Indexing Data
and
Tree Indexing

R&G Chapters 8,10

(slides adapted from content by J.Gehrke, J.Shanmugasundaram, and/or C.Koch)

Indexing

- Can we take advantage of extremely selective WHERE conditions to minimize work done?
  - Scans are expensive!
  - Create a datastructure to look up which pages contain useful information. (an Index)
- Simpler index = less help, but easier to manage.

Breaking Up Conditions

Boolean formulas can create complex conditions

```
(Officer.Ship = ‘1701A’
   AND Officer.Rank > 2)
        OR Officer.Rank > 3
```

Breaking Up Conditions

Boolean formulas can create complex conditions

```
(Officer.Ship = '1701A'
   AND Officer.Rank > 2)
       OR Officer.Rank > 3
```

First convert all conditions to Conjunctive Normal Form (CNF)

```
    (Officer.Ship = '1701A'
            OR Officer.Rank > 3)
AND (Officer.Rank > 2
            OR Officer.Rank > 3)
```

CNF is easier to manipulate. Each clause in a CNF formula MUST be true. Any estimate on the selectivity of a single clause is a lower bound on the estimate of the entire condition.

Breaking Up Conditions

Boolean formulas can create complex conditions

```
(Officer.Ship = ‘1701A’
   AND Officer.Rank > 2)
        OR Officer.Rank > 3
```

First convert all conditions to Conjunctive Normal Form (CNF)

```
(Officer.Ship = ‘1701A’
        OR Officer.Rank > 3)
            AND Officer.Rank > 2
```

Simplification may be possible

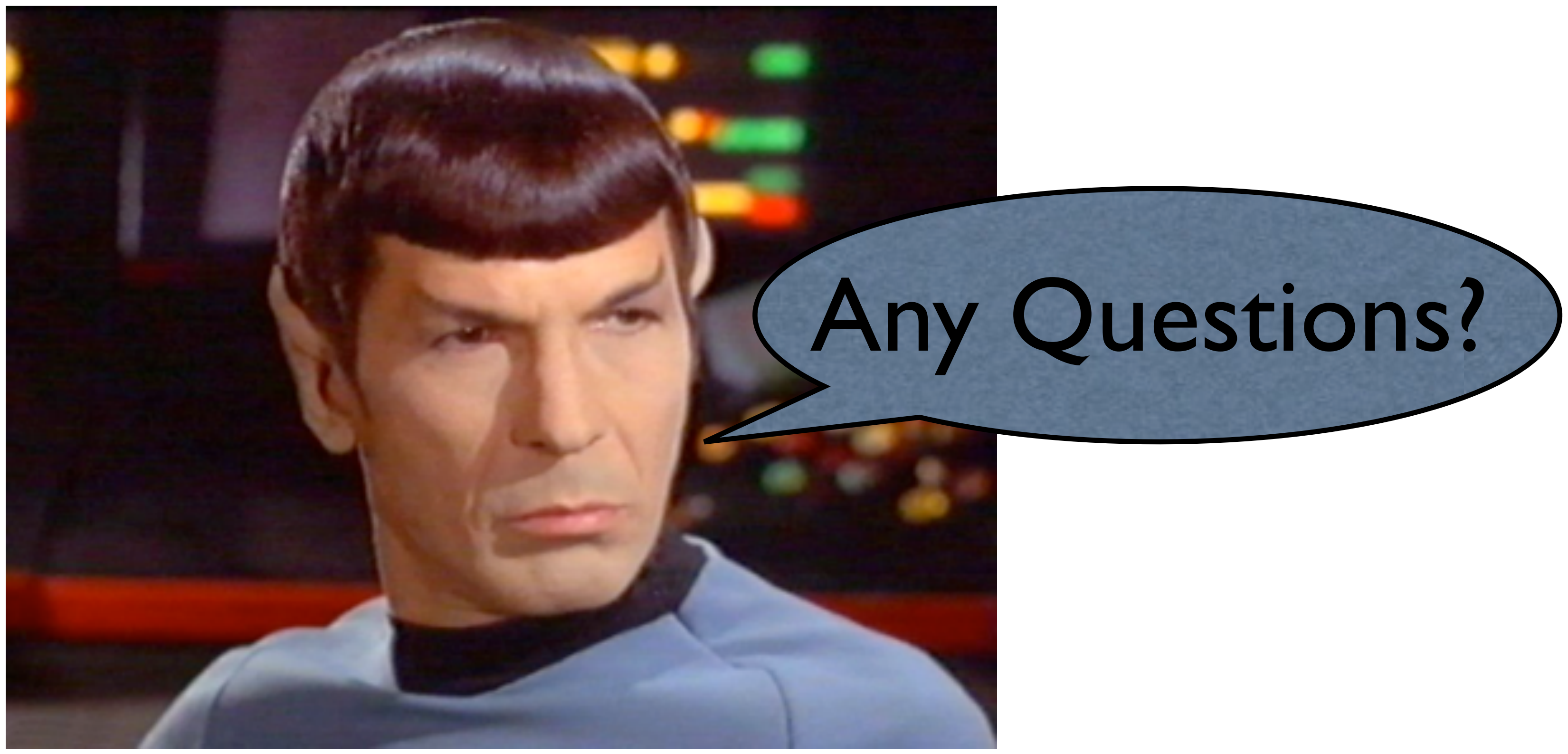
We’ll start off assuming that we’re working with conditions containing no ORs (just ANDs). Time permitting, we’ll discuss how to implement such conditions. See the text if you’re interested.

Indexing

- Each clause in a CNF boolean formula must be true.
- Index API: Give me all records (or record IDs) that satisfy this predicate (these predicates)
  - Equality search: All records with field X = ‘Y’
    - `Officer.Ship = ‘1701A’`
  - Range search: All records with field X ∈ [Y, Z]
    - `Officer.Rank ∈ [3, +∞)`

Indexing

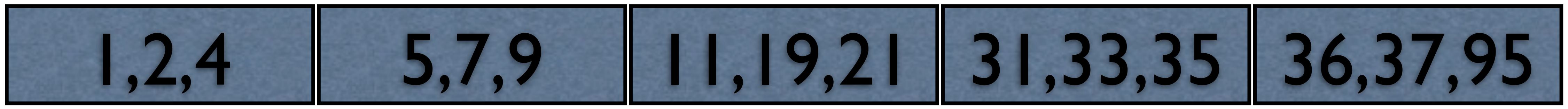
- Indexes are typically built over one (key) field k
- Index stores mappings from key k to :
  - k → The full tuple with key value k
  - k → Record ID for Tuple with key value k
  - k → List of Record/RecordIDs with key value k
- The choice of data to store is orthogonal to the choice of how to map key to value.


Any Questions?

Tree Indices

- Use a tree structure to organize key/value pairs.
  - Efficiently supports both equality and range searches
- Can be used to support the physical layouts of a data file (e.g., sorted order).

Sorted File

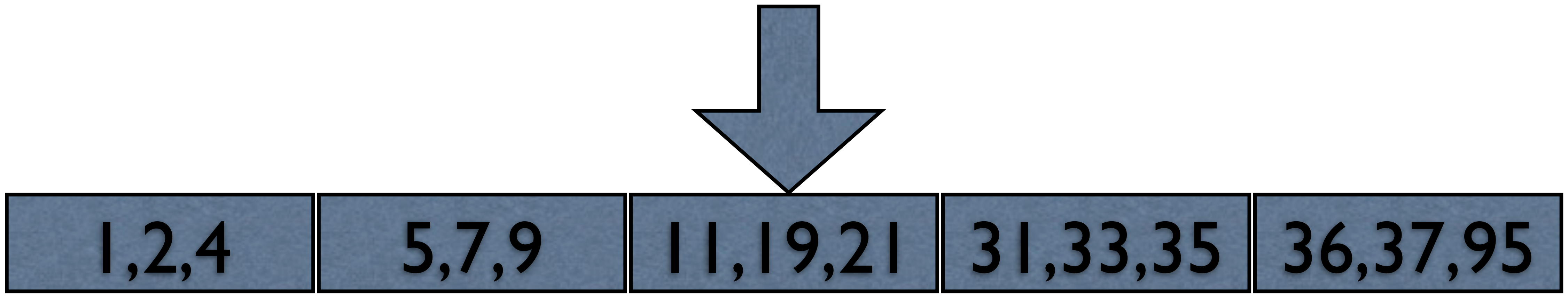


5 pages of data on disk, in sorted order

How many IOs does a binary search for [37] take?

Log(N) IOs.

Sorted File

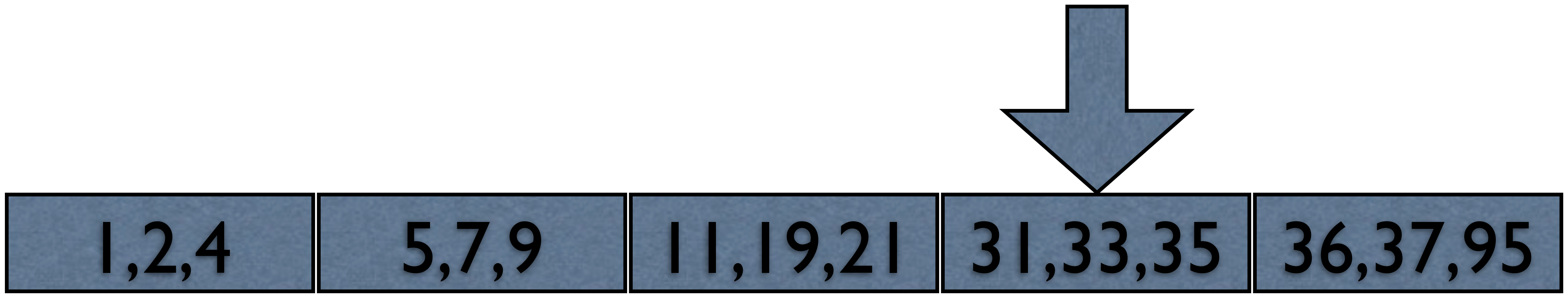


5 pages of data on disk, in sorted order

How many IOs does a binary search for [37] take?

Log(N) IOs.

Sorted File

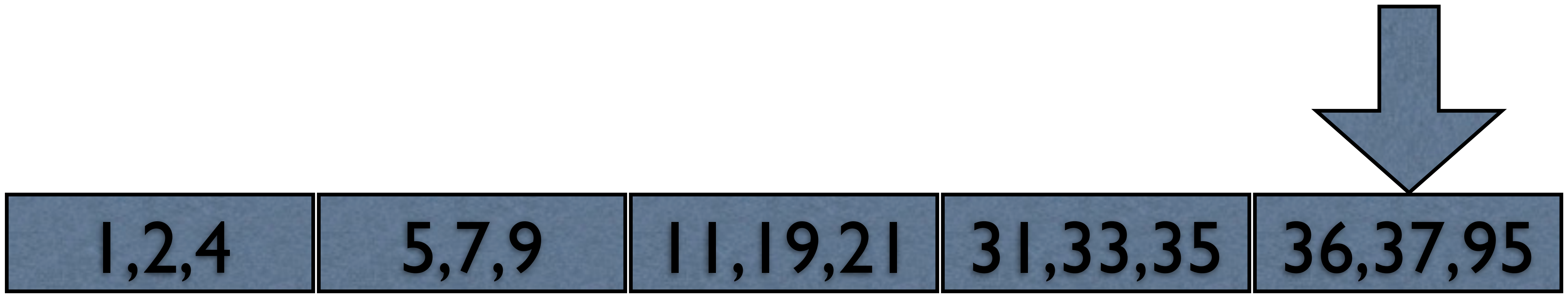


5 pages of data on disk, in sorted order

How many IOs does a binary search for [37] take?

Log(N) IOs.

Sorted File



5 pages of data on disk, in sorted order

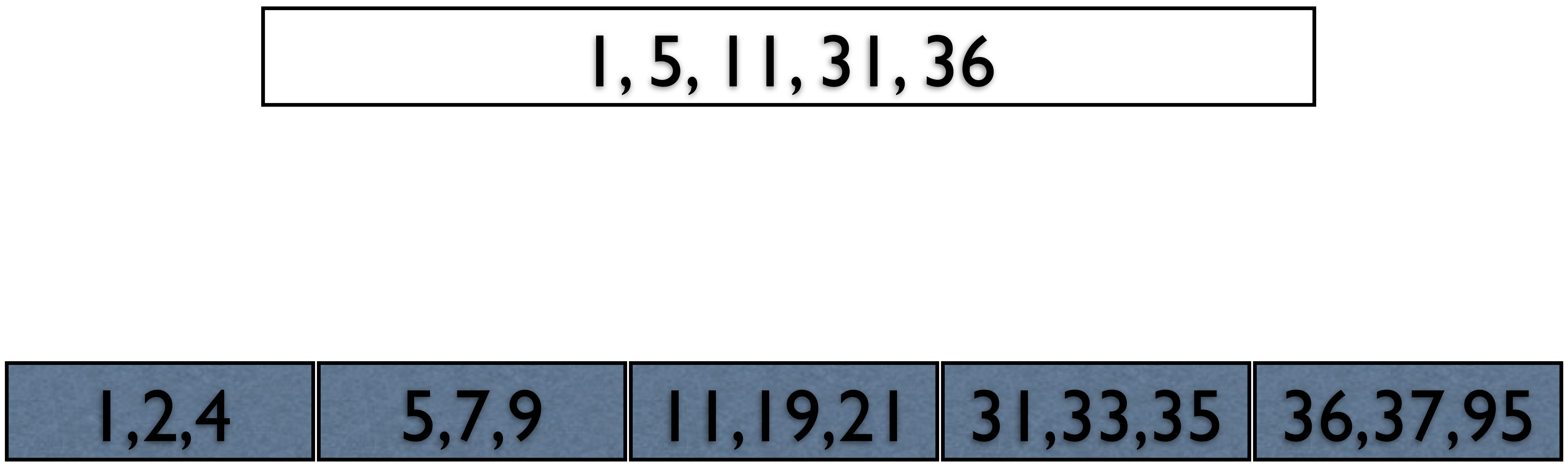
How many IOs does a binary search for [37] take?

3 Full IOs (can't be parallelized)
How many IOs in general?

Log(N) IOs.

Tree Index

Simple Idea: Create an Index File

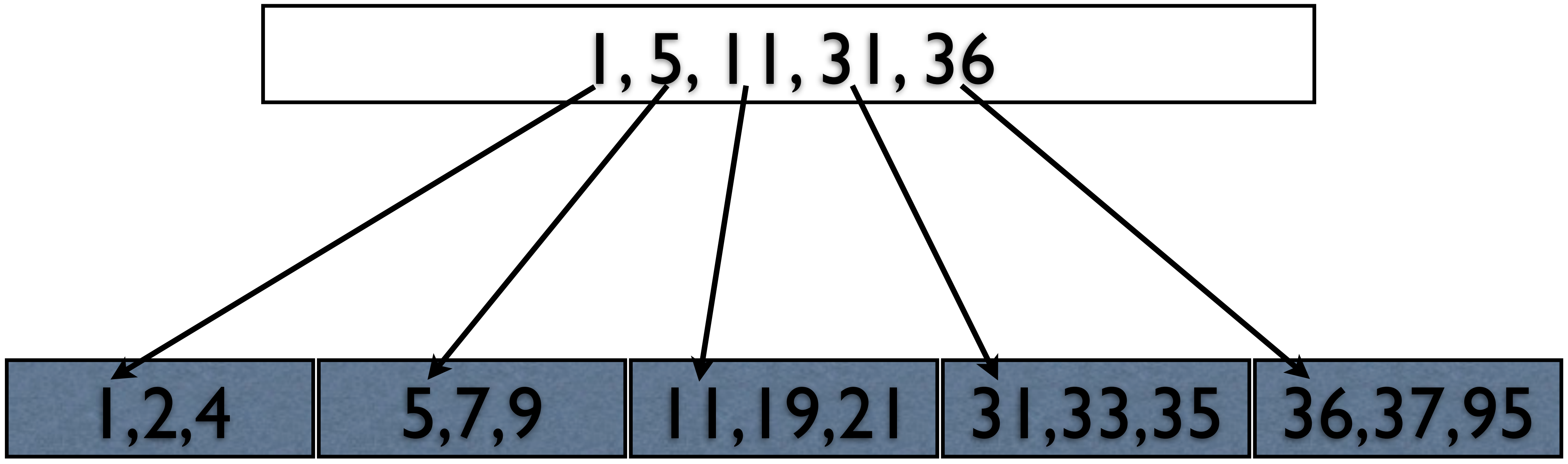


Index file stores record of first key on each page
Binary Search on the index is cheaper!

If we say 3 index/data records per page, the index takes 2 pages to store. For a data file of size N, the index takes N/3 pages to store. More generally, if we can store K index records per page (typically ~1000s), the index takes N/K pages. Even with ~1000 index records per page, 10 PB of data still requires a 10 GB index.

Tree Index

Simple Idea: Create an Index File



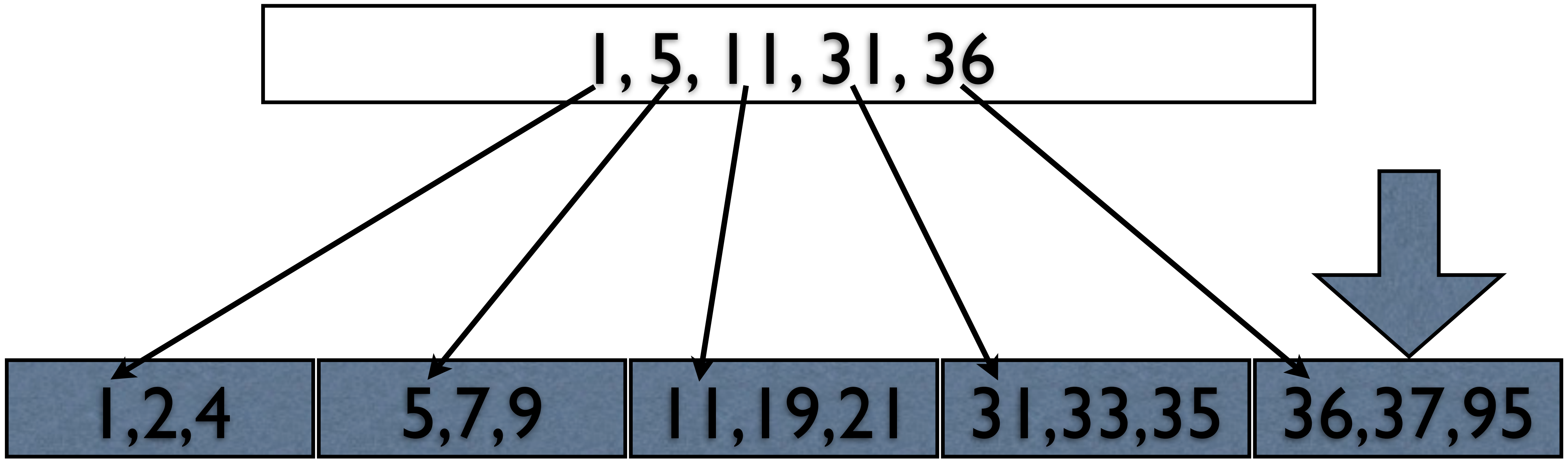
Index file stores record of first key on each page

Binary Search on the index is cheaper!

If we say 3 index/data records per page, the index takes 2 pages to store. For a data file of size N, the index takes N/3 pages to store. More generally, if we can store K index records per page (typically ~1000s), the index takes N/K pages. Even with ~1000 index records per page, 10 PB of data still requires a 10 GB index.

Tree Index

Simple Idea: Create an Index File



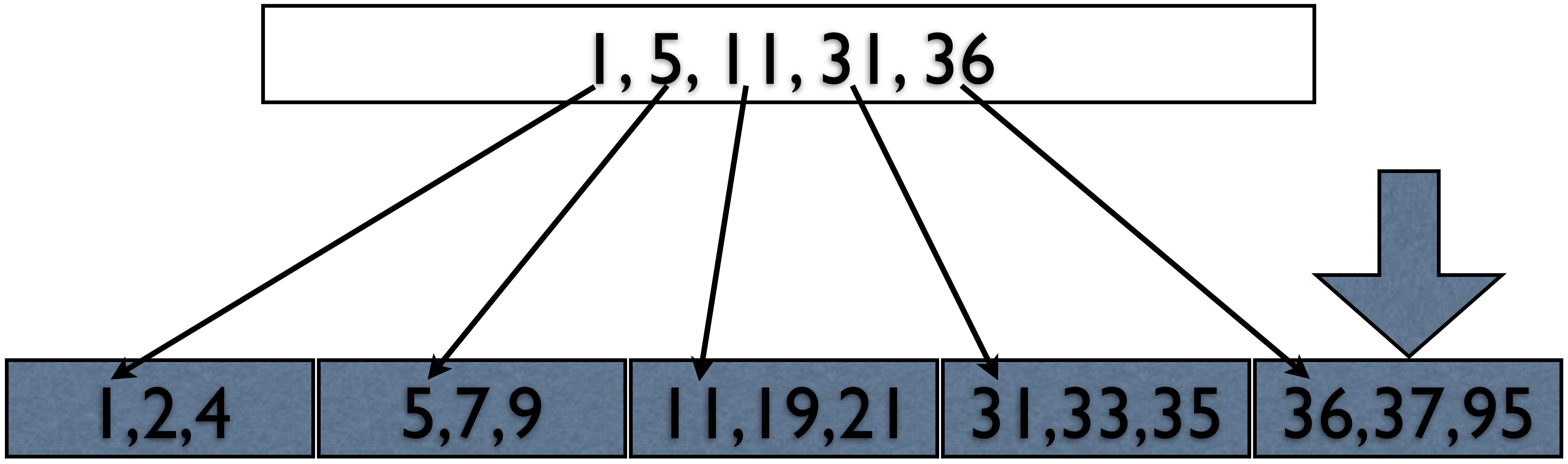
Index file stores record of first key on each page
Binary Search on the index is cheaper!

If we say 3 index/data records per page, the index takes 2 pages to store. For a data file of size N, the index takes N/3 pages to store. More generally, if we can store K index records per page (typically ~1000s), the index takes N/K pages. Even with ~1000 index records per page, 10 PB of data still requires a 10 GB index.

Tree Index

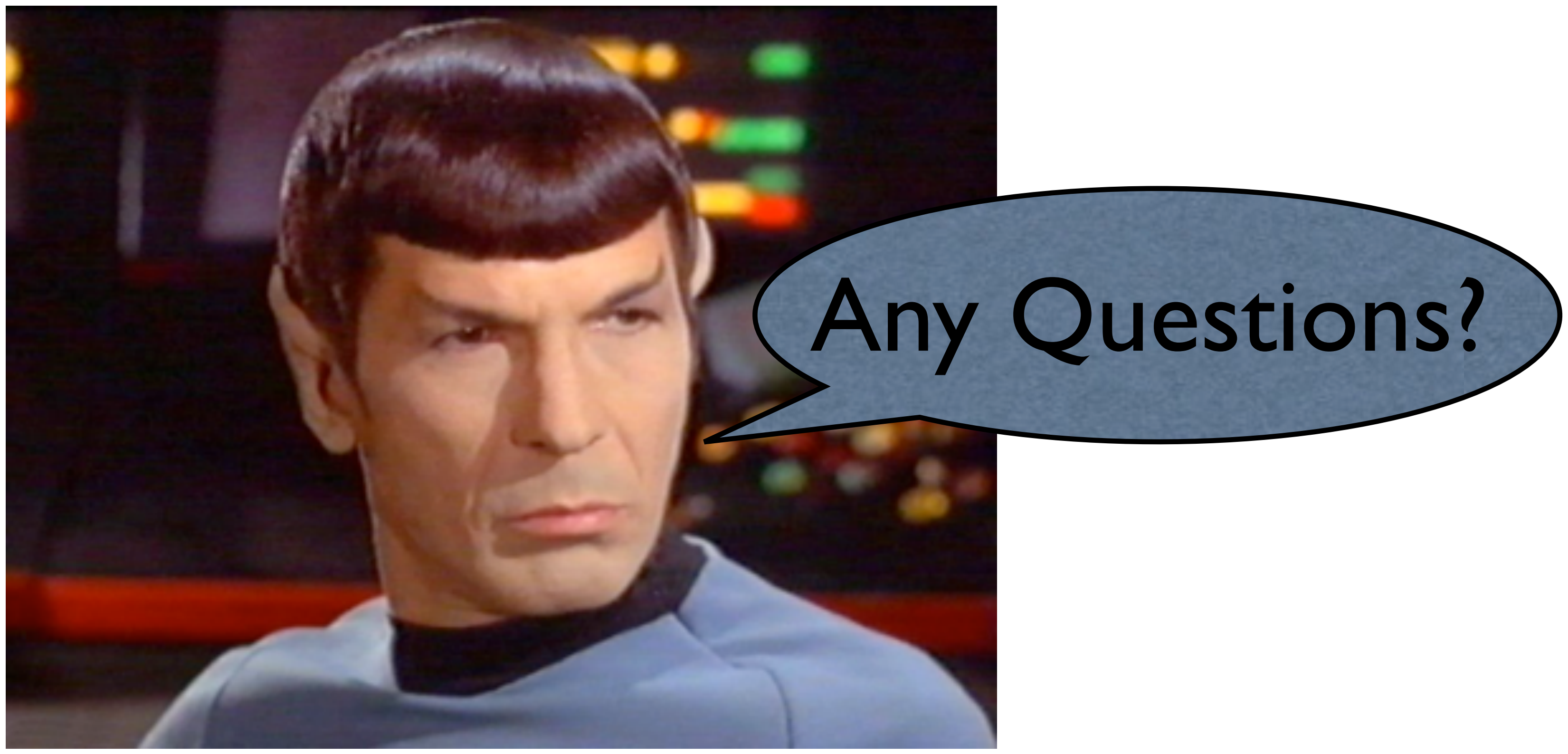
Simple Idea: Create an Index File

But the Index can still get Big!



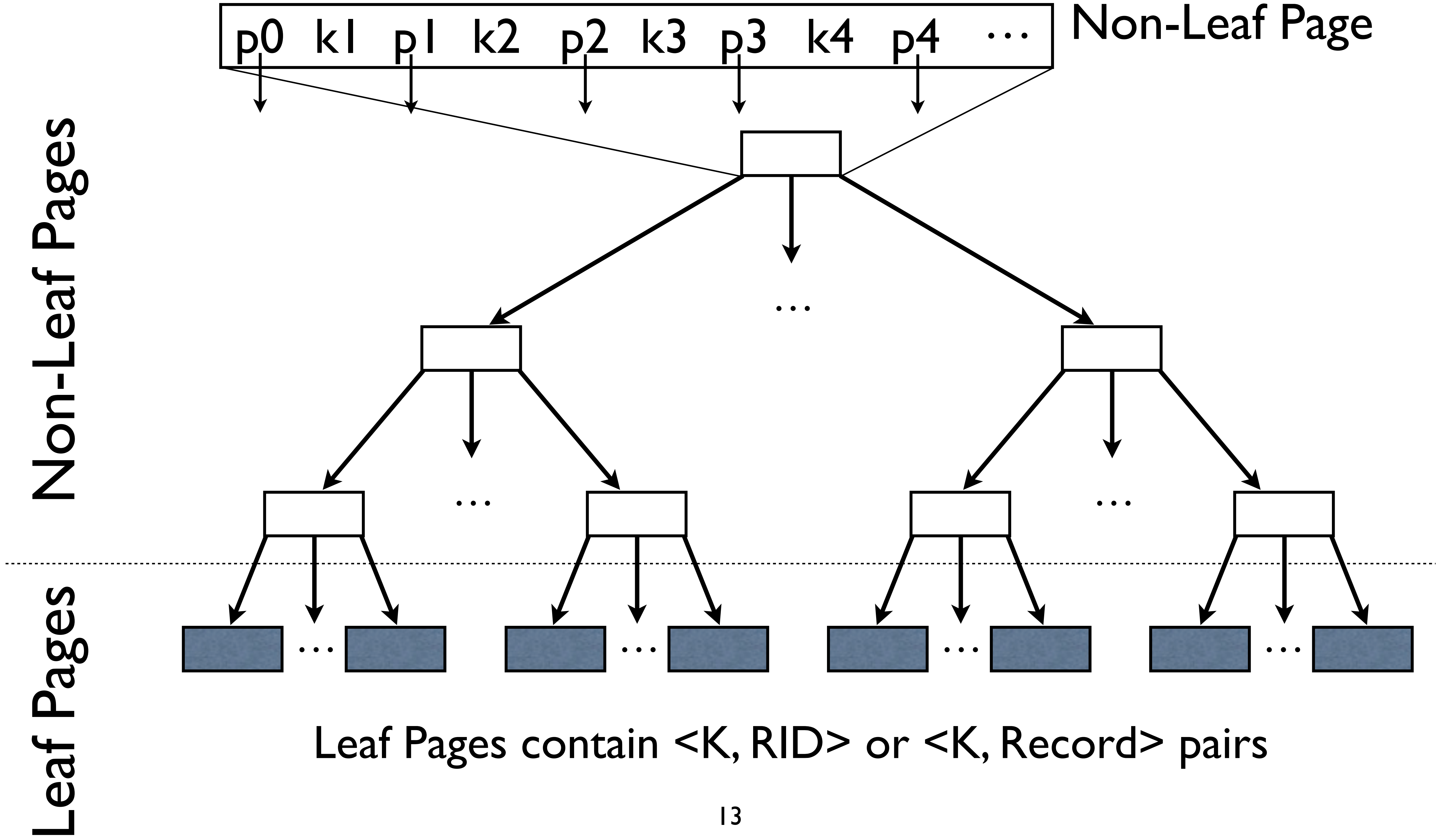
Index file stores record of first key on each page
Binary Search on the index is cheaper!

If we say 3 index/data records per page, the index takes 2 pages to store. For a data file of size N, the index takes N/3 pages to store. More generally, if we can store K index records per page (typically ~1000s), the index takes N/K pages. Even with ~1000 index records per page, 10 PB of data still requires a 10 GB index.



Monday, February 4, 13

The ISAM Datastructure



Leaf Pages contain <K, RID> or <K, Record> pairs

ISAM: Index–structured access method

Constructing an ISAM Index

Constructing an ISAM Index

1) Allocate (sequential) leaf pages

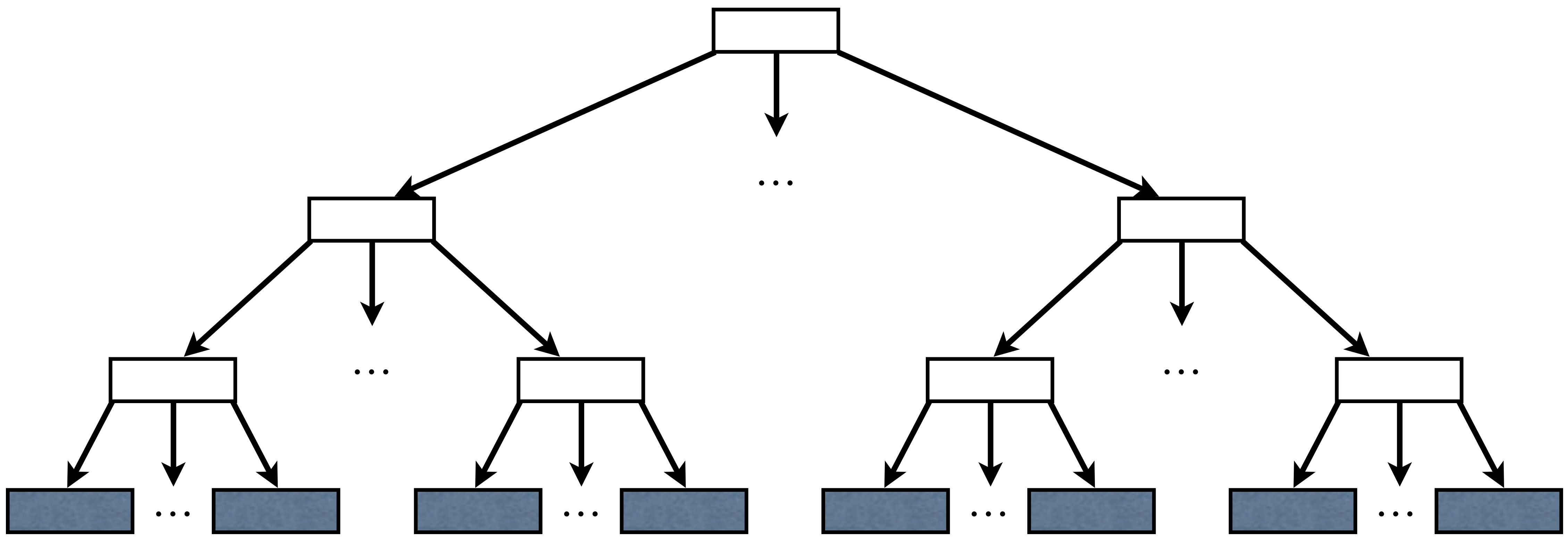
2) Ensure that the data on the leaf pages is sorted

Constructing an ISAM Index

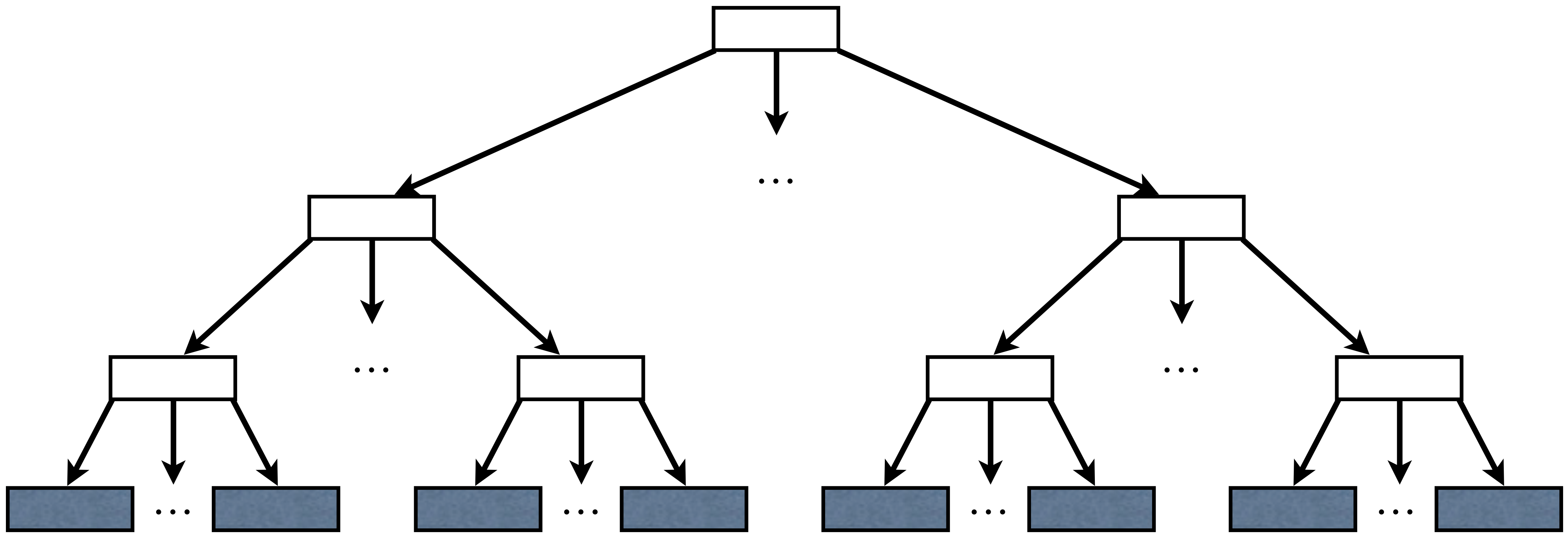
1) Allocate (sequential) leaf pages

2) Ensure that the data on the leaf pages is sorted

3) Build the non-leaf pages (in arbitrary order)



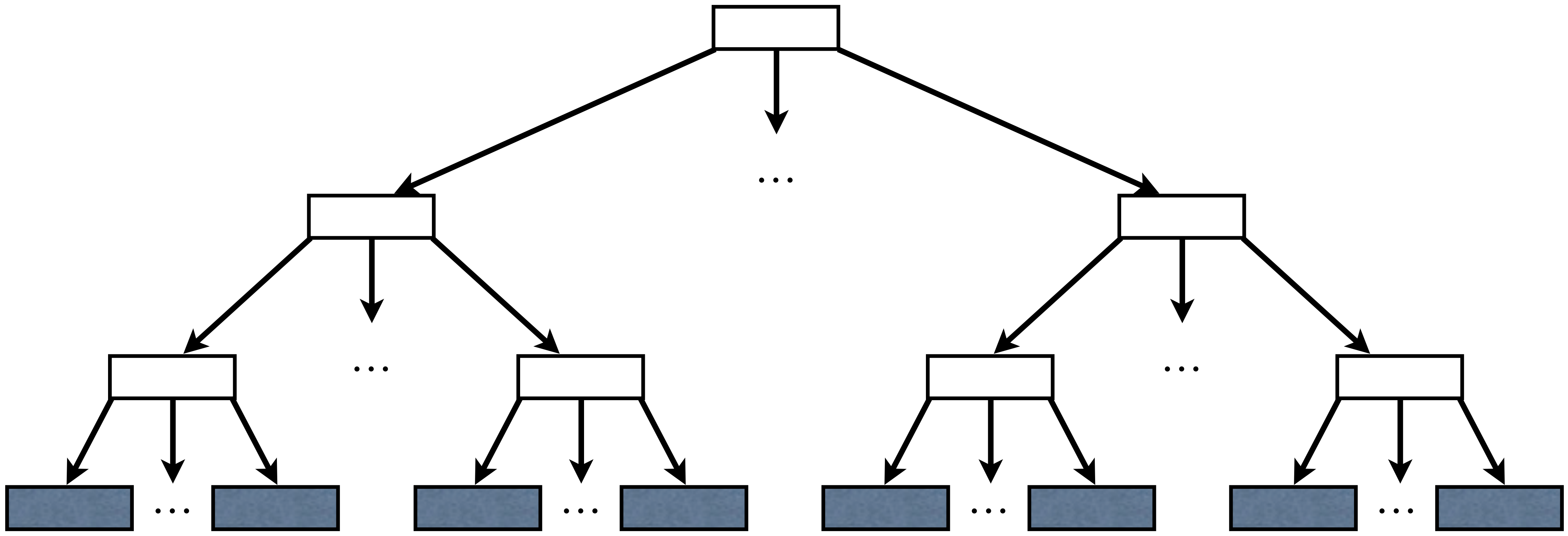
ISAM Index Searches



What's the access cost for an ISAM equality search? (for K pointers/page, Log_K(N))
range search? (2Log_K(N) + #pages returned)

ISAM Index Searches

Equality: Start at root, use key comparisons to find leaf

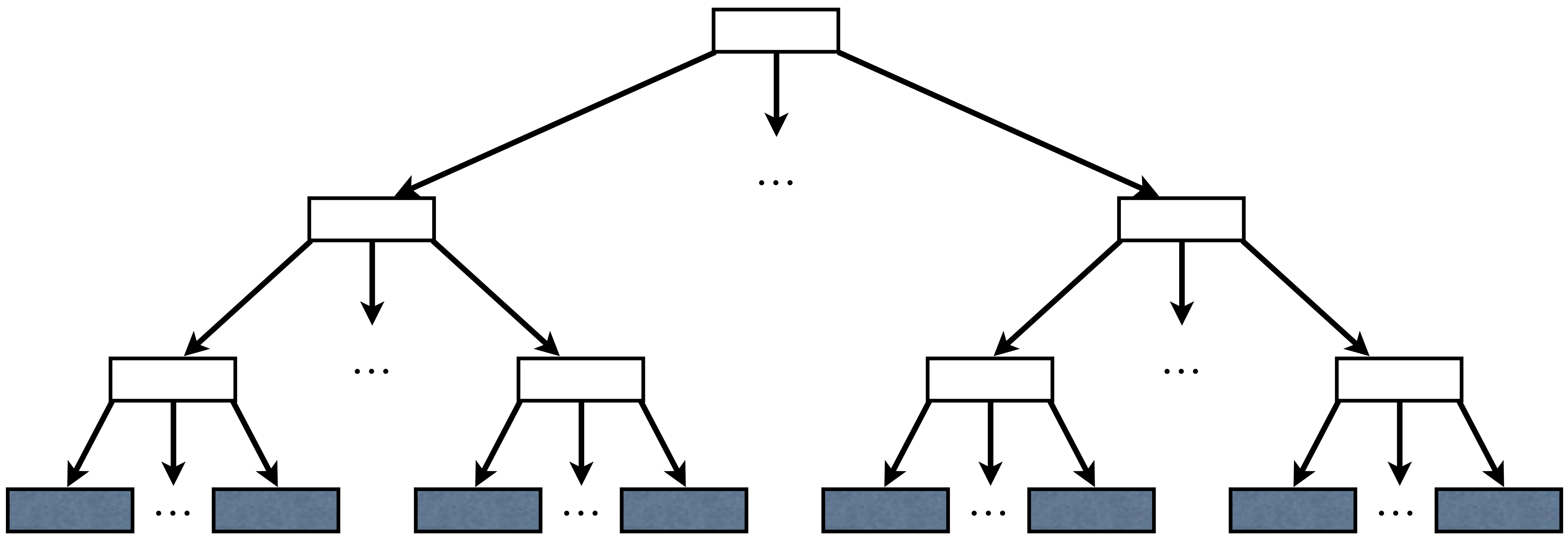


What’s the access cost for an ISAM equality search? (for K pointers/page, Log_K(N))
range search? (2Log_K(N) + #pages returned)

ISAM Index Searches

Equality: Start at root, use key comparisons to find leaf

Range: Use key comparisons to find start and end page
Scan all pages in between start/end leaves.



What's the access cost for an ISAM equality search? (for K pointers/page, Log_K(N))
range search? (2Log_K(N) + #pages returned)

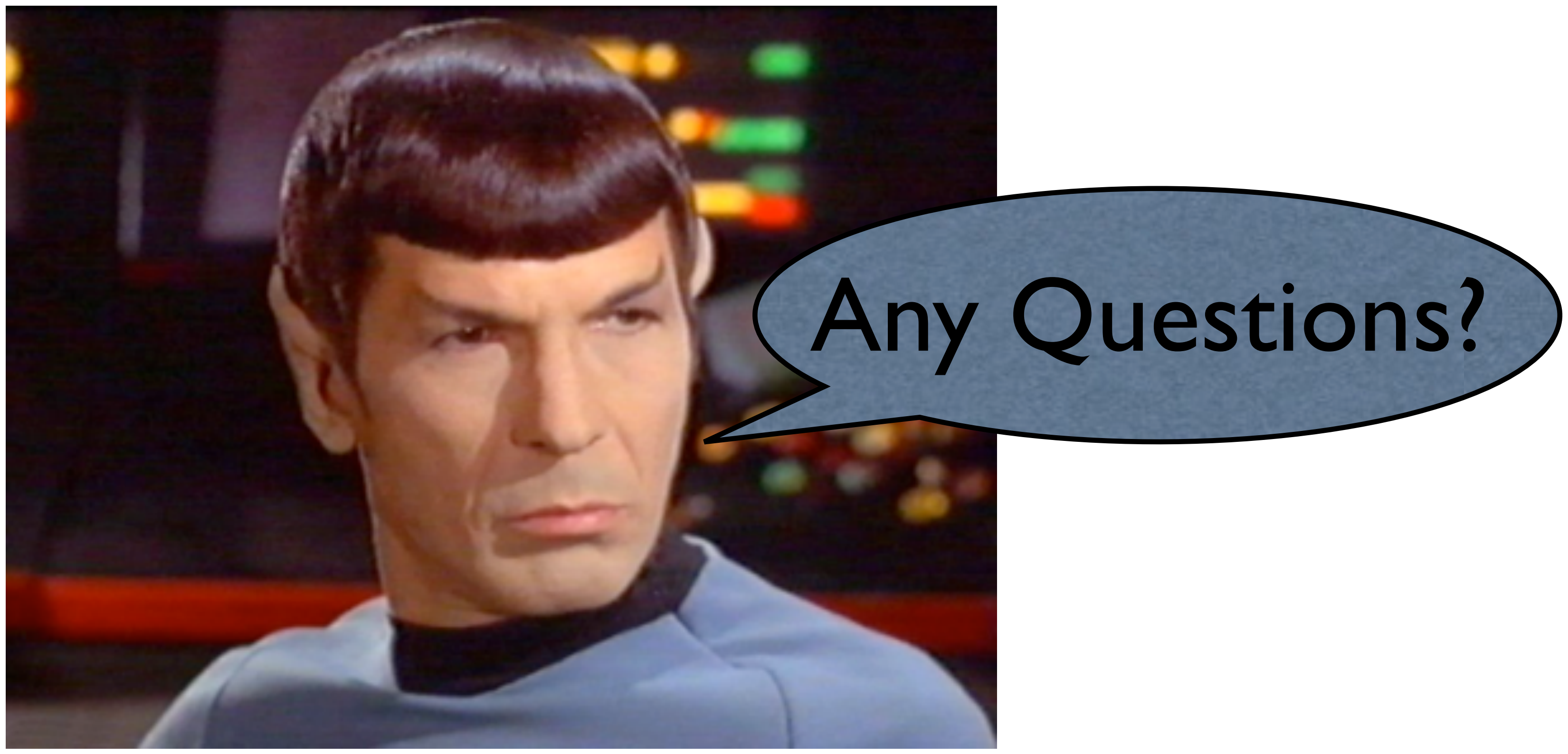
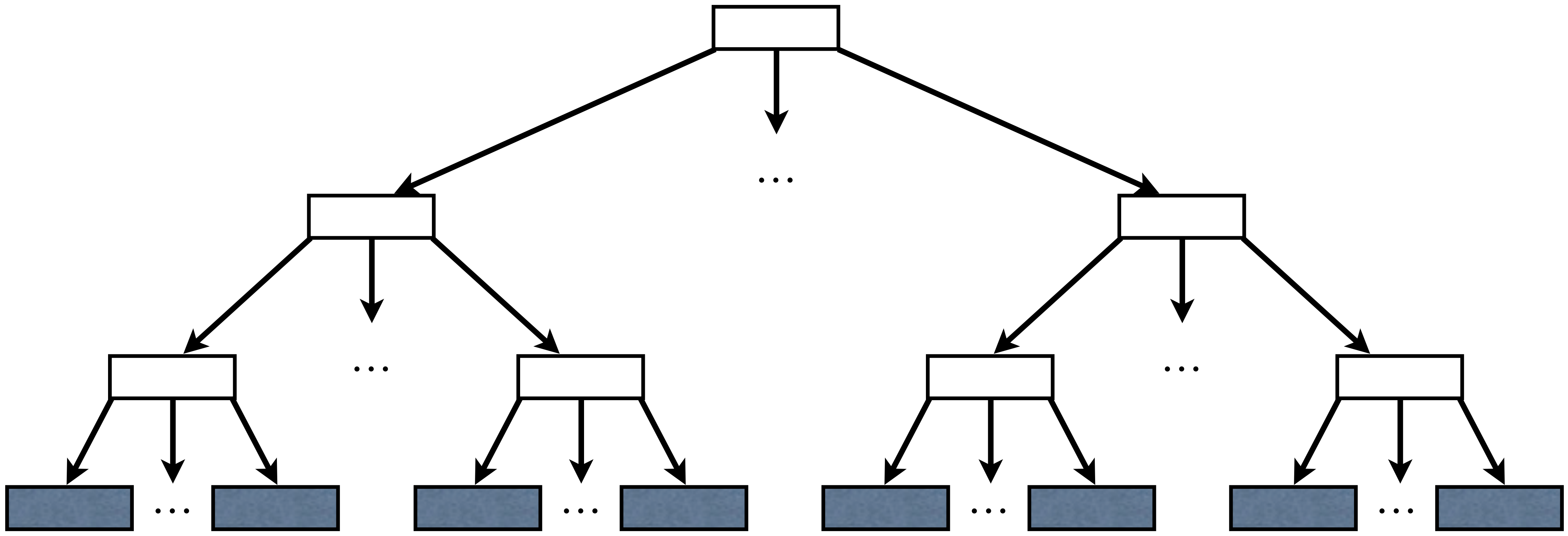


Image copyright: Paramount Pictures

Constructing an ISAM Index

Do you see any problems with this?

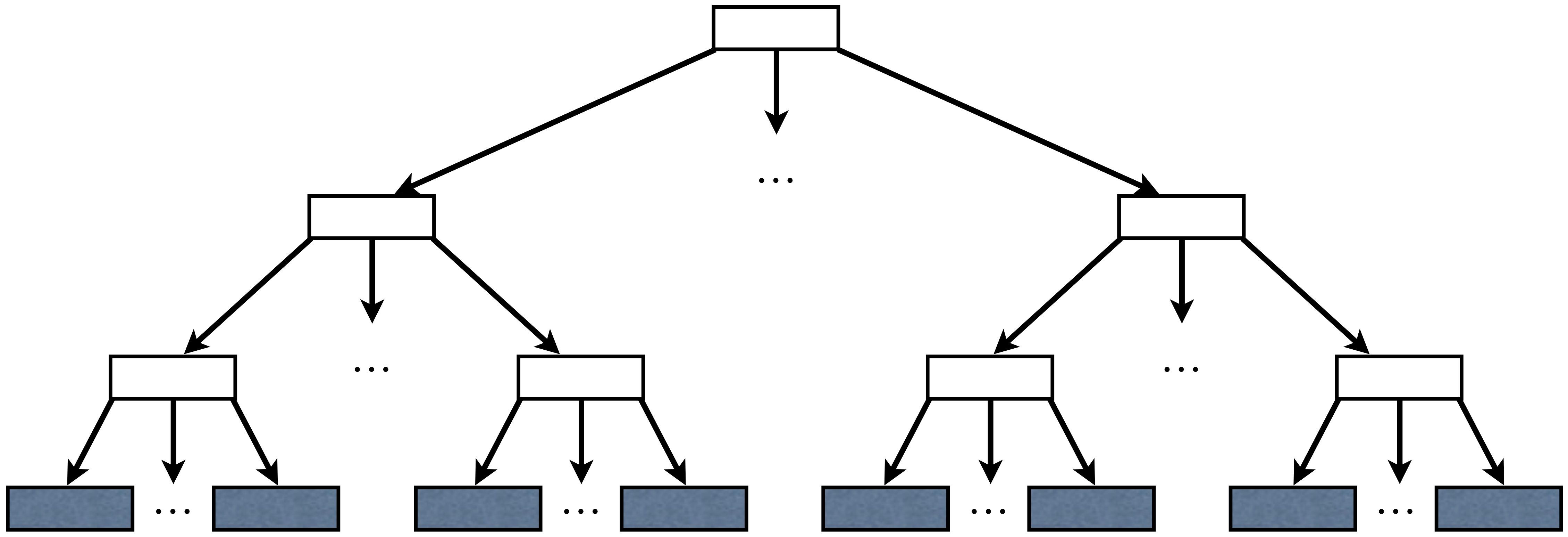


This strategy doesn't support efficient updates. Adding new records means re-sorting the leaf pages, which could mean rebuilding the entire index from scratch.

Updating an ISAM Index

1) When creating the index leave free space in each leaf page

2) *The index stays the same*, new data is added to the free space



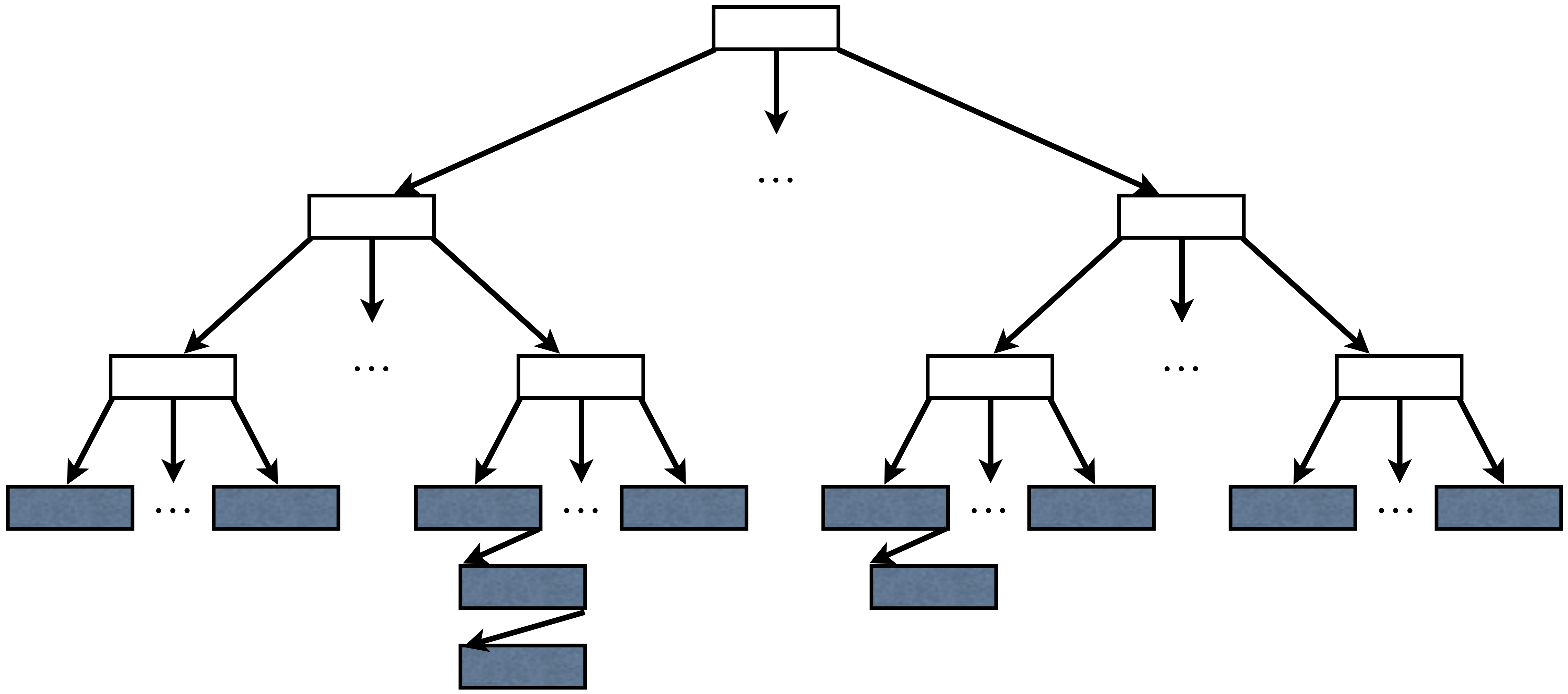
Deletion is easy - You get more free space on a page, although you might end up with empty pages.
Overflow pages are a problem. Ideally we want as few of them as possible. If we start getting lots of overflow pages, or lots of empty pages, we may want to rebuild the index.
Often DBAs will configure background processes to rebuild indexes during low-usage times.
(e.g., why many online games have regular monthly/weekly scheduled downtime events)

Updating an ISAM Index

1) When creating the index leave free space in each leaf page

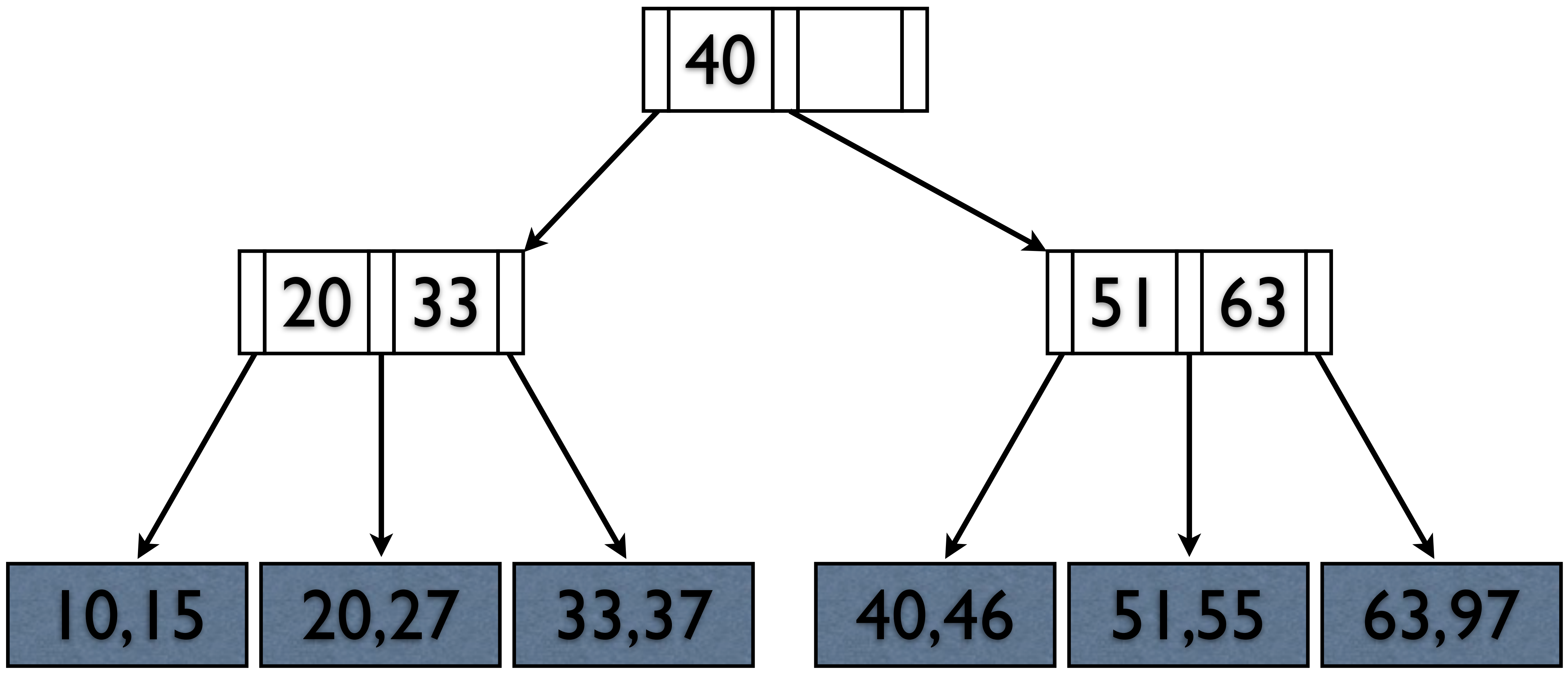
2) *The index stays the same*, new data is added to the free space

3) If a leaf page overflows, we create an overflow page (or more)



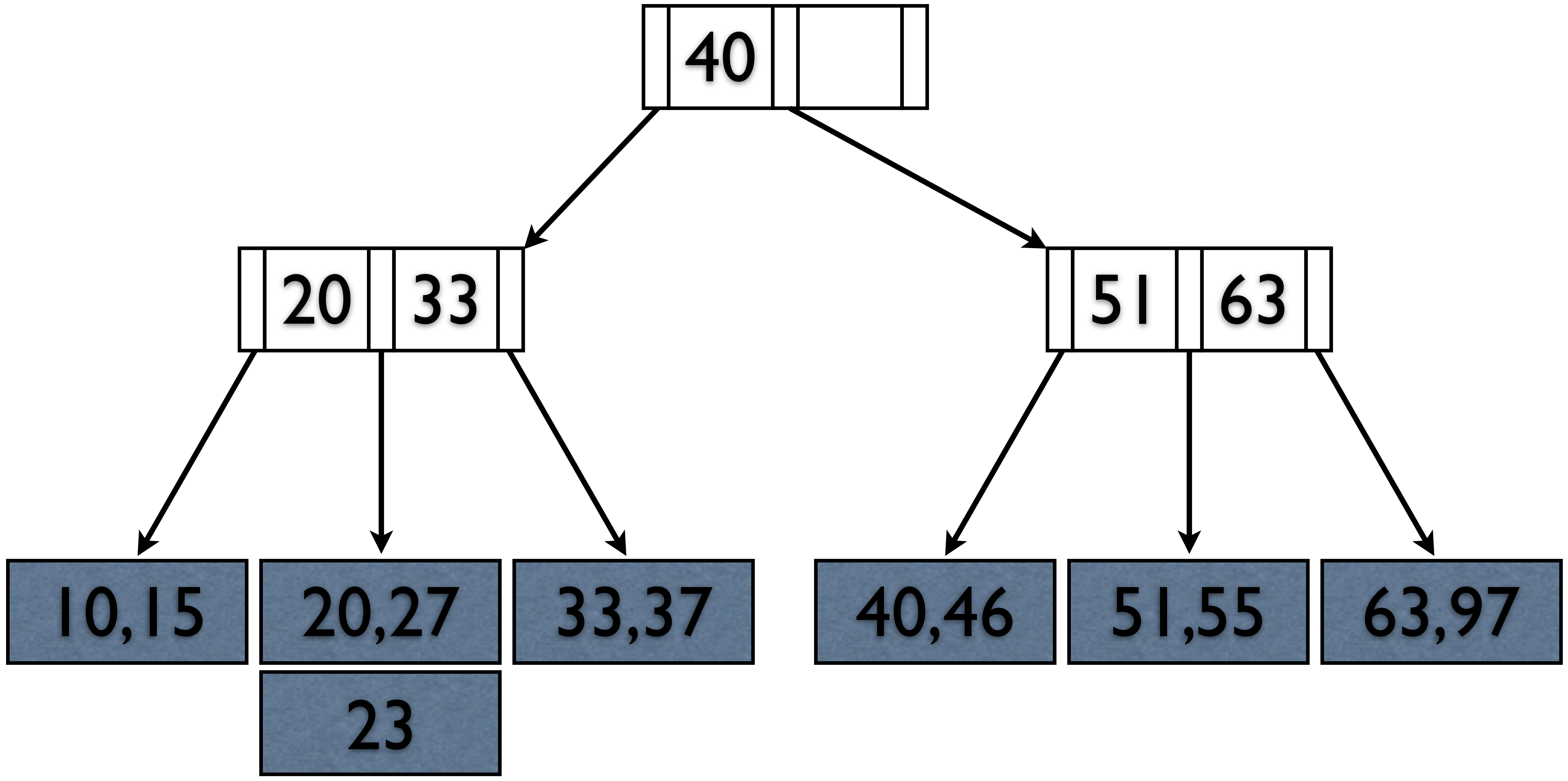
Deletion is easy - You get more free space on a page, although you might end up with empty pages.
Overflow pages are a problem. Ideally we want as few of them as possible. If we start getting lots of overflow pages, or lots of empty pages, we may want to rebuild the index.
Often DBAs will configure background processes to rebuild indexes during low-usage times.
(e.g., why many online games have regular monthly/weekly scheduled downtime events)

An Example ISAM



Insert: 23, 48, 41, 42 -- Insertions don't necessitate re-sorting the data pages
Delete: 97, 42, 51 -- Deletions free empty pages, but do not alter the index in any way

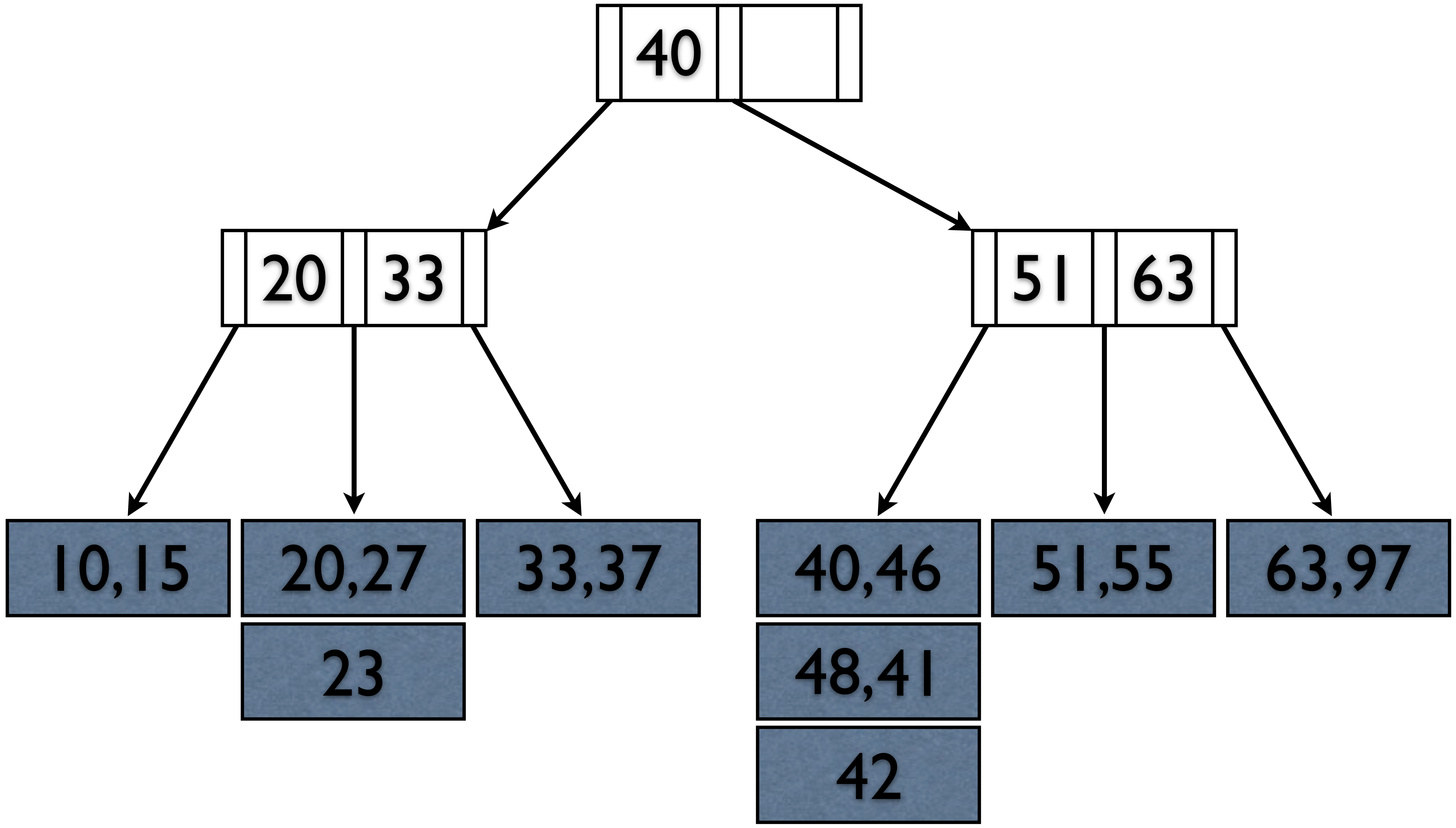
An Example ISAM



Monday, February 4, 13

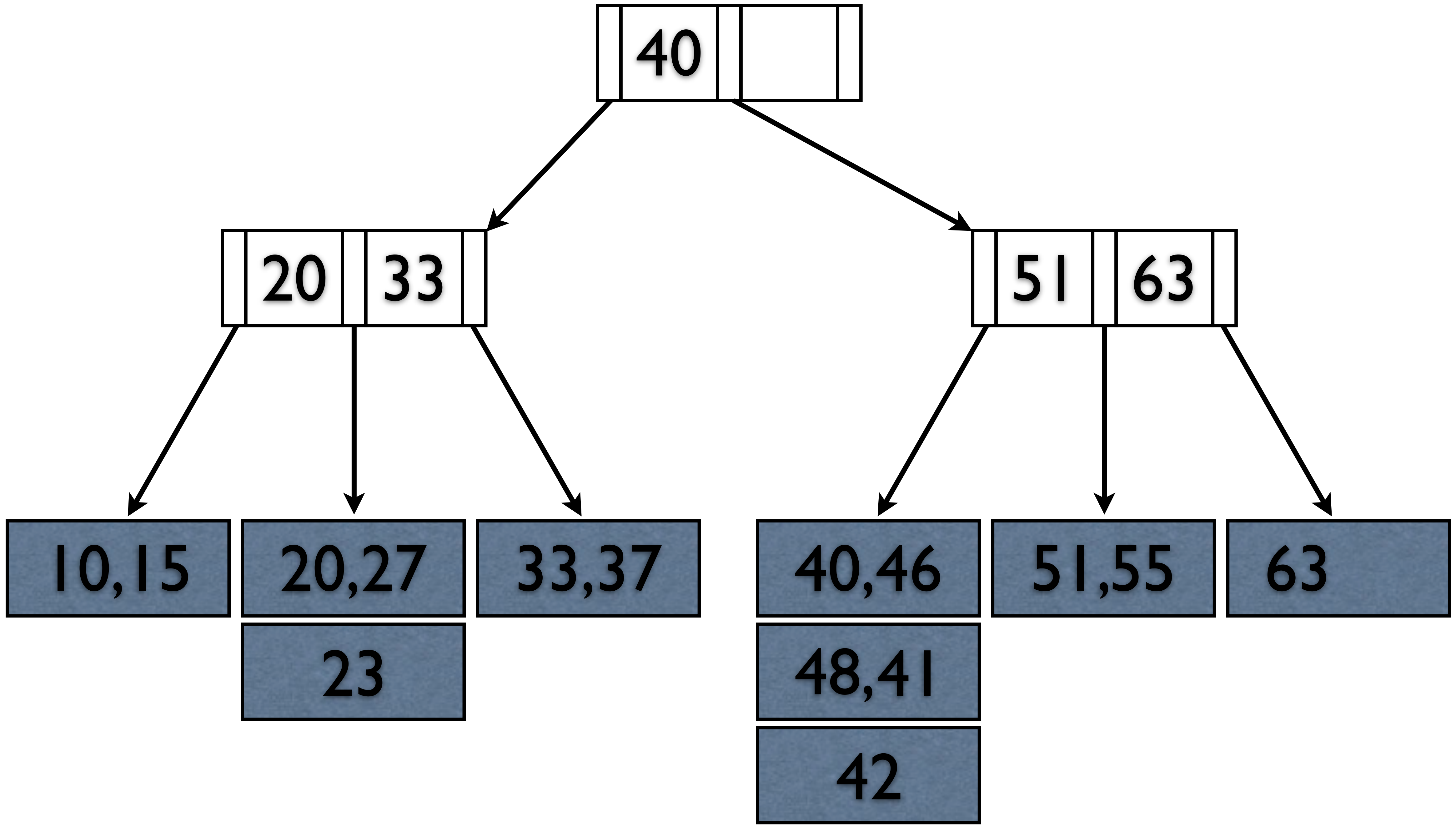
Insert: 23, 48, 41, 42 -- Insertions don't necessitate re-sorting the data pages
Delete: 97, 42, 51 -- Deletions free empty pages, but do not alter the index in any way

An Example ISAM



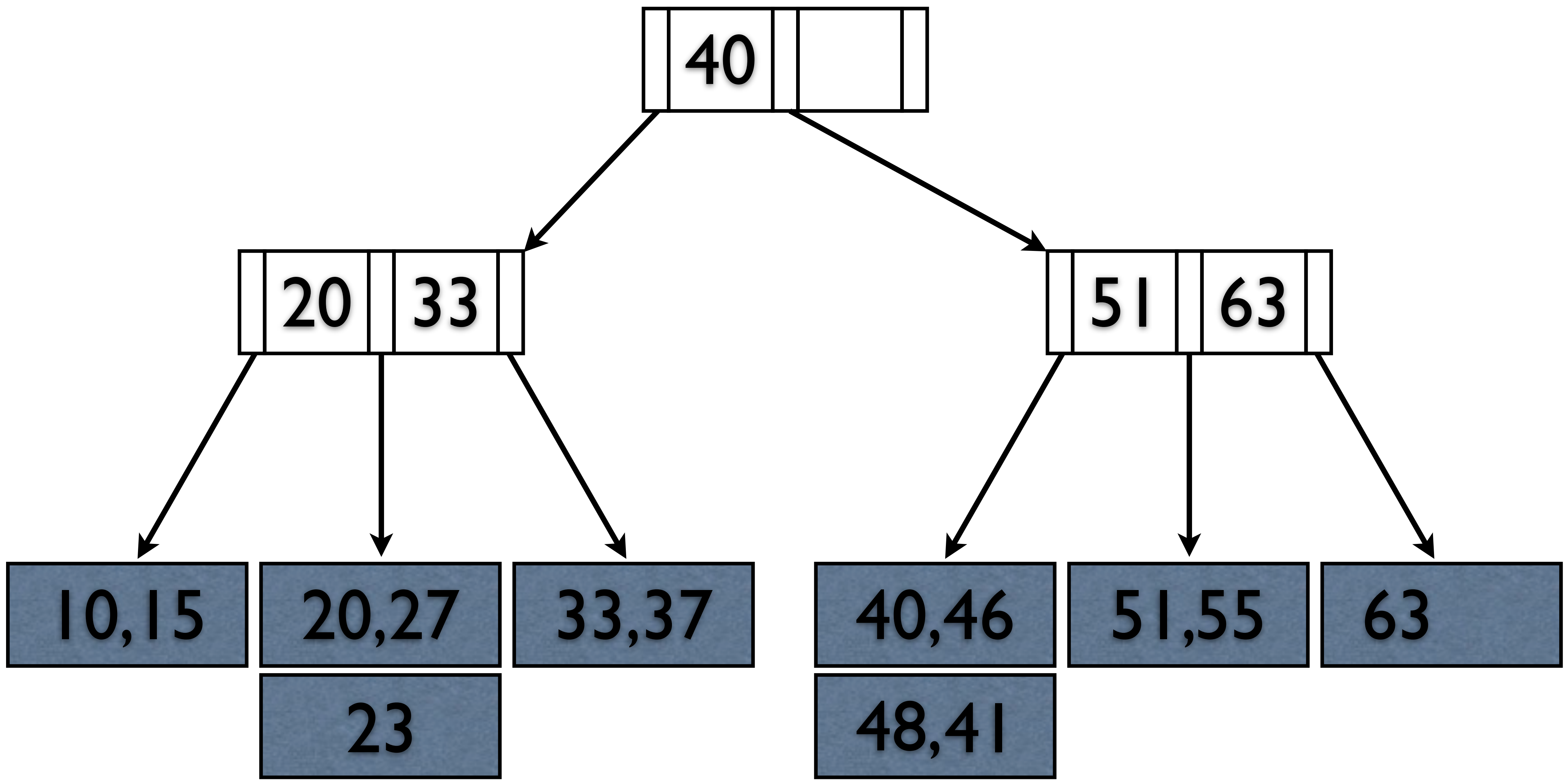
Insert: 23, 48, 41, 42 -- Insertions don't necessitate re-sorting the data pages
Delete: 97, 42, 51 -- Deletions free empty pages, but do not alter the index in any way

An Example ISAM



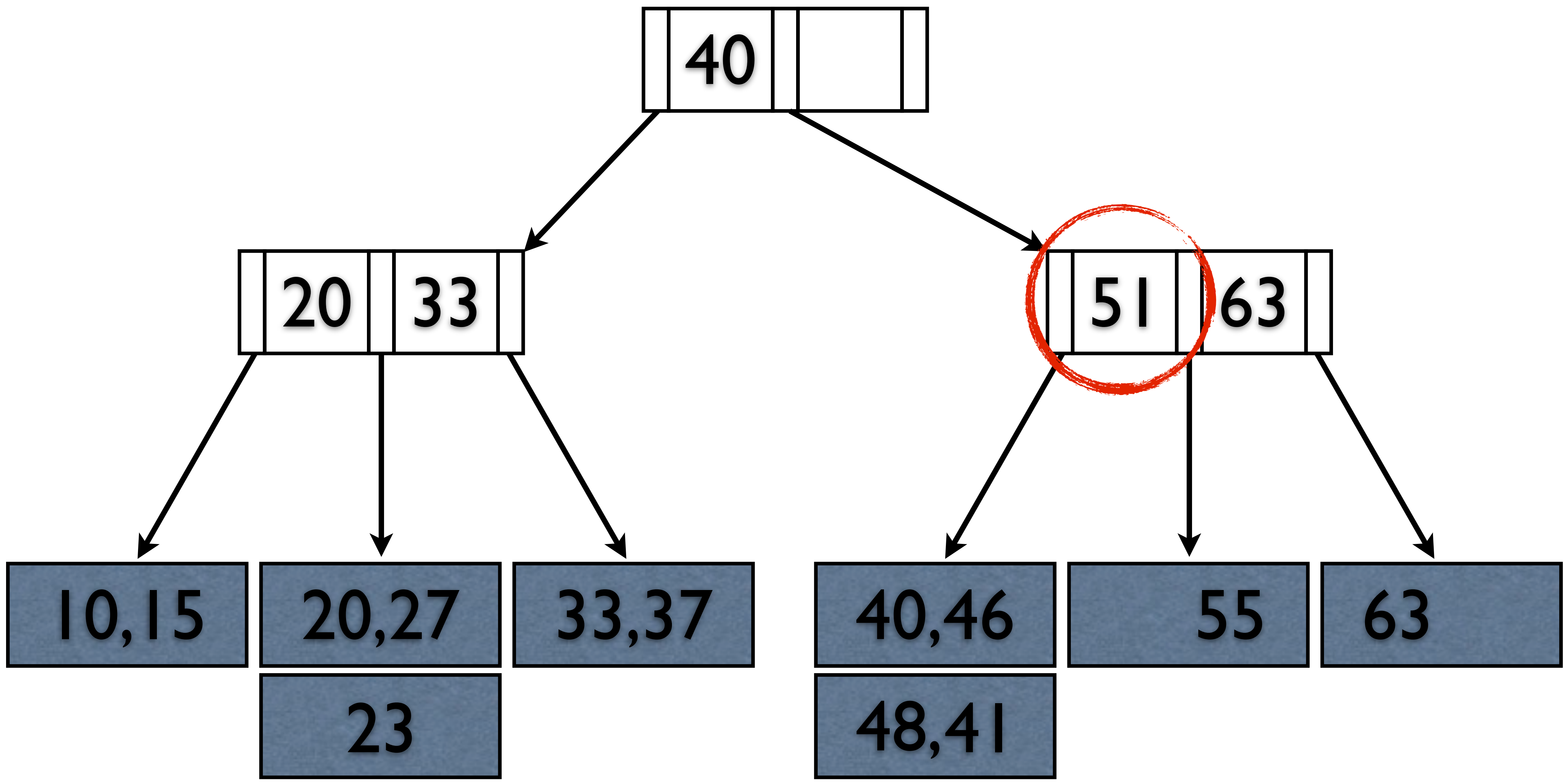
Insert: 23, 48, 41, 42 -- Insertions don't necessitate re-sorting the data pages
Delete: 97, 42, 51 -- Deletions free empty pages, but do not alter the index in any way

An Example ISAM

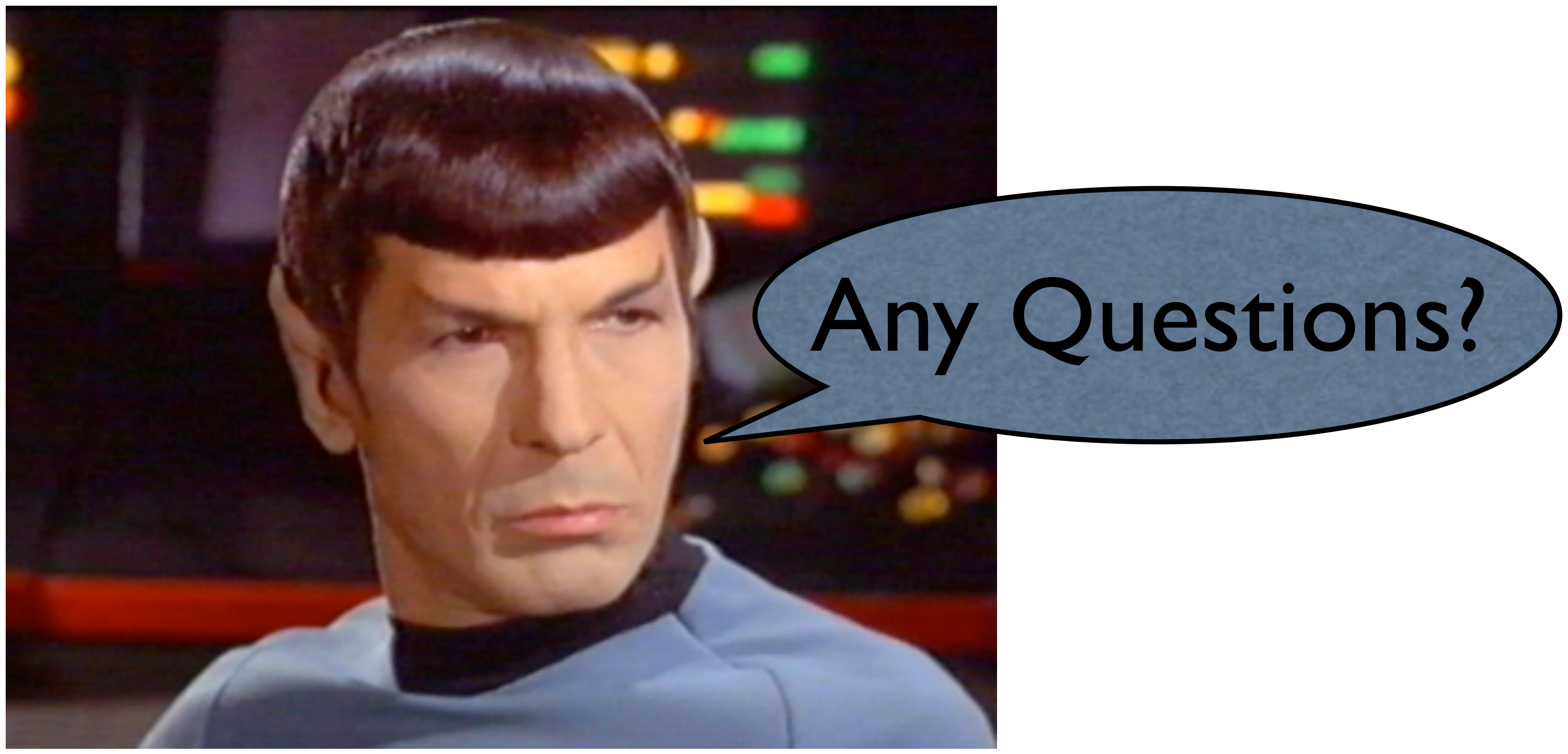


Insert: 23, 48, 41, 42 -- Insertions don't necessitate re-sorting the data pages
Delete: 97, 42, 51 -- Deletions free empty pages, but do not alter the index in any way

An Example ISAM



Insert: 23, 48, 41, 42 -- Insertions don't necessitate re-sorting the data pages
Delete: 97, 42, 51 -- Deletions free empty pages, but do not alter the index in any way


Any Questions?

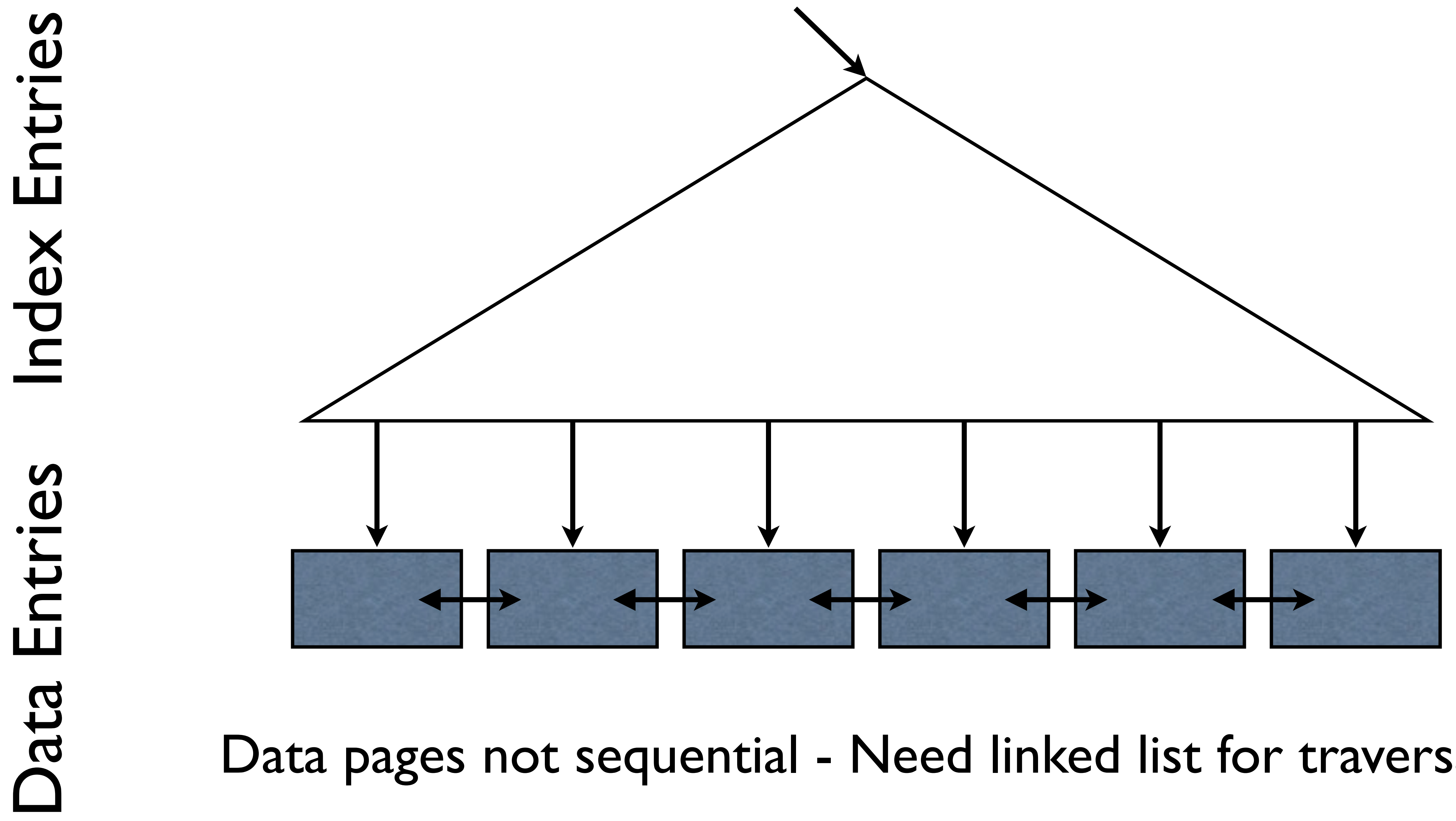
B+ Trees

- ISAM handles updates poorly: Lots of free pages/long overflow page chains are created.
- We need a (efficient) way to keep the index synched with the data.
- B+ Tree: A widely used ‘Balanced’ tree.
  - As the indexed data changes, the tree is dynamically rearranged (‘balanced’).

B+ Trees

- Operations on the tree keep it balanced
  - Each leaf contains the same amt. of data.
- Minimum occupancy of 50% for each node.
  - Each node contains d ≤ m ≤ 2d entries.
  - Occupancy often ignored for deletion.
- Tree is balanced
  - The *depth* of each leaf node is identical.

B+ Trees

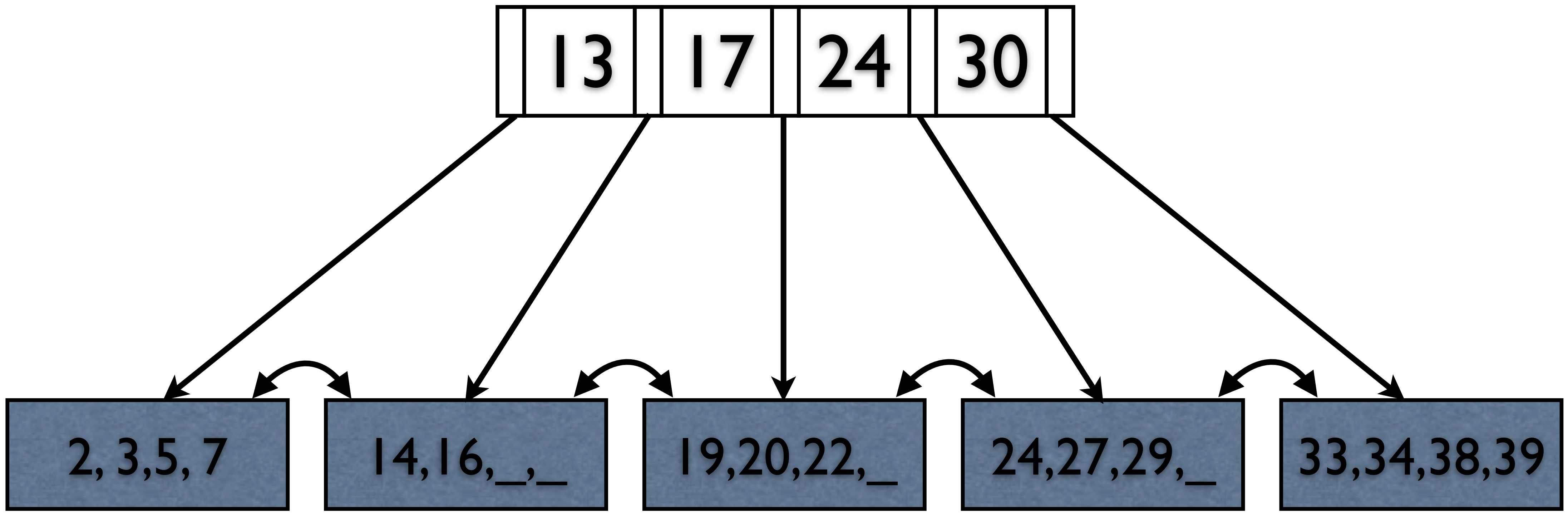


Data pages not sequential - Need linked list for traversals

B+ Trees

Search proceeds as in ISAM via key comparisons

Find 5. Find 15. Find [24,∞)

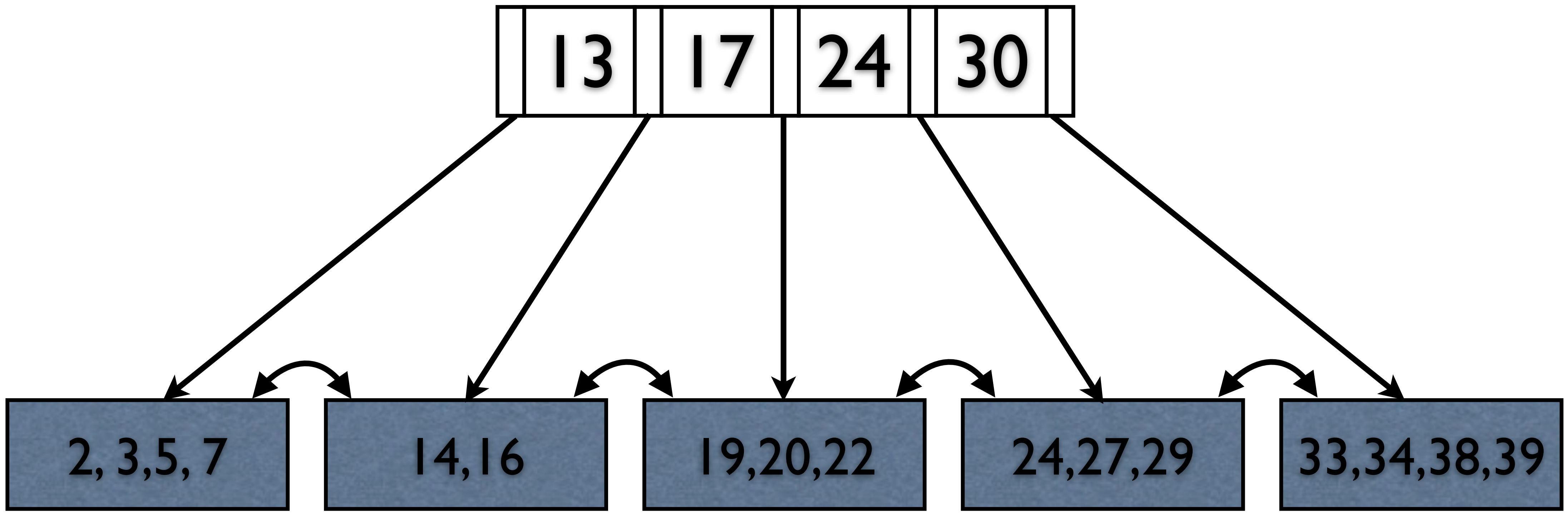


To find 5, we start at the index root: 5<13, so the first pointer is appropriate. 5 exists on the data page so we return it.
To find 15 we start at the index root: 13<15<17, so the second pointer is appropriate. 15 does not exist on the data page, so it does not exist
To find the range [24,∞), we start at the index root: Infinity doesn't require a search, so we just find the first page with a 24. As luck would have it, 24 is the lower bound on pointer 4.

Inserting into B+ Trees

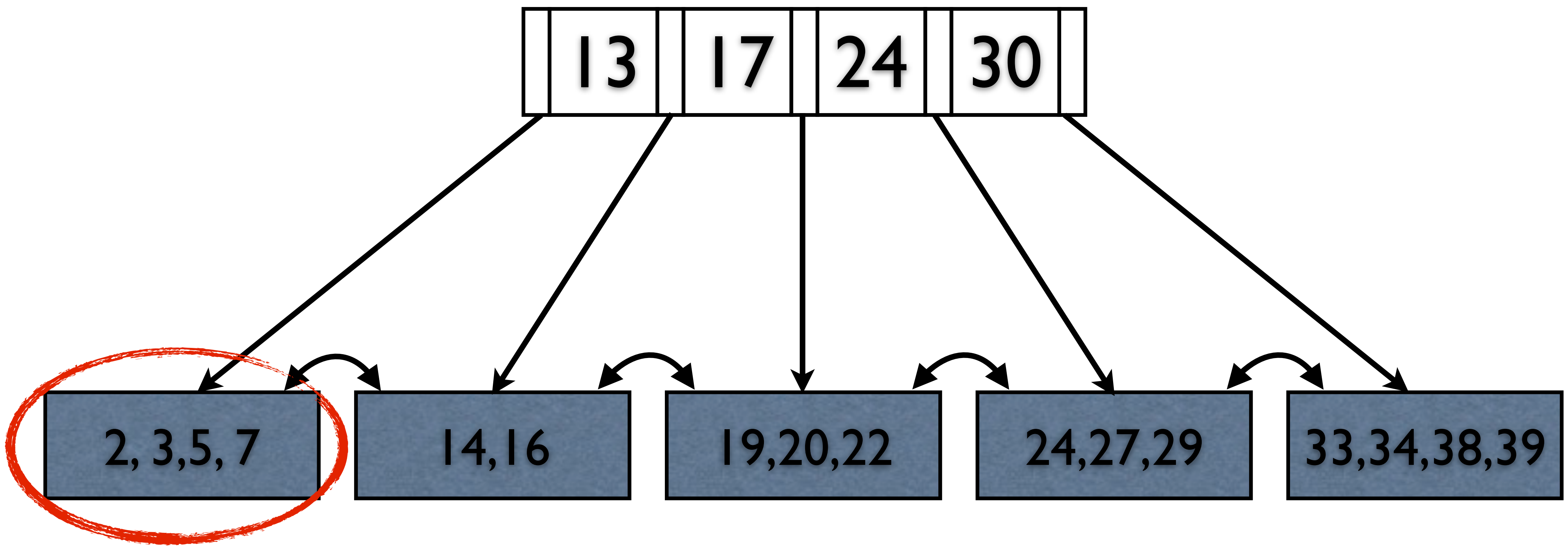
- Find the correct leaf L
- Put data entry into L
  - If L has space, done!
  - Otherwise need to split L into L and new L2
    - Split entries by median value (50% to each)
    - Insert index entry pointing to L2
    - **Copy** the median value as a separator value.
- Splits can happen recursively
  - Split on median separator value in the index.
  - **Push** up the median rather than copying it.
- Root splits increase the depth of the tree.

Inserting into B+ Trees



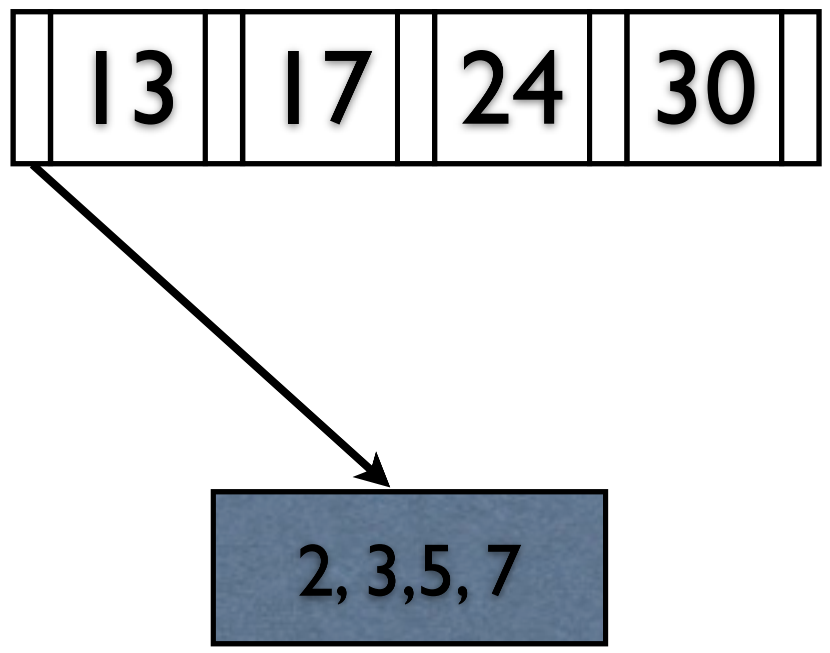
Insert 8

Inserting into B+ Trees



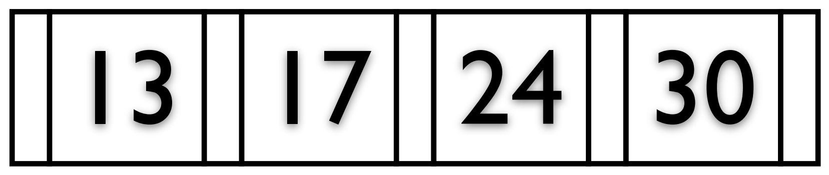
Insert 8

Inserting into B+ Trees



Insert 8

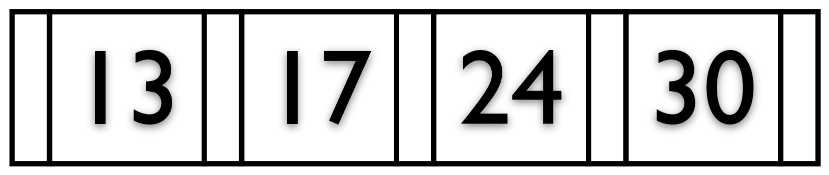
Inserting into B+ Trees



2, 3,5,7,8

Insert 8

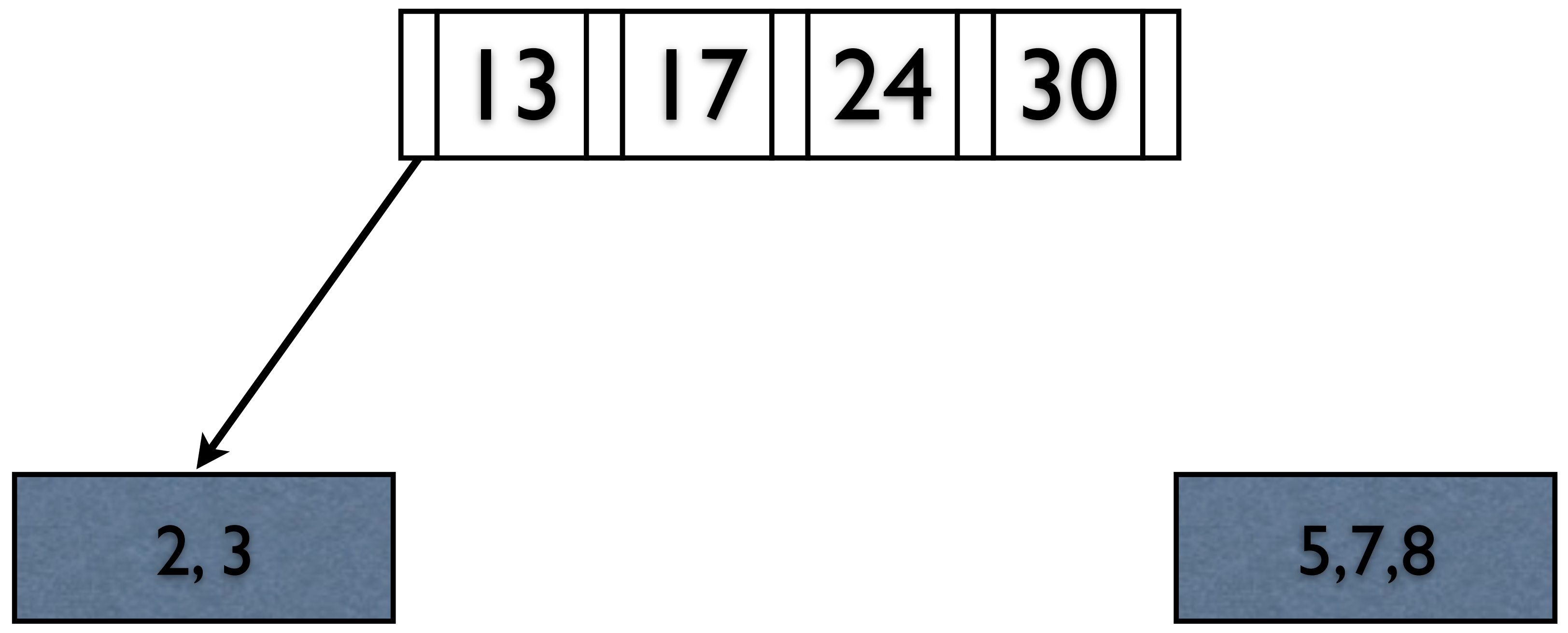
Inserting into B+ Trees



2, 3,5,7,8

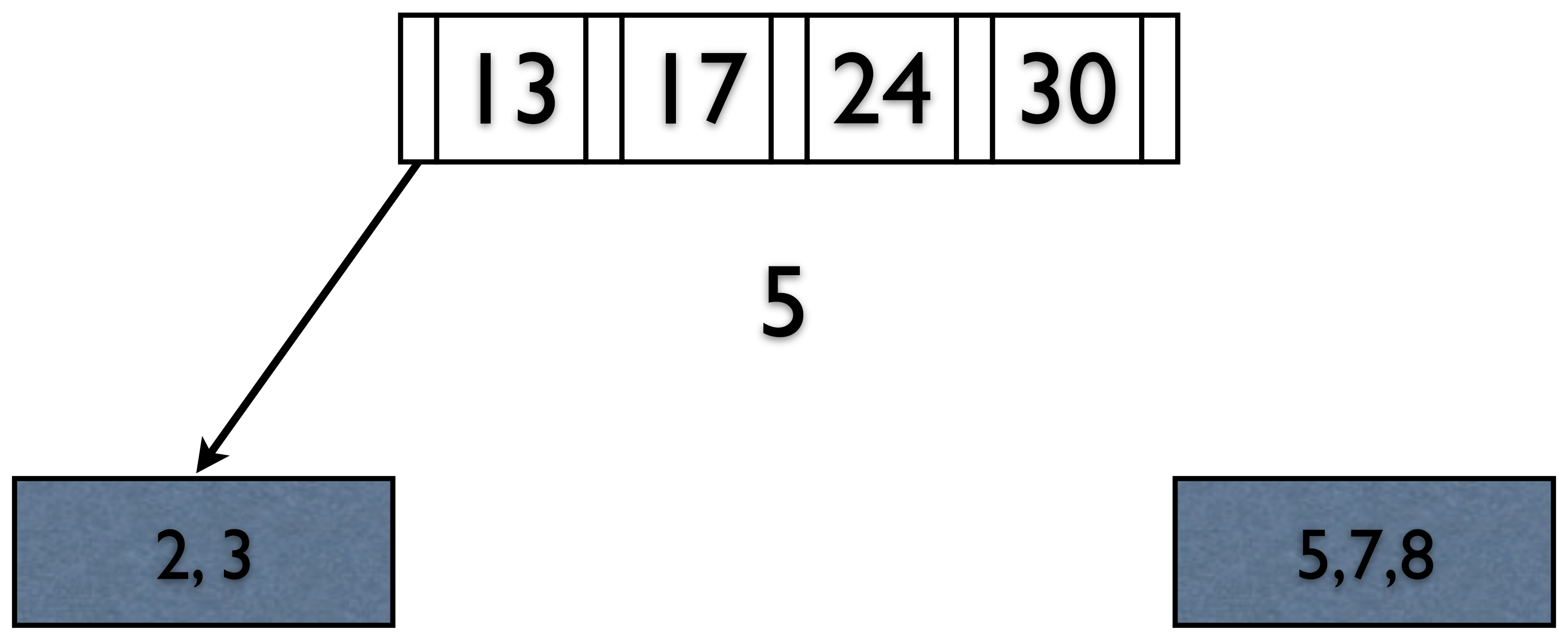
Insert 8

Inserting into B+ Trees



Insert 8

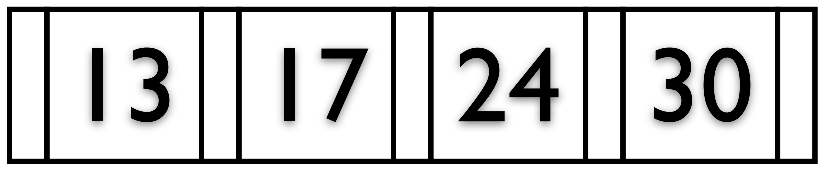
Inserting into B+ Trees



Copy <5> into parent index

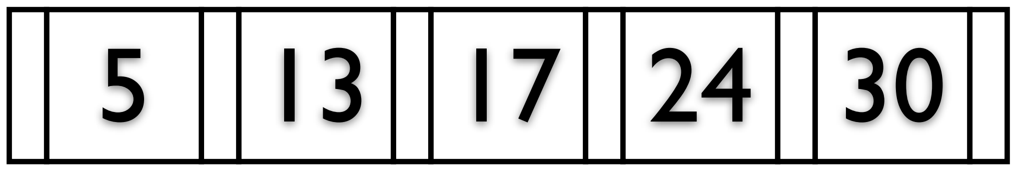
Insert 8

Inserting into B+ Trees



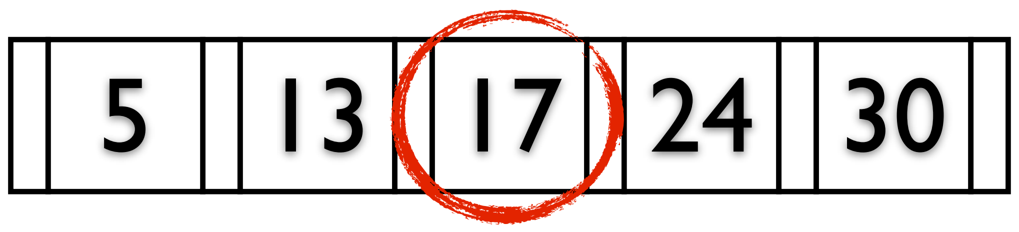
Copy <5> into parent index

Inserting into B+ Trees



Copy <5> into parent index

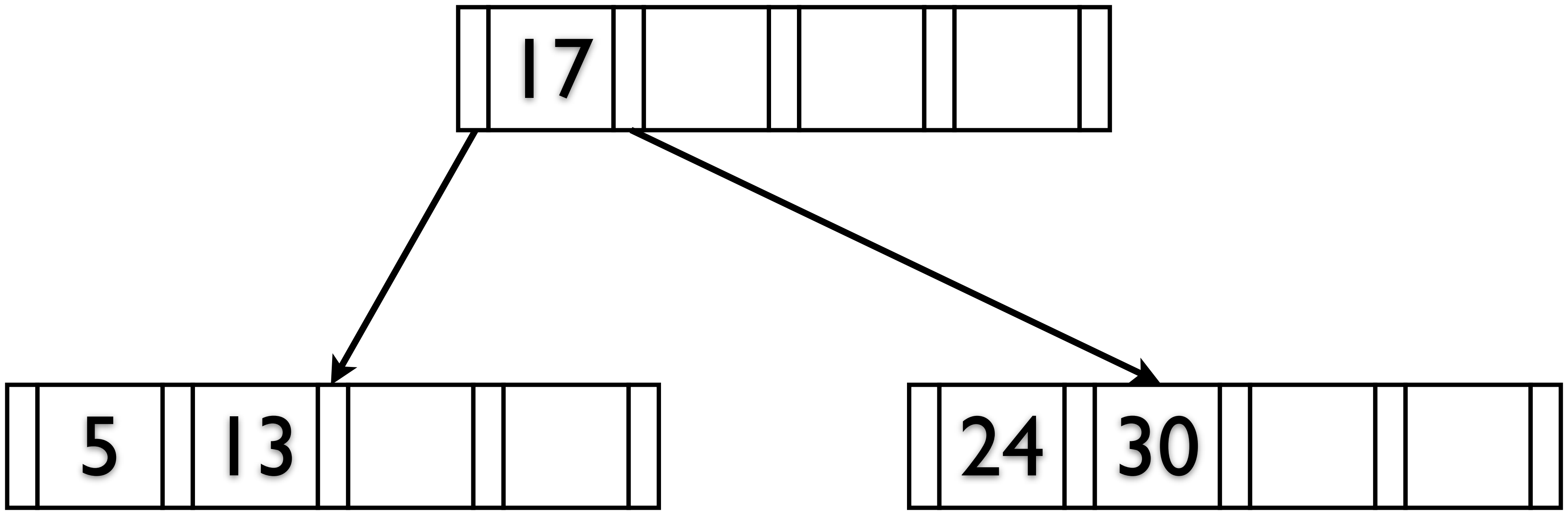
Inserting into B+ Trees



Move <17> into parent index : Root Split!

Copy <5> into parent index

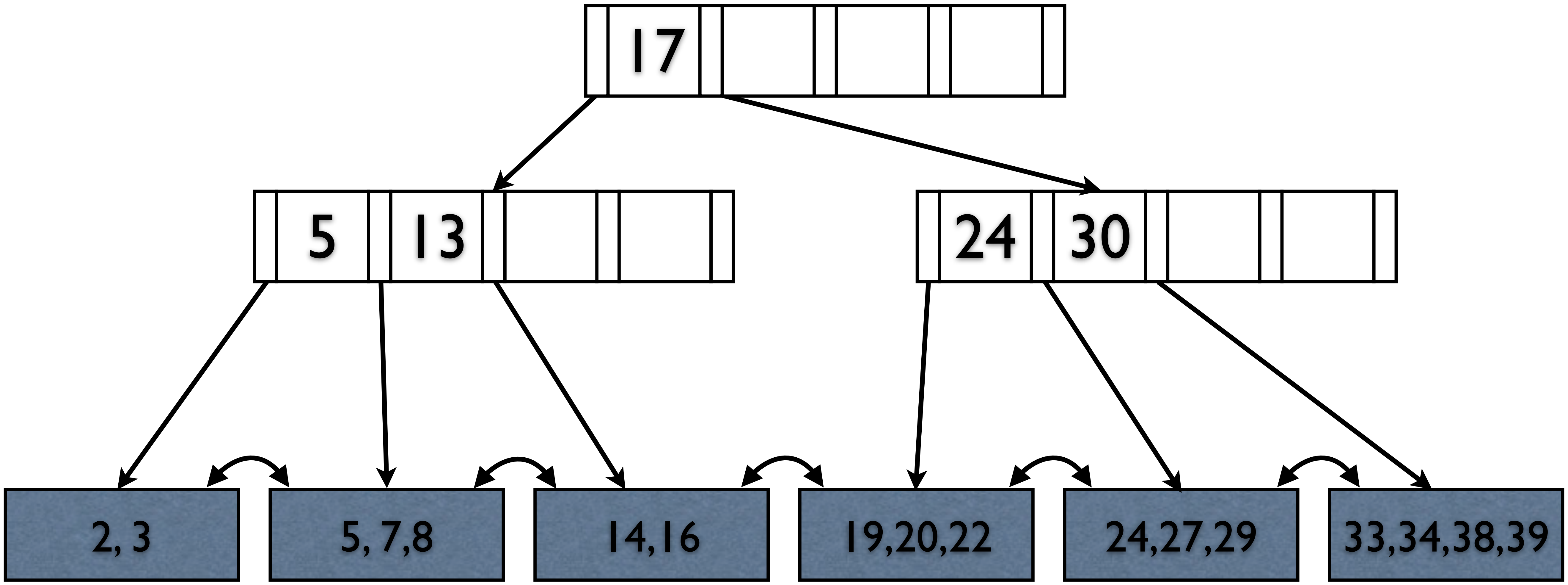
Inserting into B+ Trees



Move <17> into parent index : Root Split!

Copy <5> into parent index

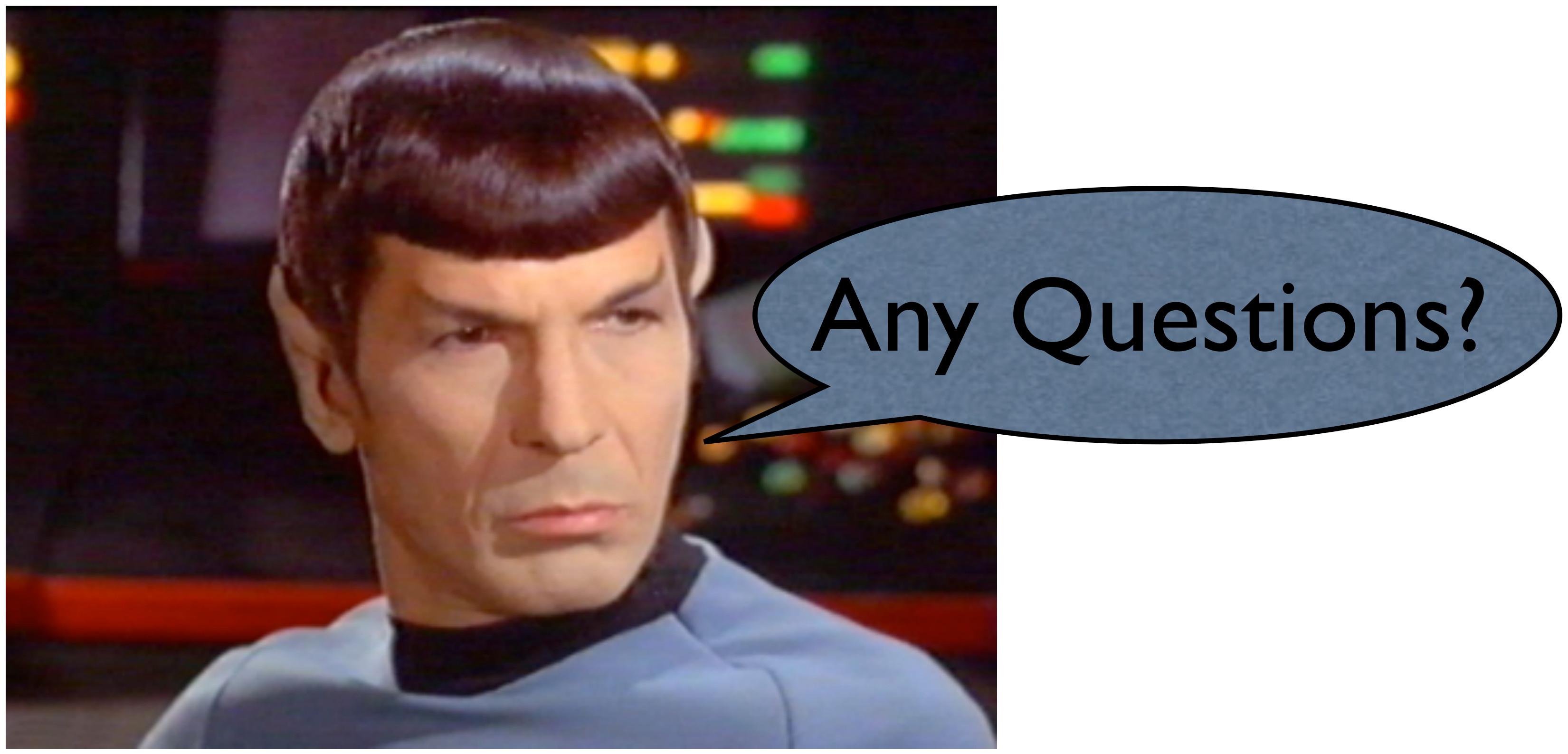
Inserting into B+ Trees



Why do we move, rather than copy the 17?

Are we guaranteed to satisfy our occupancy guarantee?

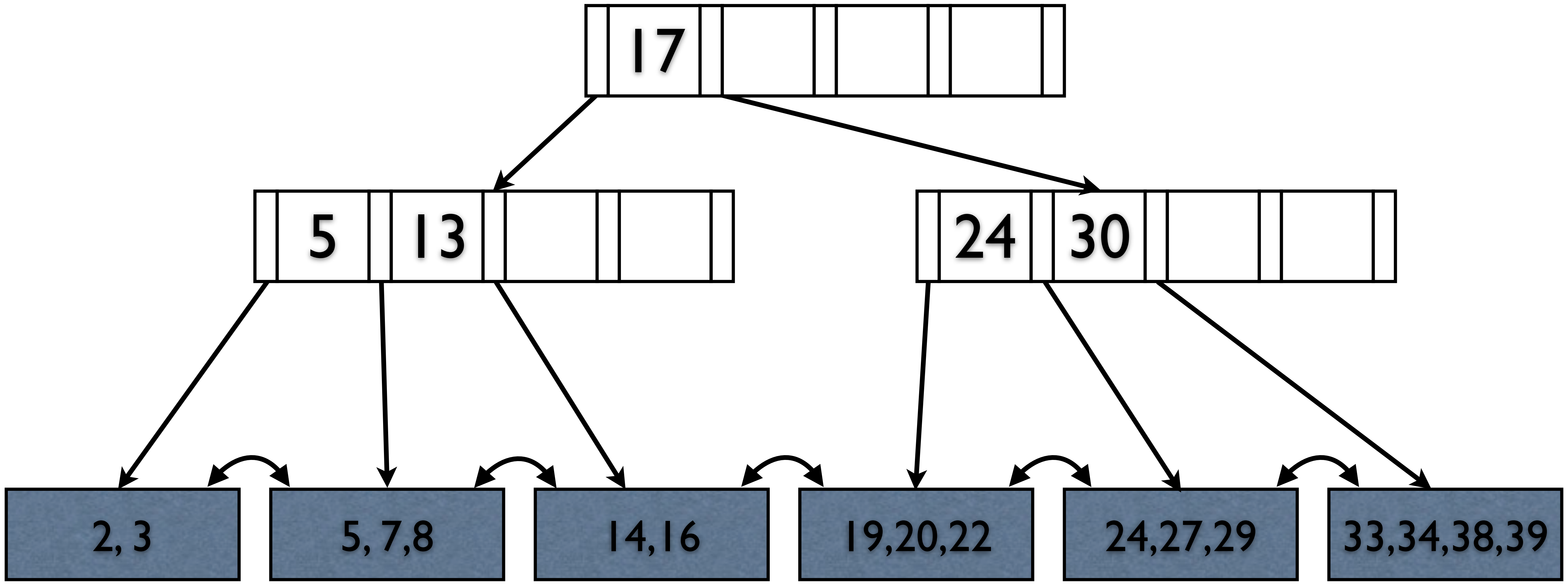
We have to visit the parent page on every search. Once we traverse into the left-hand side of this tree, we know that the upper bound on everything in this page is 17 (and similarly, the lower bound on everything on the right-hand branch).
The occupancy guarantee is satisfied, because we only split when the page is full (2d entries), and we always split in half, so we never get fewer than d entries in the resulting pages


Any Questions?

Deleting from B+ Trees

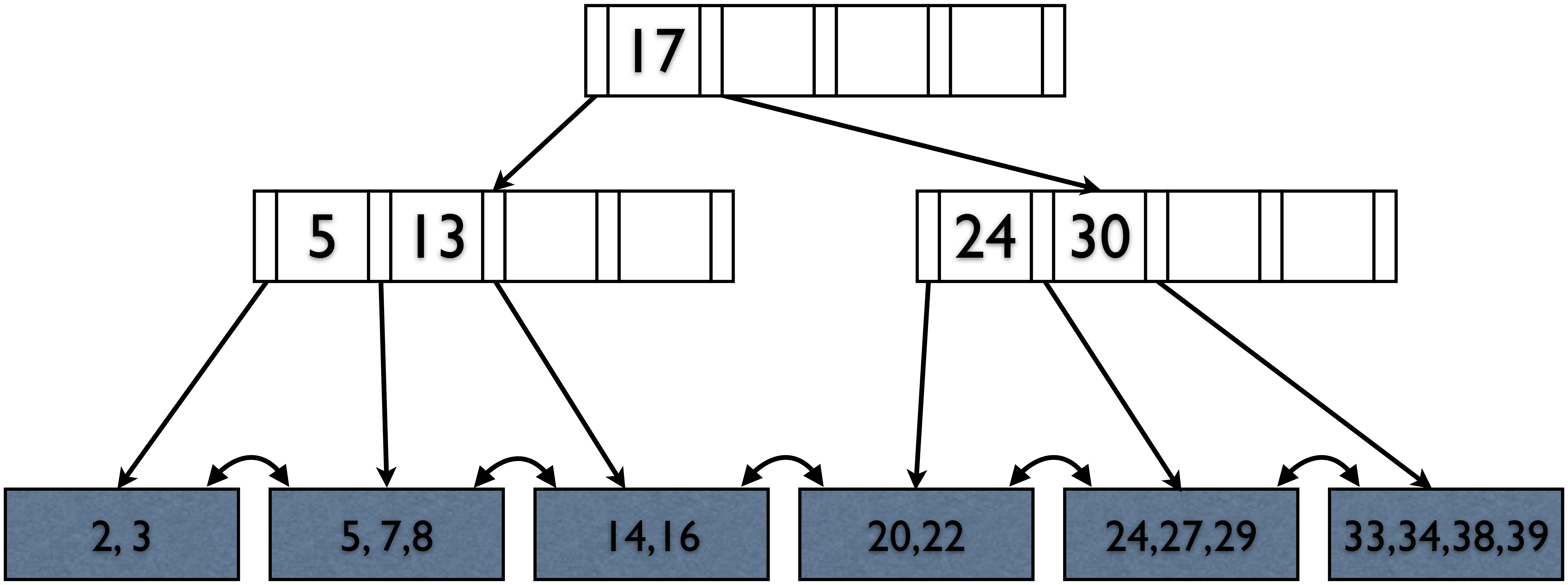
- Find the leaf page of the deleted entry
- Remove the entry
  - If the page is at least half-full, done!
  - If the page has only d-1 entries
    - Try to redistribute entries between siblings (children of the same parent)
    - If adjacent siblings have d entries, merge L and one of its siblings.
- If merged, delete the entry from L's parent.
- Merge could propagate up to the root.

Deleting from B+ Trees



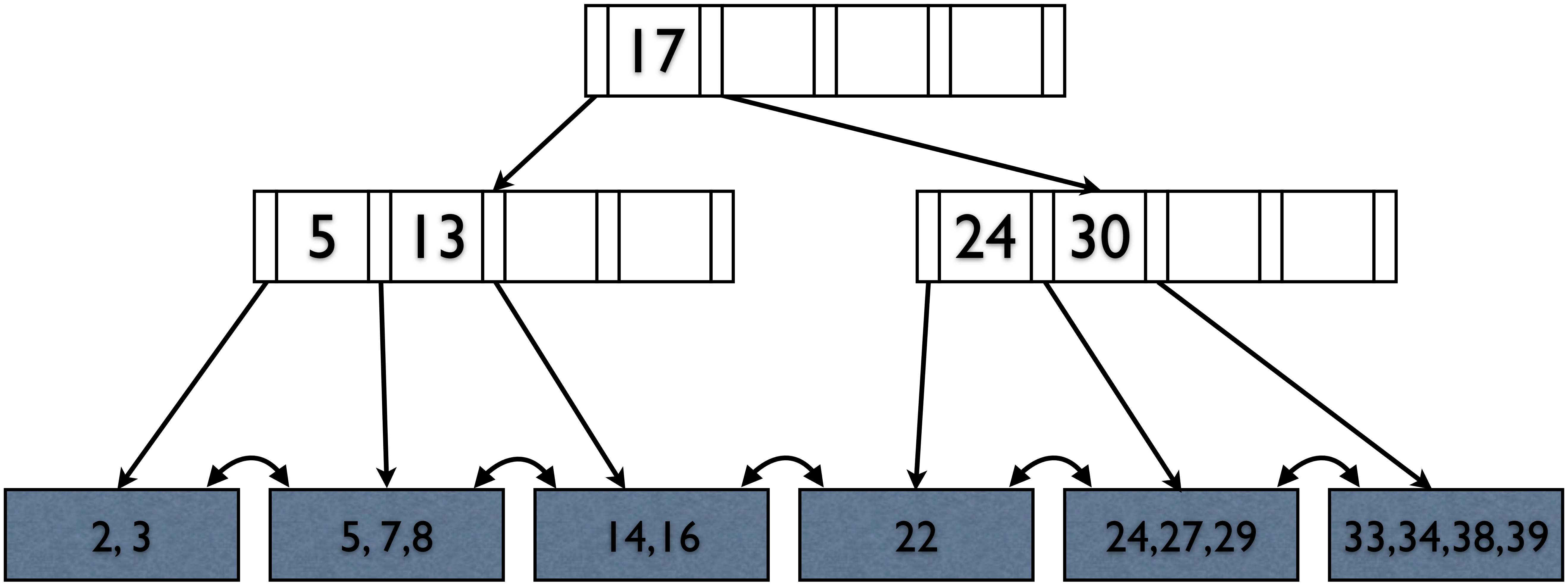
Delete 19 Delete 20

Deleting from B+ Trees



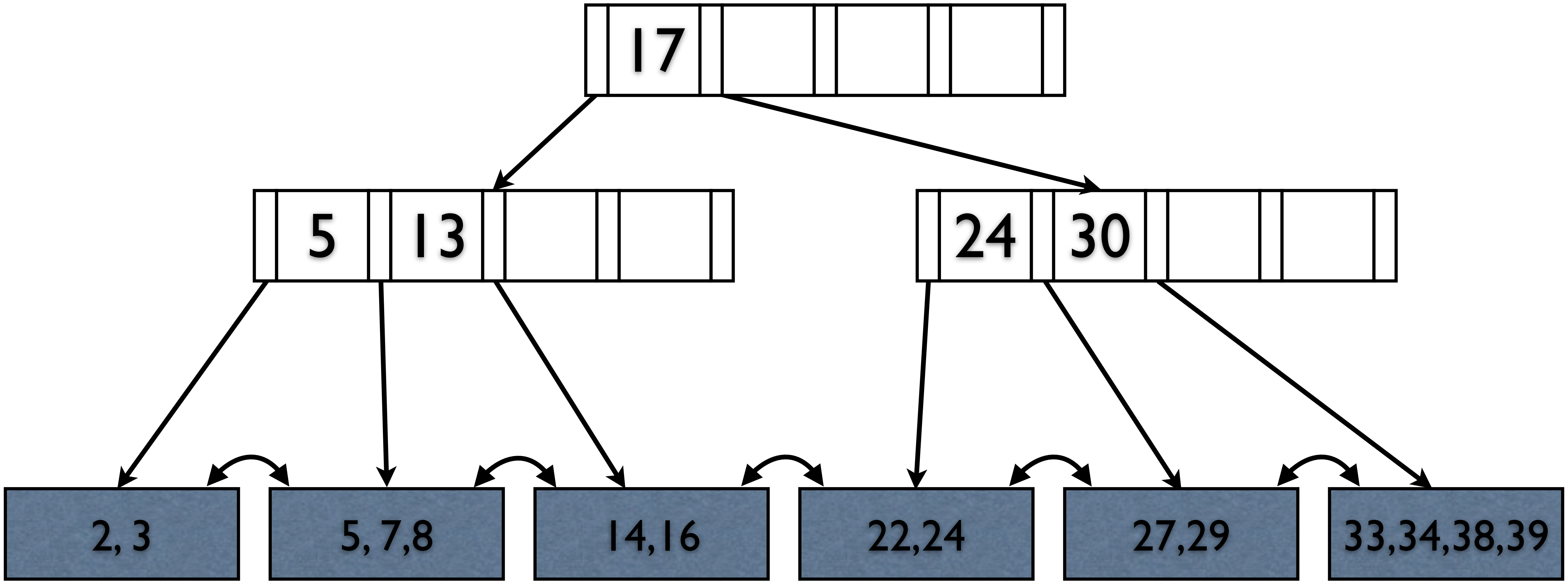
Delete 19 Delete 20

Deleting from B+ Trees



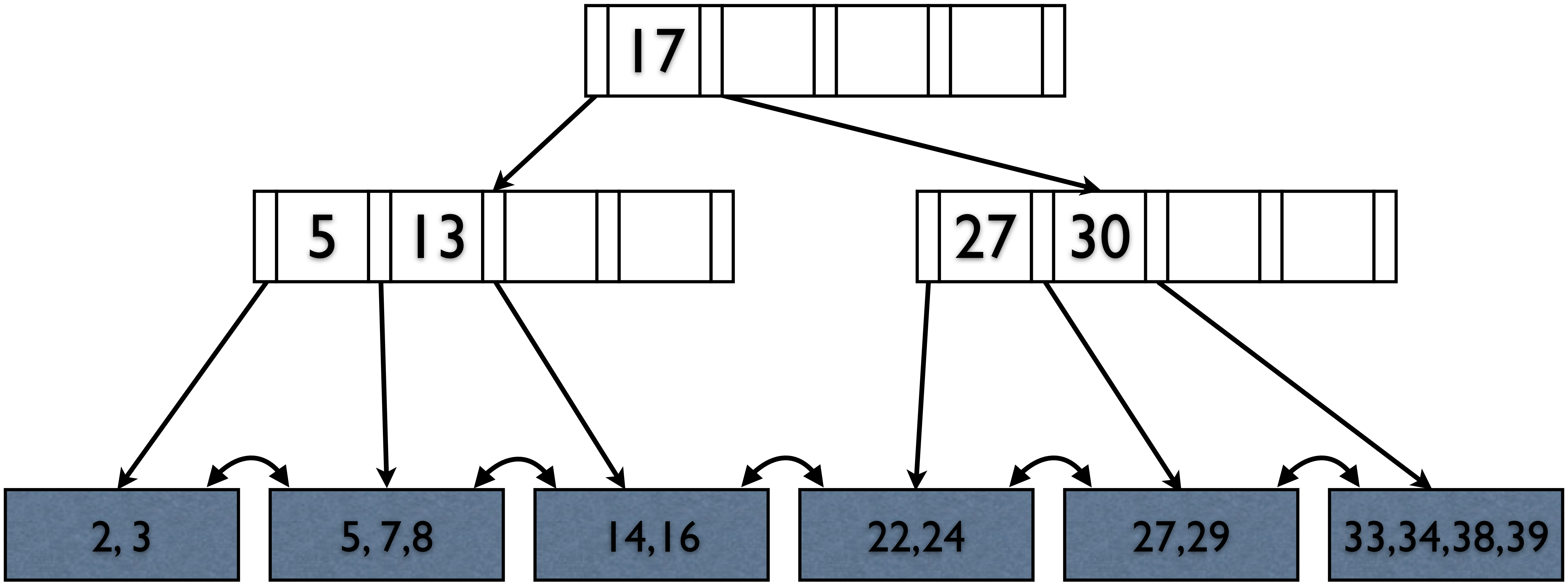
Delete 19 Delete 20

Deleting from B+ Trees



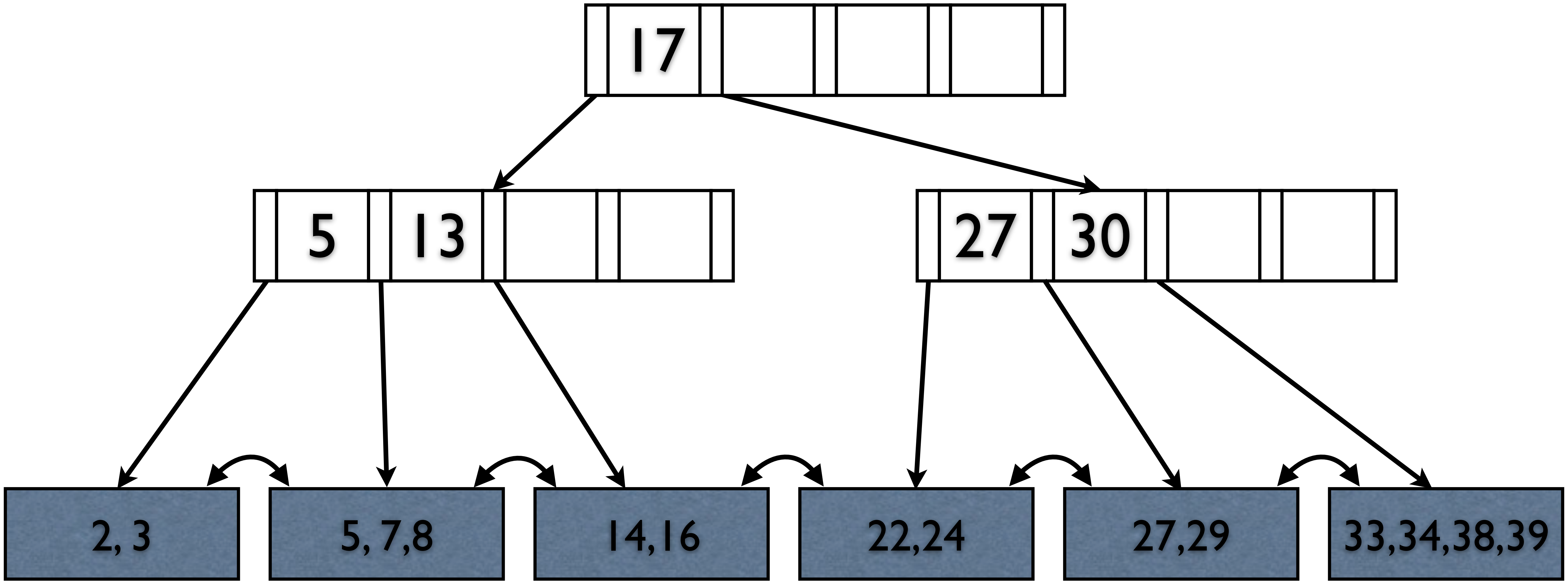
Delete 19 Delete 20

Deleting from B+ Trees



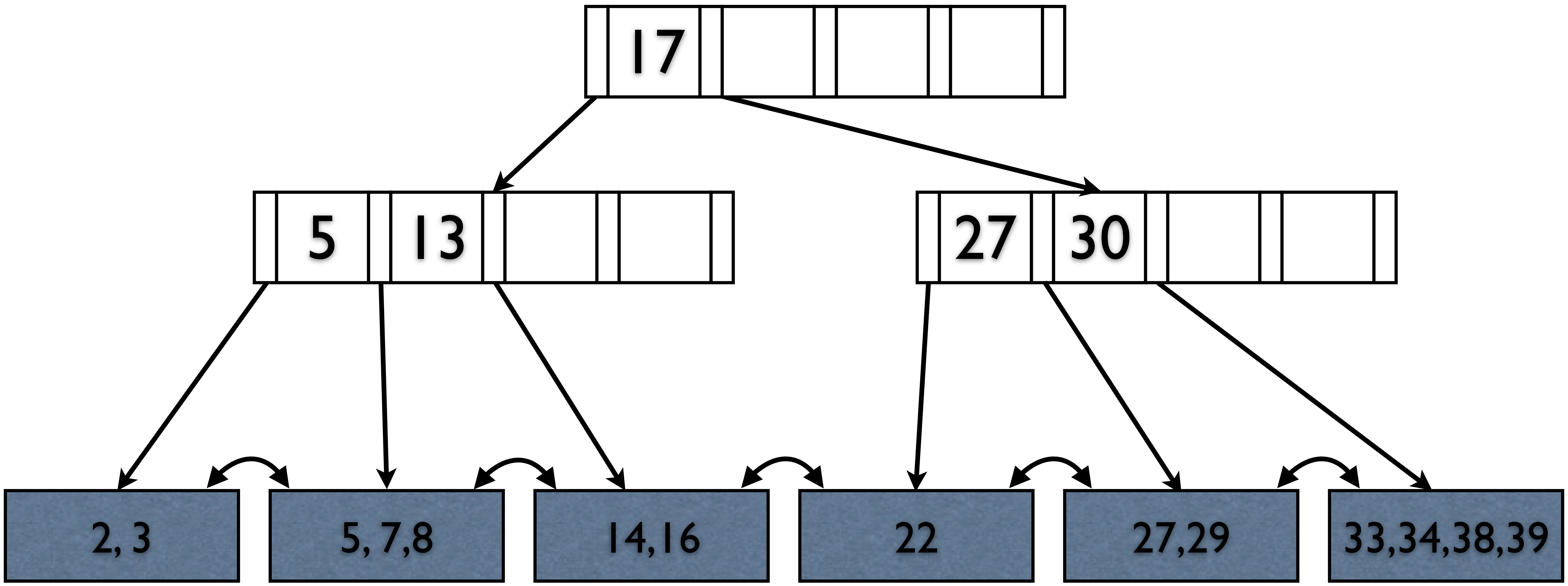
Delete 19 Delete 20

Deleting from B+ Trees



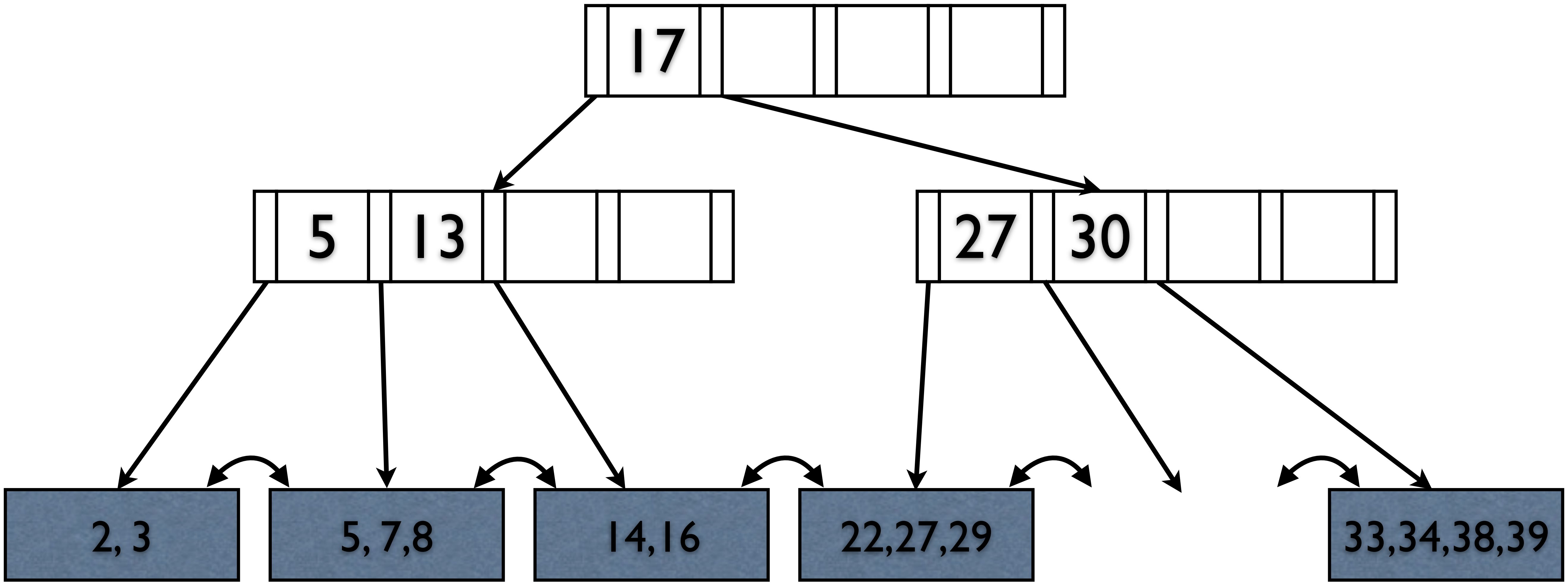
Delete 24

Deleting from B+ Trees



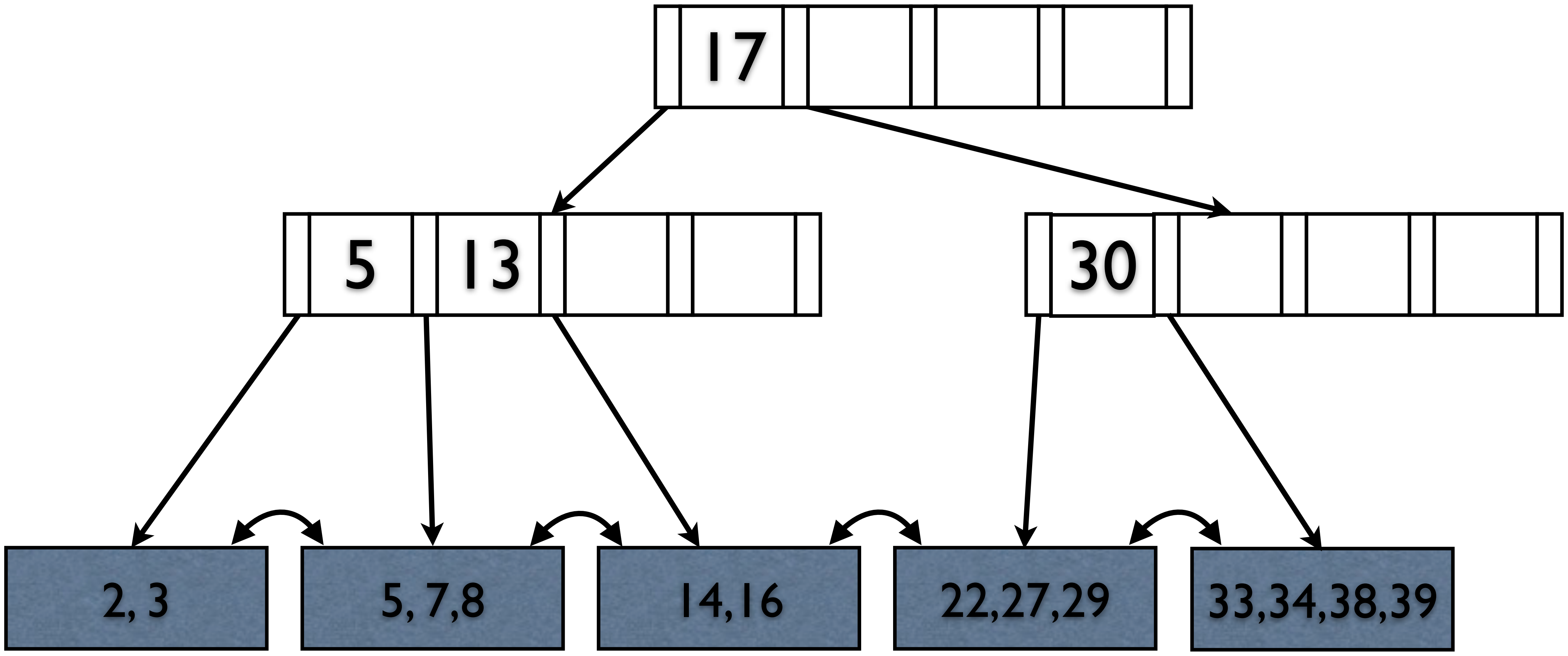
Delete 24

Deleting from B+ Trees



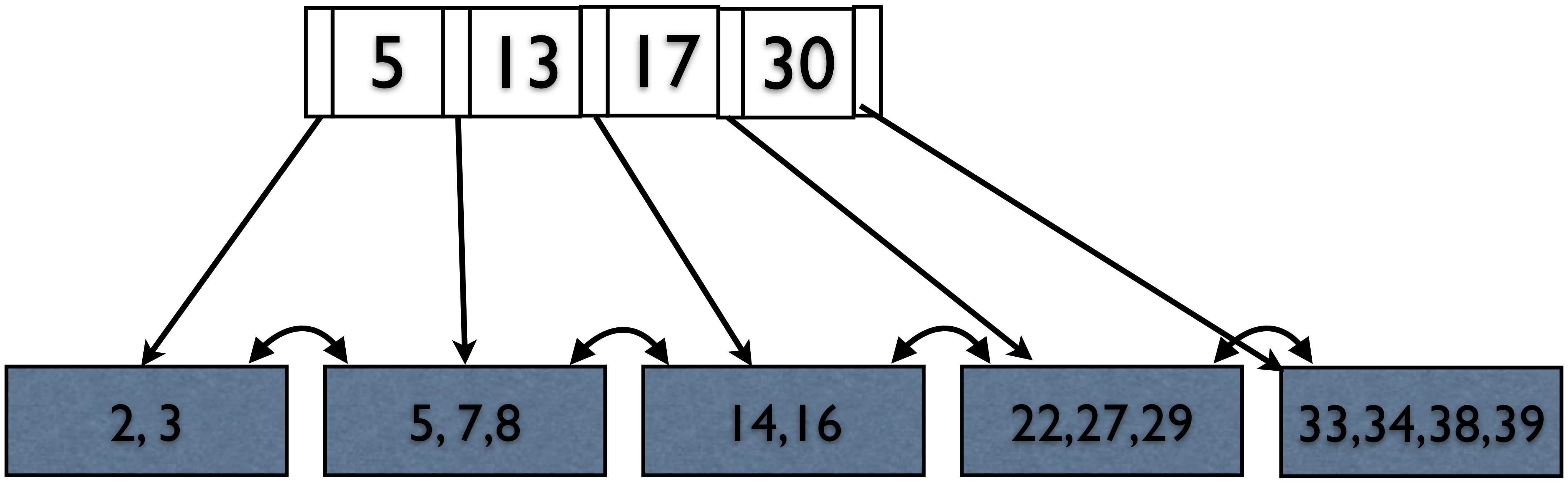
Delete 24

Deleting from B+ Trees

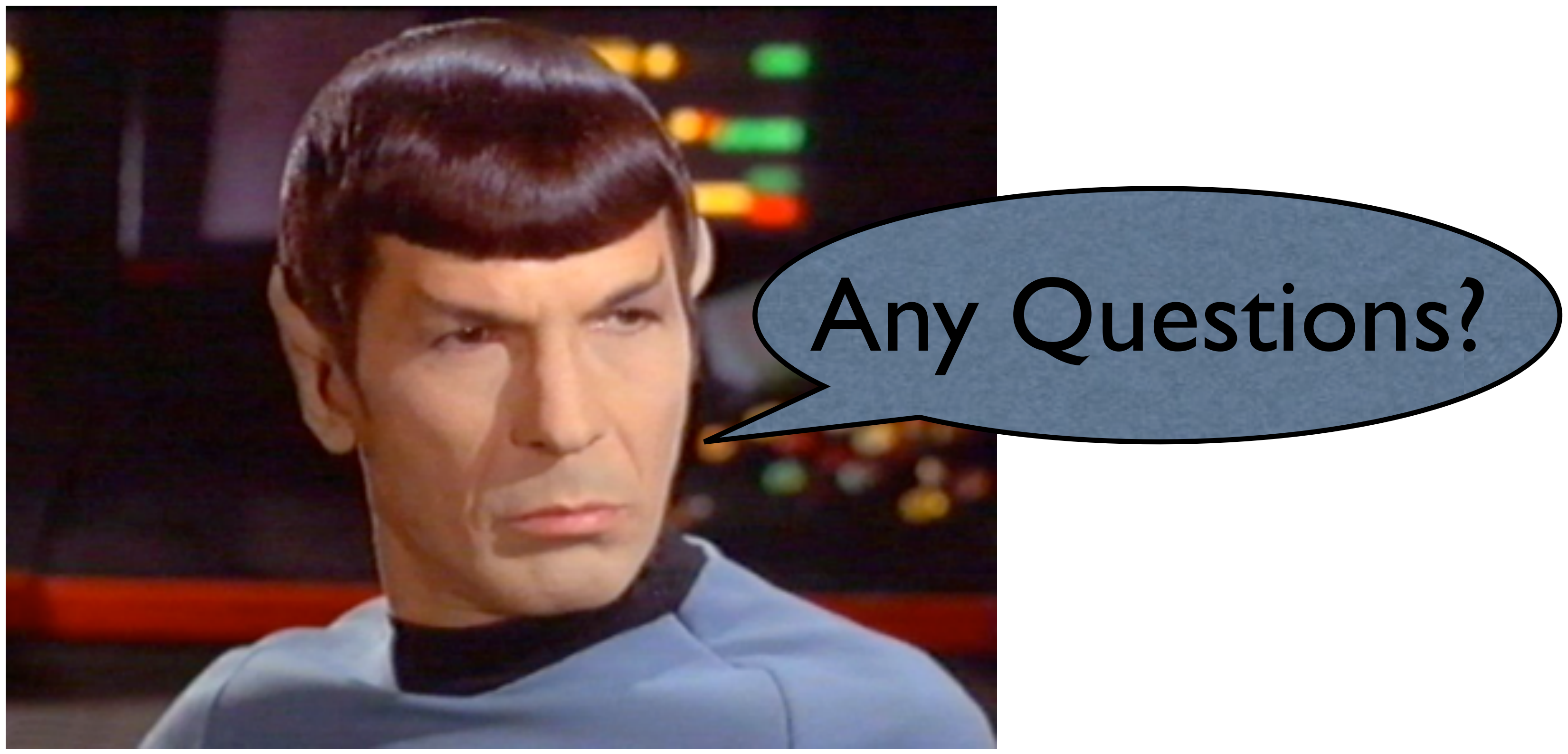


Delete 24

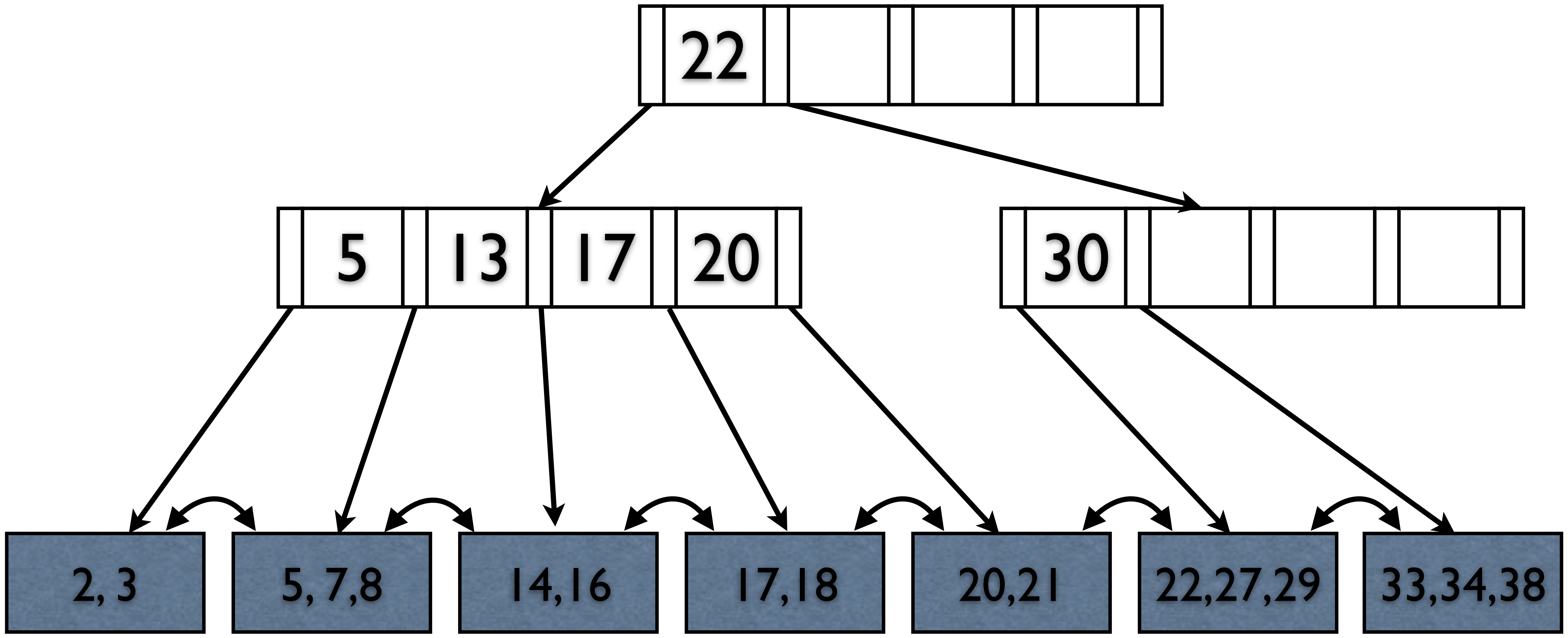
Deleting from B+ Trees



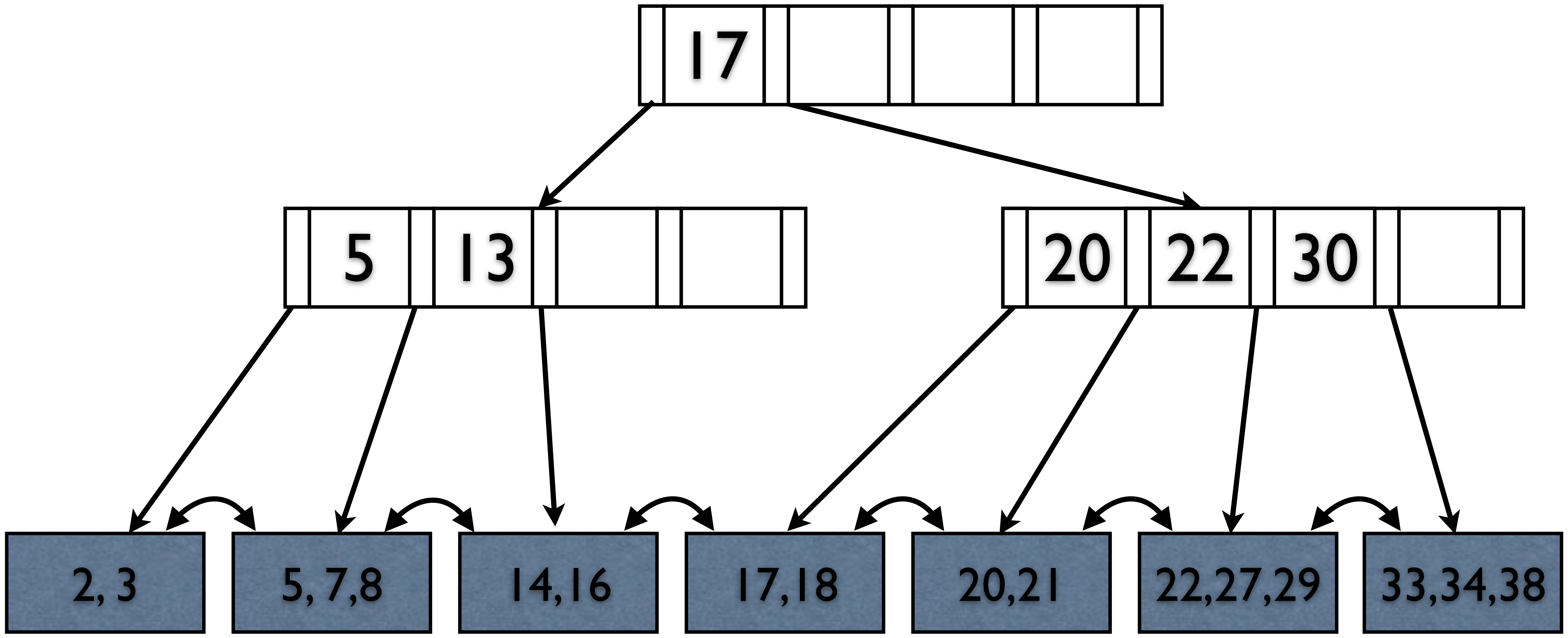
Delete 24


Any Questions?

Non-Leaf Redistribution

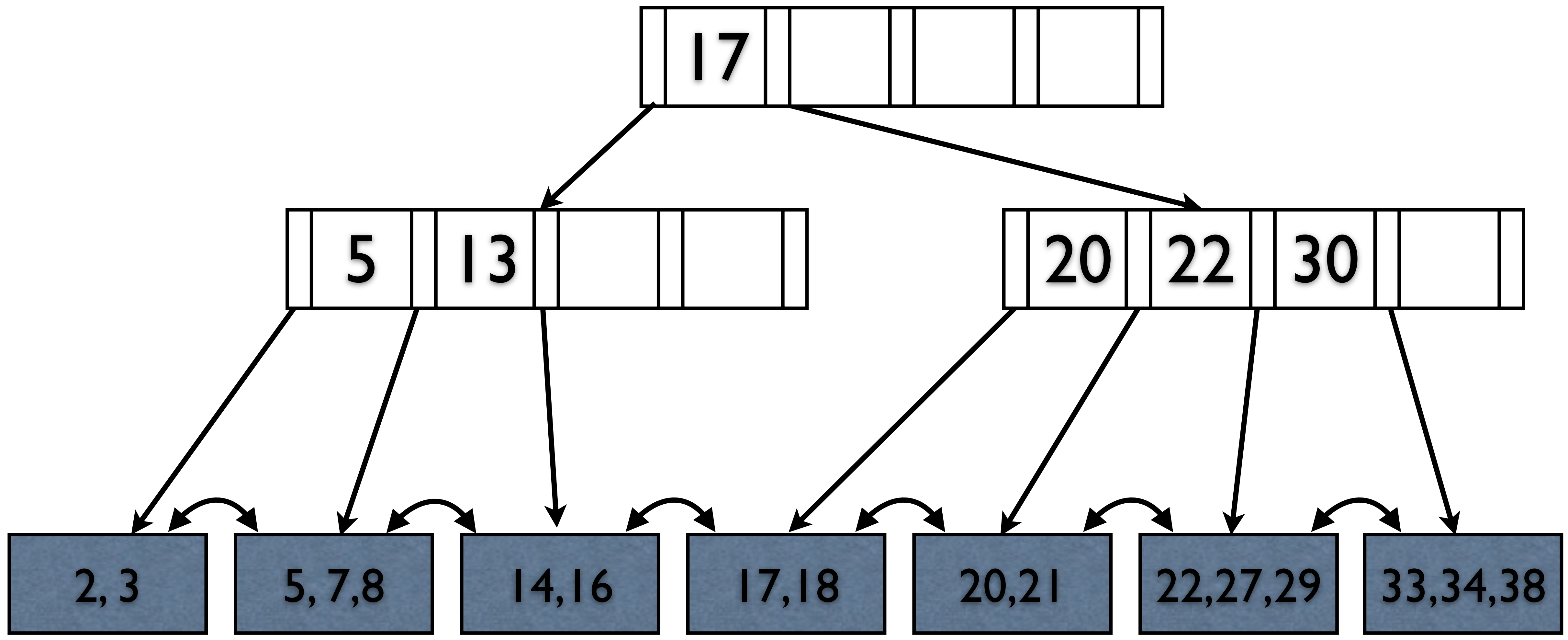


Non-Leaf Redistribution



Note that it would be sufficient to redistribute only entry 20, but we may as well even everything out.

Non-Leaf Redistribution



Intuitively, we rotate index entries 17-22 through the root

Note that it would be sufficient to redistribute only entry 20, but we may as well even everything out.

Summary

- Organize data with an index to make it easier to access data on disk (or out of cache)
- Tree structures support range searches!
  - Equality searches still O(log(n))
- Balanced trees keep access costs uniformly low!
- Two types of tree indexes
  - ISAM - Space efficient, Great for static data.
  - B+ Tree - Sacrifice space for easy updates.

More on B+ Trees next class