



University of
Southampton

Access Structures

COMP3211 Advanced Databases

Dr Nicholas Gibbins - nmg@ecs.soton.ac.uk

Overview

- Index basics
 - Sequential files
 - Dense indexes
 - Sparse indexes
 - Multi-level indexes
 - Secondary indexes
 - Indirection
- B+trees
- Hash tables

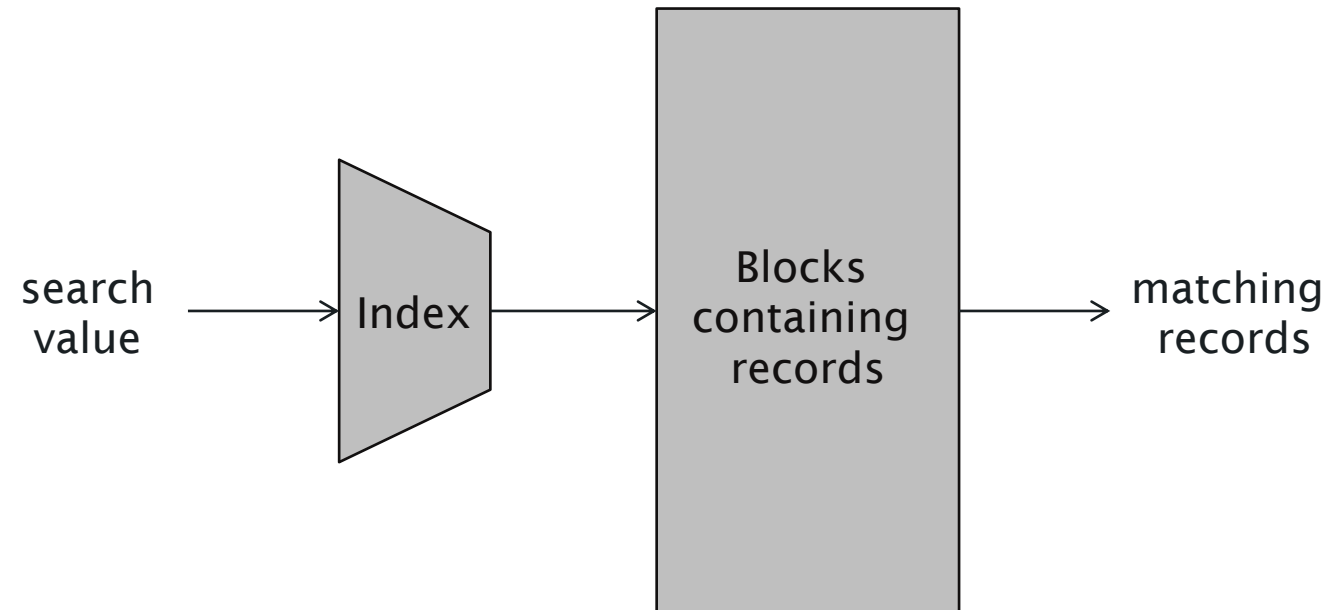
Index Basics

Index basics

- Relations are stored in files
- Files are stored as collections of blocks
- Blocks contain records that correspond to tuples in the relation

- How do we find the tuples that match some criteria?

Indexes



Sequential Files

- Tuples of a relation are sorted by their primary key
- Tuples are then distributed among blocks in that order
- Common to leave free space in each block to allow for later insertions

data file

10	
20	

30	
40	

50	
60	

70	
80	

90	
100	

110	
120	

To Index or Not To Index?

Maintaining an index costs time (processor, disk access)

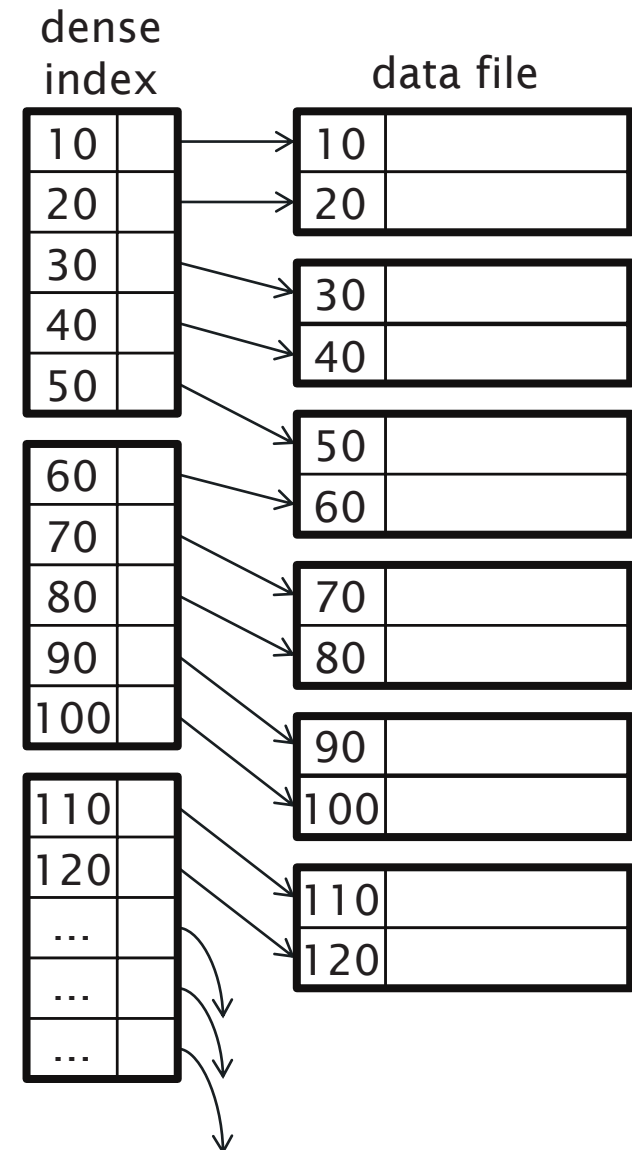
- When entries are added to the relation, index must be updated
- Index must be maintained to make good use of resources

There is a trade off between:

- Rapid access when retrieving data
- Speed of updating the database

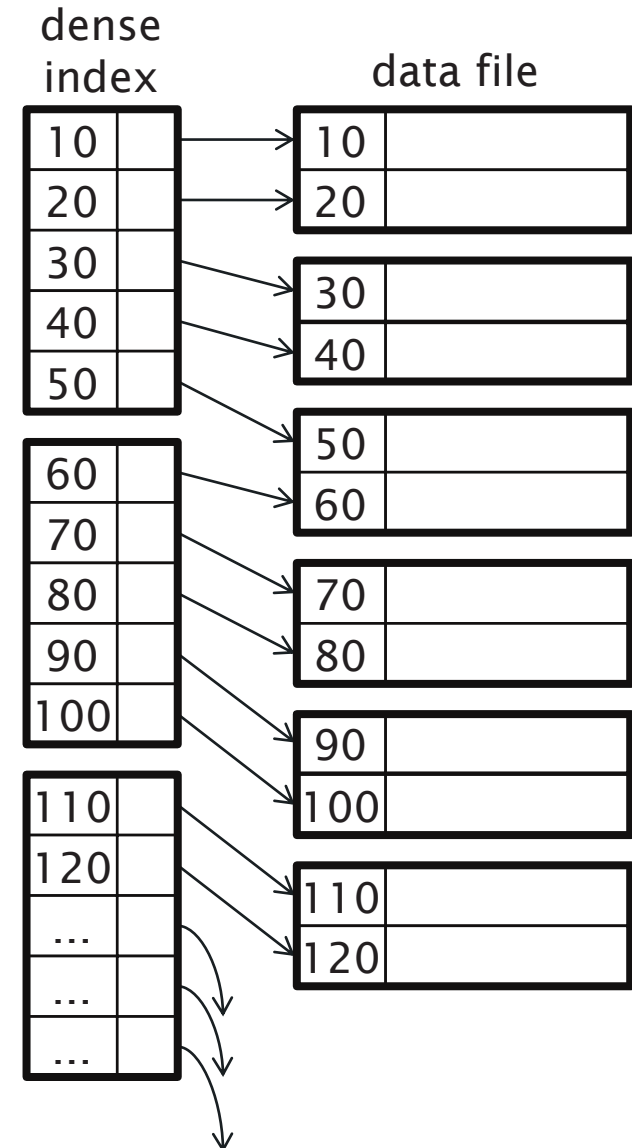
Dense Index

- Sequence of blocks holding only keys and pointers to records
- One key/pointer pair for every **record** in data file
- Blocks of index are in same order as those of the data file
- Key-pointer pair much smaller than record



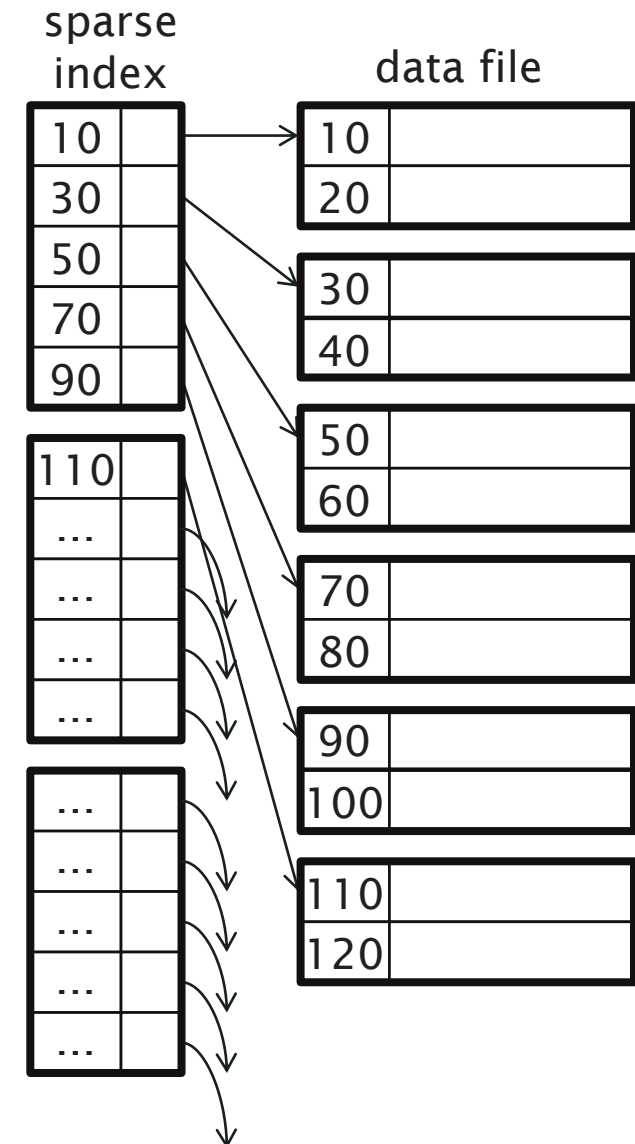
Dense Index

- Fewer blocks than data file, fewer disk accesses
- Keys are sorted, so can use binary search
- Can keep in main memory if small enough (no disk accesses)



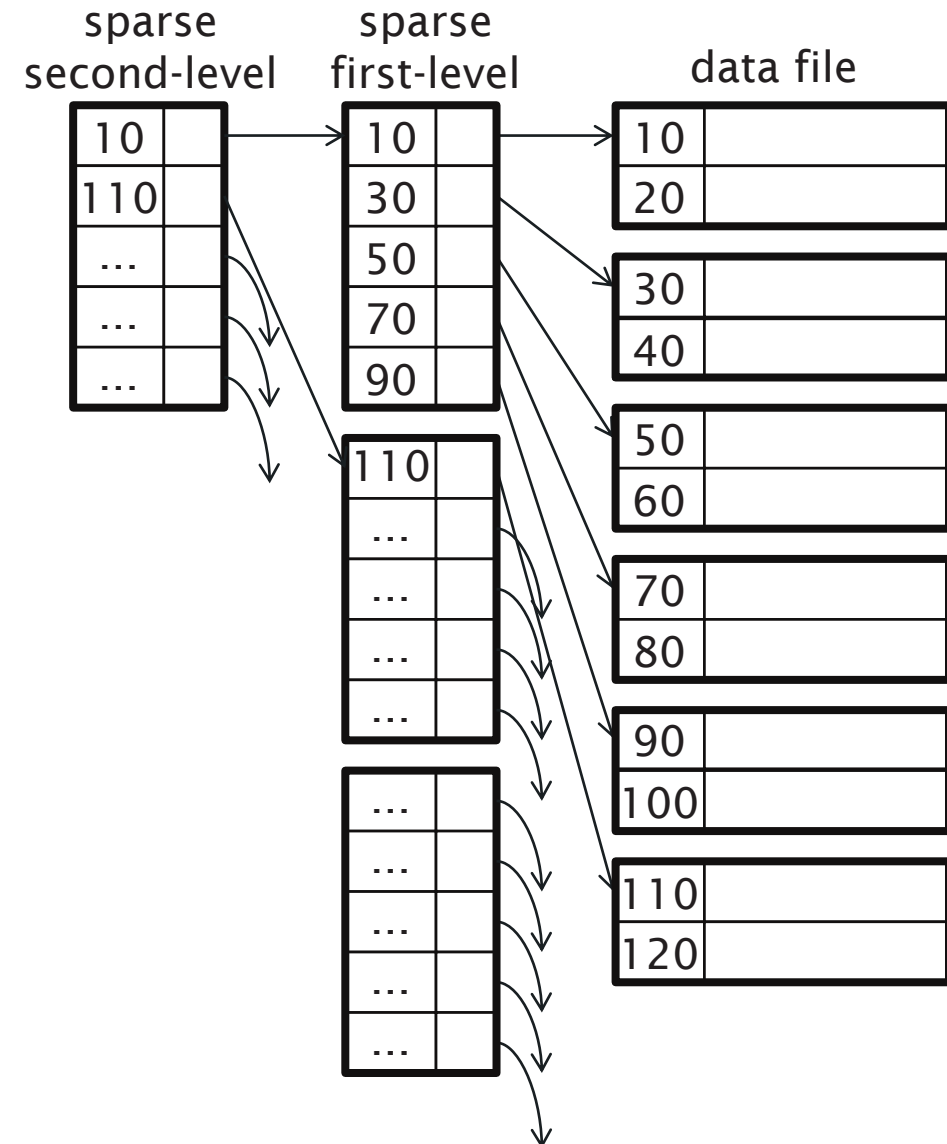
Sparse Index

- One key/pointer pair for every **block** in data file
- Can only be used if data file is sorted by search key
- Uses less space than dense index



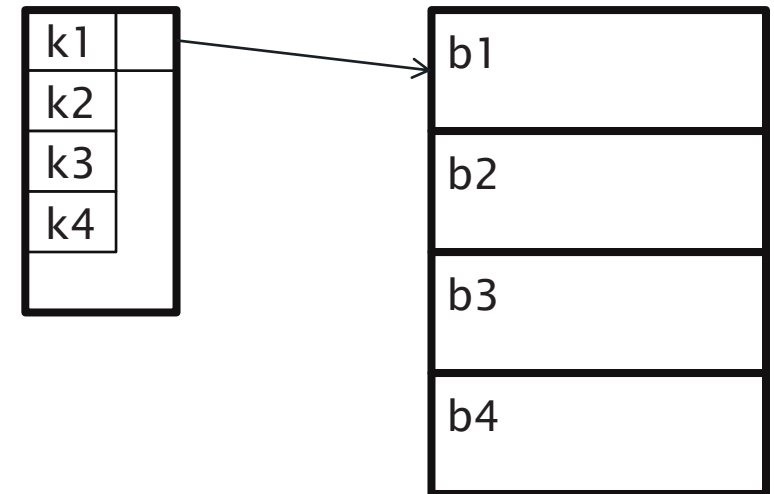
Multi-level Index

- Index file may cover many blocks
- May still need many disk accesses
- Use sparse index over the first index
 - Can't be a dense index (would use the same number of blocks as the index being indexed)
- Can create a third level index, but in general prefer B-trees



Notes on pointers:

- Block pointers (as used in sparse indexes) can be smaller than record pointers (used in dense indexes)
 - Physical record pointers consist of a block pointer and an offset
- If file is contiguous, then we can omit pointers
 - Compute offset from block size and key position
 - e.g. assuming 1kB per block and a pointer to block with key k1, to get block with key k3, use offset of $(3-1)*1 = 2\text{kB}$



Sparse vs. Dense Tradeoff

Sparse:

- Less index space per record can keep more of index in memory
- Better for insertions

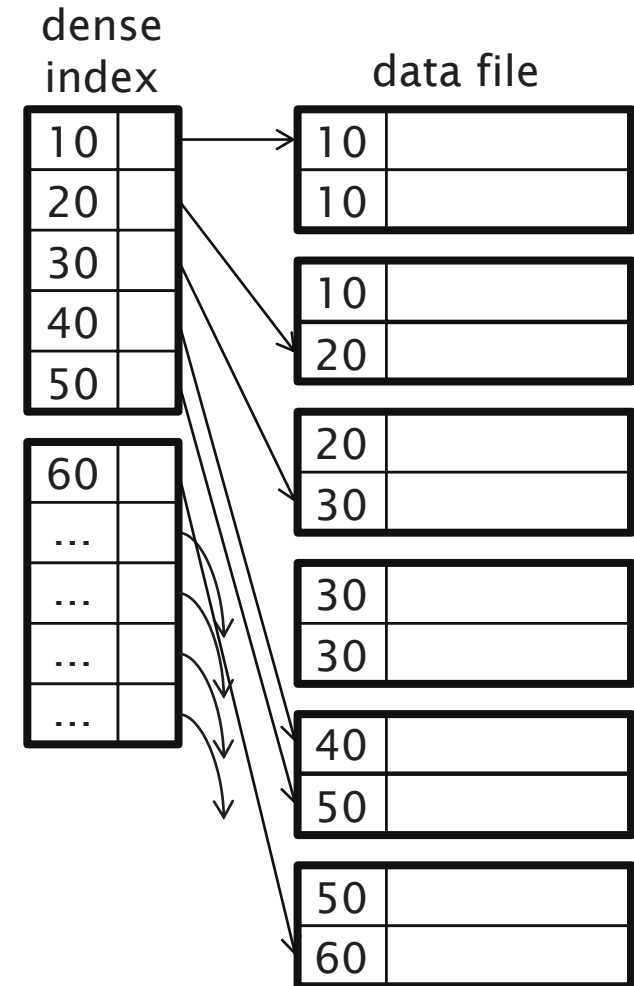
Dense:

- Can tell if a record exists without accessing file
- Needed for secondary indexes

Duplicate Keys

Dense index approach #2

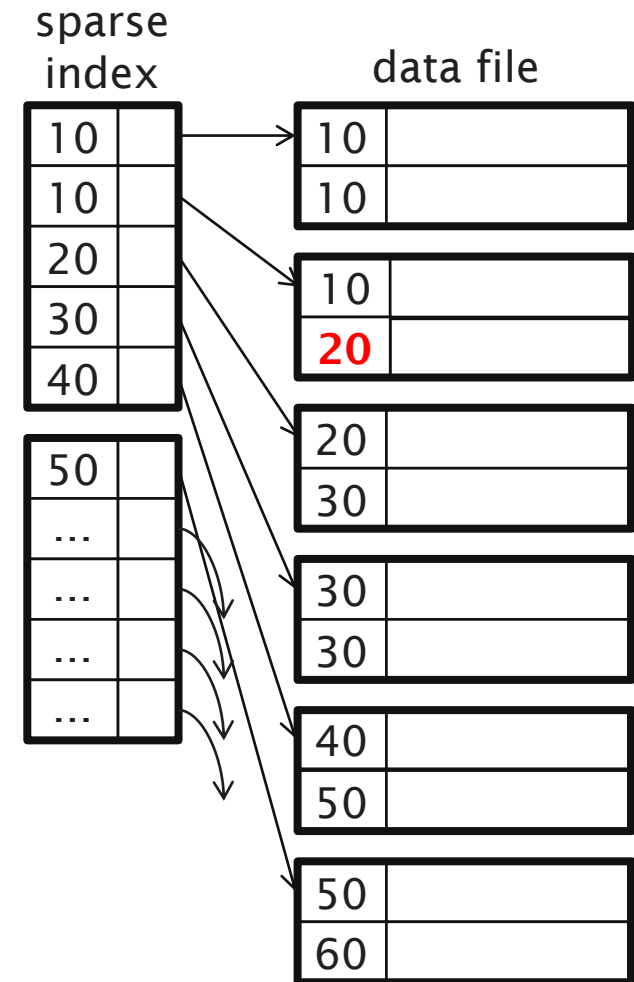
- Point at the first record with a given value
- better approach?
(smaller index)



Duplicate Keys

Sparse index approach #1

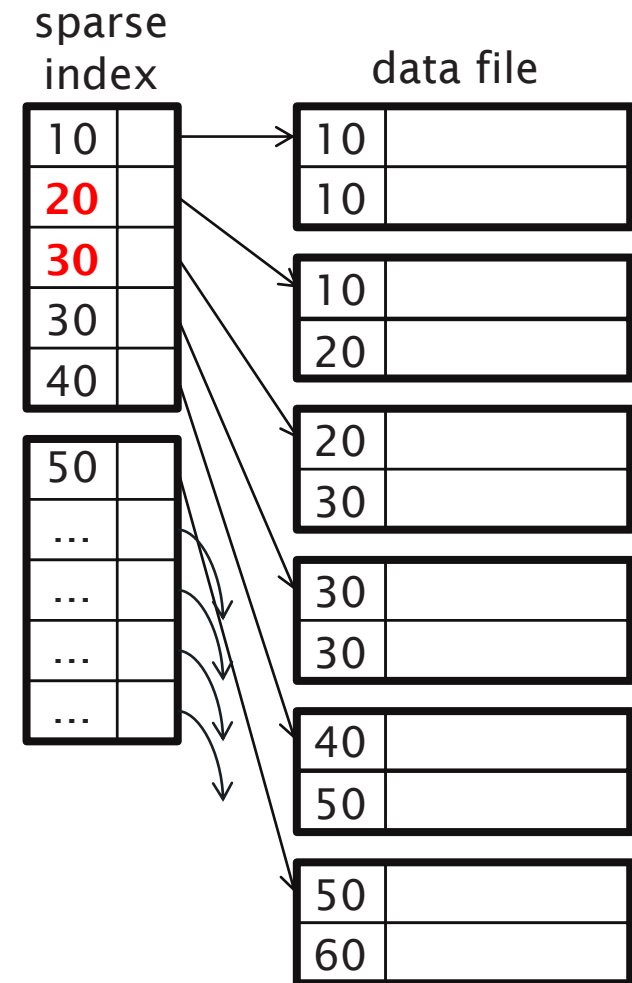
- Searching for (e.g.) 20 will give unexpected results



Duplicate Keys

Sparse index approach #2

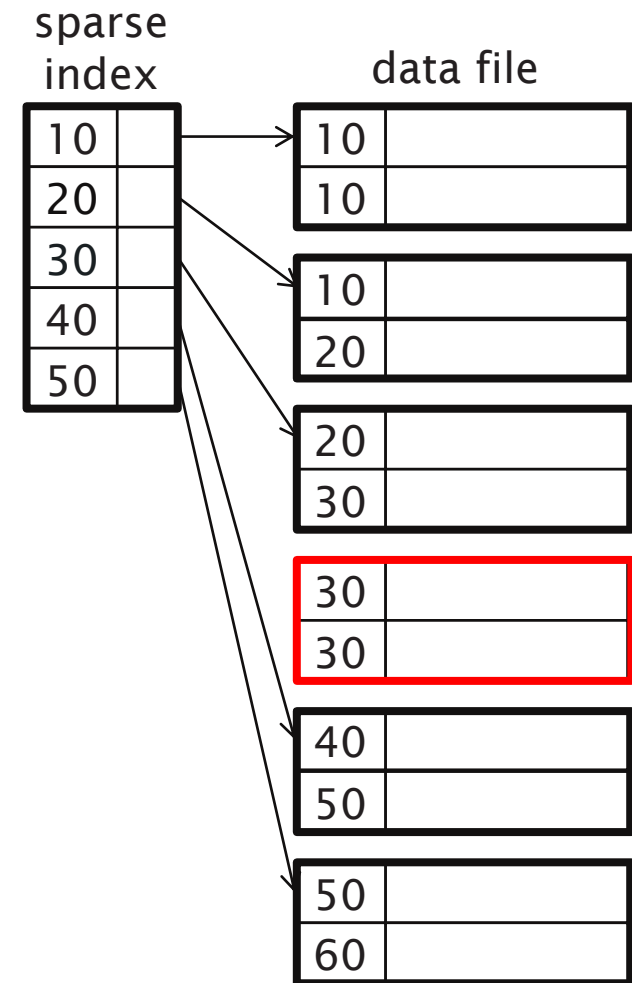
- Index contains first *new* key from each block



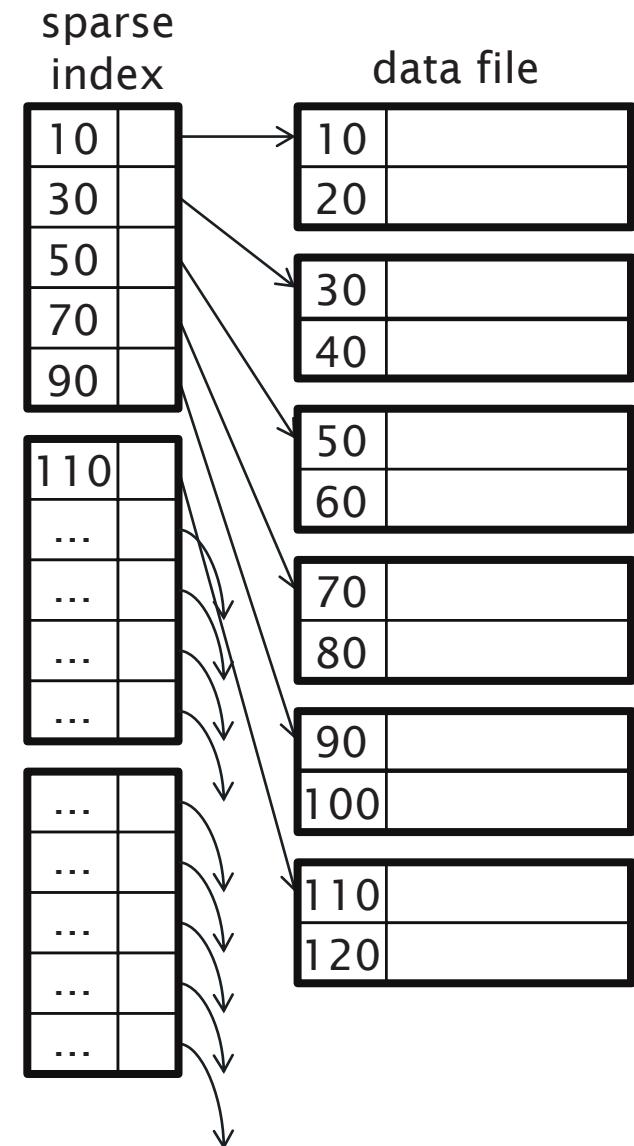
Duplicate Keys

Sparse index approach #2

- Can we exclude sequences of blocks with repeated keys?
- Point only to *first* instance of each value

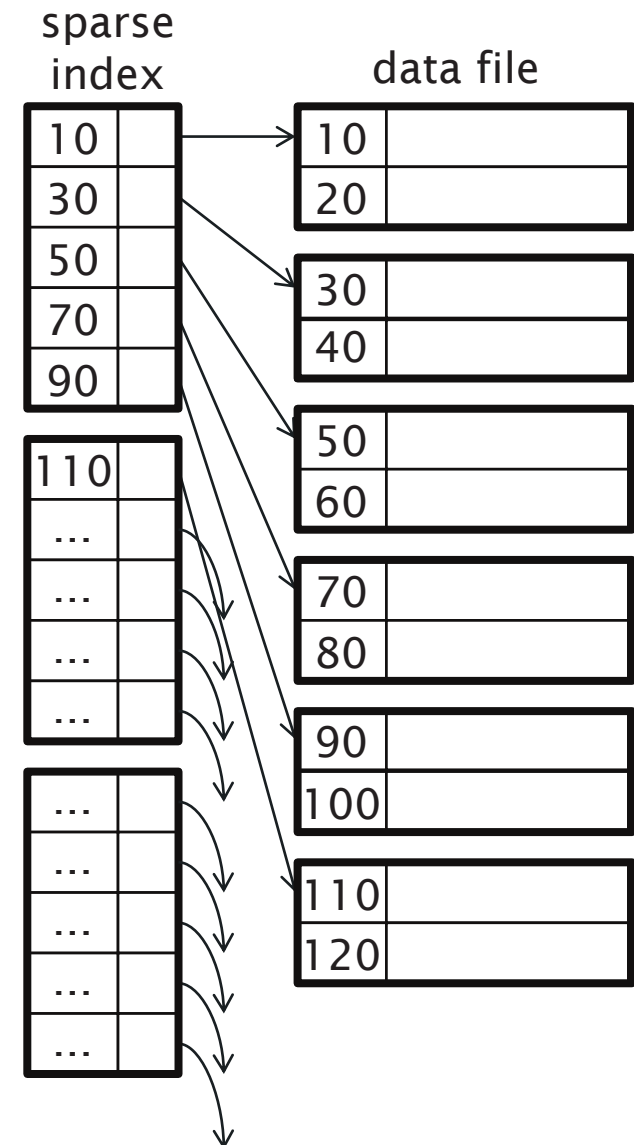


Deletion from Sparse Index



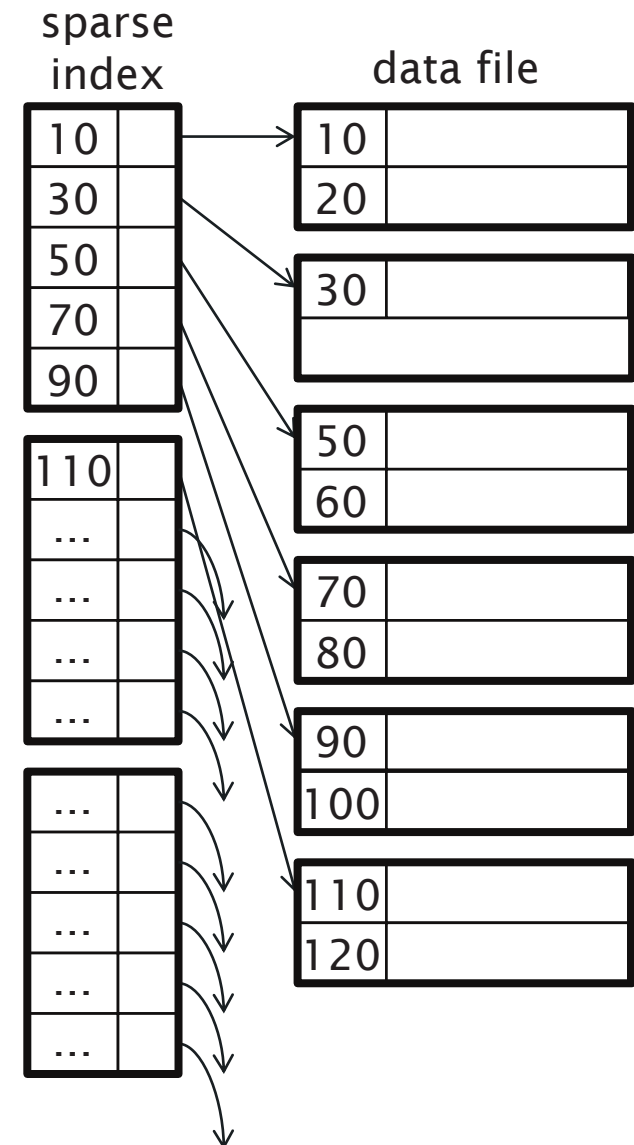
Deletion from Sparse Index

- Delete record 40



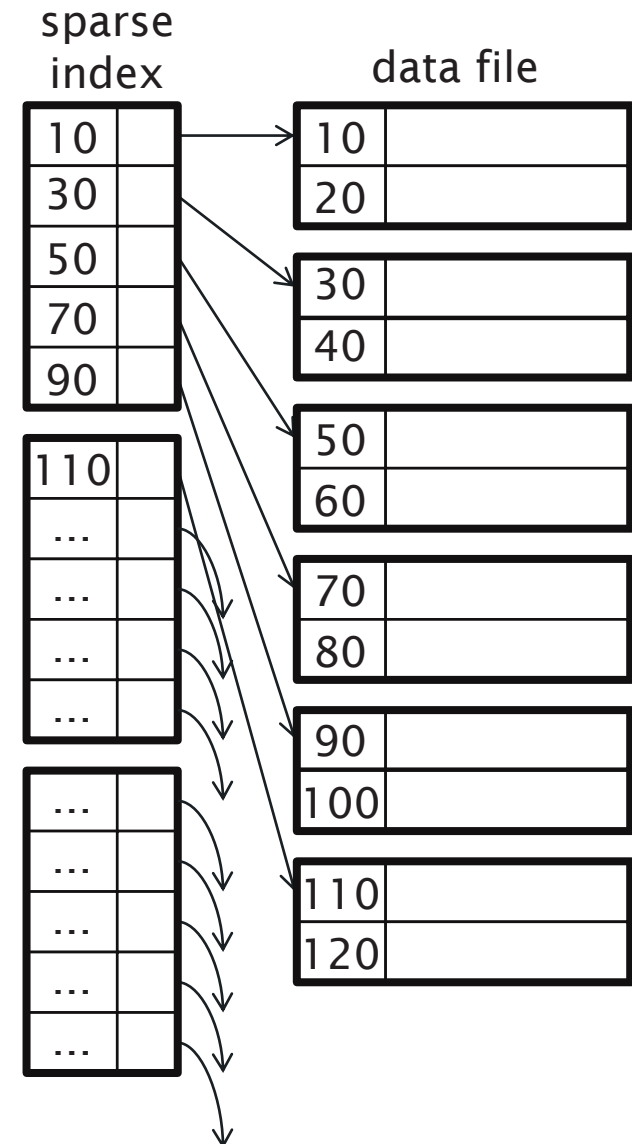
Deletion from Sparse Index

- Delete record 40



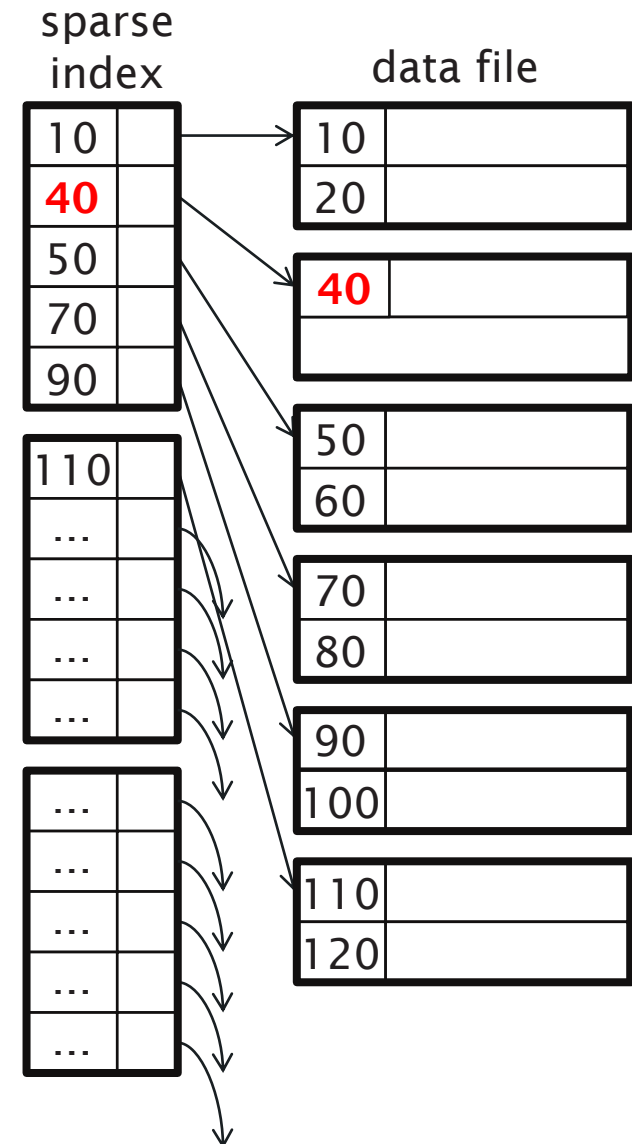
Deletion from Sparse Index

- Delete record 30
 - Delete record 30 from data file and reorder block
 - Update entry in index



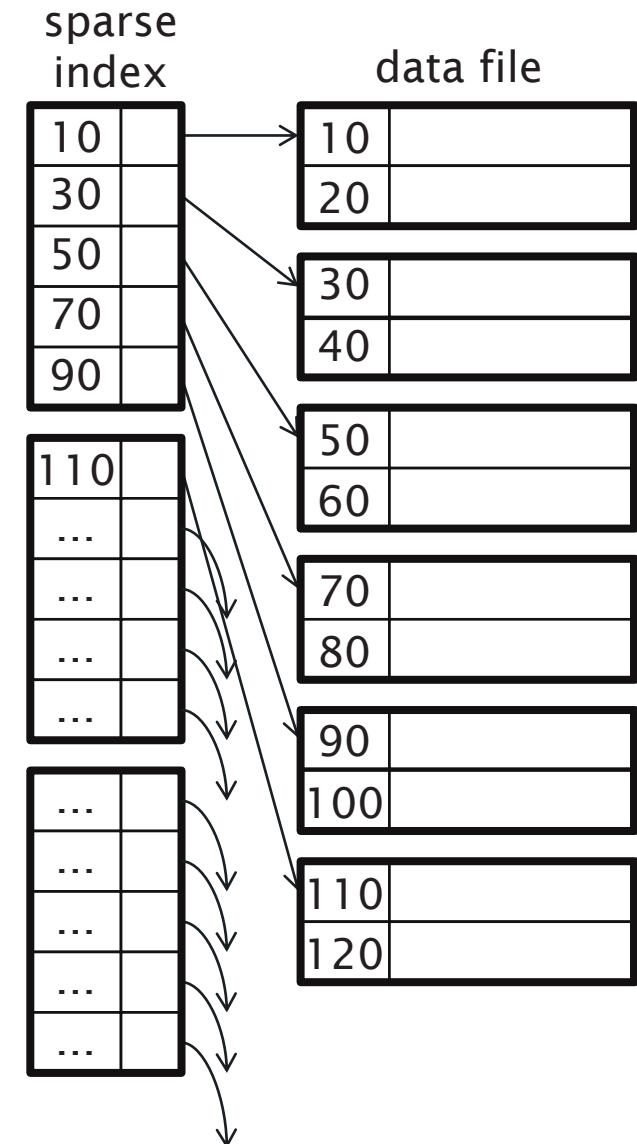
Deletion from Sparse Index

- Delete record 30
 - Delete record 30 from data file and reorder block
 - Update entry in index



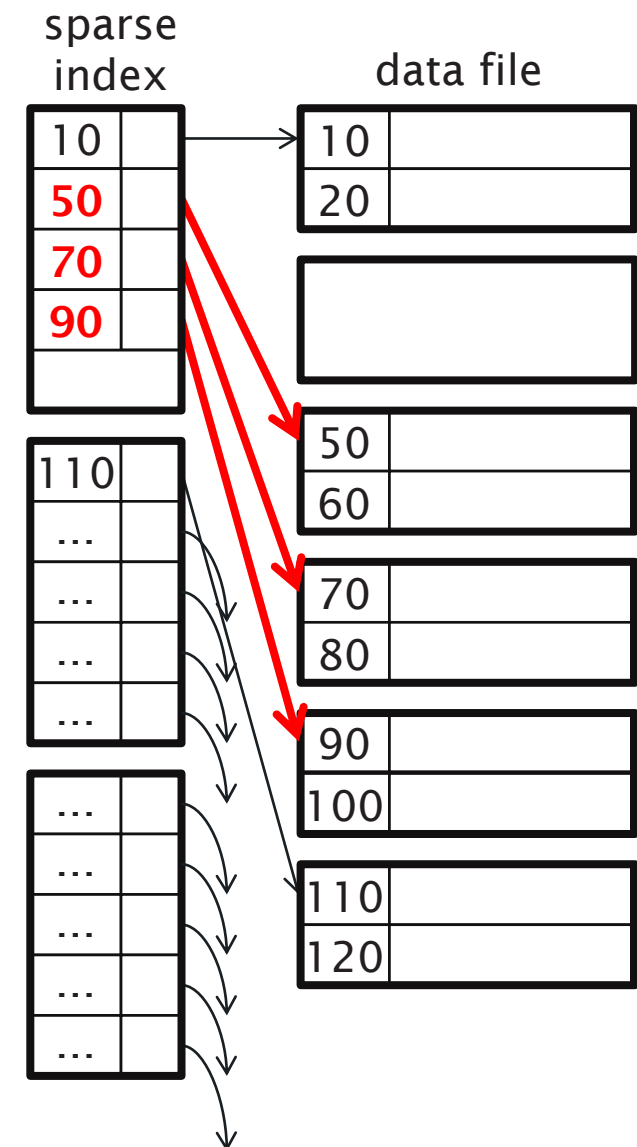
Deletion from Sparse Index

- Delete records 30 and 40
 - Delete records from data file
 - Update index



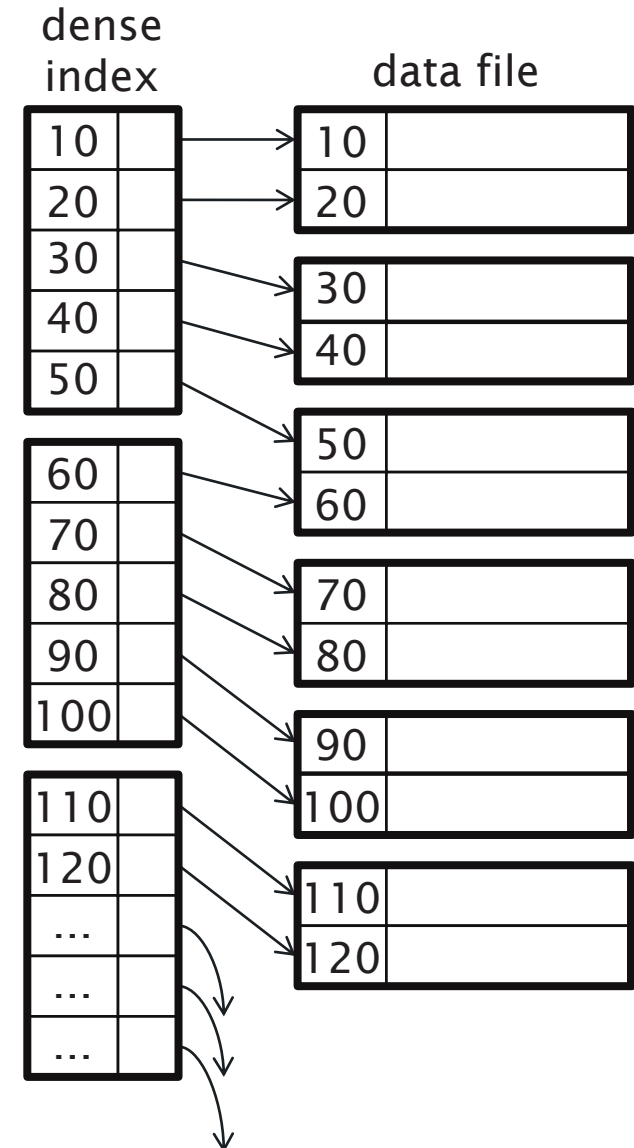
Deletion from Sparse Index

- Delete records 30 and 40
 - Delete records from data file
 - Update index



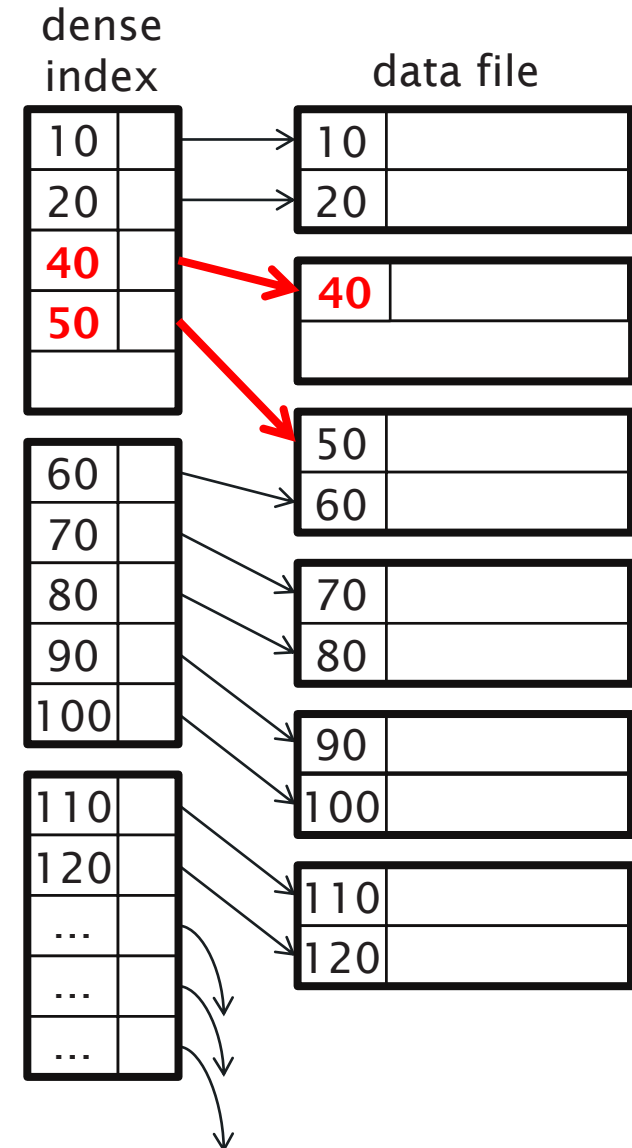
Deletion from Dense Index

- Delete record 30
 - Delete record from data file
 - Remove entry from index and update index

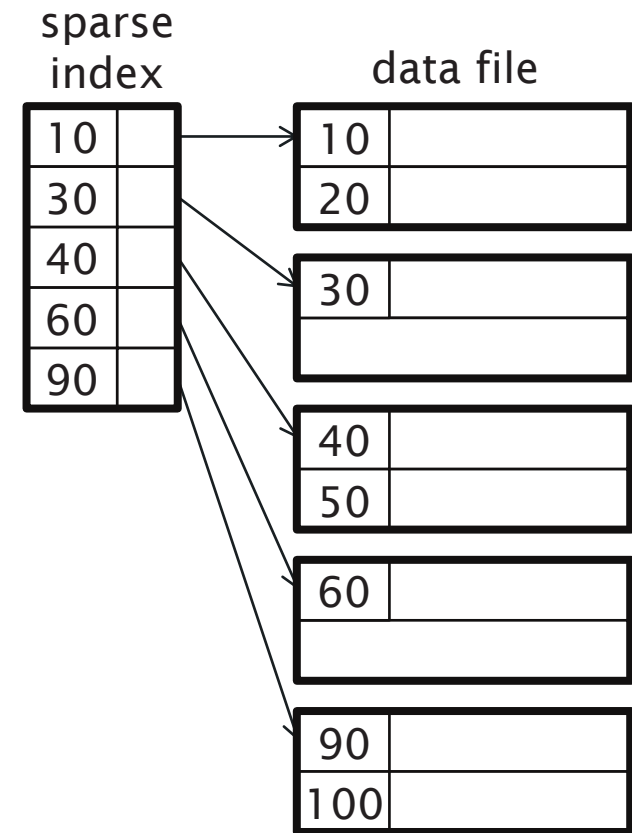


Deletion from Dense Index

- Delete record 30
 - Delete record from data file
 - Remove entry from index and update index

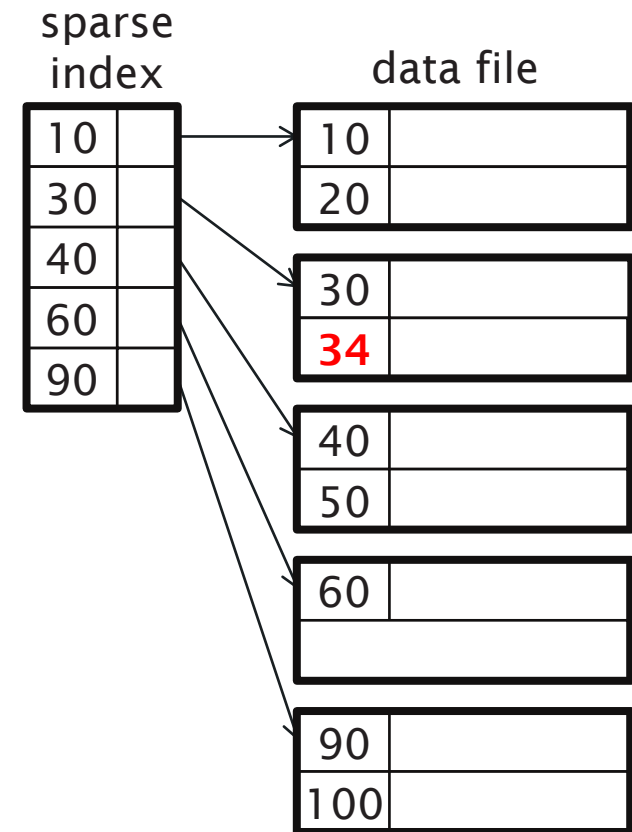


Insertion into Sparse Index



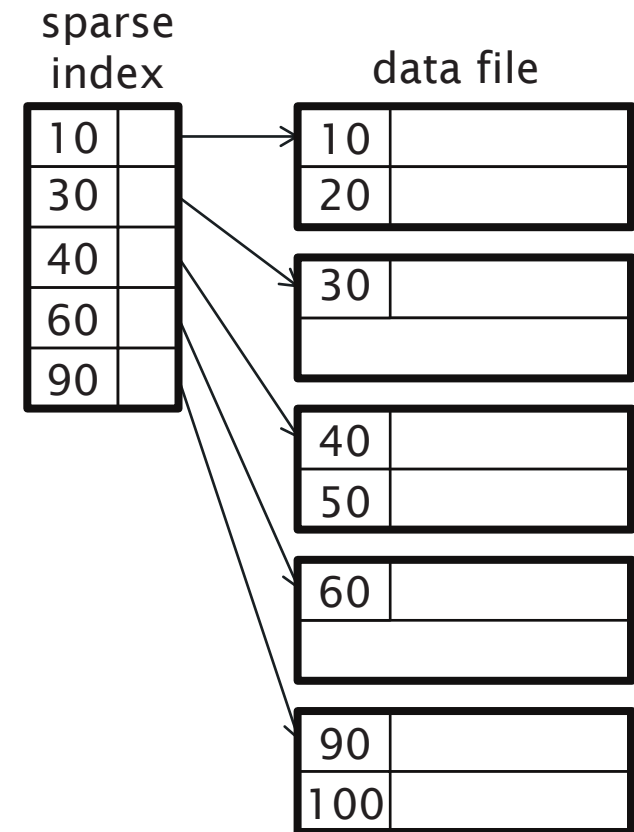
Insertion into Sparse Index

- Insert record 34
 - Easy! We have free space in the right block of the data file



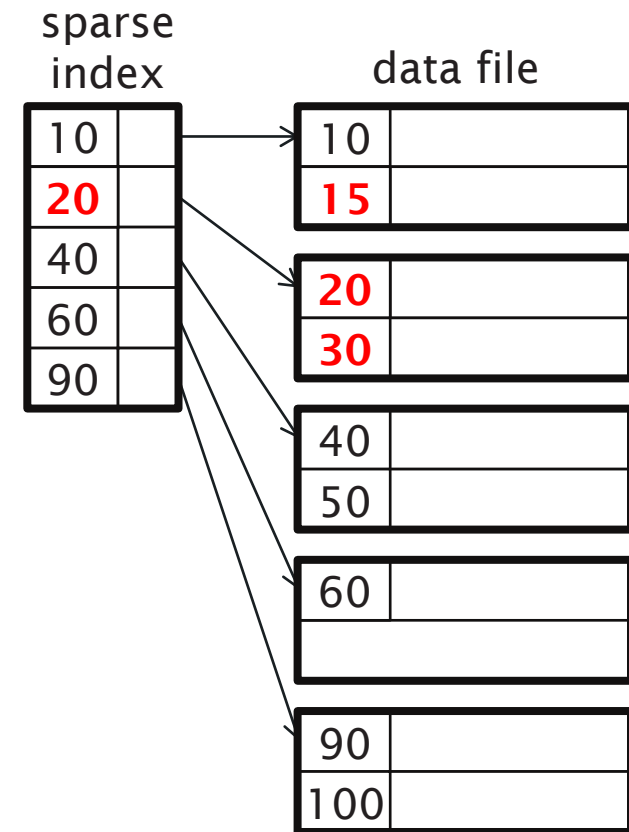
Insertion into Sparse Index

- Insert record 15
 - Add to data file and immediately reorganise
 - Update index



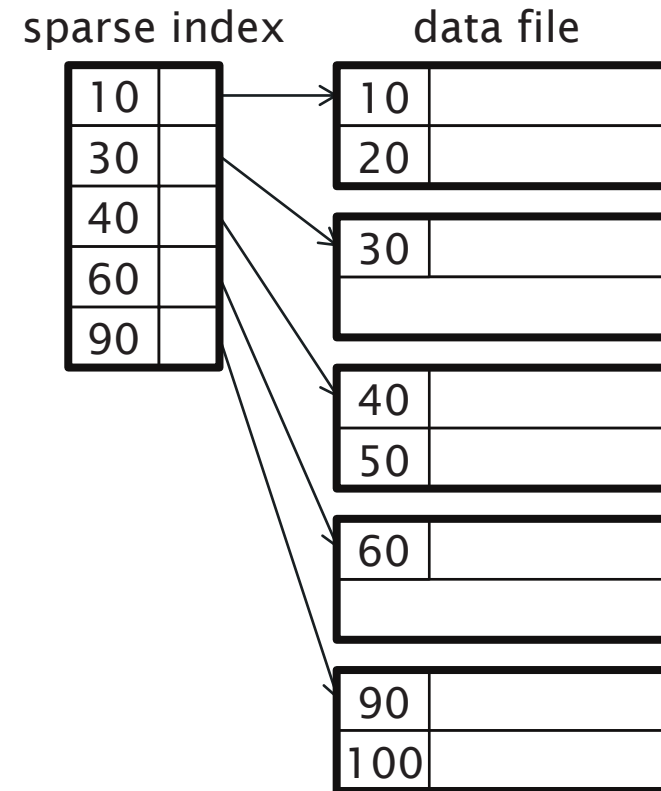
Insertion into Sparse Index

- Insert record 15
 - Add to data file and immediately reorganise
 - Update index
- Alternatively:
 - Insert new block (chained file)
 - Update index



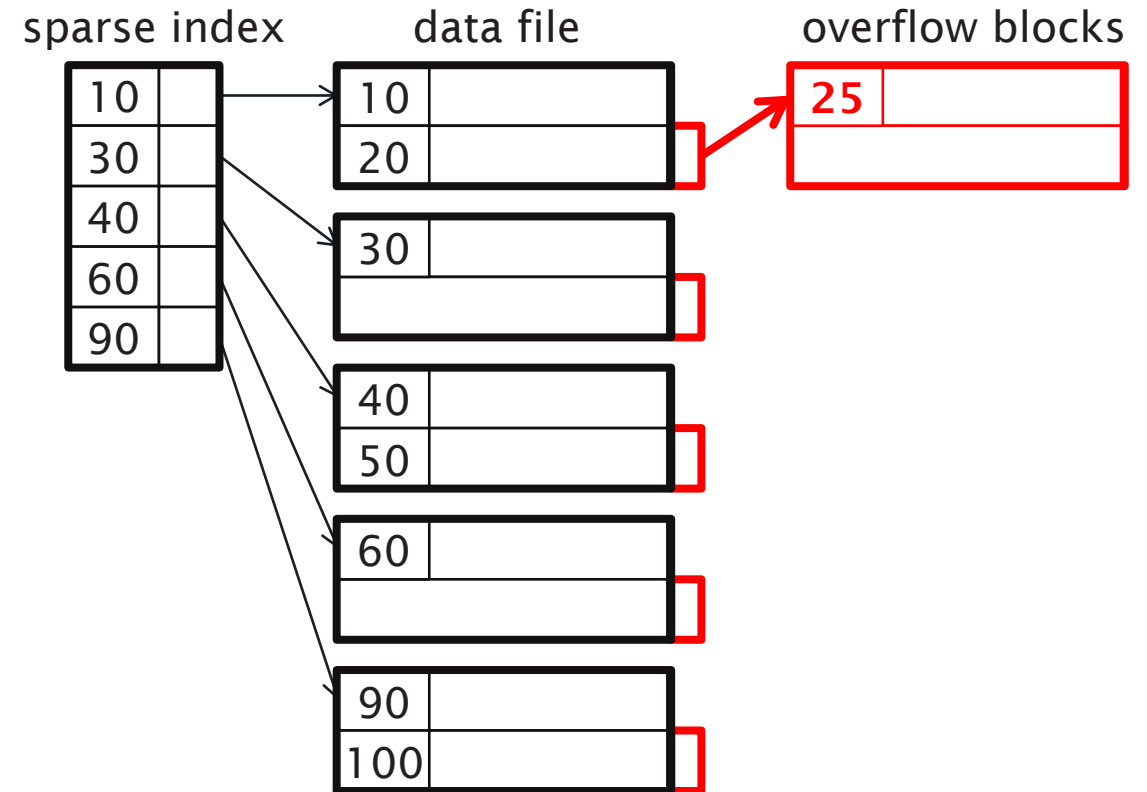
Insertion into Sparse Index

- Insert record 25
 - Block is full, so add to overflow block
 - Reorganise later...



Insertion into Sparse Index

- Insert record 25
 - Block is full, so add to overflow block
 - Reorganise later...



Secondary Indexes

- Unlike a primary index, does not determine placement of records in data file
- Location (order) of records may have been decided by a primary index on another field
- Secondary indexes are always dense
- Pointers are record pointers, not block pointers

data file

20	
40	

10	
80	

70	
50	

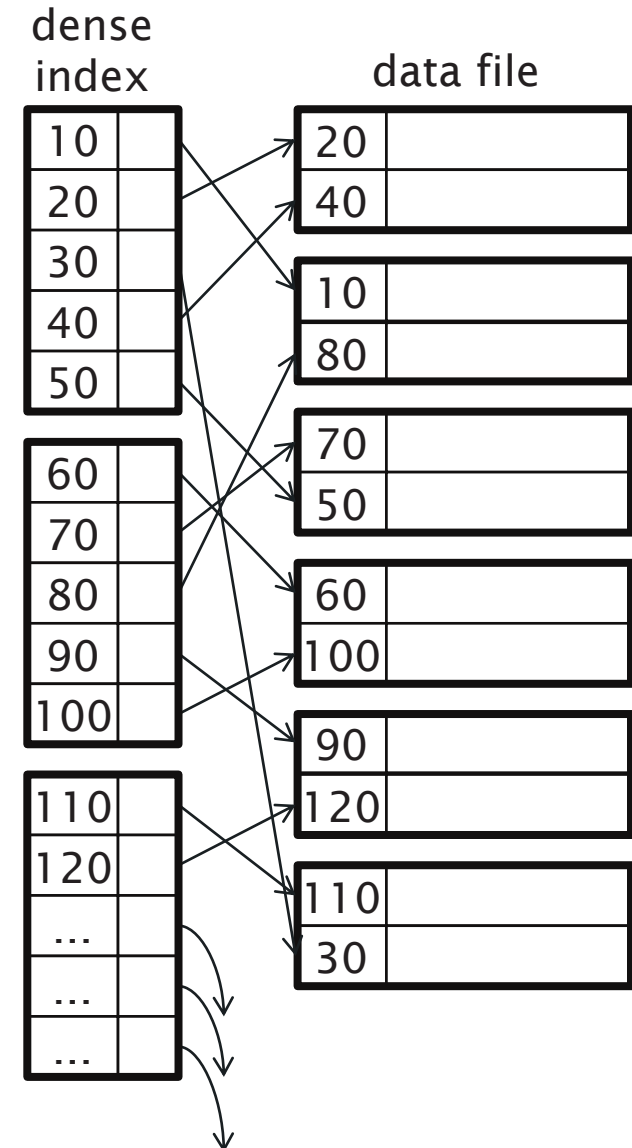
60	
100	

90	
120	

110	
30	

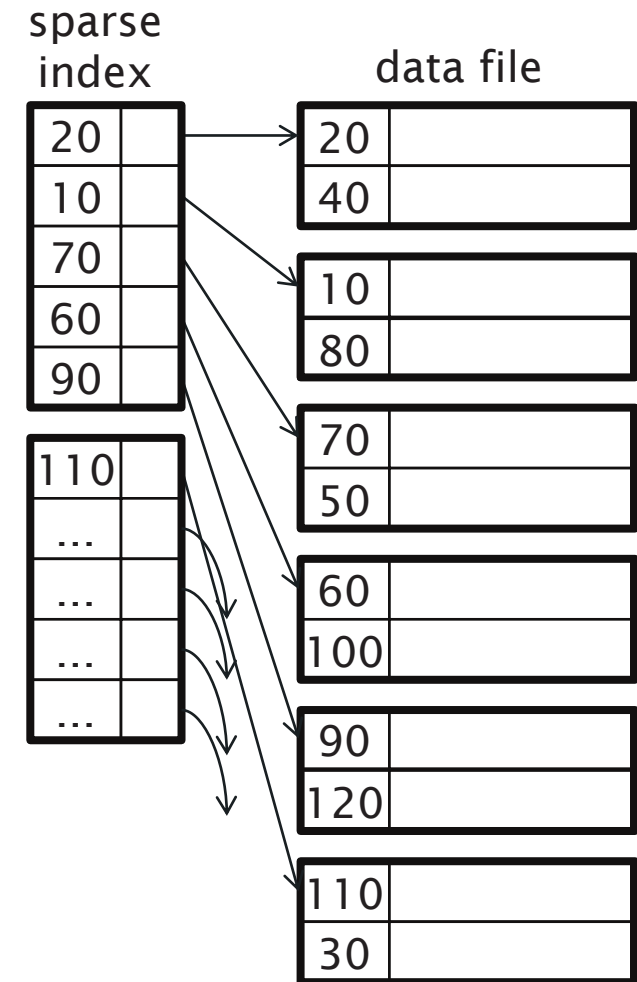
Secondary Indexes

- Unlike a primary index, does not determine placement of records in data file
- Location (order) of records may have been decided by a primary index on another field
- Secondary indexes are always dense
- Pointers are record pointers, not block pointers



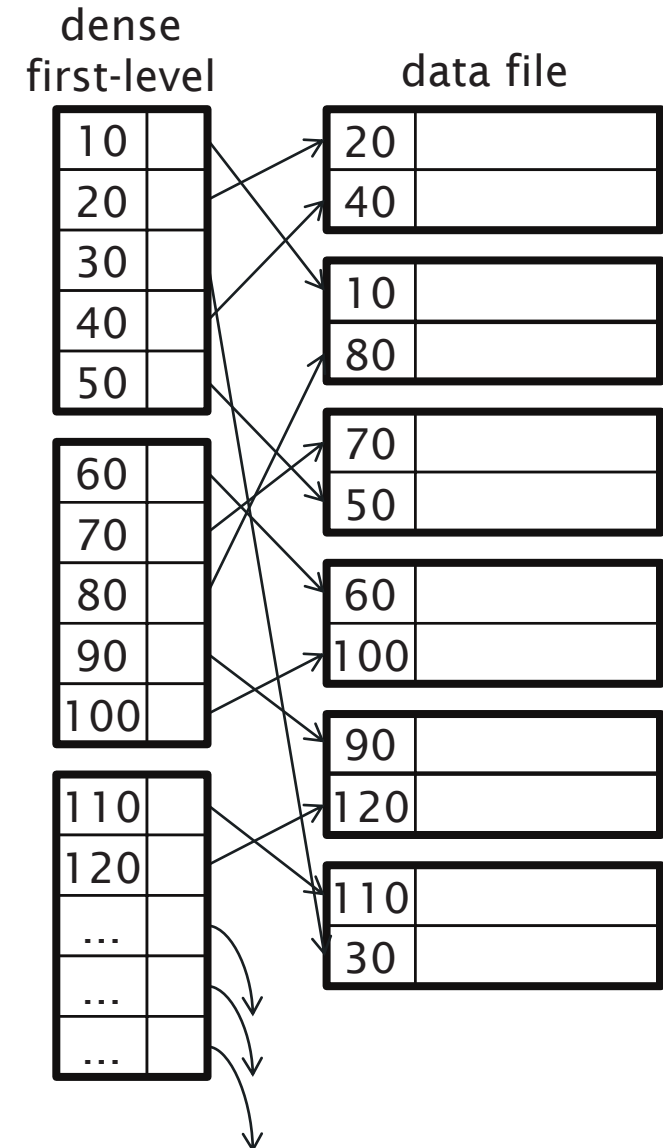
Secondary Indexes

- Sparse secondary indexes make no sense



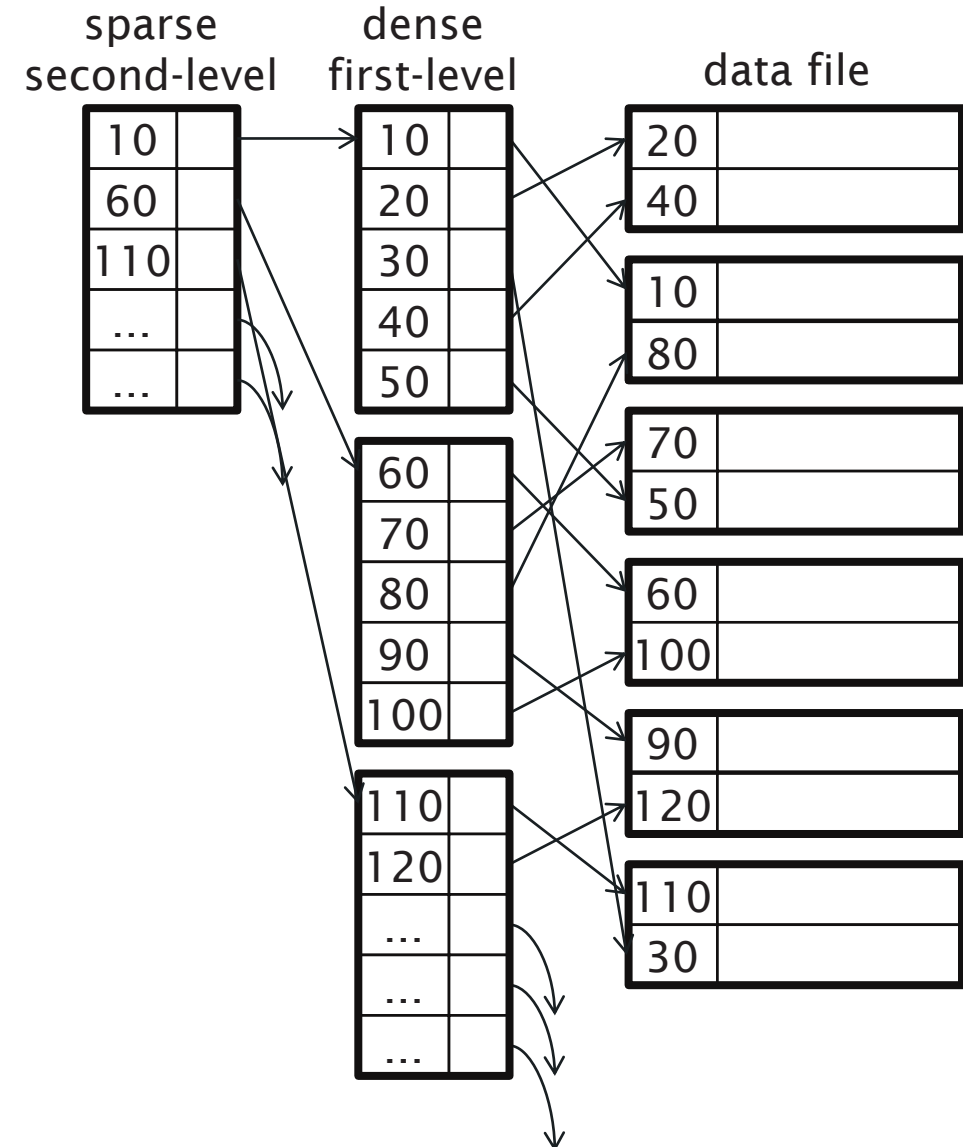
Secondary Indexes

- May have higher levels of sparse indexes above the dense index



Secondary Indexes

- May have higher levels of sparse indexes above the dense index



Duplicate values

- Secondary indexes need to cope with duplicate values in the data file

data file

20	
10	

20	
40	

10	
40	

30	
10	

20	
10	

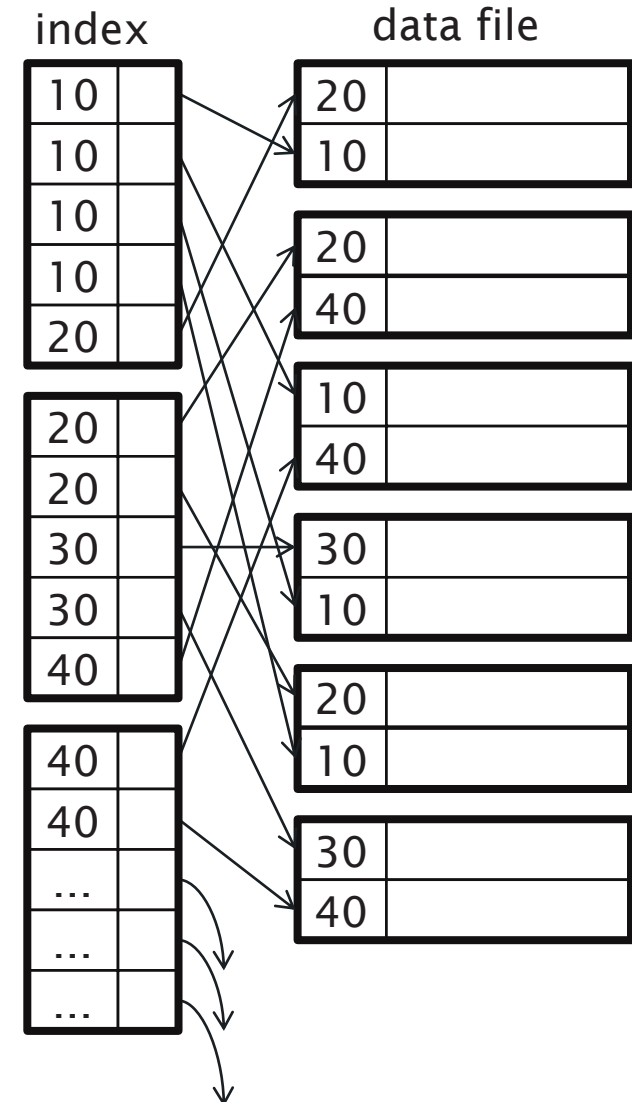
30	
40	

Duplicate values

Solution #1: repeated entries

Problems

- excess disk space
- excess search time

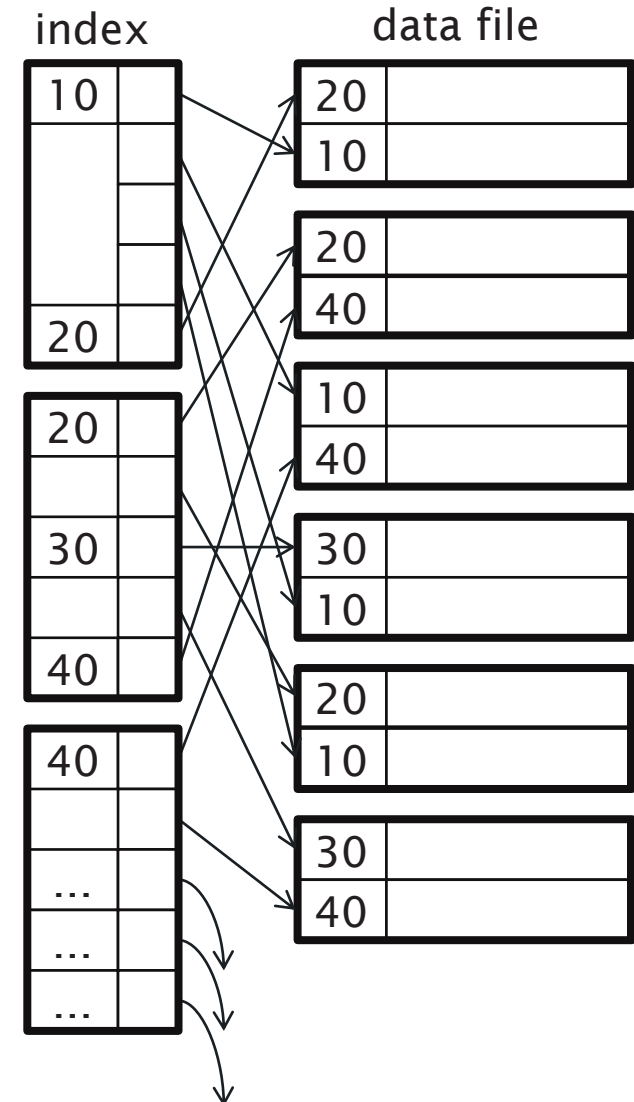


Duplicate values

Solution #2: drop repeated keys

Problems

- variable size records in index

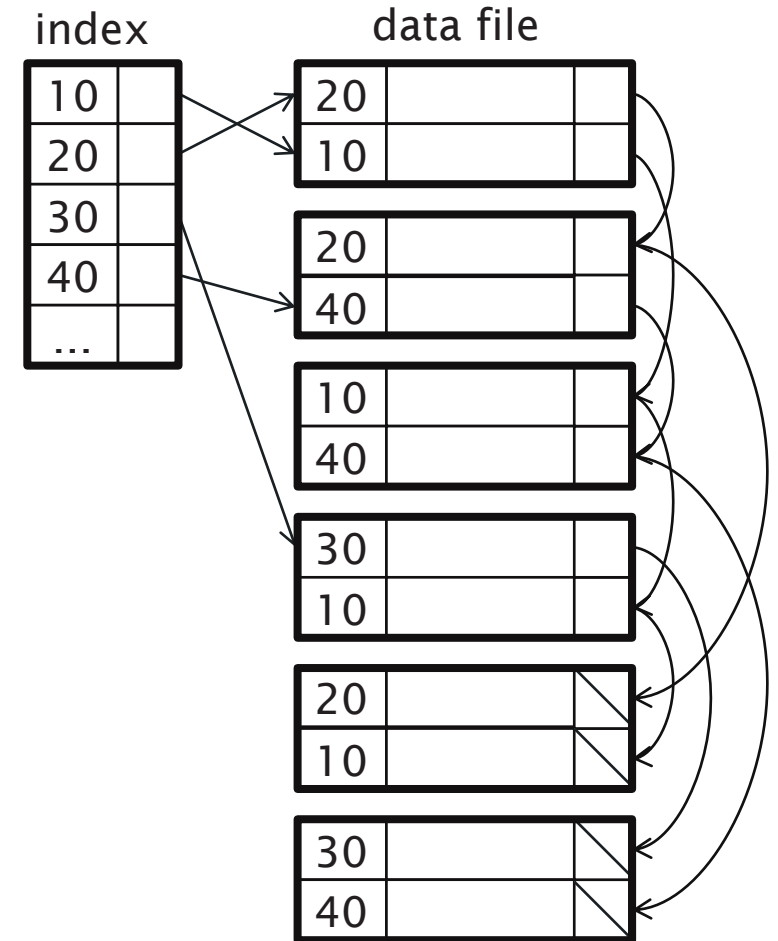


Duplicate values

Solution #3: chain records with same key

Problems

- need to add fields to records
- need to follow chain

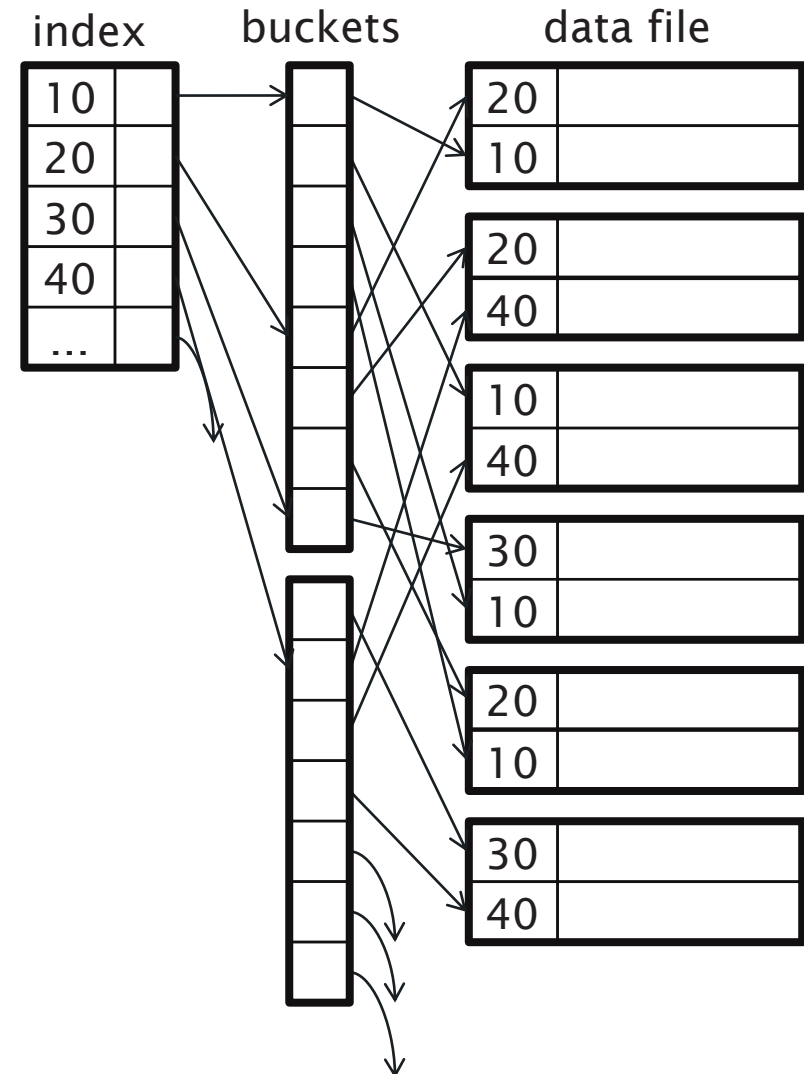


Duplicate values

Solution #4: indirection via buckets of pointers

Advantages

- If we have multiple secondary indexes on a relation, we can calculate conjunctions by taking intersections of buckets
- Don't need to examine data file!



Conventional indexes

Advantages:

- Simple
- Index is sequential file and good for scans

Disadvantages:

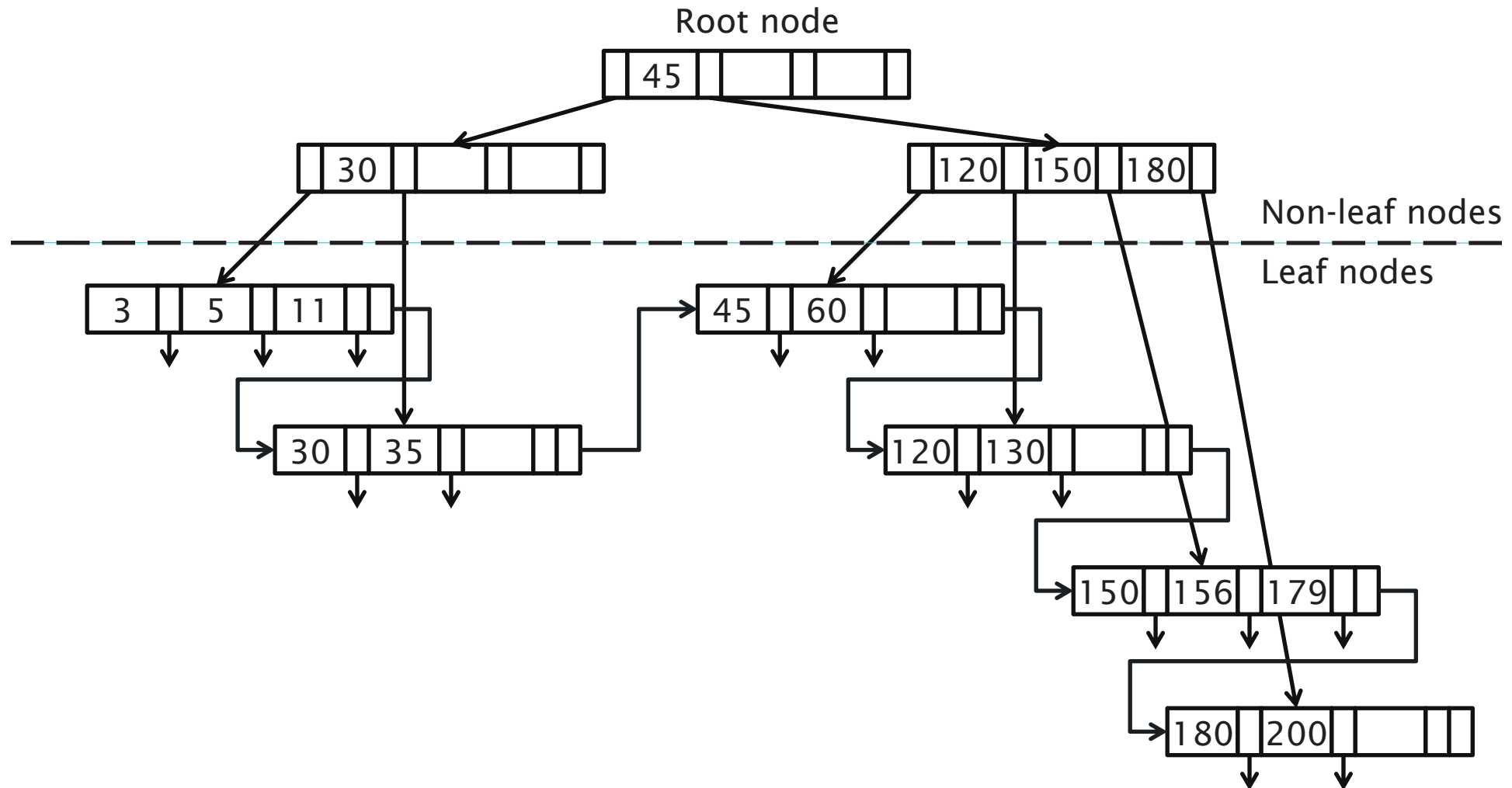
- Inserts expensive, and/or
- Lose sequentiality & balance

B+trees

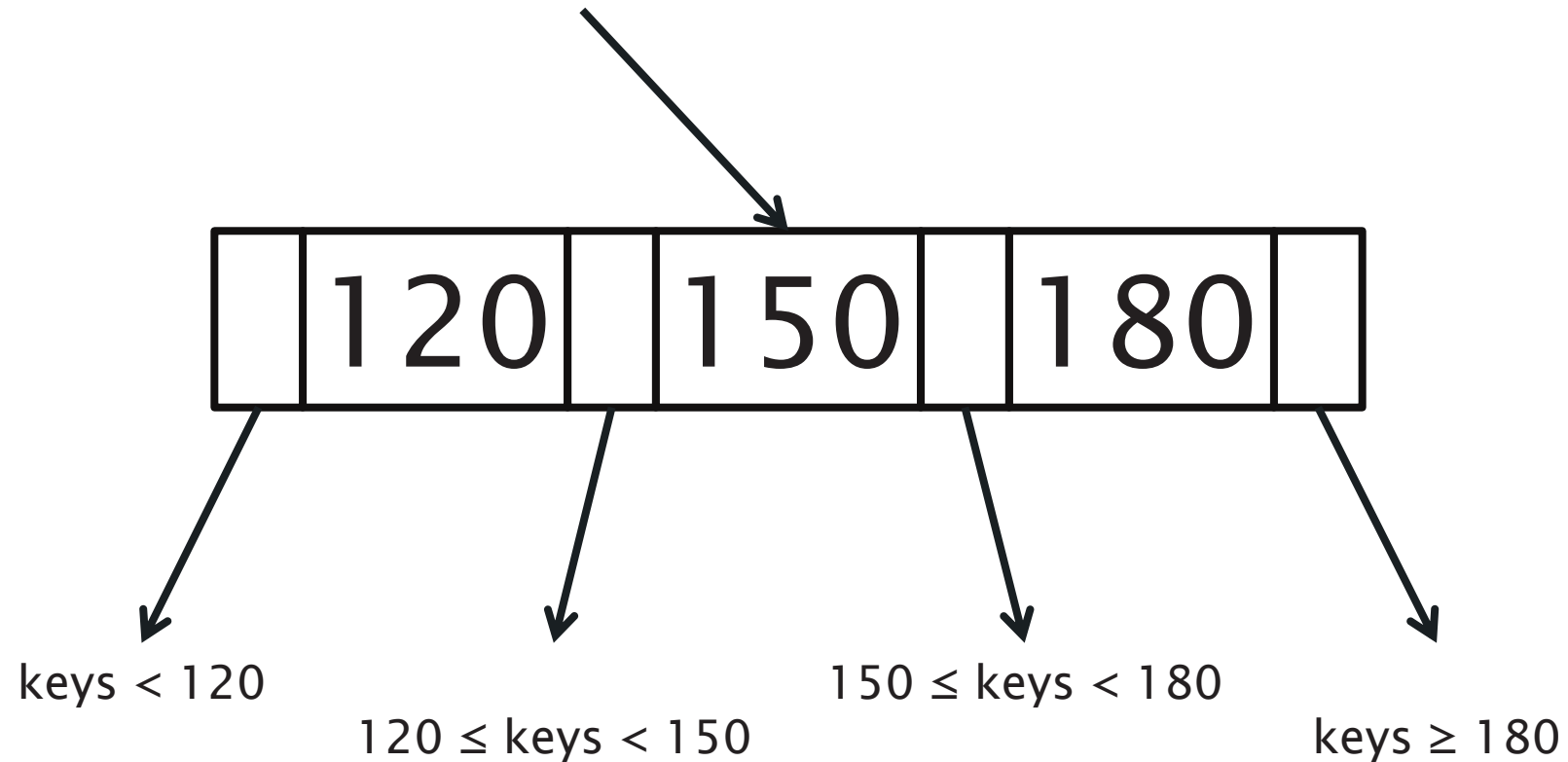
B+trees

- The most widely used tree-structured indexes
- Balanced multi-way tree
 - Yields consistent performance
 - Sacrifices sequentiality

B+tree example



Example non-leaf node

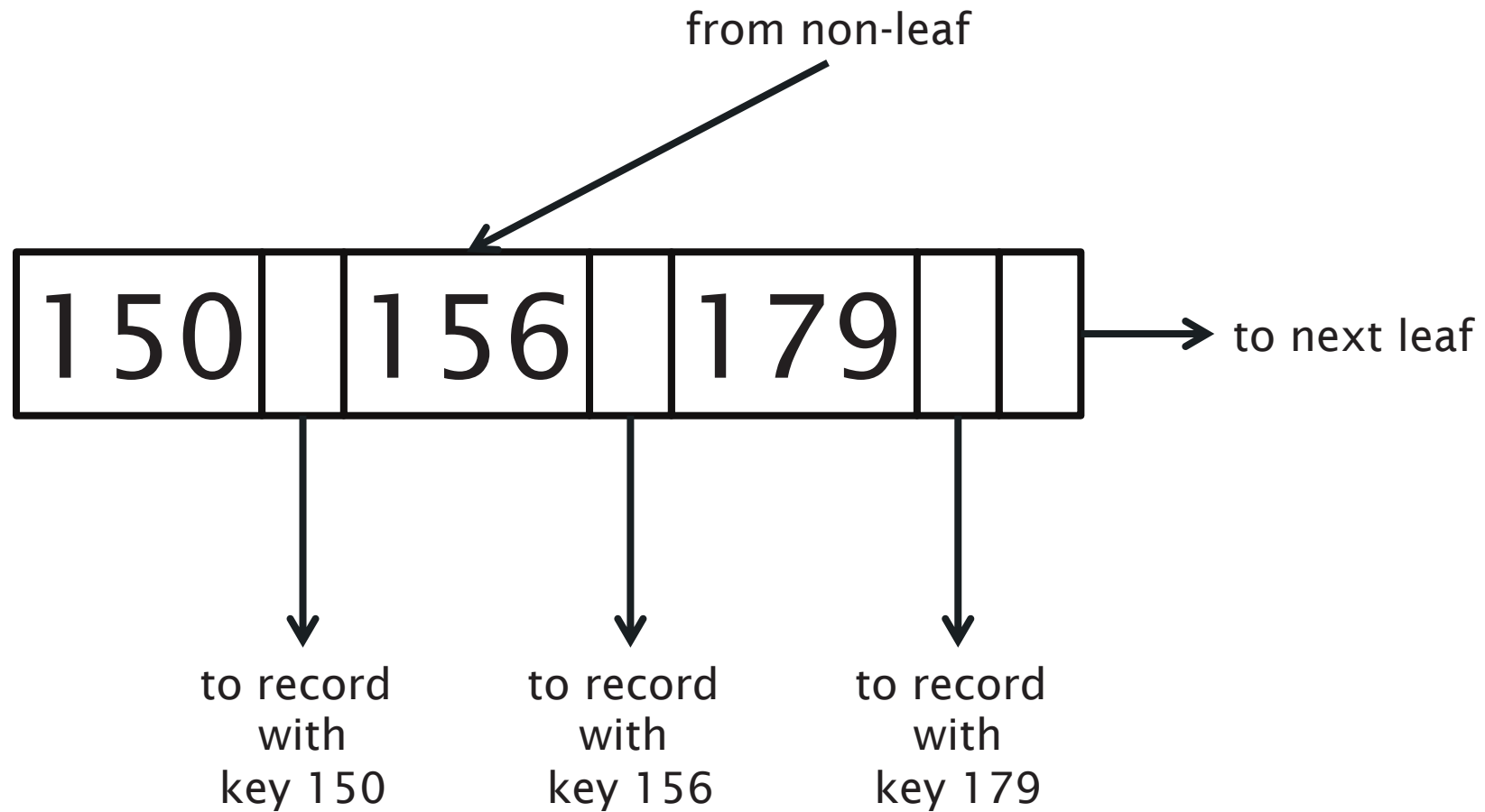


Non-leaf nodes

Root node typically kept in memory

- Entrance point to index – used as frequently as any other node
- Some nodes from second level may also be kept in memory

Example leaf node



Leaf nodes

If the index is a primary index

- Leaf nodes are records containing data, stored in the order of the primary key
- The index provides an alternative to a sequential scan

If the index is a secondary index

- Leaf nodes contain pointers to the data records
- Data can be accessed in the sequence of the secondary key
- A secondary index can point to any sort of data file, for example one created by hashing

Node size

Each node is of fixed size and contains

- n keys
- $n+1$ pointers

non-leaf



leaf



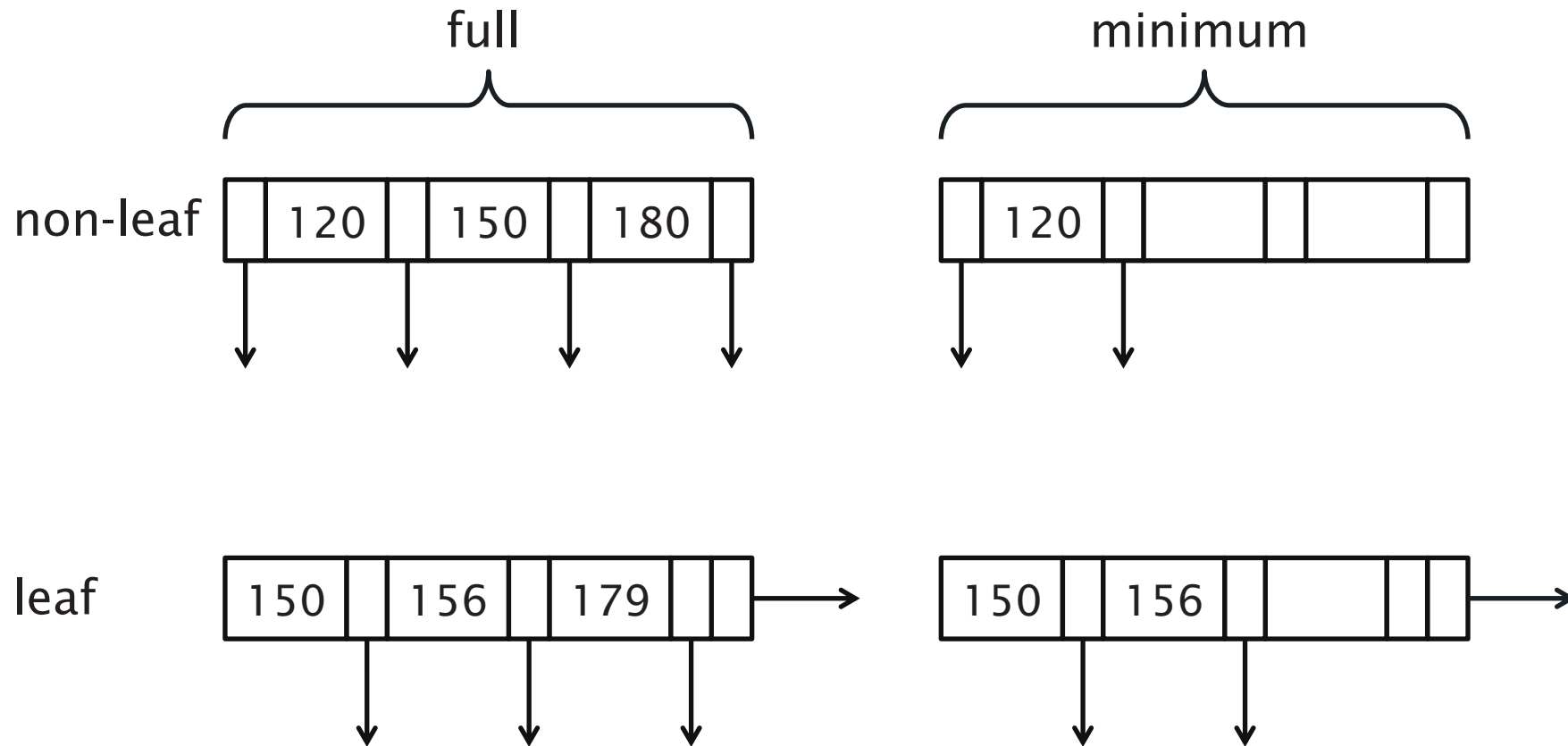
Minimum nodes

Don't want nodes to be too empty (efficient use of space)

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

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

Minimum node examples (n=3)



B+tree rules

1. All leaves same distance from root (balanced tree)
2. Pointers in leaves point to records except for “sequence pointer”
3. Number of pointers/keys for B+tree of order n:

	max ptrs	max keys	min ptrs to data	min keys
Non-leaf	n+1	n	$\lceil (n+1)/2 \rceil$	$\lceil (n+1)/2 \rceil - 1$
Leaf	n+1	n	$\lfloor (n+1)/2 \rfloor$	$\lfloor (n+1)/2 \rfloor$
Root	n+1	n	1	1

B+tree arithmetic example

First, some parameters:

- block size 4kb, of which:
b = 4000 bytes available for storage of records
- key length
k = 10 bytes
- record length
r = 100 bytes (including the key)
- block pointer
p = 6 bytes

B+tree arithmetic example

A leaf node in a primary index can accommodate l_p records, where $l_p = \lfloor (b-p)/r \rfloor = 39$ records

A leaf node in a secondary index can accommodate l_s records, where $l_s = \lfloor (b-p)/(k+p) \rfloor = 249$ records

A non-leaf node could accommodate i entries, where $i = \lfloor (b-p)/(k+p) \rfloor = 249$ records

To allow for expansion, assume initial node occupancy of u , where $u = 0.6$

B+tree primary index

For a primary index (the leaf nodes hold the records):

- A non-leaf node initially points to

$$i * u = \text{blocks}$$

- Each leaf initially contains

$$lp * u = \text{records}$$

- 1 level of non-leaf nodes initially points to

$$(lp * u)(i * u) = \text{records}$$

- 2 levels of non-leaf nodes initially point to

$$(i * u)^2 = \text{blocks}$$

$$(lp * u)(i * u)^2 = \text{records}$$

B+tree primary index

For a primary index (the leaf nodes hold the records):

- A non-leaf node initially points to

$$i * u = 149 \text{ blocks}$$

- Each leaf initially contains

$$lp * u = 23 \text{ records}$$

- 1 level of non-leaf nodes initially points to

$$(lp * u)(i * u) = 3,427 \text{ records}$$

- 2 levels of non-leaf nodes initially point to

$$(i * u)^2 = 22,201 \text{ blocks}$$

$$(lp * u)(i * u)^2 = 510,623 \text{ records}$$

B+tree secondary index

For a secondary index (the leaf nodes hold record pointers):

- A non-leaf node initially points to

$$i * u = \text{blocks}$$

- A leaf node initially points at

$$l * u = \text{records}$$

- 1 level of non-leaf nodes initially points to

$$(l * u)(i * u) = \text{records}$$

- 2 levels of non-leaf nodes initially point to

$$(l * u)(i * u)^2 = \text{records}$$

B+tree secondary index

For a secondary index (the leaf nodes hold record pointers):

- A non-leaf node initially points to

$$i * u = 149 \text{ blocks}$$

- A leaf node initially points at

$$l * u = 149 \text{ records}$$

- 1 level of non-leaf nodes initially points to

$$(l * u)(i * u) = 22,201 \text{ records}$$

- 2 levels of non-leaf nodes initially point to

$$(l * u)(i * u)^2 = 3,307,949 \text{ records}$$

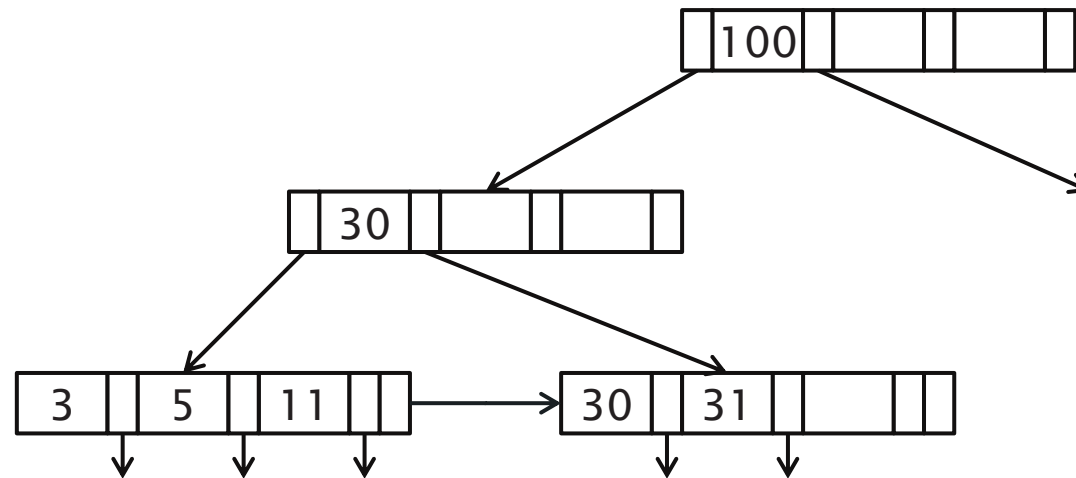
It is not normally necessary to go more than about three levels deep in the index

B+tree Insertion

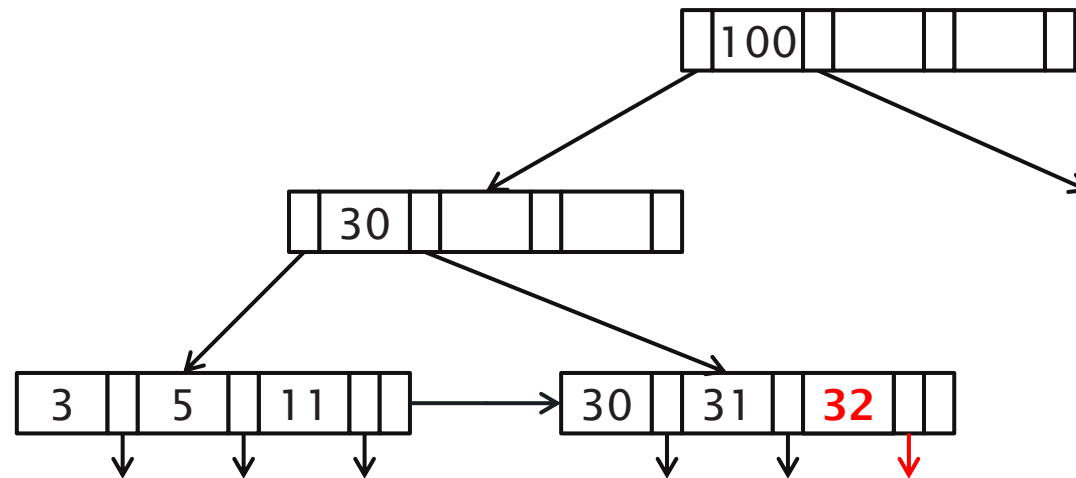
Four cases to consider:

1. Space available in leaf
2. Leaf overflow
3. Non-leaf overflow
4. New root

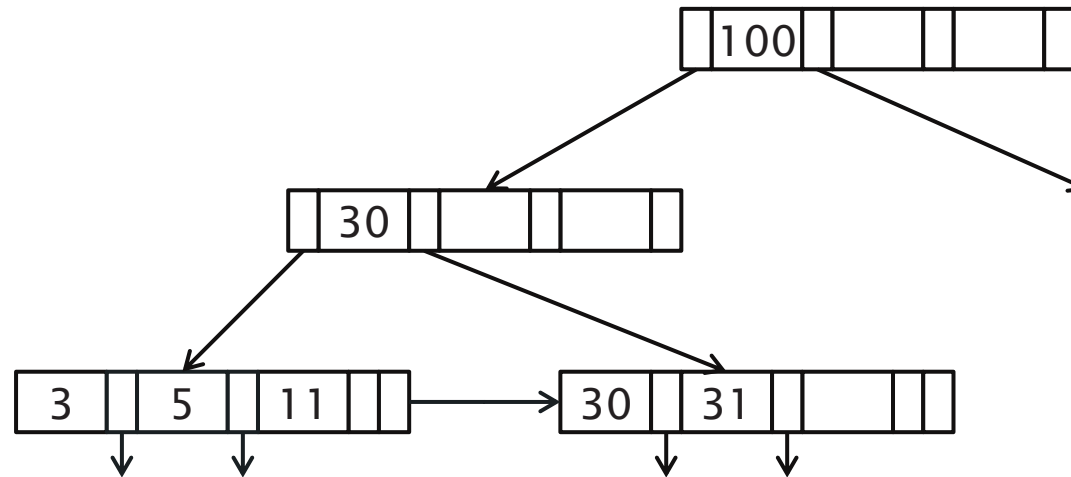
Case 1: insert key=32



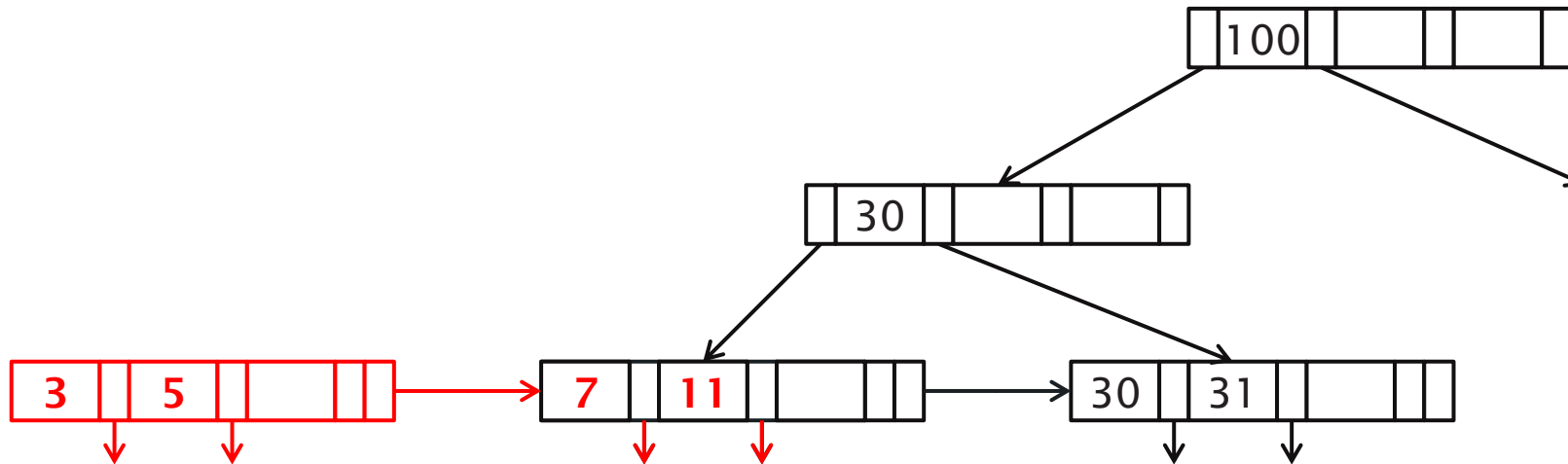
Case 1: insert key=32



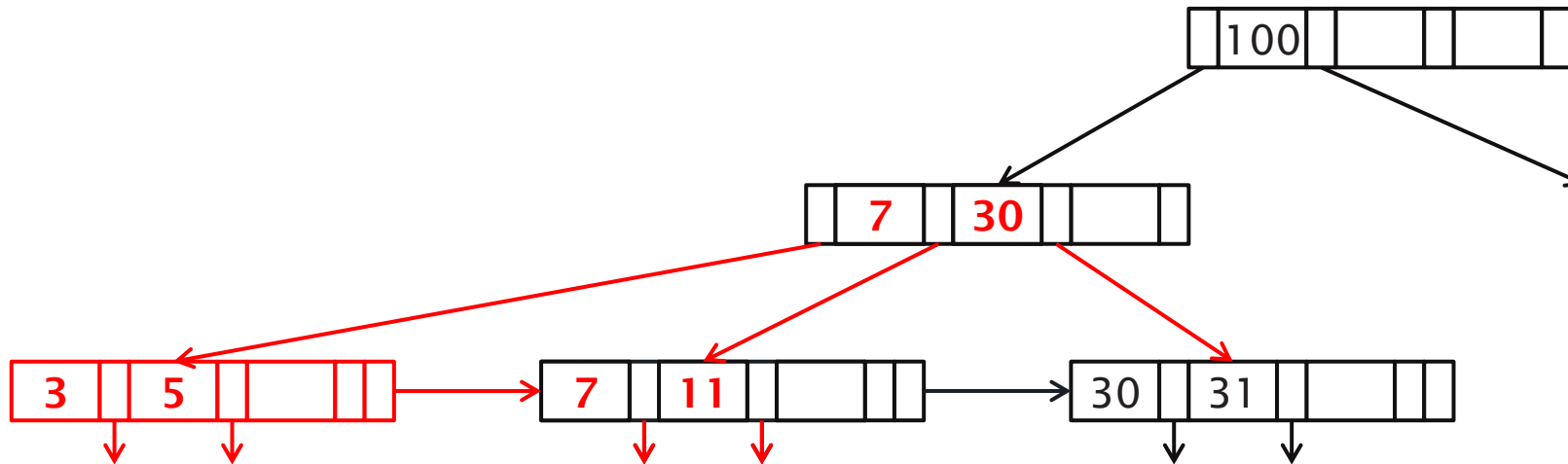
Case 2: insert key=7



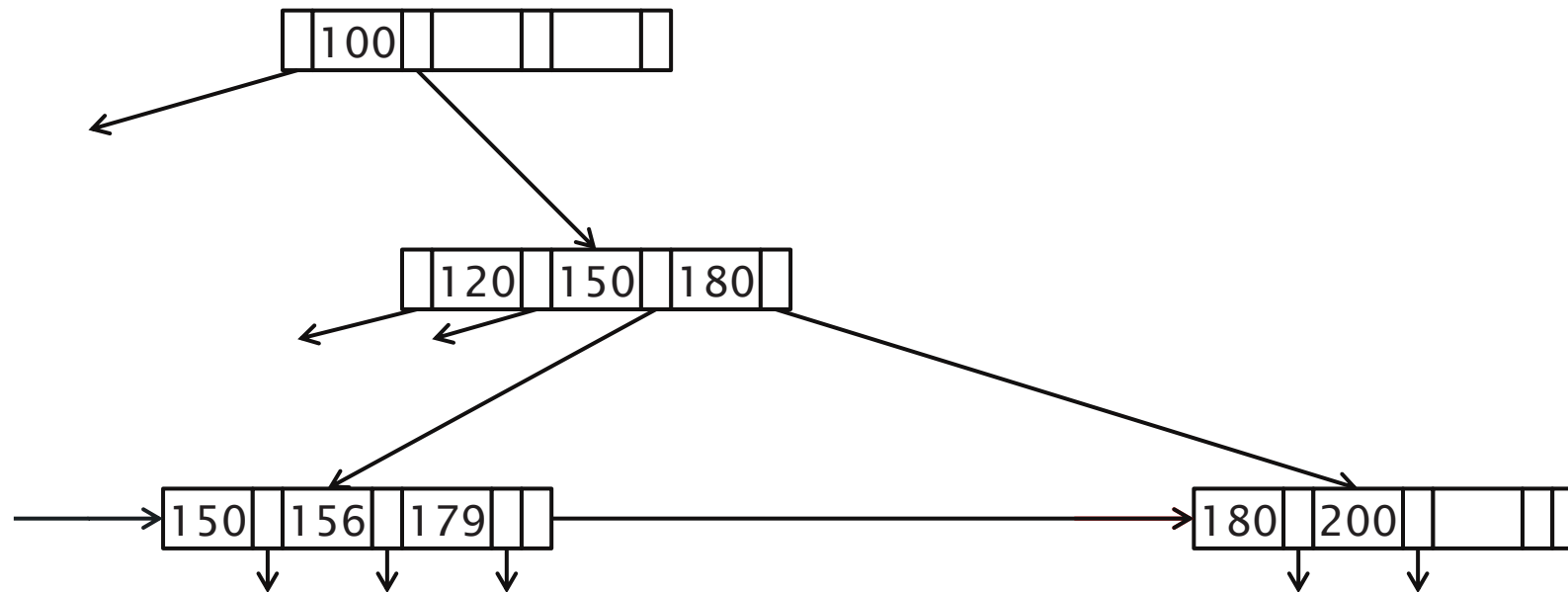
Case 2: insert key=7



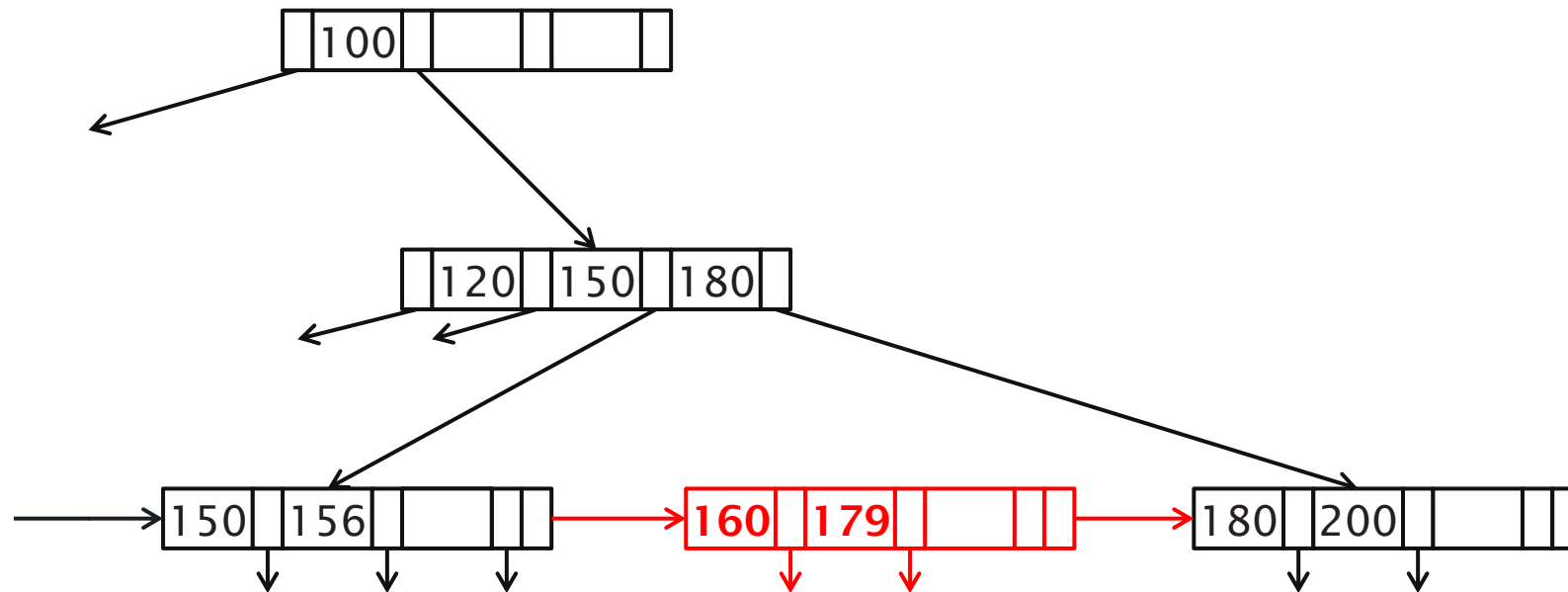
Case 2: insert key=7



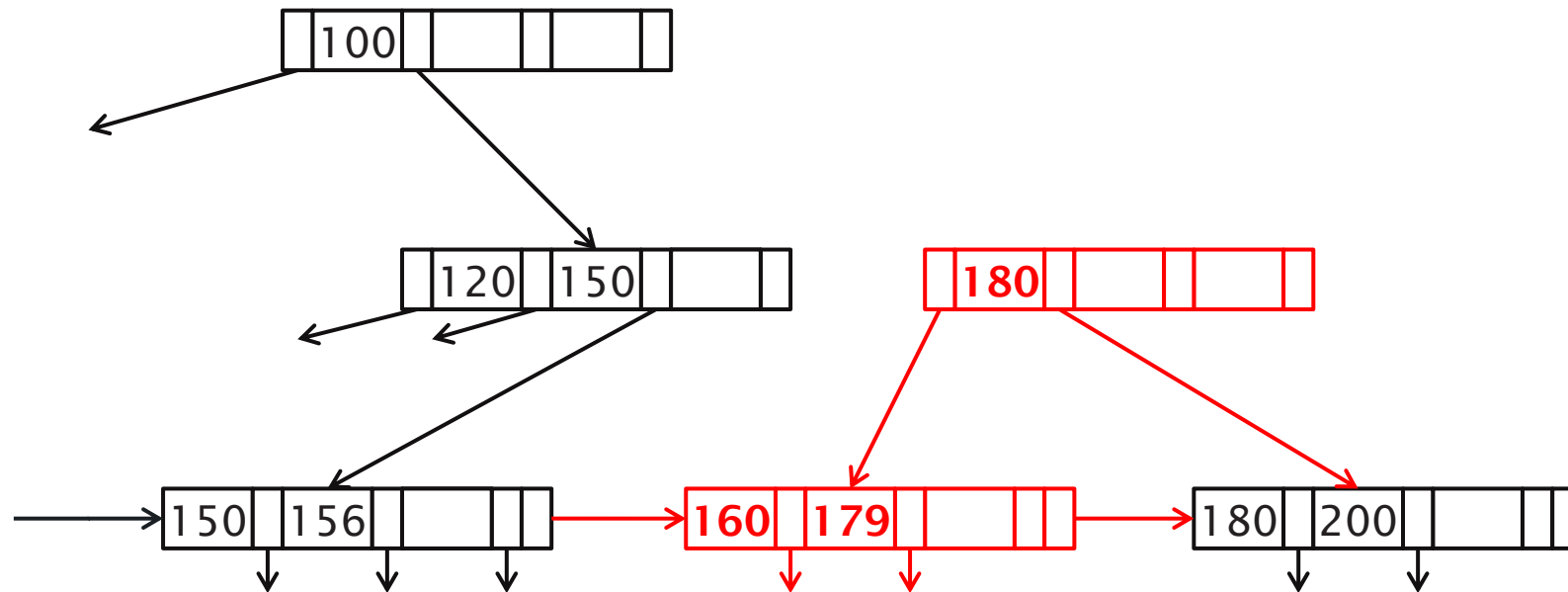
Case 3: insert key=160



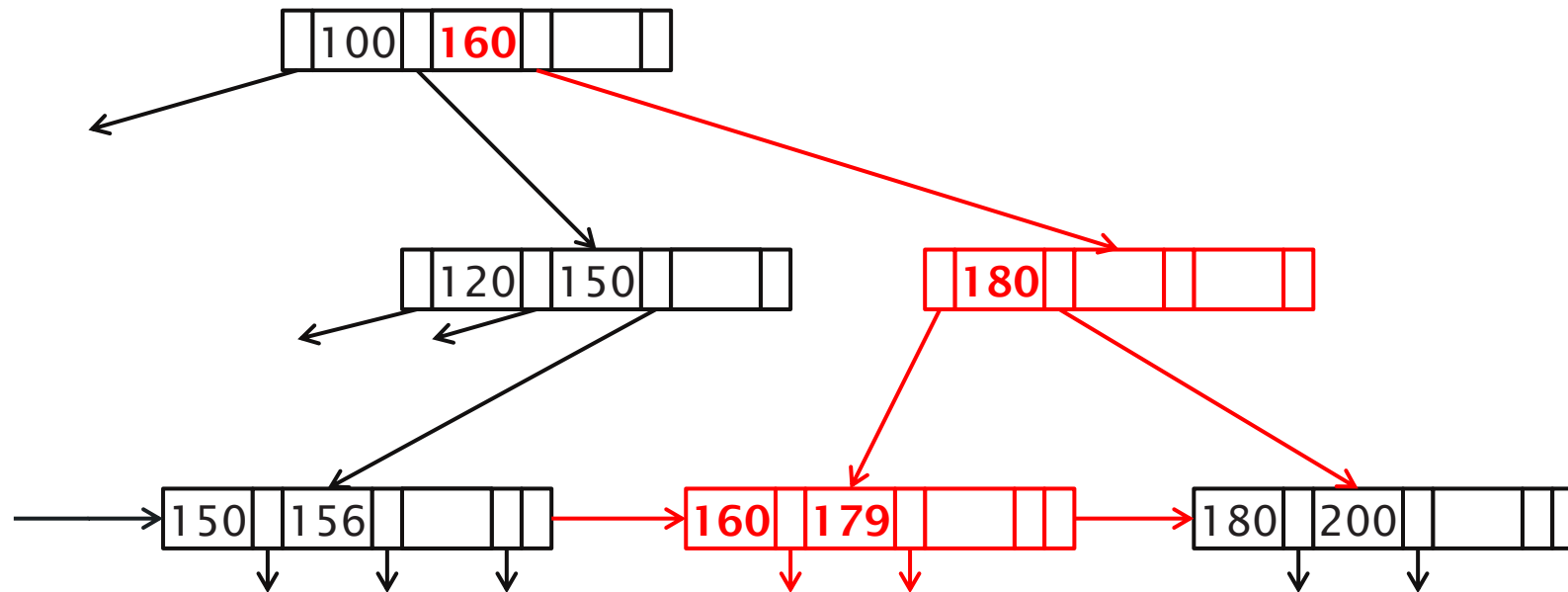
Case 3: insert key=160



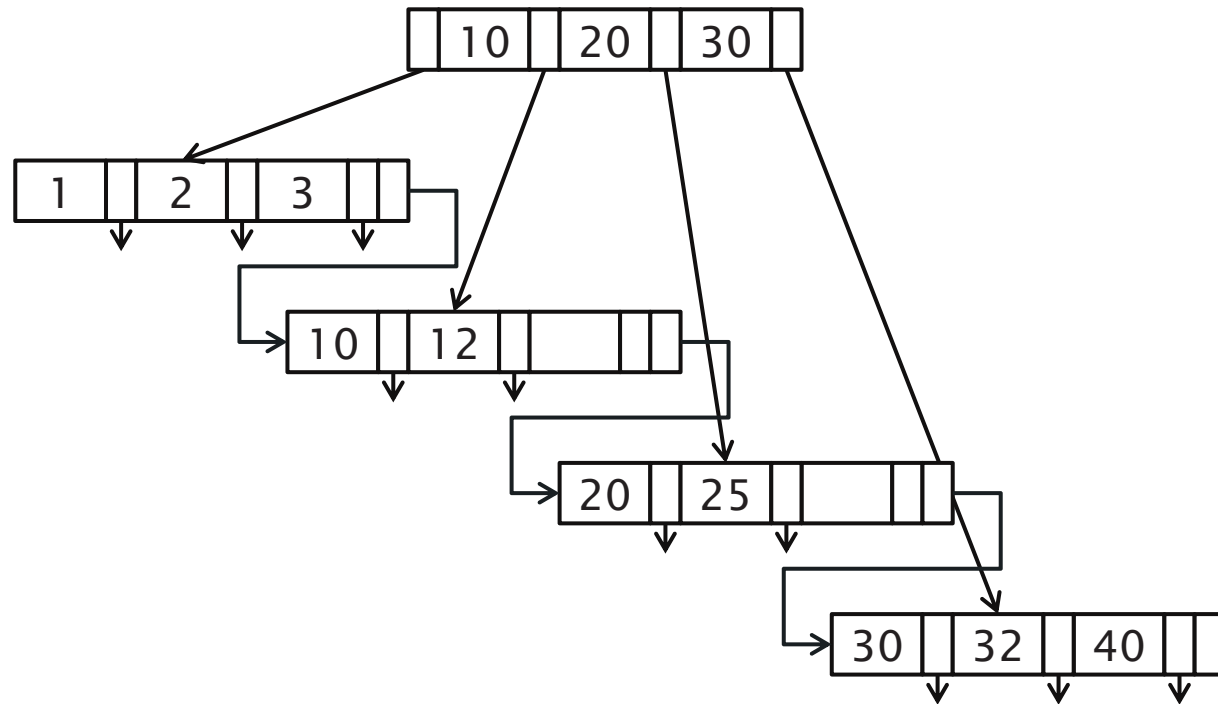
Case 3: insert key=160



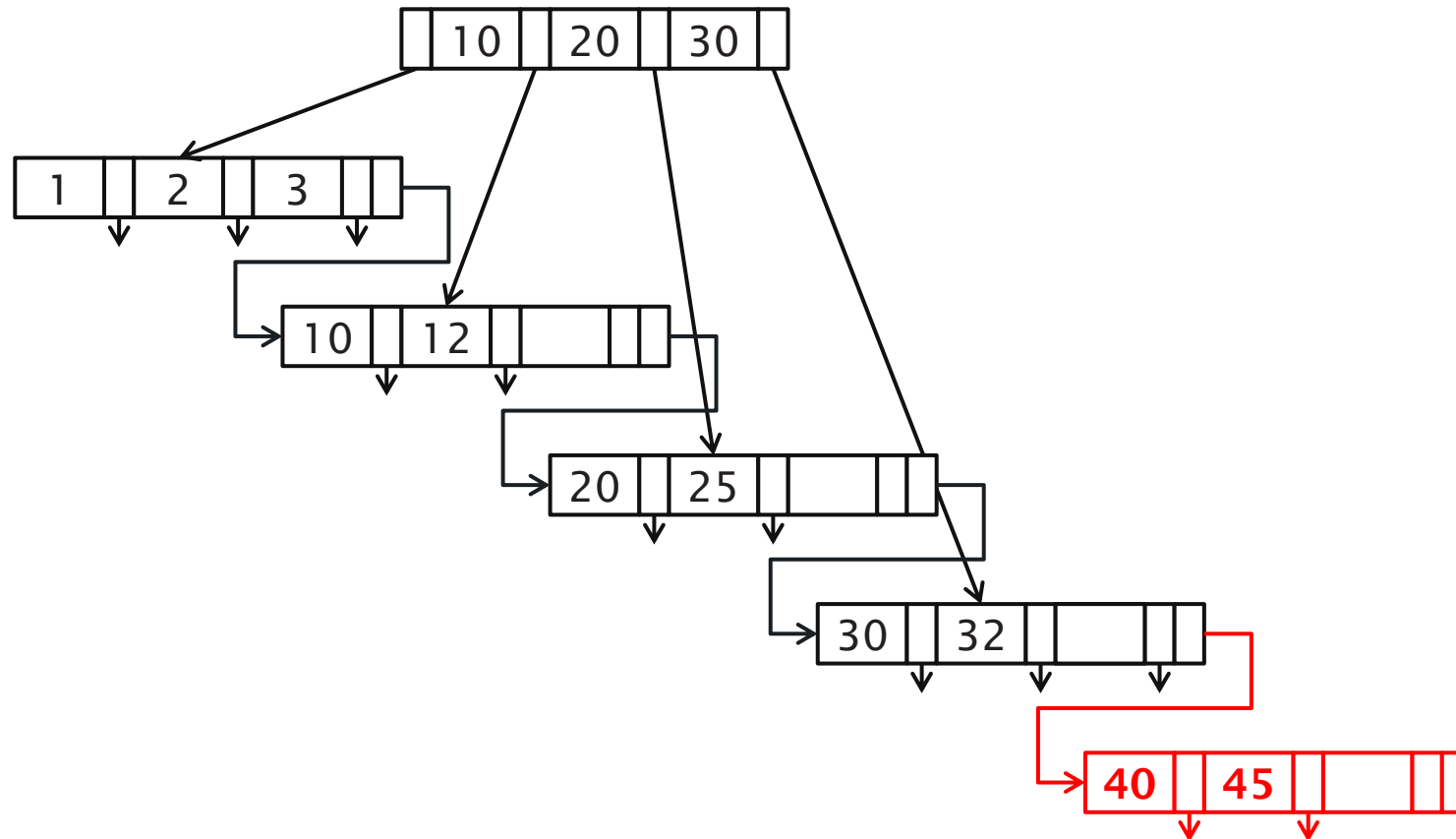
Case 3: insert key=160



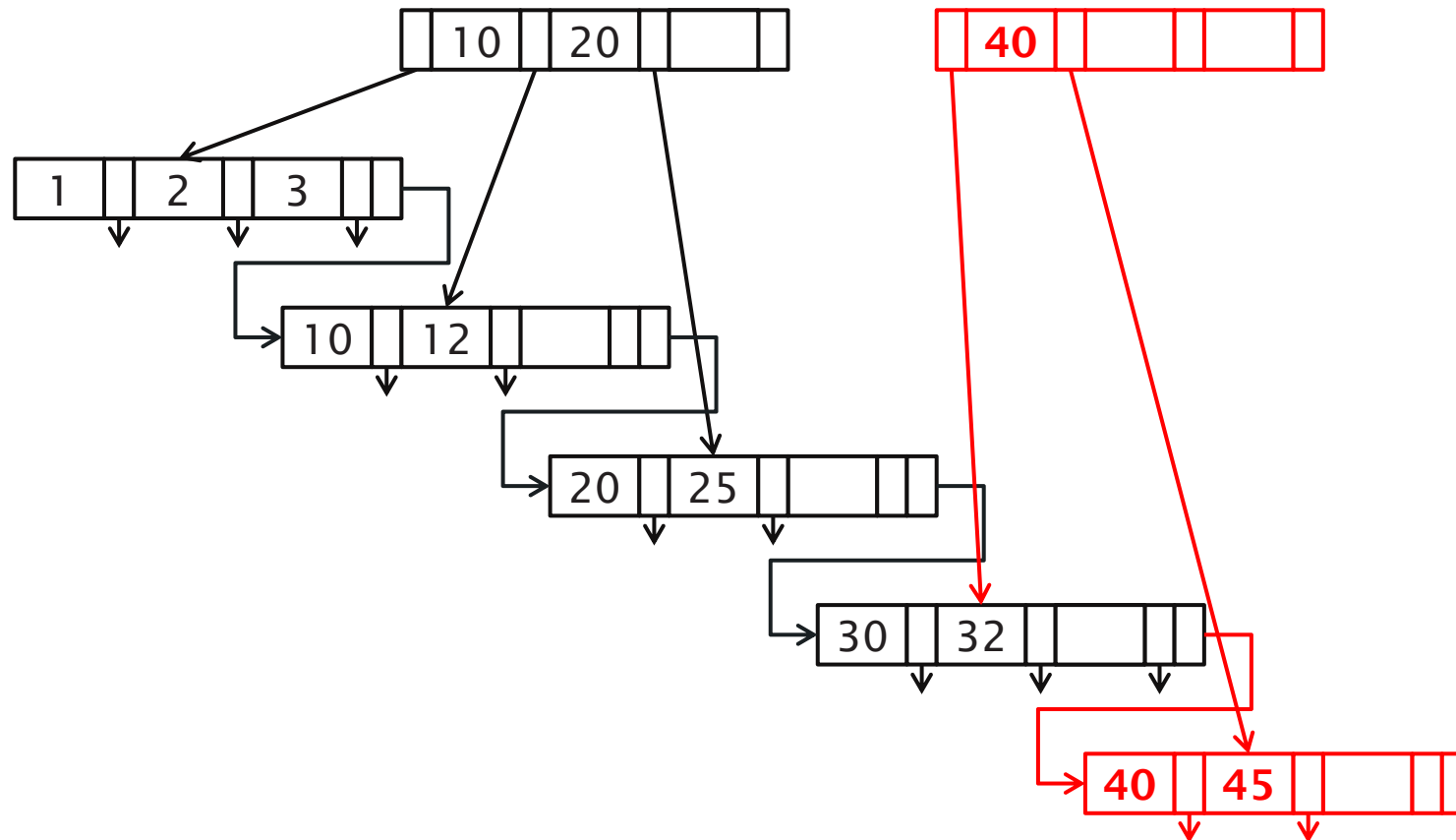
Case 4: insert 45



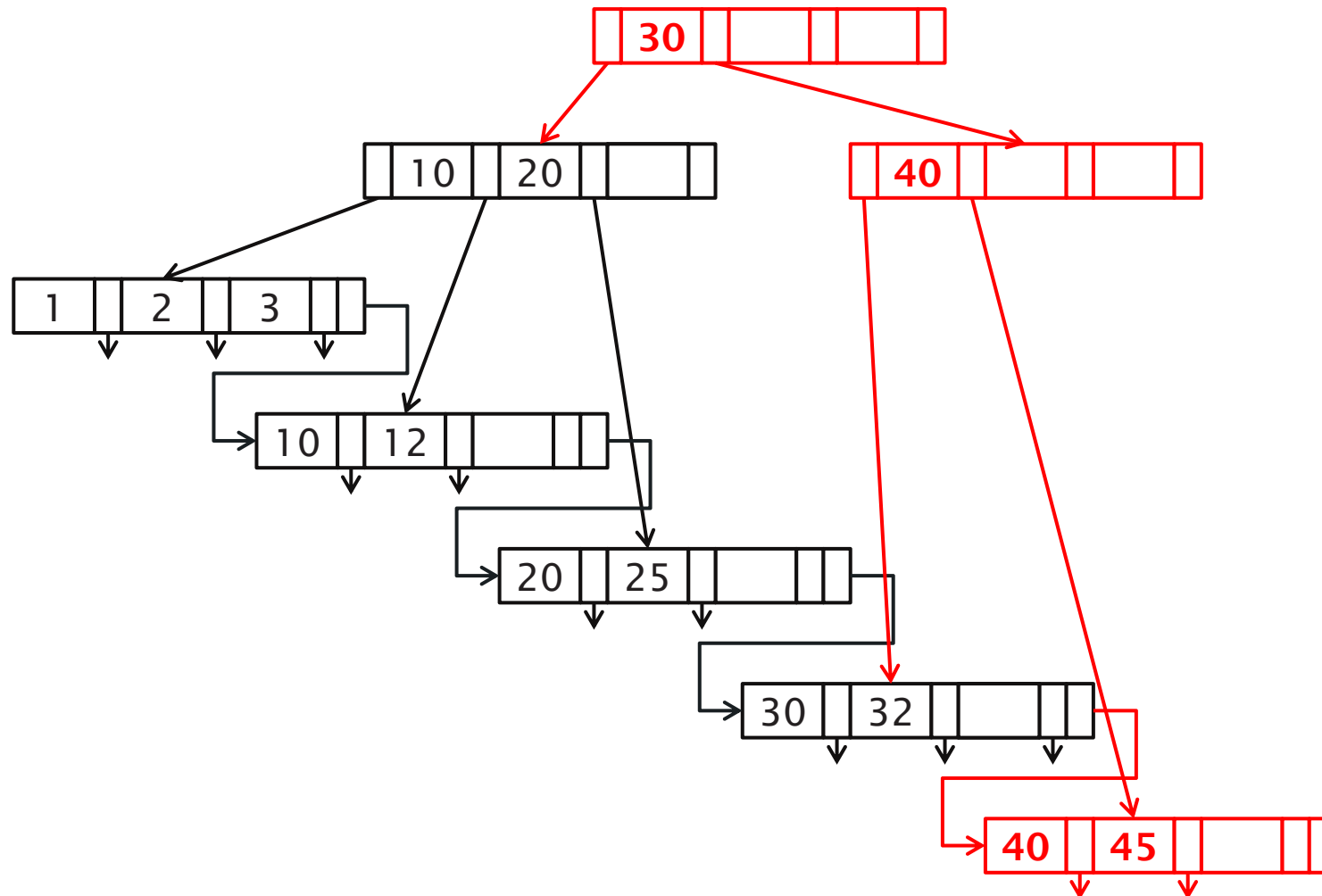
Case 4: insert 45



Case 4: insert 45



Case 4: insert 45

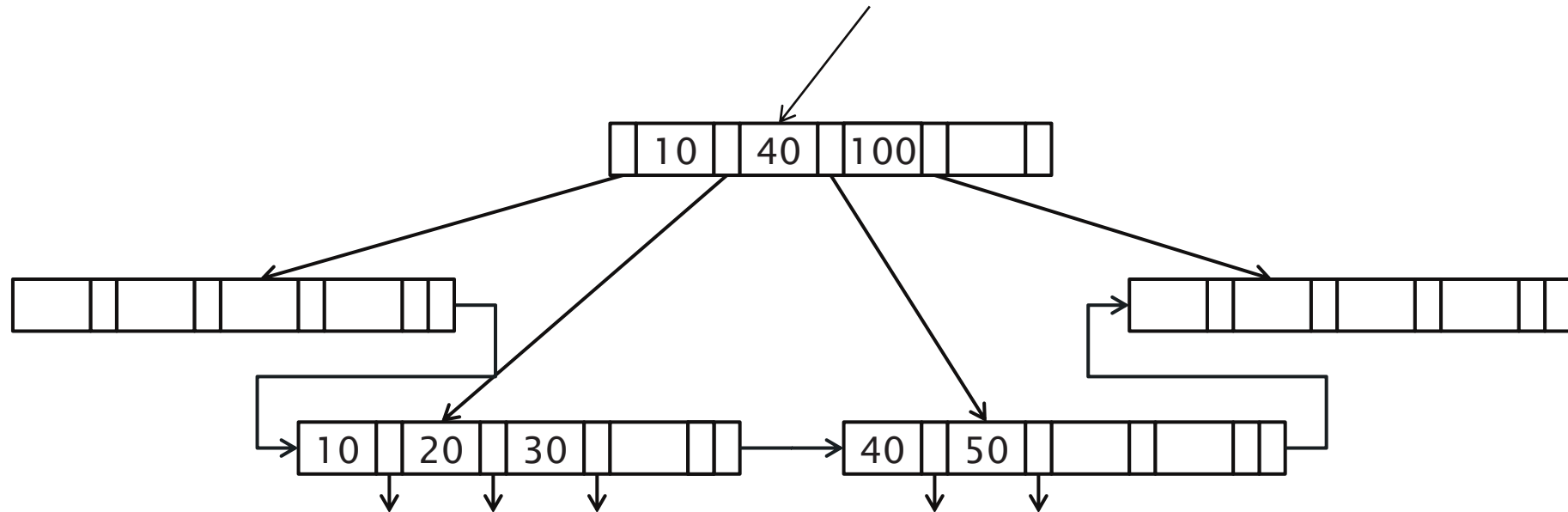


B+tree Deletion

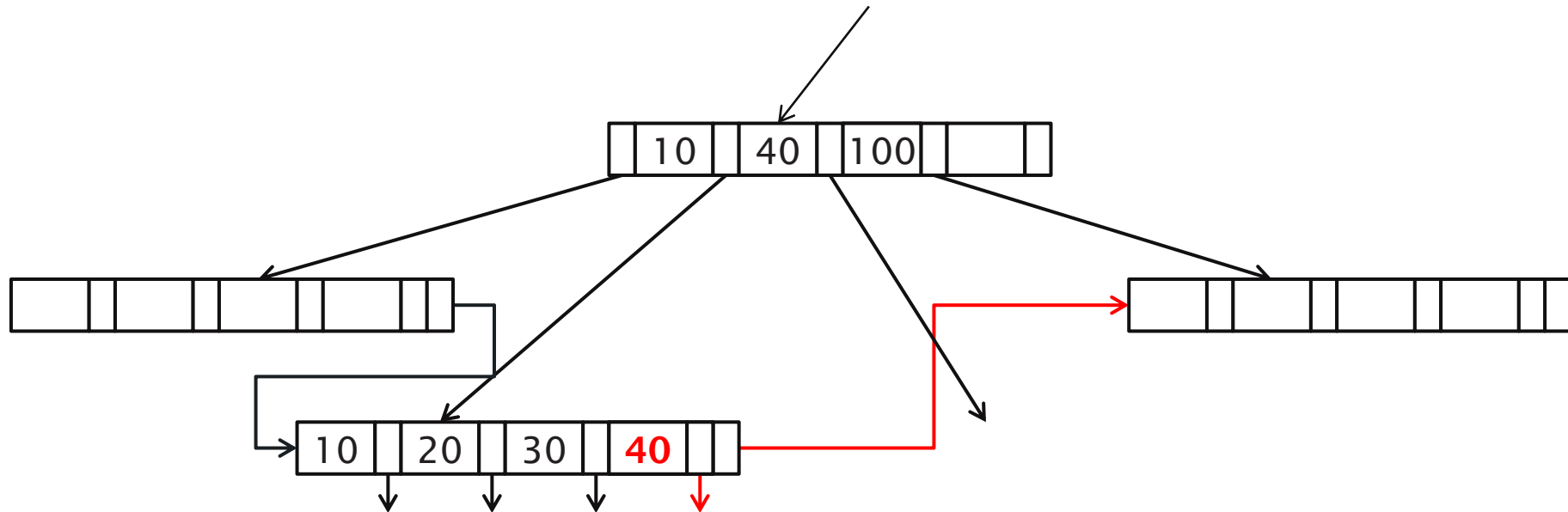
Four cases to consider:

1. Simple case
2. Coalesce with sibling
3. Re-distribute keys
4. Cases 2. or 3. at non-leaf

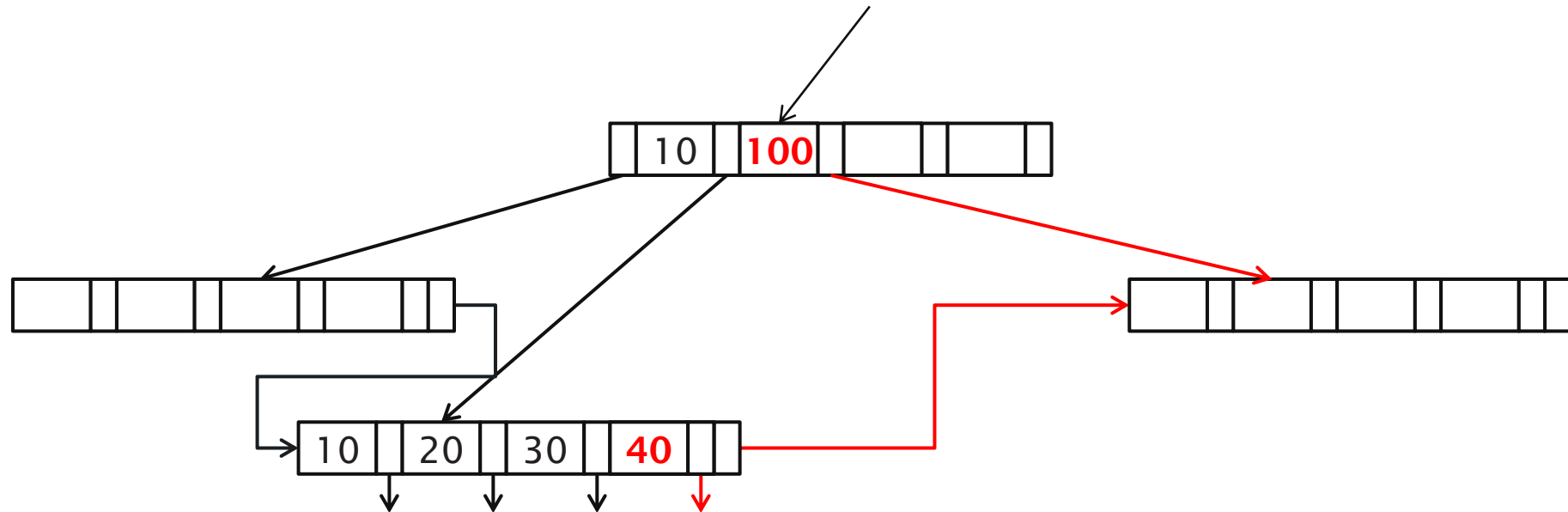
Case 2: delete key=50 (n=4)



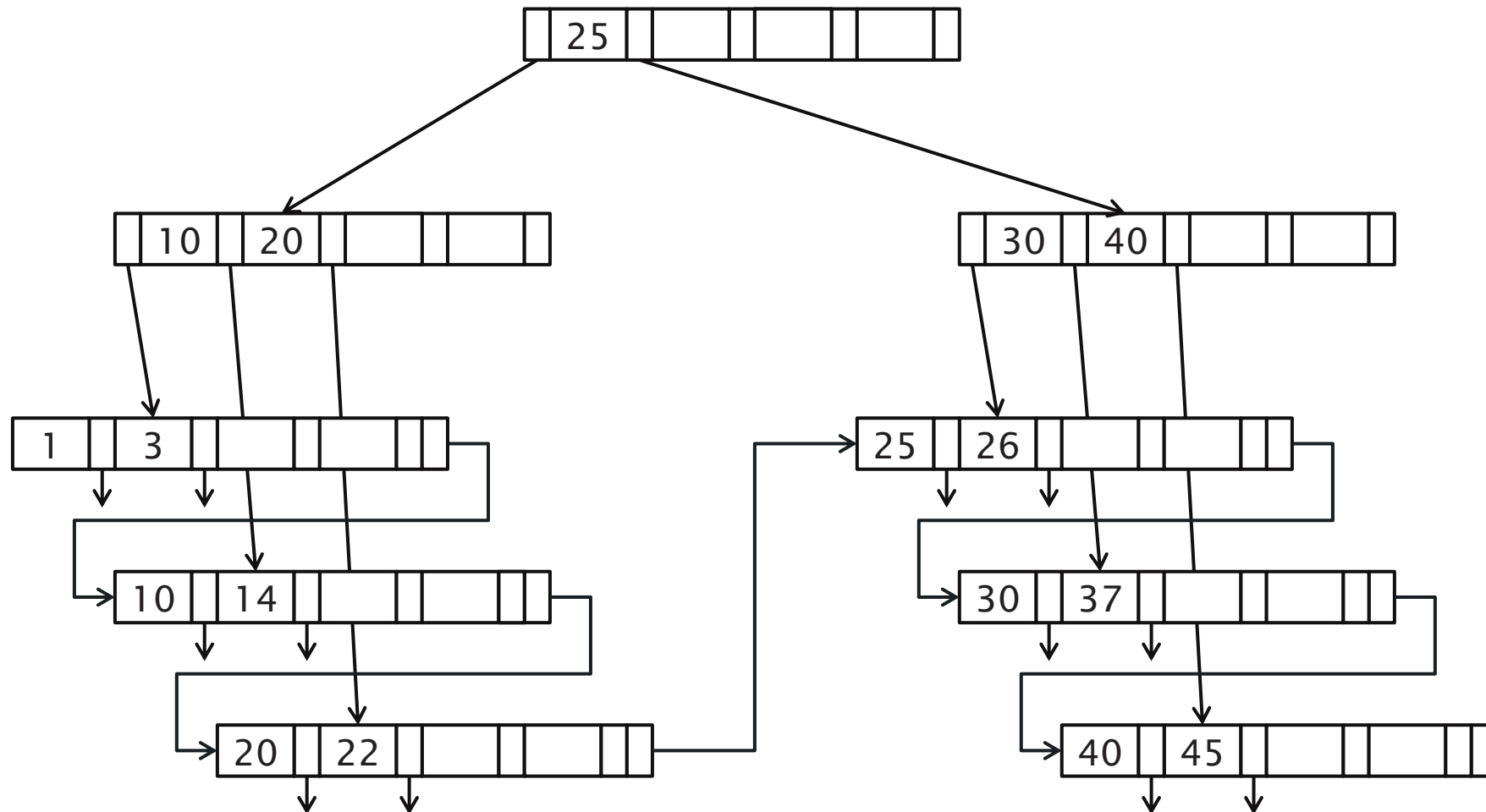
Case 2: delete key=50 (n=4)



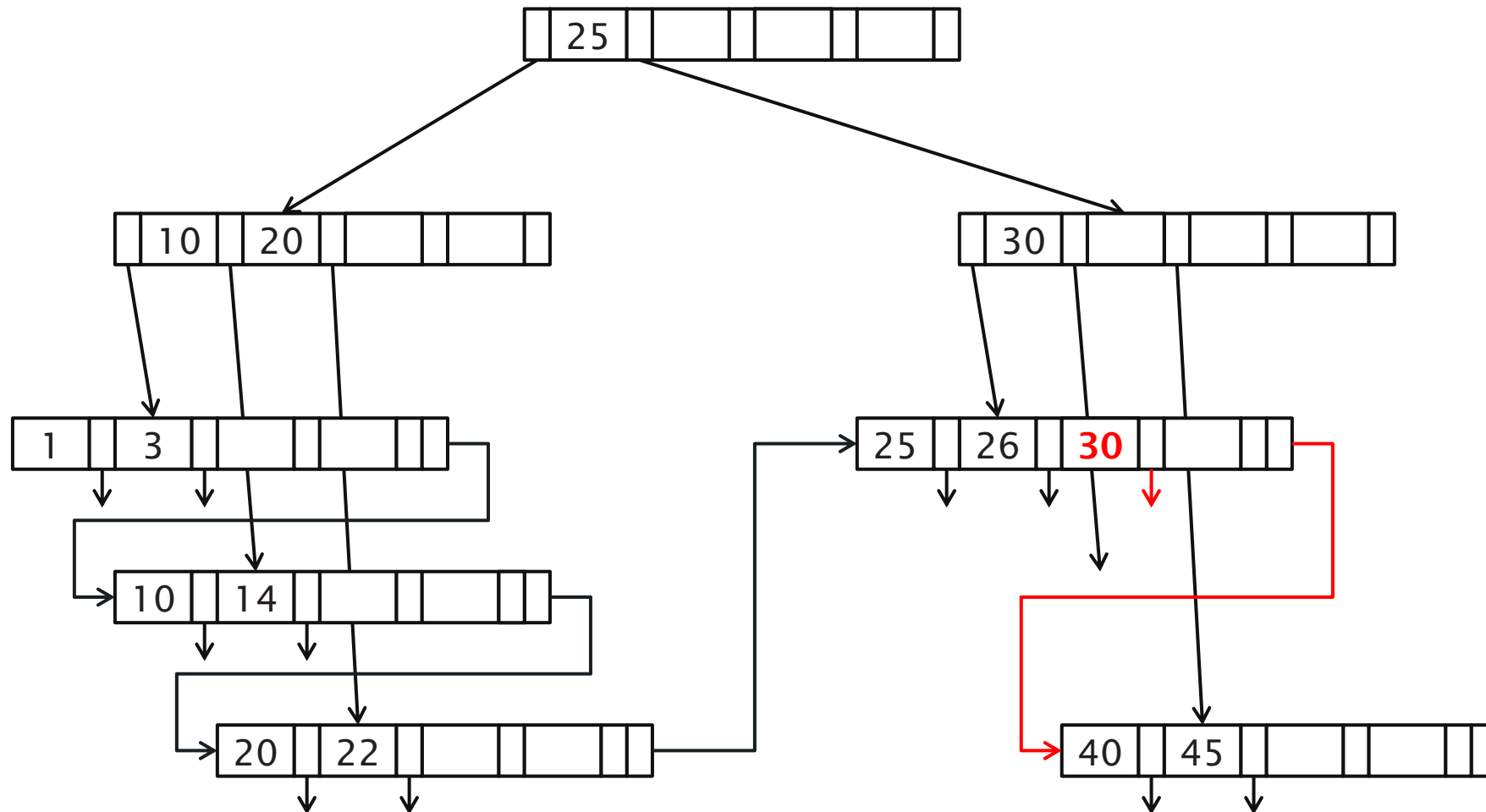
Case 2: delete key=50 (n=4)



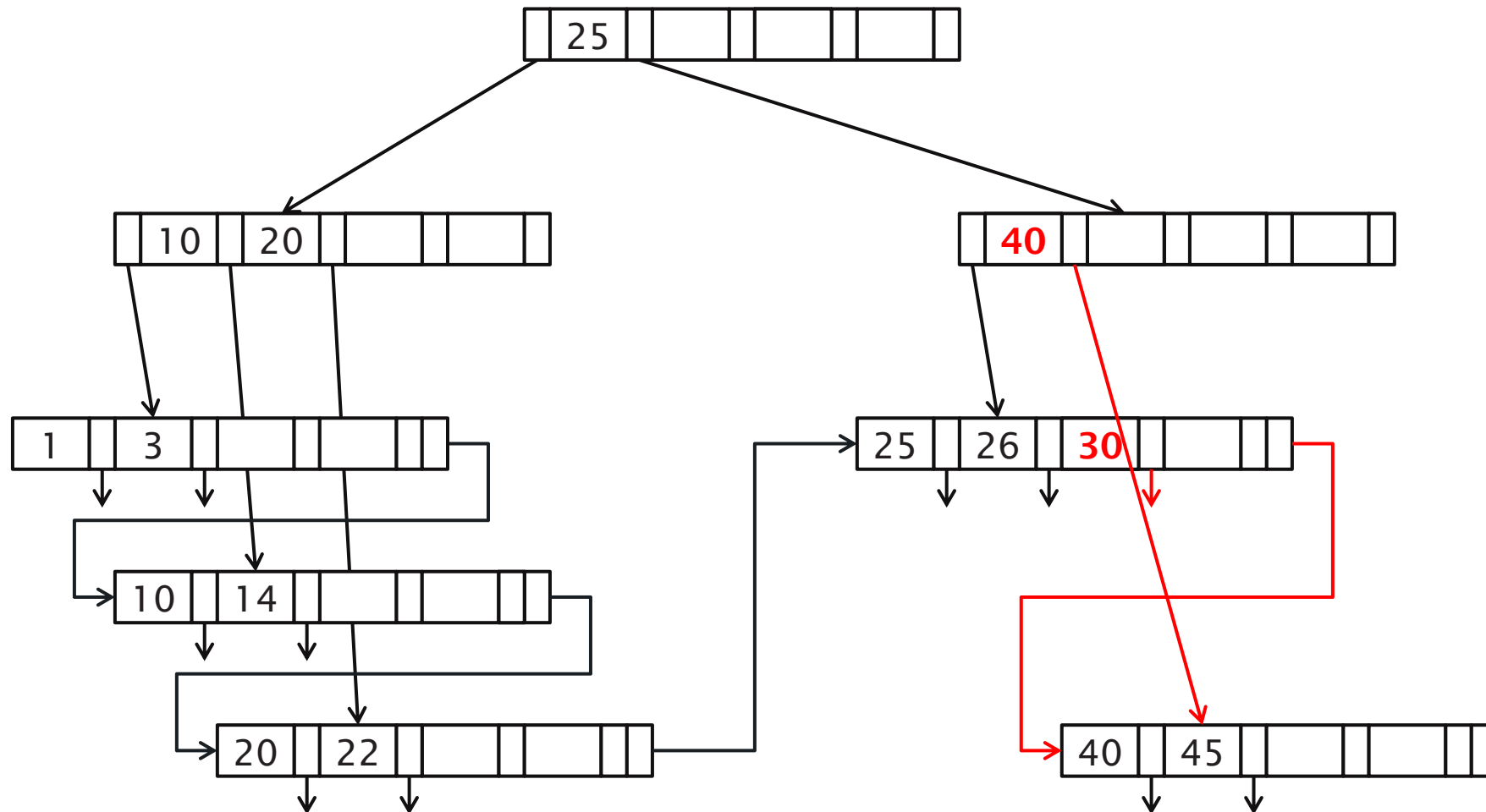
Case 4: delete key=37 (n=4)



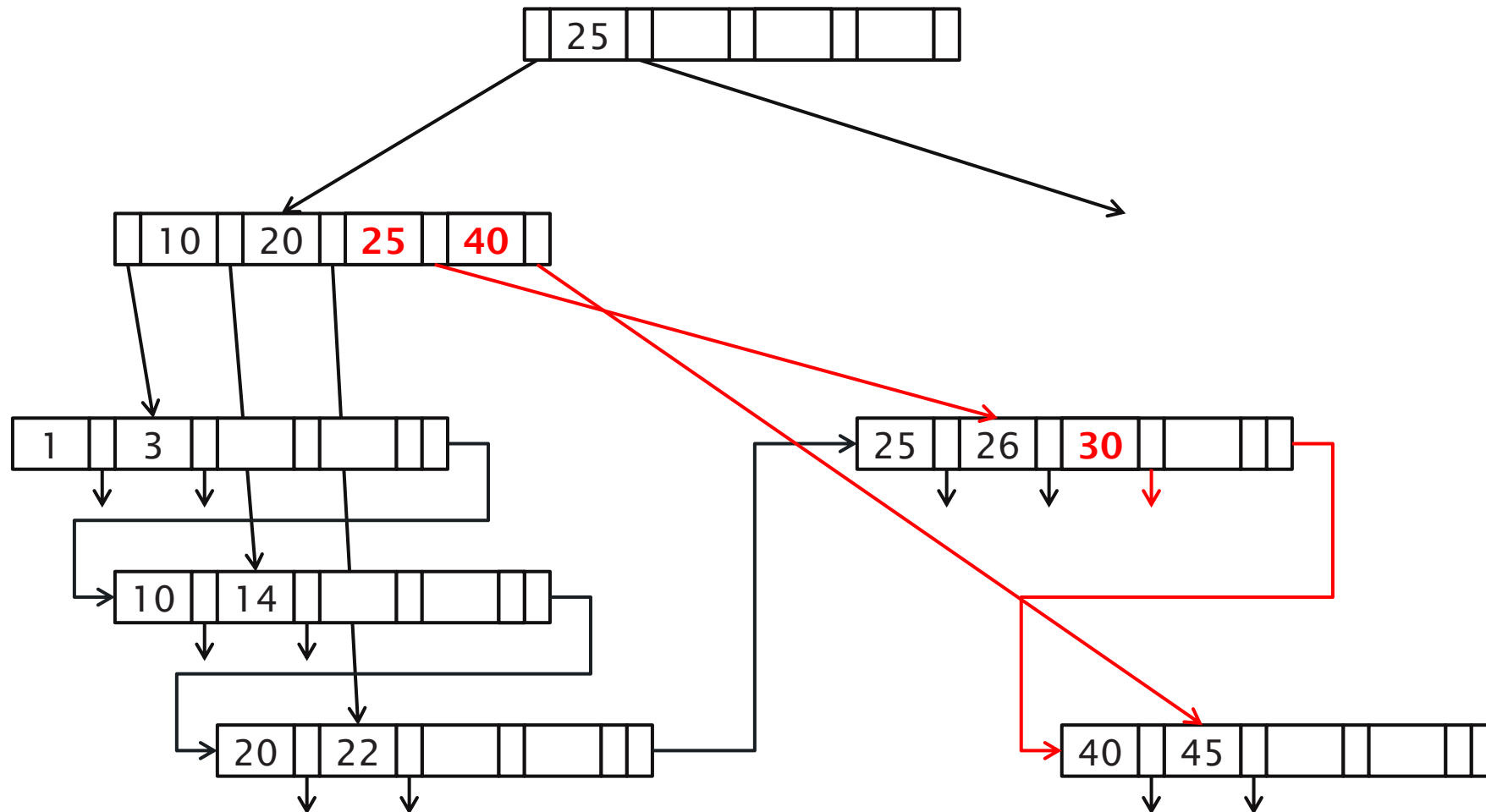
Case 4: delete key=37 (n=4)



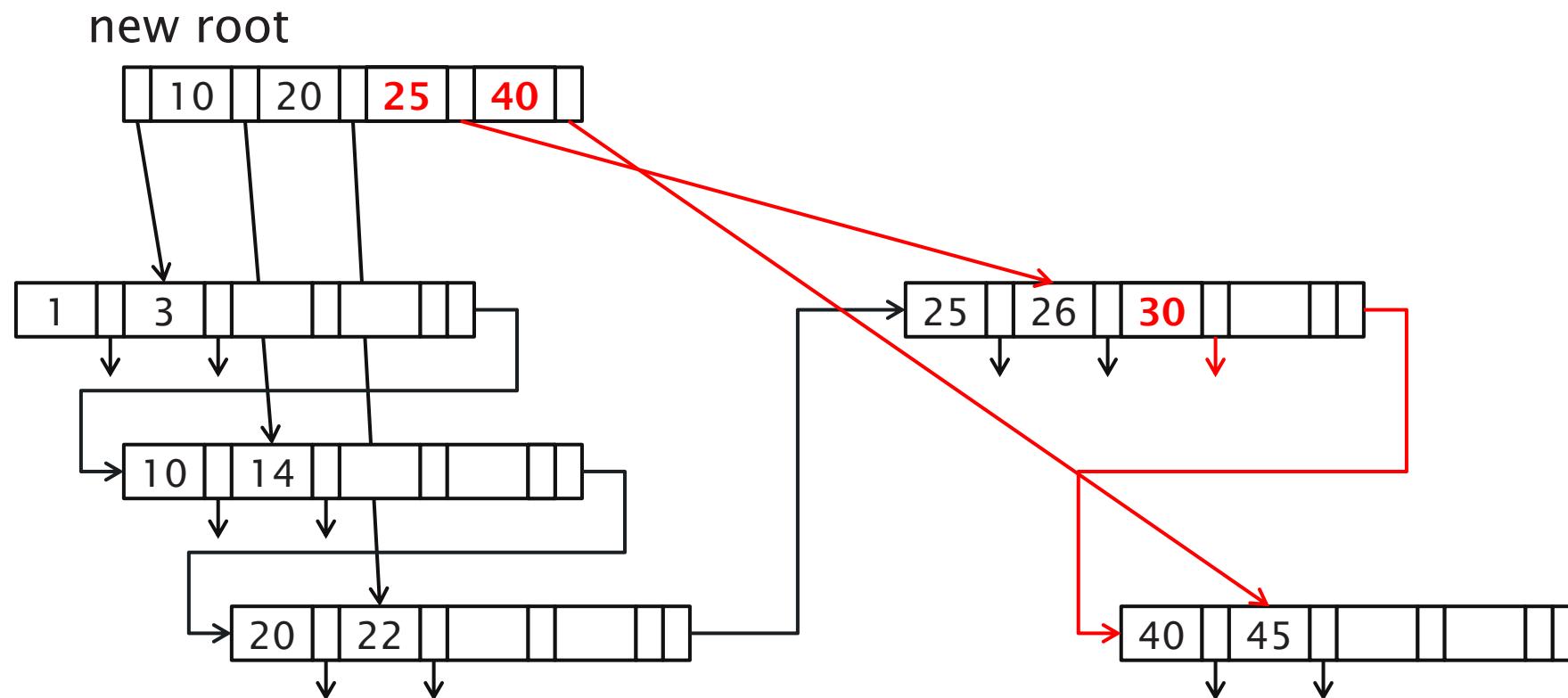
Case 4: delete key=37 (n=4)



Case 4: delete key=37 (n=4)



Case 4: delete key=37 (n=4)



B+tree deletions in practice

Often, coalescing is not implemented

- Too hard and not worth it!

B-trees versus static indexed sequential files

B-trees consume more space

- Blocks are not contiguous
- Fewer disk accesses for static indexes, even allowing for reorganisation

Concurrency control is harder in B-trees

but

DBA does not know:

- when to reorganise
- how full to load pages of new index

Hashing

Hashing

Main memory hash table

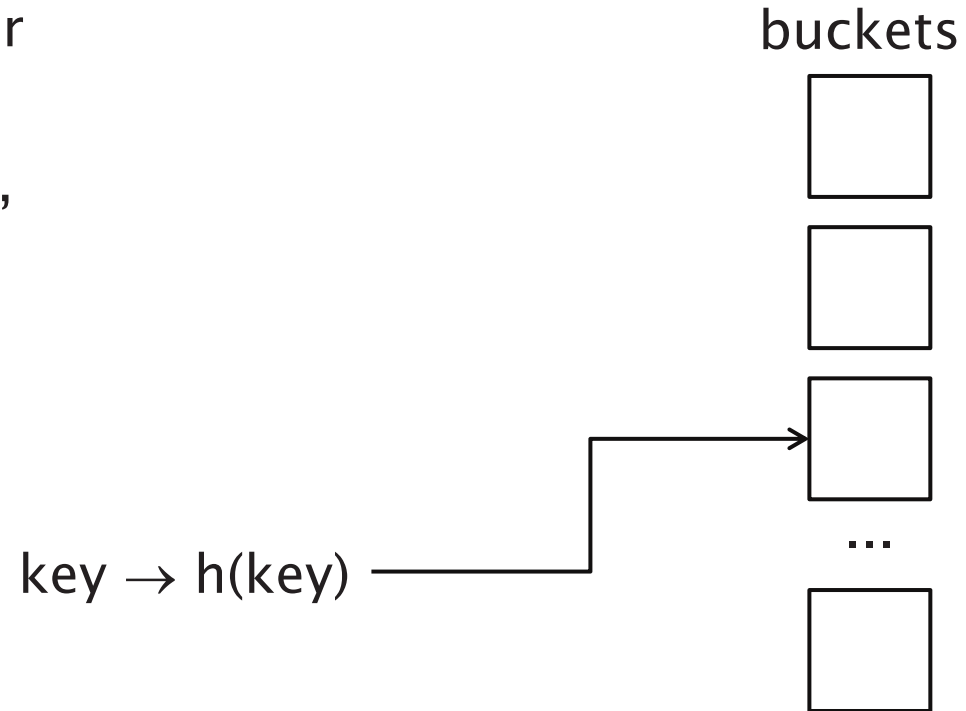
- *Hash function* $h()$ takes a key and computes an integer value
- Value is used to select a bucket from a *bucket array*
- Bucket array contains linked lists of records

Secondary storage hash table

- Stores many more records than a main memory hash table
- Bucket array consists of disk blocks

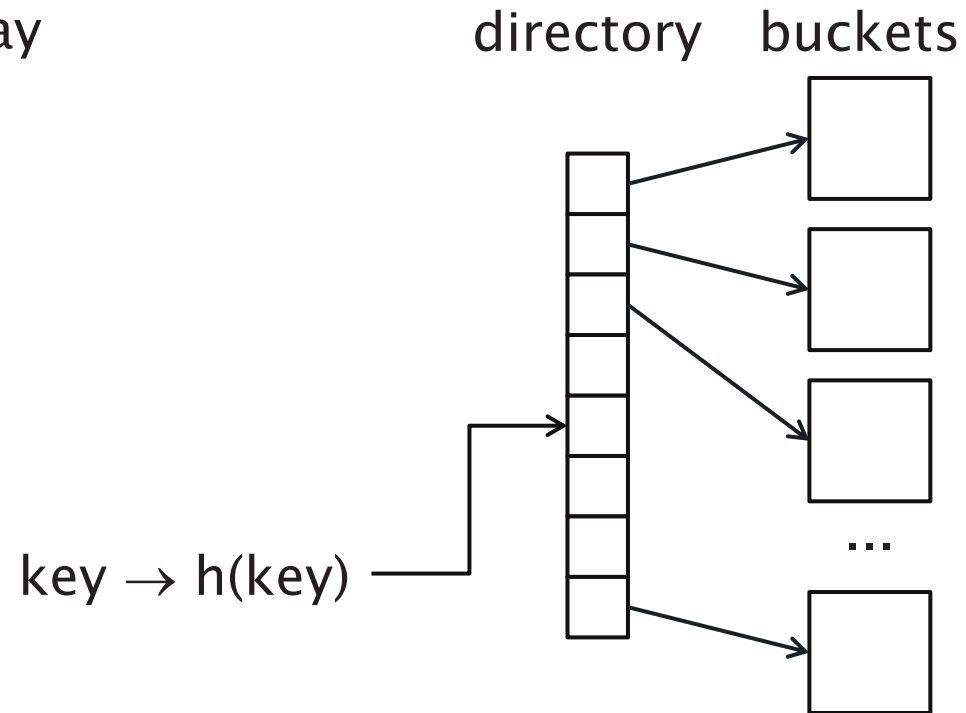
Hashing approach #1

- Hash function calculates block pointer directly, or as offset from first block
- Requires bucket blocks to be in fixed, consecutive locations



Hashing approach #2

- Hash function calculates offset in array of block pointers (directory)
- Used for “secondary” search keys



Example hash function

Key = 'x1 x2 ... xn' (n byte character string), b buckets

h: add $x_1 + x_2 + \dots + x_n$, compute sum modulo b

Not a particularly good function

Good hash function has the same expected number of keys per bucket for each bucket

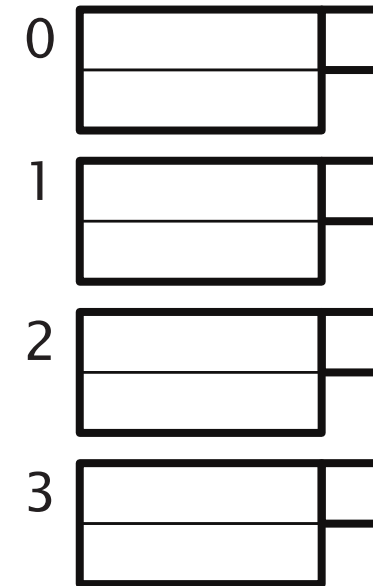
Buckets

Do we keep keys sorted?

Yes, if CPU time is critical and inserts/deletes are relatively infrequent

Hashing example

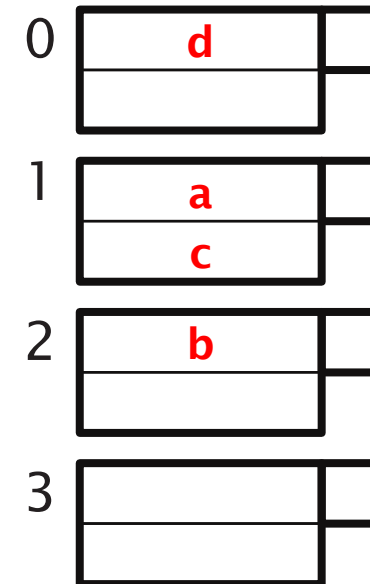
Two records per bucket



Hashing example

Insert a, b, c, d

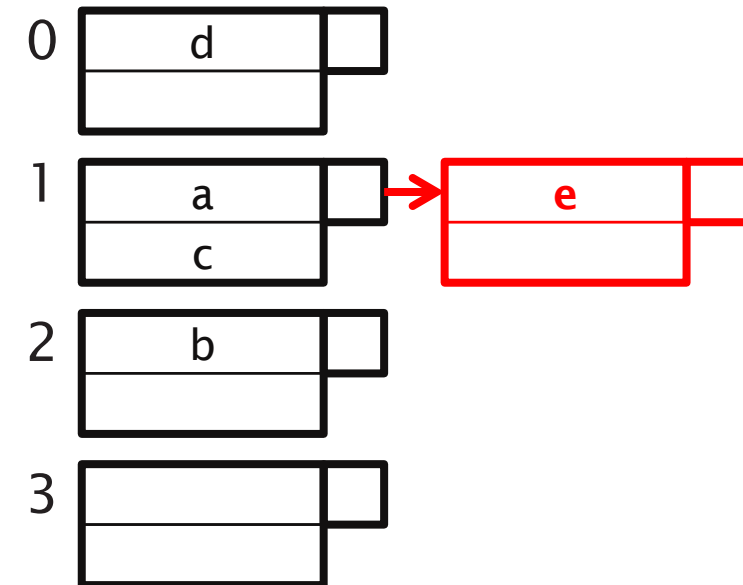
- $h(a) = 1$
- $h(b) = 2$
- $h(c) = 1$
- $h(d) = 0$



Hashing example: Overflow

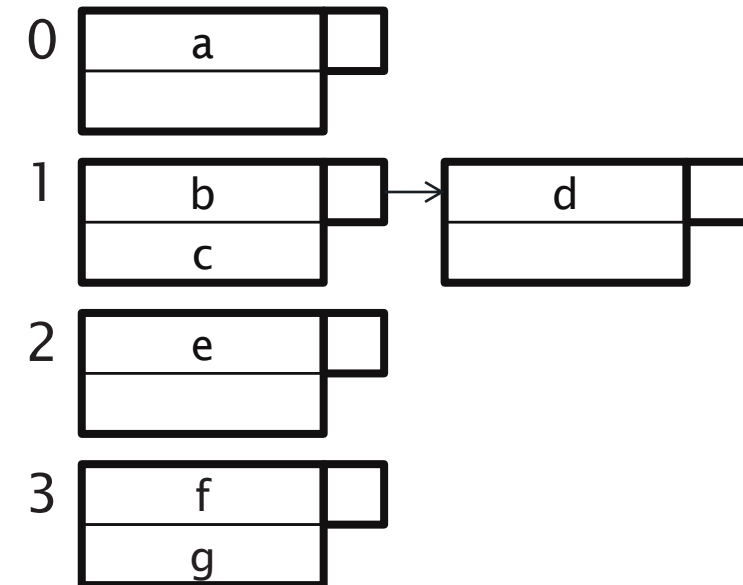
Insert e

- $h(e) = 1$



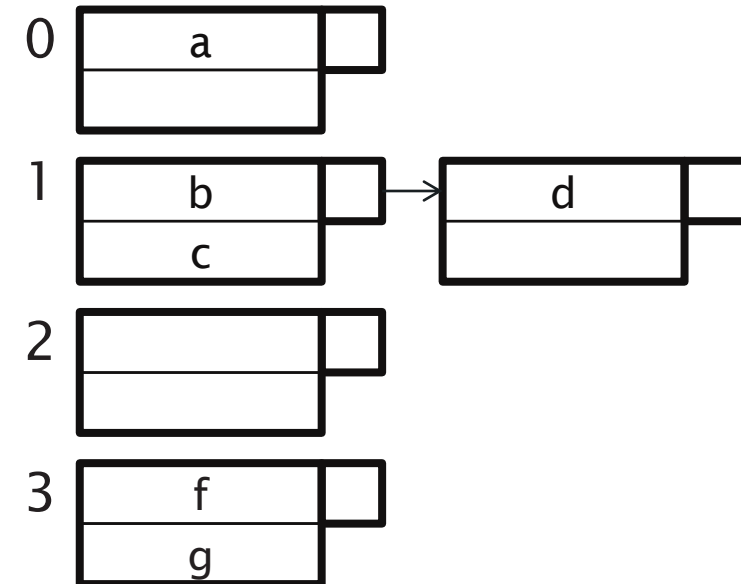
Hashing example: Deletion

Delete e



Hashing example: Deletion

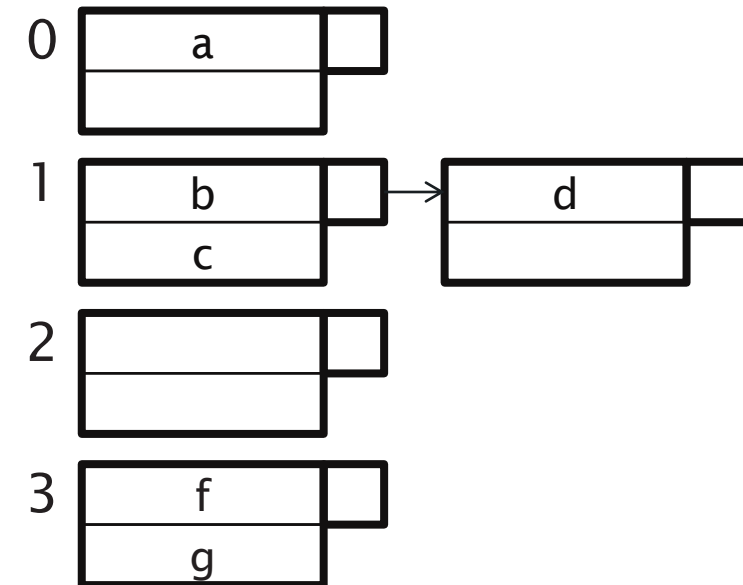
Delete e



Hashing example: Deletion

Delete f

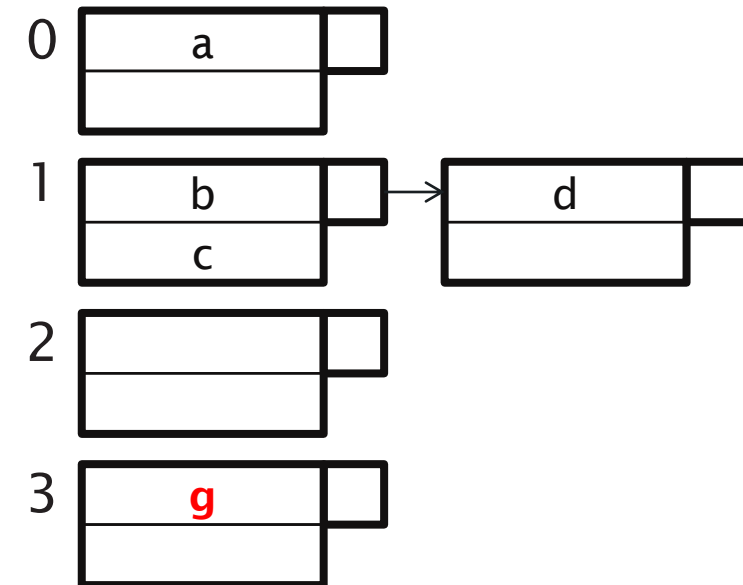
(move g up)



Hashing example: Deletion

Delete f

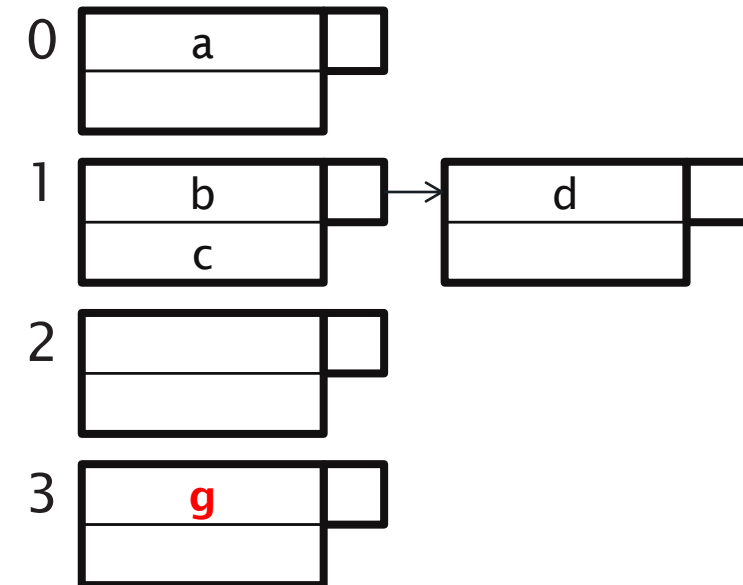
(move g up)



Hashing example: Deletion

Delete f

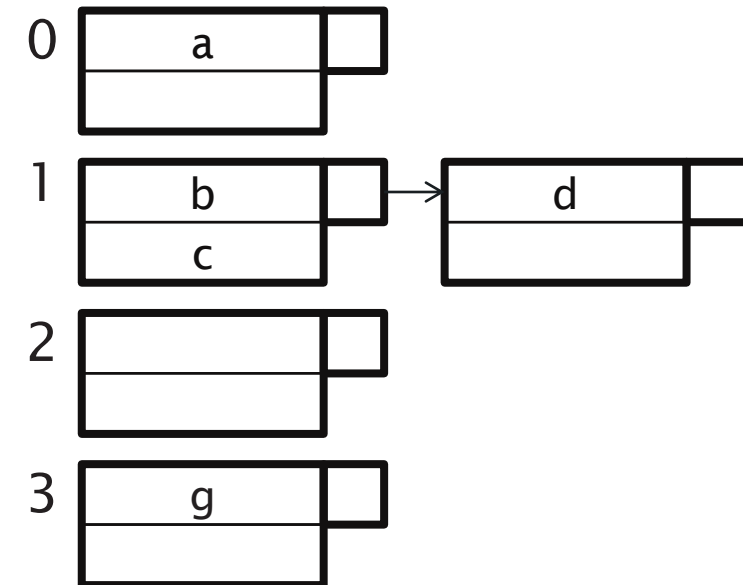
(move g up)



Hashing example: Deletion

Delete c

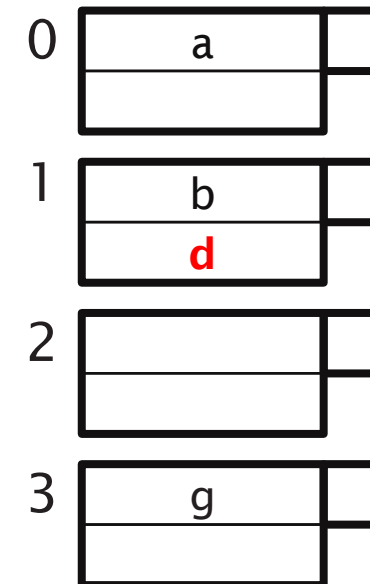
(move d from overflow block)



Hashing example: Deletion

Delete c

(move d from overflow block)



Rule of thumb:

Space utilisation should be between 50% and 80%

Utilisation = $\#keys \text{ used} / \text{total } \#keys \text{ that fit}$

If $< 50\%$, wasting space

If $> 80\%$, overflows significant

Depends on how good hash function is and on $\#keys/bucket$

How do we cope with growth?

Overflows and reorganizations

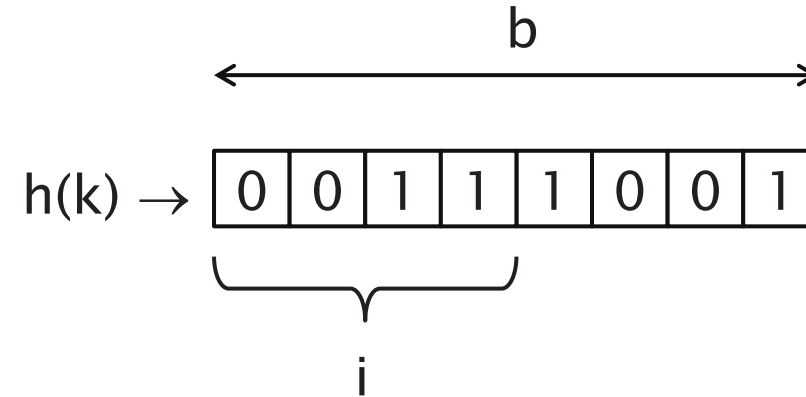
Dynamic hashing

- Extensible
- Linear

Extensible hashing

Combines two ideas:

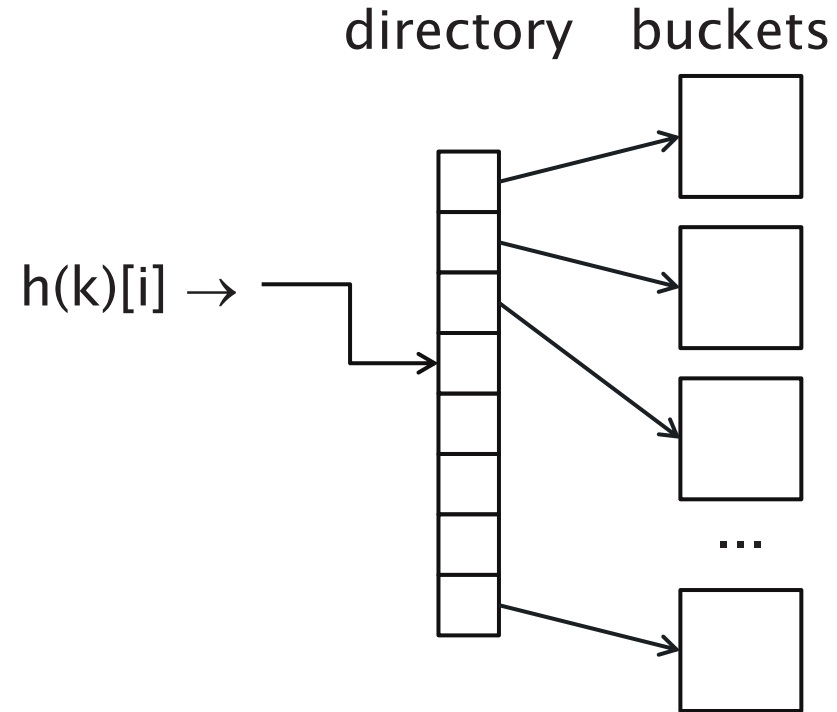
1. Use i of b bits output by hash function, where i grows over time



Extensible hashing

Combines two ideas:

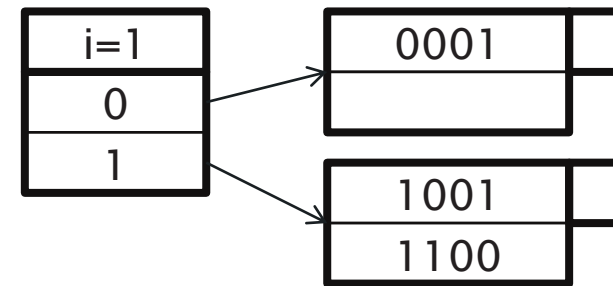
1. Use i of b bits output by hash function, where i grows over time
2. Use a directory



Example

$h(k)$ gives 4 bits

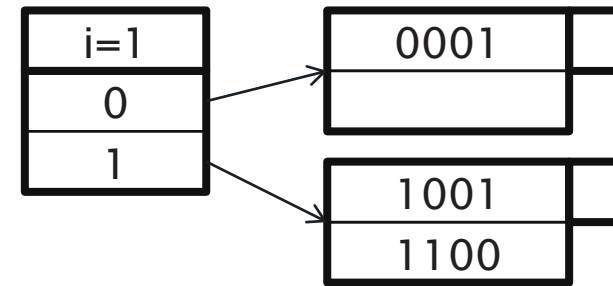
2 keys/bucket



Example

Insert 1010

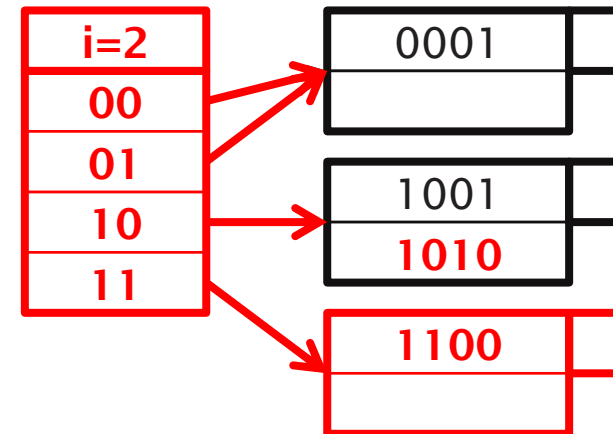
- Bucket overflow



Example

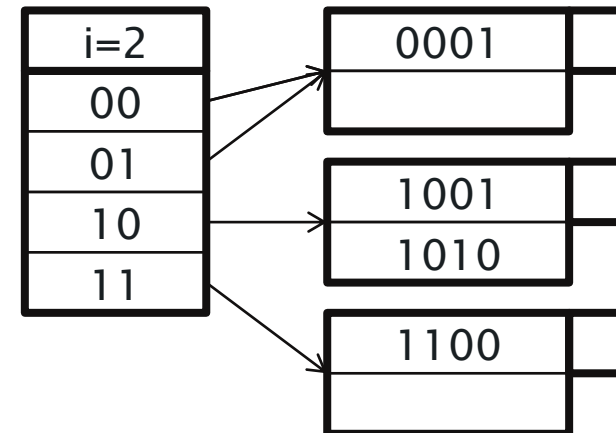
Insert 1010

- Bucket overflow
- Extend (double) directory
- Split bucket



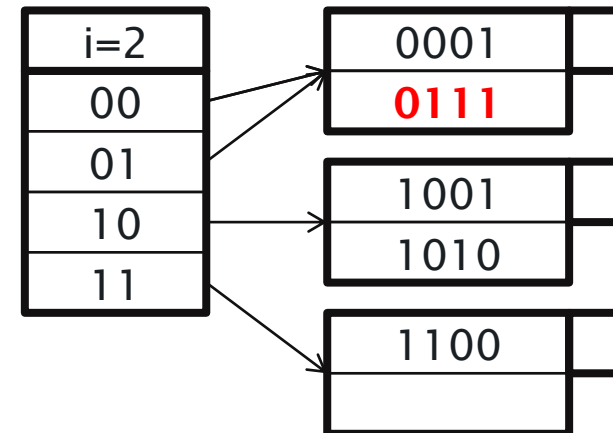
Example

Insert 0111



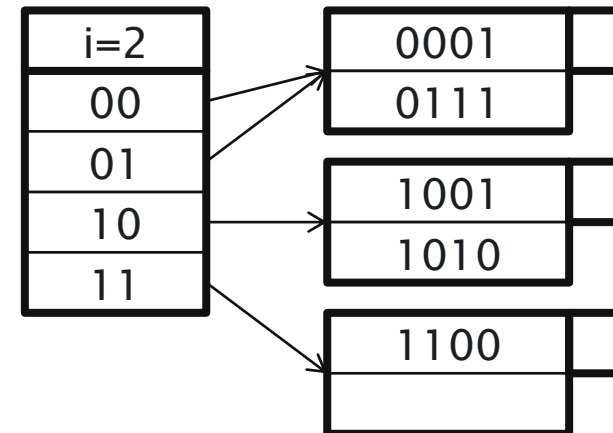
Example

Insert 0111



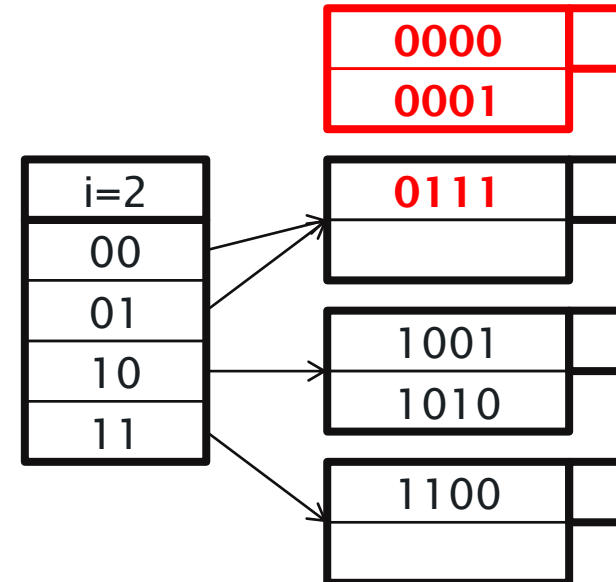
Example

Insert 0000



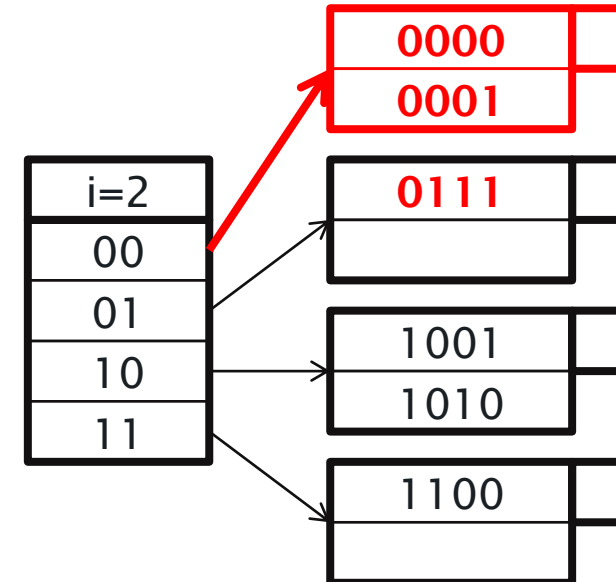
Example

Insert 0000



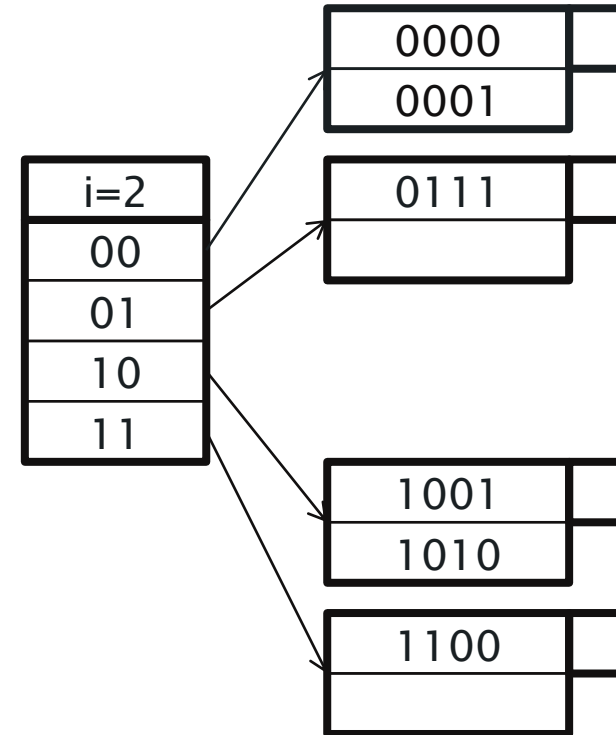
Example

Insert 0000



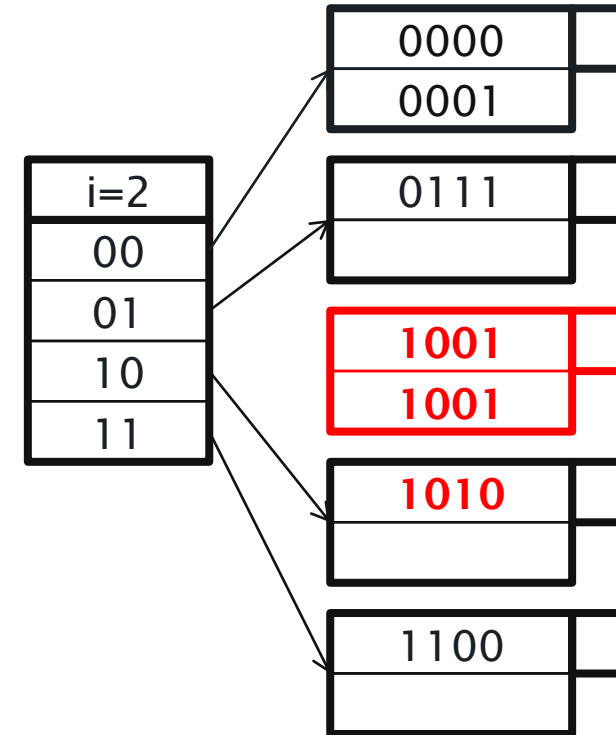
Example

Insert 1001



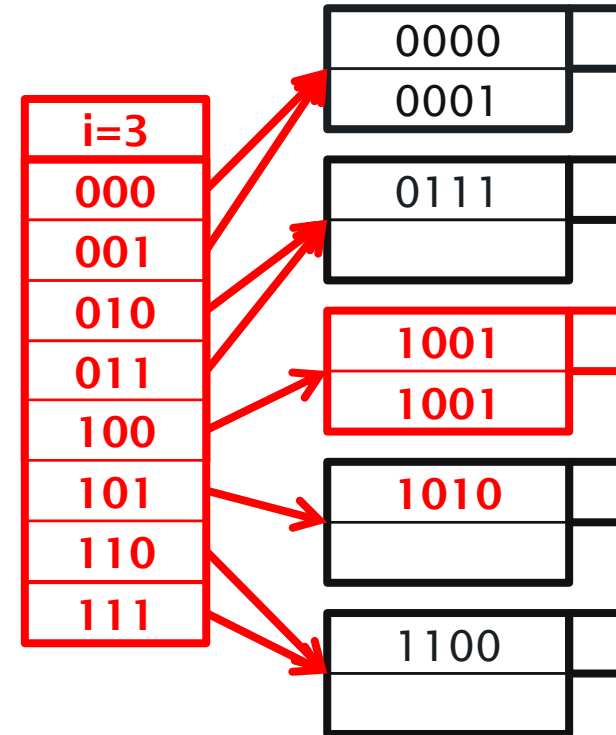
Example

Insert 1001



Example

Insert 1001



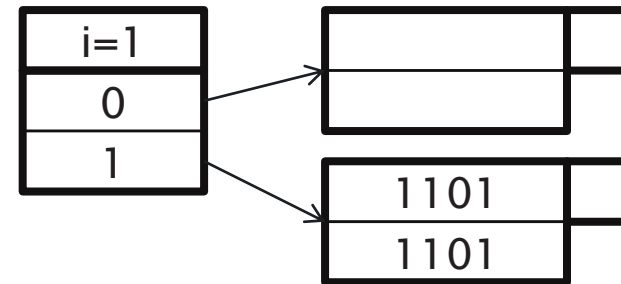
Extensible hashing: deletion

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

Overflow chains

Example: many records with duplicate keys

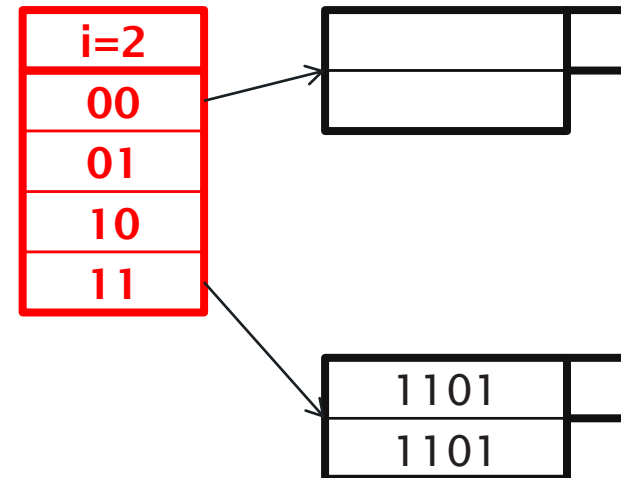
- Insert 1100



Overflow chains

Example: many records with duplicate keys

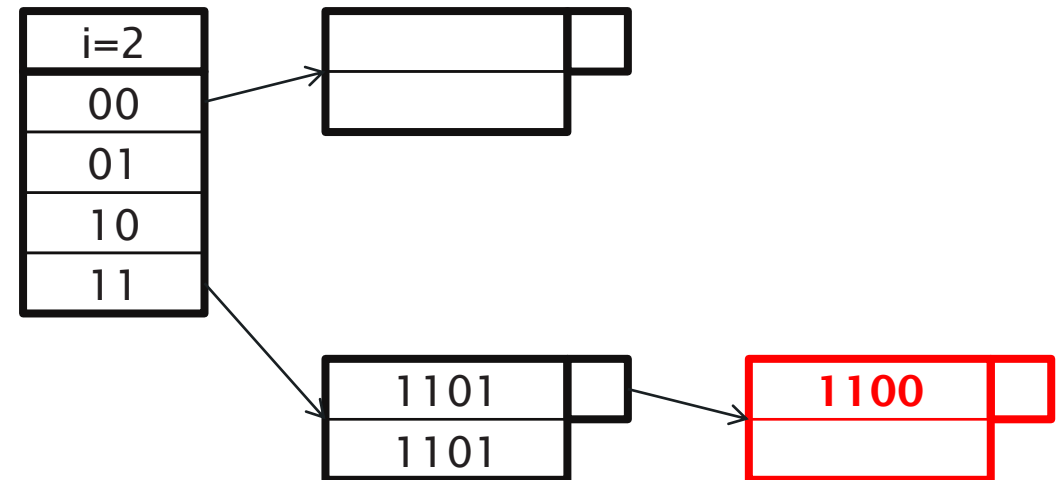
- Insert 1100



Overflow chains

Example: many records with duplicate keys

- Insert 1100
- Add overflow block



Summary

Pro

- Can handle growing files
 - with less wasted space
 - with no full reorganizations

Con

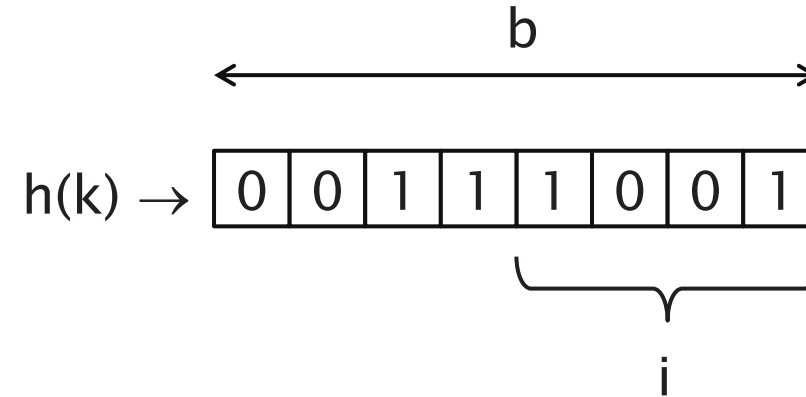
- Indirection
 - not bad if directory in memory
- Directory doubles in size
 - now it fits in memory, now it doesn't
 - suddenly increase in disk accesses!

Linear hashing

Another dynamic hashing scheme

Combines two ideas

1. Use i least significant bits of hash, where i grows over time

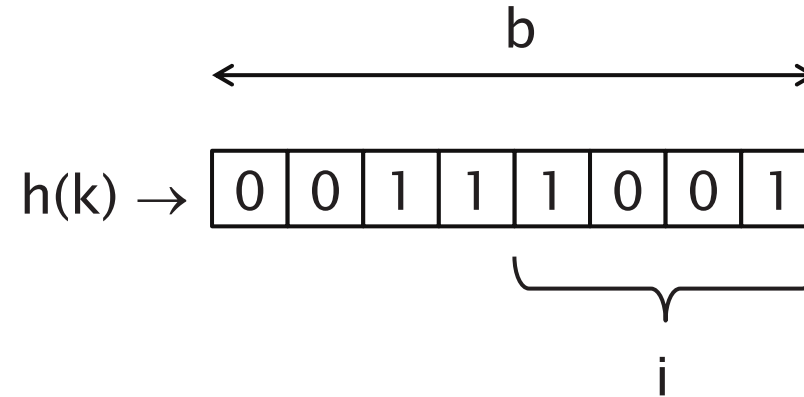


Linear hashing

Another dynamic hashing scheme

Combines two ideas

1. Use i least significant bits of hash, where i grows over time
2. Hash file grows incrementally and linearly
(unlike extensible hash file, which periodically doubles)



Linear hashing

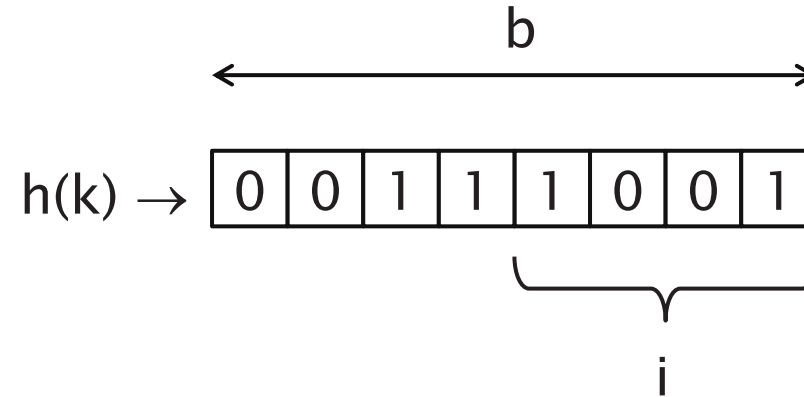
Another dynamic hashing scheme

Combines two ideas

1. Use i least significant bits of hash, where i grows over time
2. Hash file grows incrementally and linearly
(unlike extensible hash file, which periodically doubles)

Lookup rule:

if $h(k)[i] \leq m$ (maximum bucket index)
then look at bucket $h(k)[i]$
else look at bucket $h(k)[i] - 2^{i-1}$

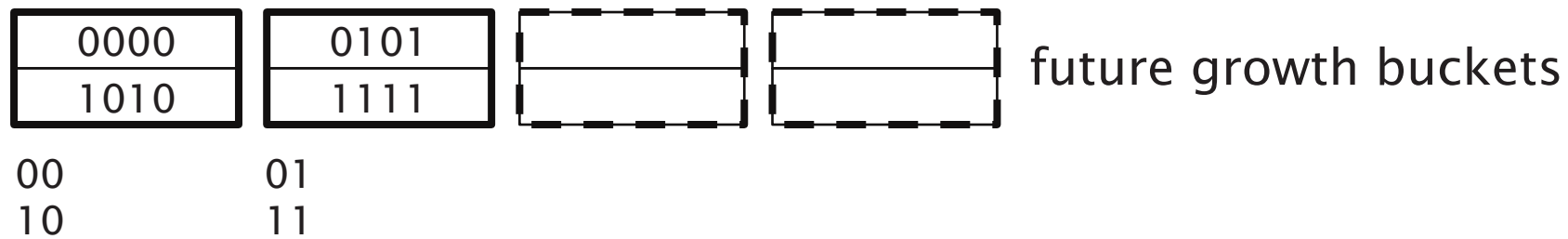


Example: $b=4$ bits, $i=1$, 2 keys/bucket

0000	0101
1010	1111
0	1

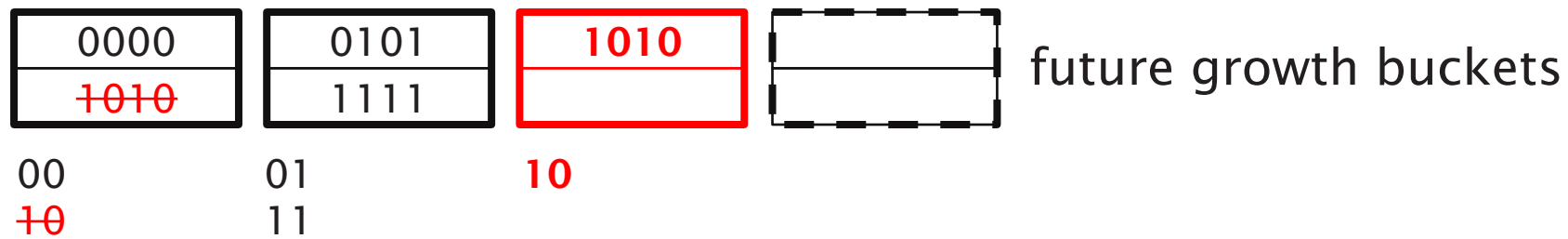
$m = \text{max used bucket} = 1$

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



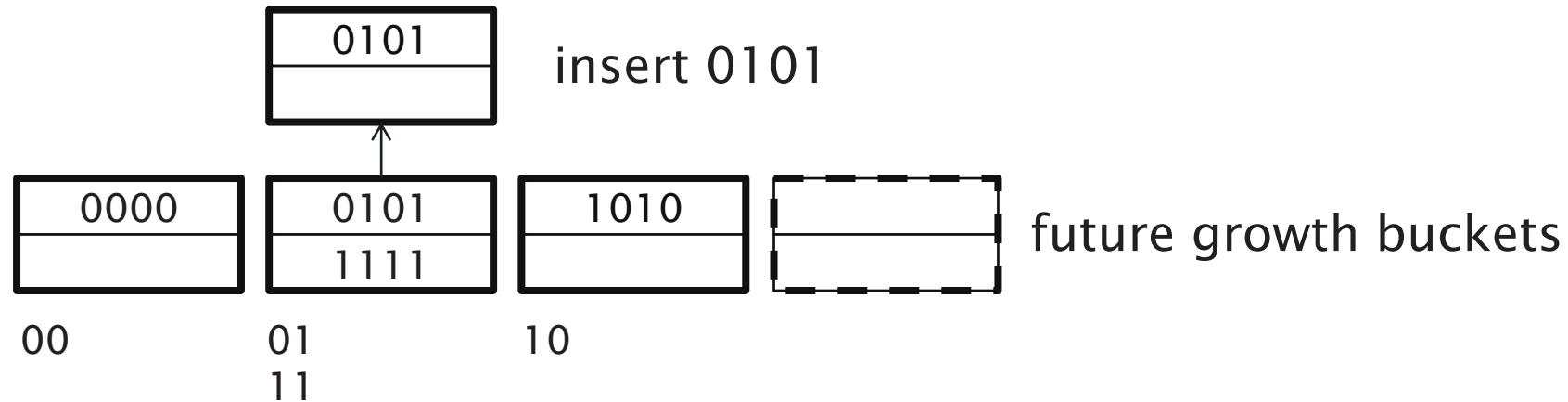
$m = \text{max used bucket} = 01$

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



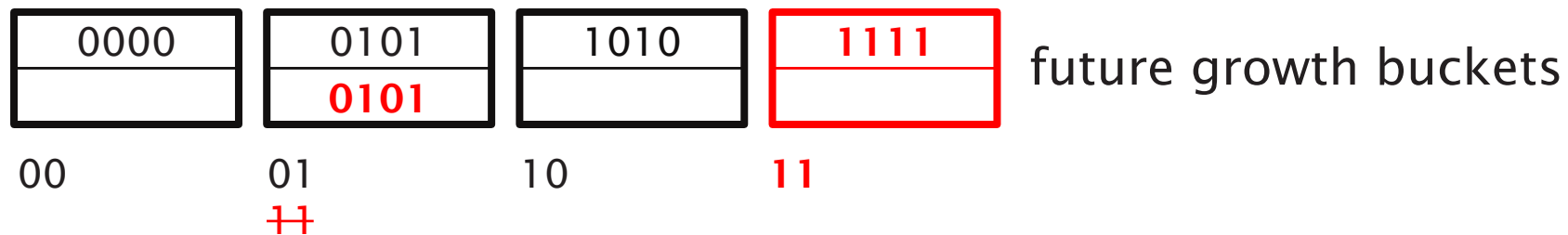
$m = \text{max used bucket} = 10$

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



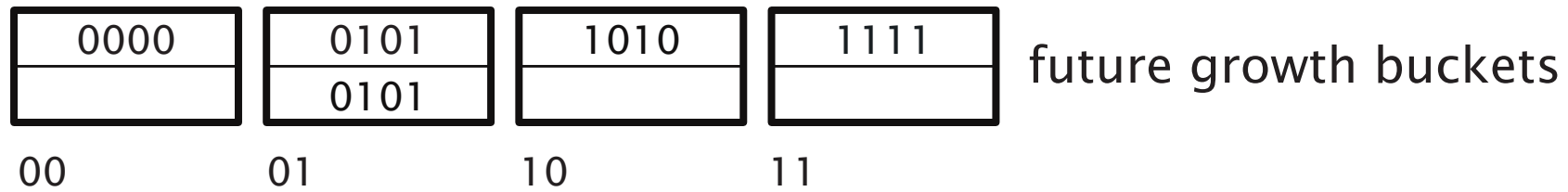
$m = \text{max used bucket} = 10$

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



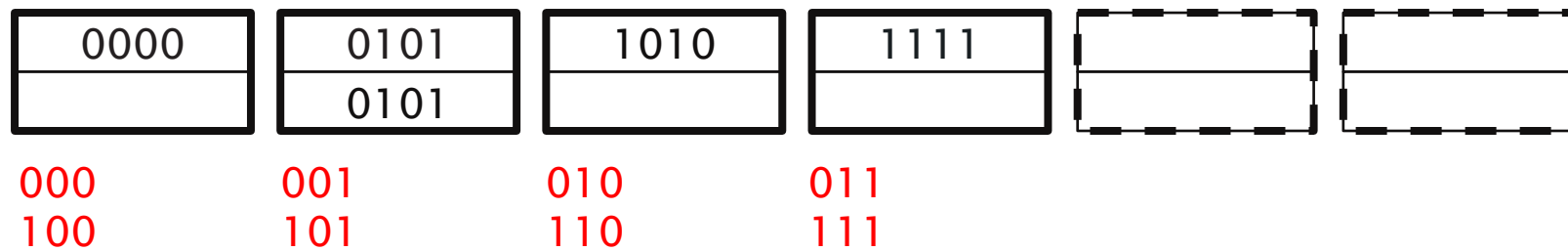
$m = \text{max used bucket} = 11$

Example: further growth



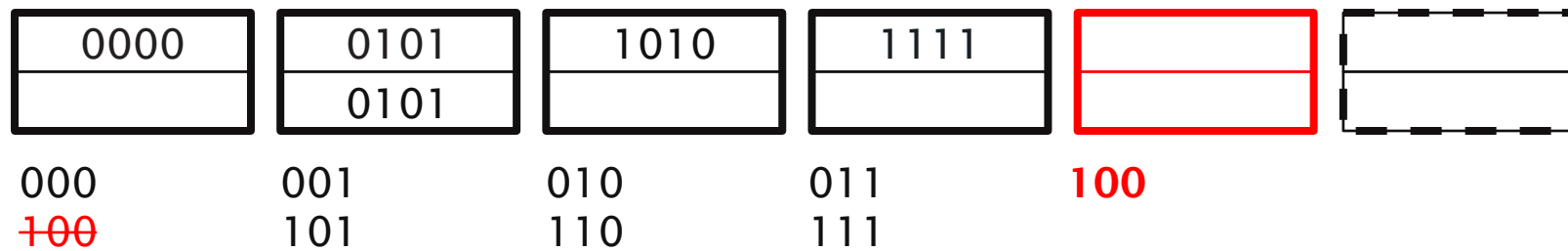
$m = \text{max used bucket} = 11$

Example: $i=3$



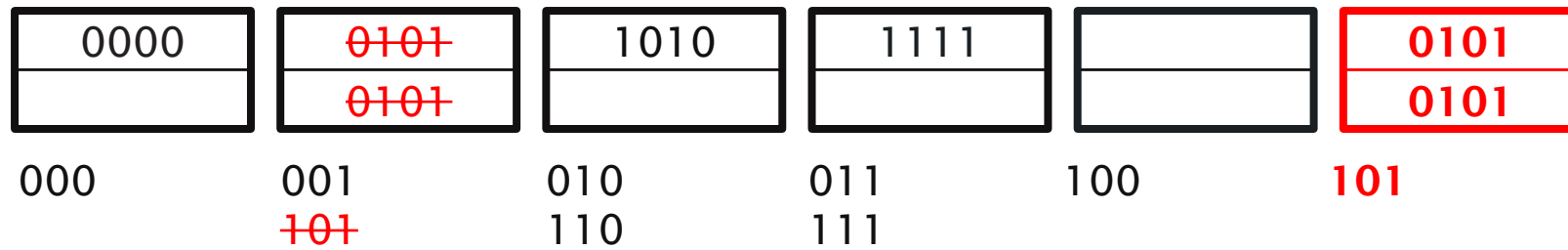
$m = \text{max used bucket} = 11$

Example: $i=3$



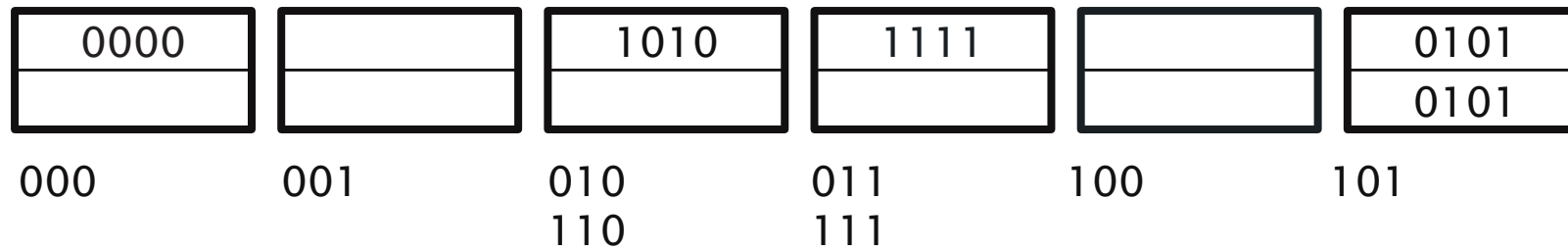
$m = \text{max used bucket} = \mathbf{100}$

Example: $i=3$



$m = \text{max used bucket} = \mathbf{101}$

Example: $i=3$



$m = \text{max used bucket} = 101$

When do we expand file?

Keep track of utilisation

$$U = \text{\#used slots} / \text{total \#slots}$$

If $U > \text{threshold}$, then increase m (and maybe i)

Linear Hashing

Pro

- Can handle growing files
 - with less wasted space
 - with no full reorganizations
- No indirection like extensible hashing

Con

- Can still have overflow chains

Indexing versus Hashing

Indexing vs Hashing

Hashing good for *probes* given a key:

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

Indexing vs Hashing

Indexing (Including B-trees) good for *range searches*:

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

Multidimensional Access Structures

COMP3211 Advanced Databases

Dr Nicholas Gibbins – nmg@ecs.soton.ac.uk

Overview

- Conventional indexes
- Hash-like
 - grid files, partitioned hashing
- Hierarchical indexes
 - multiple key, kd-trees, quad trees, r-trees, ub-trees
- Bitmap indexes

Multidimensional Access Structures

Indexes discussed so far are one-dimensional

- assume a single search key
- require a single linear order for keys (B-trees)
- require that the key be completely known for any lookup (hash tables)

Applications

Geographic information systems

- partial match queries
- range queries
- nearest-neighbour queries

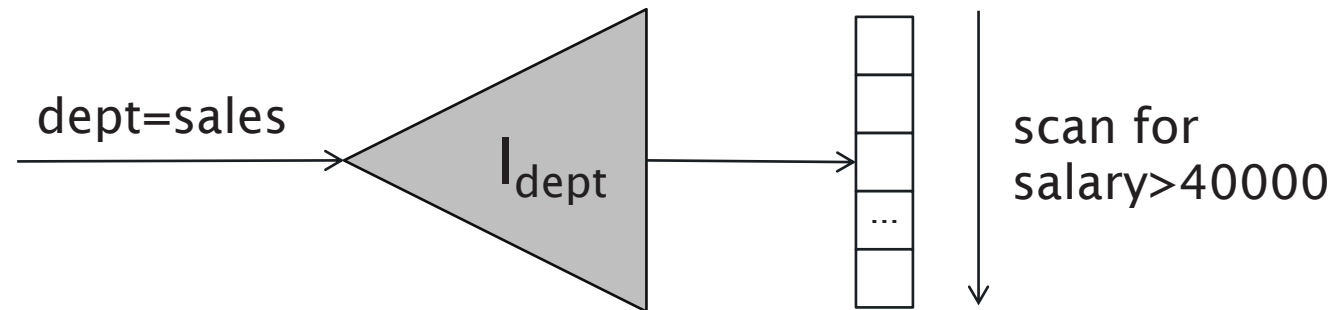
Conventional Indexes

Scenario

- Personnel database
- EMPLOYEE table with attributes
 - dept
 - salary
- How can we find employees who work in the sales department and have salaries greater than £40,000?

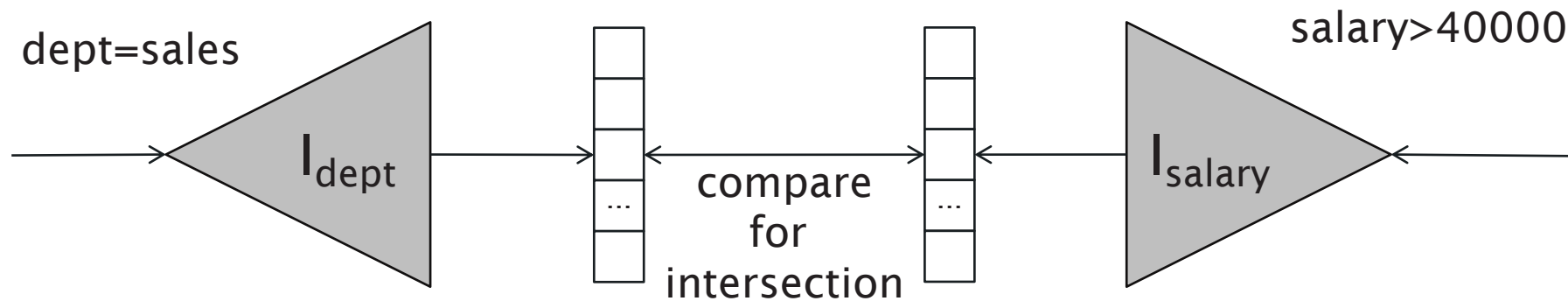
Approach #1

1. Get all matching records using an index on one attribute
2. Check values of other attribute on those records



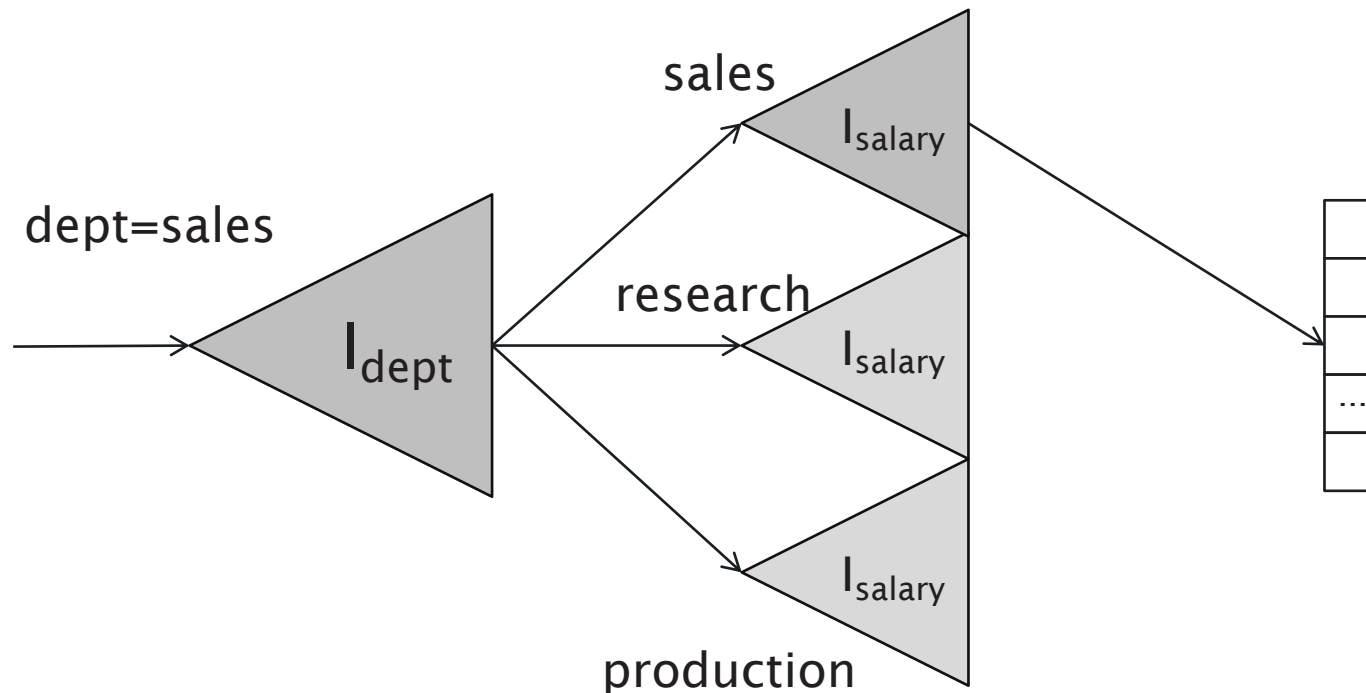
Approach #2

1. Use secondary indexes on each attribute to get two sets of record pointers
2. Take intersection of sets



Approach #3

1. Use secondary index on one attribute to select suitable index on other attribute
2. Get all matching records using selected index



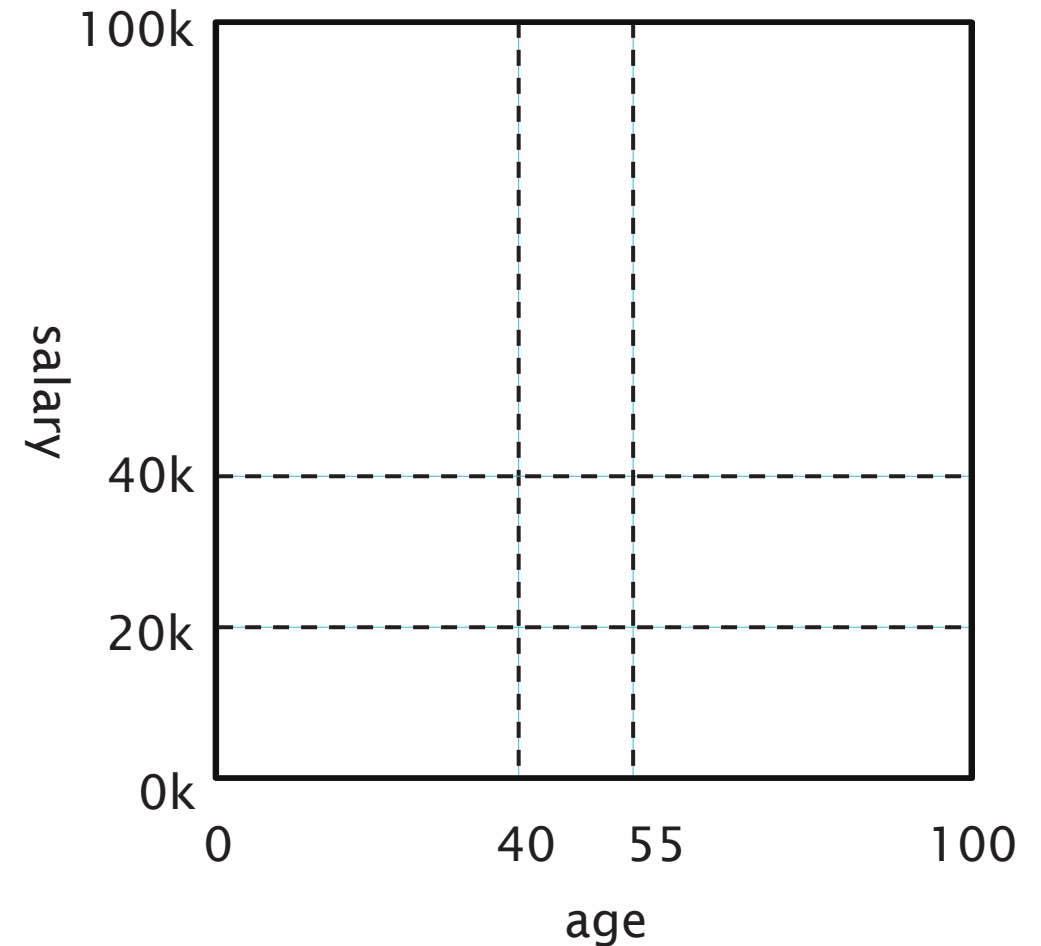
For which queries is this index good?

- `dept=sales ^ salary=40000`
- `dept=sales ^ salary>40000`
- `dept=sales`
- `salary = 40000`

Grid Files

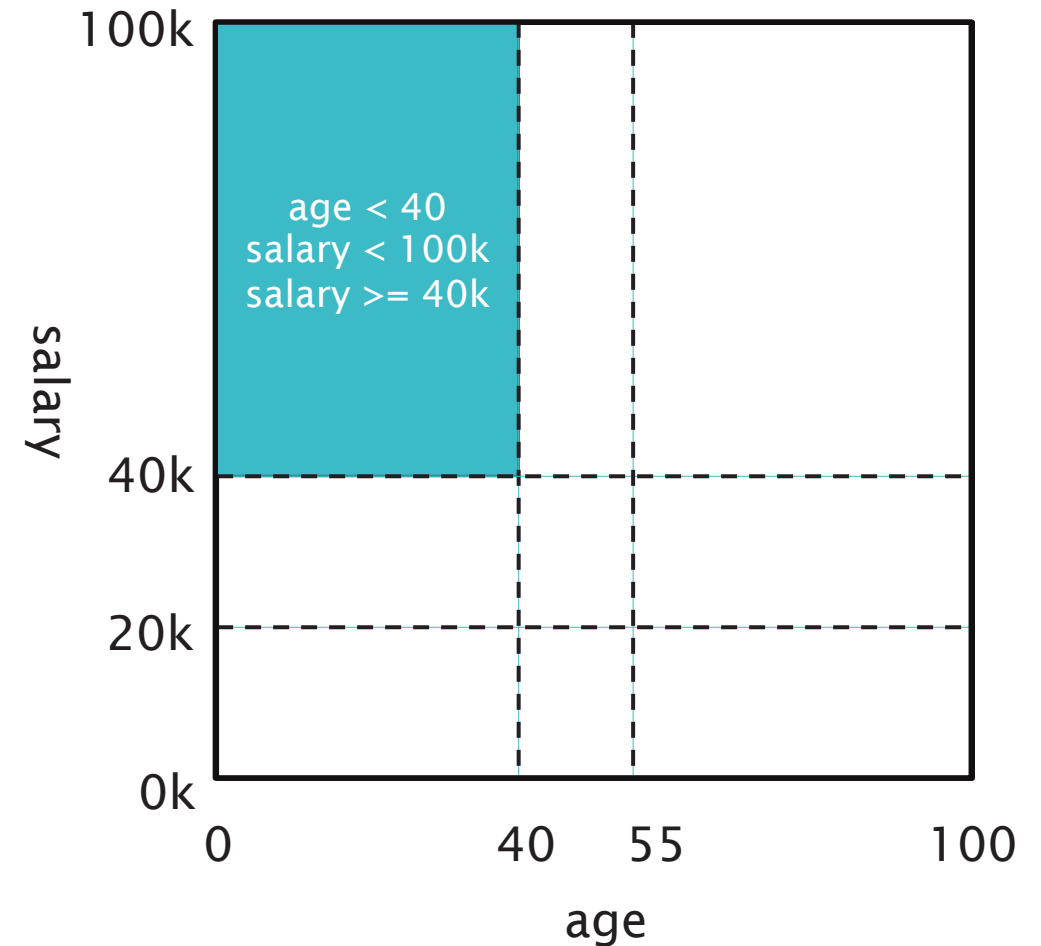
Grid File

- Partition multi-dimensional space with a grid
- Grid lines partition space into stripes
- Intersections of stripes from different dimensions define regions



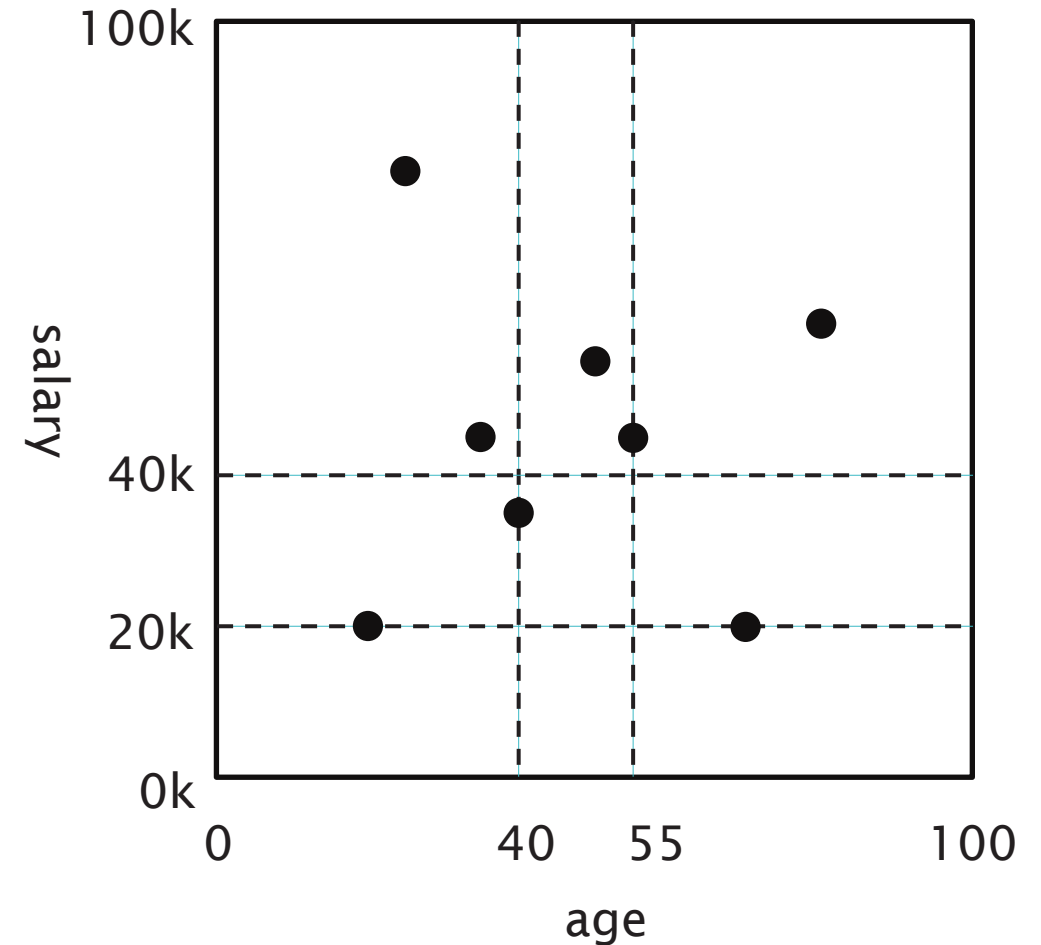
Grid File

- Partition multi-dimensional space with a grid
- Grid lines partition space into stripes
- Intersections of stripes from different dimensions define regions



Grid File

- Each region associated with a pointer to a bucket of record pointers
- Attribute values for record determine region and therefore bucket
- Fixed number of regions – overflow blocks used to increase bucket size as necessary
- Can index grid on value ranges



Grid files

Pro

- Good for multiple-key search
- Supports partial-match, range and nearest-neighbour queries

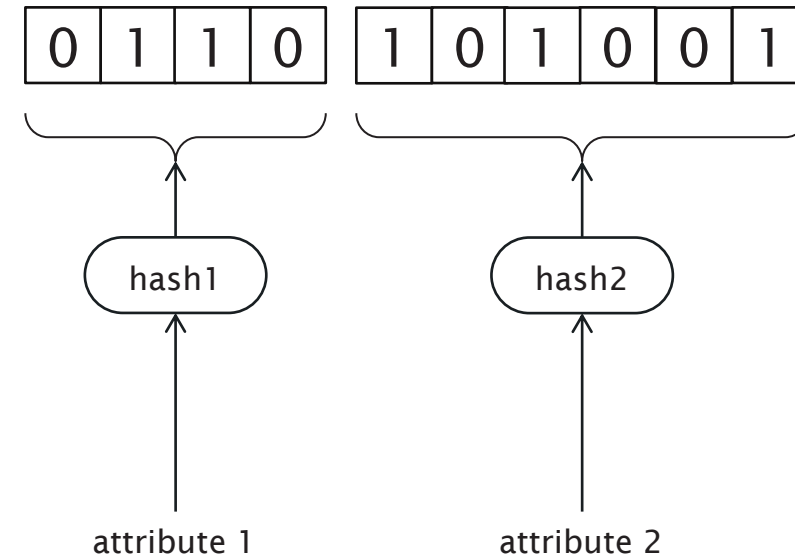
Con

- Space, management overhead (nothing is free)
- Need partitioning ranges that evenly split keys

Partitioned Hash

Partitioned Hash

- Hash function takes a list of attribute values as arguments
- Bits of hash value divided between attributes
 - Effectively, a hash function per attribute



Example

hash1(sales) = 0

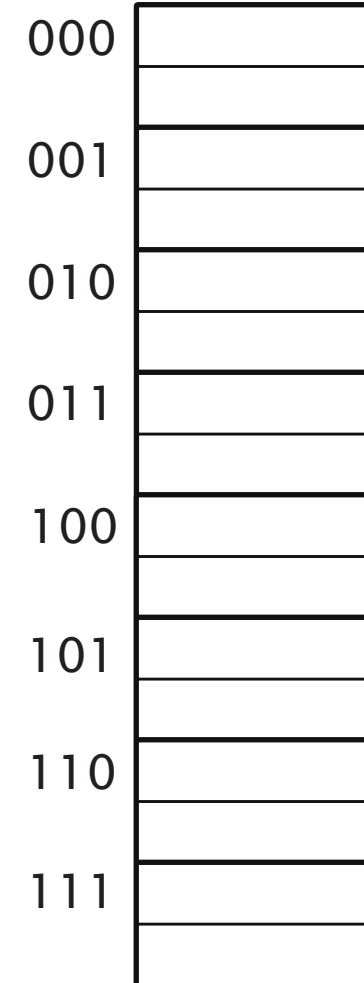
hash1(research) = 1

hash2(10000) = 00

hash2(20000) = 01

hash2(40000) = 10

hash2(100000) = 11



Insertion

hash1(sales) = 0

hash1(research) = 1

hash2(10000) = 00

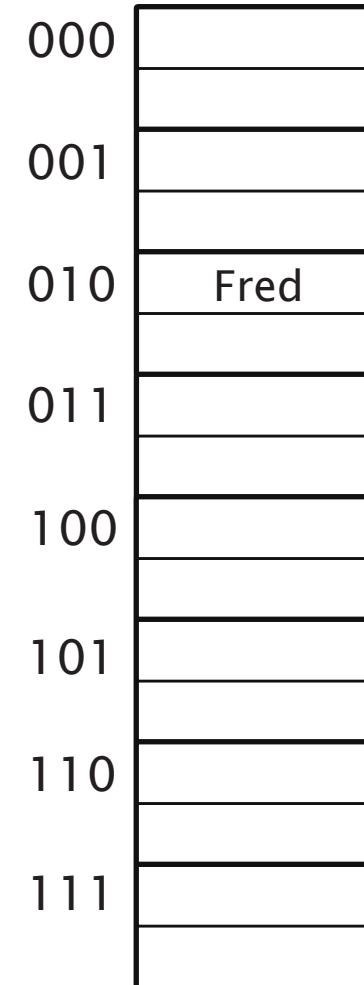
hash2(20000) = 01

hash2(40000) = 10

hash2(100000) = 11

Fred works in sales

Fred's salary is £40,000



Retrieval

hash1(sales) = 0

hash1(research) = 1

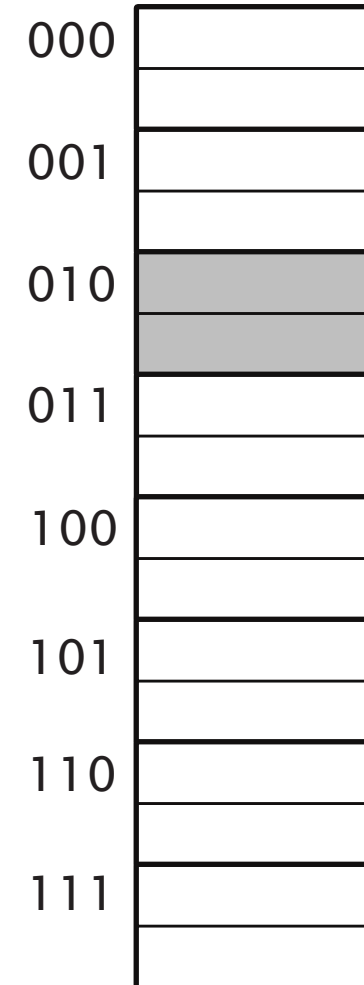
hash2(10000) = 00

hash2(20000) = 01

hash2(40000) = 10

hash2(100000) = 11

dept=sales \wedge salary=40000



Retrieval

hash1(sales) = 0

hash1(research) = 1

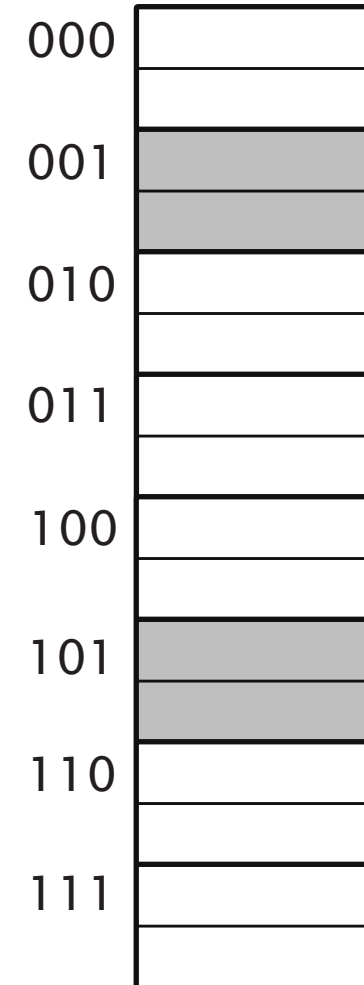
hash2(10000) = 00

hash2(20000) = 01

hash2(40000) = 10

hash2(100000) = 11

salary=20000



Retrieval

hash1(sales) = 0

hash1(research) = 1

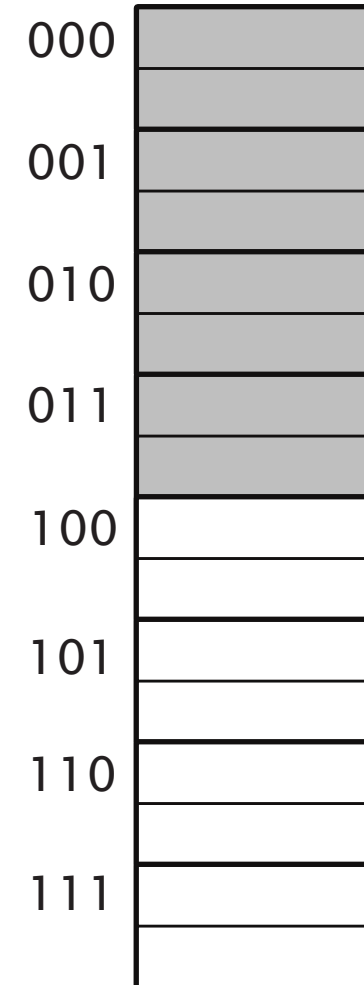
hash2(10000) = 00

hash2(20000) = 01

hash2(40000) = 10

hash2(100000) = 11

dept=sales



Partitioned hash

Pro

- Good hash function will evenly distribute records between buckets
- Supports partial-match queries

Con

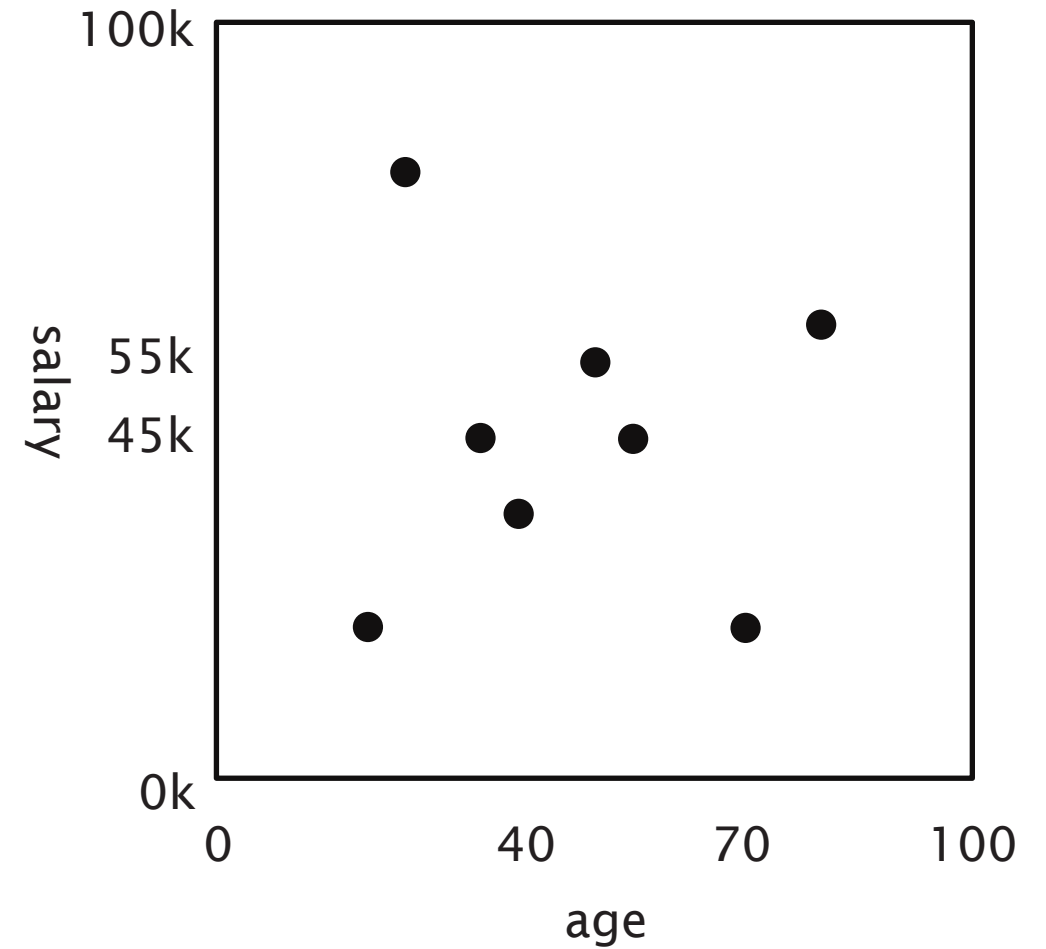
- No good for nearest-neighbour or range queries

kd-Tree

kd-Tree

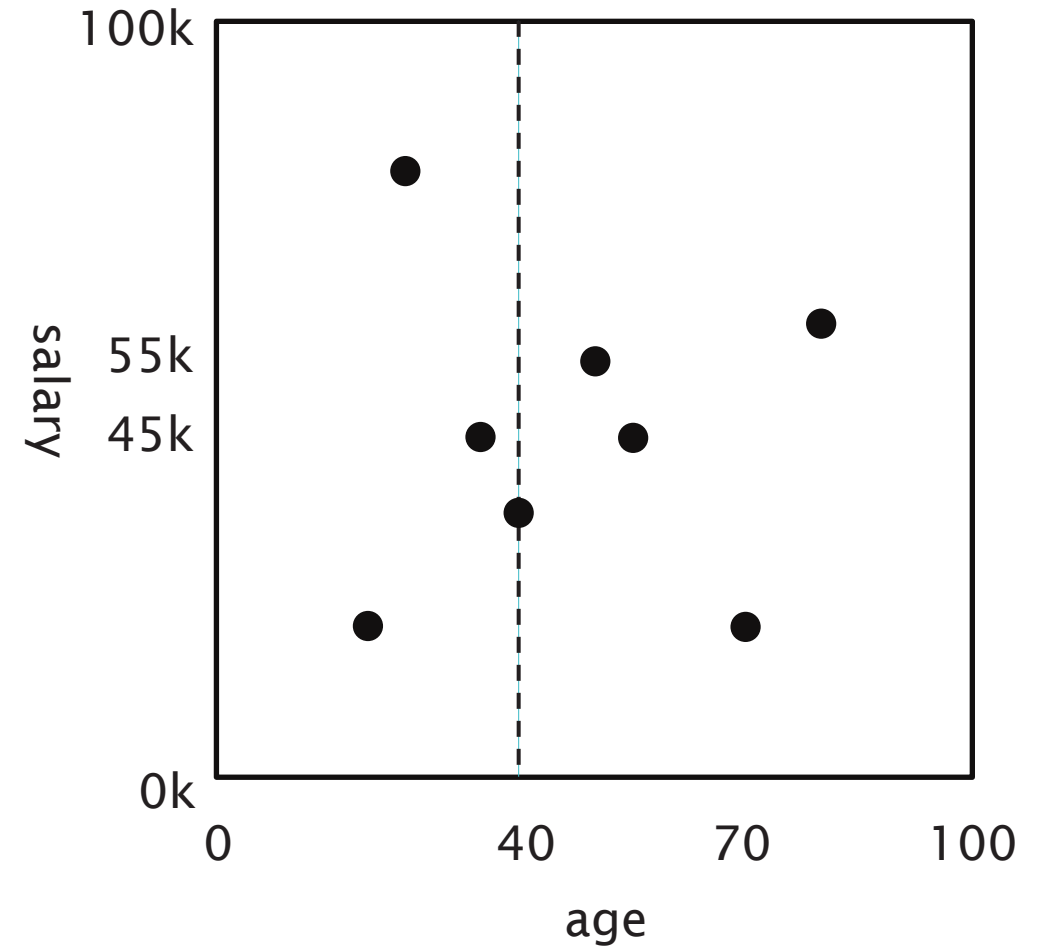
- Multidimensional binary search tree
- Each node splits the k-dimensional space along a hyperplane
- Nodes contain
 - an attribute-value pair
 - a pair of pointers
- All nodes at the same level discriminate for the same attribute
- Levels rotate between attributes of all dimensions

Example, $k=2$

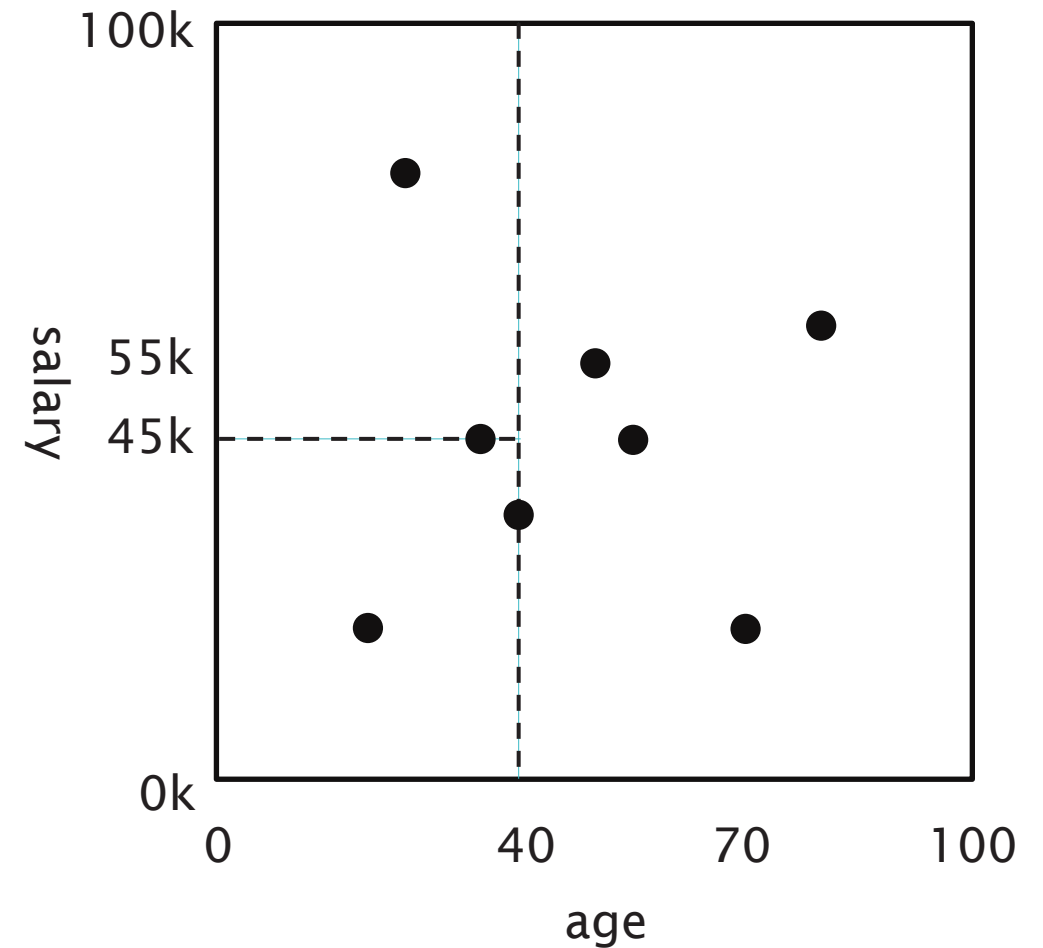
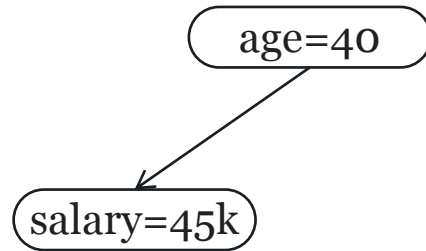


Example, $k=2$

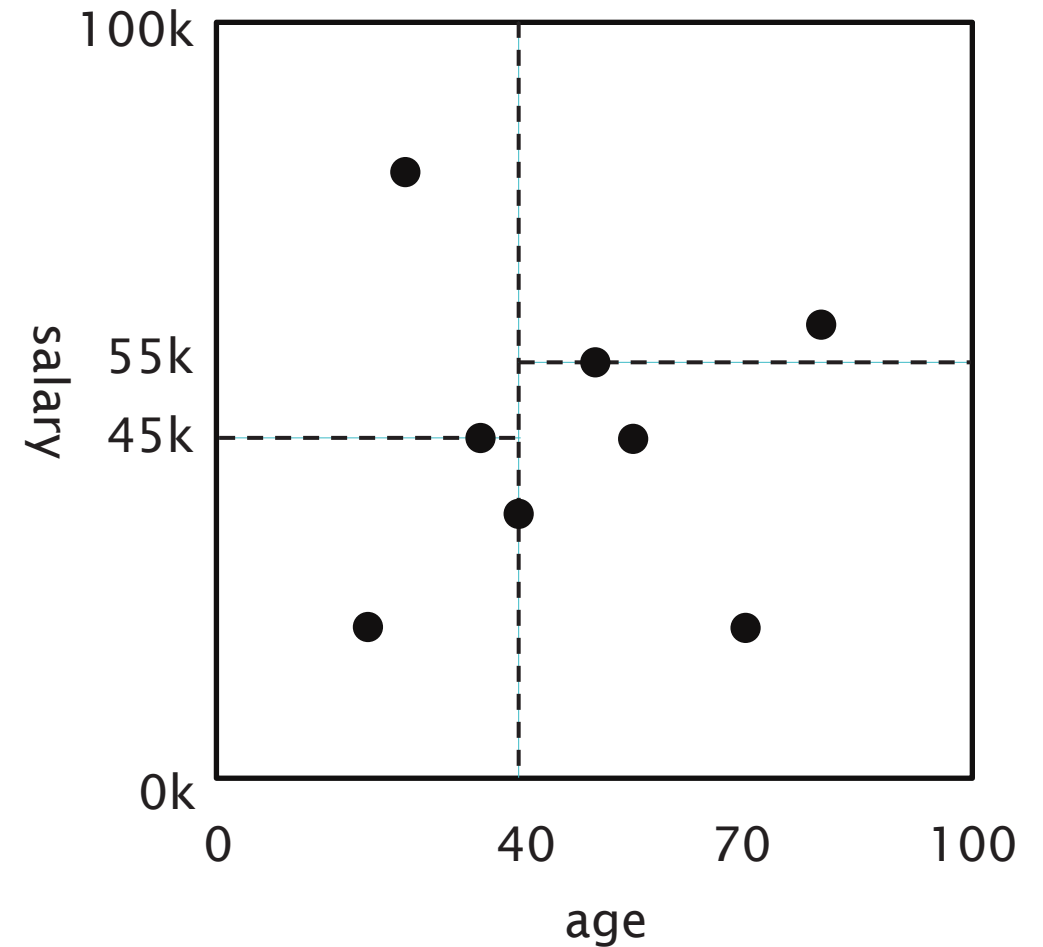
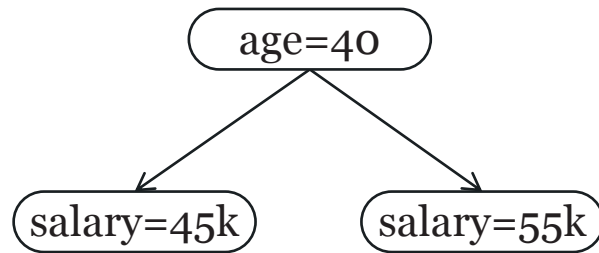
age=40



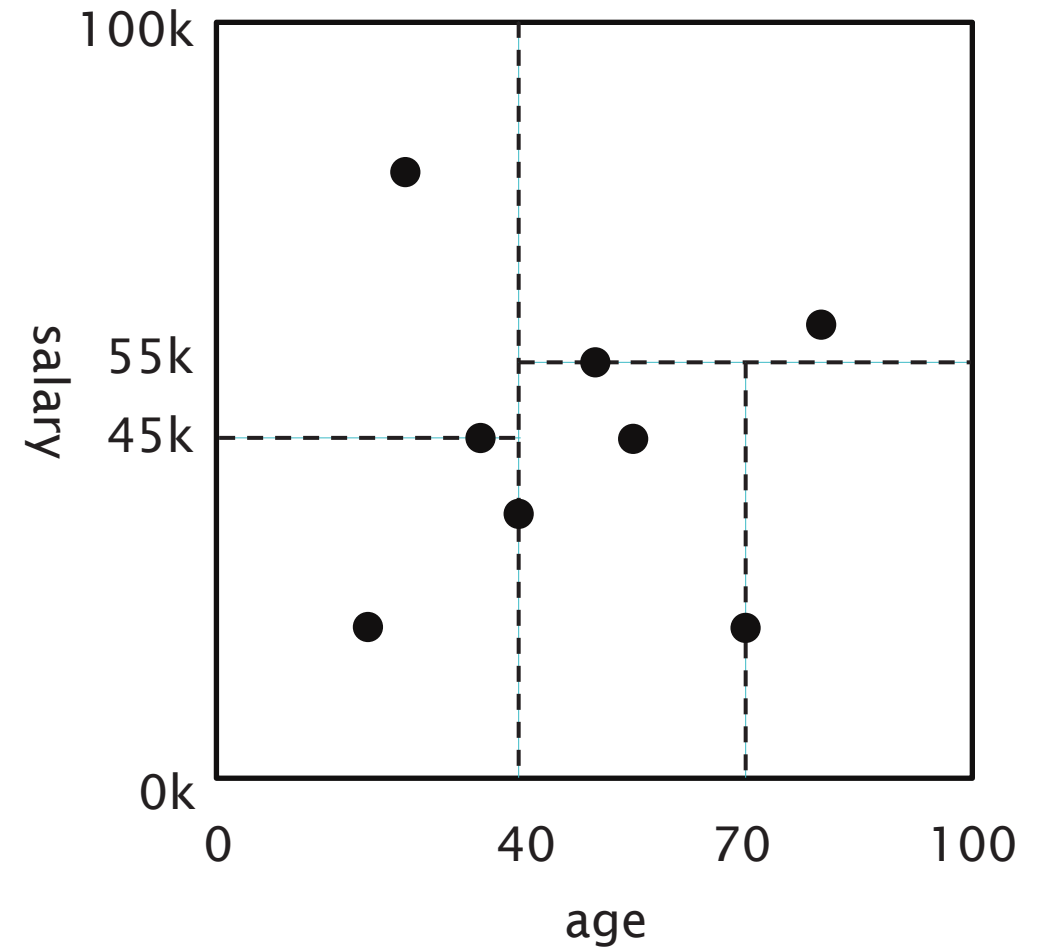
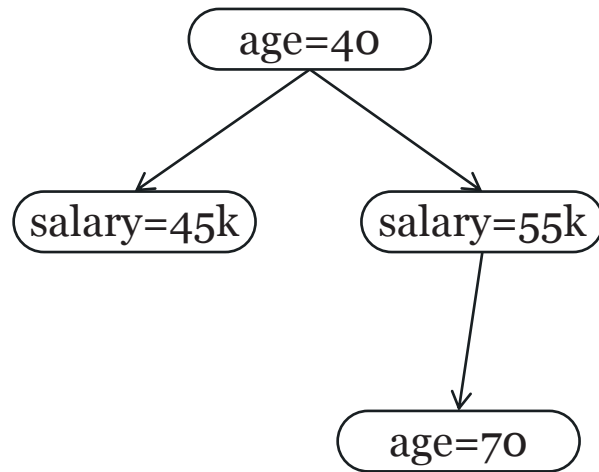
Example, $k=2$



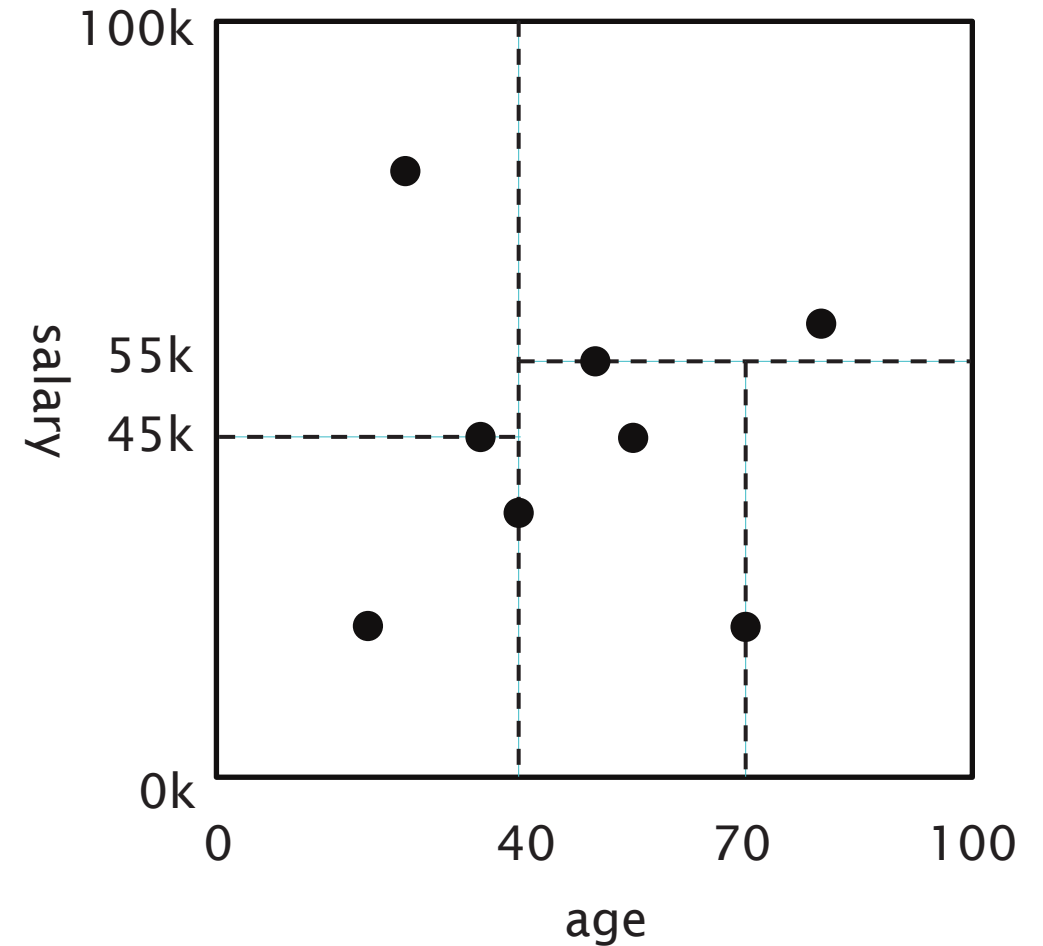
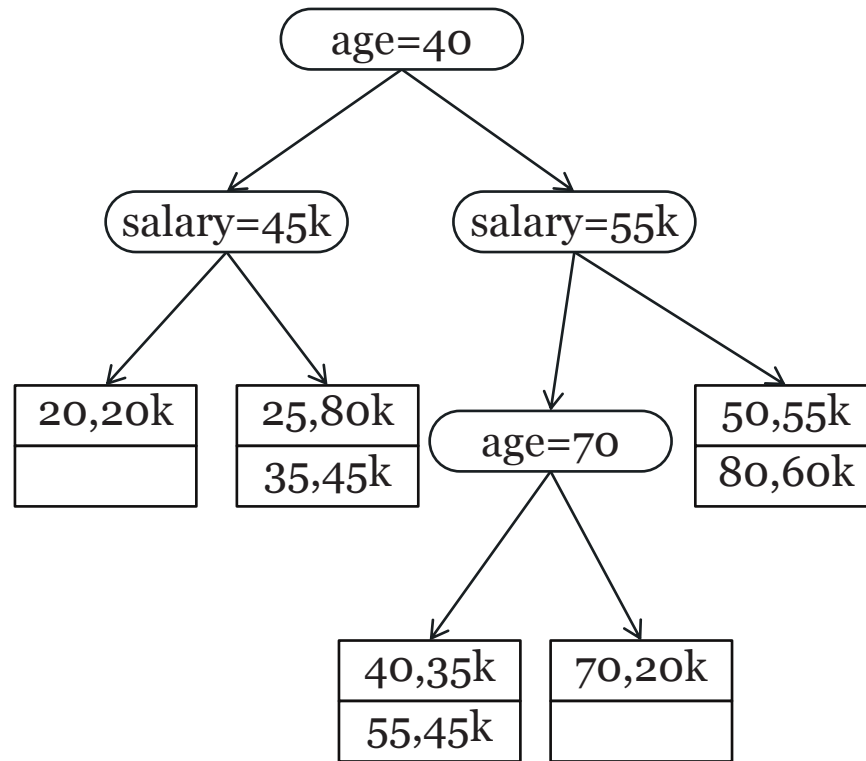
Example, $k=2$



Example, $k=2$



Example, k=2



Partial-Match Queries

- If we know value of attribute, we can choose which branch to explore
- If we don't know value of attribute, must explore both branches

Adapting kd-Trees to Secondary Storage

Average path length from root to leaf: $\log_2 n$

Disk accesses should be kept as few as possible

Two approaches:

1. Multiway nodes (split values into n ranges)
2. Group nodes in blocks (node plus descendants to a given ply)

Quad-Tree

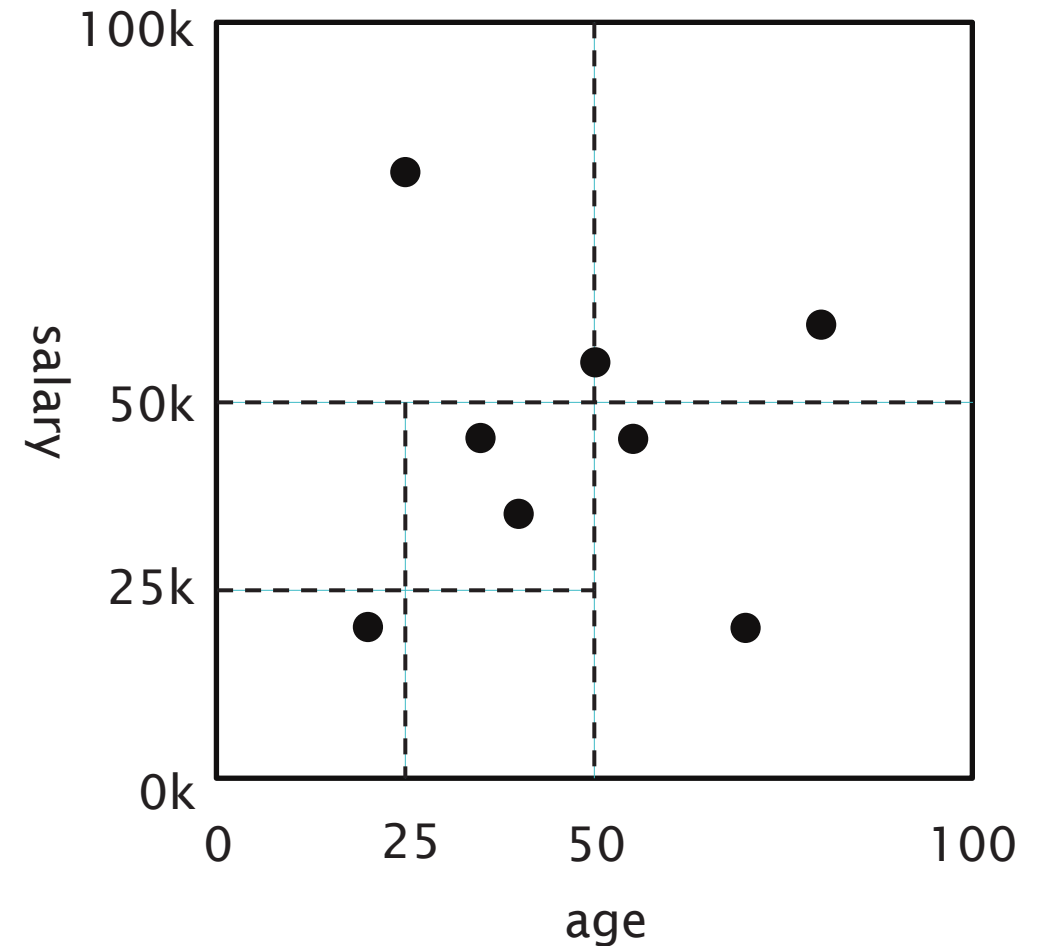
Quad-Trees

Two main types:

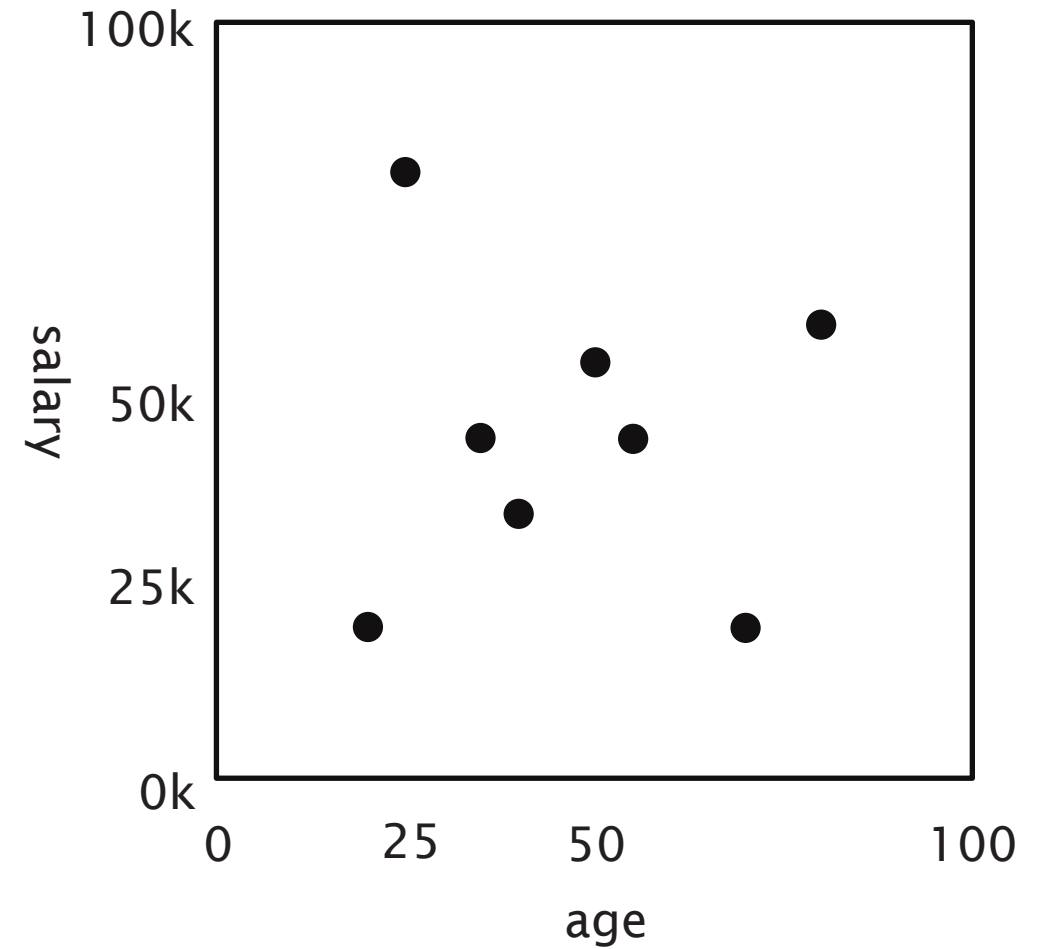
- Region quad-tree
- Point quad-tree

Region Quad-tree

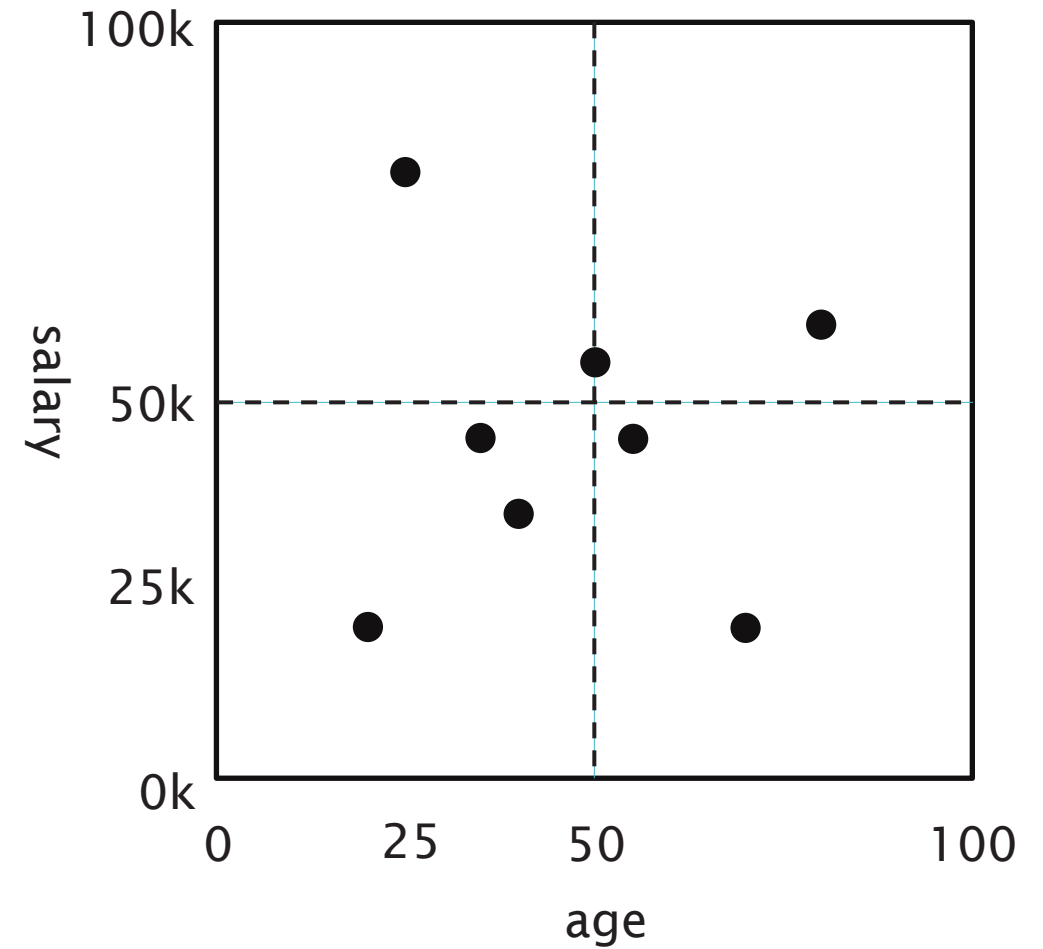
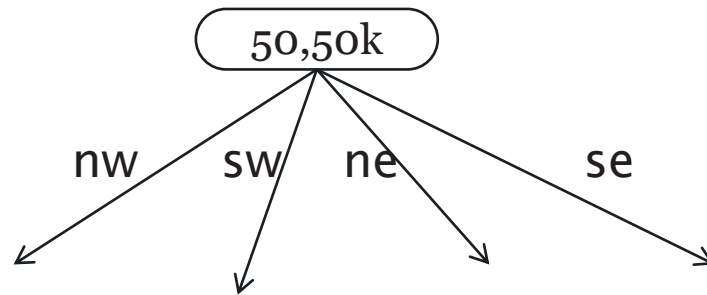
- Each partition divides the space into four equal area sub-regions
 - ne, nw, se, sw
- Split regions if they contain more records than will fit into a block
- Operations similar to those for kd-trees



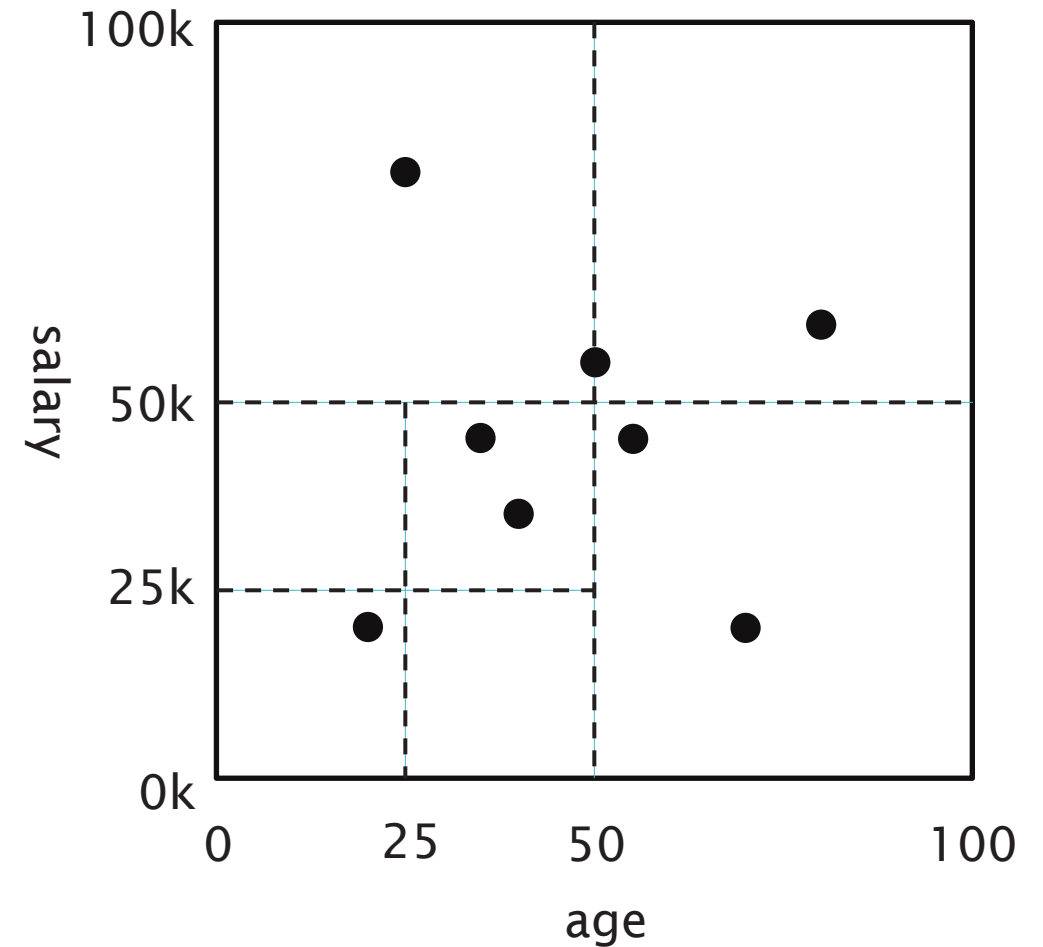
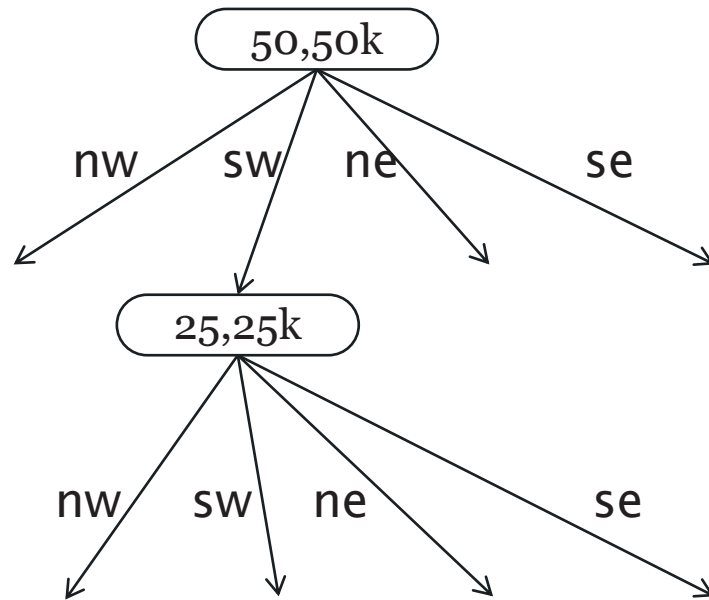
Region Quad-tree



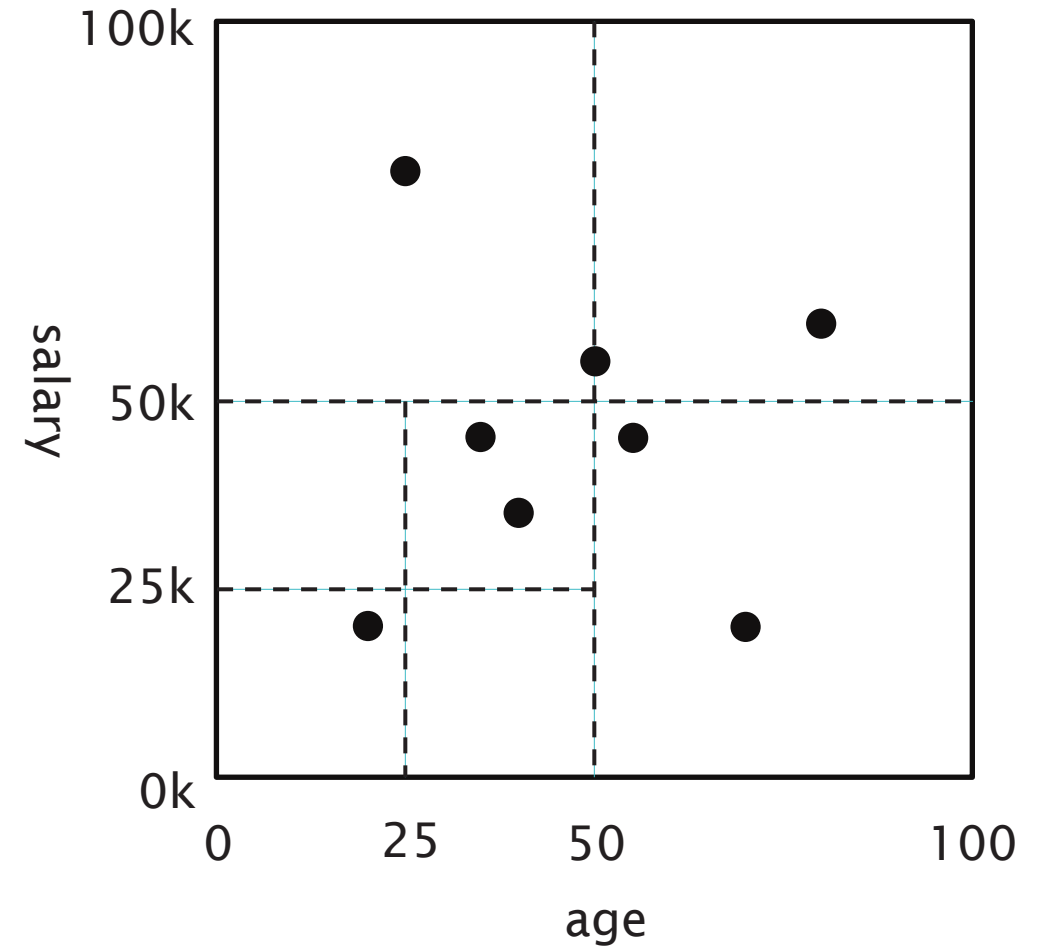
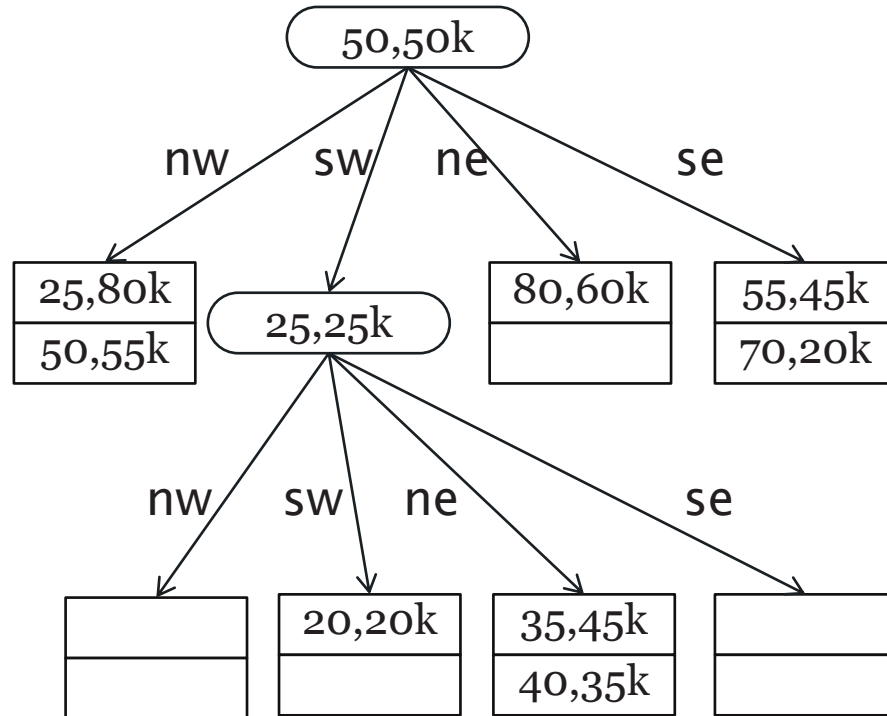
Region Quad-tree



Region Quad-tree

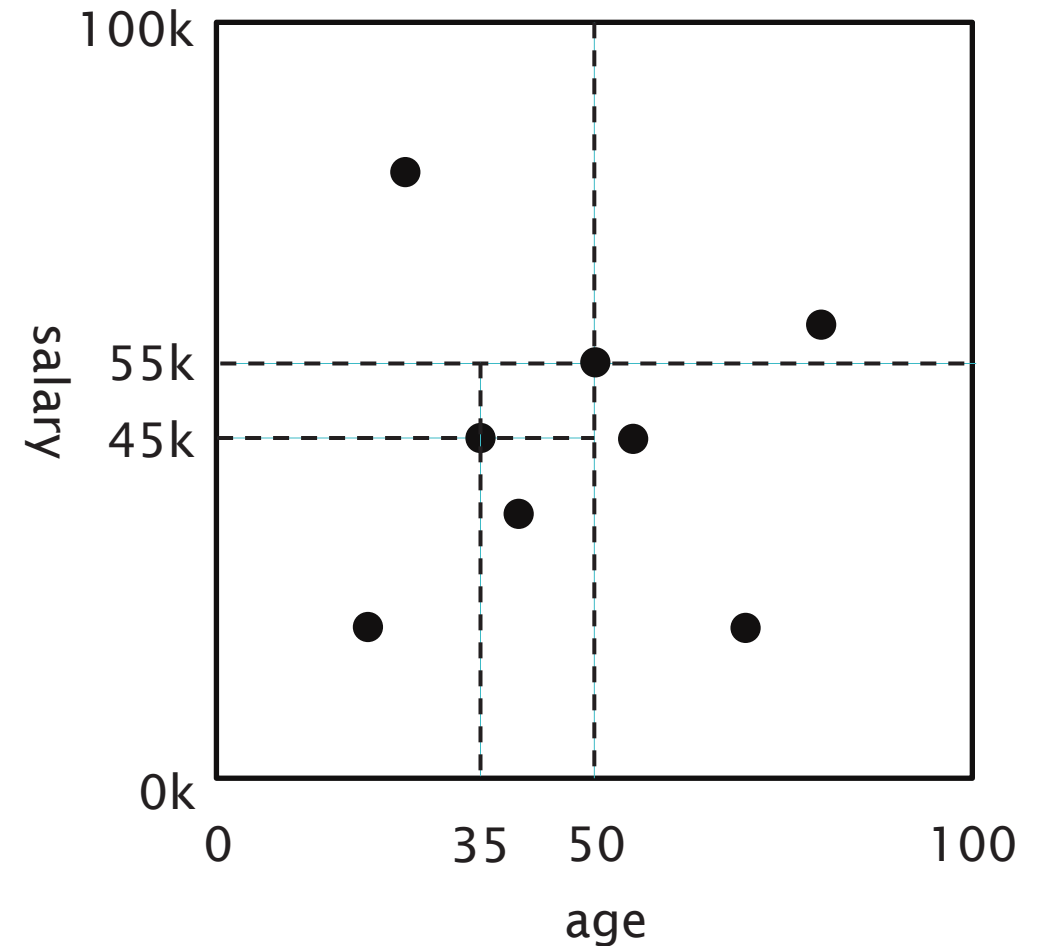


Region Quad-tree

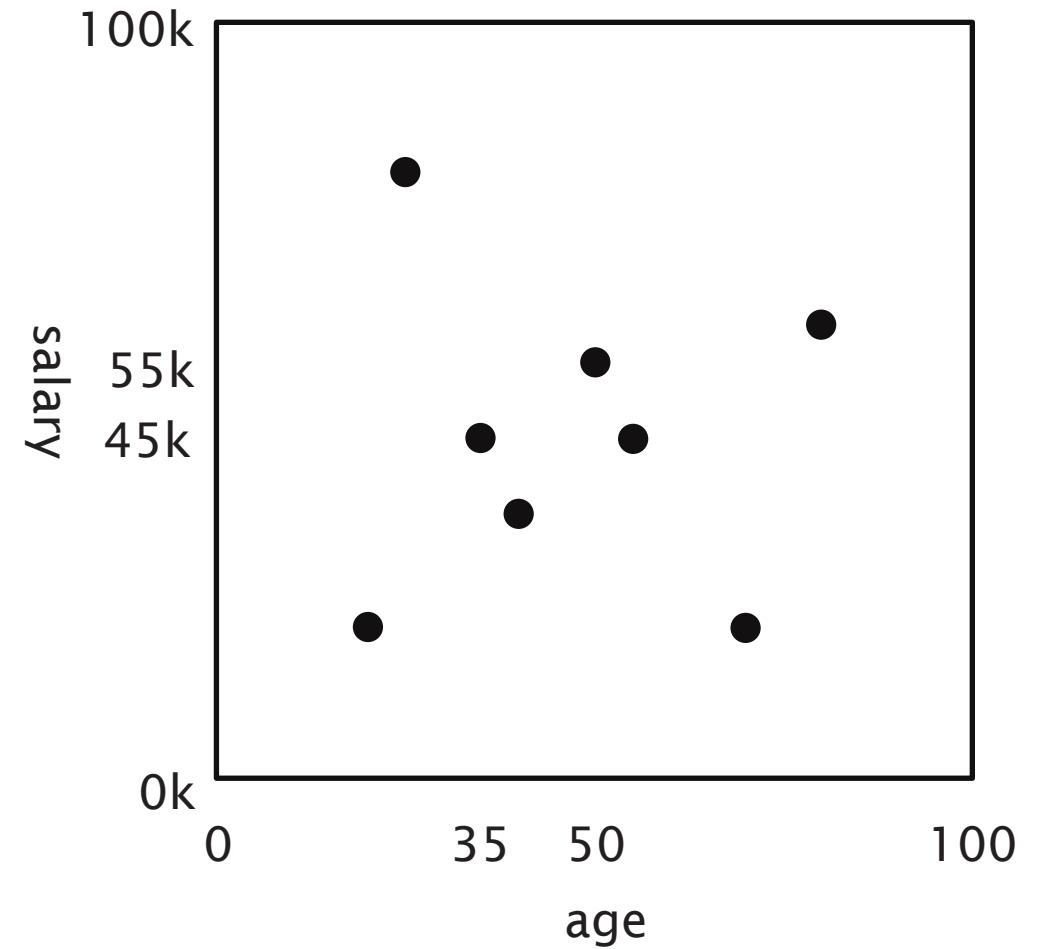


Point Quad-Tree

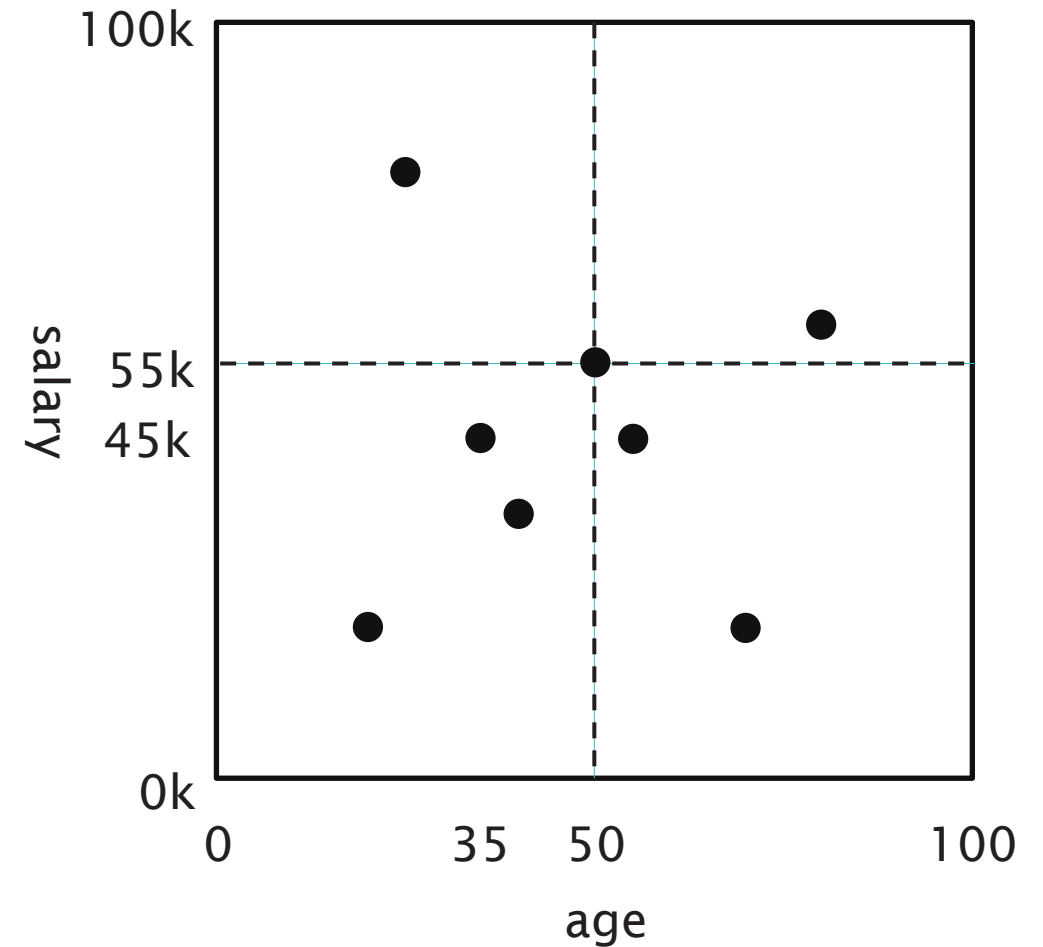
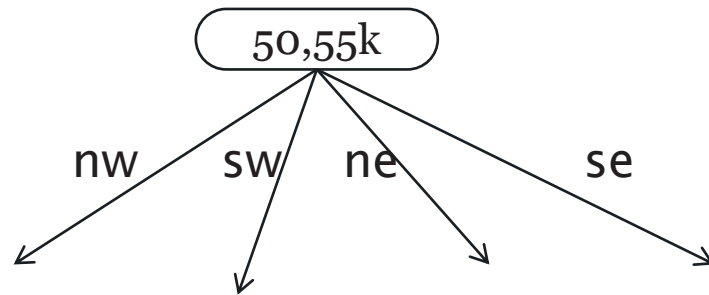
- Partitions are not equal area
 - Split lines centred on data points
 - ne/nw/se/sw sub-regions
- Otherwise, equivalent to region quad-tree



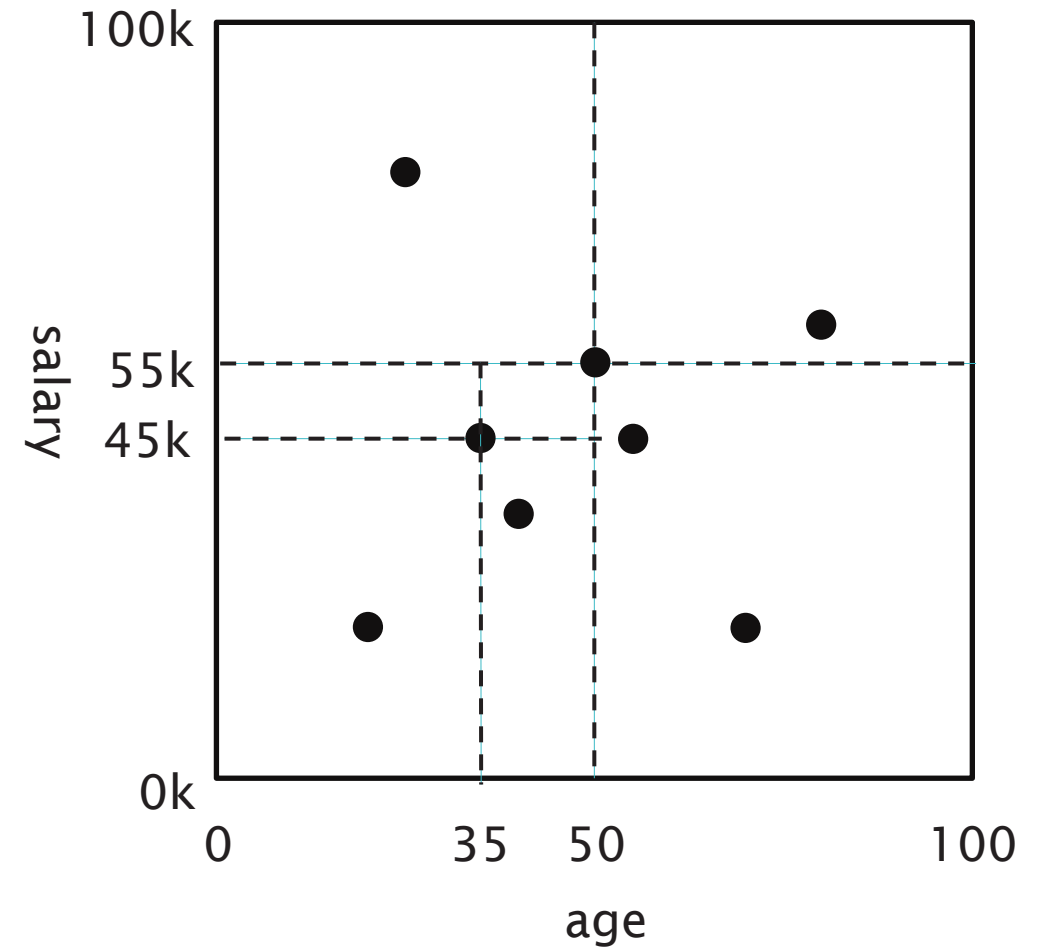
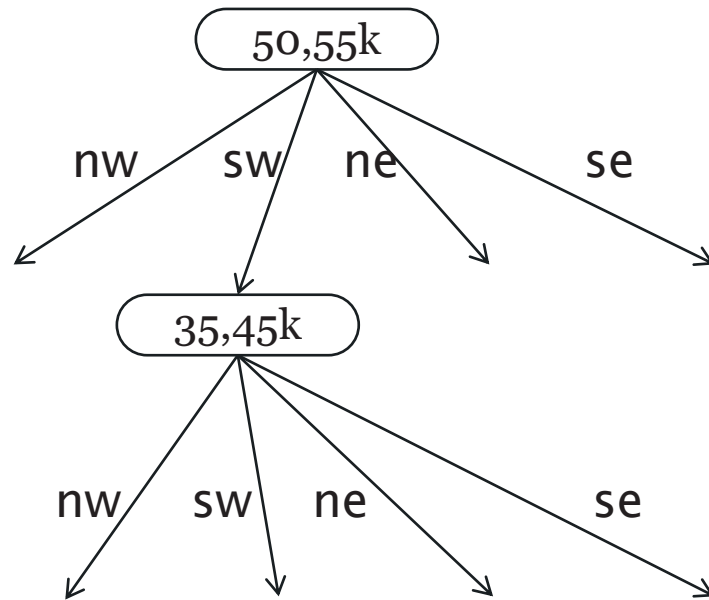
Point Quad-Tree



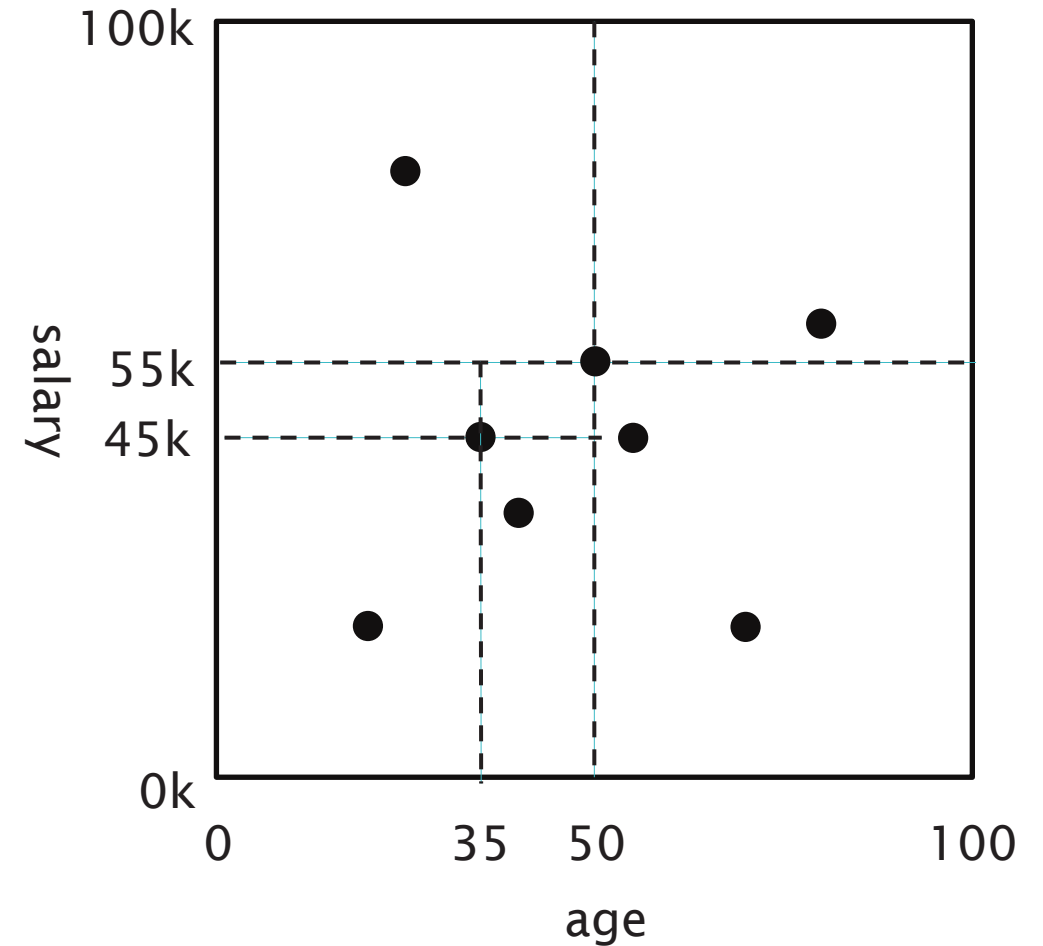
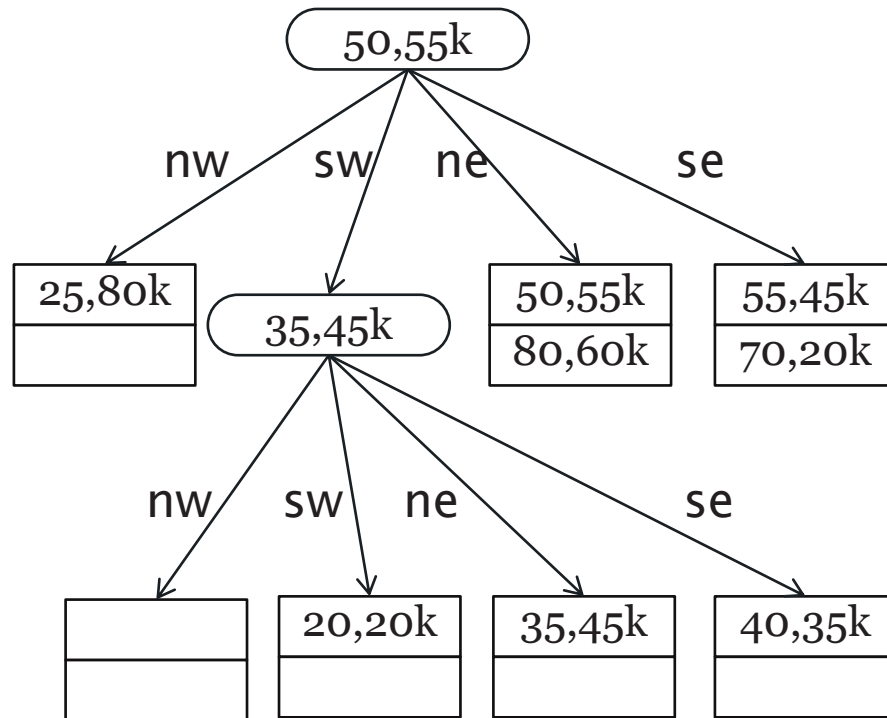
Point Quad-Tree



Point Quad-Tree



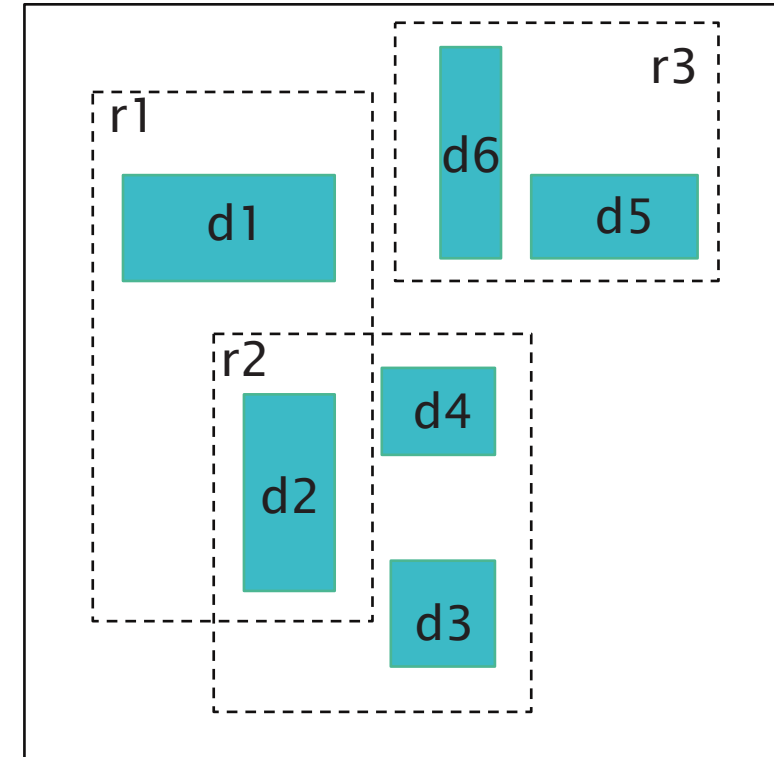
Point Quad-Tree



R-Tree

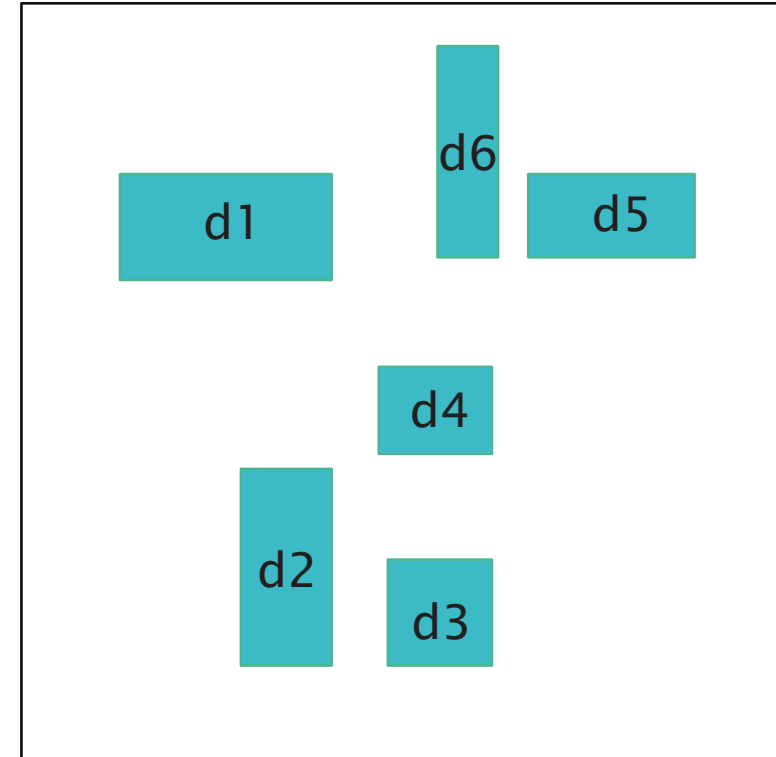
R-Trees

- Used to represent data that consists of k -dimensional *data regions*
- Internal nodes of tree represent regions that contain data regions
- Regions typically defined as top-right, bottom-left coordinates

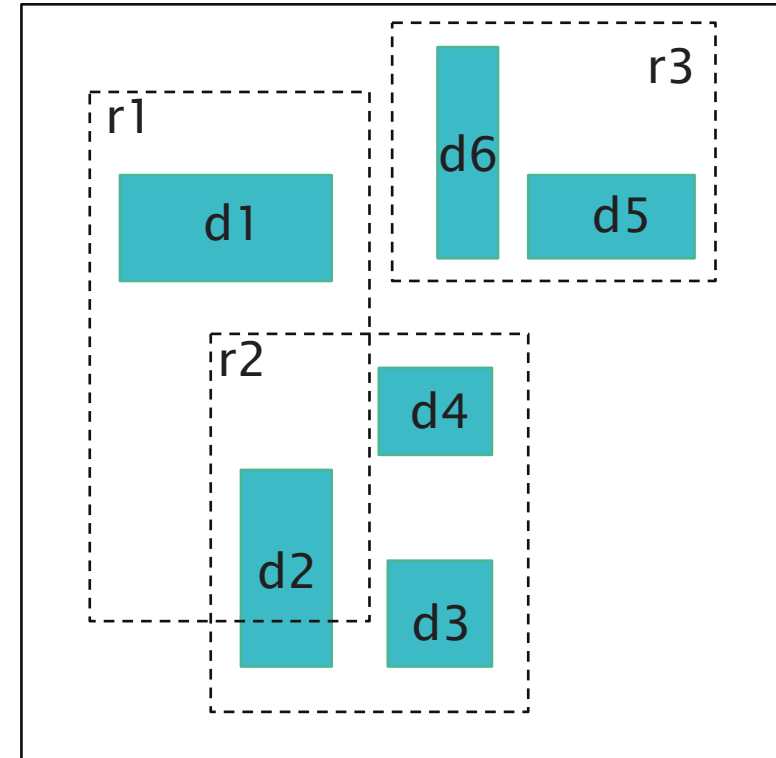
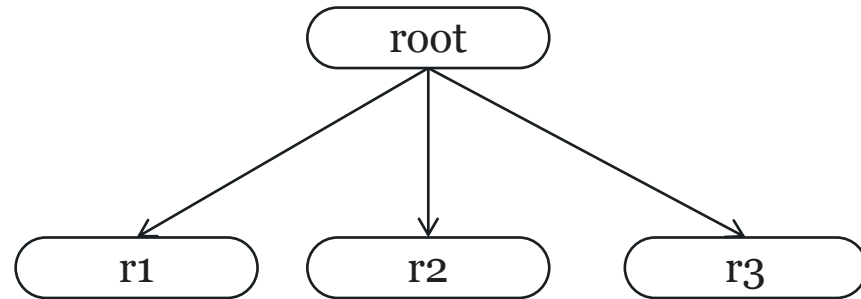


R-Trees

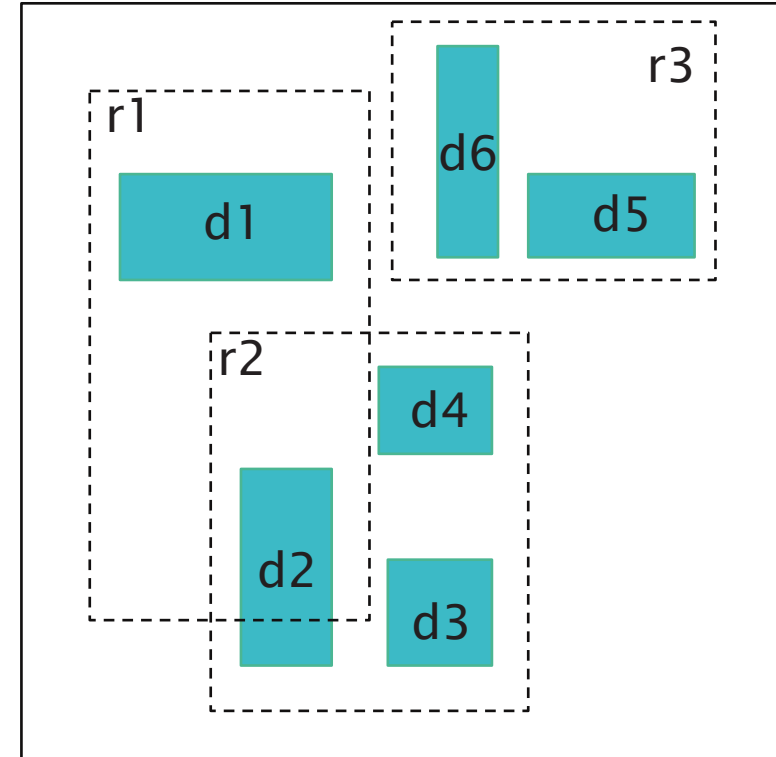
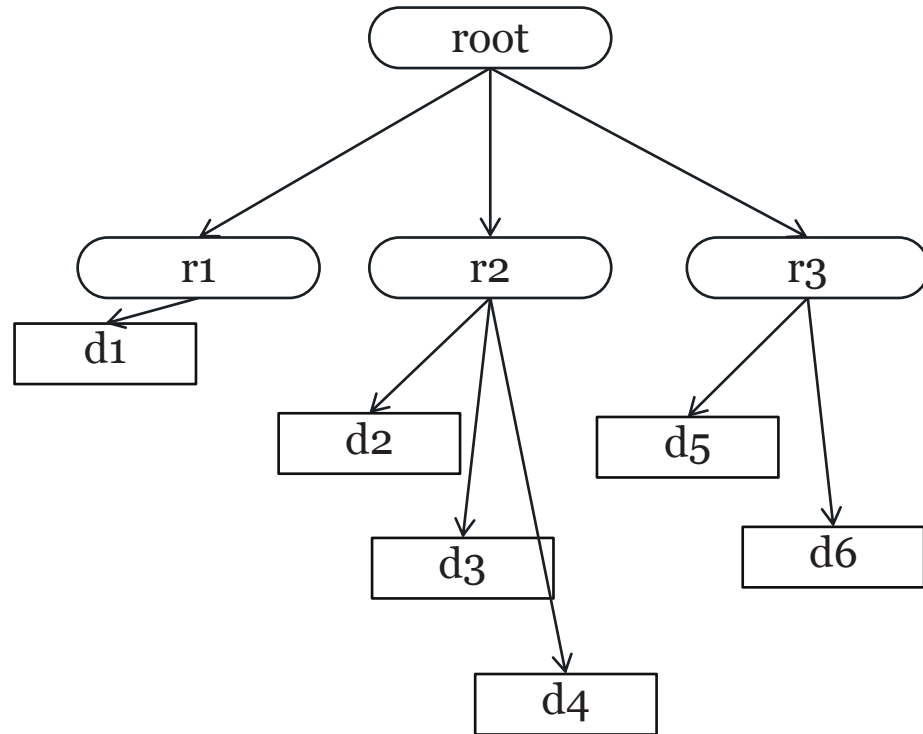
root



R-Trees



R-Trees

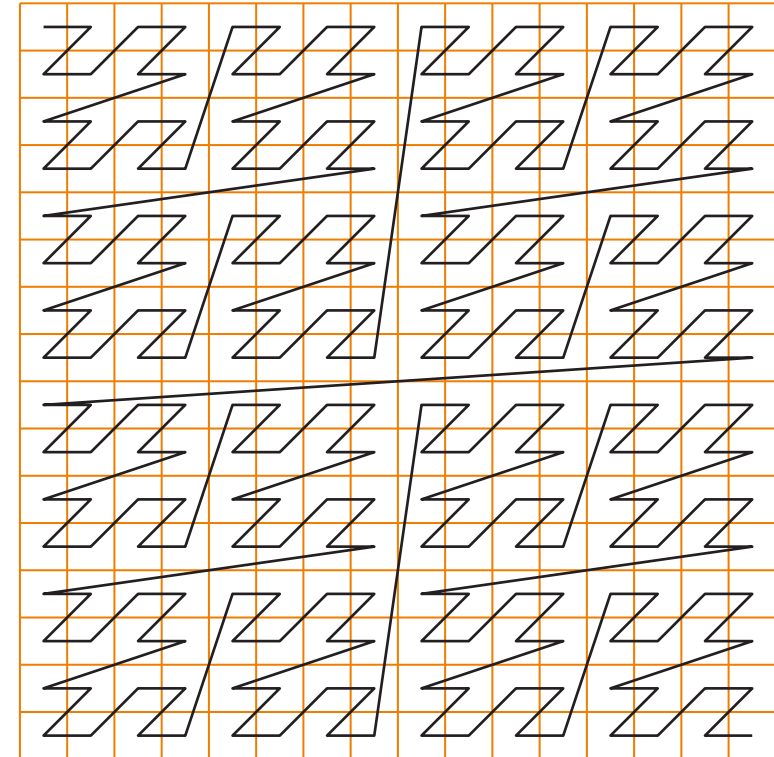


UB-Tree

UB-Tree

Basic approach:

- Map n-dimensional space onto a 1-dimensional line using a fractal space-filling curve
- Partition ranges and index using a B+tree
- When querying, identify regions of n-d space (= segments of 1-d line) that intersect with query rectangle



Z-Index

Map domain of each attribute onto n-bit integer

Order of points on Z-curve given by bit-interleaving the positions on the axes

$$X = X_1 X_2$$

$$Y = Y_1 Y_2$$

$$z\text{-index} = y_1 x_1 y_2 x_2$$

Z-Index

Map domain of each attribute onto n-bit integer

Order of points on Z-curve given by bit-interleaving the positions on the axes

$$X = X_1 X_2$$

$$Y = Y_1 Y_2$$

$$z\text{-index} = y_1 x_1 y_2 x_2$$

	00	01	10	11
00				
01				
10				
11				

Z-Index

Map domain of each attribute onto n-bit integer

Order of points on Z-curve given by bit-interleaving the positions on the axes

$$X = X_1X_2$$

$$Y = Y_1Y_2$$

$$z\text{-index} = y_1x_1y_2x_2$$

	00	01	10	11
00	0000	0001	0100	0101
01	0010	0011	0110	0111
10	1000	1001	1100	1101
11	1010	1011	1110	1111

Z-Index

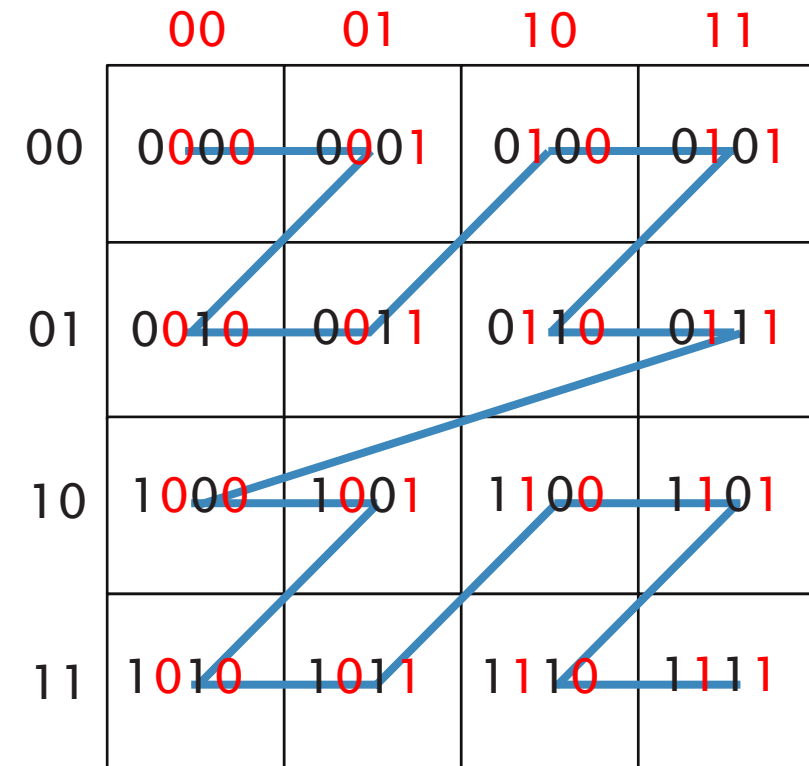
Map domain of each attribute onto n-bit integer

Order of points on Z-curve given by bit-interleaving the positions on the axes

$$X = X_1X_2$$

$$Y = Y_1Y_2$$

$$z\text{-index} = y_1x_1y_2x_2$$



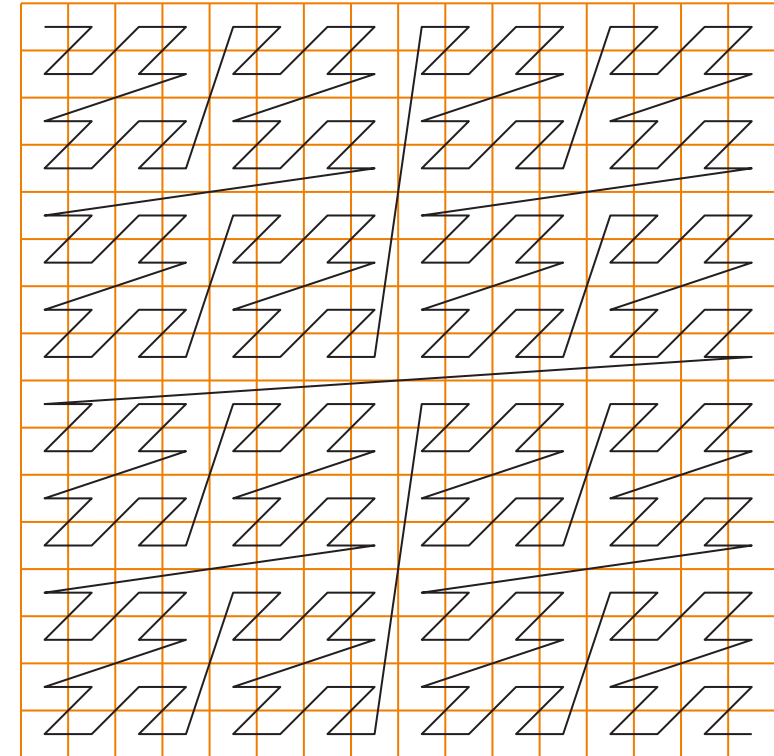
Z-Region Partition

Z-curve partitioned into contiguous ranges (*z-regions*)

- Note that these may not be contiguous regions in the multidimensional space

Z-regions mapped to leaf nodes of a B+tree

- A leaf node contain pointers to records whose attribute value locate them within the associated Z-region



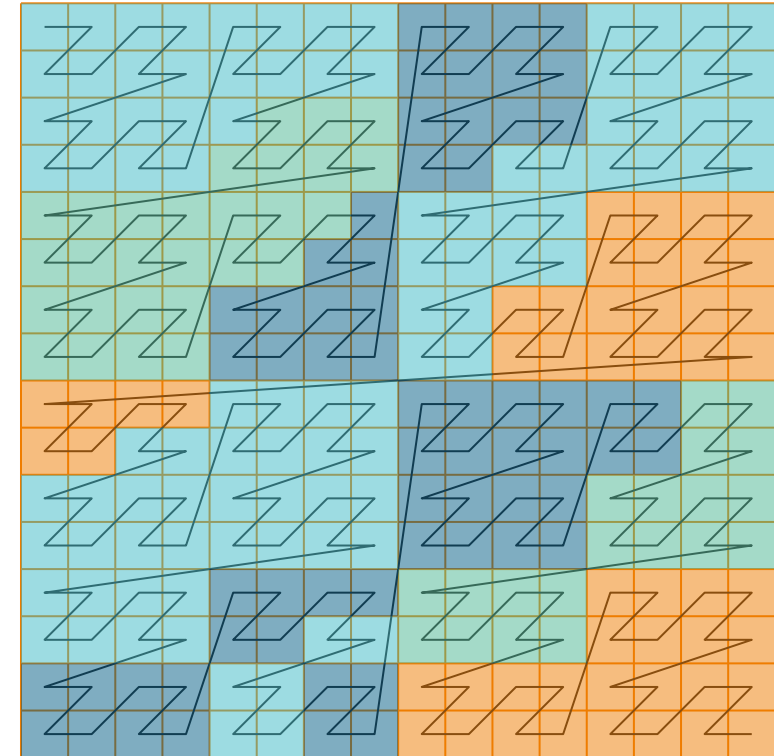
Z-Region Partition

Z-curve partitioned into contiguous ranges (*z-regions*)

- Note that these may not be contiguous regions in the multidimensional space

Z-regions mapped to leaf nodes of a B+tree

- A leaf node contain pointers to records whose attribute value locate them within the associated Z-region



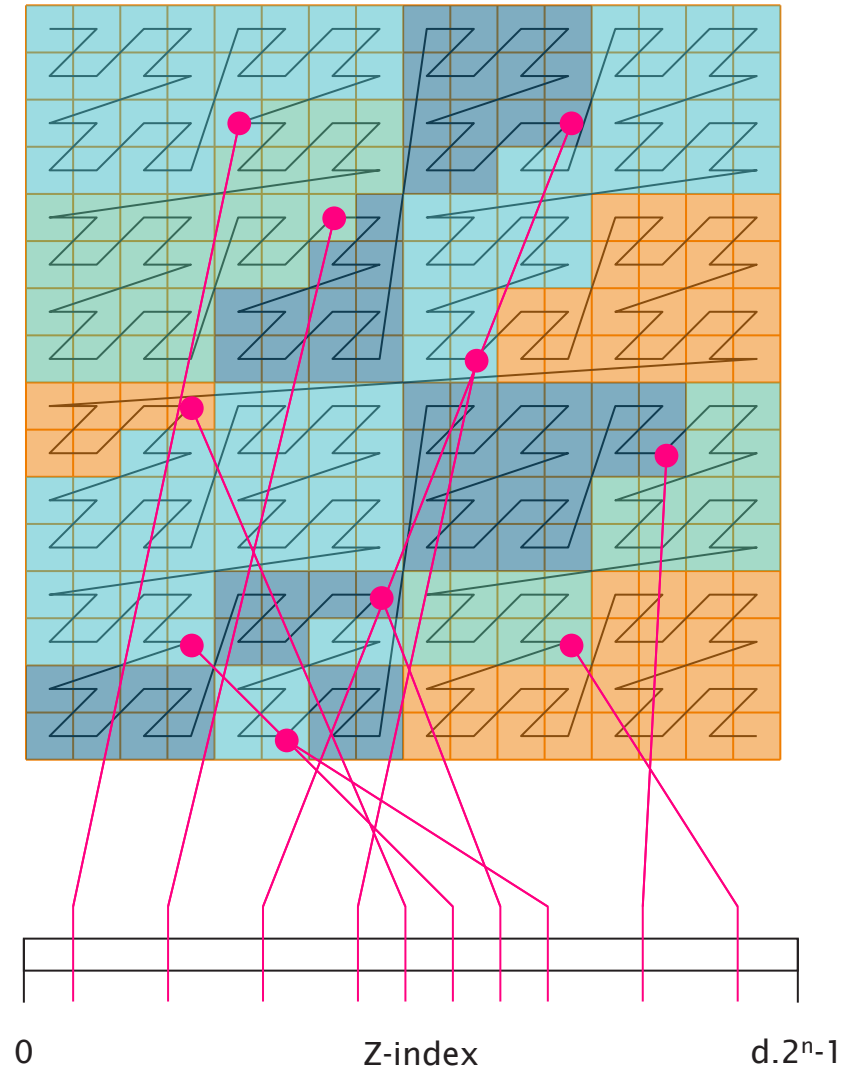
Z-Region Partition

Z-curve partitioned into contiguous ranges (*z-regions*)

- Note that these may not be contiguous regions in the multidimensional space

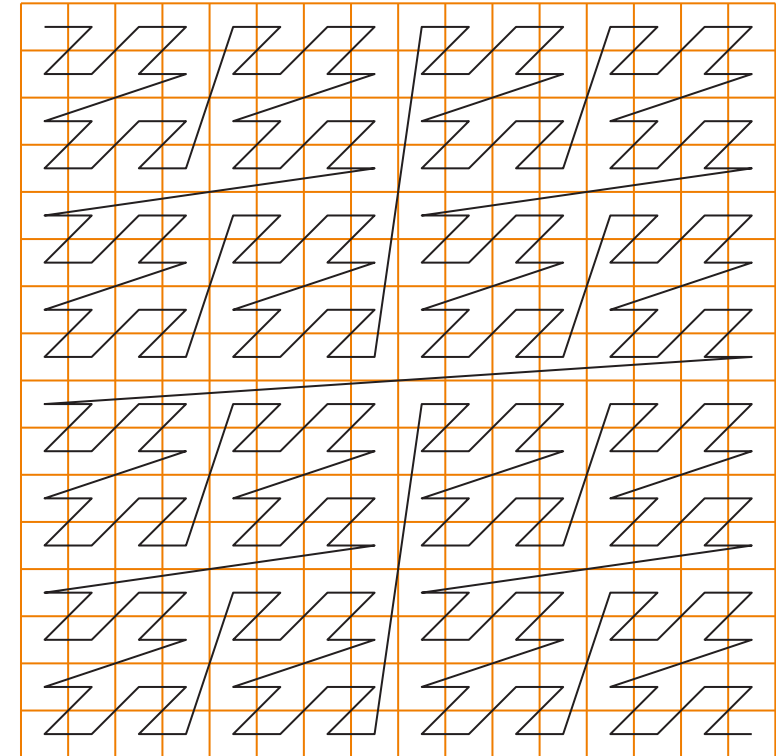
Z-regions mapped to leaf nodes of a B+tree

- A leaf node contain pointers to records whose attribute value locate them within the associated Z-region



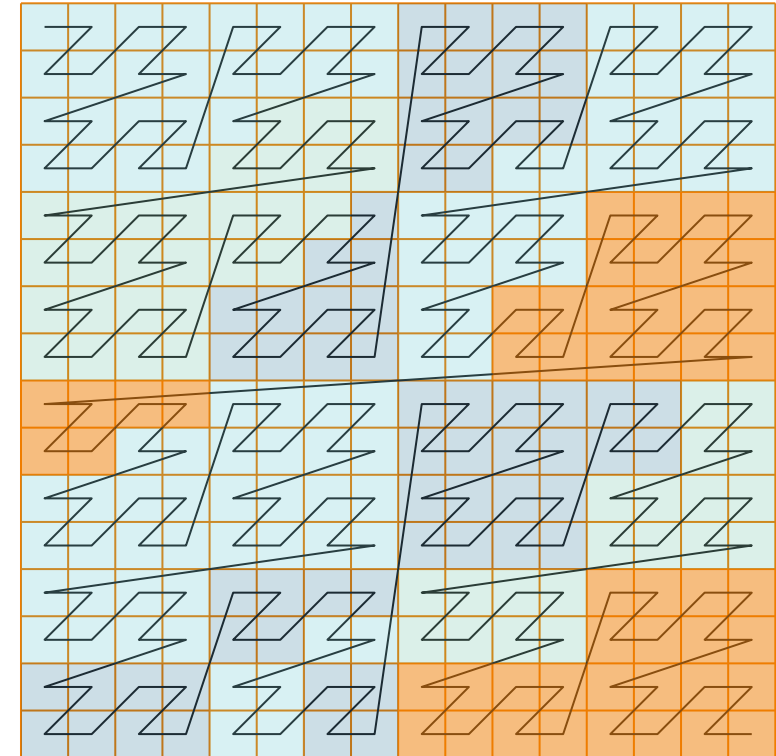
Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



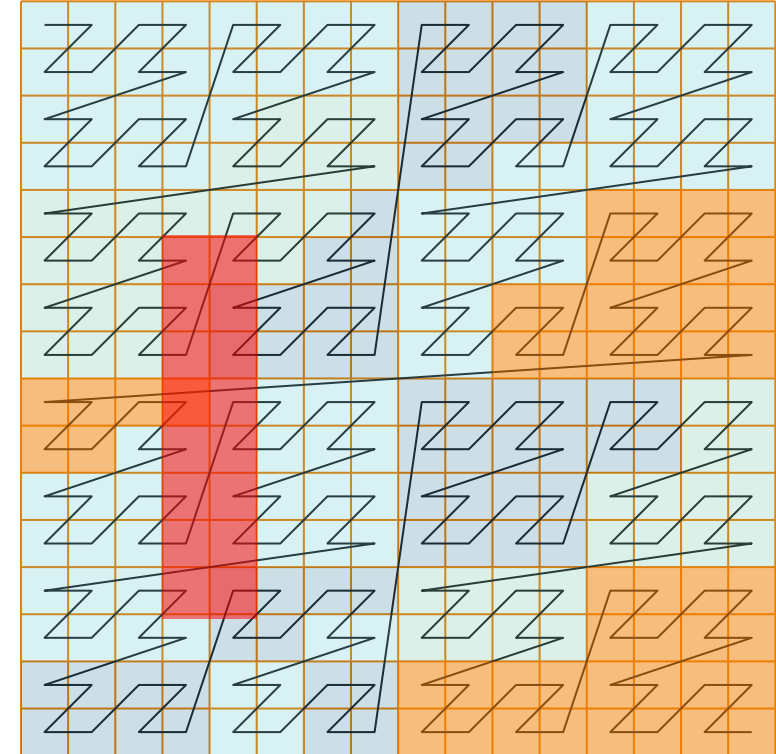
Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



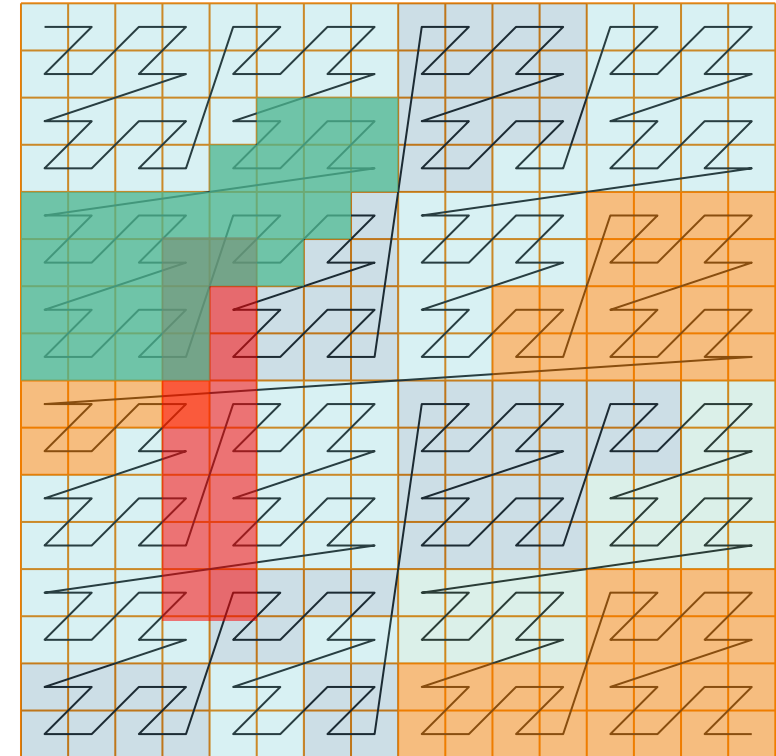
Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



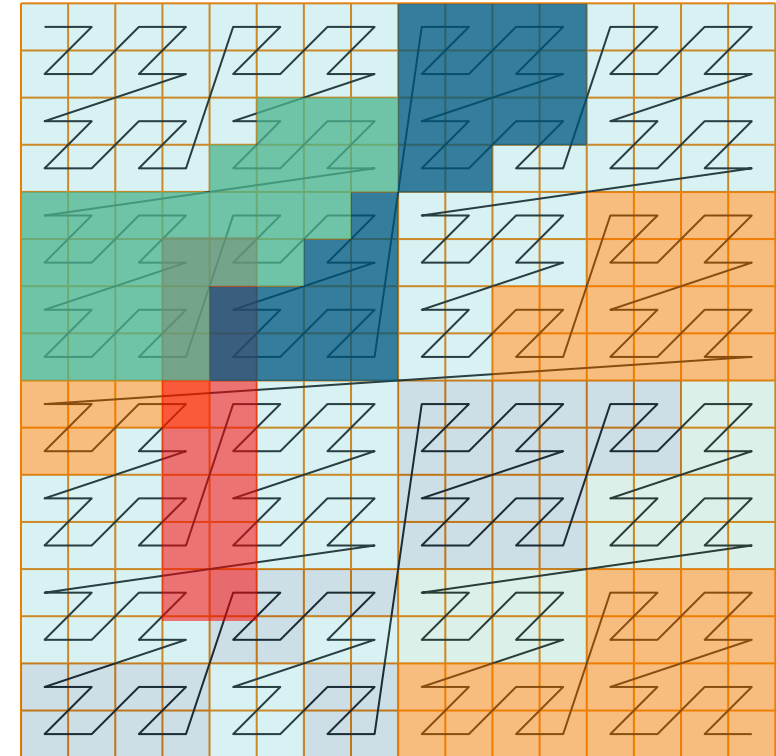
Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



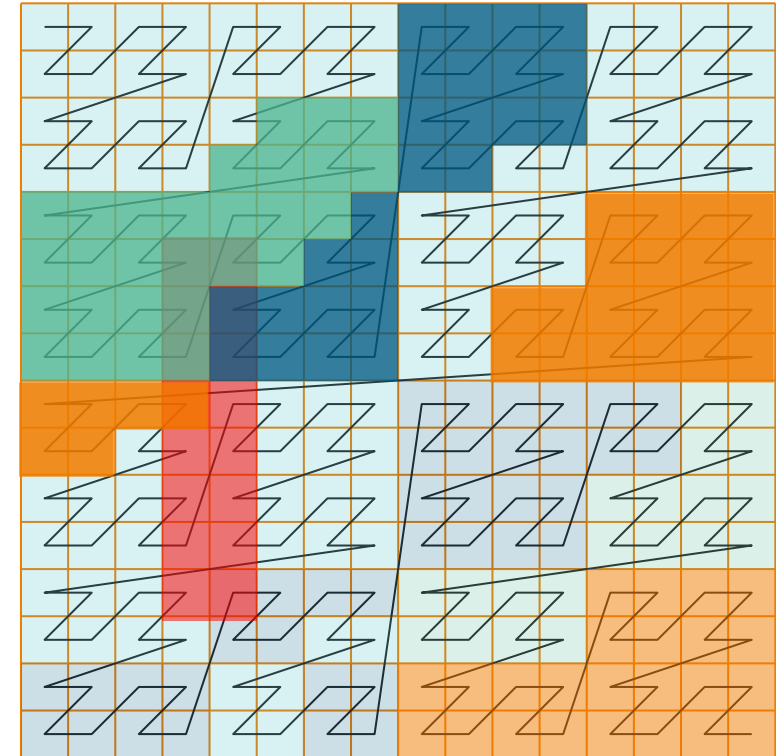
Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



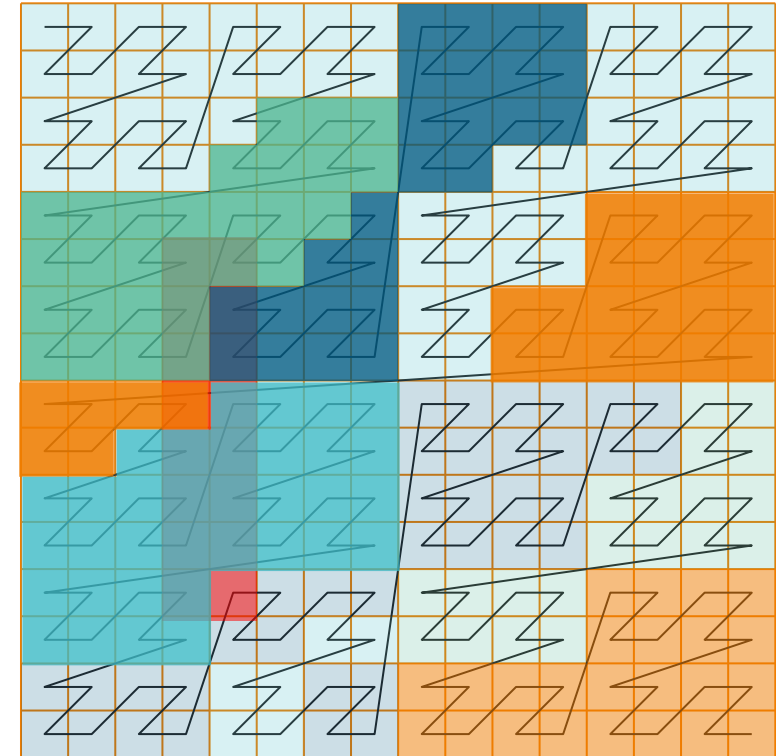
Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



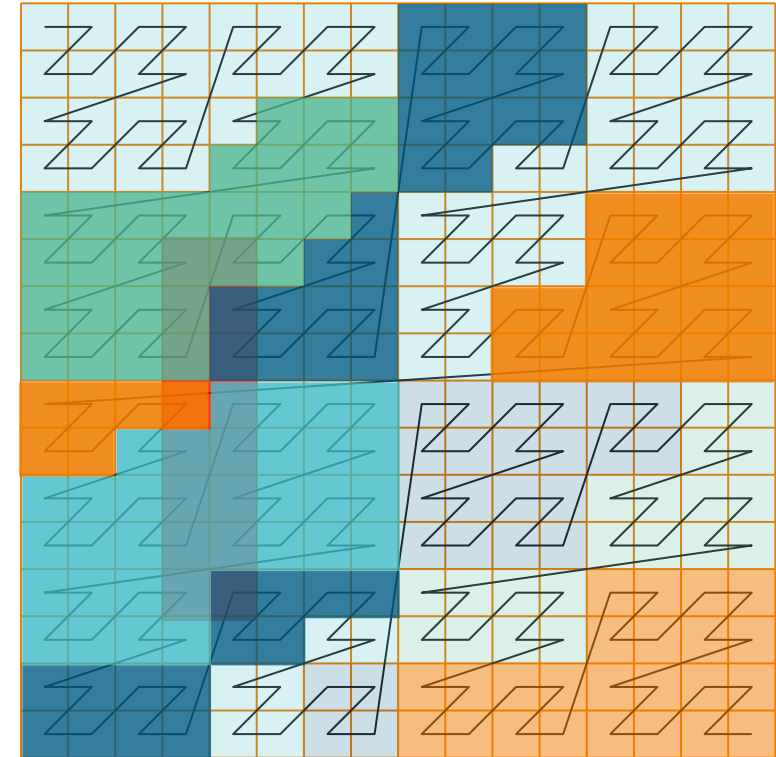
Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



Querying UB-Trees

- Multidimensional range query can be considered as a k-dimensional rectangle
- Algorithm identifies z-regions that intersect with the query rectangle



Bitmap Indexes

Bitmap indexes

Collection of bit-vectors used to index an attribute

- One bit-vector for each unique attribute value
- One bit for each record

Querying index involves combining bit-vectors with bitwise operators (&, |)

- A 1 in the i th position indicates that record i is a match

Example

An online homeware vendor sells products p1...p10

- Products p3 and p5 cost £100
- Product p1 costs £200
- Products p2, p7 and p10 cost £300
- Products p4, p6, p8 and p9 cost £400
- Products p1, p4, p5 and p9 are designed for lounges
- Products p5 and p7 are designed for dining rooms
- Products p3, p5, p6 and p10 are designed for kitchens

Example bitmap index

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
£100	0	0	1	0	1	0	0	0	0	0
£200	1	0	0	0	0	0	0	0	0	0
£300	0	1	0	0	0	0	1	0	0	1
£400	0	0	0	1	0	1	0	1	1	0
Lounge	1	0	0	1	1	0	0	0	1	0
Dining	0	0	0	0	1	0	1	0	0	0
Kitchen	0	0	1	0	1	1	0	0	0	1

Example bitmap index

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
£100	0	0	1	0	1	0	0	0	0	0
£200	1	0	0	0	0	0	0	0	0	0
£300	0	1	0	0	0	0	1	0	0	1
£400	0	0	0	1	0	1	0	1	1	0
Lounge	1	0	0	1	1	0	0	0	1	0
Dining	0	0	0	0	1	0	1	0	0	0
Kitchen	0	0	1	0	1	1	0	0	0	1

price=£300 \wedge room=kitchen

Example bitmap index

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
£100	0	0	1	0	1	0	0	0	0	0
£200	1	0	0	0	0	0	0	0	0	0
£300	0	1	0	0	0	0	1	0	0	1
£400	0	0	0	1	0	1	0	1	1	0
Lounge	1	0	0	1	1	0	0	0	1	0
Dining	0	0	0	0	1	0	1	0	0	0
Kitchen	0	0	1	0	1	1	0	0	0	1

price=£300 \wedge room=kitchen

0100001001 & 0010110001 = 0000000001

Example bitmap index

	p1	p2	p3	p4	p5	p6	p7	p8	p9	p10
£100	0	0	1	0	1	0	0	0	0	0
£200	1	0	0	0	0	0	0	0	0	0
£300	0	1	0	0	0	0	1	0	0	1
£400	0	0	0	1	0	1	0	1	1	0
Lounge	1	0	0	1	1	0	0	0	1	0
Dining	0	0	0	0	1	0	1	0	0	0
Kitchen	0	0	1	0	1	1	0	0	0	1

price=£300 \wedge room=kitchen

0100001001 & 0010110001 = 0000000001

p10 is matching product

Compression

- Bit-vectors are typically sparse, with few 1 bits
 - Large amount of wasted space
 - Run-length encoding of bit-vectors to reduce stored size
- Bitwise operators must be applied to original bit-vectors
 - Can decode RLE bit-vectors one run at a time

Bitmap indexes

Pro

- Efficient answering of partial-match queries

Con

- Requires fixed record numbers
- Changes to data file require changes to bitmap index

Further Reading

Further Reading

- Chapter 14 of Garcia-Molina et al
 - Sections 14.1-14.3

- Next lecture: Multi-key Indexing
 - Sections 14.4-14.7

Next Lecture: Relational Algebra