PRINCIPLES OF DATA ORGANISATION

Hierarchical Indexing - Basics

MOTIVATION

- Key, pointer pairs ~ index
- Search trees (binary tree, a-b tree,…)
- Unlike hashing, trees allow retrieving a set of records with keys from a given range.
- Tree structures use “clustering” to efficiently filter out non-relevant records from the data set

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”
  - We do not want a tree that looks like a list
- Inserting or deleting a record in B-tree causes only local changes and not rebuilding of the whole index

B-TREE

Bonus fact: … Experiments have been performed with indexes up to 100 000 keys.

An index of size 15 000 (100 000) can be maintained with an average of 9 (at least 4) transactions (update, delete, search) per second on an IBM System/360 Model 44 with a 2 311 disc drive.

B-TREE

B-trees are balanced $\boldsymbol{m}$-ary trees fulfilling the following conditions:

- The root has at least two children unless it is a leaf
- Every inner node except the root has at least $\lceil \boldsymbol{m/2} \rceil$ and at most $\boldsymbol{m}$ children
    - Each node is at least half full
- Every node contains at least $\lceil \boldsymbol{m/2} \rceil \boldsymbol{-1}$ and at most $\boldsymbol{m-1}$ (pointers to) data records
    - Pointers to data, discriminators and pointers to children are tightly coupled
- Each branch has the same length

Node organisation:

$$p_0, (k_1, p_1, d_1), (k_2, p_2, d_2), \ldots, (k_n, p_n, d_n), u$$

p_i – pointers to the child nodes
k_i – keys/discriminators
d_i – data
u – unused space
$\lceil \boldsymbol{m/2} \rceil \boldsymbol{-1} \leq n \leq \boldsymbol{m-1}$

Records (k_i, p_i, d_i) are sorted with respect to k_i.
For all k_j in subtree pointed by p_i : $k_i < k_j < k_{i+1}$

B-TREE

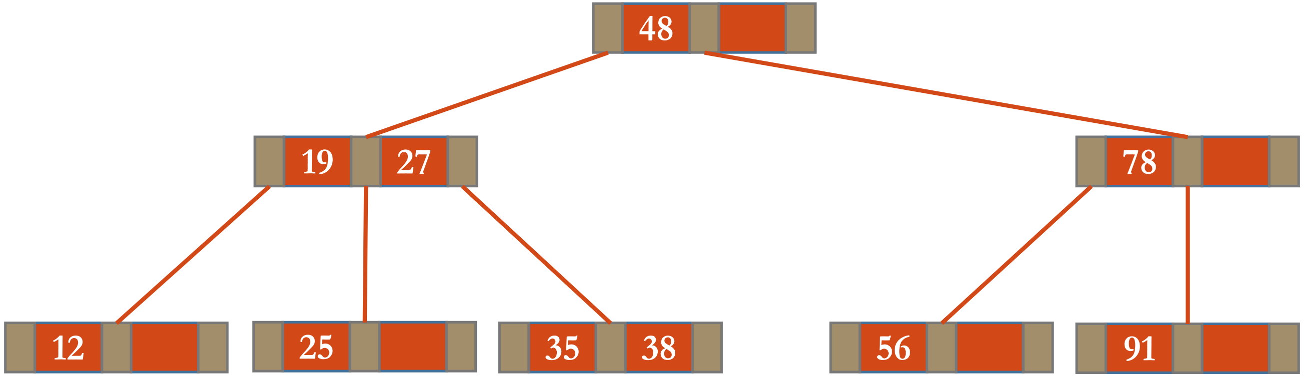
Non-redundant

- The presented definition introduced the non-redundant B-tree
- Each key value occurred just once in the whole tree
- Pointers to data are stored with values

Redundant

- Redundant B-trees store the data values in the leaves and thus have to allow repeating of keys in the inner nodes.
  - I.e. use $\leq$ instead of $<$ in the last condition
- The inner nodes do not contain pointers to the data records
  - Higher blocking factor
- More widespread

B-TREE EXAMPLE



Is is redundant or non-redundant?

B-TREE IMPLEMENTATION

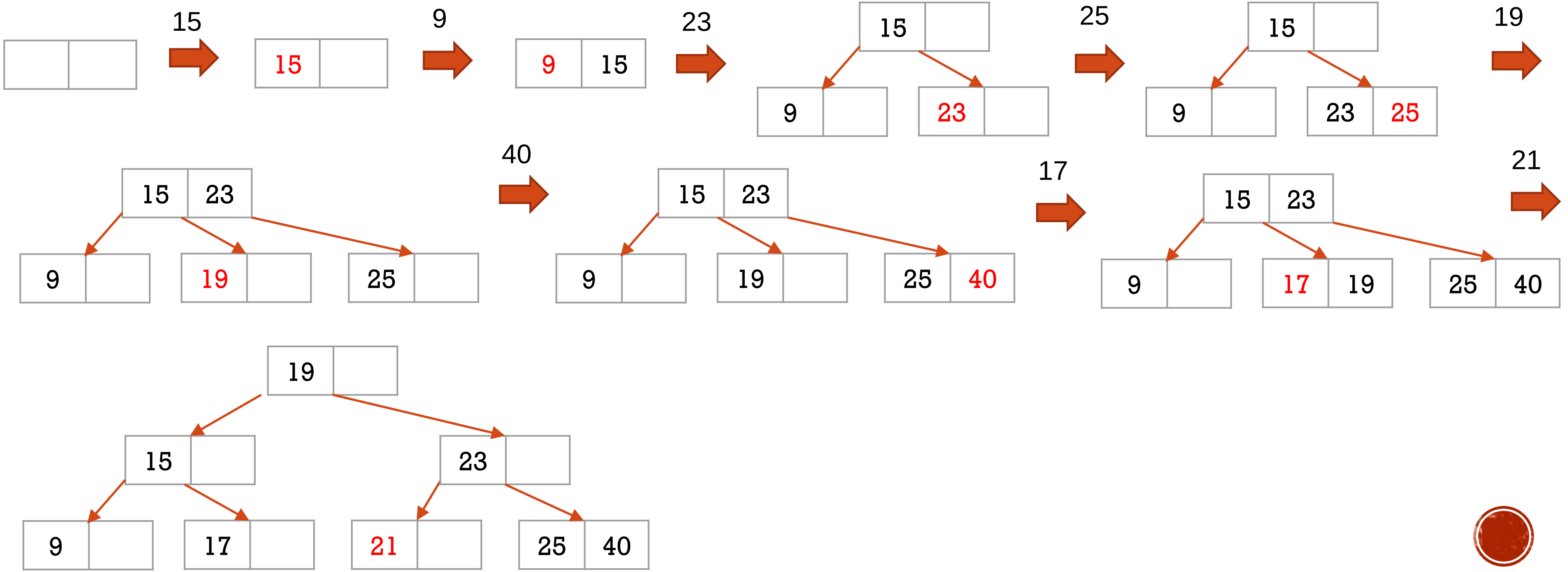
- Usually one page/block contains one node

Existing database management system:

- One page usually takes 8KB
- Redundant B-trees
  - Higher blocking factor of inner nodes
  - Range queries – values in leaves
- Data are not stored in the indexing structure itself but addressed from the leaf nodes
  - Multiple indices

EXAMPLE – NON-REDUNDANT B-TREE, INSERT

- Insert values:15,9,23,25,19,40,17,21
- m = 3



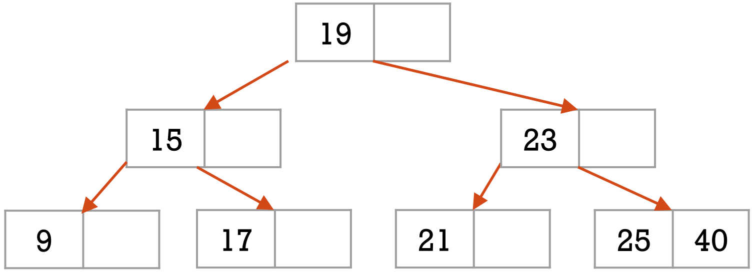
B-TREE – SEARCH

Searching a (non-redundant) tree T for a record with key k :

1. Enter the tree in the root node.
2. If the node contains a key k_i such that $k_i = k$ return the data associated with d_i.
3. Else if the node is leaf, return NULL.
4. Else find lowest i such that $k < k_i$ and set $j = i - 1$. If there is no such i set j as the rightmost index with existing key.
5. Fetch the node pointed to by p_j.
6. Repeat the process from step 2.

Example: search for 40

Remember: one node = one block



B-TREE – UPDATE

The logarithmic complexity is ensured by the condition that every node has to be at least half full.

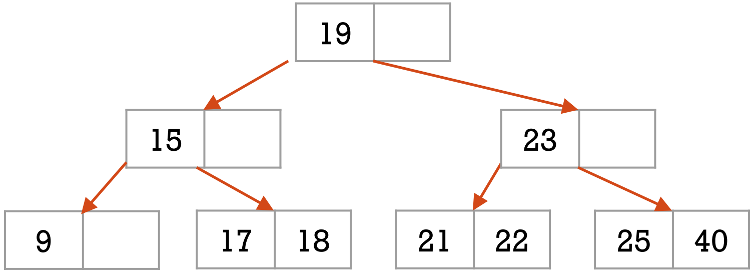
Inserting

- 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.
- Split cascade.

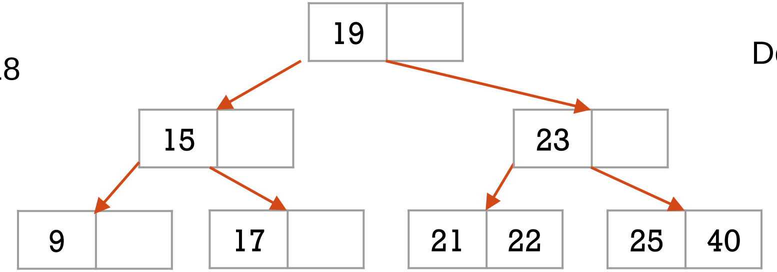
Deleting

- 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.
- Delete cascade

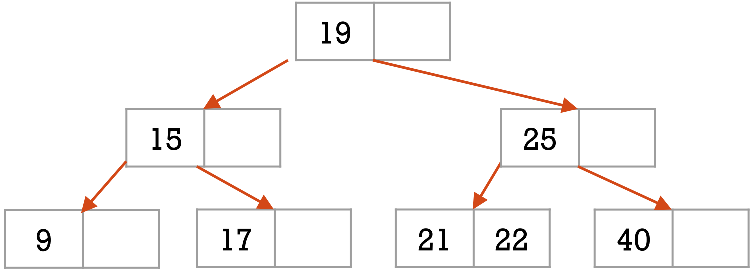
EXAMPLE – NON-REDUNDANT B-TREE, DELETE



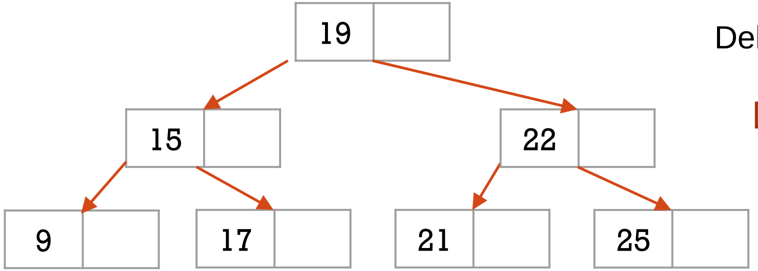
Delete 18



Delete 23



Delete 40



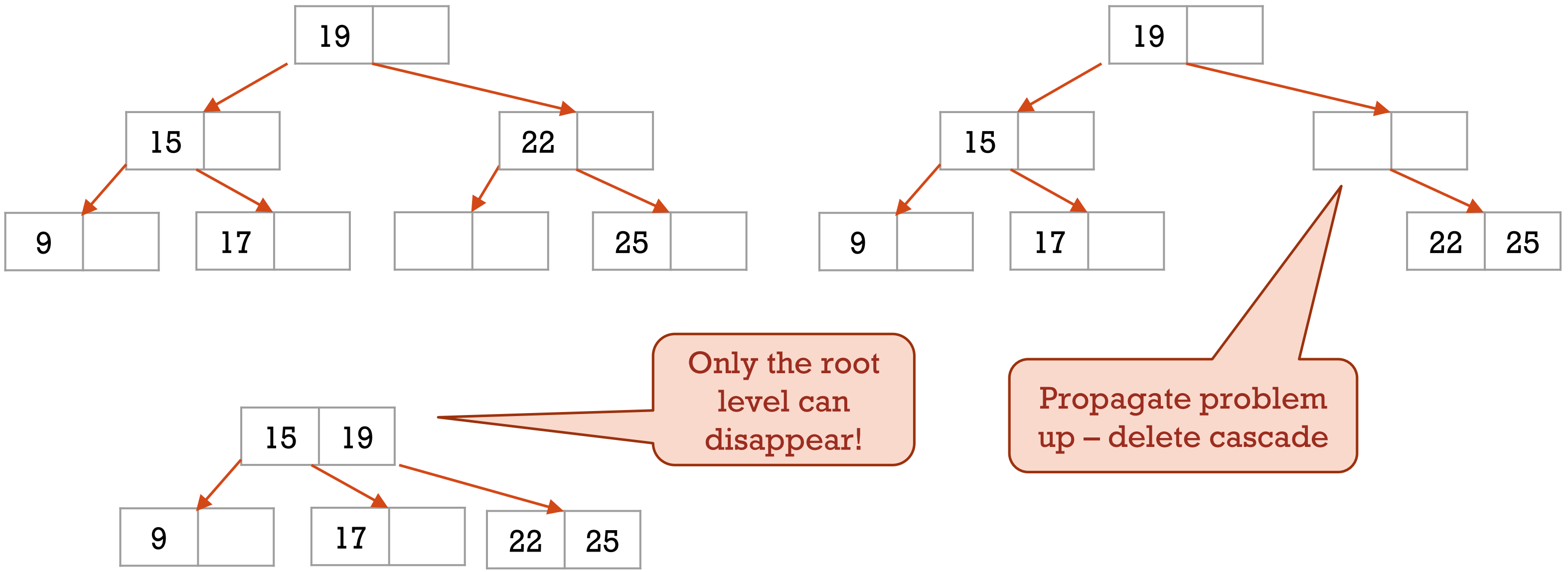
Delete 21

Borrow max from left or min from right subtree

Borrow from parent and nearest siblings

EXAMPLE – NON-REDUNDANT B-TREE, DELETE

The last step gradually:



B-TREE – INSERT

Insert into a (non-redundant) tree T for a record r with key k :

1. If the tree is empty, allocate a new node, insert the key k and (pointer to record) r and return.
2. Else find the leaf node L where the key k belongs.
3. If L is not full insert r and k into L in such a position that the keys are sorted and return.
4. Else create a new node L'.
5. 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'.
6. If P overflows, repeat step 5 for P else return.

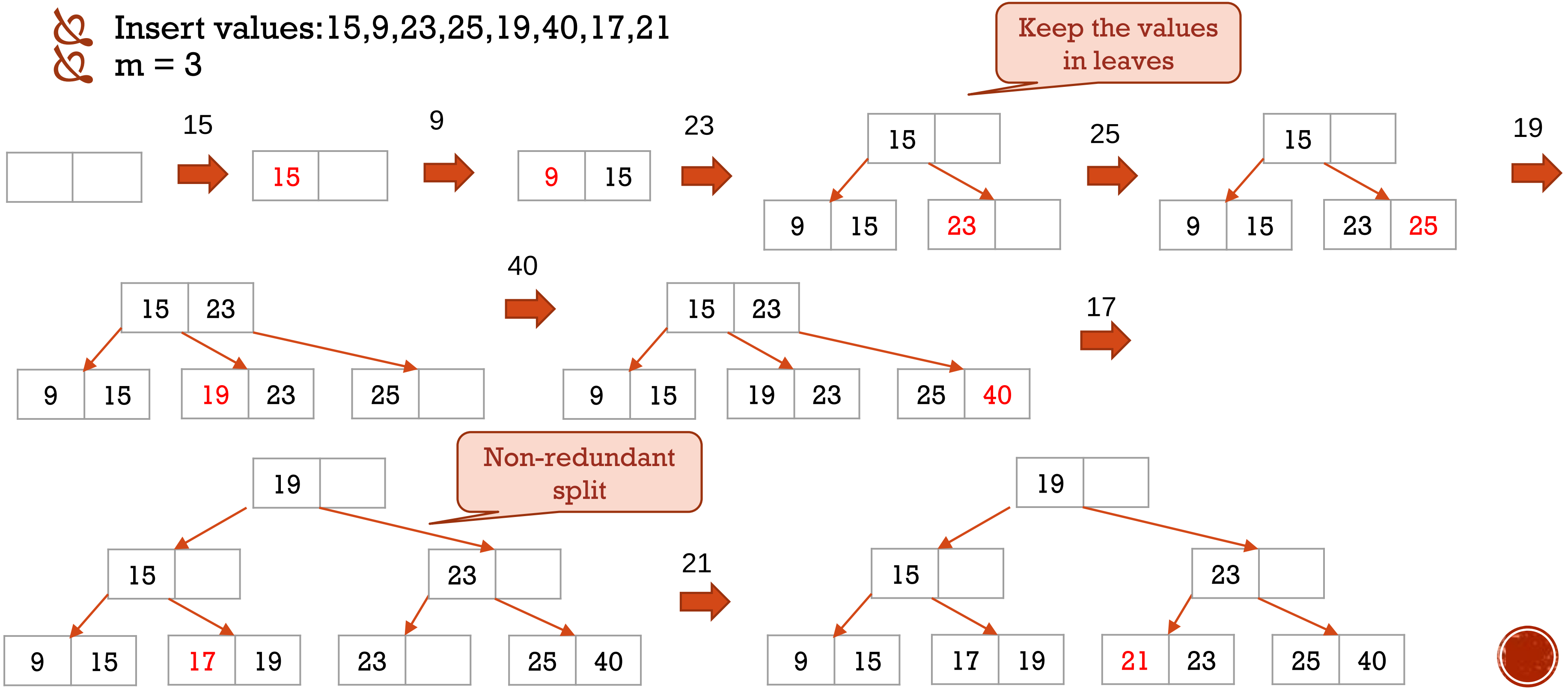
B-TREE – DELETE

Delete from tree T for a record r with key $\boldsymbol{k}$:

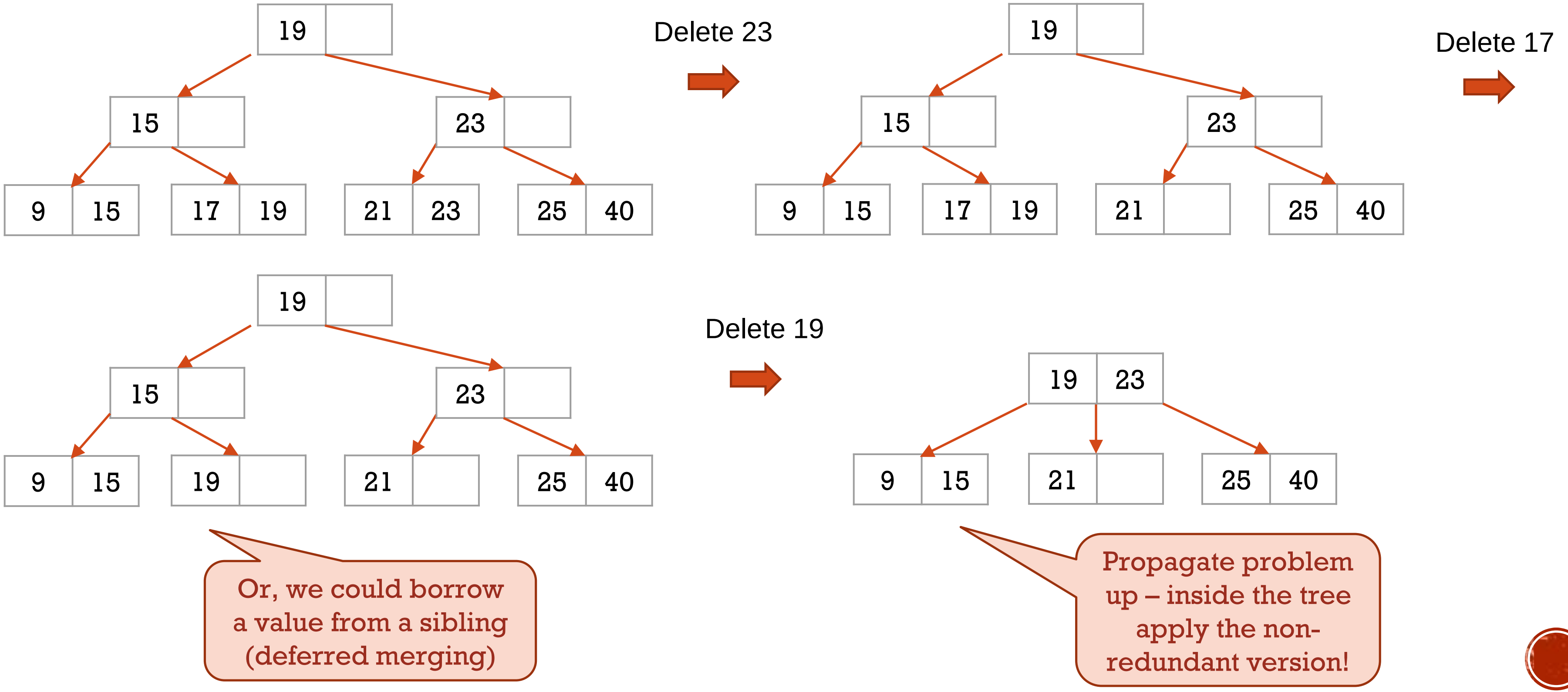
1. Find a node $\boldsymbol{N}$ containing the key $\boldsymbol{k}$.
2. Remove $\boldsymbol{r}$ from $\boldsymbol{N}$.
3. If number of keys in $\boldsymbol{N} \geq \lceil \boldsymbol{m/2} \rceil - \mathbf{1}$, return.
4. Else, if possible, merge $\boldsymbol{N}$ with either right or left sibling (includes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

EXAMPLE – REDUNDANT B-TREE, INSERT

- Insert values:15,9,23,25,19,40,17,21
- m = 3



EXAMPLE – REDUNDANT B-TREE, DELETE



B-Tree – Complexity / Capacity

|page| = 8 KiB
|key| = 10 B
|node pointer| = 8 B
|data pointer| = 9 B
m … arity (blocking factor)

m * |node pointer| + (m – 1) * (|key| + |data pointer|) ≤ |page|
m ≤ (8192 + 19 / 27) = 304

With ⅔ utilization, 202 records per node, we got:

upper limit on number of reads required to search the index

Tree height	# Records
0	202
1	40.804
2	8.242.408
3	1.664.996.416