Chapter 14: Indexing

Database System Concepts, 7th Ed.

Outline

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- B-Tree Index Files
- Hashing
- Write-optimized indices
- Spatio-Temporal Indexing

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library

- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

  - **Search Key** - attribute to set of attributes used to look up records in a file.

- Index files are typically much smaller than the original file

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - Records with a specified value in the attribute
  - Records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value.

- **Clustering index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **primary index**
  - The search key of a primary index is usually but not necessarily the primary key.

- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index.**

- **Index-sequential access file (ISAM):** sequential file ordered on a search key, with a clustering index on the search key.
  - Designed for applications that require both sequential and random access to individual/set of records.

Types of Ordered Indices

1. Dense Index Files

Dense index — Index record appears for every search-key value in the file.

– E.g. index on *ID* attribute of *instructor* relation

10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

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	

Identifier of a disk bock and an offset within the disk block

Dense Index Files (Cont.)

- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

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	
33465	Gold	Physics	87000	

In a dense secondary (non-clustering) index, the index must store a list of pointers to all records with the same search-key value.

2. Sparse Index Files

Sparse Index: contains index records for only some search-key values.

- Applicable when records are sequentially ordered on search-key

- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points

10101
32343
76766

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

Compared to dense indices:

- Less space and less maintenance overhead for insertions and deletions.
- Generally slower than dense index for locating records.

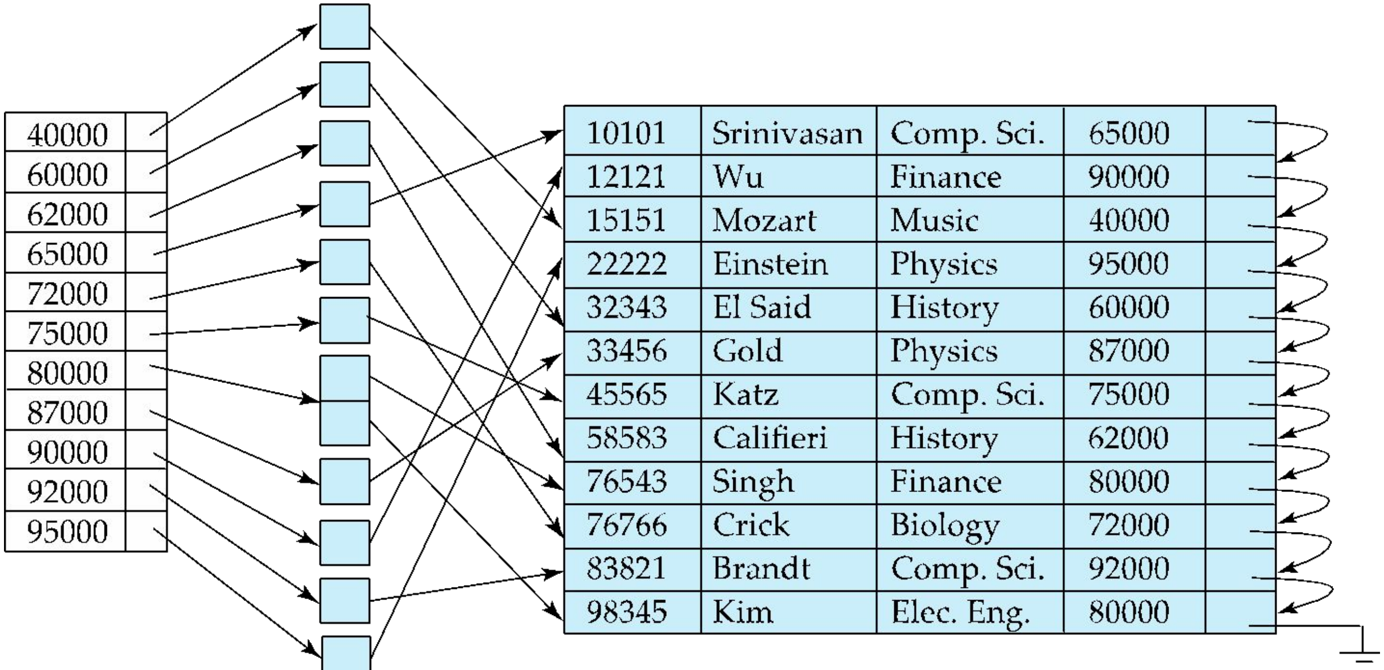
Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values

- We can have a secondary index with an index record for each search-key value

Secondary Indices Example

- Secondary index on salary field of instructor



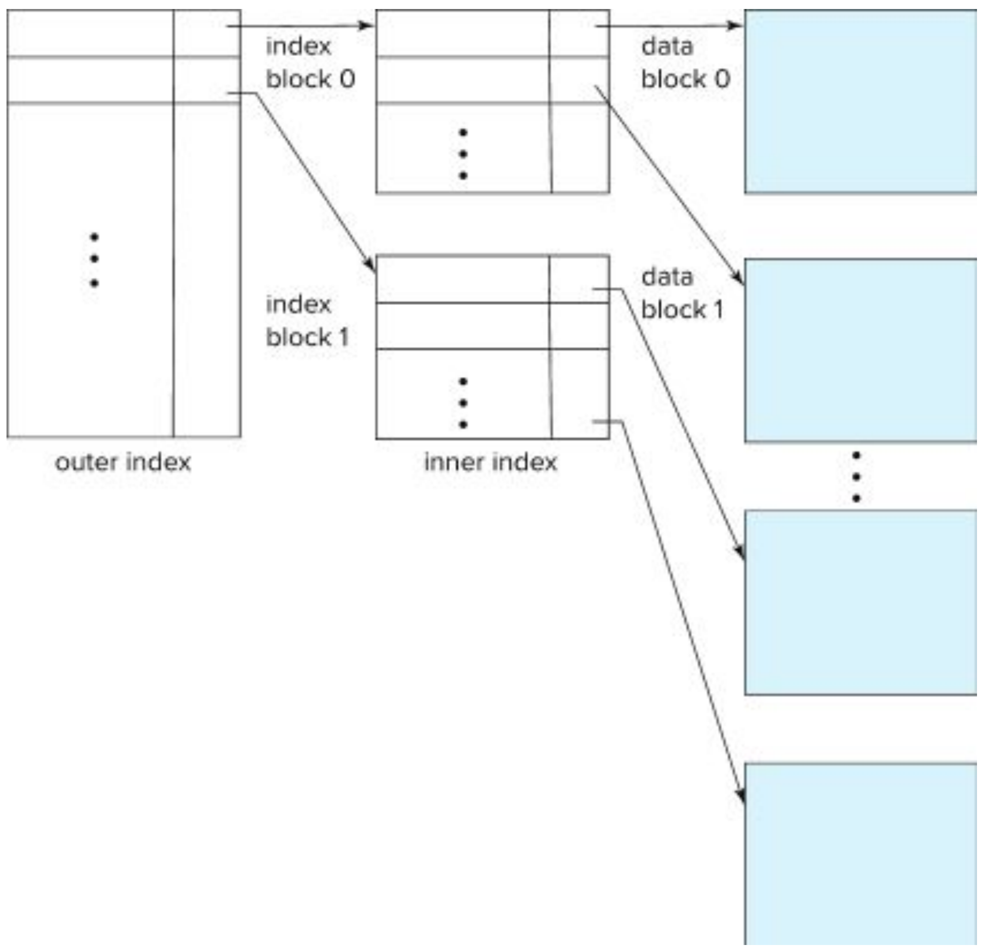
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Q: Does it make sense to create a Sparse index file for secondary index?

Multilevel Index

- If index does not fit in memory, access becomes expensive.
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of the basic index
  - inner index – the basic index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: Deletion

10101	
32343	
76766	

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	

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.

- **Single-level index entry deletion:**
  - **Dense indices** – deletion of search-key is similar to file record deletion.
  - **Sparse indices** –
    - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Index Update: Insertion

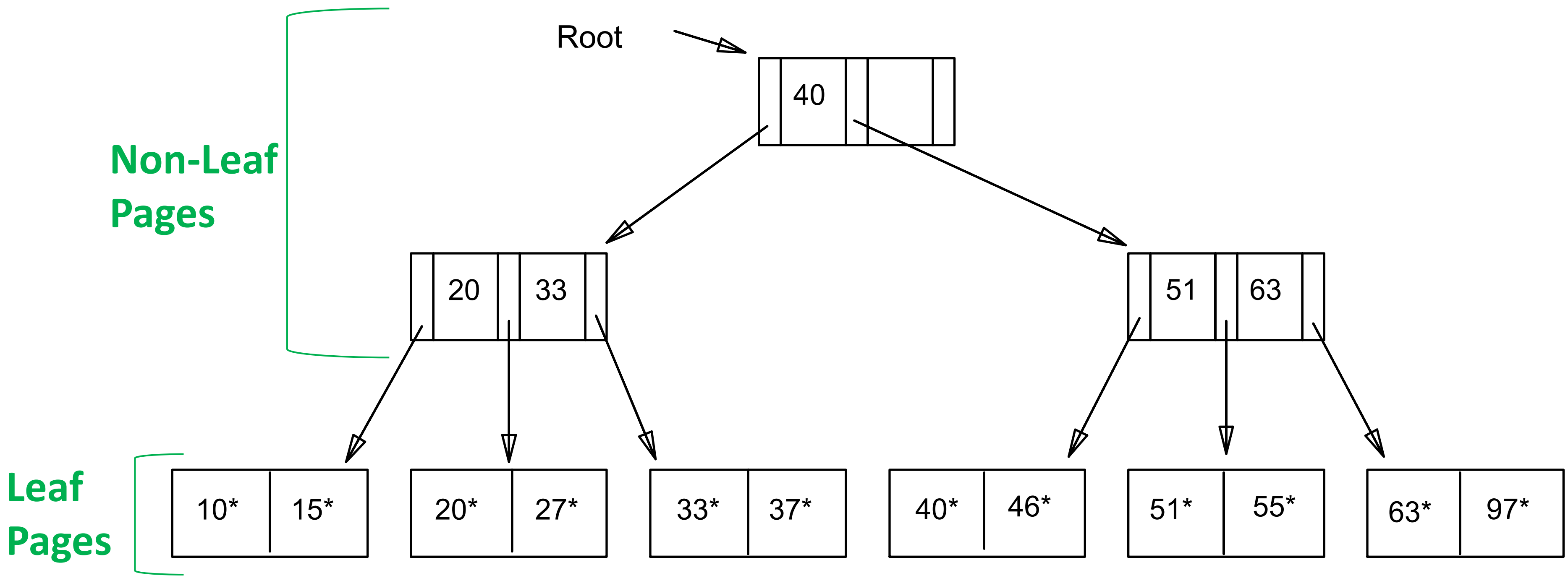
- **Single-level index insertion:**
  - Perform a lookup using the search-key value of the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it
    - Indices are maintained as sequential files
    - Need to create space for new entry, overflow blocks may be required
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms

Indices on Multiple Keys

- **Composite search key**
  - E.g., index on *instructor* relation on attributes (*name, ID*)
  - Values are sorted lexicographically
    - E.g. (John, 12121) < (John, 13514) and
      (John, 13514) < (Peter, 11223)
  - Can query on just *name*, or on (*name, ID*)

ISAM Trees

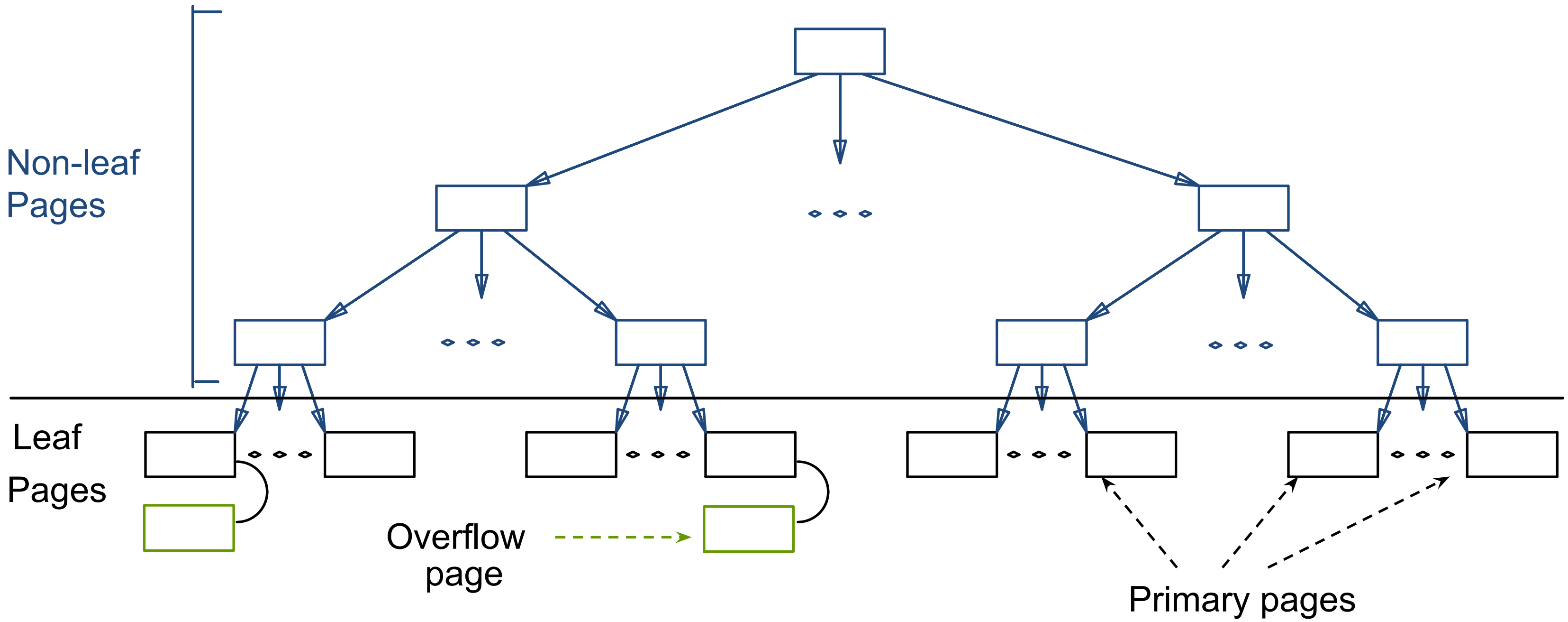
- Indexed Sequential Access Method (ISAM) trees are *static*



E.g., 2 Entries Per Page

ISAM Trees: Page Overflows

- What if there are a lot of insertions after creating the tree?

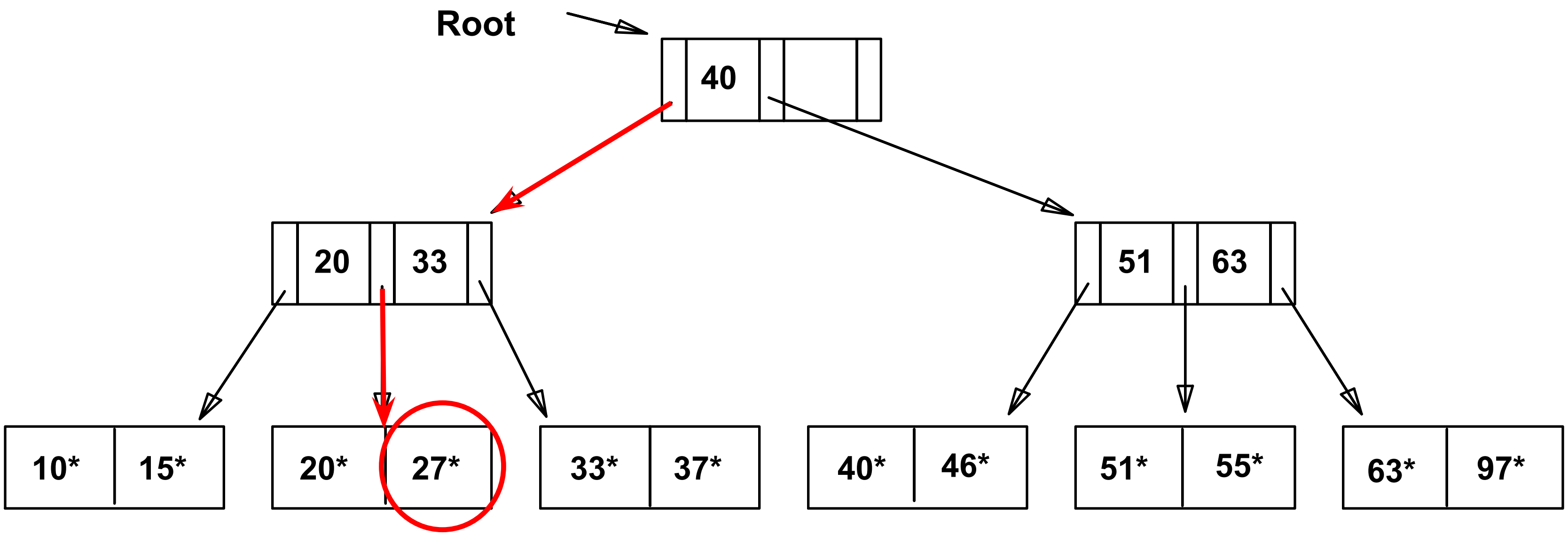


ISAM File Creation

- How to create an ISAM file?
  - All leaf pages are allocated *sequentially* and *sorted* on the search key value
  - Or the data records are created and *sorted* before allocating leaf pages
  - The non-leaf pages are subsequently allocated

ISAM: Searching for Entries

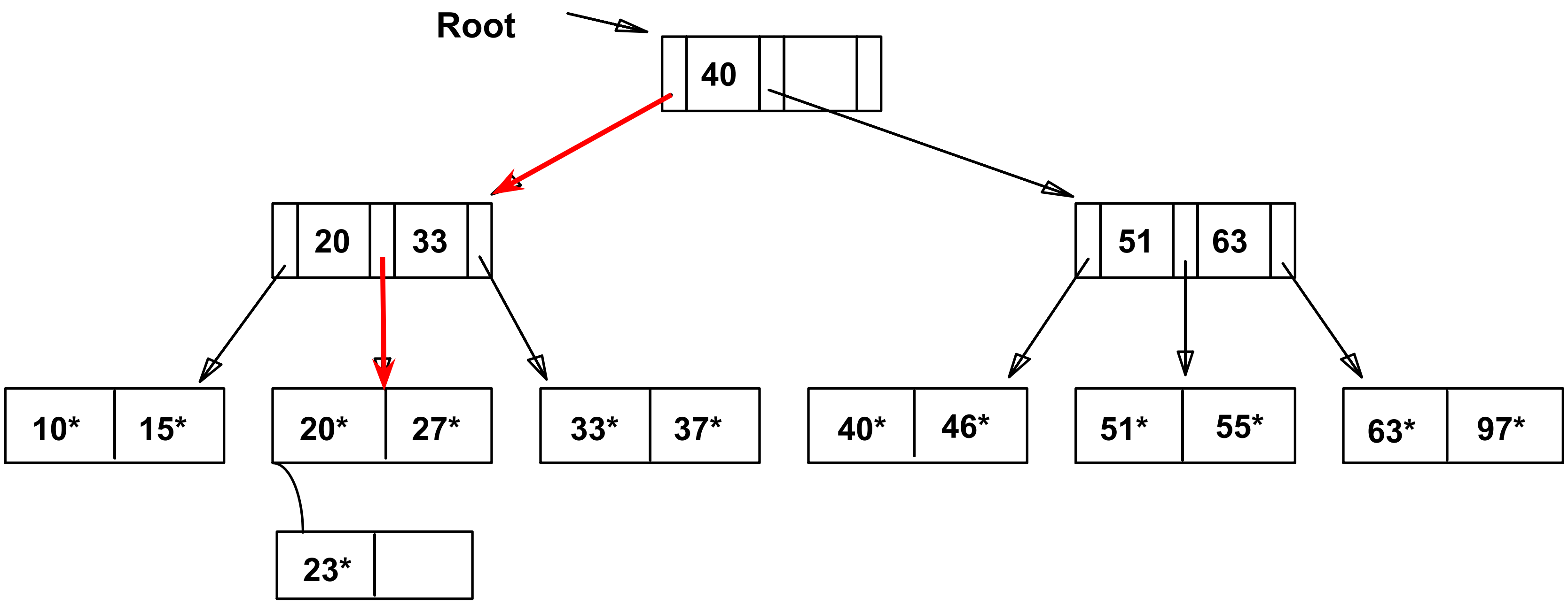
- Search begins at root, and key comparisons direct it to a leaf
- Search for 27*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

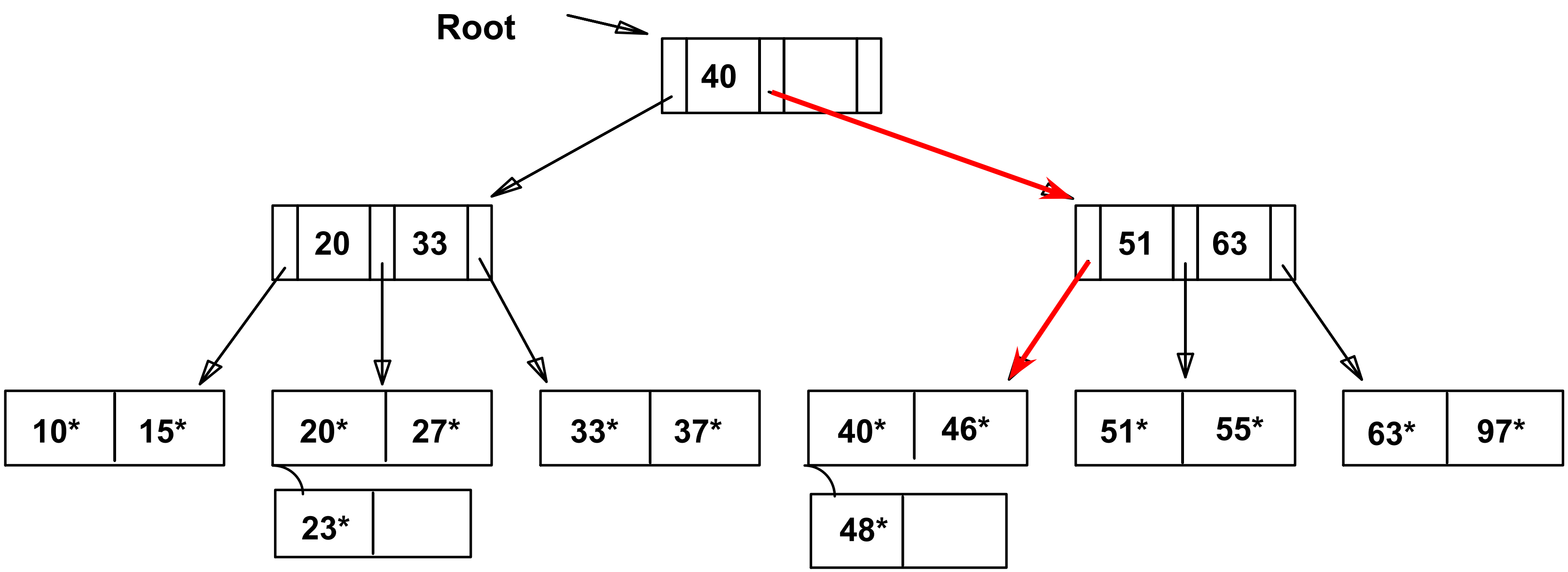
- Insert 23*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

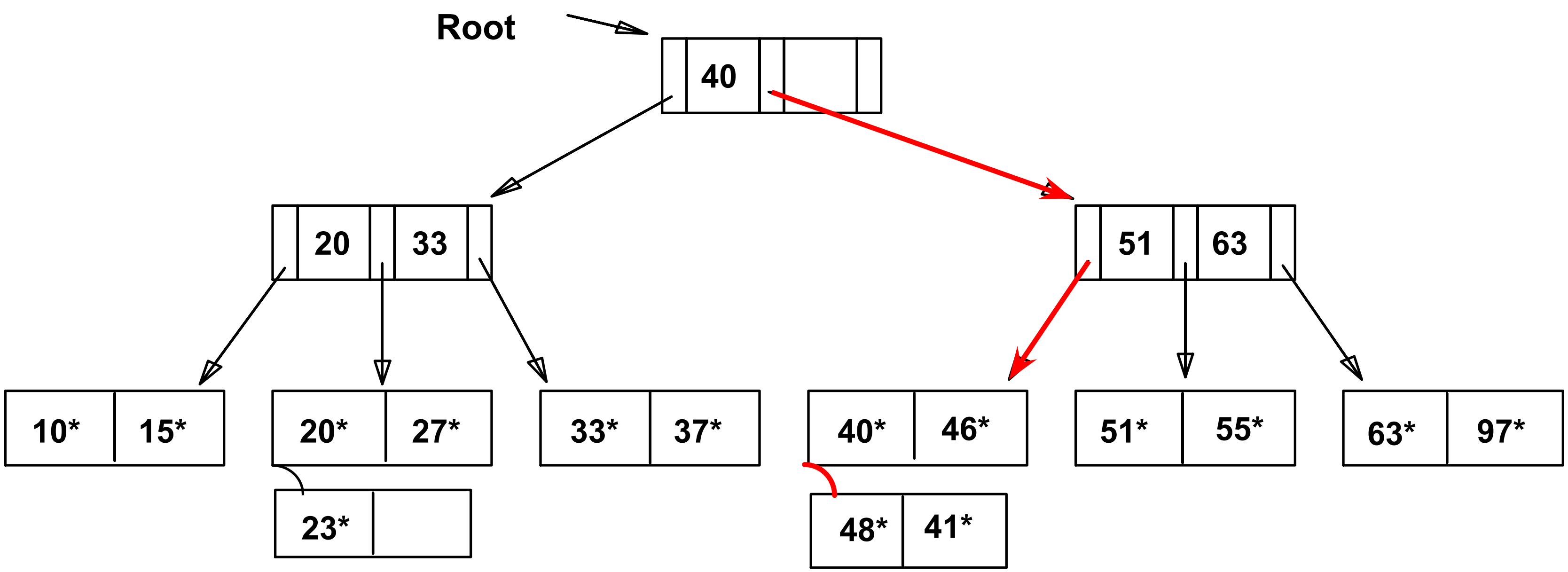
- Insert 48*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

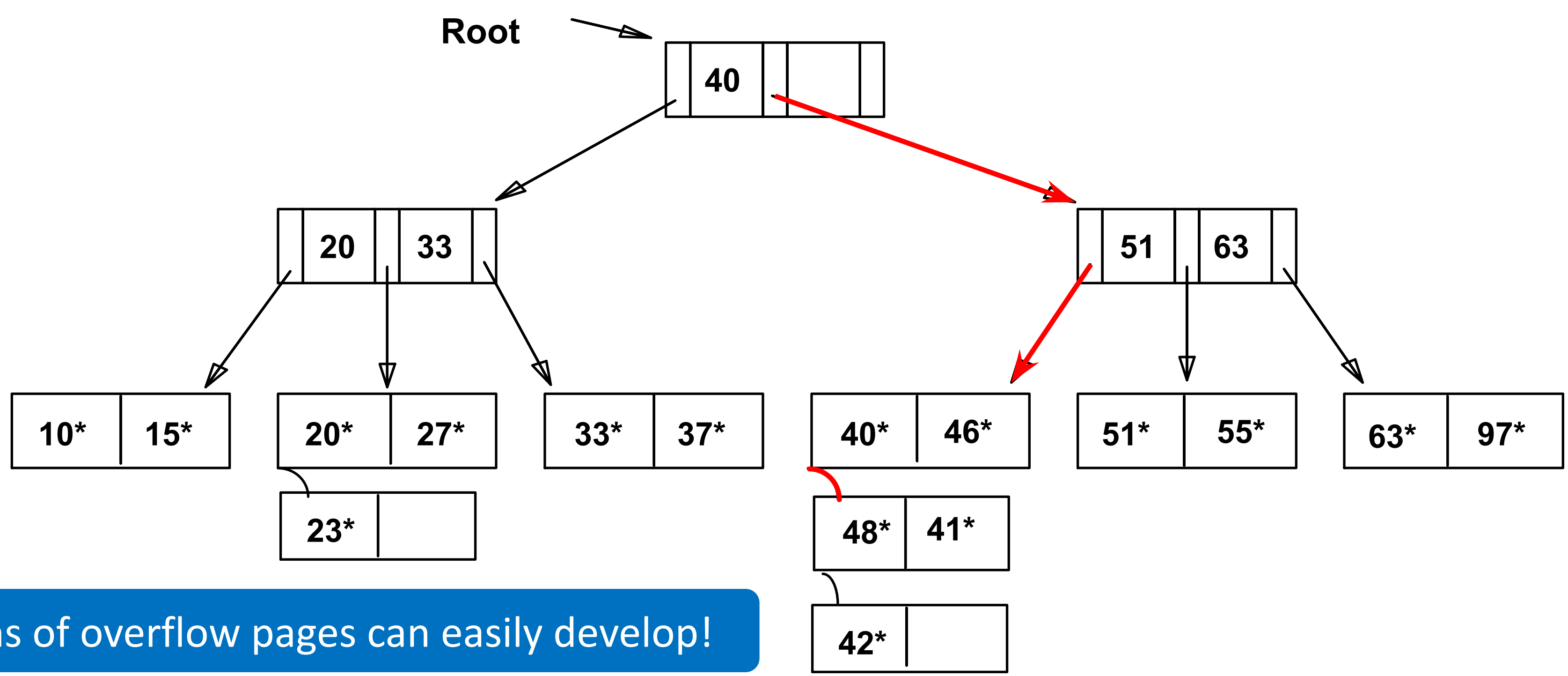
- Insert 41*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

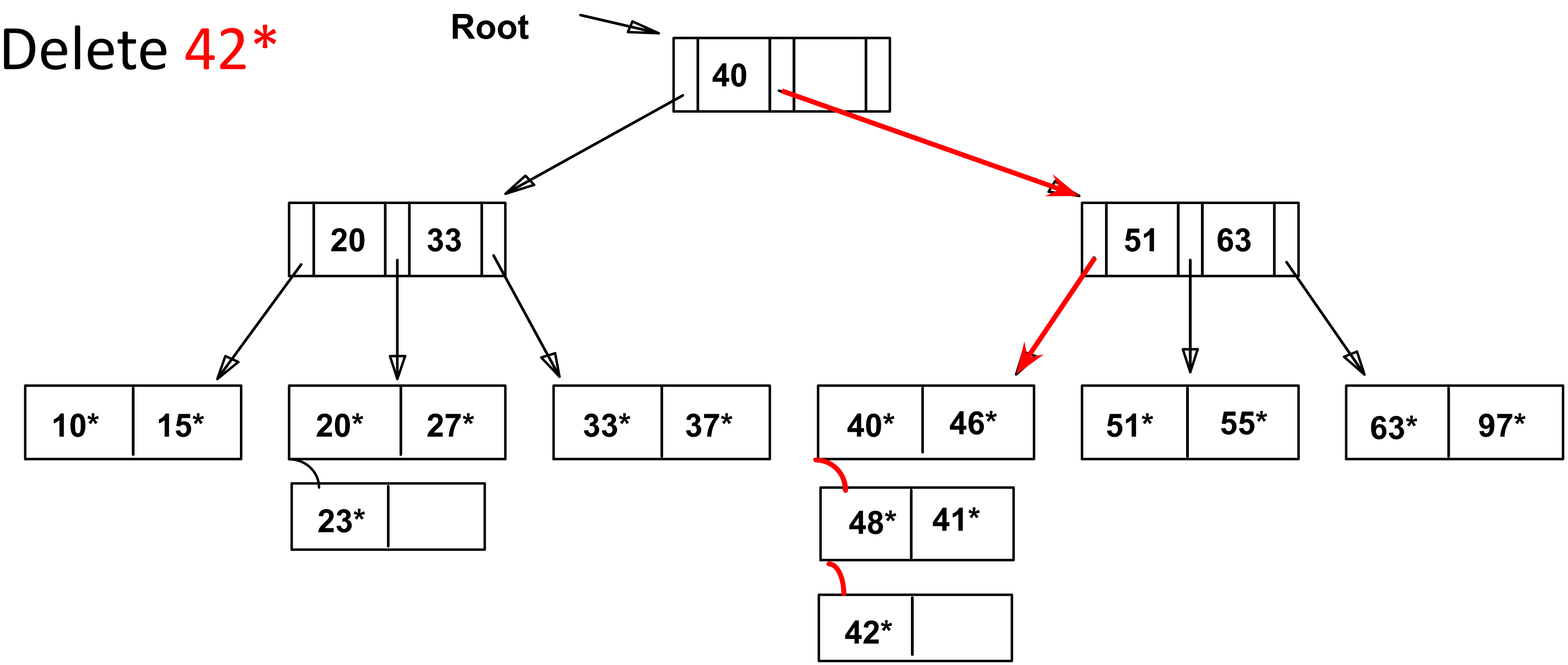
- Insert 42*



Chains of overflow pages can easily develop!

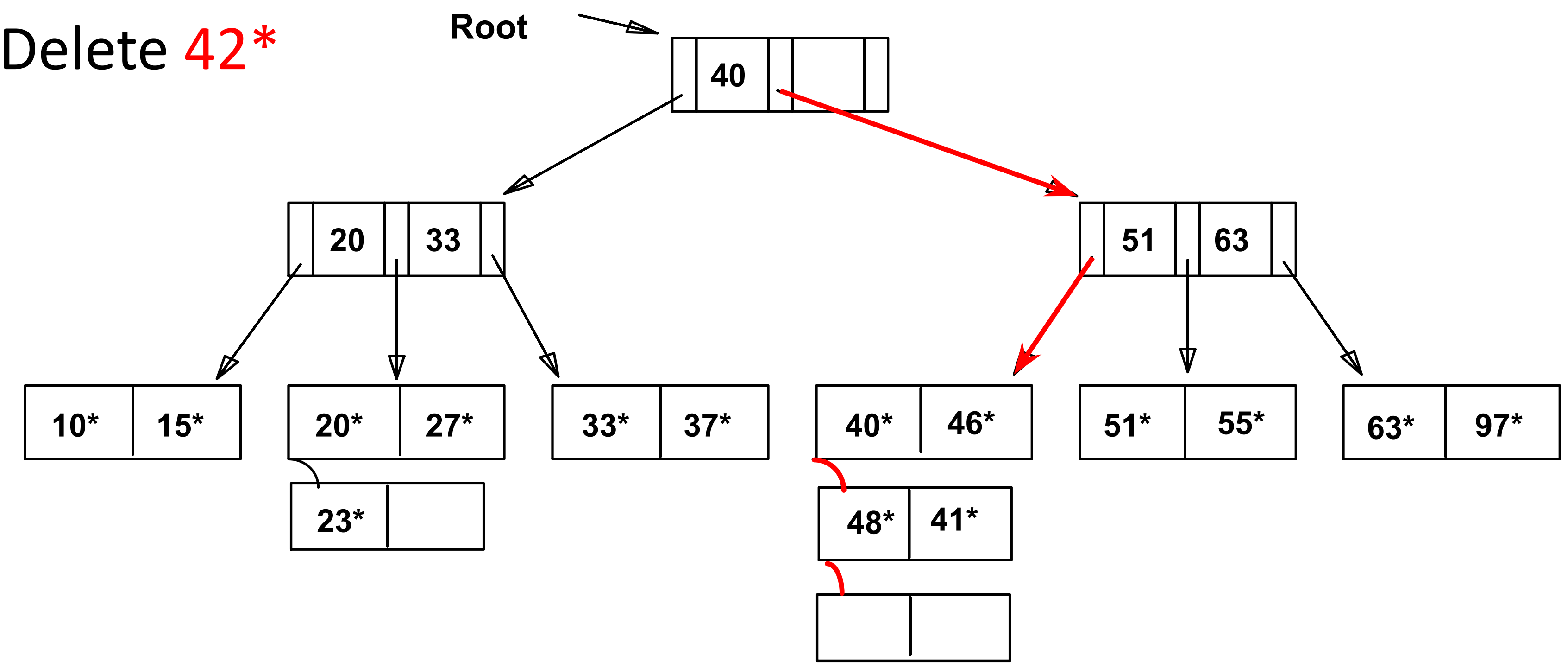
ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)
- Delete 42*



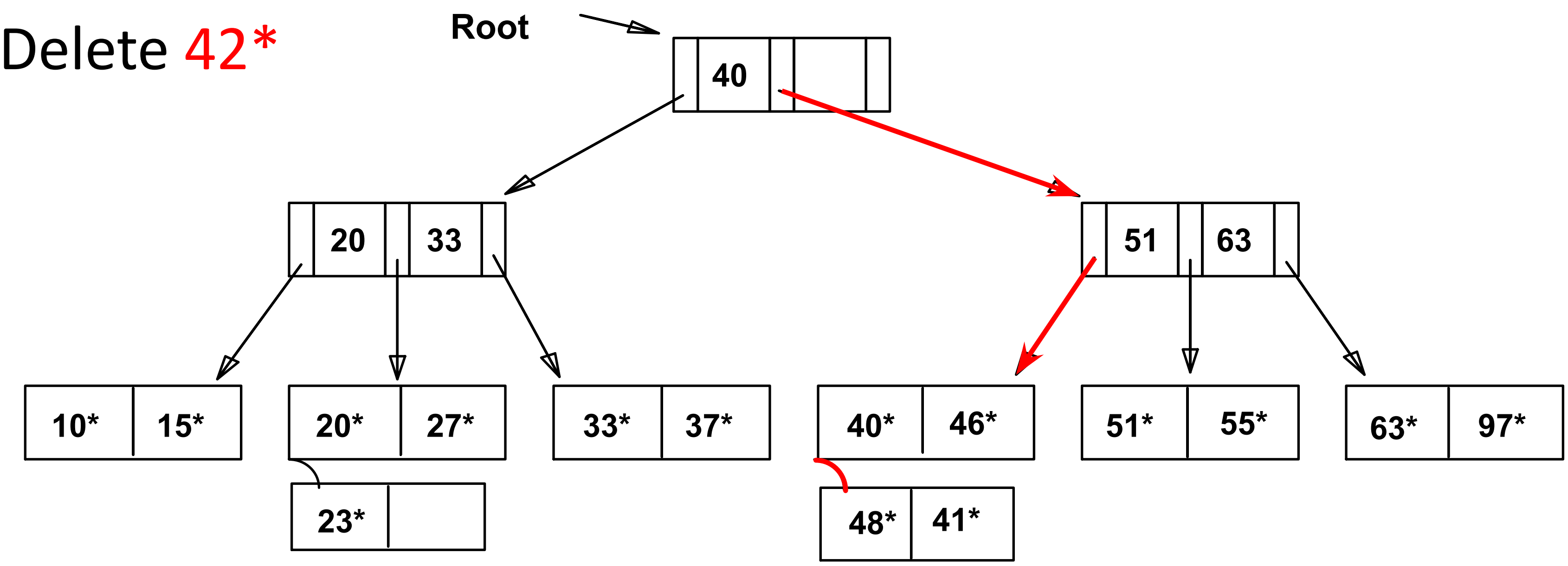
ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)
- Delete 42*



ISAM: Deleting Entries

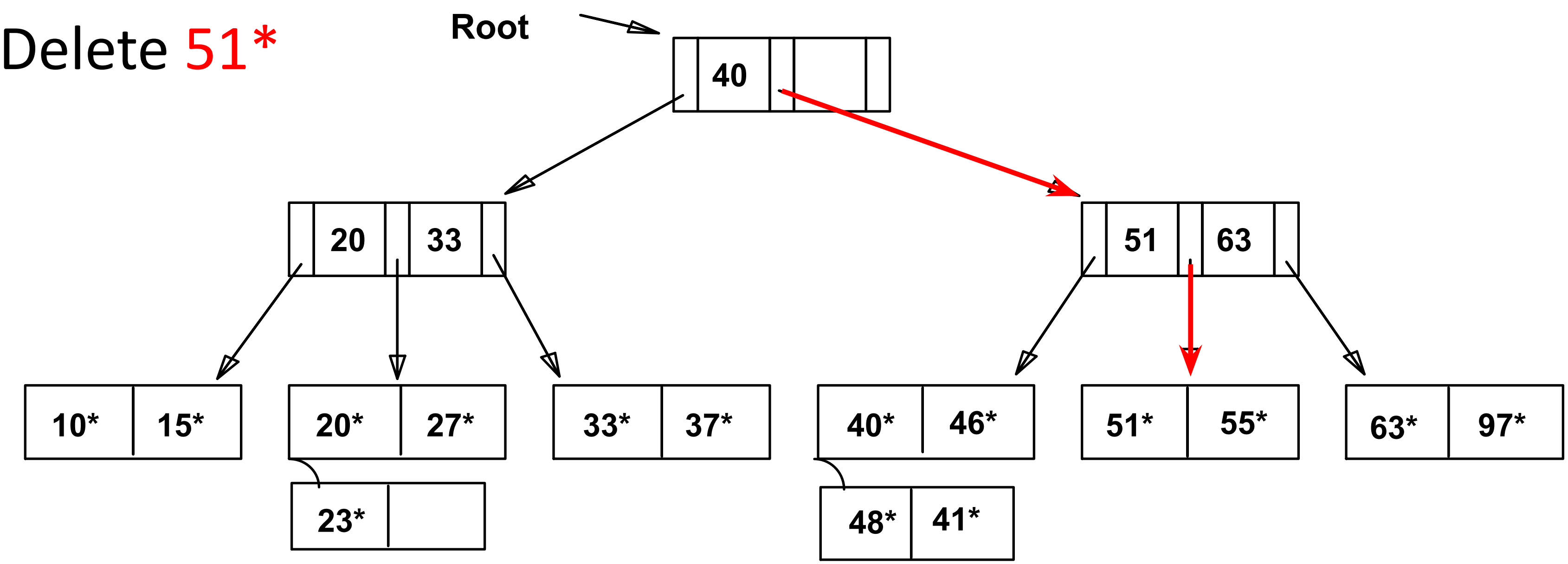
- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)
- Delete 42*



ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

- Delete 51*

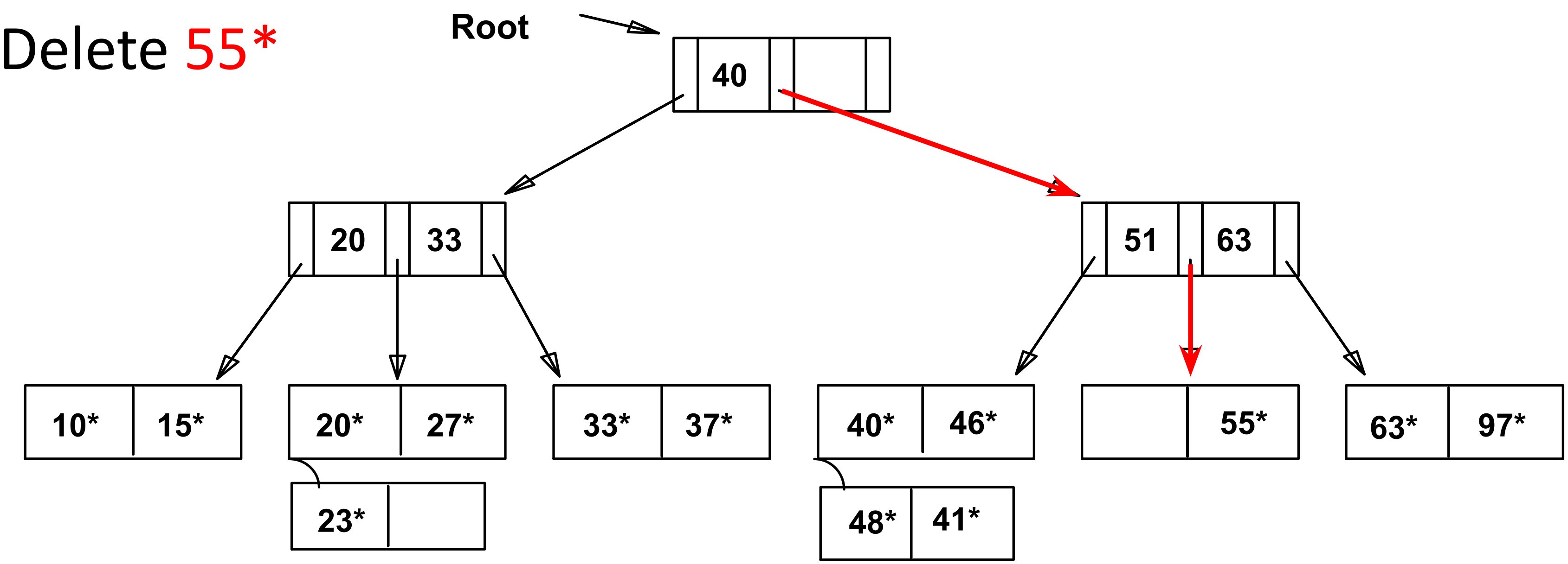


Note that 51 still appears in an index entry, but not in the leaf!

ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

- Delete 55*



Primary pages are NOT removed, even if they become empty!

ISAM: Some Issues

- Once an ISAM file is created, insertions and deletions affect only the contents of leaf pages (i.e., *ISAM is a static structure*!)

- Since index-level pages are *never* modified, there is no need to *lock* them during insertions/deletions
  - Critical for concurrency!

- Long overflow chains can develop easily
  - The tree can be initially set so that ~20% of each page is free

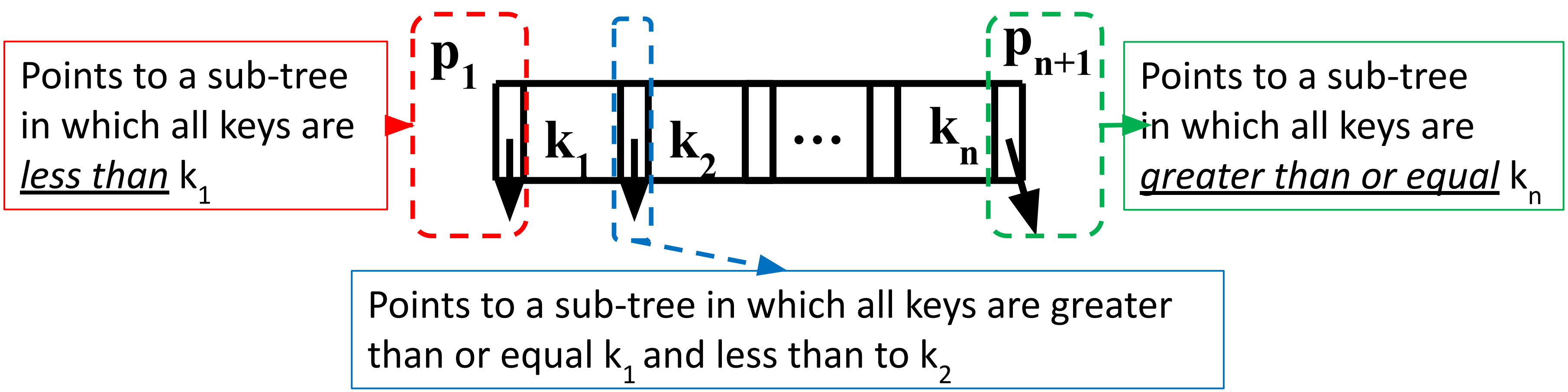
- If the data distribution and size are relatively static, ISAM might be a good choice to pursue!

Dynamic Trees

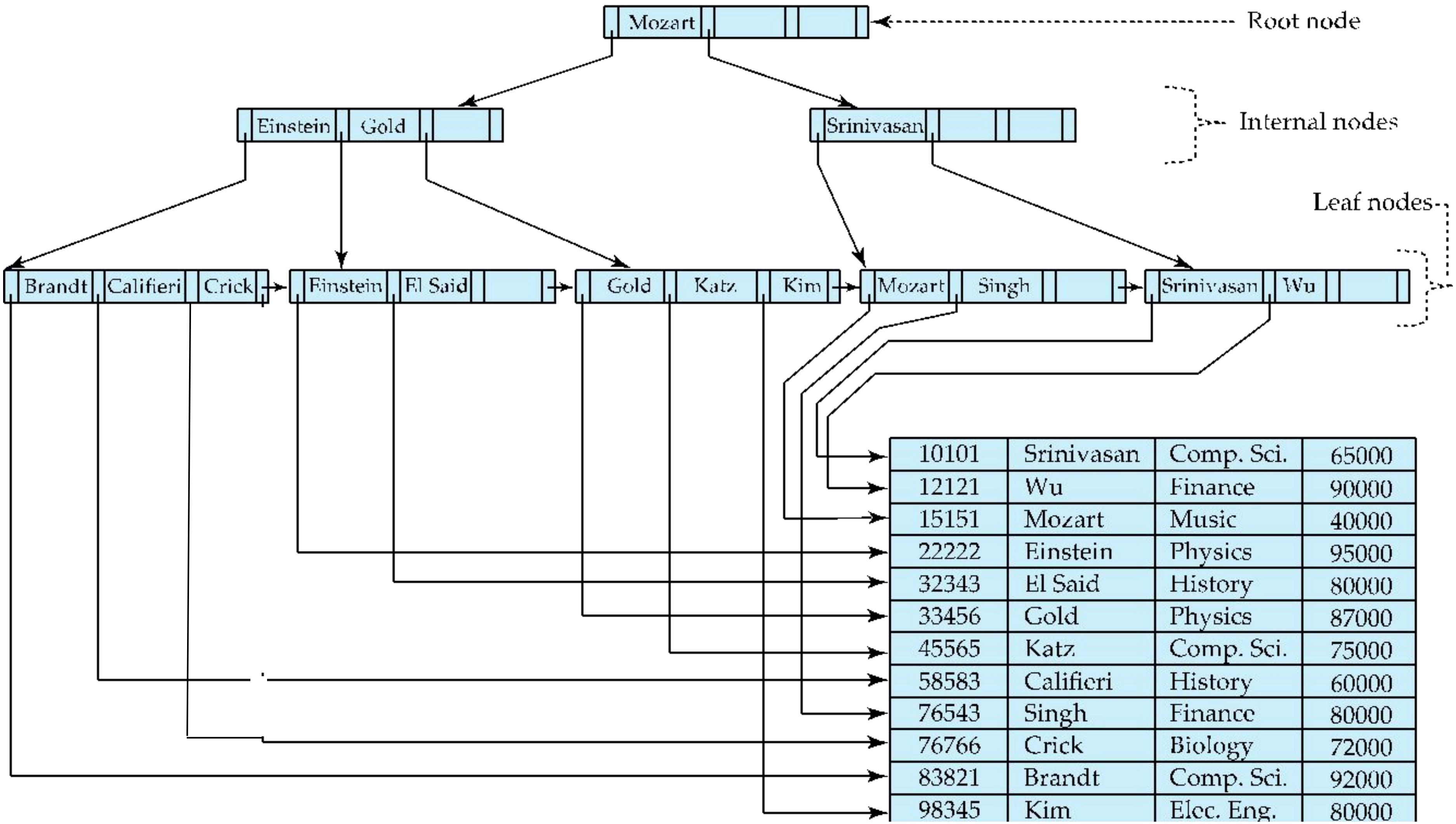
- ISAM indices are static
  - Long overflow chains can develop as the file grows, leading to poor performance
- This calls for more flexible, *dynamic* indices that adjust gracefully to insertions and deletions
  - No need to allocate the leaf pages sequentially as in ISAM
- Among the **most successful** dynamic index schemes is the B+ tree

B+ Tree Properties

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.



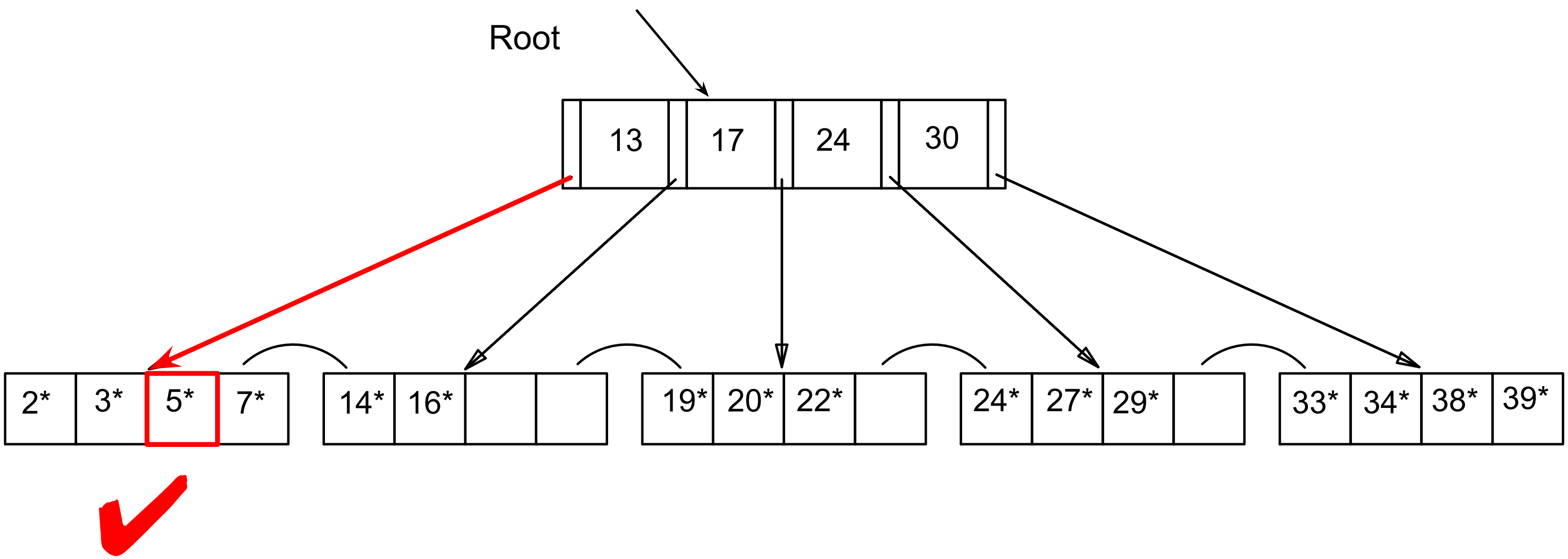
Example of B$^+$-Tree



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

B+ Tree: Searching for Entries

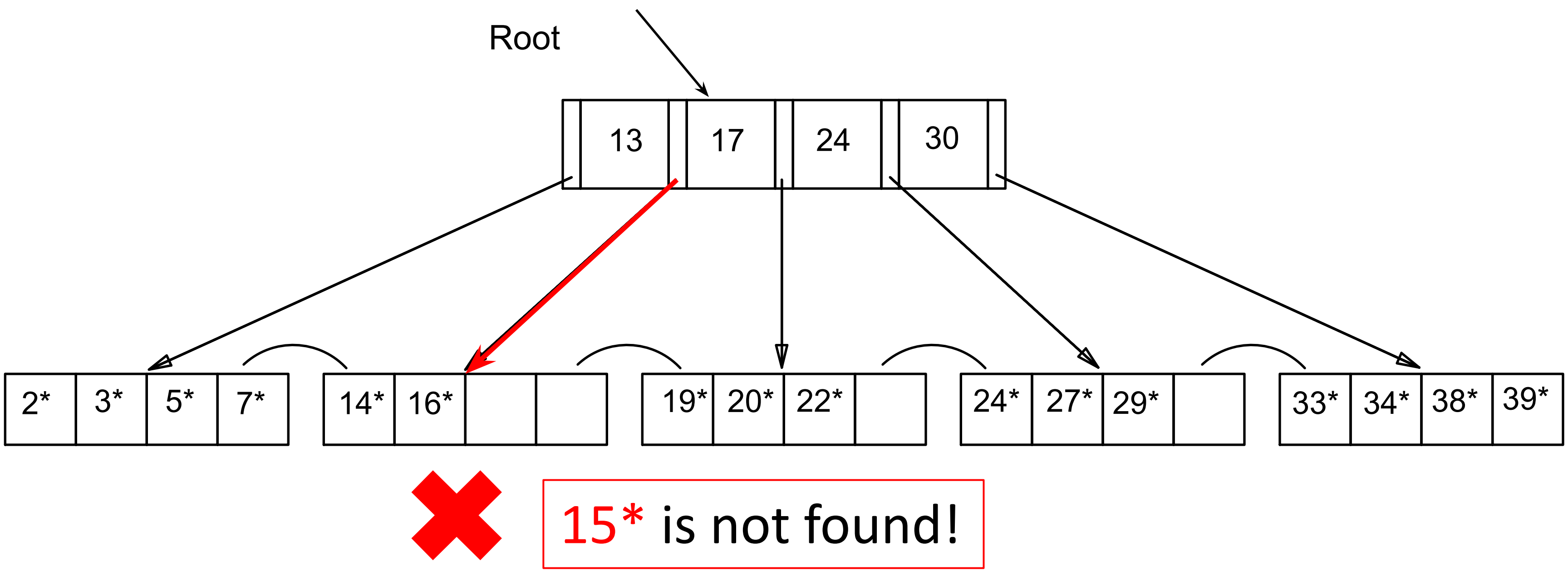
- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)
- Example 1: Search for entry 5*



B+ Tree: Searching for Entries

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)

- Example 2: Search for entry 15*

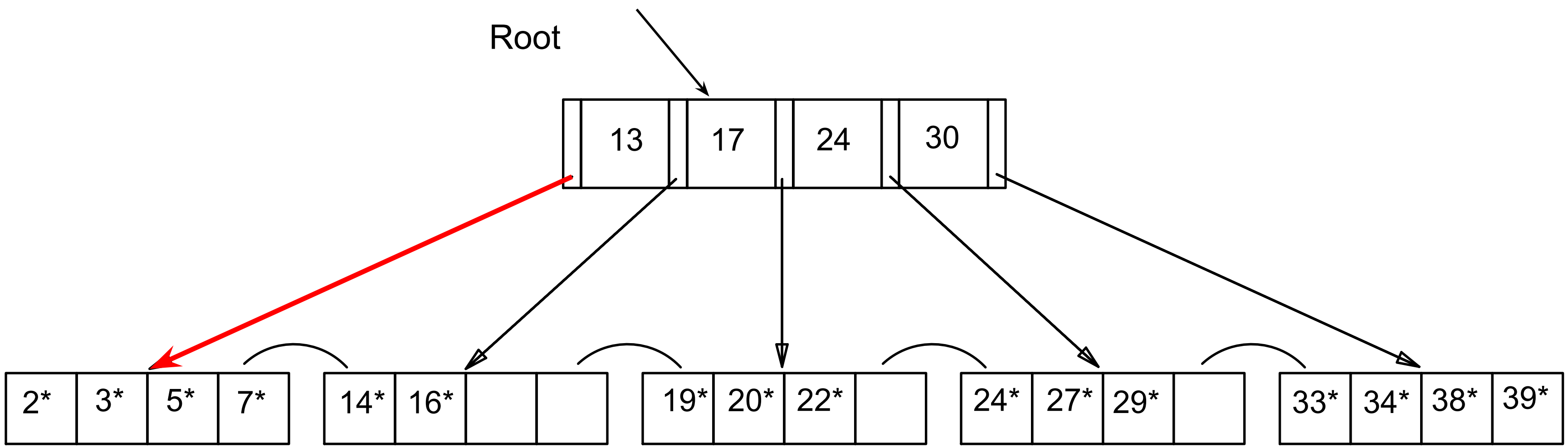


B+ Trees: Inserting Entries

- Find correct leaf ***L***
- Put data entry onto ***L***
  - If ***L*** has enough space, *done*!
  - Else, *split* ***L*** *into* ***L*** *and a new node* $\boldsymbol{L_2}$
    - Re-partition entries *evenly*, *copying up* the middle key
- Parent node may *overflow*
  - *Push up* middle key (splits "grow" trees; a root split increases the height of the tree)

B+ Tree: Examples of Insertions

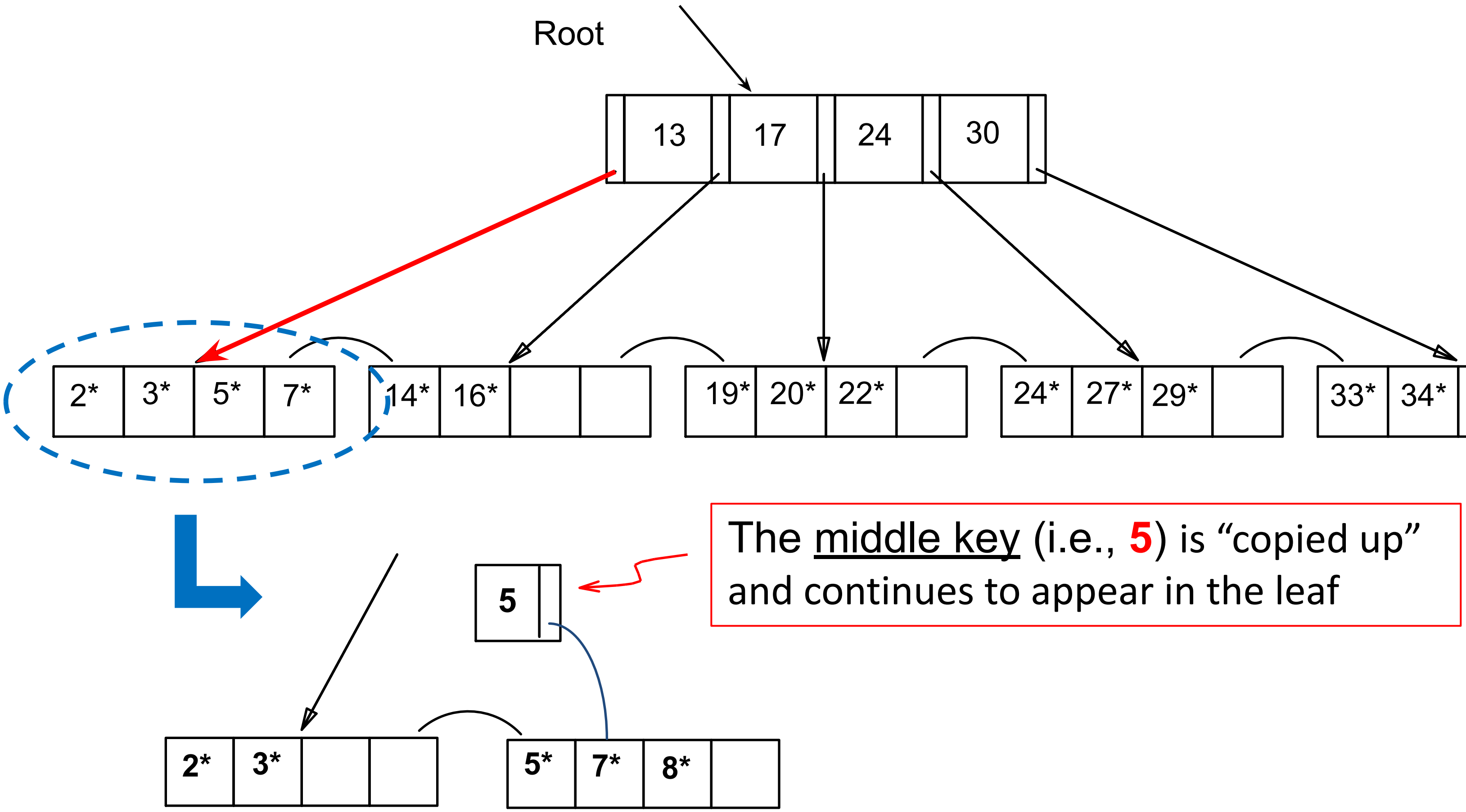
- Insert entry 8*



Leaf is ***full***; hence, split!

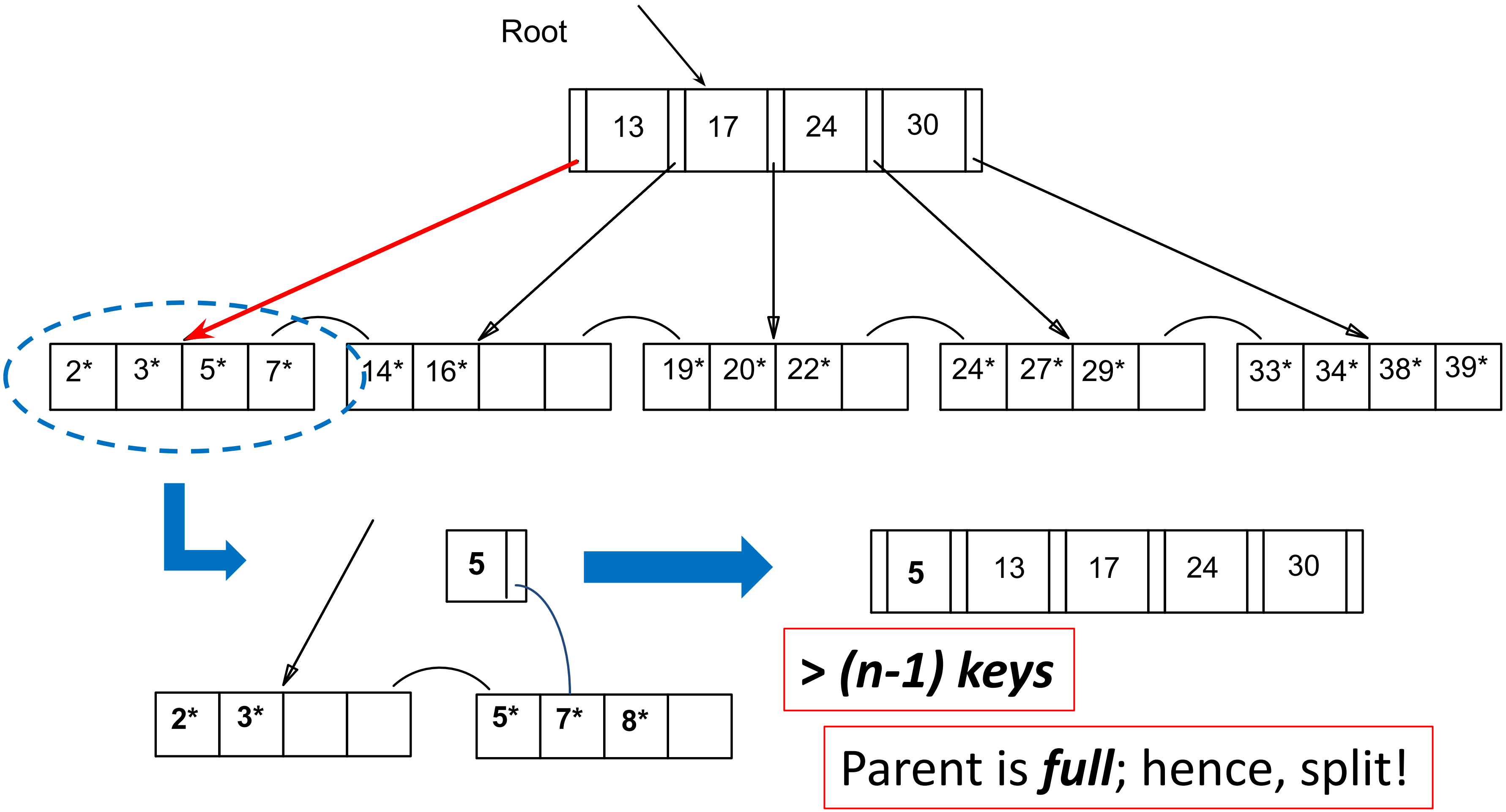
B+ Tree: Examples of Insertions

- Insert entry 8*



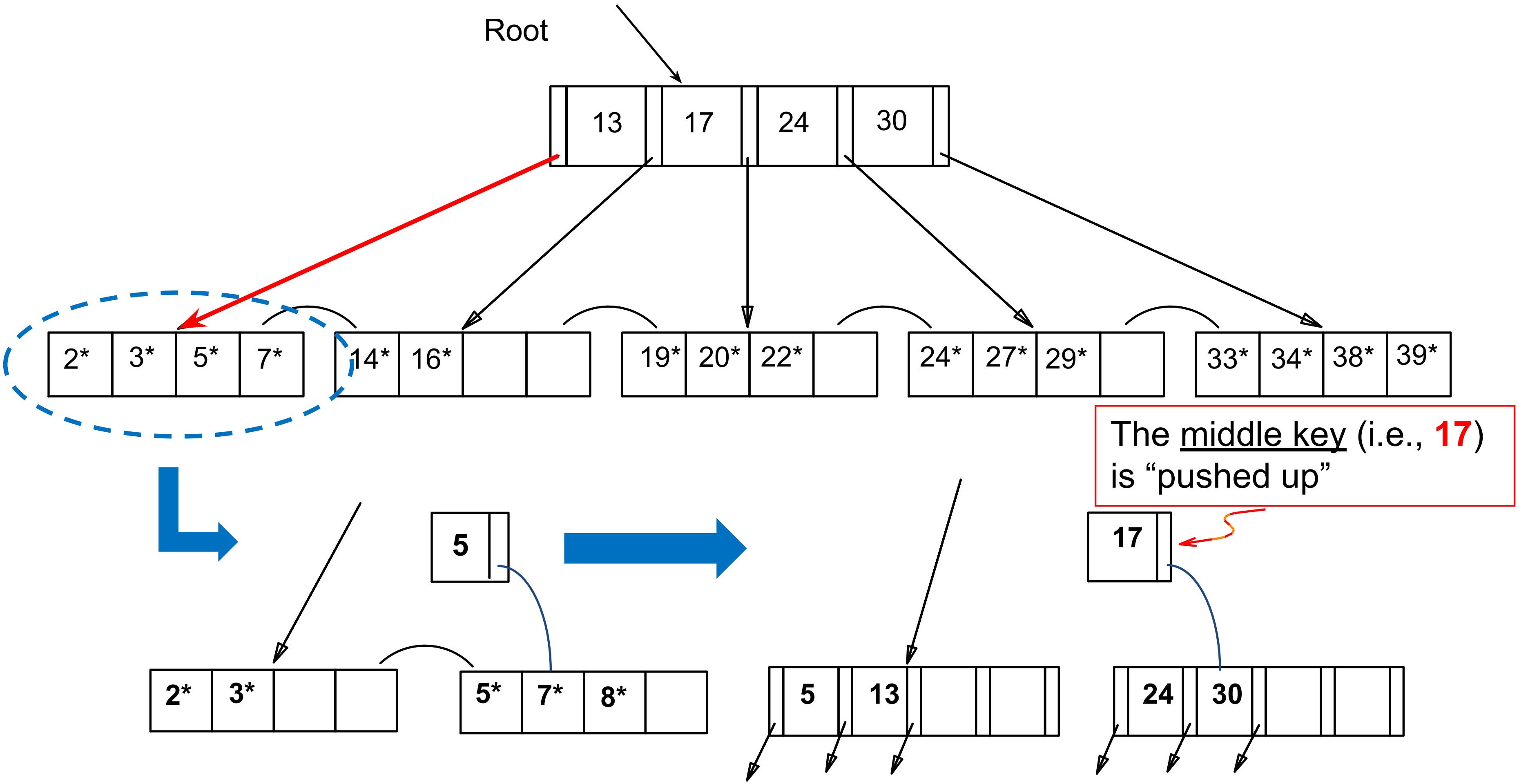
B+ Tree: Examples of Insertions

- Insert entry 8*



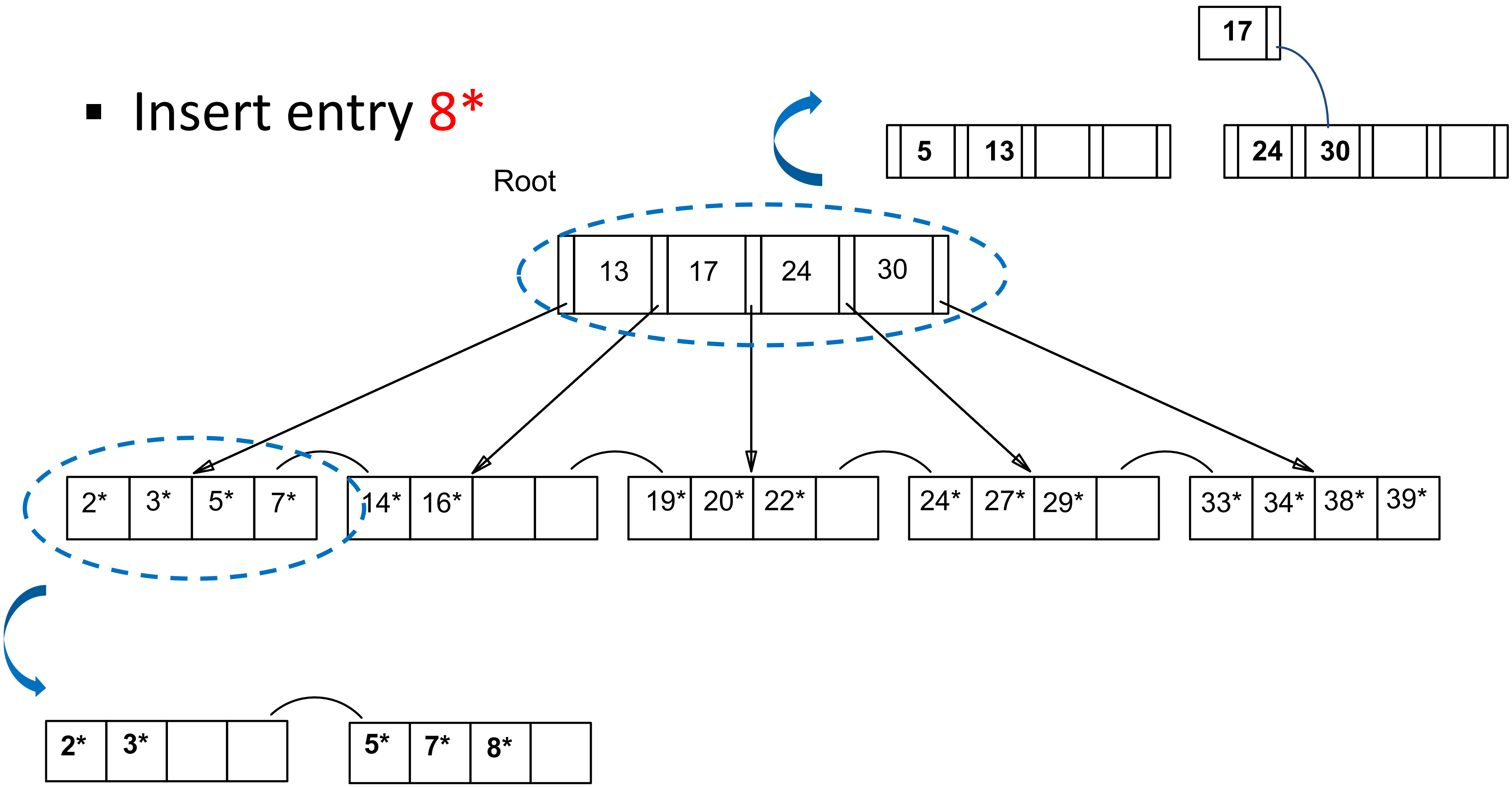
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

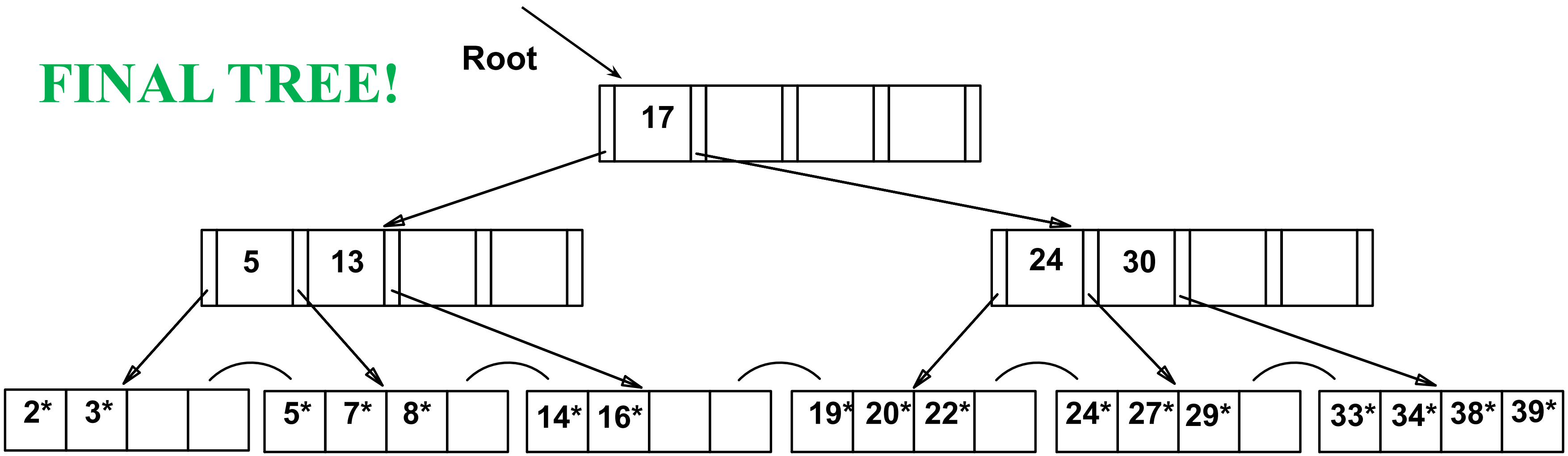
- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*

FINAL TREE!

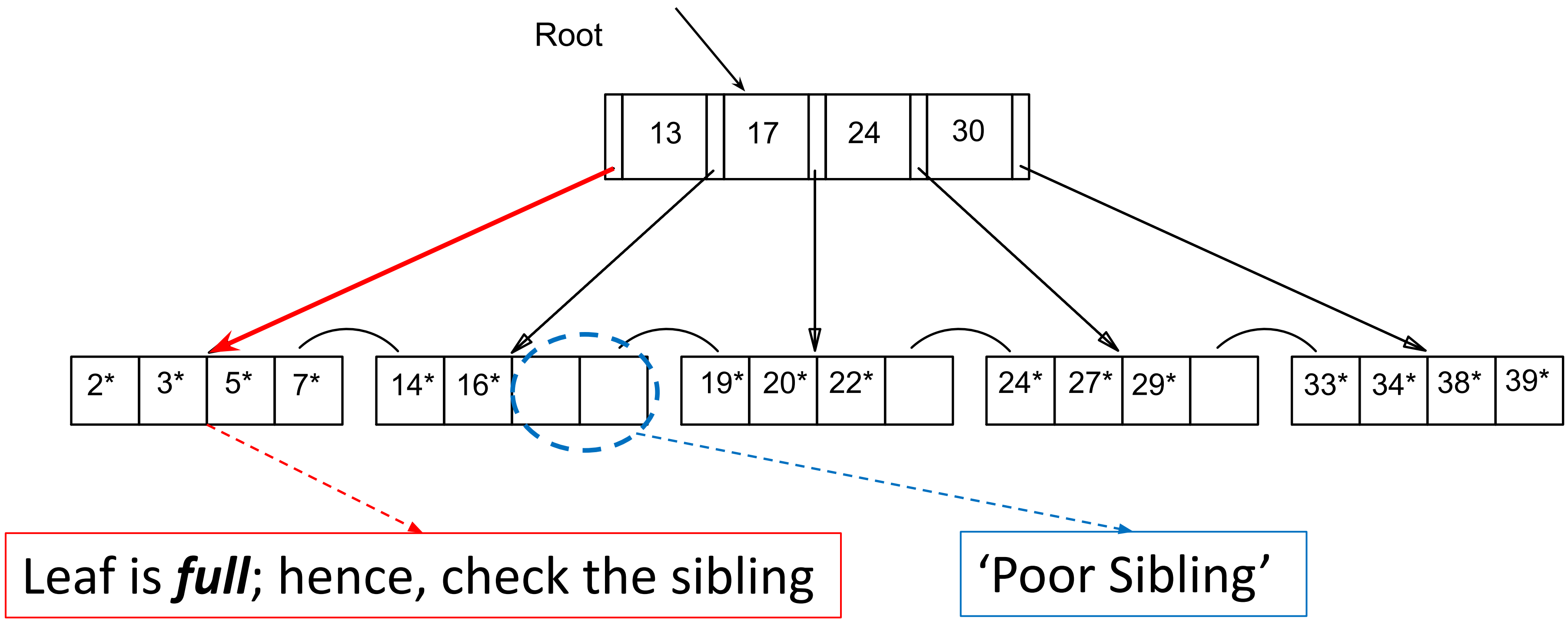


Splitting the root lead to an increase of height by 1!

What about *re-distributing* entries instead of *splitting* nodes?

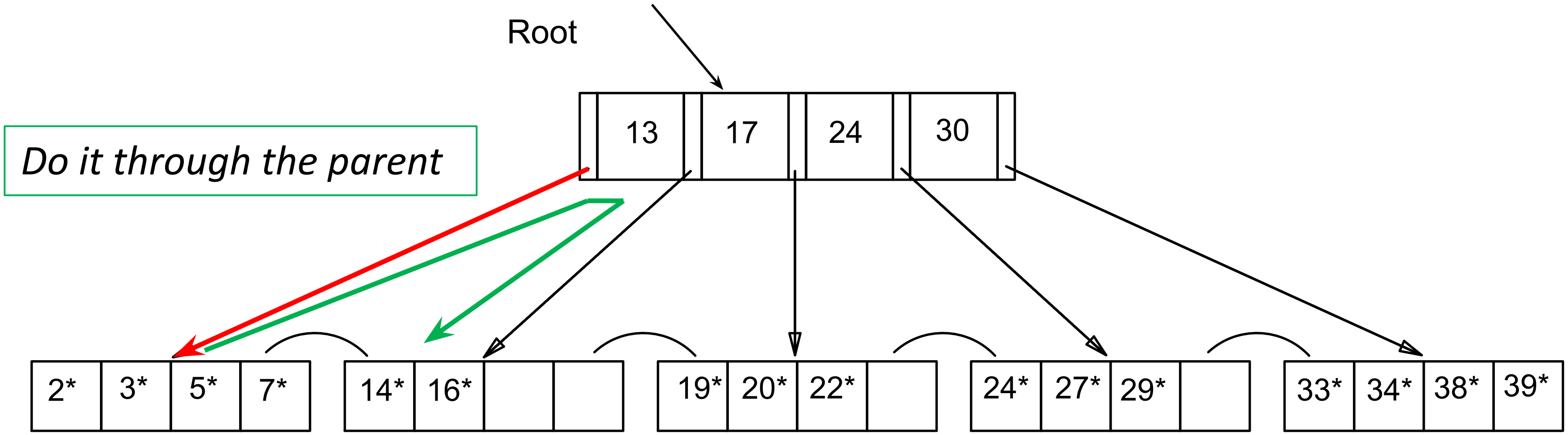
B+ Tree: Examples of Insertions

- Insert entry 8*



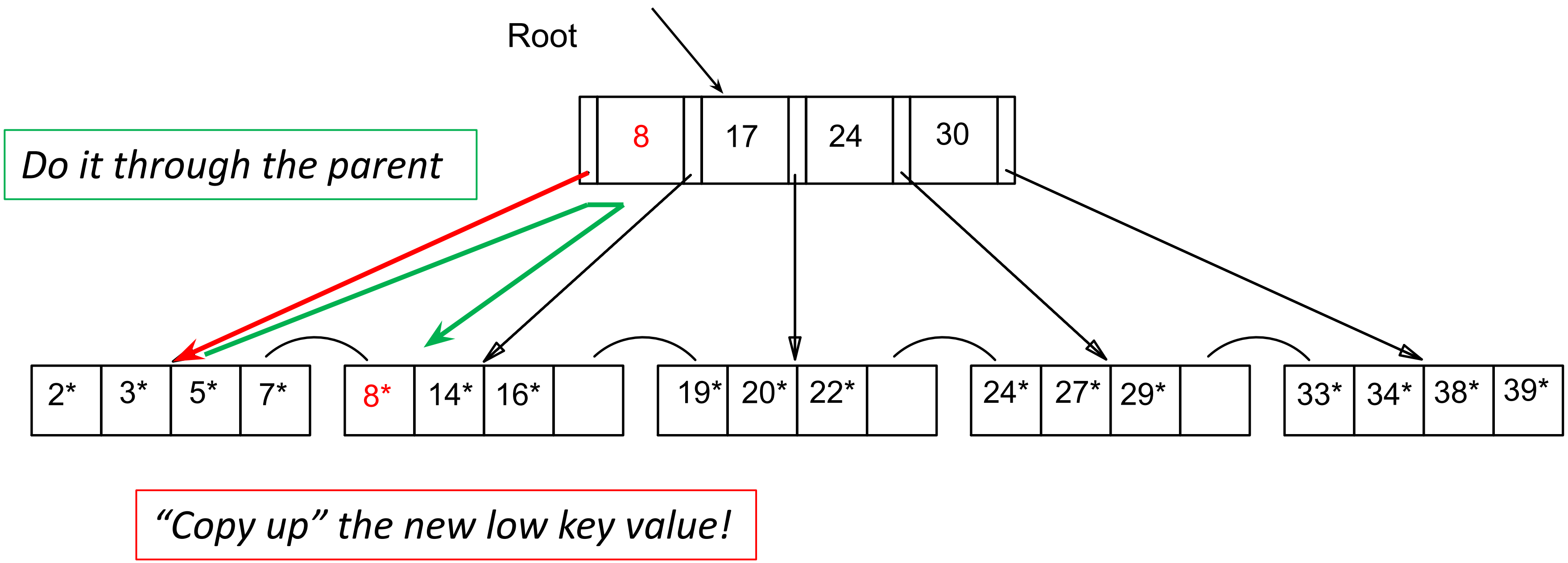
B+ Tree: Examples of Insertions

- Insert entry 8*



B+ Tree: Examples of Insertions

- Insert entry 8*



But, when to *redistribute* and when to *split*?

Splitting vs. Redistributing

- Leaf Nodes
  - Previous and next-neighbor pointers must be updated upon insertions (*if splitting is to be pursued*)
  - Hence, checking whether redistribution is possible does not increase I/O
  - Therefore, if a sibling can spare an entry, re-distribute

- Non-Leaf Nodes
  - Checking whether redistribution is possible *usually* increases I/O
  - Splitting non-leaf nodes typically pays off!

B+ Insertions: Keep in Mind

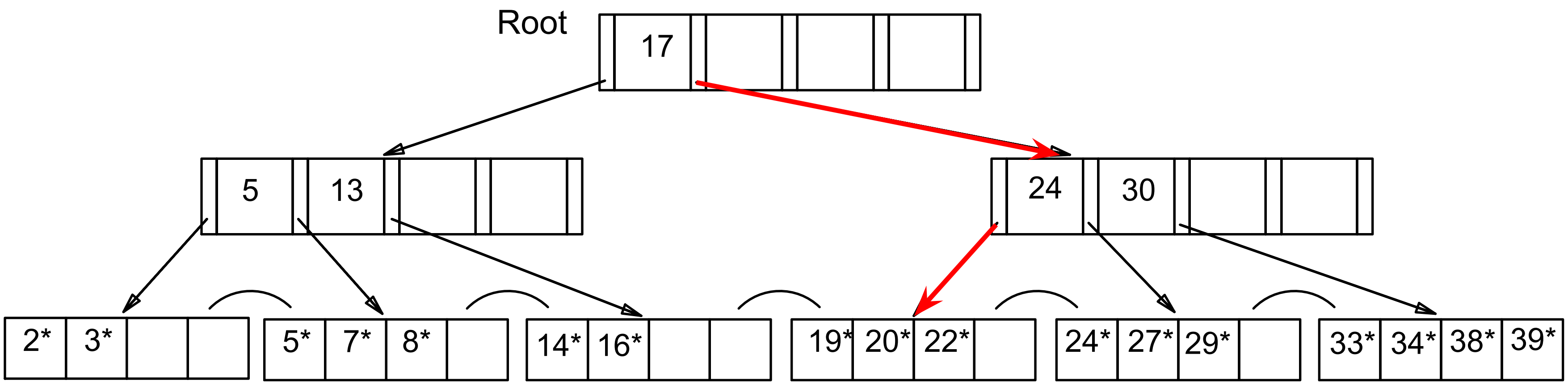
- Every data entry must appear in a leaf node; hence, “copy up” the middle key upon splitting
- When splitting index entries, simply “push up” the middle key
- Apply splitting and/or redistribution on leaf nodes
- Apply only splitting on non-leaf nodes

B+ Trees: Deleting Entries

- Start at root, find leaf ***L*** where entry belongs
- Remove the entry
  - If ***L*** is at least half-full, *done!*
  - If ***L*** *underflows*
    - Try to re-distribute (i.e., borrow from a "rich sibling" and "copy up" its *lowest key*)
    - If re-distribution fails, *merge* ***L*** and a "poor sibling"
      - Update parent
      - And possibly merge, recursively

B+ Tree: Examples of Deletions

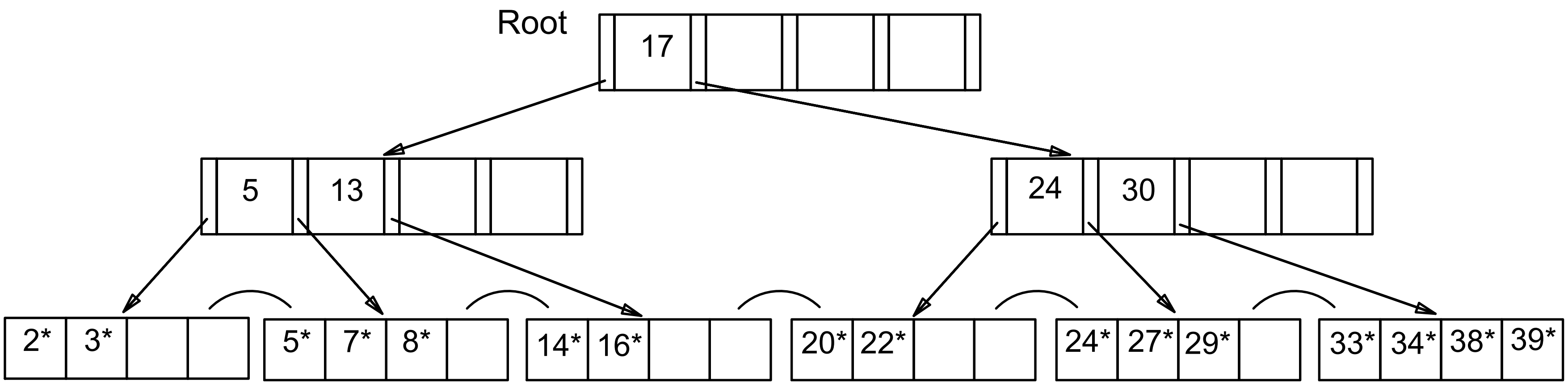
- Delete 19*



Removing 19* does not cause an underflow

B+ Tree: Examples of Deletions

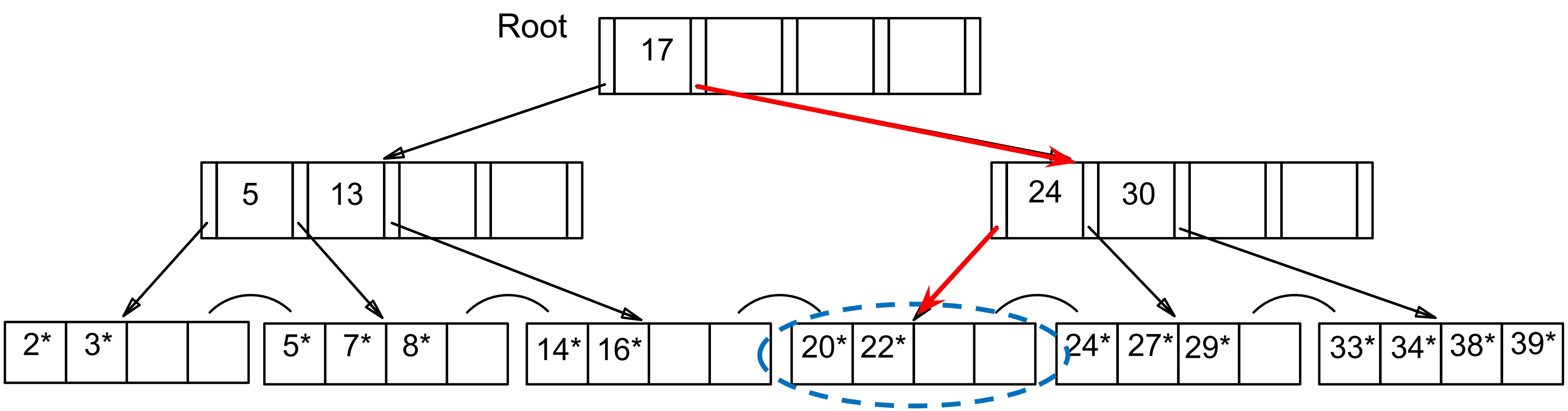
- Delete 19*



FINAL TREE!

B+ Tree: Examples of Deletions

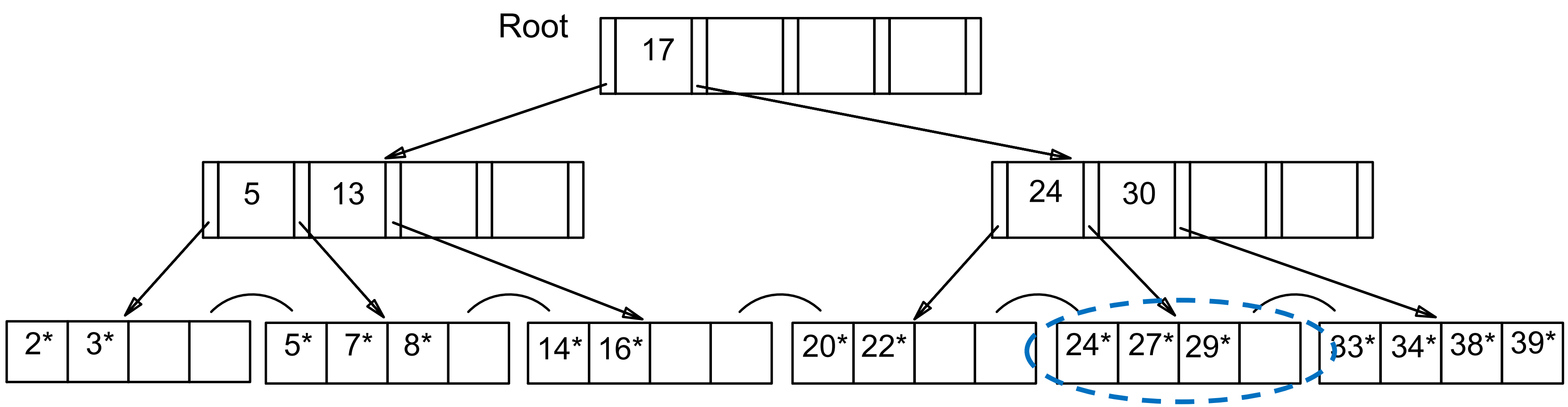
- Delete 20*



Deleting 20* causes an underflow; hence, check a sibling for redistribution

B+ Tree: Examples of Deletions

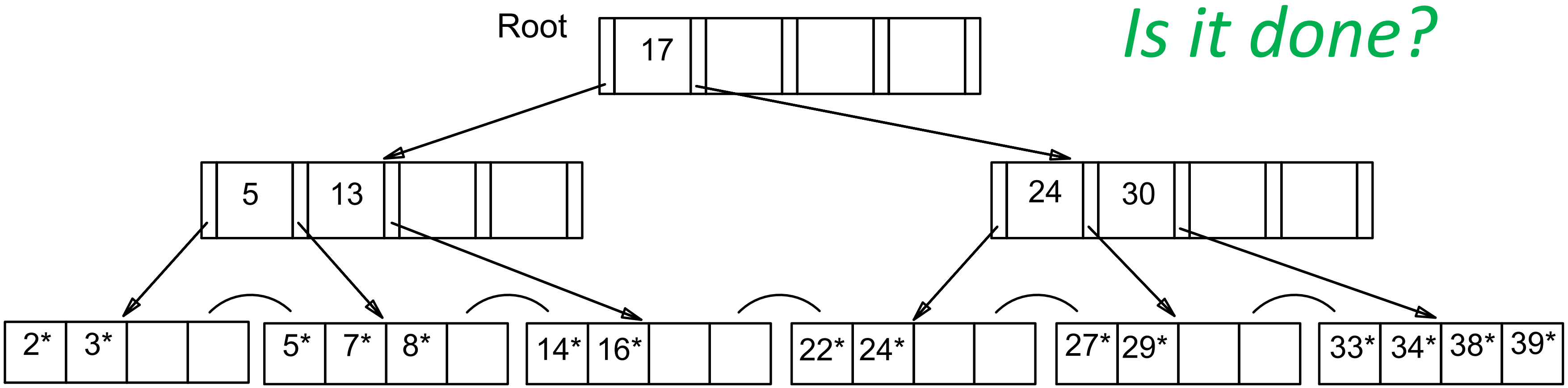
- Delete 20*



The sibling is ‘rich’ (i.e., can lend an entry); hence, remove 20* and redistribute!

B+ Tree: Examples of Deletions

- Delete 20*

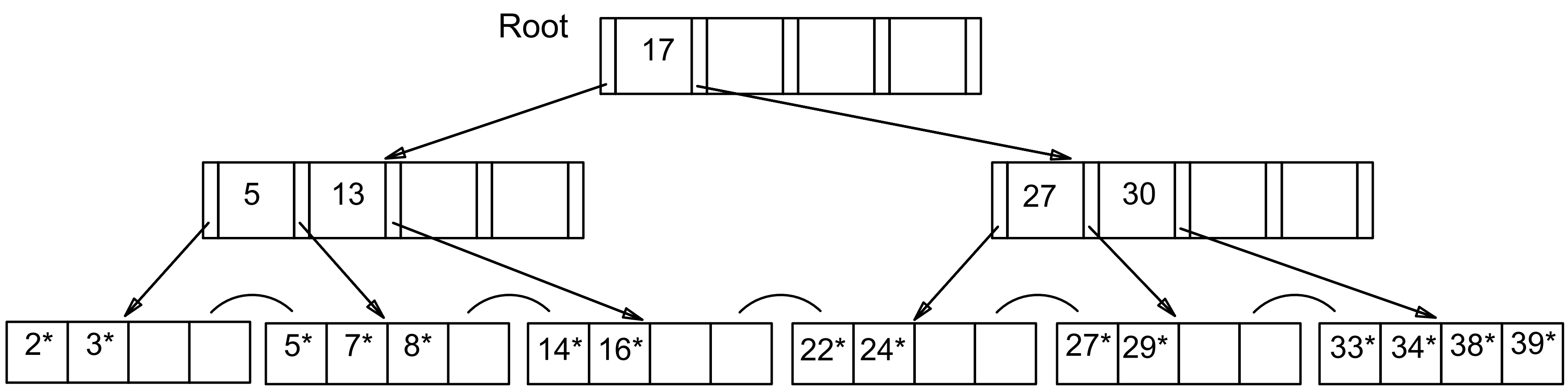


Is it done?

"Copy up" 27*, the lowest value in the leaf from which we borrowed 24*

B+ Tree: Examples of Deletions

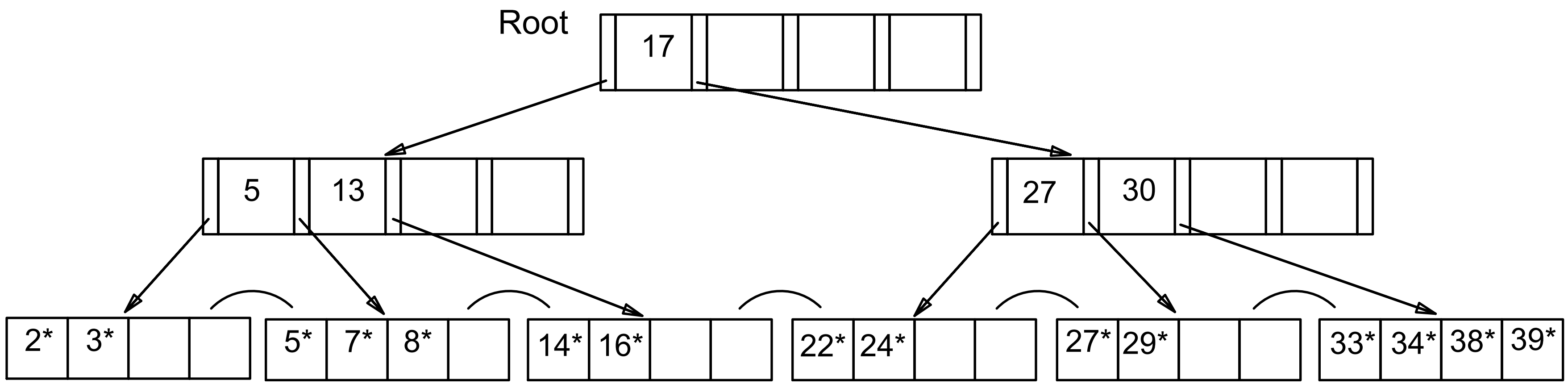
- Delete 20*



"Copy up" 27*, the lowest value in the leaf from which we borrowed 24*

B+ Tree: Examples of Deletions

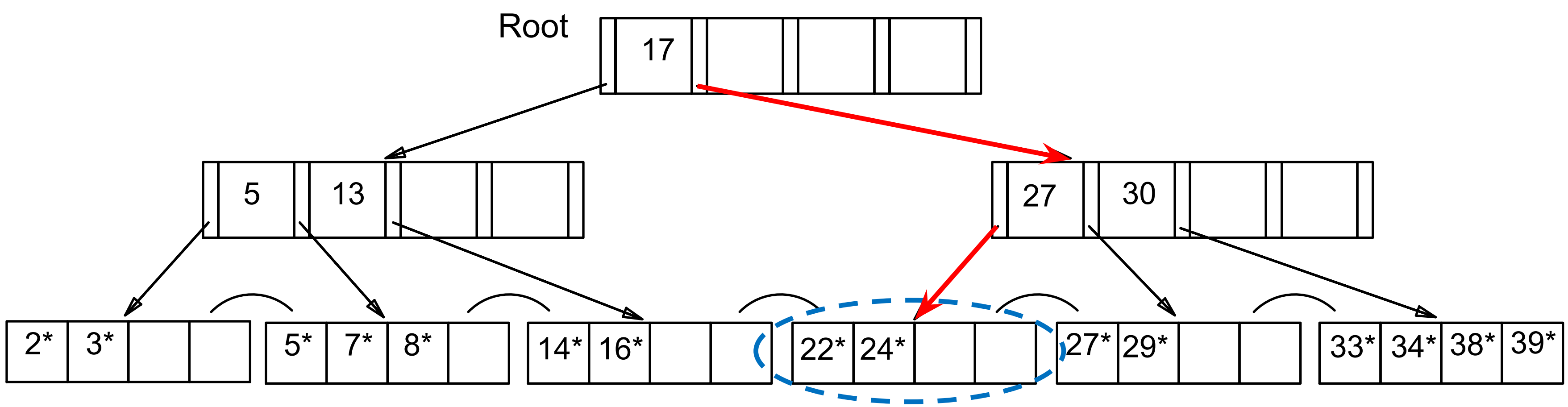
- Delete 20*



FINAL TREE!

B+ Tree: Examples of Deletions

- Delete 24*

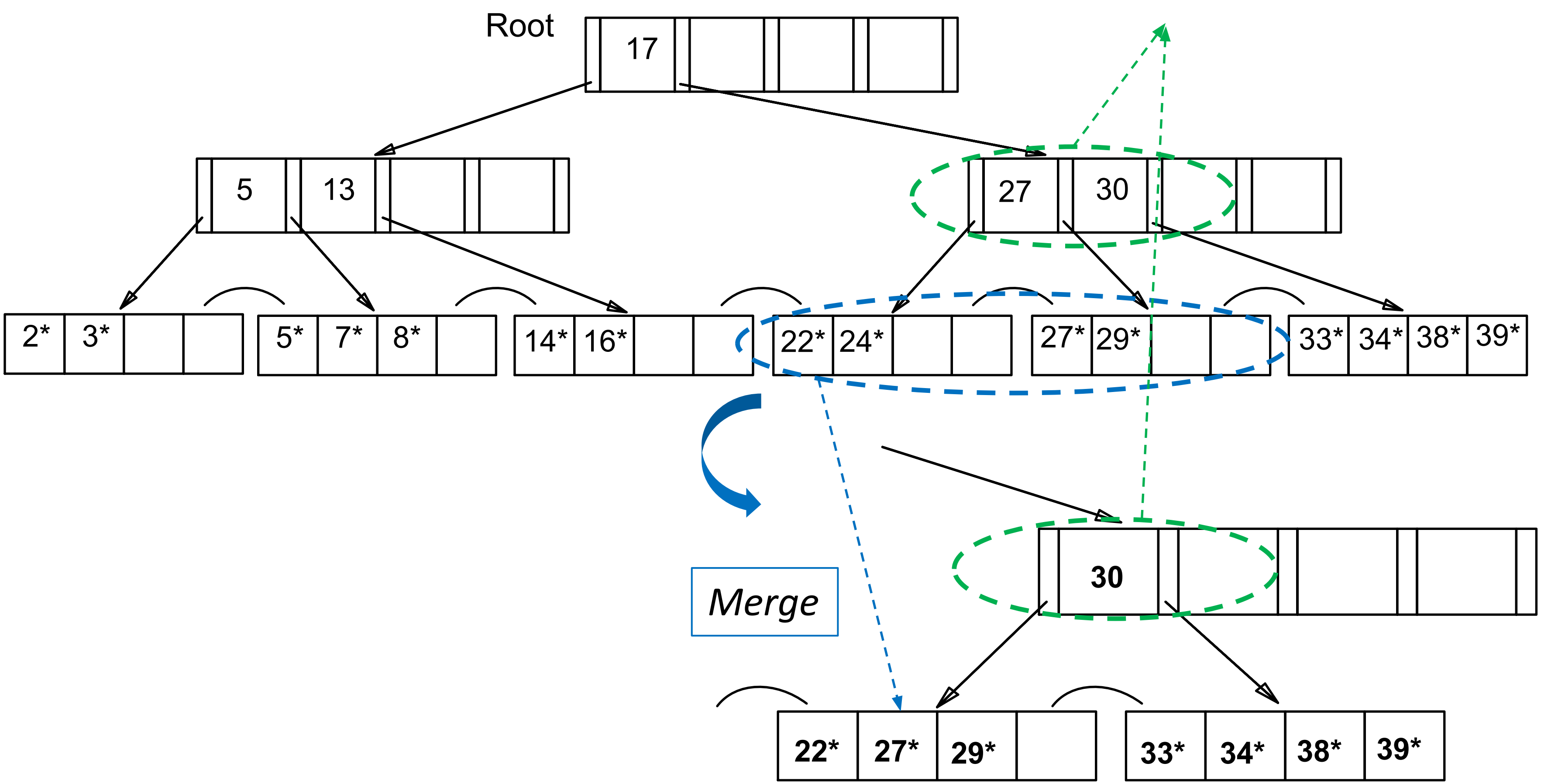


The affected leaf will contain only 1 entry and the sibling cannot lend any entry (i.e., redistribution is not applicable); hence, ***merge***!

B+ Tree: Examples of Deletions

- Delete 24*

"Toss" 27 because the page that it was pointing to does not exist anymore!

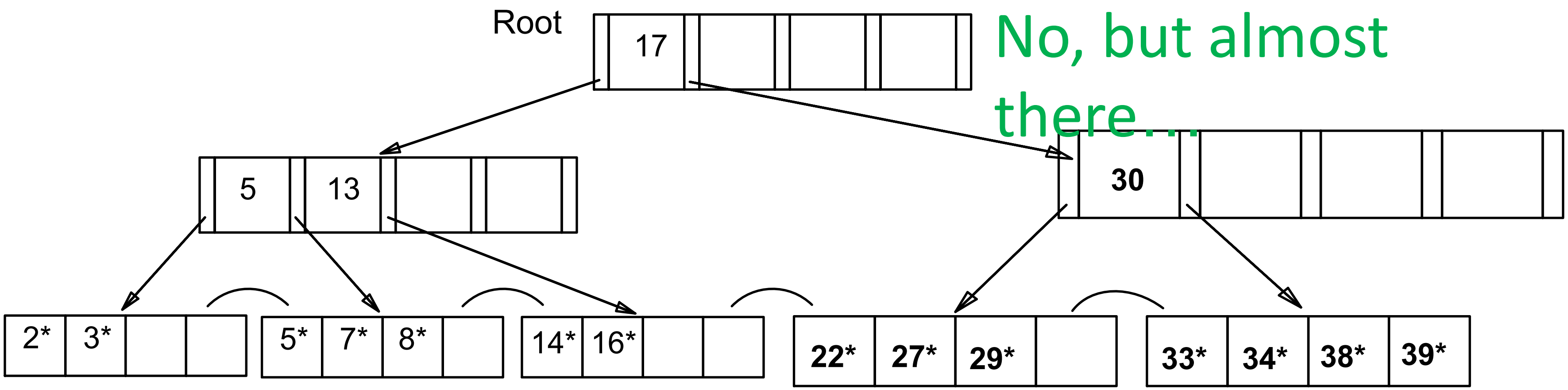


B+ Tree: Examples of Deletions

- Delete 24*

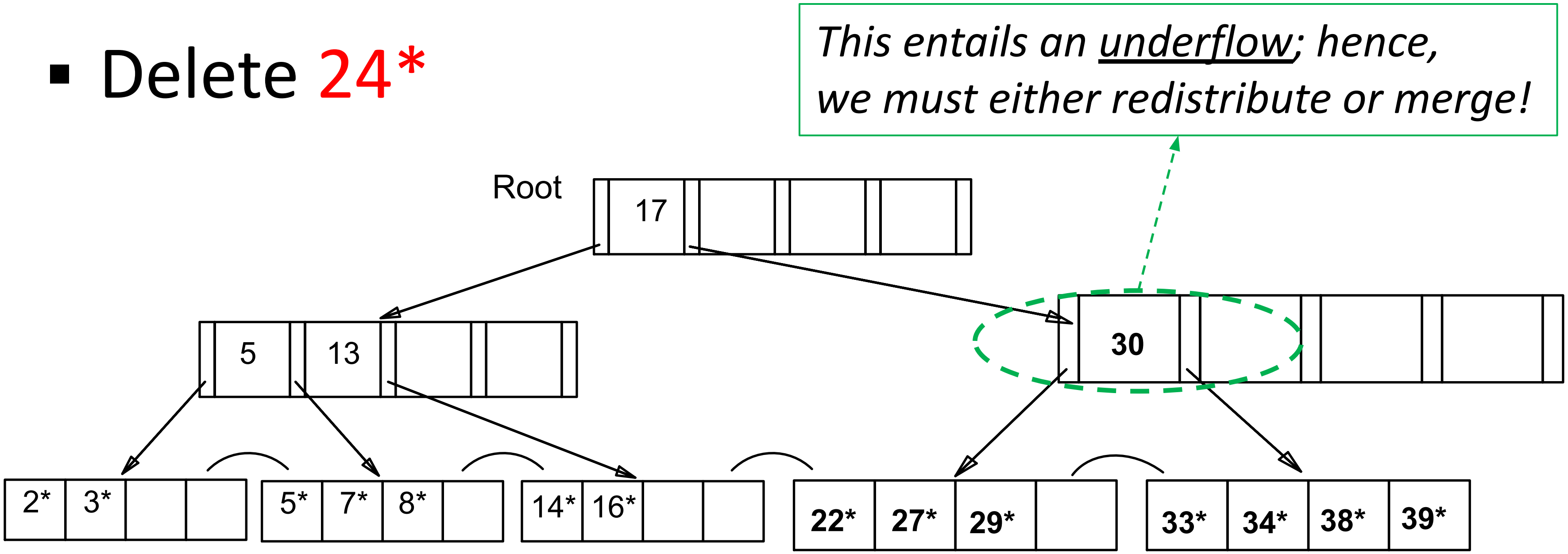
Is it done?

No, but almost there…



B+ Tree: Examples of Deletions

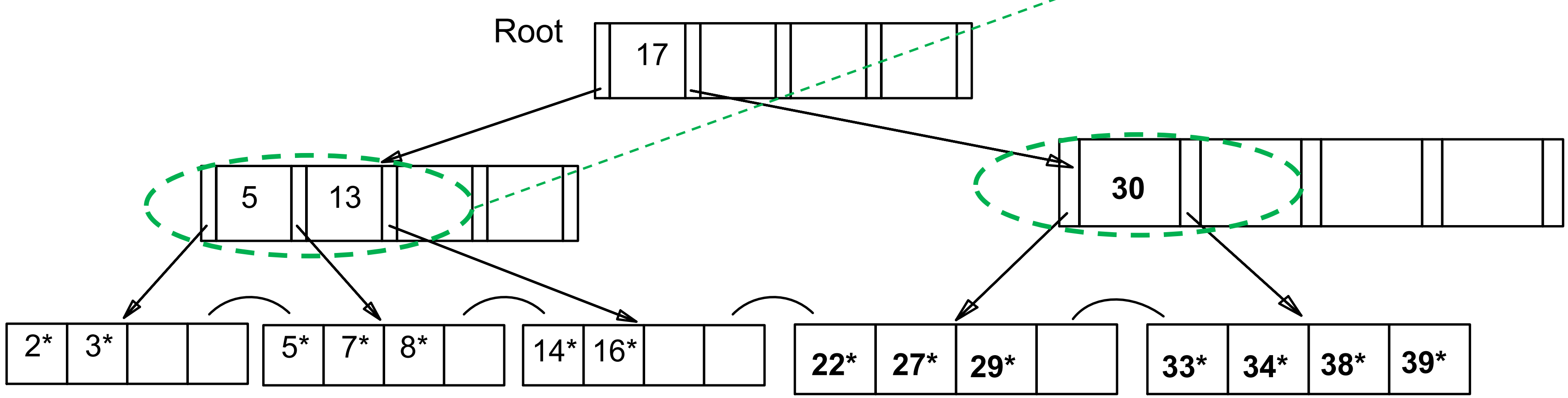
- Delete 24*



B+ Tree: Examples of Deletions

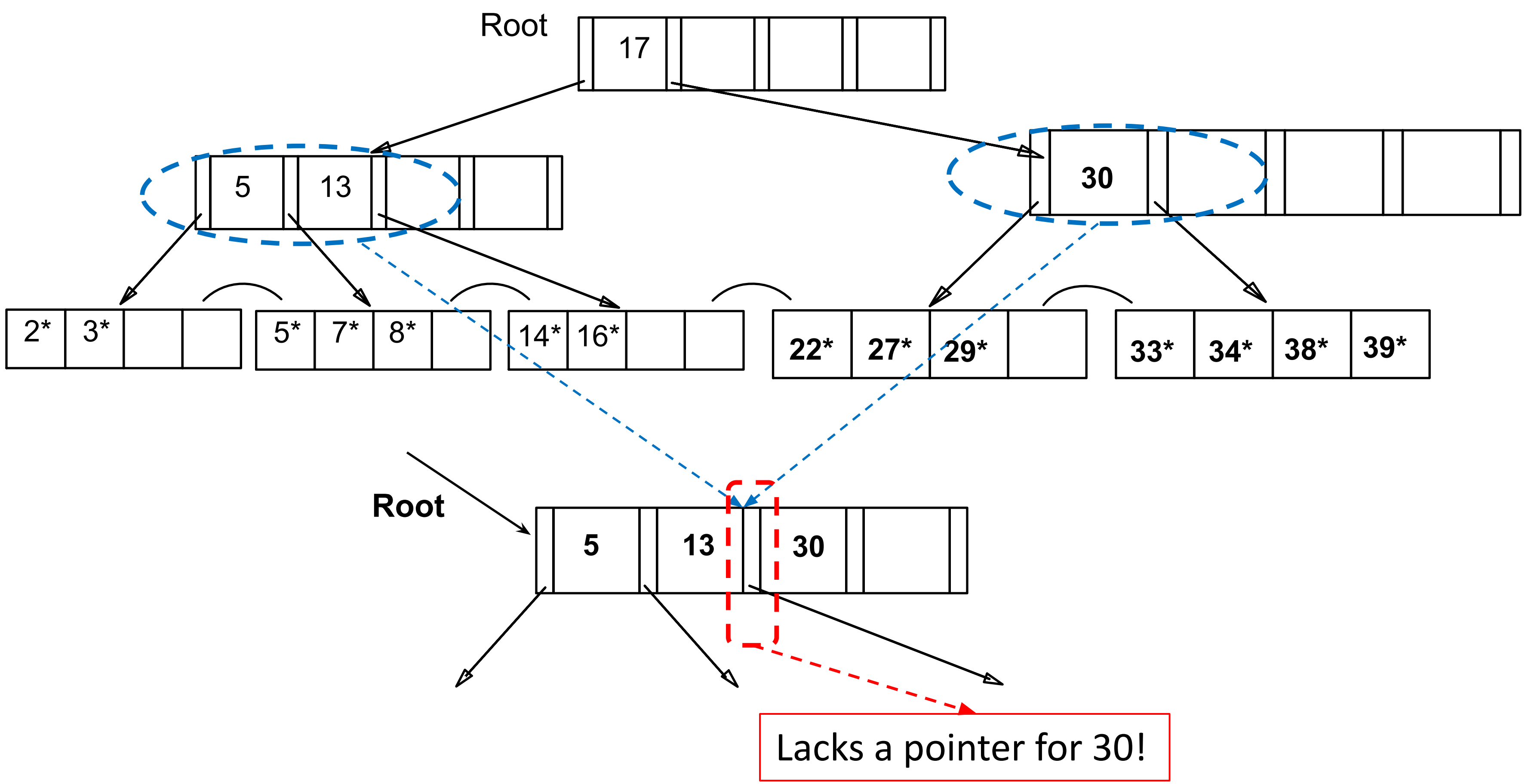
- Delete 24*

The sibling is "poor" (i.e., redistribution is not applicable)*; hence, merge!*



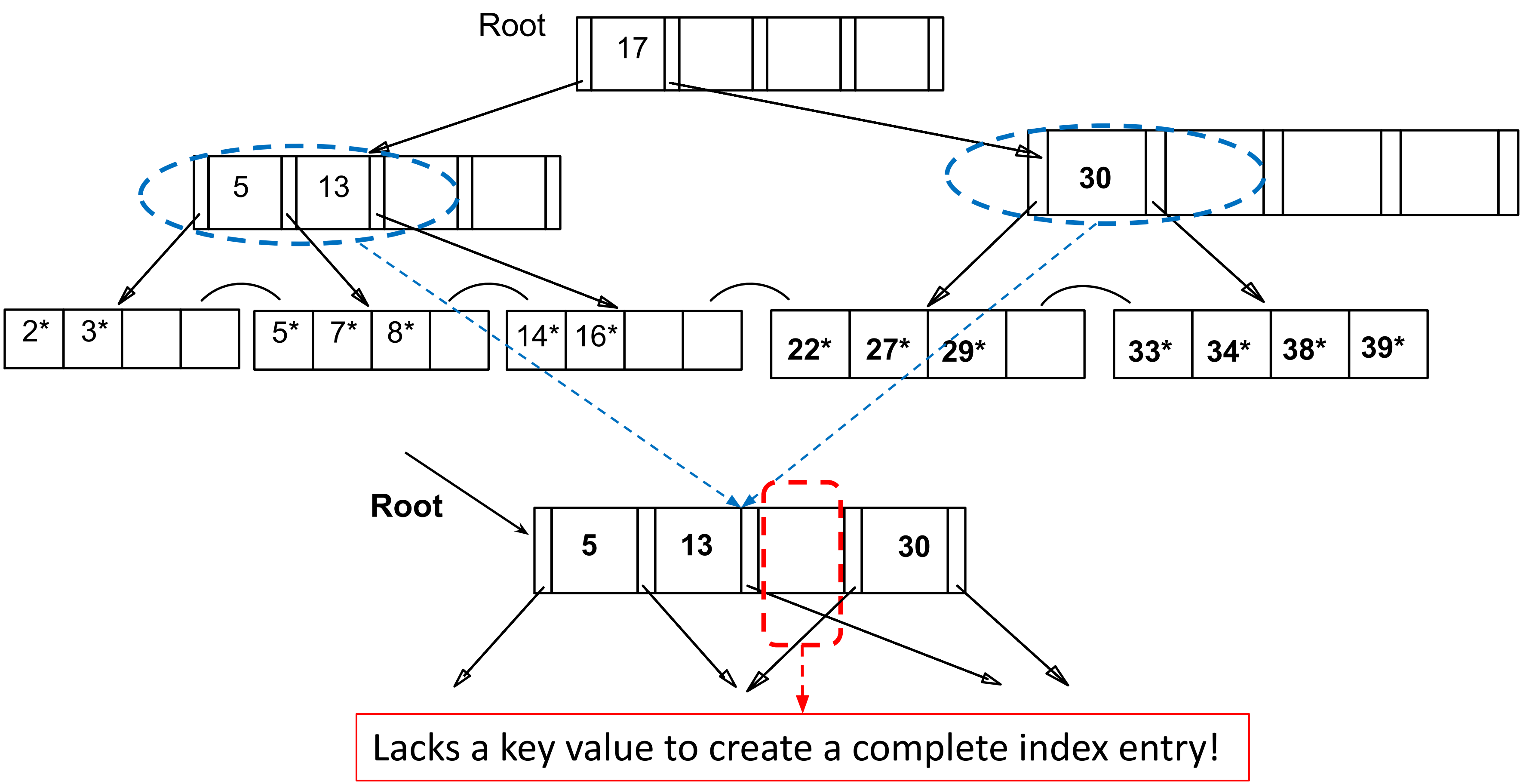
B+ Tree: Examples of Deletions

- Delete 24*



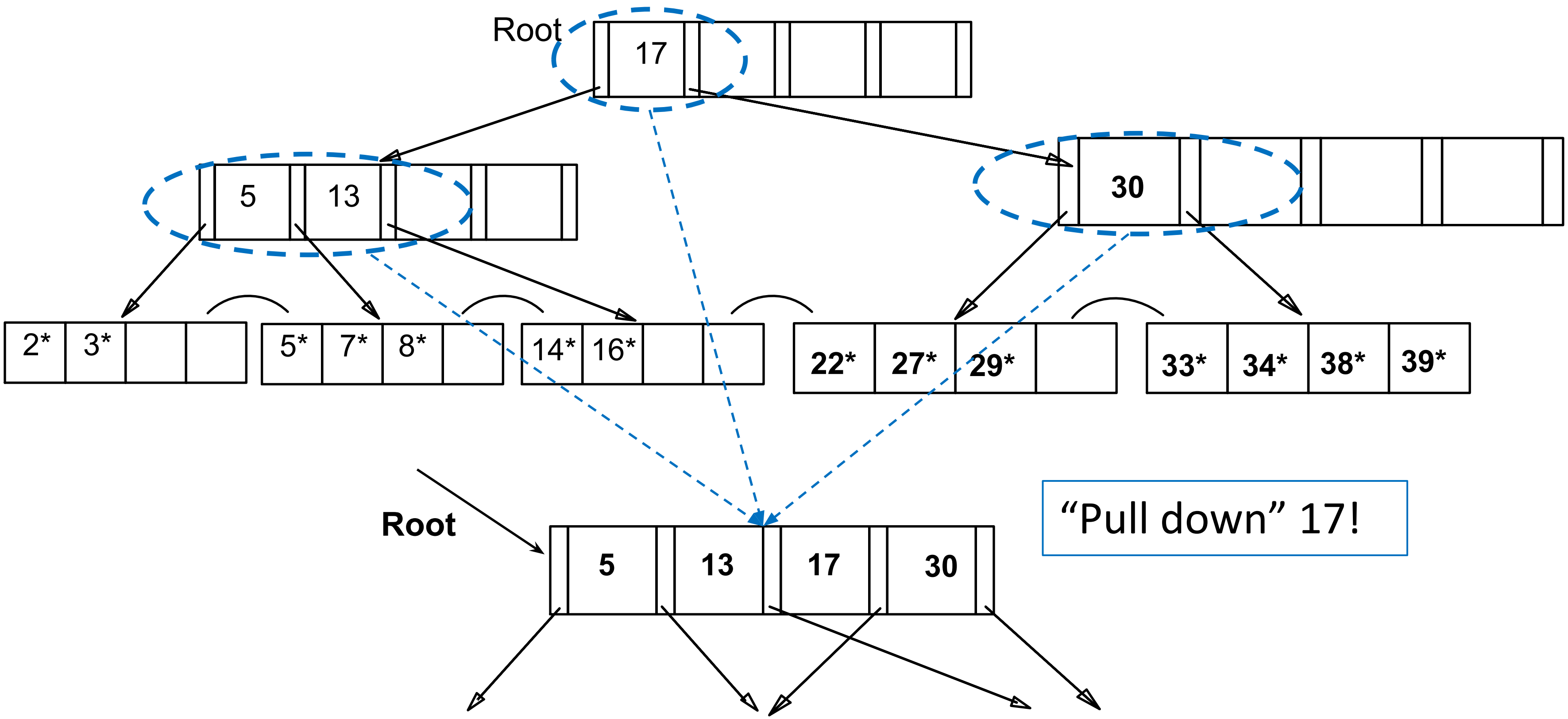
B+ Tree: Examples of Deletions

- Delete 24*



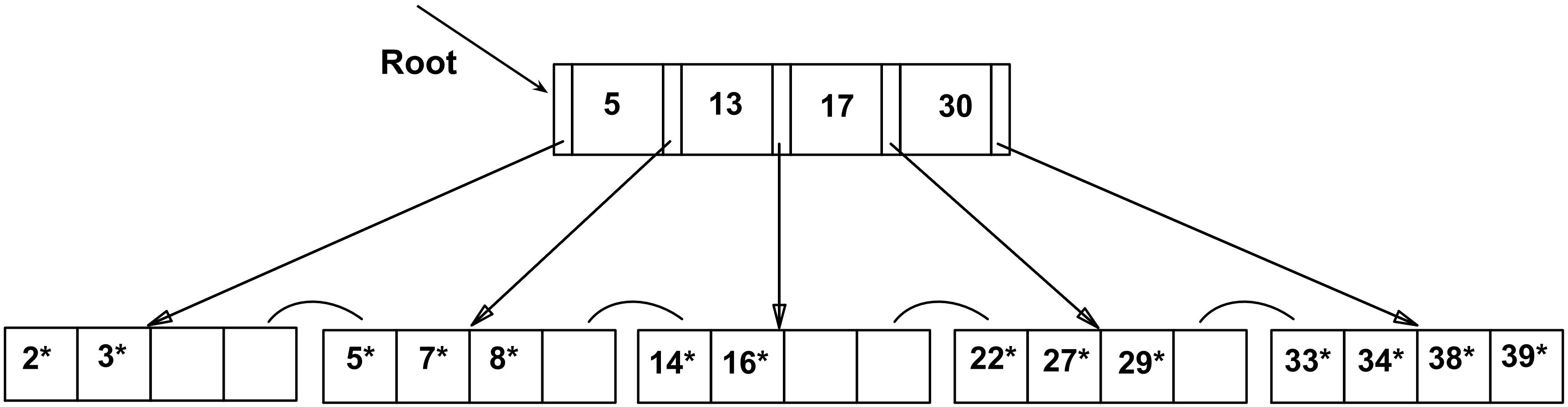
B+ Tree: Examples of Deletions

- Delete 24*



B+ Tree: Examples of Deletions

- Delete 24*



FINAL TREE!

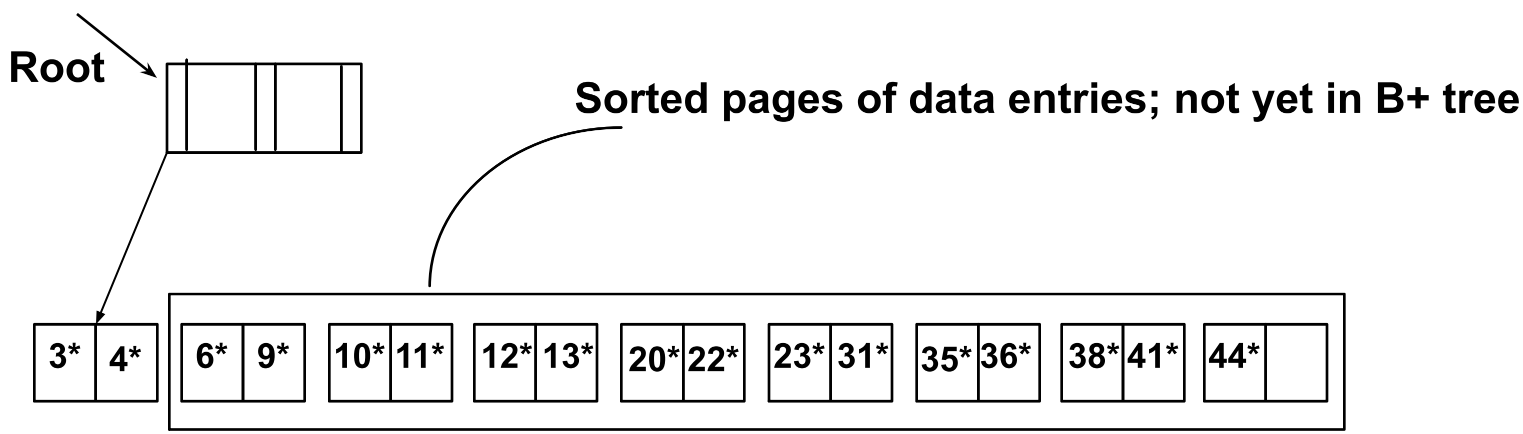
B+ Tree: Bulk Loading

- Assume a collection of data records with an ***existing*** B+ tree index on it
  - How to add a new record to it?
    - Use the B+ tree insert() function

- What if we have a collection of data records for which we want to create a B+ tree index? (i.e., we want to *bulk load* the B+ tree)
  - Starting with an empty tree and using the insert() function for each data record, *one at a time*, is expensive!
    - This is because for each entry we would require starting again from the root and going down to the appropriate leaf page

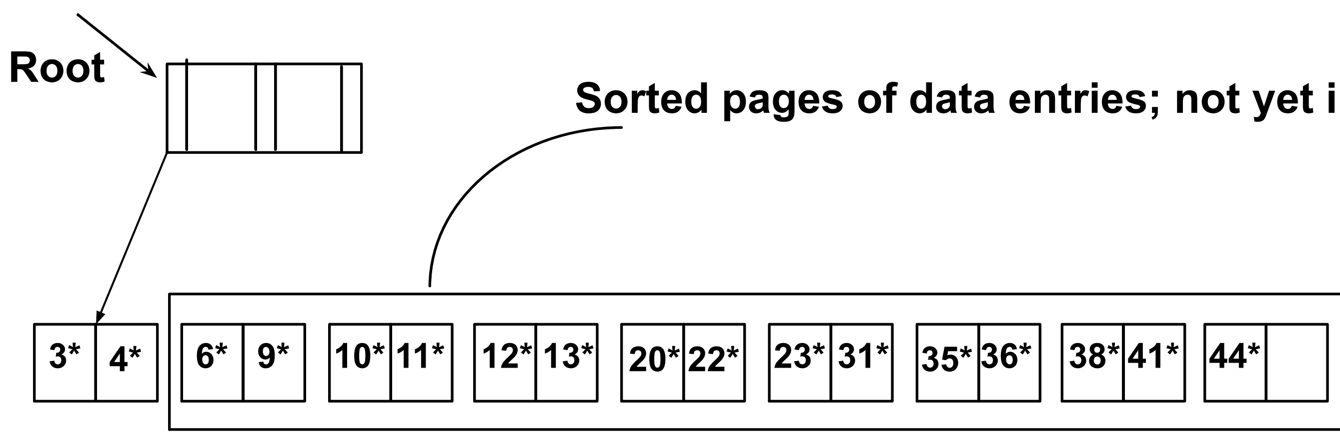
B+ Tree: Bulk Loading

- What to do?
  - *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page



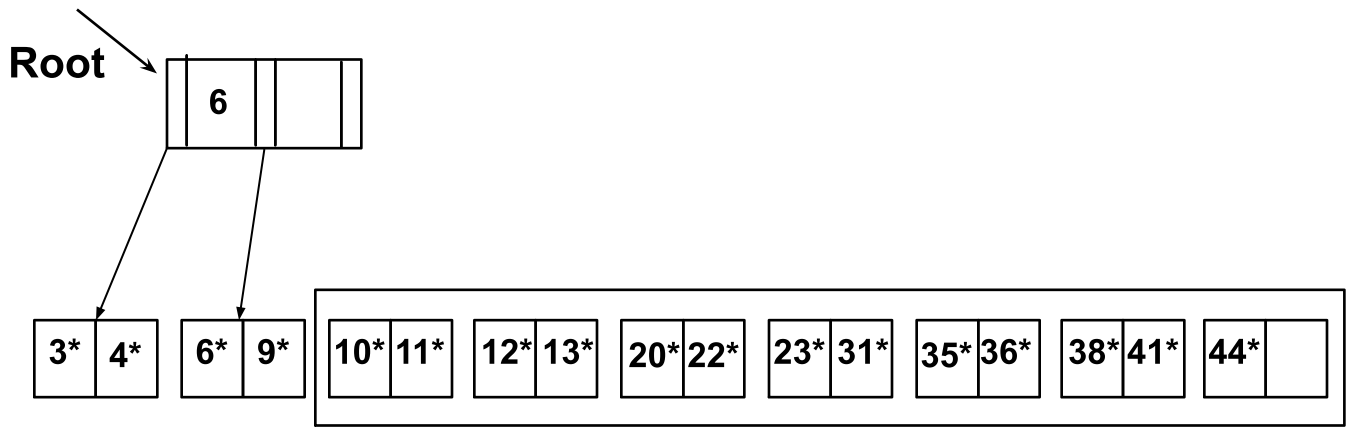
B+ Tree: Bulk Loading

- What to do?
  - Add one entry to the root page for each subsequent page of the sorted data entries *(i.e., <lowest key value on page, pointer to the page>)*



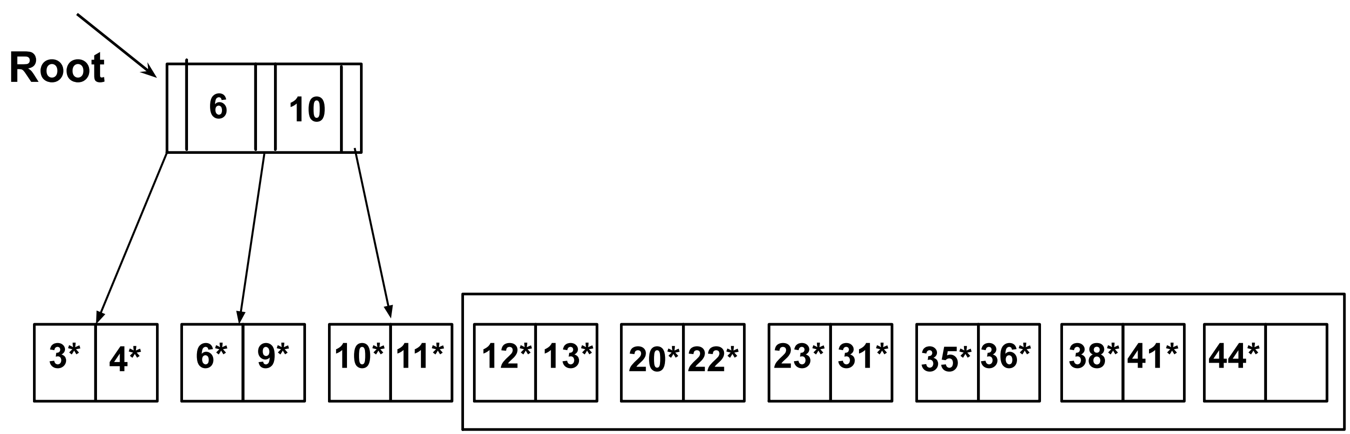
B+ Tree: Bulk Loading

- What to do?
  - Add one entry to the root page for each subsequent page of the sorted data entries *(i.e., <lowest key value on page, pointer to the page>)*



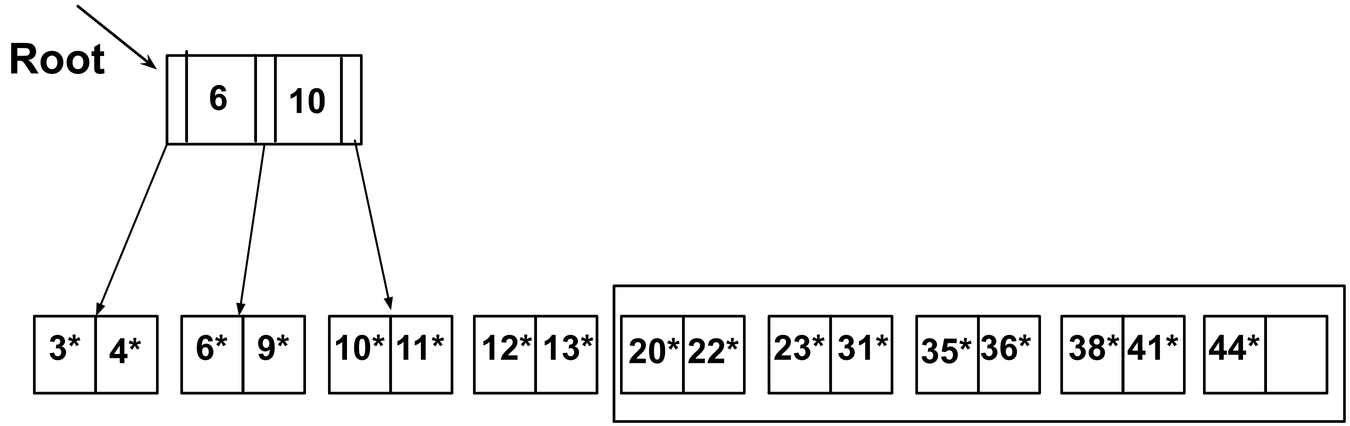
B+ Tree: Bulk Loading

- What to do?
  - Add one entry to the root page for each subsequent page of the sorted data entries *(i.e., <lowest key value on page, pointer to the page>)*



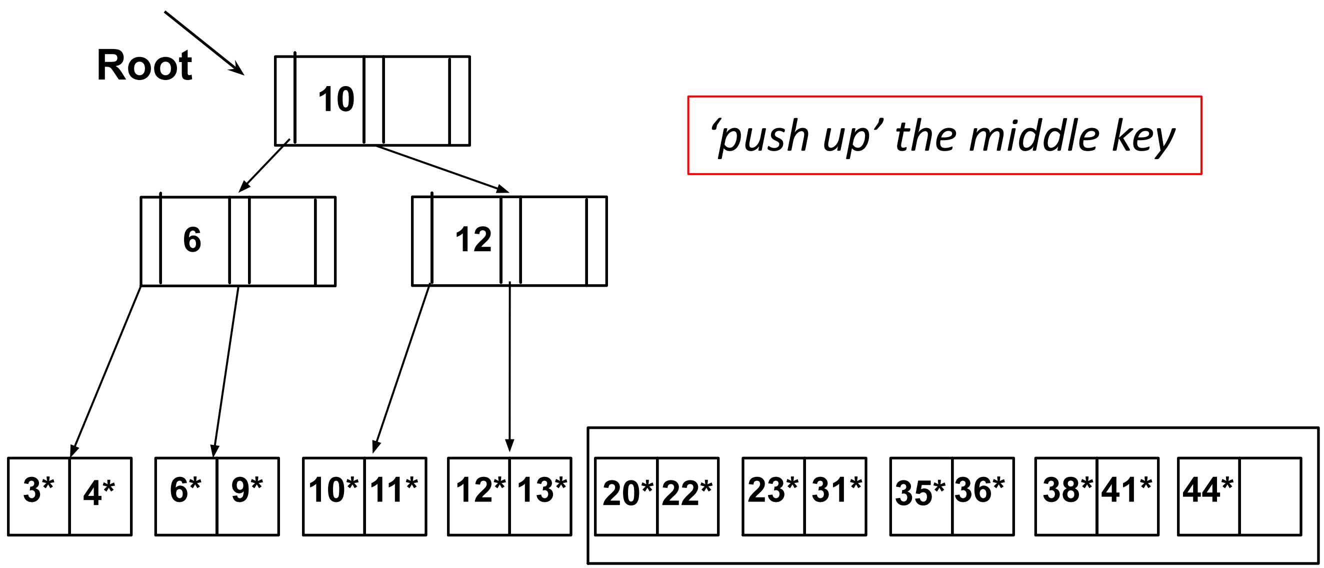
B+ Tree: Bulk Loading

- What to do?
  - Split the root and create a new root page



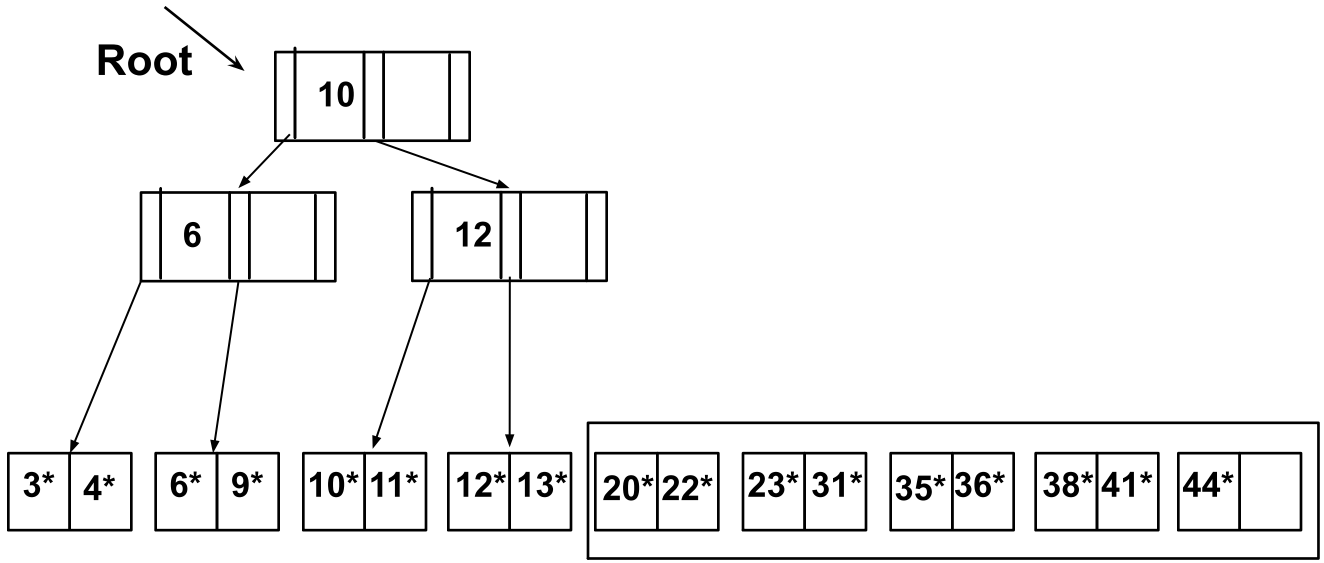
B+ Tree: Bulk Loading

- What to do?
  - Split the root and create a new root page



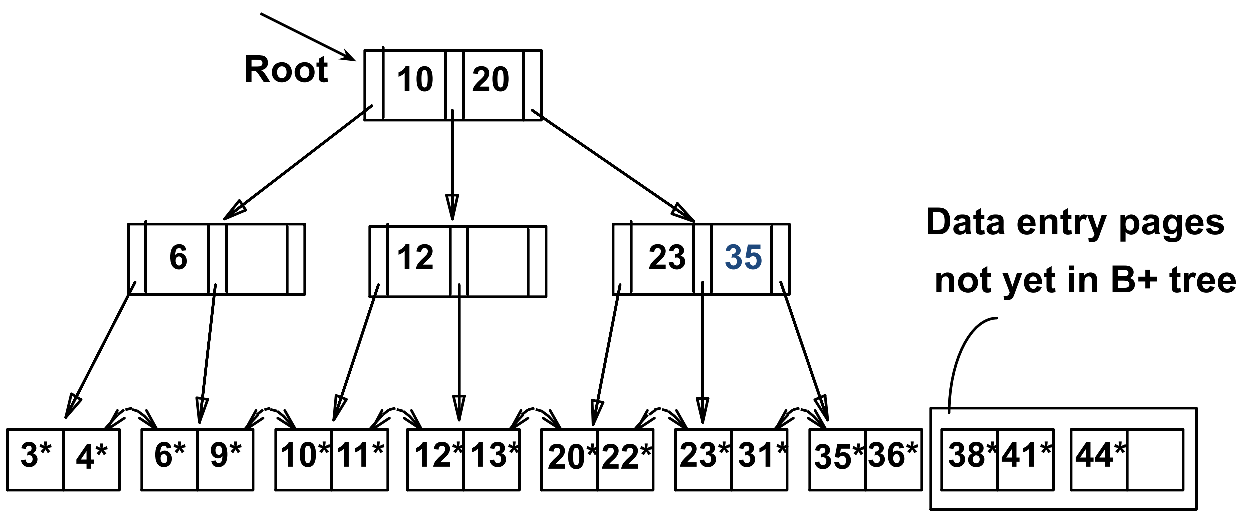
B+ Tree: Bulk Loading

- What to do?
  - Continue by inserting entries into the right-most index page just above the leaf page; split when fills up



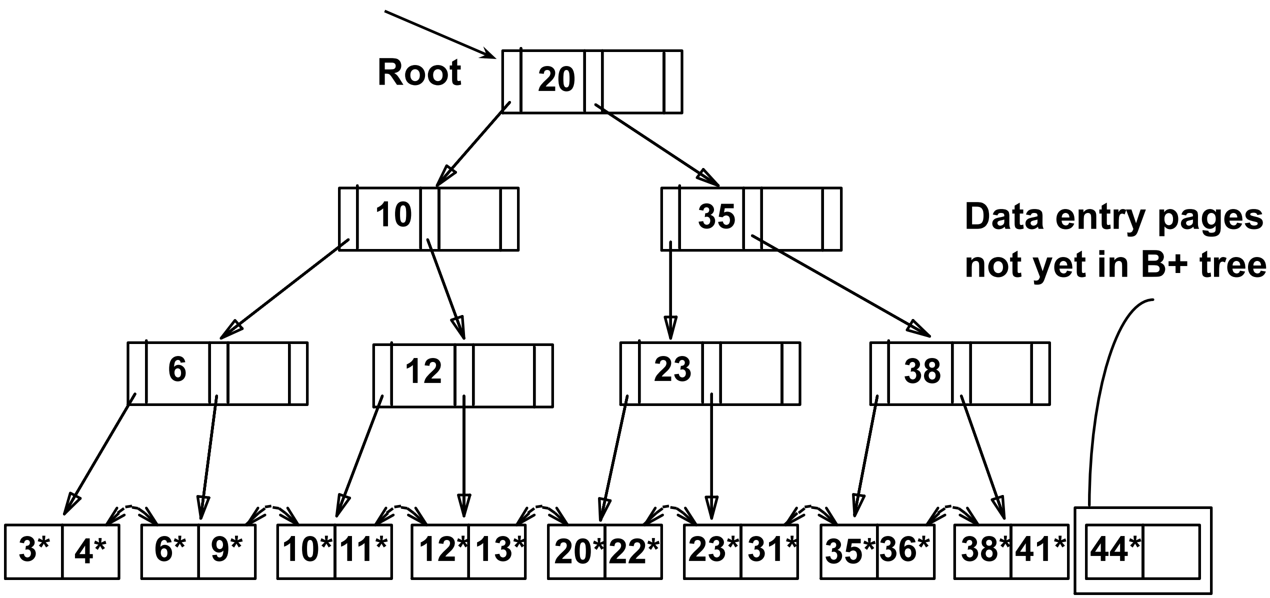
B+ Tree: Bulk Loading

- What to do?
  - Continue by inserting entries into the right-most index page just above the leaf page; split when fills up



B+ Tree: Bulk Loading

- What to do?
  - Continue by inserting entries into the right-most index page just above the leaf page; split when fills up



Hashing

Hash-Based Indexing

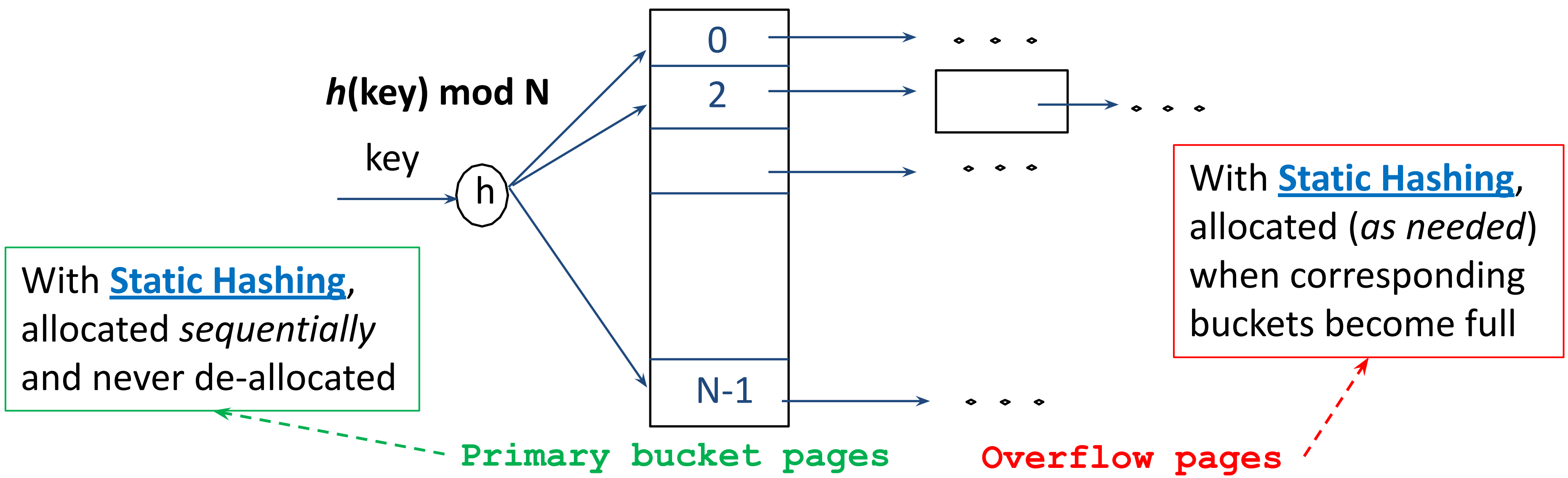
- What indexing technique can we use to support *range searches* (e.g., “Find s_name where gpa >= 3.0)?
  - Tree-Based Indexing

- What about *equality selections* (e.g., “Find s_name where sid = 102”?
  - Tree-Based Indexing
  - Hash-Based Indexing (*cannot support range searches!*)

- Hash-based indexing, however, proves to be very useful in implementing relational operators (e.g., joins)

Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- A hash function ***h*** is used to map keys into a range of *bucket numbers*
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

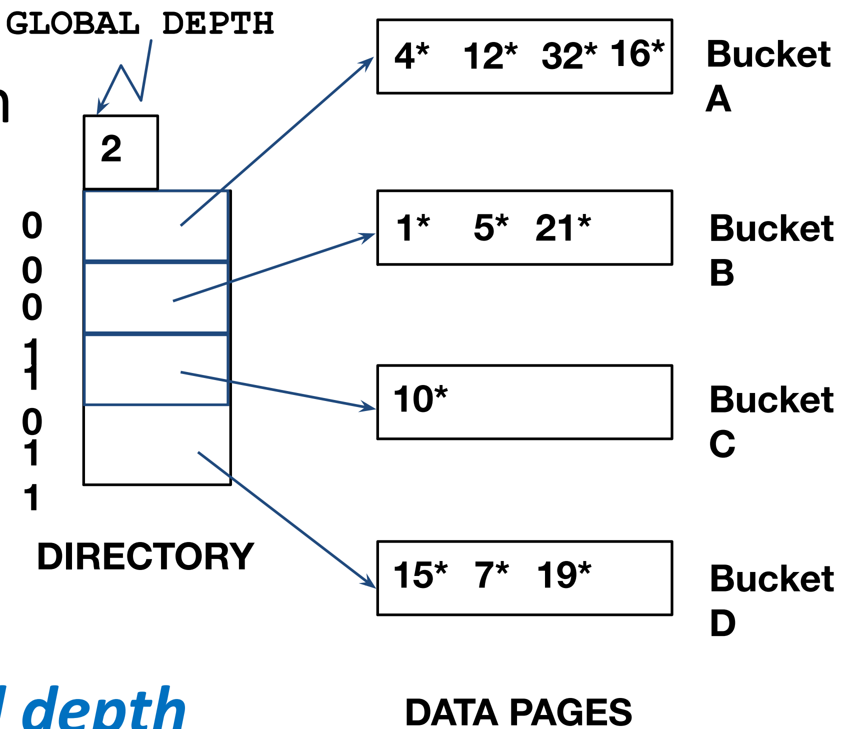


Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Extendible Hashing

- Extendible Hashing uses a directory of pointers to buckets
- The result of applying a hash function ***h*** is treated as a *binary number* and the last ***d*** bits are interpreted as an offset into the directory

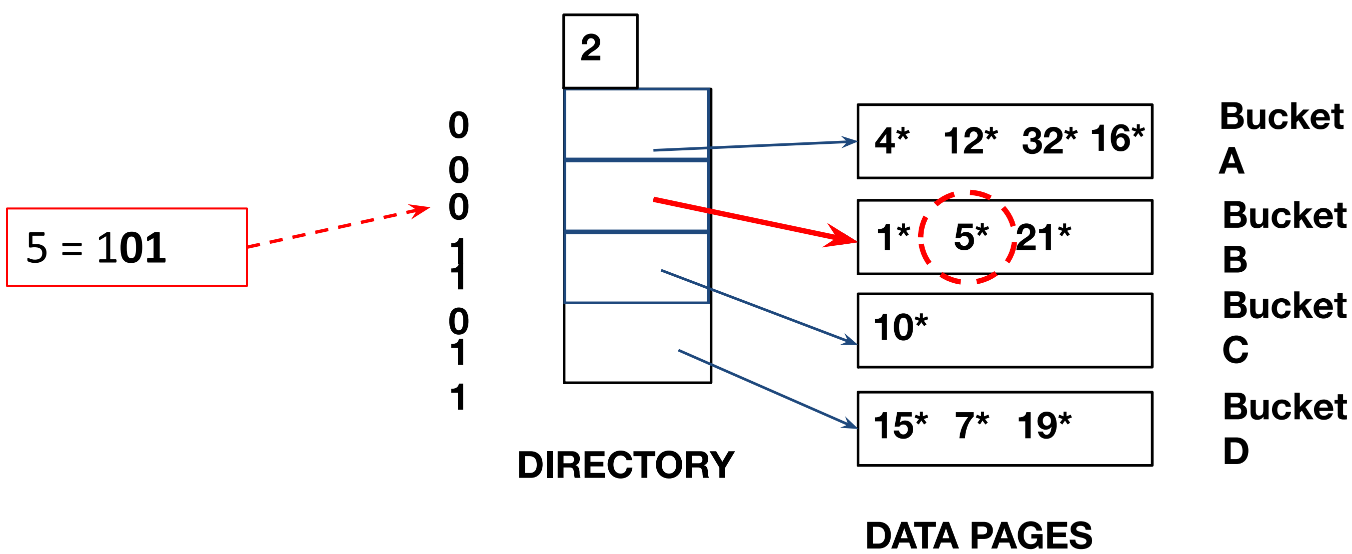


- ***d*** is referred to as the ***global depth*** of the hash file and is kept as part of the header of the file

Extendible Hashing: Searching for Entries

- To search for a data entry, apply a hash function ***h*** to the key and take the last ***d*** bits of its binary representation to get the bucket number

- Example: search for 5*



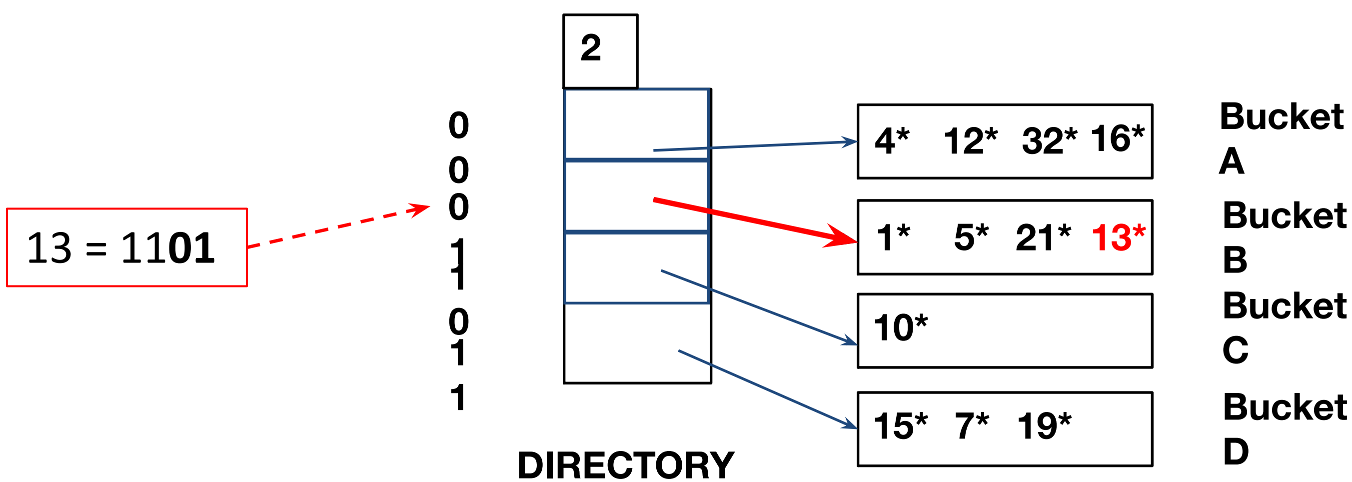
Extendible Hashing: Inserting Entries

- An entry can be inserted as follows:
  - Find the appropriate bucket (*as in search*)
  - Split the bucket *if full* and *redistribute* contents (including the new entry to be inserted) across the old bucket and its ***"split image"***
  - Double the directory *if necessary*
  - Insert the given entry

Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

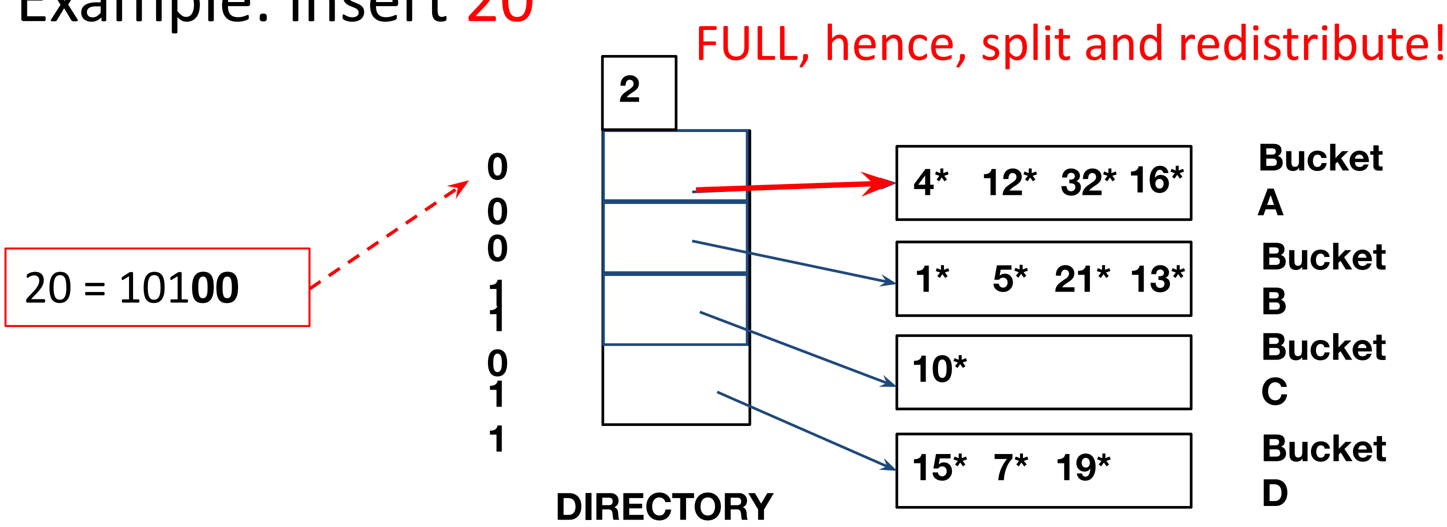
- Example: insert 13*



Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

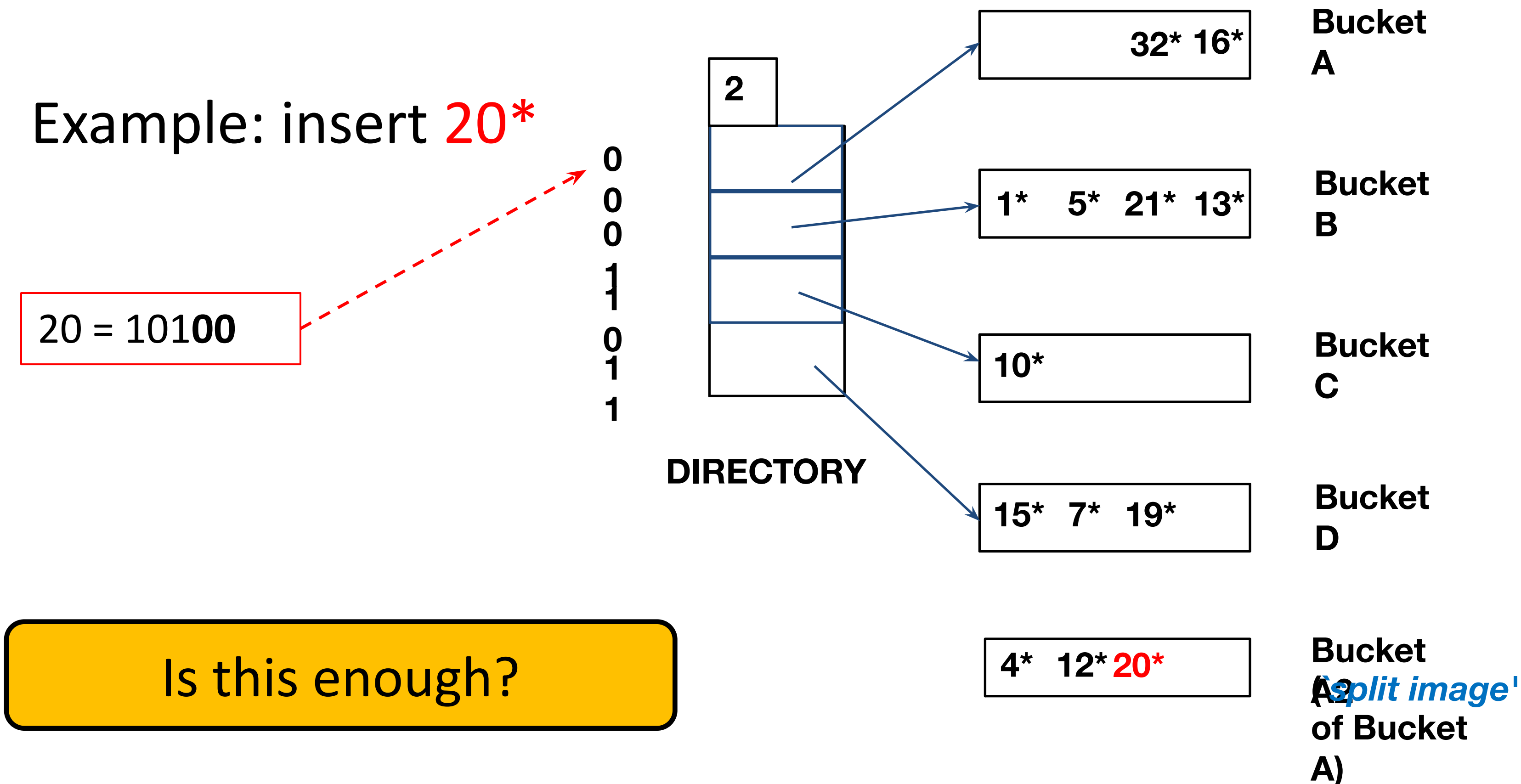
- Example: insert 20*



Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*

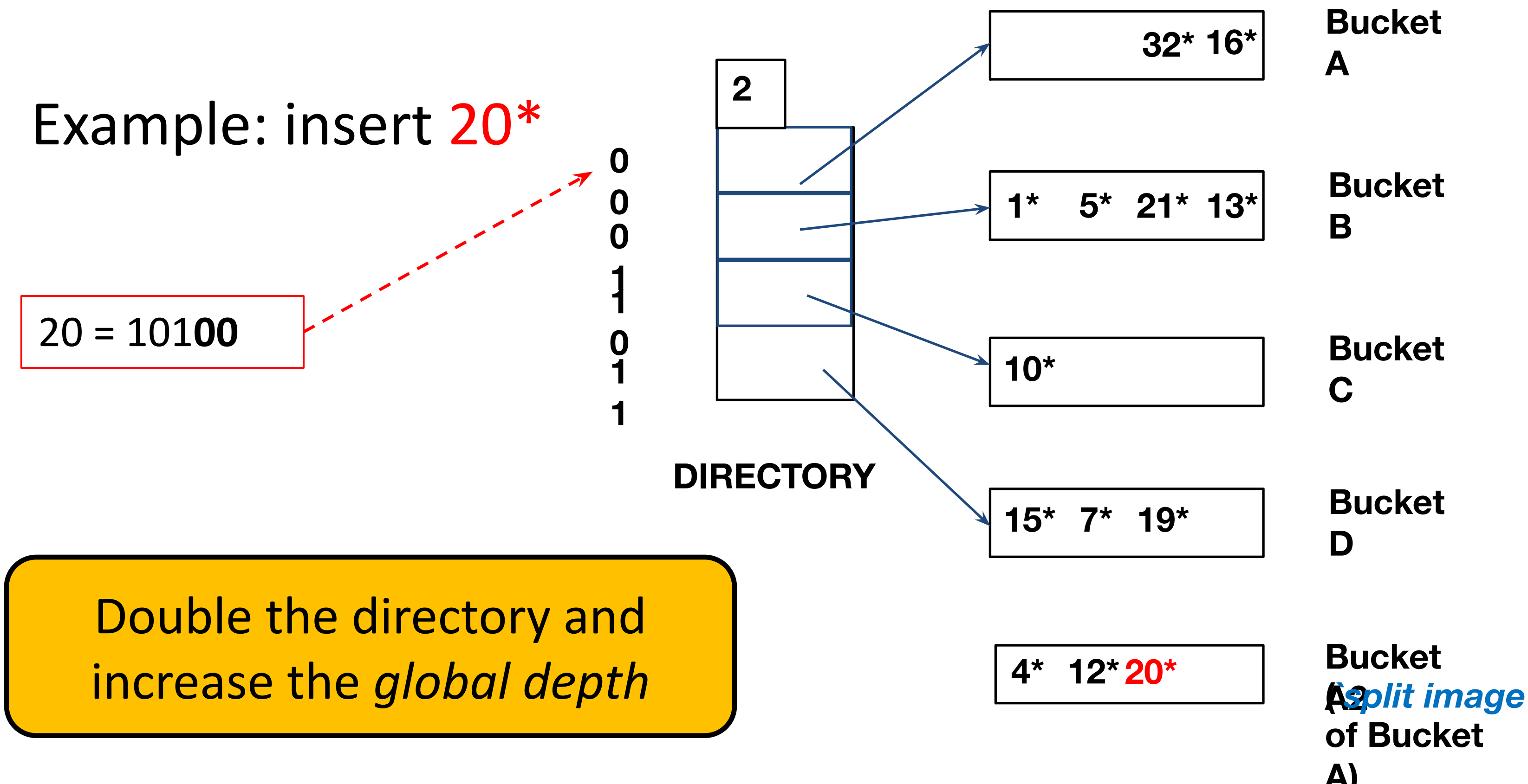


Is this enough?

Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*



Double the directory and increase the *global depth*

Extendible Hashing: Inserting Entries

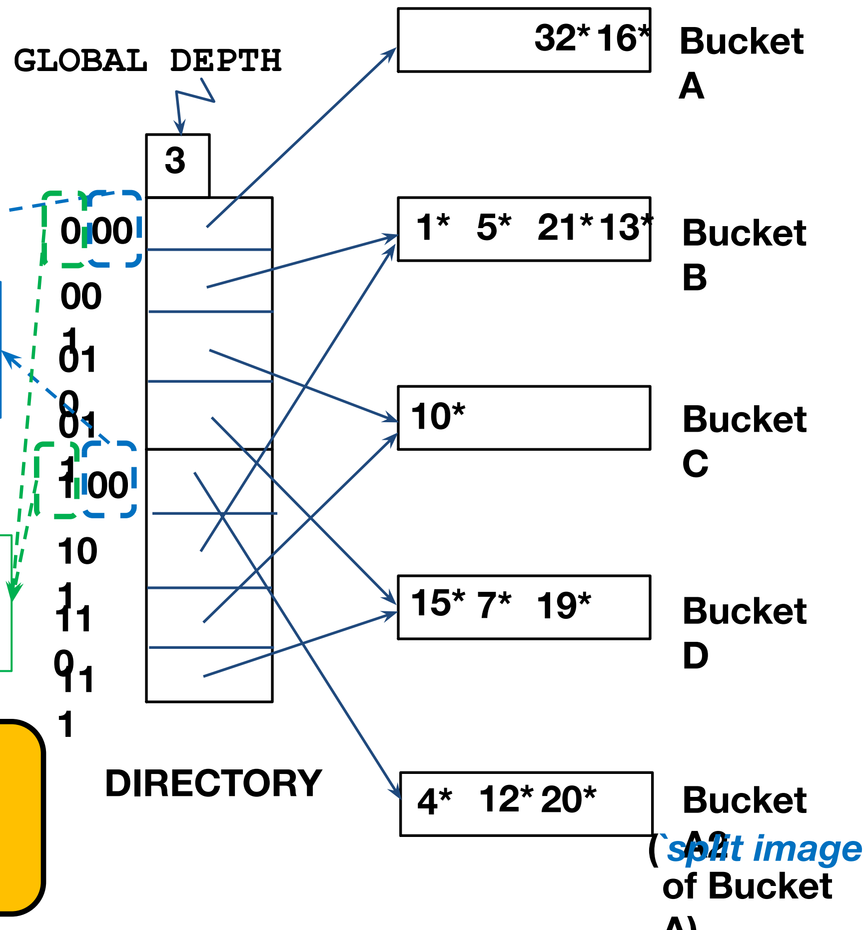
- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 20*

These two bits indicate a data entry that belongs to one of these two buckets

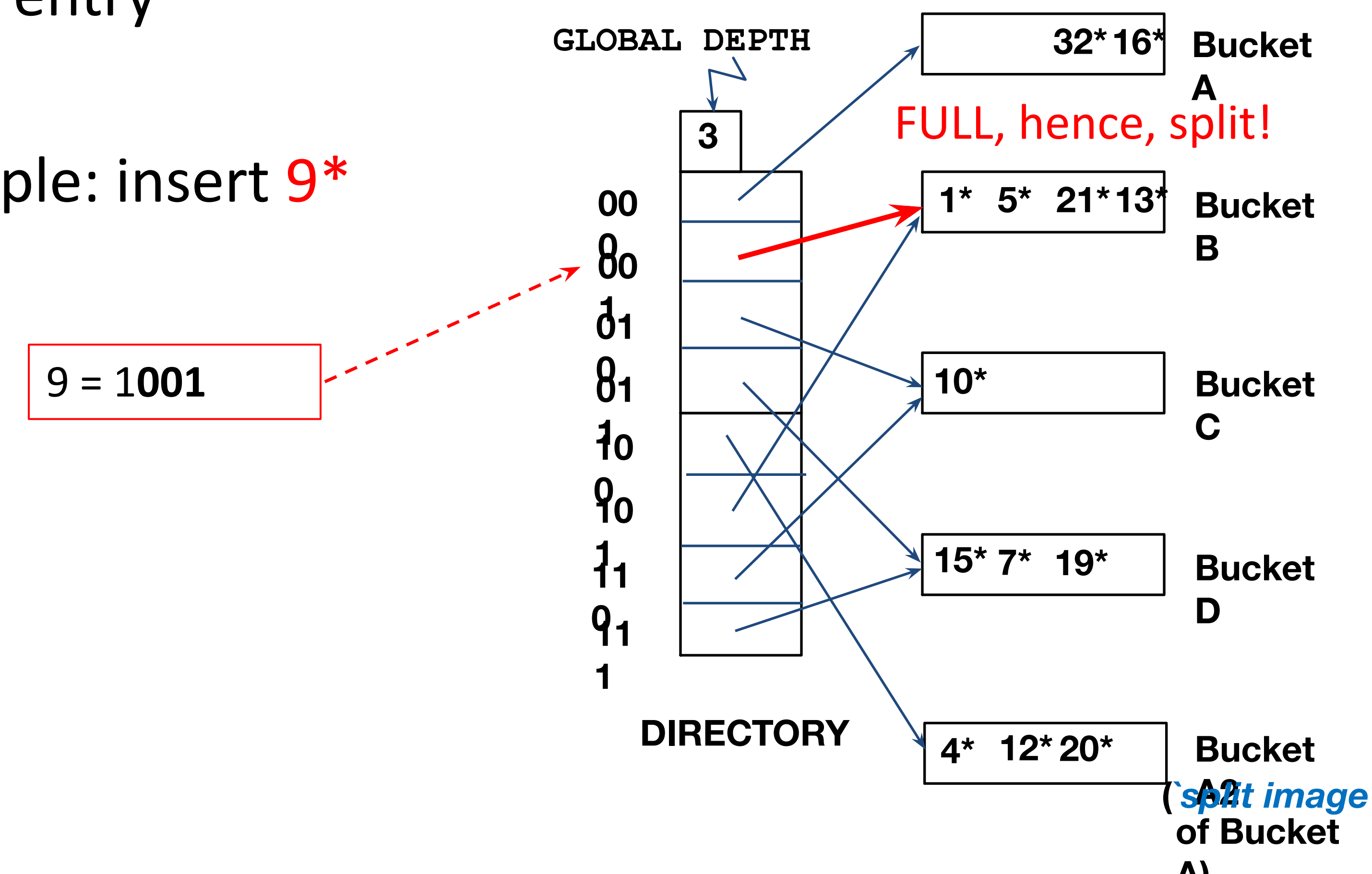
The third bit distinguishes between these two buckets!

But, is it necessary always to double the directory?



Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry
- Example: insert 9*

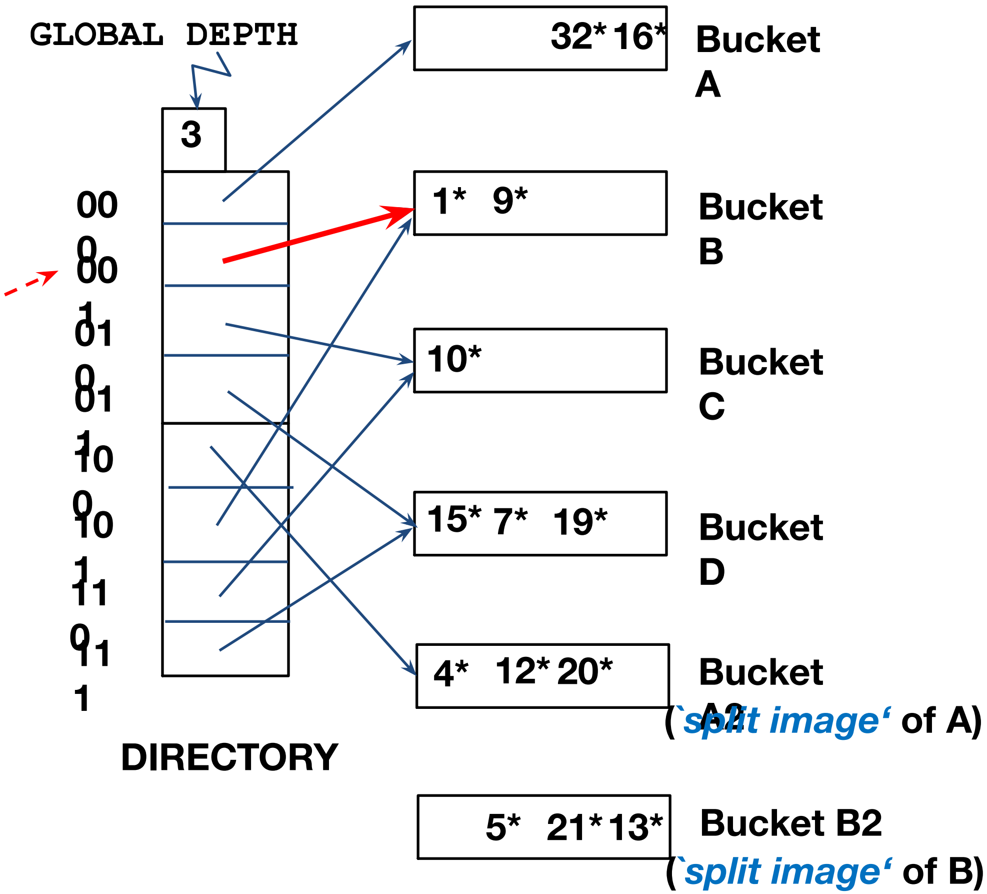


Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 9*

9 = 1001



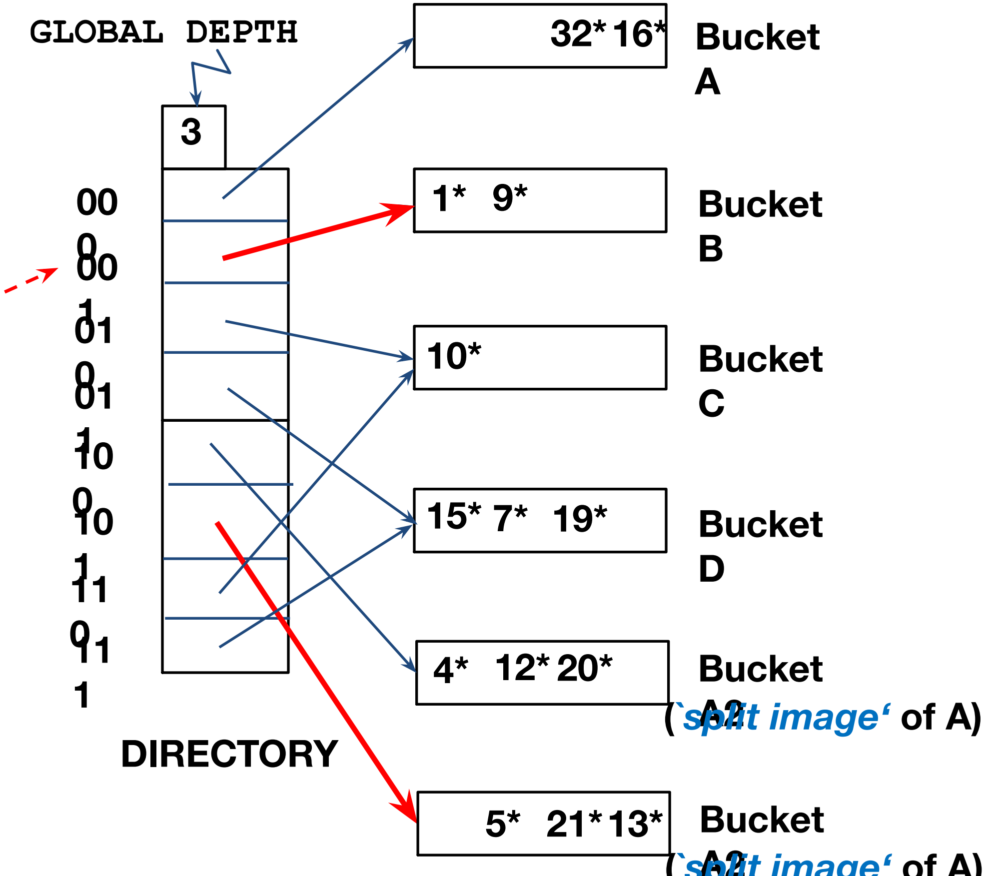
Almost there…

Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 9*

9 = 1001



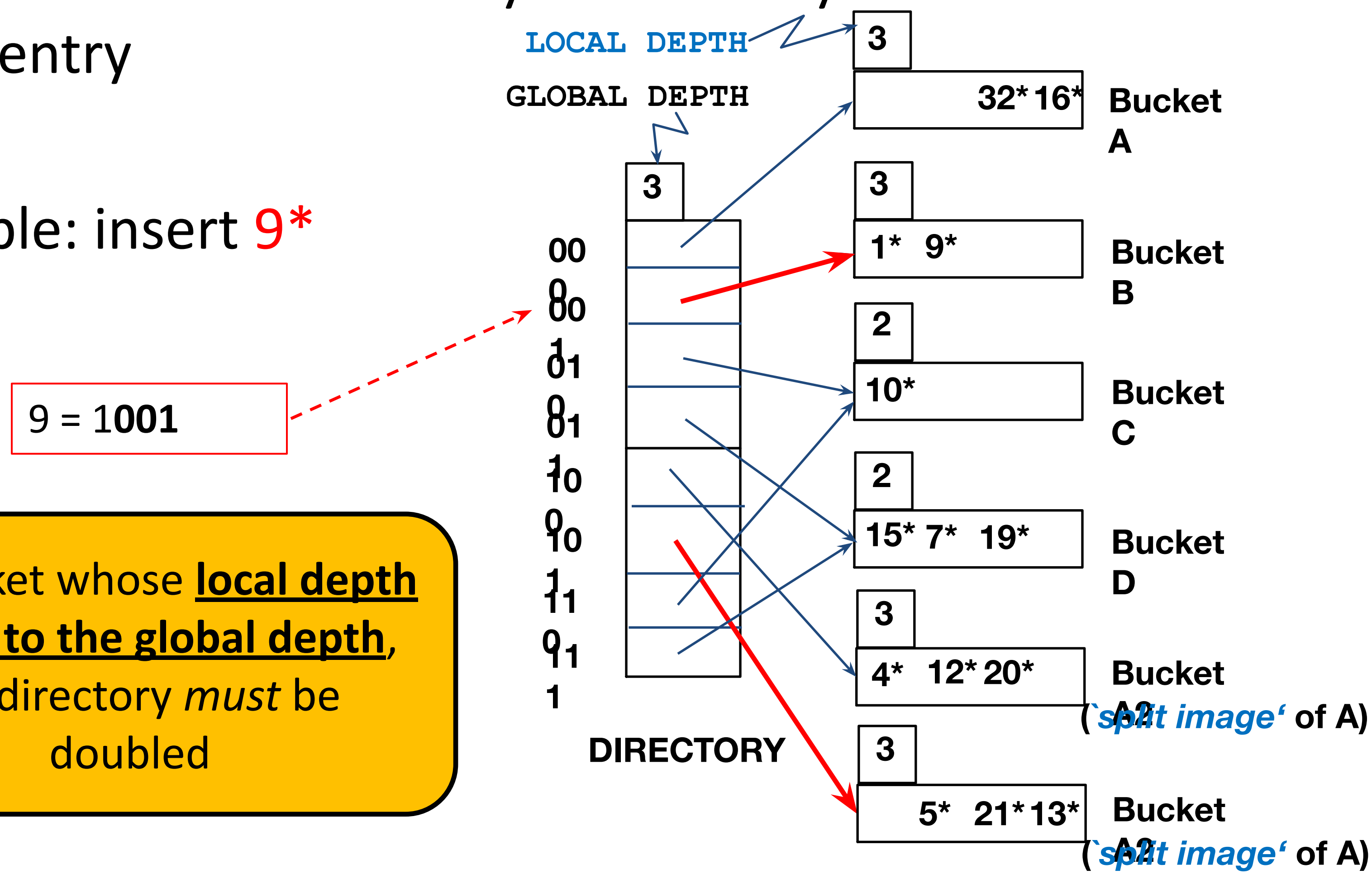
There was no need to double the directory!

When NOT to double the directory?

Extendible Hashing: Inserting Entries

- Find the appropriate bucket (as in search), split the bucket if full, double the directory if necessary and insert the given entry

- Example: insert 9*



If a bucket whose **local depth equals to the global depth**, the directory *must* be doubled

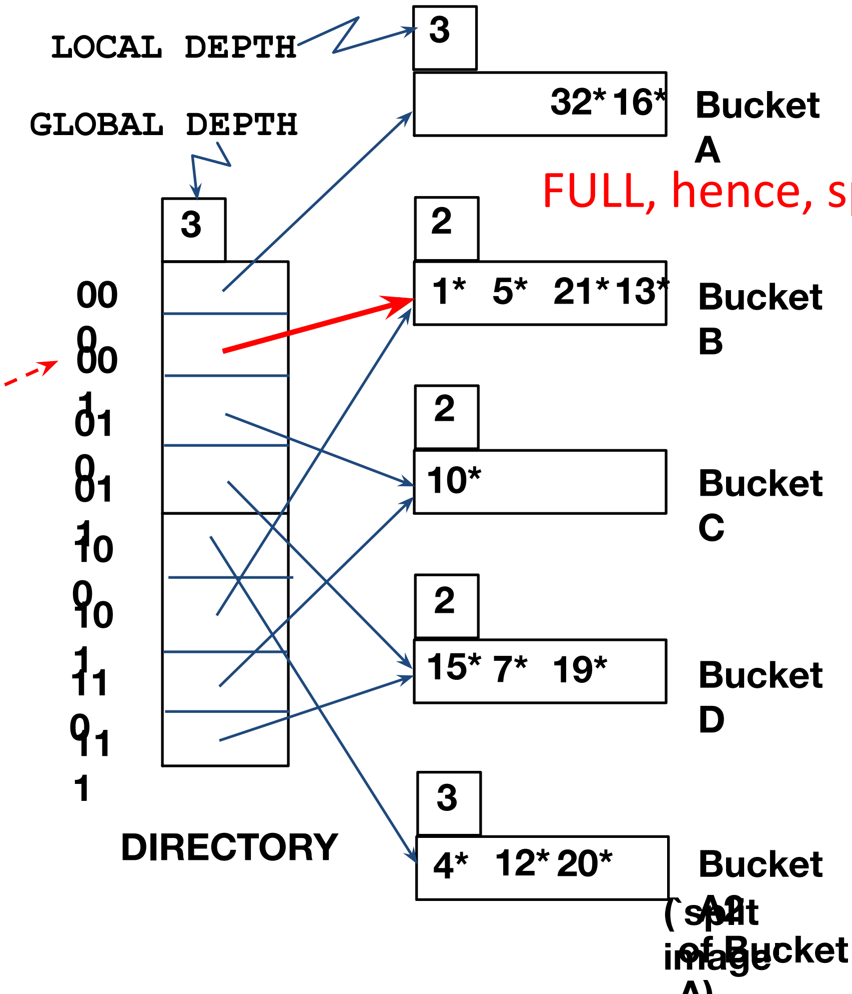
Extendible Hashing: Inserting Entries

- Example: insert 9*

Repeat…

9 = 1001

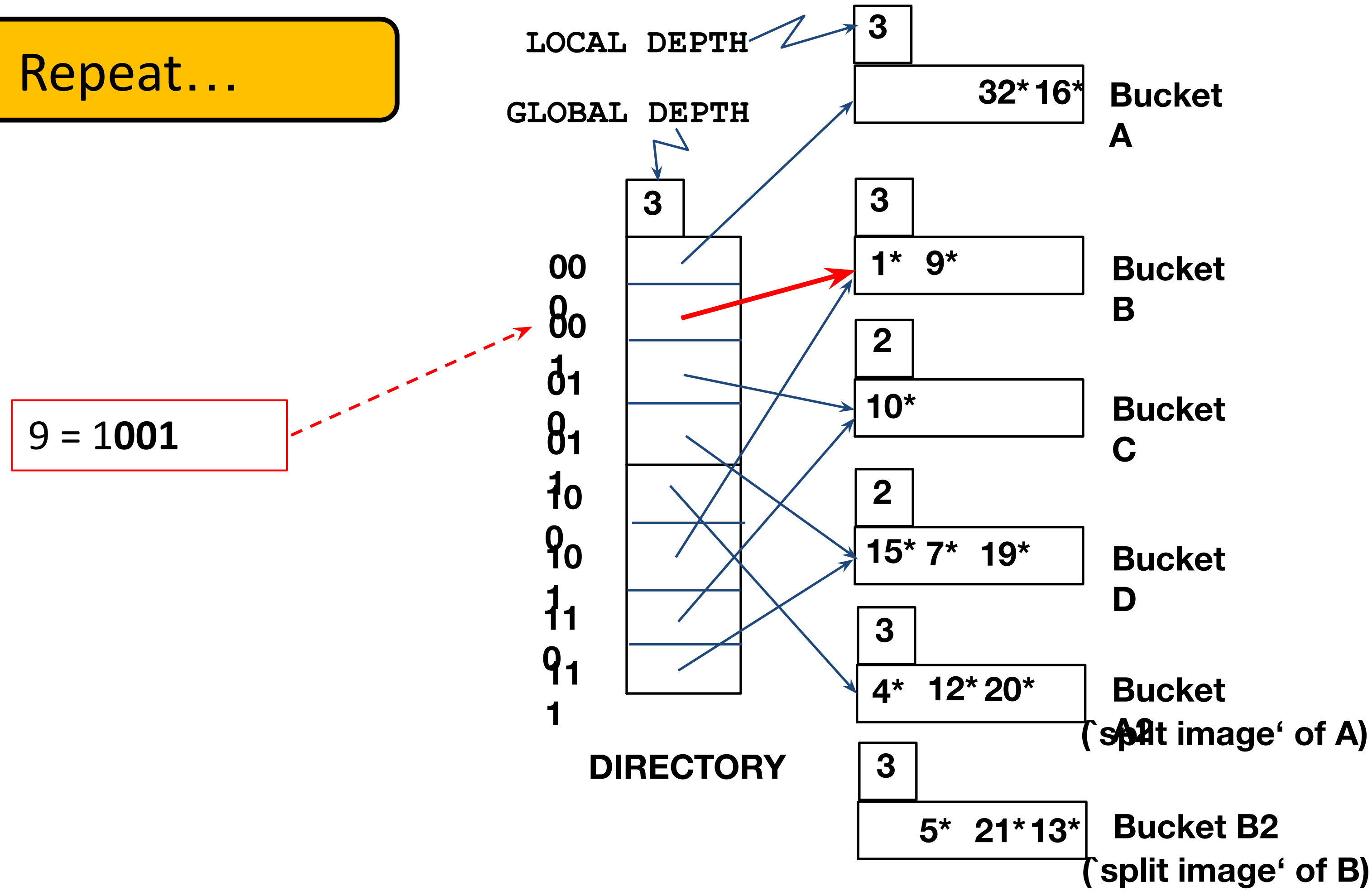
Because the local depth (i.e., 2) is *less than* the global depth (i.e., 3), NO need to double the directory



Extendible Hashing: Inserting Entries

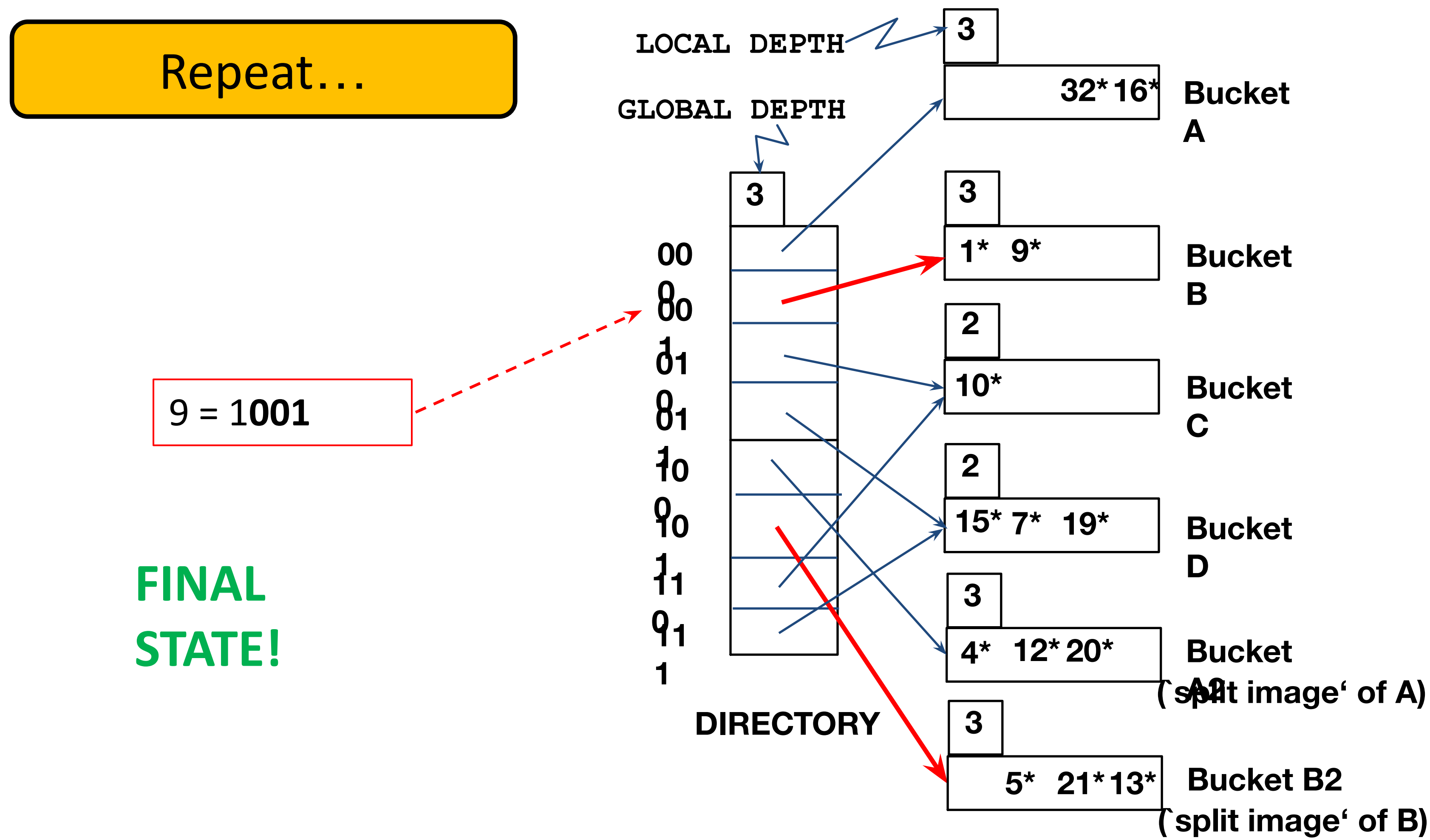
- Example: insert 9*

Repeat…



Extendible Hashing: Inserting Entries

- Example: insert 9*



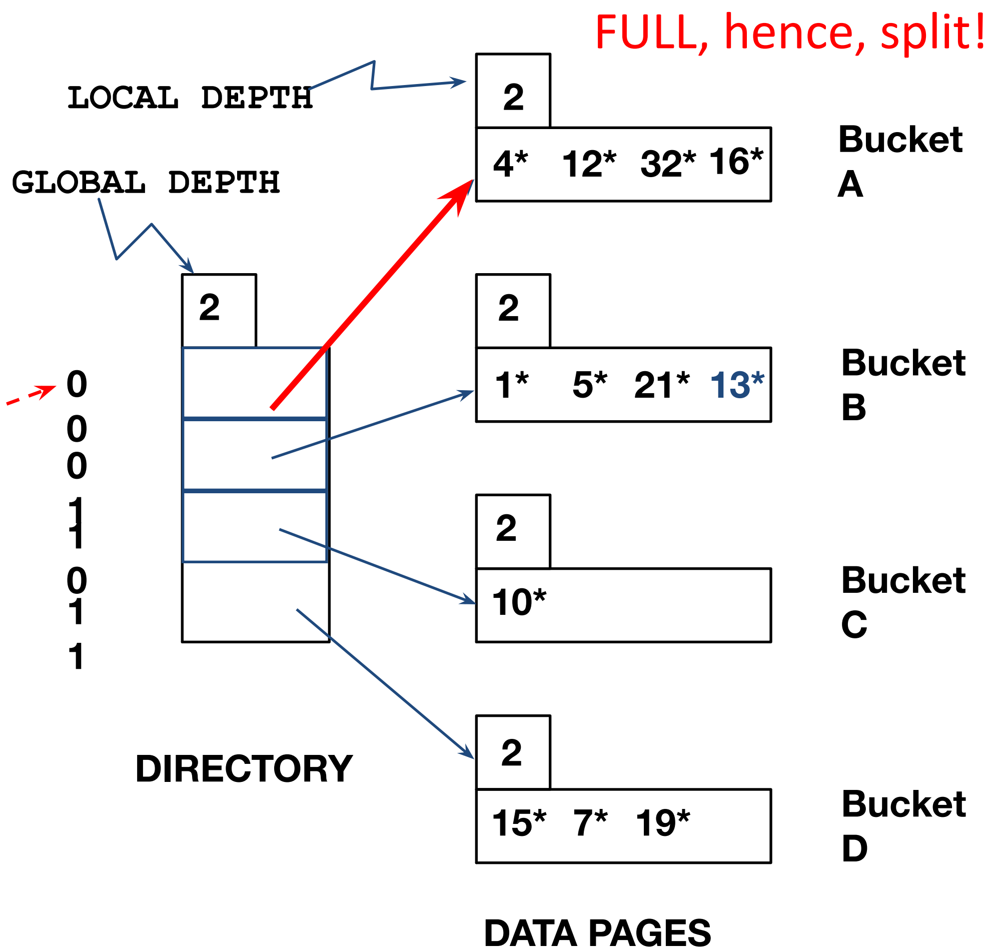
Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

20 = 10100

Because the local depth and the global depth are both 2, we *should* double the directory!

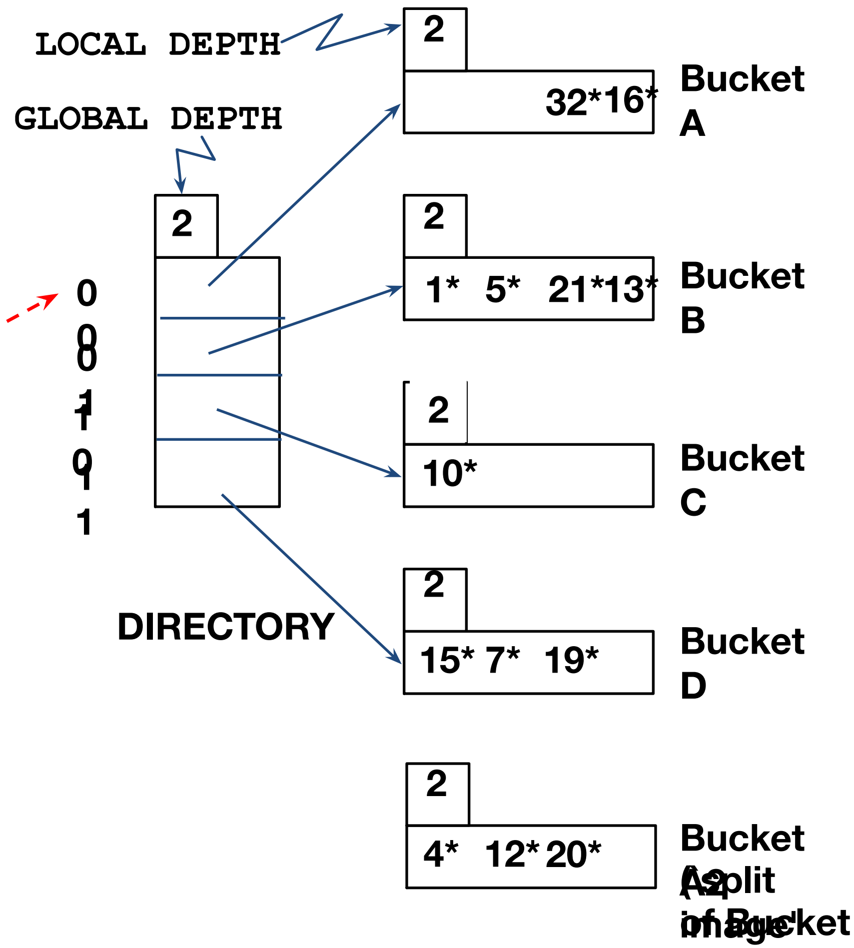


Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

20 = 10100



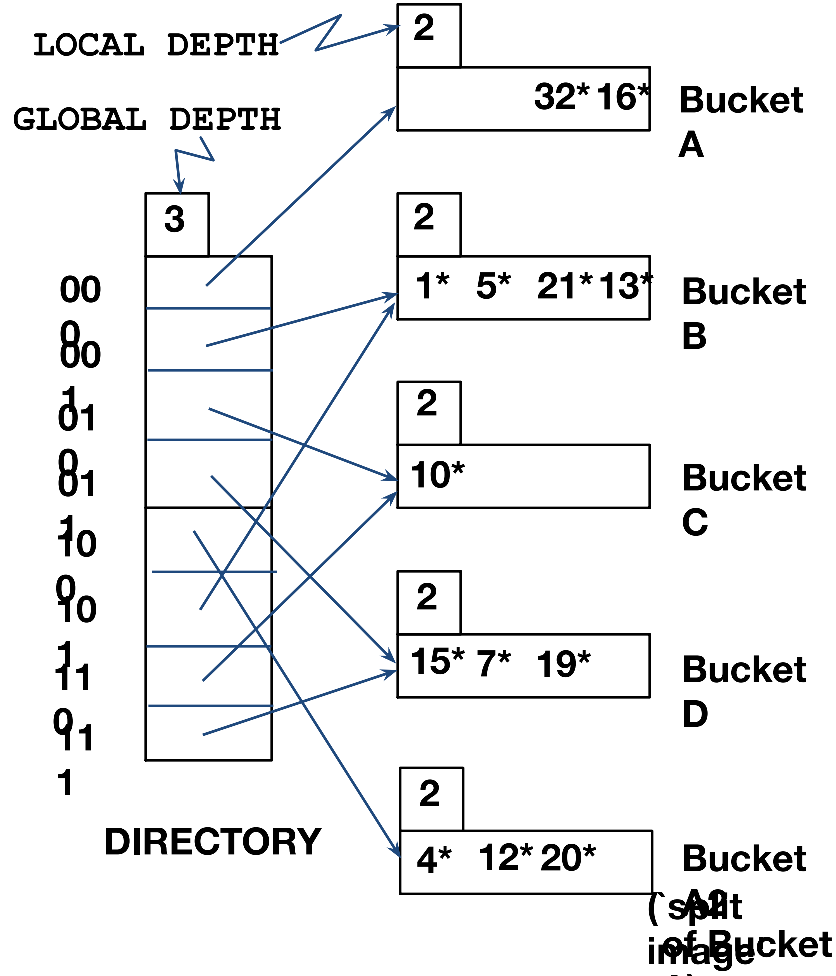
Is this enough?

Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

Is this enough?

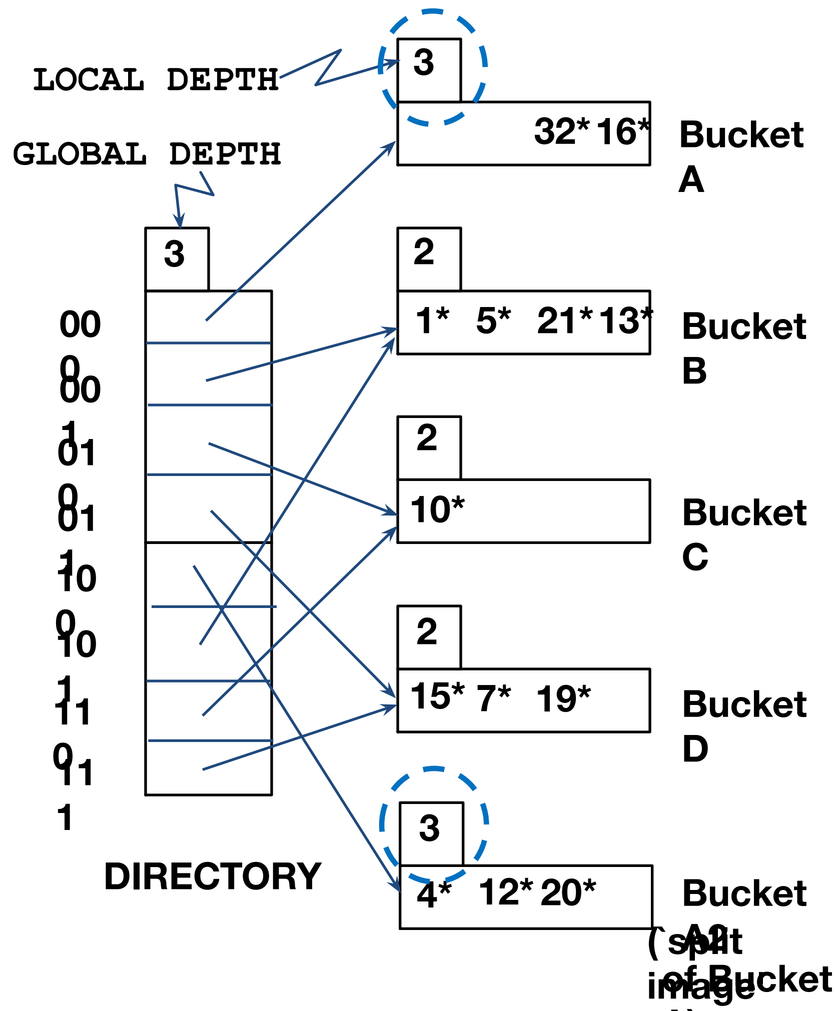


Extendible Hashing: Inserting Entries

- Example: insert 20*

Repeat…

FINAL STATE!



Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise
- Example

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g., 100110 AND 110011 = 100010

    100110 OR 110011 = 110111

    NOT 100110 = 011001
  - Males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster

- Bitmap indices generally very small compared with relation size
  - E.g., If number of distinct attribute values is 8, bitmap is only 1% of relation size

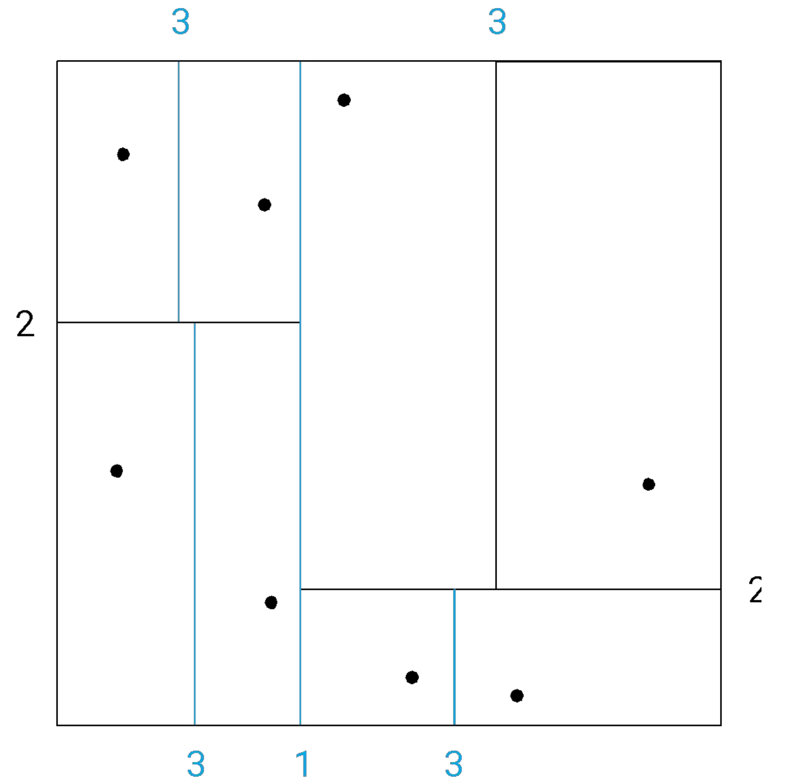
Spatial and Temporal Indices

Spatial Data

- Databases can store data types such as lines, polygons, in addition to raster images
  - allows relational databases to store and retrieve spatial information
  - Queries can use spatial conditions (e.g. contains or overlaps).
  - queries can mix spatial and nonspatial conditions
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- **Spatial join** of two spatial relations with the location playing the role of join attribute.

Indexing of Spatial Data

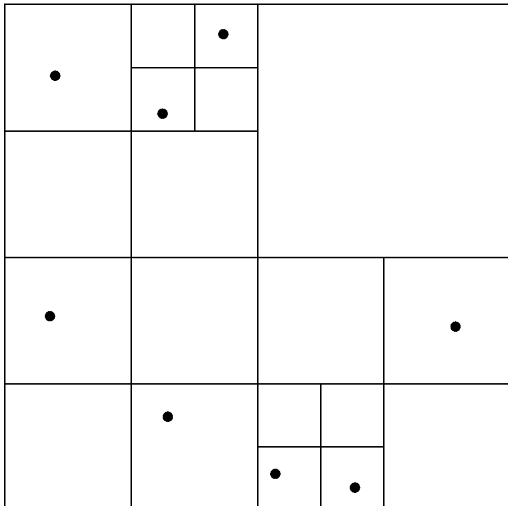
- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - Choose one dimension for partitioning at the root level of the tree.
  - Choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.



- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).

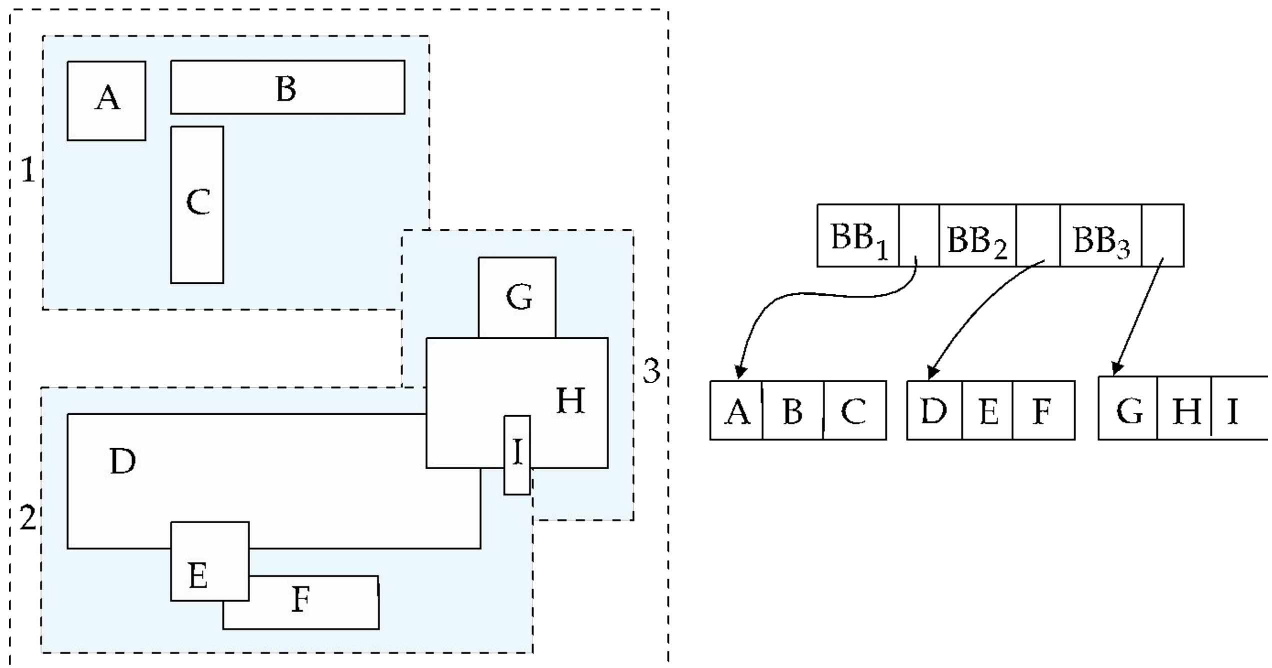


R-Trees

- **R-trees** are a N-dimensional extension of B$^+$-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R$^+$-trees and R*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an
  N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*

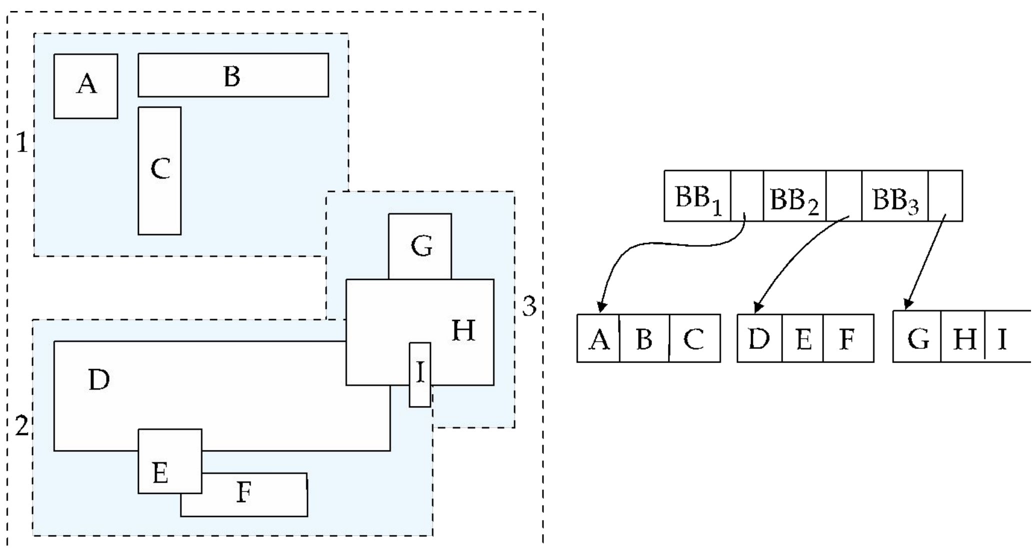
Example R-Tree

- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.



Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched, but works acceptably in practice.



Indexing Temporal Data

- Temporal data refers to data that has an associated time period (interval)
  - Example: a temporal version of the *course* relation

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

- Time interval has a start and end time
  - End time set to infinity (or large date such as 9999-12-31) if a tuple is currently valid and its validity end time is not currently known
- Query may ask for all tuples that are valid at a point in time or during a time interval
  - Index on valid time period speeds up this task

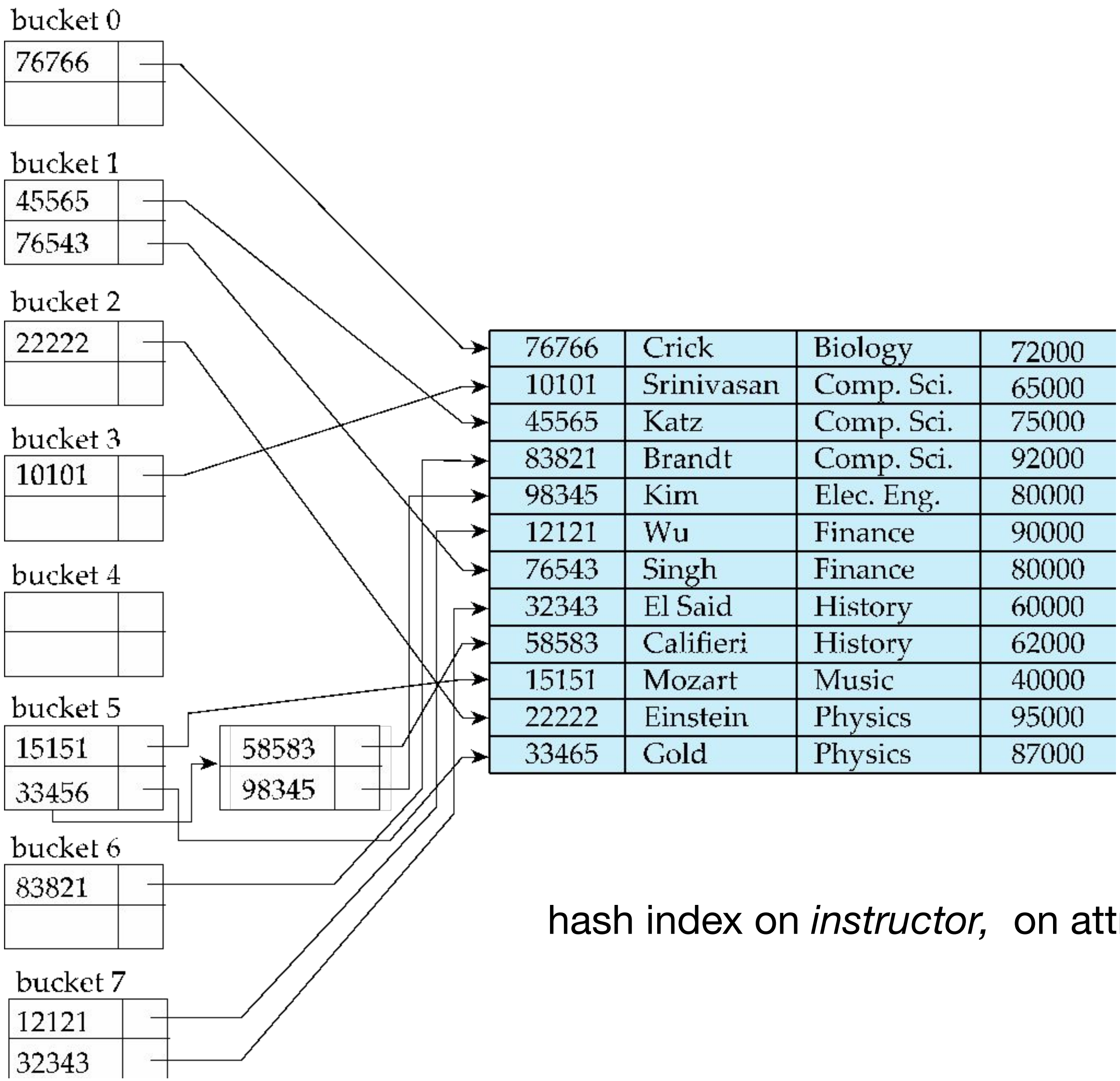
Indexing Temporal Data (Cont.)

- To create a temporal index on attribute *a*:
  - Use spatial index, such as R-tree, with attribute *a* as one dimension, and time as another dimension
    - Valid time forms an interval in the time dimension
  - Tuples that are currently valid cause problems, since value is infinite or very large
    - Solution: store all current tuples (with end time as infinity) in a separate index, indexed on (*a, start-time*)
      - To find tuples valid at a point in time *t* in the current tuple index, search for tuples in the range (*a, 0*) to (*a,t*)
- Temporal index on primary key can help enforce temporal primary key constraint

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

End of Chapter 14

Example of Hash Index



hash index on *instructor*, on attribute *ID*