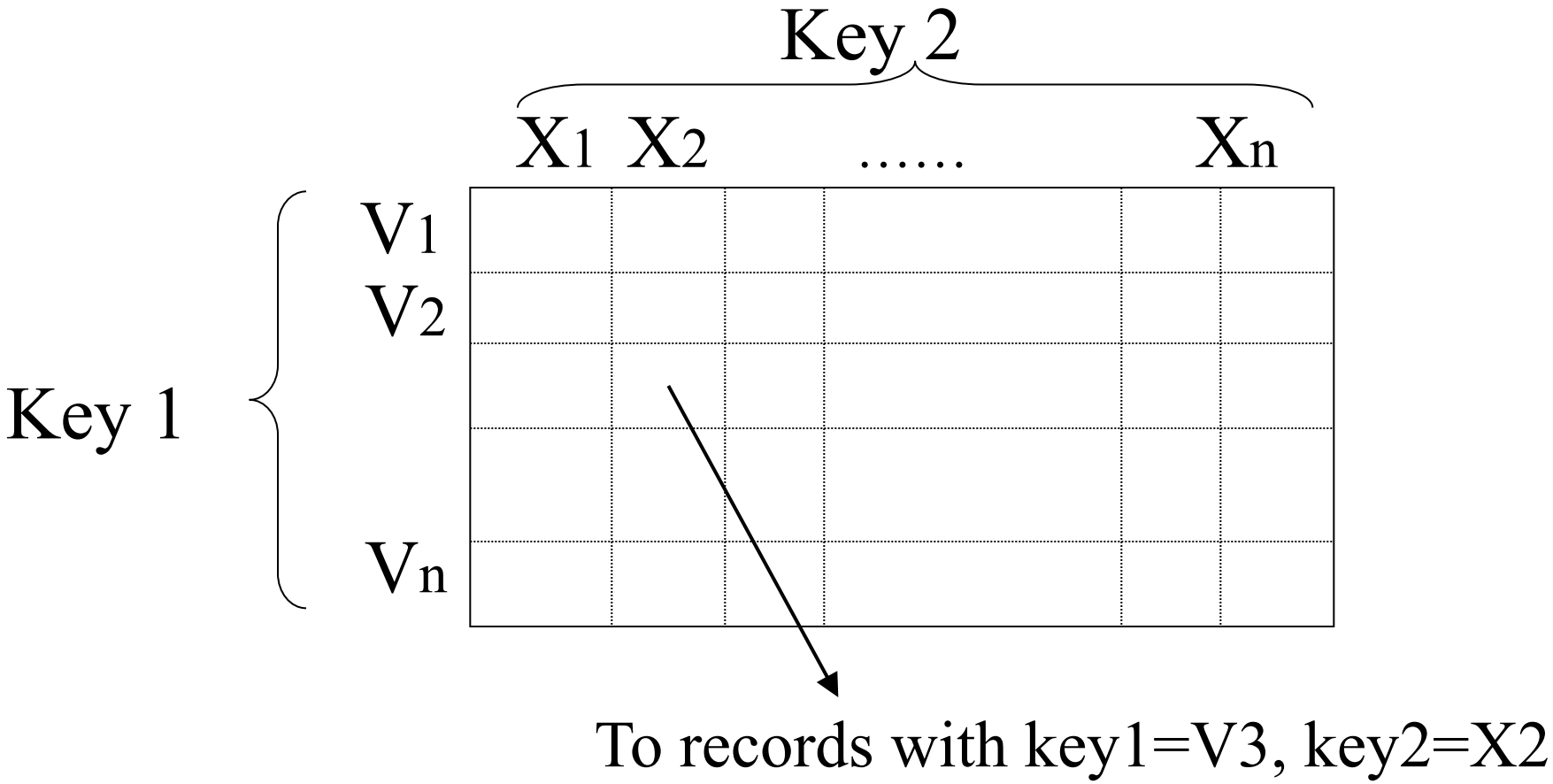
# For which queries is this index good?

- ☐ Find RECs Dept = “Sales” $\wedge$ SAL=20k
- ☐ Find RECs Dept = “Sales” $\wedge$ SAL $\geq$ 20k
- ☐ Find RECs Dept = “Sales”
- ☐ Find RECs SAL = 20k

# Types of multidimensional indexes

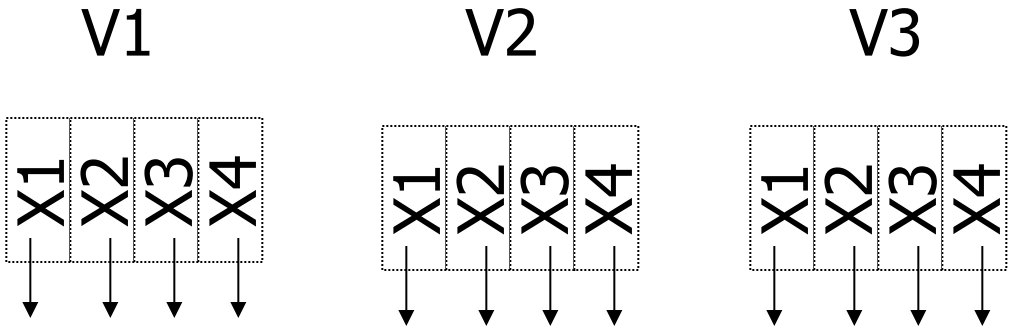
- Grid Files
- Partitioned Hash
- Tree like structures
  - Multi-key indexes (previous slide)
  - kd-tree
  - Quad tress
  - R trees

# Grid Index



# Storing Grid Index

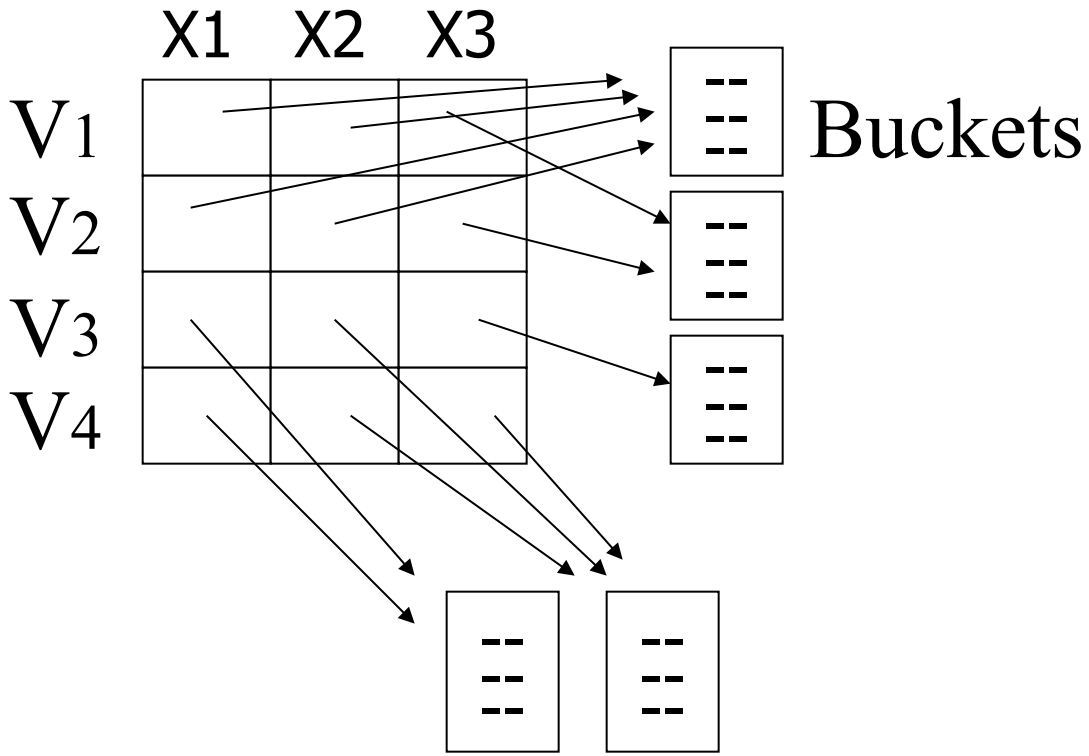
Like Array...




- Each <Vi,Xj> entry should be of uniform size (array elements must be of same size).

➔ Store a single pointer to a bucket (of tuples/pointers).

# I.e., use indirection



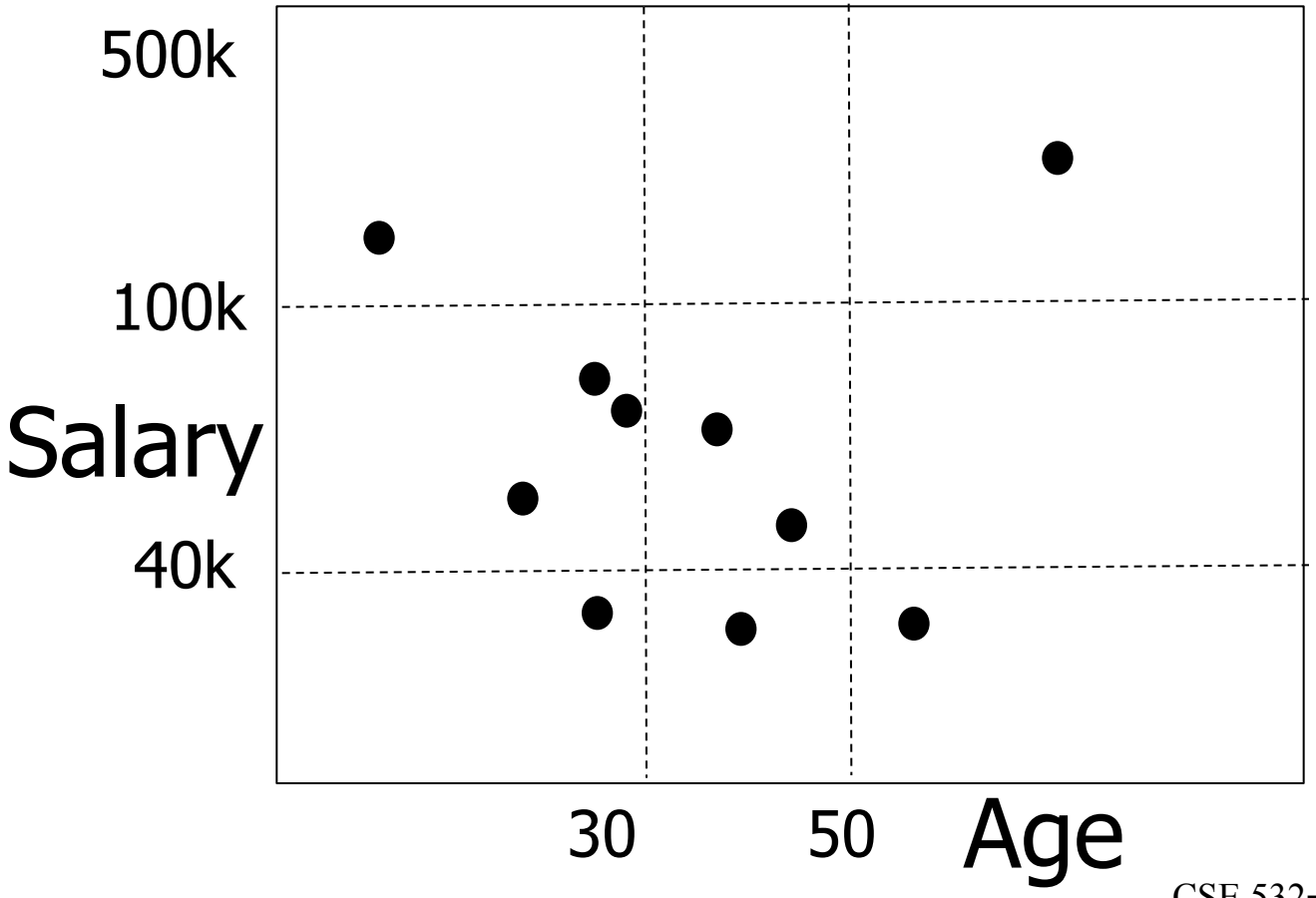
# CLAIM

- Can quickly find records with
  - key 1 = $V_i$ ∧ Key 2 = $X_j$
  - key 1 = $V_i$
  - key 2 = $X_j$

- And also ranges….
  - E.g., key 1 ≥ $V_i$ ∧ key 2 < $X_j$

# Grid Files with Value-Ranges

- Visualize a grid file as "stripes" in each dimension.

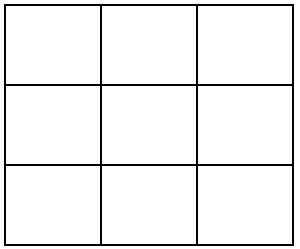


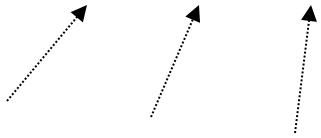
# Grid Files with Value-Ranges

Salary

| | |
|---|---|
| 0-40K | 1 |
| 40K-100K | 2 |
| 100K-500k | 3 |

Grid





Age

| 1 | 2 | 3 |
|---|---|---|
| 0-30 | 30-50 | 50-100 |

# Grid files

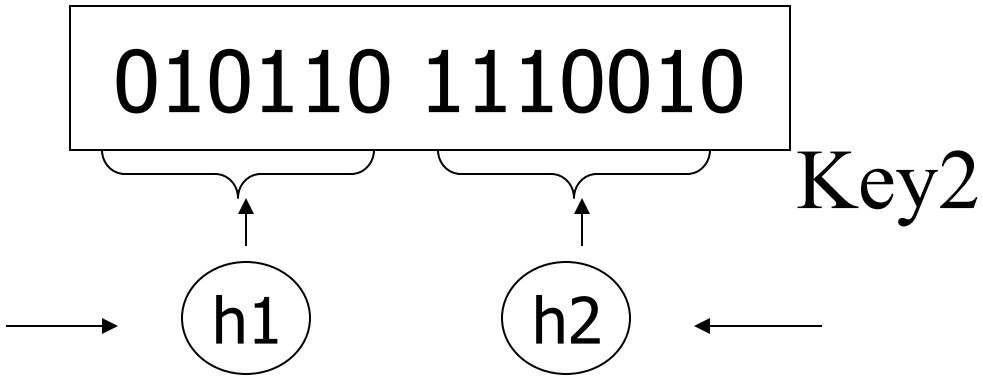
- Need to carefully split the ranges.
- May need extra indexes to determine row and/or column numbers from a key value
  - These indexes may be small enough to reside in main memory.
  - Most cases, these indexes could be replaced by "binary search".
- Good for partial-match and range queries.

# Types of multidimensional indexes

- Grid Files
- Partitioned Hash
- Tree like structures
  - Multi-key indexes (previous slide)
  - kd-tree
  - Quad tress
  - R trees

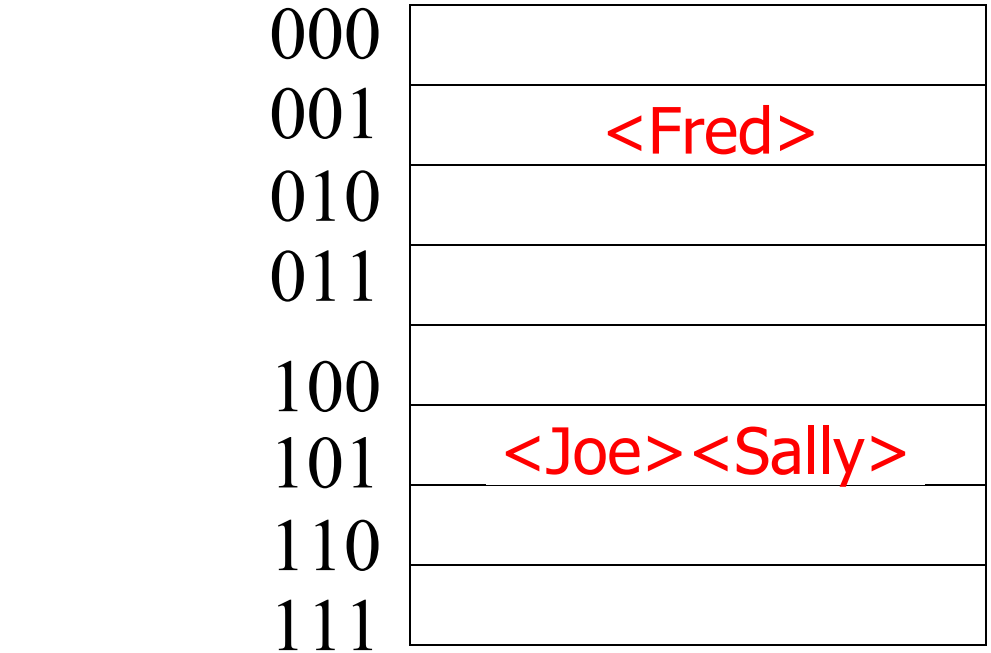
# Partitioned Hash Function

Idea:



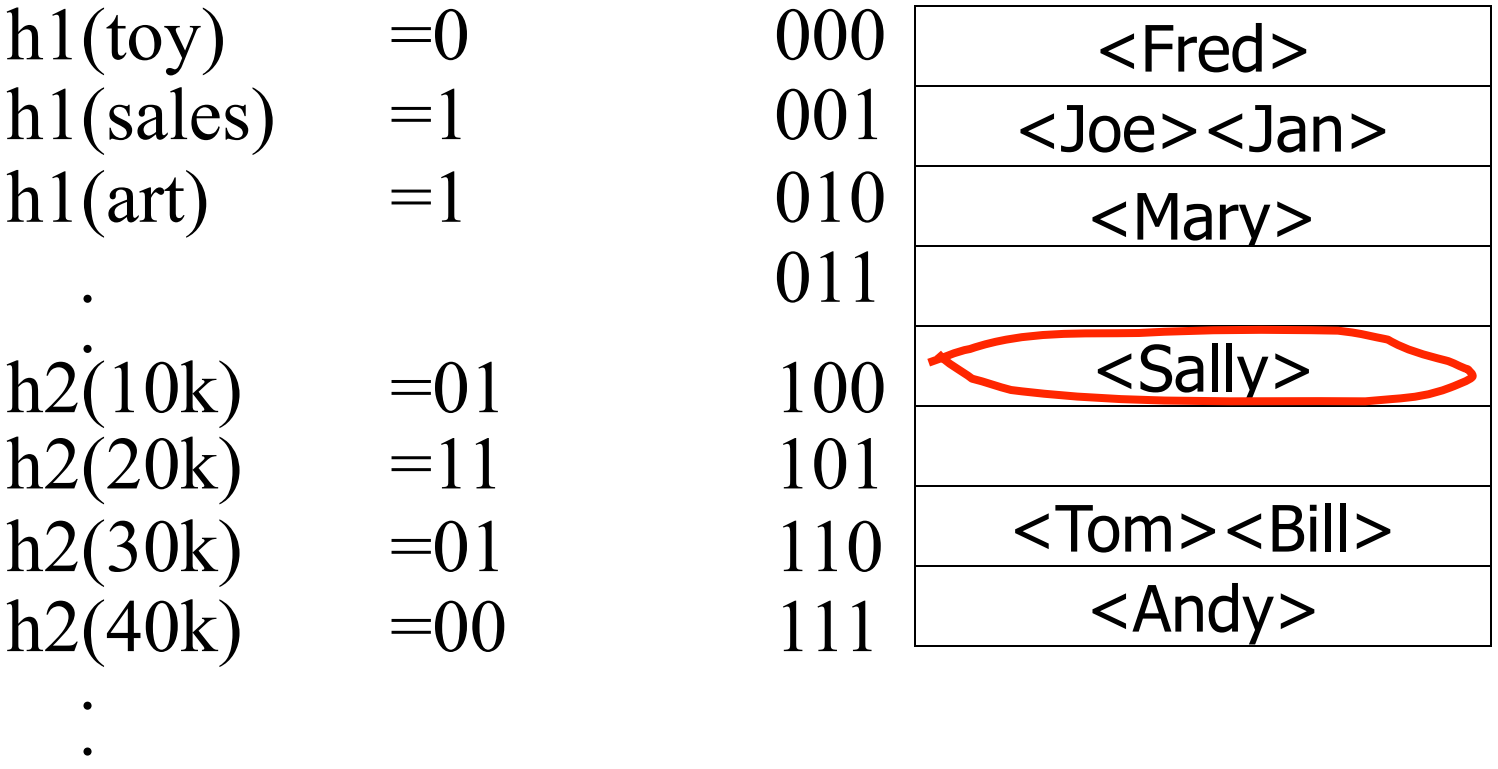
# EX:

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.



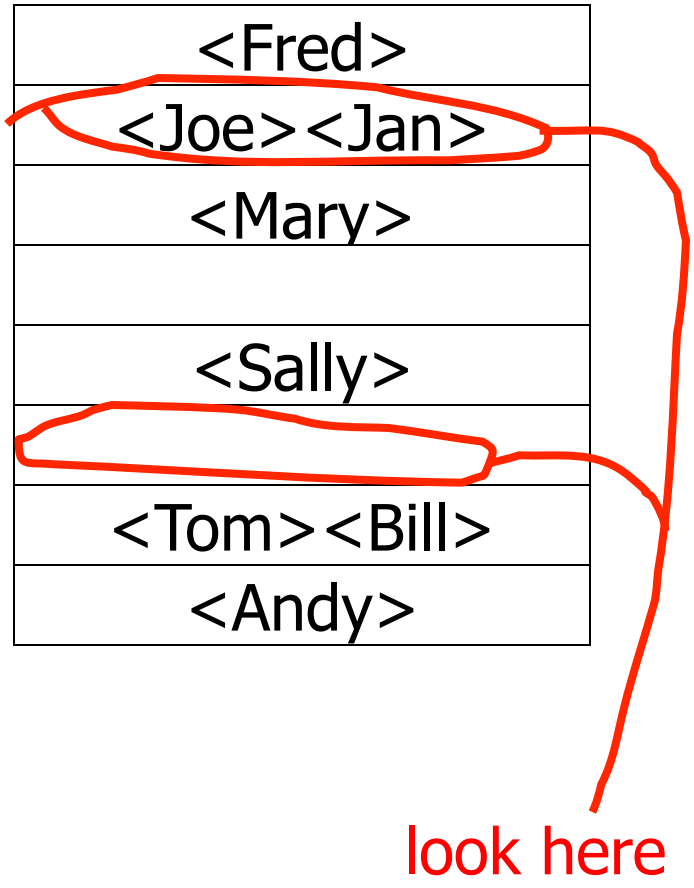

Insert

<Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>



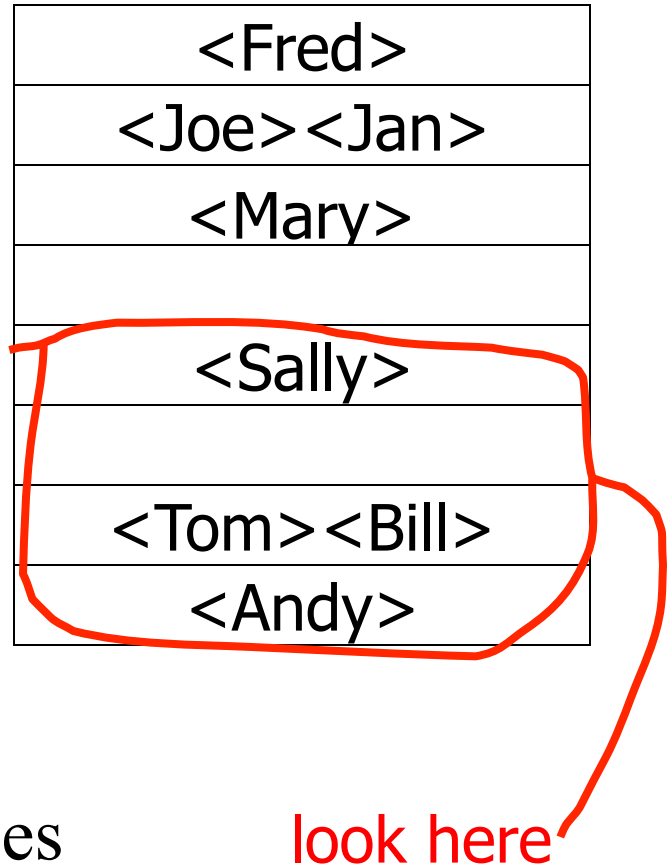

- Find Emp. with Dept. = Sales ∧ Sal=40k

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.




- Find Emp. with Sal=30k

h1(toy) =0
h1(sales) =1
h1(art) =1
.
.
h2(10k) =01
h2(20k) =11
h2(30k) =01
h2(40k) =00
.
.




- Find Emp. with Dept. = Sales

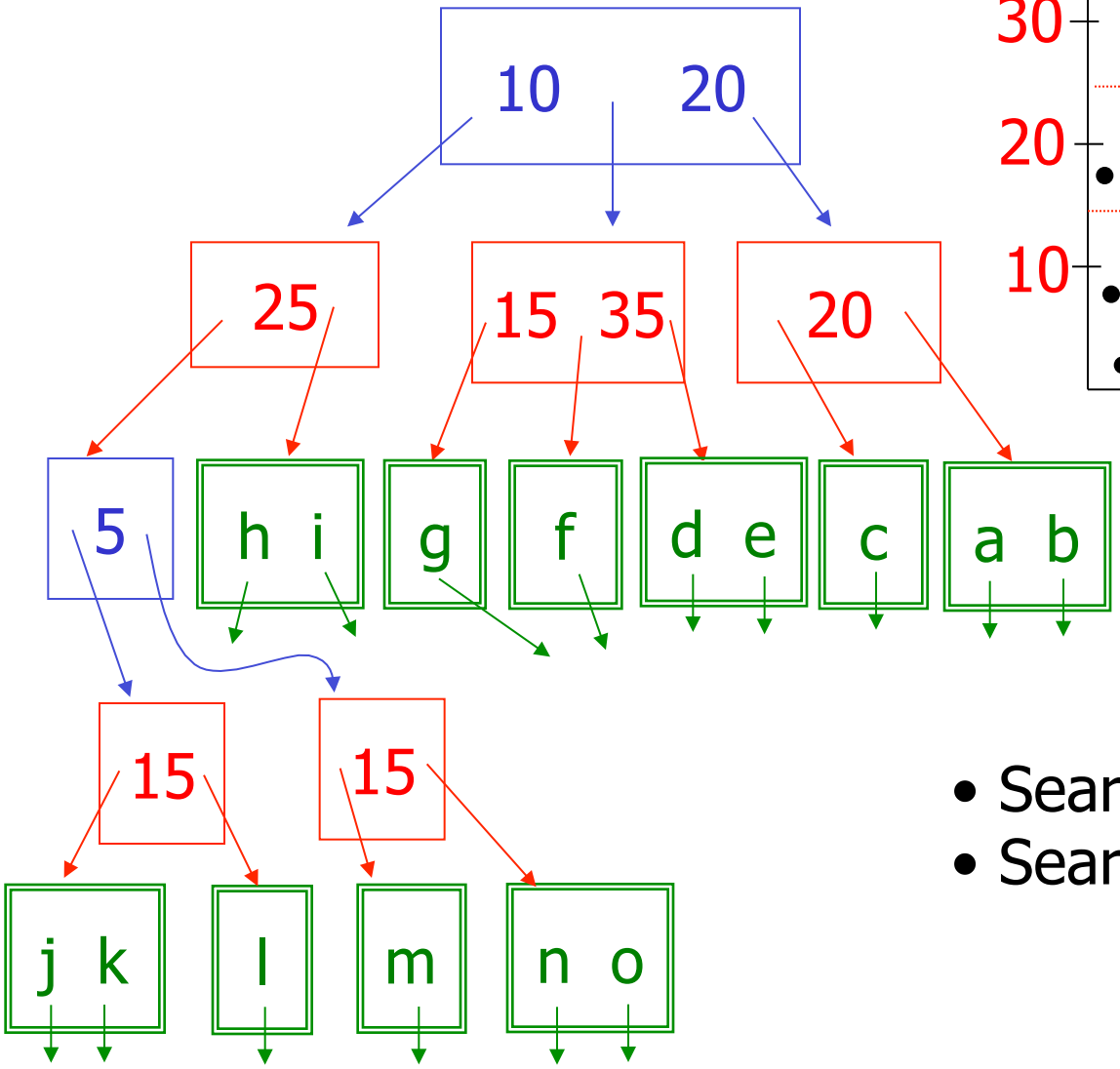
# Grid Files vs. Partitioned Hash

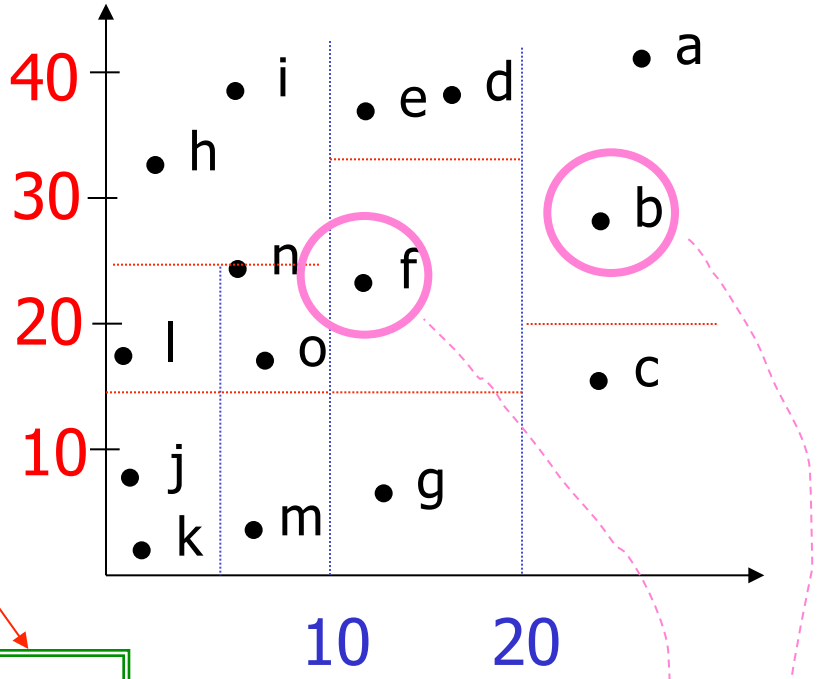
- Grid Files are good for range-queries, partial-matches. Can be used for nearest-neighbor queries (by considering bigger and bigger "ranges")
- Partitioned Hash are good only for partial-matches.

# Types of multidimensional indexes

- Grid Files
- Partitioned Hash
- Tree like structures
  - Multi-key indexes (previous slide)
  - kd-tree
  - Quad tress
  - R trees

# kd-Tree




- Search points near f
- Search points near b

# Trees-like structures

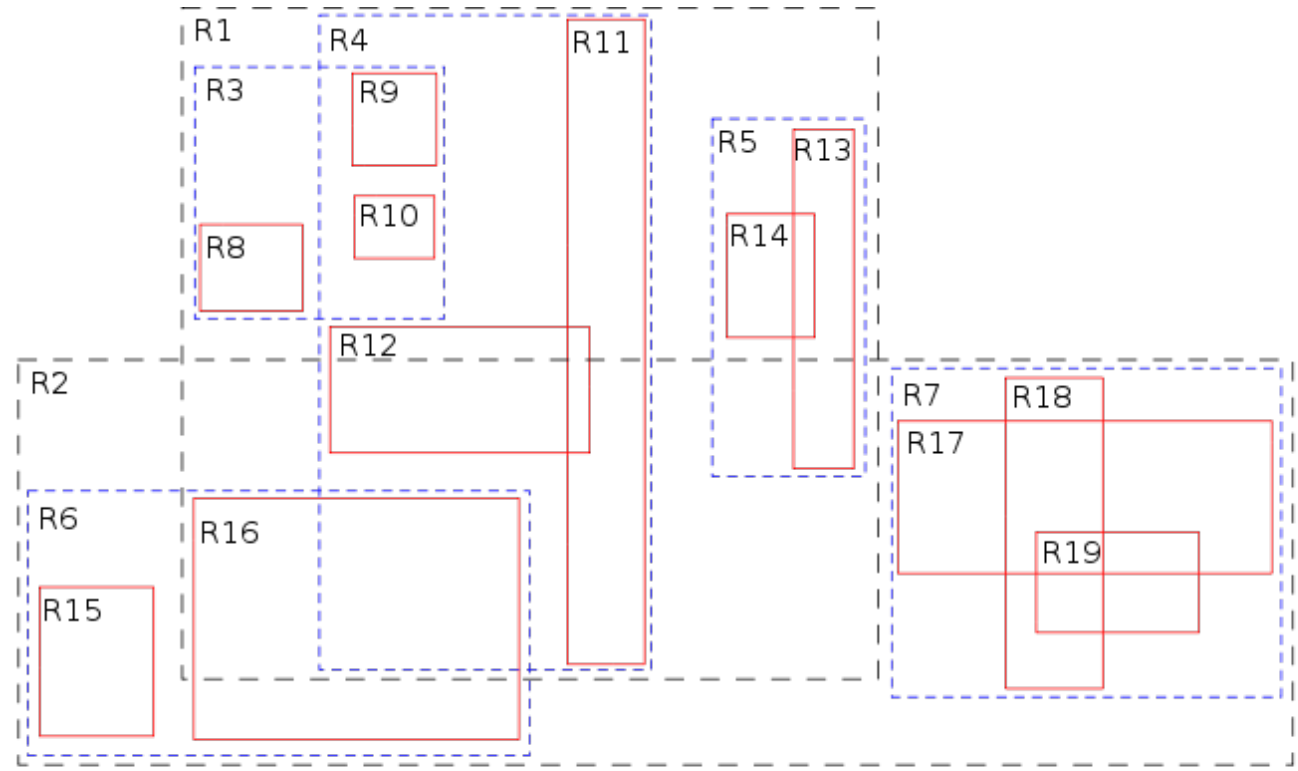
- kd-Trees
  - Interior node (A,V) has two children
  - (A-value < V) and (A-value > V)
- Quad Trees
  - Each node corresponds to a rectangle
  - Partitions its region into sub-regions using a point (thus, each level divides across all dimensions)
- R-tree (NEXT .. ) *used to store regions.*
  - Similar to Quad tree, except partitions may not be disjoint; may not cover the entire region.

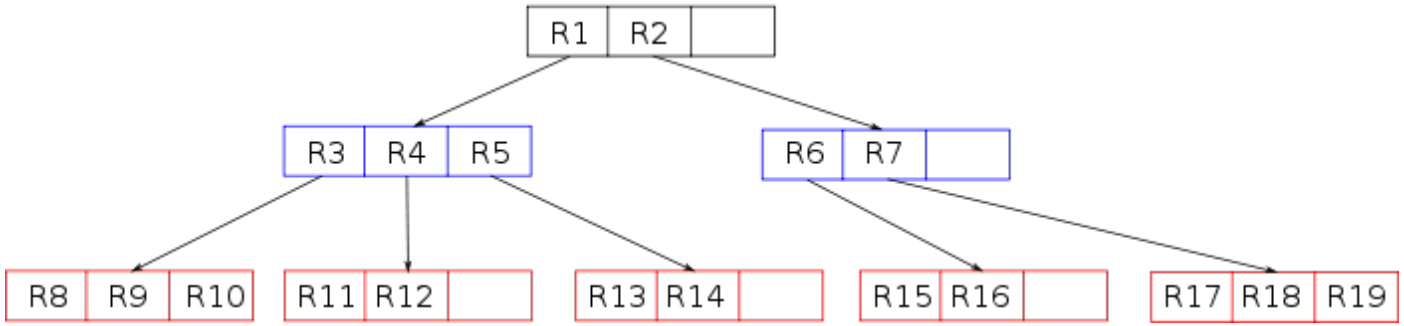
# R-trees

- Suppose we want to store "regions" instead of points.
- E.g., to store objects of geographic maps, we must store points, roads, buildings, spatial-regions, etc. A typical query could be:
  - Find all universities within 100km of the given point?
  - Find all roads that intersect with a given region?

# R-trees

- R-trees are similar to kd-trees, wherein each node represents rectangle but:
  - The "subrectangles" corresponding to the children may not be disjoint, and together they may not completely cover the parent's rectangle.
- R-tree can also be looked upon as built "bottom-up" – where given objects/regions are grouped into bigger rectangles and so on.


R1
R4
R11
R3
R9
R5
R13
R10
R14
R8
R12
R2
R7
R18
R17
R6
R16
R19
R15


R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
R15
R16
R17
R18
R19

# R-trees

- Insertion: Given a rectangle to insert.
  - Start from the root, and pick one of the children that contains the given rectangle. Continue till you reach the leaf. Store.
- Where-am-I Query: Given a rectangle/point, find intersection/containing regions:
  - Start from the root, and traverse all of the children that intersect/contain the given rectangle. Continue till you reach the leaf.

# Outline

- Dense/Sparse Indexes
- B Trees
- Hashing (Extensible, Linear)
- Multi-dimensional Indexes (Grid, Partitioned Hash, kd-tree, quad-trees, R-trees, BitMaps)

# BitMap Indexes

- Store binary vectors for each attribute value. The vector represents the record numbers that contain that value. E.g., a vector 010100001 for "dog" signifies that the 2nd, 4th, and 9th records contain the value "dog".
- Need to store such vectors for each value and each attribute.
- Records must have permanent numbers.

# BitMaps: Challenges/Issues

- Good for range, partial-match (bucket intersections) queries.
- Need to create other indexes (secondary or binary) for overall operation: to find a bit-vector for a value, and then to find the i-th record of the table.
- Vectors can be compressed for efficiency. Note that a vector is expected to have few 1s.
- Record Insertions: New record number.
- Record Deletions: "Retire" the number, thus, not effecting the numbering.