# Data Organization and Processing

Hierarchical Indexing

**(NDBI007)**

David Hoksza, **Škoda Petr**
http://siret.ms.mff.cuni.cz/hoksza

# Outline

- Background
  - graphs
  - **search trees**
    - binary trees
    - m-ary trees
- Typical tree type structures
  - **B-tree**
  - **B+-tree**
  - B*-tree
    - prefix tree
    - …
- **Newer types** of tree structures

# Motivation

- Similarly as in hashing, the **motivation** is **to find record(s)** given a query key using only **a few operations**
  - unlike hashing, trees allow to retrieve set of records with keys from given **range**

- Tree structures use "**clustering**" to efficiently **filter out non relevant records** from the data set

- Tree-based indexes practically implement the index file data organization
  - for each **search key,** an **indexing structure** can be maintained

# Drawbacks of Index(-Sequential) Organization

- **Static** nature
  - when **inserting** a record at the beginning of the file, the whole index needs to be **rebuilt**
    - an overflow handling policy can be applied → the performance degrades as the file grows
  - **reorganization** can take **a lot of time**, especially for large tables
  - **not possible** for online transactional processing (**OLTP**) scenarios (as opposed to OLAP – online analytical processing) where many insertions/deletions occur

# Tree Indexes

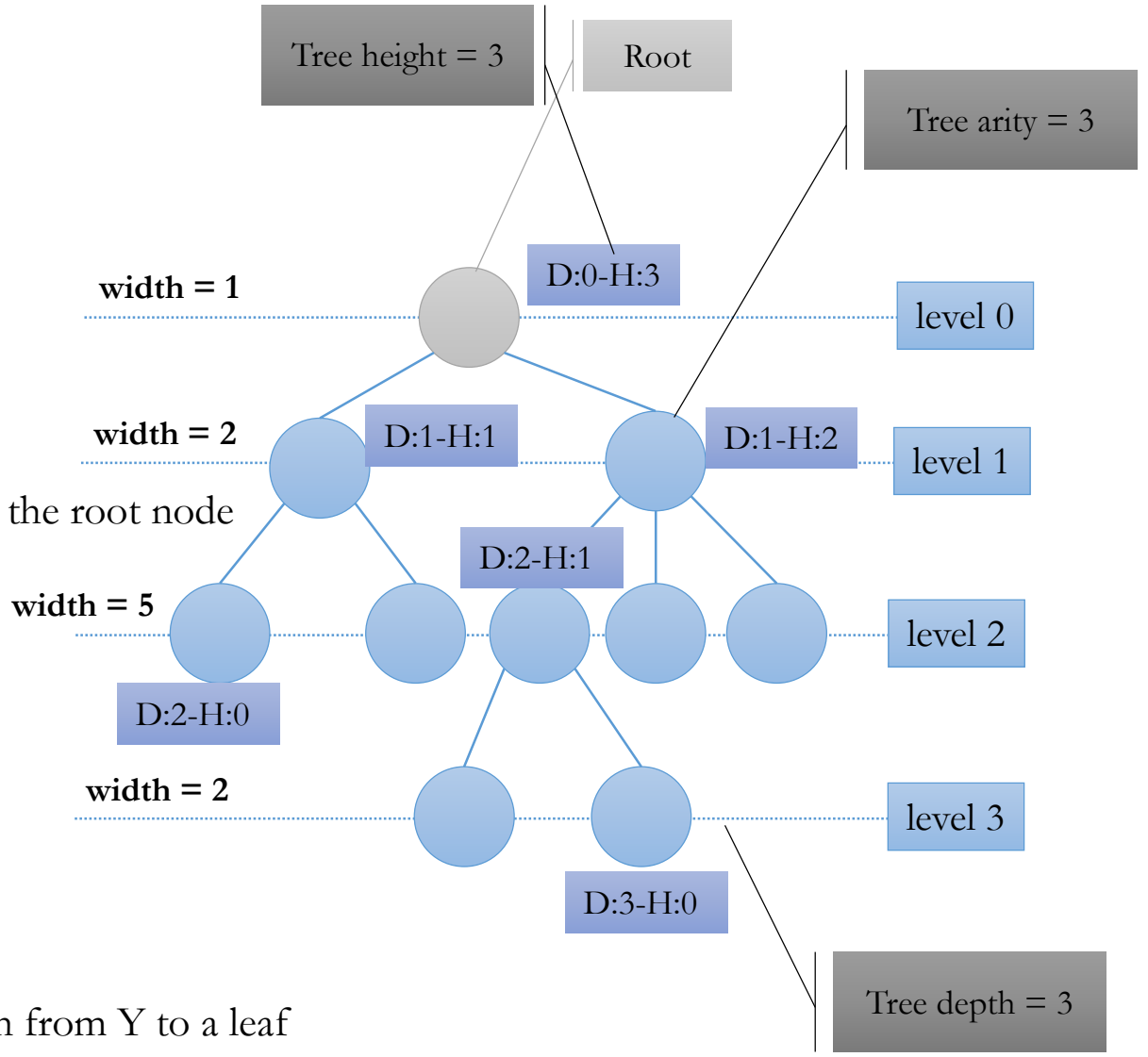
- **Most common** dynamic indexing structure for external memory
- When **inserting/deleting into/from the primary file**, the **indexing structure(s)** residing in the secondary file is **modified** to accommodate the new key
- The **modification** of a tree is implemented by **splitting/merging** nodes
- Used not only in databases
  - NTFS directory structure is built over B+ tree

# Trees

- **Undirected graph without cycles**
  - we will be interested in rooted trees
    → one node designated as the root
    → orientation
- Nodes are in **parent-child relation**
  - every **parent** has a **finite set of descendants (children nodes)**
    - from a parent an edge points to child
  - every **child** has **exactly one parent**
- **root**
  - is the only node without parent – root of the hierarchy
- **leaves**
  - nodes without children
- **inner nodes**
  - nodes with children
- **branch**
  - path from a leaf to the root

# Tree Characteristics (1)

- **Tree arity/degree**
  - maximum number of children of any node
- **Node depth**
  - the length of the simplest path (number of edges) from the root node
  - depth(root) = 0
- **Tree depth**
  - maximum of node depths
- **Tree level**
  - set of nodes with the same depth (distance from root)
- **Level width**
  - number of nodes at given level
- **Node height**
  - number of edges on the **longest** downward simple path from Y to a leaf
- **Tree height**
  - height of the root node



# Tree Characteristics (2)

- **Balanced** *(vyvážený)* **tree**
  - a tree whose subtrees differ in height by no more than one and the subtrees are height-balanced as well
- **Unsorted tree**
  - general tree – the descendants of a node are not sorted at all
- **Sorted tree**
  - unlike general tree the order of children of each node matters
  - children of a node are sorted based on a given key

# Binary Tree

- **Each inner node** contains at most **two child nodes** (tree arity = 2)

- Types
  - **perfect** *(perfektní)* **binary tree** – every non-leaf node has two child nodes
  - **complete** *(komletní)* **binary tree** – full tree except for the first non-leaf level where nodes are as far left as possible

- Characteristics
  - **number of nodes** of a **perfect** binary tree of height $h$

$$\#nodes_{perf} = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

  - **number of leaf nodes** of a **perfect** binary tree of height $h$

$$\#leaf\_nodes_{perf} = 2^h$$

# Binary Search Tree (BST)

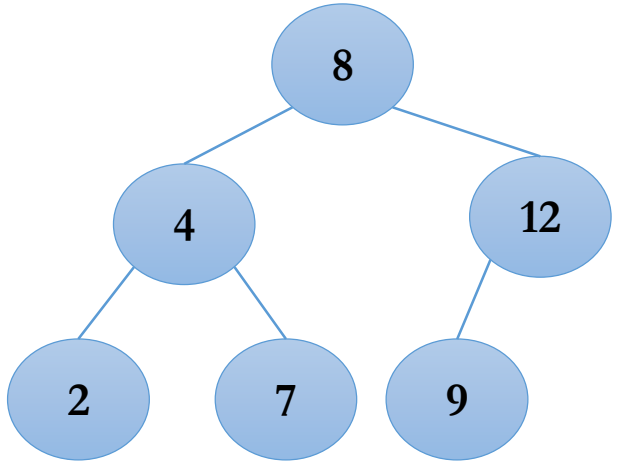
- Binary sorted tree where
  - each node is assigned a value corresponding to one of the keys
  - all nodes in the **left subtree** of a node $X$ correspond to keys with **lower** values than the value of $X$
  - all nodes in the **right subtree** of a node $X$ correspond to keys with **higher** values than the value of $X$
  - when using BST for storing records, the records can be stored directly in the nodes or addressed from them

- Searching a record with a key $k$
  1. Enter the tree at the root level
  2. Compare $k$ with the value $v$ of the current node
  3. If $k = v$ **return the** record corresponding to the current node key
  4. If $k < v$ descend to the left subtree
  5. If $k > v$ descend to the right subtree
  6. Repeat steps 2-5 until a leaf is reached. If the condition in step 3 is not met at any level, no record with the key $k$ is present

# Redundant vs. Non-Redundant BST

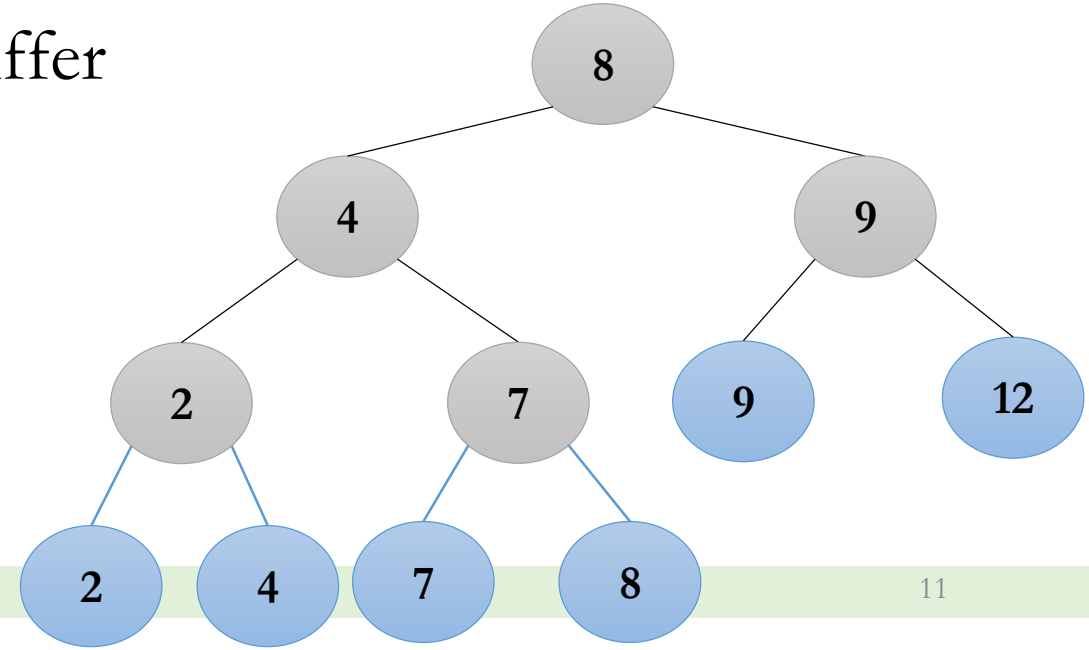
**Non-Redundant BST**

- Records stored in (addressed from) both inner and leaf nodes




**Redundant BST**

- Records stored in (addressed from) the leaves
- Structure of inner and leaf nodes differ



# Applications of Binary Trees (not related to storing data records)

- **Expression trees**
  - leaves = variables
  - inner nodes = operands

- **Huffman coding**
  - leaves = data
  - coding along a branch leading to give leaf = leaf's binary representation

- **Query optimizers in DBMS**
  - query can be represented by an algebraic expression which can be in turn represented by a binary tree

# M-ary Trees (1)

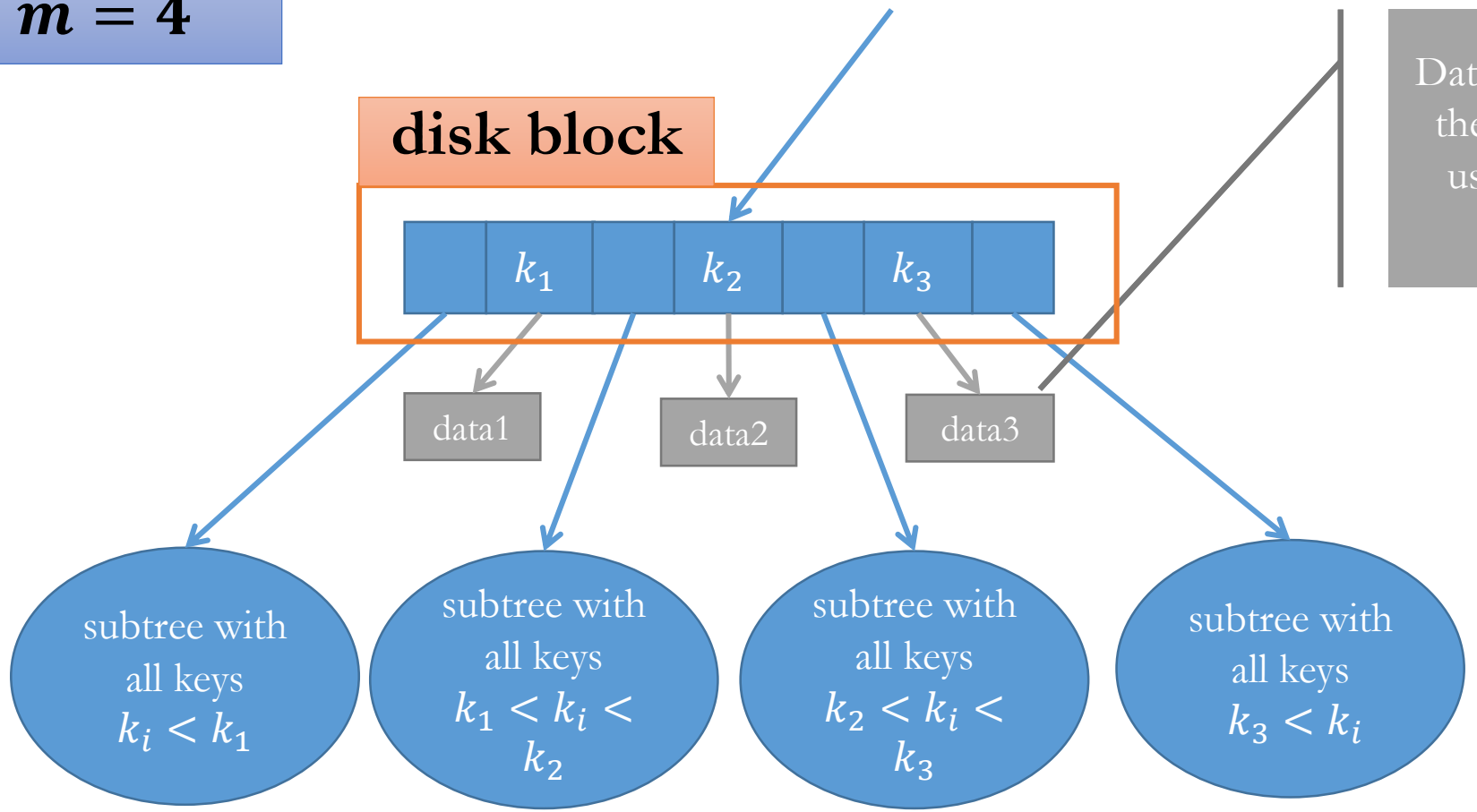
- Motivation
  - binary trees are not suitable for magnetic disks because of their height – indexing 16 items would need an index of height 4 → up to 4 disk accesses (potentially in different locations) to retrieve one of the 16 records
  - $\log_2 1000 = 10$ , $\log_2 1{,}000{,}000 = 20$
  - **But increasing arity leads to decreasing the tree height**

# M-ary Trees (2)

- **Generalization of binary search trees**
  - A binary tree is a special case of an m-ary tree for $m = 2$
    - only m-ary trees with $m > 2$ are usually considered as true m−ary trees
  - unlike binary tree, **m-ary tree** contains $\boldsymbol{m-1}$ **discriminators (keys)** in every node
  - every node $N$ points to up to $\boldsymbol{m}$ **child nodes** (subtrees)
  - every **subtree** of $N$ contains records with **keys restricted** by the **pair of discriminators** of $N$ between which the subtree is rooted
  - the **leftmost subtree** contains only records with keys having **lower values** than **all the discriminators** in $N$
  - the **rightmost subtree** contains only records with **keys having higher values** than **all the discriminators** in $N$

# M-ary Trees (3)

$m = 4$




Data could be stored directly in the node as well. But it is not usual in real-world database environments.

# Characteristics of M-Ary Trees (1)

- **Maximum number of nodes**
  - tree height $h$
  - $level_0 = m^0 = 1, level_1 = m^1, level_2 = m^2, \ldots level_h = m^h$

$$\sum_{i=0}^{h} m^i = \frac{\boldsymbol{m^{h+1} - 1}}{\boldsymbol{m - 1}}$$

- **Maximum number of records/keys**
  - every node contains up to $m - 1$ keys/records

$$\frac{m^{h+1} - 1}{m - 1} \times (m - 1) = \boldsymbol{m^{h+1} - 1}$$

  - $m = 3, h = 3 \rightarrow \#records \leq 3^4 - 1 = 80$
  - $m = 100, h = 3 \rightarrow \#records \leq 100^4 - 1 = 100{,}000{,}000$

# Characteristics of M-Ary Trees (2)

- **Minimum height**

$$h = \lceil \log_m n \rceil$$

  - minimum height of the tree corresponds to the **minimum number of disk operations** needed to **fetch** a record → search complexity $O(\log_m n)$

- **Maximum height**

$$h \doteq \frac{n}{m}$$

  - maximum height of the tree corresponds to the **maximum number of disk operations** needed to **fetch** a record → search complexity $O(n)$

- The challenge is to **keep the complexity logarithmic**, that is to keep the tree more or less balanced

# B-tree

- Bayer & McCreight, 1972

- B-tree is a sorted **balanced m-ary** (not binary) **tree** with additional **constraints restricting the branching** in each node thus causing the tree to be reasonably "wide"

- Inserting or deleting a record in B-tree causes only local changes and not rebuilding of the whole index

# B-tree definition (1)

- B-trees **are balanced m-ary trees** fulfilling the following conditions:

1. The **root** has **at least two children** unless it is a leaf.
2. Every **inner node** except of the root **has at least** $\left\lceil \frac{m}{2} \right\rceil$ and **at most** $m$ **children**.
3. Every **node** contains **at least** $\left\lceil \frac{m}{2} \right\rceil - 1$ and **at most** $m - 1$ **(pointers to) data records**.
4. Each **branch** has the **same length**.

# B-tree definition (2)

5. The **nodes** have following **organization**:

$$\boldsymbol{p_0}, (\boldsymbol{k_1}, \boldsymbol{p_1}, \boldsymbol{d_1}), (\boldsymbol{k_2}, \boldsymbol{p_2}, \boldsymbol{d_2}), \dots, , (\boldsymbol{k_n}, \boldsymbol{p_n}, \boldsymbol{d_n}), \boldsymbol{u}$$

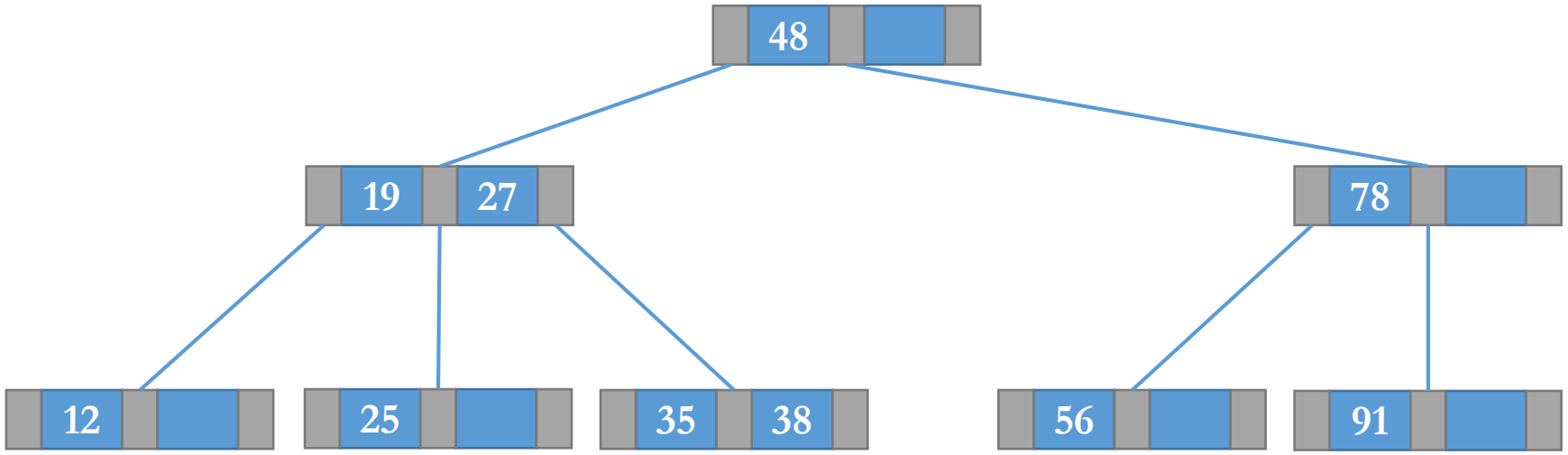
- where $\boldsymbol{p_0}, \boldsymbol{p_1}, \dots, \boldsymbol{p_n}$ are **pointers to the child nodes**, $\boldsymbol{k_1}, \boldsymbol{k_2}, \dots, \boldsymbol{k_n}$ are **keys**, $\boldsymbol{d_1}, \boldsymbol{d_2}, \dots, \boldsymbol{d_n}$ are associated **data, $\boldsymbol{u}$** is **unused space** and the **records $(\boldsymbol{k_i}, \boldsymbol{p_i}, \boldsymbol{d_i})$ are sorted** in increasing order with respect to the keys and

$$\left\lceil \frac{\boldsymbol{m}}{\boldsymbol{2}} \right\rceil - \boldsymbol{1} \leq \boldsymbol{n} \leq \boldsymbol{m} - \boldsymbol{1}$$

6. If a **subtree $\boldsymbol{U(p_i)}$** corresponds to the pointer $p_i$ then:
   - **i.** $\forall \boldsymbol{k} \in \boldsymbol{U}(\boldsymbol{p_{i-1}}): \boldsymbol{k} < \boldsymbol{k_i}$
   - **ii.** $\forall \boldsymbol{k} \in \boldsymbol{U}(\boldsymbol{p_i}): \boldsymbol{k} > \boldsymbol{k_i}$

# Example of a B-Tree

- Keys: 25, 48, 27, 91, 35, 78, 12, 56, 38, 19



# Redundant B-Tree

- **Redundant** vs. **non-redundant** B-trees
  - the presented definition introduced the **non-redundant** B-tree where **each key value** occurred **just once** in the whole tree
  - **redundant B-trees** store **the data values in the leaves** and thus have to allow **repeating of keys in the inner nodes**
    - restriction 6(i) is modified so that
      - $\forall \boldsymbol{k} \in \boldsymbol{U}(\boldsymbol{p_{i-1}}): \boldsymbol{k} \leq \boldsymbol{k_i}$
    - moreover, the **inner nodes do not contain pointers** to the data records

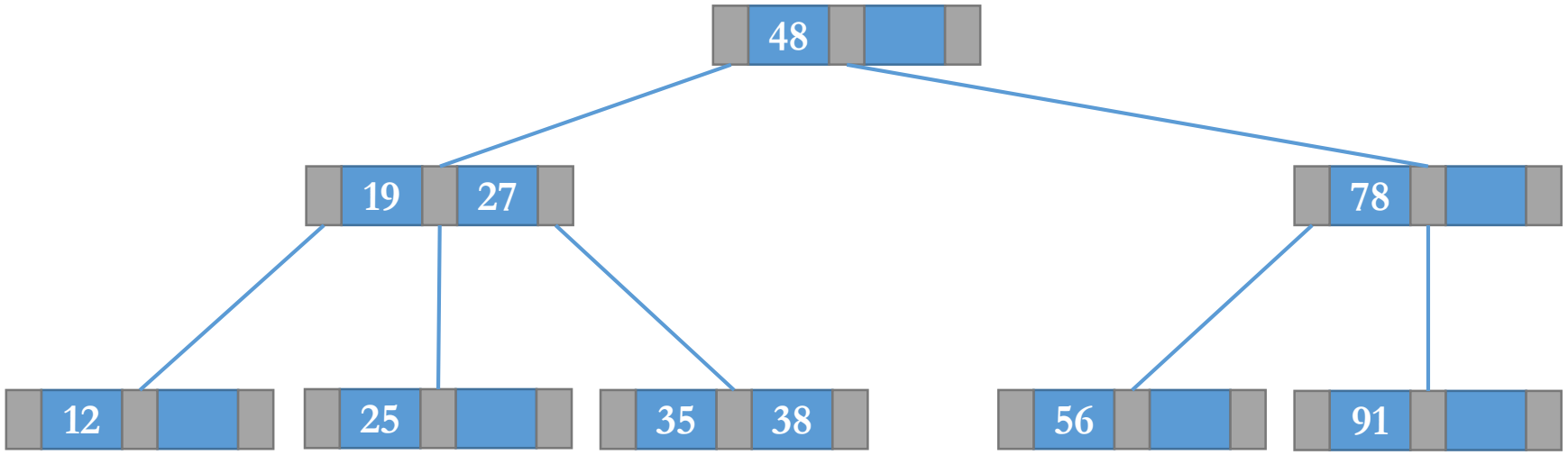
# B-Tree implementations

- Nodes vs. pages/blocks
  - usually **one page/block contains one node**

- **Existing DBMS implementations**
  - one **page** usually takes **8KB**
  - **redundant B-trees**
    - inner nodes contain only pairs $(k_1, p_1)$ → inner nodes and leaf nodes differ in structure and size
    - **data** are not stored in the indexing structure itself but **addressed from the leaf nodes**
    - if the key is of type 64-bit integer (8B) and the DMBS runs on a 64-bit OS (8B pointers) with 8KB blocks → $|(k_1, p_1)| = 16B$ → one page can accommodate $8192/16 = 512$ pairs → arity of the B-tree is then about 500

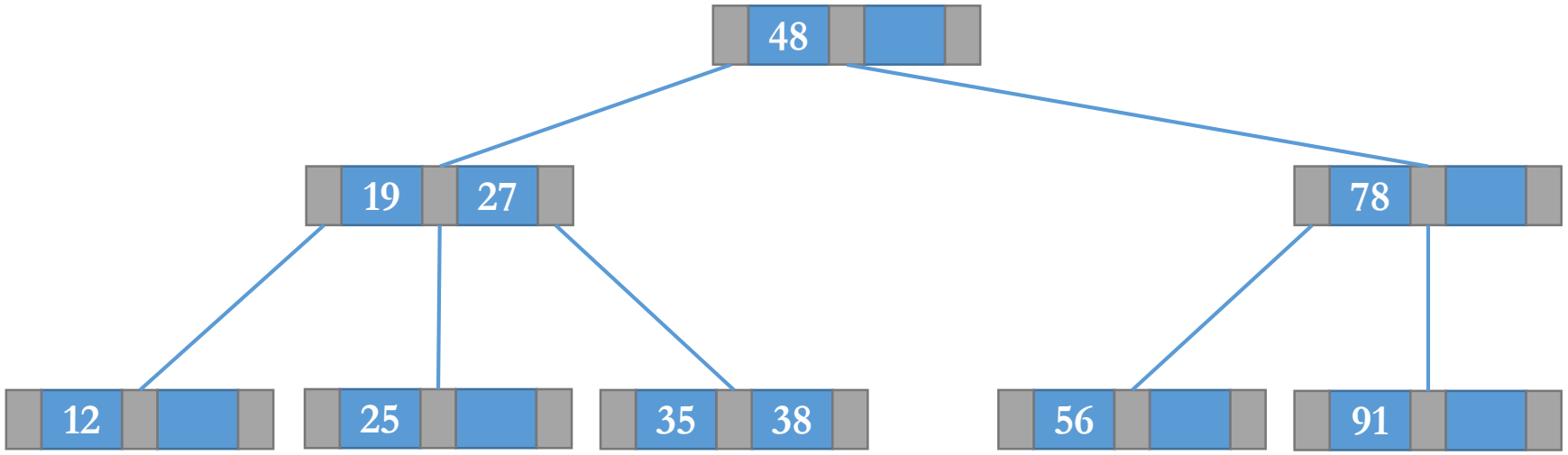
# B-Tree - Search

- Search: 25, 27, 91, 19



# B-Tree - Search

- Search: 25, 27, 91, 19, 49



# B-Tree - Search

- Searching a (non-redundant) tree $T$ for a record with key $k$
  - **Enter** the tree in the **root node**.
  - If the node **contains a key** $\boldsymbol{k_i}$ such that $\boldsymbol{k_i = k}$ return the data associated with $\boldsymbol{d_i}$.
  - Else if the node is **leaf, return** NULL.
  - Else find **lowest** $\boldsymbol{i}$ such that $\boldsymbol{k < k_i}$ and set $\boldsymbol{j = i - 1}$. If there is no such $i$ set $\boldsymbol{j}$ **as the rightmost** index with existing key.
  - Fetch the **node** pointed to by $\boldsymbol{p_j}$.
  - **Repeat** the process from step 2.

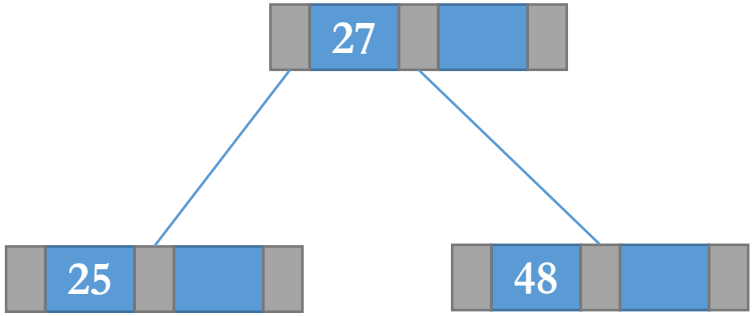
# Updating a B-Tree

- The logarithmic complexity is ensured by the condition that **every node has to be at least half full**
- **Inserting**
  - finding a leaf where the new record should be inserted
  - when inserting **into a not yet full node no splitting occurs**
  - when inserting **into a full node**, the **node is split** in such a way that **the two resulting nodes are at least half full**
    - splitting a node increments the number of subtrees of the parent node and thus possibly causes splitting of the parent node → **split cascade**
- **Deleting**
  - when deleting a record from a **node more than half full**, **no reorganization** happens
  - deleting **in a half full node** induces **merging of the neighboring nodes**
    - merging decreases the number of child nodes in the parent node thus possibly causing merging of the parent node→ **merge cascade**

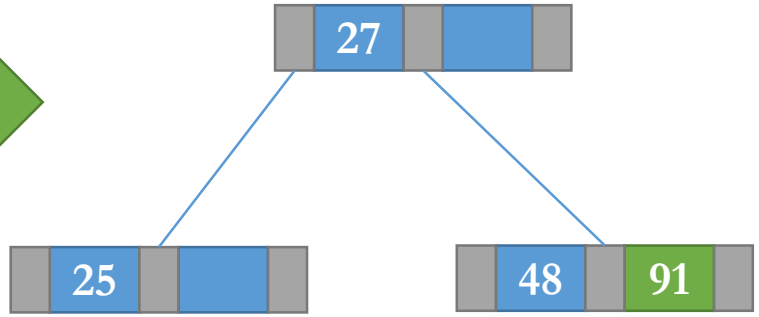
# Inserting into a B-tree (1)

- **Insert** into a (non-redundant) tree $T$ a record $r$ with key $k$
  - If the tree is **empty**, allocate a **new node**, insert the key $k$ and (pointer to record) $r$ and **return**.
  - Else find the **leaf node $L$** where the key **$k$ belongs**.
  - If **$L$ is not full insert $r$** and $k$ into $L$ in such a position that the keys are sorted and **return**.
  - Else create a **new node $L'$**.
  - Leave lower **half records** (all the items from $L$ plus $r$) in **$L$** and the higher **half records** into **$L'$ except** of the item with the middle key $k'$.
    - **a) If L is the root**, create **a new root node**, move the record with key $k'$ to the new root and **point it to $L$ and $L'$** and return.
    - b) Else move the record with key $k'$ to the **parent node $P$** into appropriate position based on the value $k'$ **and point the "left" pointer to L and the "right" pointer to $L'$**.
  - **If $P$ overflows**, repeat step 5 for $P$ else return.

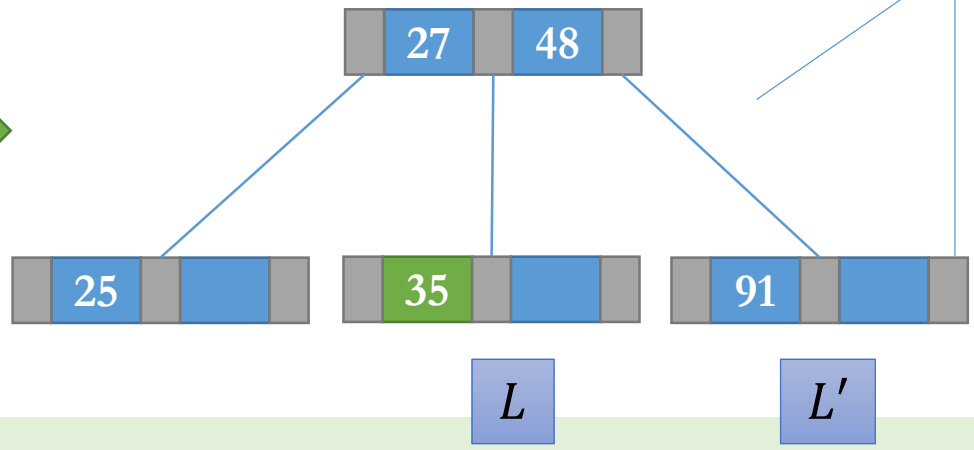
# Inserting into a B-tree (2)





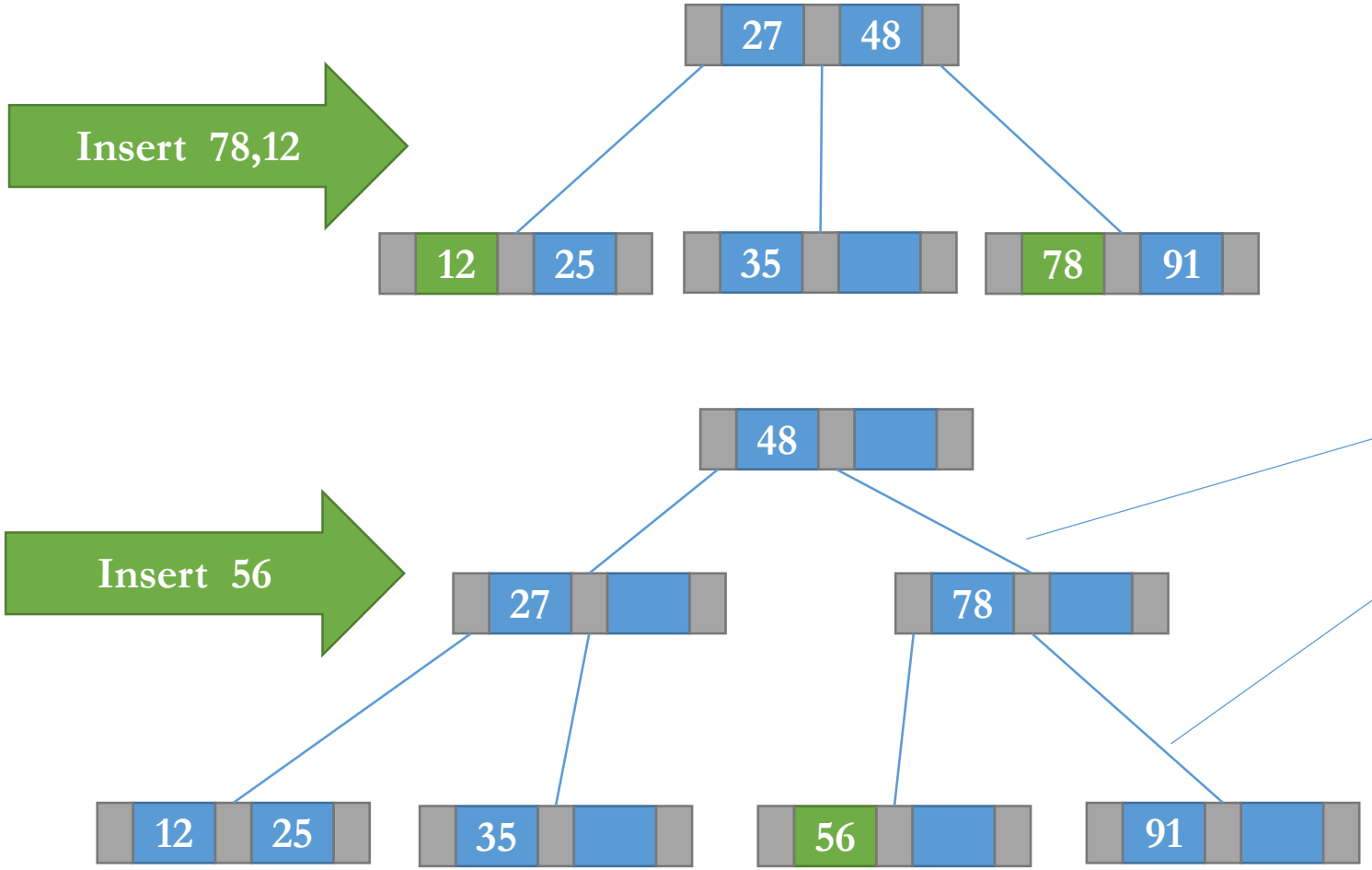







35 falls into the node (48,91) → (35, 48, 91) → middle key (48) moves to the parent node

# Inserting into a B-tree (3)




78 moves into the node (27, 48) → (27, 48, 78) → middle key (48) moves to the parent node → new root

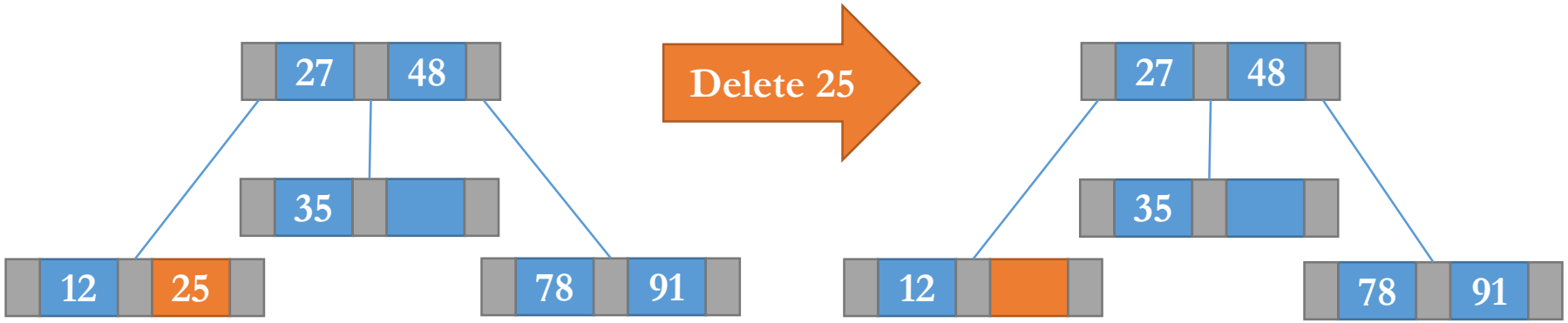
56 falls into the node (78,91) → (56, 78, 91) → middle key (78) moves to the parent node

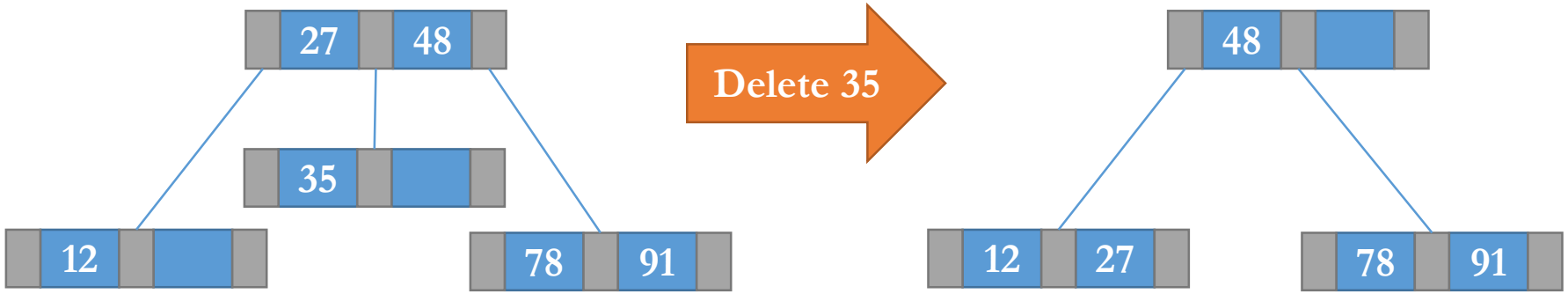
# Deleting from a B-tree (1)

- **Delete** from tree $T$ record $r$ with key $k$

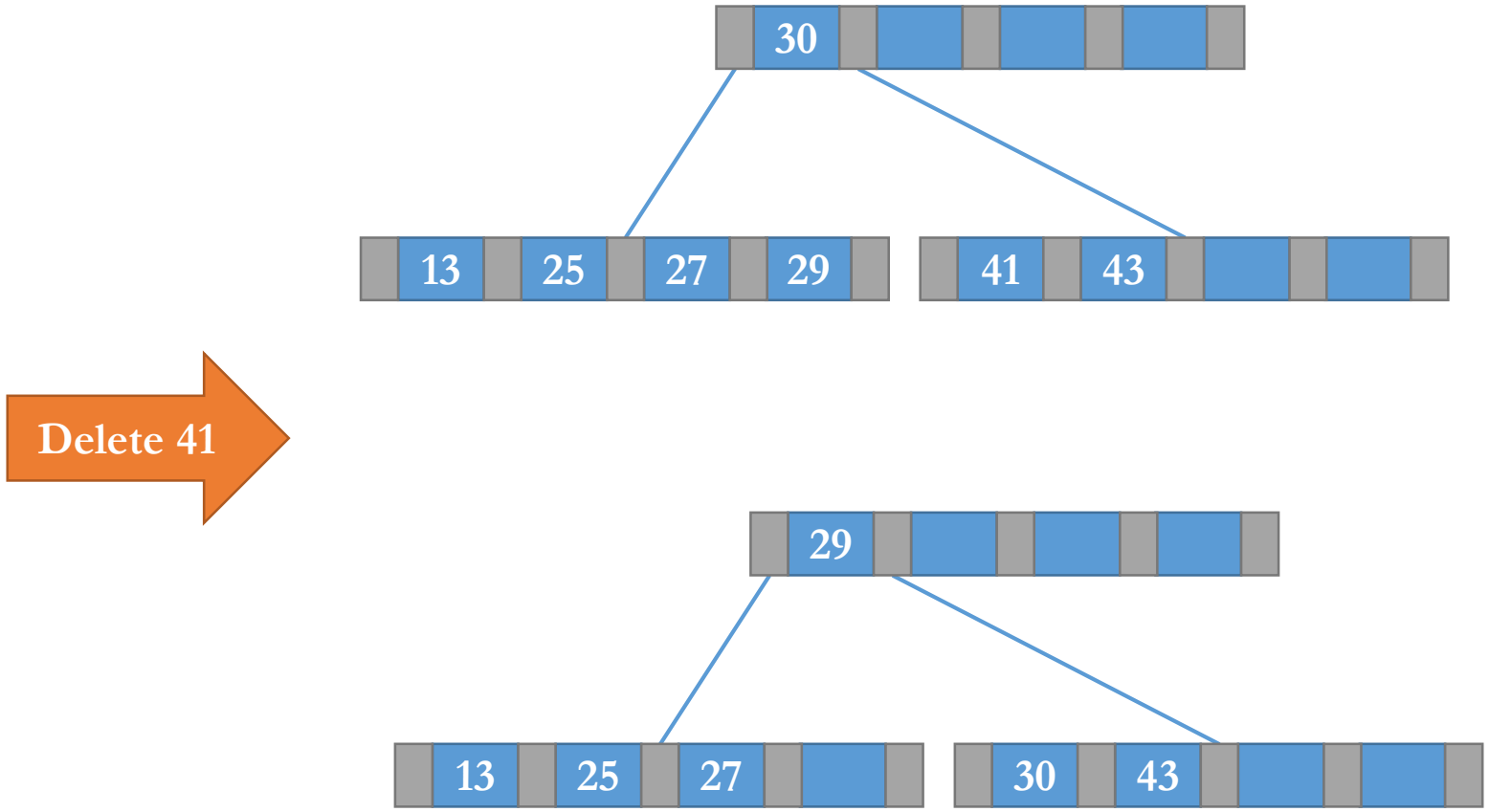
  - **Find a node $N$** containing the key $k$.
  - **Remove $r$** from $N$.
  - **If** number of keys in $N \geq \left\lceil \frac{m}{2} \right\rceil - 1$, **return**.
  - **Else,** if possible, **merge $N$ with either right or left sibling** (includes update of the parent node accompanied by the decrease of the number of keys in the parent node).
  - **Else reorganize records among $N$ and its sibling and the parent node**.
  - If needed, reorganize the **parent node** in the same way (steps 3 – 5).

# Deleting from a B-tree (2)



# Deleting from a B-tree (3)



# B-Tree - Complexity

- Let us have
  - 8 KB = $8 \times 2^{10}$ B — page size ($\boldsymbol{R}$)
  - 10 B — key ($\boldsymbol{K}$)
  - 8 B — pointer to a tree node ($\boldsymbol{P_T}$)
  - 9 B — pointer to data (pointer to a data page + offset) ($\boldsymbol{P_D}$)

- **Arity** of a B-tree with such values can be

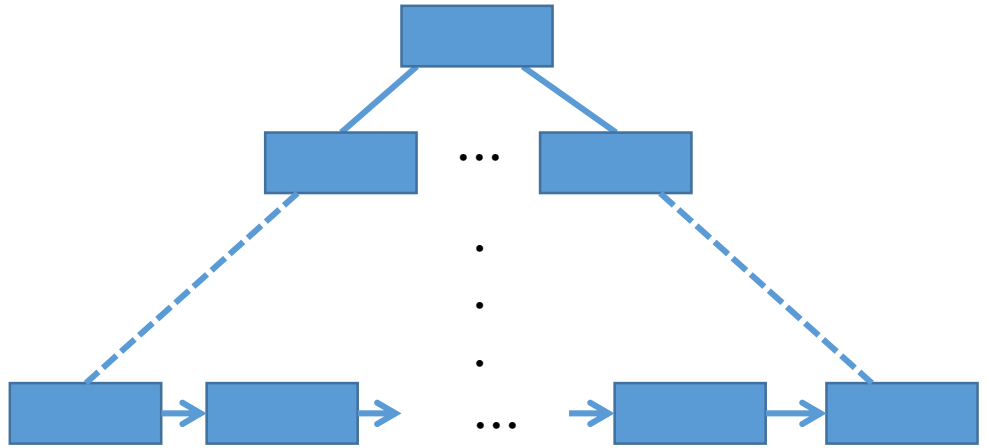
$$m \times P_T + (m-1) \times (P_D + K) \leq R$$

$$m \times 8 + (m-1) \times (9 + 10) \leq 8192$$

$$\boldsymbol{m \leq \frac{8192 + 19}{27} = 304}$$

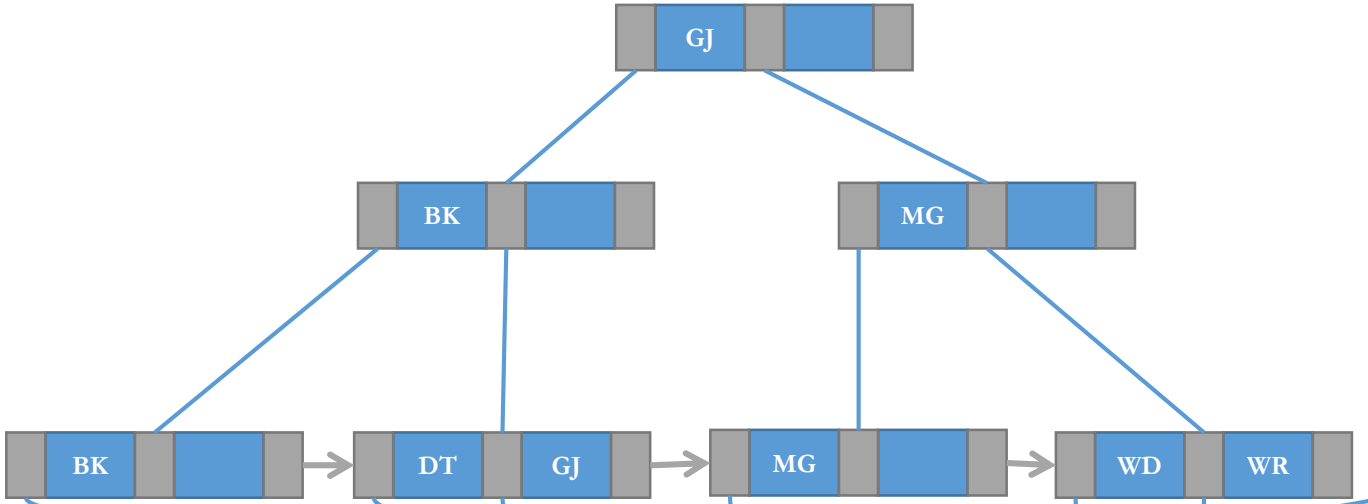
  - **every node** can accommodate **up to 303 records** (m-1)
  - if every node shows **utilization 2/3** then every node contains **202 records**
  - thus on average
    - **tree of height 0 → 202**
    - **tree of height 1 → 40,804**
    - **tree of height 2 → 8,242,408**
    - **tree of height 3 → 1,664,966,416**

# B+-tree



- **Redundant** B-tree
  - **records** are pointed to **from the leaf nodes** only
- **The leaf level is chained**
  - the leaf nodes do not have to be physically next to each other since they are connected using pointers
- **Preferred** in existing DBMS
- **In some DBMS** (e.g., Microsoft SQL Server), **also the non-leaf** levels are chained

# B+-Tree – Example (m=3)




| Name | Department |
|---|---|
| Brown Kevin | PR |
| Brown Kevin | Purchasing |
| Brown Kevin | Marketing |
| Duffy Terri | Developlment |
| Duffy Terri | Research |
| Galvin Janice | Purchasing |
| Matthew Gigi | Purchasing |
| Walters David | Production |
| Walters Rob | Marketing |
| Walters Rob | Developlment |
| Walters Rob | PR |

# B- vs. B+-Tree

**B-tree Advantages**

- Non-redundant
  - **can be redundant** as well
- A record can be identified faster (if it resides in an inner level)

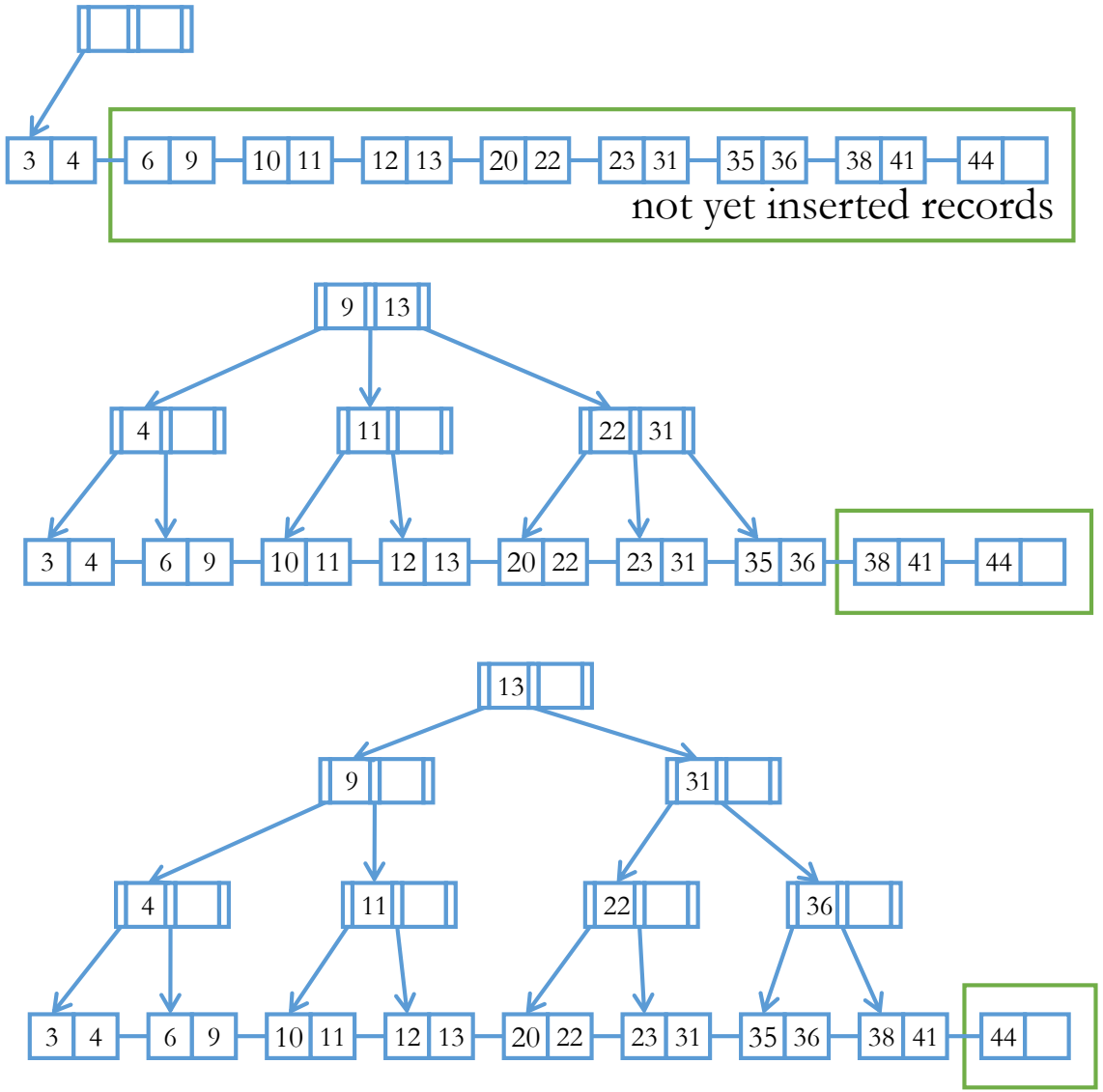
**B+-tree Advantages**

- **Smaller inner nodes** (no pointers to the data) → nodes can **accommodate more records** → **lower tree** height → **faster search**
- Insert and delete operations are simpler
- Simple implementation
  - especially of **range queries**

# Bulk Loading B-Trees

- When indexing a large collection, inserting records one by one can be tedious → **bulk loading**

1. Sort the data based on the search key in the pages
2. Insert pointer to the leftmost page into a new root
3. Move over the data file and insert the respective keys into the rightmost index page above the leaf level. If the rightmost page overflows, split.



# Page Balancing

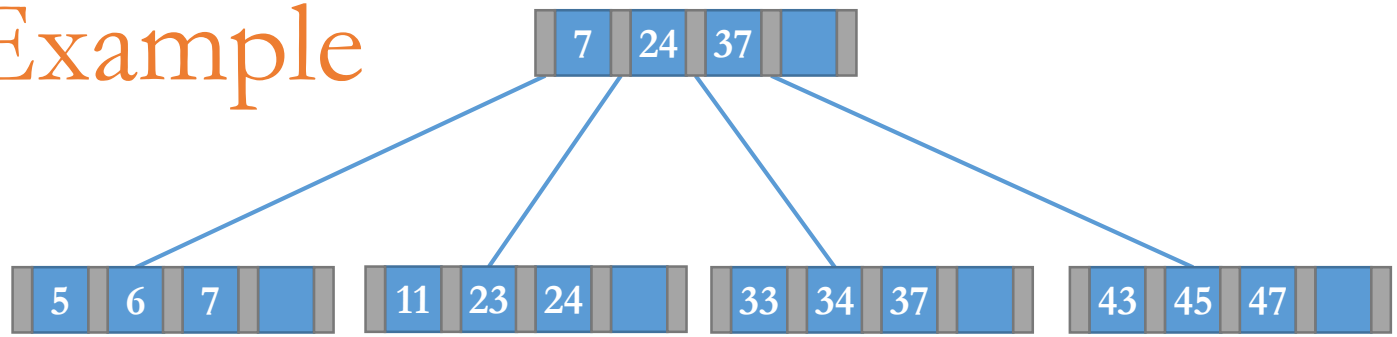
- Modification of B-tree where **an overflow does not have to lead to a page split**
- When a page overflows
  - **sibling** pages are **checked**
  - the content of the **overflowed page is joined** into a set $X$ with the left or right neighbors
  - The record to be **inserted** is **added into $X$** and the content is equally distributed into the two nodes
  - the changes are projected into the parent node where the keys have to be modified (but no new key is inserted → **no split cascade**)
- For high $m$ this change leads to about 75% utilization in the worst case

# B*-Tree

- Generalization of page balancing
  - the **root** node has **at least 2 children**
  - all the **branches** are of **equal** length
  - **every node** different from the root has **at least $\lceil (2m-1)/3 \rceil$ children**

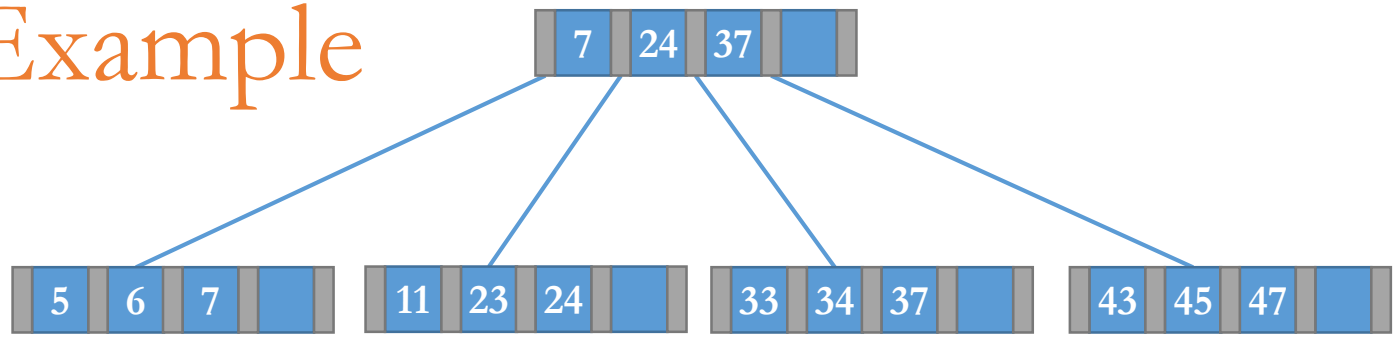
- **Tree modification**
  - **two full** pages are **split into 3** pages (one new page)
    - if a **node is full but none of its neighbors** is full, **page balancing** takes place
    - if the **insert** occurs in a **full page** which has **full left or right neighbor**
      - their **content is joined** into a set *X*
      - the **new record is added** into *X*
      - a **new page** *P* is allocated
      - the **records** from *X* are **equally distributed** into the 3 pages (the 2 existing and *P*)
      - a **new key** is **added into the parent node** and the keys are adjusted
  - the delete operation is handled similarly

# B*-Tree - Example

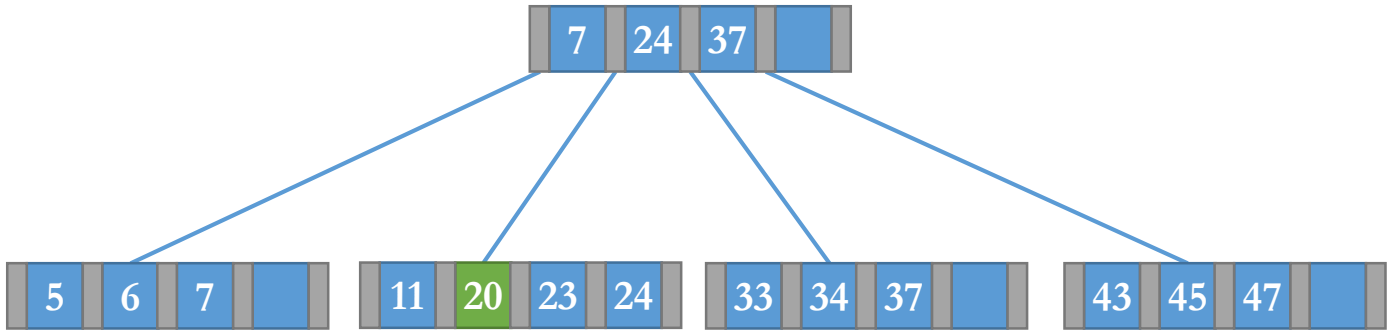



Insert 20

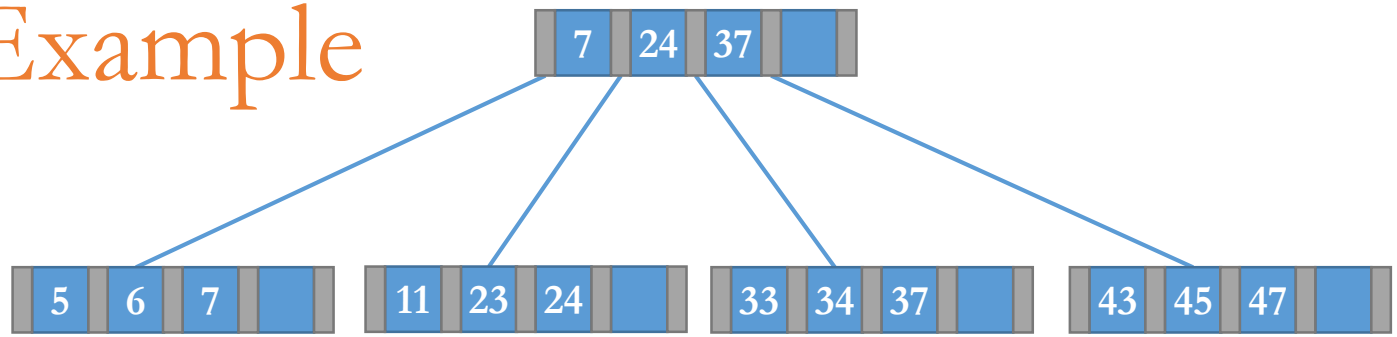
# B*-Tree - Example



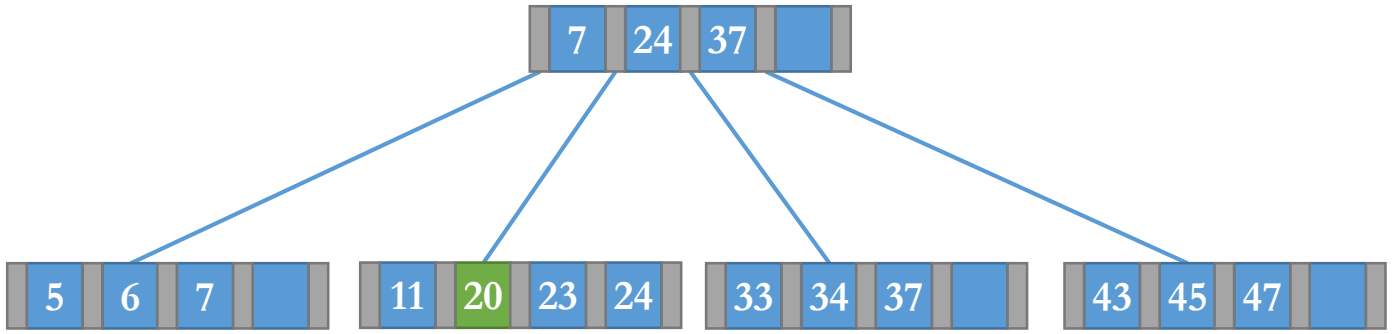

Insert 20



# B*-Tree - Example

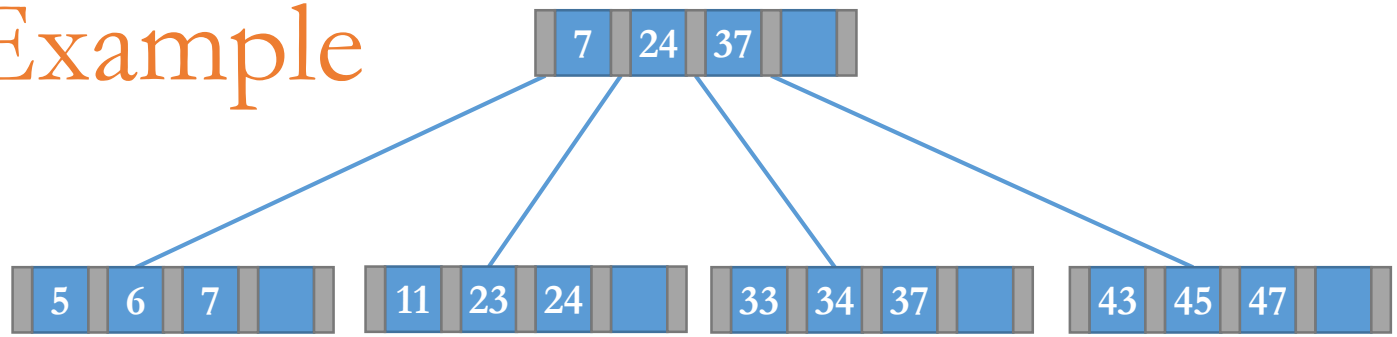



Insert 20

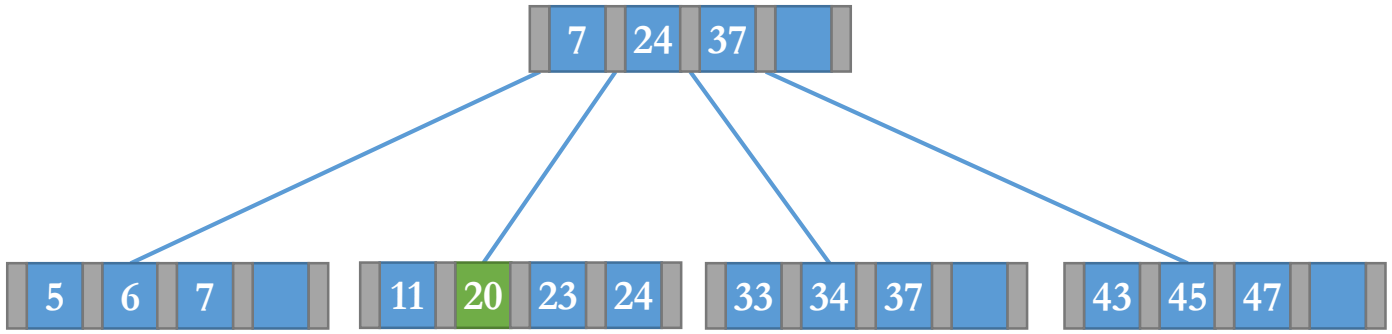



Insert 22

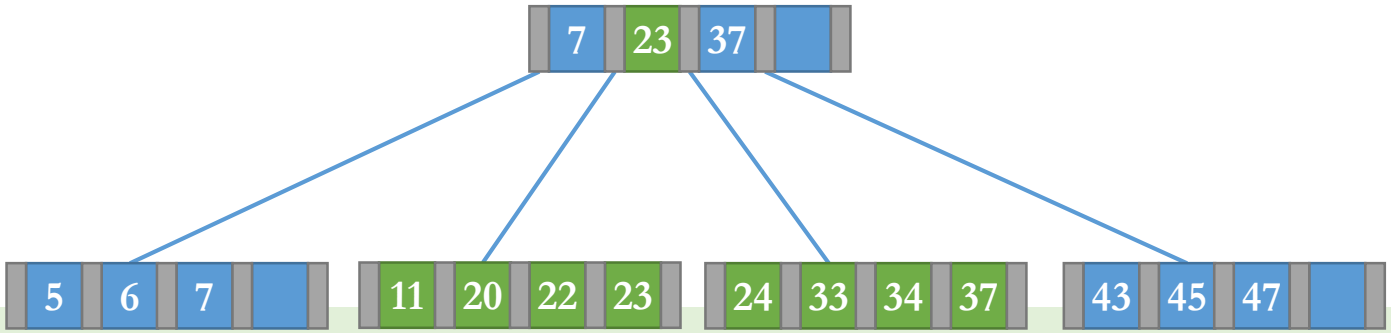
# B*-Tree - Example



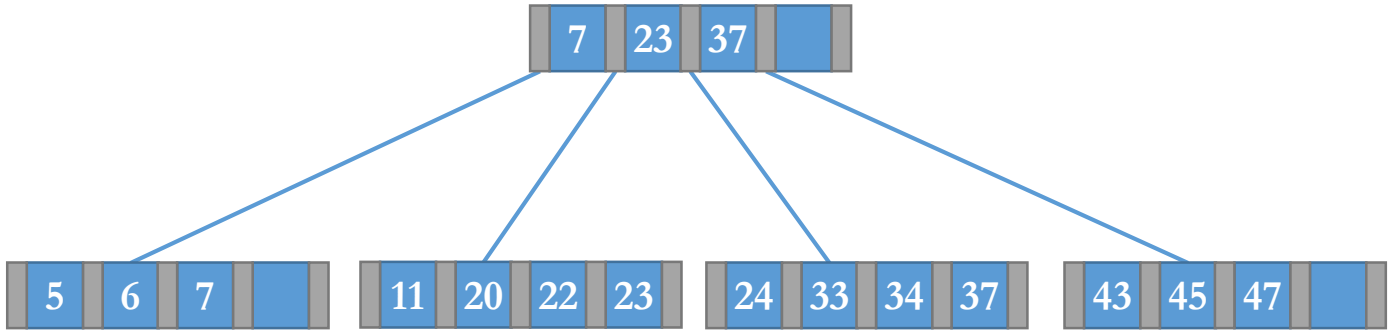

Insert 20




Insert 22

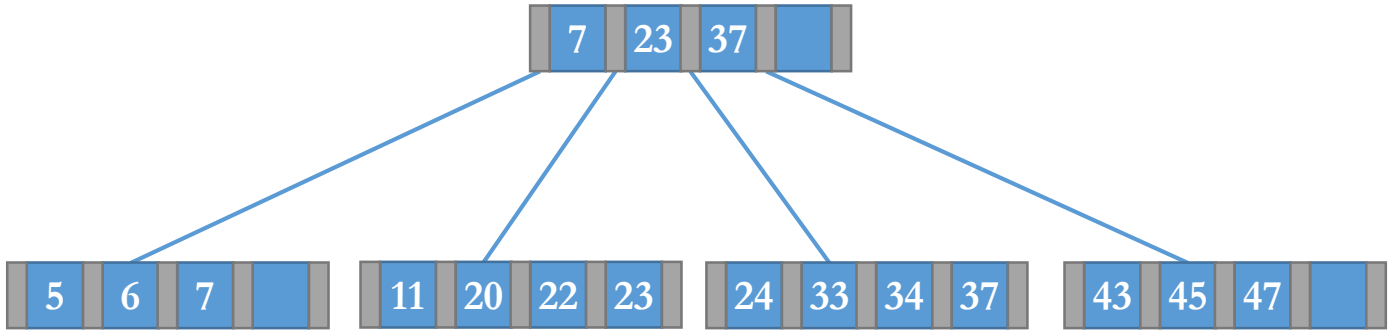


# B*-Tree – Example (cont.)

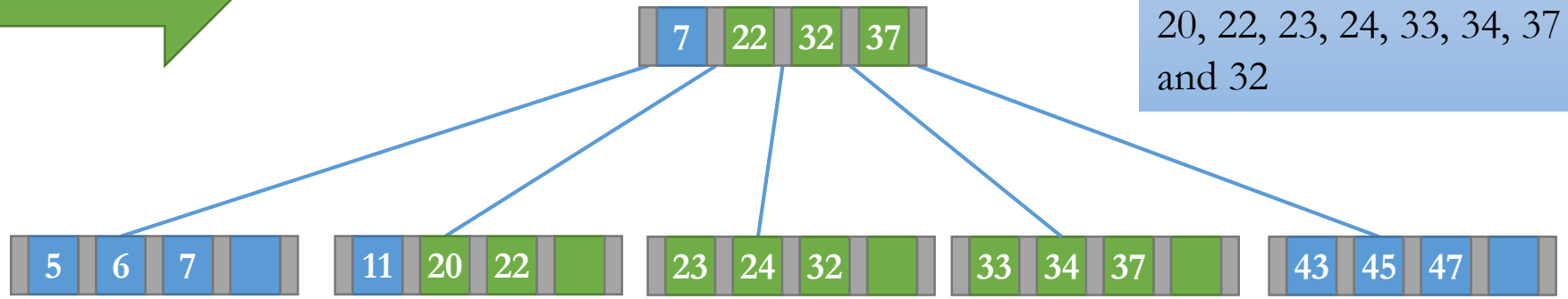



Insert 32

# B*-Tree – Example (cont.)







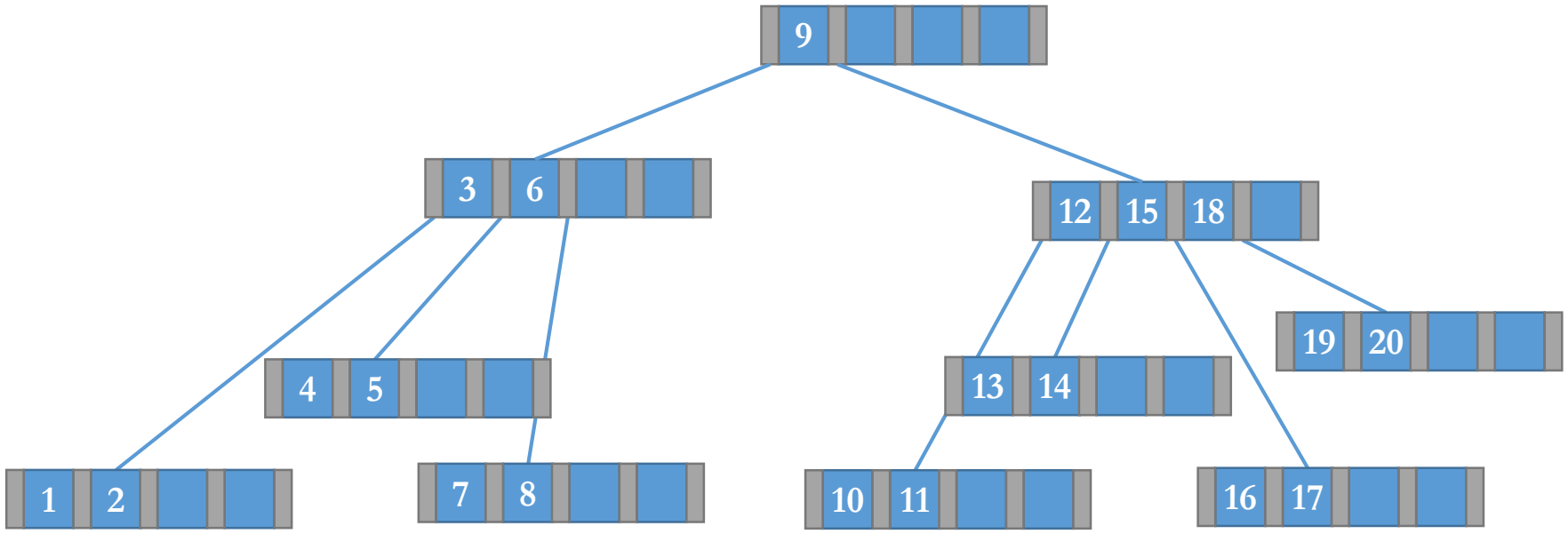

To be redistributed: 11, 20, 22, 23, 24, 33, 34, 37 and 32

# Deferred Splitting

- In the **original B-trees**, certain sequences of inserts **can lead to only half utilization** → **deferred splitting**
  - let us keep an **overflow page for each node**
  - when a **page overflows**, the overflown record is **inserted into** the respective **overflow page**
  - when **both the original and the overflow page are full**, the original page is **split** and the overflow page is emptied

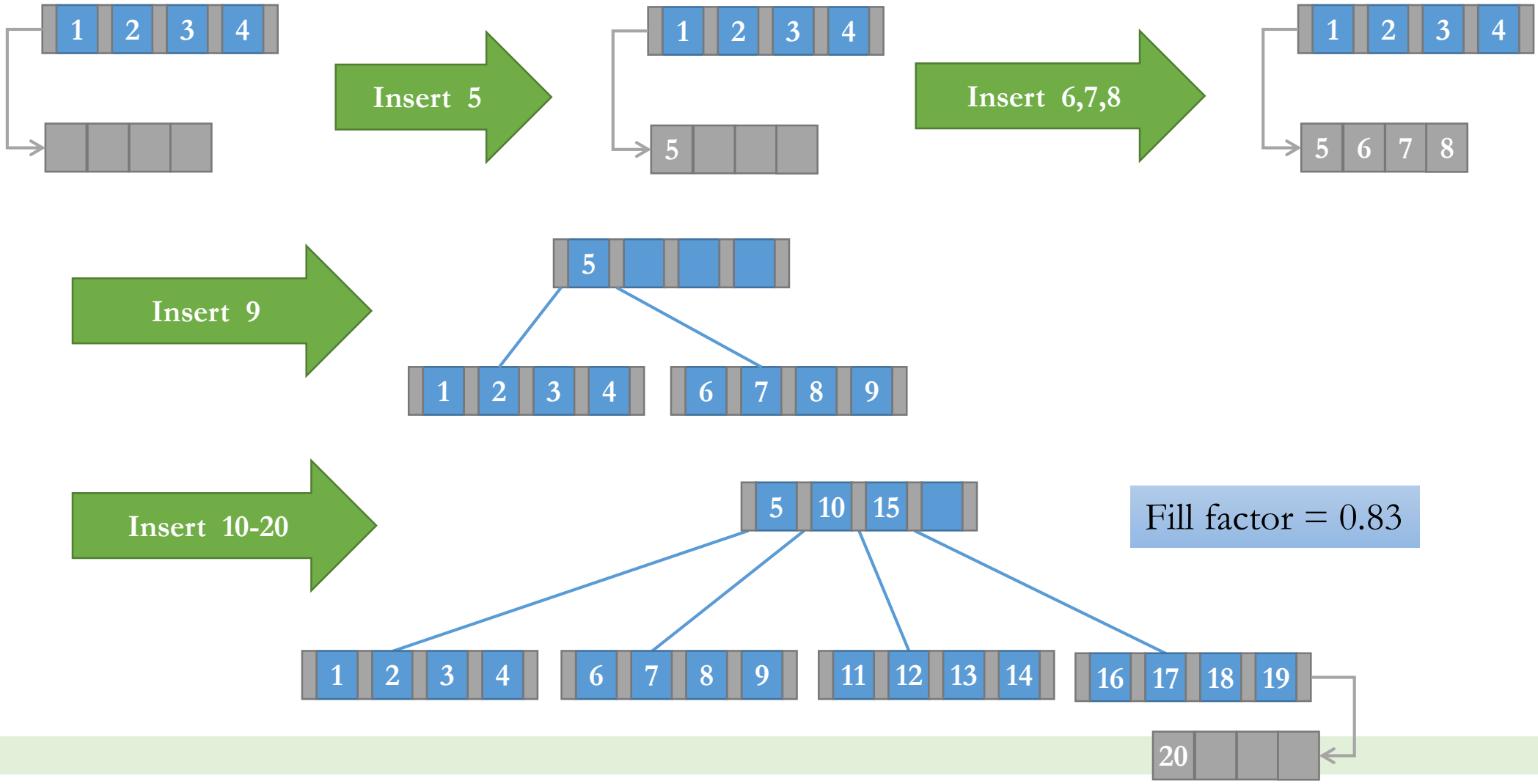
# Deferred Splitting - Example

Original nonredundant B-tree with m=5




Fill factor = 0.5 (worst possible)

# Deferred Splitting – Example (cont.)



# Variable Length Records

- Often we want to index not only numbers but also **strings** → **variable length-records (VLR)** → **different $m$ for different nodes**

- When **splitting** a page with VLR, rather **length of the records is taken into account** than the number of records → distribution is driven by the resulting length → **can lead to violation** of the condition regarding the **minimum number of records** in a B-tree

  - longer records tend to get closer to the root → **lower arity close to the root**
  - when **merging**, a short record can be replaced by a longer one → **height increase**

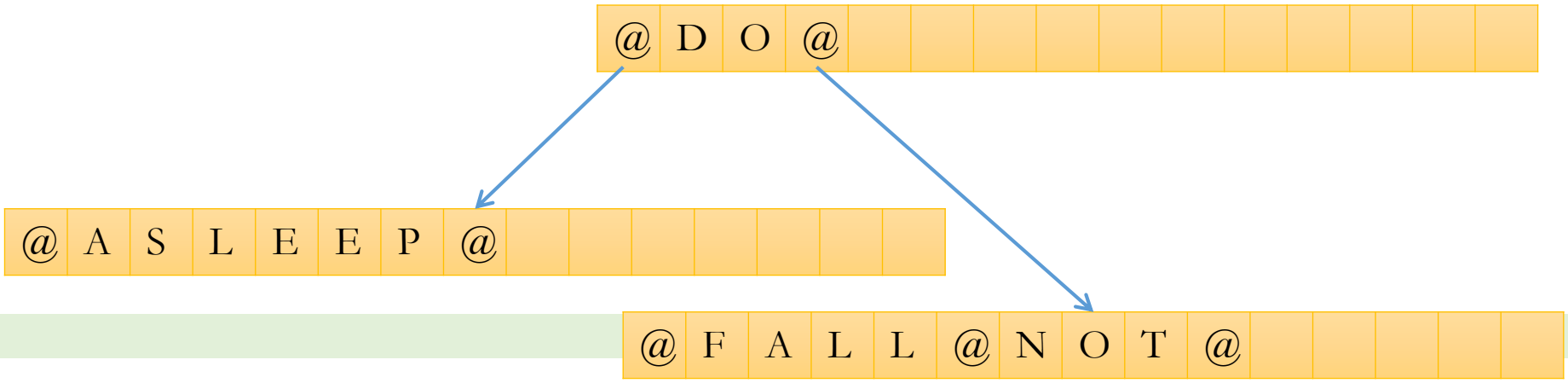
# VLR - Example

Representing the sentence: "DO NOT FALL ASLEEP"

- node size is 15
- pointers represented by @
  - for sake of simplicity let us consider size of a pointer to be identical to the size of a character

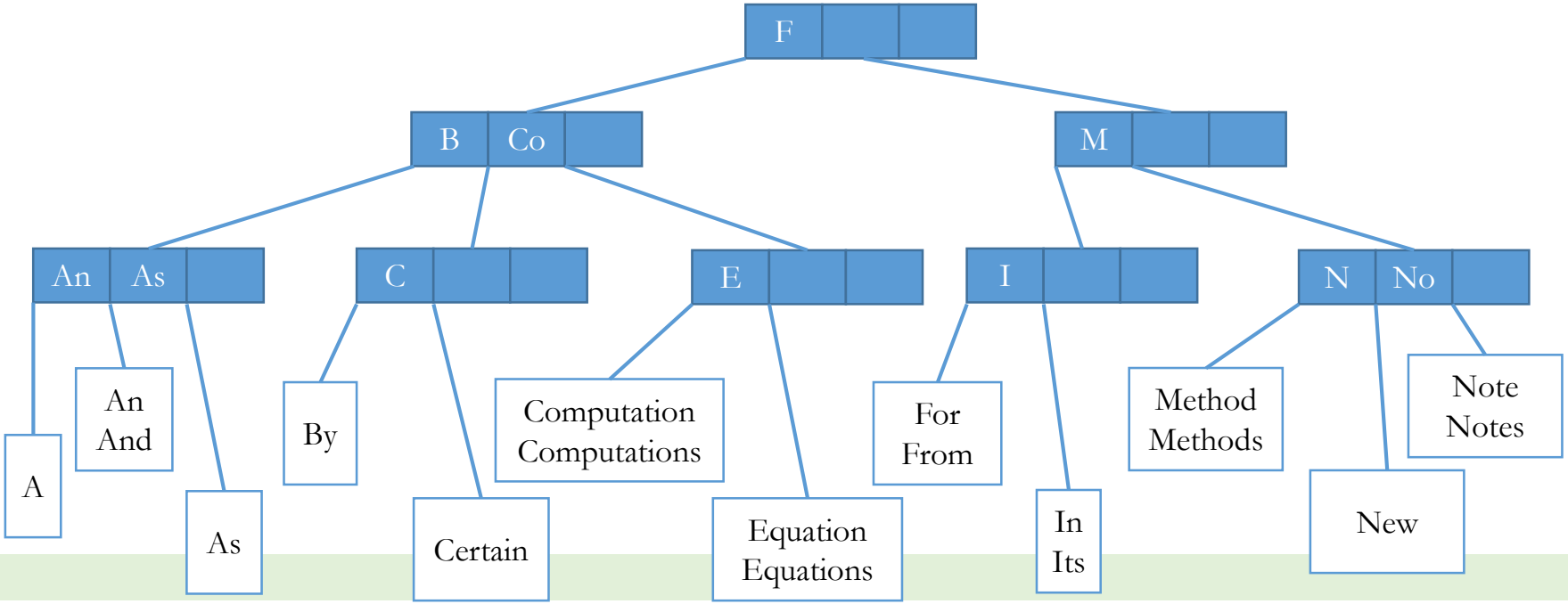
| @ | D | O | @ | F | A | L | L | @ | N | O | T | @ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|

- Inserting "ASLEEP" causes overflow → splitting
- Sequence to be split: @ASLEEP@D**O**@FALL@NOT@
  - → O is the middle character

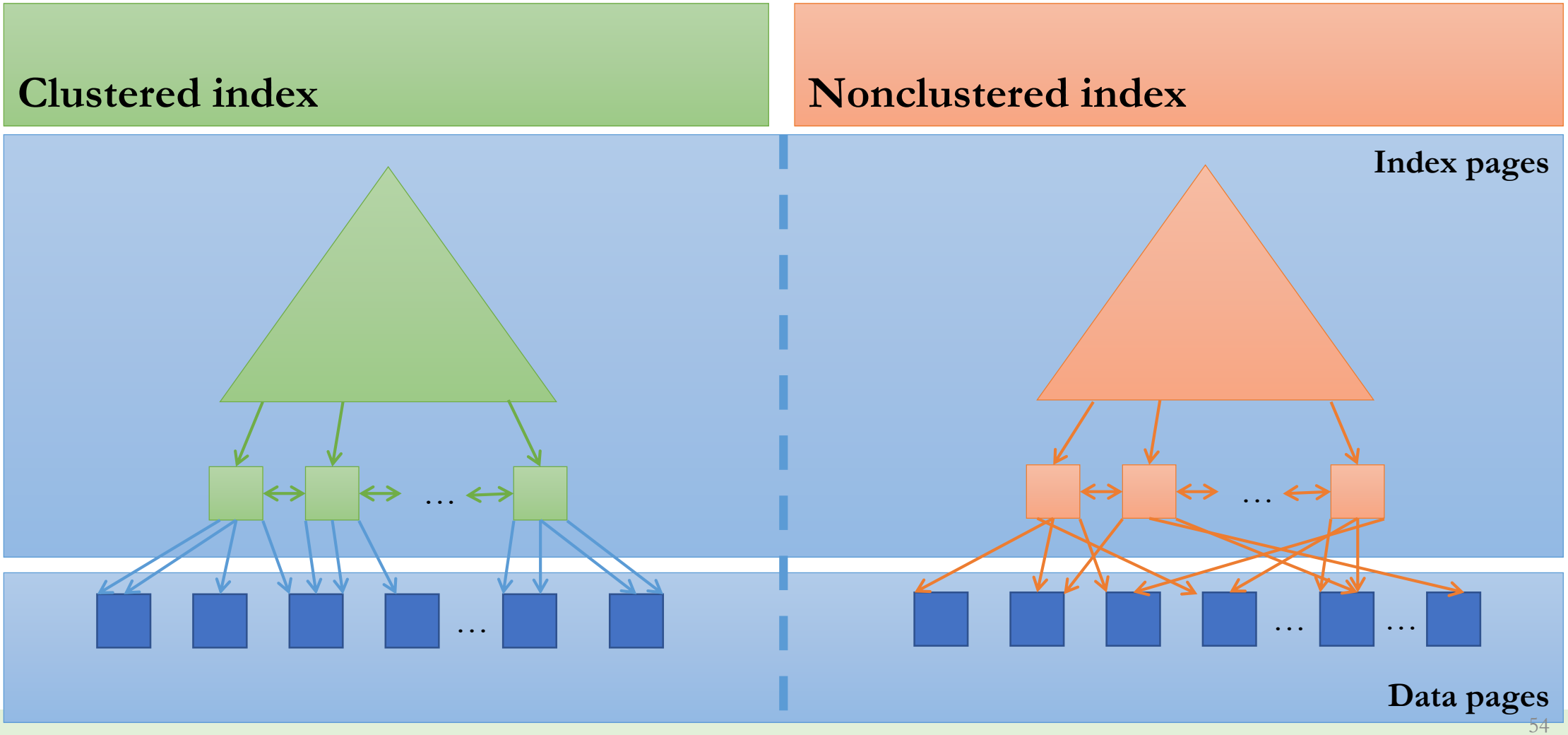


# Prefix (B-)Tree

- Modification of the redundant B-tree
  - inner nodes keys do not have to be subsets of the keys in the leaf level
  - **inner nodes keys have to separate**
  - **smaller keys** lead to **higher node capacity** → lower trees → faster access
  - suitable choice of separators are **prefixes** of the keys



# Clustered vs Nonclustered Indexes



# Clustered vs Nonclustered Indexes

- **Clustered index**
  - **logical order** of the key values **determines the physical order** of the corresponding data records, i.e., the order of the data in the data file follows the order of data in the index file
  - a file can be **clustered over at most one search key** → on **clustered index**
  - basically corresponds to the idea of index-sequential file organization
- **Nonclustered index**
  - order of data in the index and the primary file is not related
  - multiple nonclustered indexes can exist
- **Motivation**
  - when range range querying over a **nonclustered index, every record** might reside in a **different data page**
  - clustered index should be defined over an attribute over which range scan often happens

# (Non)Clustered Indexes - SQL

```
CREATE TABLE Product(
     id INT PRIMARY KEY NONCLUSTERED,
     code NVARCHAR(5),
     name NVARCHAR(50),
     type INT);

CREATE NONCLUSTERED INDEX ixProductCode ON
Product(Code);

CREATE CLUSTERED INDEX ixProductName ON
Product(Name);
```

# Indexes in Existing Leading Database Systems

| | Oracle 11g | Microsoft SQL Server 2016 | PostgreSQL 9.2 | MySQL 5.5 |
|---|---|---|---|---|
| Standard index | B+-tree | B+-tree | B+-tree | B+-tree |
| Bitmap index | Yes | No | No | No |
| Hash index | Yes (clustering) | Yes (clustering) | Yes | Yes |
| Spatial index | R-tree | B+-tree (mapping 2D to 1D using Hilbert space filling curve) | R-tree | R-tree |

# Less Traditional Approaches to Indexing

- Update-optimized structures
  - **Buffered repository tree (BRT)**
  - **Streaming B-tree**

- Cache-oblivious structures
  - **Cache-oblivious B-tree**
  - **Streaming B-tree**

# B-Tree Insert Drawback

- **Inserting** in (amortized) time $\boldsymbol{O(\log_b N)}$ is **suboptimal**

- With search/insert tradeoff one can get much **faster inserts** for **increase in the search** time
  - suitable for applications with lot of inserts (e.g., streaming databases (sensor data), file systems, …)

| | Search | Insert/Delete |
|---|---|---|
| B-tree | $O(\log_b \frac{N}{b})$ | $O(\log_b \frac{N}{b})$ |
| **BR-tree** | $\boldsymbol{O(\log_2 \frac{N}{b})}$ | $\boldsymbol{O(\frac{1}{b}\log_2 \frac{N}{b})}$ * |
| $B^{\epsilon}$-tree | $O(\frac{1}{\epsilon}\log_b \frac{N}{b})$ | $O(\frac{1}{\epsilon b^{1-\epsilon}}\log_b \frac{N}{b})$ * |
| $B^{1/2}$-tree | $O(2\log_b \frac{N}{b})$ | $O(\frac{2}{\sqrt{b}}\log_b \frac{N}{b})$ * |

* amortized

# Buffered Repository Tree

- Buchsbaum,Goldwasser,Venkatasubramanian,Westbrook, 2000 (BGVW2000)

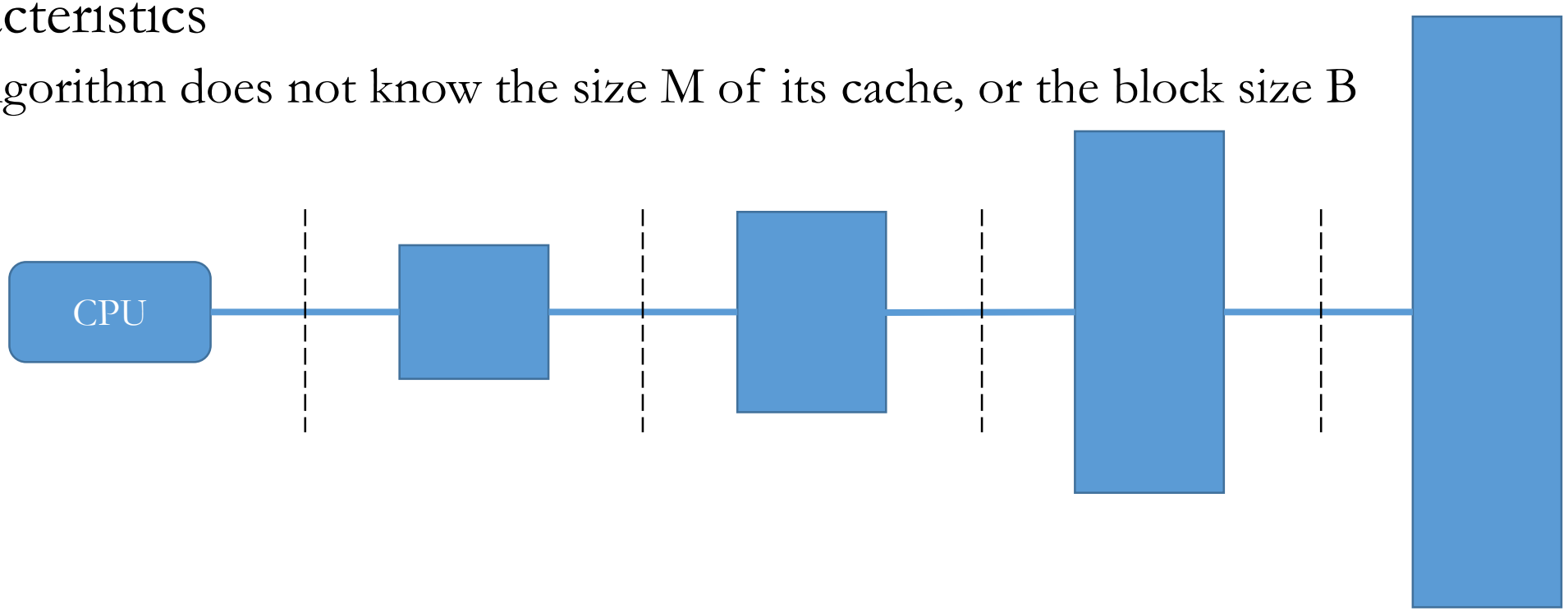
- The buffered repository tree is an **augmented (2,4) tree**
  - each **leaf node** holds up to $\boldsymbol{b}$ **items**
  - each **inner node** contains **keys** (2-4) and **one buffer** holding up to $\boldsymbol{b}$ **items**
  - all items stored in descendant leaves and buffers of a node obey the standard search criteria with respect to the rank among its sibling(s)

- **Insert**
  - record is **inserted into the root's buffer**
  - when a **buffer overflows**, the **data are redistributed to the descendants** and the key items in that node
    - Any time a buffer is distributed to children nodes during an insert, we apportion the $O(1)$ I/Os uniformly to the B items that are distributed. Each item is thus charged $O(\frac{1}{b})$ I/Os per level

- **Search**
  - the **root buffer is scanned**
  - if the item is not found, the procedure **recursively descends into the next level** based on the keys and scans the respective buffer at given level → $\boldsymbol{O(\log_2 N)}$

# Cache-oblivious model

- Full memory hierarchy where each memory can have different characteristics
  - Algorithm does not know the size M of its cache, or the block size B




Full memory hierarchy (different types of caches and external memory types)

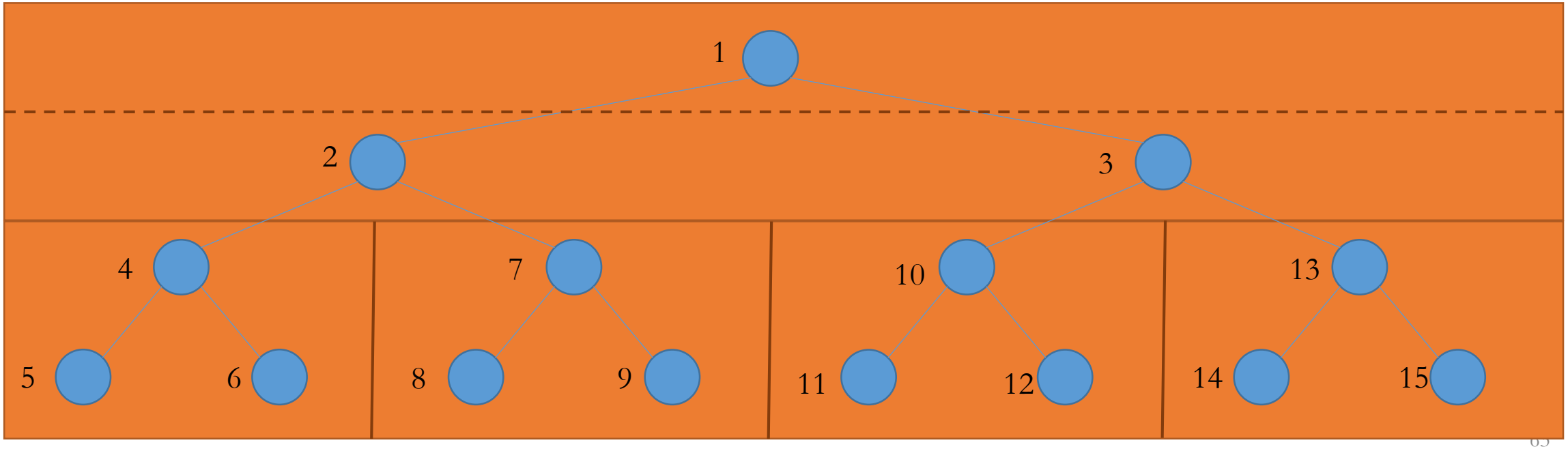
# COM search trees

- Search tree in **RAM model** $O(\log_B N)$ search, insert, delete
  - Constant factor for search 1
- Search tree in **COM model** $O(\log_B N)$ search, insert, delete
  - Constant factor for search $e$ (we will show for 4)

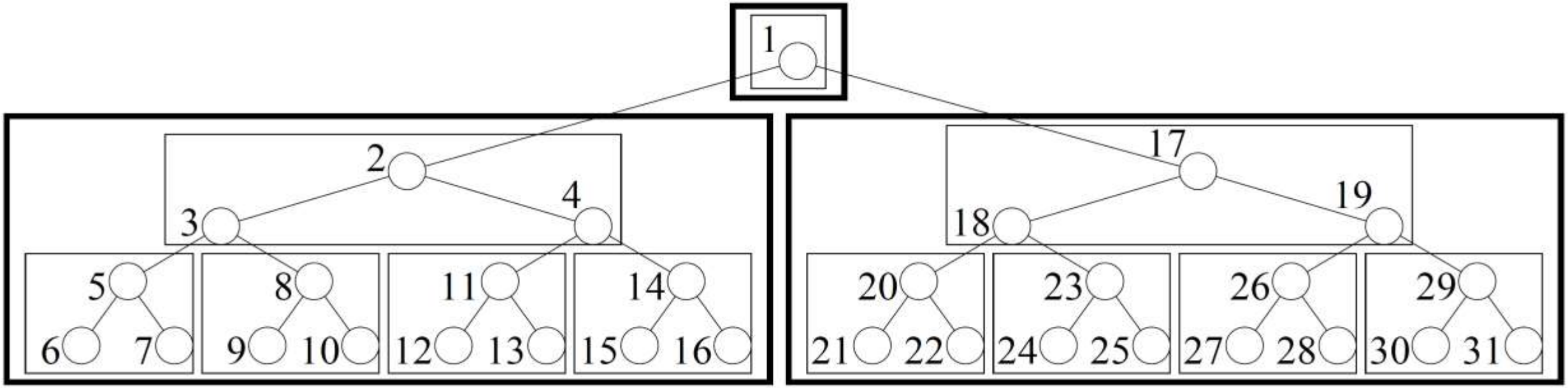
- **Based on binary trees** → good enough mapping between the order of nodes and the sequential organization (array) so that traversal does not touch too many blocks on disk

# COM static search tree - idea

- **van Emde Boas ordering**
  - Cut off the tree in half → extract individual trees of size $\sqrt{N}$ → repeat recursively → concatenate + label



# vEB ordering example




source: Bender, Michael, Erik D. Demaine, and Martin Farach-Colton. "Cache-oblivious B-trees." Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on. IEEE, 2000.

# COM static search trees - complexity

- For given block size $B$, let us stop the recursion so that **size of the substrees $(T_S) \leq B$** (and $T_s$ is of maximum size)

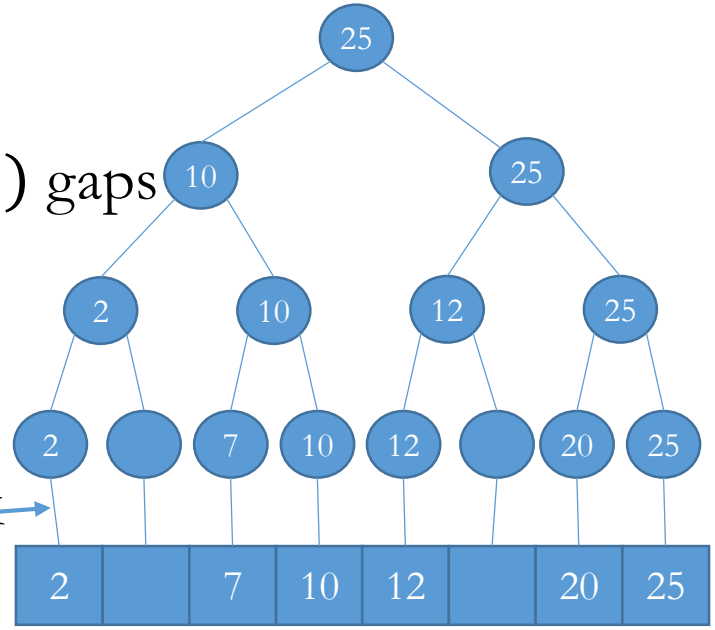
- $T_S$ can span **at most 2 blocks**

- Height of each $T_S$ is in $(\frac{1}{2}\log_2 B\,;\log_2 B) \rightarrow$ to read the tree from root to leaf takes

$$2\frac{\log_2 N}{\frac{1}{2}\log_2 B} = \mathbf{4\log_B N}$$

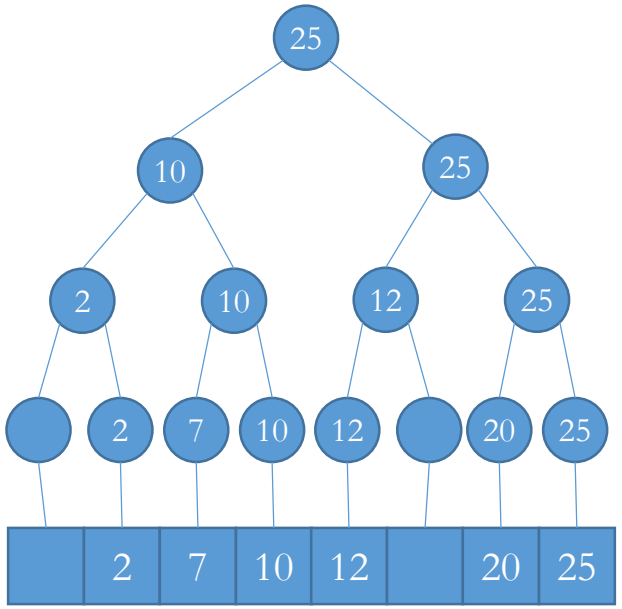
# COM B-tree

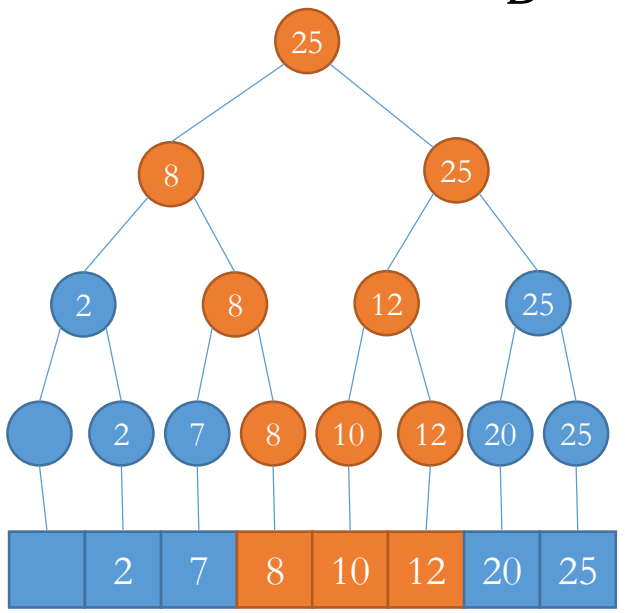
- Requires **ordered file maintenance** data structure
  - [Itai, Alon, Alan G. Konheim, and Michael Rodeh. *A sparse table implementation of priority queues*. Springer Berlin Heidelberg, 1981]
  - Enables to store N entries in given order in $O(N)$ space with $O(1)$ gaps (between two elements there is at most constant number of gaps)
  - Enables to insert/delete at given position in by moving entries in interval of size $O(\log_2^2 N)$ amortized by O(1) scans
- We stack **static search tree over ordered file** and crosslink (even the gaps)
  - Leafs contain the keys from the ordered file
  - Inner nodes contain maximum of left and right subtree



# COM B-tree update

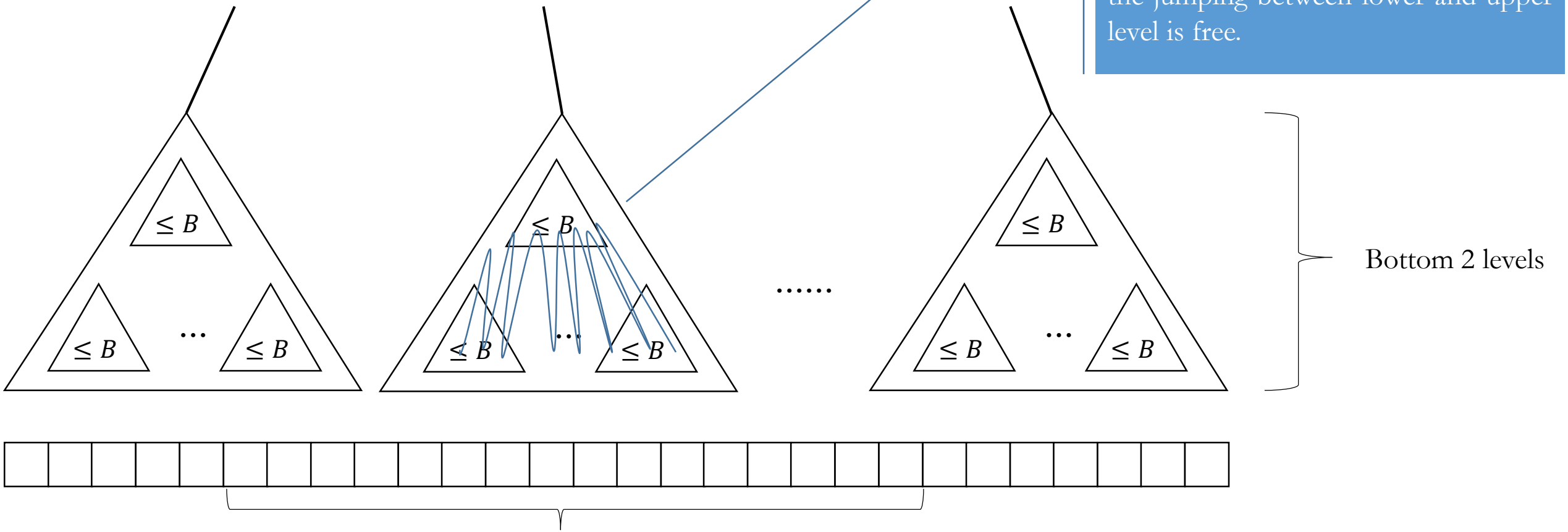
1. Search for the position in the ordered file $O(\log_B N)$
2. Update ordered file $O(\frac{\log_2^2 N}{B}) \rightarrow O(\log_B N)$
3. Propagate changes (post-order) $O(\log_B N + \frac{\log_2^2 N}{B}) \rightarrow O(\log_B N)$



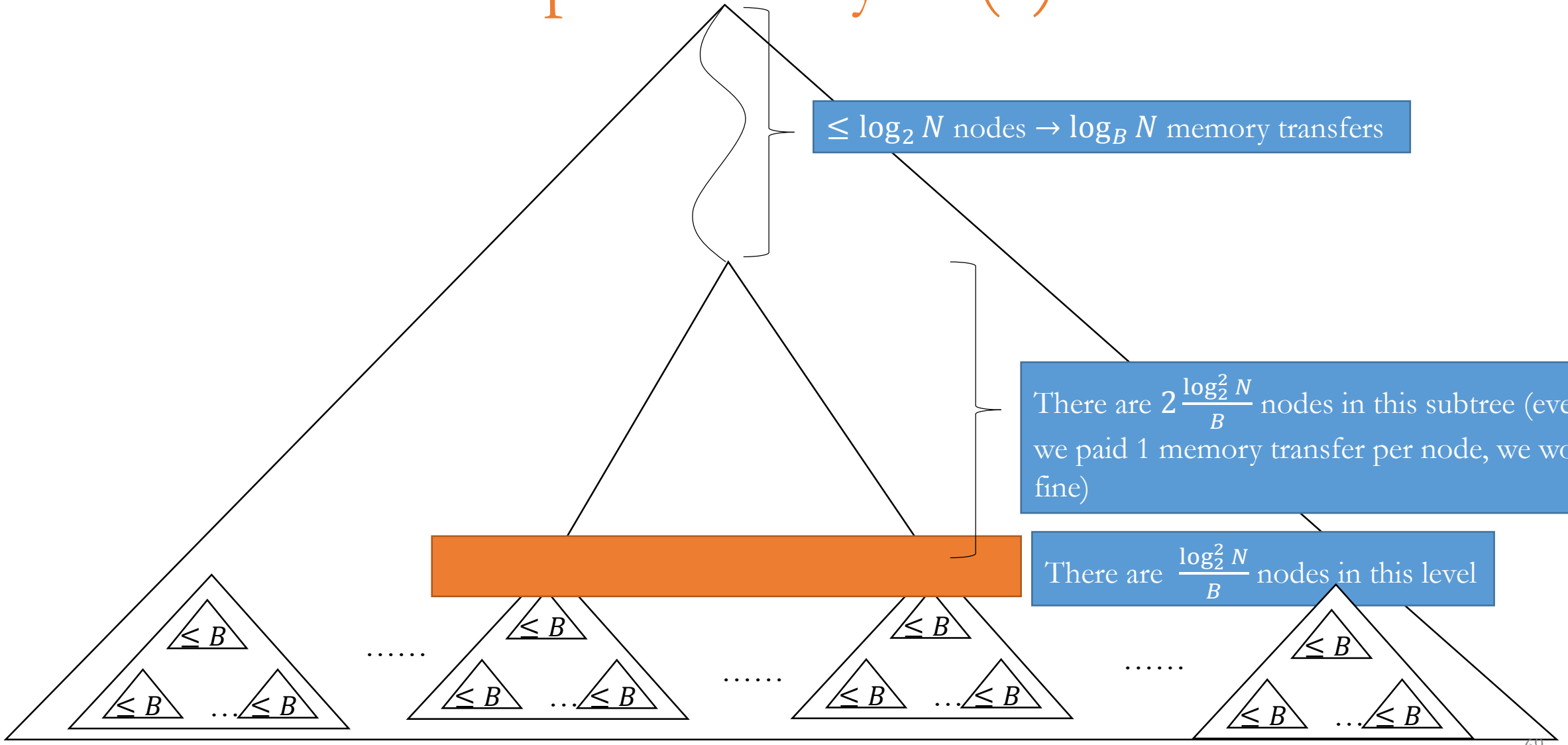
# COM B-tree update analysis (1)

Since we update post-order, to update the whole tree of size $B^2$, we first update maxes of the lower leftmost subtree, then go to the next one and so on. Each subtree is in at most two pages, so if we have cache of size 4, the jumping between lower and upper level is free.




$\log_2^2 N$ amortized $\rightarrow O(1 + \frac{\log_2^2 N}{B})$ trees (blocks) need to be visited in bottom 2 levels

# COM B-tree update analysis (2)



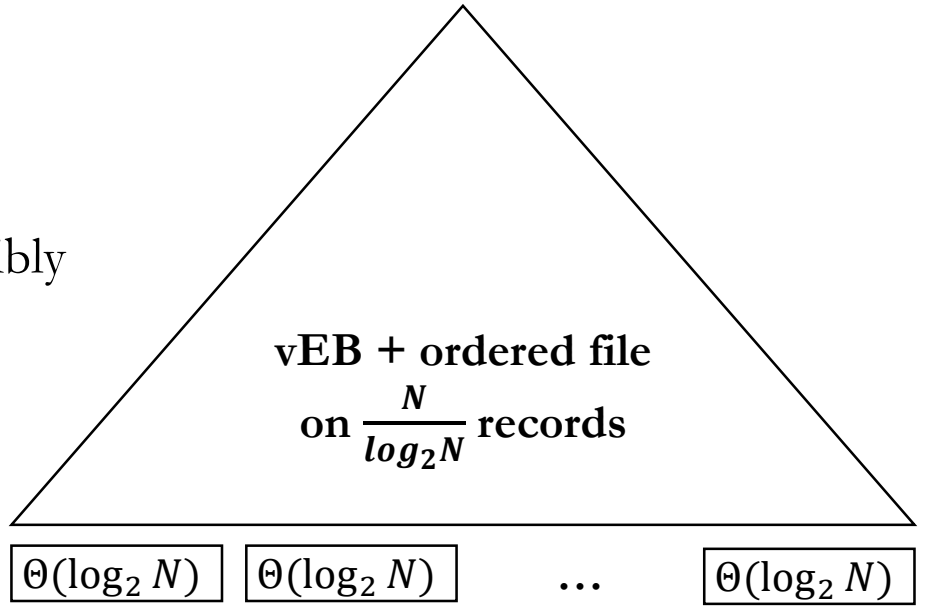
# COM B-tree update analysis (3)

Search

Propagation in the upper part of the tree

Update of the file and propagation in the lower part of the tree

- The cost of update is thus $O\left(\log_B N + \frac{\log_2^2 N}{B}\right)$
  - For $B$ big enough this is as good as B-tree
  - Using indirection to get $O(\log_B N)$
    - Split items into blocks of size $\Theta(\log_2 N)$
    - For each block, store only min in the structure
    - Search in the tree + block $(\frac{\log_2 N}{B})$
    - Update involves rewrite of the whole block + possibly split/merge of the blocks
      - utilization $\in (\frac{1}{4}\log_2 N\,; \log_2 N]$




$$O\left(\frac{\log_B N + \frac{log_2^2 N}{B}}{\log_2 N} + \frac{\log_2 N}{B}\right) = O\left(\frac{\log_2 N}{B}\right) < O(\log_B N)$$

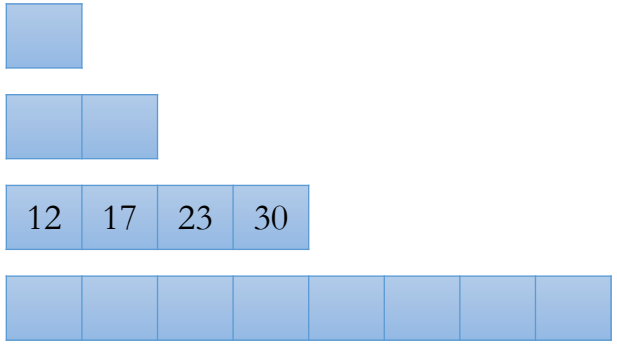
# Streaming B-trees

- Implemented in TokuDB engine of MySQL → **Fractal tree**
  - We show its older implementation which is based on COLA (current implementation is based on $B^{\epsilon}$-tree)

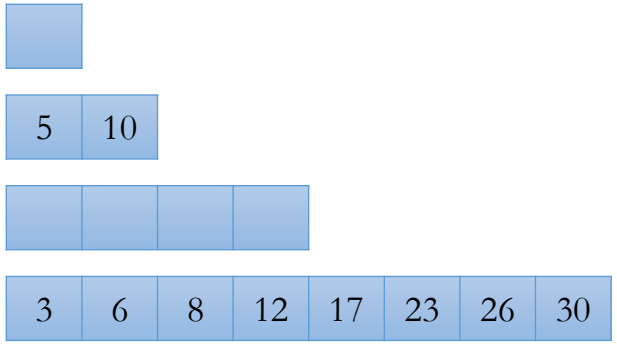
- Cache-oblivious lookahead array (COLA)
  - consists of $\lceil \log_2 N \rceil$ arrays where the $i$-th array stores $2^i$ elements
  - each array is either completely full or empty
  - CO since size of the block does not appear in the data structure
- Invariants
  - The $k$-th array contains items if and only if the $k$-th least significant bit of the binary representation of N (number of records) is a 1
  - Each array contains its items in ascending order by key
  - When a new item is inserted we effectively perform a carry
    - we create a list of length one with the new item, and as long as there are two lists of the same length, we merge them into the next bigger size

# Simplified fractal tree - example

A fractal tree with 4 records

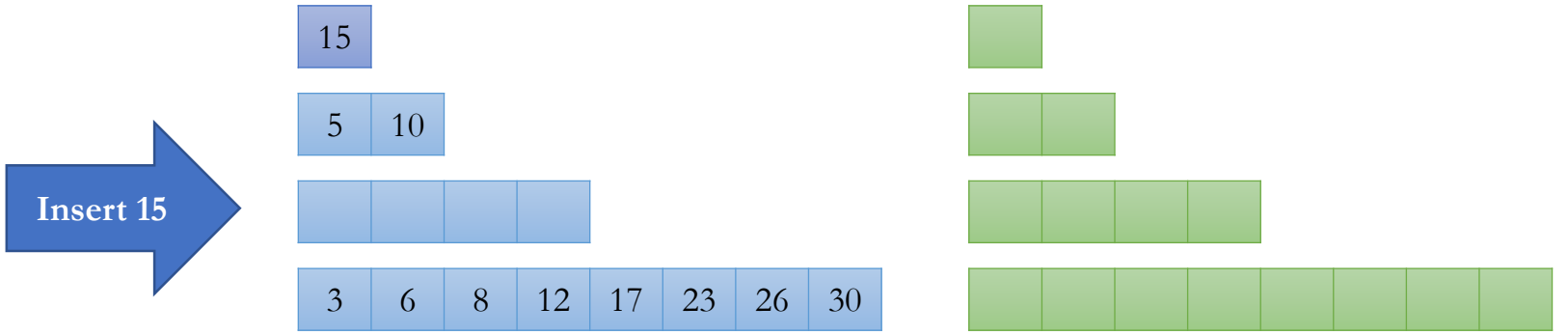



A fractal tree with 10 records




Every number can be written as the sum of powers of 2 → the non-zero members correspond to the non-empty arrays.

# Fractal tree - Insert

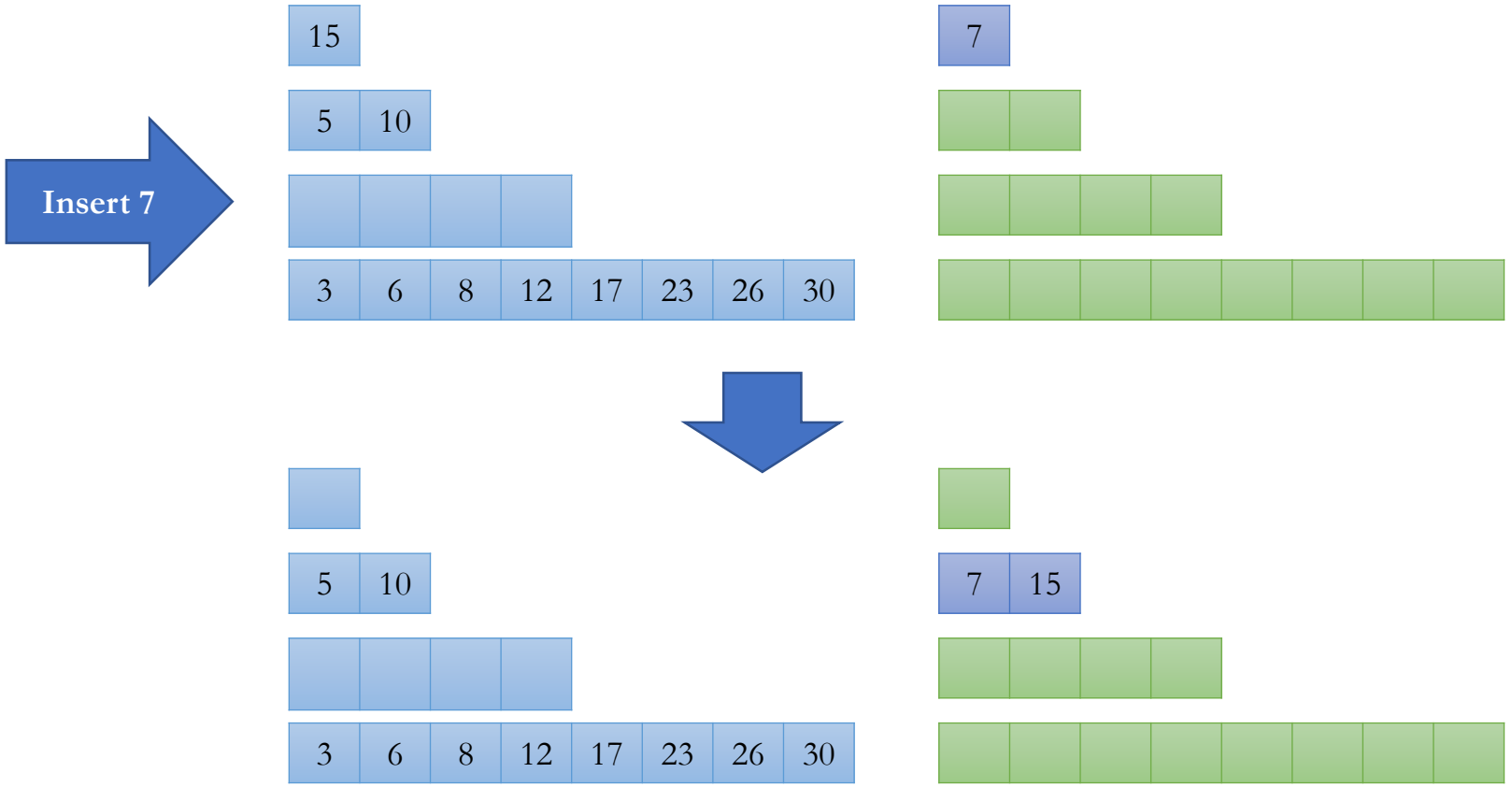



Record with key 15 fits into the 1-array

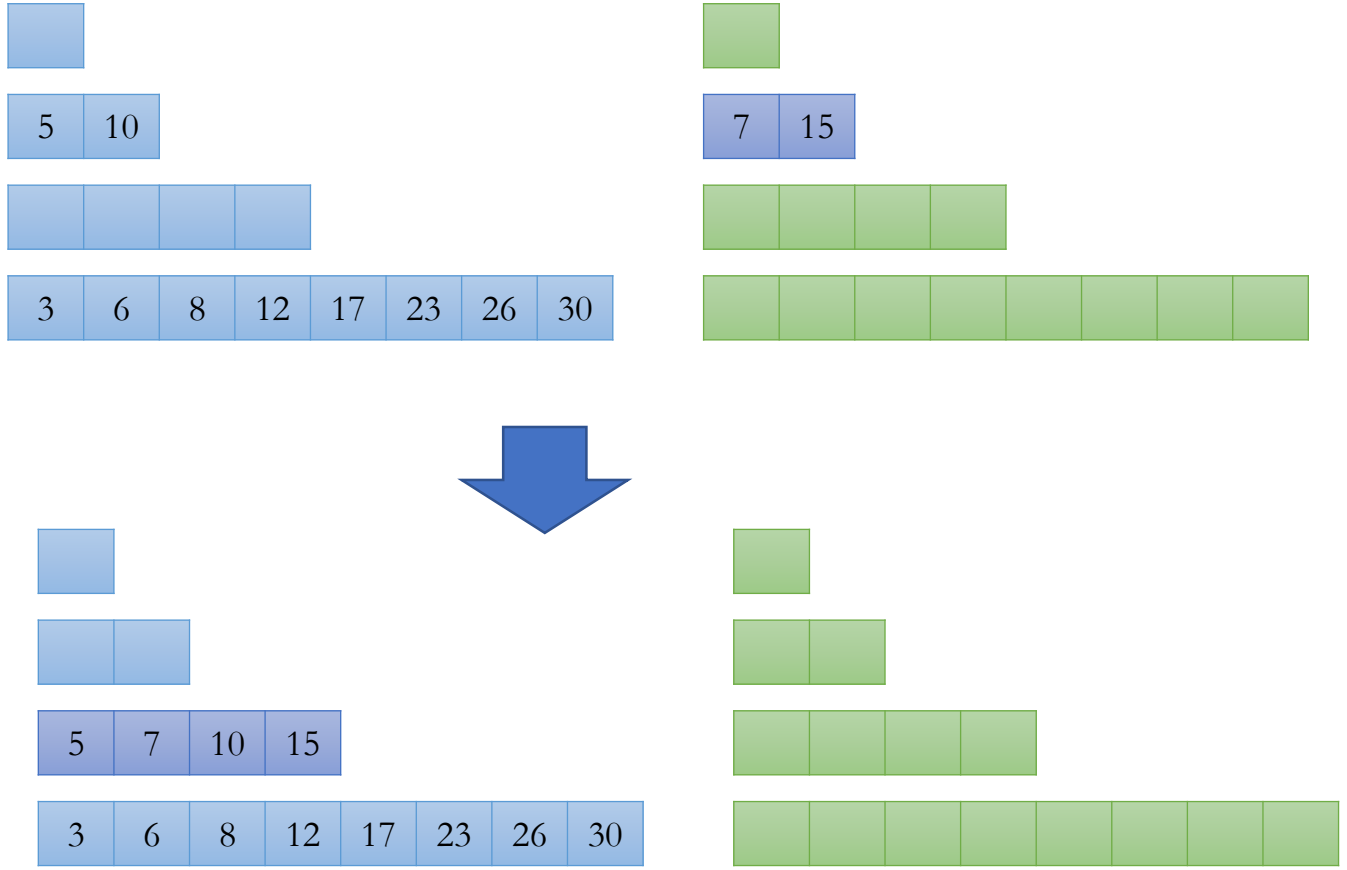
In order for insert to work, each array has a temporary storage.

At the beginning of each stage, all the temporary arrays are empty.

# Fractal tree – Insert (cont.)

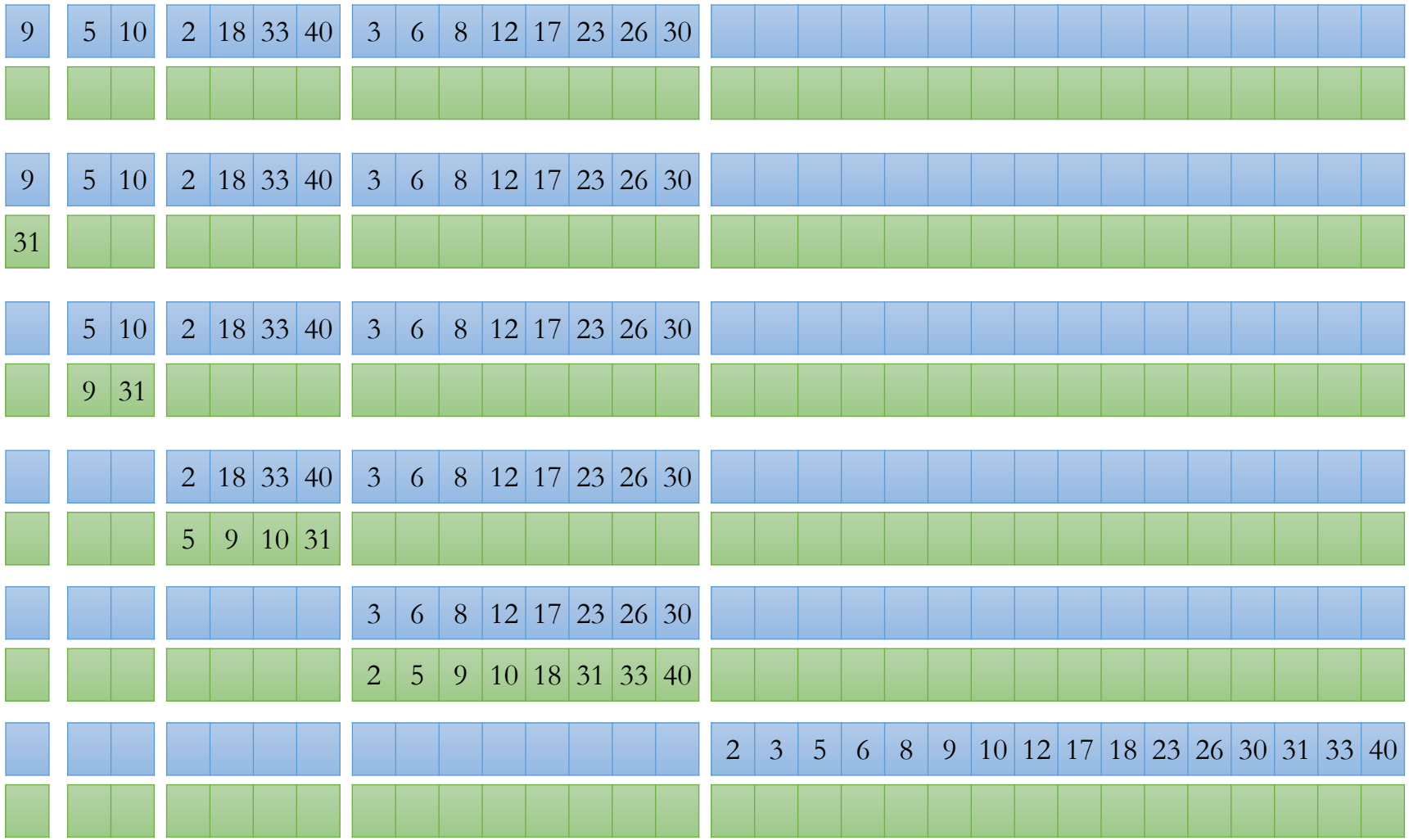


# Fractal tree – Insert (cont.)



# Fractal tree – Insert (cont.)

In some cases the cascade can be very long.

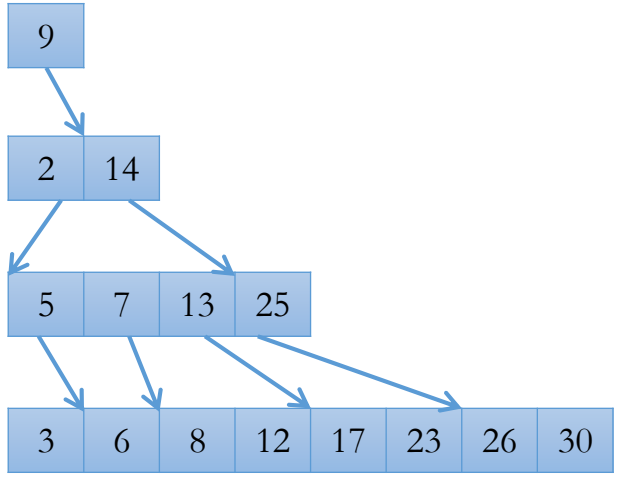


# Fractal tree - complexity

- **Insert**
  - $\boldsymbol{O(\frac{1}{b} log_2 N)}$ **amortized** and $O(\log_2 N)$ worst-case block transfers
    - each element is merged at most $O(\log_2 N)$ times
    - cost to merge 2 arrays of size X is $O(X/b)$ block I/Os
    - cost per element to merge is $O(1/b)$ since we merge $X/b$ blocks each having $b$ items
- **Search**
  - $\boldsymbol{O(\log_2^2 N)}$ **block transfers**
    - **naïve solution** where each level is searched **using binary search** which takes $O(\log_2 N)$ time (basic COLA)
  - $\boldsymbol{O(\log_2 N)}$
    - **using forward pointers**

# Forward pointers

- **Fractional cascading** (Chazelle, Guibas, 1986)
  - **each element** contains a **forward pointer** to where it fits in the next-level array
  - when searching for an element, only a part of the array needs to be scanned



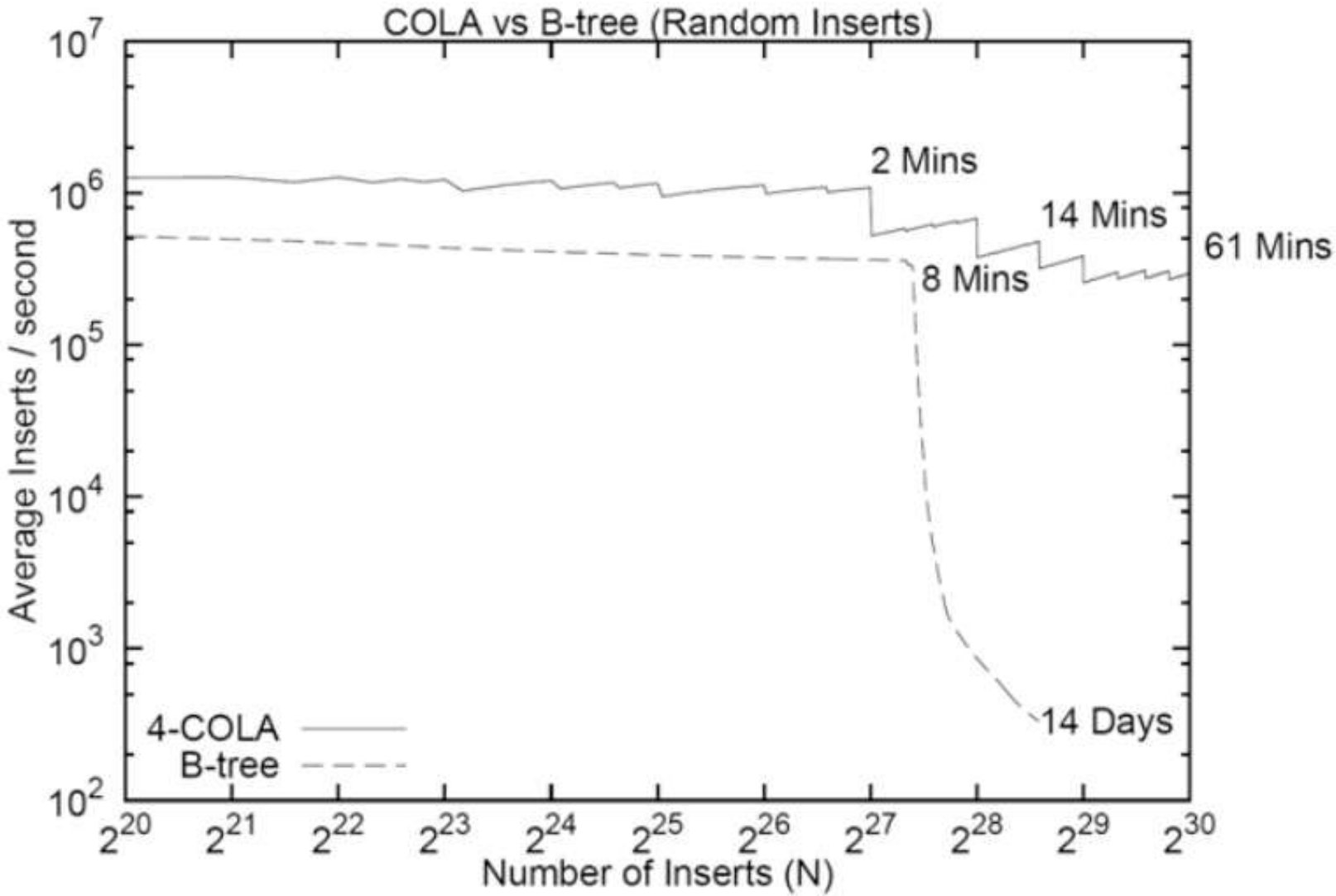
# Fractal tree – insert (Time)

- **B-tree:** $O(\log_B N) = \boldsymbol{O(\frac{\log_2 N}{\log_2 B})}$
- **Fractal tree:** $\boldsymbol{O(\frac{\log_2 N}{B})}$

- $N = 10^9, B = 4096$

$$\boldsymbol{\log_2 10^9 = 30}$$
$$\boldsymbol{\log_2 4096 = 12}$$

- **Inserting into B-tree** requires $\left\lceil \frac{\log_2 N}{\log_2 B} \right\rceil = \left\lceil \frac{30}{12} \right\rceil =$ **3 disk accesses**
- **Inserting into fractal tree** requires $\left\lceil \frac{\log_2 N}{B} \right\rceil = \left\lceil \frac{30}{4096} \right\rceil =$ **0.008 disk accesses**


COLA vs B-tree (Random Inserts)
Average Inserts / second
Number of Inserts (N)
$10^7$
$10^6$
$10^5$
$10^4$
$10^3$
$10^2$
$2^{20}$ $2^{21}$ $2^{22}$ $2^{23}$ $2^{24}$ $2^{25}$ $2^{26}$ $2^{27}$ $2^{28}$ $2^{29}$ $2^{30}$
2 Mins
14 Mins
61 Mins
8 Mins
14 Days
4-COLA
B-tree