
University of
Southampton

University of Southampton

Access Structures

COMP3211 Advanced Databases

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

University of Southampton

Overview

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

University of Southampton

Index Basics

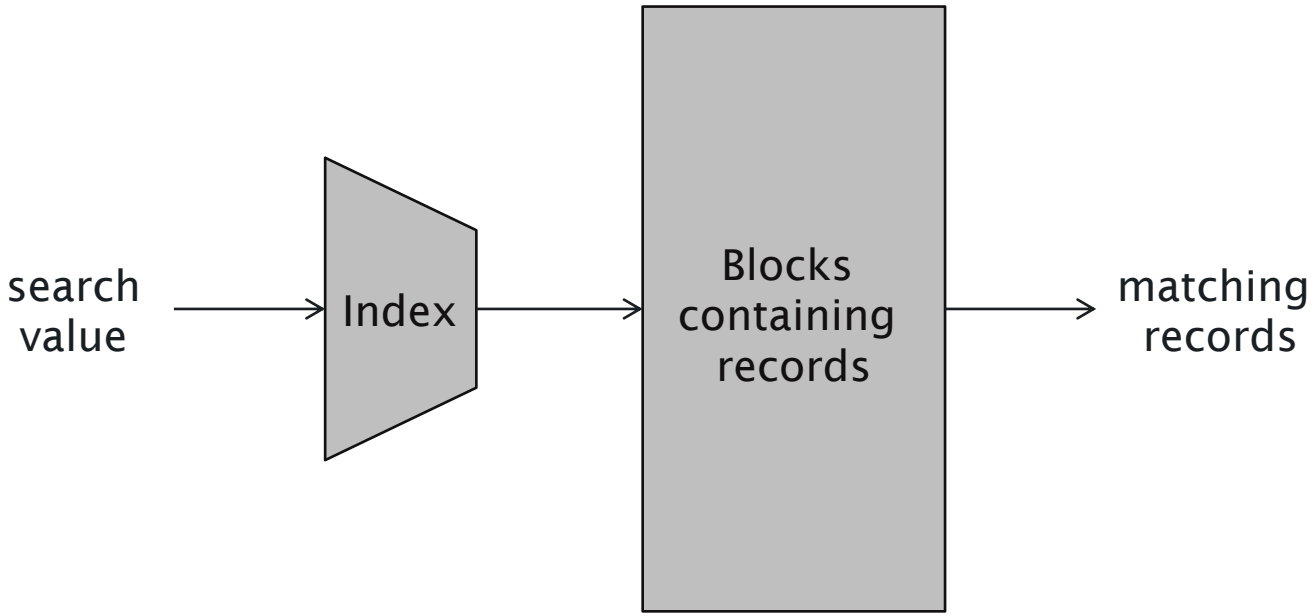
University of Southampton

Index basics

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

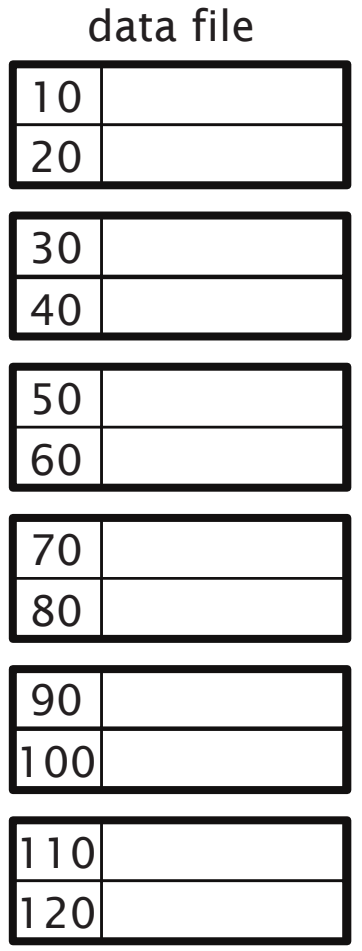
- How do we find the tuples that match some criteria?

Indexes


search value
Index
Blocks containing records
matching records

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



University of Southampton

To Index or Not To Index?

Maintaining an index costs time (processor, disk access)

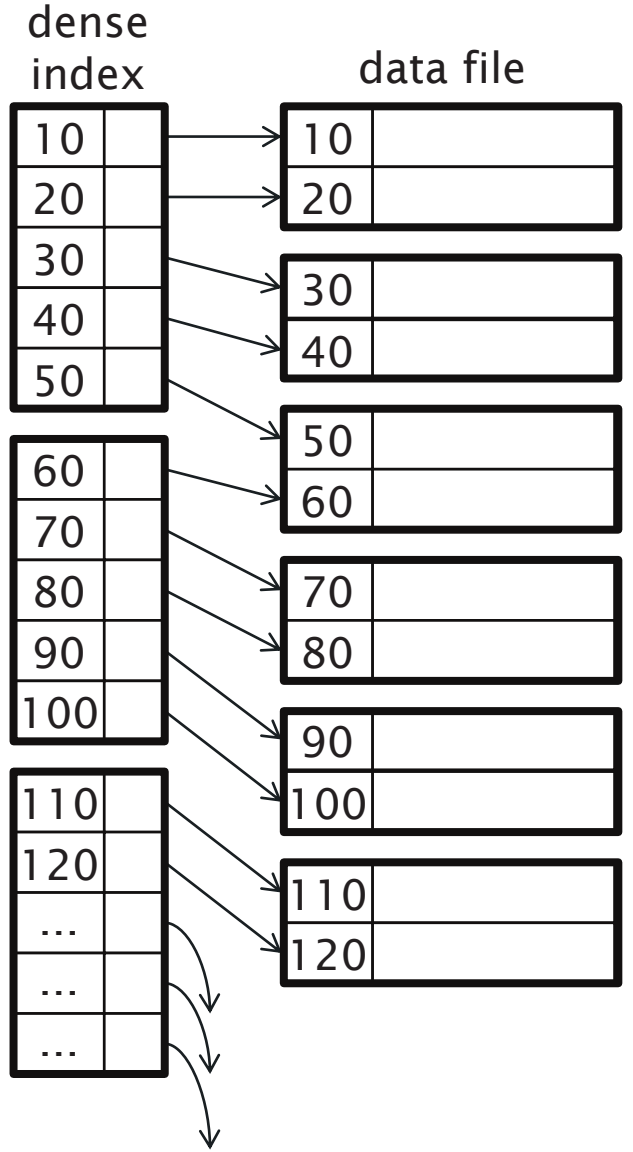
- When entries are added to the relation, index must be updated
- Index must be maintained to make good use of resources

There is a trade off between:

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

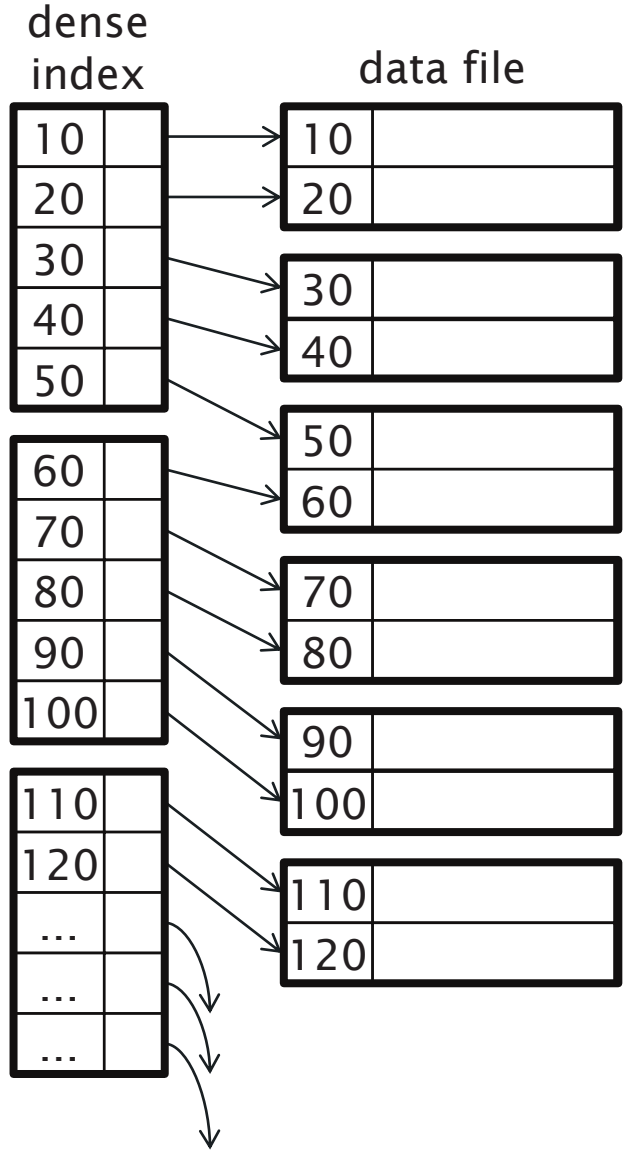
Dense Index

- 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



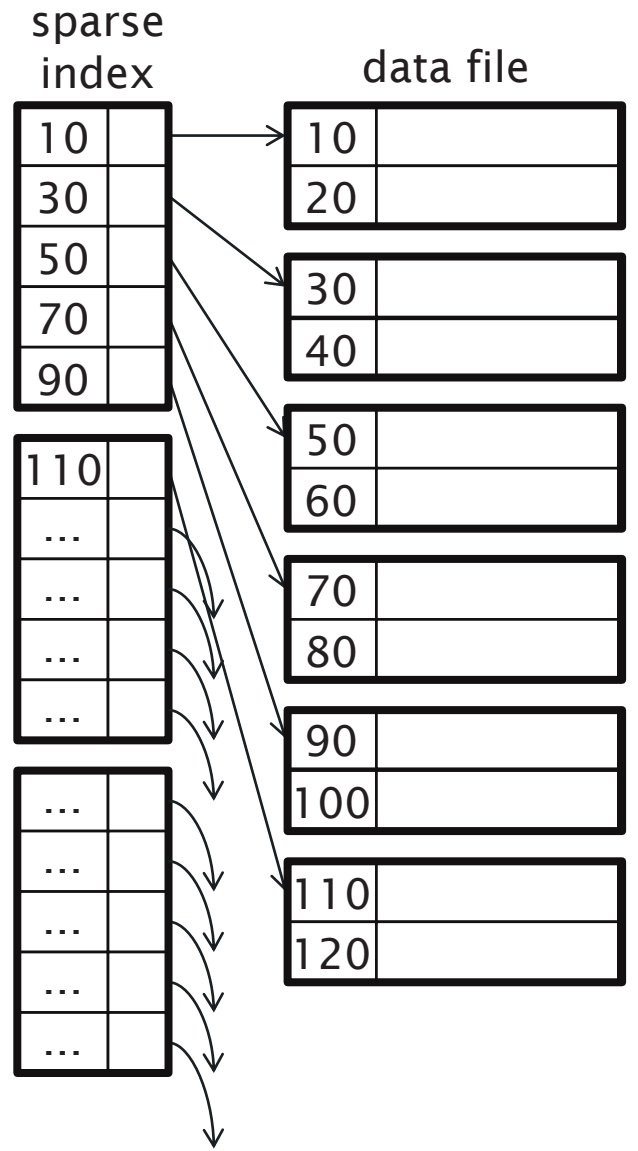
Dense Index

- 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)



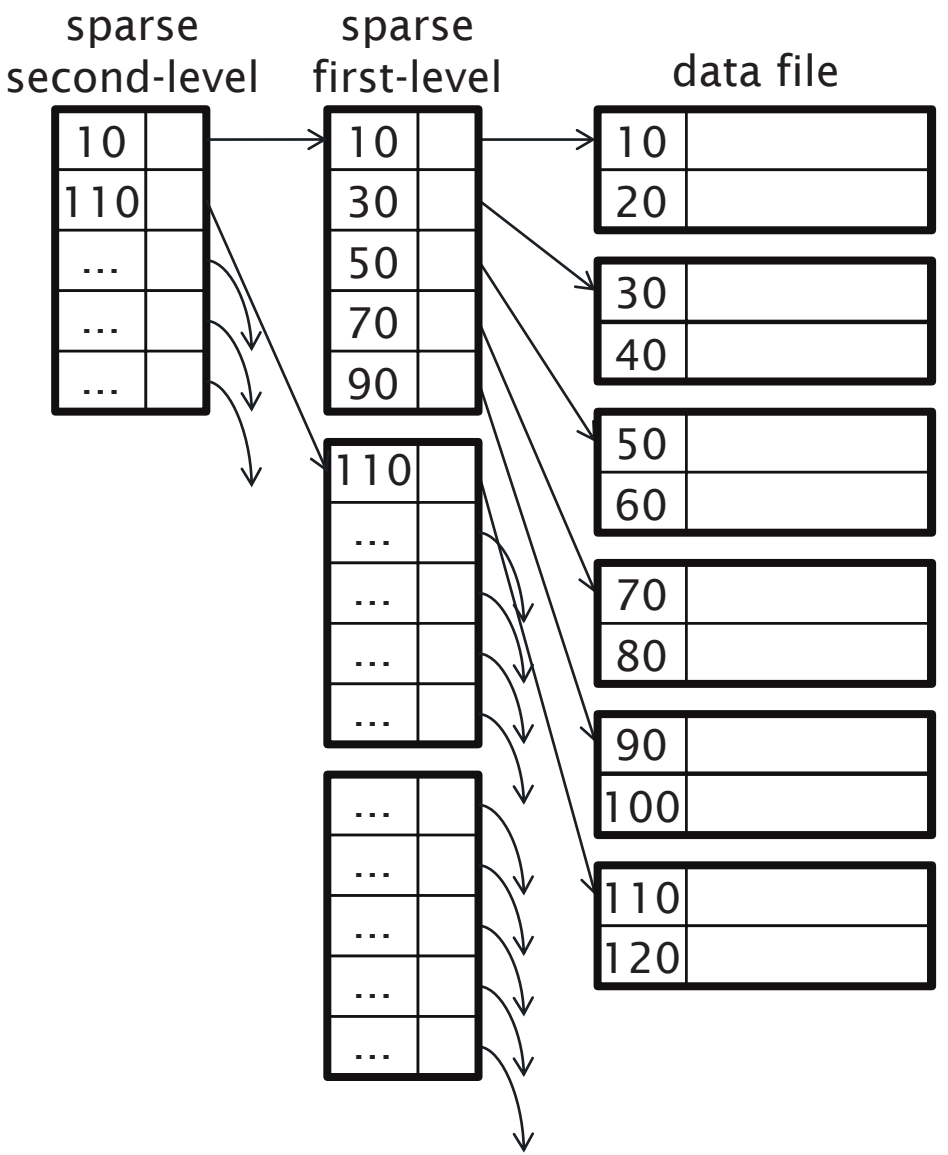
Sparse Index

- 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
- Potentially takes longer to find key than dense index (



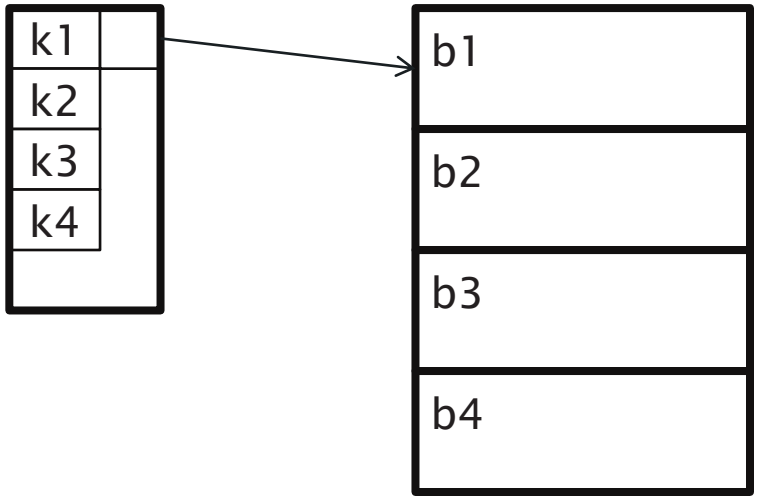
Multi-level Index

- 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 = 2kB



University of Southampton

Sparse vs. Dense Tradeoff

Sparse:

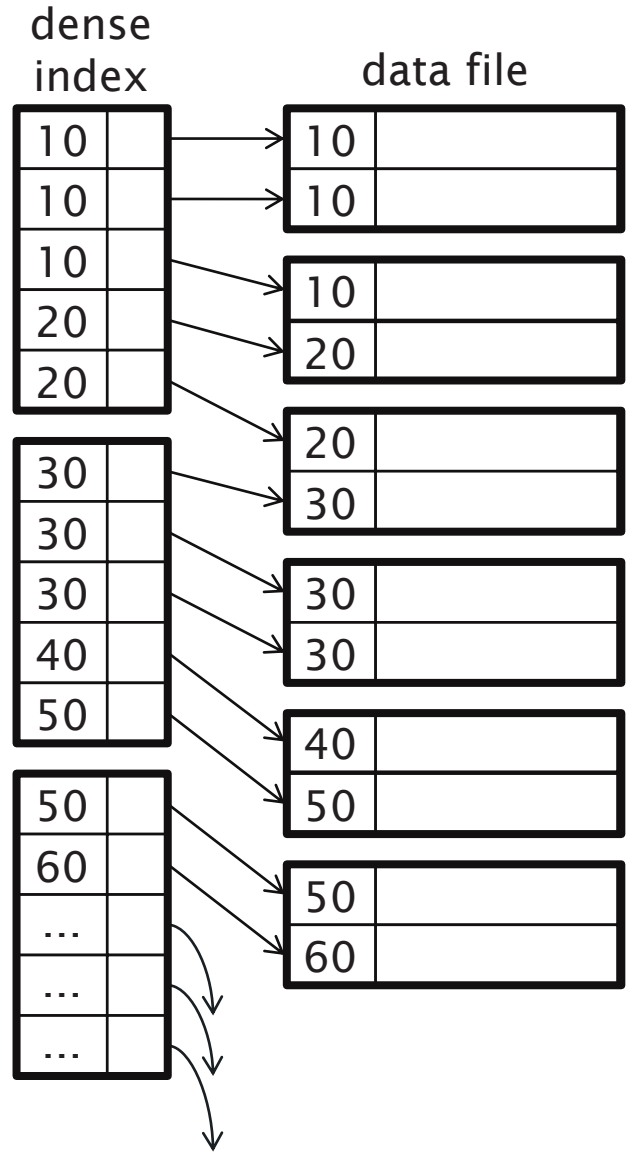
- Less index space per record can keep more of index in memory
- Better for insertions

Dense:

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

Duplicate Keys

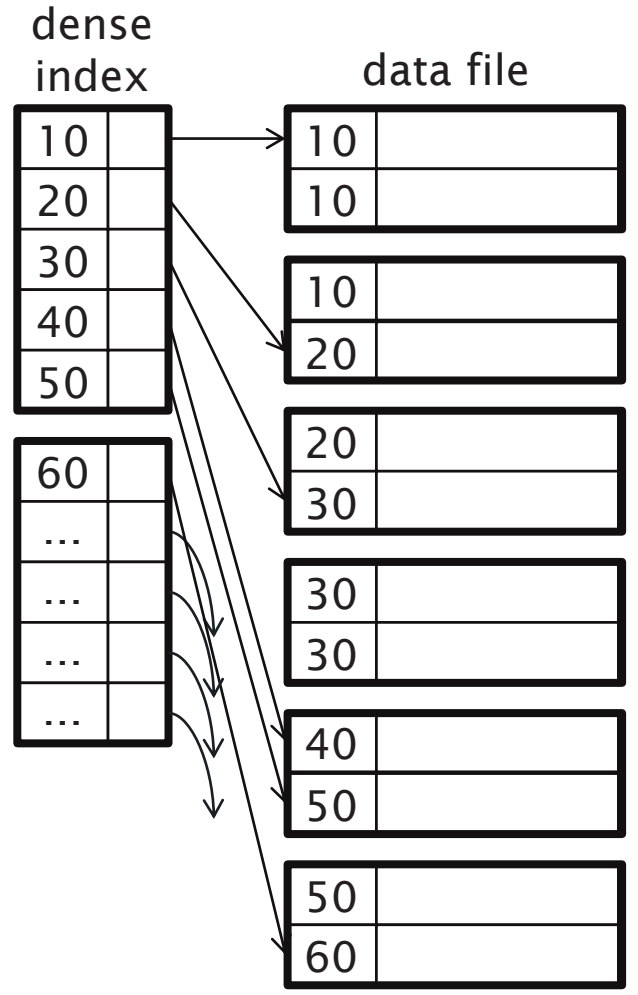
Dense index approach #1



Duplicate Keys

Dense index approach #2

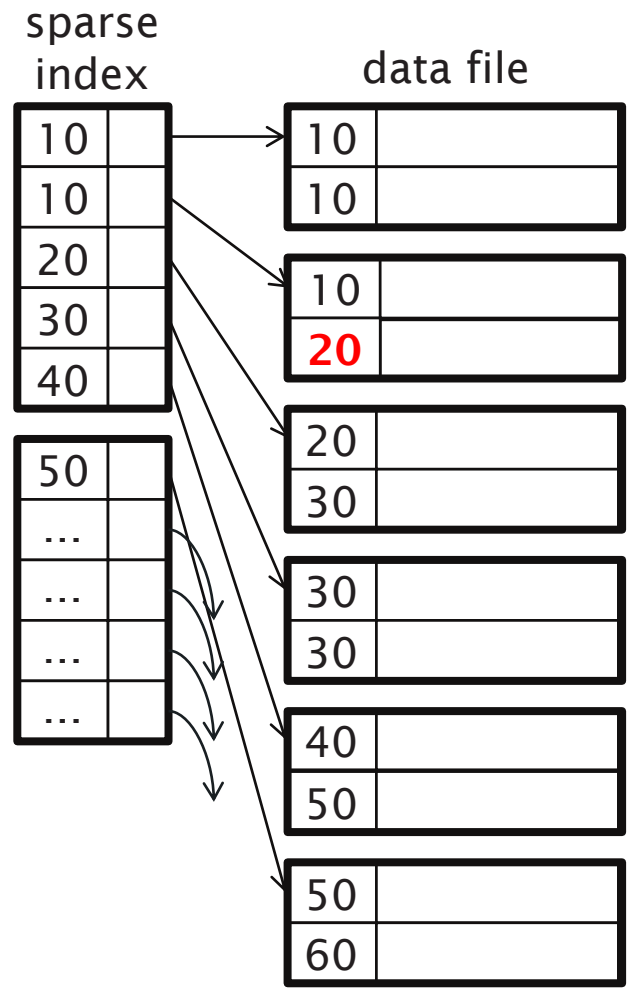
- Point at the first record with a given value
- better approach?
  (smaller index)



Duplicate Keys

Sparse index approach #1

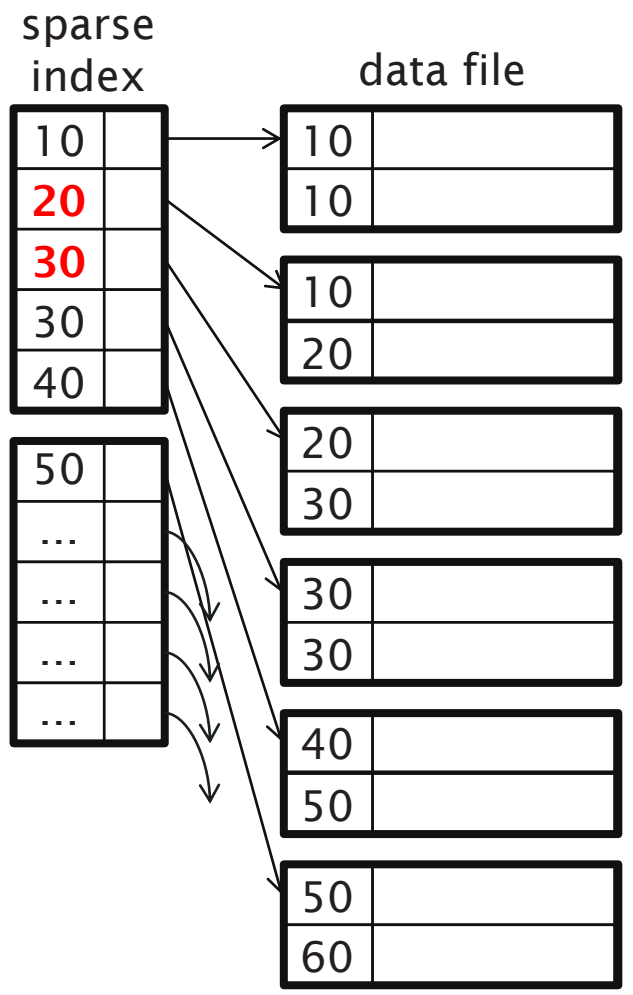
- Searching for (e.g.) 20 will give unexpected results



Duplicate Keys

Sparse index approach #2

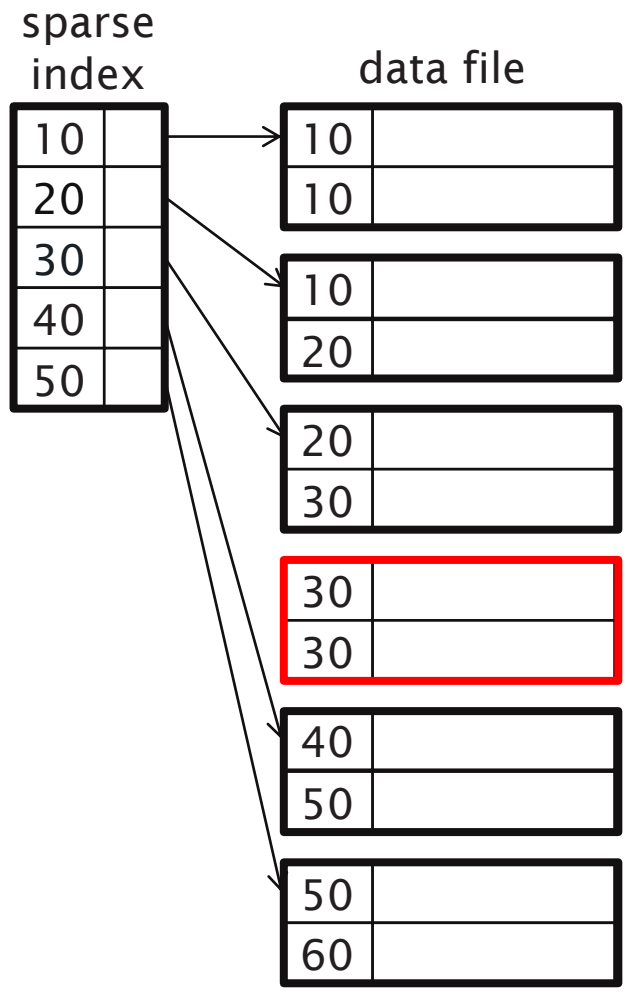
- Index contains first *new* key from each block



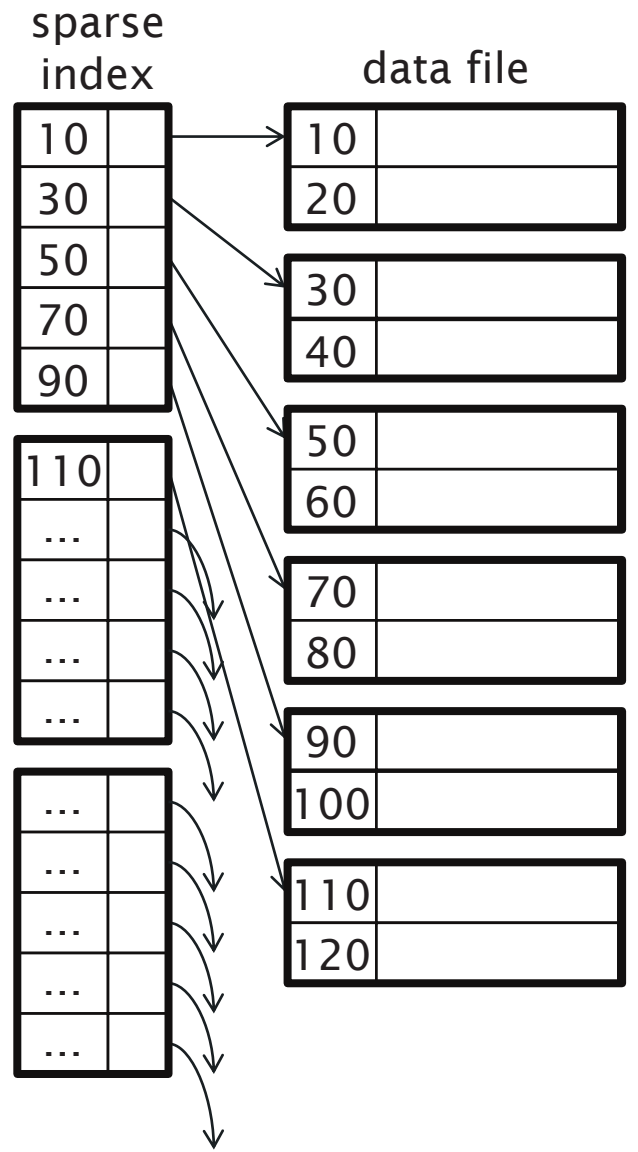
Duplicate Keys

Sparse index approach #2

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

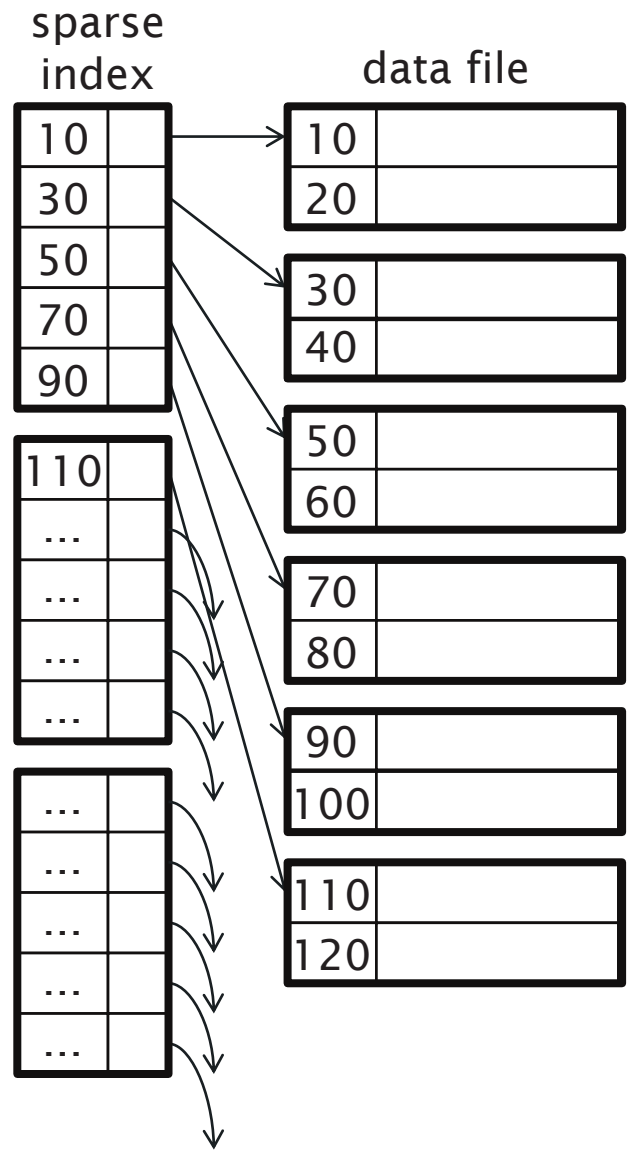


Deletion from Sparse Index



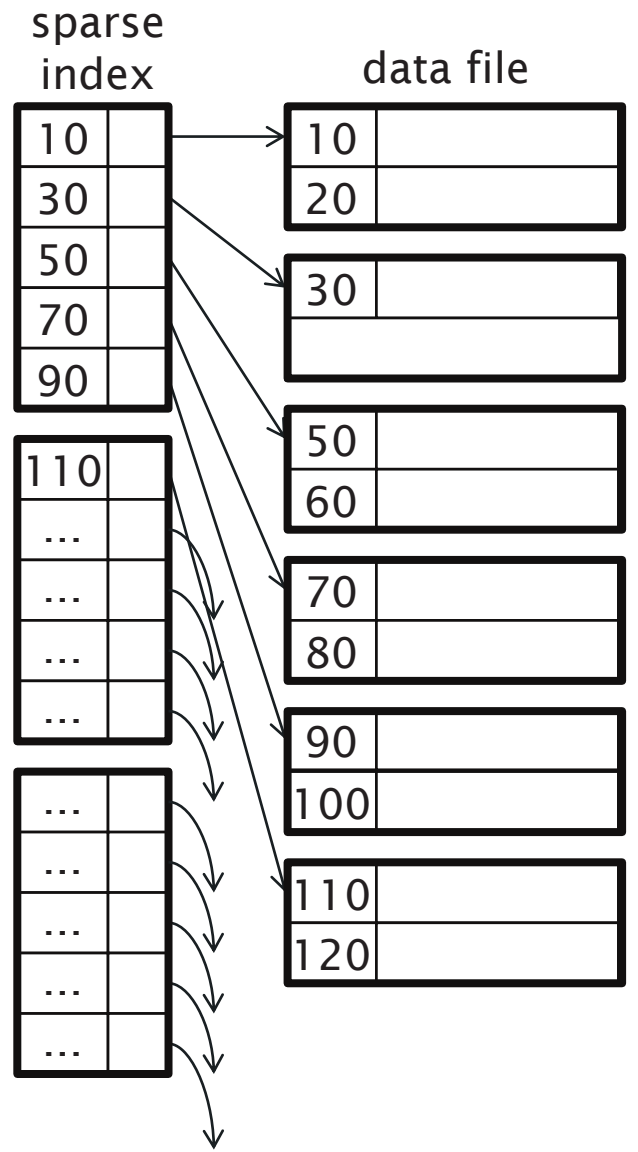
Deletion from Sparse Index

- Delete record 40



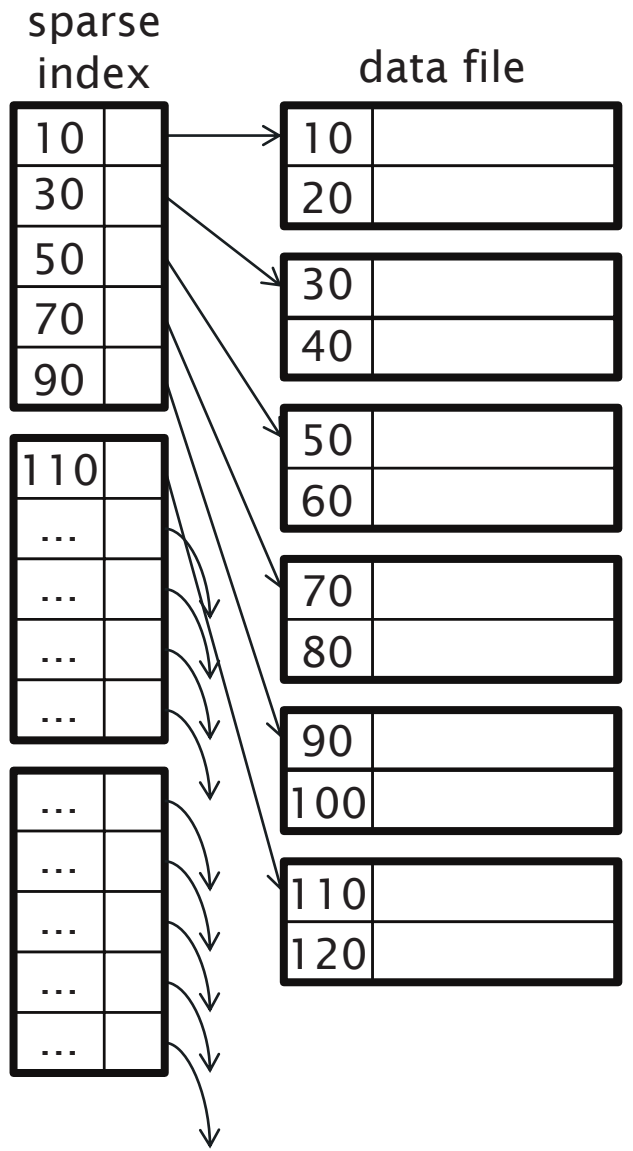
Deletion from Sparse Index

- Delete record 40



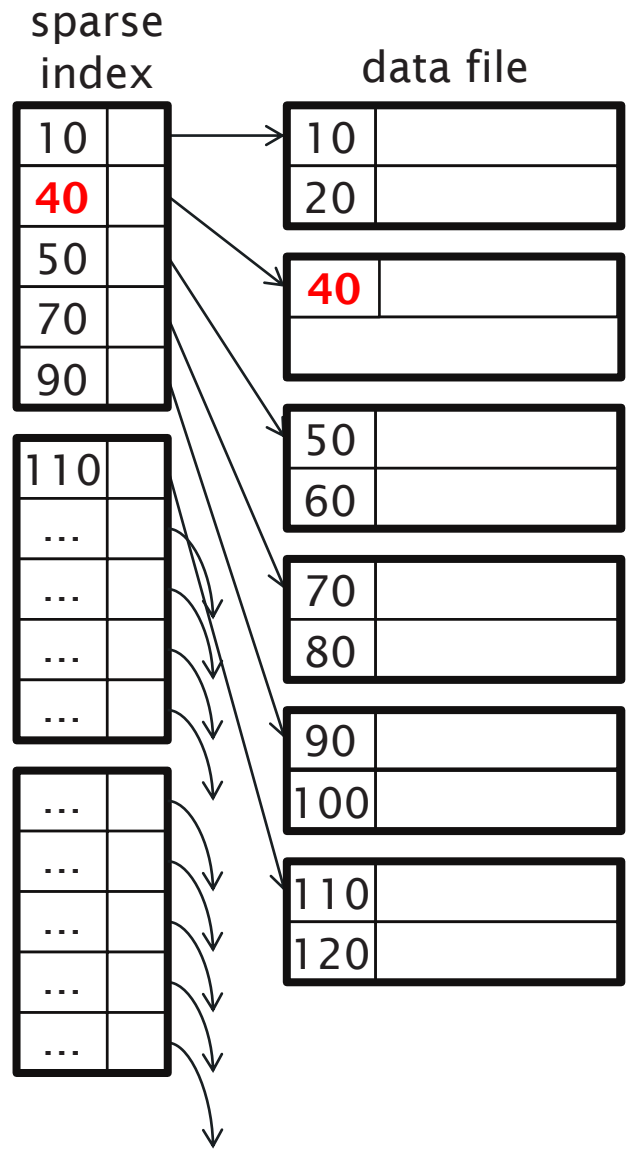
Deletion from Sparse Index

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



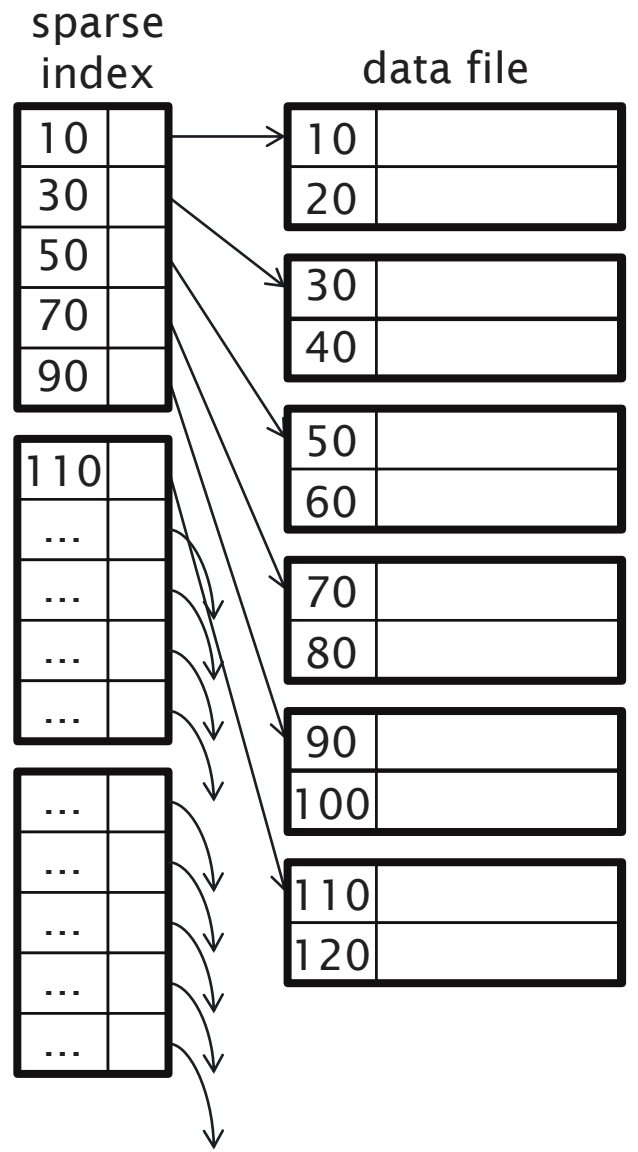
Deletion from Sparse Index

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



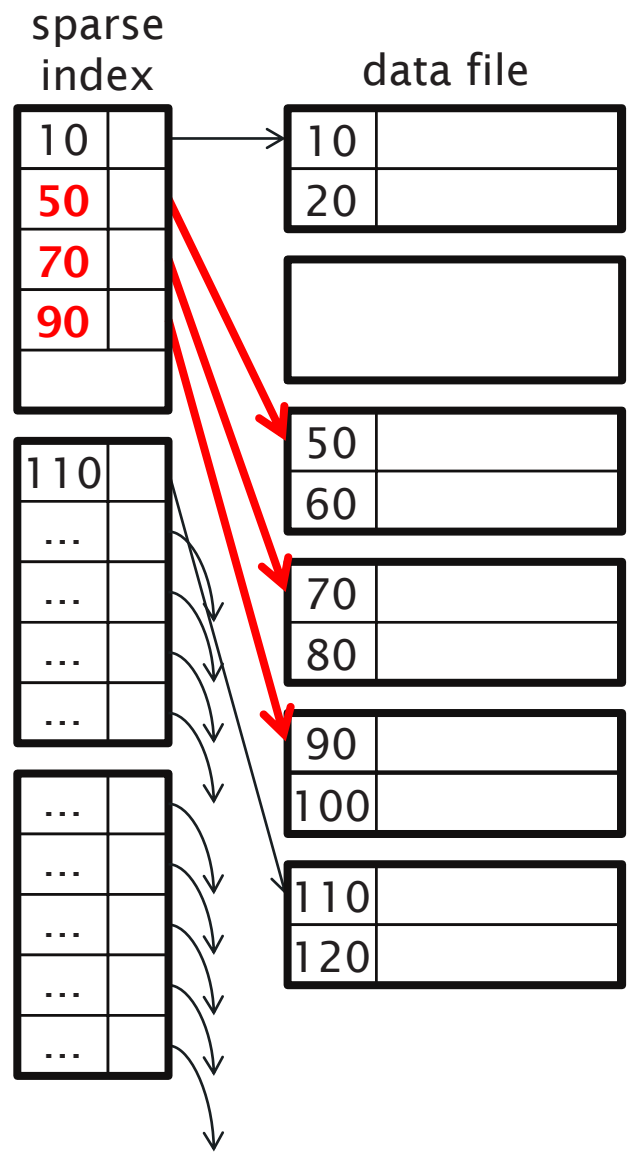
Deletion from Sparse Index

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



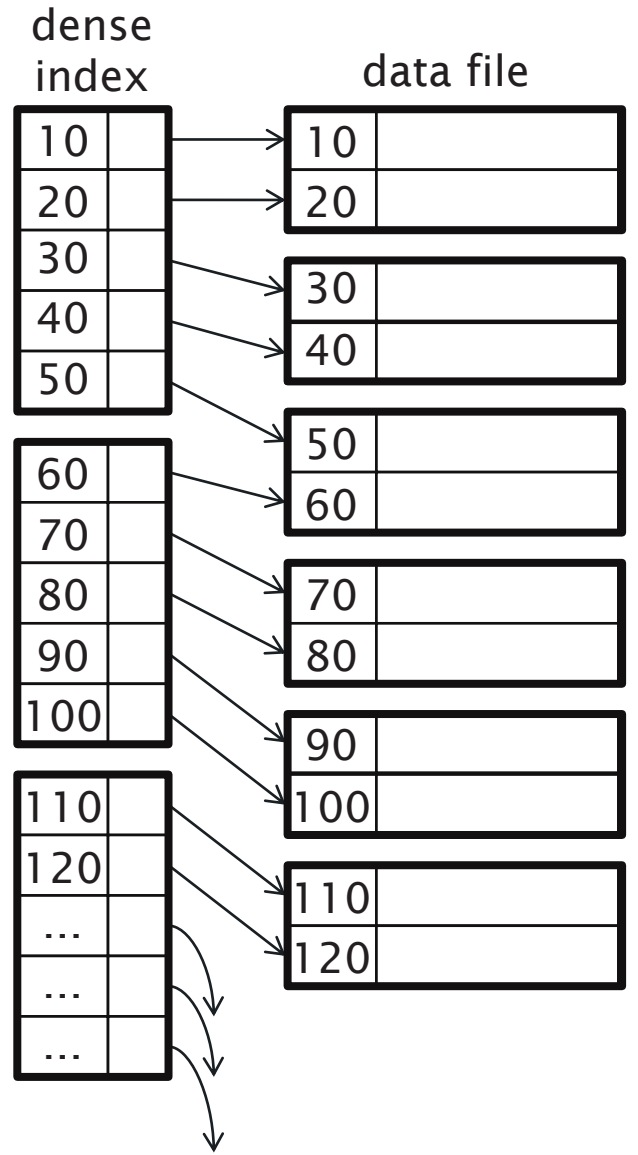
Deletion from Sparse Index

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



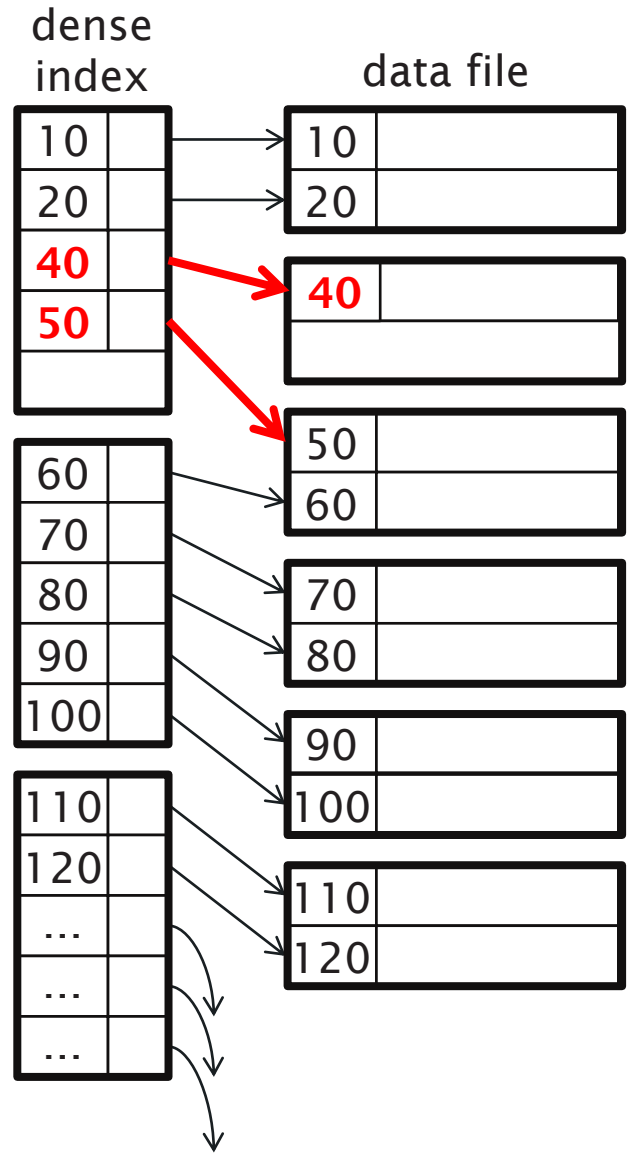
Deletion from Dense Index

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

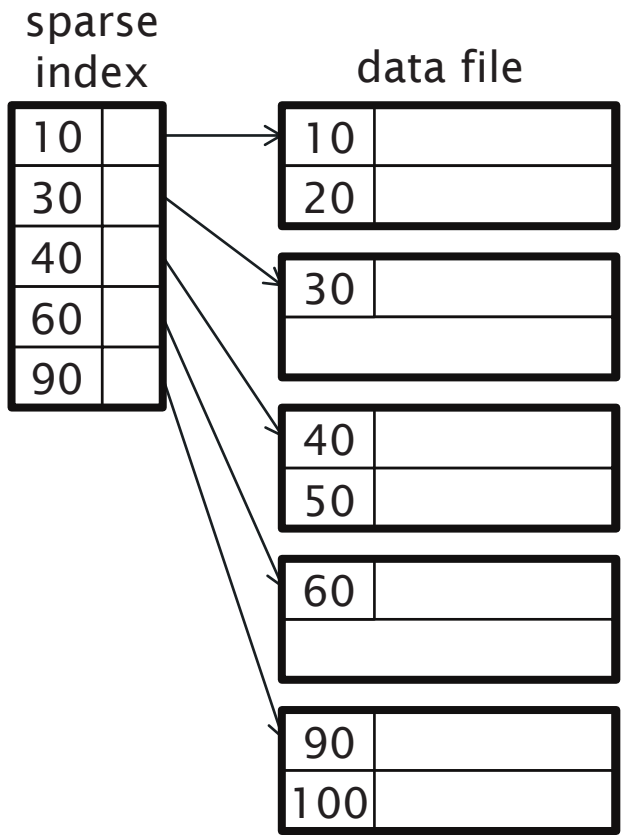


Deletion from Dense Index

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

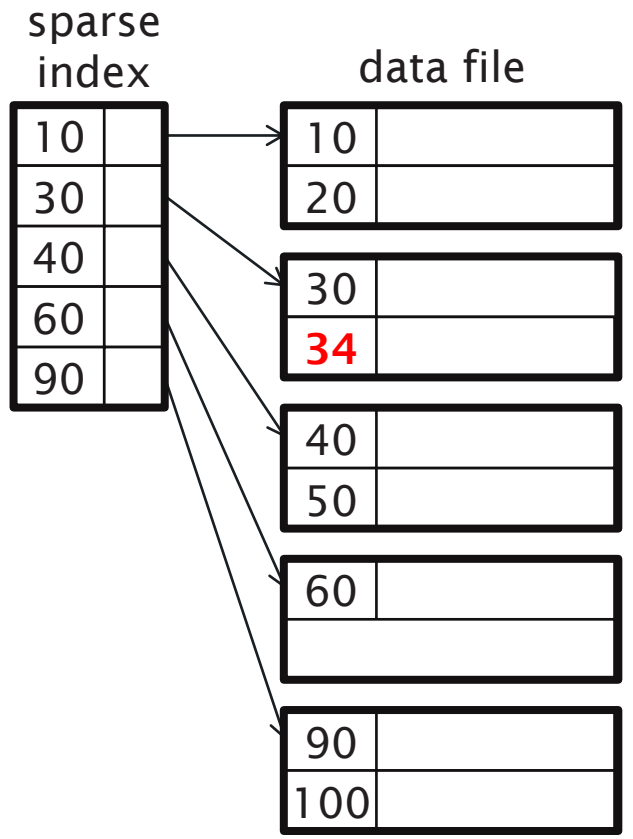


Insertion into Sparse Index



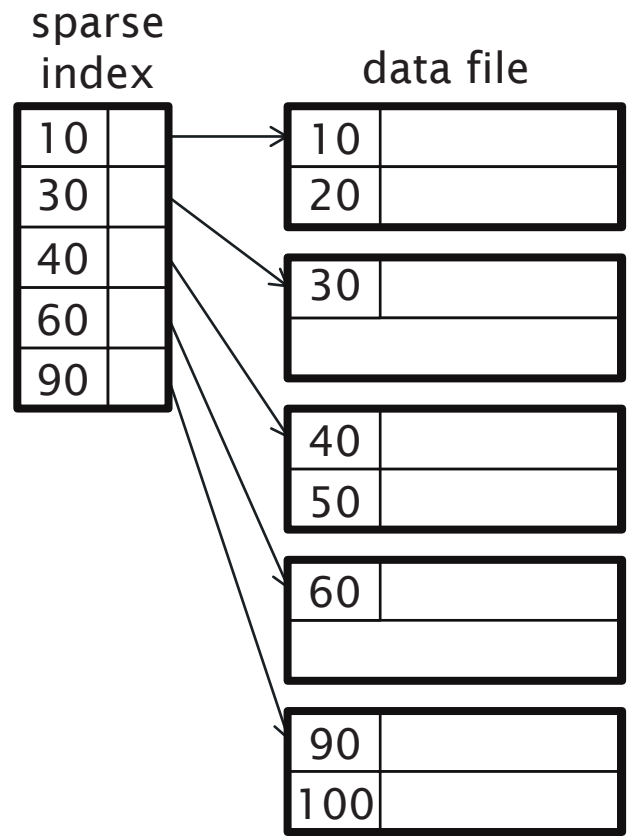
Insertion into Sparse Index

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



Insertion into Sparse Index

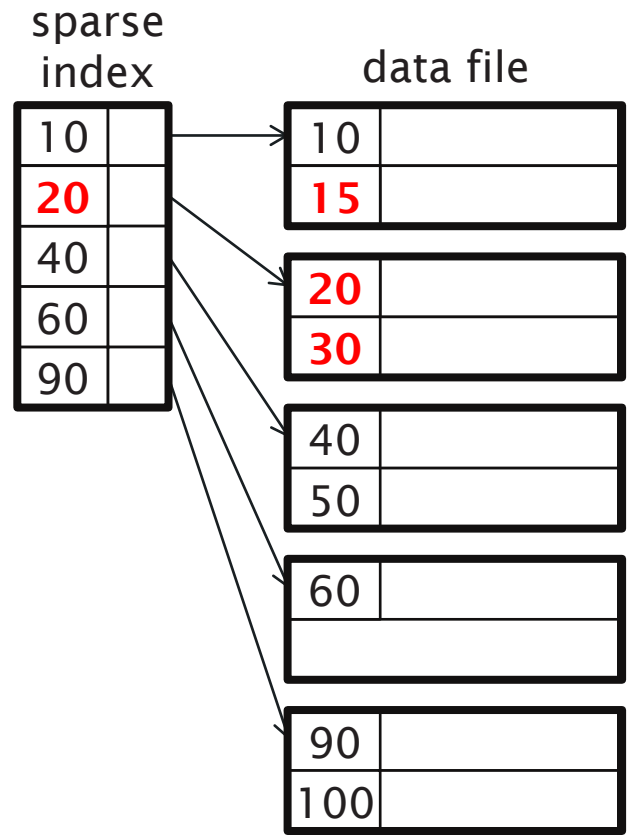
- Insert record 15
  - Add to data file and immediately reorganise
  - Update index



Insertion into Sparse Index

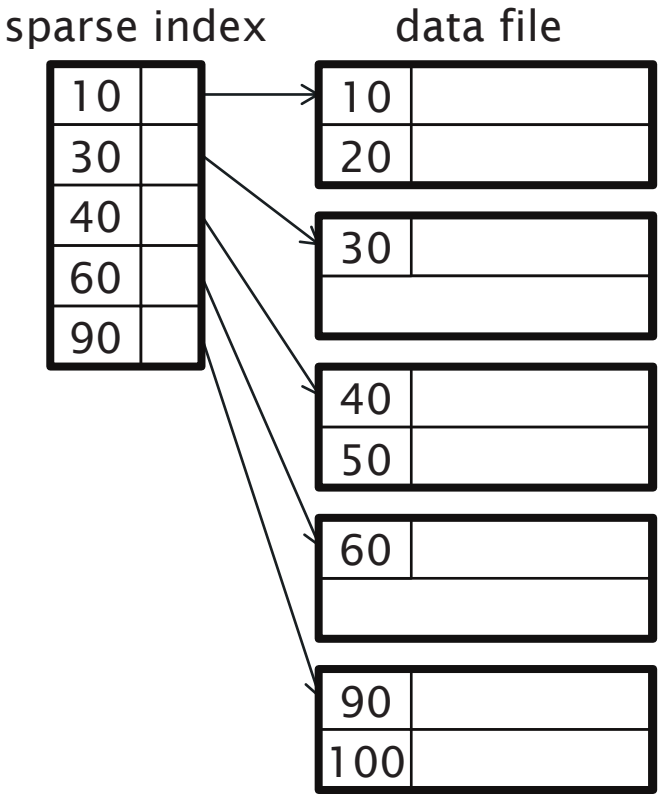
- Insert record 15
  - Add to data file and immediately reorganise
  - Update index

- Alternatively:
  - Insert new block (chained file)
  - Update index



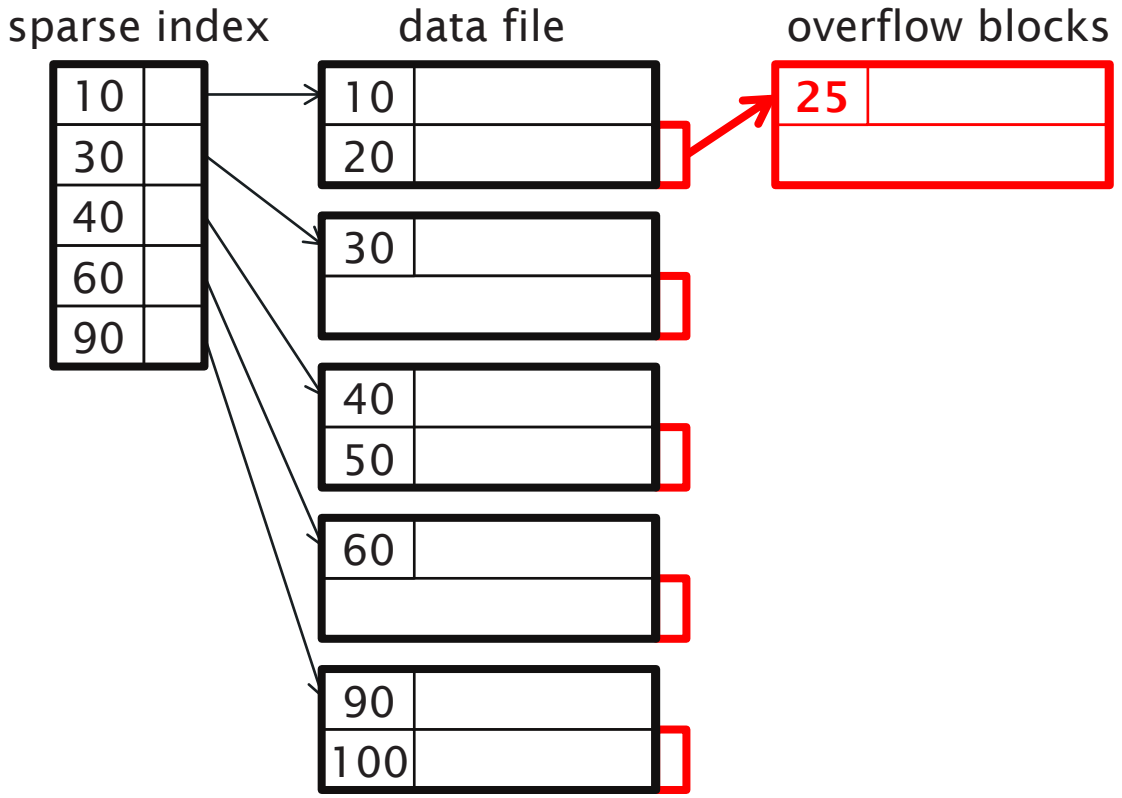
Insertion into Sparse Index

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



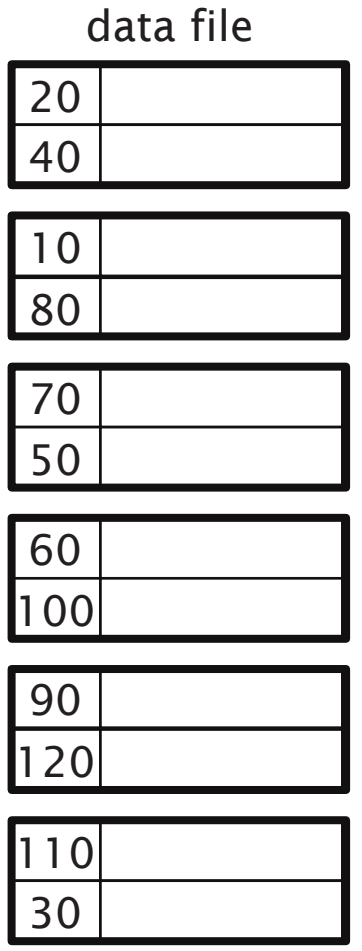
Insertion into Sparse Index

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



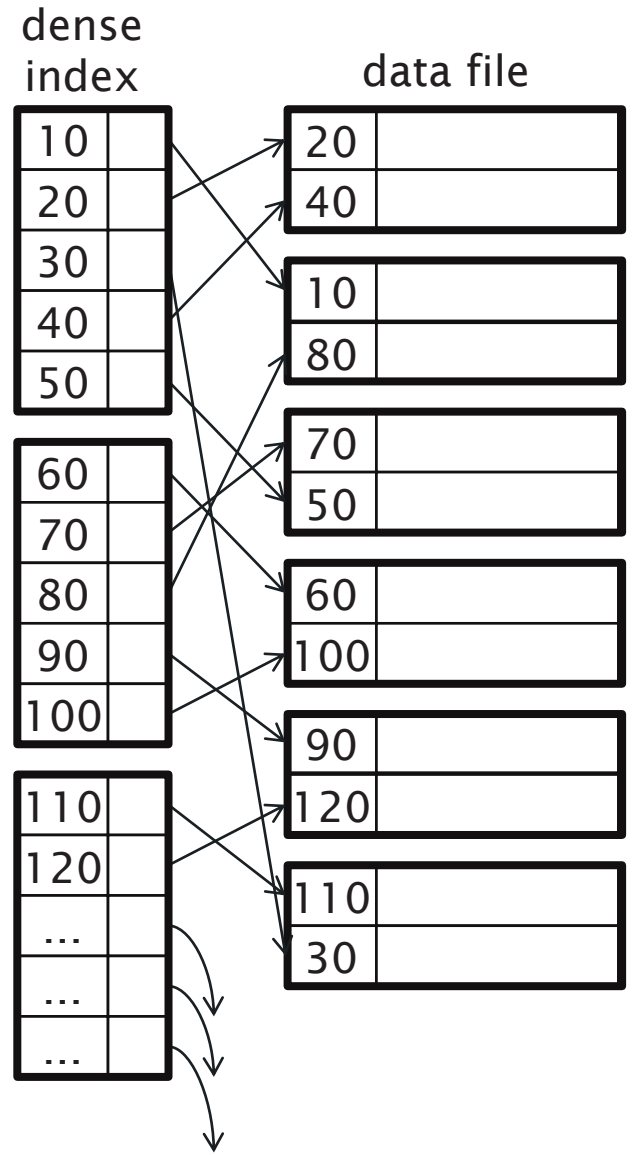
Secondary Indexes

- 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



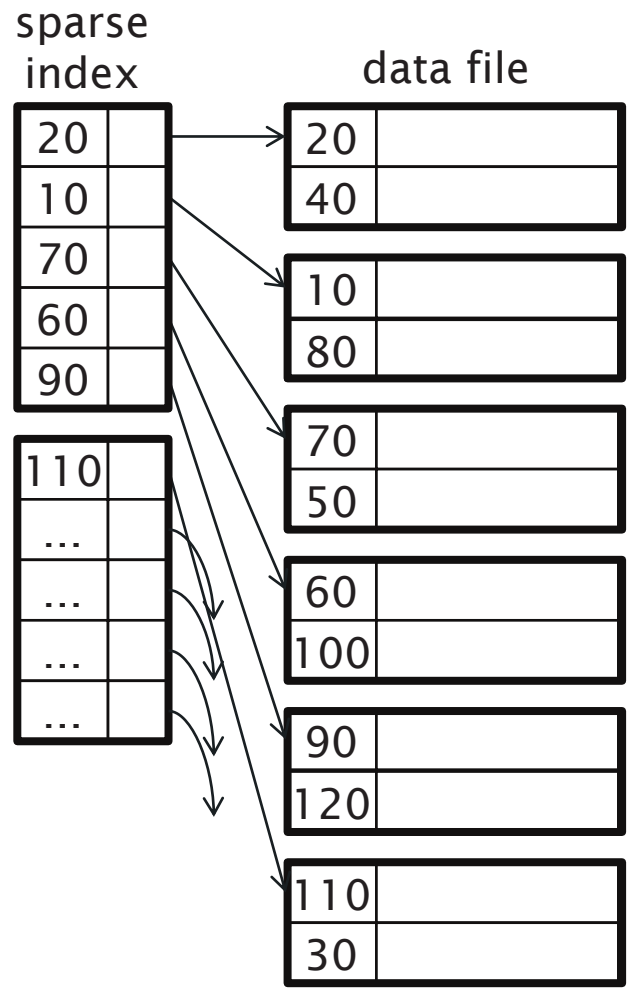
Secondary Indexes

- 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



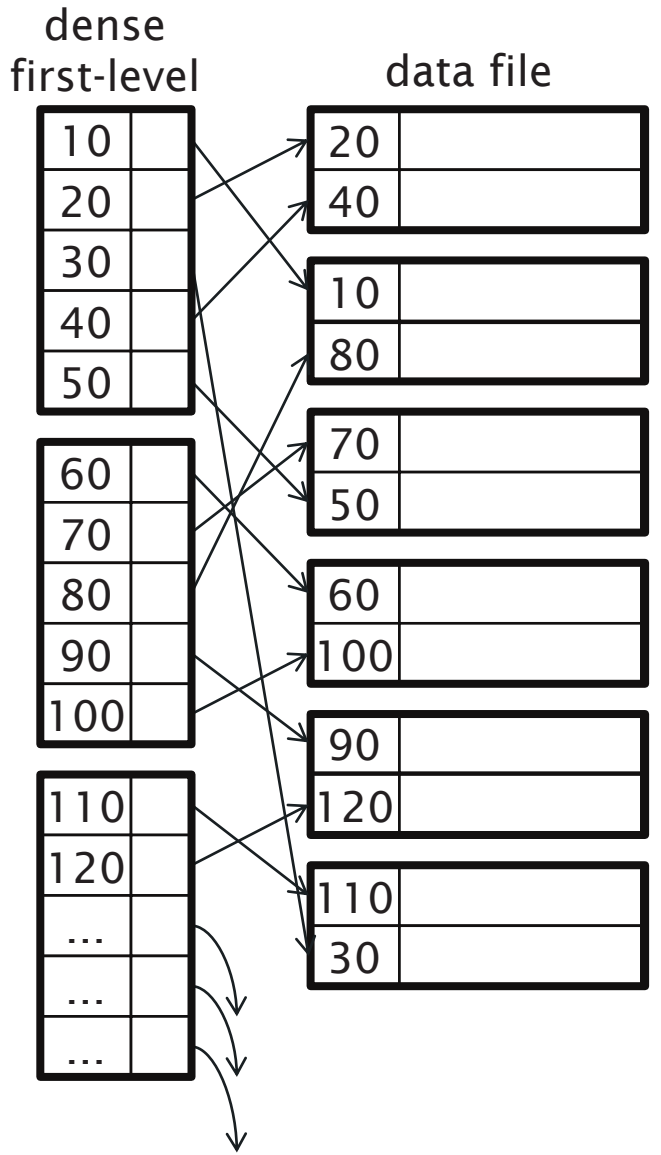
Secondary Indexes

- Sparse secondary indexes make no sense



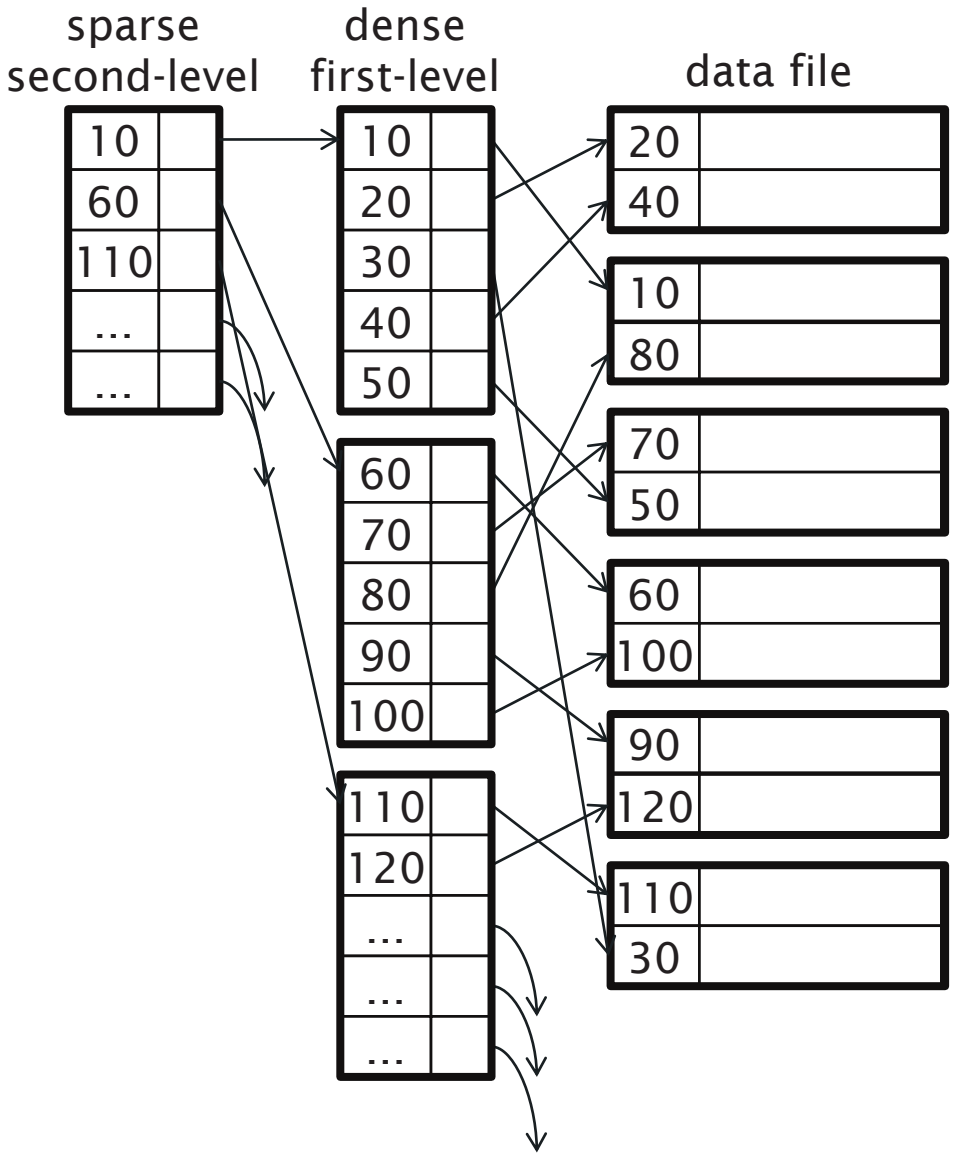
Secondary Indexes

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



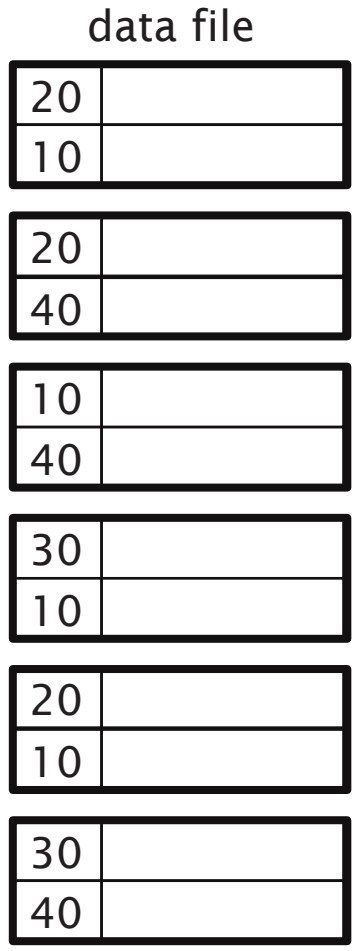
Secondary Indexes

- 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

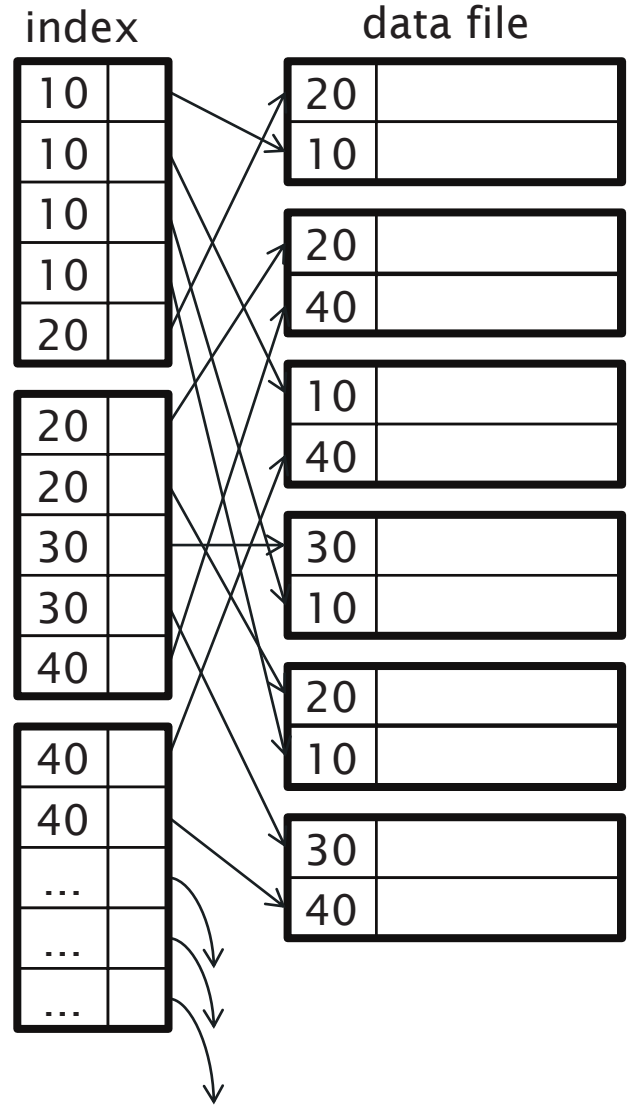


Duplicate values

Solution #1: repeated entries

Problems

- excess disk space
- excess search time

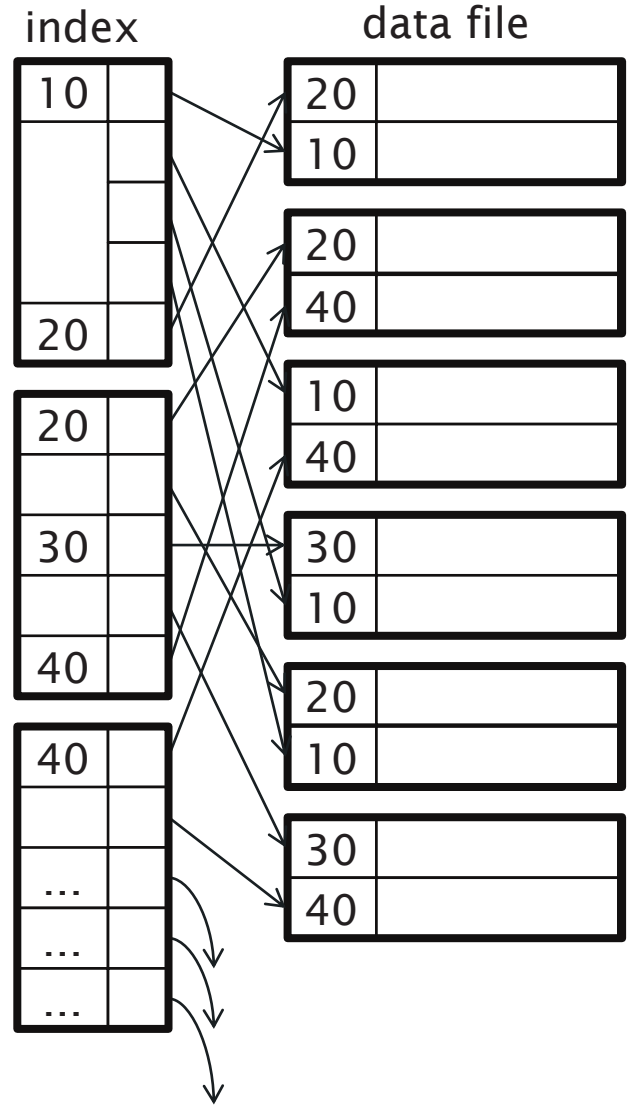


Duplicate values

Solution #2: drop repeated keys

Problems

- variable size records in index

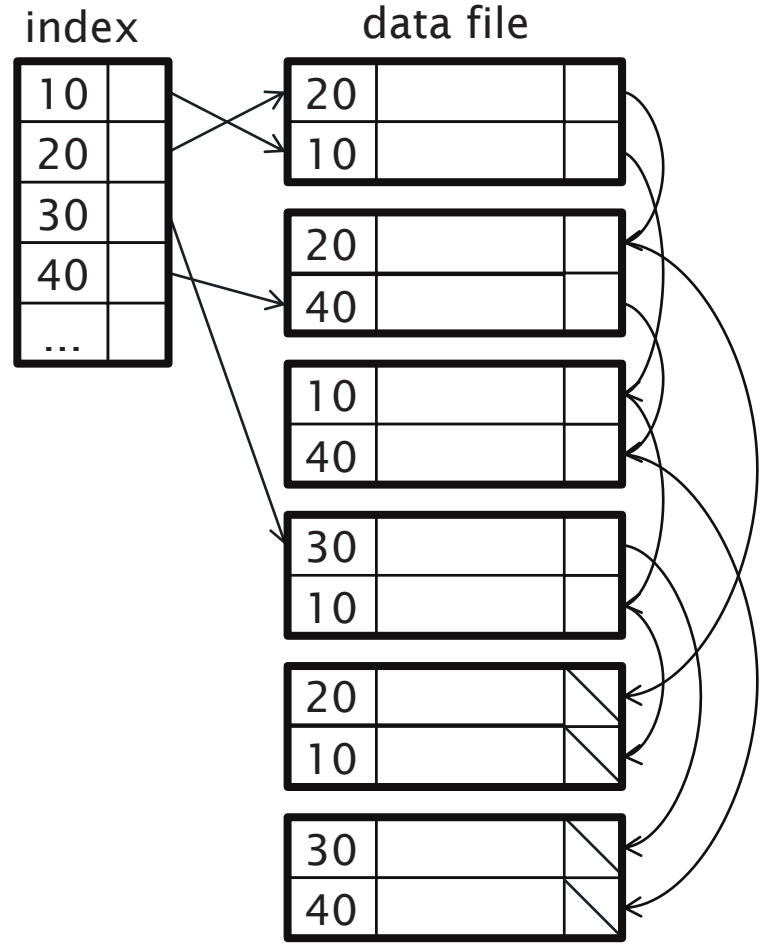


Duplicate values

Solution #3: chain records with same key

Problems

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

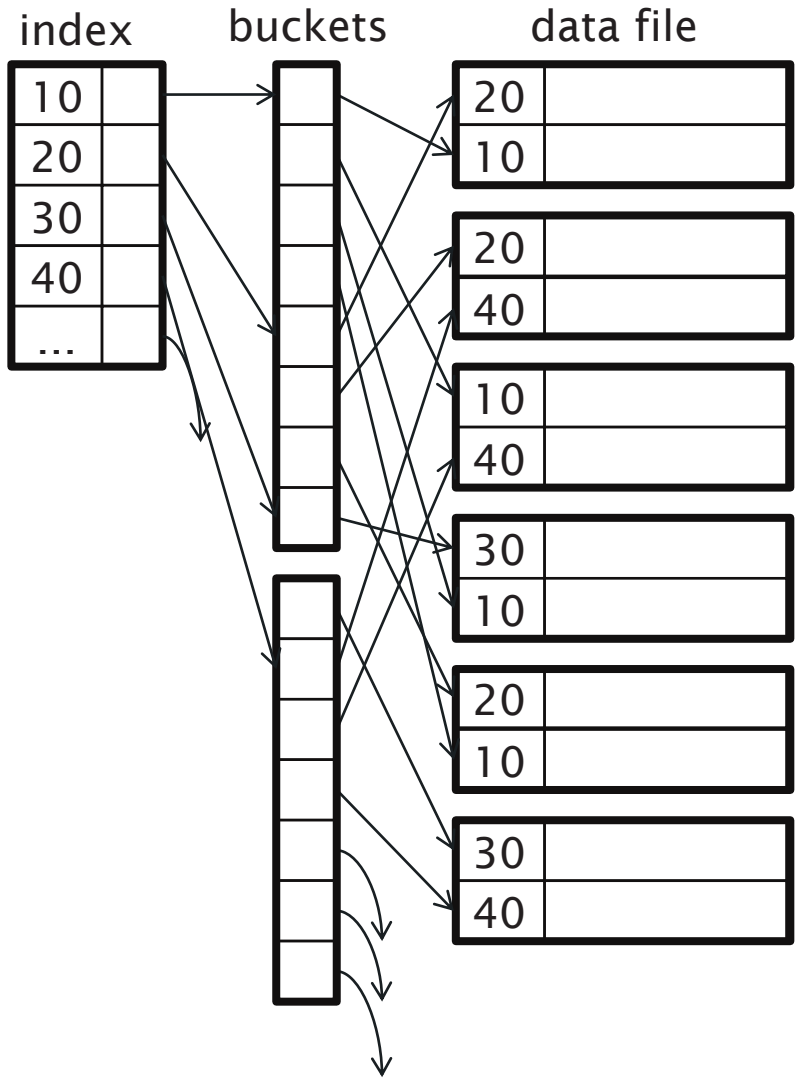


Duplicate values

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!



University of Southampton

Conventional indexes

Advantages:

- Simple
- Index is sequential file and good for scans

Disadvantages:

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

University of Southampton

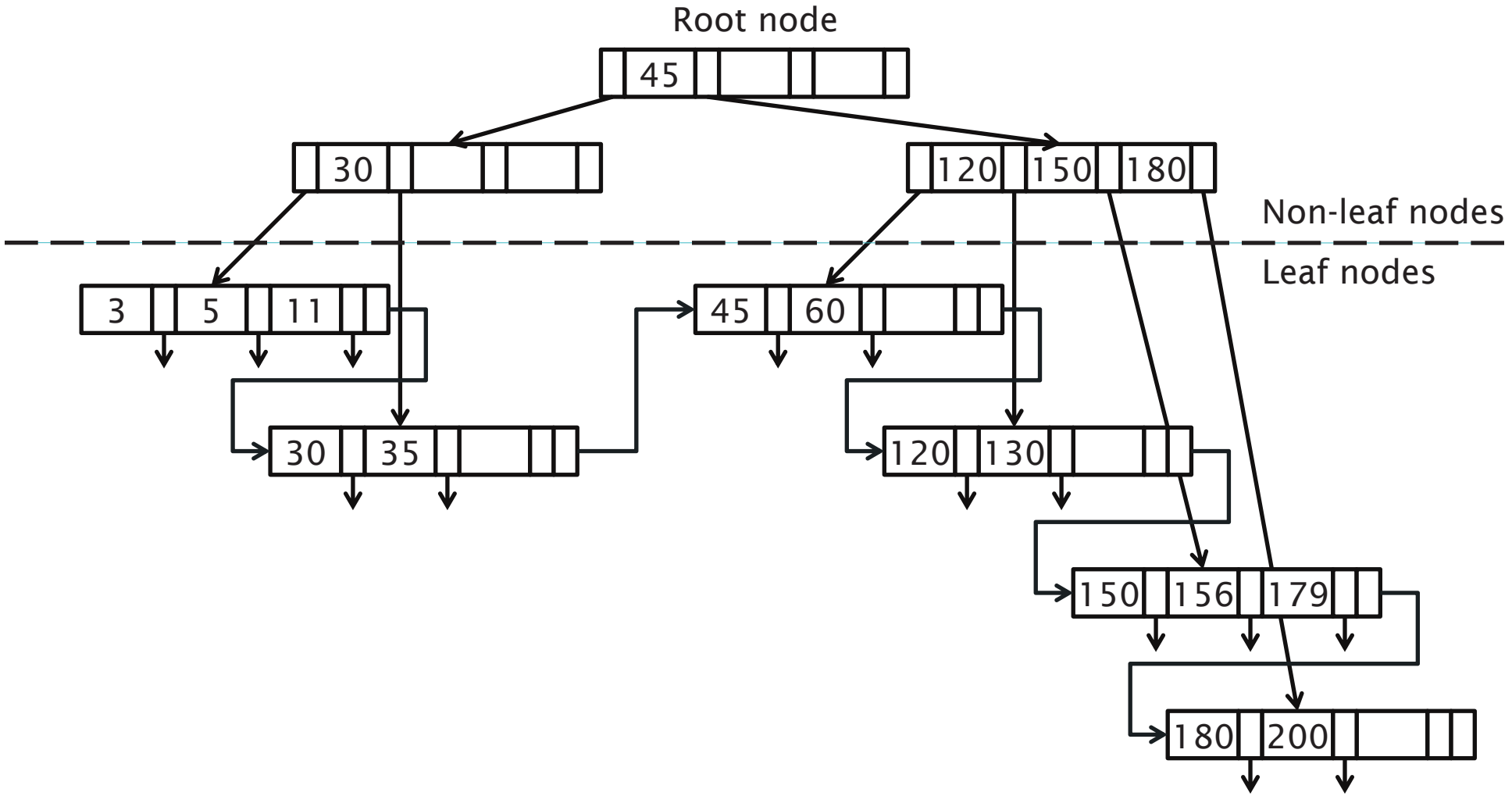
B+trees

University of Southampton

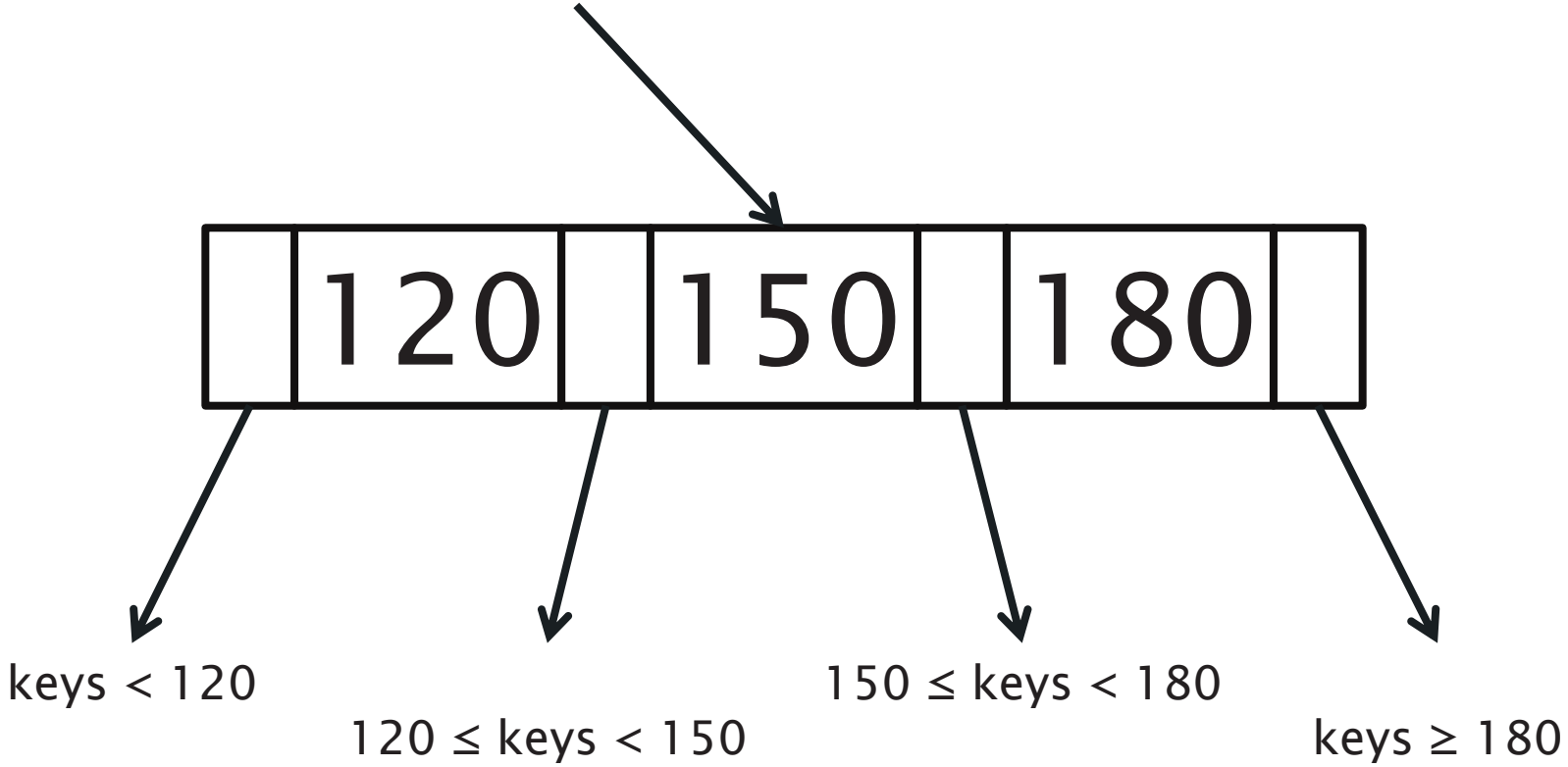
B+trees

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

B+tree example



Example non-leaf node



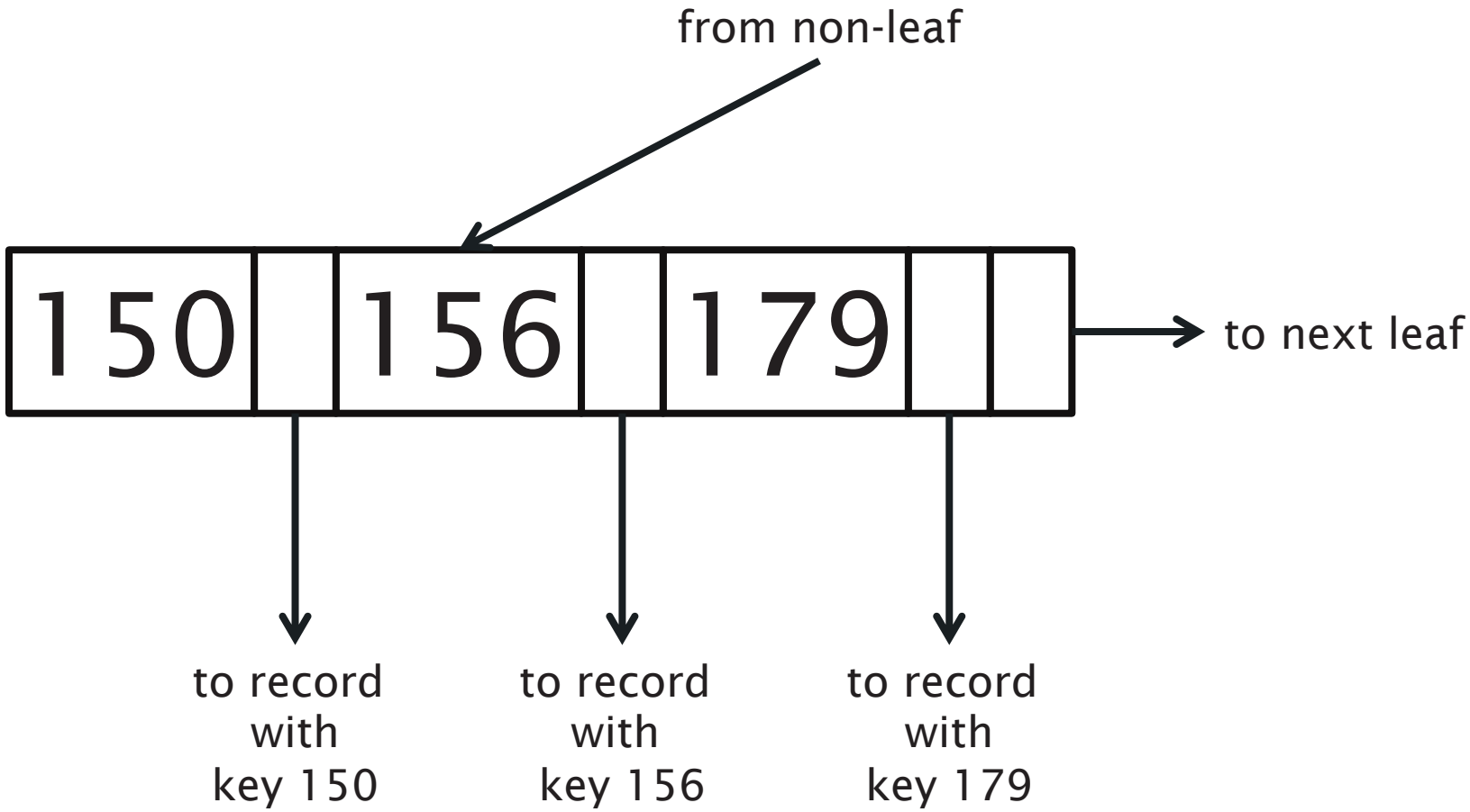
University of Southampton

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

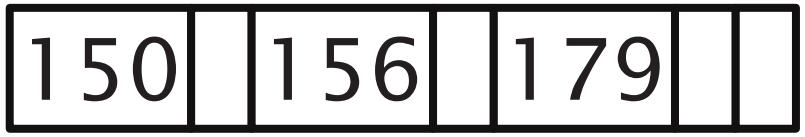
Node size

Each node is of fixed size and contains

- n keys
- n+1 pointers

non-leaf

leaf



University of Southampton

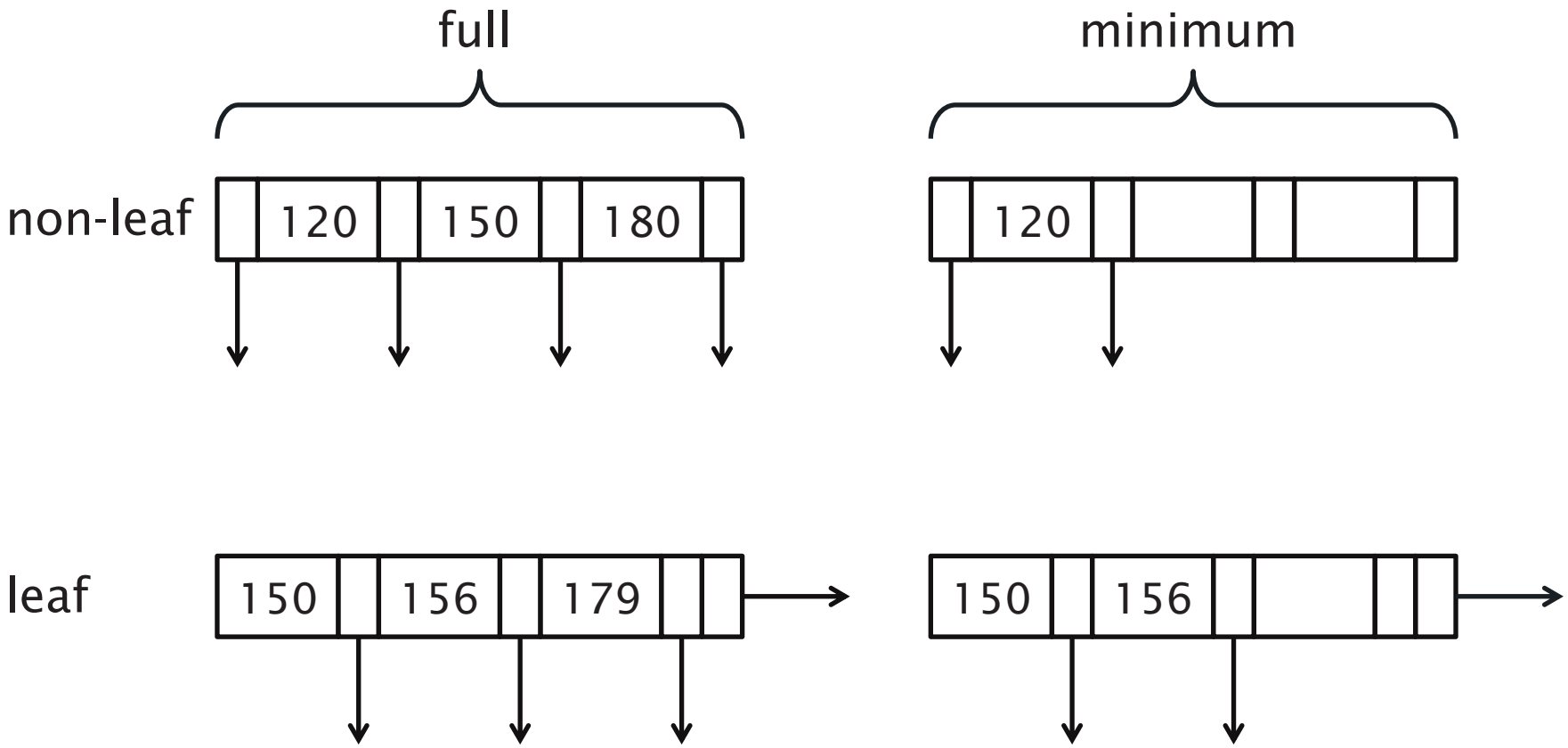
Minimum nodes

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

University of Southampton

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 lp records, where lp = $\lfloor (b-p)/r \rfloor$ = 39 records

A leaf node in a secondary index can accommodate ls records,
where ls = $\lfloor (b-p)/(k+p) \rfloor$ = 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
  $i*u$ = blocks
- Each leaf initially contains
  $lp*u$ = records
- 1 level of non-leaf nodes initially points to
  $(lp*u)(i*u)$ = records
- 2 levels of non-leaf nodes initially point to
  $(i*u)^2$ = blocks
  $(lp*u)(i*u)^2$ = 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 blocks
- Each leaf initially contains
  $lp*u$ = 23 records
- 1 level of non-leaf nodes initially points to
  $(lp*u)(i*u)$ = 3,427 records
- 2 levels of non-leaf nodes initially point to
  $(i*u)^2$ = 22,201 blocks
  $(lp*u)(i*u)^2$ = 510,623 records

B+tree secondary index

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

- A non-leaf node initially points to
  $i*u$ = blocks
- A leaf node initially points at
  $ls*u$ = records
- 1 level of non-leaf nodes initially points to
  $(ls*u)(i*u)$ = records
- 2 levels of non-leaf nodes initially point to
  $(ls*u)(i*u)^2$ = 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 blocks
- A leaf node initially points at
  ls*u = 149 records
- 1 level of non-leaf nodes initially points to
  (ls*u)(i*u) = 22,201 records
- 2 levels of non-leaf nodes initially point to
  $(ls*u)(i*u)^2$ = 3,307,949 records

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

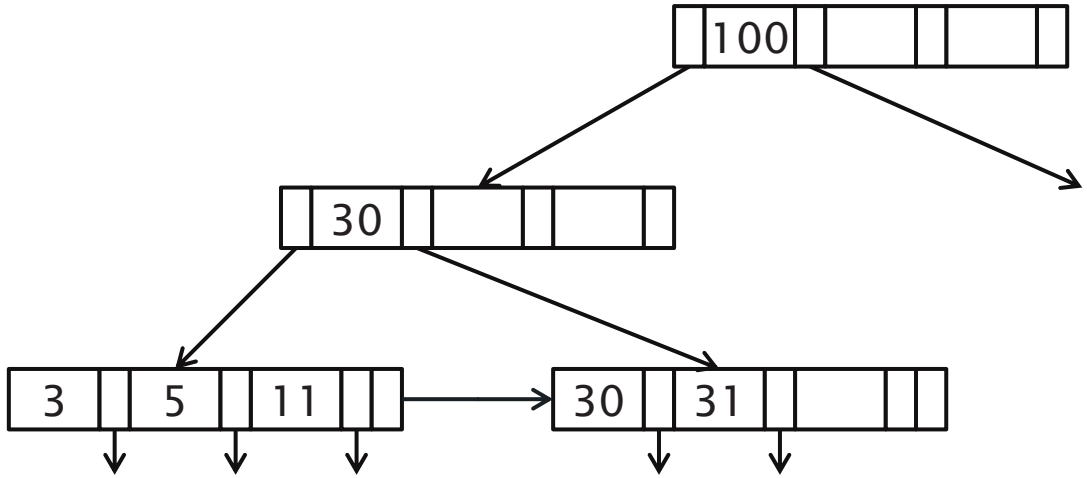
University of Southampton

B+tree Insertion

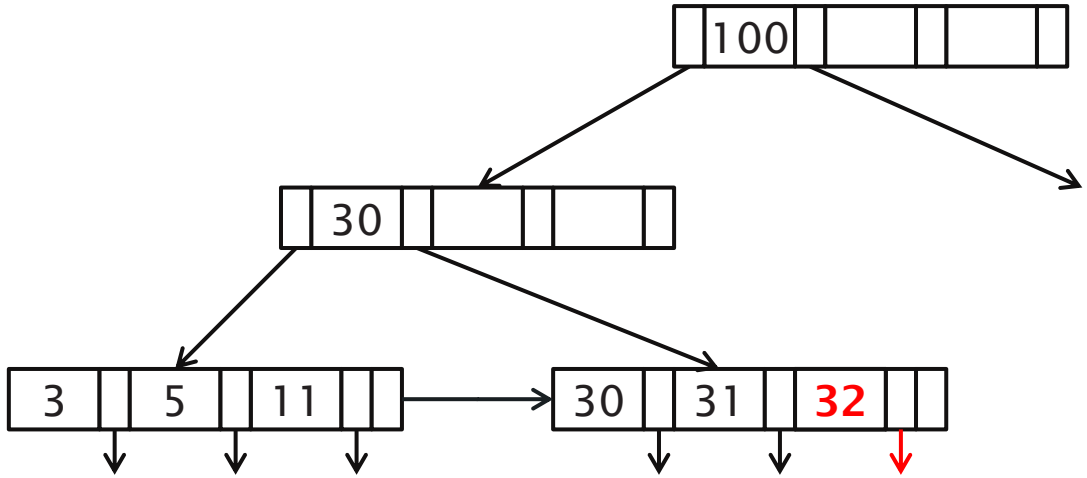
Four cases to consider:

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

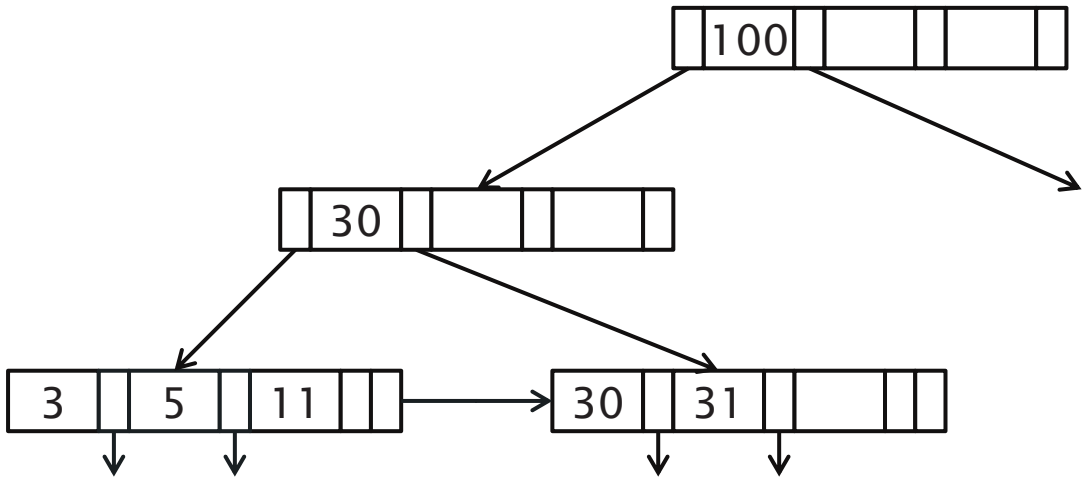
Case 1: insert key=32



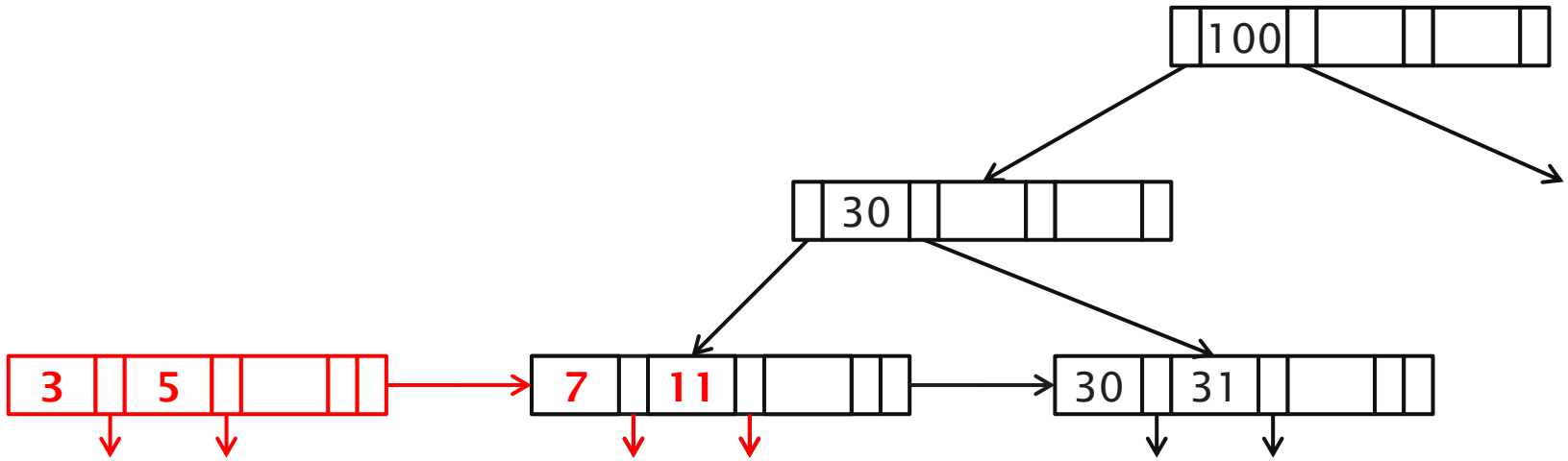
Case 1: insert key=32



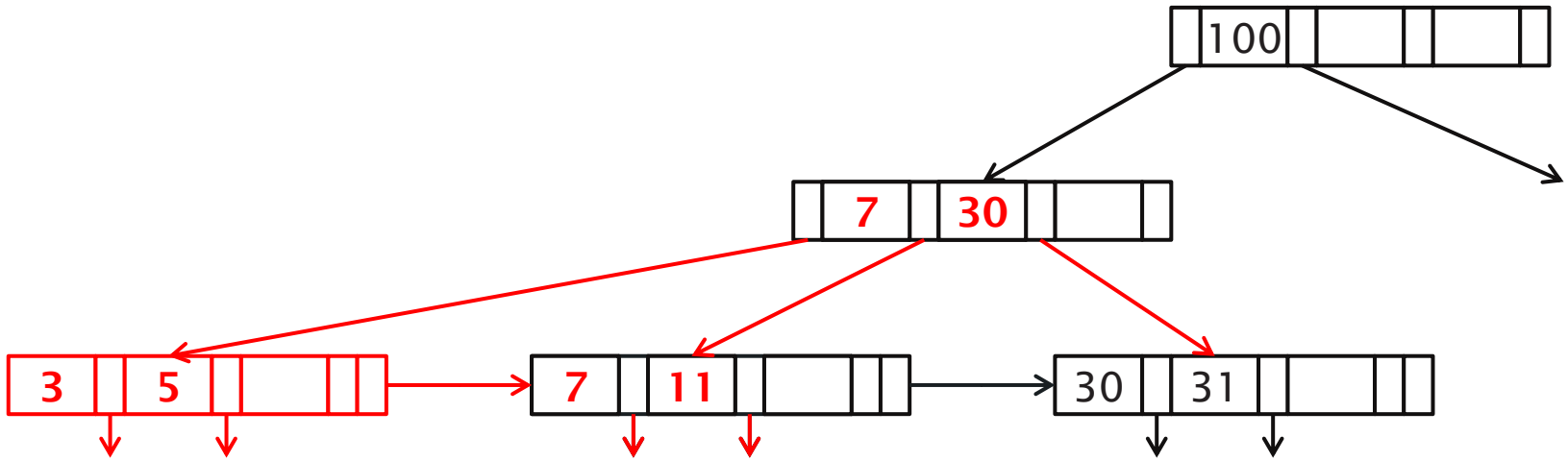
Case 2: insert key=7



Case 2: insert key=7

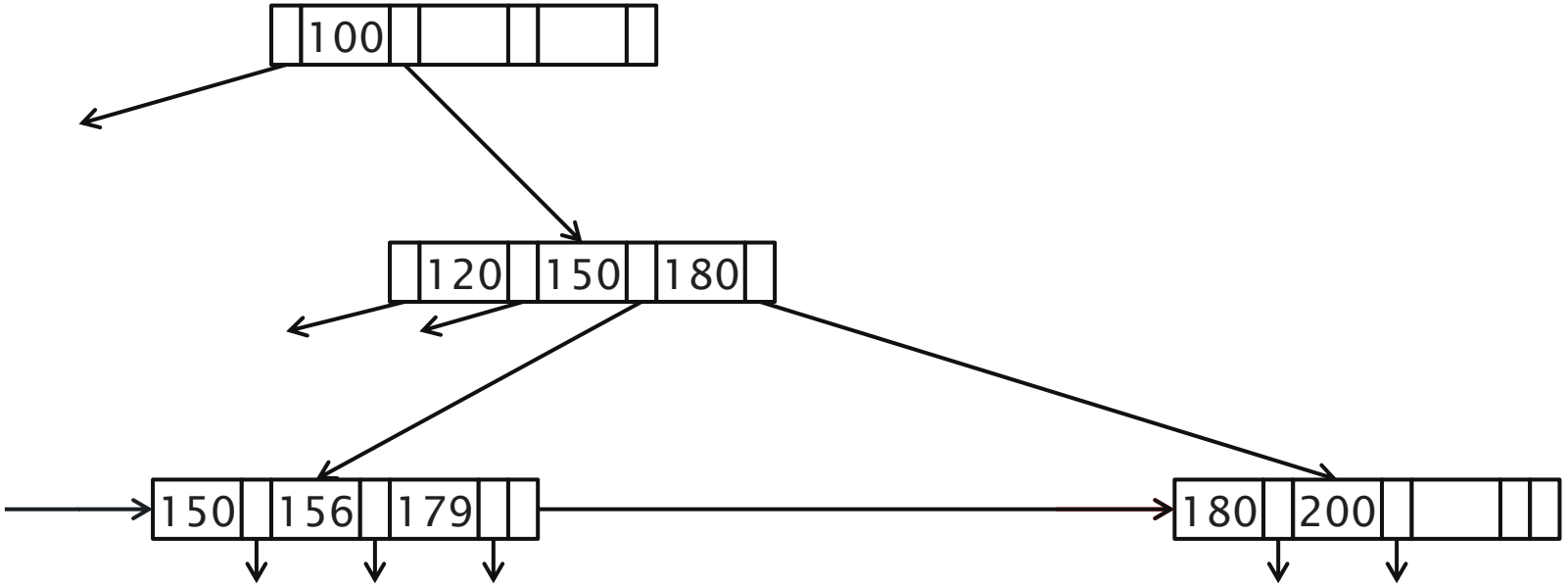


Case 2: insert key=7

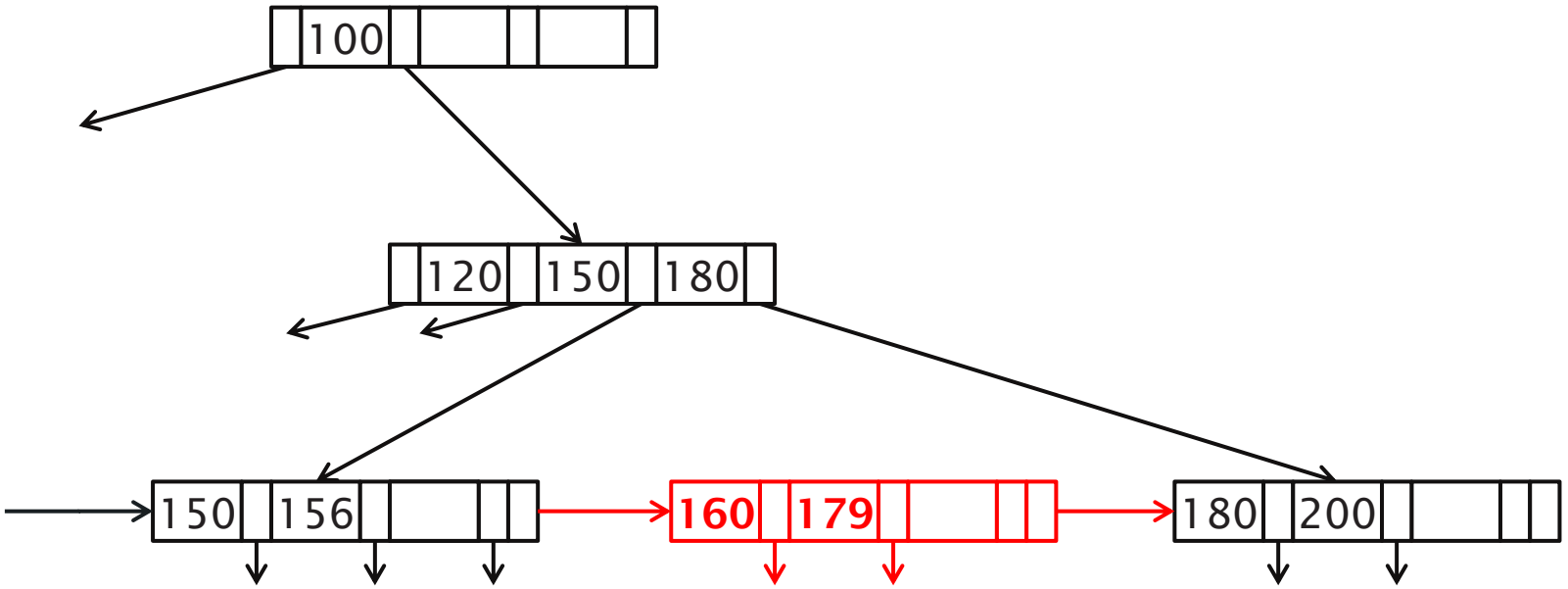


University of Southampton

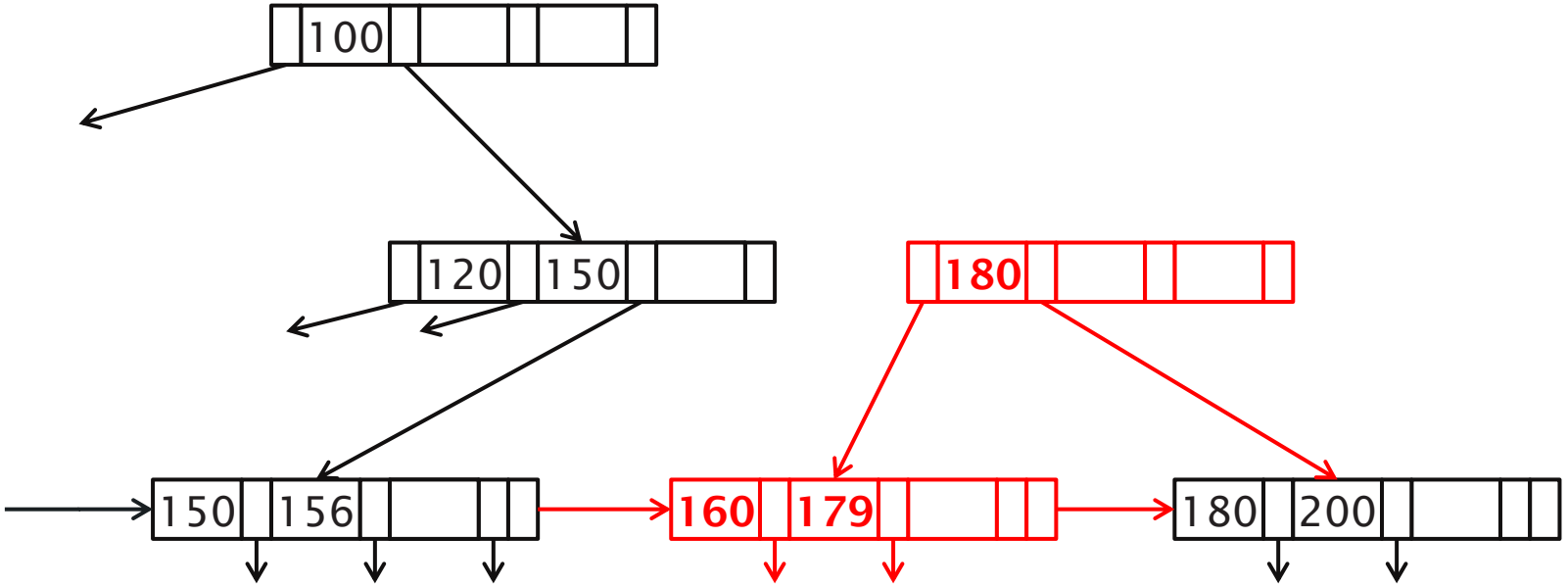
Case 3: insert key=160



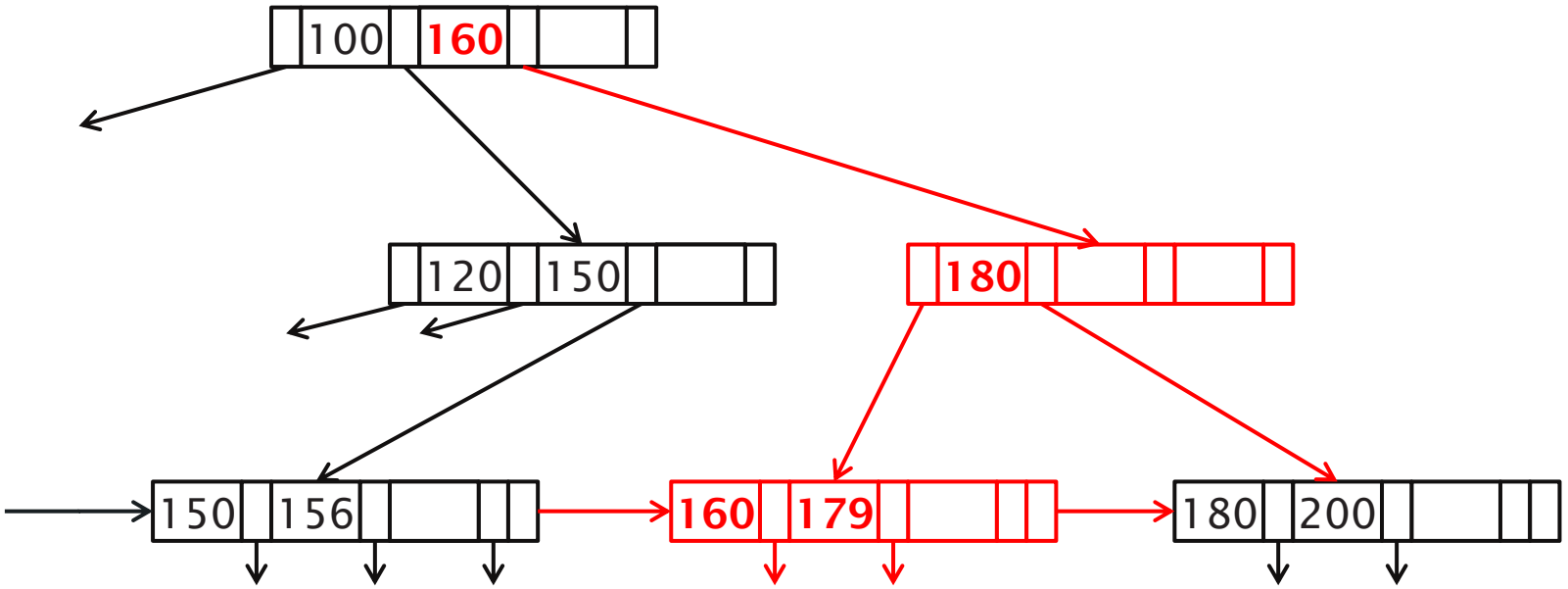
Case 3: insert key=160



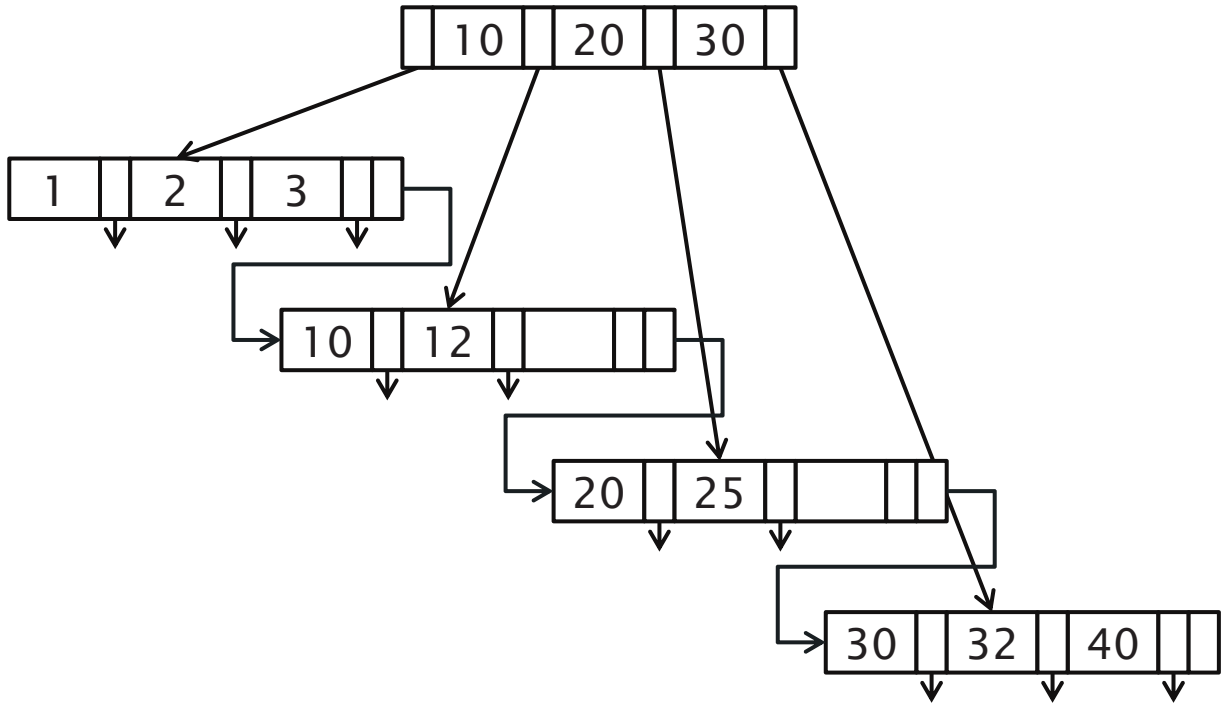
Case 3: insert key=160



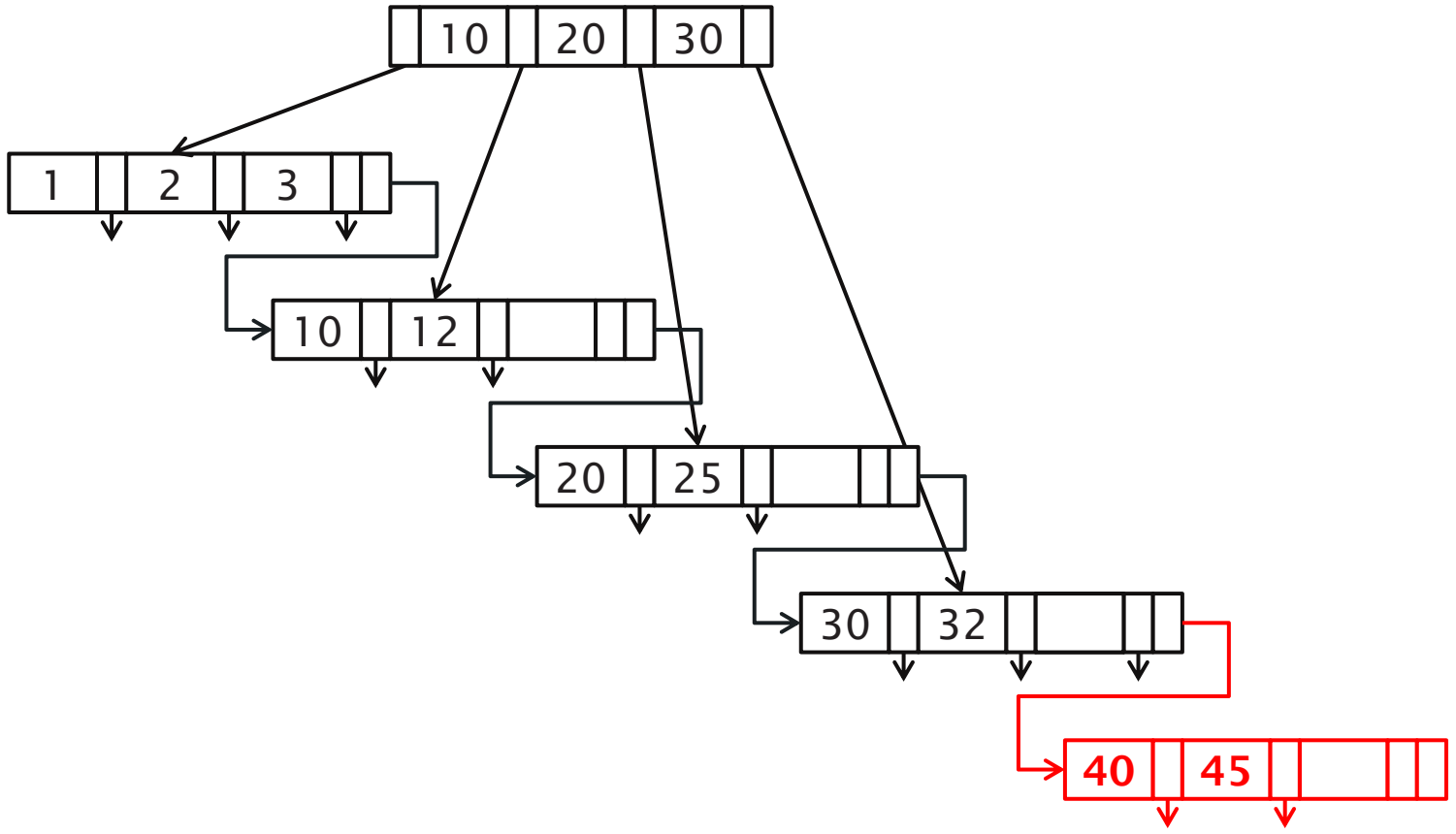
Case 3: insert key=160



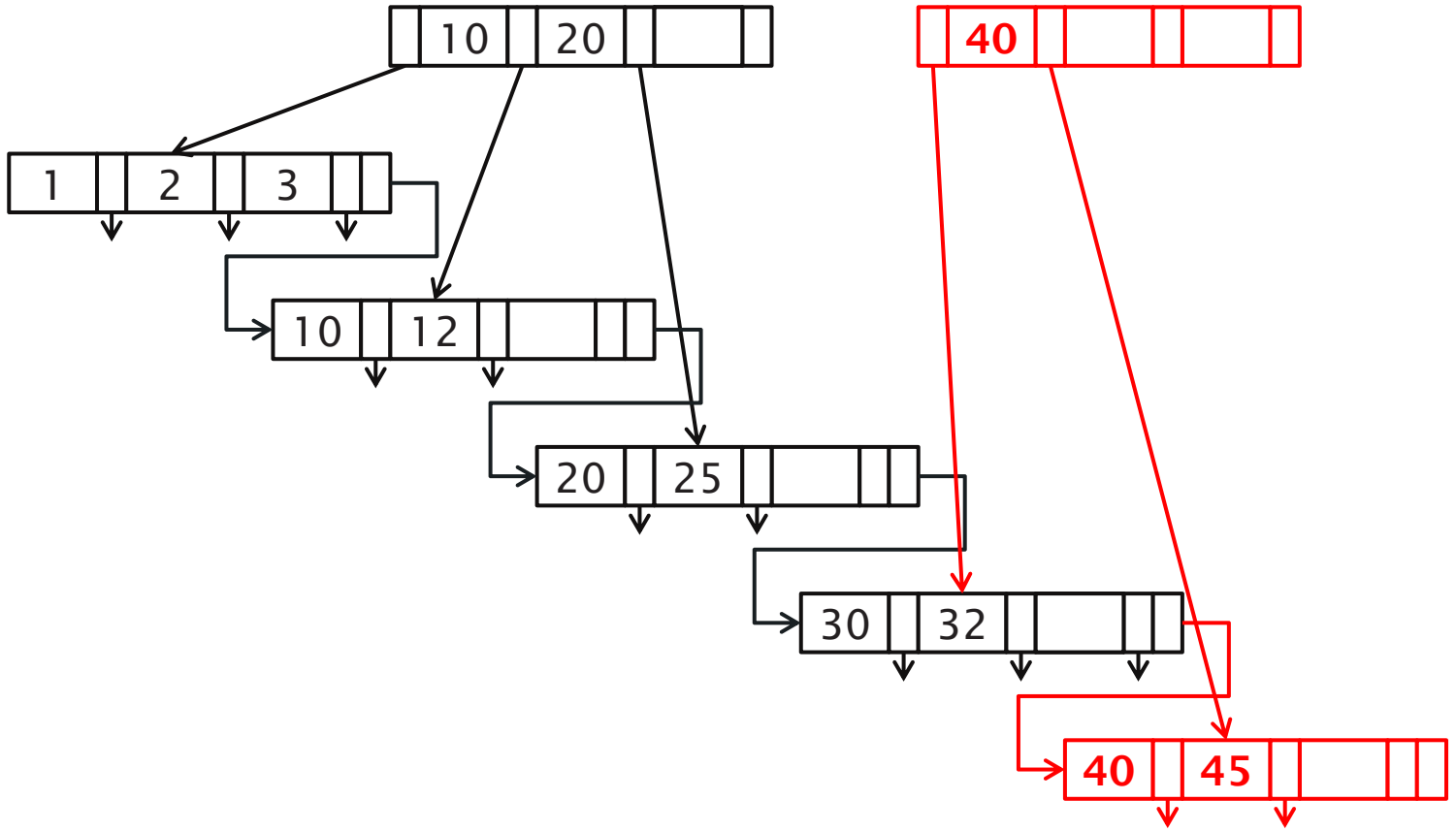
Case 4: insert 45



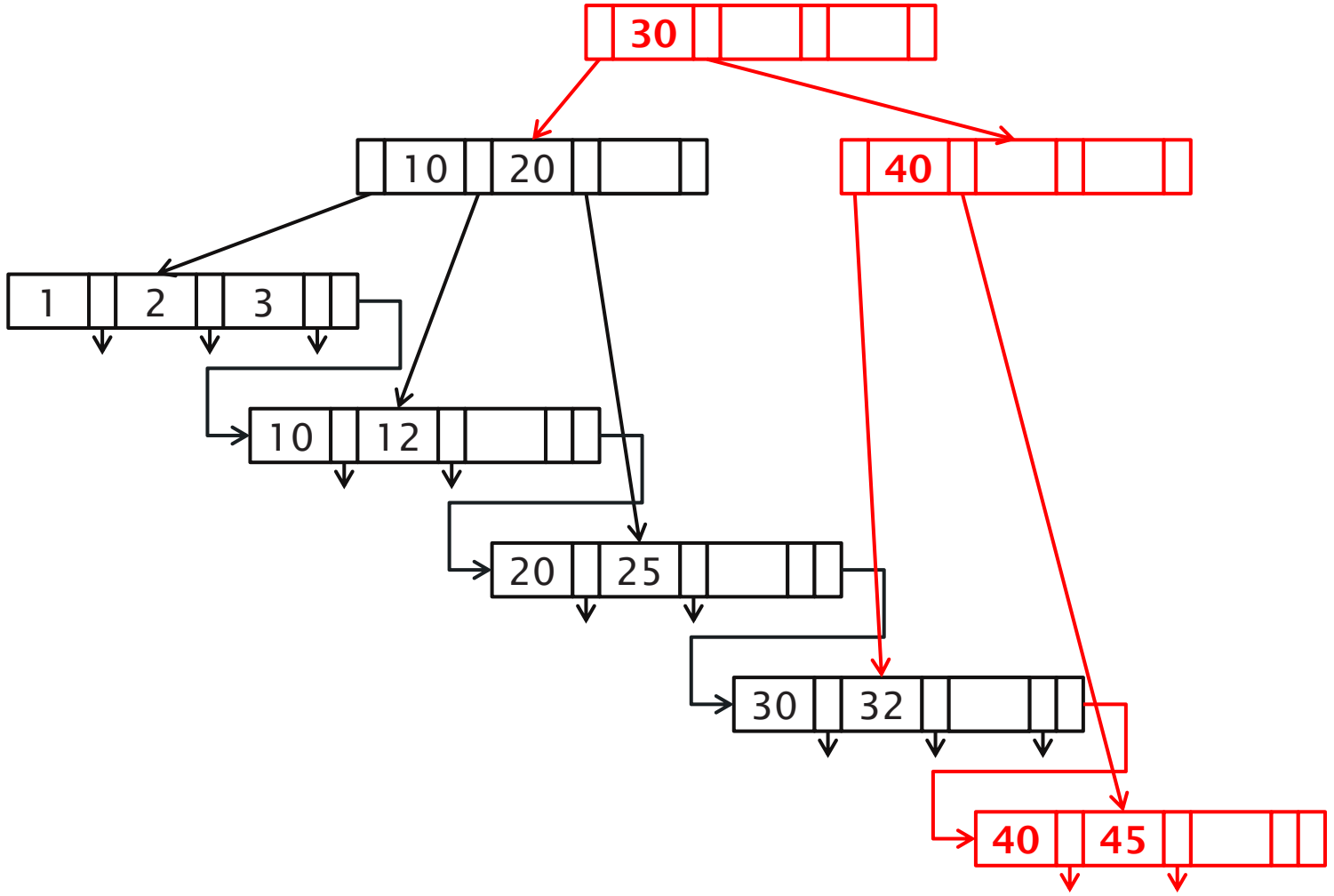
Case 4: insert 45



Case 4: insert 45



Case 4: insert 45

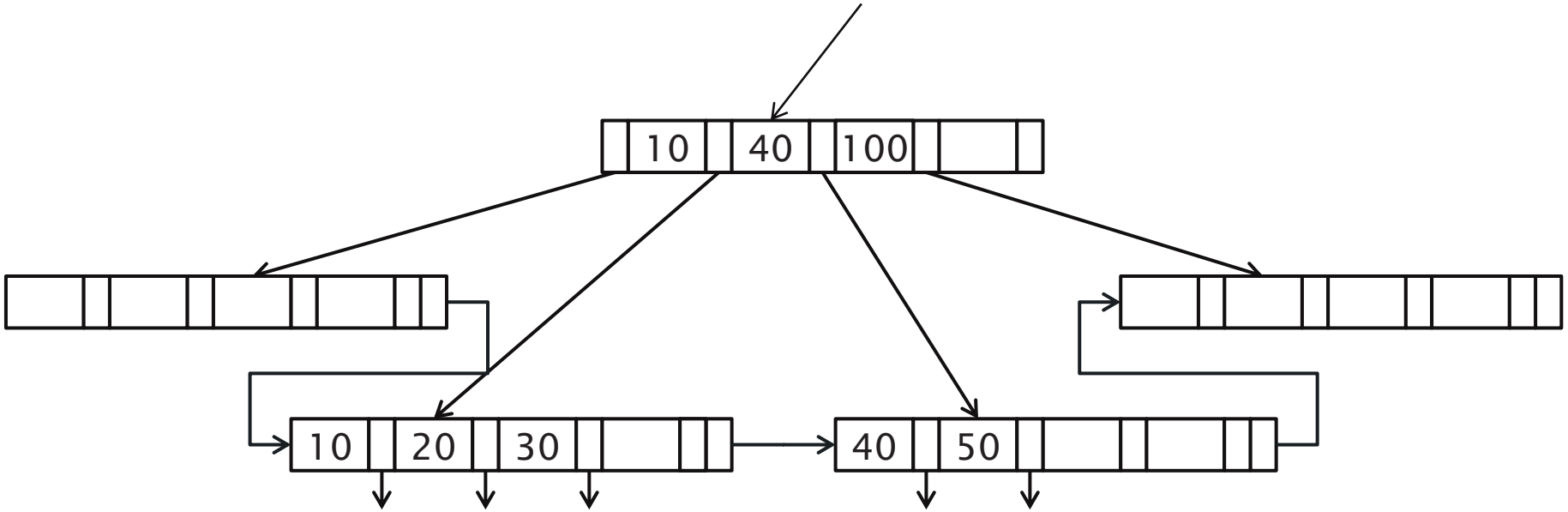


B+tree Deletion

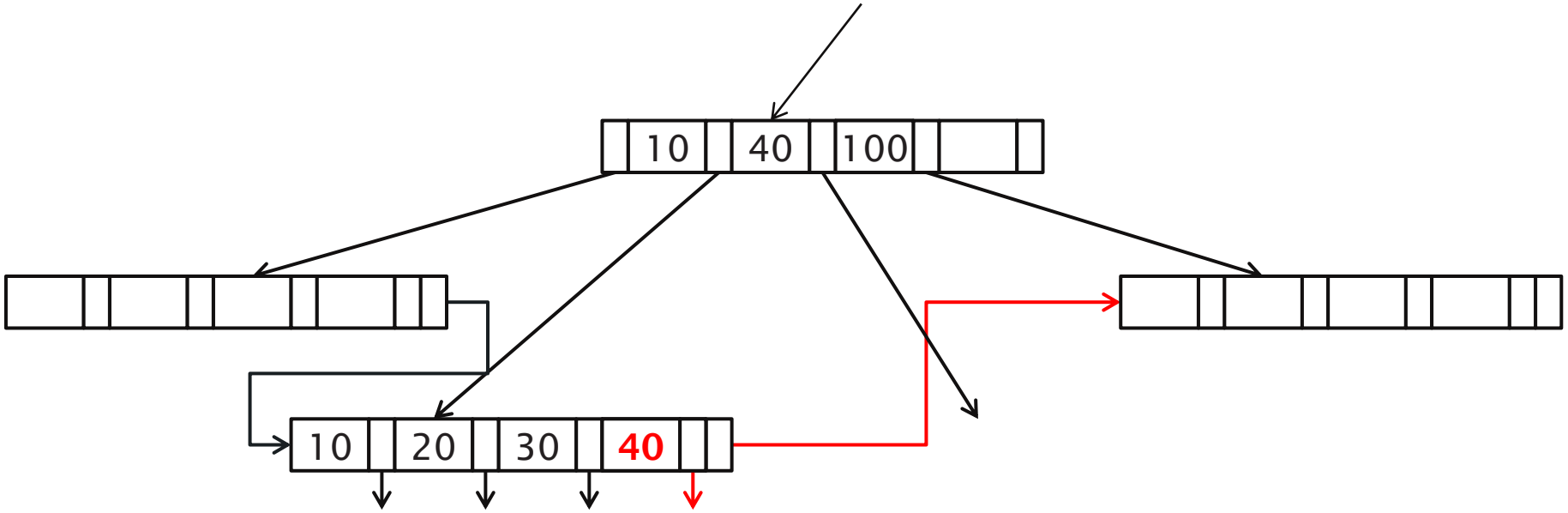
Four cases to consider:

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

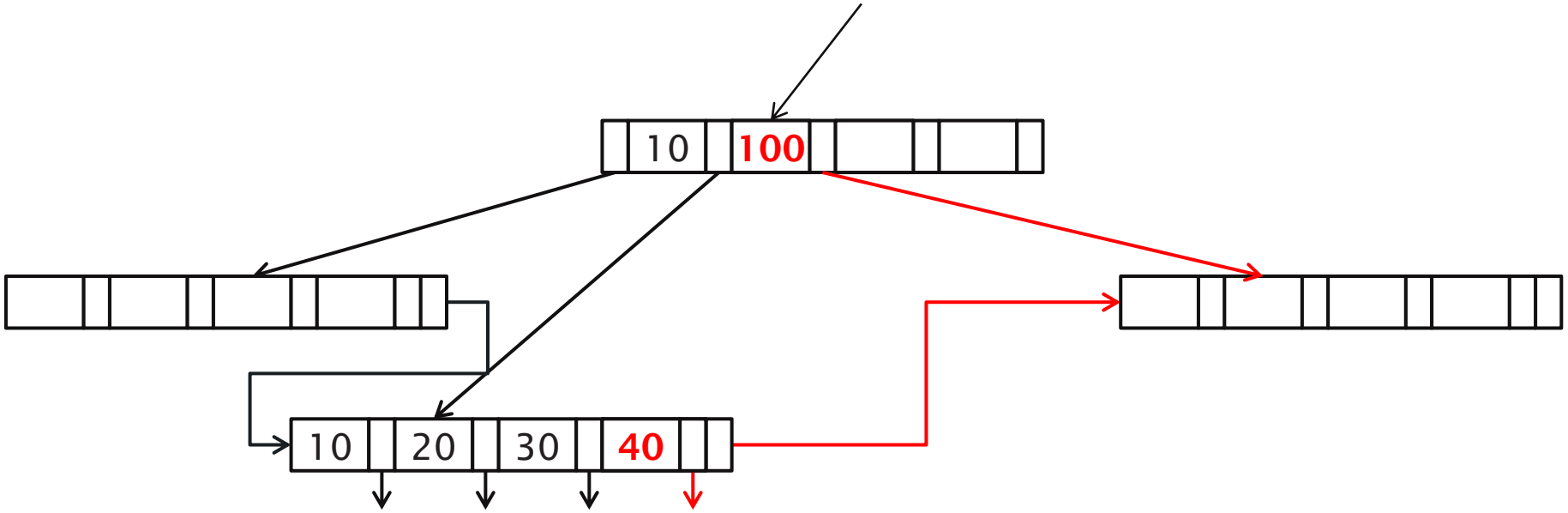
Case 2: delete key=50 (n=4)



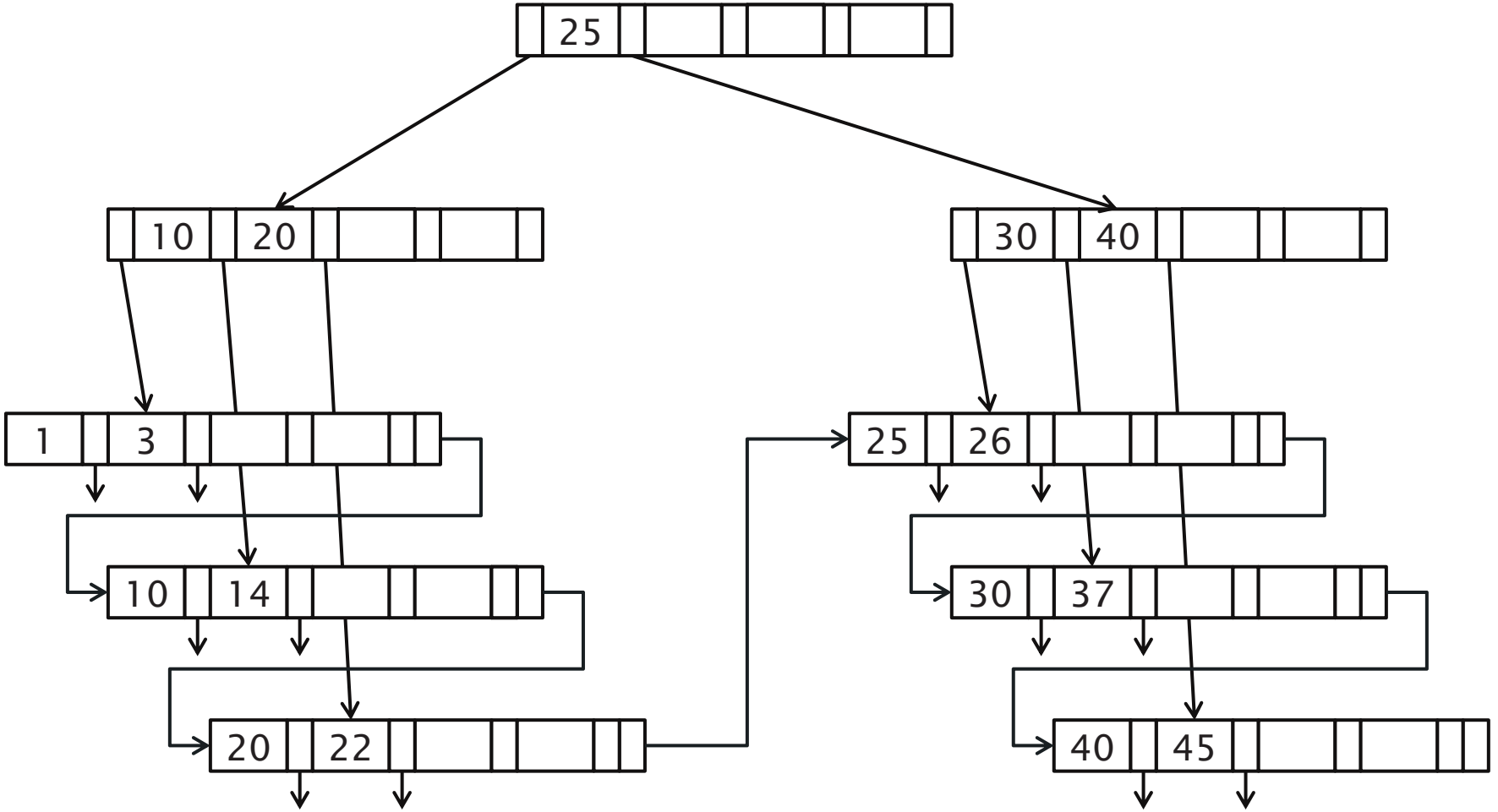
Case 2: delete key=50 (n=4)



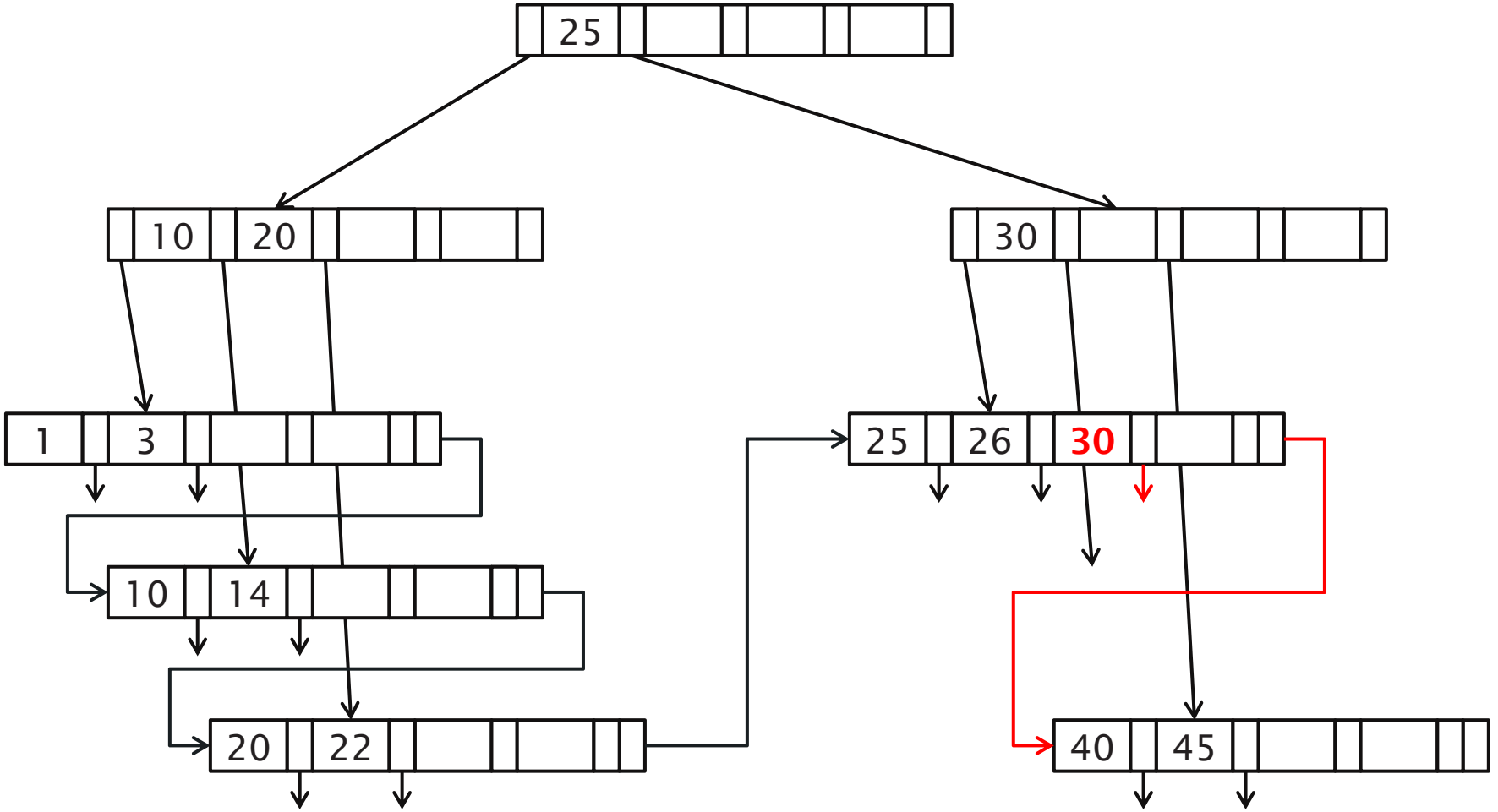
Case 2: delete key=50 (n=4)



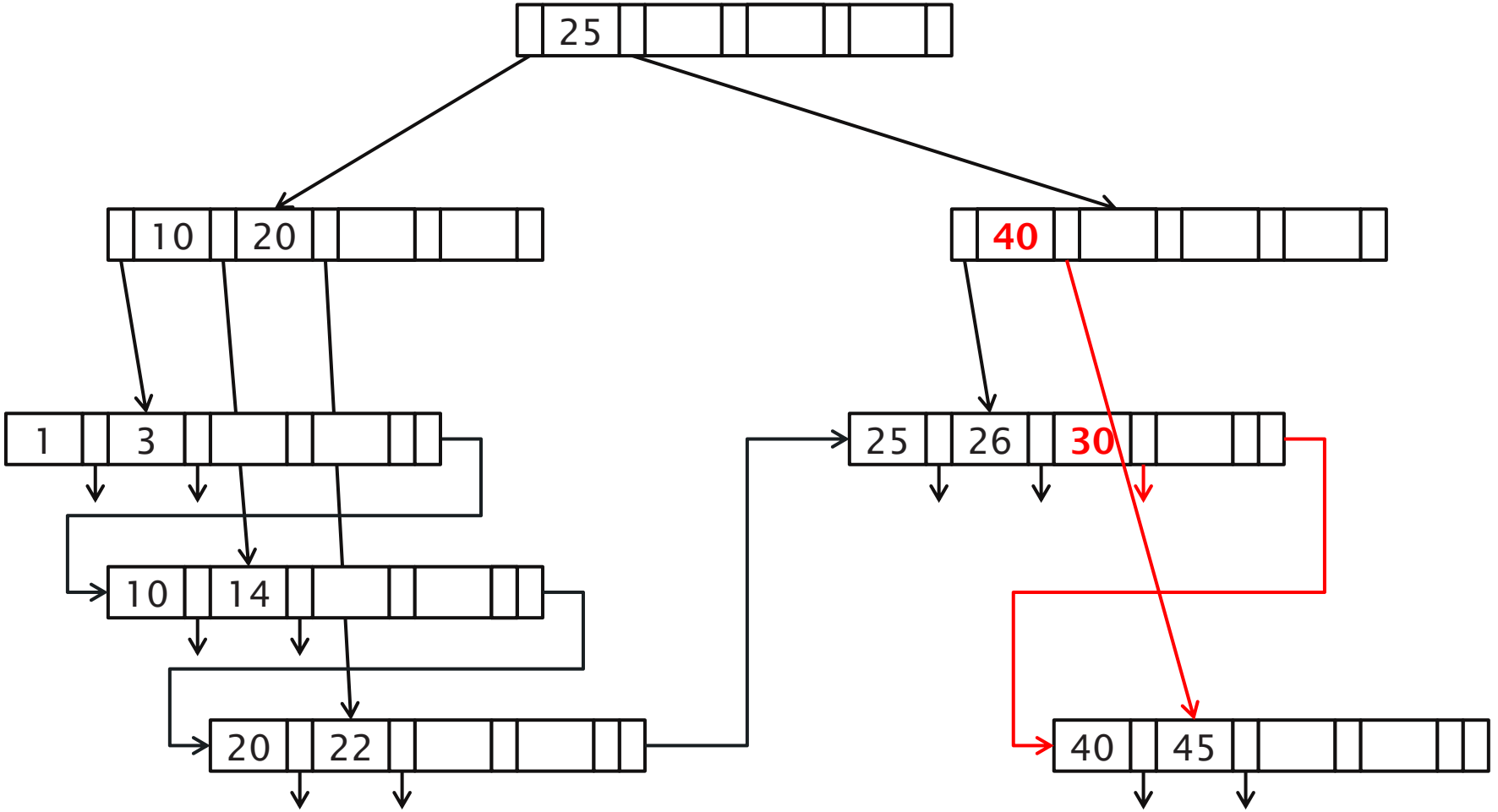
Case 4: delete key=37 (n=4)



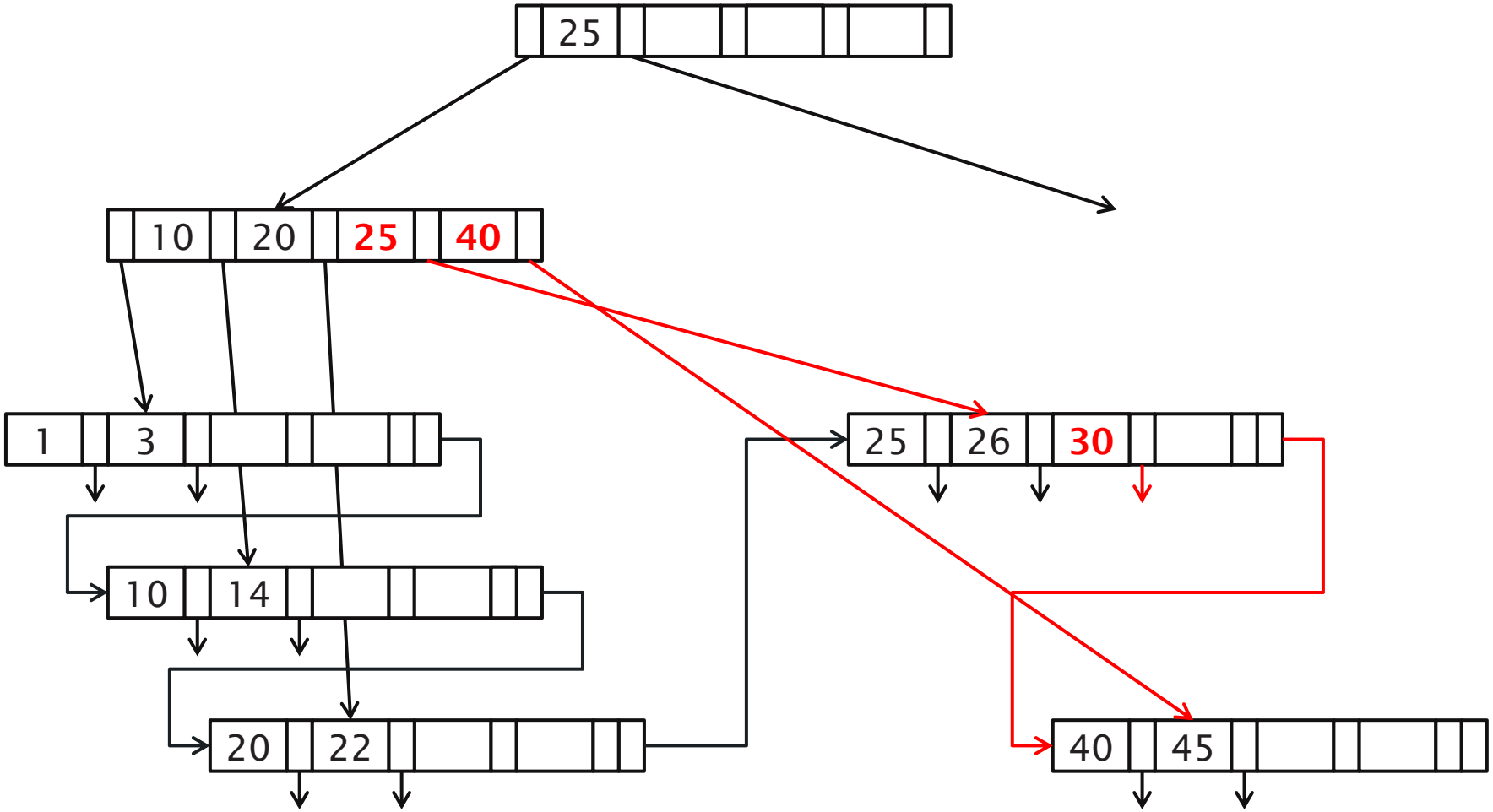
Case 4: delete key=37 (n=4)



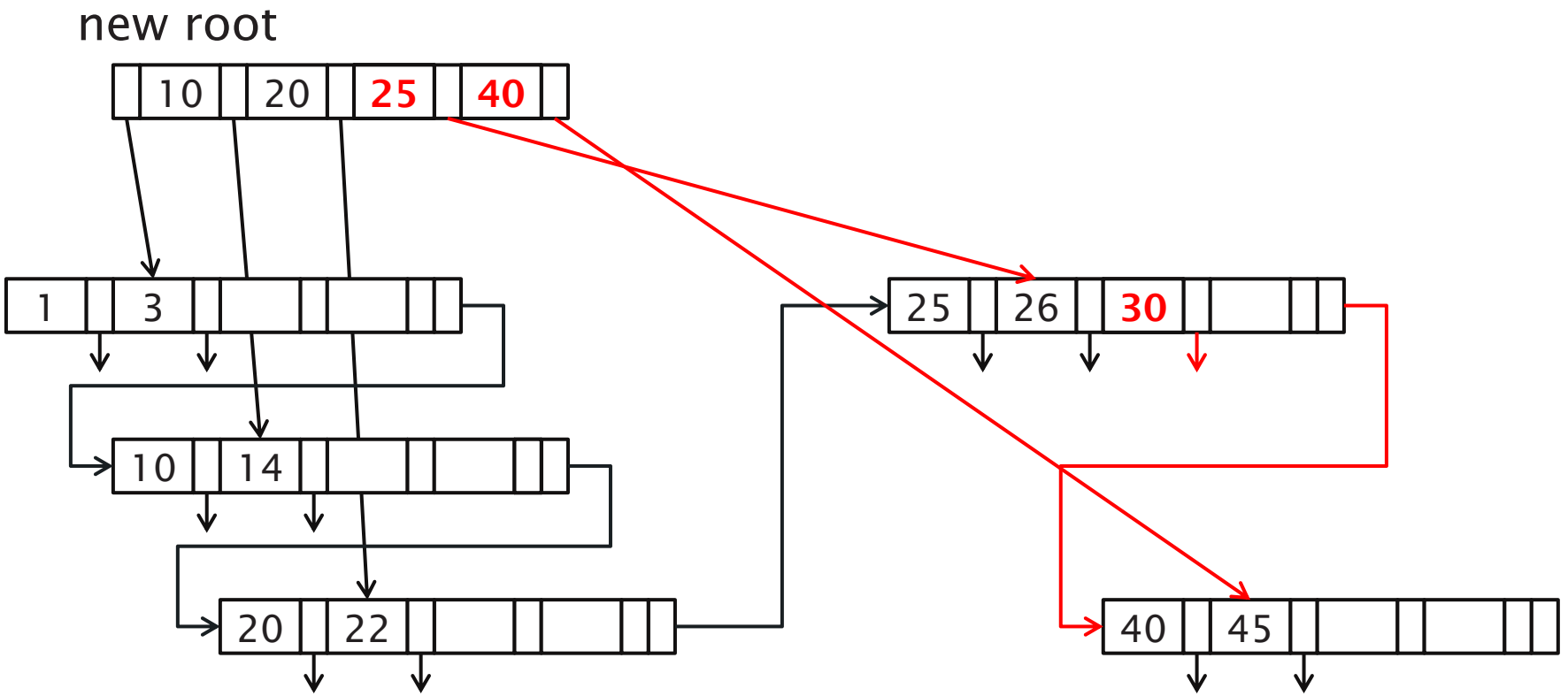
Case 4: delete key=37 (n=4)



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



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



University of Southampton

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

University of Southampton

Hashing

University of Southampton

Hashing

Main memory hash table

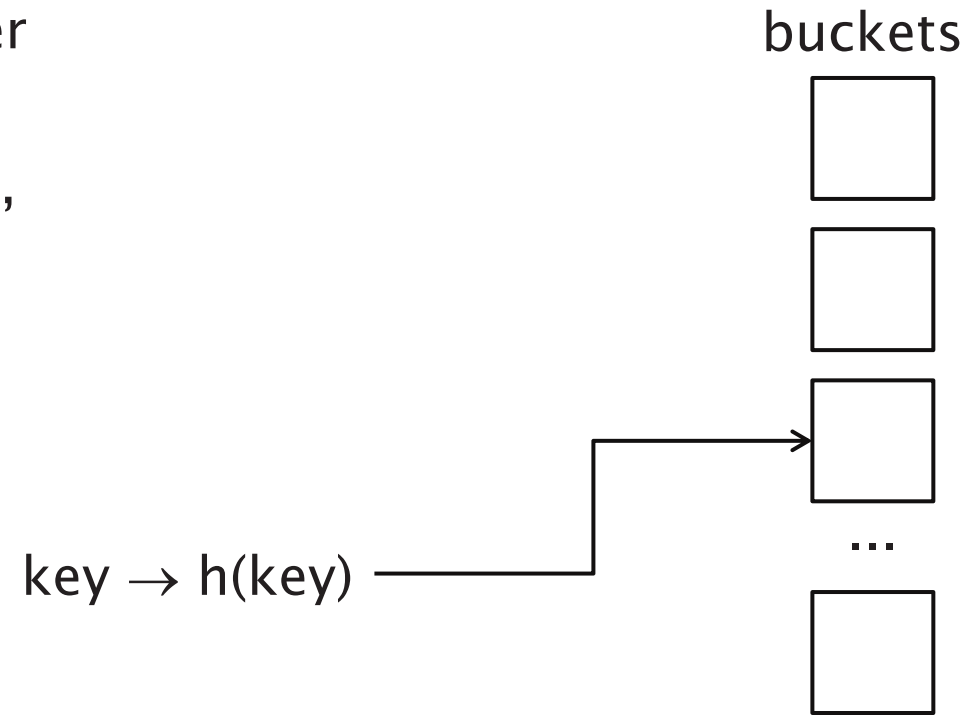
- *Hash function* h() takes a key and computes an integer value
- Value is used to select a bucket from a *bucket array*
- Bucket array contains linked lists of records

Secondary storage hash table

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

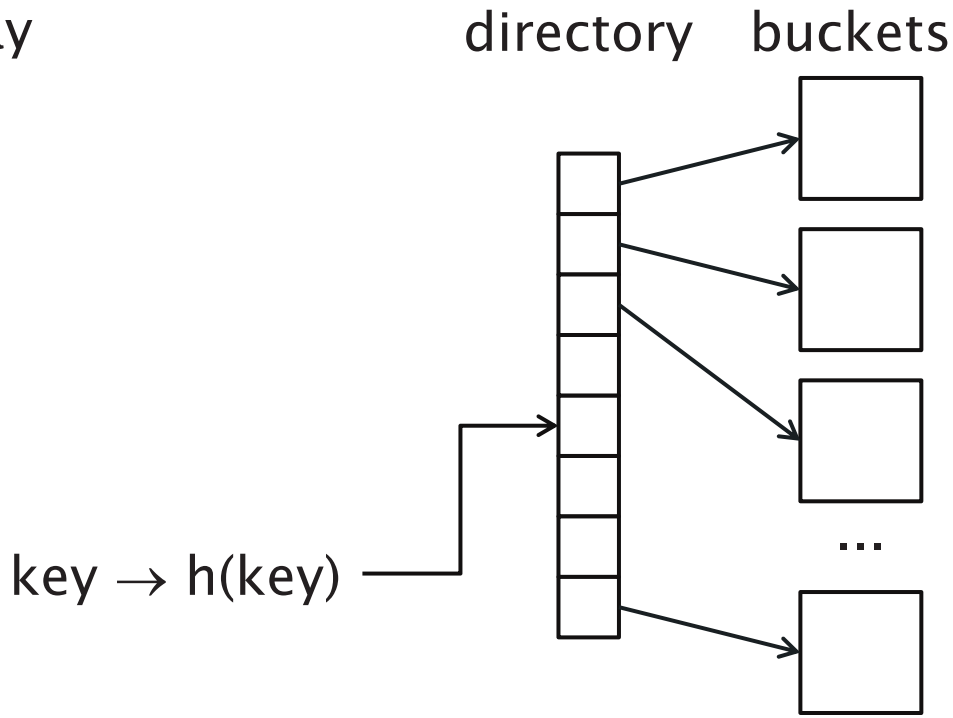
Hashing approach #1

- 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 x1 + x2 + ….. xn, compute sum modulo b

Not a particularly good function

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

University of Southampton

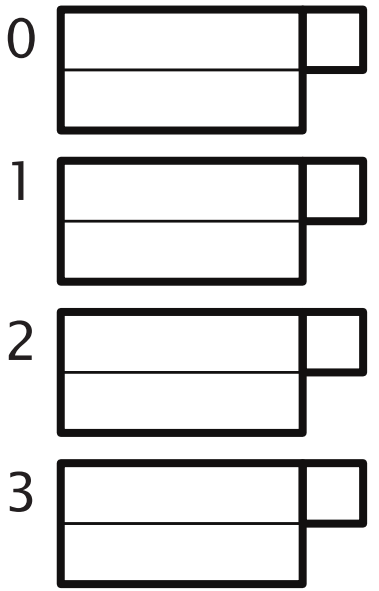
Buckets

Do we keep keys sorted?

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

Hashing example

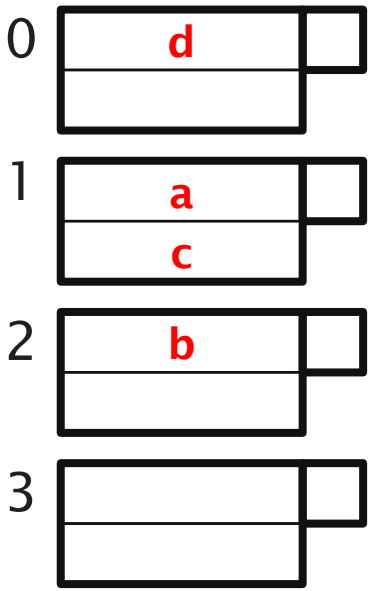
Two records per bucket



Hashing example

Insert a, b, c, d

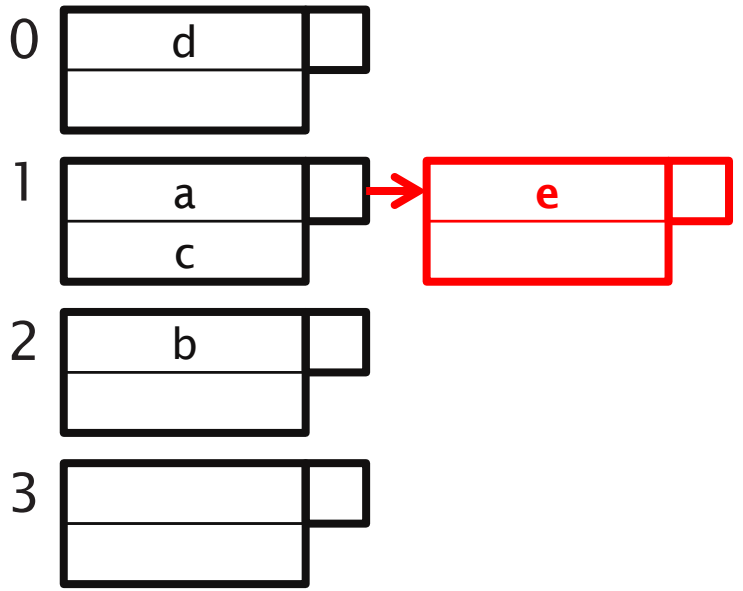
- h(a) = 1
- h(b) = 2
- h(c) = 1
- h(d) = 0



Hashing example: Overflow

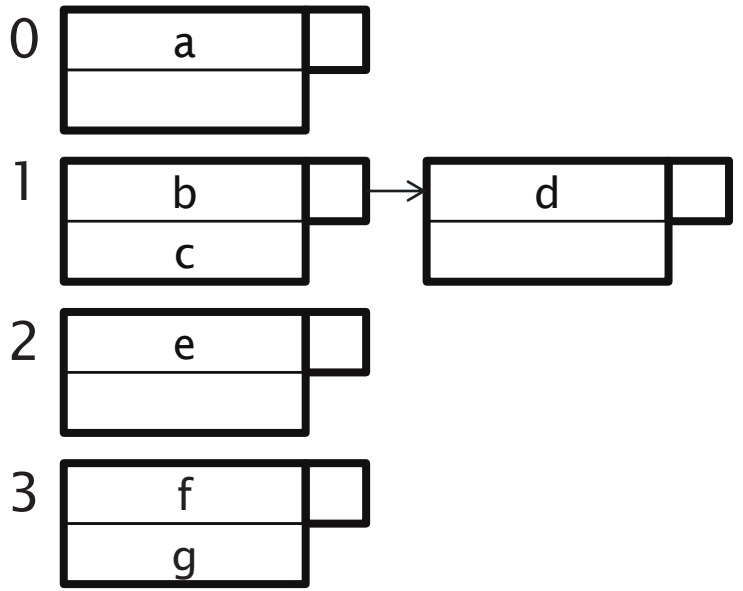
Insert e

- h(e) = 1



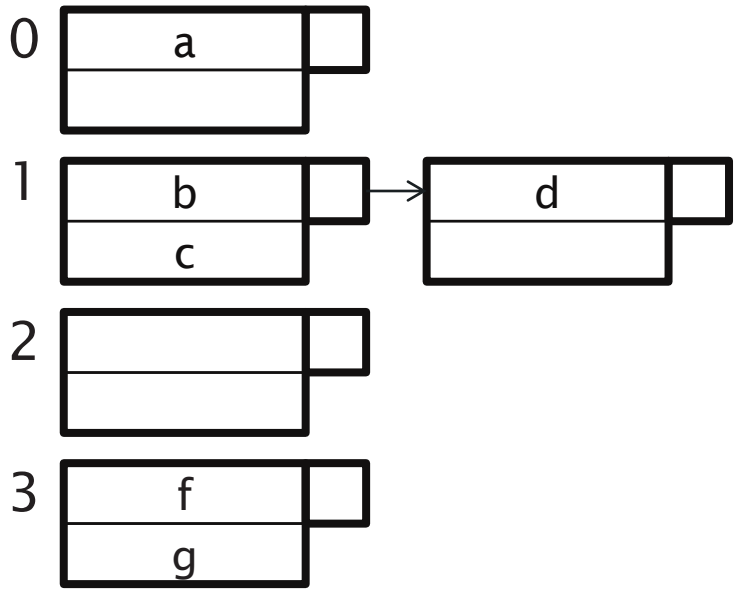
Hashing example: Deletion

Delete e



Hashing example: Deletion

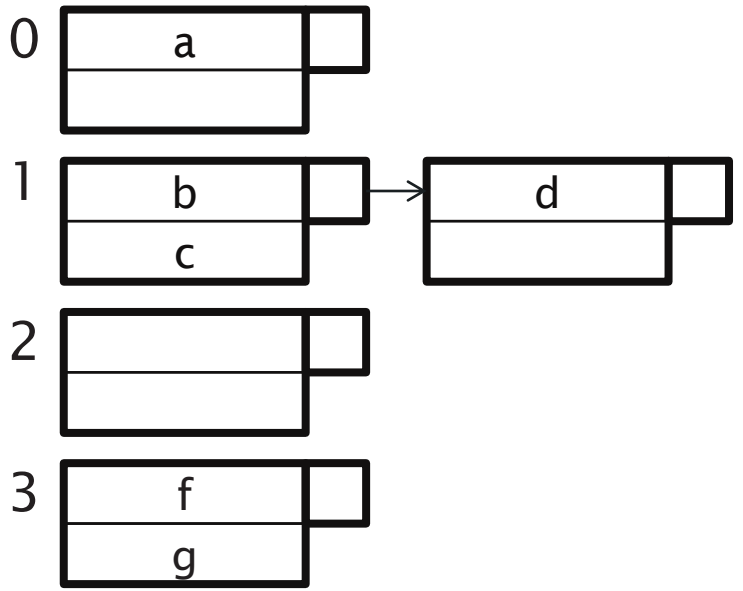
Delete e



Hashing example: Deletion

Delete f

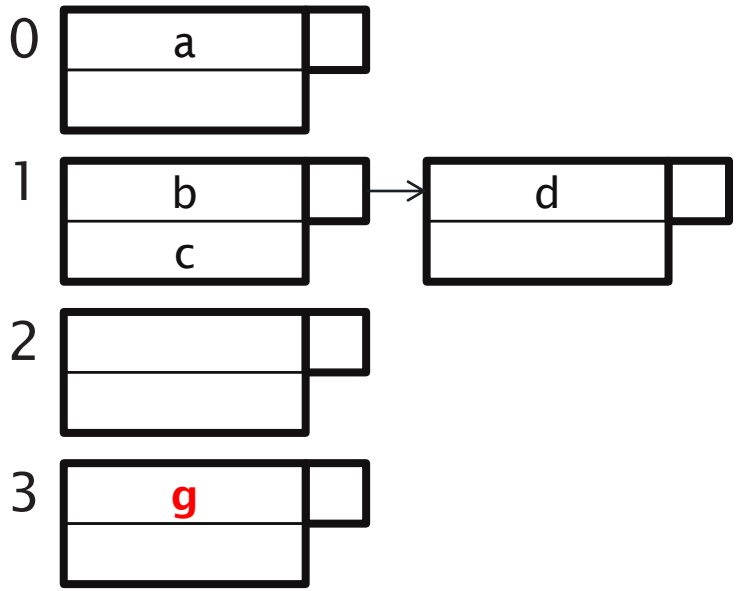
(move g up)



Hashing example: Deletion

Delete f

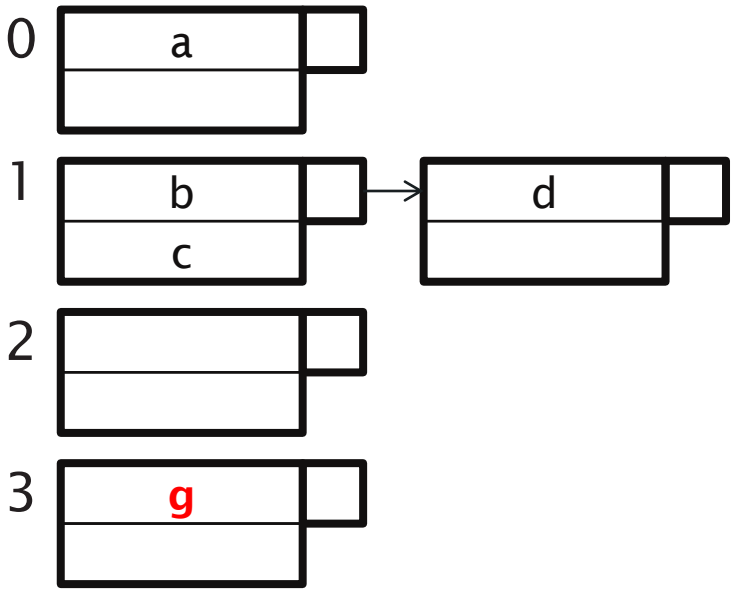
(move g up)



Hashing example: Deletion

Delete f

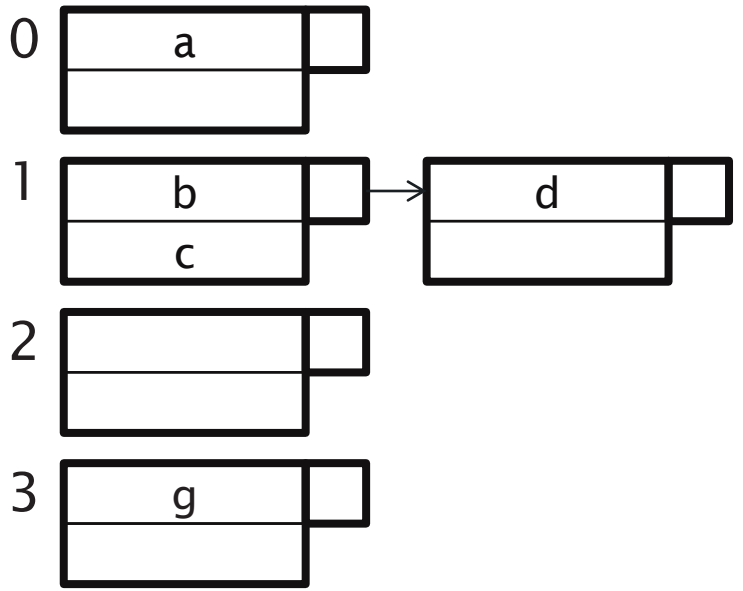
(move g up)



Hashing example: Deletion

Delete c

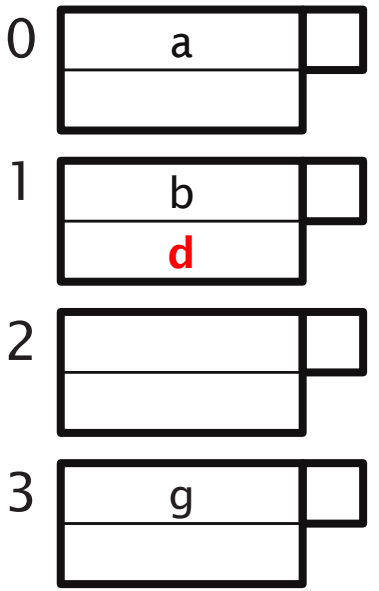
(move d from overflow block)



Hashing example: Deletion

Delete c

(move d from overflow block)



University of Southampton

Rule of thumb:

Space utilisation should be between 50% and 80%

Utilisation = #keys used / total #keys 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

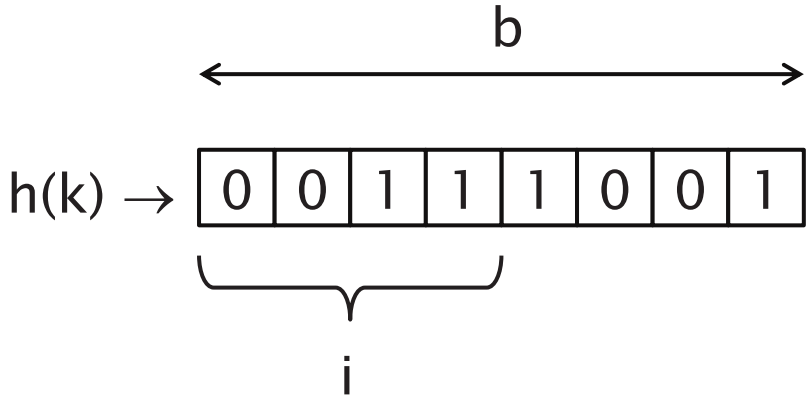
Dynamic hashing

- Extensible
- Linear

Extensible hashing

Combines two ideas:

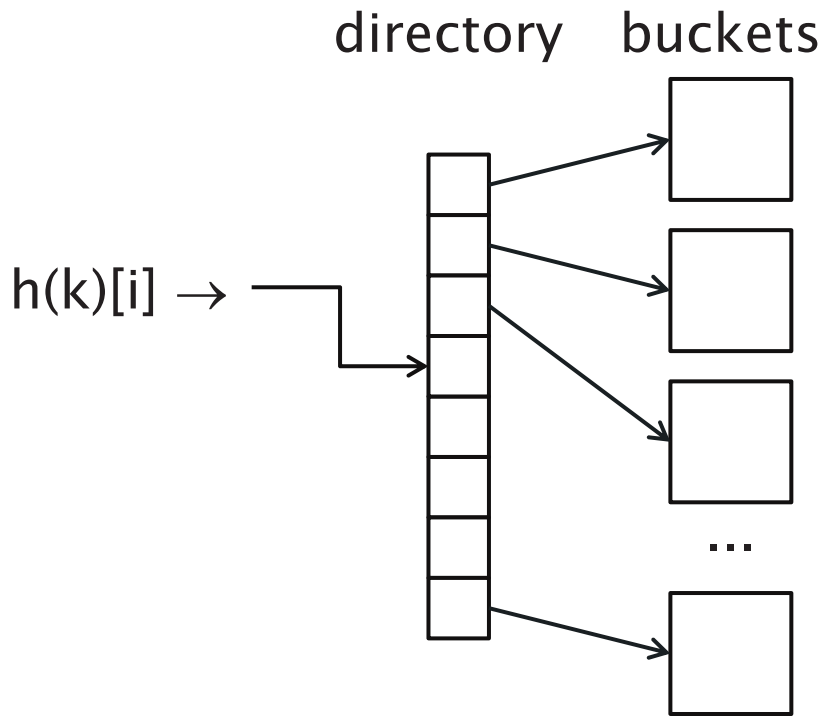
1. Use i of b bits output by hash function, where i grows over time



Extensible hashing

Combines two ideas:

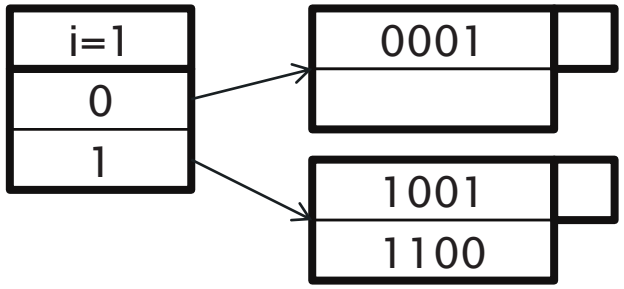
1. Use i of b bits output by hash function, where i grows over time
2. Use a directory



Example

h(k) gives 4 bits

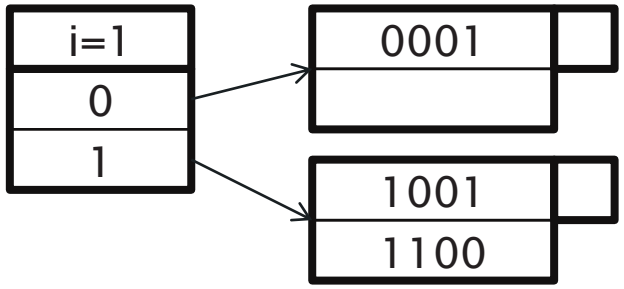
2 keys/bucket



Example

Insert 1010

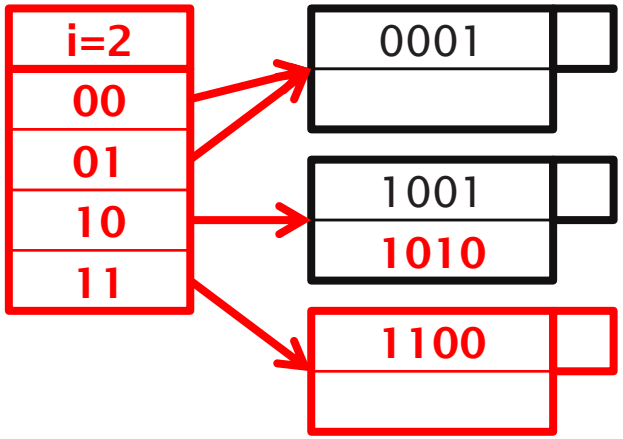
- Bucket overfull



Example

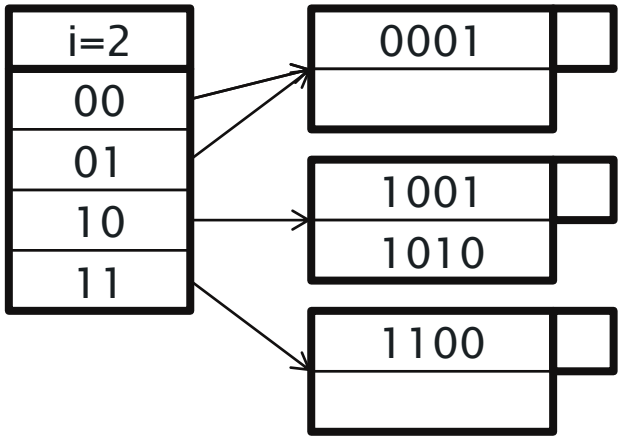
Insert 1010

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



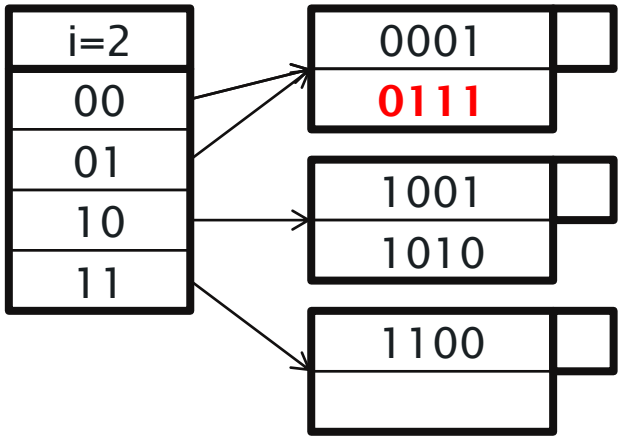
Example

Insert 0111



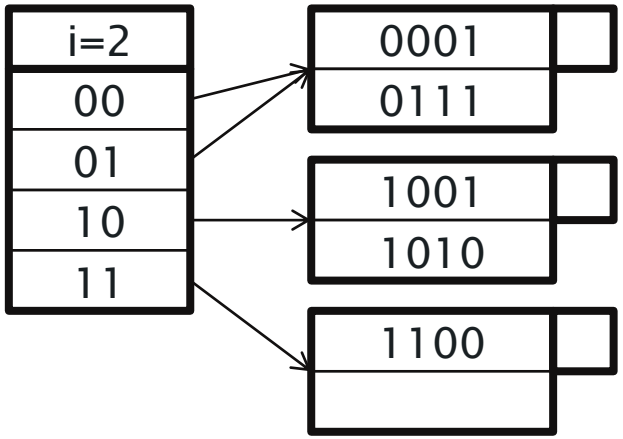
Example

Insert 0111



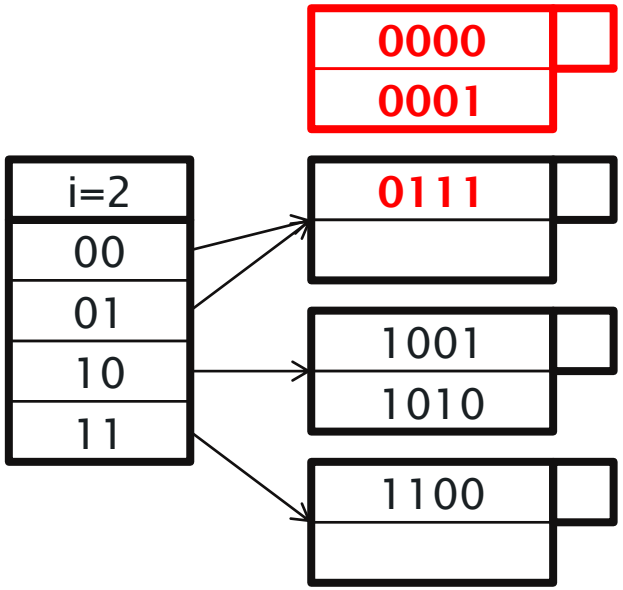
Example

Insert 0000



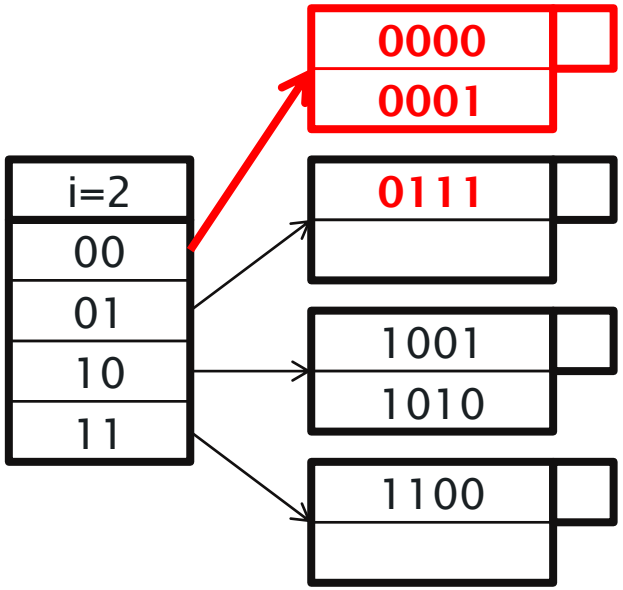
Example

Insert 0000



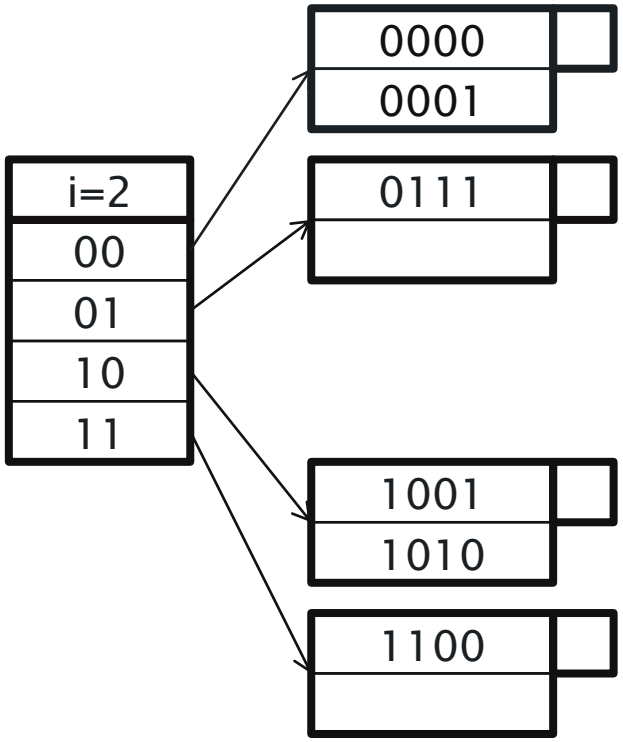
Example

Insert 0000



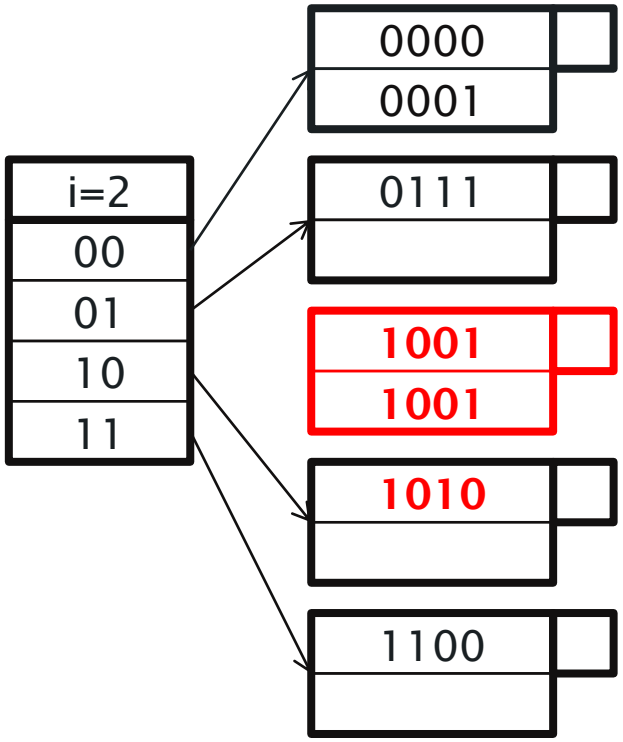
Example

Insert 1001



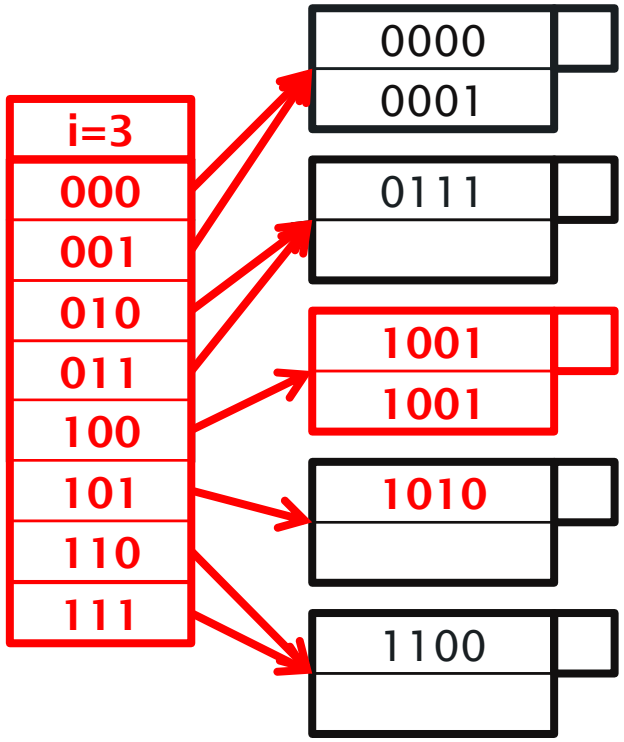
Example

Insert 1001



Example

Insert 1001



University of Southampton

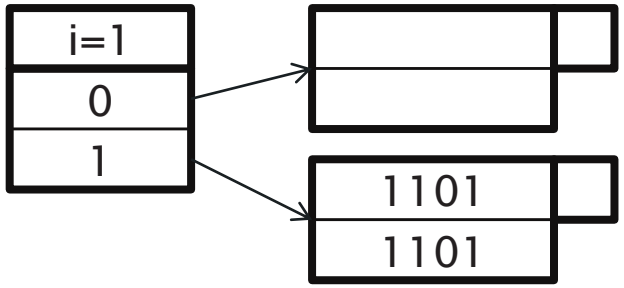
Extensible hashing: deletion

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

Overflow chains

Example: many records with duplicate keys

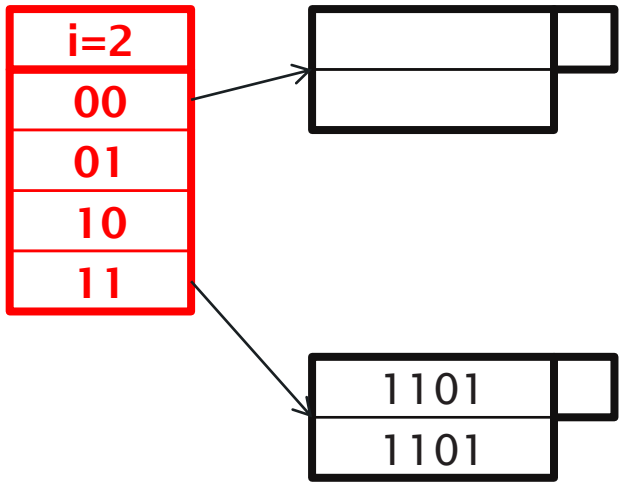
- Insert 1100



Overflow chains

Example: many records with duplicate keys

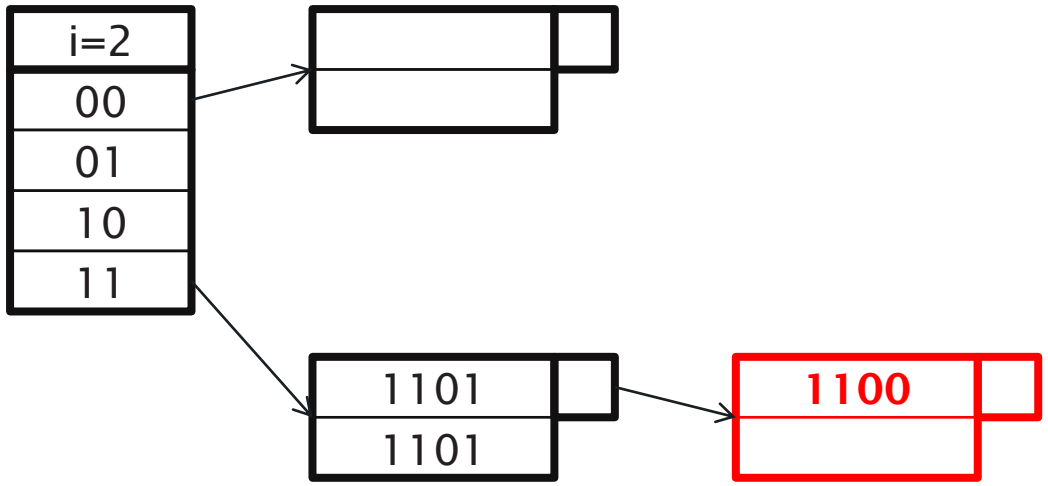
- Insert 1100



Overflow chains

Example: many records with duplicate keys

- Insert 1100
- Add overflow block



University of Southampton

Summary

Pro

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

Con

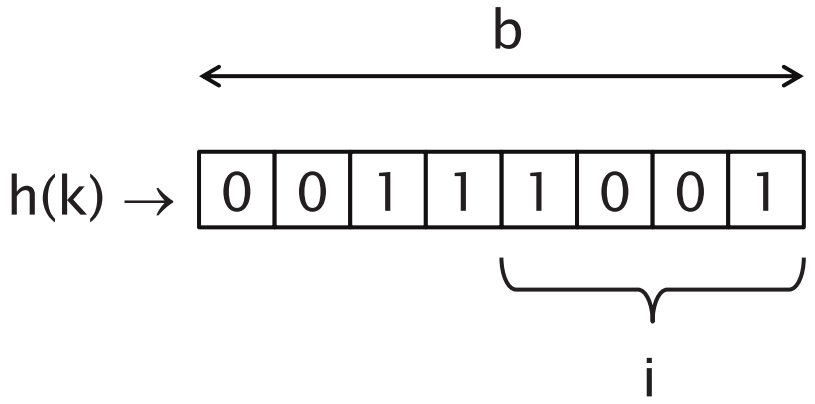
- Indirection
  - not bad if directory in memory
- Directory doubles in size
  - now it fits in memory, now it doesn't
  - suddenly increase in disk accesses!

Linear hashing

Another dynamic hashing scheme

Combines two ideas

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

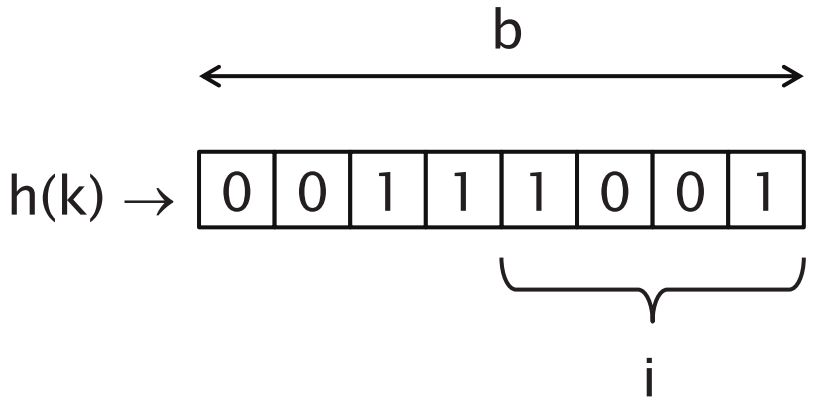


Linear hashing

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

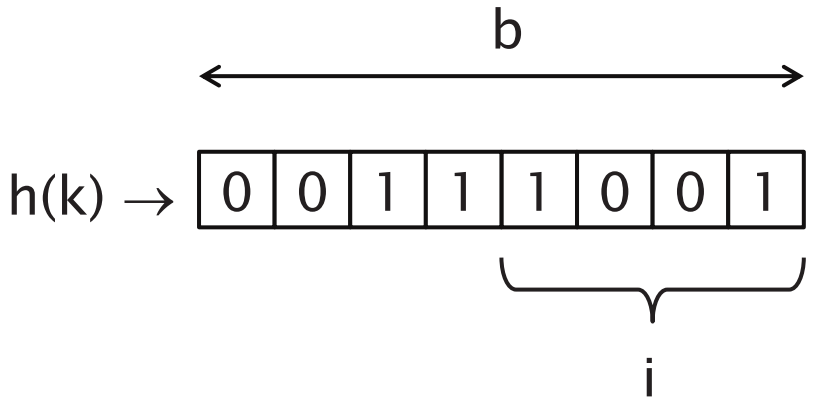
Another dynamic hashing scheme

Combines two ideas

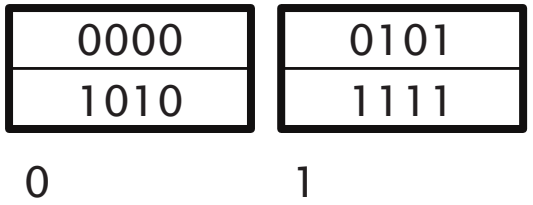
1. Use i least significant bits of hash, where i grows over time
2. Hash file grows incrementally and linearly
   (unlike extensible hash file, which periodically doubles)

Lookup rule:

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

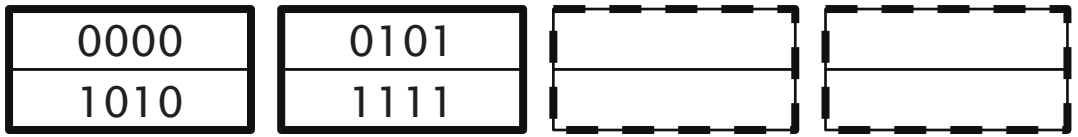


Example: b=4 bits, i=1, 2 keys/bucket



m = max used bucket = 1

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



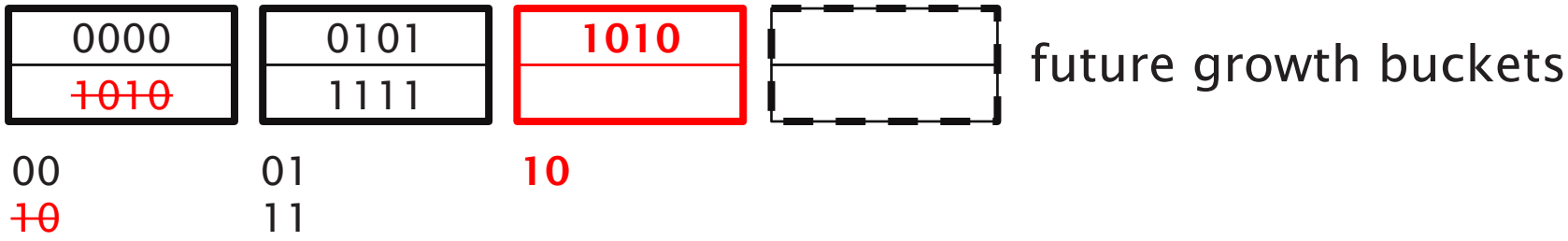
future growth buckets

00
10

01
11

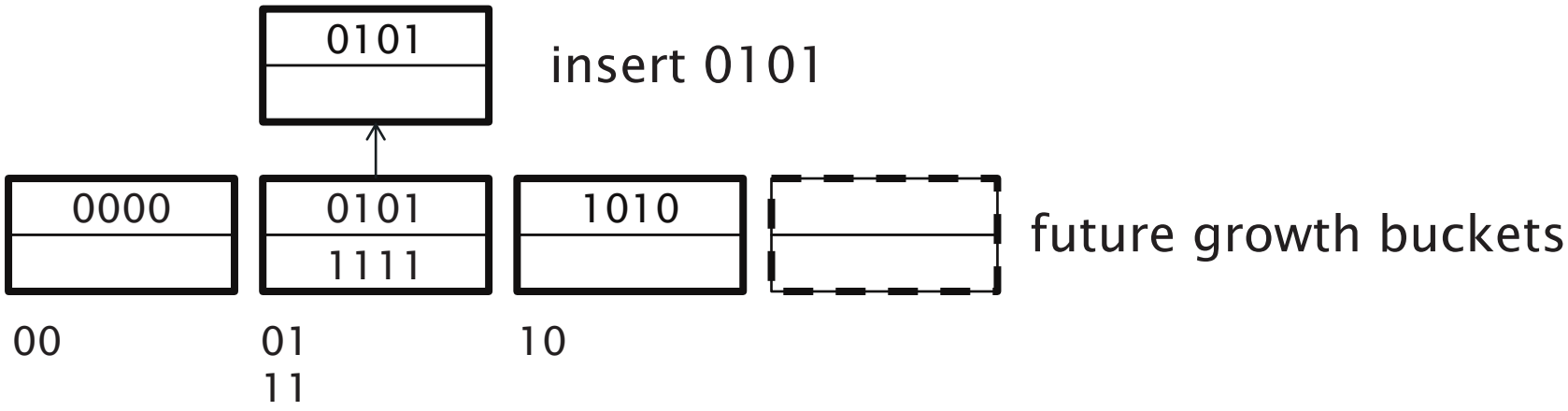
m = max used bucket = 01

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



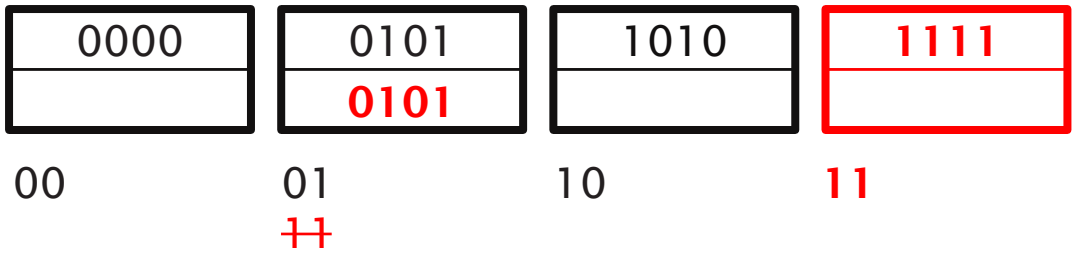
m = max used bucket = **10**

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



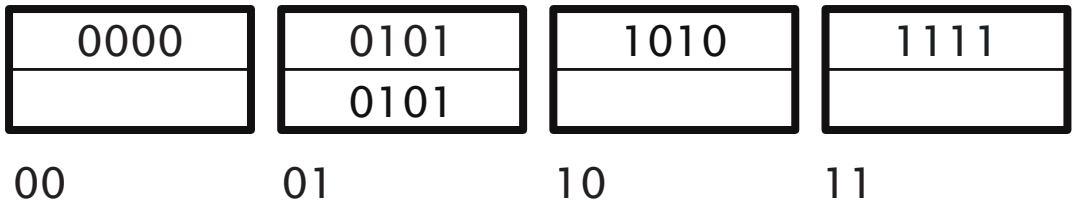
m = max used bucket = 10

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



m = max used bucket = **11**

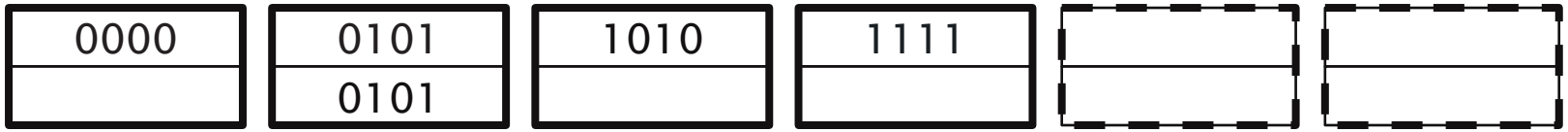
Example: further growth



future growth buckets

m = max used bucket = 11

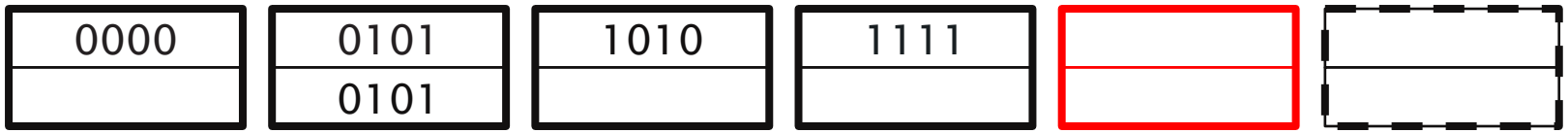
Example: i=3



000 001 010 011
100 101 110 111

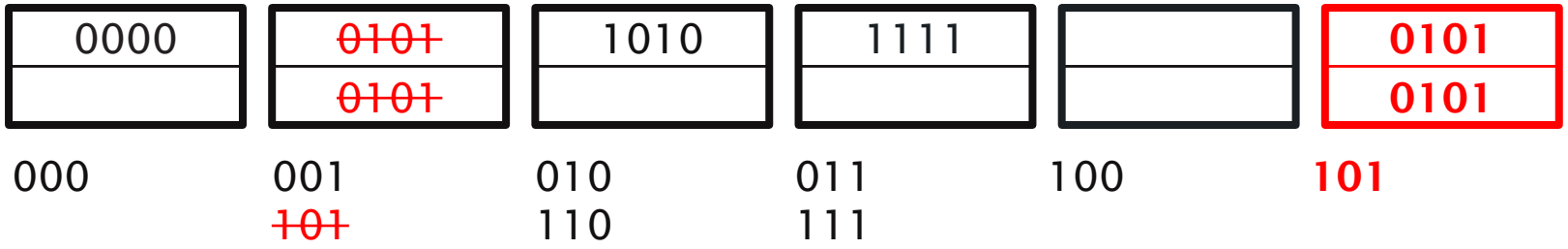
m = max used bucket = 11

Example: i=3



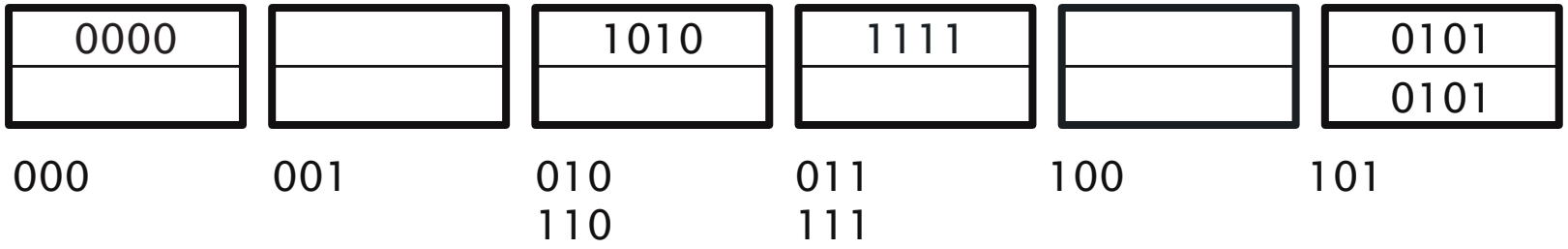
m = max used bucket = **100**

Example: i=3



m = max used bucket = **101**

Example: i=3



m = max used bucket = 101

University of Southampton

When do we expand file?

Keep track of utilisation

U = #used slots / total #slots

If U > threshold, then increase m (and maybe i)

University of Southampton

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

University of Southampton

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
```

University of Southampton

Further Reading

University of Southampton

Further Reading

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

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

University of Southampton

Next Lecture:
Multidimensional Access Structures