Physical Database Design

These slides are mostly taken verbatim, or with minor
changes, from those prepared by
Stephen Hegner (http://www.cs.umu.se/ hegner/)
of Umeå University

Data Independence — a Basic Consideration

Data Independence refers to the condition that the functionality of the external user interface(s) to the DBMS be independent of the internal storage representation of the data.

- One of the fundamental features of the relational model is that it exhibits such independence by design.
- Nevertheless, it is important for the sophisticated user to have some understanding of the internal storage model, because certain choices of approach to queries may affect performance substantially.
  - ... although it should never affect correctness.

Types of Access

- There are many different types of access which a comprehensive DBMS must support:

Key-based: Retrieval of data based upon the values of specific keys suggests an indexed or hashed strategy.

Sequential processing: Retrieval of large amounts of data in some order suggests that the data themselves should be stored in some appropriate order.

Range queries: Retrieval of data for which certain parameters fall within a range of values suggests that the above two approaches need to be combined.

- It is generally not possible to provide optimal access for all of these possibilities.
- Nevertheless, much is known about how to obtain such access with reasonable performance.

Records

Record: The basic entity of storage in a DBMS.

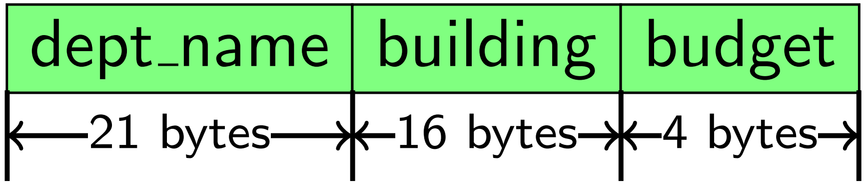
- In the ubiquitous *row-based* implementation of the relational model, each tuple is represented as a record.

Field: The basic physical data item.

- Each record is divided into one or more fields.
- In the usual implementation of the relational model, each field of a record corresponds to an attribute, with the field containing the value for that attribute.

Fixed-length record: The most common implementation is to allocate a fixed-size field for each attribute.

```
CREATE TABLE department
    (dept_name          VARCHAR(20),
     building           VARCHAR(15),
     budget             NUMERIC(12,2)
                   CHECK (budget > 0),
     PRIMARY KEY (dept_name)
    );
```



dept_name	building	budget
21 bytes	16 bytes	4 bytes

Variable-Length Records of a Fixed Type

Variable-length records: There are a number of situations in which it is useful to allow the length of a record of a given type to vary.

- Most often, this possibility arises because the length of one or more fields is variable.

  Predominately-null fields: If a field is null in most records, it may be advantageous to represent the null value with a bit marker.

  Fields with sets of values: In object-relational models (supported in the latest SQL standard), it is possible to define fields which take multisets or arrays as values.

Fields whose size varies greatly: These are typically handled other ways.

  - Large objects such as BLOBs and CLOBs are stored separately, with the record containing only a (fixed-size) pointer to the object.
  - It is not common (although possible) to represent `VARCHAR` fields using variable-length constructions.

Implementation of Variable-Length Records

- A variable-length record may be implemented as shown below for three fixed-length fields and two variable-length fields..

Fixed Field$_1$	Fixed Field$_2$	Fixed Field$_3$	Var Field Count	Var Field$_1$ Loc	Var Field$_2$ Loc	Var Field Data

Var Field Count: Indicates the number of variable fields.

Var Field$_i$ Loc: Describes how to find the i^{th} variable field in Var Field Data.

  - Start offset + size
  - Start offset + end offset

Drawback: It takes more time to retrieve an item which is stored in a variable-length format than to retrieve the same data in a fixed-length format.

Principle: Memory (primary and secondary) has become much less expensive, so it is effective to use variable-length records only when the amount of space to be saved is substantial.

Physical Storage of Records

Blocks: Records are stored in units called *blocks*.

- A block usually corresponds to the sector size for the hard disk, or a small multiple of that size.

Blocking factor: The number of records which are stored in a block.

  - Depends upon the type of record.
  - Variable per record type if the records are variable record length.
  - Variable if several different types of record are stored in the same block.
  - In these cases, averages are typically used.

Unspanned blocking: Each record is contained entirely in one block.

Spanned blocking: A record may be split over (usually two) blocks.

  - Relatively rare in modern systems.

Organization of Records in Storage

- There are three fundamental ways in which records may be stored.
- These approaches are typically per record type, so distinct record types may have distinct methods of storage.

Heap: Any record may be stored anywhere.

  - Typically, different record types are stored in distinct *files*.
  - Access is entirely via indices.

Ordered: The records are stored in the order defined by the value(s) of one more more attributes, typically but not always the primary key of the associated relation.

  - Indices may still be used to facilitate both sequential and non-sequential access.

Hashed: The records are distributed into *buckets* according to some *hashing function*.

  - Within each bucket, the records may be further organized according to one of the above two approaches.

Sequential Organization

Question: What does it mean for records on disk to be ordered?

- Here ordering on the ID (first) field of the `Instructor` relation is illustrated.

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

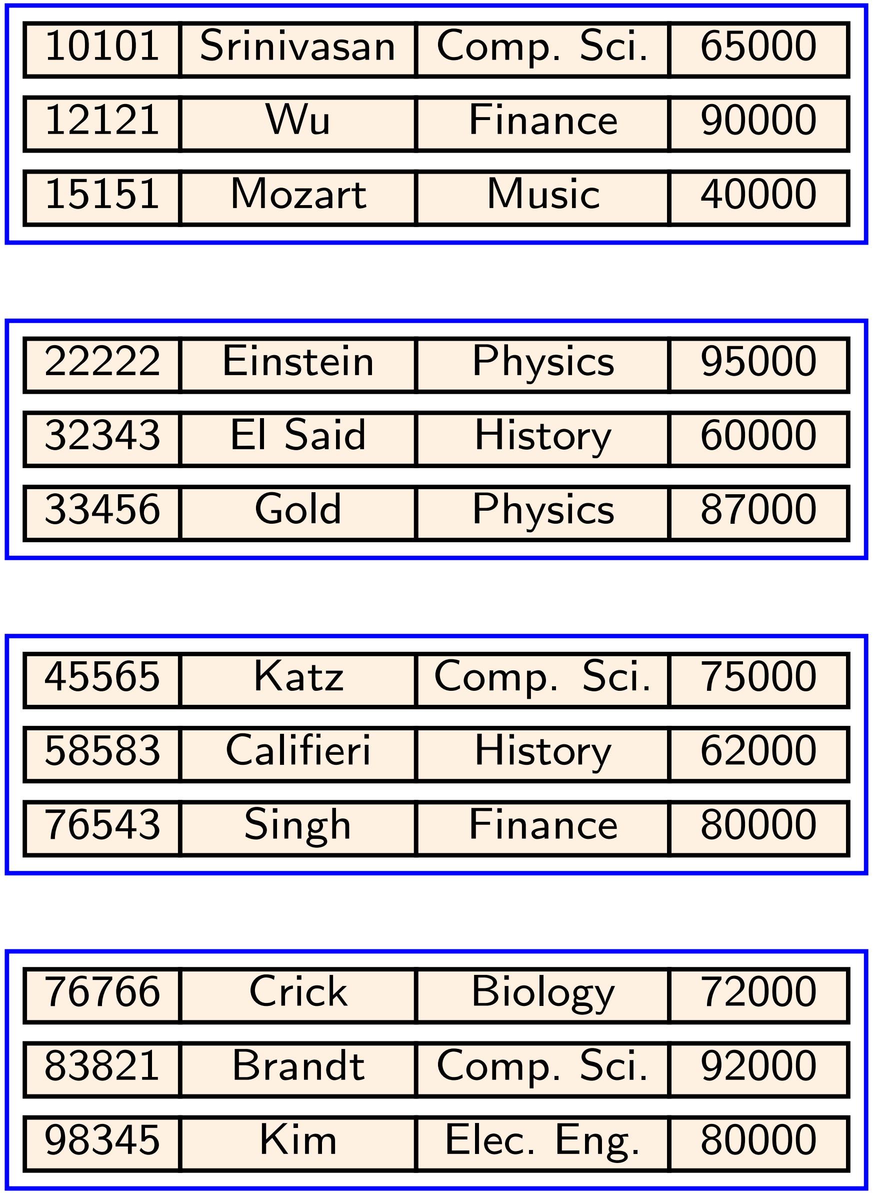
Sequential Organization

Question: What does it mean for records on disk to be ordered?

- Here ordering on the ID (first) field of the `Instructor` relation is illustrated.

Question: But the records are stored in blocks. How are the blocks ordered?

- It is true that modern hard drives use *LBA* (Logical Block Addressing), so that it is technically possible to represent the order via the disk address of the containing block.
- However this is not feasible in practice, since insertions and deletions would result in the need to move massive amounts of data.



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000

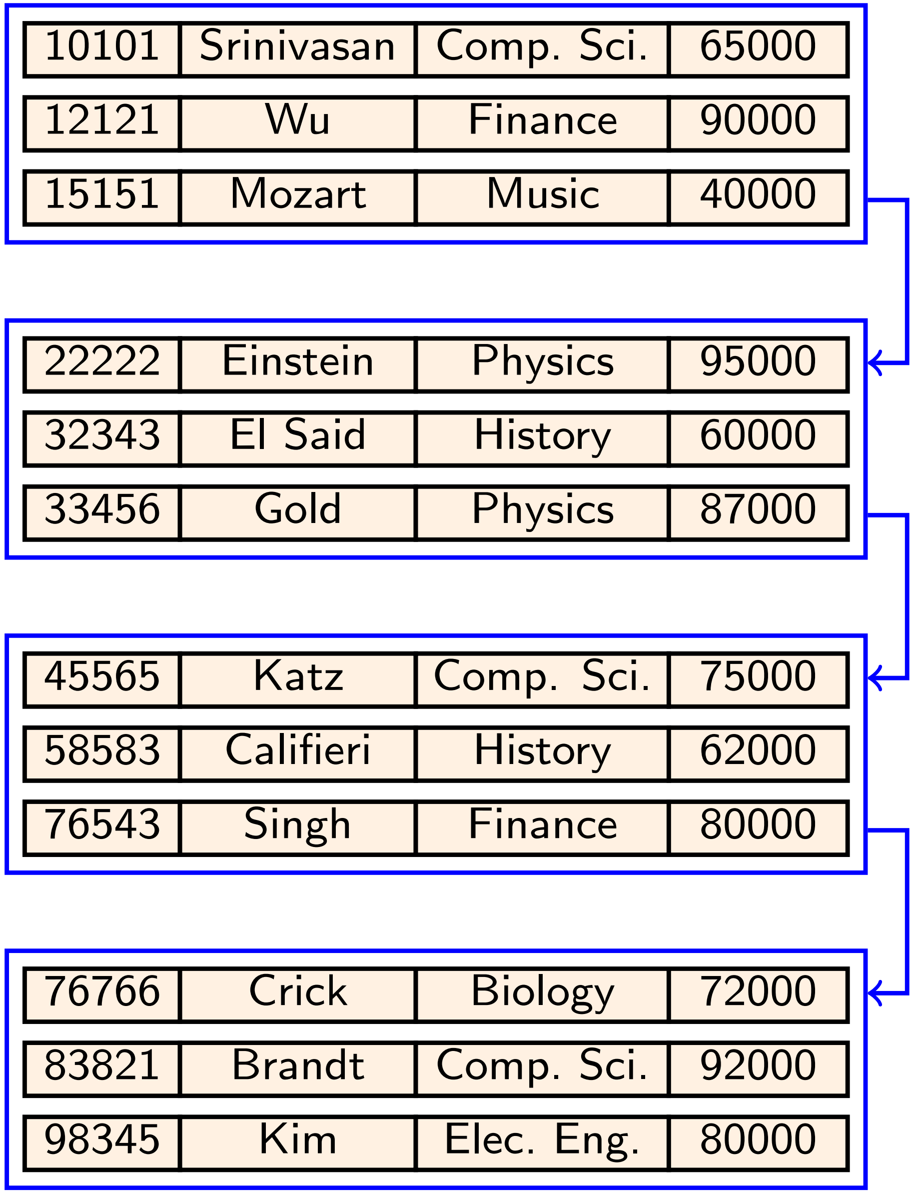
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000

76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Sequential Organization 2

- Information on the logical order of the blocks is maintained by the system,
- Here links are shown, but other ways are possible.
- As noted, the system tries to keep blocks which are logical neighbors as physical neighbors as well.
- Within each block, the entries are ordered on the selected field.



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000

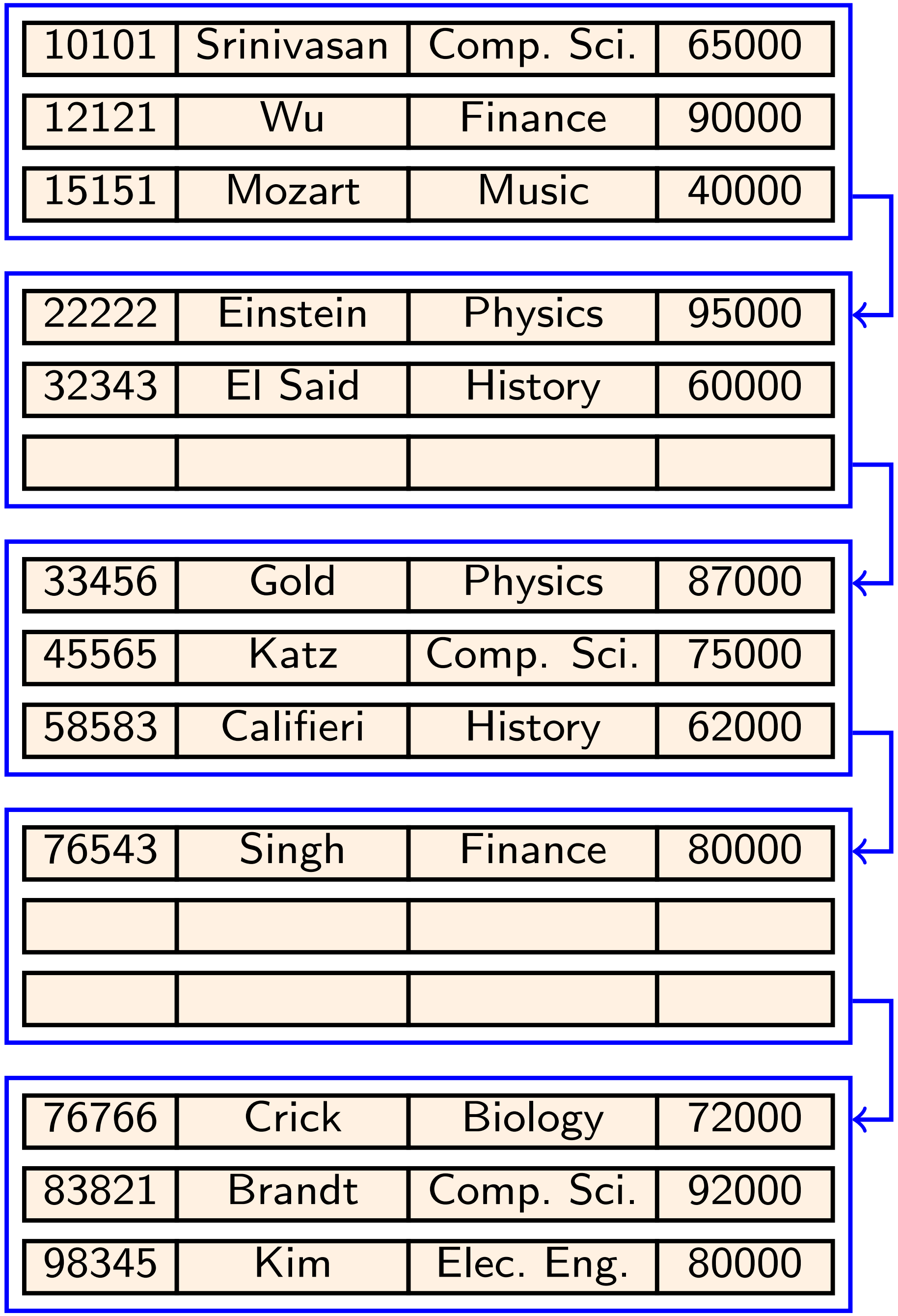
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000

76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Sequential Organization 2

- Information on the logical order of the blocks is maintained by the system,
- Here links are shown, but other ways are possible.
- As noted, the system tries to keep blocks which are logical neighbors as physical neighbors as well.
- Within each block, the entries are ordered on the selected field.
- Blocks need not be full, but there may be a requirement on how “empty” they may be.



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000

22222	Einstein	Physics	95000
32343	El Said	History	60000

33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000

76543	Singh	Finance	80000

76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Classification of Indices

Index: An index is an access structure to records.

- The elements of the index are usually ordered for easy searching.

Classification: Indices may be classified along several dimensions.

Primary vs. secondary:

Primary (or clustering): Based upon the attribute(s) used to order the records.

- Need not be built on the primary key (but often is).
- ☡ Some authors limit the term *clustering index* to indices on non-key attributes.
- These authors use the term *primary index* for clustering indices on key attributes.

Secondary (or non-clustering): Not primary.

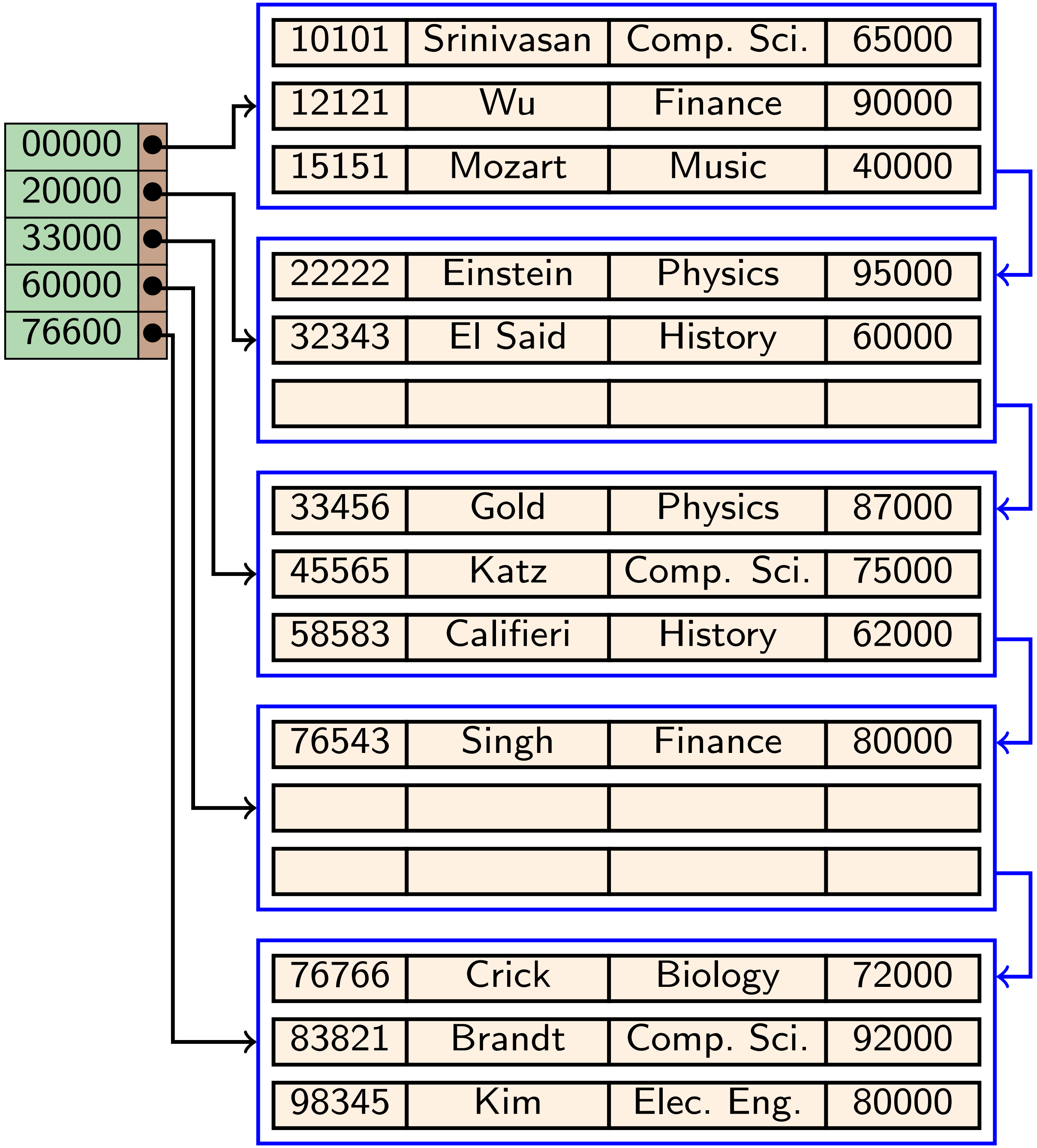
Dense vs. non-dense:

Dense: There is an index entry for each value of the search key which occurs in the file.

Non-dense (or sparse): Not dense.

A Sparse Clustering Index on the Primary Key

- The index values need not be key values of records which are currently in the database.
- Each link points to the first block containing an entry greater than or equal to the index value.
- Usually, with such a non-dense index, if an index link points to a block B, then all entries in B are greater than or equal to the index value.



00000
20000
33000
60000
76600

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000

22222	Einstein	Physics	95000
32343	El Said	History	60000

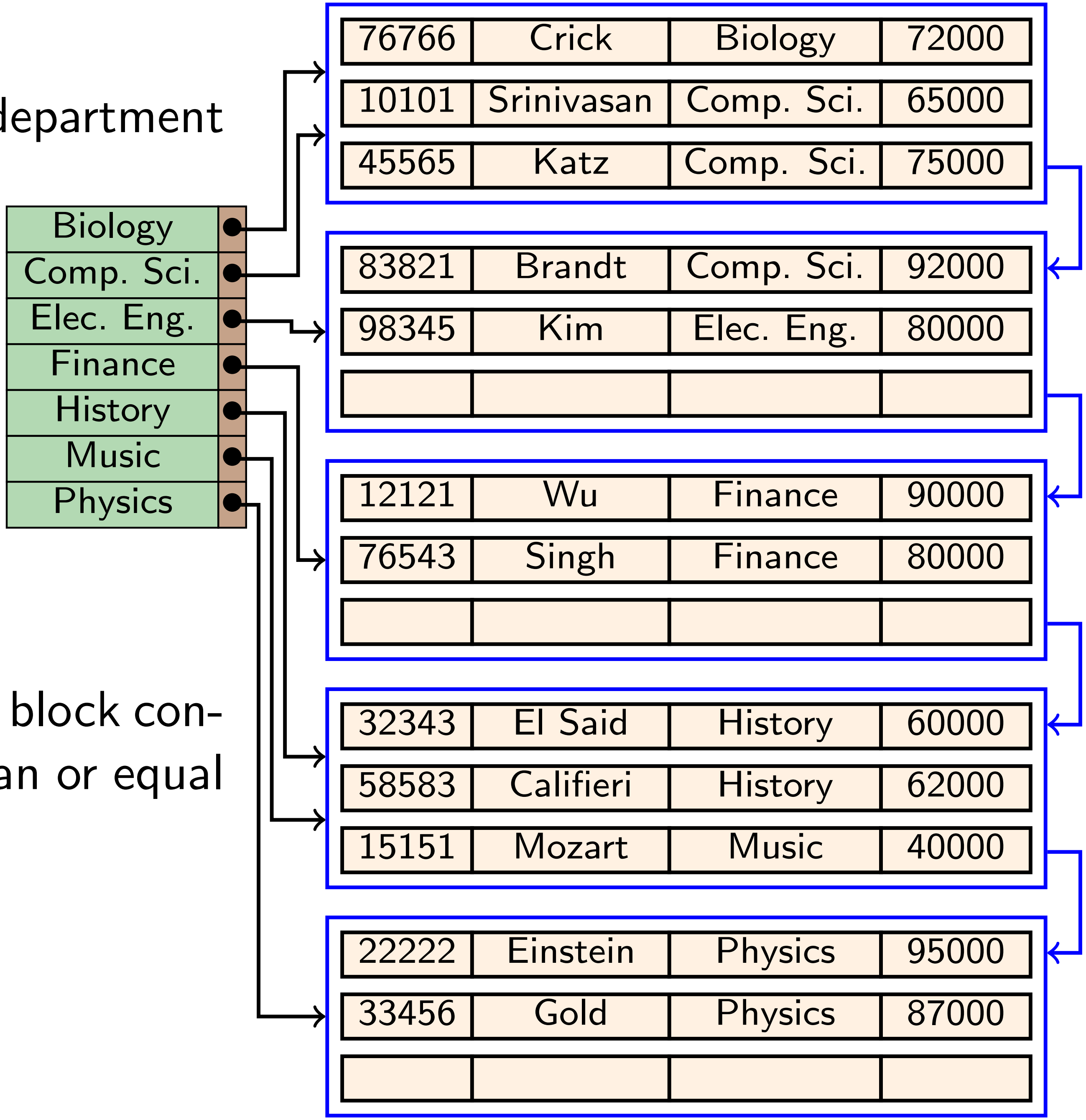
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000

76543	Singh	Finance	80000

76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

A Dense Clustering Index Not on the Primary Key

- The records are sorted by department name in this example.
- There is an index entry for every department name which occurs in the database, but not for every possible department name.
- Each link points to the first block containing an entry greater than or equal to the index value.



Biology	●
Comp. Sci.	●
Elec. Eng.	●
Finance	●
History	●
Music	●
Physics	●

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000

83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

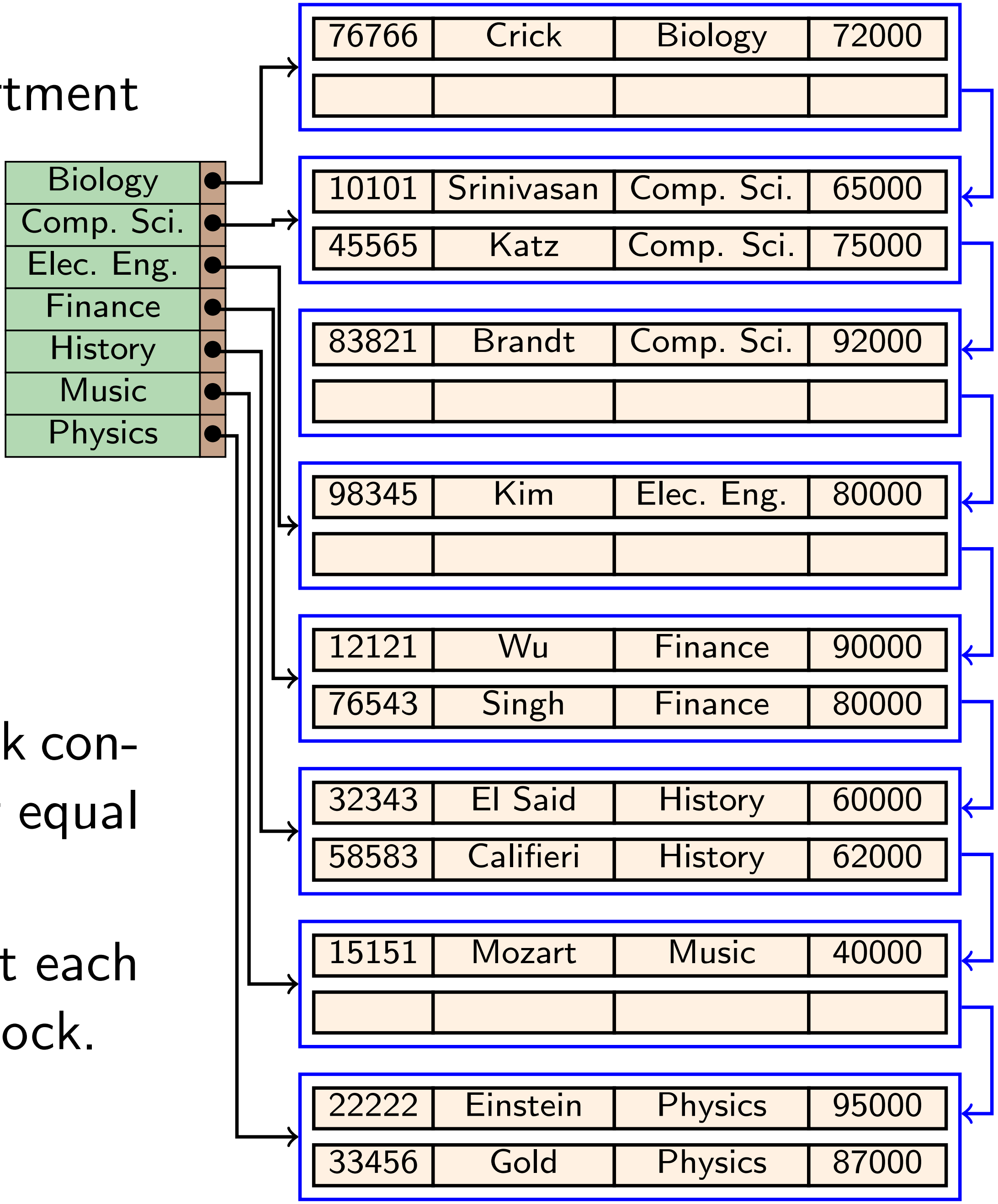
12121	Wu	Finance	90000
76543	Singh	Finance	80000

32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000

22222	Einstein	Physics	95000
33456	Gold	Physics	87000

A Dense Clustering Index Not on the Primary Key

- The records are sorted by department name in this example.
- There is an index entry for every department name which occurs in the database, but not for every possible department name.
- Each link points to the first block containing an entry greater than or equal to the index value.
- It is also possible to require that each new index value begin a new block.

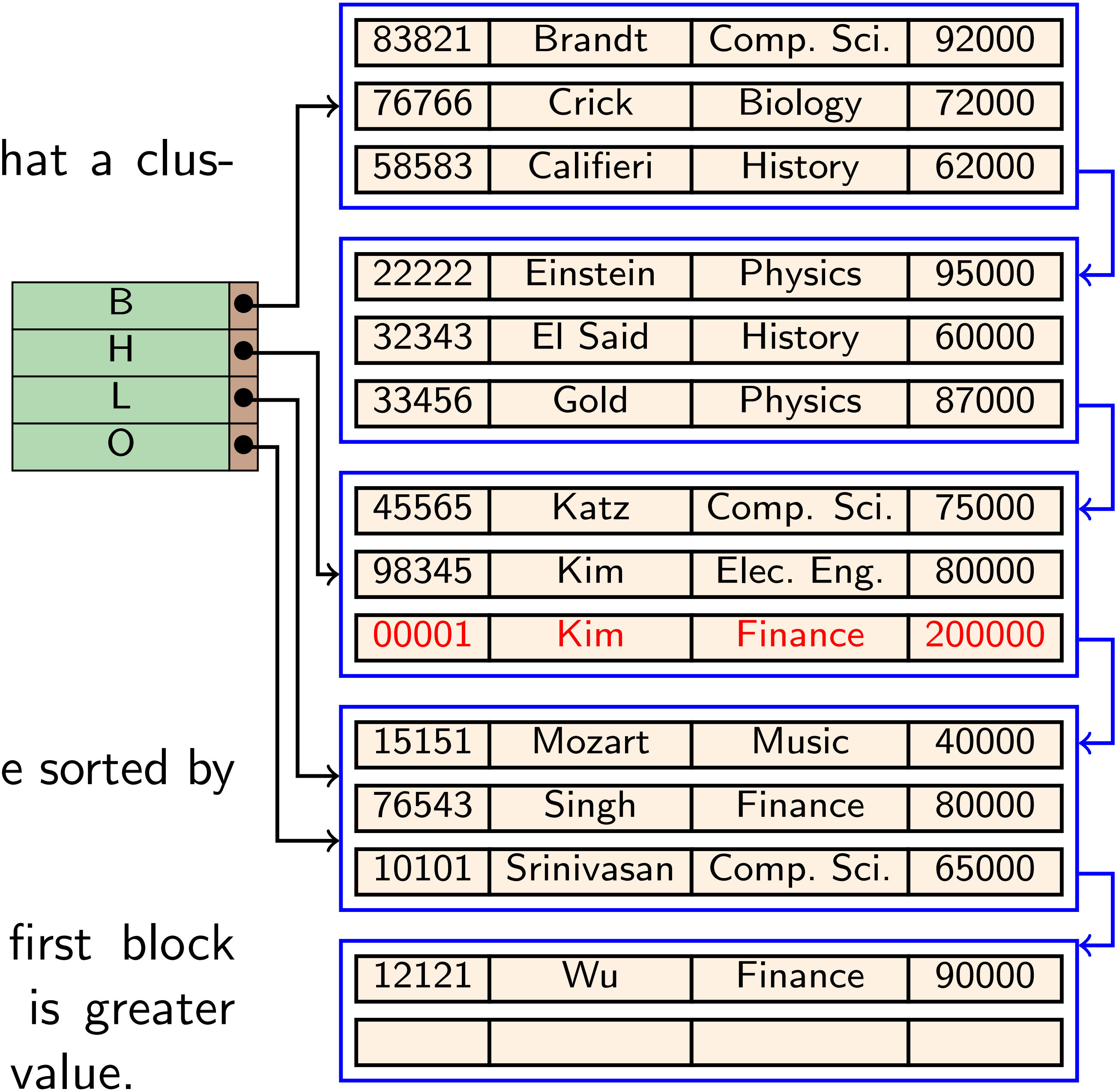


Biology
Comp. Sci.
Elec. Eng.
Finance
History
Music
Physics

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

A Sparse Clustering Index on a “Near” Key

- There is no requirement that a clustering index be on a key.
- In particular, if the field on which the records are sorted is “almost” a key, then a non-dense clustering index may be useful.
- The records to the right are sorted by instructor name.
- The index points to the first block containing a record which is greater than or equal to the index value.



B
H
L
O

83821	Brandt	Comp. Sci.	92000
76766	Crick	Biology	72000
58583	Califieri	History	62000

22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000

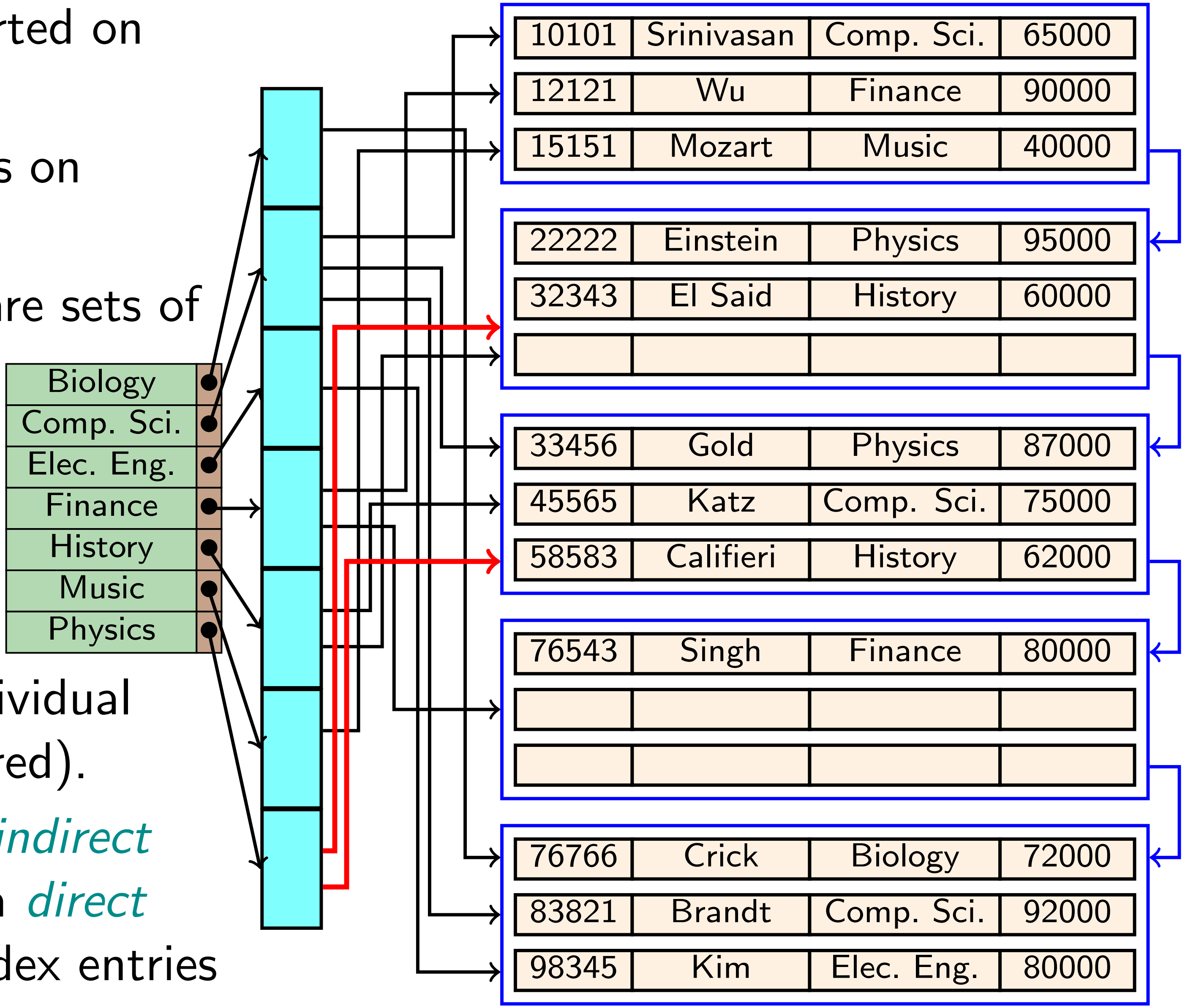
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
00001	Kim	Finance	200000

15151	Mozart	Music	40000
76543	Singh	Finance	80000
10101	Srinivasan	Comp. Sci.	65000

12121	Wu	Finance	90000

A Non-Clustering Index

- The example file is sorted on employee ID.
- The secondary index is on department.
- The blocks in aqua are sets of pointers for the given value of the index attribute.
- Note that a pointer from such a set leads to a block, not an individual record. (Examples in red).
- This is also called an *indirect* index, as opposed to a *direct* index, in which the index entries point directly to the record blocks.



Biology
Comp. Sci.
Elec. Eng.
Finance
History
Music
Physics

10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000

22222	Einstein	Physics	95000
32343	El Said	History	60000

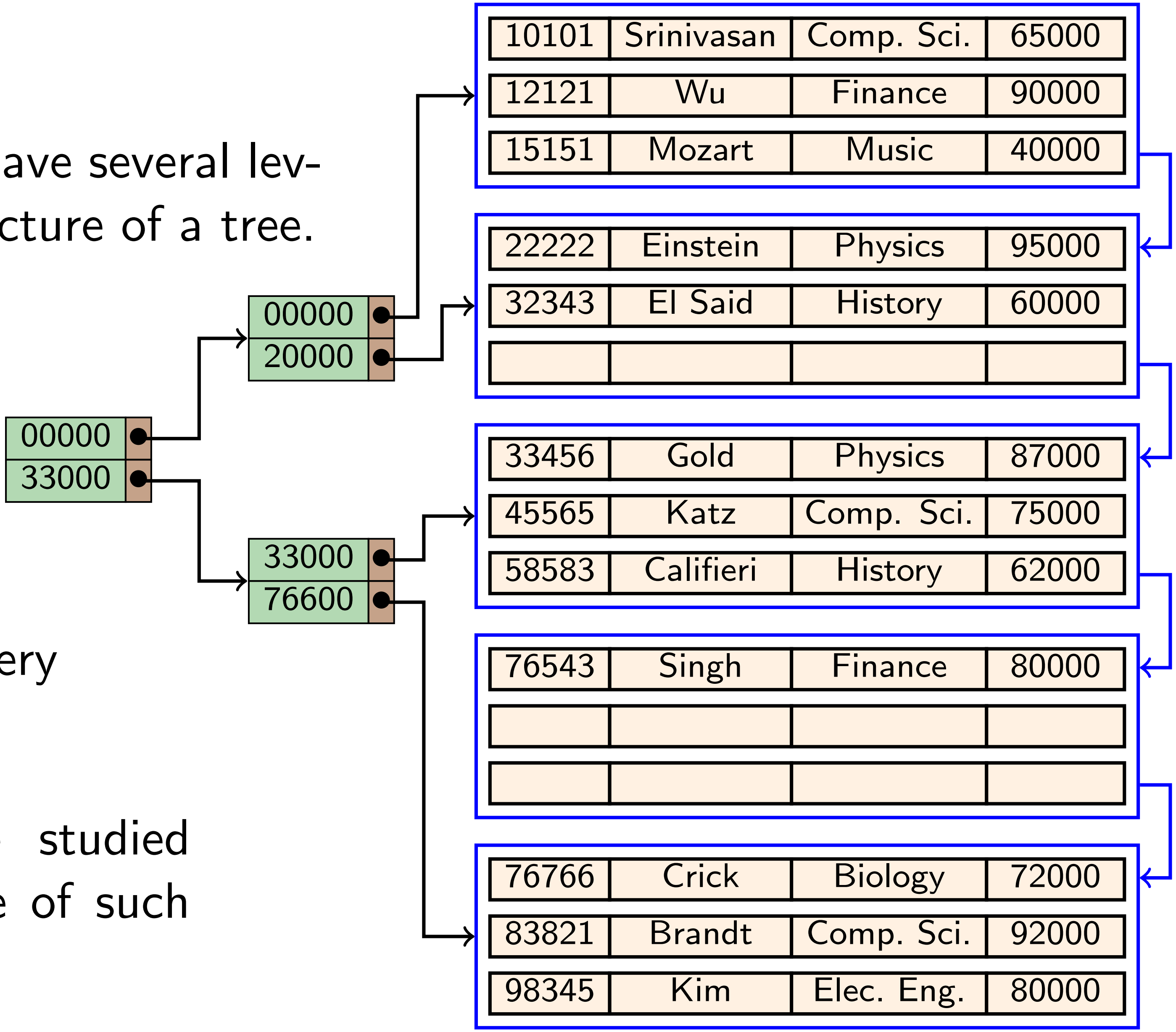
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000

76543	Singh	Finance	80000

76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

A Multi-Level Index

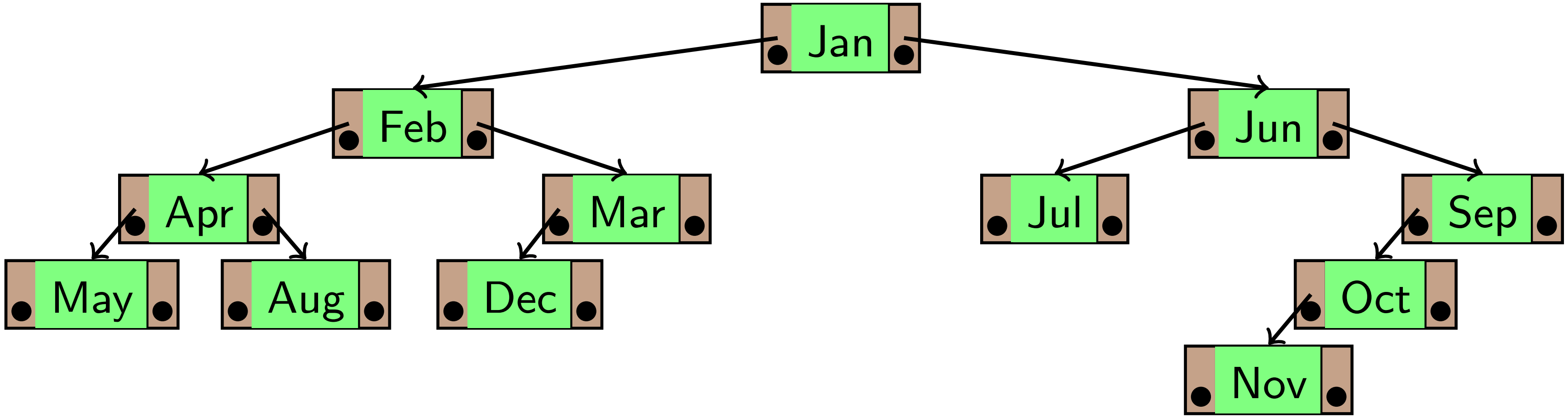
- The index itself may have several levels, usually in the structure of a tree.
- Illustrated here is a multi-level nondense primary index on the instructor ID.
- Such indices are very common.
- The B^+-tree, to be studied shortly, is an example of such an index structure.



B-Trees and B^+-Trees

- The most important form of index structure in database systems is the *B^+-tree*.
- While it is possible to present B^+-trees directly (as does the textbook), the easiest way to understand B^+-trees is to understand *B-trees* first.
- B-trees are a direct extension of the classical and ubiquitous binary search tree (which everyone in this class should already know.)

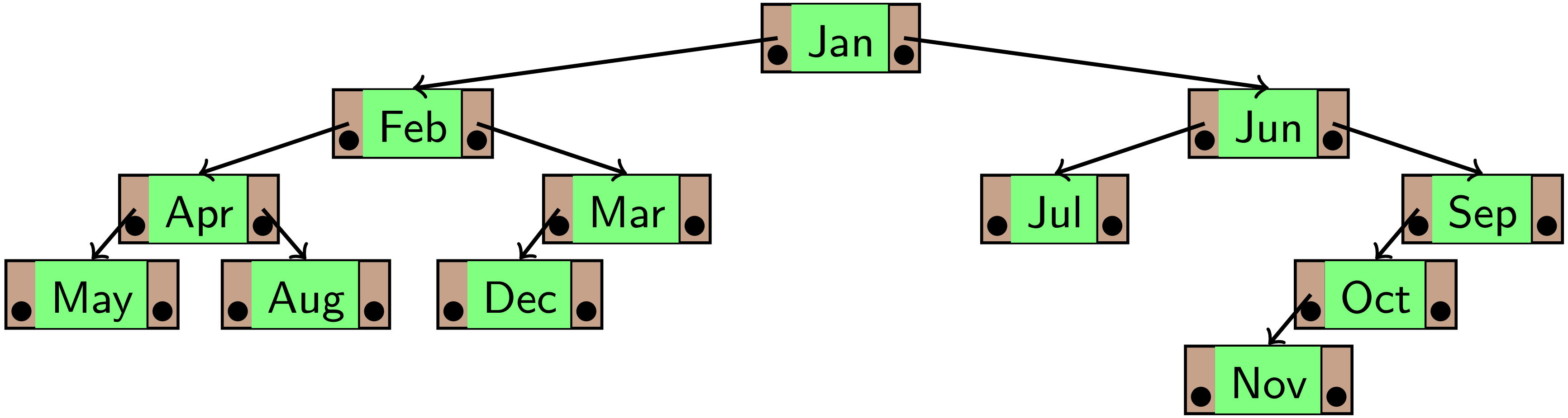
Review of Binary Search Trees



- Shown above the binary tree obtained by inserting, into an initially empty tree, the three-letter abbreviations for the months, in chronological order.

- The method of search is standard:
  - Begin at the root.
  - If the element is found, stop.
  - Otherwise, go left if the item sought is less than the value of the current vertex, otherwise go right.
  - Repeat until found or an empty pointer is reached.

Shortcomings of Binary Trees for Database Storage



- Binary search trees have two shortcomings which renders them a poor choice for database storage.

No guaranteed balance: Binary search trees need not be balanced, and unless special measures are taken, can grow far out of balance.

  - Lack of balance can lead to long searches, with even average case time $O(n)$ rather than $O(\log(n))$, $n =$ number of vertices.

Much pointer following: One pointer must be followed for each decision in the search process.

  - In the DBMS context, following a pointer often involves a disk read, rendering the approach unusably slow.

B-Trees to the Rescue

- B-trees are designed to overcome these shortcomings of the traditional binary search tree in two ways.

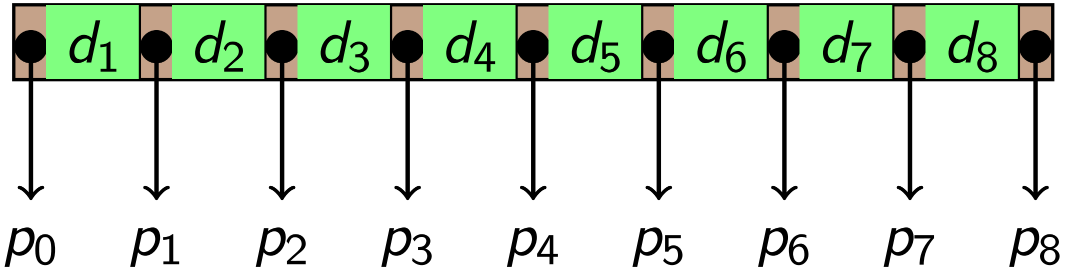
Guaranteed balance: In a B-tree every path from the root to a leaf has exactly the same length.

  - A search is thus guaranteed to run in worst-case time $O(\log(n))$, with n the number of data items stored in the tree.

Multiple data items per vertex: Instead of storing only one data item per vertex, in a B-tree many data items may be stored in the same vertex.

  - This leads to searches which require far fewer pointers chases, and consequently far fewer disk accesses.

The Structure of a Vertex of a B-tree



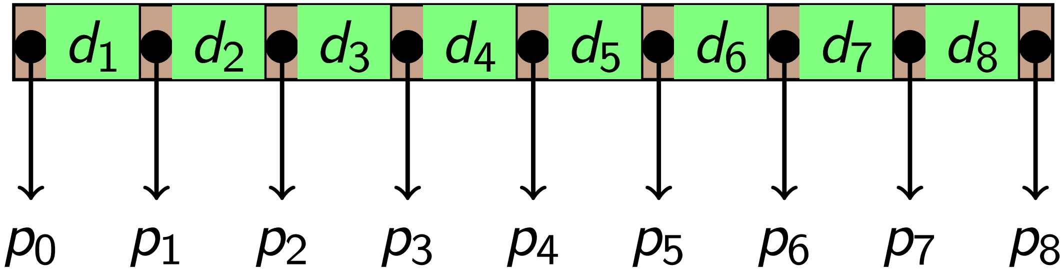
- A vertex of a B-tree is a generalization of that of a binary search tree.
- A vertex of a B-tree of *order* n has n pointers and d $n - 1$ data fields.
- The form for $n = 9$ is depicted above.
- A B-tree is a *rooted* tree, just as is a binary search tree.

Some authors define the order to be $\lfloor n/2 \rfloor$ relative to the above definition.

  - The definition of order used here coincides with that of Knuth (Vol. 3 of *The Art of Computer Programming*).
  - The other definition leads to ambiguities in maximum size.

- The conditions on a B-tree are more complex than those of a binary search tree, and are described next.

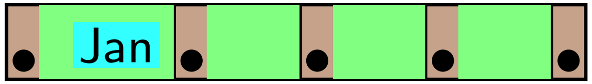
Conditions on a B-tree of Order n



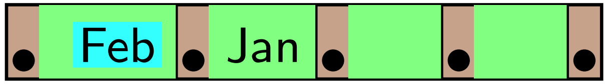
- Each pointer and each data field is either *used* or *unused*.
- Both pointers and data field are used from left to right:
  - There is a k, $1 \leq k \leq n$, such that p_i and d_i are used iff $i \leq k$.
- Every vertex, except the root, must be at least half full: $k \geq \lfloor (n-1)/2 \rfloor$.
- The root must contain at least one data value: $k \geq 1$.
- The data elements in a given vertex are in sort order, from left to right.
- All used pointer fields of a leaf vertex are null.
- For an internal vertex, each used pointer p_j must point to another vertex of the tree, with all used data fields d in the subtree satisfying $d_j < d < d_{j+1}$.
  - To make this work, take the fictitious data fields d_0 and d_{n+1} to contain the largest and smallest possible values, respectively.
- The tree is balanced; all paths from the root to a leaf are the same length.

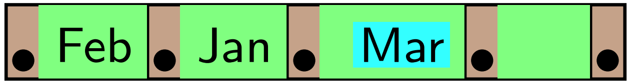
A Simple Example of Repeated Insertion into a B-tree

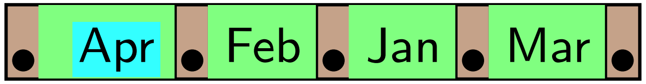
- The operations on a B-tree are best learned by example.
- In this example, the three-letter abbreviations for the months of the year will be inserted, in chronological order, into a B-tree of order four.
- Formally, there is no such thing as an empty B-tree, so begin with the tree containing just Jan:



- The insertions of Feb, Mar, and Apr are straightforward, with the inserted element shaded in aqua:

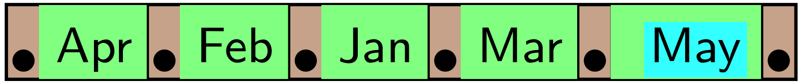




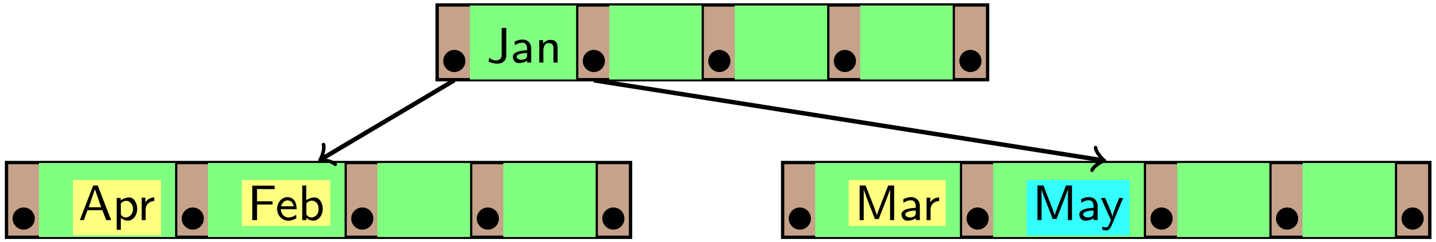


An Simple Example of Repeated Insertion into a B-tree — 2

- Insertion of May using this method would require a B-tree vertex of order five, which lies outside of the model being used.



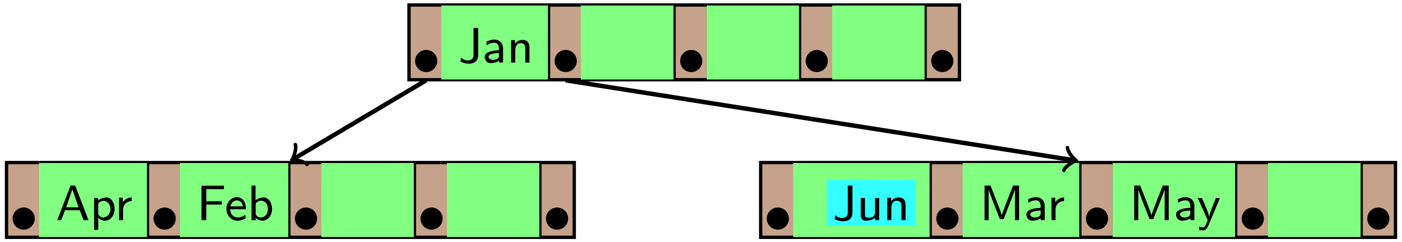
- The solution is to *split* this fictitious vertex, retaining the middle element as the sole value of the new root, with two half-full children:

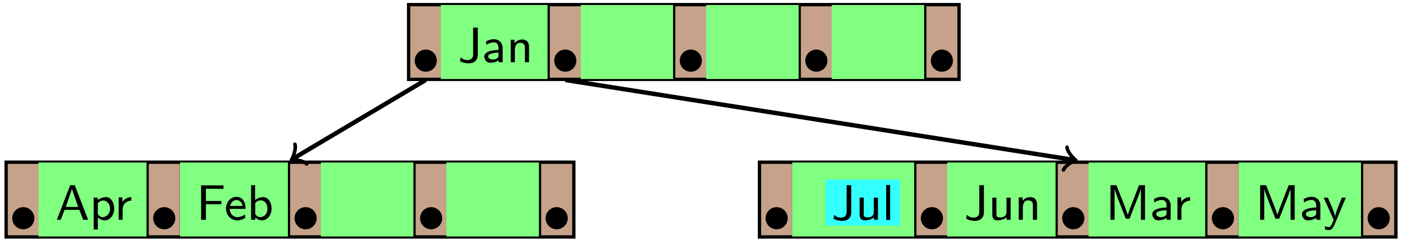


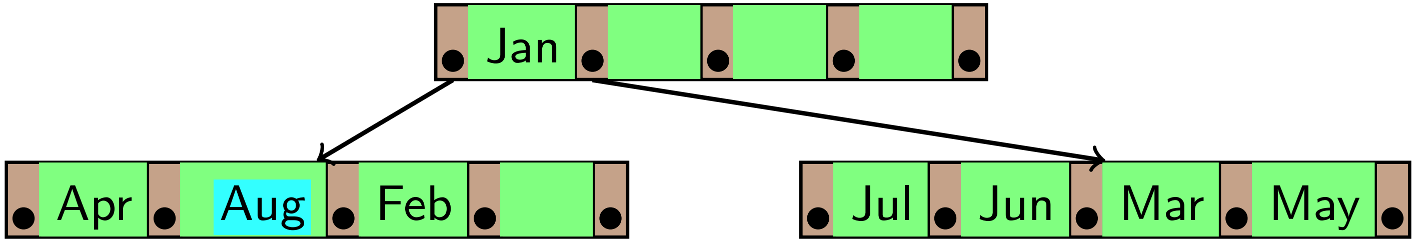
- The values marked in yellow are moved to a different vertex in the process.

An Simple Example of Repeated Insertion into a B-tree — 3

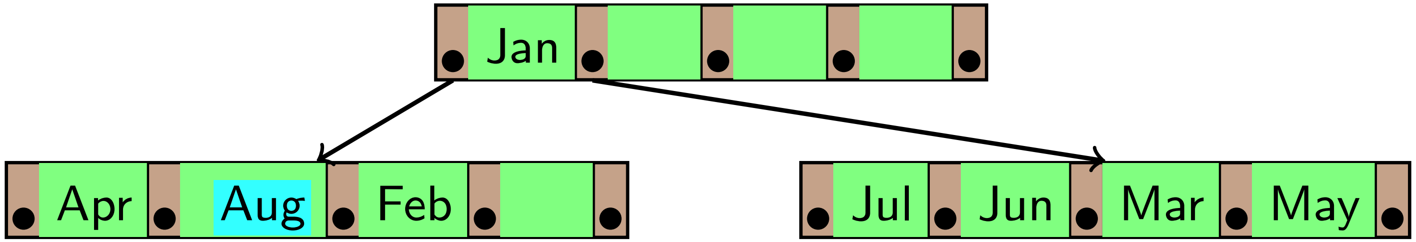
- The insertions of Jun, Jul, and Aug are simple leaf insertions.



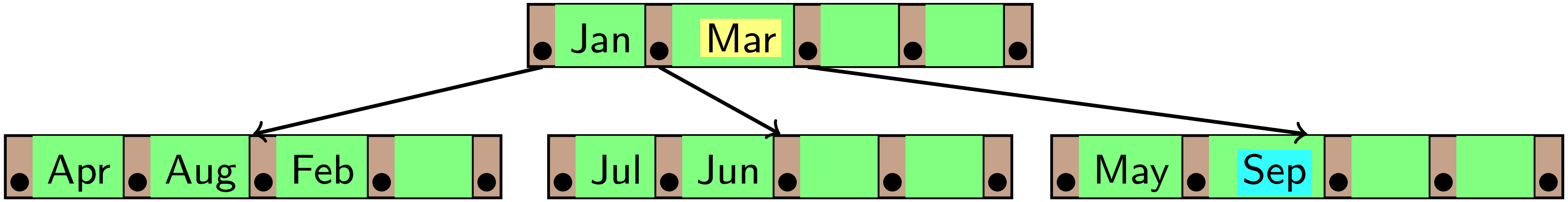




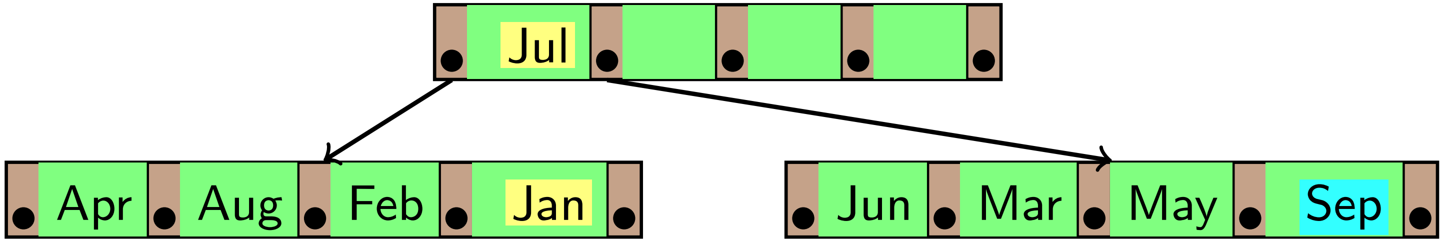
An Simple Example of Repeated Insertion into a B-tree — 4



- There are two possibilities for the insertion of Sep.
- The first is to do a split of the full vertex, moving the middle element to the parent.

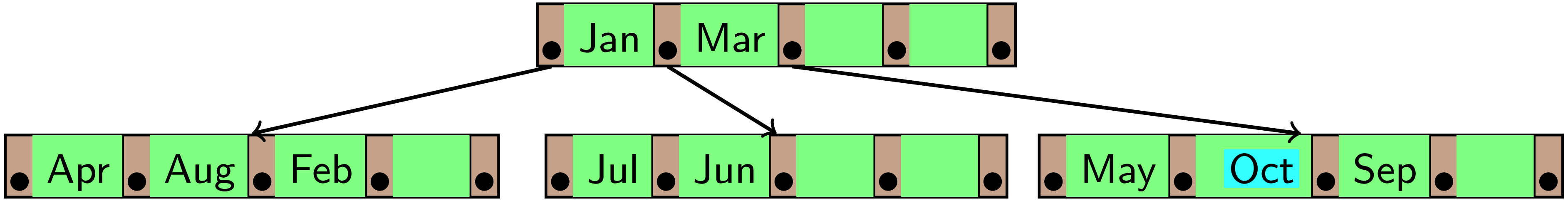


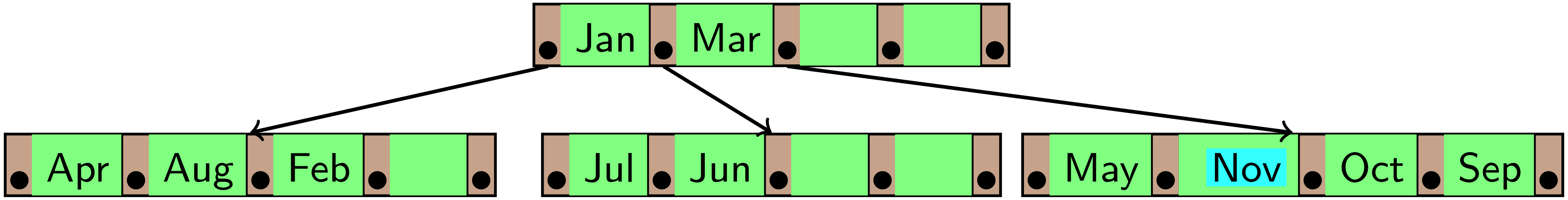
- The second performs a *rotation* of values, through the parent to the left sibling.

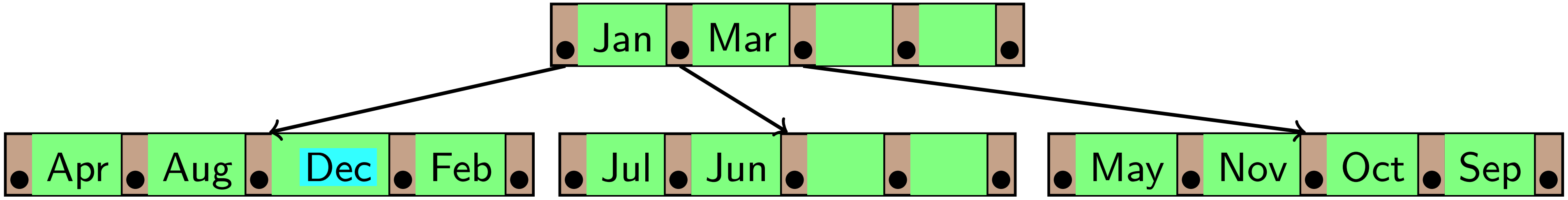


An Simple Example of Repeated Insertion into a B-tree — 5

- The insertions of Oct, Nov, and Dec are simple leaf insertions to the first alternative on the previous slide.

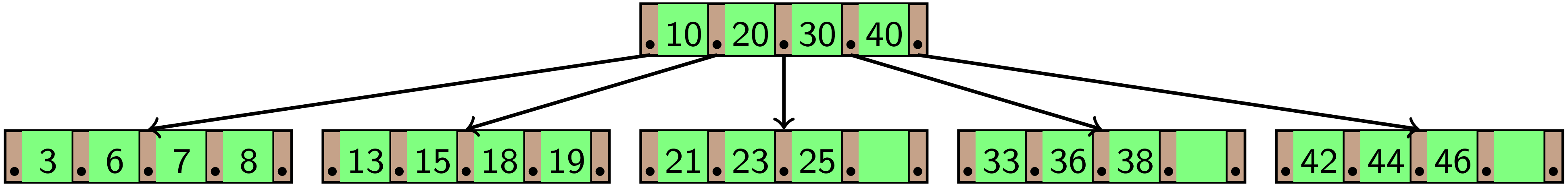




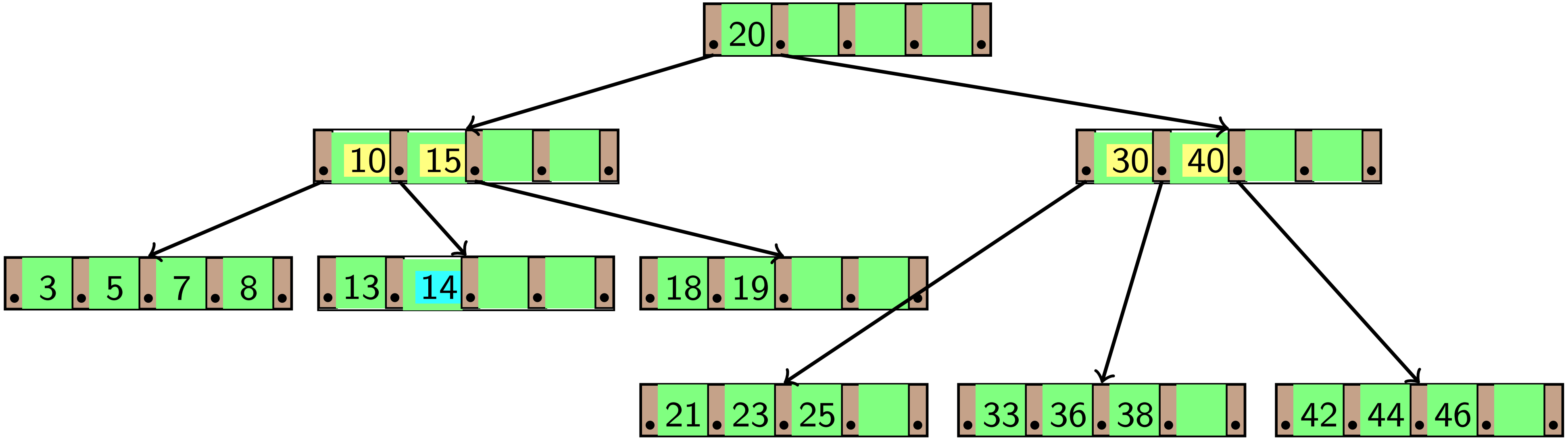


Insertions on B-Trees Involving Root Splitting

- Insertion of 14 into the following B-tree implies a split of the second child from the left.



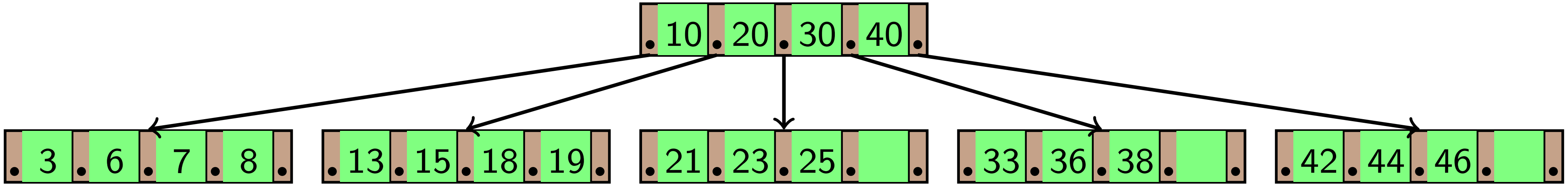
- This in turn forces a split of the root.



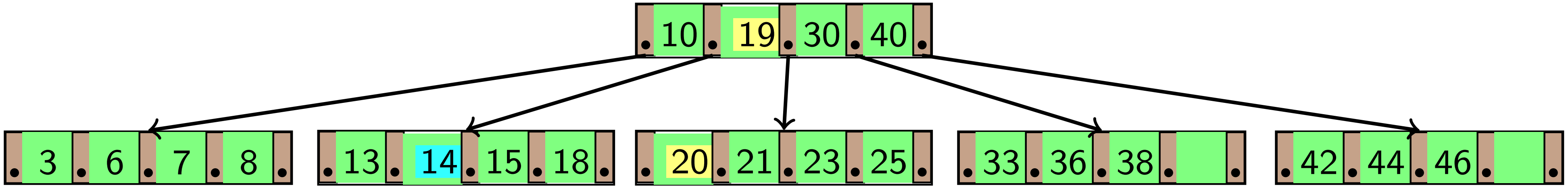
- Such splits of the root are the only way in which a B-tree can grow in depth, and guarantee that it remains balanced.

Insertions on B-Trees Realized via Redistribution

- Insertion of 14 into the following B-tree implies a split of the second child from the left.



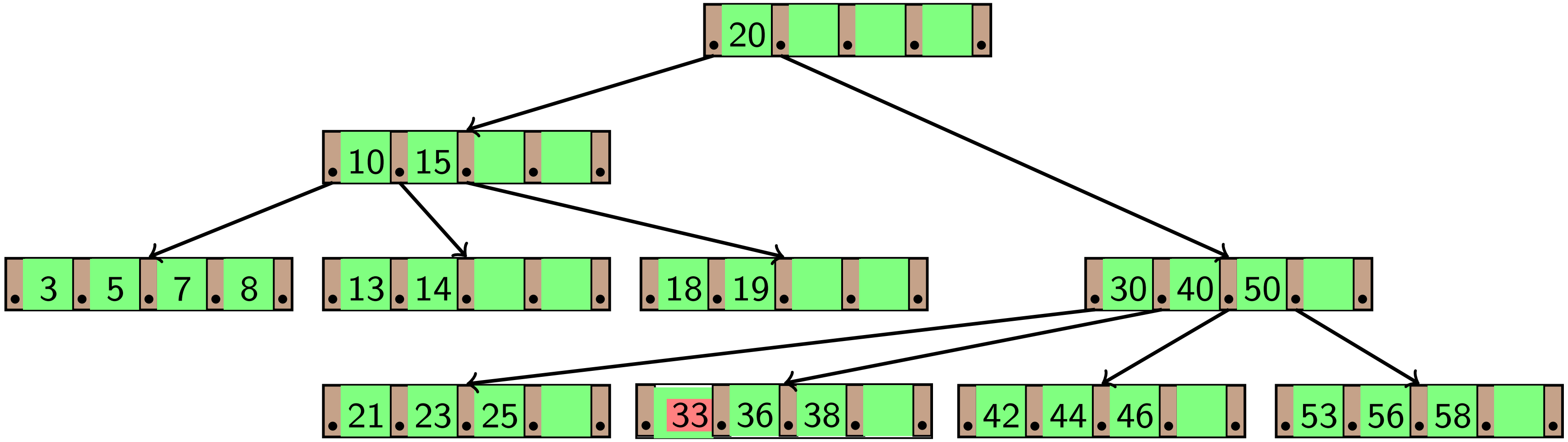
- In this case, insertion of 14 could also be realized by a redistribution of values, without splitting any vertices.



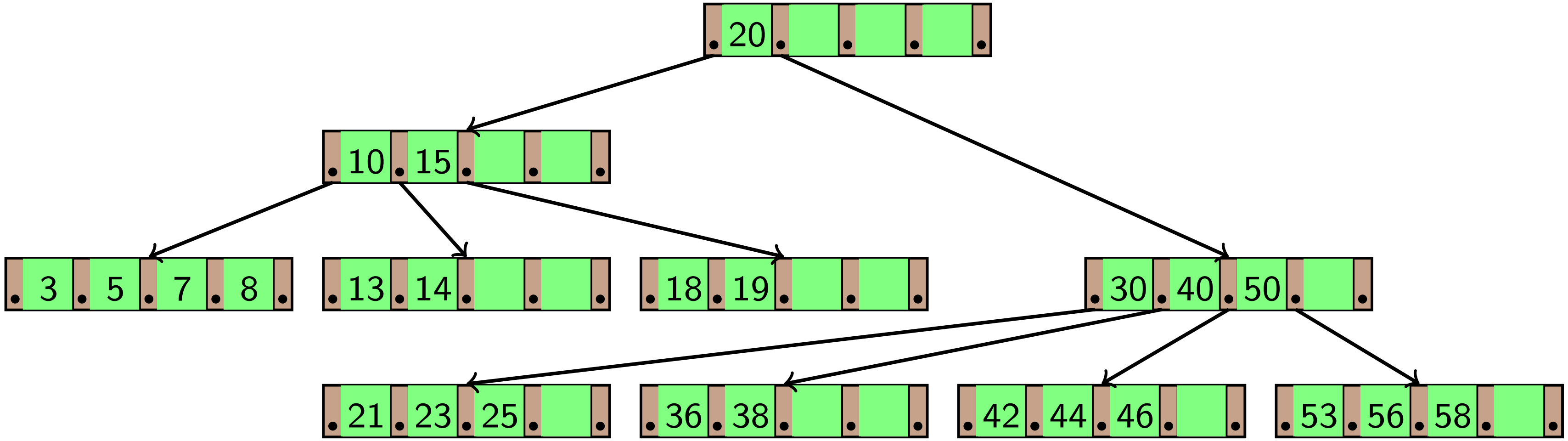
- The choice of strategy is more of a heuristic than a hard-and-fast rule.

Simple Deletions on B-trees

- Consider the deletion of 33 from the following B-tree:

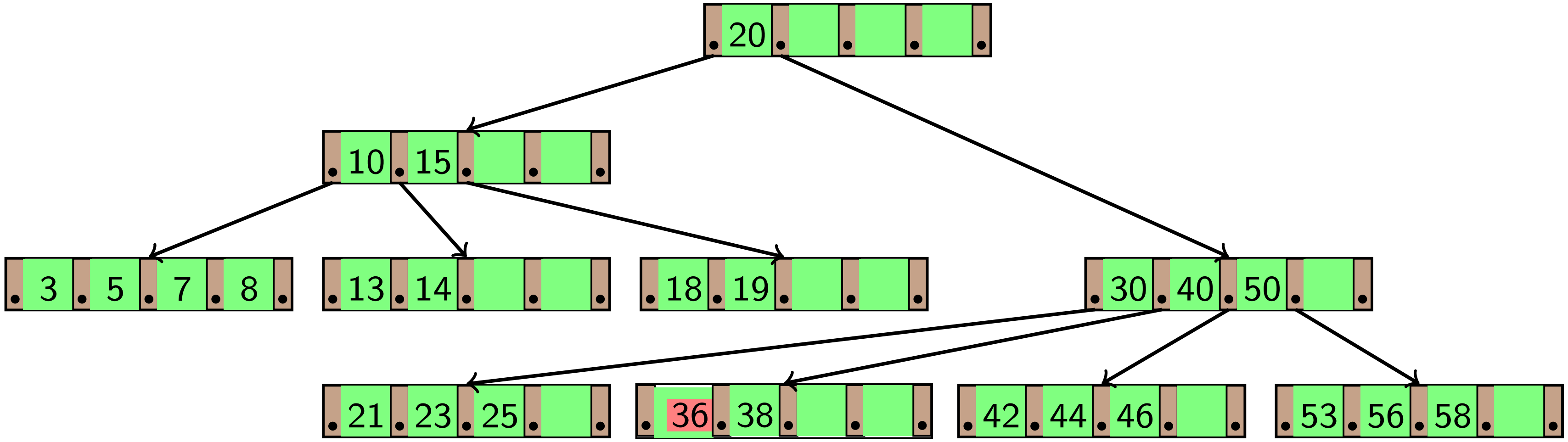


- It is a simple matter, since there is no underfill.

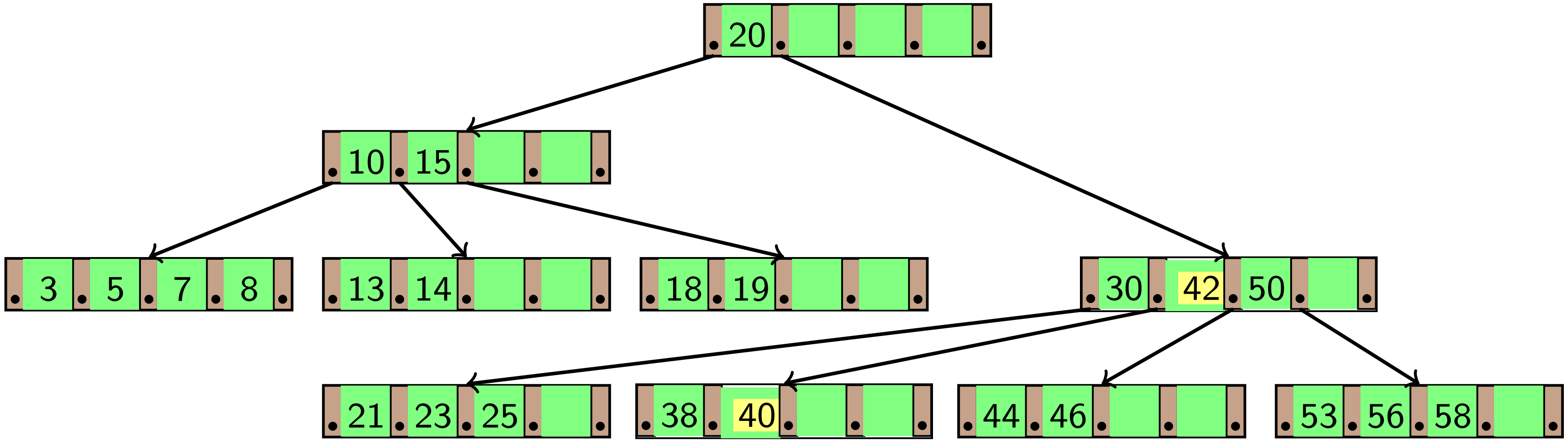


Deletions on B-trees — Underfill Solved via Redistribution

- The subsequent deletion of 36 results in an vertex with too few values:



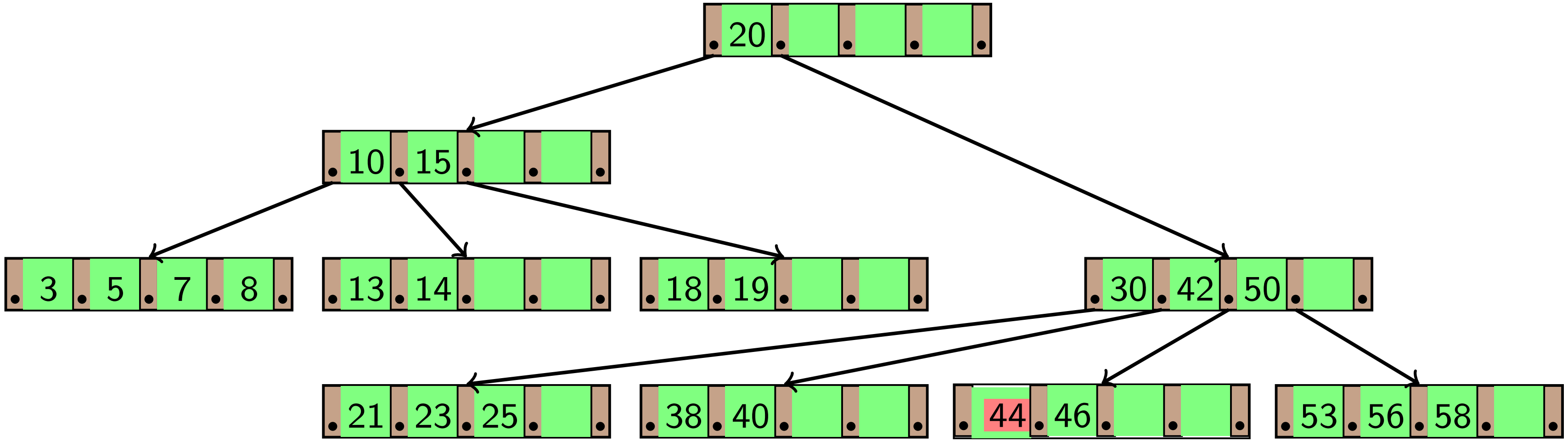
- which may be remedied via a redistribution:



Deletions on B-trees with a Choice of Solutions

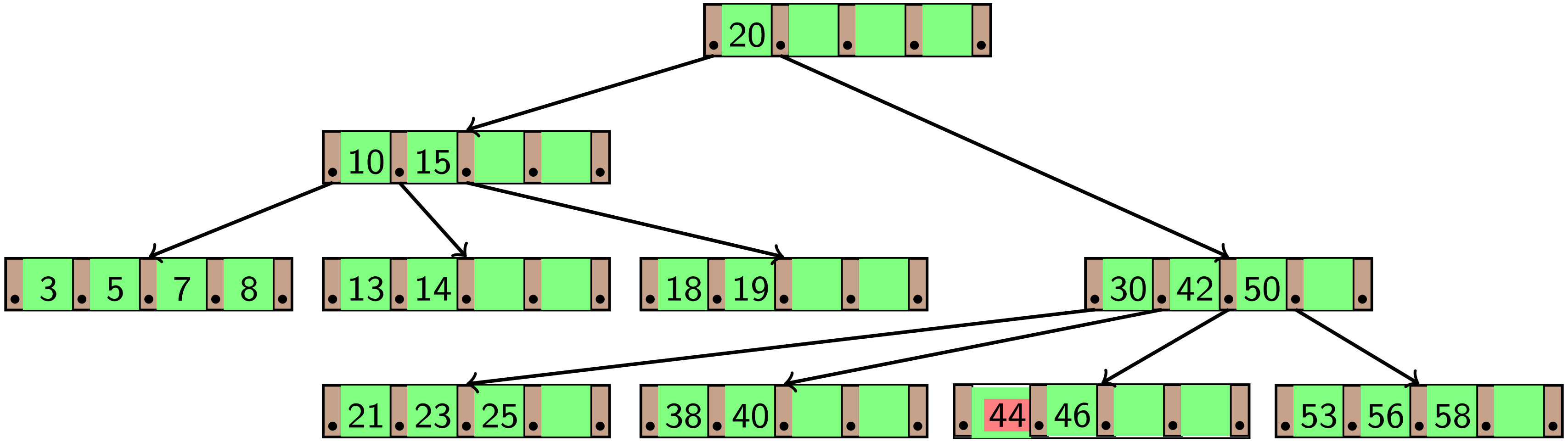
- Sometimes, there is a choice between a redistribution and a combination of vertices.

- Continue with the result of the previous deletion, this time with the further deletion of 44.

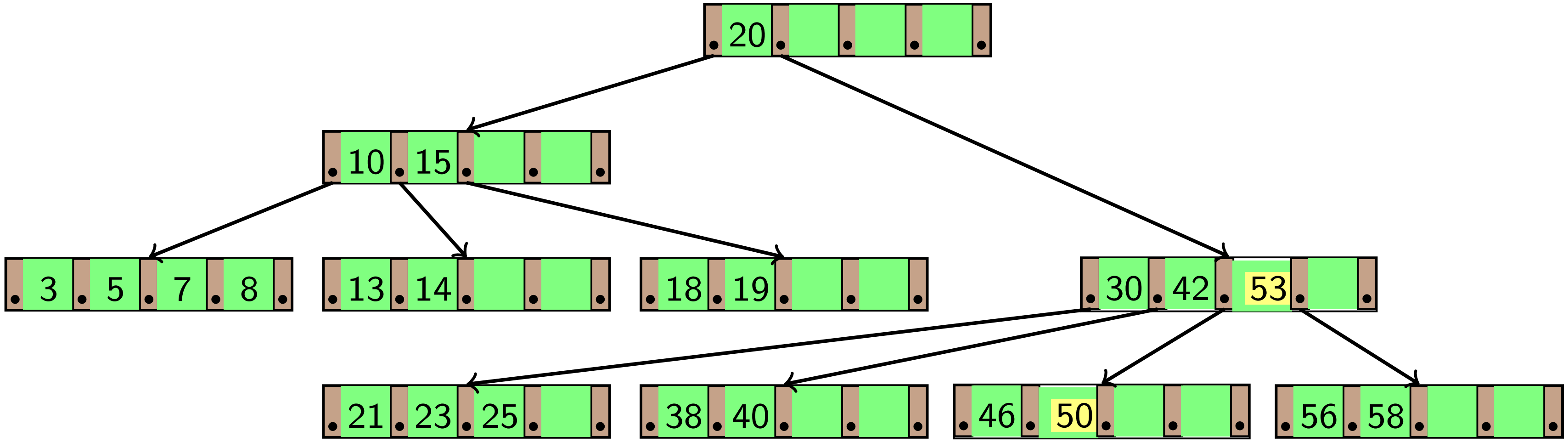


- There are two ways to support this update, as shown on the following two slides.

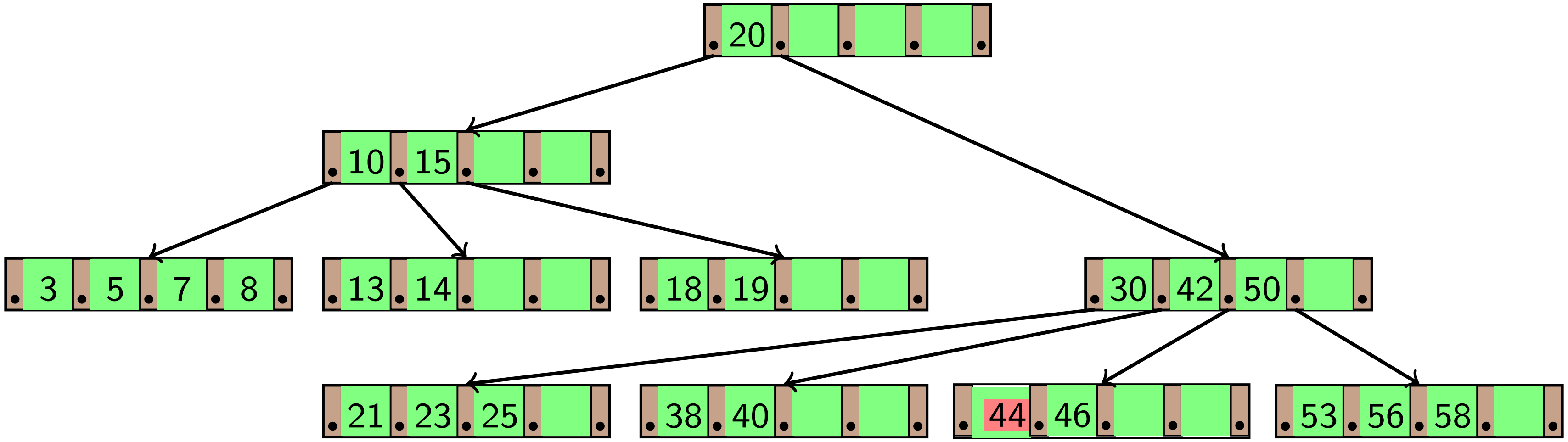
Deletions on B-trees with a Choice of Solutions — 2



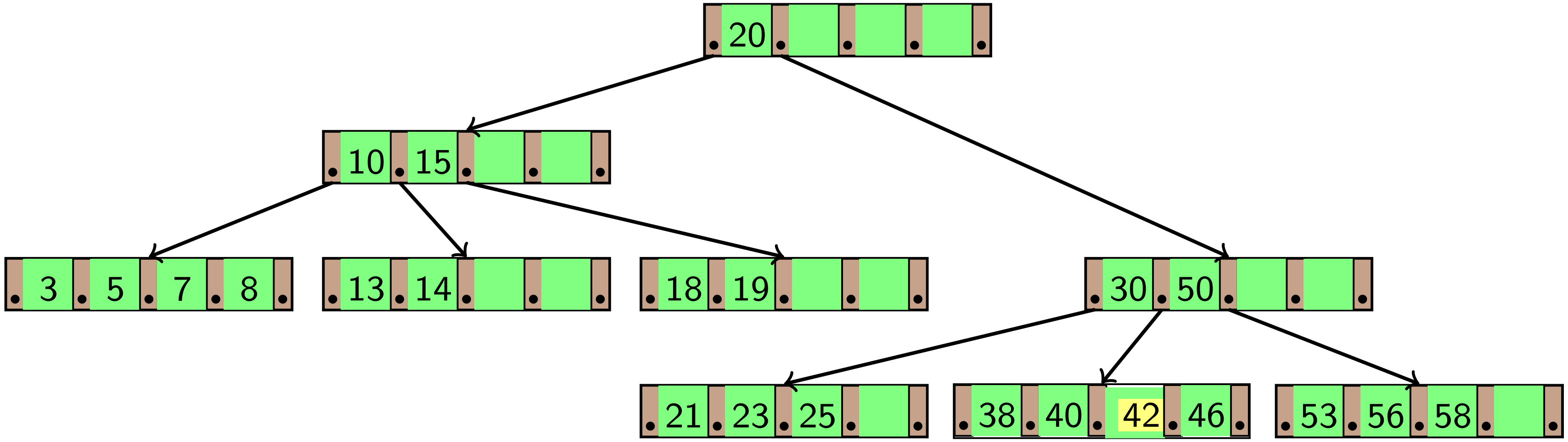
- The first solution involves a redistribution, much as in the previous example.



Deletions on B-trees with a Choice of Solutions — 3

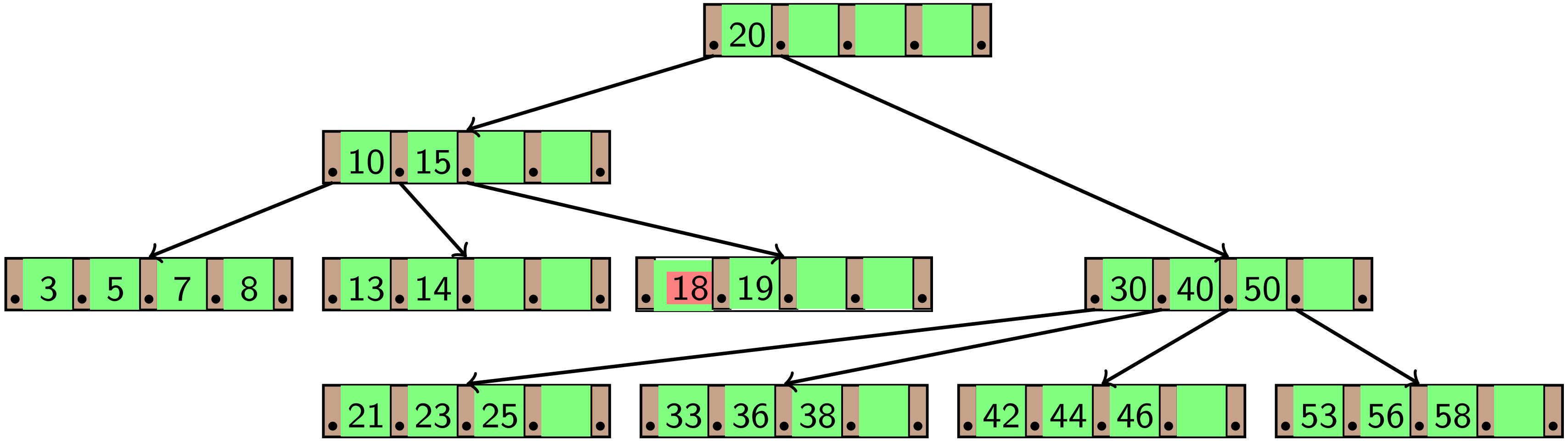


- The second solution involves a combination of the underfull vertex with its sibling, together with the movement one data field down from the parent.

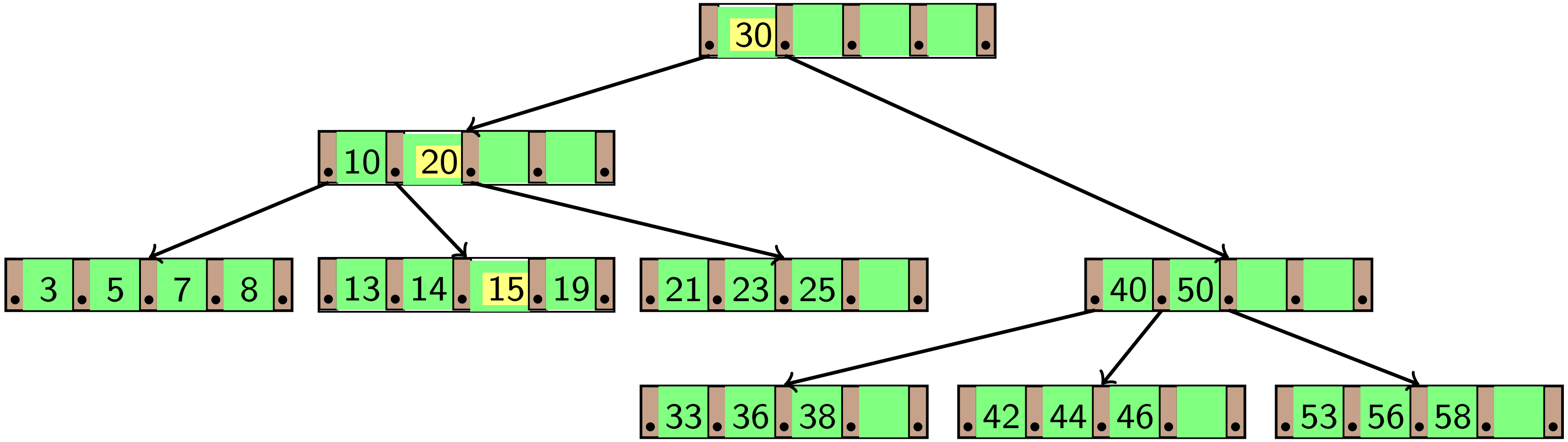


Deletions on B-trees Involving Redistribution through the Root

- Consider deleting 18 from the following B-tree:



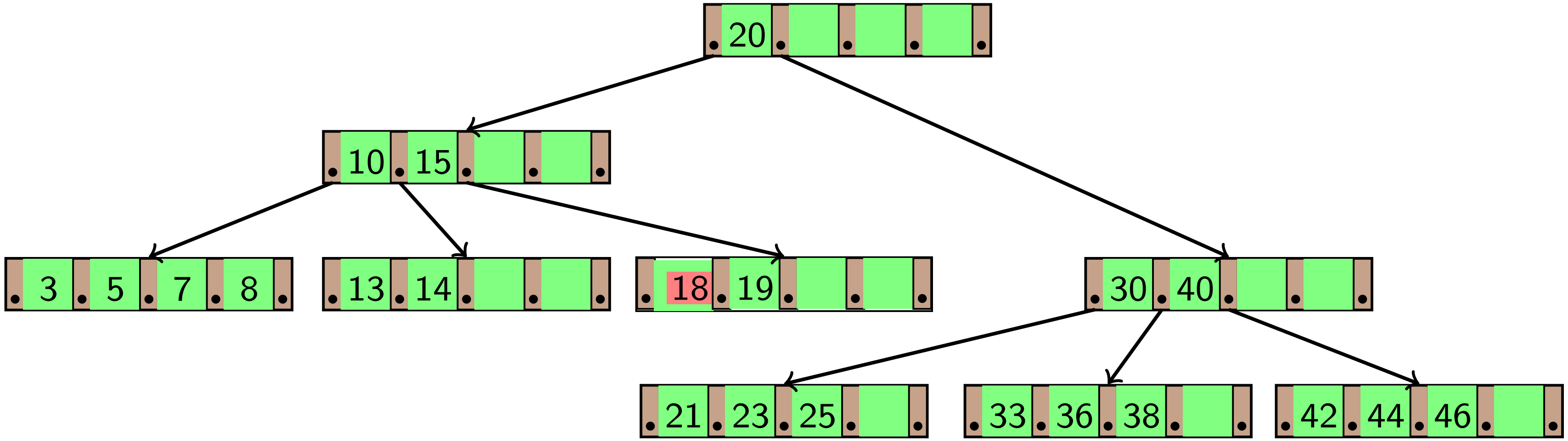
- This may be realized via redistribution up through the root.



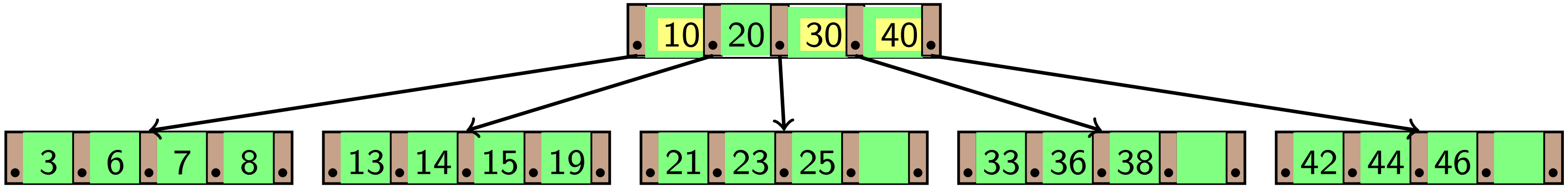
- Notice the movement of the 21-23-25 vertex.

Deletions on B-trees Requiring Depth Reduction

- Deletion of 18 from the following B-tree requires a height adjustment (unless very long-range moves are permitted).



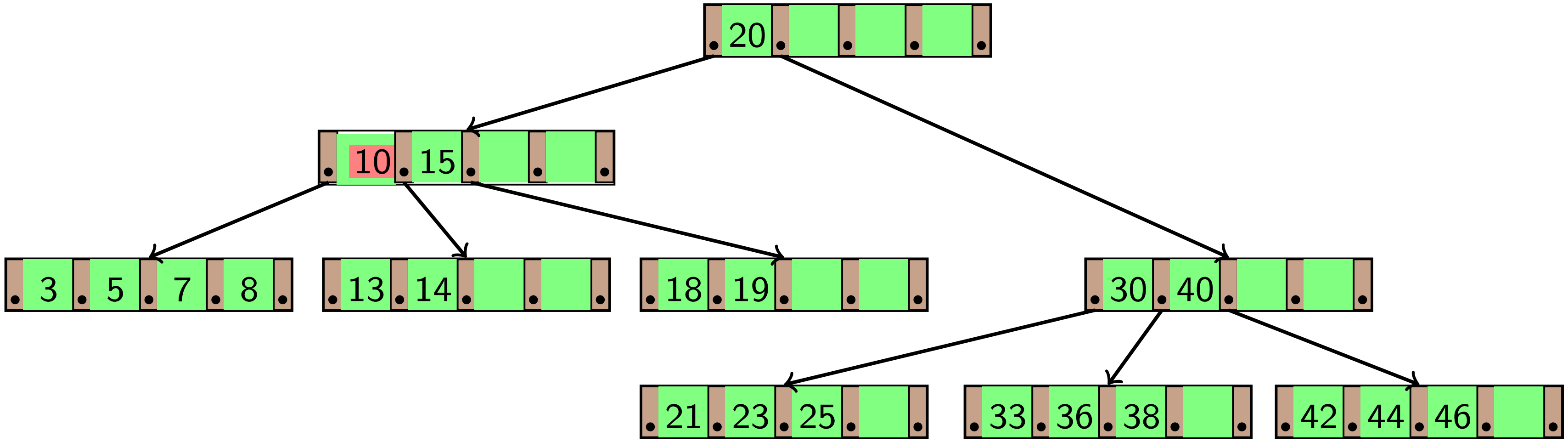
- Here is the result of the deletion.



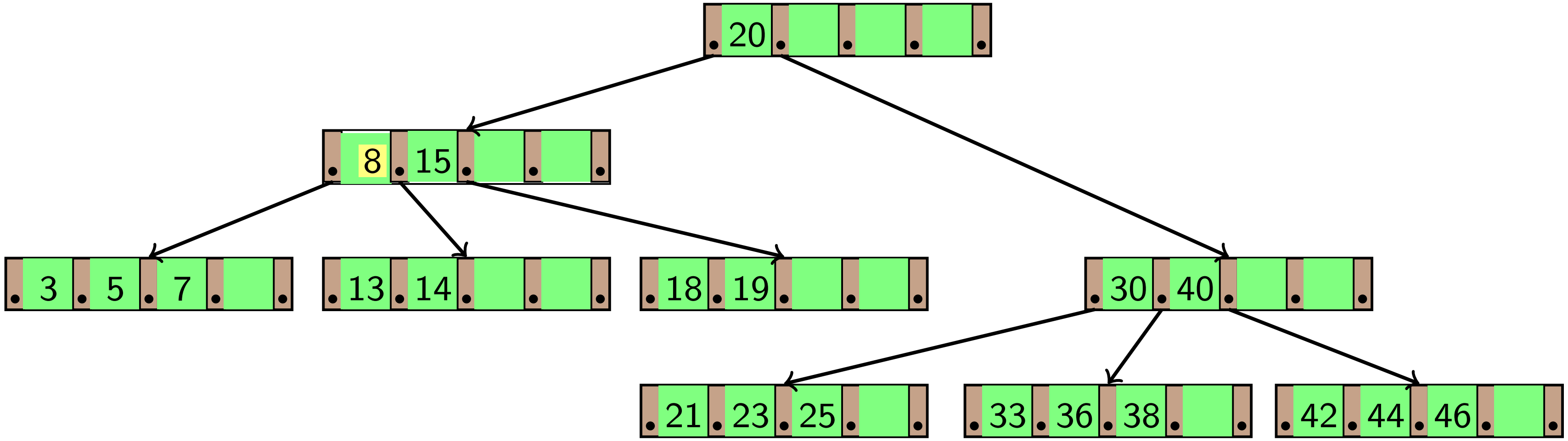
- This is the only way that a B-tree may shrink in depth.

Deletions of Non-Leaf Fields on B-trees

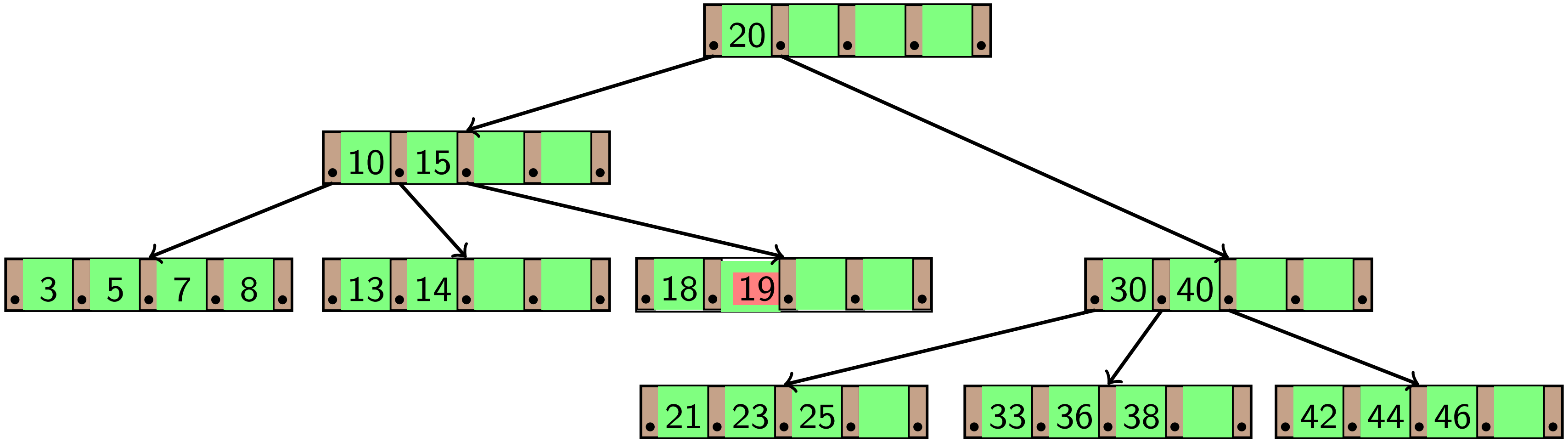
- It is sometimes possible to realize deletions within non-leaf vertices via redistribution.



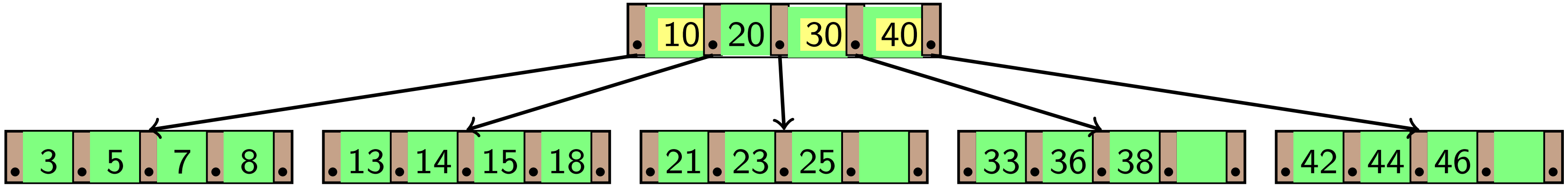
- Deletion of 10 may be achieved as follows:



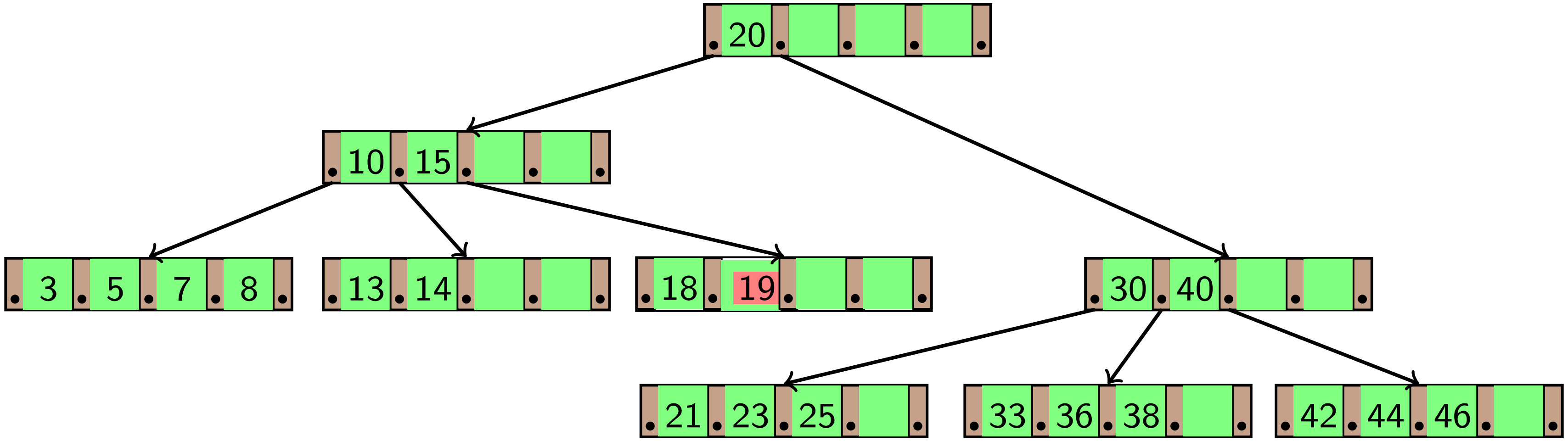
Deletions on B-trees with Alternative Solutions



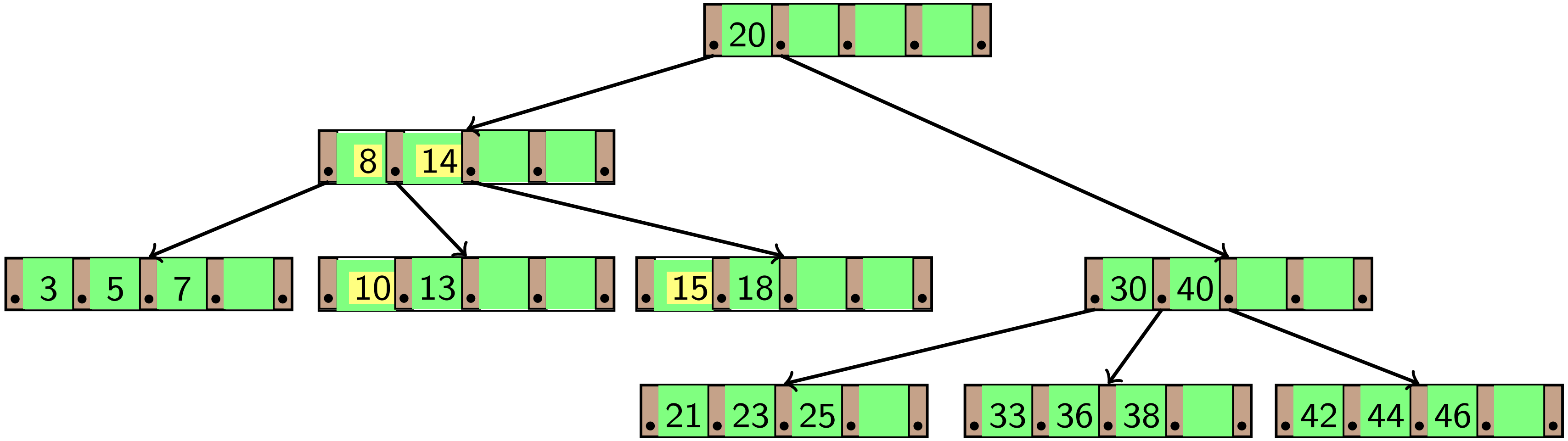
- Deletion of 19 appears to require adjustment at the second level, and then combination with the root.



Deletions on B-trees with Alternative Solutions — 2



- However, it is possible in this case to do a long-range, multiple readjustment.



Heuristics for B-Trees

If possible, avoid operations which involve cascaded splitting or combining of vertices: Such operations are generally very expensive.

- Choose them (if avoidable) only if it is imminent that they will be needed soon anyway.
- For example, if the operations are dominated by insertions, then allowing cascading splitting is reasonable.
- However, if future operations are expected to be dominated by deletions, then splitting should be avoided if possible.

Redistribute evenly: When redistributing elements to accommodate an insertion or a deletion, redistribute so that the number of elements in each sibling is about the same.

- This happens automatically in the simple examples here in which the order of the vertices is only four.
- However, it is far from automatic when the order is much larger.

Depth of a B-Tree

- It is very useful to be able to estimate the depth of a B-tree, given configuration parameters and the number of records.
- Such an estimate will help provide key information on expected access time.

Example setting:

Page size:	2 KBytes	
Record size:	128 Bytes	
Pointer size:	4 Bytes	(4 GBytes address space)
Total records	10^6	

- Maximum order n of the B-tree:

$$(n \times \text{PtrSize}) + ((n-1) \times \text{RecSize}) \leq \text{PageSize}$$

$$n = \left\lfloor \frac{\text{PageSize} + \text{RecSize}}{\text{PtrSize} + \text{RecSize}} \right\rfloor = \left\lfloor \frac{2048 + 128}{4 + 128} \right\rfloor = 16$$

Maximum-Depth B-Trees – Example Computation

Minimum density: A B-tree will have maximum depth when it has minimum *density* — as few records per vertex as possible.

- All vertices except the root will contain $\lfloor (n-1)/2 \rfloor = 7$ records.
- The root will contain one record.
- First, to see how to approach the problem, compute the necessary sizes by brute force.

Level	Vertices at the level	Records at the level	Total records
root	1	1	1
1	2	$2 \times 7 = 14$	15
2	$2 \times 8 = 16$	$16 \times 7 = 112$	127
3	$16 \times 8 = 128$	$128 \times 7 = 896$	1023
4	$128 \times 8 = 1024$	$1024 \times 7 = 7168$	8191
5	$1024 \times 8 = 8192$	$8192 \times 7 = 57344$	65535
6	$8192 \times 8 = 65536$	$65536 \times 7 = 458752$	524287
7	$65536 \times 8 = 524288$	$524288 \times 7 = 3670016$	4194303

- The maximum depth is thus 6, since a depth of 7 would require at least 4194303 records.

Parameters of B-Trees

- The brute force approach becomes tedious, particularly when the depth becomes substantial.
- It is instructive to develop more general, closed formulas.
- The general parameters are as follows:

Parameter	Meaning
d	depth of the B-tree
m	number of records in the root vertex
r	number of records in all other vertices

- It is very rare that all non-root vertices will contain exactly the same number of records.
- These parameters are therefore used in approximation.
- A B-tree which satisfies these conditions will be called (m, r, d)-*uniform*.

Maximum-Depth B-Trees — Formulas

- Here is a computation of the number of vertices at each level.

Level	Vertices	Records
root	1	m
1	$m+1$	$(m+1)\cdot r$
2	$(m+1)\cdot(r+1)$	$(m+1)\cdot(r+1)\cdot r$
3	$(m+1)\cdot(r+1)^2$	$(m+1)\cdot(r+1)^2\cdot r$
4	$(m+1)\cdot(r+1)^3$	$(m+1)\cdot(r+1)^3\cdot r$
$\dots$	$\dots$	$\dots$
d	$(m+1)\cdot(r+1)^{d-1}$	$(m+1)\cdot(r+1)^{d-1}\cdot r$

- Thus, the total number of records $R(m,r,d)$ in an (m,r,d)-uniform B-tree is given by

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

Maximum-Depth B-Trees — Formulas 2

- Continuing with

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

- The general law

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

which may be derived from

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

leads to

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

which simplifies to

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

Maximum-Depth B-Trees — Formulas 3

- Continuing with

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

- To find the value for d with minimum density, with N the total number of records to be stored, begin as follows:

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

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

- To solve for d, take the log for base $r+1$ of each side:

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

Maximum-Depth B-Trees — Using the Formulas on the Example

- Continuing with:

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

- In the example, $r = 7$, $N = 1000000$, and $m = 1$, so

$$d \leq = \frac{\log_e\left(\frac{1000000+1}{1+1}\right)}{\log_e(7+1)} = \frac{\log_e(500000.5)}{\log_e(8)} = 6.31$$

- Since the depth of a B-tree must be an integer, it follows that it cannot be greater than 6, in agreement with the brute-force approach.

Minimum-Depth B-Trees – Example Computation

Maximum density: A B-tree will have minimum depth when it has maximum density — as many records per vertex as possible.

- All vertices, including the root, will contain $n - 1 = 15$ records.
- First, to see how to approach the problem, compute the necessary sizes by brute force.

Level	Vertices at the level	Records at the level	Total records
root	1	15	15
1	16	$16 \times 15 = 240$	255
2	$16^2 = 256$	$256 \times 15 = 3840$	4095
3	$16^3 = 4096$	$4096 \times 15 = 61440$	65535
4	$16^4 = 65536$	$65536 \times 15 = 983040$	1048575

- The minimum depth is thus 4, since a depth of 3 would hold at most 65535 records, while a depth of 4 can hold more than 10^6.

Minimum-Depth B-Trees — Formulas

- Recall:

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

- To solve for the value for r with maximum density, with N the total number of records to be stored, this time:

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

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

- Since $m = r$,

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

so, taking the log base $r+1$ of each side:

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

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

Minimum-Depth B-Trees — Using the Formulas on the Example

- Continuing with:

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

- In the example, $r = 15$, $N = 1000000$, so

$$d \geq = \frac{\log_e(1000000+1)}{\log_e(15+1)} - 1 = \frac{\log_e(1000001)}{\log_e(16)} - 1 = 3.9828$$

- Since the depth of a B-tree must be an integer, it follows that it must be at least 4, in agreement with the brute-force approach.

- The fact that d is very close to 4 suggests that by adding just a few more vertices to N, a tree of depth five would be required. The "brute-force" chart confirms this; the largest $(15, 15, 4)$-uniform B-tree has 1048575 vertices, only 48575 more than 100000.

Computing the Total Number of Records — Formula

- The basic formula below is useful in other ways.

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

- For example, if the total number of records, as well as depth d and root record count m of a (m, r, d)-uniform B-tree is known, then the record density r can be computed as well:

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

- To solve for r, take the d^{th} root of both sides, and subtract 1:

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

Computing the Total Number of Records — Examples

- Consider again the example of maximum depth with 10^6 records.
- The known parameters are $m = 1$ (given) and $d = 6$ (computed previously).
- To find the value r which identifies the number of records in each vertex:

$$r = \sqrt[d]{\frac{R(m, r, d) + 1}{m + 1}} - 1 = \sqrt[6]{\frac{10^6 + 1}{1 + 1}} - 1 = \sqrt[6]{\frac{1000001}{2}} - 1 = 7.90$$

- This means that a $(1, r, 6)$-uniform B-tree would have 7.90 records in each of its non-root vertices.
- Of course, it is impossible to have a tree with 7.90 records per vertex.
- This result is thus just an estimate.
- A real B-tree, as balanced as possible, would have between 7 and 8 records per vertex.

Computing the Total Number of Records — Examples 2

- Continue with this example, and suppose that two records are now in the root vertex.
- To find the value r which identifies the number of records in each vertex:

$$r = \sqrt[d]{\frac{R(m,r,d)+1}{m+1}} - 1 = \sqrt[6]{\frac{10^6+1}{2+1}} - 1 = \sqrt[6]{\frac{1000001}{3}} - 1 = 7.32$$

- By creating slightly more fan-out at the root vertex, the lower vertices are much less densely populated.
- In fact, the density is just barely adequate, since the minimum is 7.
- Now suppose that the root contains three records.

$$r = \sqrt[d]{\frac{R(m,r,d)+1}{m+1}} - 1 = \sqrt[6]{\frac{10^6+1}{3+1}} - 1 = \sqrt[6]{\frac{1000001}{4}} - 1 = 6.93$$

- This value does not define a valid situation; the minimum depth is 7.

Observation: Not any mix of parameters will result in a valid approximation to a real B-tree.

Computing the Total Number of Records — Examples 3

- Consider again the example of minimum depth with 10^6 records.
- The known parameters are $m = 15$ (given) and $d = 4$ (computed previously).
- To find the value r which identifies the number of records in each vertex:

$$r = \sqrt[d]{\frac{R(m,r,d)+1}{m+1}} - 1 = \sqrt[4]{\frac{10^6+1}{15+1}} - 1 = \sqrt[4]{\frac{1000001}{16}} - 1 = 14.81$$

- The average record density of the vertex is extremely high, as is expected, since a $(15, r, 4)$-uniform tree can have a maximum of 1048481 records.
- If the fan-out at the root is reduced by just one, to $m = 14$:

$$r = \sqrt[d]{\frac{R(m,r,d)+1}{m+1}} - 1 = \sqrt[4]{\frac{10^6+1}{14+1}} - 1 = \sqrt[4]{\frac{1000001}{15}} - 1 = 15.06$$

- This value does not define a valid situation; max records/vertex $= 15$.
- Indeed, a uniform (14,15,4) B-tree has $(m+1)\cdot(r+1)^d - 1 = 15 \cdot 16^4 - 1 = 983041$ as the maximum number of records, which is only slightly less than 10^6.

Average Path Length in a B-Tree

Question: What is the average path length from the root to a vertex in a B-tree.

- This question is readily examined in the context of (m, r, d)-uniform B-trees.
- From previous computations:

$$\text{Number of records at level } d = (m+1) \cdot (r+1)^{d-1} \cdot r$$

$$\text{Total number of records} = (m+1) \cdot (r+1)^{d} - 1$$

- Thus, the percentage of records which are situated in leaf vertices is approximately:

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

Average Path Length in a B-Tree — 2

- Continuing with:

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

- If r is reasonably large, most of the records will reside in the leaf vertices.

r	$\frac{r}{r+1}$
1	0.500
4	0.800
7	0.875
15	0.938
32	0.970
100	0.990

- Thus, even for the simple examples considered here, it can be expected that close to 90% of the records will reside in the leaf vertices.

Implications of Most Records Residing in Leaves

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

- With four or five disk accesses per fetch, this is unacceptable.

Solutions: There are several ways to reduce the number of disk accesses.

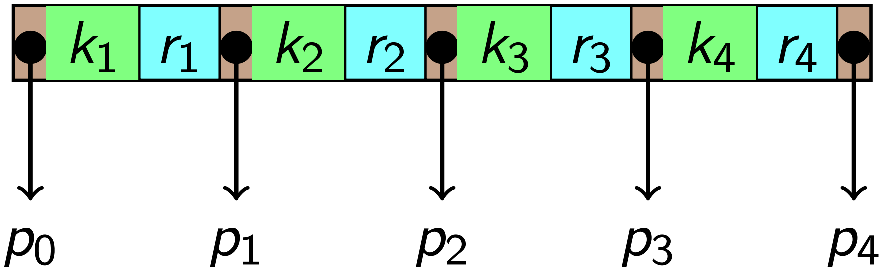
Keep the top few levels in main memory: By keeping (copies of) the first k levels of the B-tree in main memory, the number of disk accesses is reduced by k.

Build an index into the B-tree: This is possible, but there are better solutions (such as the B^+-tree).

Store pointers rather than records in the B-tree: This solution will be discussed in more detail.

B-Trees of Keys and Pointers

- Instead of storing an entire record in the B-tree, an alternative is to store only the key value and a pointer to the full record.
- This is the approach described in the textbook.
- A (non-leaf) vertex appears as follows:



- Each r_i is a pointer to the record whose key is k_i.
- Typically, $k_1 + r_i$ is much smaller than an entire record.
- Thus, the number of items per vertex will be much greater, and so the tree will be much less deep.
- A drawback to this approach is that the storage of neighboring records can become very fragmented.
- For example, distinct disk accesses may be necessary to retrieve r_1 and r_2.
- The B$^+$-tree typically offers a better solution in this regard.

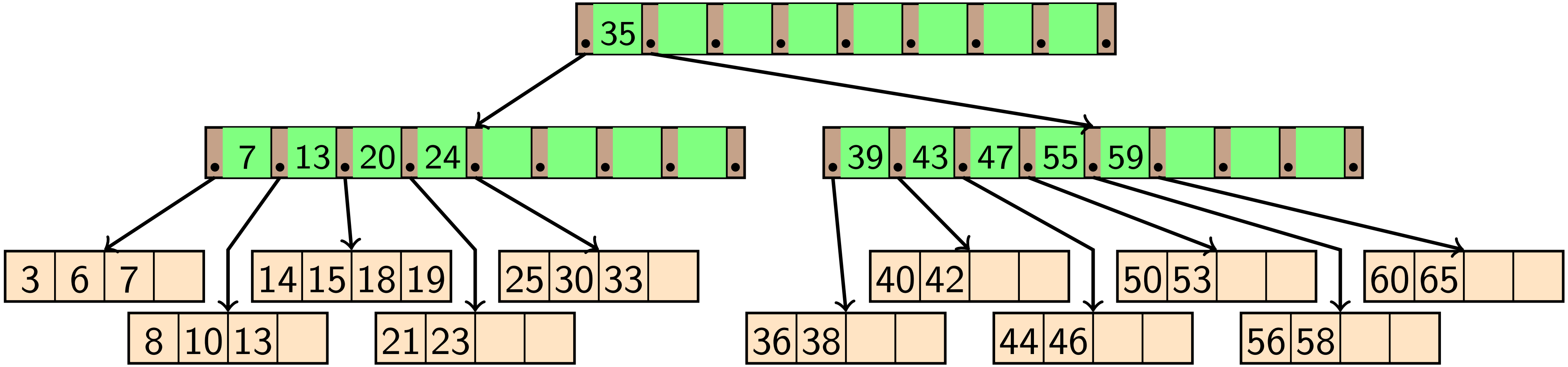
The B^+-Tree

- The B^+-tree differs from the B-tree in the following fundamental way.
  - All records are stored in the leaves.
  - The internal vertices contain the index only.

Advantages:

- Since index fields are typically much smaller than record fields, many index values may be stored in a single internal vertex.
- This implies that the fanout in the non-leaf vertices will be very high.
- This implies, in turn, that the index will be relatively small and not very deep.
- The leaf vertices (left to right) form an ordered sequential representation, thus facilitating sequential processing.

Visualization of a B$^+$-Tree



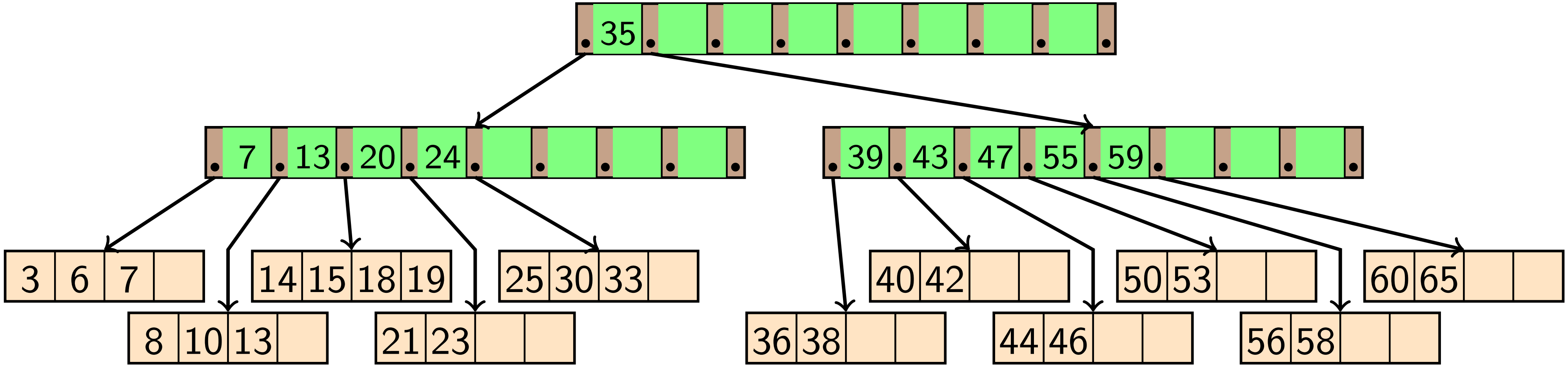
- Shown above is a B$^+$-tree of *order* (9,4).
  - The order of a non-leaf vertex is defined exactly as in a B-tree.
  - The order of a leaf vertex is defined to be the maximum number of records which can be stored in it.

- Note that leaf vertices do not have any pointer fields (none are needed).

- The values which are stored in the non-leaf vertices are just *possible* keys, and do not need to be key values of records stored in the leaves.

- A key value does not occur more than once in the index.

Convention for Index Paths in a B^+-Tree



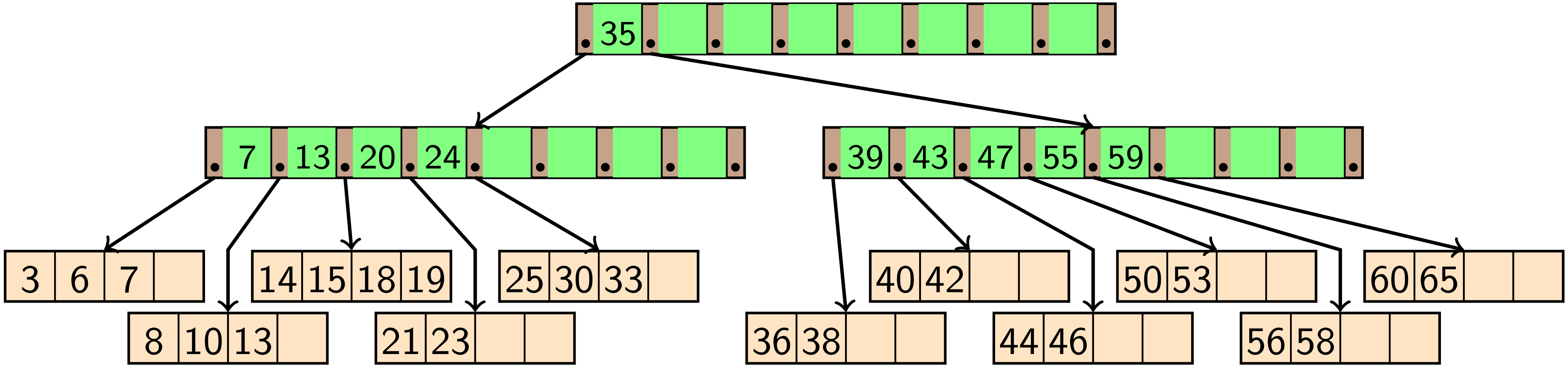
Convention for pointers of index vertices:

Pointer to the left of key k: All further indices and records with keys which are $\leq k$.

Pointer to the right of key k: All further indices and records with keys which are $> k$.

- In other words, for a search value which is equal to the index value, go left, not right.

Fullness Conditions on the Vertices of a B$^+$-Tree



- As in the case of a B-tree, all vertices except the root must be at least "half full".

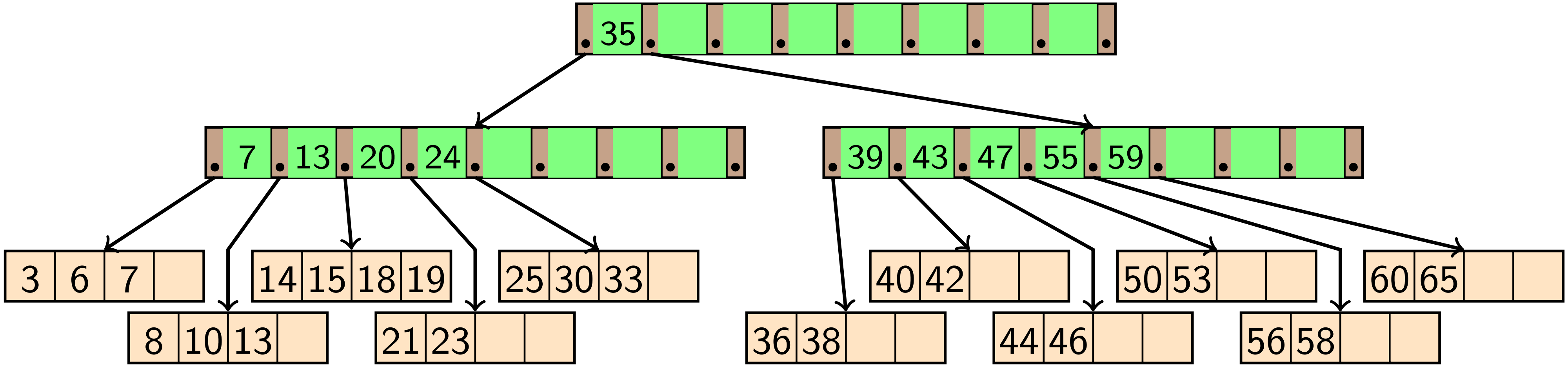
Internal (index) vertices: The condition for internal (index vertices) is exactly the same as for B-trees:

  - Each vertex except the root must contain at least $\lfloor (n_{\text{int}} - 1)/2 \rfloor$ vertices, where n_{int} is the order (number of pointers) in such a vertex.

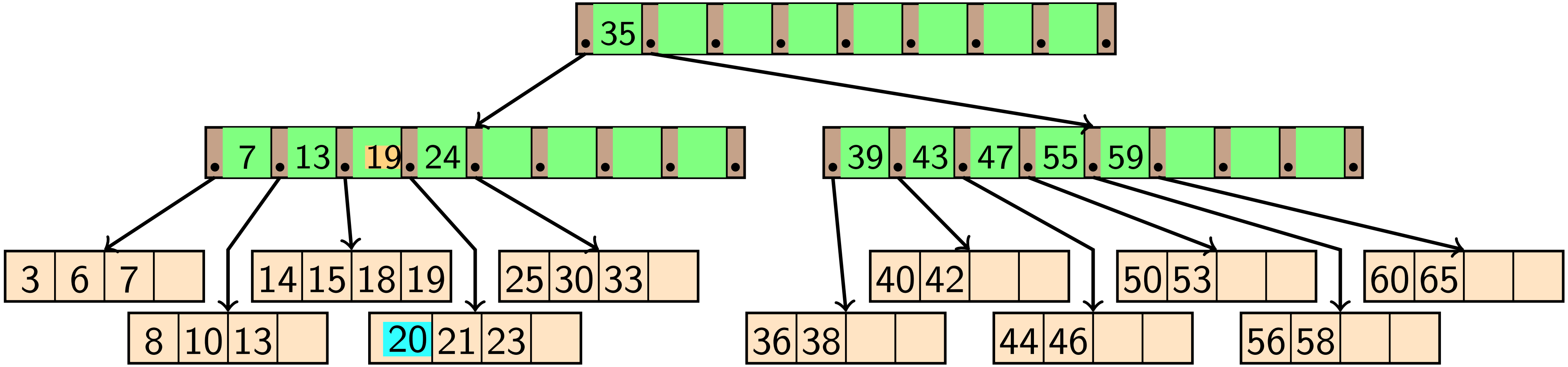
Leaf vertices: The condition for leaf vertices stipulates that if the maximum number of records is odd, then half full is defined by "round up".

  - Each leaf must contain at least $\lceil (n_{\text{ext}})/2 \rceil$ vertices, where n_{ext} is the order (number of possible records) in such a vertex.

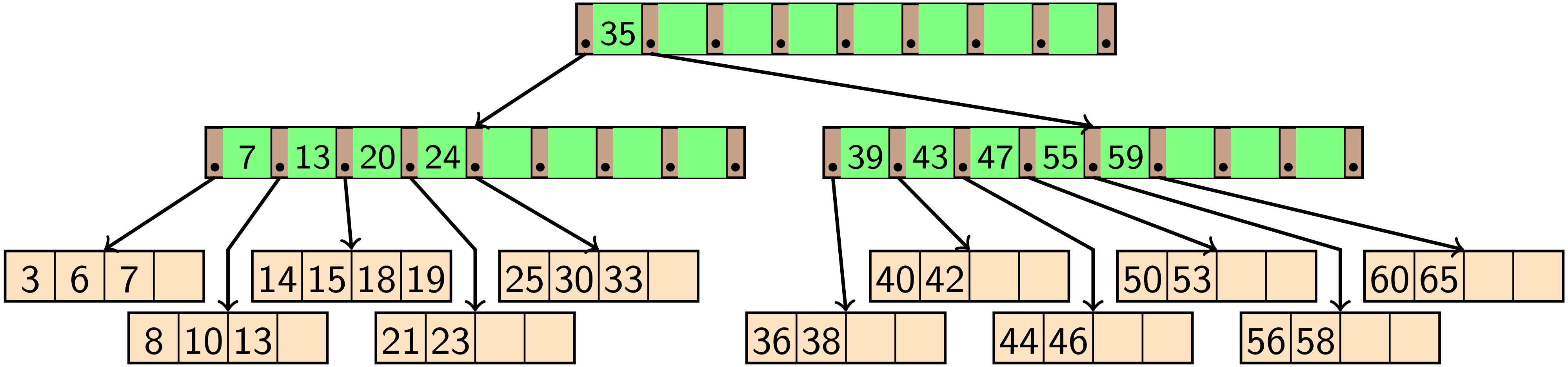
Insertion into a B^+-Tree



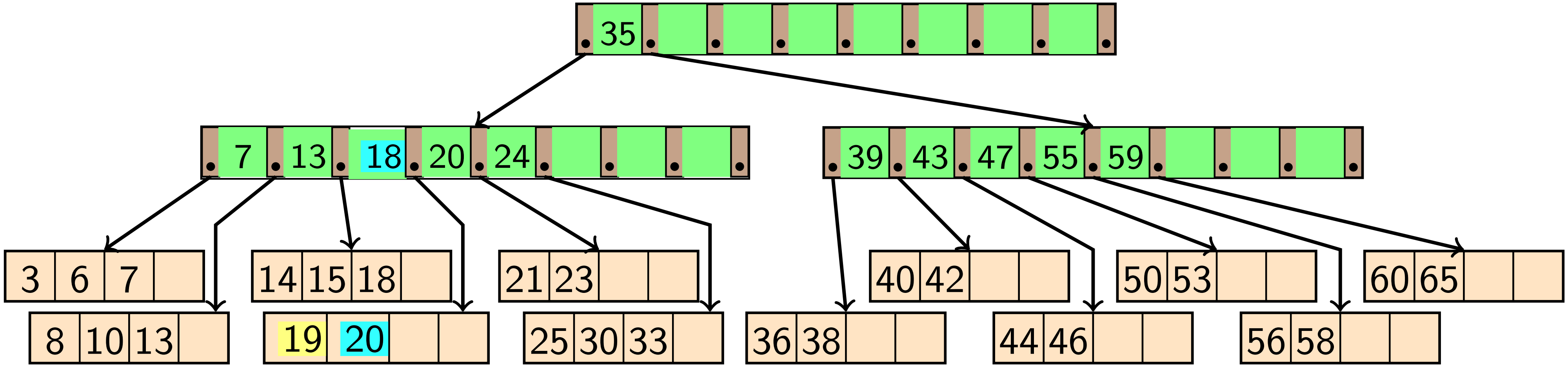
- Consider insertion of a record with key 20 into the above tree.
- The index value 20 must be changed to 19 (changes shown in orange).
- Alternatively, a straightforward rotation may be used.



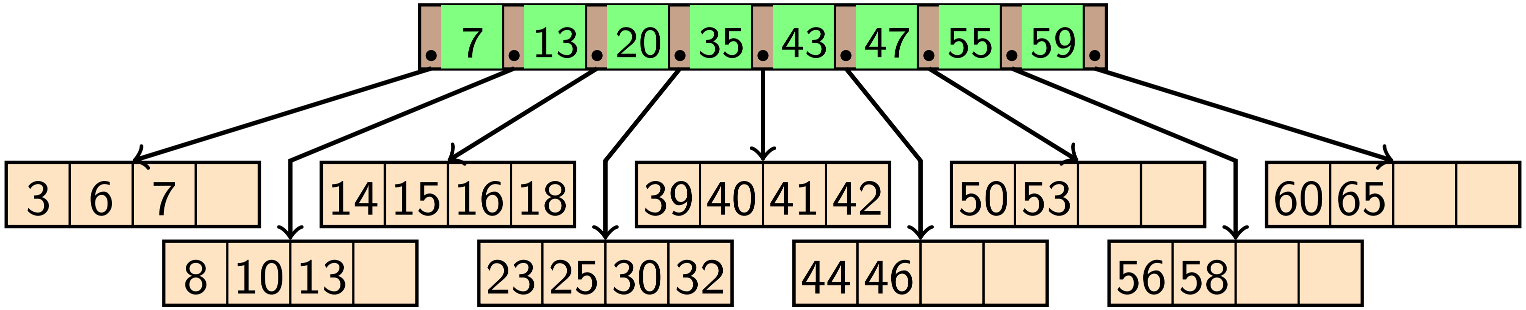
Insertion into a B$^+$-Tree — 2



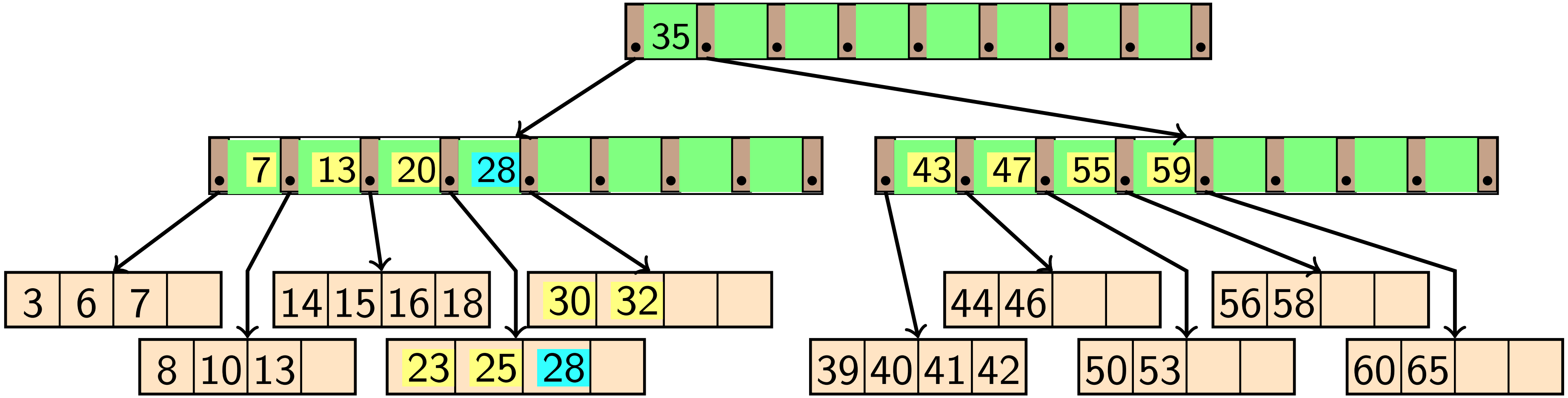
- It is possible to solve this same insertion of 20 via a split of the leaf vertex together with the insertion of a new index value.



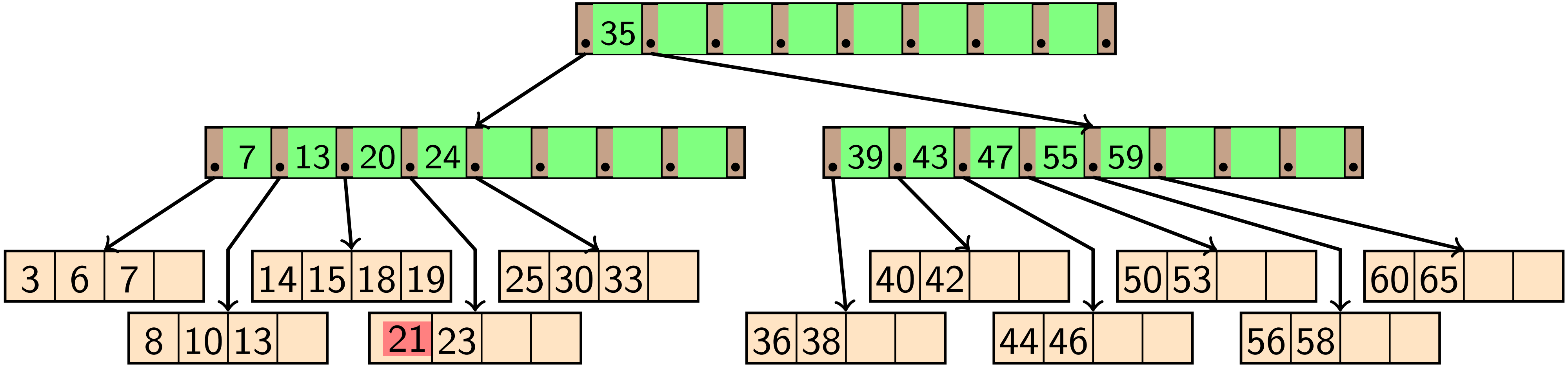
Insertion into a B^+-Tree — 3



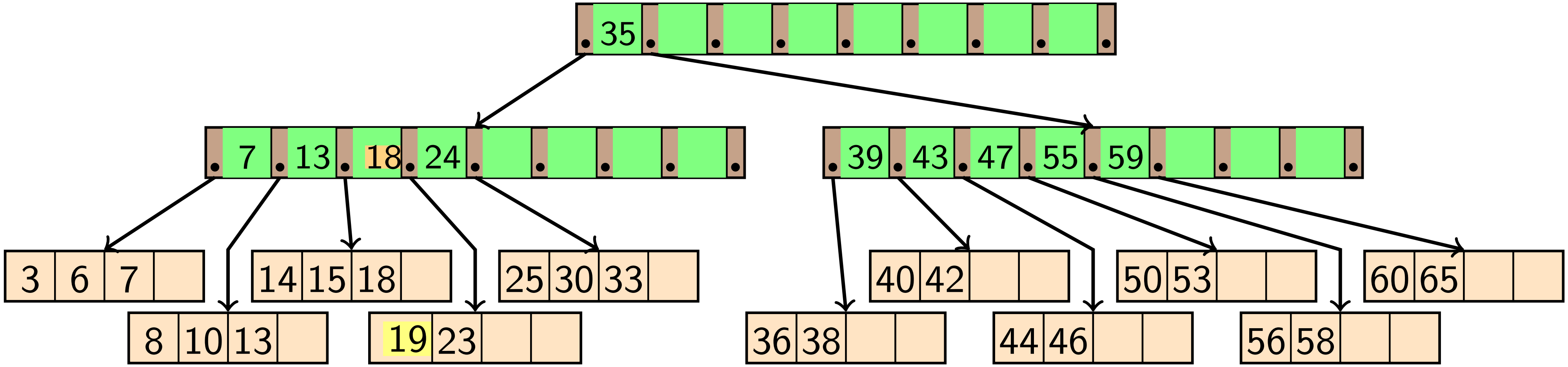
- Insertion of a record with key 28 into the above tree requires a split of the vertex at the second level as well as the root.
- The inserted internal key (not record) 28 could be either of 28 or 29.
- This is the only way which the depth of a B^+-tree may increase.



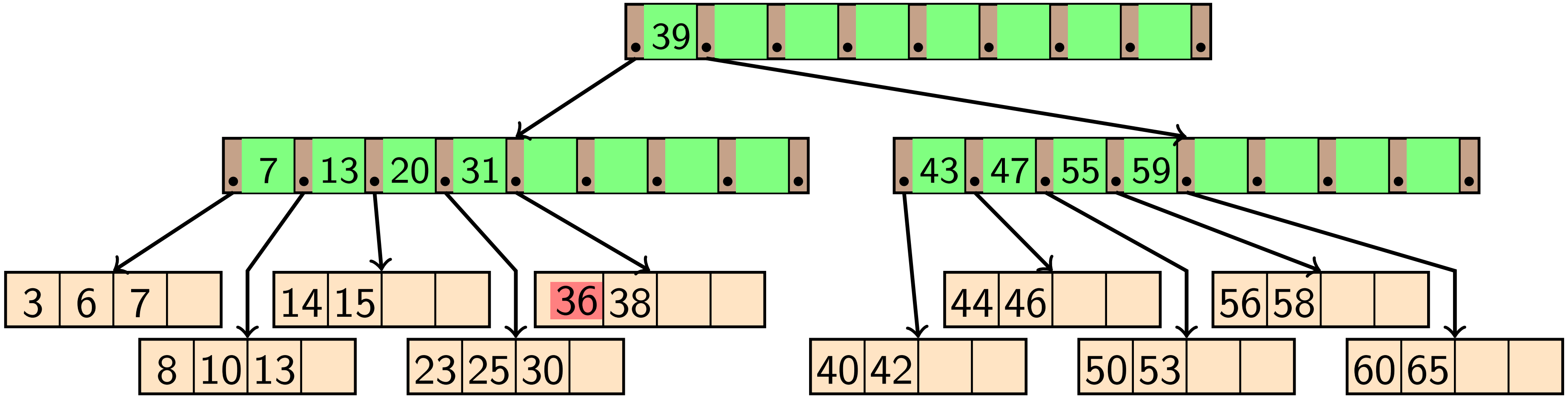
Deletion from a B$^+$-Tree



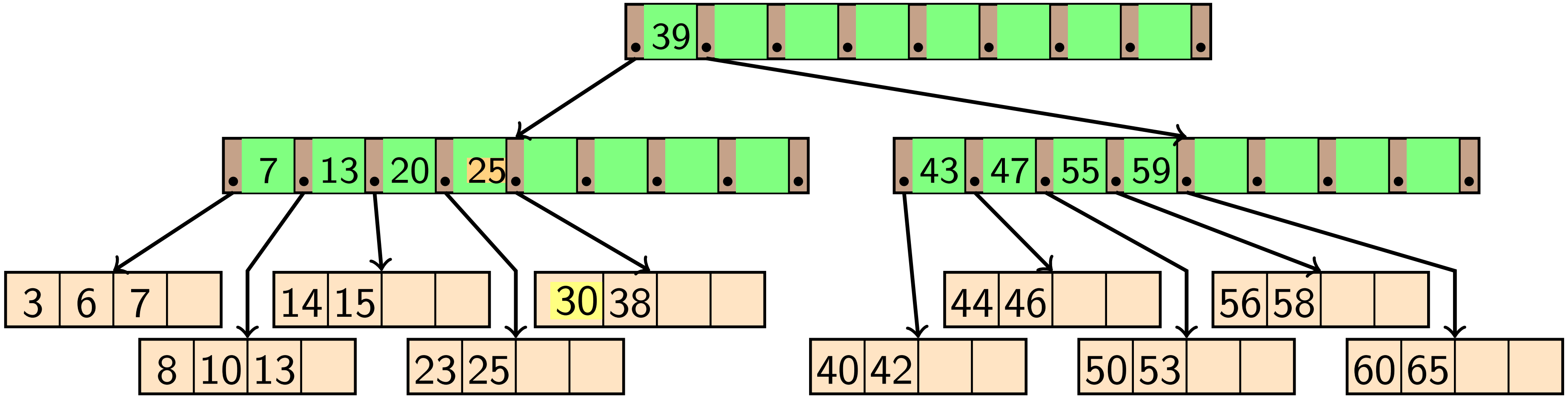
- Deletion of 21 from the above tree is realized as shown below.
- A simple rotation and change of key value is required.



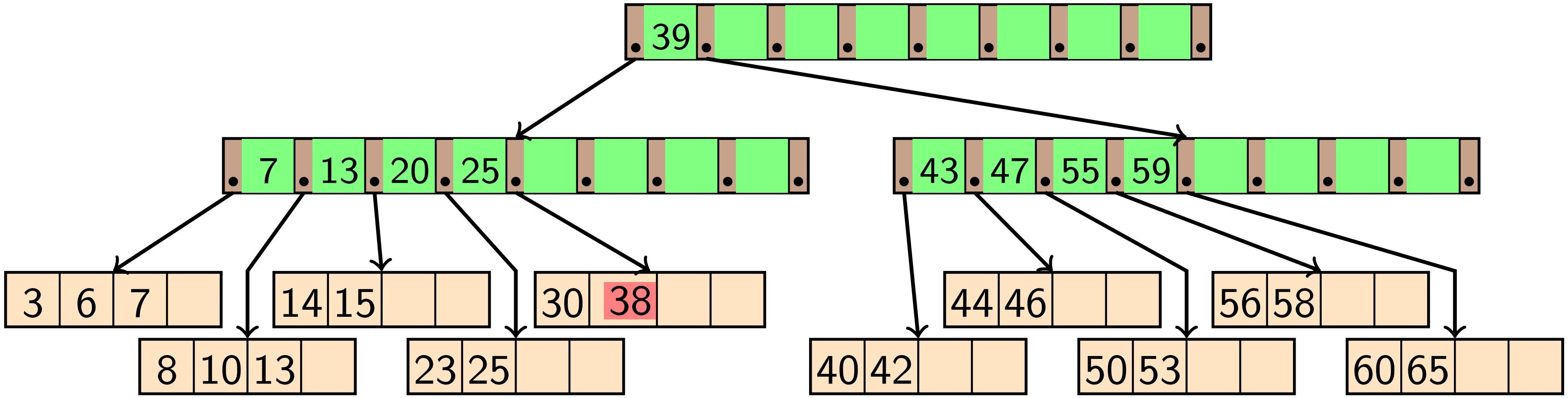
Deletion from a B^{+}-Tree — 2



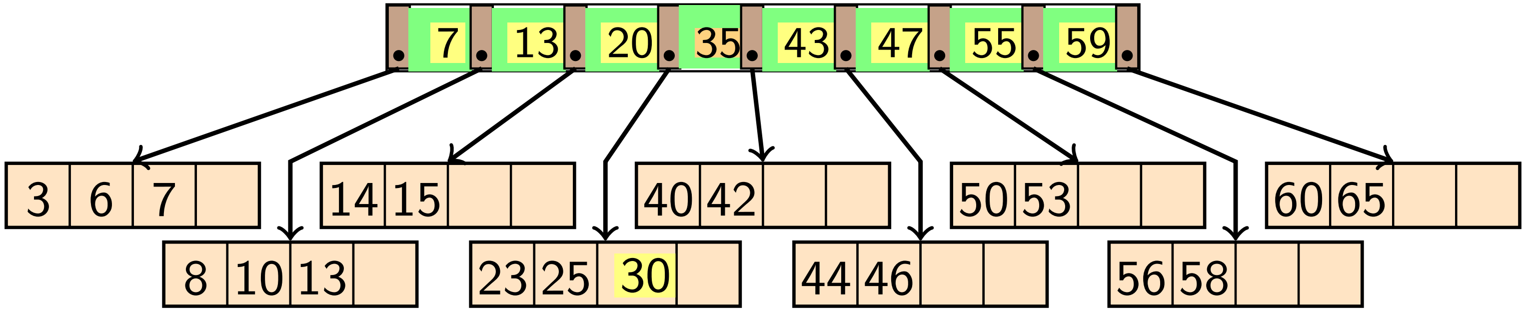
- Deletion of 36 from the above tree is realized as shown below.
- A simple rotation and change of key value is required.



Deletion from a B^+-Tree — 3

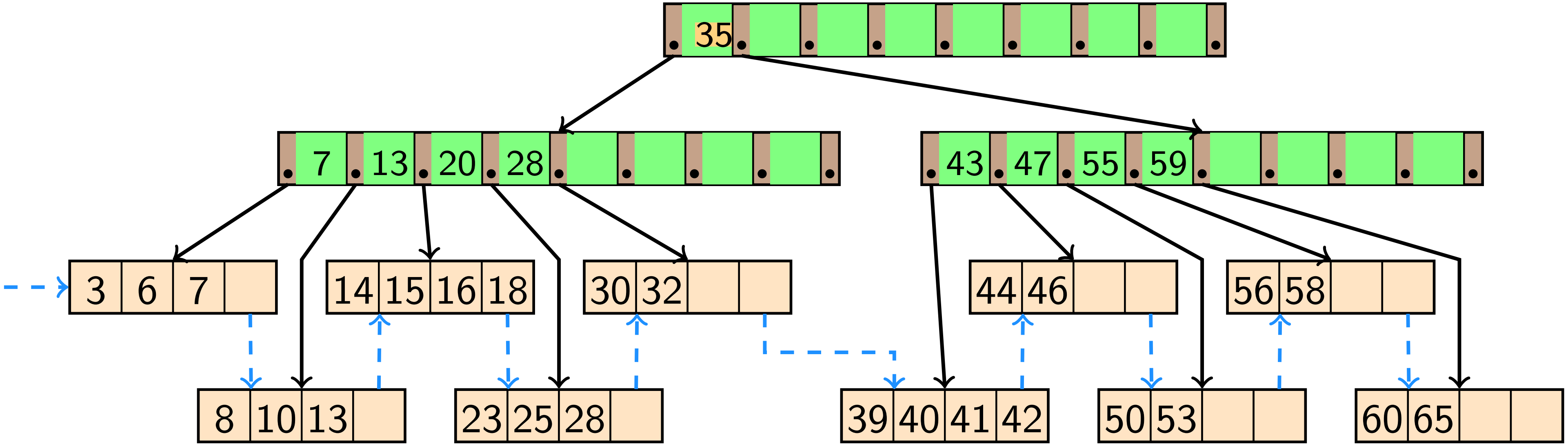


- Continuing with the previous result, deletion of 38 requires a combination of both vertices and keys, together with shrinking of the depth.
- The new value for the key obtained by combining 20 and 39 (35) could be any value 30-39.
- This is the only way which the depth of a B^+-tree may become smaller.



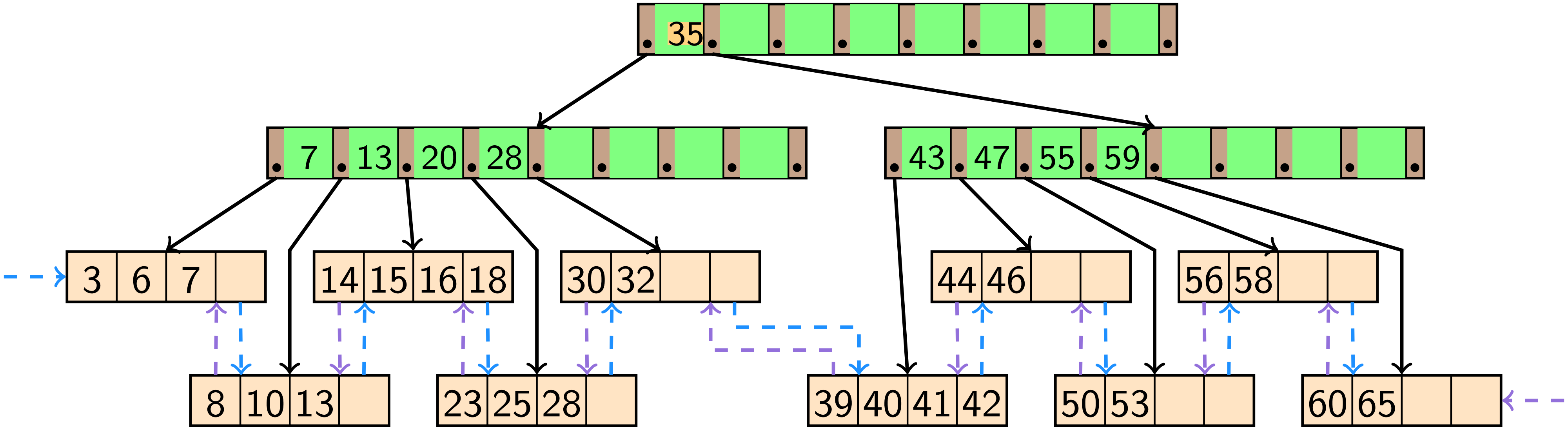
Sequential Access in B$^+$-Trees

- Sequential access may be obtained by linking the leaves together.



Sequential Access in B$^+$-Trees

- Sequential access may be obtained by linking the leaves together.
- Usually, links are provided in both directions, so that reverse as well as forward sequential access is possible.
- This also provides efficient access to neighboring data vertices.
- For best performance, adjacent leaf vertices should be sequential neighbors on the disk as well, insofar as possible.



Depth of B^+-Tree

Example setting:

Page size:	2 KBytes
Record size:	128 Bytes
Pointer size:	4 Bytes
Bytes per internal key	16
Total records	10^6
Total bytes for sequential pointers in leaves	8

- Maximum order n for the internal vertices:

$$(n \times \text{PtrSize}) + ((n-1) \times \text{KeySize}) \leq \text{PageSize}$$

$$n = \left\lfloor \frac{\text{PageSize} + \text{KeySize}}{\text{PtrSize} + \text{KeySize}} \right\rfloor = \left\lfloor \frac{2048 + 16}{4 + 16} \right\rfloor = 103$$

- Maximum number of records r_{max} per leaf vertex:

$$(r_{\text{max}} \times \text{RecSize}) + \text{SeqPtrsSize} \leq \text{PageSize}$$

$$r_{\text{max}} = \left\lfloor \frac{\text{PageSize} - \text{SeqPtrsSize}}{\text{RecSize}} \right\rfloor = \left\lfloor \frac{2048 - 8}{128} \right\rfloor = 15$$

Maximum-Depth B$^+$-Trees – Example Computation

Minimum density: A B$^+$-tree will have maximum depth when it has minimum *density* — as few keys per internal vertex and as few records per leaf as possible.

- Internal vertices other than the root will contain $\lfloor (n-1)/2 \rfloor = \lfloor 102/2 \rfloor = 51$ keys.
- The root will contain one key.
- Record vertices will contain $\lceil r_{\text{max}}/2 \rceil = \lceil 15/2 \rceil = 8$ records.
- Brute force:

Level	Vertices at level	Keys at the level	Min Leaf Records
root	1	1	$2 \cdot 8 = 16$
1	2	$2 \times 51 = 102$	$2 \times 52 \times 8 = 832$
2	$2 \times 52 = 104$	$104 \times 51 = 5304$	$104 \times 52 \times 8 = 58240$
3	$104 \times 52 = 5408$	$5408 \times 51 = 275808$	$5408 \times 52 \times 8 = 2249728$

- The maximum depth of the index is thus 2, since a depth of 3 would require at least 2249728 records.
- The tree itself, including leaves, has a maximum depth of 3.

Parameters of B$^+$-Trees

- The brute-force approach becomes tedious, particularly when the depth becomes substantial.
- It is instructive to develop more general formulas.
- The general parameters are as follows:

Parameter	Meaning
m	number of keys in the root vertex
q	number of keys in other internal vertices
r	number of records in a leaf vertex
d	depth, from root to leaf

- It is very rare that all non-root vertices will contain exactly the same number of records.
- These parameters are therefore used in approximation.
- In the above example, $m = 1$, $q = 51$, $r = 8$, and d is to be computed.
- A B$^+$-tree which satisfies these conditions will be called *(m, q, r, d)-uniform*.

Maximum-Depth B$^+$-Trees — Formulas

- Here is a computation of the number of vertices at each level.

Level	Index Vertices	Keys	Total Rec Next Level
root	1	m	$(m+1)\cdot r$
1	$m+1$	$(m+1)\cdot q$	$(m+1)\cdot(q+1)\cdot r$
2	$(m+1)\cdot(q+1)$	$(m+1)\cdot(q+1)\cdot q$	$(m+1)\cdot(q+1)^2\cdot r$
3	$(m+1)\cdot(q+1)^2$	$(m+1)\cdot(q+1)^2\cdot q$	$(m+1)\cdot(q+1)^3\cdot r$
$\cdots$	$\cdots$	$\cdots$	$\cdots$
$d-1$	$(m+1)\cdot(q+1)^{d-2}$	$(m+1)\cdot(q+1)^{d-2}\cdot q$	$(m+1)\cdot(q+1)^{d-1}\cdot r$
d	$(m+1)\cdot(q+1)^{d-1}$	$(m+1)\cdot(q+1)^{d-1}\cdot q$	$(m+1)\cdot(q+1)^{d}\cdot r$

- The total number of records $R(m,q,r,d)$ in an (m,q,r,d)-uniform B$^+$-tree is given by choosing the value for level $d-1$ (the last level of indices) in the table:

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

- Solving for d:

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

Maximum-Depth B^+-Trees — the Formulas on the Example

- Continuing with:

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

- In the example, $r = 8$, $N = 1000000$, $m = 1$ and $q = 51$, so

$$d = \frac{\log_e\left(\frac{1000000+1}{(1+1)\cdot 8}\right)}{\log_e(51+1)} + 1 = \frac{\log_e(62500)}{\log_e(52)} + 1 = 3.79$$

- Since the depth of a B^+-tree must be an integer, it follows that it cannot be greater than $\lfloor 3.79 \rfloor = 3$, in agreement with the brute-force approach.

Minimum-Depth B^+-Trees – Example Computation

Maximum density: A B^+-tree will have minimum depth when it has maximum *density* — as many keys per internal vertex and as many records per leaf as possible.

- Internal vertices, including the root, will contain $n - 1 = 102$ records.
- Record vertices will contain $r_{\text{max}} = 15$ records.
- Brute force:

Level	Vertices at level	Keys at the level	Leaf Records
root	1	102	$103 \cdot 15 = 1545$
1	103	$103 \times 102 = 10506$	$103^2 \times 15 = 159135$
2	103^2	$103^2 \times 102 = 1082116$	$103^3 \times 15 = 16390905$

- The minimum depth of the index is thus 2, since a depth of 1 would support at most 159135 records.
- The tree itself, including leaves, thus has a maximum depth of 3.
- The minimum and maximum depths are the same for this example!

Minimum-Depth B^+-Trees — Applying the Formula

- Recall:

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

- In the example, $r = 15$, $N = 1000000$, $m = q = 102$, so

$$d = \frac{\log_e\left(\frac{1000000+1}{(102+1)\cdot 15}\right)}{\log_e(102+1)} + 1 = \frac{\log_e(647.24)}{\log_e(103)} + 1 = 2.39$$

- Since the depth of a B^+-tree must be an integer, it follows that it cannot be less than $\lceil 2.39 \rceil = 3$, in agreement with the brute-force approach.

Maximum-Depth B^+-Trees — Adjustment Example

- It is not always possible to find a maximum-depth B^+-tree with only one key in the root.
- Consider a $(1, ?, 8, 3)$-uniform B^+-tree with exactly 2249728 data records.

$$q = \sqrt[d-1]{\frac{R(m,r,d)+1}{(m+1)\cdot r}} - 1 = \sqrt[2]{\frac{2249728+1}{(1+1)\cdot 8}} - 1 = 373.98$$

- This value is larger than the maximum value $q_{\mathrm{max}} = 102$, so no such B^+-tree is possible.
- To find the minimum value for m which will work:

$$m_{\mathrm{min}} \geq \frac{R(m,r,d)}{(q_{\mathrm{max}}+1)^{d-1}\cdot r} - 1 = \frac{2249728+1}{(102+1)^2\cdot 8} - 1 = 26.04$$

- Thus, $m_{\mathrm{min}} = 27$ and so

$$q = \sqrt[d-1]{\frac{R(m,r,d)+1}{m_{\mathrm{min}}+1}\cdot r} - 1 = \sqrt[2]{\frac{2249728+1}{(27+1)\cdot 8}} - 1 = 100.21.$$

- Similar examples for minimum-depth B^+-trees, and even for B-trees, are handled analogously.

The Number of Index Vertices in a B$^+$-Tree

- Using the table on a previous slide, it is easy to see that the total number of index (interior) vertices in an (m, q, r, d)-uniform B$^+$-tree 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 $(1, 51, 8, 4)$-uniform B$^+$-tree, $\Rightarrow$ 2249728 data records $\Rightarrow$ 5515 index vertices.

- Consider a $(102, 102, 15, 3)$-uniform B$^+$-tree, $\Rightarrow$ 16390905 data records $\Rightarrow$ 10713 index vertices.

- This is a small example; even much larger ones have small indices, which may often be kept in main memory.

Bulk Loading of B$^+$-Trees

Problem: Given a large collection of records, build a B$^+$-tree index for it.

Observation: Insertion of records into an initially empty tree, one by one, will be very slow.

Bulk loading is the process of creating an entire index for a collection of records.

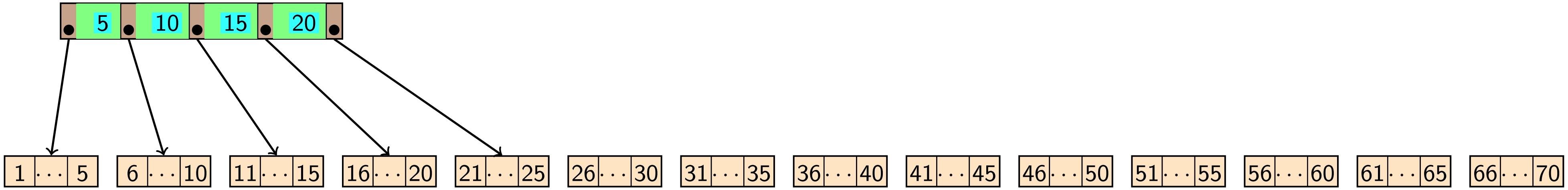
- The first step is to sort the records, and then place them into leaf vertices.
- Shown below is a small sorted collection of 70 records in 14 vertices.
- They need not be full, but they must all be half full.
- The idea is to build an index on top of this sequence of leaf vertices, from left to right.

1 · · · 5	6 · · · 10	11 · · · 15	16 · · · 20	21 · · · 25	26 · · · 30	31 · · · 35	36 · · · 40	41 · · · 45	46 · · · 50	51 · · · 55	56 · · · 60	61 · · · 65	66 · · · 70

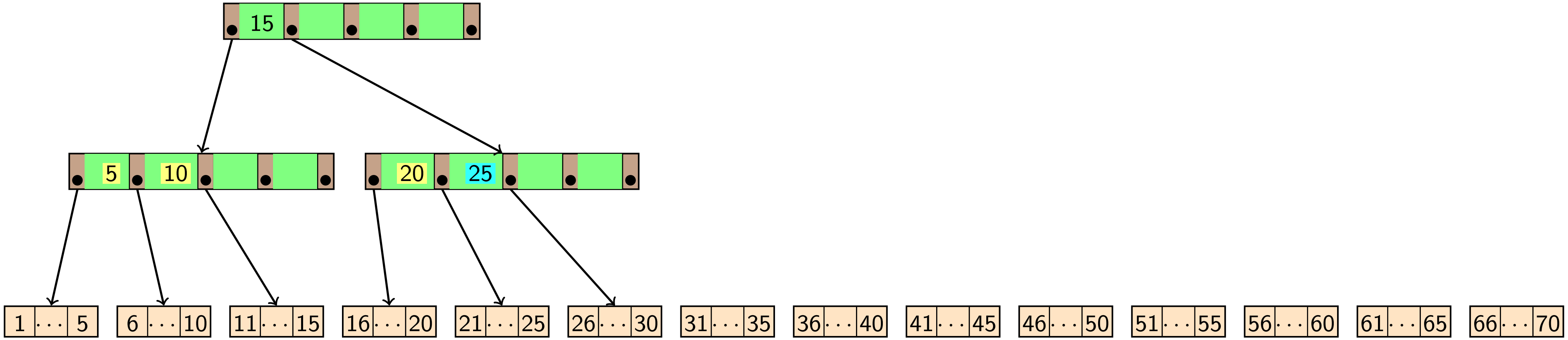
Bulk Loading of B$^+$-Trees — 2

- The first step is to create a top level index for as many leaf vertices as a single index vertex will support.

- Leaf vertices are always added left to right.

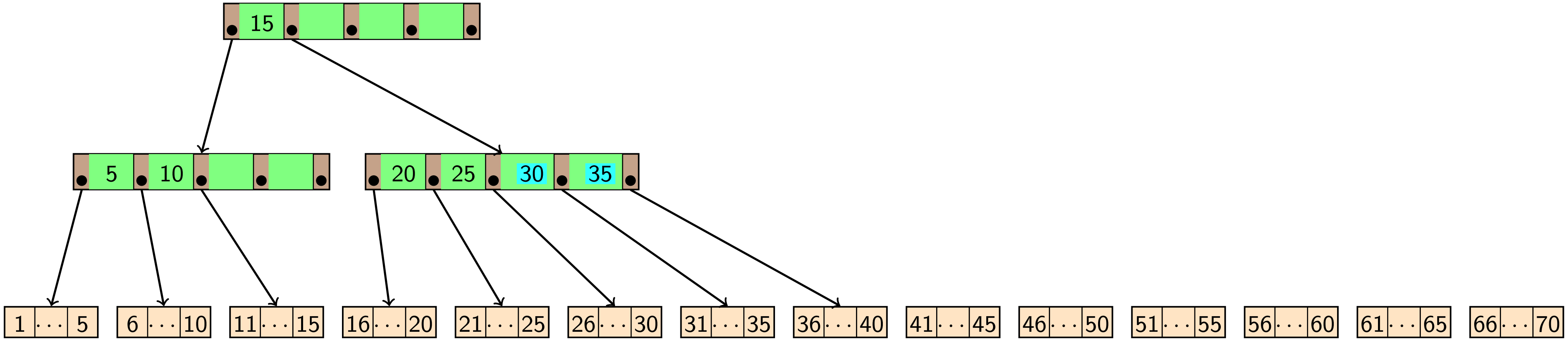


- Adding the next leaf vertex forces a split of the root.

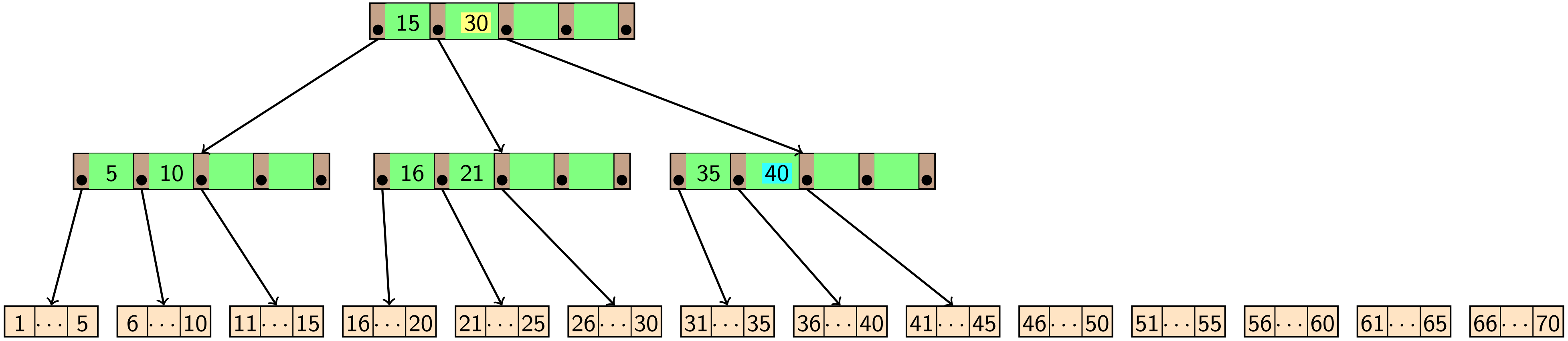


Bulk Loading of B$^+$-Trees — 2

- Now add leaf vertices until the rightmost index vertex is full.

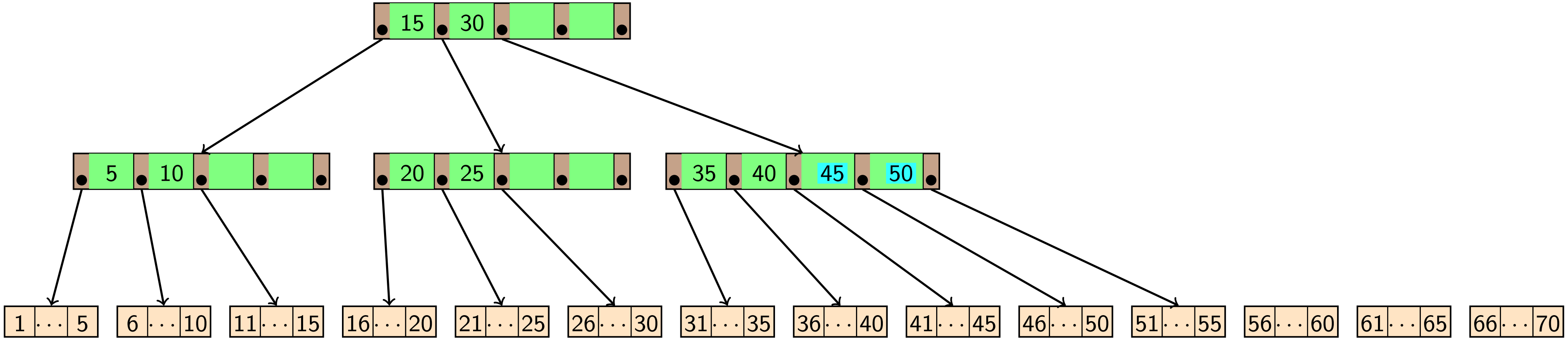


- Adding the next leaf vertex forces a split of the rightmost leaf vertex.

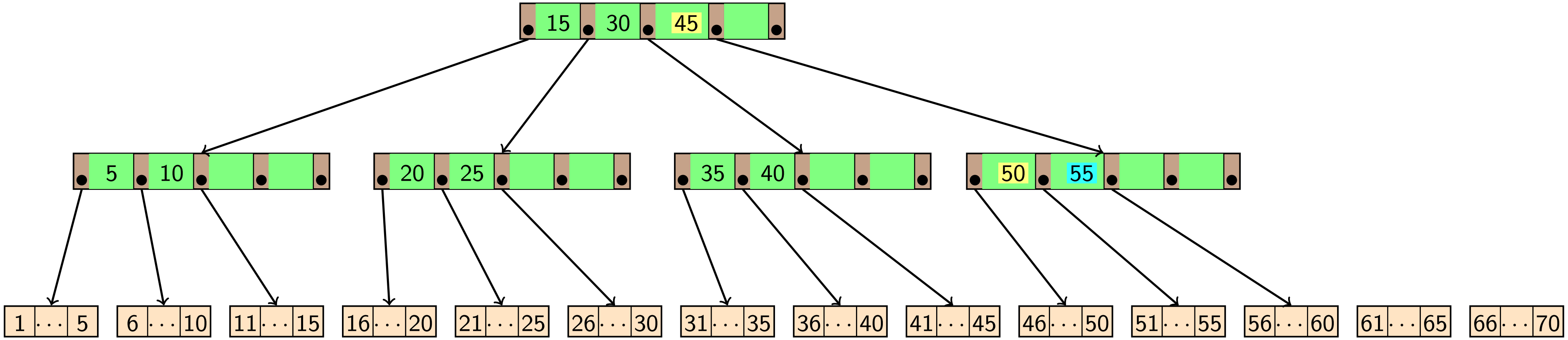


Bulk Loading of B$^+$-Trees — 3

- Again add leaf vertices until the rightmost index vertex is full.

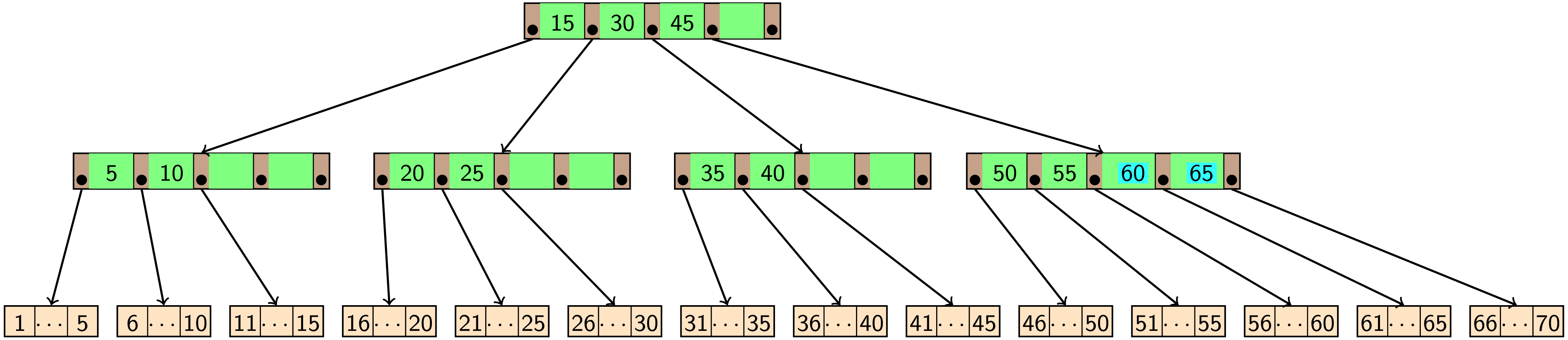


- Adding the next leaf vertex again forces a split of the rightmost leaf vertex.



Bulk Loading of B^+-Trees — 4

- Keep going until all leaf records are incorporated into the tree.



- The tree always grows by adding new vertices from the right, just below the leaves.
- Keys are added directly only to the rightmost index vertex which points to leaves.
- Eventually, the parent of the rightmost index vertex will fill up and must be split.
- Note that all index vertices, save for those which are on the rightmost path from the root, remain only half full.

Bulk Loading vs. Bulk Insertion

Bulk loading: Build a new index on top of a sorted list of leaf vertices.

Bulk insertion: Insert a large set of new records into an existing B$^+$-tree.

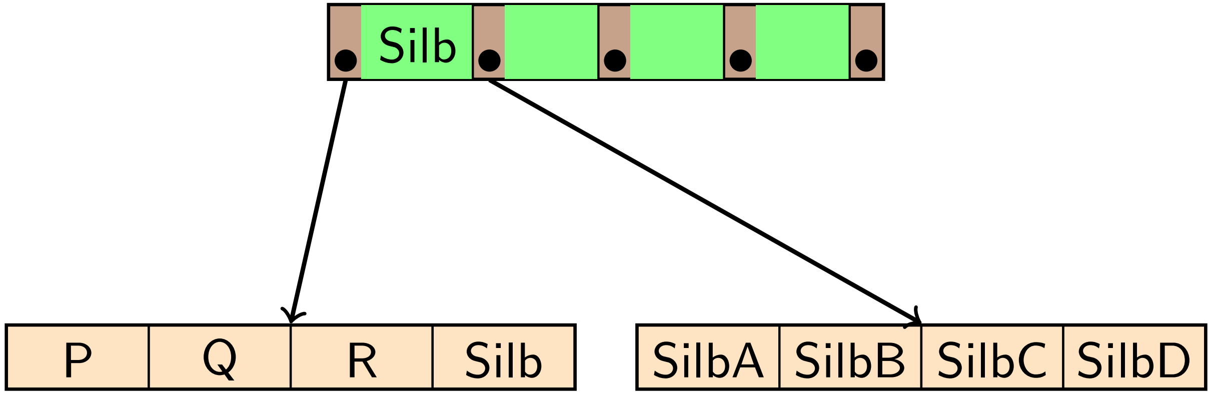
- Bulk insertion is much more difficult to do efficiently than bulk loading.
- There are no clear-cut winners, but there are some heuristics which can be followed.

Insert in order: The most important heuristic to follow when doing bulk insertion is to insert the records in order.

- This will minimize the number of writes to leaf vertices.
- This will allow several elements to be inserted at once, provided there is room in the leaf vertex.

Prefix Compression

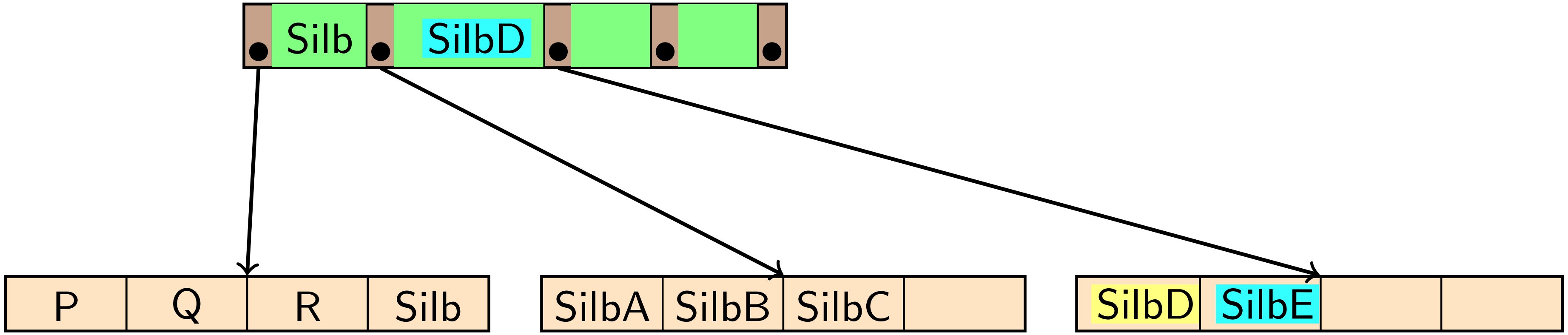
- The length of a full key can be quite long.
- For example, in the instructor relation of the university schema, the name field is `VARCHAR(20)`.
- An index for that key would require index vertices with 20 bytes reserved for each key value.
- This would result in relatively few keys per index, and a consequently deep tree.
- One way around this would be to use only a fixed-length prefix of the full string.
- An example for a prefix length of four is shown below.



Prefix Compression — 2

Problem: If too many records begin with the same prefix, a problem occurs.

- Consider inserting `SilbE` into the tree on the previous slide, as shown below.



- Now the key in the index must be increased in length from four to five.

- This implies that for such a prefix compression scheme to work, variable-length key fields in the index must be allowed.

- It is possible to do this by varying the number of keys in an index vertex.

Prefix Compression — 3

- To allow a variable-length key field in a vertex of fixed size, the number of key fields must be variable.
- This, however, creates a slowdown in accessing the k^{th} index in an index vertex, because the offset is not fixed.
- The performance degradation can be minimized by having as single bit in the vertex which indicates whether any of the indices are over the fixed length.
- If the bit is not set, access can proceed following the fixed-length model of a key.

Prefix Compression and Multi-Attribute Keys

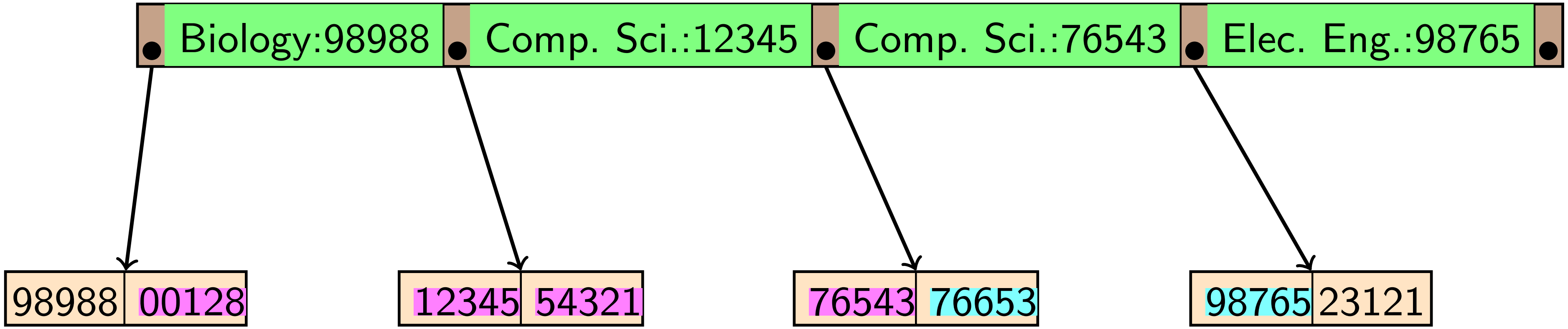
- In the case of multi-attribute, variable-length keys, the compression problem is even more severe.

Example: Suppose that both (instructor) `name` (`VARCHAR(20)`) and and `dept_name` (`VARCHAR(20)`) are used as a combined index.

- If the two are to be concatenated to form a single string for the key, then at least the first string must be padded out with spaces, which wastes space.
- The solution is to use a clever encoding which actually produces two strings, one for comparison for greater than, and a second for less than.
- The details are not presented here.

Non-Unique Search Keys for B$^+$-Trees

- It is possible to use a B$^+$-tree index even if the index field is not a (candidate) key.
- In this case, without further measures, an index value may identify several records.
- This can cause inefficiencies in both searching and in update operations.
- The usual solution is to append a key to the search index.
- This is illustrated below for an index by department on the `student` relation, with the student `ID` appended.
- The keys in fuchsia identify Computer Science students, while those in cyan identify Electrical Engineering students.



Biology:98988 | Comp. Sci.:12345 | Comp. Sci.:76543 | Elec. Eng.:98765

98988 | 00128

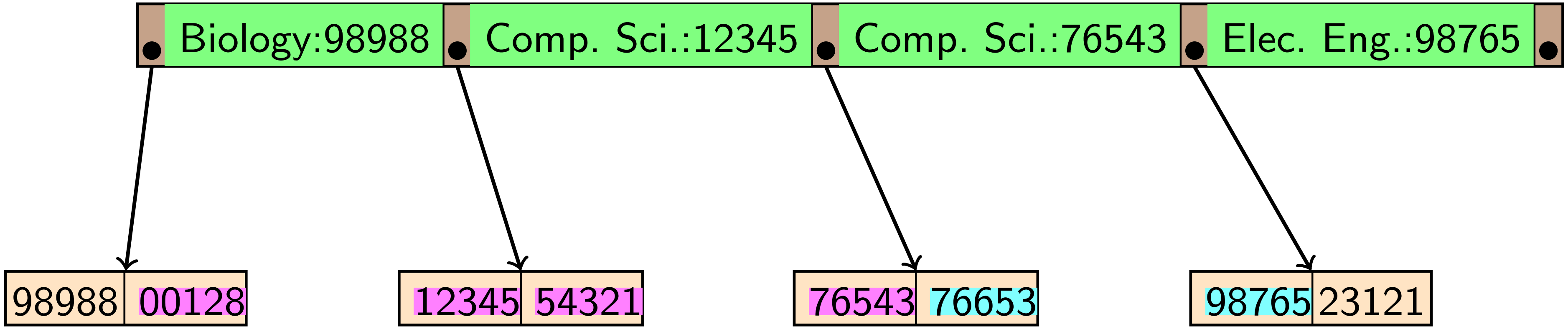
12345 | 54321

76543 | 76653

98765 | 23121

Secondary Search Keys for B$^+$-Trees

- The records of a B$^+$-tree can only be ordered on one attribute.
- If a second index is created, the leaf vertices contain either a key or else a pointer identifying the actual record.
- If a key is kept, a second search using an index based upon that key will be required.
- If a pointer to the record is kept, that pointer must be updated if the record is moved (due to operations on the B$^+$-tree for the index using the key.)
- It is a performance decision to choose which is best for a given situation.



Biology:98988 Comp. Sci.:12345 Comp. Sci.:76543 Elec. Eng.:98765

98988 00128

12345 54321

76543 76653

98765 23121

B*-Trees

- A *B*-tree* is structurally identical to a B-tree; however, the insertion and deletion algorithms are designed to ensure that every non-root vertex is two-thirds full, not just half full.
- B*-trees thus make better use of storage space.
- In a B-tree, when it is necessary to insert into a full vertex, there are often two possibilities:

  Split: Split the vertex into two; move the middle element to the parent.

  Rotate through the parent: If a sibling has some room, rotate through the parent in order to make room for the insertion.
- Similarly, when it is necessary to delete from a half-full vertex, there are often two possibilities:

  Combine: Combine the vertex with one of its neighbors, moving the common parent element down as well.

  Rotate through the parent: If a sibling is more than half full, rotate through the parent in order to leave the vertex full enough after the deletion.
- In a *B*-tree*, such rotation is mandatory whenever possible.

B*-Trees — 2

Deletion in B*-trees: Deletion for B*-trees is more complex than for B-trees in that to preserve two-thirds fullness, it may be necessary to combine three siblings into two rather than two into one.

  - However, the idea of the algorithm is straightforward.

- In short, B*-trees are structurally identical to B-trees; they just make use of insertion and deletion algorithms which ensure a higher level of fullness.

Extension to B$^+$-trees: These ideas extend to B$^+$-trees as well.

  - The ideas are similar and will not be elaborated here.

Higher levels of fullness: In principle, it is possible to guarantee an even higher level of fullness by working with a greater number of siblings at once.

  - However, the complexity of the algorithm outweighs the benefits and so the idea is seldom seen in practice.

Bitmap Indices

- Suppose that some survey data are given.
- Suppose further that range queries on `Age` and `Gender` are to be supported, for example:

```
SELECT * FROM Survey WHERE (SEX='F') AND (60 <= AGE) AND (AGE < 79);
```

Survey

ID	Sex	Age	Amount	City
11111111	F	46	5321	Stockholm
22222222	F	63	5000	Göteborg
33333333	M	62	7125	Trelleborg
44444444	F	23	9100	Tillberga
55555555	M	28	1200	Tillberga
66666666	F	68	5500	Malmö
77777777	F	42	5500	Simrishamn

Bitmap Indices

- Suppose that some survey data are given.
- Suppose further that range queries on `Age` and `Gender` are to be supported, for example:

  `SELECT * FROM Survey WHERE (SEX='F') AND (60 <= AGE) AND (AGE < 79);`
- It may then be useful to have a *bitmap index* which allows such retrieval based upon matching of bits.

Survey

ID	Sex	Age	Amount	City
11111111	F	46	5321	Stockholm
22222222	F	63	5000	Göteborg
33333333	M	62	7125	Trelleborg
44444444	F	23	9100	Tillberga
55555555	M	28	1200	Tillberga
66666666	F	68	5500	Malmö
77777777	F	42	5500	Simrishamn

Bitmap

ID	Sex	0-19	20-39	40-59	60-79	80-
11111111	1	0	0	1	0	0
22222222	1	0	0	0	1	0
33333333	0	0	0	0	1	0
44444444	1	0	1	0	0	0
55555555	0	0	1	0	0	0
66666666	1	0	0	0	1	0
77777777	1	0	0	1	0	0

Bitmap Indices

- Suppose that some survey data are given.
- Suppose further that range queries on `Age` and `Gender` are to be supported, for example:

```
SELECT * FROM Survey WHERE (SEX=’F’) AND (60 <= AGE) AND (AGE < 79);
```

- It may then be useful to have a *bitmap index* which allows such retrieval based upon matching of bits.
- The *bitmap* may be represented compactly as a single string.
- Standard hardware instructions for bit manipulation may then be used for rapid processing.
- The bitmap is represented as a relation, but is in fact an index on `ID` and may be implemented in a number of ways.

Survey

ID	Sex	Age	Amount	City
11111111	F	46	5321	Stockholm
22222222	F	63	5000	Göteborg
33333333	M	62	7125	Trelleborg
44444444	F	23	9100	Tillberga
55555555	M	28	1200	Tillberga
66666666	F	68	5500	Malmö
77777777	F	42	5500	Simrishamn

Bitmap

ID	Sex	0-19	20-39	40-59	60-79	80-
11111111	1	0	0	1	0	0
22222222	1	0	0	0	1	0
33333333	0	0	0	0	1	0
44444444	1	0	1	0	0	0
55555555	0	0	1	0	0	0
66666666	1	0	0	0	1	0
77777777	1	0	0	1	0	0

Compact_Bitmap

ID	BitMap
11111111	100100
22222222	100010
33333333	000010
44444444	101000
55555555	001000
66666666	100010
77777777	100100

Bitmap Indices — Additional Compactification

- To represent n conditions, only $\lceil \log(n) \rceil$ bits are required.
- This suggests the compact representation given below, using the following table.

Age Range	Encoding $A_1A_2A_3$
0-20	000
21-39	001
40-59	010
60-79	011
80-	100

Bitmap

ID	Sex	A_1	A_2	A_3
11111111	1	0	1	0
22222222	1	0	1	1
33333333	0	0	1	1
44444444	1	0	0	1
55555555	0	0	0	1
66666666	1	0	1	1
77777777	1	0	1	0

Bitmap

ID	Sex	A_1	A_2	A_3
11111111	1	0	1	0
22222222	1	0	1	1
33333333	0	0	1	1
44444444	1	0	0	1
55555555	0	0	0	1
66666666	1	0	1	1
77777777	1	0	1	0

Compact_Bitmap

ID	BitMap
11111111	1010
22222222	1011
33333333	0011
44444444	1001
55555555	0001
66666666	1011
77777777	1010

Extendible Hashing

- The goal of extendible hashing is to realize the advantage of hashing within the context of data on secondary storage:
  - Fast (constant-time) random access

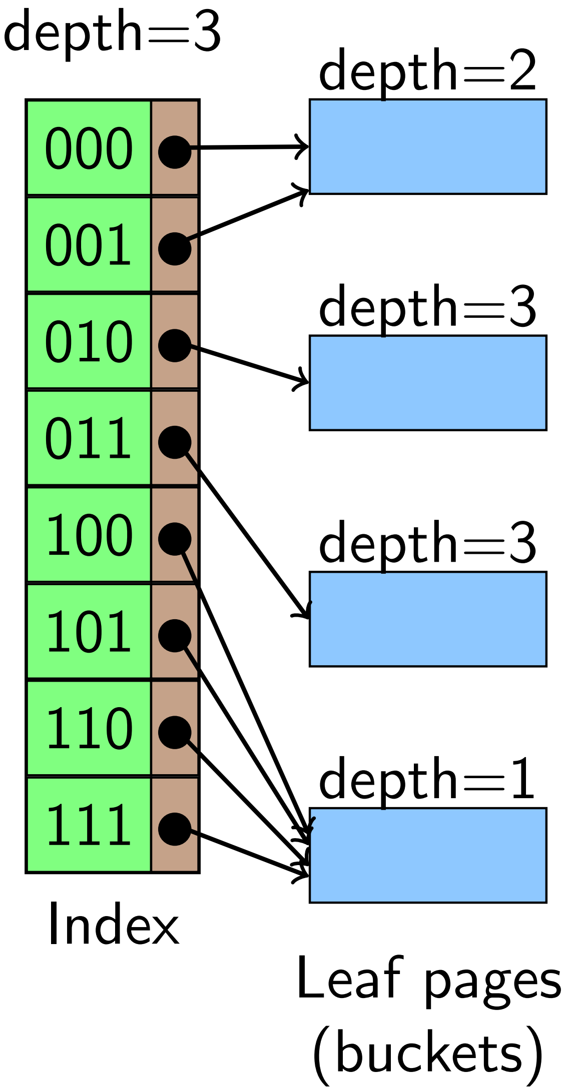
Idea: The hashing function h : keys → hash values is broken into two pieces: (Directory address, Leaf address).

Toy example: Suppose that a two-byte hash address is used:

Directory address size: 3 bits

Hash address size: 13 bits

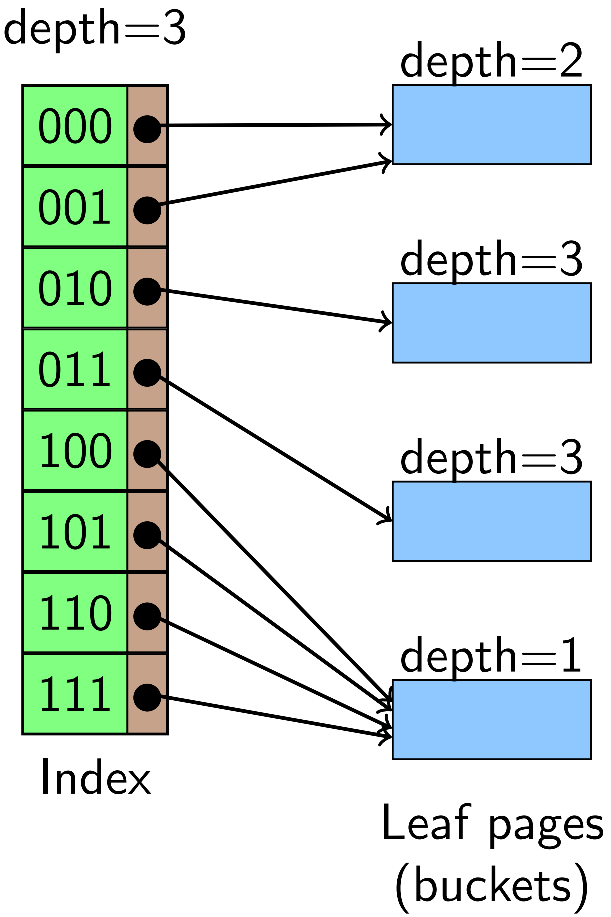
- Suppose that k is a key with the property that $h(k) = 1010111010110001$.
- Then, Directory address $= 101$, Leaf address $= 0111010110001$.
- This assumes that the first three bits are used as the directory address.



Extendible Hashing — 2

- The *depth* of an index is the number of bits of the hash value which is used as the index value.
- The *depth* of a leaf page is:

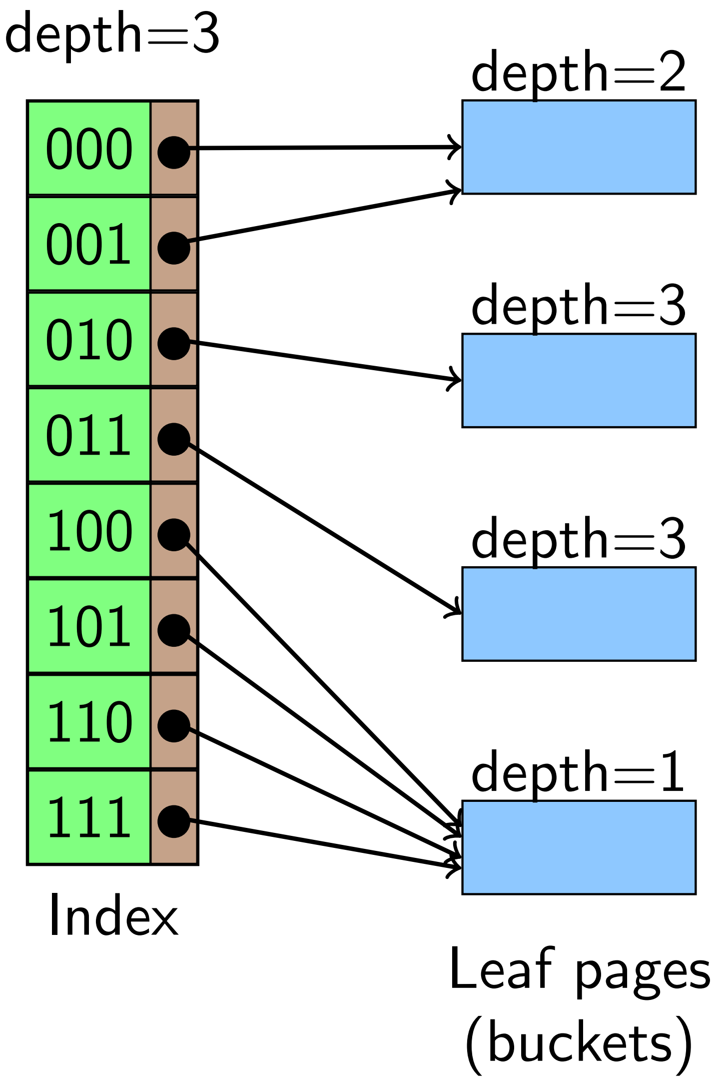
$$\text{Index depth} - \log_2(\text{number of index entries which point to that bucket})$$



- The approach supports insertions quite well.
- It is less efficient at handling deletions.
- Some examples will be used to illustrate the idea.

Extendible Hashing — Bucket Expansion

- Suppose that the bucket which is shared by 000 and 001 becomes full.
- To allow further insertions for keys beginning with 00, a split of this bucket is necessary.


depth=3

000

001

010

011

100

101

110

111

Index

depth=2

depth=3

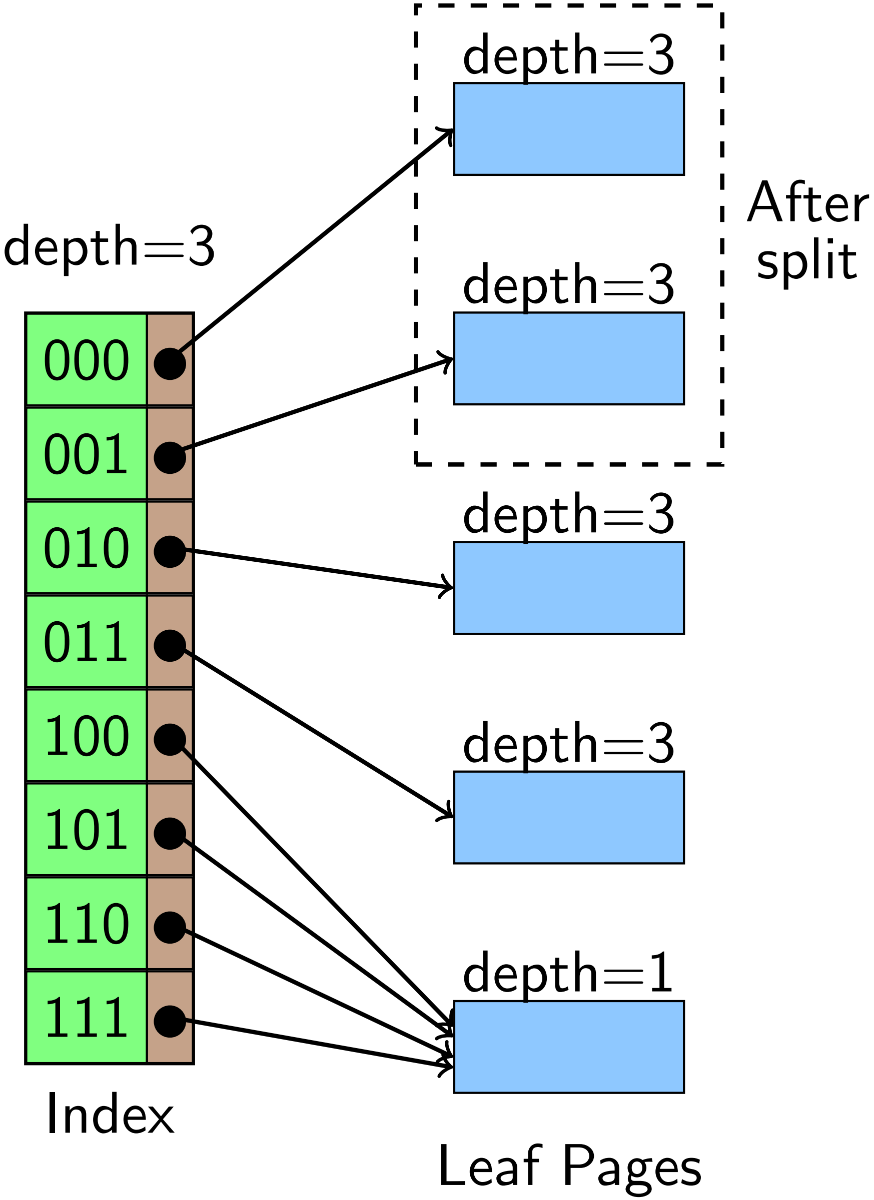
depth=3

depth=1

Leaf pages (buckets)

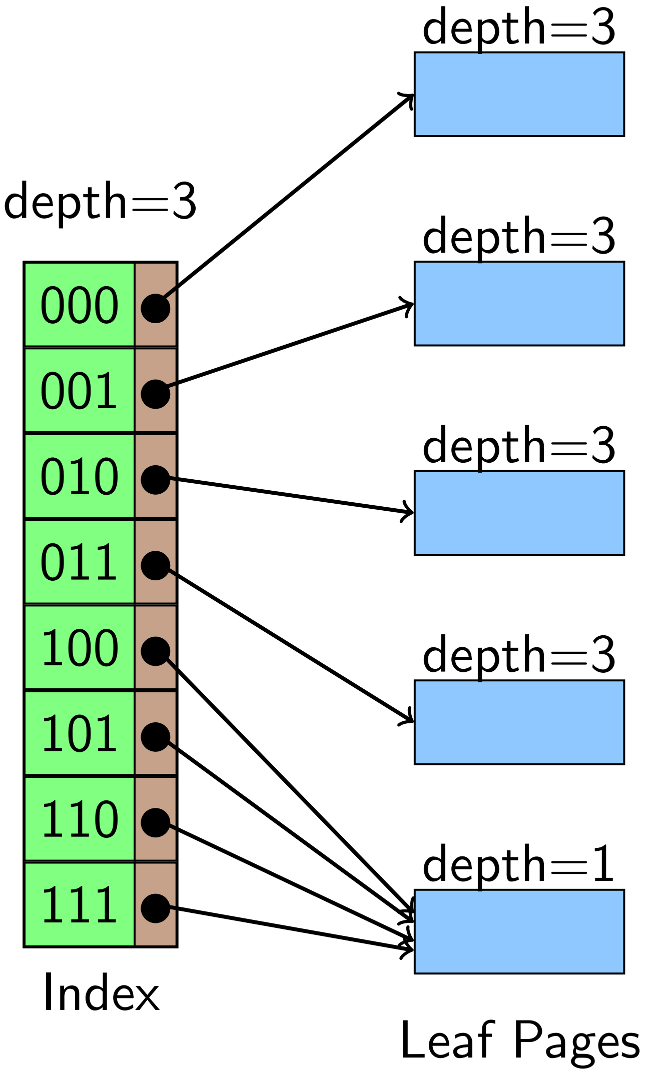
Extendible Hashing — Bucket Expansion

- Suppose that the bucket which is shared by 000 and 001 becomes full.
- To allow further insertions for keys beginning with 00, a split of this bucket is necessary.
- Notice that 000 and 001 now each have their own buckets.
- The entries of the old 000+001 bucket are divided appropriately between these two.



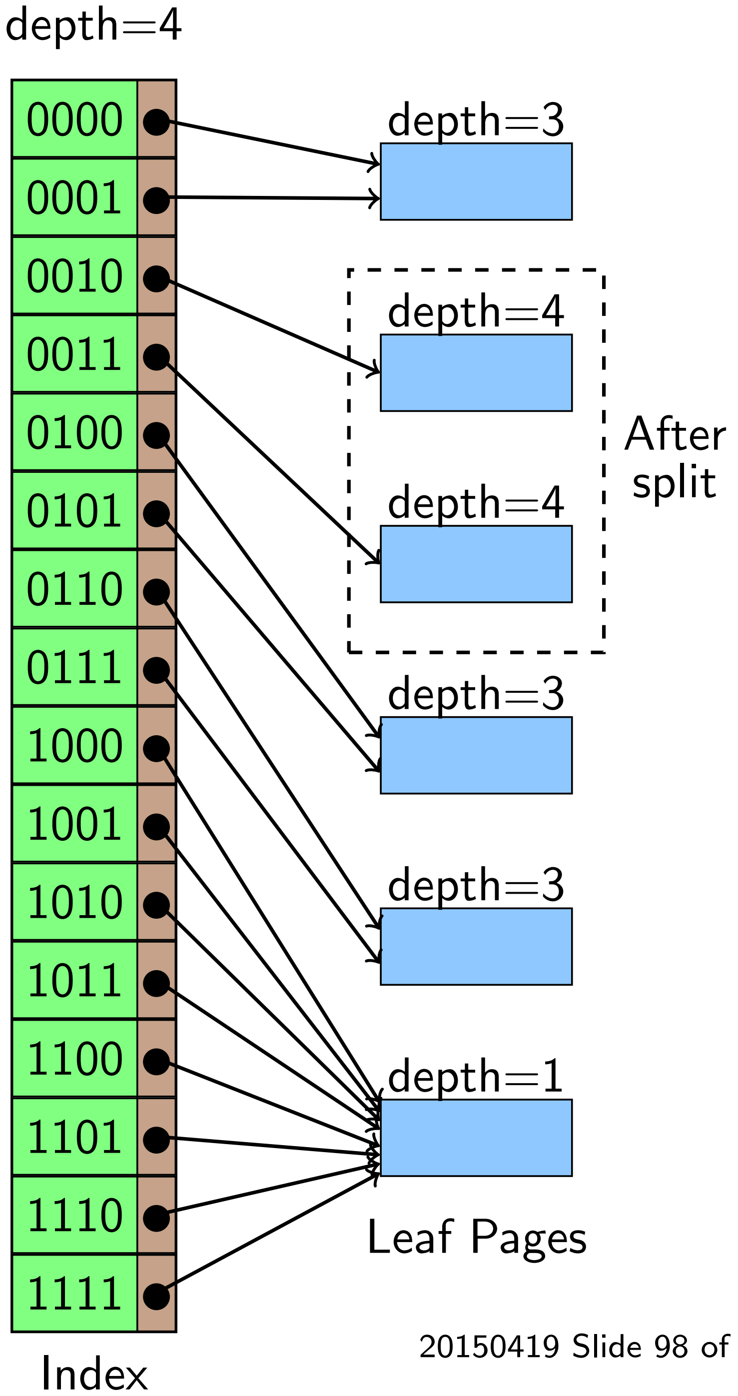
Extendible Hashing — Index Expansion

- Suppose that the bucket for 001 becomes full.
- To allow further insertions for keys beginning with 001, the index itself must be split.



Extendible Hashing — Index Expansion

- Suppose that the bucket for 001 becomes full.
- To allow further insertions for keys beginning with 001, the index itself must be split.
- The depth of the index becomes four, and the number of index entries doubles.
- The entries of the old 001 bucket are are divided appropriately between the 0010 bucket and the 0011 bucket.



Remarks Regarding Extensible Hashing

- Extendible hashing works best when insertions and modifications are the dominant forms of update.
- Random-access time may be somewhat superior to that for B+-trees, particularly when memory is limited.
  - The index for extendible hashing may be 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.