Physical Database Design

Basic considerations:

Data independence:

- The user should be insulated from physical database design.
- It is perhaps acceptable (desirable) to allow the user to make suggestions for things such as which attributes should be indexed for faster access.

Types of access needed:

- Fast retrieval based upon specific keys (suggests hashing strategies)
- Processing tuples in order based upon a key (suggests a sequential structure)
- Partial-match queries: Requests based upon several attributes (suggests multi-dimensional data structures)

Overriding concern:

- Data are stored on external devices (disk)
- Access is much slower (1000 to 10000 times) than access to main memory.
- For efficiency, the number of disk accesses must be minimized.

High-level implementation strategy:

- Use the file system of the OS.
- Use a DBMS-specific partition.
- Roll-your-own file system.

Basic concepts of disk access:

Physical concepts:
- Platter
- Head
- Cylinder
- Track
- Sector

- CHS addressing
- Logical sector addressing

Microcomputer disk interfaces:
- IDE / ATA
- EIDE
- ATA4 / Ultra DMA-33
- ATA5 / ATA-66
- ATA6 / ATA-100
- ATA-133, ATA-150
- Serial ATA
- SCSI II
- Wide SCSI II
- Ultra SCSI (various flavors)
- Serial-attached SCSI
- Fiber channel

Time parameters:
- Seek time
- Rotational latency
- Block transfer time
- Bulk transfer rate

A high-end SCSI drive:

Seagate Cheetah 15K.5:
ST3300655LW/LC/FC/SS (300 GByte)
ST3146655LW/LC/FC/SS (146.8 GByte)
ST373655LW/LC/FC/SS (73.4 Gbyte)

Specification	Value
Formatted capacity	300 / 146.8 / 73.4 Gbyte.
Interface	Ultra320 SCSI 4 Gbit/sec fibre channel 3 Gbit/sec serial-attached SCSI
Rotational speed	15000 RPM
Seek time (avg. read)	3.5 msec.
Seek time (avg. write)	4.0 msec.
Rotational latency (avg.)	2.0 msec.
Platters	4 / 2 / 1
Heads	8 / 4 /2
Nonrecoverable error rate	1 sector / 10^{16} bits
Acoustics idle (bels -- sound power)	3.0 - 3.6

A high-quality SATA drive for a PC:

Seagate Barracuda ES:
- ST3750640NS (750 GByte)
- ST3500630NS (500 GByte)
- ST3400620NS (400 Gbyte)
- ST3320620NS / ST3320820NS (320 Gbyte)
- ST3250620NS / ST3250820NS (250 Gbyte)

Specification	Value
Formatted capacity	750/500/400/320/200 Gbyte.
Interface	SATA 3Gbit/sec.
Rotational speed	7200 RPM
Seek time (avg. read)	8.5 msec.
Seek time (avg. write)	9.5 msec.
Rotational latency (avg.)	4.16 msec.
Platters	?
Heads	?
Nonrecoverable error rate	1 sector / 10^{14} bits
Acoustics idle (bels -- sound power)	2.5 – 2.7

RAID:
Redundant Array of Inexpensive Disks
Redundant Array of Independent Disks

Mulitiple disks are used in RAID configurations in two main ways:

Striping (RAID 0): The data are distributed in blocks over several disks in order to speed up access.

- Disadvantage: If one disk in the array fails, all data are lost.
- RAID 2 and RAID 3 are variants with different types of striping.

Redundancy via mirroring (RAID 1): The data are replicated over several disks in order to increase reliability in case of failure.

- Mirroring provides first and foremost a reliability advantage.
- It can also provide a performance advantage for reading, with some loss of perfomance in writing.
- Not all RAID controllers provide such performance advantage.

Combination: A number of RAID configurations provide both striping and redundancy:

- RAID 0+1: A mirror of stripes
- RAID 1+0: A stripe of mirrors
- RAID 5: Block-level striping plus redundancy via parity data
  - In case of a failure of a disk, all data can be recovered from the remaining disks using the parity data.
- There are numerous other configurations, each with its advantages and disadvantages.

RAID may be implemented:

- in the hardware (via a special disk controller);
- in the software (*e.g.*, Linux kernel).
  - It is difficult to boot from a software RAID partition.

Some Basic Concepts:

- Field:
  - Smallest unit of logical storage
  - Typically corresponds to one column of a relation.
  - Length has two dimensions:
    - Fixed vs. variable
    - Logical vs. physical
    - In Access:
      - Variable logical length allowed.
      - All fields are fixed-length physically.

- Record:
  - A collection of fields (similar to PL notion).

  - Physical record: Stored as a single accessible unit.

  - Logical record:
    - Corresponds to a logical notion in the data model (*e.g.* tuple)
    - May or may not be stored as a physical record.
    - Length: (see Figure 13.5; (5.7 in 3rd Ed.))
      - Fixed-length records
      - Variable-length records
      - May arise in two distinct ways:
        - Variable-length fields
        - Variable number of fields

- Blocking factor:
  - A *block* is the unit of data which is transferred in a single disk access.
  - The *blocking factor* is the number of records stored in a single block.

    B = size of block
    R = size of record
    bfr = $\lfloor B/R \rfloor$

- Block organization: (Figure 13.6 (5.8 in 3rd Ed.))

  - Unspanned: Every record is contained in a single block:
    - Unused space per block = B – (bfr $\times$ R)

  - Spanned: To avoid wasted space, a record may be split over blocks.
    - Spanning makes retrieval slower, however.

Processing Needs:

- A good physical database design must be based upon perceived processing needs.

Access:

- *Random access:* Retrieve records individually based upon the value of a key.
- *Batch access:* Retrieve and process all of the records, in any order.
- *Sequential access:* Retrieve all of the records, in order, based upon the values of a selected key.
    - *Primary key only:* Access based upon one key only.
    - *Multi*-key: Multiple access requirements, based upon different keys.
- *Range access:* Retrieve all of the records which satisfy certain range constraints on one or more key attributes.

Basic Organizations:

- Heap:
  - Records are stored without any logical regard to order.
  - Order is typically the “insertion order.”
  - Access:
    - Linear search, block-by-block
    - Via secondary indices (later)
  - Insertion: very easy
  - Deletion:
    - Physical removal is very slow.
    - Marking of deleted records may also be used, but periodic garbage collection is necessary.

This organization is seldom used in a DBMS without further structural support (*e.g.* indices).

Sequential: (Figure 13.7 (5.9 in 3rd Ed.))

- Records are stored in order, based upon some field used as a key.
  - Block-by-block +
  - Within each block
- Access:
  - Via binary search.
  - Still need one disk access per “division” step in the binary search.
- Insertion:
  - Very slow.
  - May be partially remedied with an *overflow file.*
- Deletion:
  - May use the same strategies as for a heap.
  - Same advantages and disadvantages.

This organization is seldom used in a DBMS without further structural support (*e.g.* indices).

Indexed Organizations:

- Direct (Figure 14.1 (6.1 in 3rd Ed.))
  - Records are accessed based upon the direct value of one or more keys.
  - Advantage: Rapid sequential processing is possible.
  - Disadvantage:
    - Relatively large indexing structure
    - Nonuniform distribution may require frequent reorganization of the index.

- Hashed
  - A key is transformed to another value, and the record is stored based upon that computed value:
  - Advantages:
    - Relatively small indexing structure
    - A well-chosen transformation (hashing function) can result in a very uniform distribution of records within the storage space, even when the key values are very clustered.
  - Random and batch access times are improved.
  - Disadvantage:
    - The capability for rapid sequential processing is lost.
  - Special hashing techniques exists for structures on secondary storage.

We will look at direct indexing first, and then return to hashing.

Indices:

- Primary:
  - A *primary index* is one which is tied to the physical order of the records. (Figure 14.1 (6.1 in 3rd Ed.))

- Secondary:
  - A *secondary index* is one which is not tied to the physical order of the records.

- Density

  - Dense:

    - A *dense index* is one which has a distinct index entry for each record.
      - Secondary indices are almost always dense. (Figure 14.4 (6.4 in 3rd Ed.))

  - Nondense:
    - In a *nondense index* , a single index entry may reference many records. (Figure 14.1 (6.1 in 3rd Ed.))
      - Primary indices may be nondense.

- Clustering:

  - For a field which does not have a distinct value for each record, a *clustering index* may be used. (Figures 14.2 and 14.3 (6.2, 6.3 in 3rd Ed.))

- Direct vs. indirect:
  - In a *direct index*, the index entry points directly to the associated record(s).
  - In an *indirect index*, the index entry points to a (block of) pointer(s) to the associated record(s). (Figure 14.5 (6.5 in 3rd Ed.))
    - Advantages:
      - Ease of implementation of non-dense indices.
      - Less burden during file re-organization.

- Single-level vs. multi-level:
  - The index itself may be organized as a multi-level entity (*e.g.,* a tree).
  - Advantage: more rapid search of the index.

Question: Do the analyses of access time in the text make sense?

Would you keep an index which is 1 Mb. in size on disk, and bring it into memory in 2 Kbyte. blocks?

More on this later.

Specific examples of structures which use multi-level indices:

- B-trees
- B^+-trees

We will first examine B-trees.

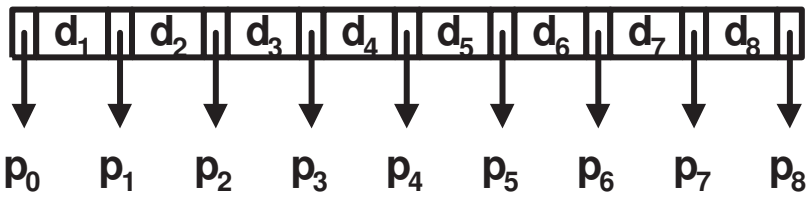
B-trees:

Recall that in a *binary search tree*, each node has one data entry and two pointers:

- Left subtree
- Right subtree

In a B-tree, this arrangement is generalized. In a *B-tree of order n*, there are n-1 data entries and n pointers.

Here is a node for a B-tree of order 8:



- The p_i's are pointers.
- The d_i's are data fields.

Note that a binary tree node is just a B-tree node of order 1.

However, B-trees have special properties not shared by all binary trees.

Because of their more general structure, the rules for maintaining B-trees are more complex than for binary trees. Here are some basics:

- Any node, except for the root, must be at least half full, in the precise sense that if the nodes contain m data fields, then at least $\lfloor m/2 \rfloor$ must be nonempty. (Round down for odd values.)

- The root must contain at least one data value; *i.e,* at least two pointers.

- Data fields are used from left to right, with unused fields empty

- The data elements in a given node are sorted.

- All pointer fields of a leaf node are null.

- For internal nodes, if a data field is not empty, then neither its left pointer nor its right pointers may not null.

- If a data field is empty, then its right pointer must be null.

- A non-null pointer identifies a subtree containing values which are between the values of the keys surrounding that pointer.

- The tree is always balanced: the length of the path from the root to a given leaf is the same as for any other leaf.

The operations are best illustrated with examples.

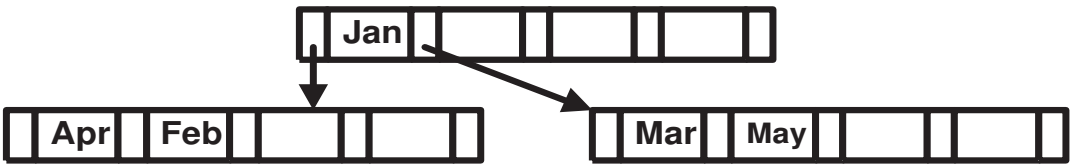
Example: Inserting the months, in chronological order. (Sort in alphabetical order.)

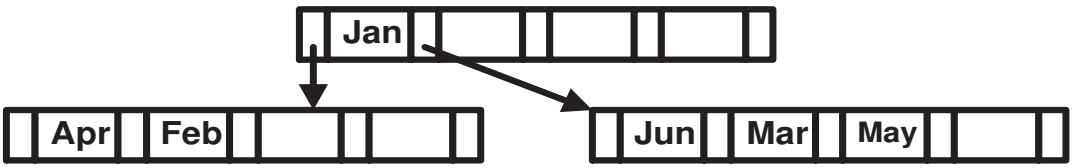
Jan

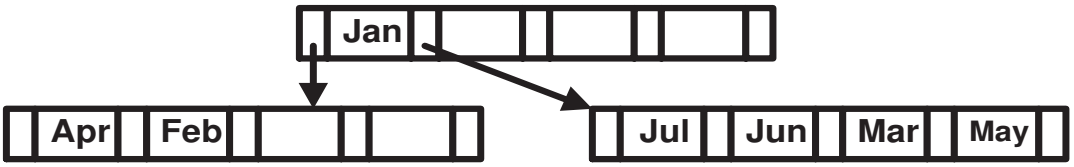
Feb
Jan

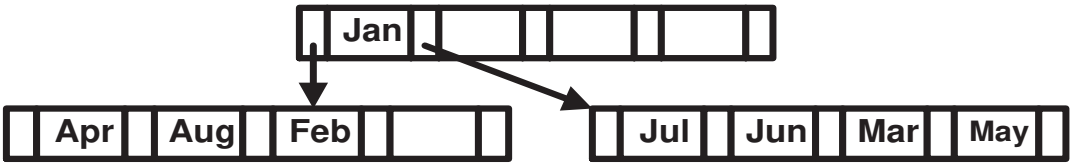
Feb
Jan
Mar

Apr
Feb
Jan
Mar


Jan
Apr
Feb
Mar
May

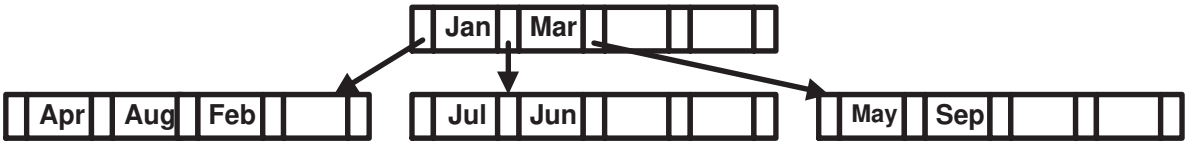

Jan
Apr
Feb
Jun
Mar
May

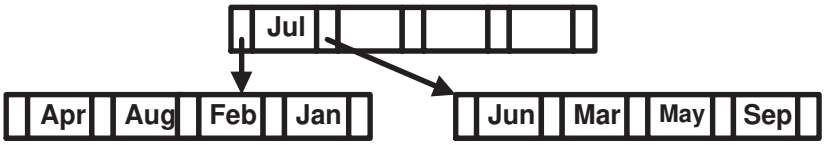

Jan
Apr
Feb
Jul
Jun
Mar
May

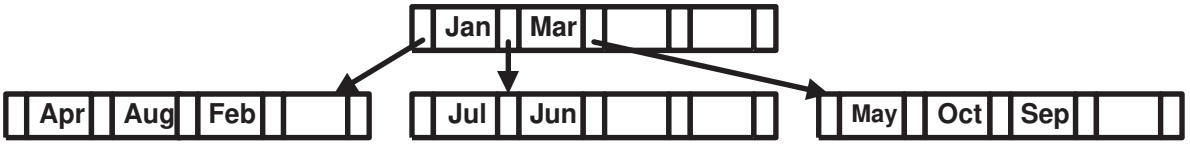

Jan
Apr
Aug
Feb
Jul
Jun
Mar
May

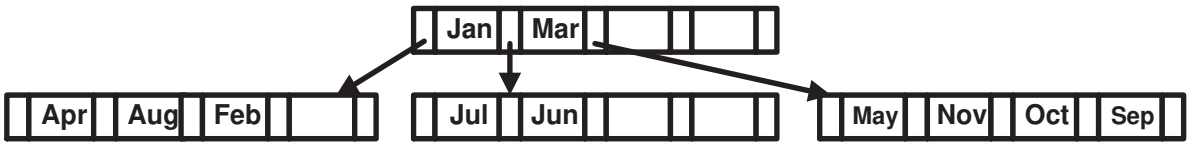
The last four months:

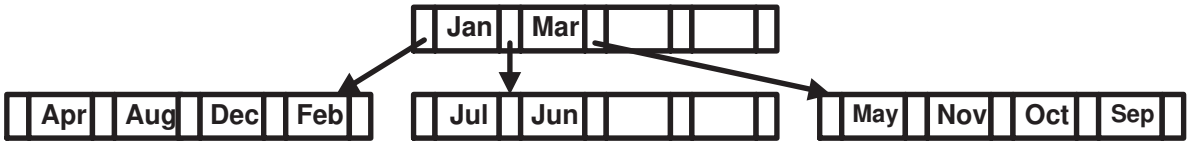
(This first two trees provide different alternatives for the insertion of Sep.)



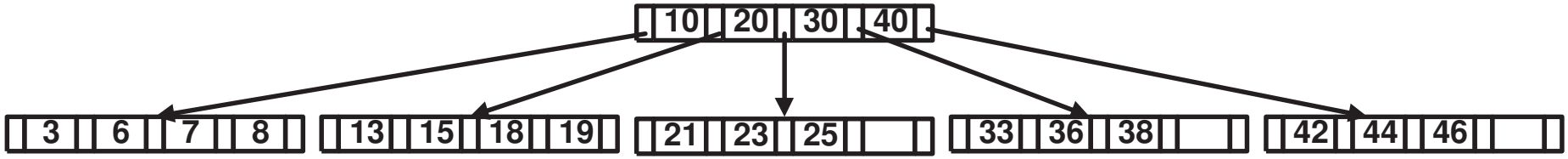




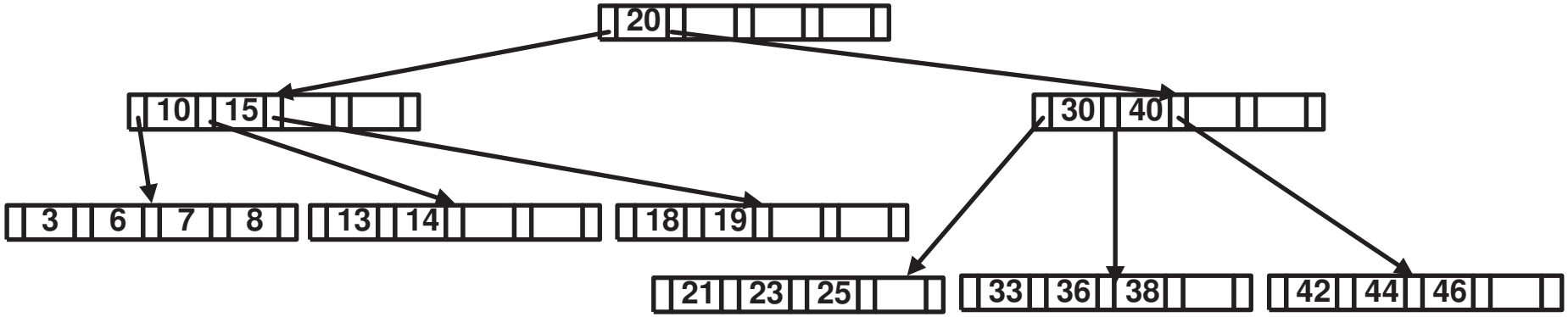




Example: Insert 14 into the following structure:



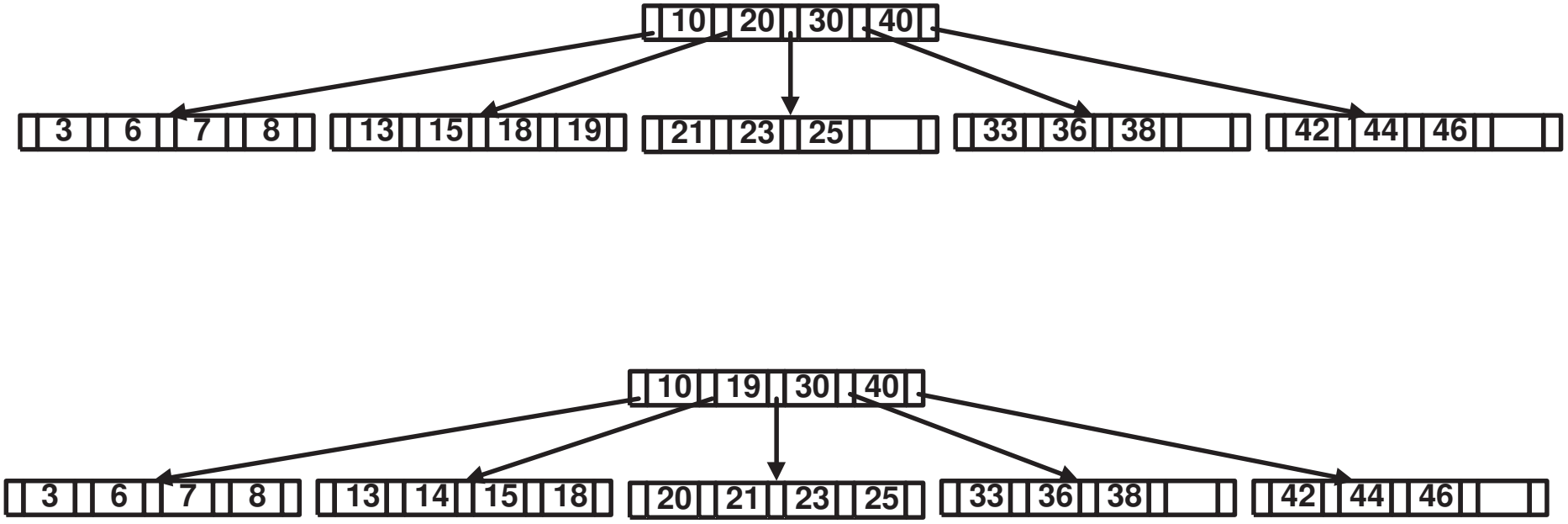
A split of the overfull node which propagates to the root is the usual solution:



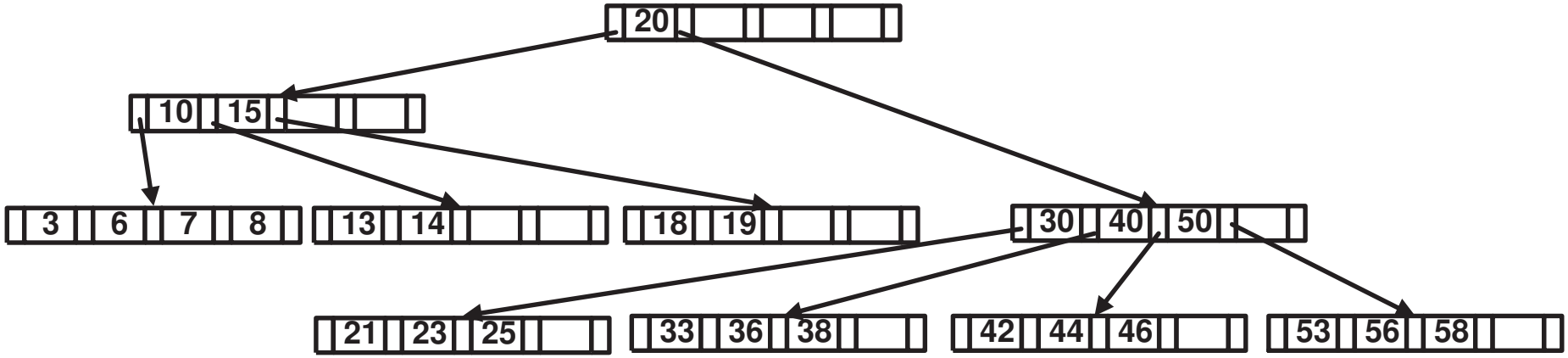
This is the *only* way in which the tree can grow in depth.

In this particular case, the insertion of 14 could also be realized by a rotation of values.

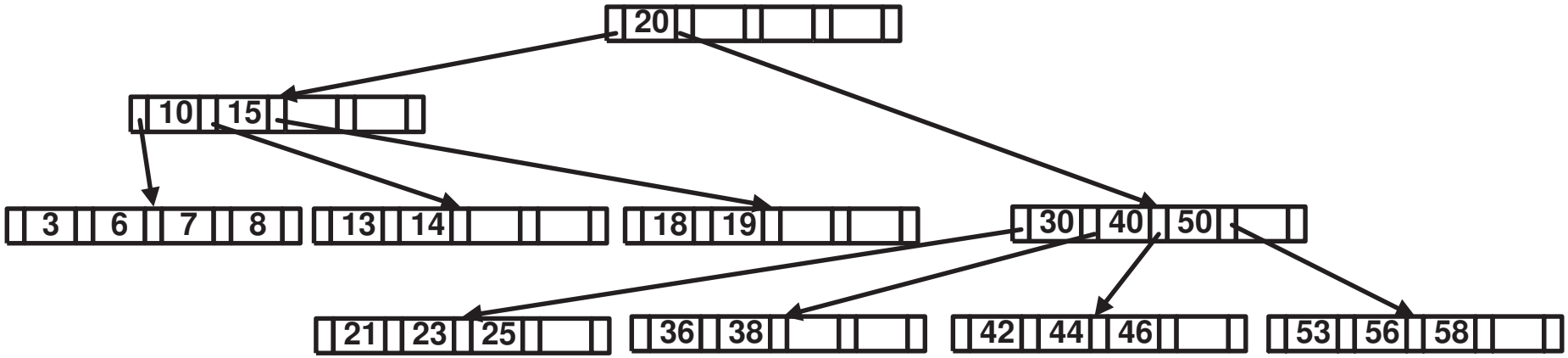
- The choice of strategy is more of a heuristic than a hard-and-fast rule.



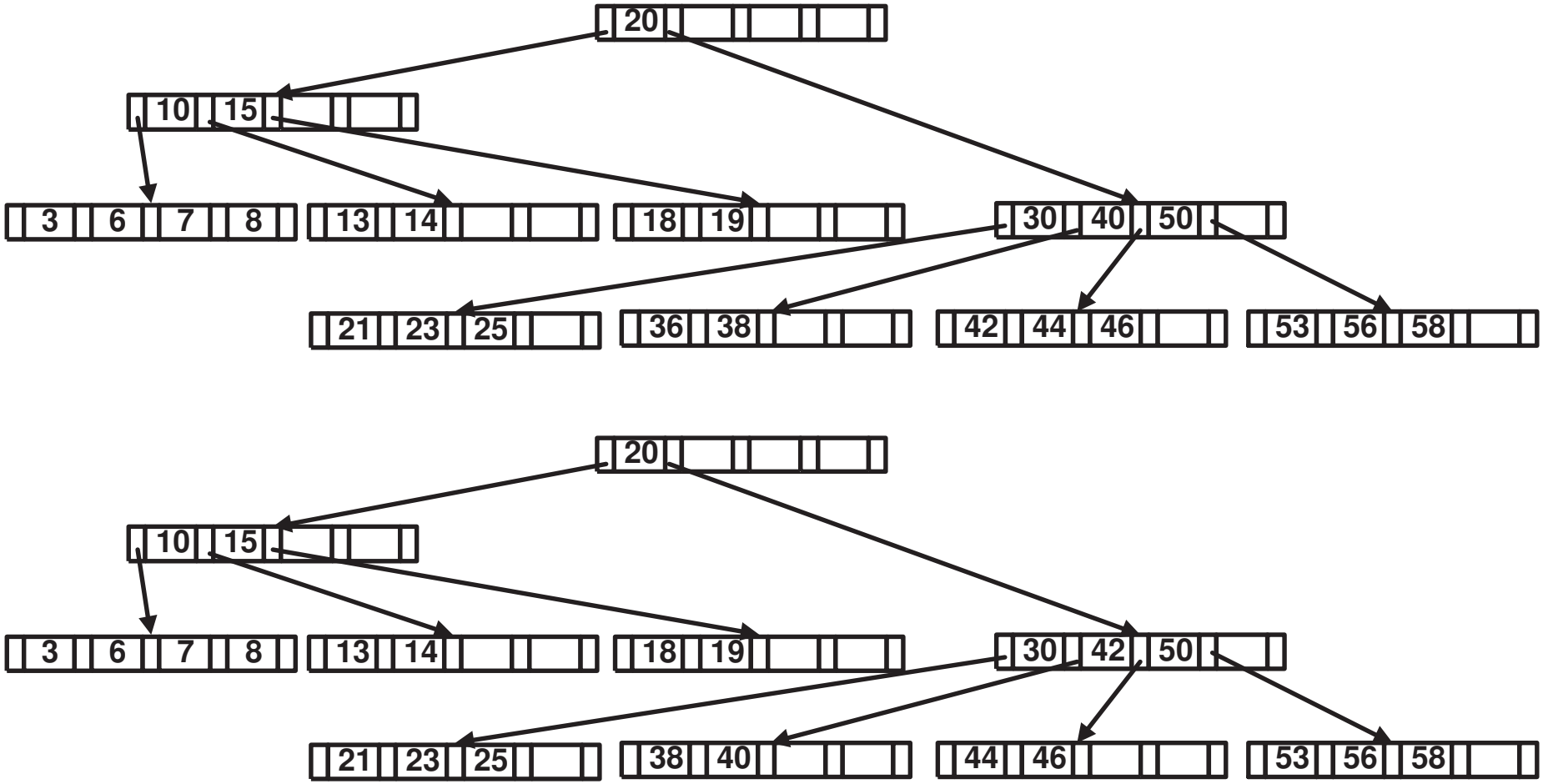
Now consider the deletion of 33 from the following tree:



It is a simple matter, since there is no underflow:



In the deletion of 36, an underflow occurs, which may be remedied with a redistribution:

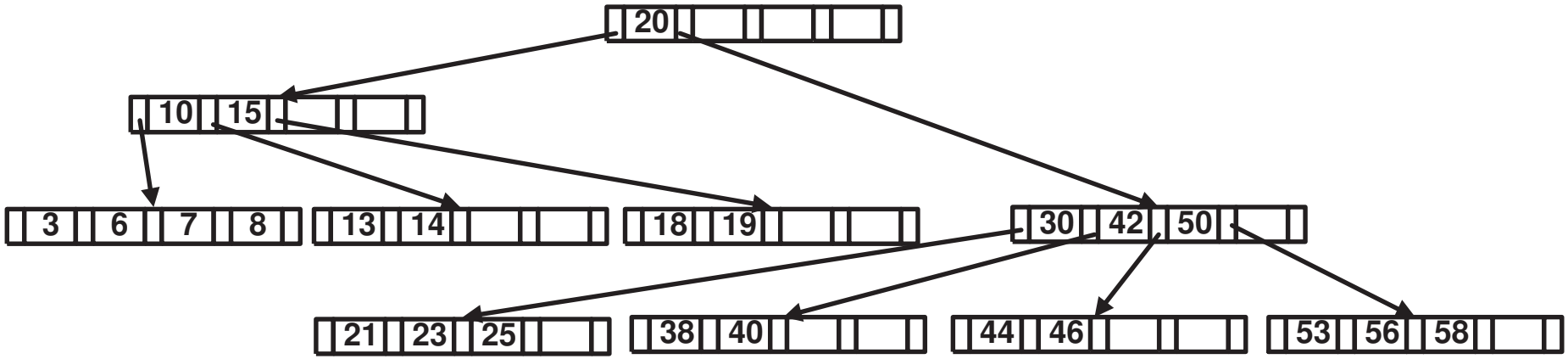


- Heuristic: When redistributing, make the number of elements in each sibling (about) the same. This will lessen the likelihood of another redistribution in the near future.

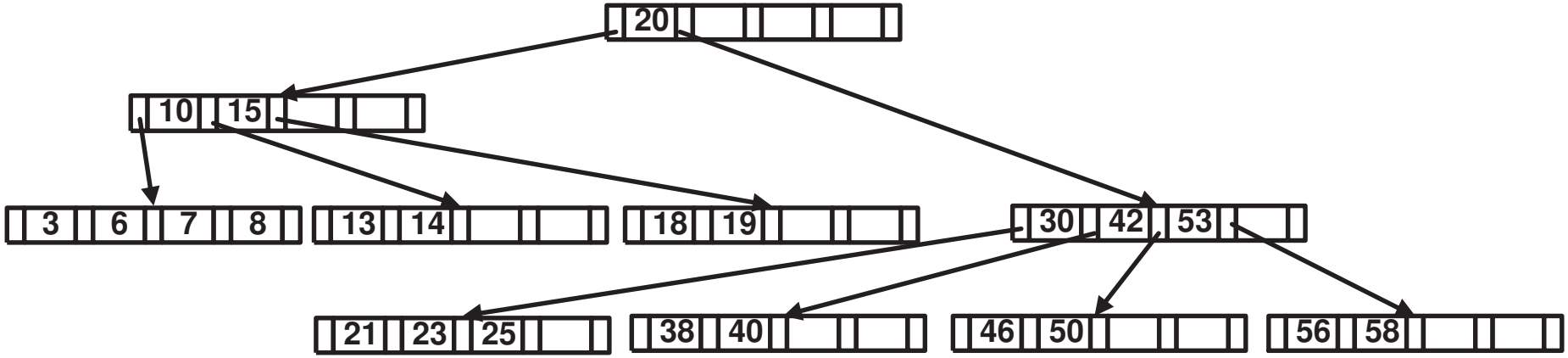
(This happens automatically in our simple example with only four data fields per node, but is far from automatic with a large number of data fields.)

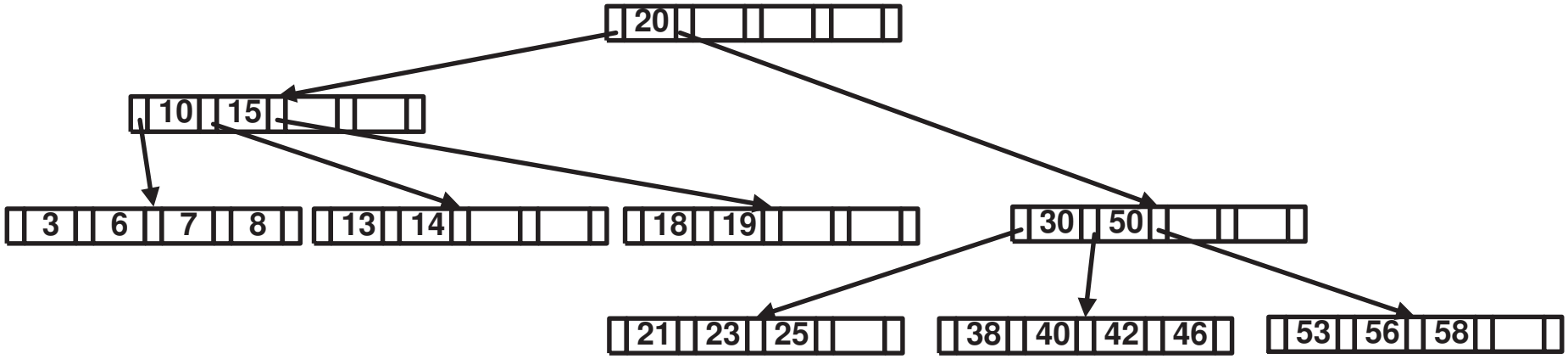
Now consider the deletion of 44. There are two possibilities:

- Combine the underfull node with the right sibling (no reduction required).
- Combine the underfull node with the left sibling (reduction required).

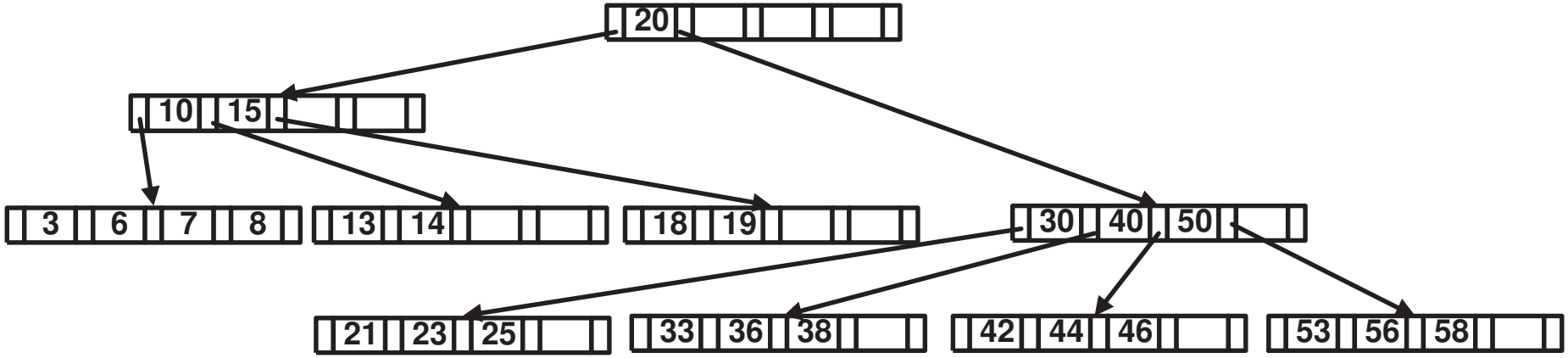


Here are the two solutions

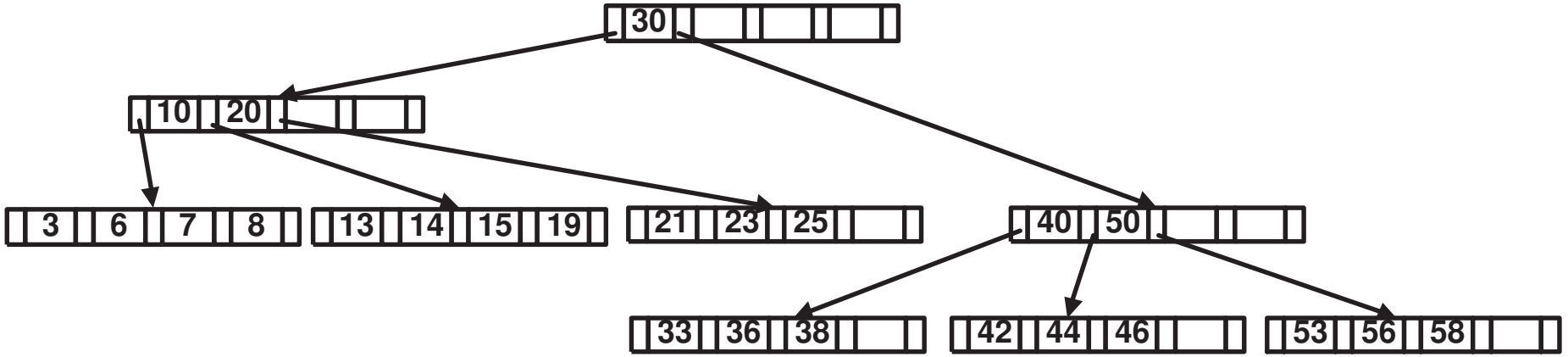




- Now consider deleting 18 from the following structure:

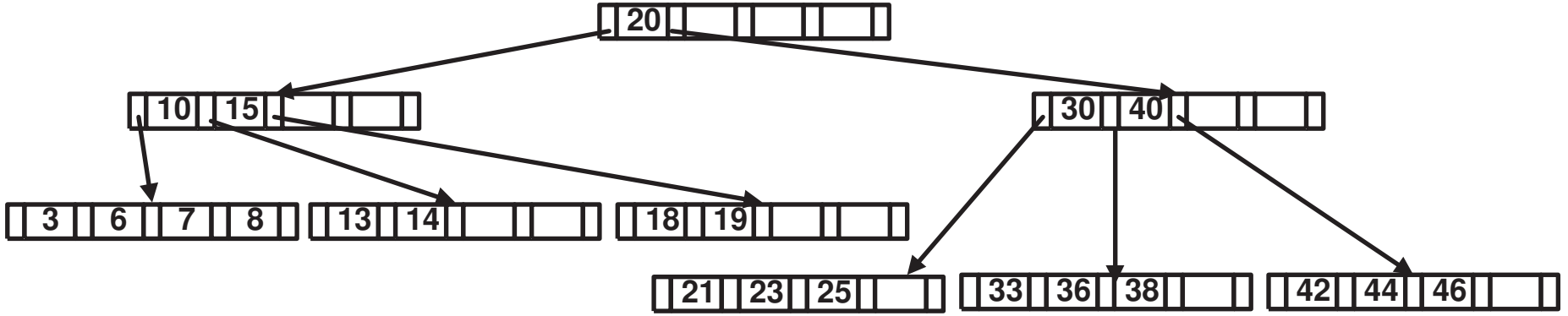


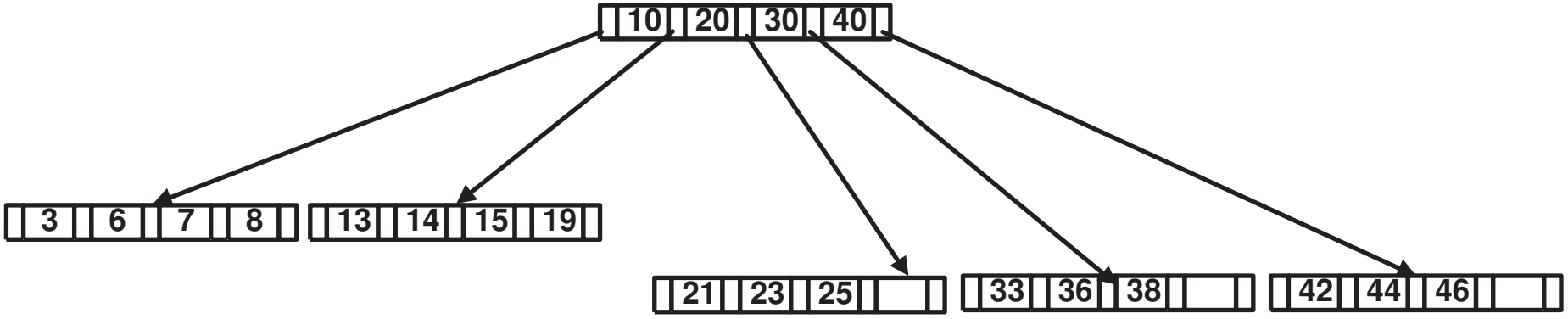
We can redistribute values one level up from the leaves:



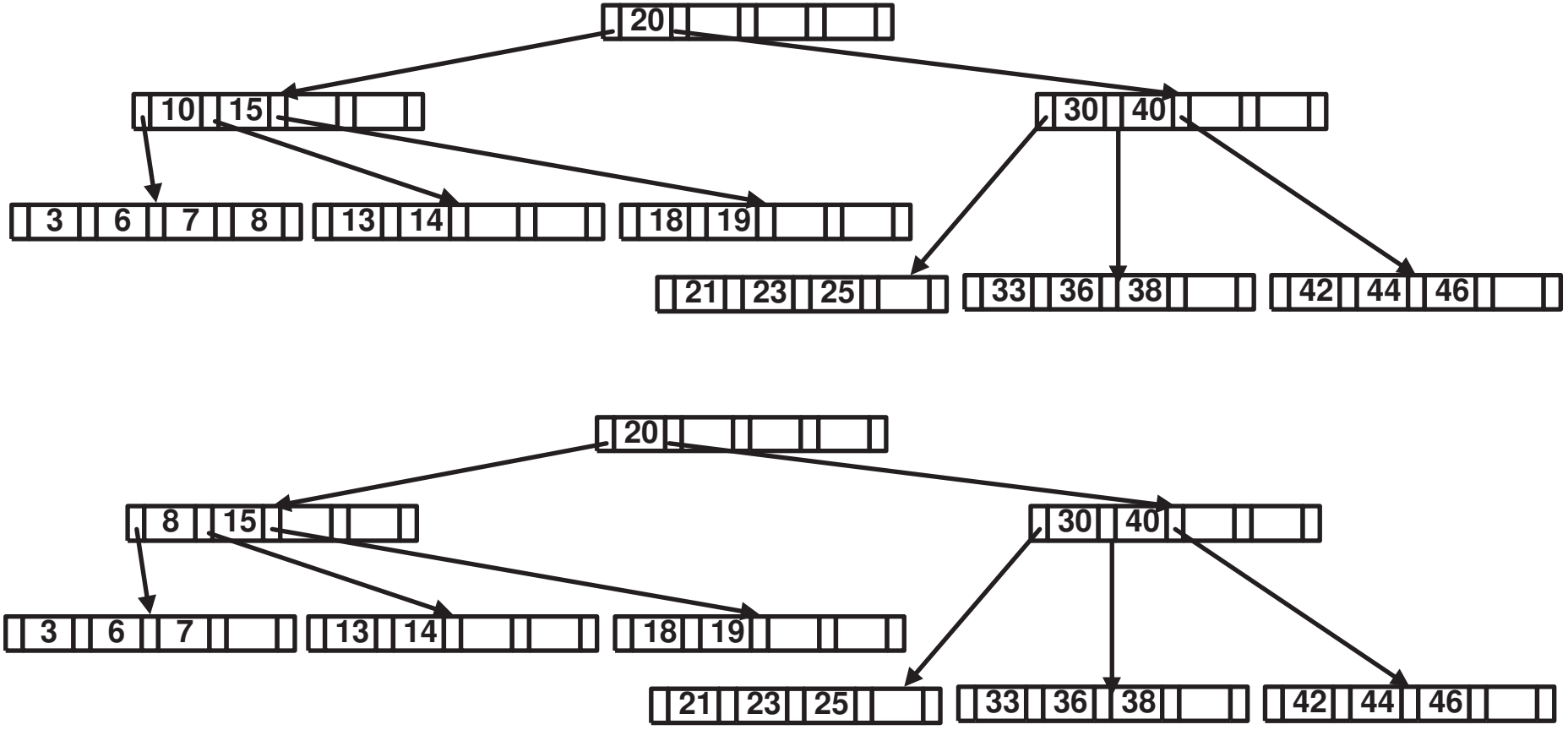
- If we delete 18 from the following tree, however, we must adjust the height.

(Unless we do a long-range re-adjustment – illustrated later.)

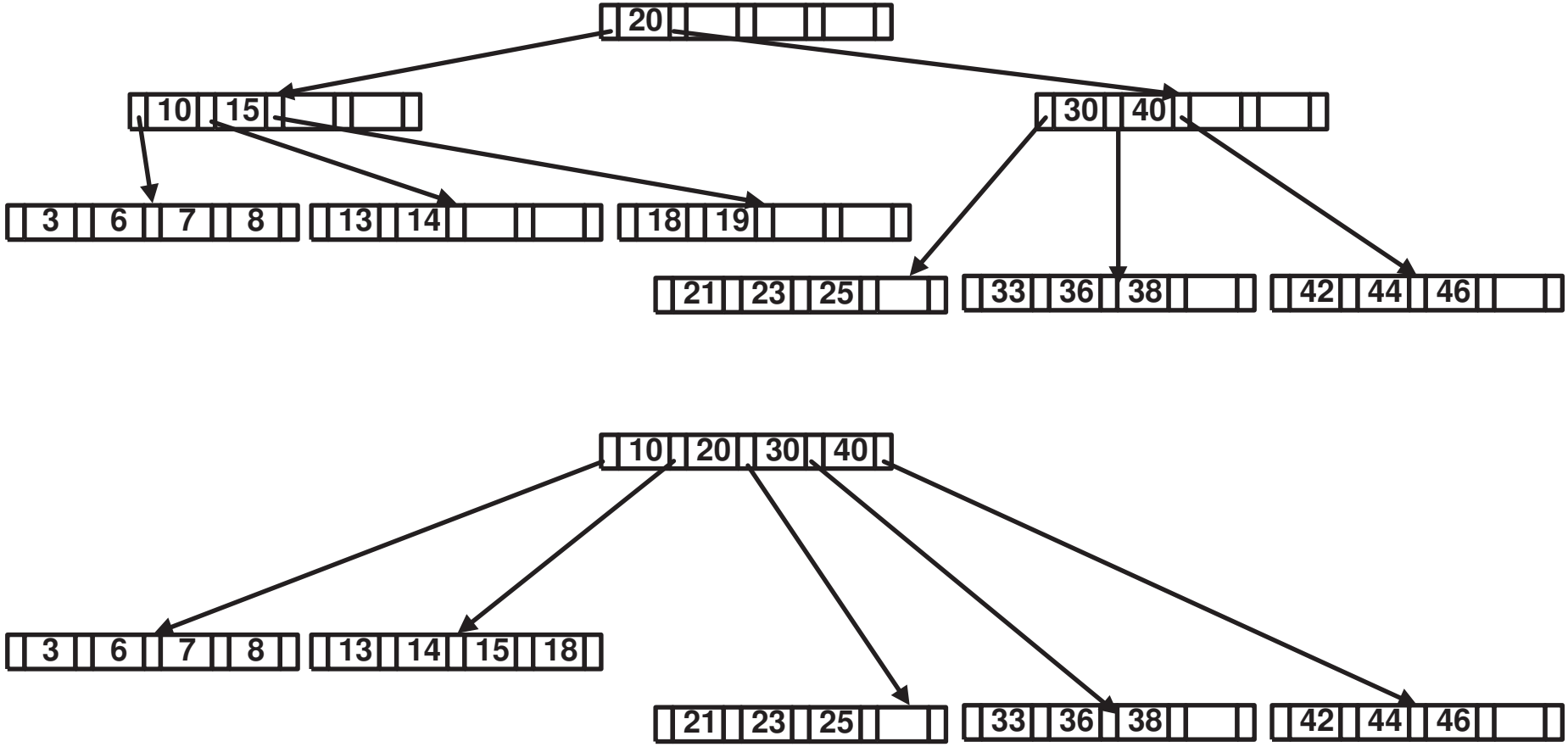




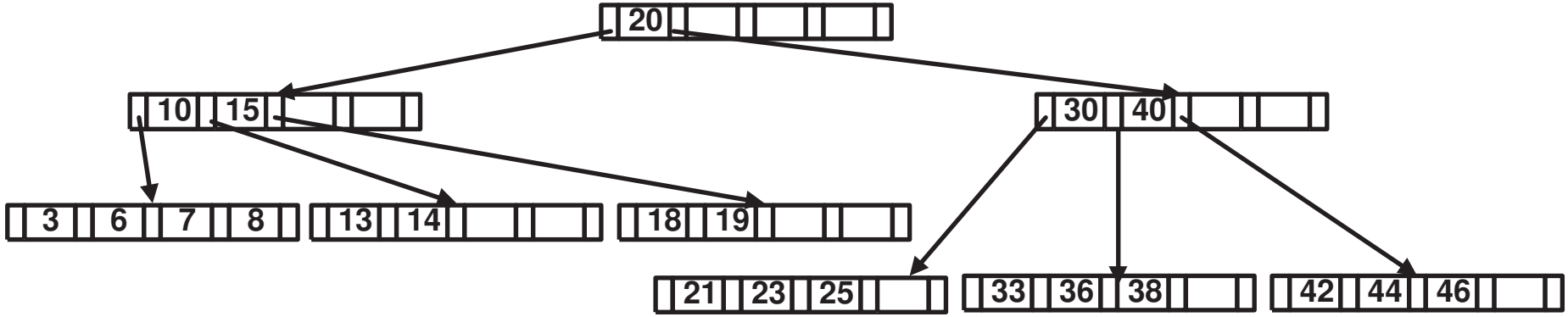
- With the deletion of non-leaf nodes, it is sometimes possible to redistribute. Consider the deletion of 10 from the following tree:

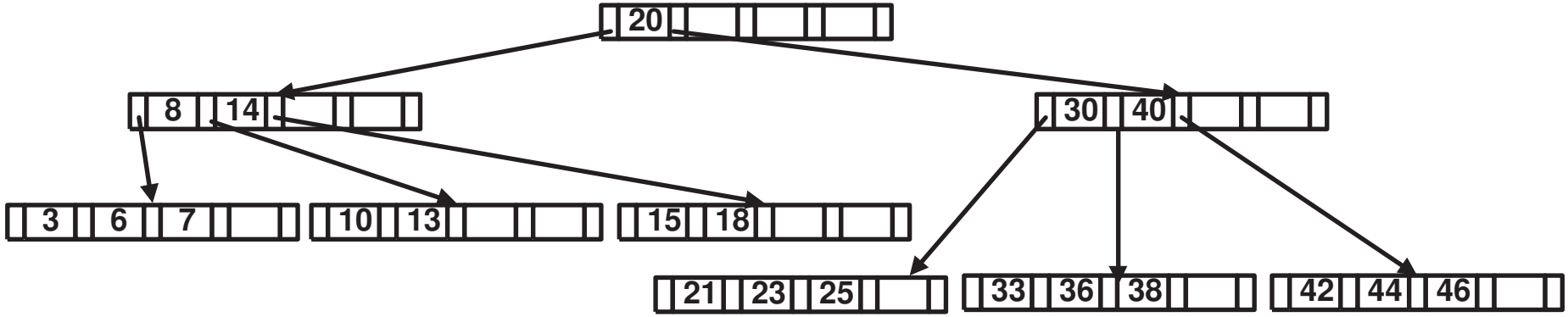


- Deletion of 19 appears to require adjustment at the second level, and then combination with the root:



However, it is sometimes possible to do a long-range multiple re-adjustment. Here is an alternate solution to the deletion of 19:





Some general heuristics for B-trees.

- When making adjustments, always try to keep sibling nodes with about the same number of keys.
- Avoid depth changes whenever possible.

Disadvantages of B-trees for DBMS's:

- There are too many disk accesses for large files.
- Cannot perform sequential processing efficiently.

A look at the number of disk accesses:

Example:

Assumptions:

- 2 Kbytes. pages.
- 128 bytes per record (very conservative for a DBMS).
- 4 bytes per pointer (4 Gbyte. address space).

Maximum number of records per page:

$$4 + n \bullet (128+4) \leq 2048$$

So: $n = \lfloor 2044/132 \rfloor = 15$ records per page.

- Suppose that we have 10^6 records.

What is the depth of the tree?

- The answer depends upon the *fullness* of the nodes. A minimum and a maximum can be computed.

Maximum depth / Minimum density:

- The tree will have maximum depth when the nodes have minimum density.
- For minimum density, it is assumed that nodes are at least half full, but no more full than necessary.
- The sole exception is the root node, which need contain only one data value.

A node which is half full contains $\lfloor 15 \bullet (½) \rfloor = 7$ records.

- Note that “half full” entails “round down” here.

What is the maximum depth of the tree?

- First, this problem may be solved with a "brute force" technique, using a table.

Level	Nodes	Records at this Level	Total Records
Root	1	1	1
1	2	2•7 = 14	15
2	2•8 = 16	16•7 = 112	127
3	16•8 = 128	128•7 = 896	1023
4	128•8 = 1024	1024•7 = 7168	8191
5	1024•8 = 8192	8192•7 = 57344	65535
6	8192•8 = 65536	65536•7 = 458752	524287
7	65536•8 = 524288	524288•7 = 3670016	4194303

The maximum depth of the tree is thus 6, because a B-tree of depth of 7 would require a minimum of 4194303 records.

This "brute-force" approach becomes tedious, particularly when the depth becomes substantial, and it applies only to this special case.

It is instructive to derive a more general formula relating depth to the number of nodes in a B-tree.

The starting point is a B-tree with the following parameters:

- d denotes the depth of the B-tree.
- m denotes the total number of records in the root node.
- All nodes other than the root node contain exactly r records.

Note that not all B-trees have this structure!!

However, a formula for such B-trees will nonetheless prove very useful.

Such B-trees will be called *uniform* (m,r,d) *B-trees*.

Consider the following table, which computes the number of nodes and records at each level.

Level	Nodes	Records
Root	1	m
1	m+1	(m+1)•r
2	(m+1)•(r+1)	(m+1)•(r+1)•r
3	(m+1)•$(r+1)^2$	(m+1)•$(r+1)^2$•r
4	(m+1)•$(r+1)^3$	(m+1)•$(r+1)^3$•r
…	…	…
d	(m+1)•$(r+1)^{d-1}$	(m+1)•$(r+1)^{d-1}$•r

Thus, the total number of records R(m,r,d) in a uniform (m,r,d) B-tree is

$$m+(m+1)\cdot r\cdot\sum_{i=0}^{d-1}(r+1)^i$$

The general law

$$\sum_{i=0}^{d}k^i=\frac{k^{d+1}-1}{k-1}$$

which may be derived from:

$$(1+k+k^2+\ldots+k^n)(1-k)=(1-k^{n+1})$$

leads to

$$R(m,r,d)=m+(m+1)\cdot((r+1)^d-1)$$

- This formula easily simplifies to

$$R(m, r, d) = (m+1) \cdot (r+1)^d - 1$$

Now, reconsider the problem of finding the maximum depth of a B-tree with a given number of nodes. Instead of the "brute-force" approach, the above formula will be used.

The idea is to find the greatest depth d of a uniform (1,r,d) B-tree which has the property that the total number of records does not exceed the specified number of records N.

- The value of m is 1 in this case, since a tree of maximum depth is sought, and therefore as few records as possible are placed in the root node.
- The value of r is 7 for the example.
- The value of N for the example is 1000000.

Thus,

$$(m+1) \cdot (r+1)^d - 1 \leq N$$

or

$$(r+1)^d \leq \frac{N+1}{(m+1)}$$

- To solve, take the log base r+1 of each side.

$$d \leq log_{r+1}\left(\frac{N+1}{m+1}\right) = \frac{log_e\left(\frac{N+1}{m+1}\right)}{log_e(r+1)}$$

Plugging in r=7, N=1000000, and m=1 yields

$$d \leq \frac{log_e(500000.5)}{log_e(8)} = 6.31$$

Since the depth of a B-tree must be an integer, it follows that it must be no larger than 6, in agreement with the brute-force approach.

Minimum depth / Maximum density:

- The tree will have minimum depth when the nodes have maximum density.
- For maximum density, it is assumed that all nodes are full, including the root.

A node which is full contains 15 records.

What is the minimum depth of the tree in this case?

- First, this problem may be solved with a "brute force" technique, using a table.

Level	Nodes	Records at this Level	Total Records
Root	1	15	15
1	16	16•15 = 240	255
2	16^2 = 256	256•15 = 3840	4001
3	16^3 = 4096	4096•15 = 61440	65441
4	16^4 = 65536	65536•15 = 983040	1048481

Since the "Total Records" entry is now the maximum number for the given depth, the tree must have depth at least 4, since a tree of depth 3 can hold at most 65441 records.

This problem can also be solved using the general R(m,r,d) formula. This time:

- m = r =15, since each node, including the root, contains the maximum number of records.
- N = 1000000, as before.

Thus,

$$(m+1)\cdot(r+1)^d - 1 \geq N$$

or

$$(r+1)^d \geq \frac{N+1}{m+1}$$

Since m=r

$$(r+1)^{d+1} \geq N+1$$

Thus

$$d+1 \geq log_{r+1}(N+1) = \frac{log_e(N+1)}{log_e(r+1)}$$

or

$$d \geq \frac{log_e(N+1)}{log_e(r+1)} - 1$$

Plugging in N=1000000 and r=15 yields

$$d \geq \frac{log_e(1000001)}{log_e(16)} - 1 = 3.9828$$

Since d must be an integer, it follows that it must be at least 4. Again, this is in agreement with the brute-force approach.

The fact that d is very close to 4 suggests that by adding just a few more nodes to N, a tree of depth five would be required. The "brute-force" chart confirms this; the largest uniform (15,15,4) B-tree 1048481 nodes, only 48481 more than 1000000.

The "R(m,r,d)" formula is useful in other ways. For example, if the total number of records, as well as depth d and root record count m of a uniform (m,r,d) B-tree is known, then the record density r can be computed as well. Starting with

$$R(m,r,d)=(m+1)\cdot(r+1)^d-1$$

then

$$(r+1)^d=\frac{R(m,r,d)+1}{m+1}$$

To solve for r, one simply takes the d^{th} root of both sides, and then moves the 1 over:

$$r=\sqrt[d]{\frac{R(m,r,d)+1}{m+1}}-1$$

Let us illustrate the utility of this formula with some concrete examples.

First of all, consider the example of a maximum depth / minimum density tree with 10^6 records.

Specifically, consider a uniform (1,r,6) tree with 10^6 records. We may ask what the value of r is.

$$r = \sqrt[d]{\frac{R(m,r,d)+1}{m+1}} - 1 = \sqrt[6]{\frac{10^6+1}{1+1}} - 1 = 7.90$$

- This says that a *uniform* (1,r,6) B-tree would have 7.90 records in its (non-root) nodes.

- Of course, it is impossible to have a tree with 7.90 records per node. This result is thus just an estimate. A real B-tree, as balanced as possible, would have between 7 and 8 records per node.

Suppose now that we put 2 records in the root node. The average value of r for the other nodes then becomes

$$r = \sqrt[d]{\frac{R(m,r,d)+1}{m+1}} - 1 = \sqrt[6]{\frac{10^6+1}{2+1}} - 1 = 7.32$$

- Thus, by creating slightly more fan-out at the root node, the lower nodes are much less densely populated. In fact, the density is just barely adequate, since the minimum is 7.

Finally, suppose that we put 3 records in the root node. The average value of r for the other nodes then becomes

$$r = \sqrt[d]{\frac{R(m, r, d)+1}{m+1}} - 1 = \sqrt[6]{\frac{10^6+1}{3+1}} - 1 = 6.93$$

- Here the value of r is not enough for a valid tree, since the minimum is 7.
- Thus, it should not be assumed that a legal value for r will always result.

Now let us look at the example for minimum depth / maximum density.

The tree of 10^6 records with m=15 and d=4 is characterized as follows:

$$r = \sqrt[d]{\frac{R(m, r, d)+1}{m+1}} - 1 = \sqrt[4]{\frac{10^6+1}{15+1}} - 1 = 14.81$$

- The average record density of the nodes is extremely high, as is expected, since a uniform (15,r,4) tree can have a maximum of 1048481 nodes.

If the fan-out at the root is reduced by just one, to m=14, we obtain

$$r = \sqrt[d]{\frac{R(m, r, d)+1}{m+1}} - 1 = \sqrt[4]{\frac{10^6+1}{14+1}} - 1 = 15.06$$

- This does not characterize a real B-tree, since the maximum value for r is 15.

Indeed, a uniform (14,15,4) B-tree has as the maximum number of records:

$$(m+1)\cdot(r+1)^d - 1 = 15 \cdot 16^4 - 1 = 983041$$

which is only slightly less than 10^6.

Average Path Length

Now consider the following question, the answer to which is a major factor in computing access time:

- For a given uniform (m,r,d) B-tree, what is the average path length from the root to a node?

From previous computations, we know that

- Number of records at level d = $(m+1)\bullet(r+1)^{d-1}\bullet r$
- Total number of records = $(m+1)\bullet(r+1)^{d}-1$

Thus, the percentage of records which are situated in leaves is approximately

$$\frac{(m+1)\cdot(r+1)^{d-1}\cdot r}{(m+1)\cdot(r+1)^{d}-1} \approx \frac{(m+1)\cdot(r+1)^{d-1}\cdot r}{(m+1)\cdot(r+1)^{d}} = \frac{r}{r+1}$$

If r is reasonably large (as is typically the case with a B-tree), then most of the records will reside in the leaf nodes.

r	r/(r+1)
1	0.500
8	0.888
15	0.938
32	0.970
100	0.990

Thus, even for the example which we have considered, it can be expected that around 90% of the records will reside at leaf nodes.

Implication:

- If there is one disk request per access to a B-tree node, then the average access time will be the time for a single access times the depth of the tree.

- For the example, four or five disk access per record fetch is excessive! Even at 10 ms. per access, this would result in 40 to 50 ms. per record access! (Of course, a good caching strategy would help immensely.)

- Can this be improved upon?

There is an interesting improvement.

- Since pointer fields are not needed in leaf nodes, we could have two types of nodes.
  - For interior nodes, use the design we have already described.
  - For leaf nodes, have data fields only.

- In our example, without pointers, we could fit 16 records in such a leaf node, instead of just 15.

How substantial an improvement is this?

- The total number of leaf nodes in a uniform (m,r,d) B-tree is $(m+1)\bullet(r+1)^{d-1}$.
- The total number of records in the tree is $(m+1)\bullet(r+1)^{d}-1$.
- Assume that by eliminating pointer fields, an additional k records may be placed in a node.

Then, the capacity of the tree, for r set to the maximum number of records per ordinary node, is increased by the factor

$$\frac{(m+1)\cdot(r+1)^{d-1}\cdot k}{(m+1)\cdot(r+1)^{d}-1} \approx \frac{(m+1)\cdot(r+1)^{d-1}\cdot k}{(m+1)\cdot(r+1)^{d}} = \frac{k}{r+1}$$

For the running example, k would be just 1, so the improvement would be 1/(r+1), which is a a rather small amount; for r=9, it would be about 10%.

Still, this is a simple improvement which may be made with little or no programming overhead.

Additional comments on B-trees

There are two distinct flavors of implementation:

1. Actual records are stored in the data fields.

Advantages:

- Rapid access to adjacent records.

Disadvantages:

- Low density of records per node results in a very large structure.

2. The entire B-tree is merely an index to the actual records; the data fields of the B-tree are pointers to the actual records .

Advantages:

- High density of keys per node, as only key values and pointers need be stored.

Disadvantages:

- Potentially extreme fragmentation of actual data.

A better approach: the B^+-tree

A B^+-tree differs from a B-tree in the following fundamental way:

- All of the data records are held at the leaves. The interior nodes are used solely as an index structure.

- Leaf nodes must be at least half full of *records.*

Advantages:

- Since index fields are much smaller than record fields, the index (non-leaf nodes) will be relatively small.

- The non-leaf index structure is often small enough so that it may be kept in main memory.
    - This gives constant-time access. (only one disk access per data request!)

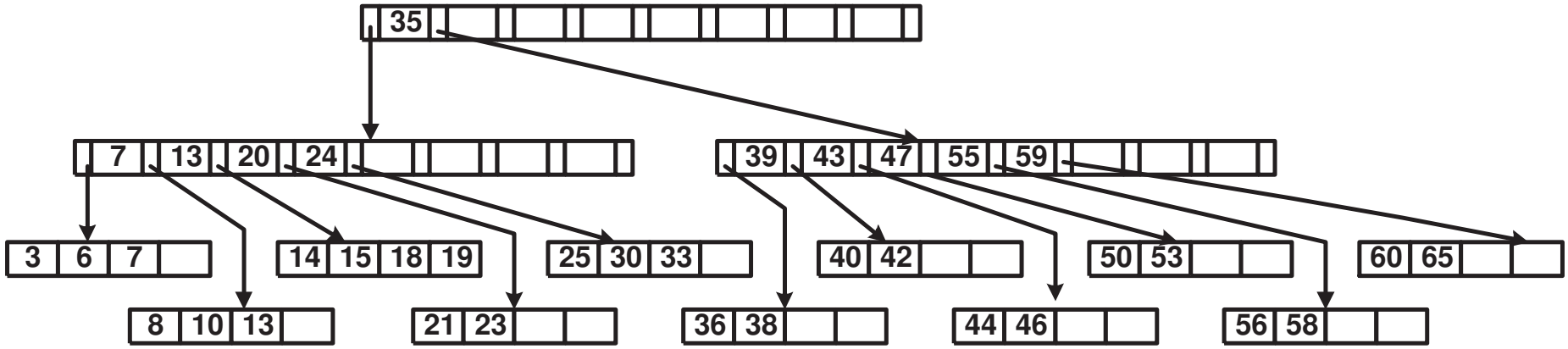
- The leaf nodes may be linked together to provide a simple means of sequential processing.

- The insertion and deletion algorithms are similar to those for B-trees.

- Adjustments to non-leaf nodes are easier, since field values are not records, but merely index value.

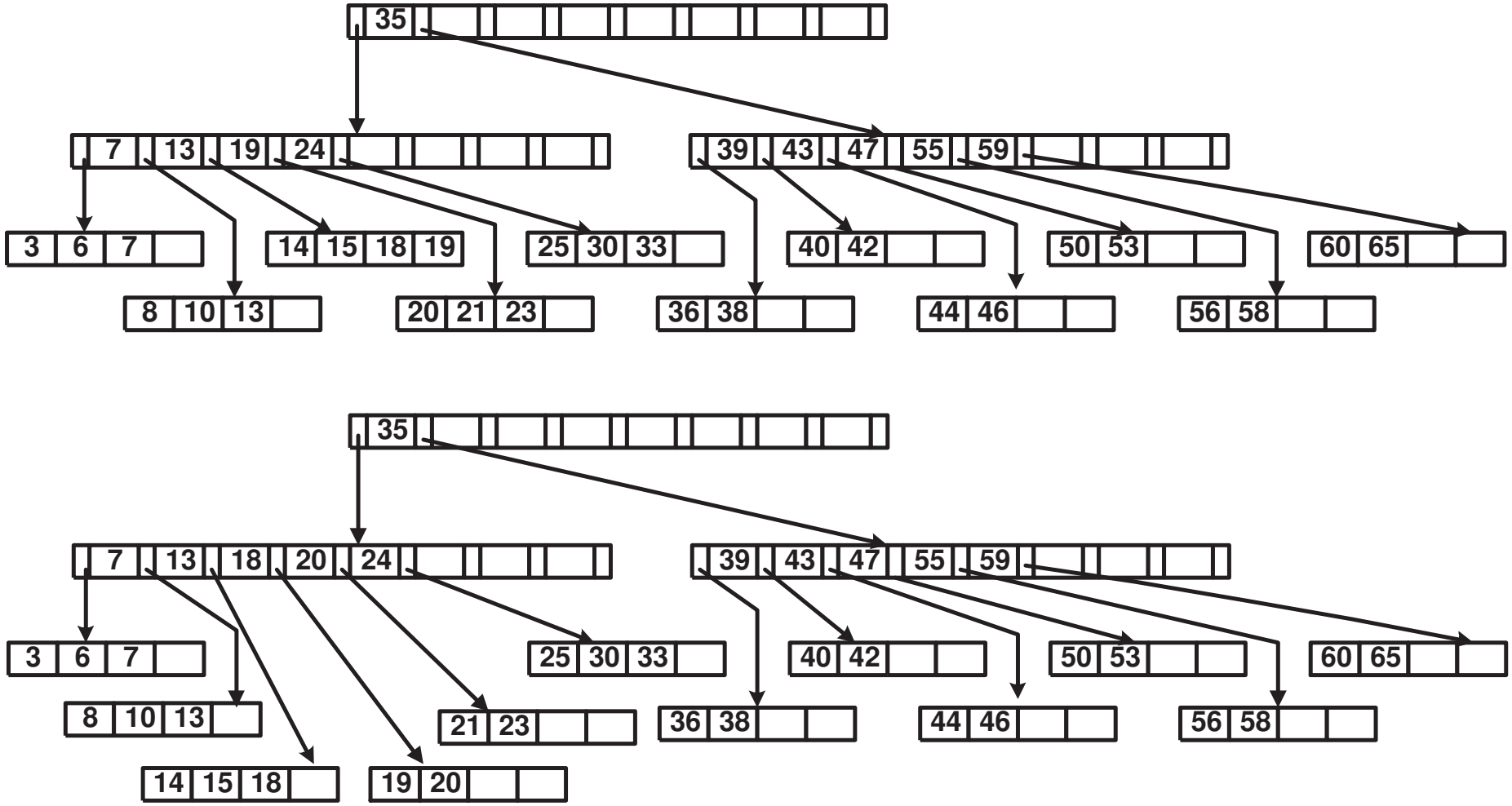
An example of a B^+-tree:

(Convention: Records whose keys match an index entry are found to the left of that index entry)

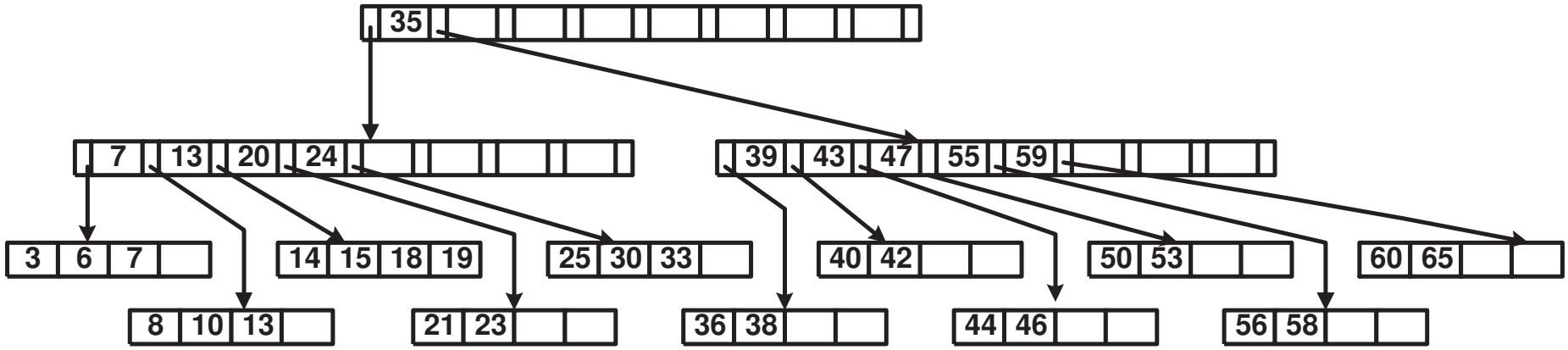


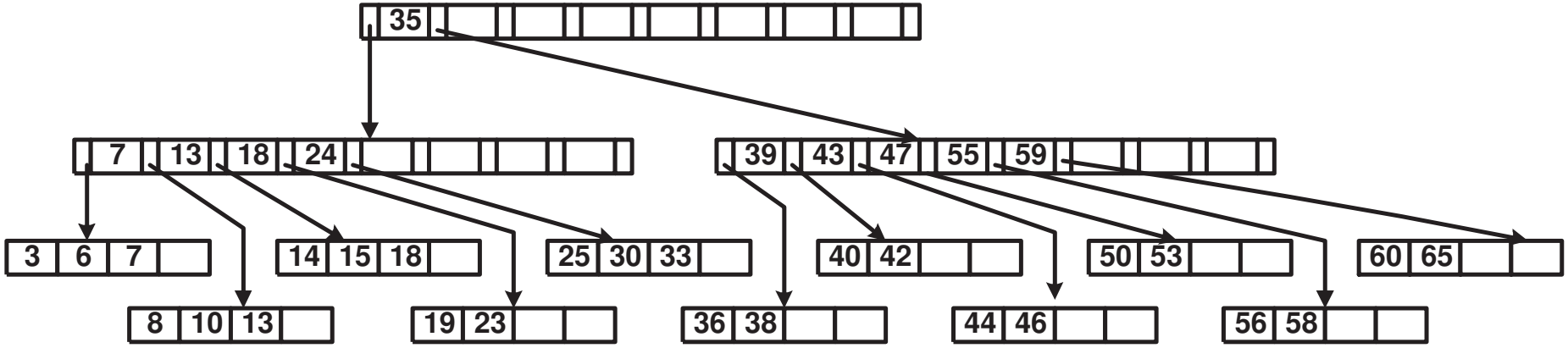
We now illustrate insertion and deletion.

Example of insertion of 20 into the above B^+-tree (two different ways):



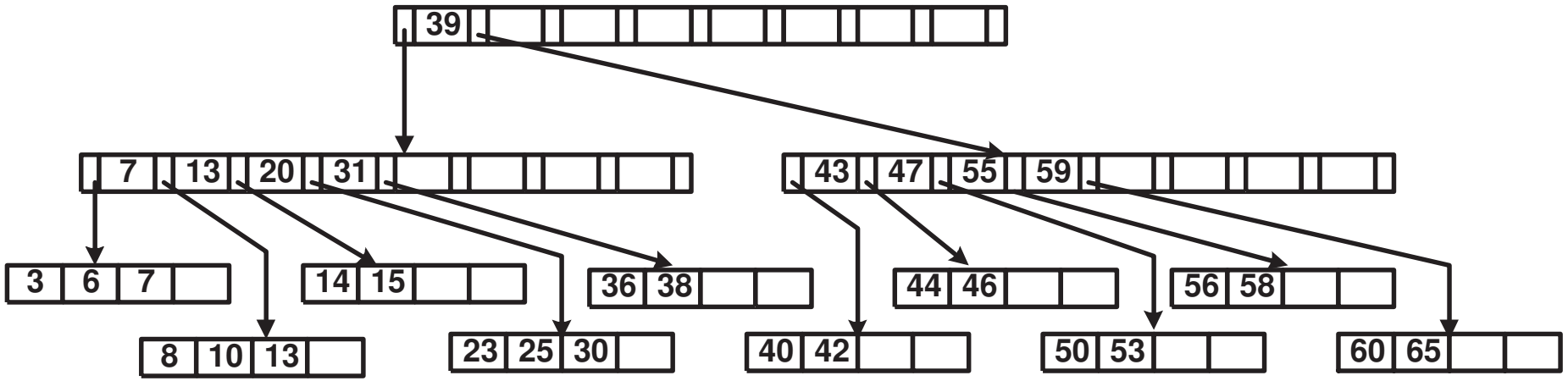
- An example of deletion: Deletion of the value 21:



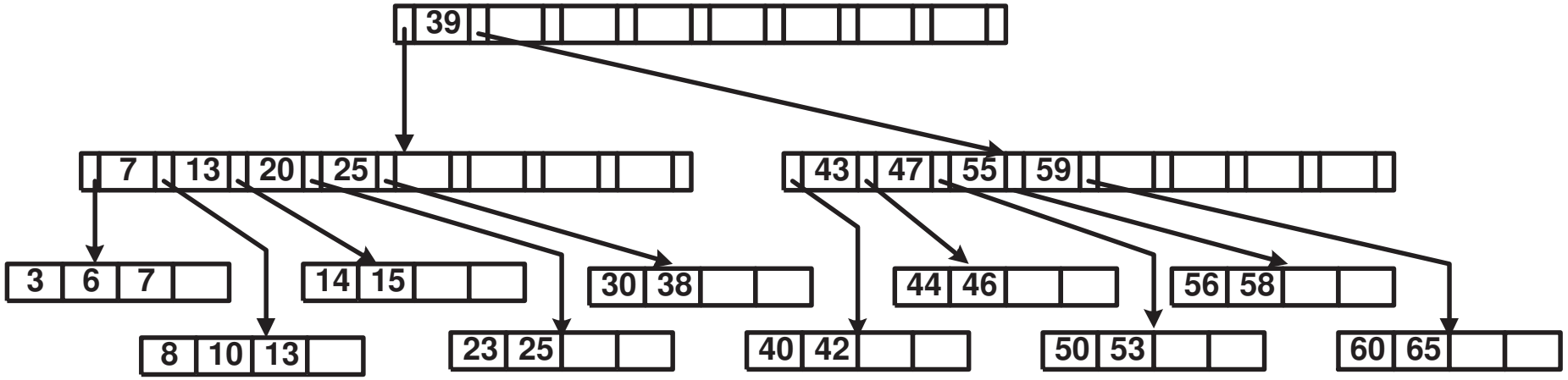


The deletion of 36 and then 38 illustrates a merging at level one.

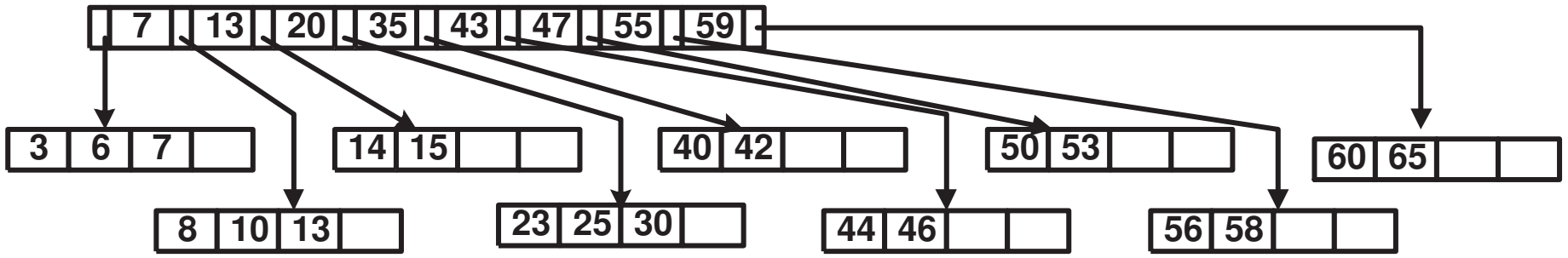
This is the initial tree:



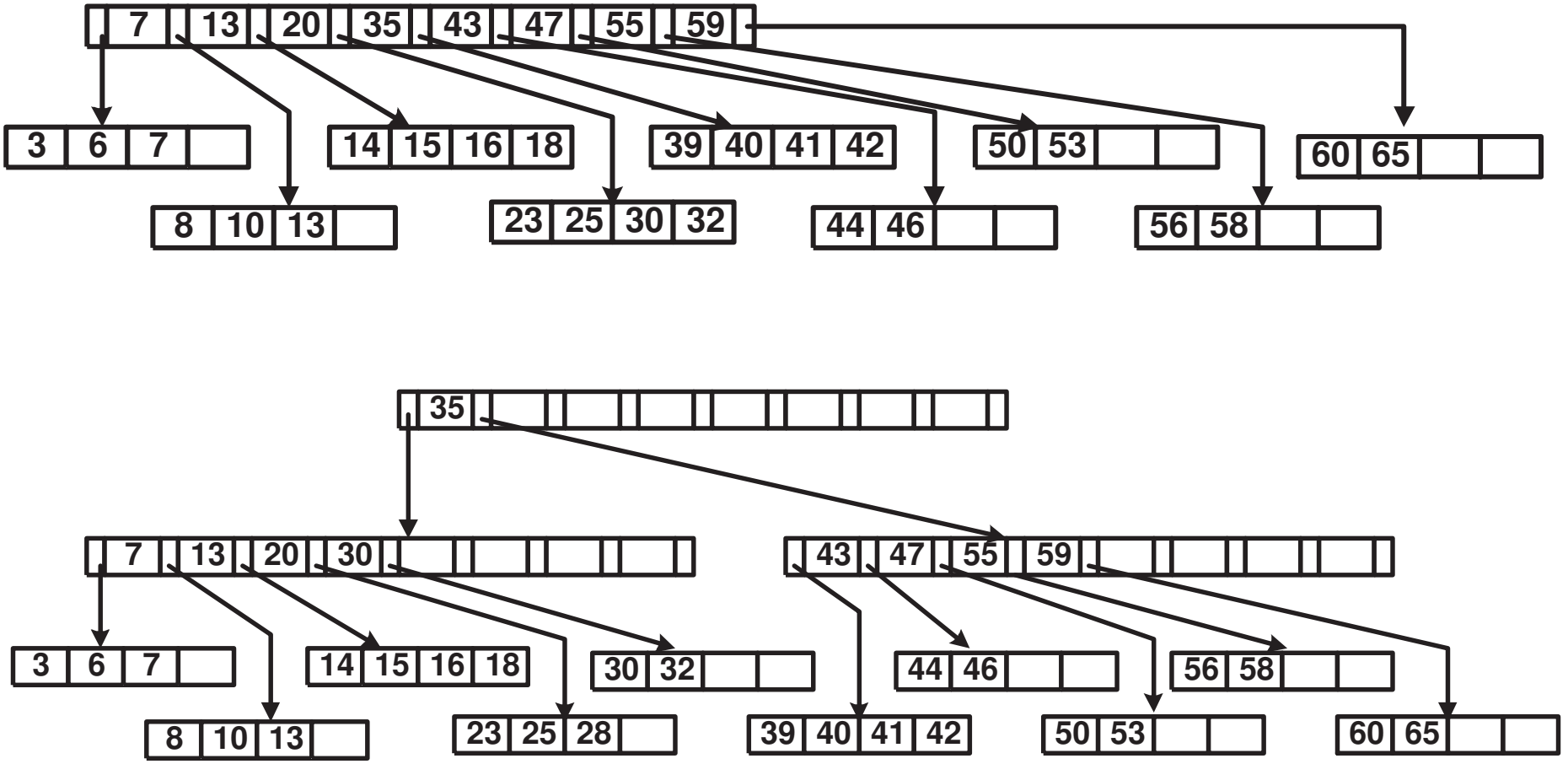
The deletion of 36 is accomplished via a rotation of values:



Continuation of the deletion example; deletion of 38 with a reduction in depth:



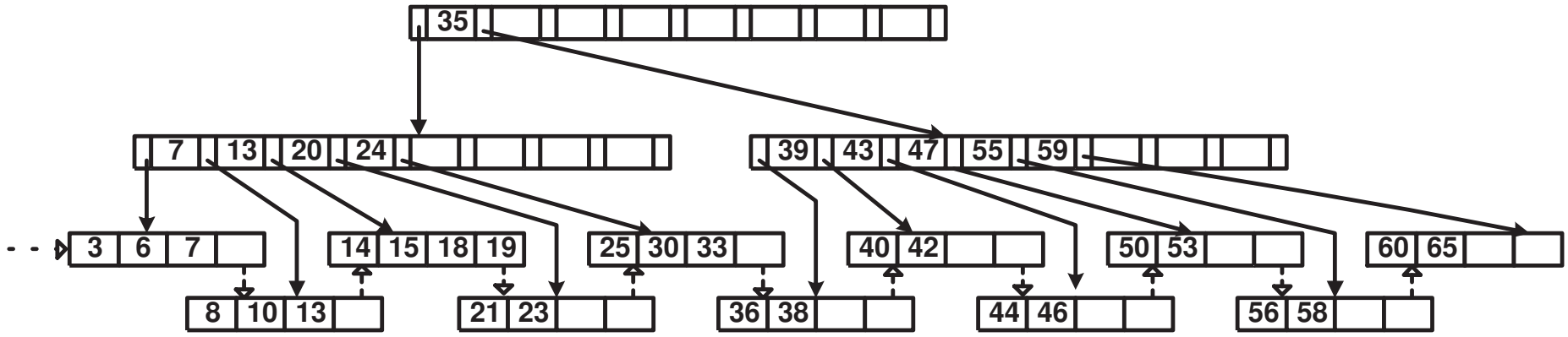
An example of expansion; the insertion of a record with key 28:



- To admit sequential processing of the file, the leaves of a B^+-tree may be linked together.
  - See the links shown with dashed lines.

- To admit sequential processing of the file, the leaves of a B^+-tree may be linked together.
  - See the links shown with dashed lines.

- Usually, two-way linking (doubly linked list) is used, to allow easy insertion, as well as to allow reverse-order traversal.



Analysis of B^+-tree size requirements:

Example:
Assumptions:

- 2 Kbyte. pages.
- 128 bytes per record (very conservative for a DBMS)
- 4 bytes per pointer (4 Gbyte. address space).
- 16 bytes per internal key.
- 10^6 records total.
- 8 bytes total for the sequential pointers of the leaf nodes.

Number of indices per internal node:

$4 + n \bullet (16+4) = 2048$
So: $n = \lfloor 2044/20 \rfloor$ = maximum 102 keys per internal node.

Number of records per leaf node:

$8 + n \bullet 128 = 2048$
So: $n = \lfloor 2040/128 \rfloor$ = maximum 15 records per external node.

(The initial eight bytes are for the sequential pointers, with links in both directions.)

Maximum depth / Minimum density

The assumptions are similar to those for a B-tree:

- The tree will have maximum depth when the nodes have minimum density.
- For minimum density, it is assumed that index nodes are at least half full of pointers, but no more full than necessary.
- Leaf nodes are at least half full of *records*.
- The sole exception is the root node, which need contain only one index value.

- An index node which is half full contains $\lfloor 102 \bullet (½) \rfloor = 51$ indices. (Round down)
- A record node which is half full contains $\lceil 15 \bullet (½) \rceil = 8$ records. (Round up)

What is the maximum depth of the index structure for tree?

- First, this problem may be solved with a "brute force" technique, using a table.

Level	Nodes	Keys at this Level	Min. Records at Leaf Level Below
Root	1	1	2•8 = 16
1	2	2•51 = 102	2•52•8 = 832
2	2•52 = 104	104•51= 5304	104•52•8 = 58240
3	104•52= 5408	5408•51 = 275808	5408•52•8 = 2249728

The maximum depth of the index structure is thus 2, because an index of depth of 3 would require a minimum of 2249728 records. The tree itself has depth bounded by 3.

In analogy to a uniform (m,r,d) B-tree, we may define the notion of a *uniform* (m,q,r,d) B^+-tree. Such trees have the following uniform parameters.

m = total number of indices at the root node.
q = total number of indices in each other index node.
r = total number of records in each leaf node.
d = depth of the tree, from the root to a leaf node.

In the above example, m=1, q=51, r=8, and d is to be computed.

Remember that not every B^+-tree is uniform. This is a special case, which is very useful for computational purposes.

Consider the following table, which computes the number of nodes and records at each level.

Index Level	Index Nodes	Keys	Total Records at Next Level
Root	1	m	(m+1)•r
1	m+1	(m+1)•q	(m+1)•(q+1)•r
2	(m+1)•(q+1)	(m+1)•(q+1)•q	$(m+1)\bullet(q+1)^2\bullet r$
3	$(m+1)\bullet(q+1)^2$	$(m+1)\bullet(q+1)^2\bullet q$	$(m+1)\bullet(q+1)^3\bullet r$
4	$(m+1)\bullet(q+1)^3$	$(m+1)\bullet(q+1)^3\bullet q$	$(m+1)\bullet(q+1)^4\bullet r$
…	…	…	…
d	$(m+1)\bullet(q+1)^{d-1}$	$(m+1)\bullet(q+1)^{d-1}\bullet q$	$(m+1)\bullet(q+1)^d\bullet r$

Let R(m,q,r,d) denote the total number of records which are stored in a uniform (m,q,r,d) B^+-tree. Then

$$R(m,q,r,d)=(m+1)\cdot(q+1)^{d-1}\cdot r$$

We can solve for d in this equation as in the one for B-trees:

$$d=log_{q+1}\left(\frac{R(m,q,r,d)}{(m+1)\cdot r}\right)+1$$

For the example (m=1, q=51, r=8, $R(m,q,r,d)=10^6$), we compute d = 3.794. Since d must be an integer, the depth of the tree must be at most 3, in agreement with the brute-force approach.

Minimum depth / Maximum density:

- The tree will have minimum depth when the nodes have maximum density.
- For maximum density, it is assumed that all nodes are full, including the root.

- An index node which is full contains 102 indices.
- A leaf node which is full contains 15 records.

What is the minimum depth of the tree in this case?

- First, this problem may be solved with a "brute force" technique, using a table.

Level	Nodes	Keys at this Level	Max. Records at Leaf Level
Root	1	102	103•15 =1545
1	103	103•102 = 10506	103^2•15 = 159135
2	103^2 = 10609	103^2•102= 1082118	103^3•15 = 16390905

Since the "Total Records" entry is now the maximum number for the given depth, the tree must have depth at least 3, since a tree of depth 2 can hold at most 159135 records. Remember that 1 must be added to the level to account for the leaf-level data nodes.

- Note that a tree of depth three will accommodate over 16M records!

The formula

$$d = log_{q+1}\left\lfloor \frac{R(m,q,r,d)}{(m+1)\cdot r} \right\rfloor + 1$$

when supplied with m=q=102, r=15, and $R(m,q,r,d)=10^6$, yields d = 2.39. Since d must be an integer, it is round *up* to 3, to get the minimum height of the tree.

Thus, the minimum and maximum heights are the same in this example!

In a uniform (m,q,r,d) B$^+$-tree, the number of index (interior) nodes is

$$1+(m+1)\cdot\sum_{i=0}^{d-2}(q+1)^i = 1+\frac{(m+1)\cdot((q+1)^{d-1}-1)}{q}$$

- Consider a uniform (1,51,8,4) B$^+$-tree, which would have 2249728 data records.
    - According to this formula, It would have just 5515 index nodes.
    - At 2KB per node, this translates to just over 11 Mbyte. of memory.

Consider a uniform (102,102,15,3) B$^+$-tree, which would have 16390905 data records.

- According to this formula, It would have just 10713 index nodes.
- At 2KB per node, this translates to just under 22 Mb. of memory.

Why not keep the whole index in main memory?

- This reduces the number of disk accesses per record to one – constant time access!

Extendible Hashing:

- The goal of extendible hashing is to realize the advantage of hashing:
    - Fast (constant-time) random access

  within the context of data on secondary storage.

Idea:

- The hashing function

  h: keys $\rightarrow$ hash values

  is broken into two pieces:
    - Directory address
    - Leaf address

Toy example:

Suppose we have a 16-bit hash address:

- Directory address size = 3 bits
- Hash address size = 13 bits

Suppose that κ is a key with the property that h(κ) = 1010111010110001.

Then,

- Directory address = 101
- Leaf address = 0111010110001

- This assumes that we use the <u>first</u> three bits as the directory address.

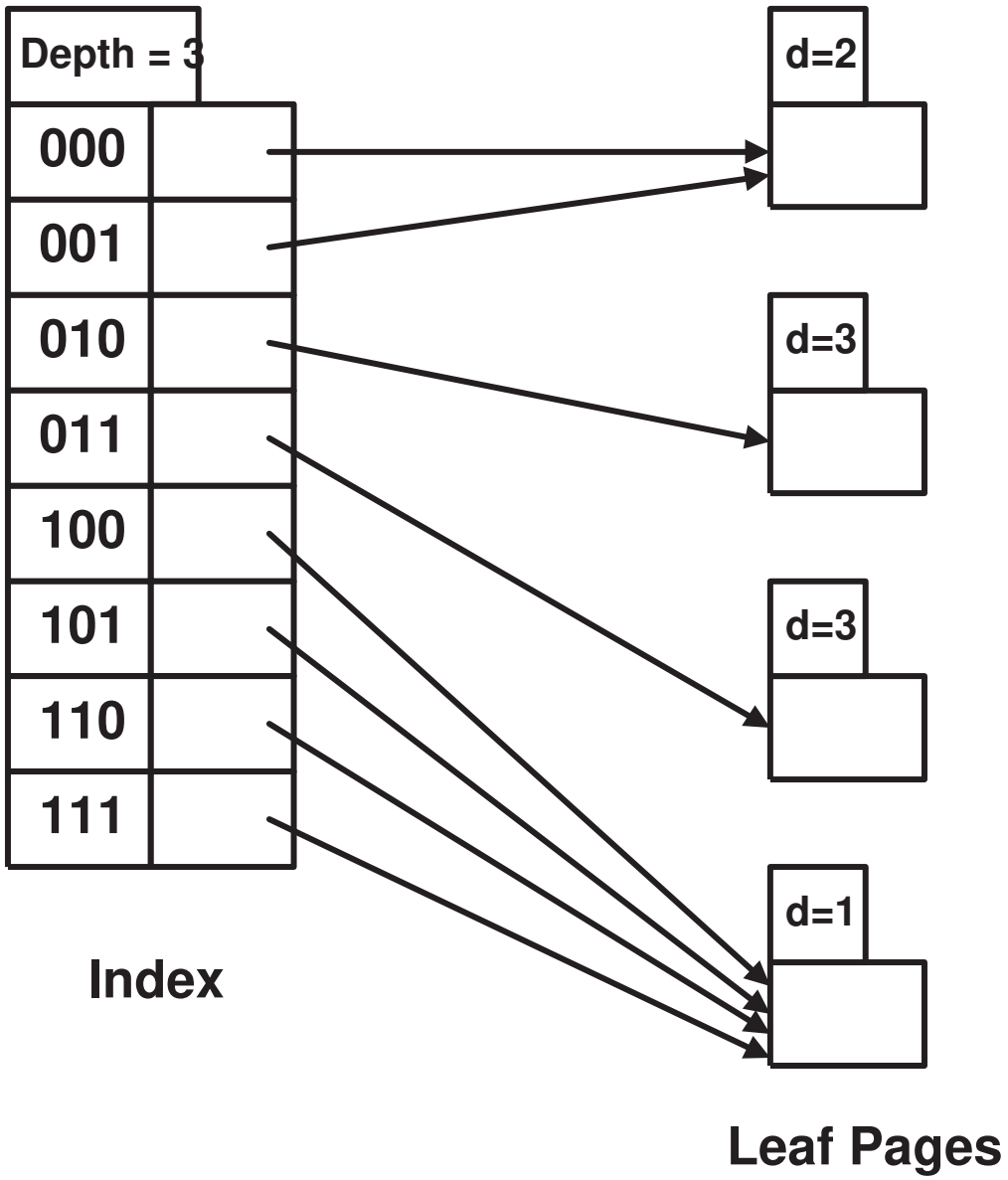
- There are other possibilities.

General structure:

- For each directory entry, there is a hash bucket.
- Directory entries may share hash buckets.
  - In a “powers-of-two” fashion

Example:

- The value d indicates the actual depth of the index entries associated with that bucket.



Notes:

- In a “real” application, the index size would be much larger (*e.g.,* 12 to 16 bits).

- The index is typically kept in main memory, since it is quite small.

- The index may be searched very rapidly, in an “array” style.
  - The i^{th} entry is found at address
    base + i $\times$ entry_size.

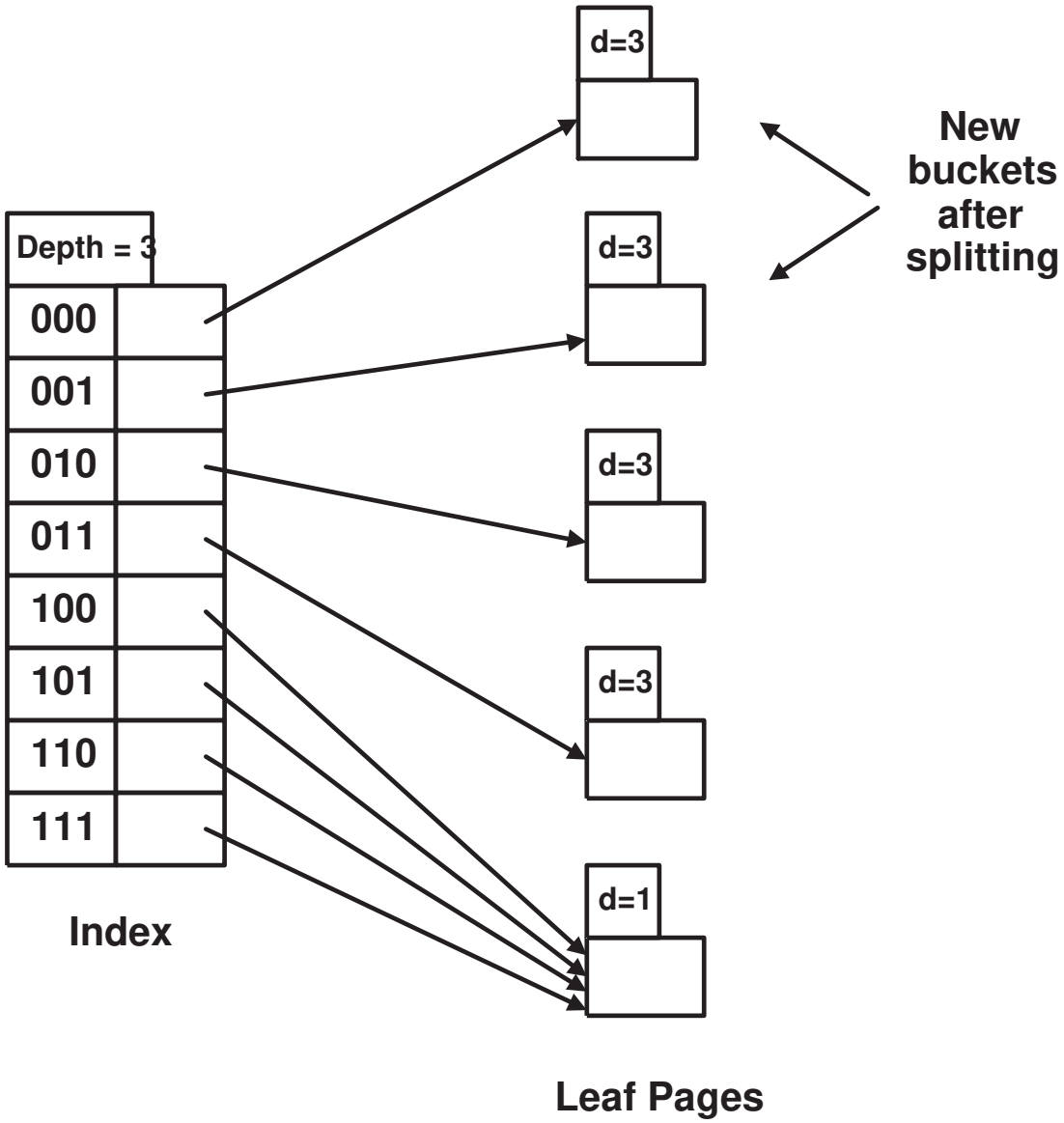
- The sharing of buckets accommodates unevenness in the distribution of hashed values.

- The arrangement of elements within a bucket is a separate issue, and may be optimized for the particular application.
    - It should be done is such a way to accommodate the “splitting” operation, which will be described next.

Now let us examine why this scheme is termed *extendible.*

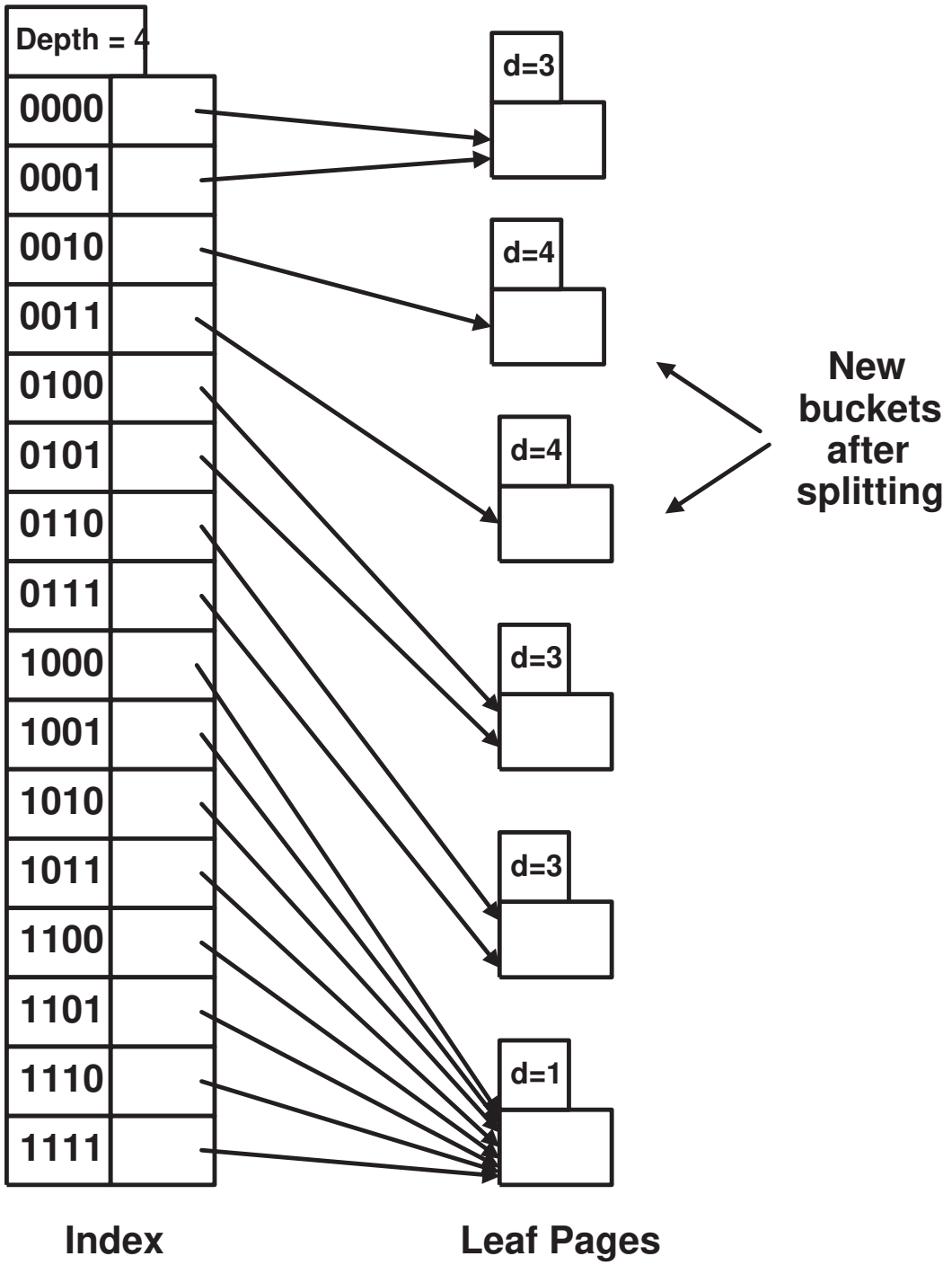
- Suppose that the 00 bucket (shared by 000 and 001) in the example becomes full.

- This bucket is then split into two, as illustrated below.



- A more serious problem occurs if a bucket which is associated with only one index entry becomes overfull.

- In this case, the index must be doubled, as illustrated below for the case that the bucket for 001 became full.



Other issues:

- It is possible to construct a block-combining strategy as well, but this is uncommon unless it is expected that the database will shrink substantially without subsequent growth.

- Random-access time may be somewhat superior to that for B^+-trees, particularly in situations in which there is relatively little memory available:
    - The index for extendible hashing is much smaller than the index for a corresponding B^+-tree.
    - No searching is required; just computation of a key-to-address transformation and an array access.
- Relative advantages diminish as memory size increases.

- With a typical hashing strategy:
    - Sequential processing becomes very slow.
    - Batch processing is still feasible.

- In some cases, it may be possible to arrange things so that sequential processing is still feasible:
    - Use a trivial KAT: the first k bits of the key become the directory address, and the rest the leaf address.
    - This may or may not result in very poor record distribution, depending upon the application.

- Reference for further information:

R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong, “Extendible hashing – a fast access method for dynamic files,” ACM Transactions on Database Systems, 4(3), September 1979, pp. 315-344.