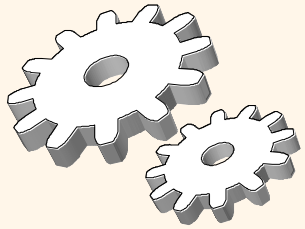
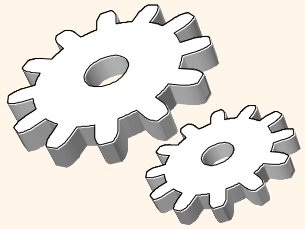
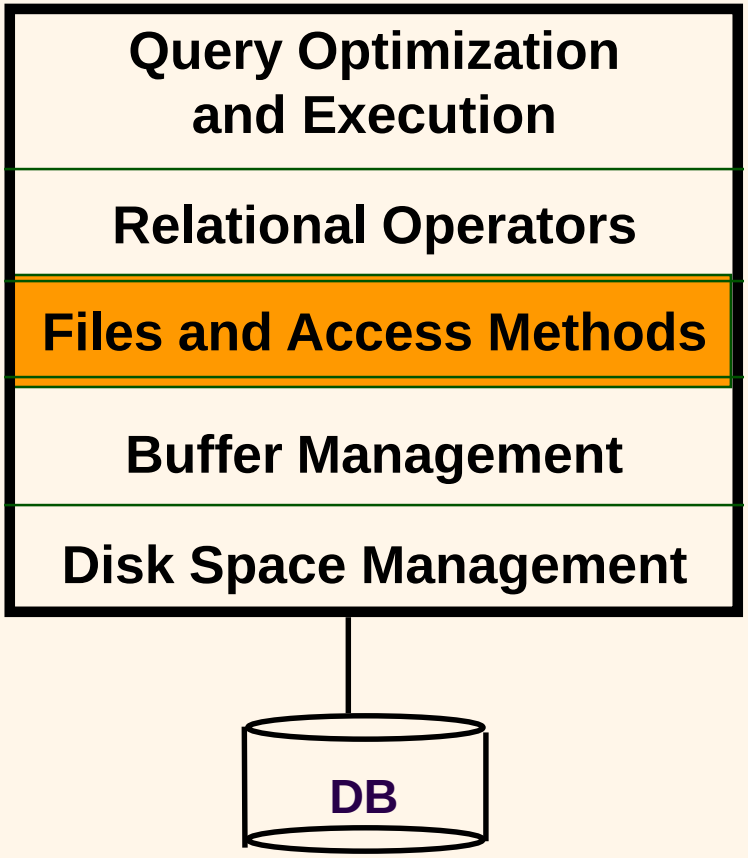


COMP7640
Database Systems & Administration

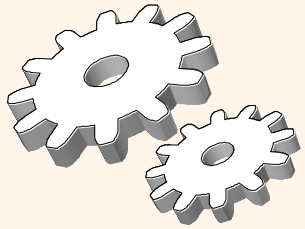
Tree-Structured Indexing

Where Are We Now?





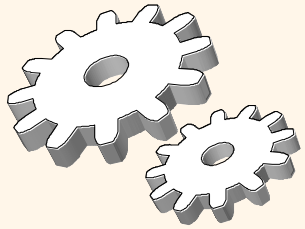
Range Search



- ``*Find all employees with age between 20 and 30''*
  - If data is in the sorted file,
    - do a binary search to find the first such employee
    - then scan to find others
  - Cost of binary search is $\lceil \log_2 B \rceil$
    - B is the number of pages for storing the records
  - This cost can be quite high.

- Solution: Create an `index' file

Tree-Structured Indexes

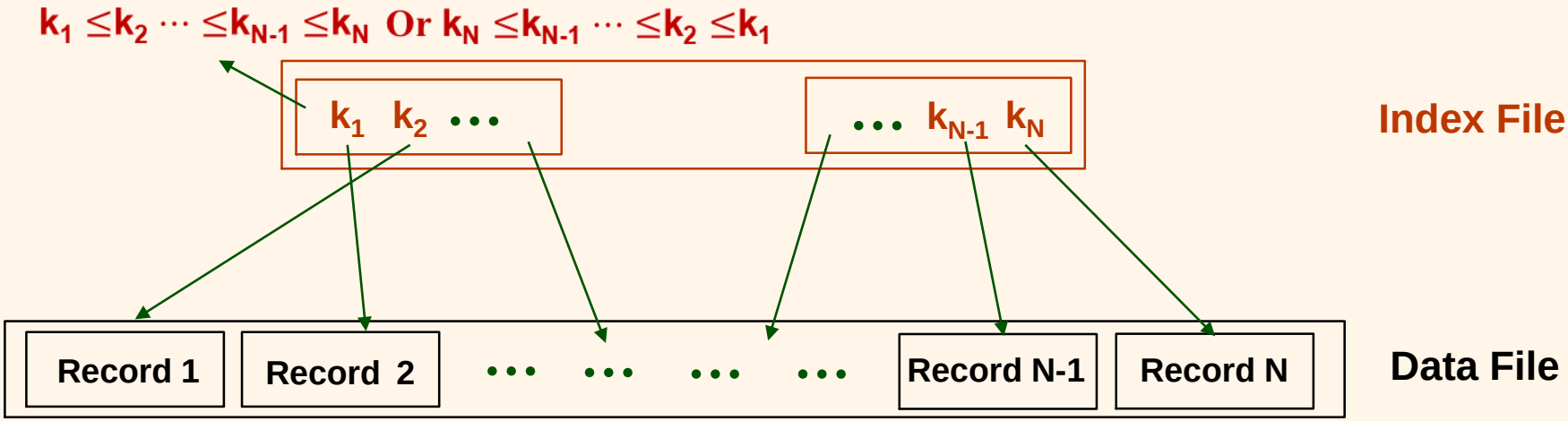


- Efficient support for *range search, insertion* and *deletion*

- *ISAM*
  - **Indexed Sequential Access Method**
  - **Static index structure**
- *B+ tree*
  - Dynamic structure that adjusts gracefully under insertions and deletions

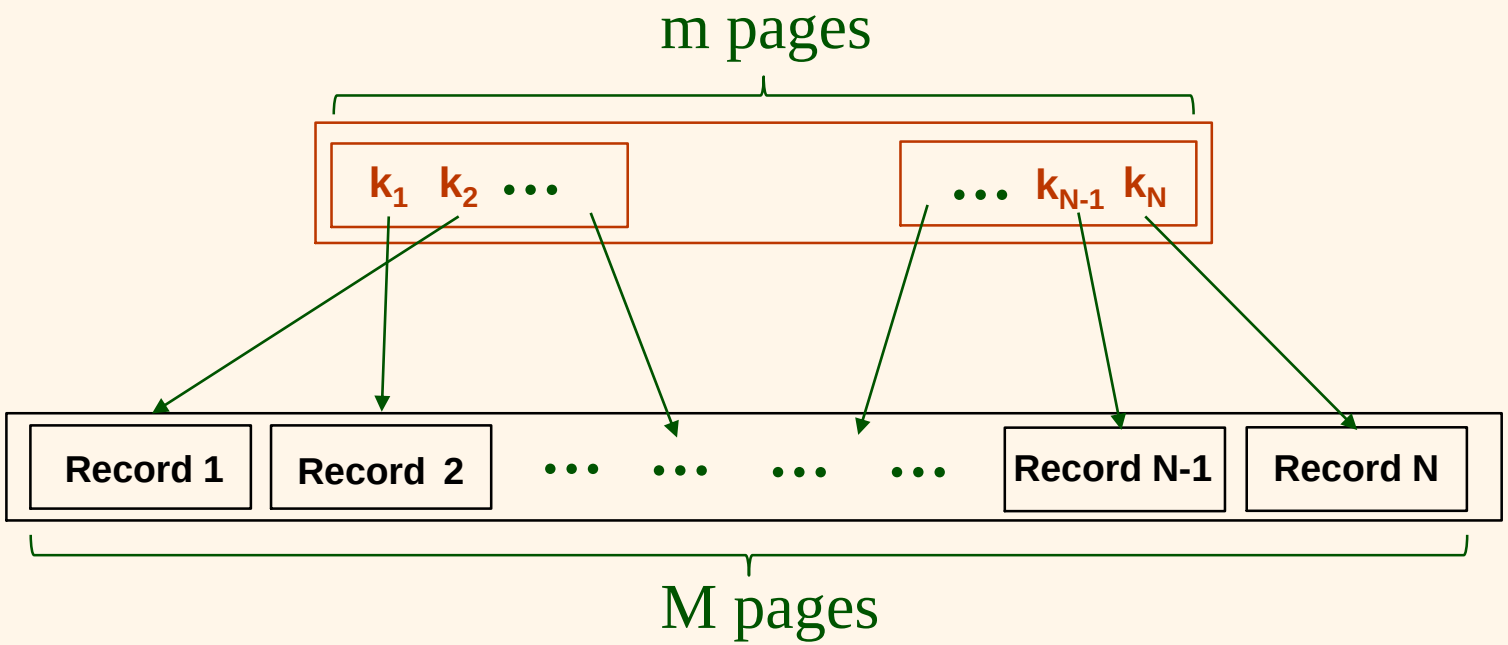
ISAM

- Index file
  - Each data entry corresponds to a record
  - Data entry: <search key value, record-id>
  - Data entries are *sorted* according to key values



Records are *not* required to be sorted according to key

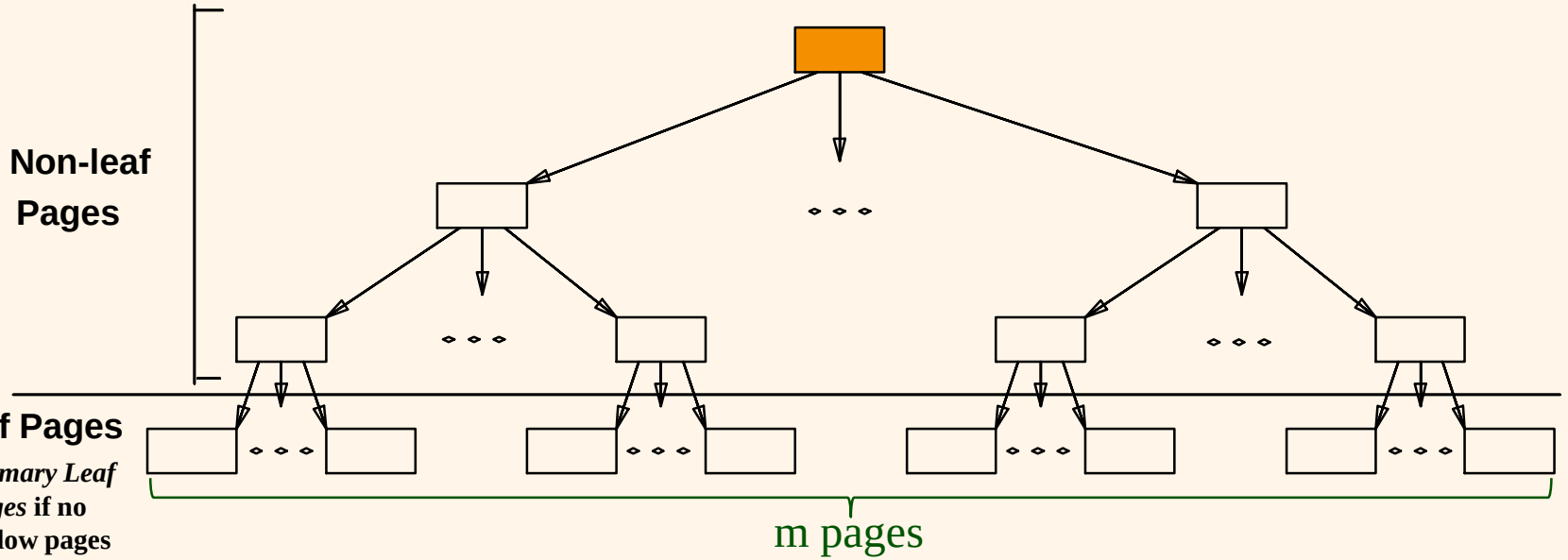
ISAM



- We can use the binary search approach to solve the range query, given a range of keys.
- Finding a range of keys in the index file has lower I/O costs compared with finding a range of keys in the data file ($\lceil \log_2 m \rceil < \lceil \log_2 M \rceil$) ☺

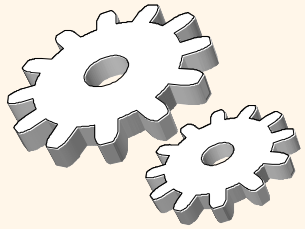
ISAM

- Index file (m pages) may still be quite large
  - Apply the idea of building an auxiliary file on the index file repeatedly until the file finally *fits on one page*



- *Leaf pages contain data entries <search key value, record-ID>.*
- *Non-leaf pages contain index entries <search key value, page-ID>.*

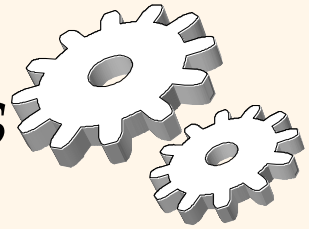
ISAM: File Creation and Search



- *File creation*
  1. Allocate leaf pages (containing data entries) sequentially, sorted by search key
  2. Allocate non-leaf (containing index entries) pages

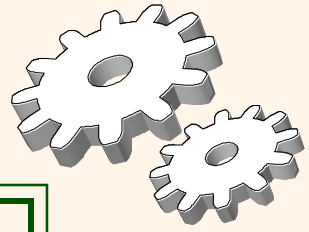
- *Search*
  - Start at root
  - Determine which subtree to continue search by comparing given key value with key values in the node

Example Relation Instance: Employees



Name	Age	Salary
Ashby	23	6003
Basu	28	4000
Bristow	31	4007
Cass	34	3500
Daniels	38	3200
Jones	42	5000
Smith	44	5008
Tracy	47	2800
Joan	50	3700
Billy	52	5006
Tony	60	4500
Bob	64	3000

Sorted File (by Age field)



- Assumptions:
  - Each page can only store one data record
  - Search key is age

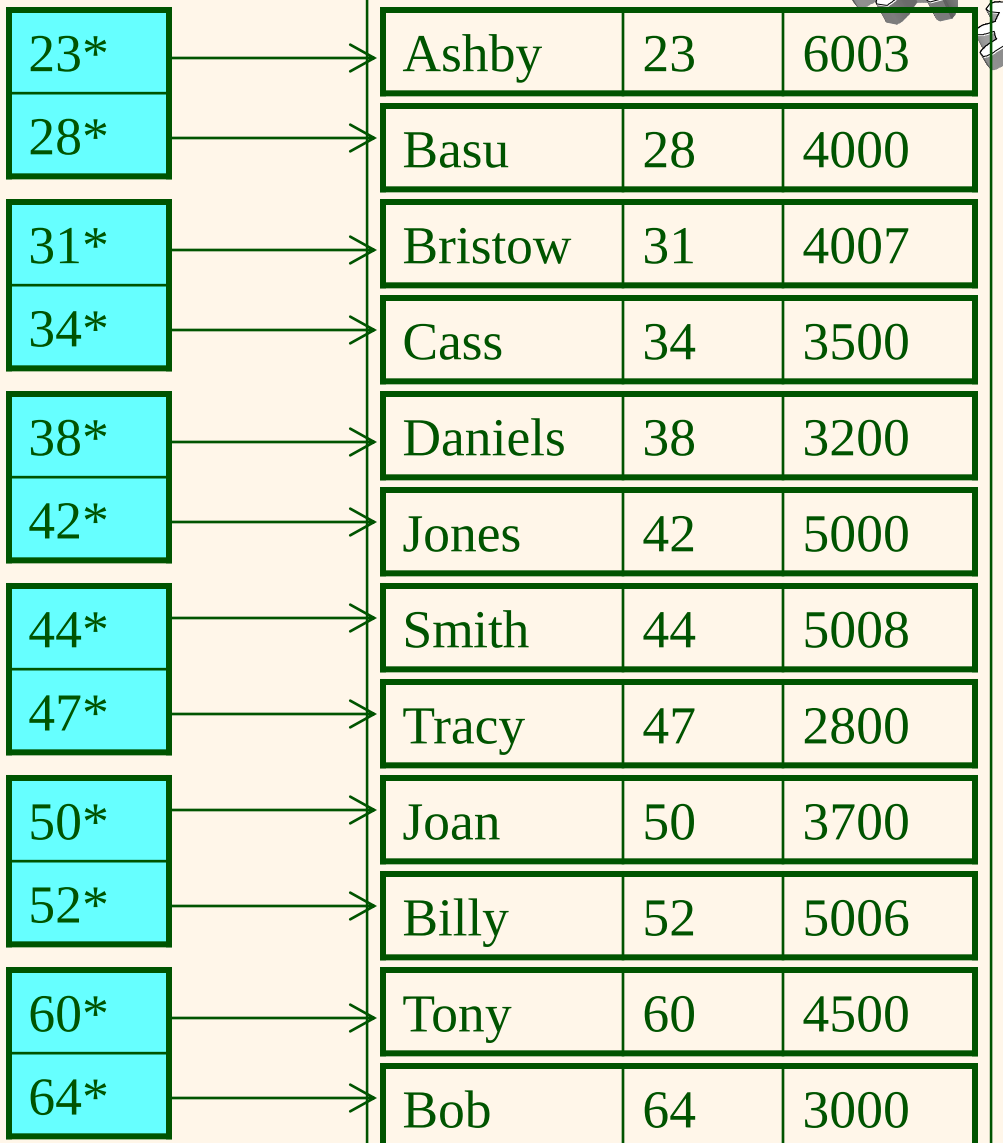
Ashby	23	6003
Basu	28	4000
Bristow	31	4007
Cass	34	3500
Daniels	38	3200
Jones	42	5000
Smith	44	5008
Tracy	47	2800
Joan	50	3700
Billy	52	5006
Tony	60	4500
Bob	64	3000

Leaf Pages of the Index File

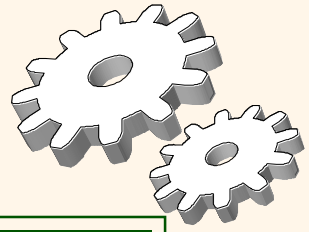
- Assumption:
  - Each page can store two data entries

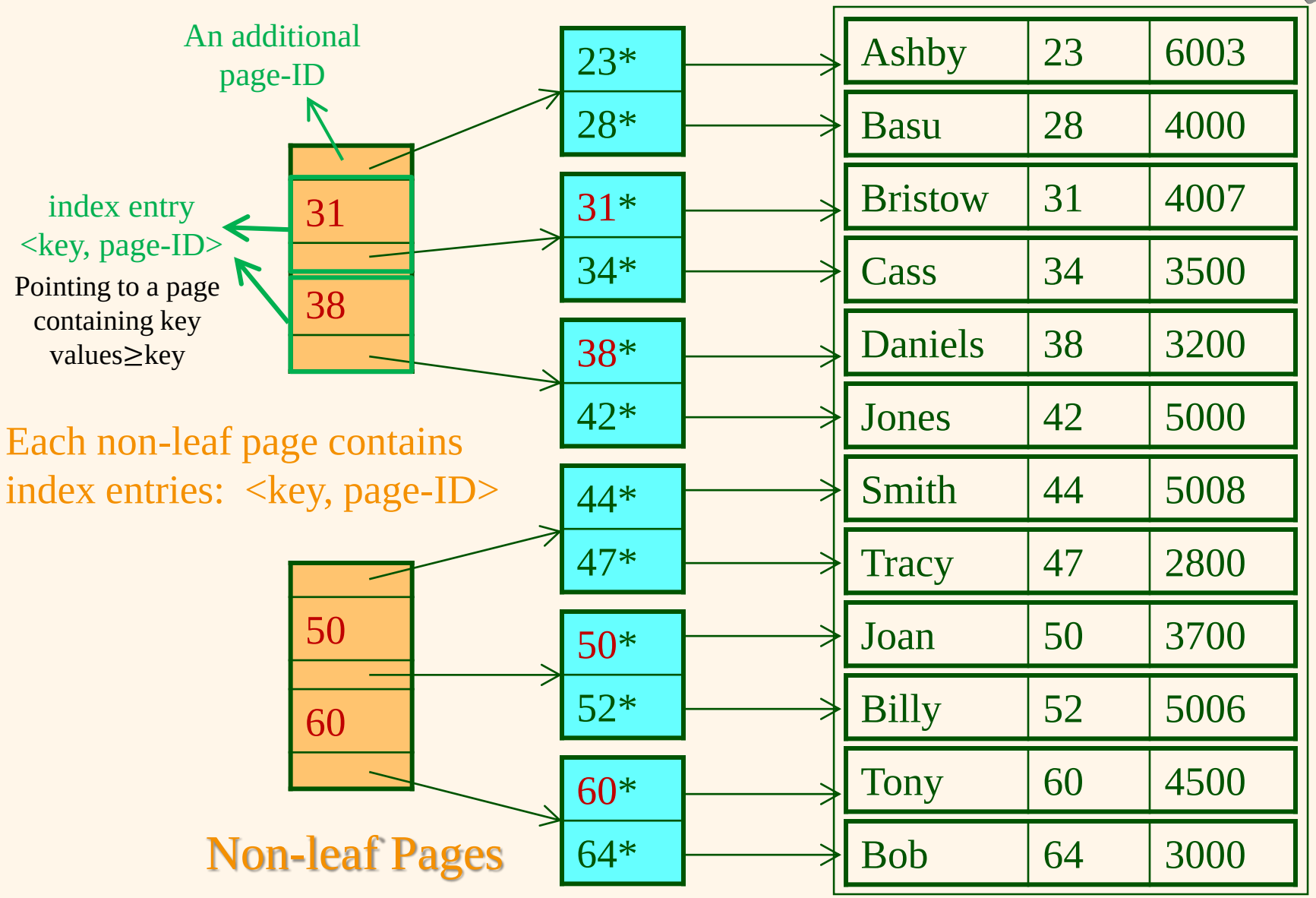
Each leaf page contains data entries: <key, record-ID>

We use key* to denote a data entry.

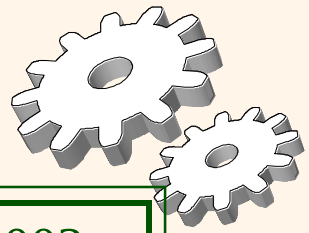


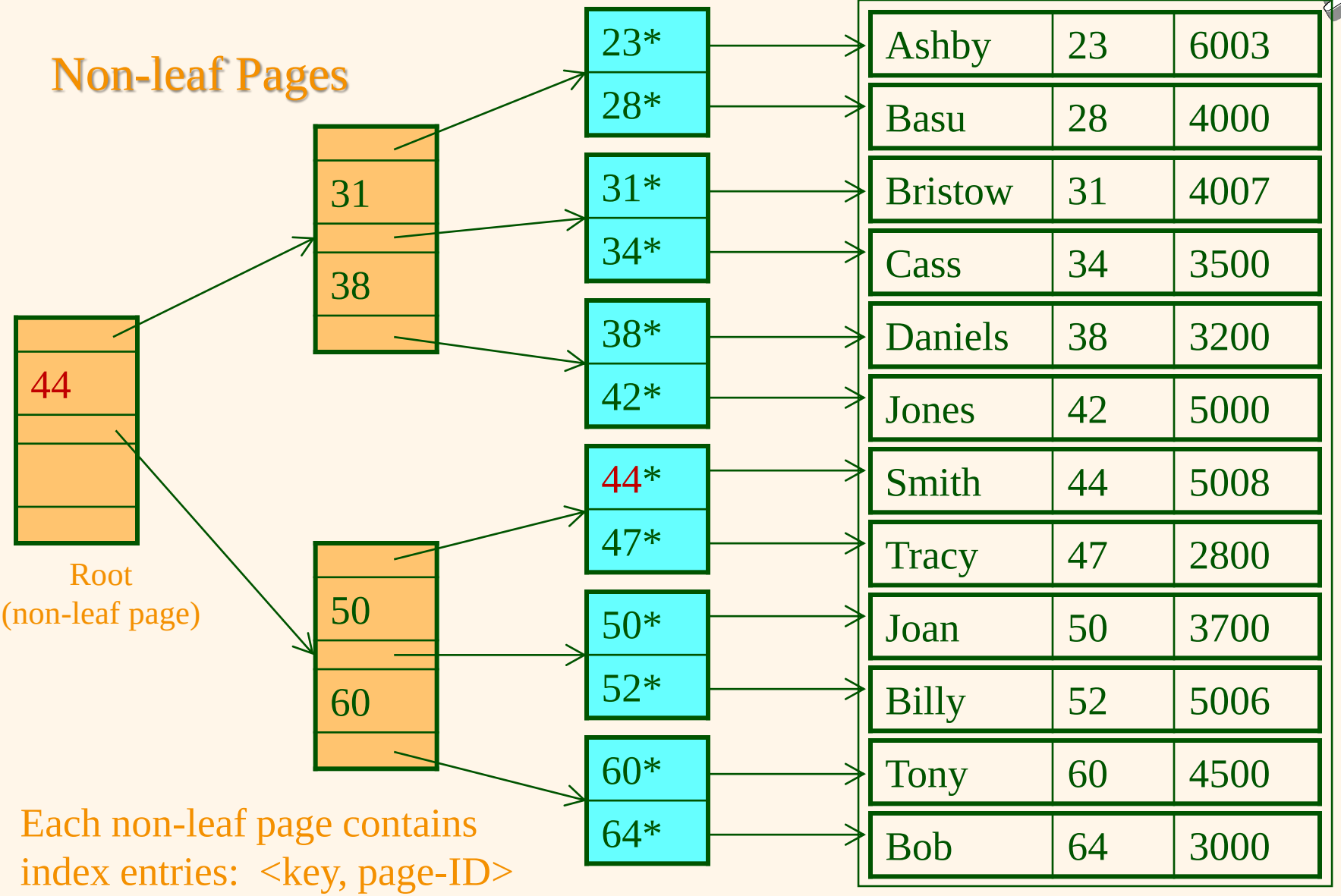
Non-leaf Pages of the Index File





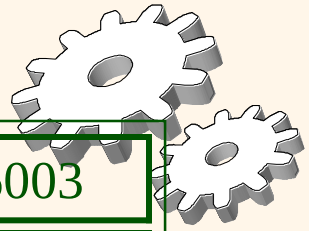
The root of the Index File

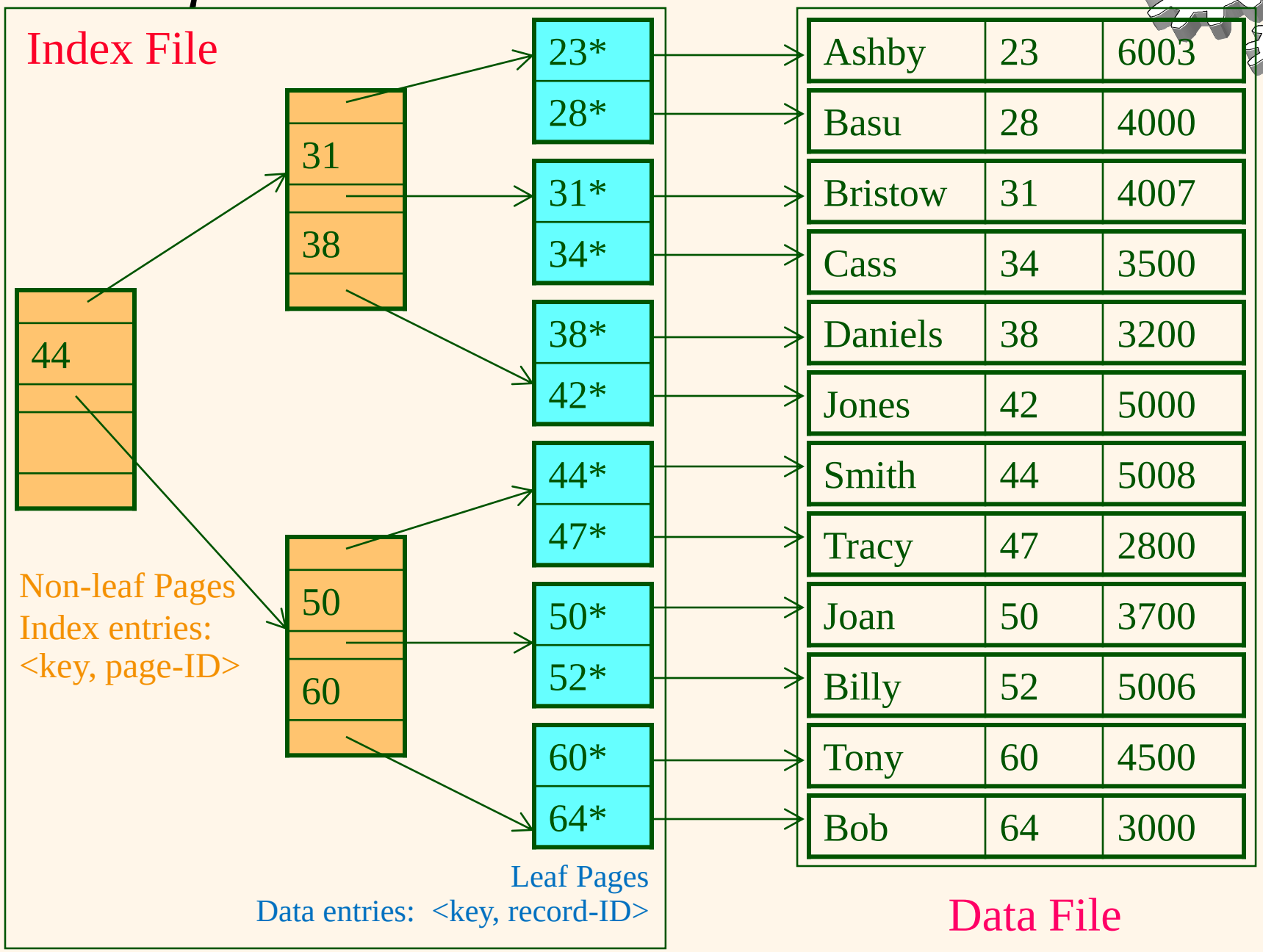




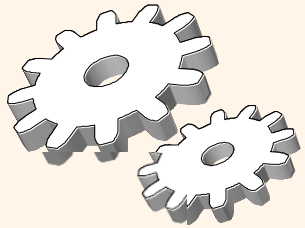
Each non-leaf page contains index entries: <key, page-ID>

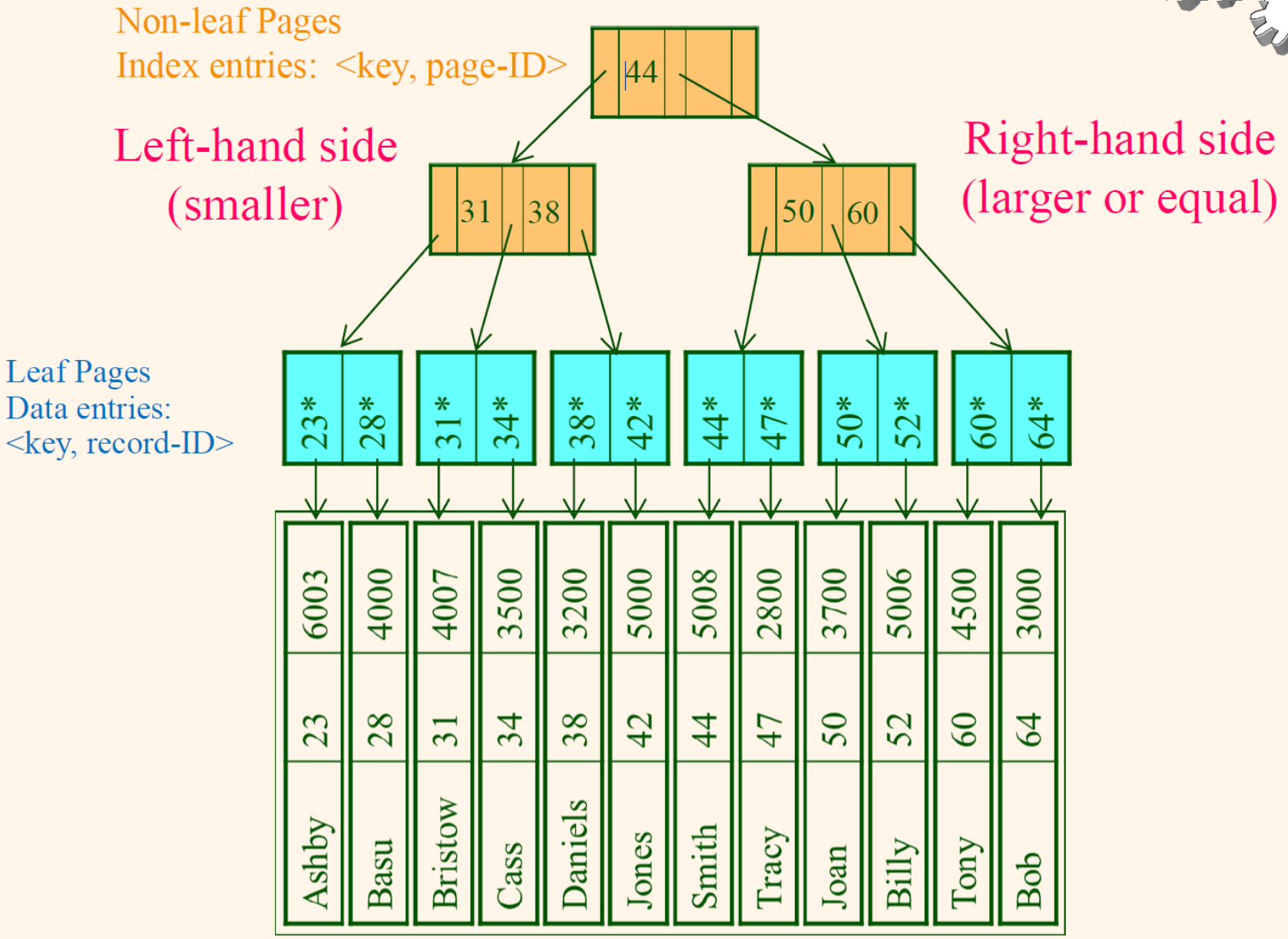
Complete Index File



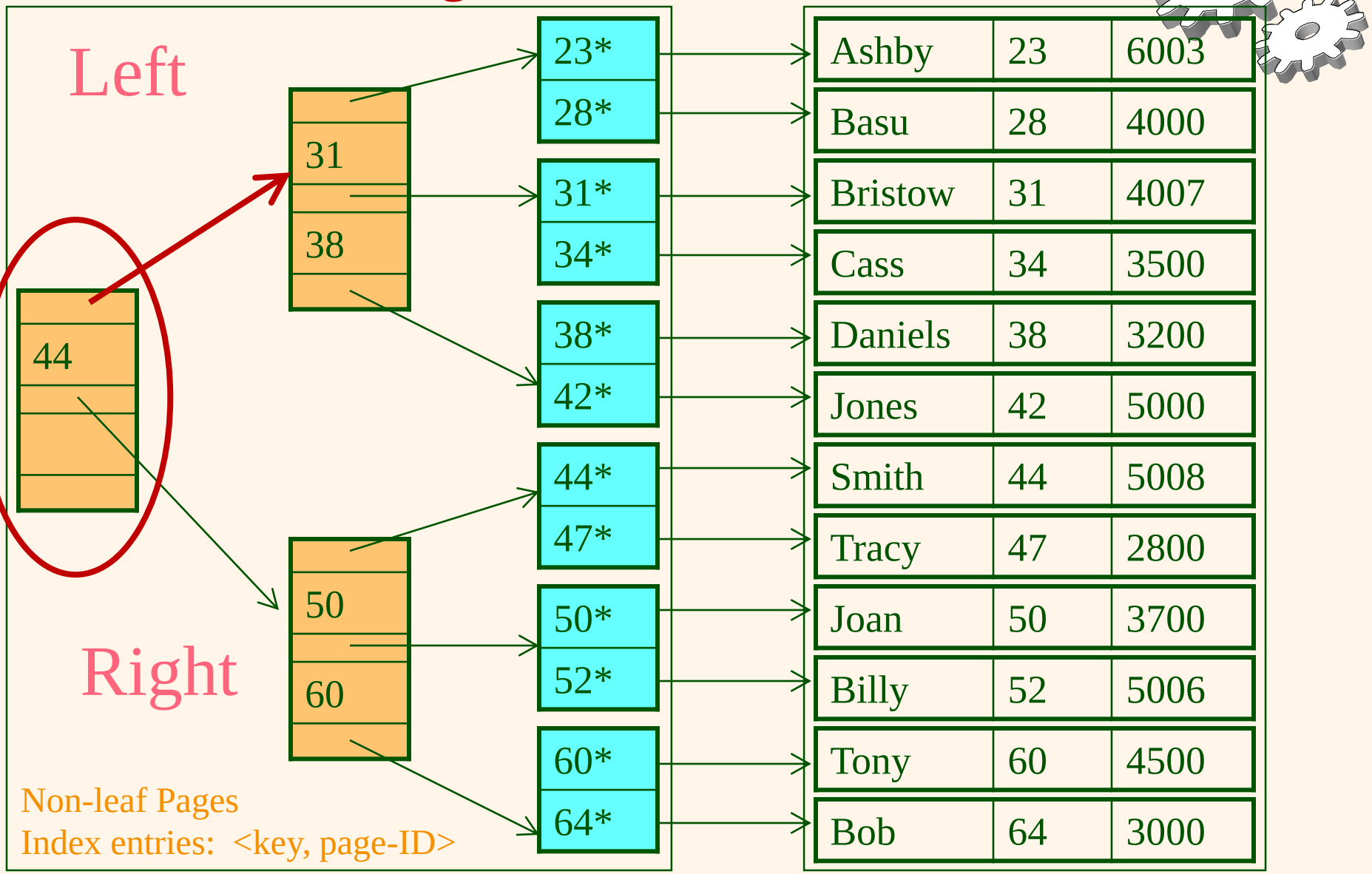


Complete Index File

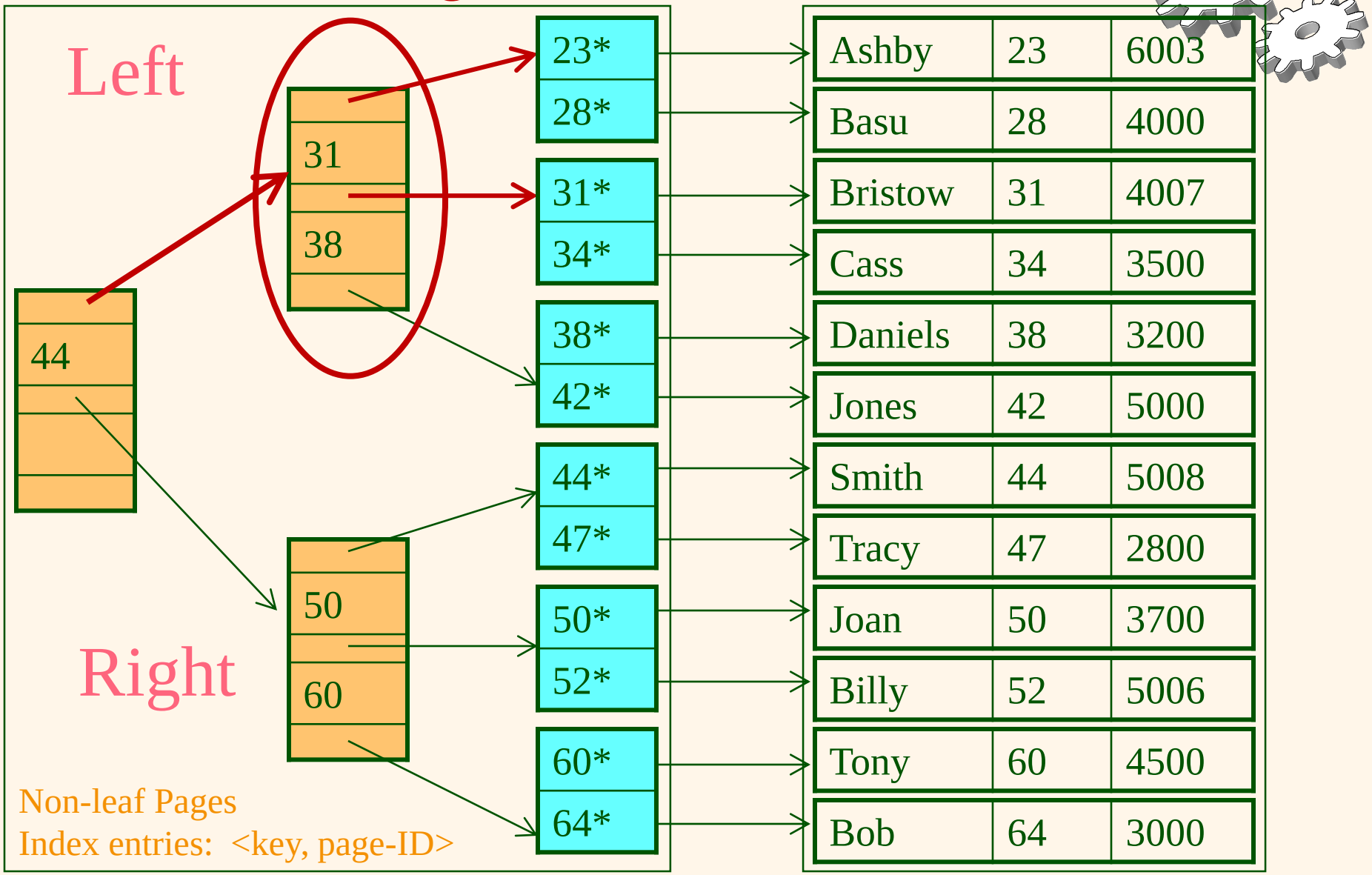




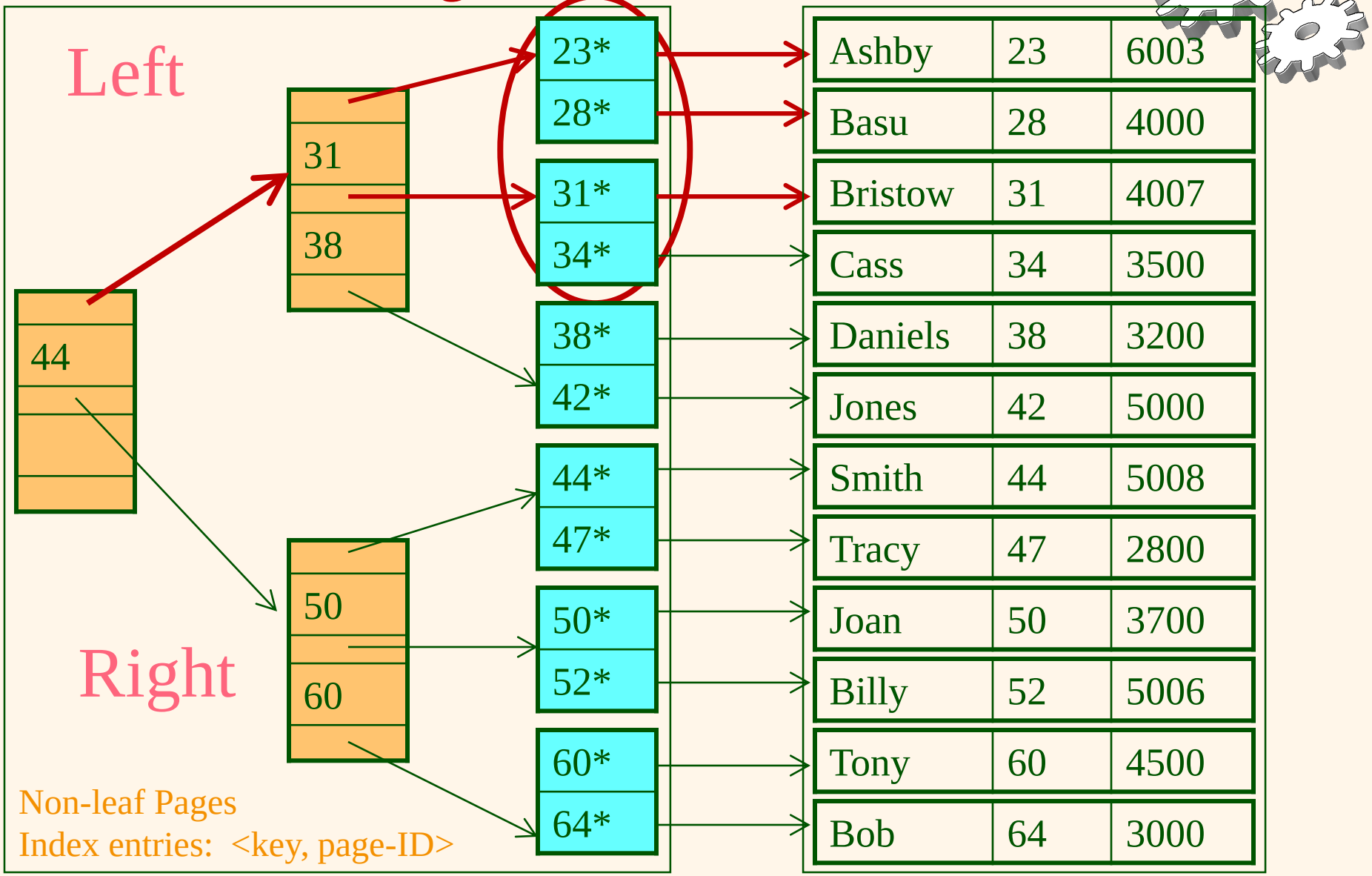
Search (22<Age<32) with the Index File



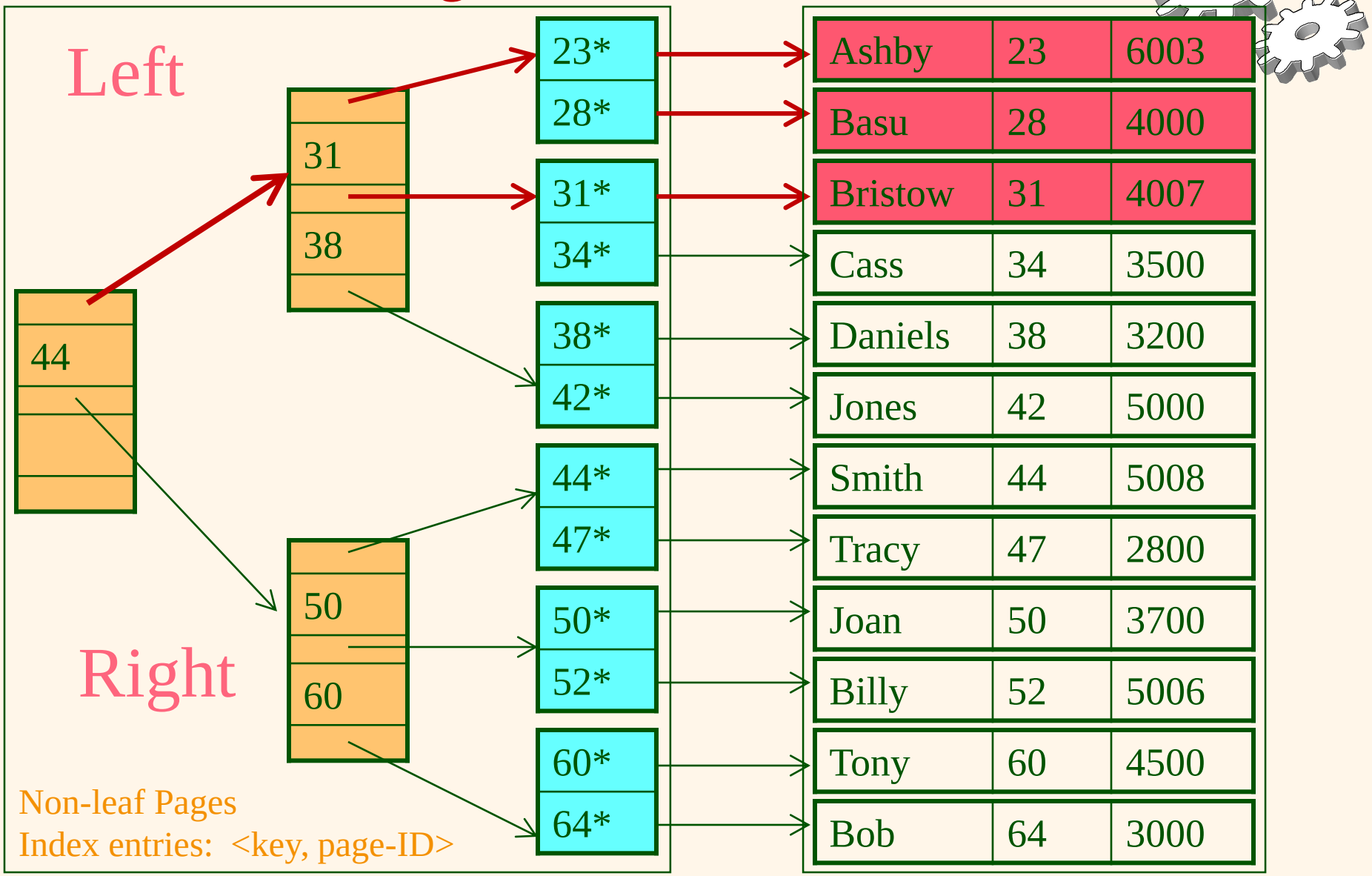
Search (22<Age<32) with the Index File



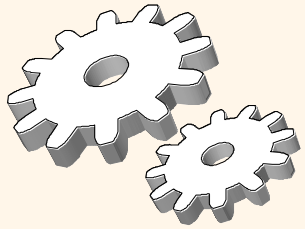
Search (22<Age<32) with the Index File



Search (22<Age<32) with the Index File



ISAM: Insert

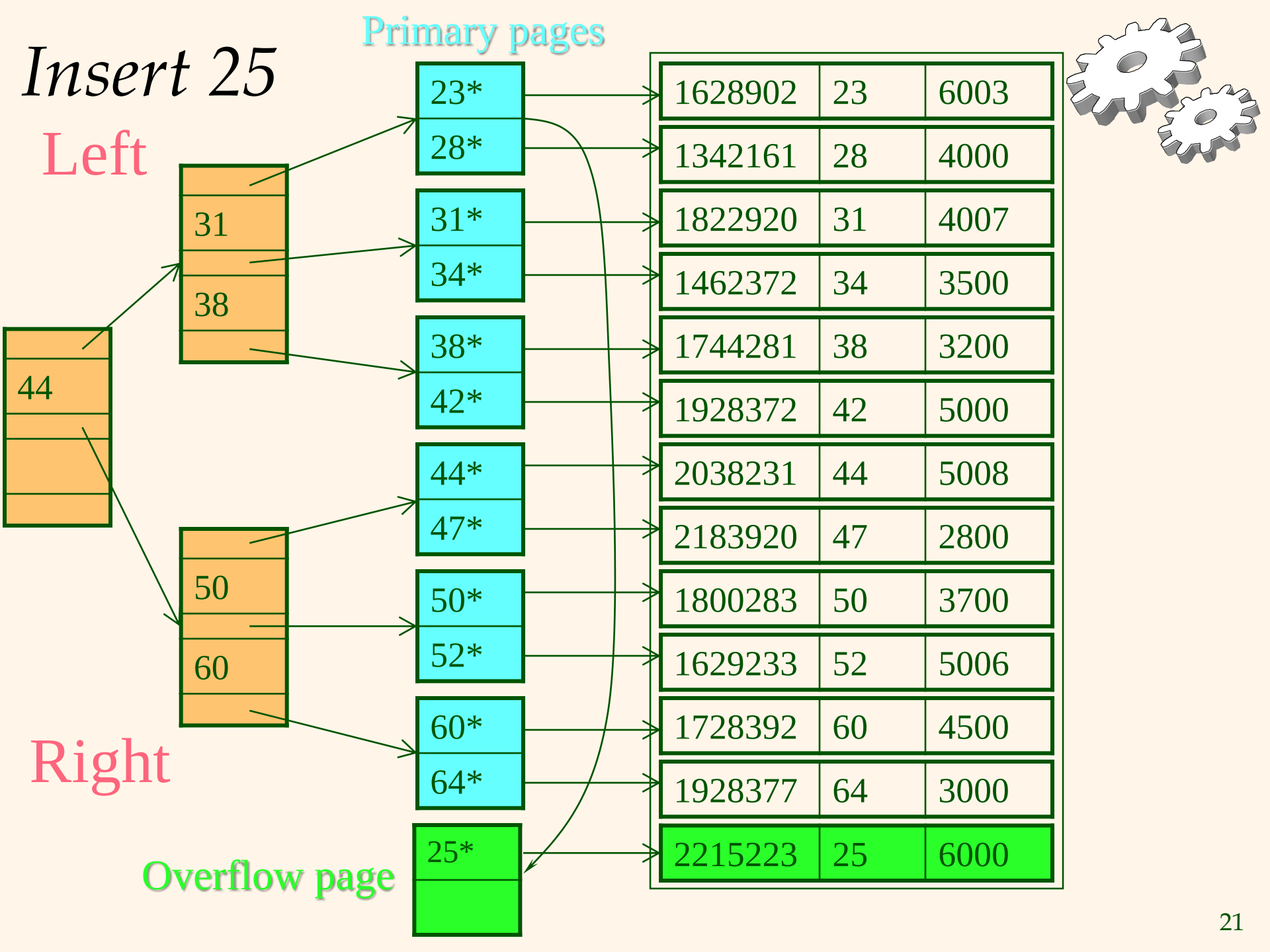


- *Insert*
  - Find the appropriate leaf page to which the data entry belongs and put it there.
  - Allocate overflow page if necessary

- Remark: Do *not* modify the index structure.
- Weakness: Can have long overflow chains ☹

 ISAM is a **static tree structure**: *inserts/deletes affect only leaf pages*.

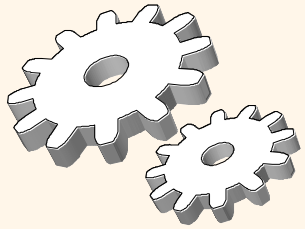
Insert 25


Primary pages
Left
Right
Overflow page
44
31
38
50
60
23*
28*
31*
34*
38*
42*
44*
47*
50*
52*
60*
64*
25*
1628902 23 6003
1342161 28 4000
1822920 31 4007
1462372 34 3500
1744281 38 3200
1928372 42 5000
2038231 44 5008
2183920 47 2800
1800283 50 3700
1629233 52 5006
1728392 60 4500
1928377 64 3000
2215223 25 6000

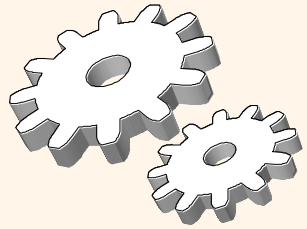
Insert 57

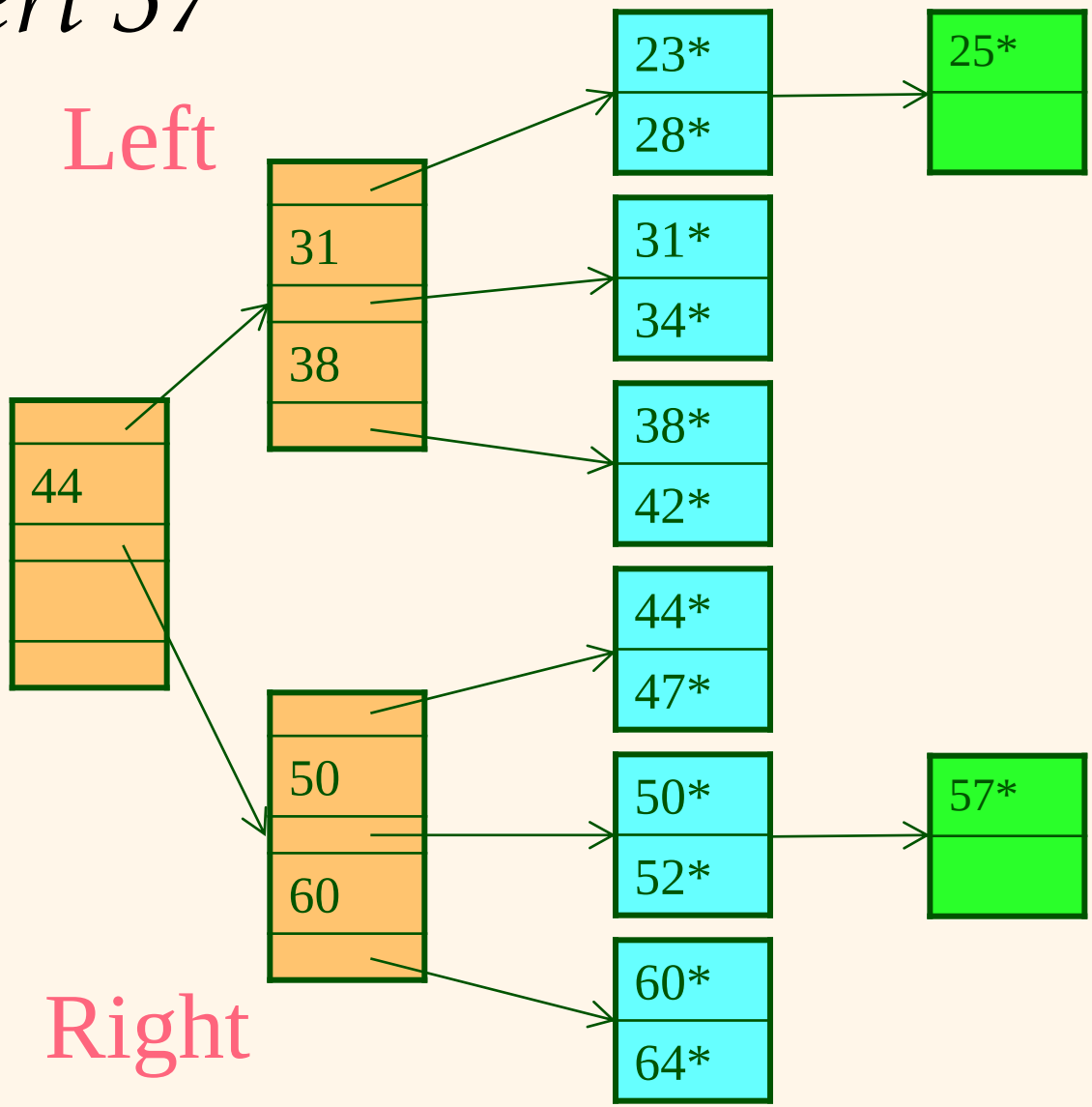
1628902	23	6003
1342161	28	4000
1822920	31	4007
1462372	34	3500
1744281	38	3200
1928372	42	5000
2038231	44	5008
2183920	47	2800
1800283	50	3700
1629233	52	5006
1728392	60	4500
1928377	64	3000
2215223	25	6000

2217283	57	4600

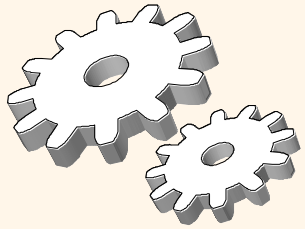


Insert 57





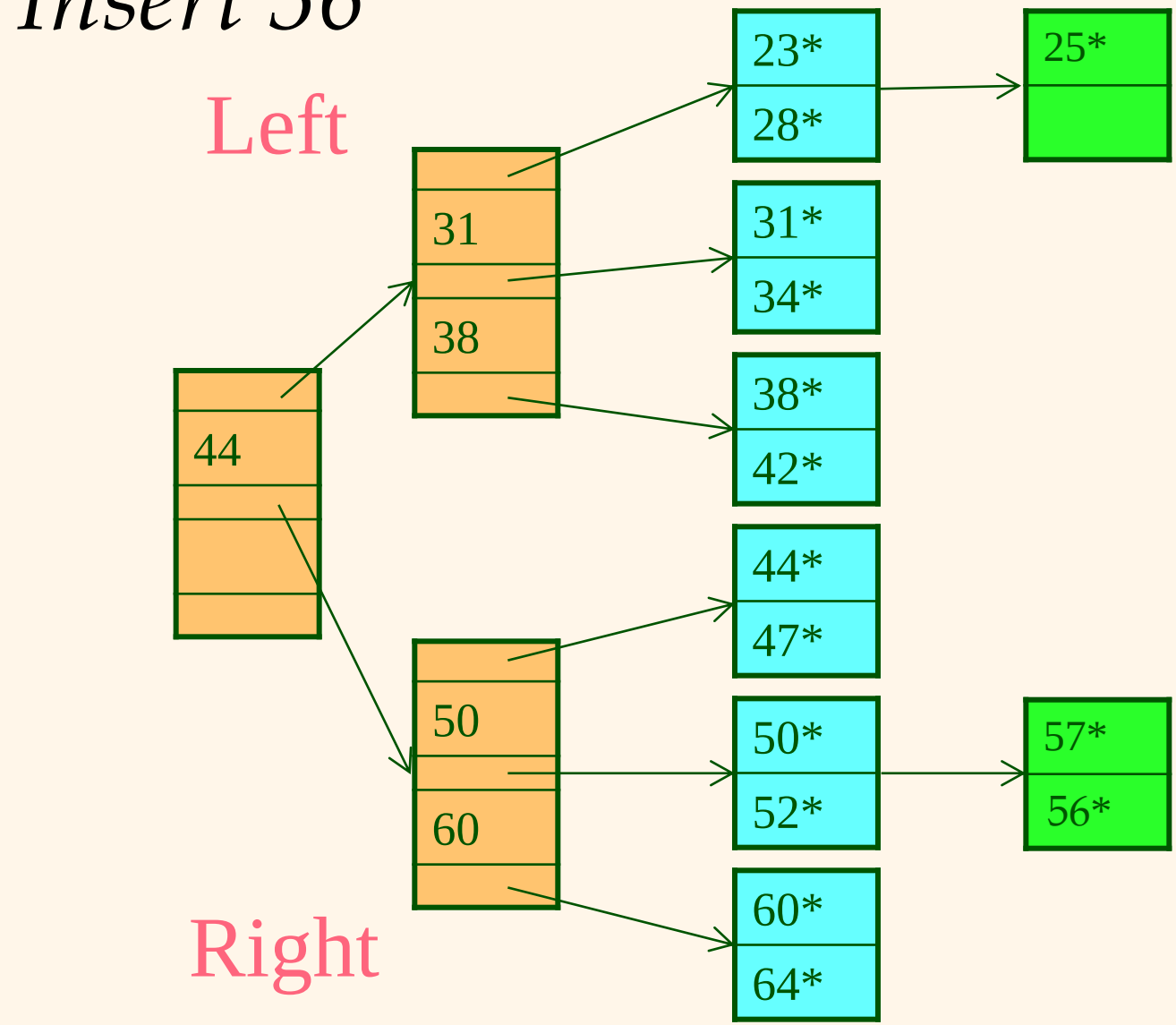
Insert 56



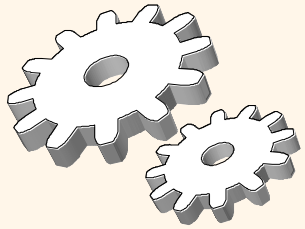
1628902	23	6003
1342161	28	4000
1822920	31	4007
1462372	34	3500
1744281	38	3200
1928372	42	5000
2038231	44	5008
2183920	47	2800
1800283	50	3700
1629233	52	5006
1728392	60	4500
1928377	64	3000
2215223	25	6000

2217283	57	4600
2264312	56	5000

Insert 56



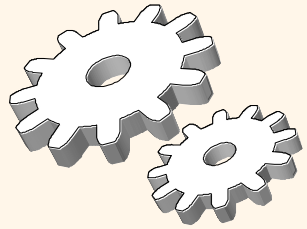
Insert 54

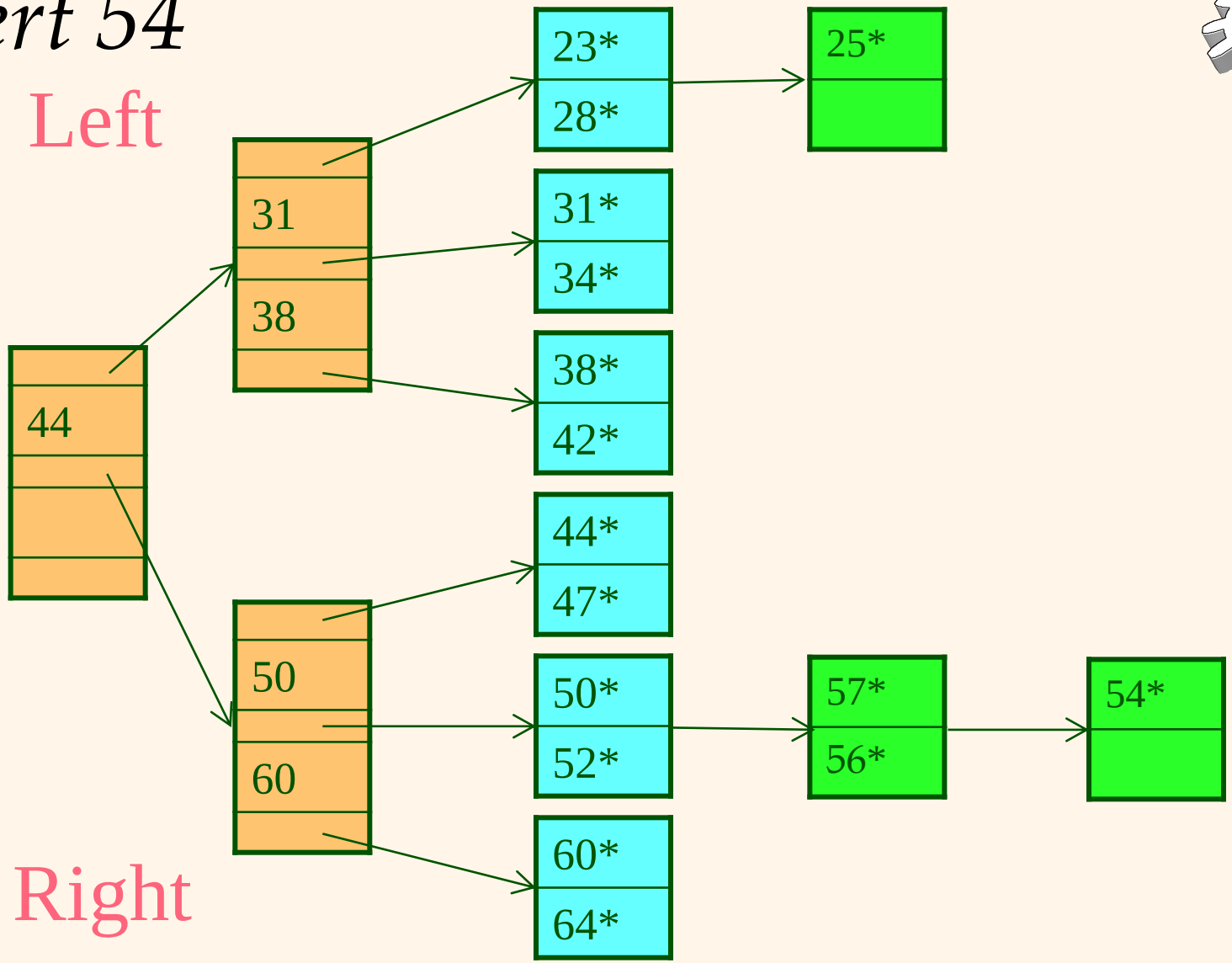


1628902	23	6003
1342161	28	4000
1822920	31	4007
1462372	34	3500
1744281	38	3200
1928372	42	5000
2038231	44	5008
2183920	47	2800
1800283	50	3700
1629233	52	5006
1728392	60	4500
1928377	64	3000
2215223	25	6000

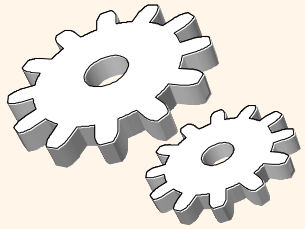
2217283	57	4600
2264312	56	5000
2245123	54	6200

Insert 54




Left
23*
28*
25*
31
31*
34*
38
38*
42*
44
44*
47*
50
50*
52*
57*
56*
54*
60
60*
64*
Right

ISAM: Delete

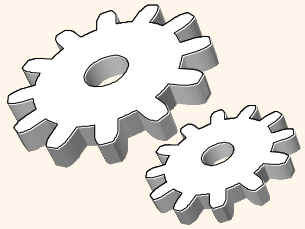


- *Delete*
  - Find the leaf page where the data entry is located and remove it
  - De-allocate the overflow page if it's empty

- Remark: No need to modify the entries of internal nodes.
- Weakness: Can result in many holes (i.e., empty space) in the index file/data file ☹

 ISAM is a **static tree structure**: *inserts/deletes affect only leaf pages.*

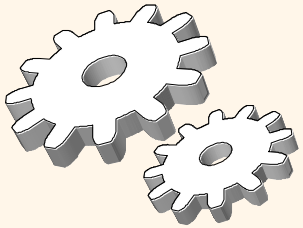
Delete 25

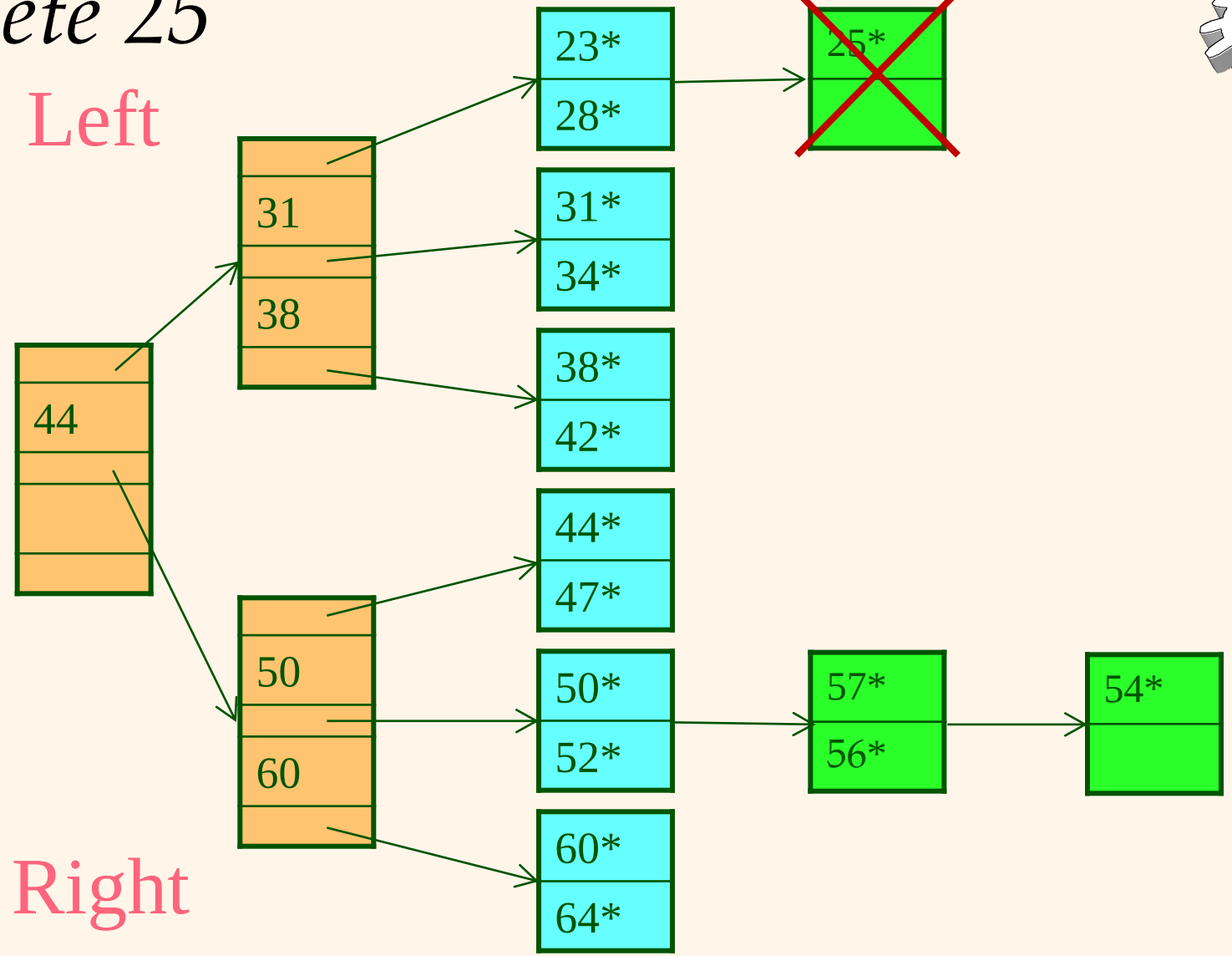


1628902	23	6003
1342161	28	4000
1822920	31	4007
1462372	34	3500
1744281	38	3200
1928372	42	5000
2038231	44	5008
2183920	47	2800
1800283	50	3700
1629233	52	5006
1728392	60	4500
1928377	64	3000

2217283	57	4600
2264312	56	5000
2245123	54	6200

Delete 25

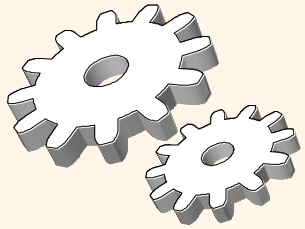



Left
23*
28*
25*
31
31*
34*
38
38*
42*
44
44*
47*
50
50*
52*
57*
56*
54*
60
60*
64*
Right

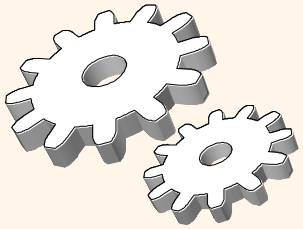
Delete 60

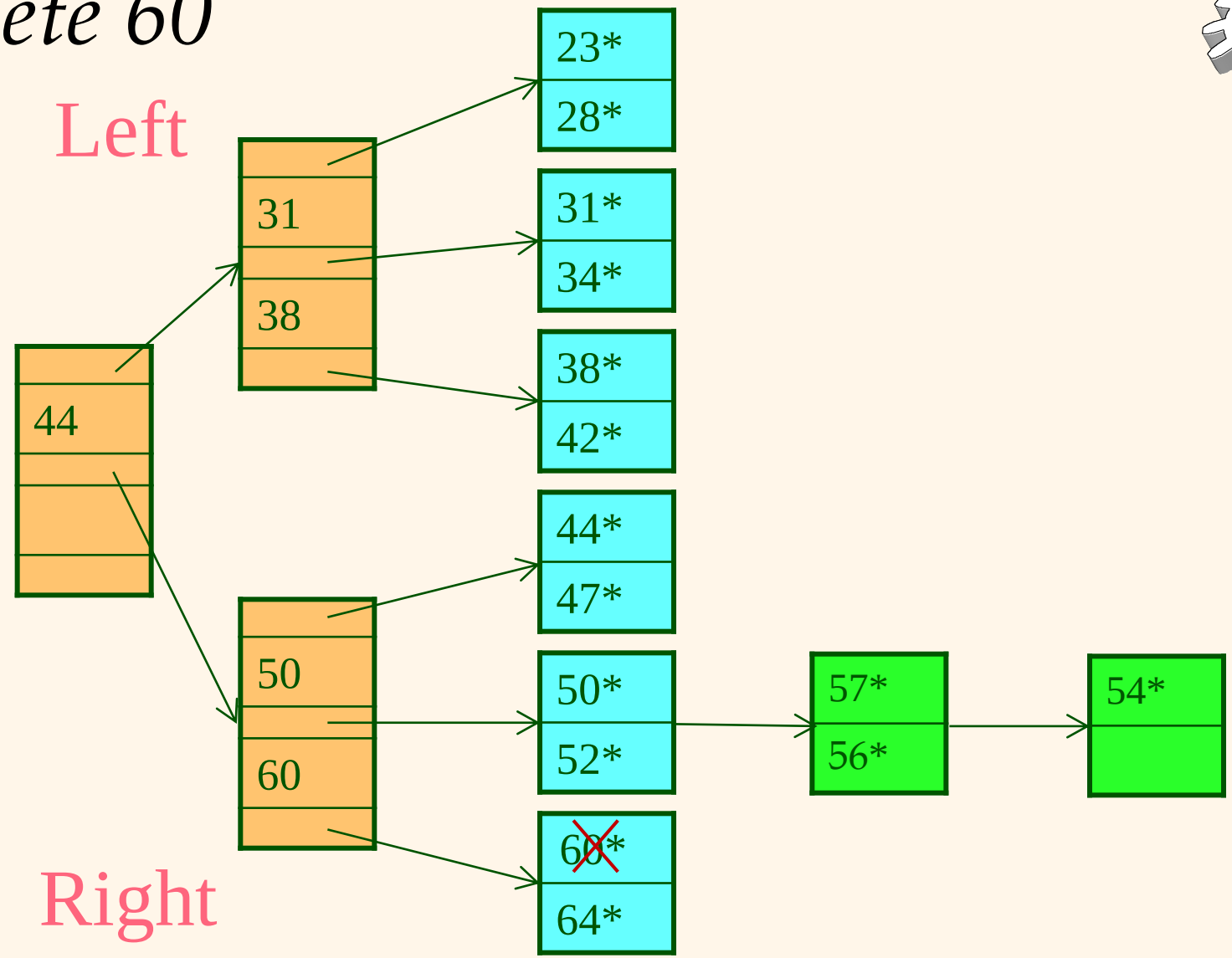
1628902	23	6003
1342161	28	4000
1822920	31	4007
1462372	34	3500
1744281	38	3200
1928372	42	5000
2038231	44	5008
2183920	47	2800
1800283	50	3700
1629233	52	5006
1928377	64	3000

2217283	57	4600
2264312	56	5000
2245123	54	6200



Delete 60

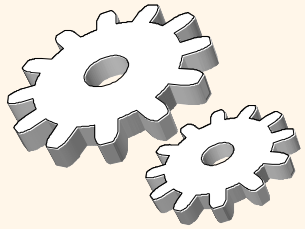




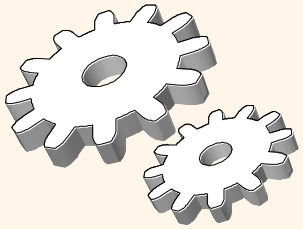
Delete 54

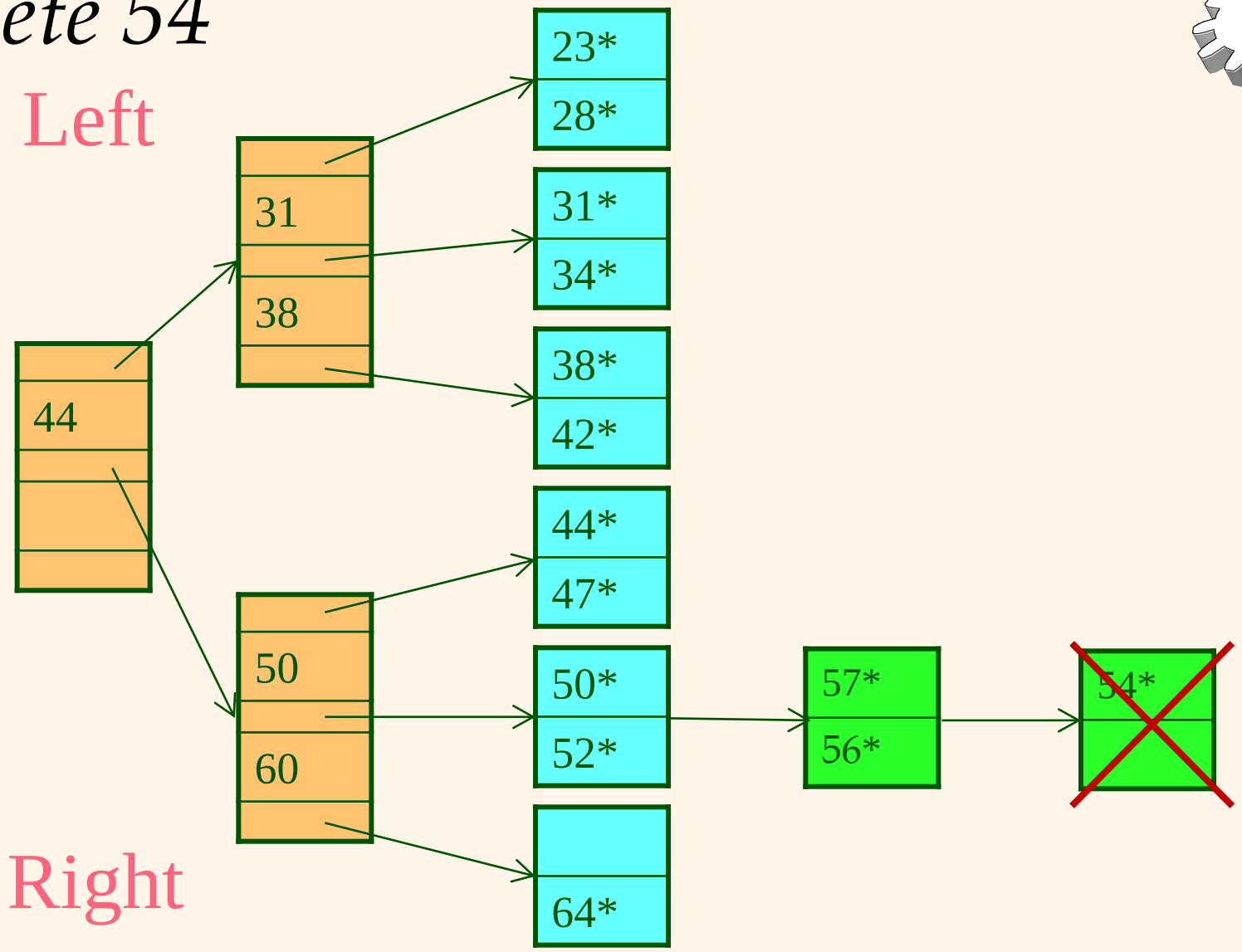
1628902	23	6003
1342161	28	4000
1822920	31	4007
1462372	34	3500
1744281	38	3200
1928372	42	5000
2038231	44	5008
2183920	47	2800
1800283	50	3700
1629233	52	5006
1928377	64	3000

2217283	57	4600
2264312	56	5000



Delete 54

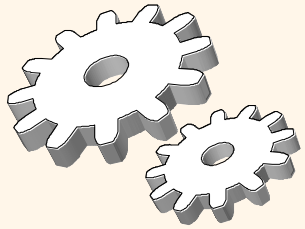




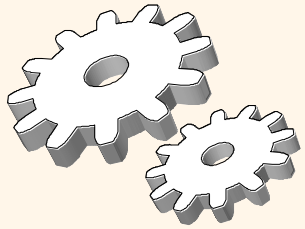
Delete 42

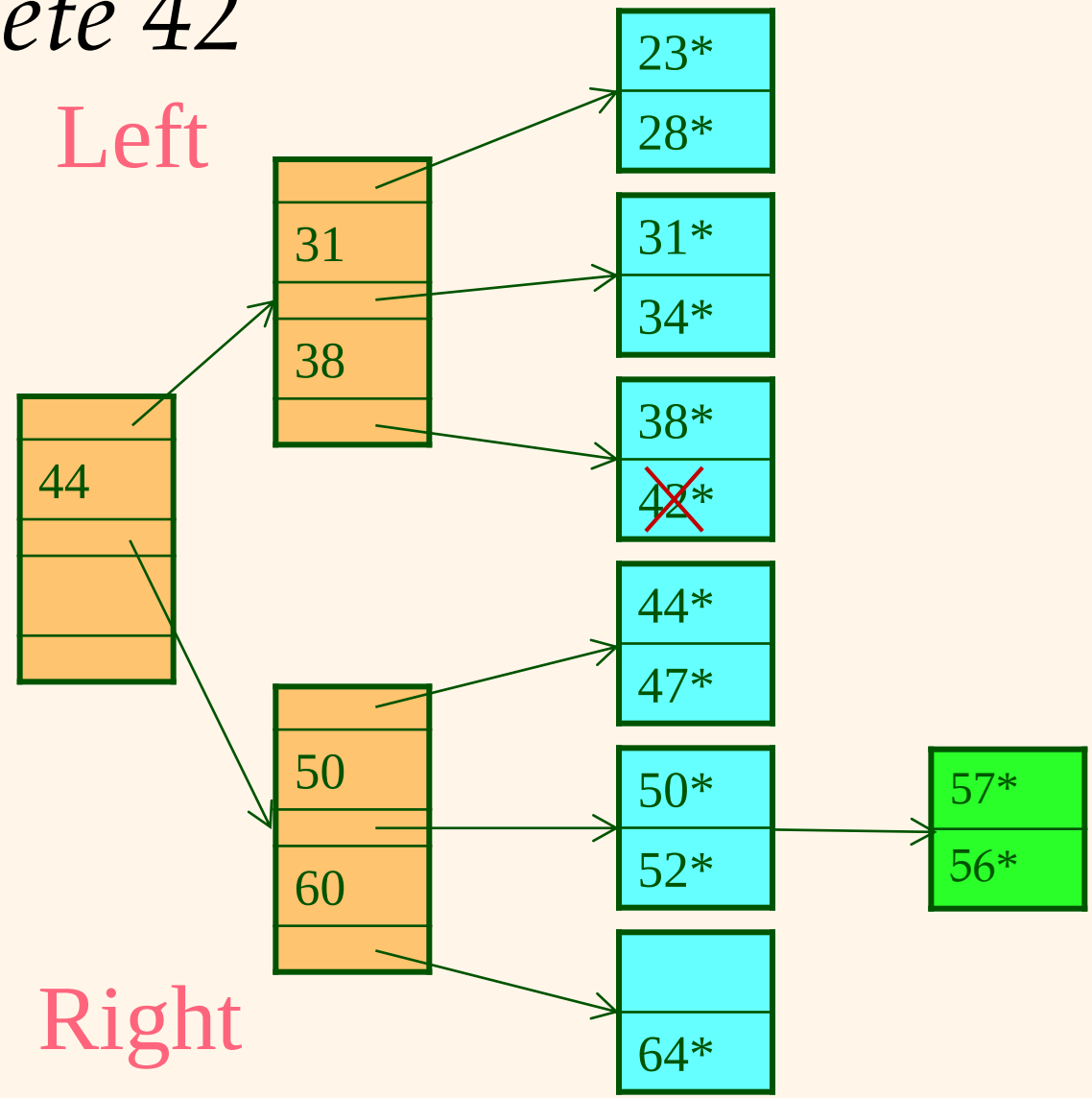
1628902	23	6003
1342161	28	4000
1822920	31	4007
1462372	34	3500
1744281	38	3200
2038231	44	5008
2183920	47	2800
1800283	50	3700
1629233	52	5006
1928377	64	3000

2217283	57	4600
2264312	56	5000



Delete 42

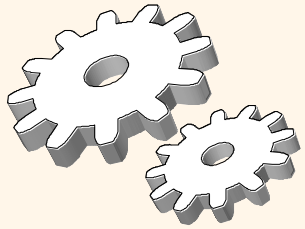




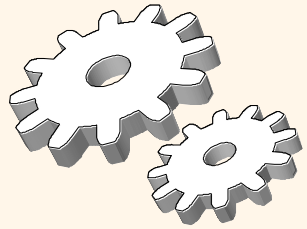
Delete 34

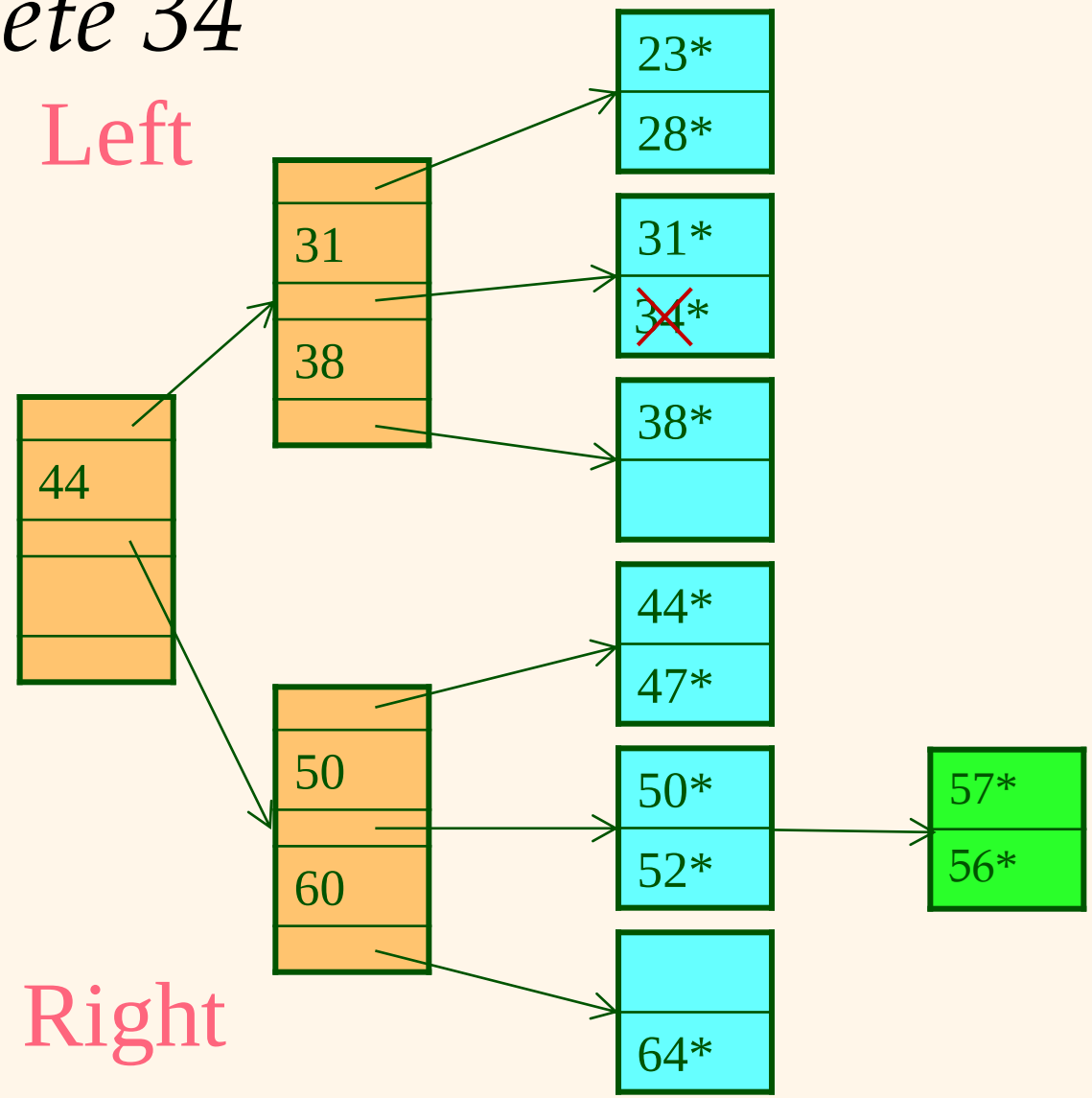
1628902	23	6003
1342161	28	4000
1822920	31	4007
1744281	38	3200
2038231	44	5008
2183920	47	2800
1800283	50	3700
1629233	52	5006
1928377	64	3000

2217283	57	4600
2264312	56	5000

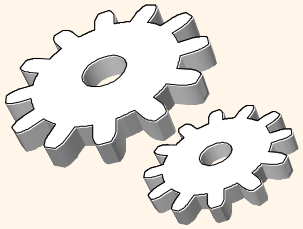


Delete 34



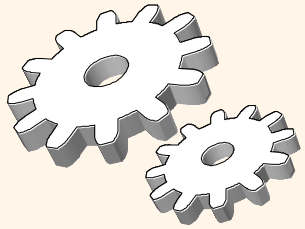


A Worse Case



degrades to a sequential scan of the index

Cost Analysis of ISAM



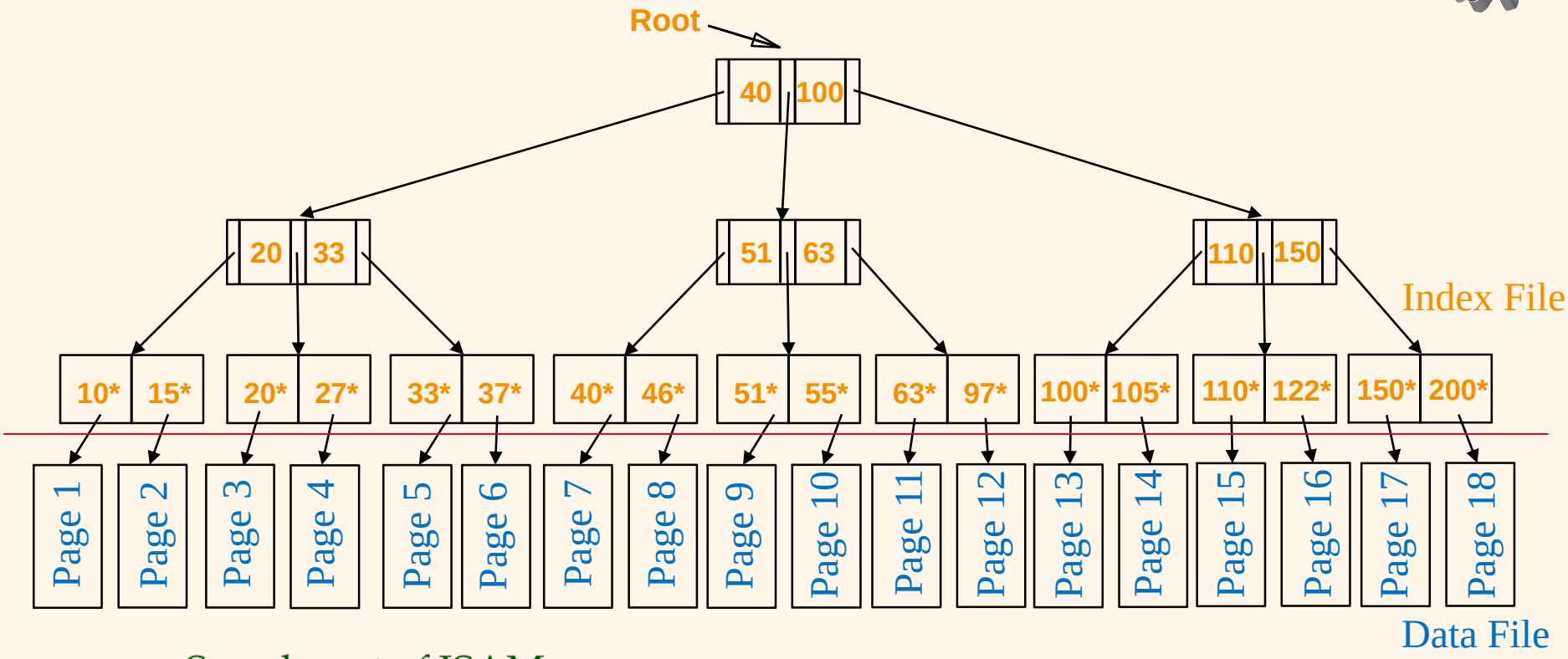
- Search cost of ISAM
  - # of disk IOs = # of levels of tree = $1+\lceil \log_F N \rceil$,
    where F = fan-out = # of children per non-leaf page
    N= # of primary leaf pages

(excluding reading records and assuming no overflow pages)

- Cost of binary search
  - $\lceil \log_2 B \rceil$, where B = # of data pages

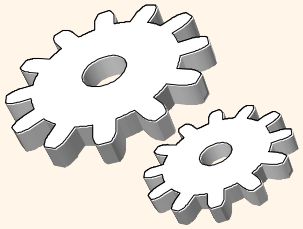
(excluding reading remaining records)

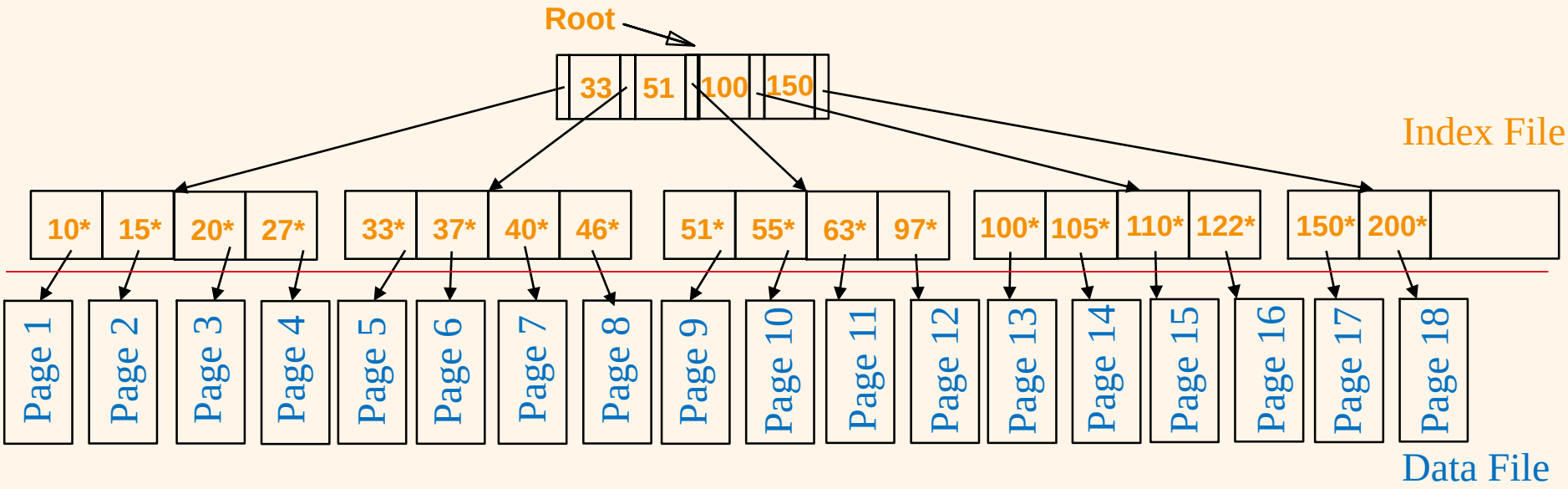
Cost Analysis of ISAM: Example



- Search cost of ISAM
  - F = fan-out = # of children per non-leaf page = 3
  - N= # of primary leaf pages = 9
  - # of disk IOs = # of levels of tree = $1+\lceil \log_F N \rceil = 3$ I/Os
- Cost of binary search
  - B = # of data pages = 18, # of disk IOs = $\lceil \log_2 B \rceil = 5$ I/Os

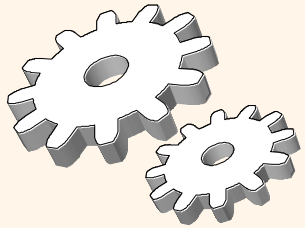
Cost Analysis of ISAM: Example





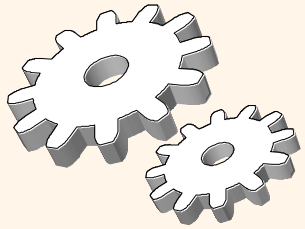
- Search cost of ISAM
  - F = fan-out = # of children per non-leaf page = 5
  - N= # of primary leaf pages = 5
  - # of disk IOs = # of levels of tree = $1+\lceil \log_F N \rceil$ = 2 I/Os
- Cost of binary search
  - B = # of data pages = 18, # of disk IOs = $\lceil \log_2 B \rceil$ = 5 I/Os

Question 1



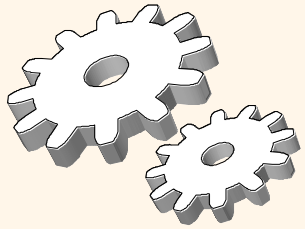
1,000,000 records, 10 data records per page, 100 data entries per (leaf) page, 100 index entries per (non-leaf) page。 Estimate the search I/O cost of ISAM and the I/O cost of binary search.

Summary



- Tree-structured indexes are ideal for range searches, also good for equality searches
- ISAM is a static structure
  - Only leaf pages modified
  - Overflow pages needed
  - Creating holes (empty spaces)

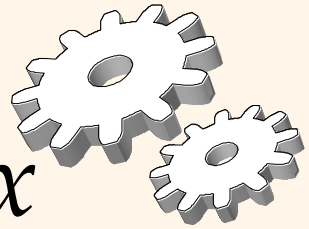
Tree-Structured Indexes



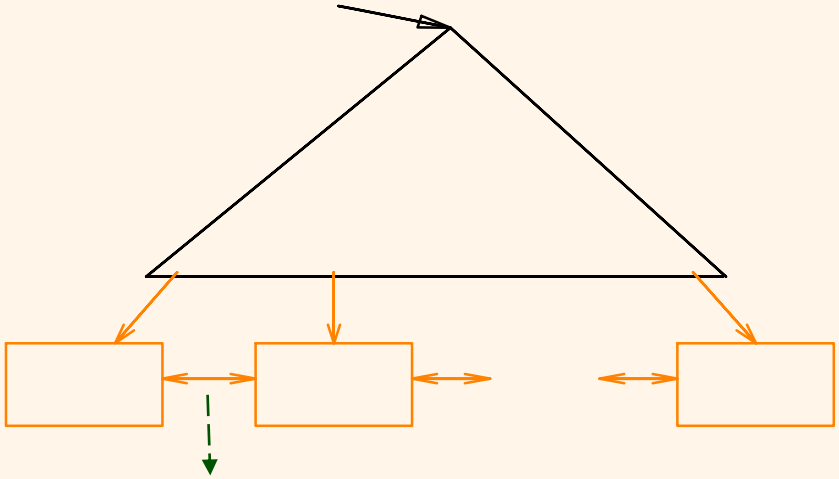
- Efficient support for *range search, insertion* and *deletion*

- *ISAM*
  - Indexed Sequential Access Method
  - Static index structure
- ***B+ tree***
  - **Dynamic structure that adjusts gracefully under insertions and deletions**

B+ Tree: Most Widely Used Index



- Dynamic index structure
  - Height-balanced tree
- Height = length of the path from root to any leaf
  - Typically 3 or 4 because of high fan-out (#children per index node (i.e., non-leaf page) in the index)



Index Nodes/Pages

- Index entries

- Direct search

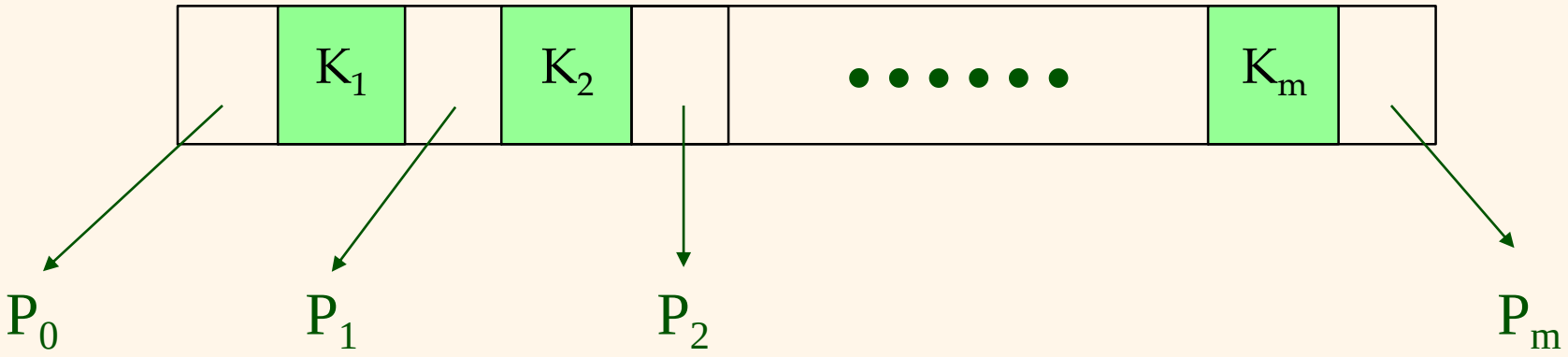
Leaf Nodes/Pages

- Data entries

- Doubly linked list

- "Sequence set"

Structure of an Index Node

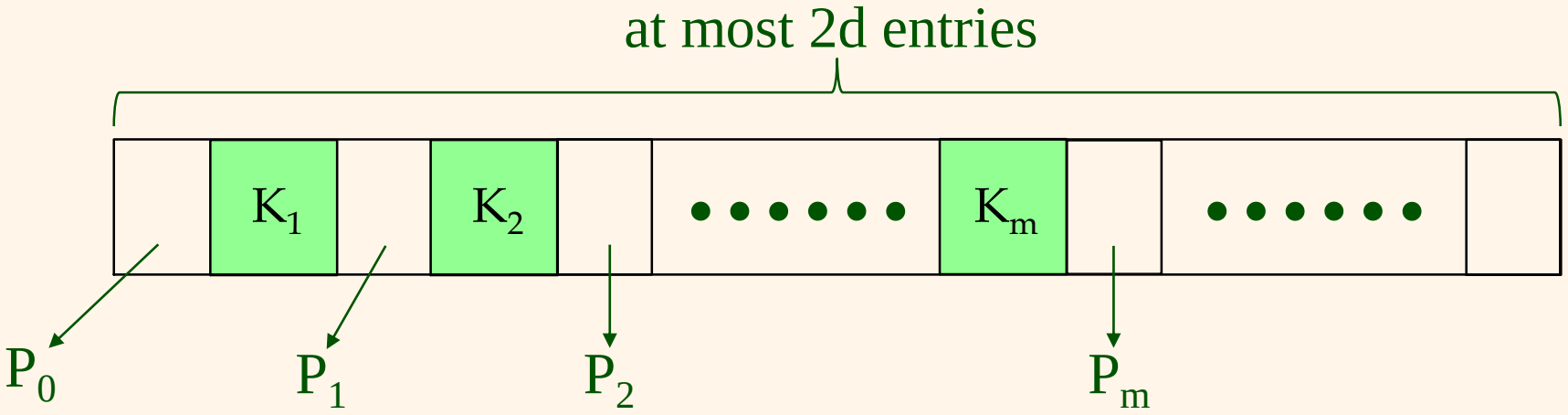


- If the node contains ***m*** entries (keys), there are m + 1 pointers to its child nodes.
- Pointer P_i ($1 \le i < m$) points to its subtree where the key values K fulfills the following condition:

$$K_i \le K < K_{i+1}$$

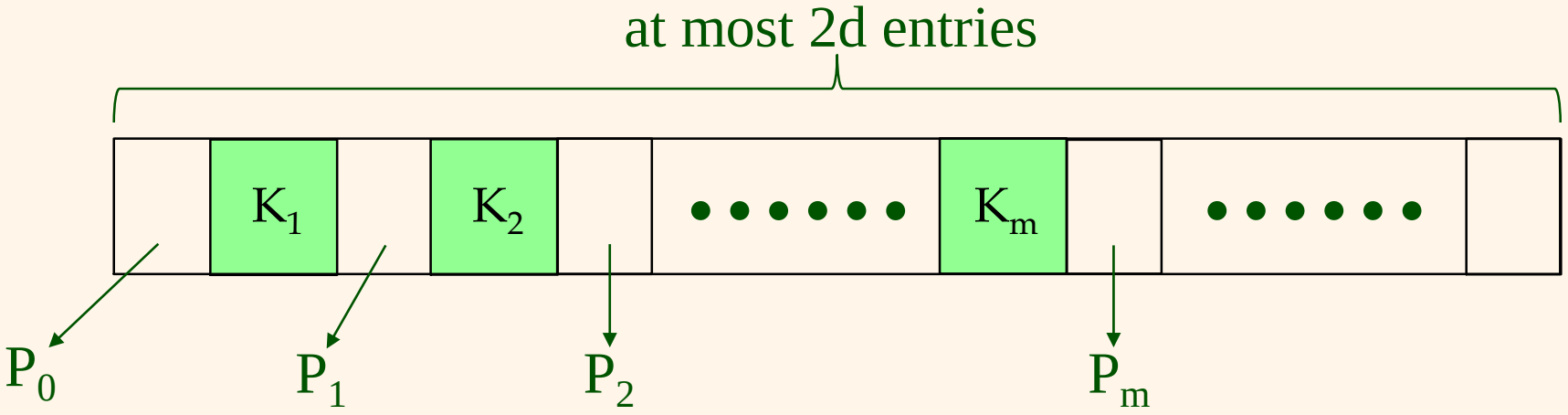
- Pointer P_0 points to its subtree with $K < K_1$.
- Point P_m points to its subtree with $K \ge K_m$.

Order of an Index Node



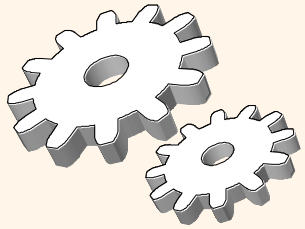
- Order is a parameter of a B+ tree, denoted by *d*, which can be used to measure the capacity of a node.
  - The maximum number of entries (keys) of a node is *2d*.
  - Let *m* be the current number of entries in a node. *m* must satisfy $d \leq m \leq 2d$.

Fanout of an Index Node



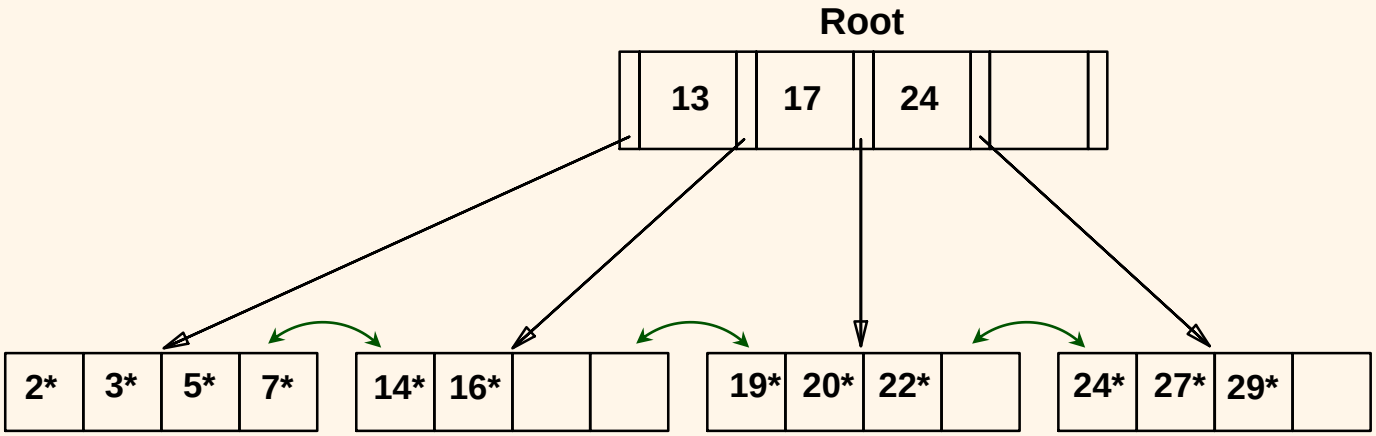
- Fanout of a node is the current number of pointers of the node
  - *m + 1* in this example
  - Maximum fanout of a node = *2d + 1*

Example B+ Tree

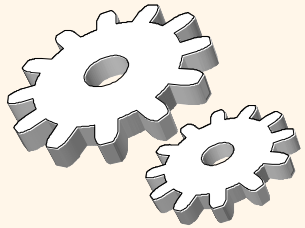


The maximum number of entries in an index node is 4

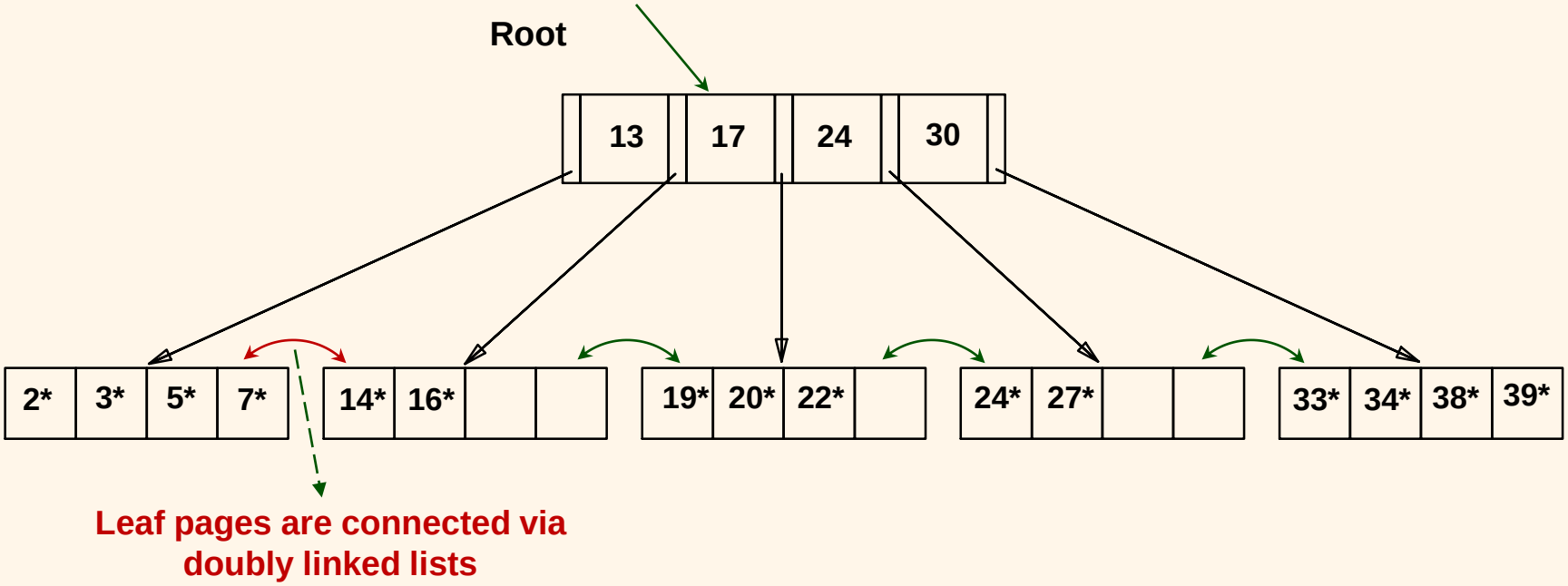
- What is the order of the tree? d=2
- What is the fanout of the root node? F=4
- What is the maximum fanout of the root node? 2d+1=5



Example: How to Search in a B+ tree?

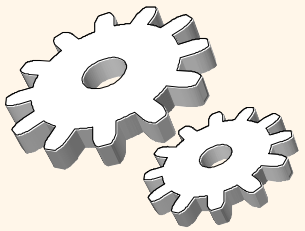


- Search begins at the root, and key comparisons direct it to a leaf (as in ISAM).
  - No need to handle overflow pages
- e.g., find 5*, 15*, all data entries 20≤k<27 ...



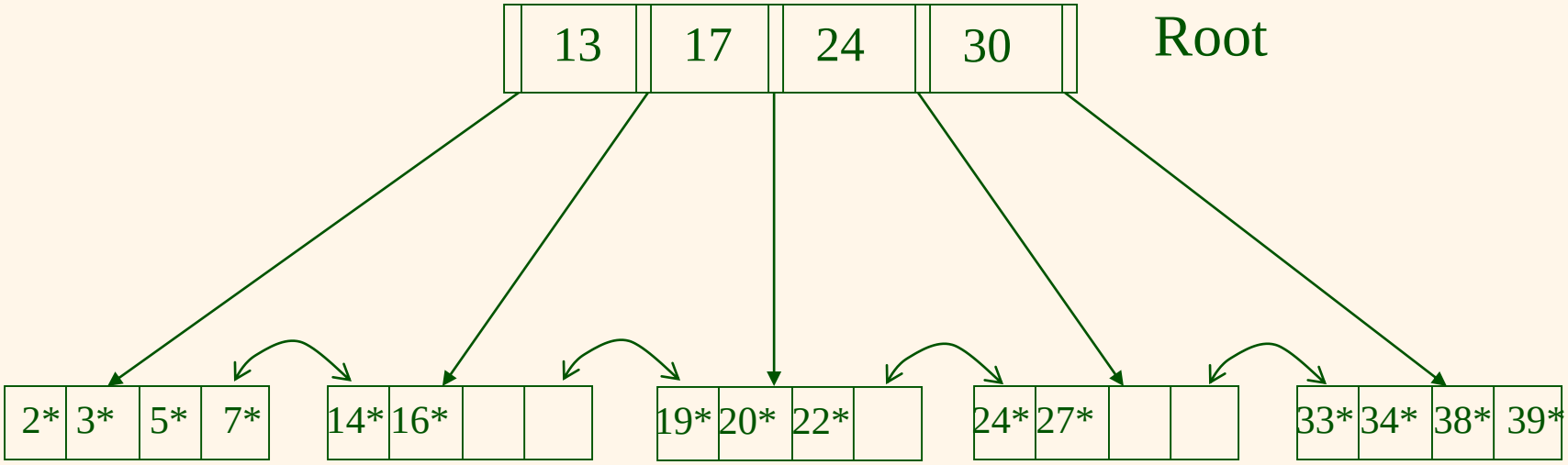
Inserting a Data Entry into a B+ Tree

- Find the correct leaf *L*
- Put data entry onto *L*
  - If *L* has enough space, *done*!
  - Else, *split L into L and a new node L2*
    - Redistribute entries evenly, **copy up the** middle key
    - Insert index entry pointing to *L2* into parent of *L*
- This can happen recursively
  - To split the index node, redistribute entries evenly, but **push up the** middle key (different from leaf splits)
- Splits "grow" tree
  - Root split increases height

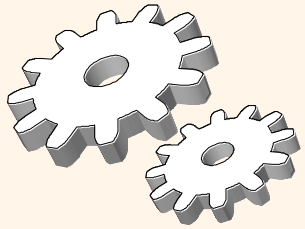


Inserting 29 (1)*

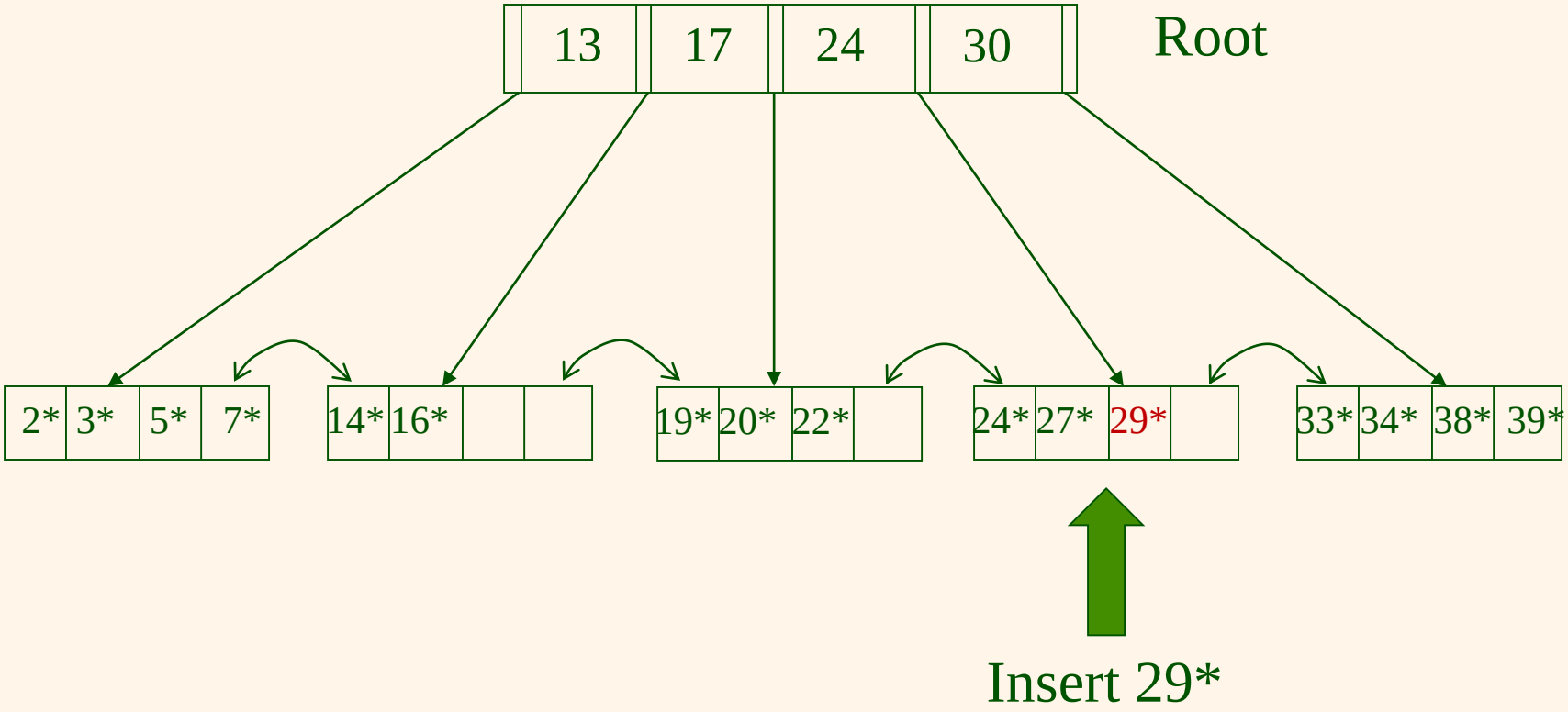
Let the order of the B+ tree be 2 (i.e., the maximum number of entries is 4).

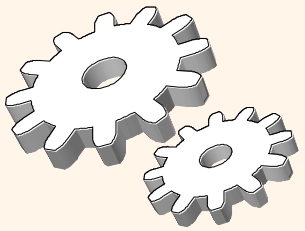


Inserting 29 (2)*



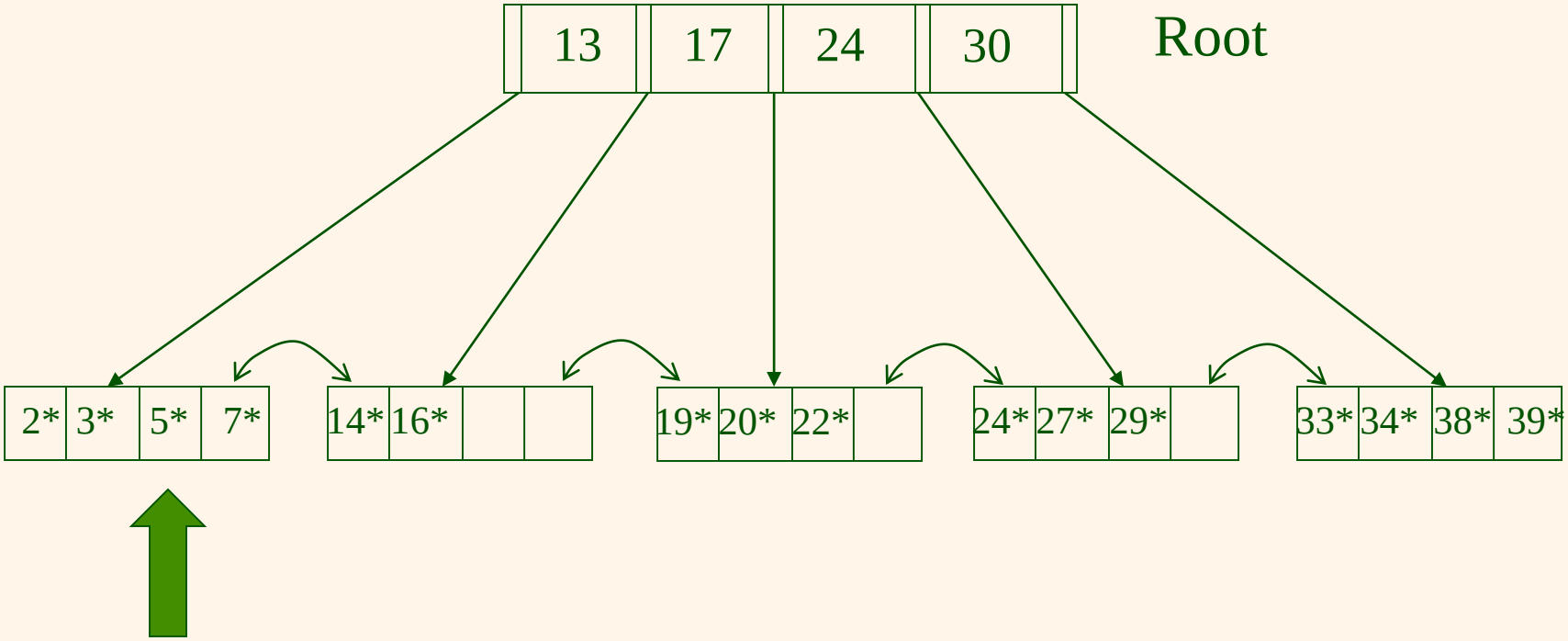
Let the order of the B+ tree be 2 (i.e., the maximum number of entries is 4).



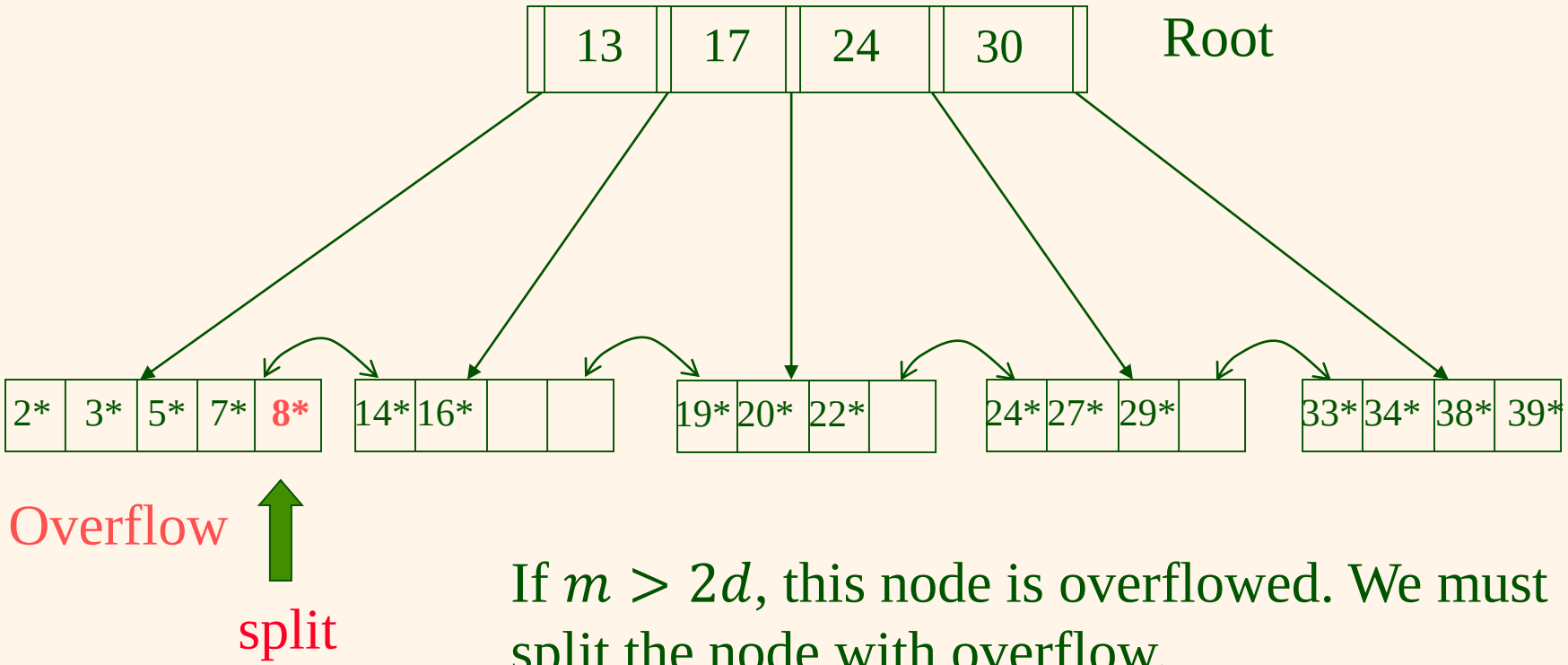


Inserting 8 (1)*

Let the order of the B+ tree be 2 (i.e., the maximum number of entries is 4).

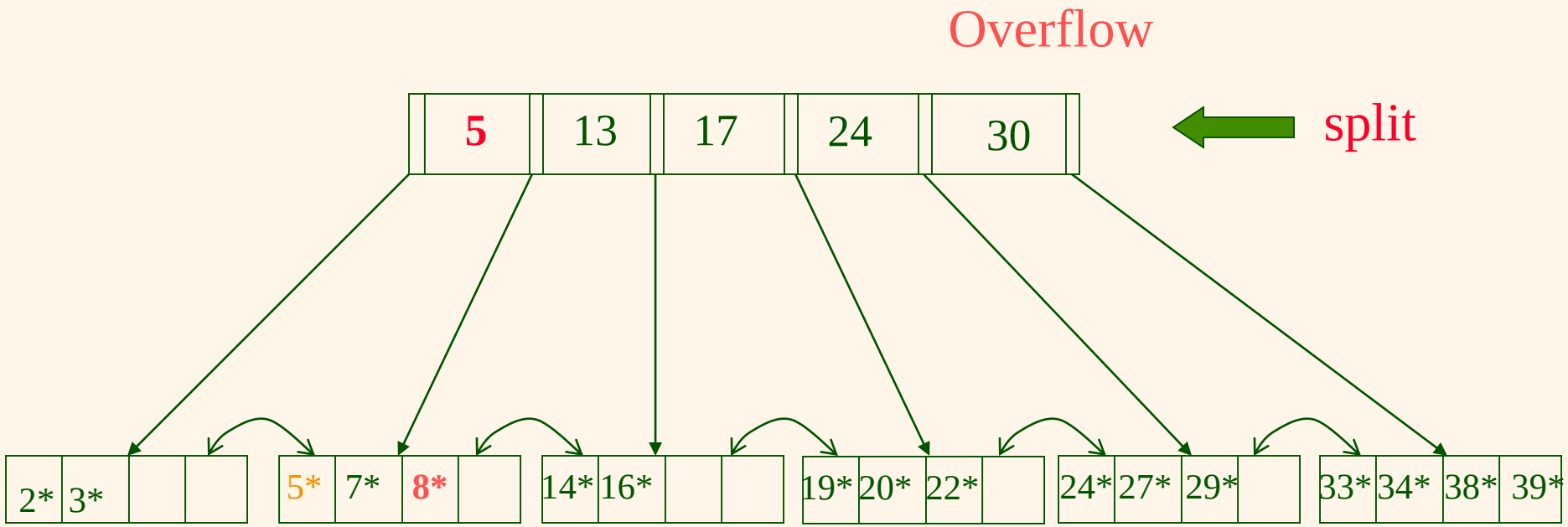


Inserting 8 (2)*



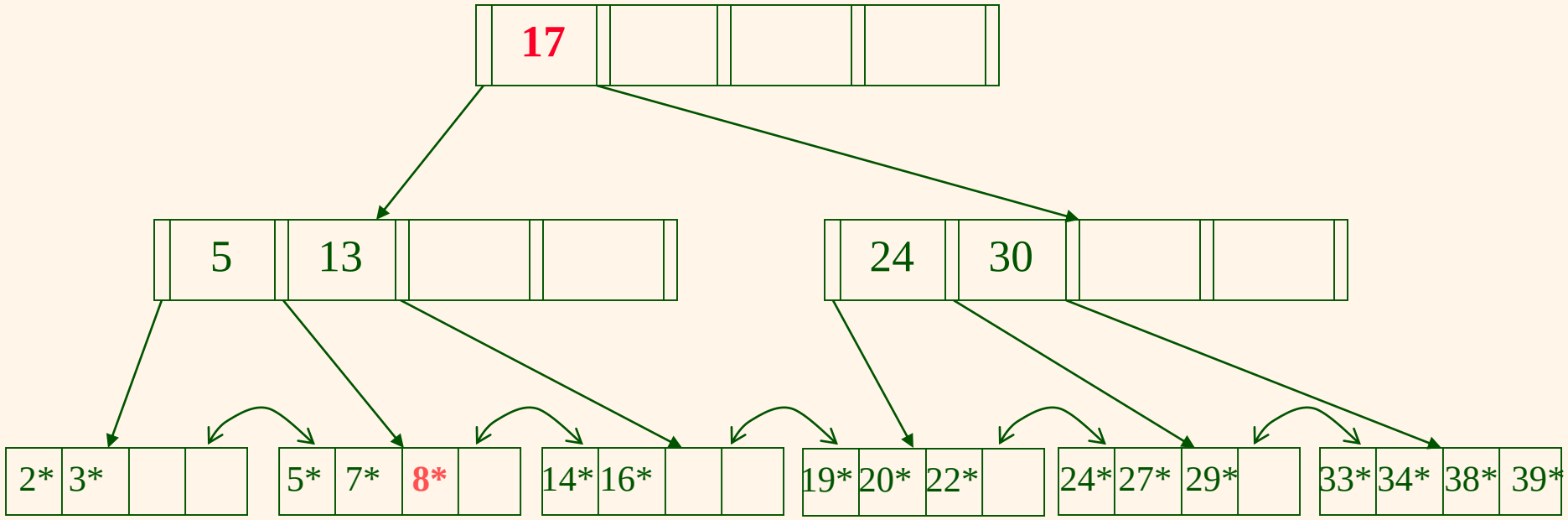
If $m > 2d$, this node is overflowed. We must split the node with overflow.

Inserting 8 (3)*



Key 5 (middle key in leaf node) is "copied up" into parent node.

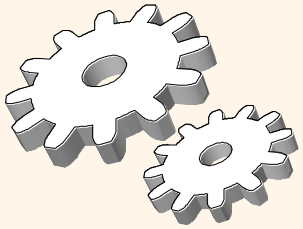
Inserting 8 (4)*

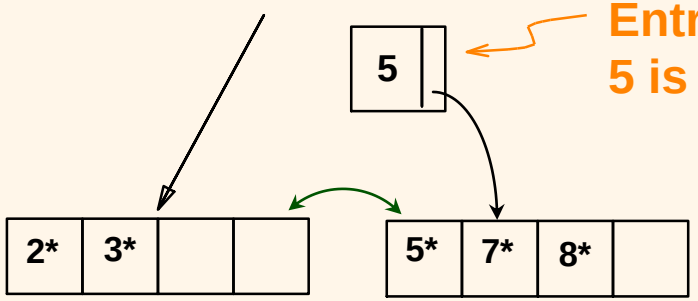


Key 17 (middle key in index node) is "pushed up" into parent node.

- Root has been split, leading to height increase

"Copy Up" vs. "Push Up"

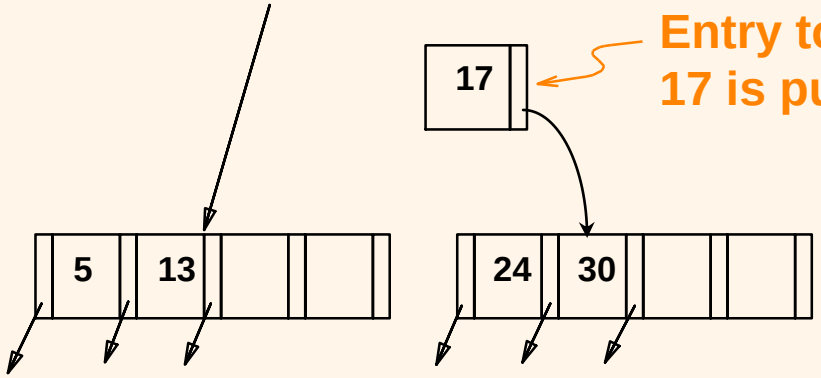




Entry to be inserted in parent node:
5 is copied up and still appear in leaf

Leaf node split

"Copy Up"

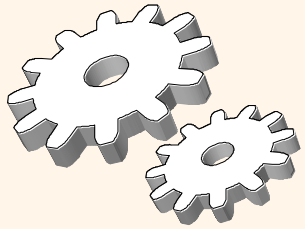


Entry to be inserted in parent node:
17 is pushed up and appears once in index

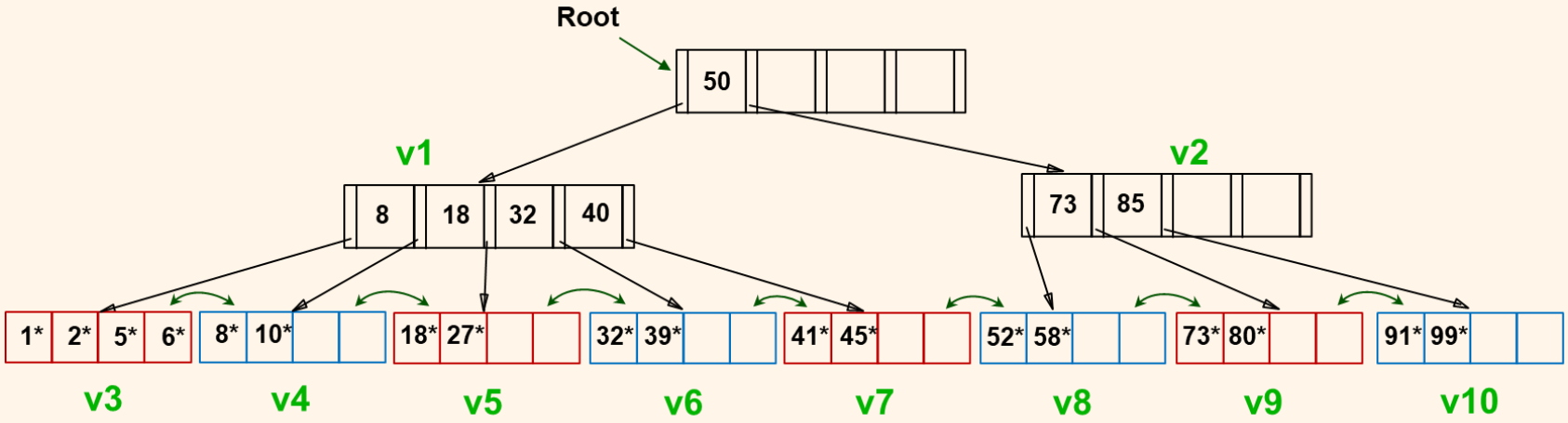
Index node split

"Push Up"

A B+ Tree Index

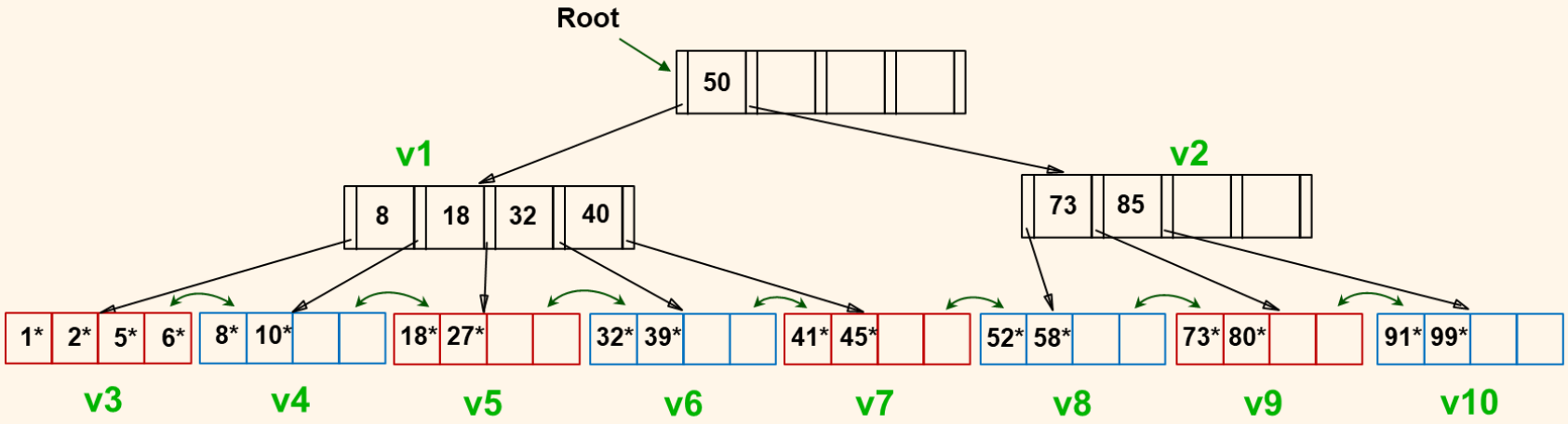


- Given a B+ tree index of order d=2 shown in the below figure, please answer Questions 2-5.



Order d = 2 means that each node at least has 2 elements and at most has 4 elements

Question 2 & 3

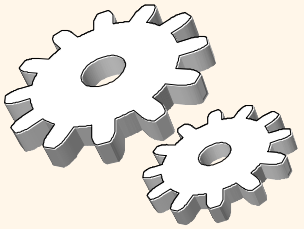


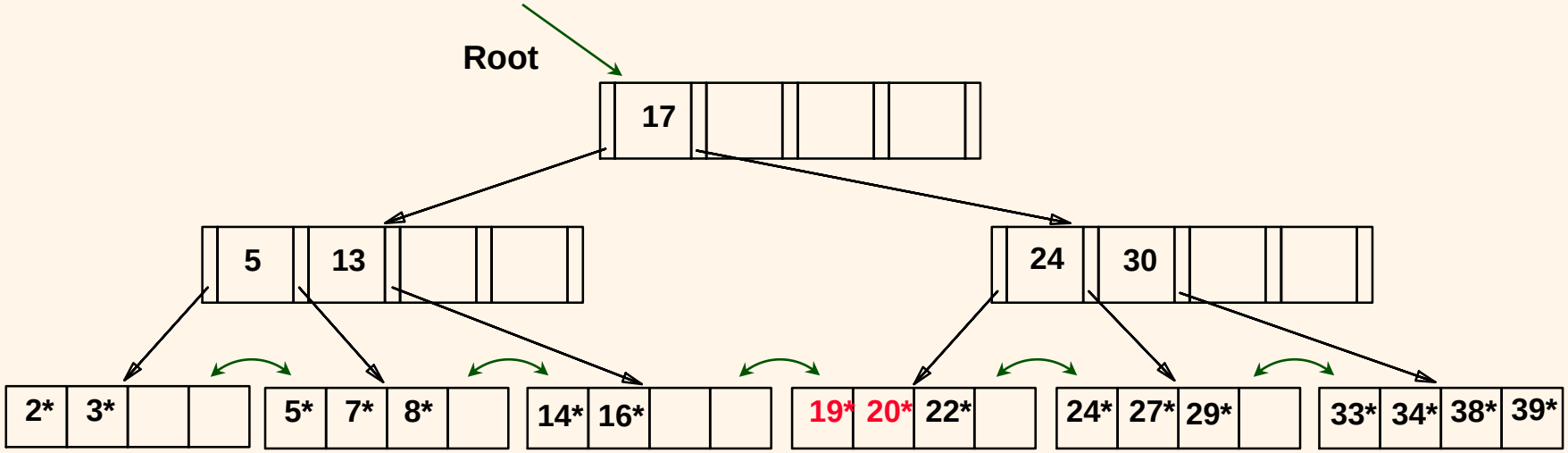
- Show the tree that would result from inserting a data entry with key 9 into this tree
- Show the B+ tree that would result from inserting a data entry with key 3 into the *original tree*.

Deleting a Data Entry from a B+ Tree

- Start at the root, find leaf *L* where entry exists
- Remove the entry
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to re-distribute by borrowing from a *sibling* *(adjacent node with the same parent as L)*
    - If re-distribution fails, *merge* *L* and sibling
    - If merge occurred, must delete the entry (pointing to *L* or sibling) from the parent of *L*
- Merge could propagate to the root, decreasing the height of tree

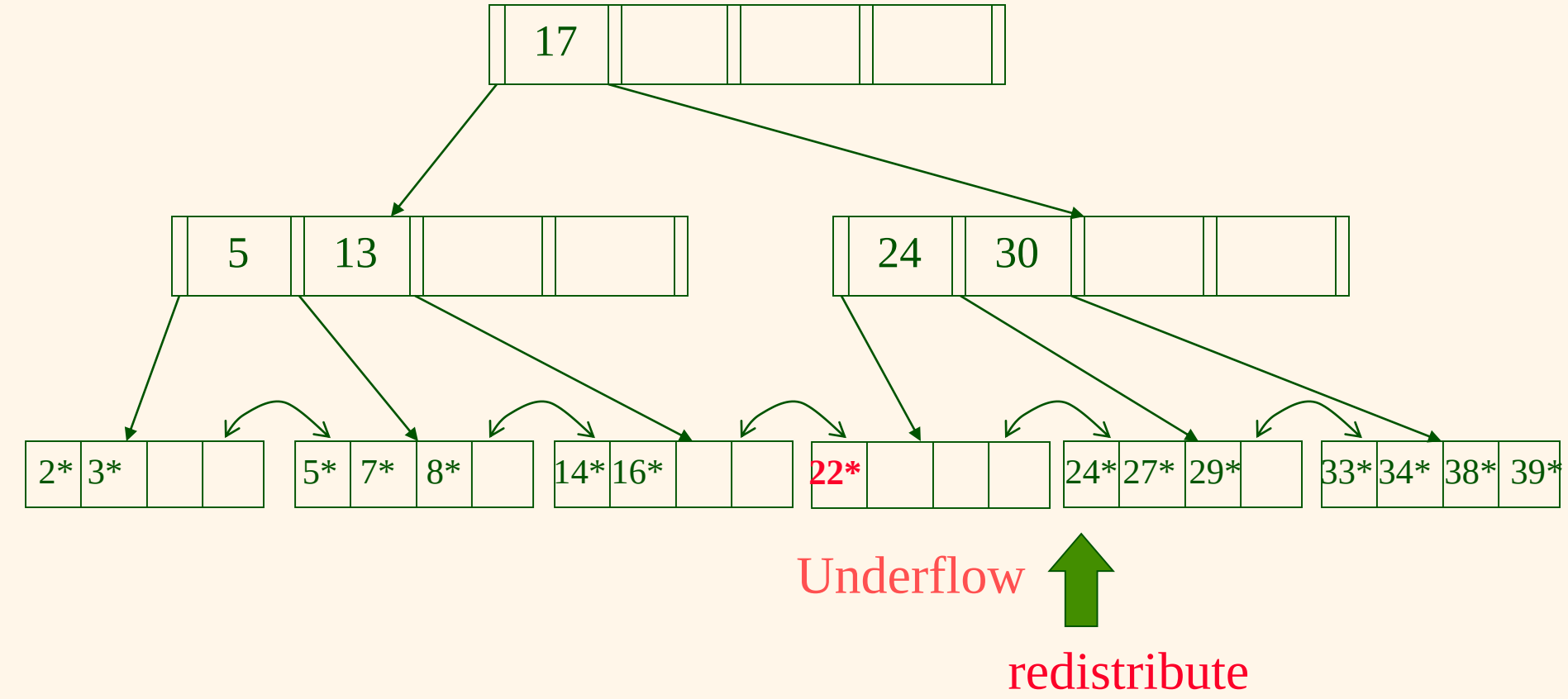
Want to Delete 19 and 20* ...*





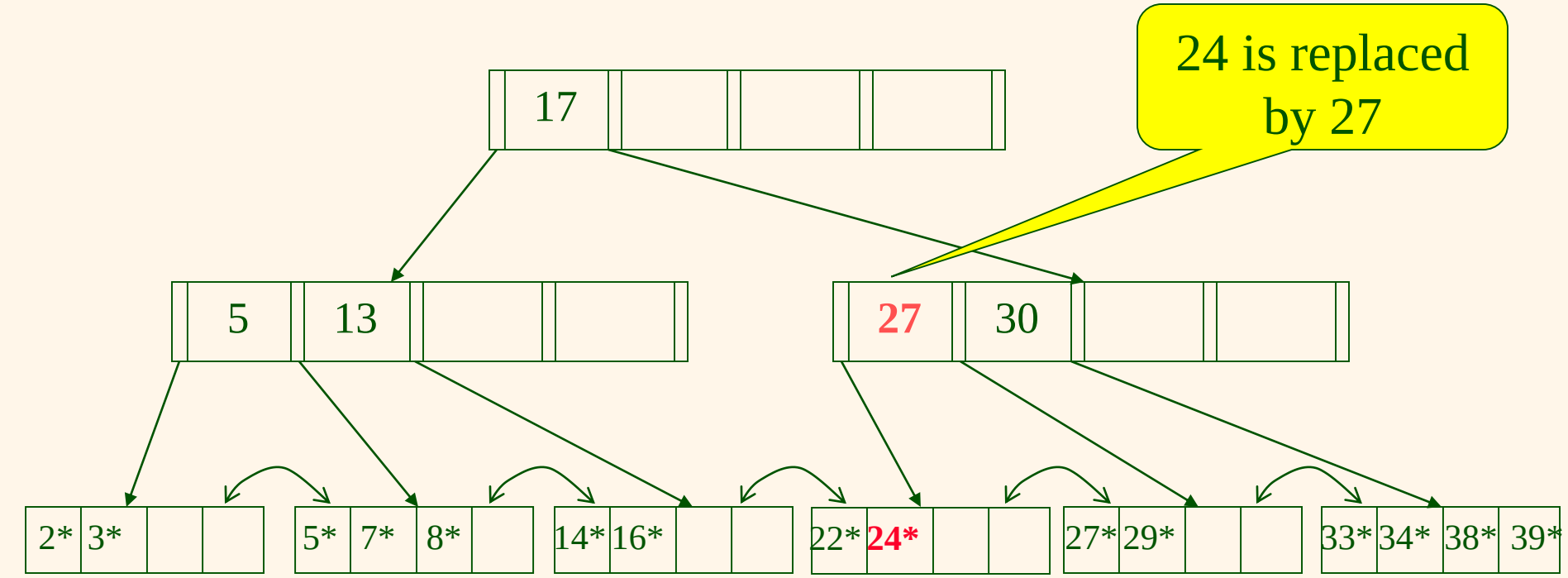
- Deleting 19* is easy
- Deleting 20* is done with re-distribution
  - middle key is *copied up*

Deleting 19, 20**



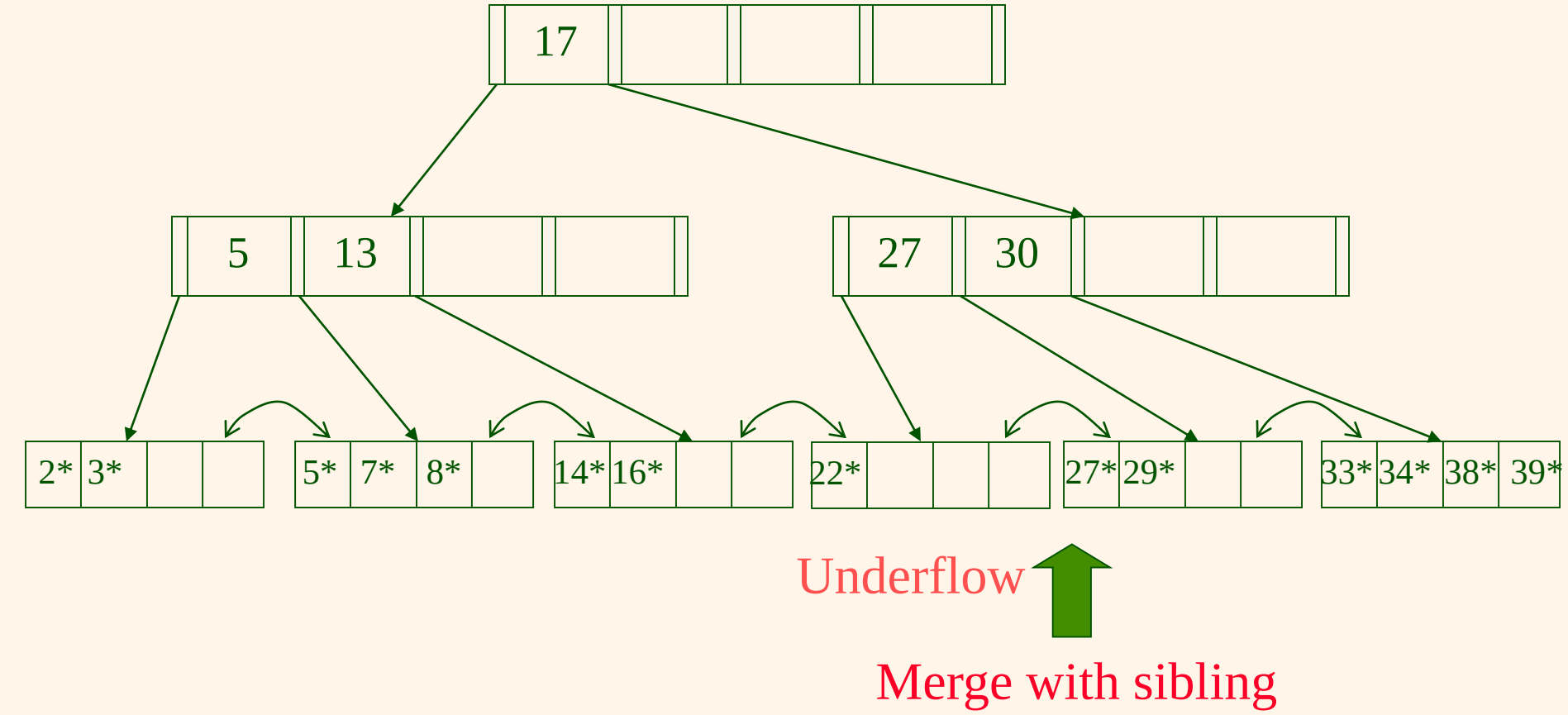
If $m < d$, this node is underflowed.

After Deleting 19, 20**

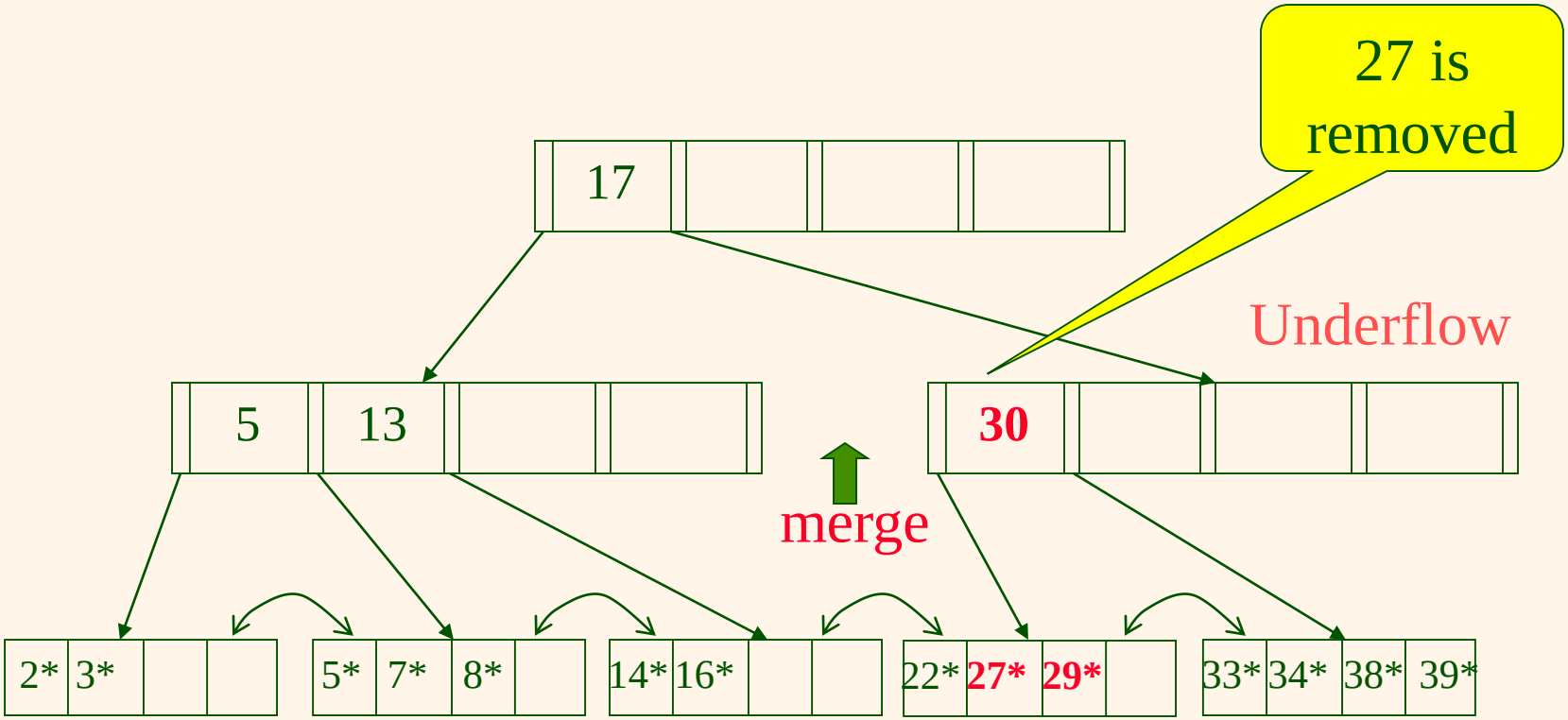


Now want to delete 24*

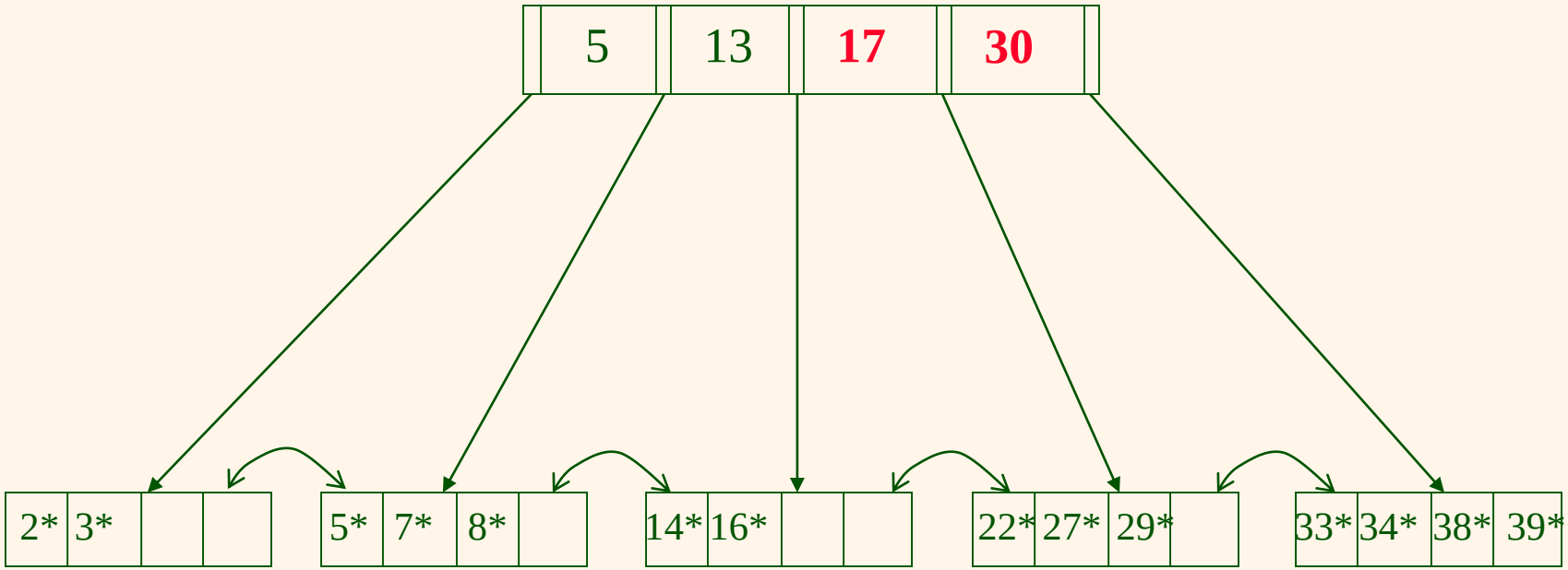
Deleting 24 (1)*



Deleting 24 (2)*

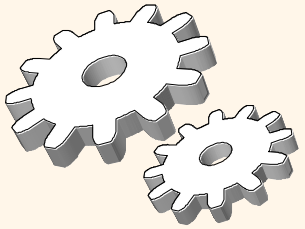


Deleting 24 (3)*



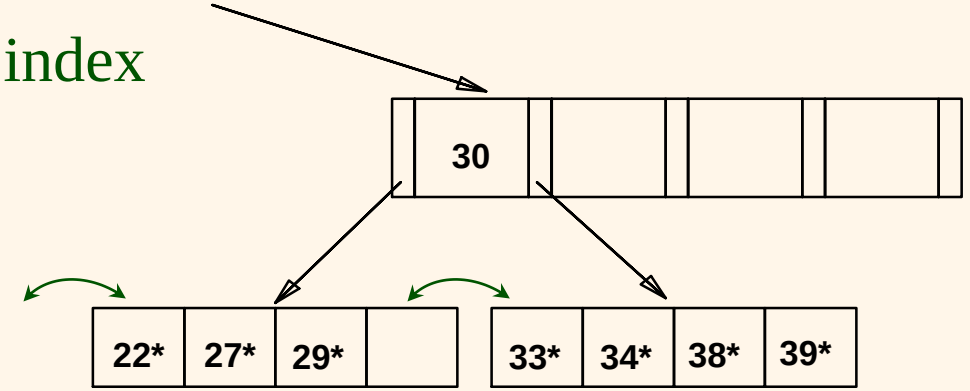
- After deleting 24*, the tree shrinks.

"Delete" vs. "Pull Down"



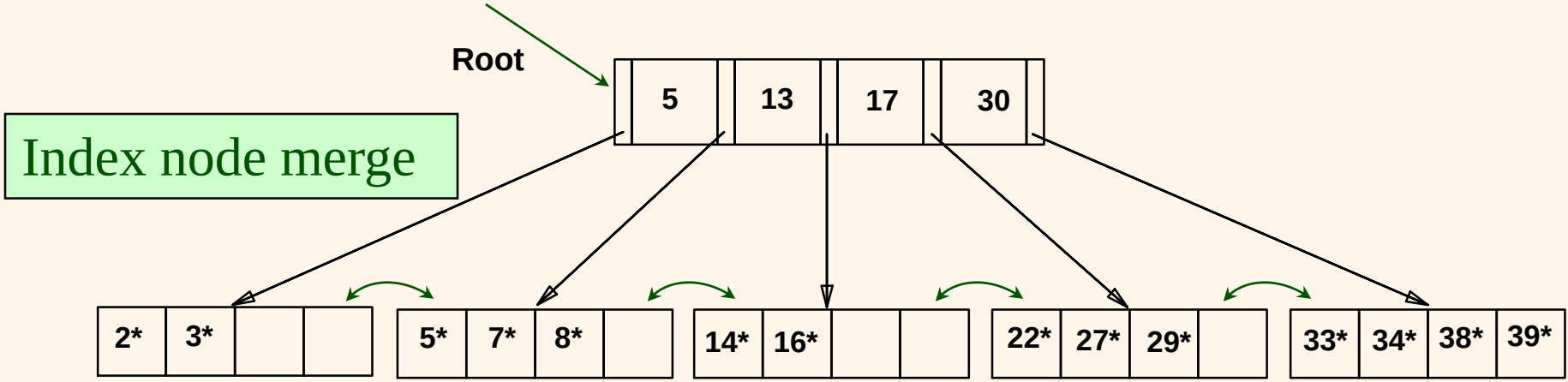
- Merge leaf nodes, *delete* index entry in parent

Leaf node merge

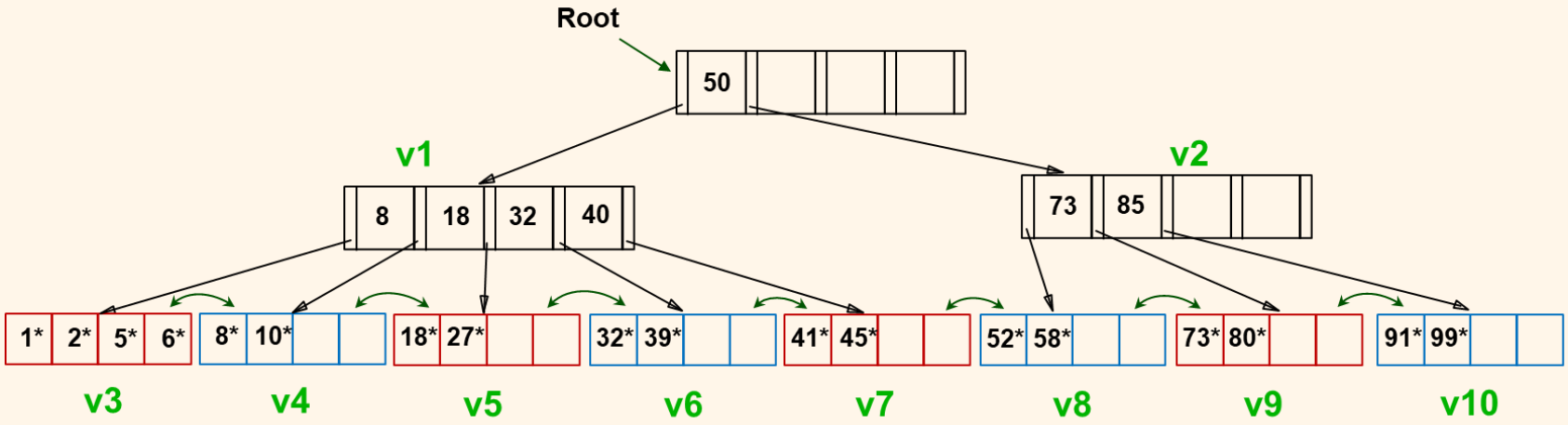


- Merge index nodes, *pull down* index entry in parent

Index node merge

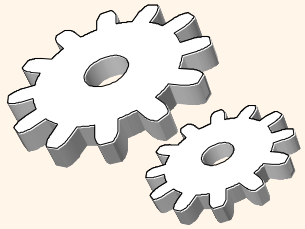


Questions 4 & 5



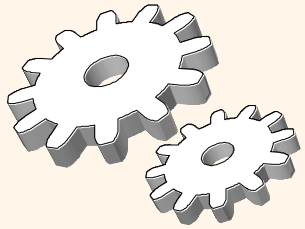
- Show the B+ tree that would result from deleting the data entry with key 8 from the *original tree*, assuming that the *left* sibling is checked for possible re-distribution.
- Show the B+ tree that would result from deleting the data entry with key 8 from the *original tree*, assuming the *right* sibling is checked for possible re-distribution.

B+ Trees in Practice



- Typical order: d = 100
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Create Index in SQL



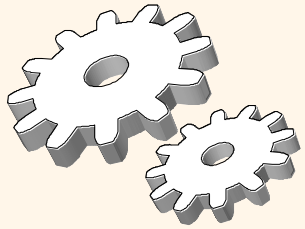
- **create index** <index-name> **on** <relation-name>(<attribute-list>)

- E.g.: **create index** *my-index* **on** STUDENTS(stu-id)

- To drop an index

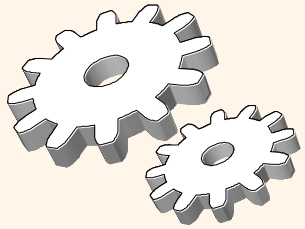
  drop index <index-name>

Summary



- Tree-structured indexes are ideal for range-searches, also good for equality searches
- ISAM is a static structure
  - Only leaf pages modified
  - Overflow pages needed
- B+ tree is a dynamic structure
  - Inserts/deletes leave tree height-balanced; cost =1+$\lceil \log_F N \rceil$
  - High fanout (**F**) means depth rarely more than 3 or 4

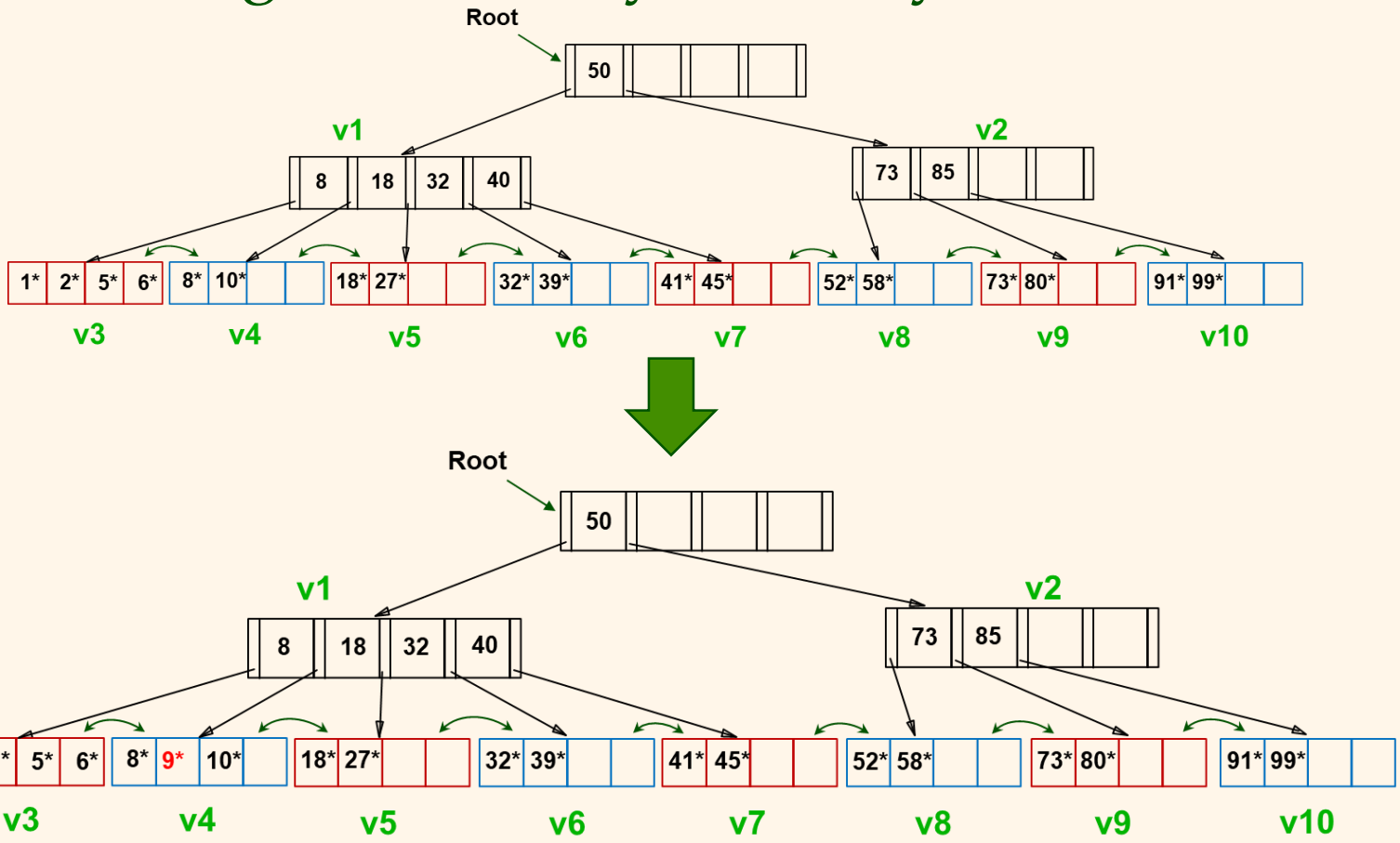
Solution to Question 1



- 1,000,000 records, 10 data records per page, 100 data entries per (leaf) page, 100 index entries per (non-leaf) page。 Estimate the search I/O cost of ISAM and the I/O cost of binary search.
- Solution:
  - B=#data pages=10^6 records/(10 records per page)=10^5
  - N=#primary leaf pages (each data entry corresponds to a record)
    = 10^6 data entries/(100 data entries per leaf page)=10^4
  - F=#children per non-leaf page=1+#index entries per non-leaf page=101
  - I/O cost for ISAM: $1+\lceil \log_F N \rceil = 1+\lceil \log_{101} 10^4 \rceil = 3$ I/Os
  - I/O cost for binary search: $\lceil \log_2 B \rceil = \lceil \log_2 10^5 \rceil = 17$ I/Os

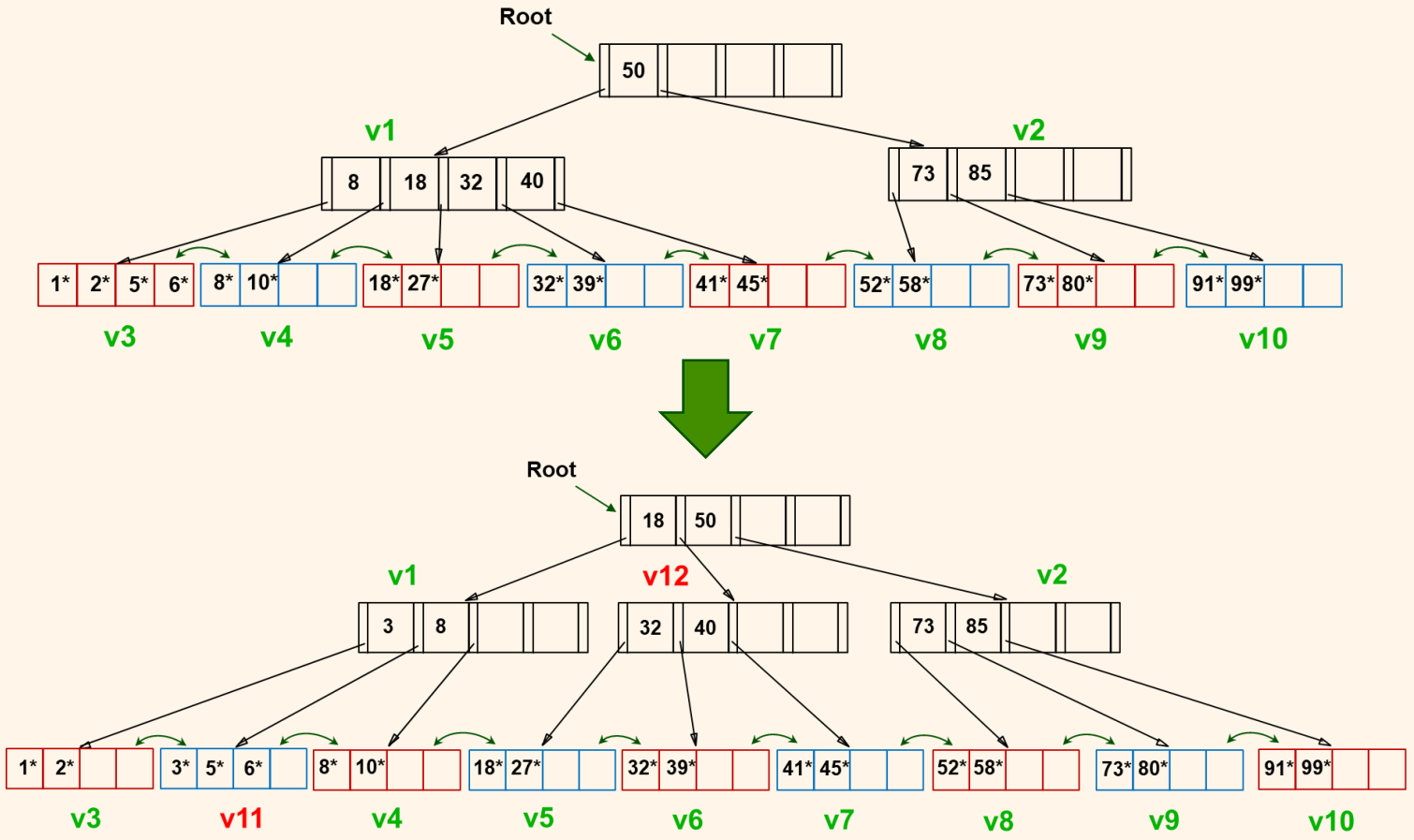
Solution to Question 2

- Show the tree that would result from inserting a data entry with key 9 into this tree

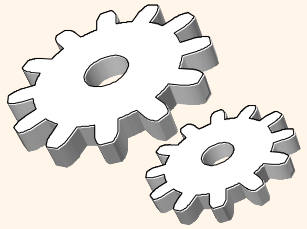


Solution to Question 3

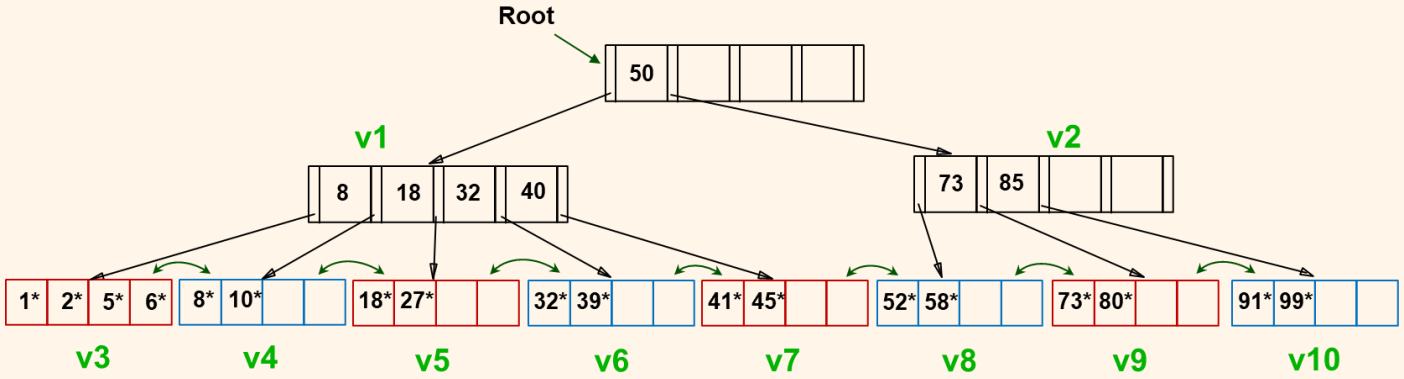
- Show the B+ tree that would result from inserting a data entry with key 3 into the original tree.



Solution to Question 4

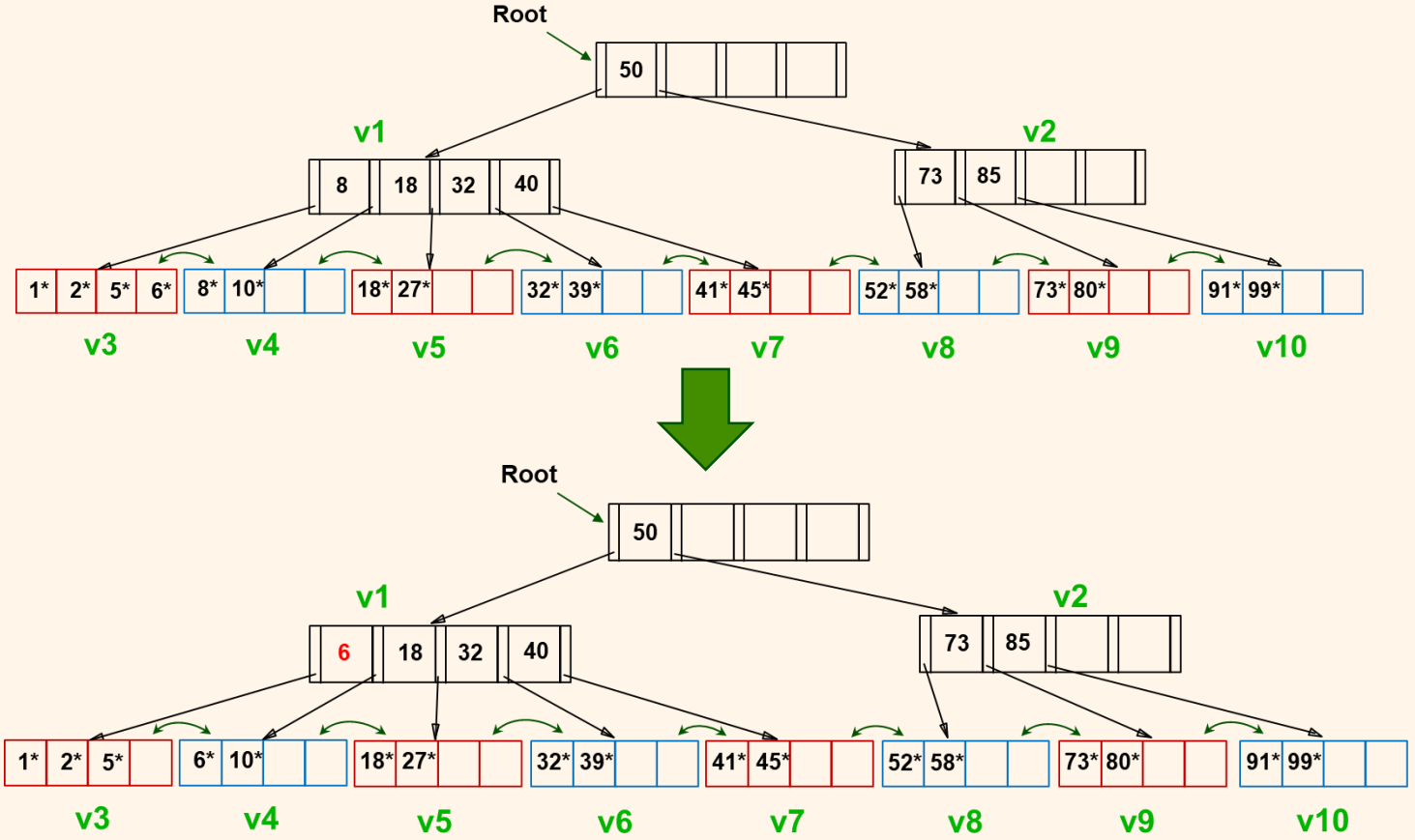


- Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the left sibling is checked for possible re-distribution.
  - The data entry with key 8 is deleted, resulting in v4 underflow
  - The left sibling v3 of v4 is checked for redistribution. Since v3 has more than 2 data entries, redistribution between v3 and v4 takes place
  - Middle key copied up

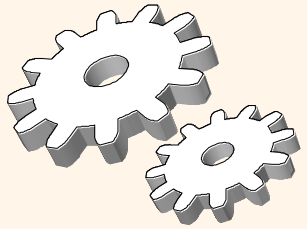


Solution to Question 4

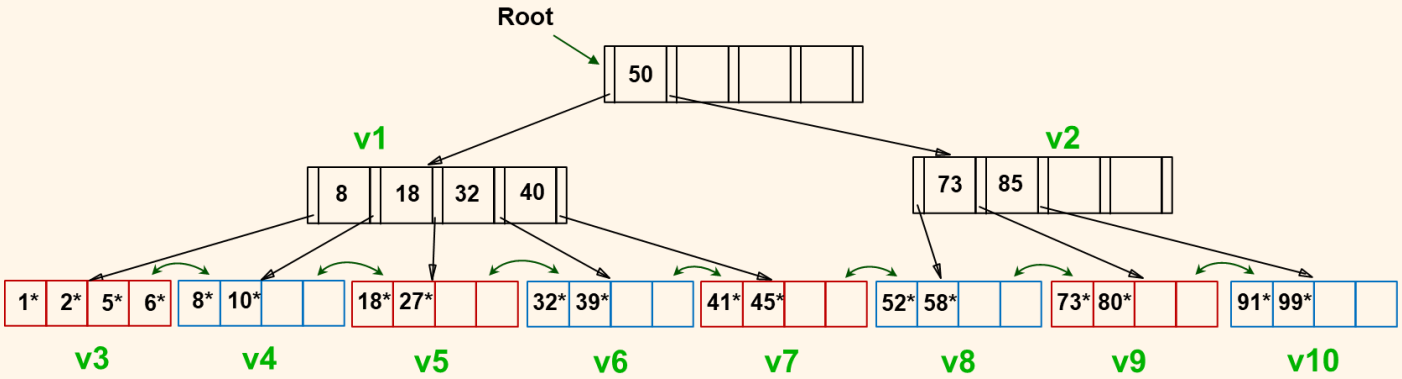
- Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming that the left sibling is checked for possible re-distribution.



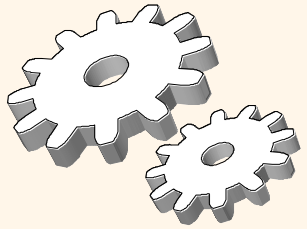
Solution to Question 5



- Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming the right sibling is checked for possible re-distribution.
  - The data entry with key 8 is deleted, resulting in v4 underflow
  - The right sibling v5 of v4 is checked for redistribution. Since v5 has two data entries, we cannot do redistribution. We merge v4 and v5 and delete the entry pointing to v5 (in its parent node)



Solution to Question 5



- Show the B+ tree that would result from deleting the data entry with key 8 from the original tree, assuming the right sibling is checked for possible re-distribution.