King Fahd University of Petroleum & Minerals

*College of Computer Science & Engineering*

Information & Computer Science Department

# Unit 9

# Multi-Way Trees

## Reading Assignment

- “Data Structures and Algorithms in Java”, 3rd Edition, Adam Drozdek, Cengage Learning, ISBN 978-9814239233
  - Chapter 7 Section 7.1 (7.1.1 and 7.1.3 only)

## Objectives

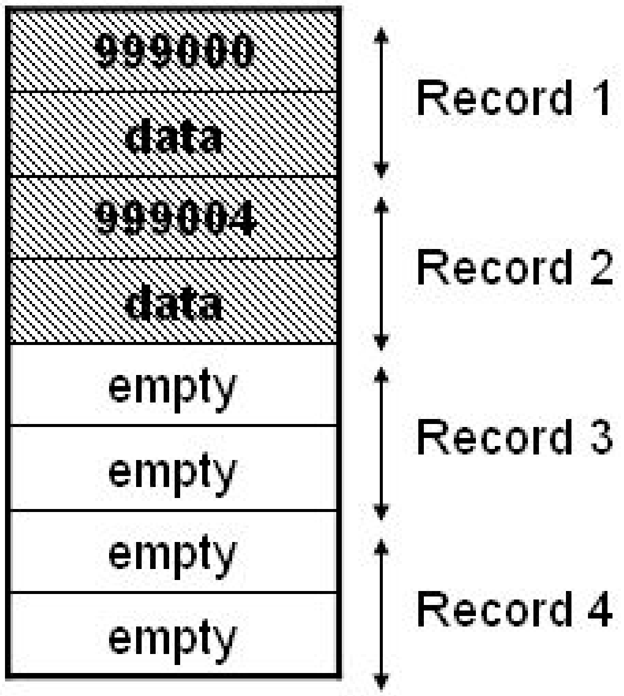
Discuss the following topics:

- Multi-Way Trees
- B-Trees
- B+-Trees

NOTE: SOME EXAMPLES IN THIS UNIT ARE ADOPTED FROM INTERNET SOURCES

## Motivation for studying Multi-Way Trees

- Data is stored on disk (i.e., secondary memory) in blocks.
- A block is the smallest amount of data that can be accessed on a disk.
- Each block has a fixed number of bytes – typically 512, 1024, 2048, 4096 or 8192 bytes
- Each block may hold many data records.

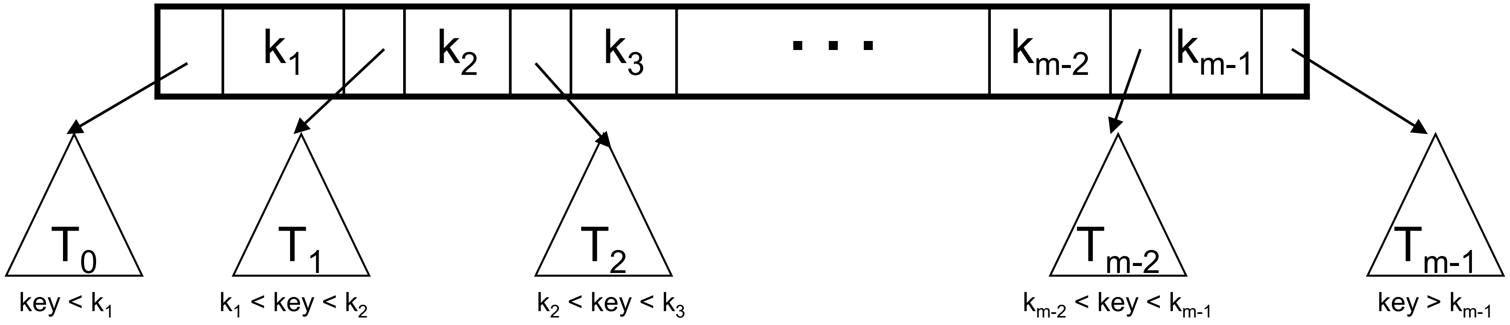


## Motivation for studying Multi-Way Trees

- A disk access is very expensive compared to a typical computer instruction (mechanical limitations) - One disk access is worth about 200,000 instructions.
- Thus, When data is too large to fit in main memory the number of disk accesses becomes important.
- Many algorithms and data structures that are efficient for manipulating data in primary memory are not efficient for manipulating large data in secondary memory because they do not minimize the number of disk accesses.
- For example, AVL trees are not suitable for representing huge tables residing in secondary memory.
- The height of an AVL tree increases, and hence the number of disk accesses required to access a particular record increases, as the number of records increases.

## What is a Multi-way tree?

- A multi-way (or m-way) search tree of order m is a tree in which
  - Each node has at-most **m** subtrees, where the subtrees **<u>may be empty</u>**.
  - Each node consists of at least **1** and at most **m-1** distinct keys
  - The keys in each node are sorted.



- The keys and subtrees of a non-leaf node are ordered as:

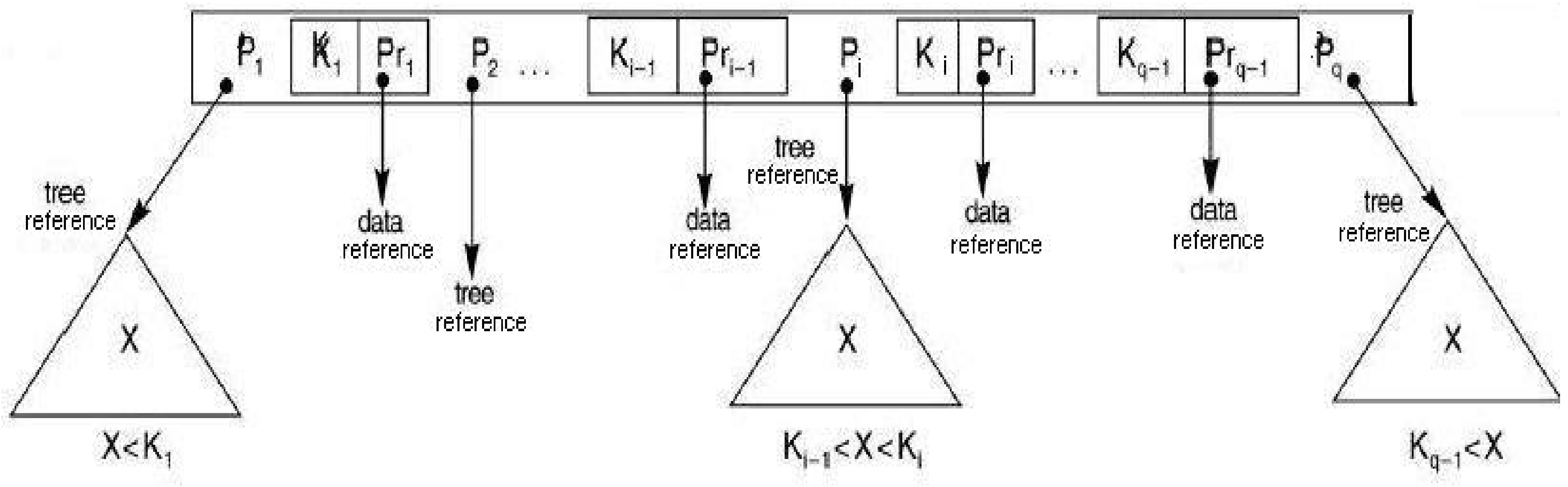
  $T_0, k_1, T_1, k_2, T_2, \ldots, k_{m-1}, T_{m-1}$ such that:

  – All keys in subtree T0 are less than k1.

  – All keys in subtree $T_i$ , $1 <= i <= m - 2$, are greater than $k_i$ but less than $k_{i+1}$.

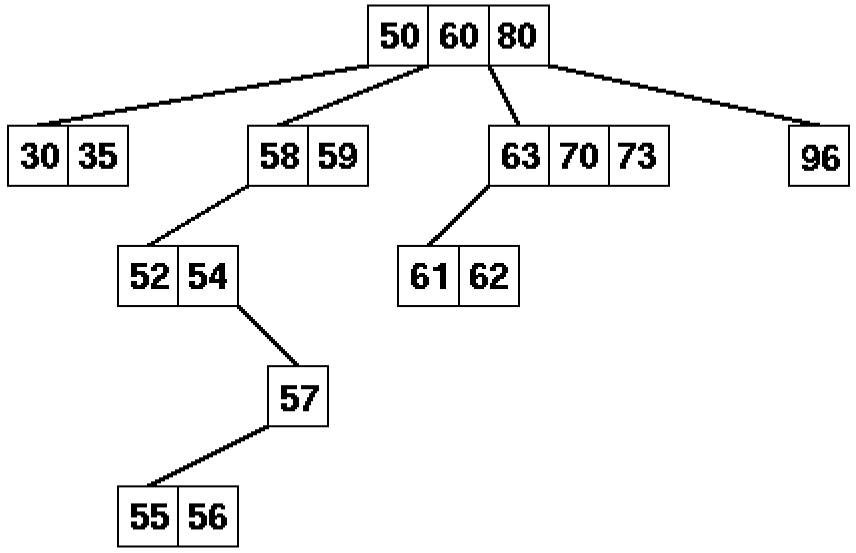
  – All keys in subtree $T_{m-1}$ are greater than $k_{m-1}$

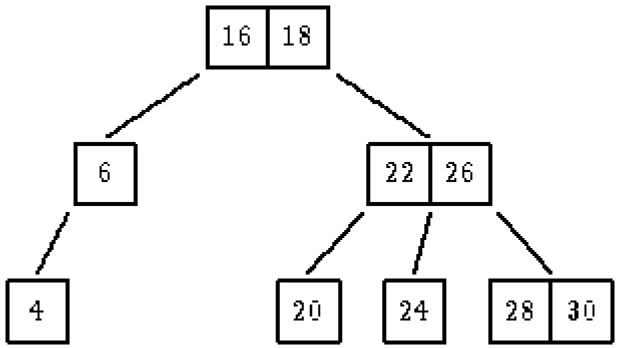
## The node structure of a Multi-way tree



- Note:
  - Corresponding to each key there is a data reference that refers to the data record for that key in secondary memory.
  - In our representations we will omit the data references.
  - The literature contains other node representations that we will not discuss.

## Examples of Multi-way Trees





- Note: In a multiway tree:
  - The leaf nodes need not be at the same level.
  - A non-leaf node with **n** keys may contain less than **n + 1** non-empty subtrees.

## B-Trees

- What is a B-tree?
- Why B-trees?
- Searching a B-tree
- Insertion in a B-tree
- Deletion in a B-tree

## What is a B-Tree?

- A B-tree of order m (or branching factor m), where m > 2, is either an empty tree or a multiway search tree with the following properties:
  - The root is either a leaf or it has at least two non-empty subtrees and at most m non-empty subtrees.
  - Each non-leaf node, other than the root, has at least $\lceil m/2 \rceil$ non-empty subtrees and at most m non-empty subtrees. (Note: $\lceil x \rceil$ is the lowest integer >= x ).
  - The number of keys in each non-leaf node is one less than the number of non-empty subtrees for that node.
  - All leaf nodes are at the same level; that is the tree is perfectly balanced.

## What is a B-tree? (cont'd)

**For a non-empty B-tree of order m:**

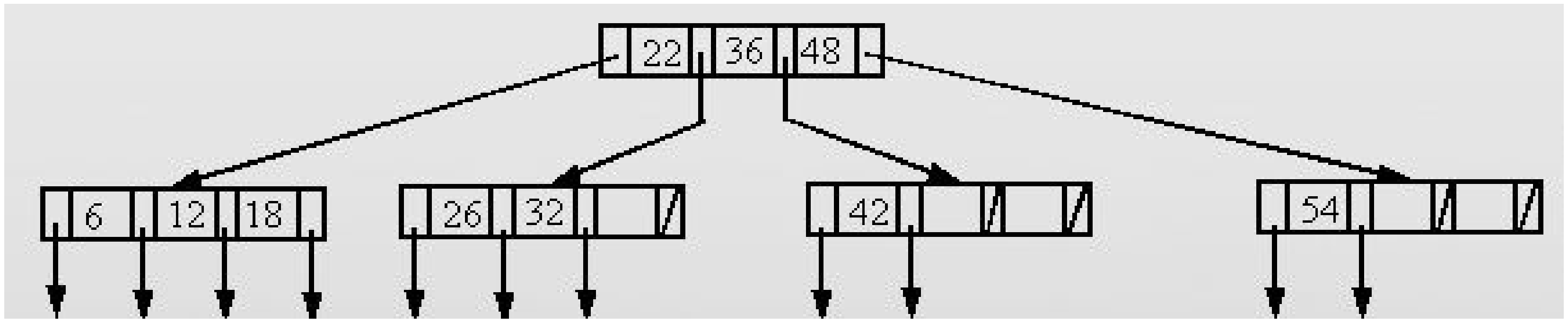
This may be zero, if the node is a leaf as well

| | Root node | Non-root node |
|---|---|---|
| **Minimum number of keys** | **1** | **$\lceil m / 2 \rceil - 1$** |
| Minimum number of non-empty subtrees | 2 | $\lceil m / 2 \rceil$ |
| **Maximum number of keys** | **m - 1** | **m – 1** |
| Maximum number of non-empty subtrees | m | m |

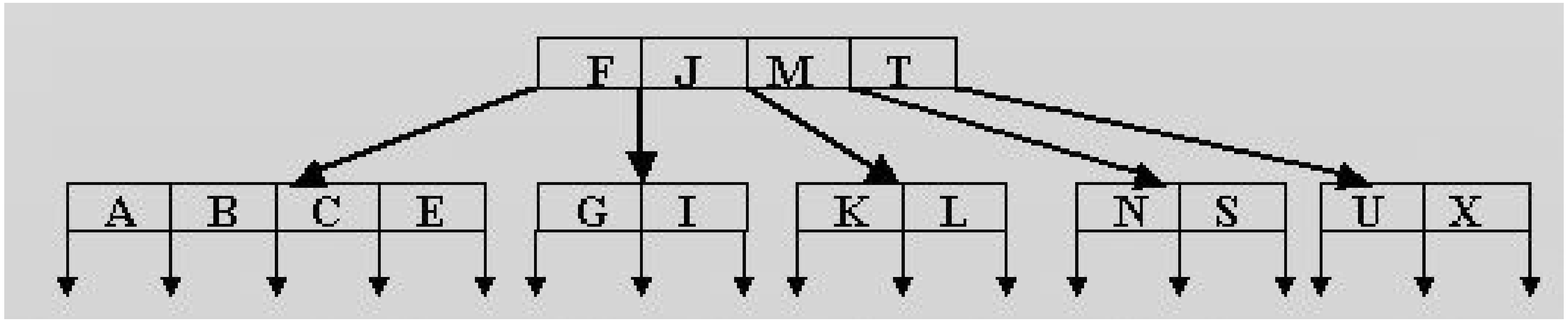
These will be zero if the node is a leaf as well

## B-Tree Examples

**Example: A B-tree of order 4**



**Example: A B-tree of order 5**



**Note:**

- The data references are not shown.
- The leaf references are to empty subtrees
- The representation of example2 is a simplification in which unused key positions are not shown

## More on Why B-Trees

- B-trees are suitable for representing huge tables residing in secondary memory because:

  1. With a large branching factor m, the height of a B-tree is low resulting in fewer disk accesses.

     **Note: As** m **increases the amount of computation at each node increases; however this cost is negligible compared to hard-drive accesses.**

  2. The branching factor can be chosen such that a node corresponds to a block of secondary memory.

  3. The most common data structure used for database indices is the B-tree. An **index** is any data structure that takes as input a property (e.g. a value for a specific field), called the search key, and *quickly* finds all records with that property.

# Comparing B-Trees with AVL Trees

- The height h of a B-tree of order m, with a total of n keys, satisfies the inequality $h \le 1 + \log_{\lceil m/2 \rceil}((n + 1) / 2)$
  - If m = 300 and n = 16,000,000 then $h \approx 4$.
  - Thus, in the worst case finding a key in such a B-tree requires 3 disk accesses (assuming the root node is always in main memory ).

- The average number of comparisons for an AVL tree with n keys is log n + 0.25 where n is large.
  - If n = 16,000,000 the average number of comparisons is 24.
  - Thus, in the average case, finding a key in such an AVL tree requires 24 disk accesses.

## Searching a B-Tree

- Searching for KEY:
  - Start from the root
  - If root or an internal node is reached:
    - Search KEY among the keys in that node
      - linear search or binary search
      - If found, return the corresponding record
    - If KEY < smallest key, follow the leftmost child reference down
    - If KEY > largest key, follow the rightmost child reference down
    - If $K_i$ < KEY < $K_j$, follow the child reference between $K_i$ and $K_j$
  - If a leaf is reached:
    - Search KEY among the keys stored in that leaf
      - linear search or binary search
    - If found, return the corresponding record; otherwise report not found

# Insertion in B-Trees

- OVERFLOW CONDITION:

  A root-node or a non-root node of a B-tree of order m overflows if, after a key insertion, it contains m keys.

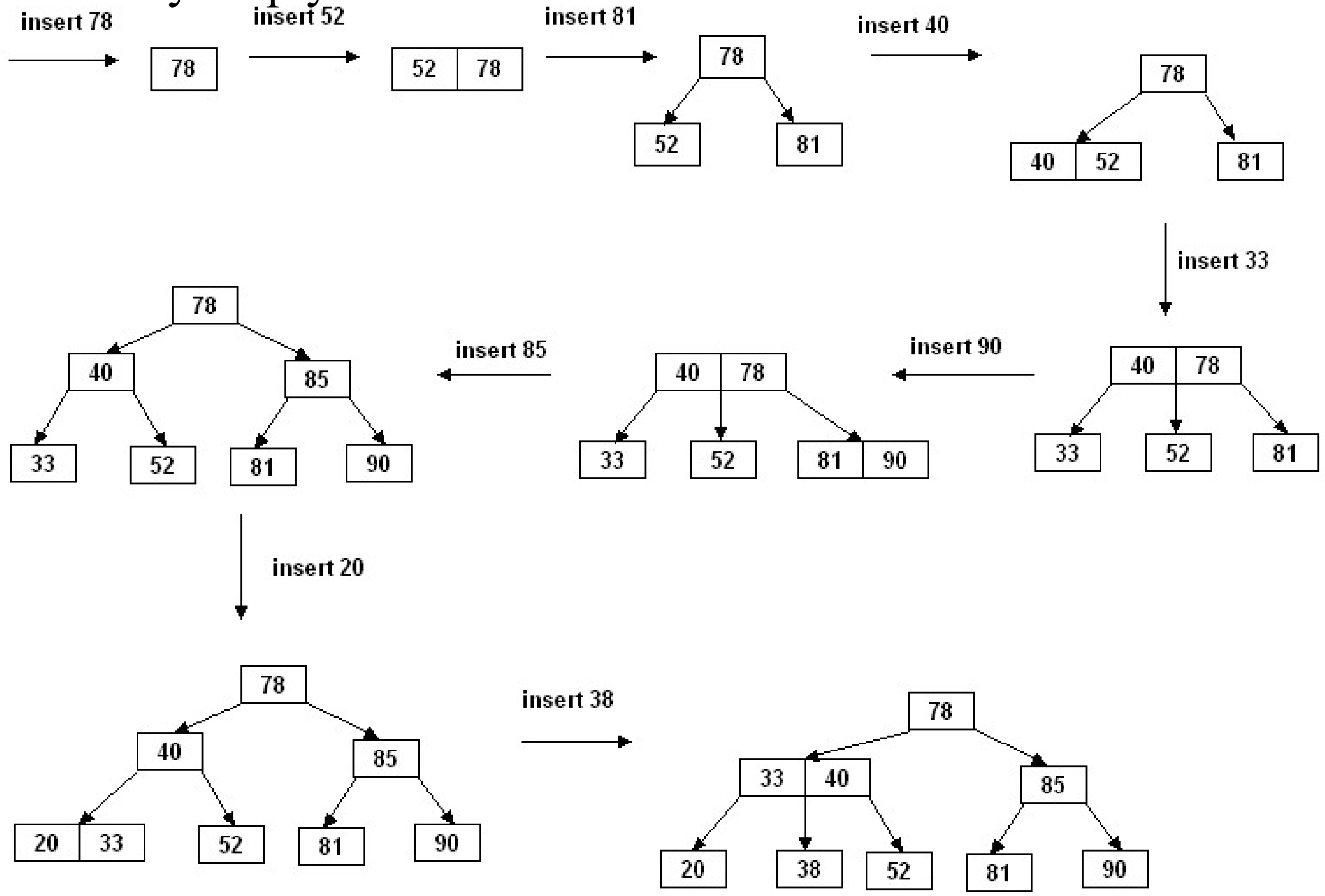
- Insertion algorithm:

  If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.

- Note: Insertion of a key always <u>starts</u> at a leaf node.

# Insertion in B-Trees

- **Insertion in a B-tree of <u>odd</u> order**
- Example: Insert the keys 78, 52, 81, 40, 33, 90, 85, 20, and 38 in this order in an initially empty B-tree of order 3



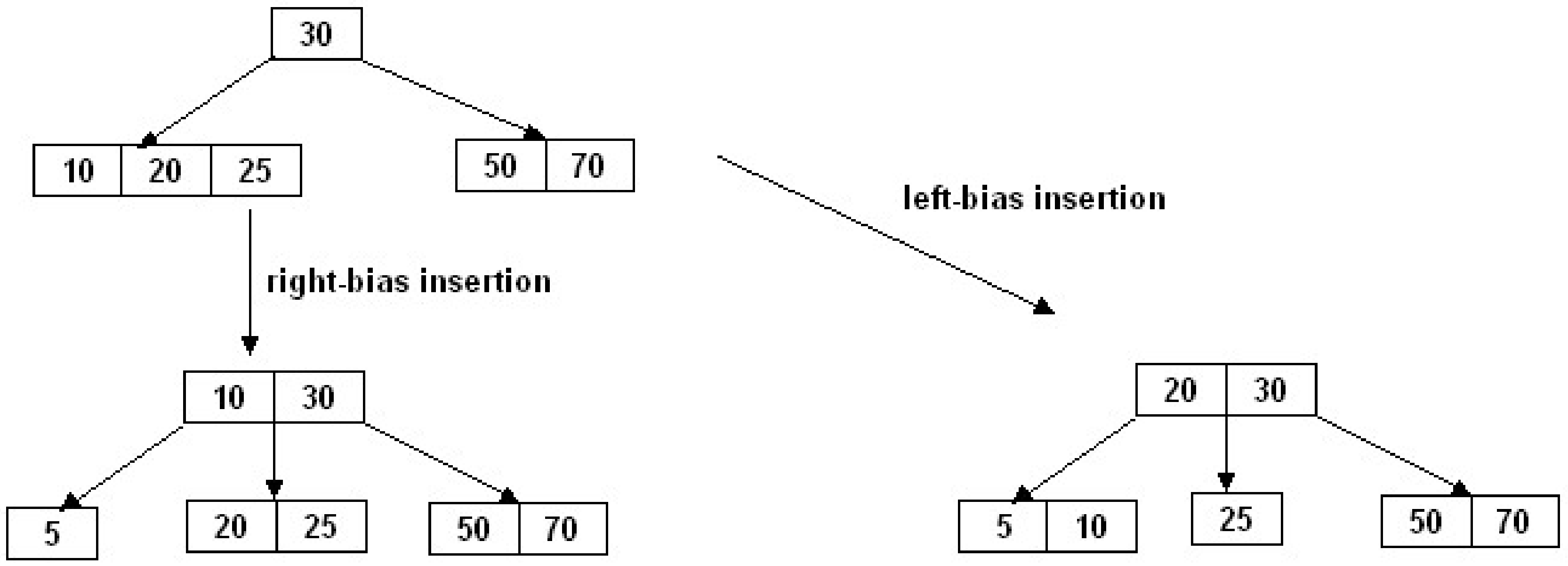
# Insertion in B-Trees

- **Insertion in a B-tree of even order**

At each node the insertion can be done in two different ways:

- **right-bias:** The node is split such that its right subtree has more keys than the left subtree.
- **left-bias:** The node is split such that its left subtree has more keys than the right subtree.

- **Example**: Insert the key **5** in the following B-tree of order **4**:



## B-Tree Insertion Algorithm

```
insertKey (x){
   if(the key x is in the tree)
       throw an appropriate exception;

   let the insertion leaf-node be the currentNode;
   insert x in its proper location within the node;

   if(the currentNode does not overflow)
     return;
    done = false;
    do{
       if (m is odd) {
           split currentNode into two siblings such that the right sibling rs has m/2 right-most keys,
           and the left sibling ls has m/2 left-most keys;
           Let w be the middle key of the splinted node;
       }
       else {          // m is even
           split currentNode into two siblings by any of the following methods:
             ■ right-bias: the right sibling rs has m/2 right-most keys, and the left sibling ls has (m-1)/2 left-most keys.
             ■ left-bias: the right sibling rs has (m-1)/2 right-most keys, and the left sibling ls  has m/2 left-most keys.
          let w be the “middle” key of the splinted node;
       }
      if (the currentNode is not the root node) {
          insert w in its proper location in the parent p of the currentNode;
         if (p does not overflow)
             done = true;
        else
            let p be the currentNode;
       }
  } while (! done && currentNode is not the root node);
```

## B-Tree Insertion Algorithm - Contd

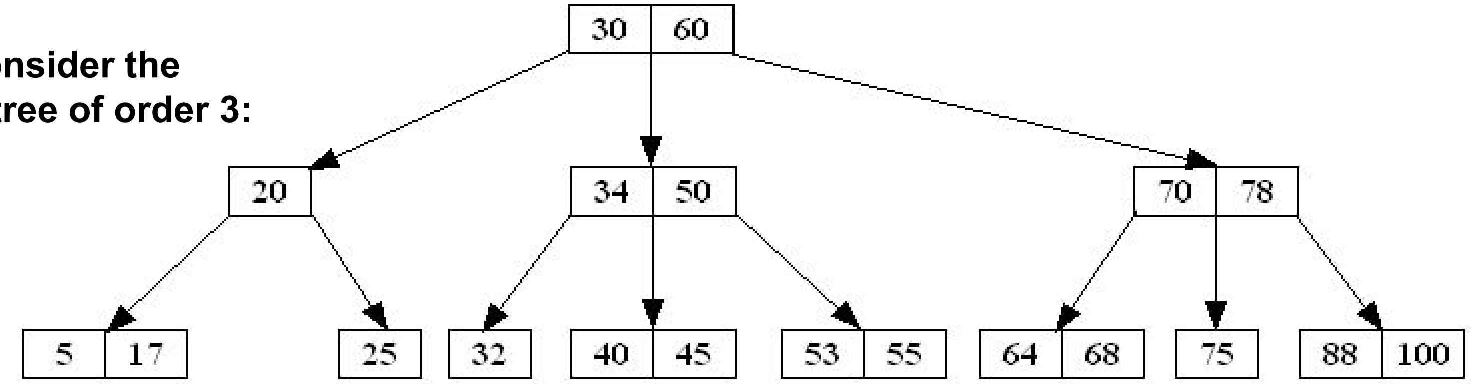
```
if (! done) {
        create a new root node with **w** as its only key;
        let the right sibling **rs** be the right child of the new root;
        let the left sibling **ls** be the left child of the new root;
  }
  return;
}
```

# Deletion in B-Tree

**Like insertion, deletion must be on a leaf node. If the key to be deleted is not in a leaf, swap it with either its successor or predecessor (each will be in a leaf).**

- The successor of a key **k** is the smallest key greater than **k**.
- The predecessor of a key **k** is the largest key smaller than **k**.
- IN A B-TREE THE SUCCESSOR AND PREDECESSOR, IF ANY, OF ANY KEY IS IN A LEAF NODE

**Example: Consider the following B-tree of order 3:**



| key | predecessor | successor |
|---|---|---|
| 20 | 17 | 25 |
| 30 | 25 | 32 |
| 34 | 32 | 40 |
| 50 | 45 | 53 |
| 60 | 55 | 64 |
| 70 | 68 | 75 |
| 78 | 75 | 88 |

## Deletion in B-Tree

- **UNDERFLOW CONDITION**
- A non-root node of a B-tree of order m underflows if, after a key deletion, it contains $\lceil \mathbf{m / 2} \rceil$ **- 2** keys

- The root node does not underflow. If it contains only one key and this key is deleted, the tree becomes empty.

## Deletion in B-Tree

- **Deletion algorithm:**

  If a node underflows, **rotate** the appropriate key from the **adjacent** right- or left-sibling if the sibling contains at least $\lceil m / 2 \rceil$ keys; otherwise perform a **merging**.

⇒ A key rotation must always be attempted before a merging

- There are **five** deletion cases:

  1. The leaf does not underflow.
  2. The leaf underflows and the adjacent right sibling has at least $\lceil m / 2 \rceil$ keys.

     perform a left key-rotation
  3. The leaf underflows and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.

     perform a right key-rotation
  4. The leaf underflows and each of the adjacent right sibling and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.

     perform either a left or a right key-rotation
  5. The leaf underflows and each adjacent sibling has $\lceil m / 2 \rceil$ - 1 keys.

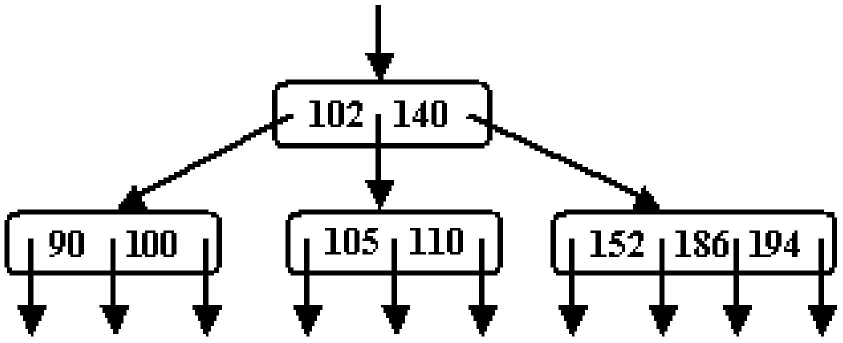
     perform a merging

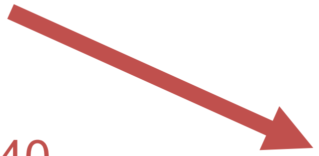
## Deletion in B-Tree

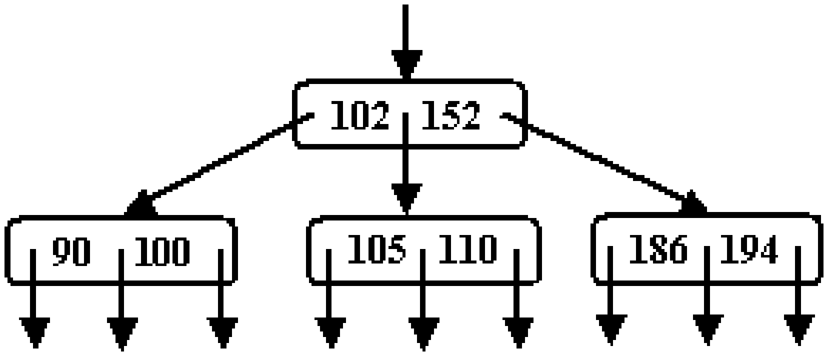
### Case1: The leaf does not underflow.

Example:

**B-tree of order 4**





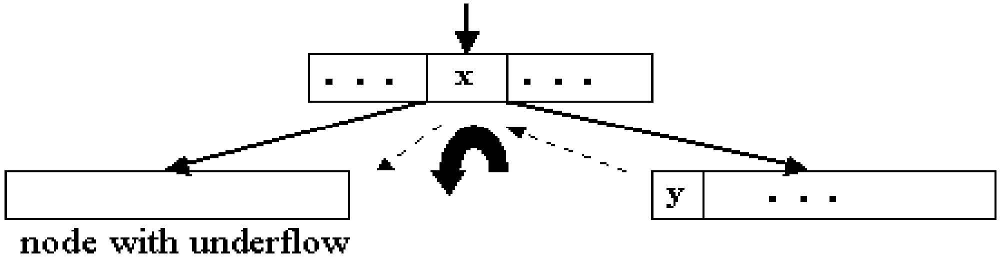


# Deletion in B-Tree (cont'd)

Case2: The leaf underflows and the adjacent right sibling has at least ⌈m / 2⌉ keys.

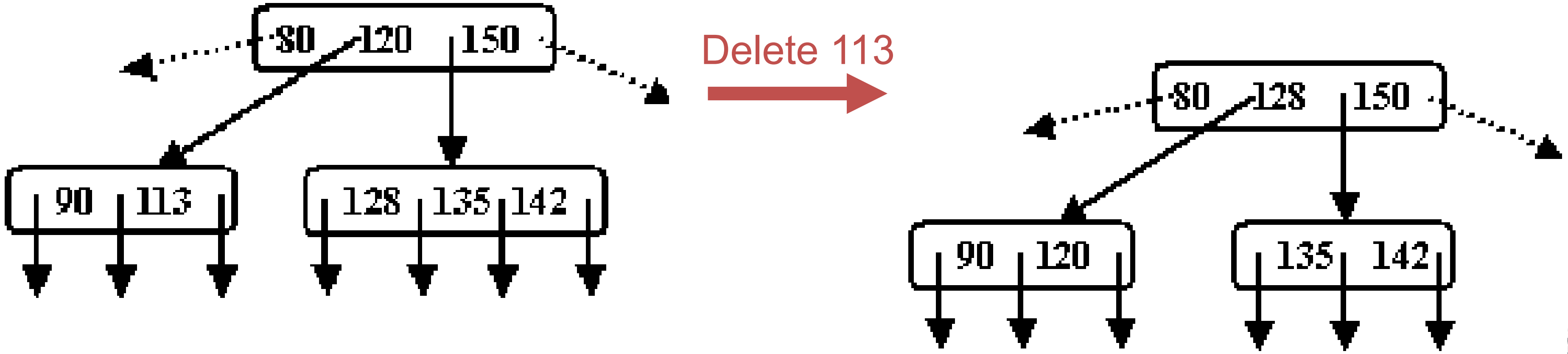
Perform a left key-rotation:

1. Move the parent key **x** that separates the siblings to the node with underflow
2. Move **y**, the minimum key in the right sibling, to where the key **x** was
3. Make the old left subtree of **y** to be the new right subtree of **x**.



Example:

**B-tree of order 5**

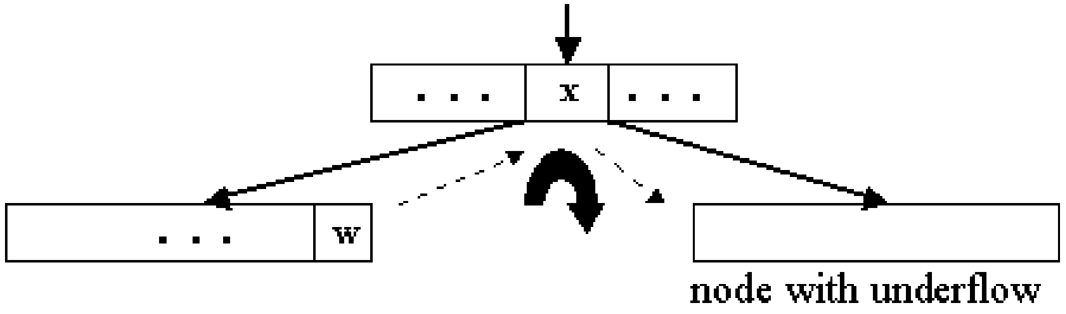


## Deletion in B-Tree (cont'd)

Case 3: The leaf underflows and the adjacent left sibling has at least $\lceil m / 2 \rceil$ keys.

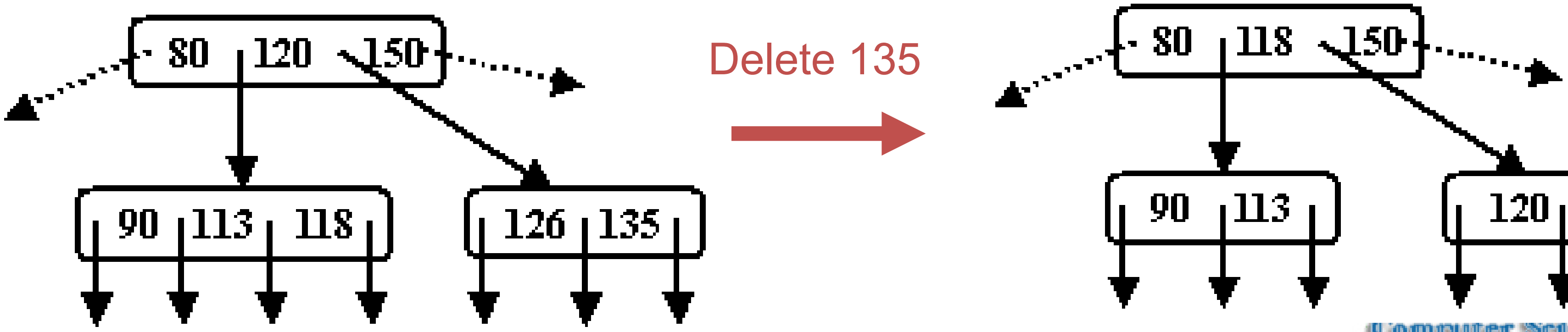
Perform a right key-rotation:

1. Move the parent key **x** that separates the siblings to the node with underflow
2. Move **w**, the maximum key in the left sibling, to where the key **x** was
3. Make the old right subtree of **w** to be the new left subtree of **x**



Example:

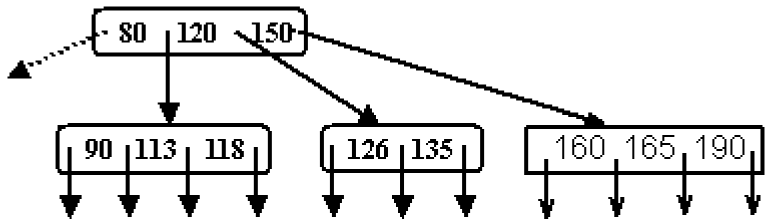
**B-tree of order 5**

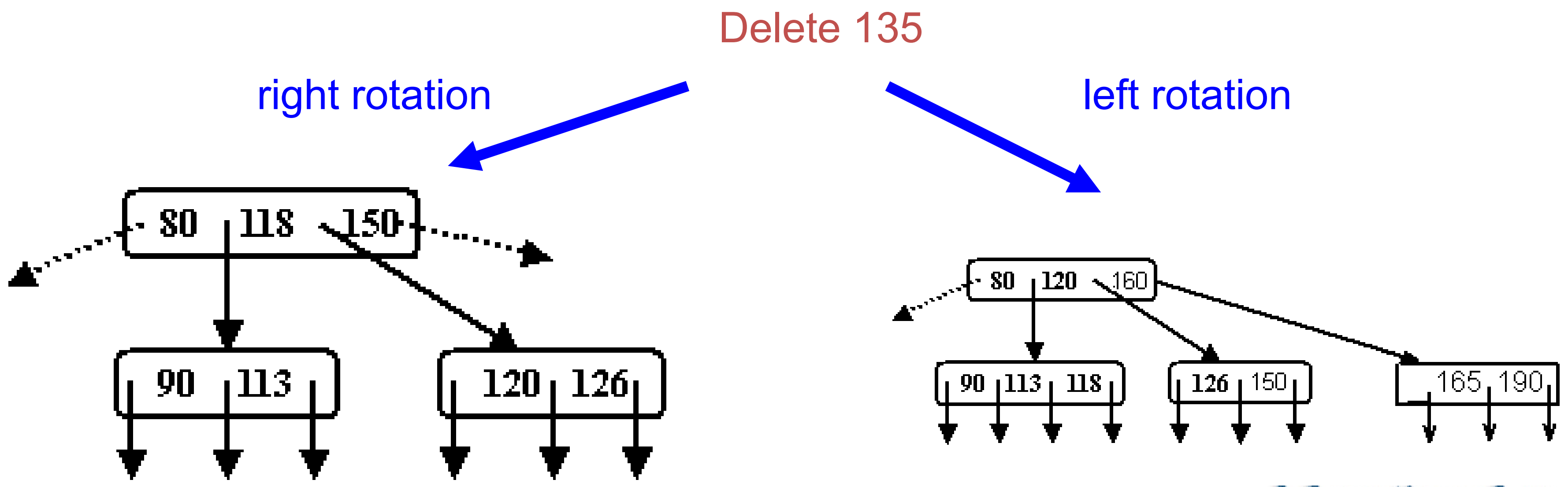


## Deletion in B-Trees (cont'd)

Case 4: The leaf underflows and each of the adjacent right sibling and the adjacent left sibling has at least ⌈m / 2⌉ keys.

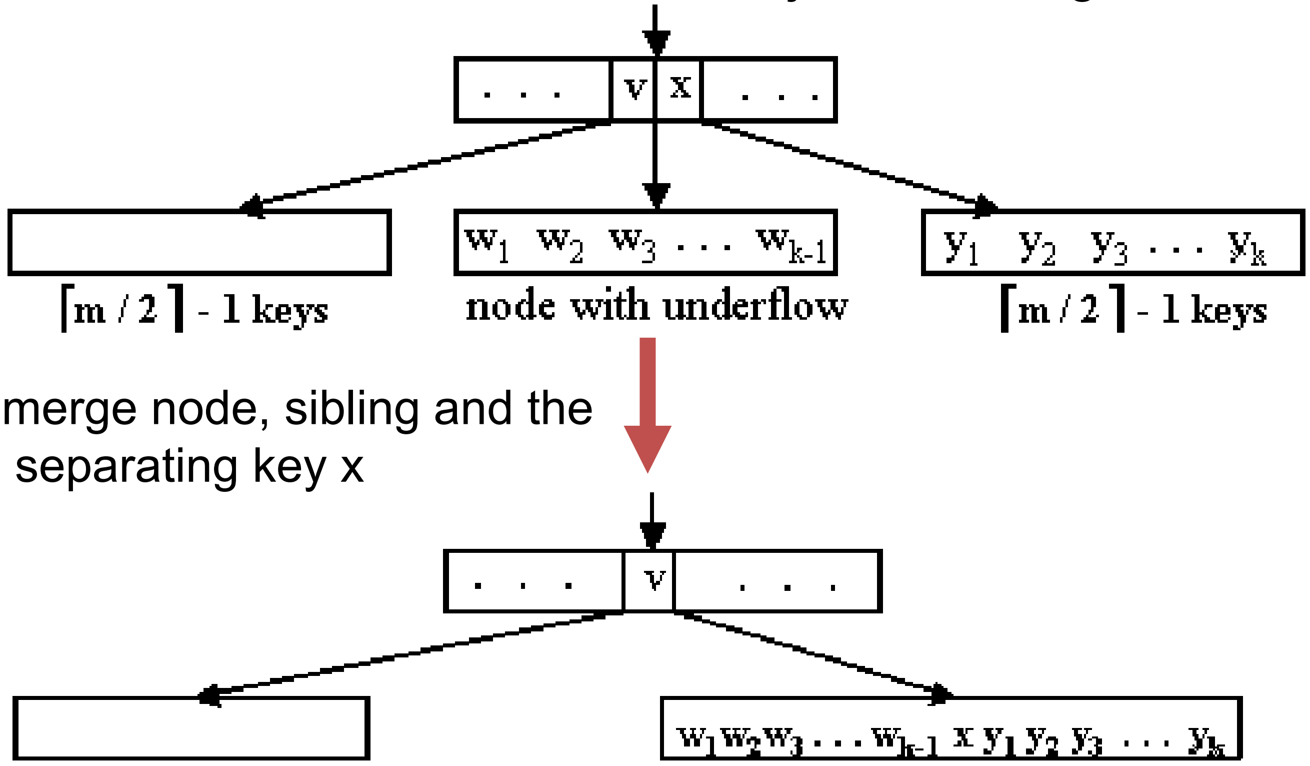
Example: **B-tree of order 5**





## Deletion in B-Tree (cont'd)

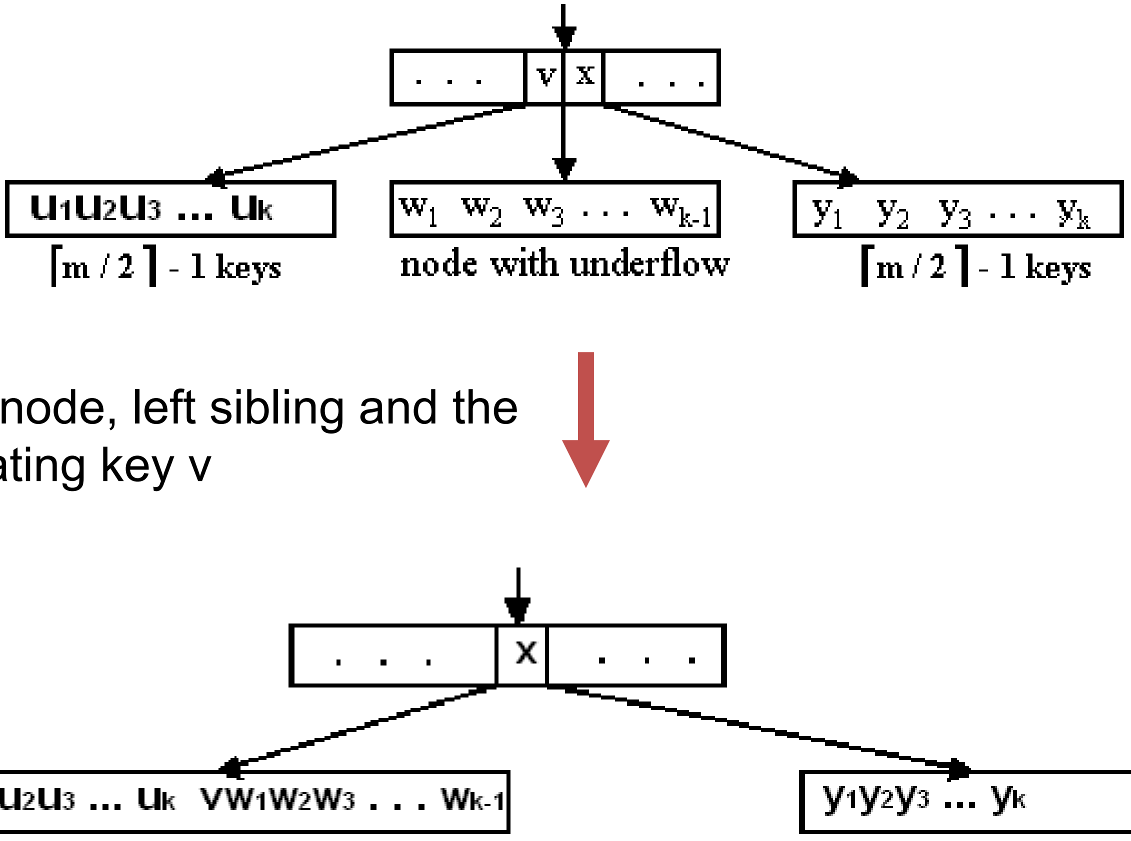
Case 5:The leaf underflows and each adjacent sibling has $\lceil m / 2 \rceil - 1$ keys.


. . . v x . . .
$w_1\ w_2\ w_3 \ldots w_{k-1}$
$y_1\ y_2\ y_3 \ldots y_k$
$\lceil m / 2 \rceil - 1$ keys
node with underflow
$\lceil m / 2 \rceil - 1$ keys
merge node, sibling and the separating key x
. . . v . . .
$w_1 w_2 w_3 \ldots w_{k-1}\ x\ y_1 y_2 y_3 \ldots y_k$

If the parent of the merged node underflows, the merging process propagates upward. In the limit, a root with one key is deleted and the height decreases by one.

## Deletion in B-Trees (cont'd)

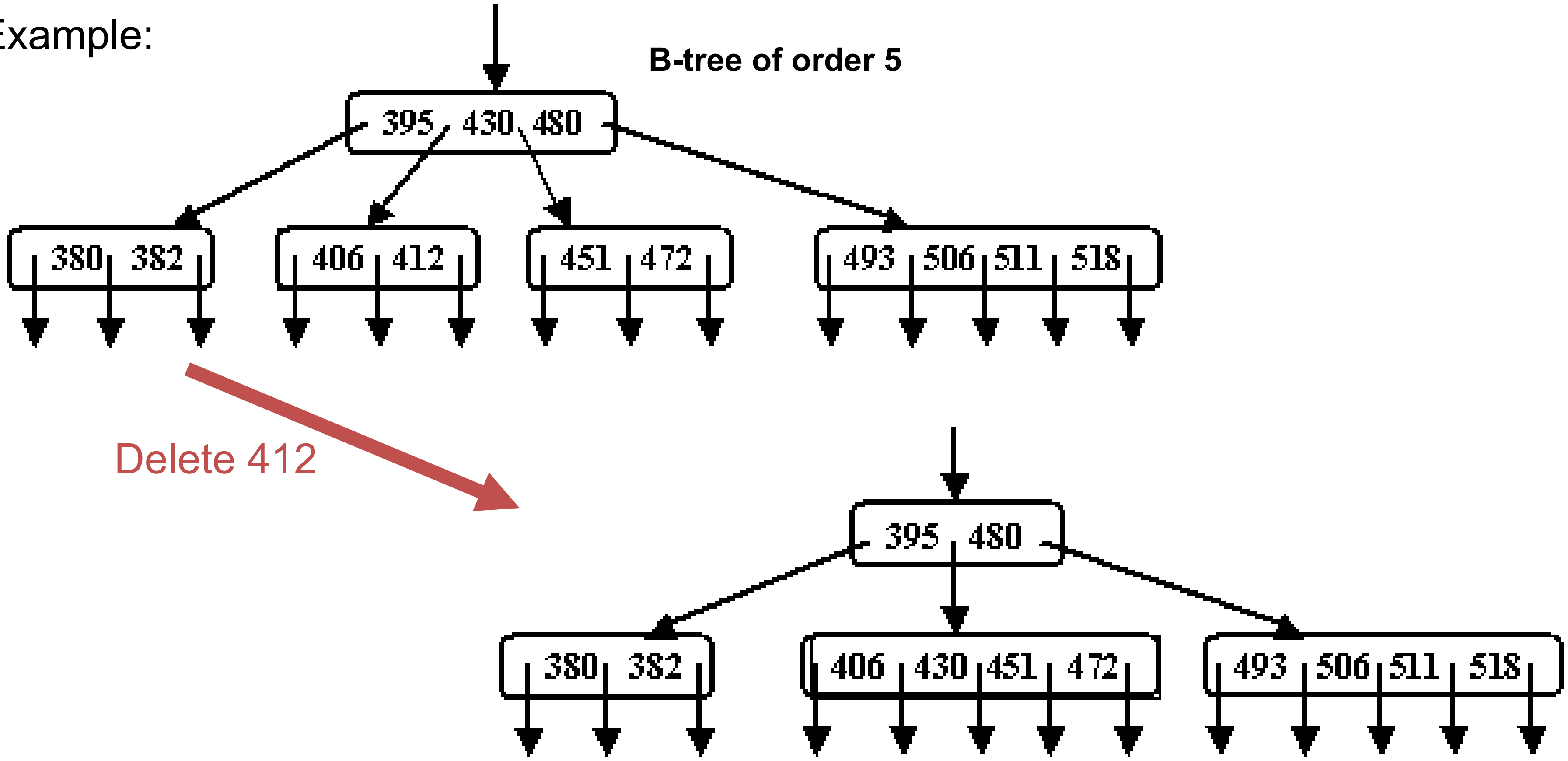
**Note**: The merging could also be done by using the left sibling instead of the right sibling.



merge node, left sibling and the separating key v

## Deletion in B-Tree (cont'd)

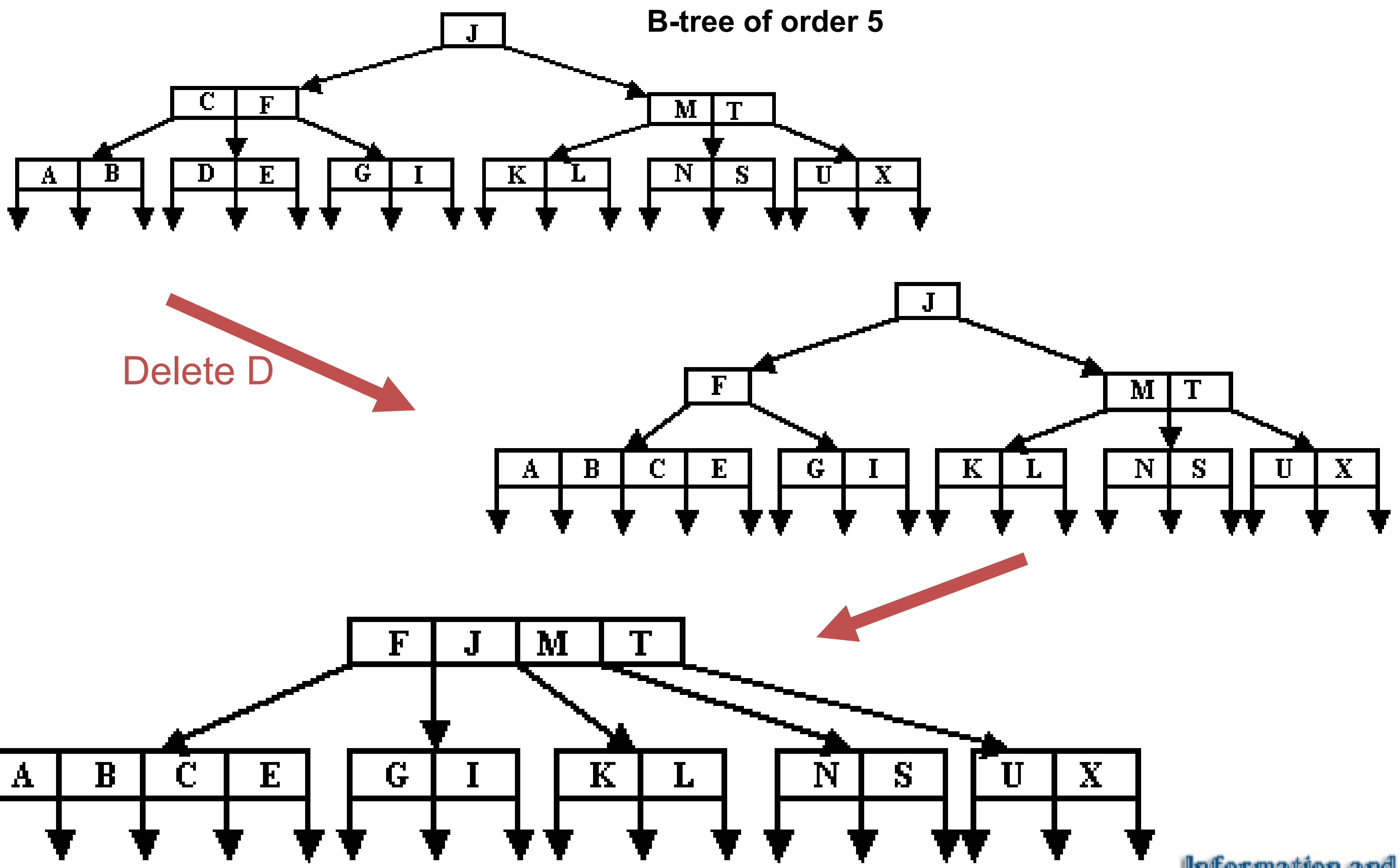
Example:



The parent of the merged node does not underflow. The merging process does not propagate upward.

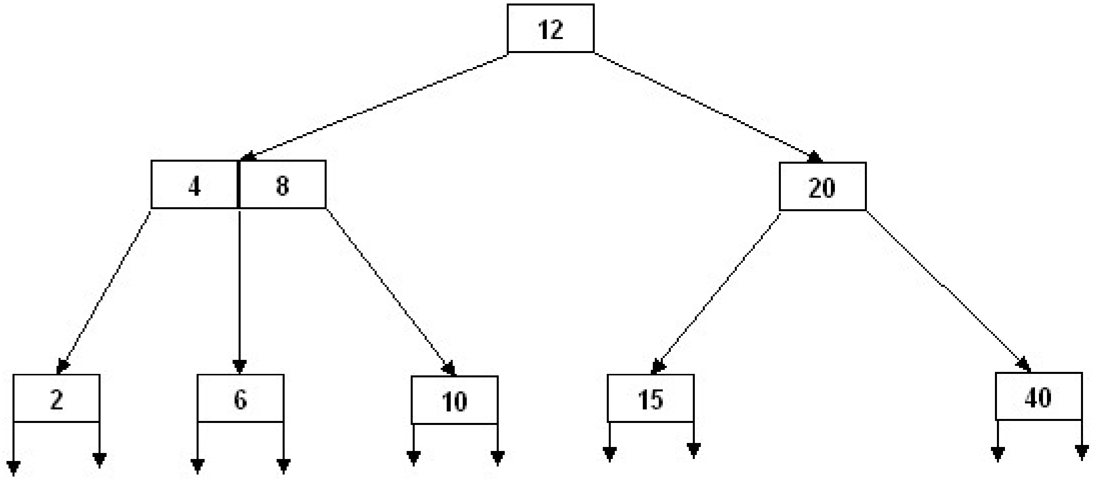
# Deletion in B-Tree (cont'd)

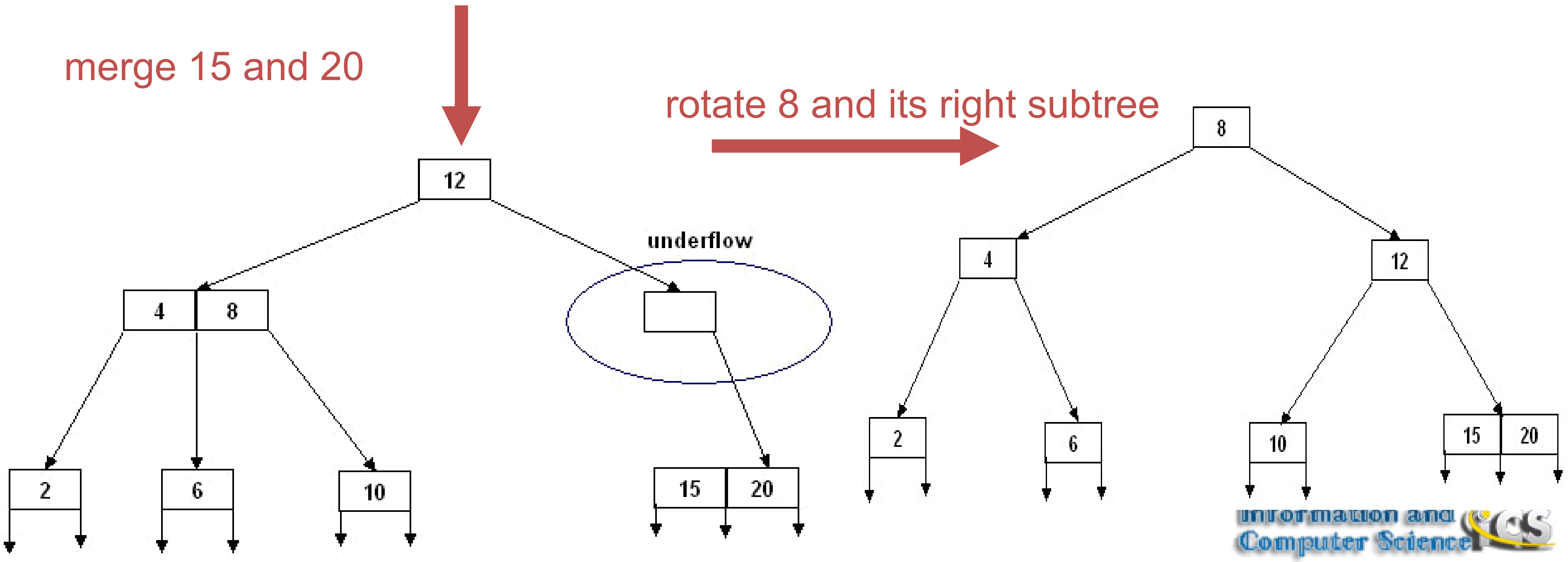
Example:



## Example involving a rotation and merging

Example: Delete the key **40** in the following B-tree of order **3**:





## B-Tree Deletion Algorithm

```
deleteKey (x) {
    if (the key x to be deleted is not in the tree)
         throw an appropriate exception;
    if (the tree has only one node) {
      delete x ;
      return;
    }
    if (the key x is not in a leaf node)
           swap x with its successor or predecessor;           // each will be in a leaf node
    delete x from the leaf node;
    if(the leaf node does not underflow) // after deletion numKeys ≥ ⌈m / 2⌉ - 1
         return;
     let the leaf node be the CurrentNode;
     done = false;
```

## B-Tree Deletion Algorithm

```
while (! done && numKeys(CurrentNode) < ⌈m / 2⌉ - 1) {        // there is underflow
    if (any of the adjacent siblings t of the CurrentNode has at least ⌈m / 2⌉ keys) { // ROTATION CASE
        if (t is the adjacent right sibling) {
            rotate the separating-parent key w of CurrentNode and t to CurrentNode;
            rotate the minimum key of t to the previous parent-location of w;
            rotate the left subtree of t, if any, to become the right-most subtree of CurrentNode;
        }
        else {      // t is the adjacent left sibling
            rotate the separating-parent key w between CurrentNode and t to CurrentNode;
            rotate the maximum key of t to the previous parent-location of w;
            rotate the right subtree of t , if any, to become the left-most subtree of CurrentNode;
        }
        done = true;
    }
    else { // MERGING CASE: the adjacent or each adjacent sibling has ⌈m / 2⌉ - 1 keys
        select any adjacent sibling t of CurrentNode;
        create a new sibling by merging currentNode, the sibling t, and their parent-separating key ;
        If (parent node p is the root node) {
            if (p is empty after the merging)
                make the merged node the new root;
            done = true;
        } else
      let parent p be the CurrentNode;
      }
  } // while
  return;
}
```

## B+-Trees

- What is a B+ tree?
- Why B+ trees?
- Searching a B+ tree
- Insertion in a B+ tree
- Deletion in a B+ tree.

# What is a B+ tree?

- A $B^+$-tree of order $M \geq 3$ is an M-ary tree with the following properties:
  - Leaves contain data items or references to data items
    - all are at the same depth
    - each leaf has $\lceil L/2 \rceil$ to L data or data references (**L** may be equal to, less or greater than **M**; but usually $L \ll M$)
  - Internal nodes contain searching keys
    - The keys in each node are sorted in increasing order
    - each node has at least $\lceil M/2 \rceil$ and at most M subtrees
    - The number of search keys in each node is one less than the number of subtrees
      - key i in an internal node is the smallest key in subtree i+1
  - Root
    - can be a single leaf, or has 2 to M children

- Nodes are at least half-full, so that the tree will not degenerate into a simple binary tree or even a linked list

# The internal node structure of a B+ tree

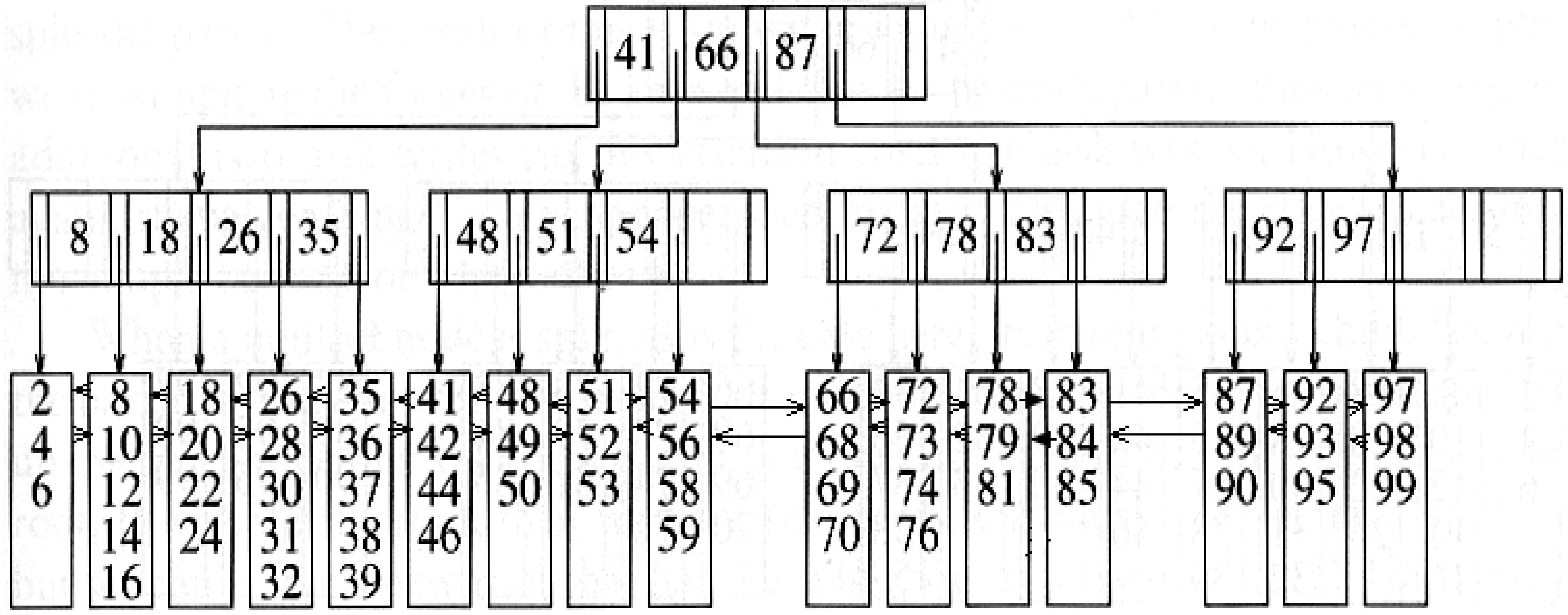
- Each leaf node stores key-data pair or key-dataReference pair. Data or data references are in leaves only.
- Leaves form a doubly-linked list that is sorted in increasing order of keys.
- Each internal node has the following structure:

$$j \quad a_1 \; k_1 \; a_2 \; k_2 \; a_3 \; \ldots \; k_j \; a_{j+1}$$

j is the number of keys in the node.
$a_i$ is a reference to a subtree.
$k_i$ is <= the smallest key in subtree $a_{i+1}$ and is > largest key in subtree $a_i$.
$k_1 < k_2 < k_3 < . . . < k_j$

# What is a B+ tree?

- Example: A B+ tree of order M = 5, L = 5



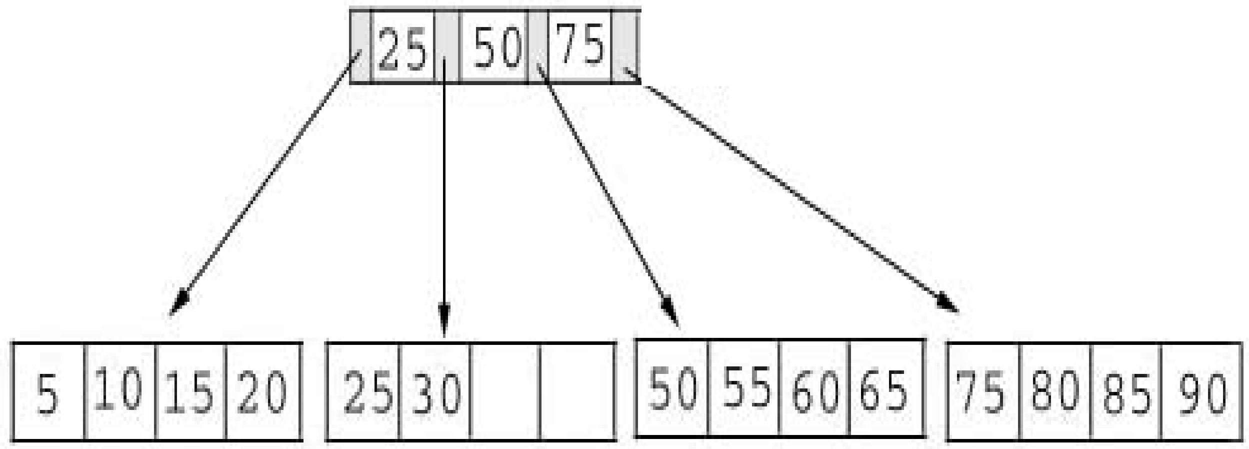
• Records or references to records are stored at the leaves, but we only show the keys here

• At the internal nodes, only keys (and references to subtrees) are stored

• **Note: The index set (i.e., internal nodes) contains distinct keys**

## What is a B+ tree?

- Example: A B+ tree of order M = 4, L = 4



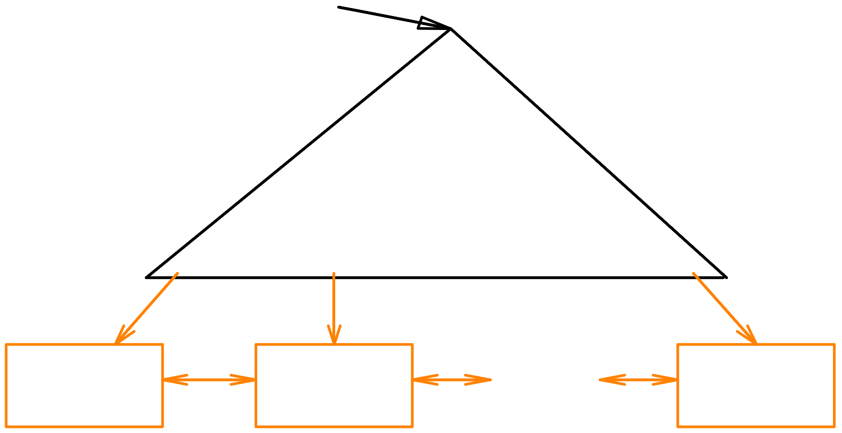
Note: For simplicity the doubly linked list references that join leaf nodes are omitted

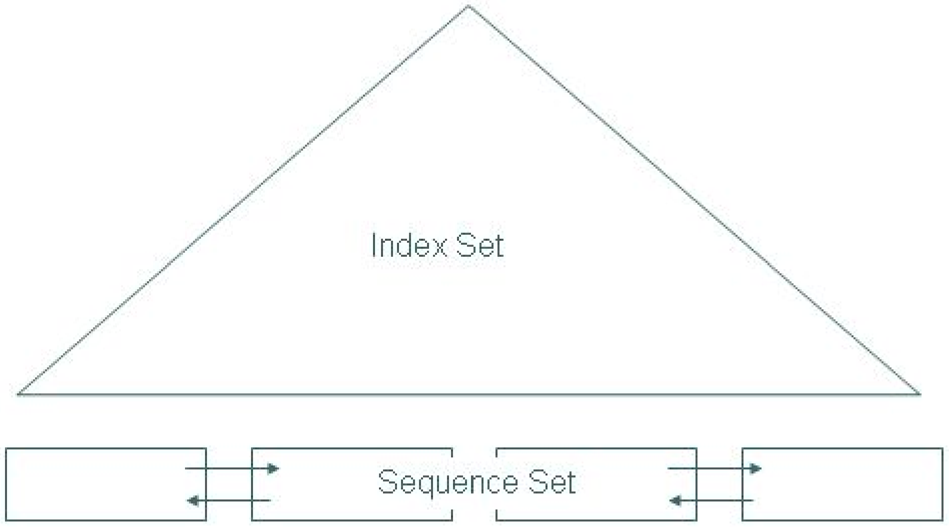
## Why B+ trees?

- Like a B-tree each internal node and leaf node is designed to fit into one I/O block of data. An I/O block usually can hold quite a lot of data. Hence, an internal node can keep a lot of keys, i.e., large M. This implies that the tree has only a few levels and only a few disk accesses can accomplish a search, insertion, or deletion.
- $B^+$-tree is a popular structure used in commercial databases. To further speed up searches, insertions, and deletions, the first one or two levels of the $B^+$-tree are usually kept in main memory.
- The reason that B+ trees are used in databases is, unlike B-trees, B+ trees support both equality and range-searches efficiently:
  - Example of equality search: Find a student record with key 950000
  - Example of range search: Find all student records with Exam grade greater than 70 and less than 90

# Why B+ trees ? (Cont'd)

A B+ tree supports equality and range-searches efficiently





## B+ Trees in Practice

- For a B+ tree of order M and L = M, with *h* levels of index, where h $\geq$ 1:
  - The maximum number of records stored is $n = M^{h-1}(L)$
    - The data records are in level *h*, where each leaf node holds L records.
  - The space required to store the tree is $O(n)$
  - Inserting a record requires $O(\log_M n)$ operations in the worst case
  - Finding a record requires $O(\log_M n)$ operations in the worst case
  - Removing a (previously located) record requires $O(\log_M n)$ operations in the worst case
  - Performing a range query with *k* elements occurring within the range requires $O(\log_M n + k)$ operations in the worst case.

- Example for a B+ tree of order M = 133 and L = 100:
  - A tree with 3 levels stores a maximum of $133^2$ (100) = 1,768,900 records
  - A tree with 4 levels stores a maximum of: $133^3$(100) = 235,263,700 records

## Searching a B+ Trees

- Searching KEY:
  - Start from the root
  - If an internal node is reached:
    - Search KEY among the keys in that node
      - linear search or binary search
    - If KEY < smallest key, follow the leftmost child reference down
    - If KEY >= largest key, follow the rightmost child reference down
    - If $K_i$ <= KEY < $K_j$, follow the child reference between $K_i$ and $K_j$
  - If a leaf is reached:
    - Search KEY among the keys stored in that leaf
      - linear search or binary search
    - If found, return the corresponding record; otherwise report not found

## Searching a B+ Trees

- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are *K* search-key values in the file, the path is no longer than
  $\lceil \log_{\lceil m/2 \rceil}(K) \rceil$.
- With 1 million search key values and *m* = 100, at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.

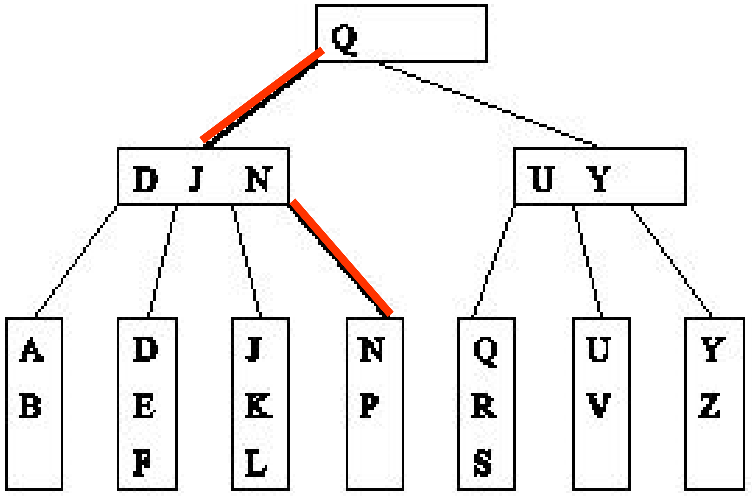
# Insertion in B+ Trees

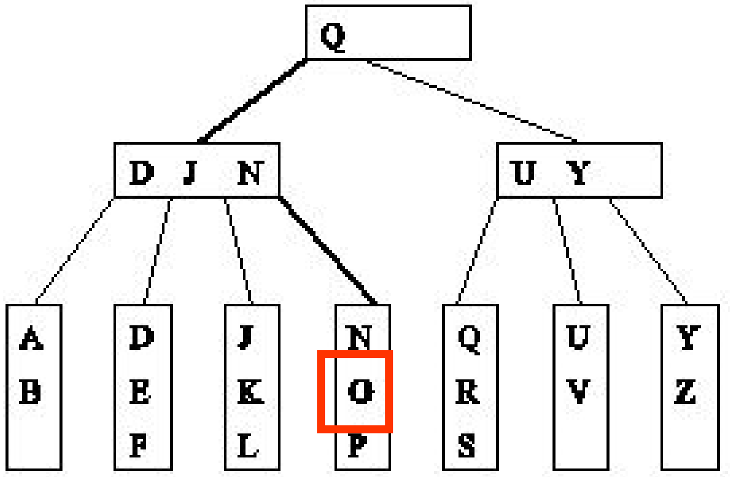
Suppose that we want to insert a key ***K*** and its associated record into the B+ tree.

- A B+ tree has two OVERFLOW CONDITIONS:
  - A leaf-node overflows if after insertion it contains **L + 1** keys
  - A root-node or an internal node of a B+ tree of order M overflows if, after a key insertion, it contains **M** keys.
- Insertion algorithm:
  - Search for the appropriate leaf node x to insert the key.
    - Note: Insertion of a key always starts at a leaf node.
  - If the key exists in the leaf node x, report an error, else insert the key in its proper sorted order in the leaf node.
  - If the leaf does not overflow (If *x* contains less than L+1 keys after insertion), the insertion is done, else
  - If a leaf node overflows, split it into two, COPY the smallest key y of the right split node and insert it (Using B-Tree Insertion Algorithm) to the parent of the node (Records with keys **<** y go to the left leaf node. Records with keys >= y go to the right leaf node).

# Insertion in B+ Trees: No overflow

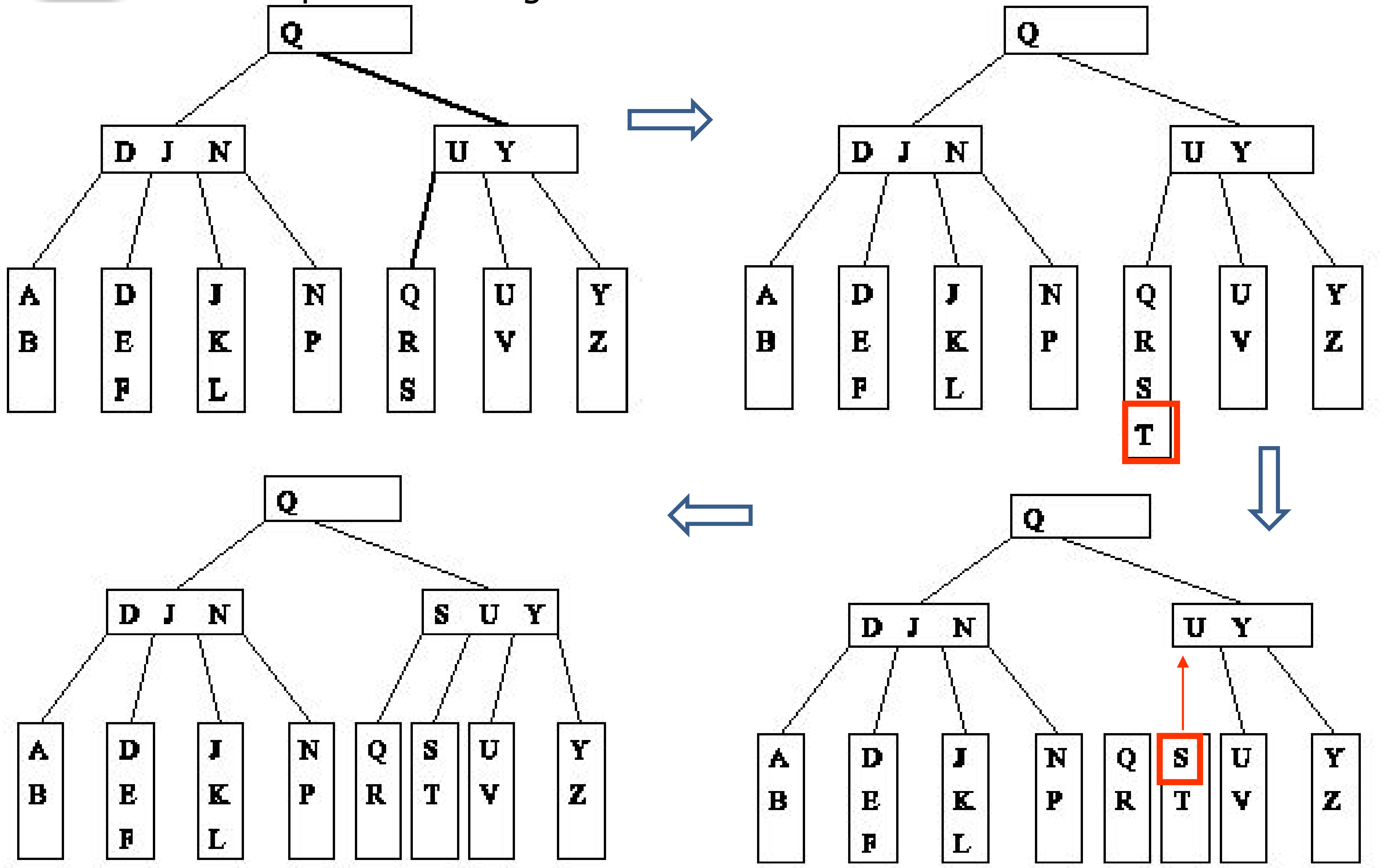
An example of inserting *O* into a B+ tree of order M = 4, L = 3.





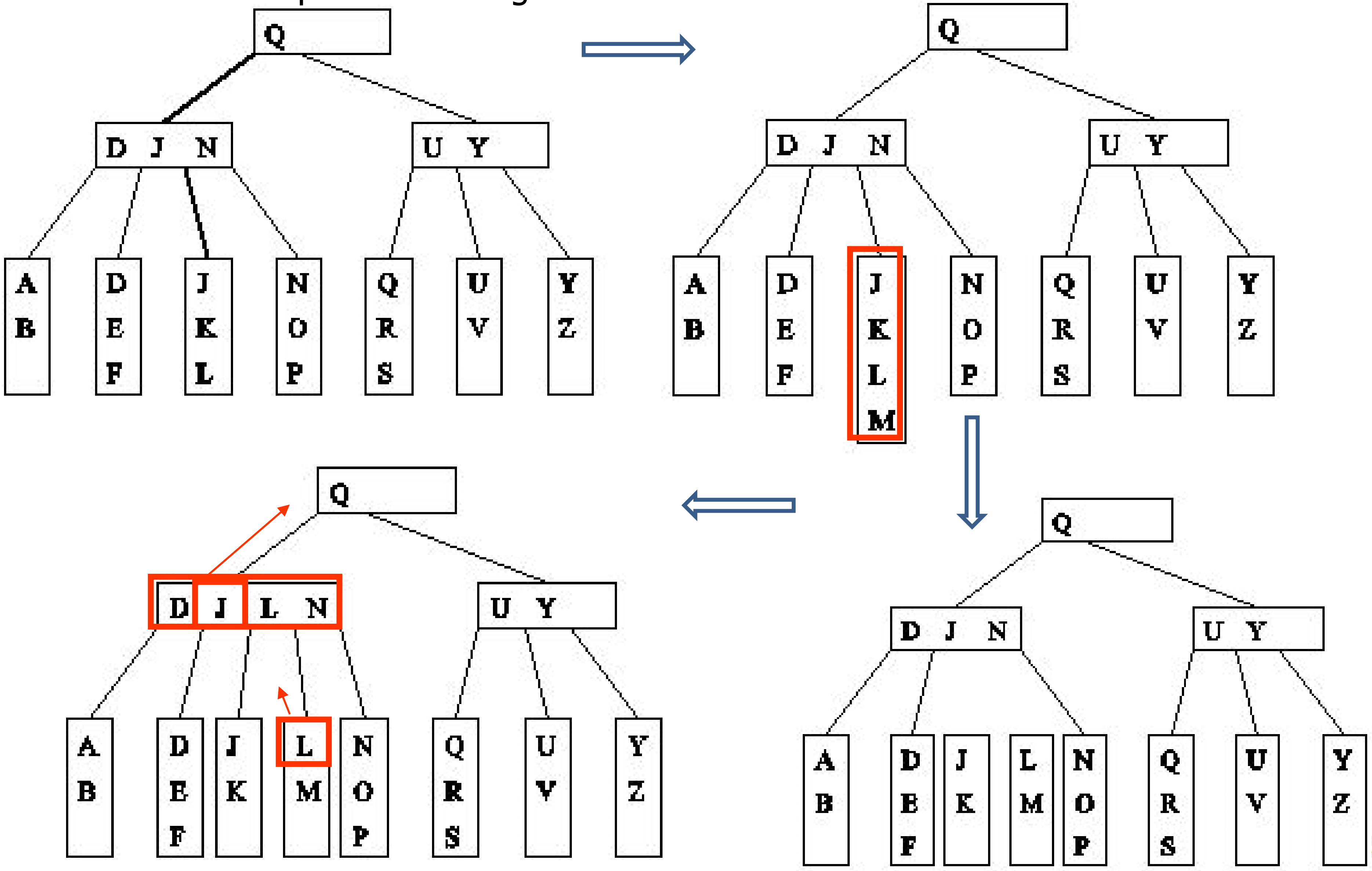
## Insertion in B+ Trees: Splitting a Leaf Node

An example of inserting *T* into a B+ tree of order M = 4 and L= 3

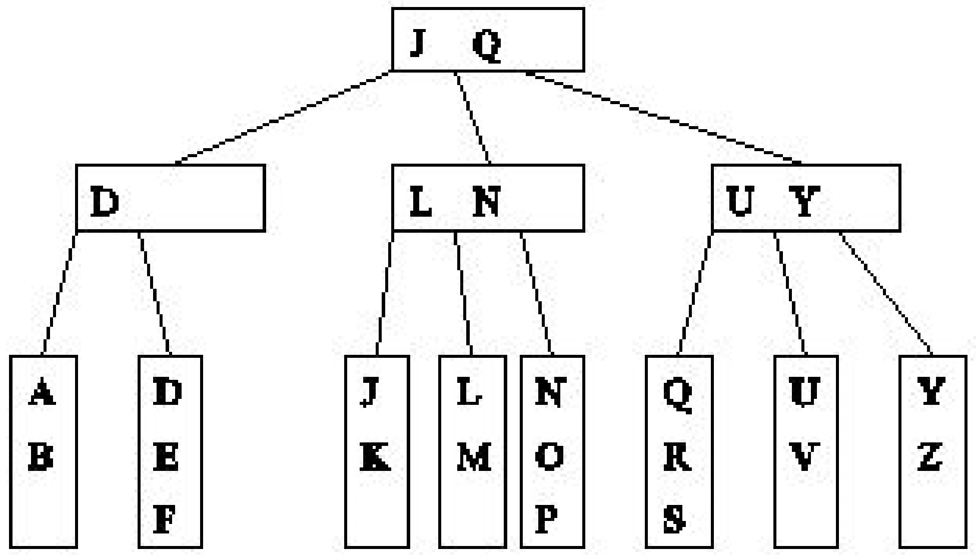


## Insertion in B+ Trees: Splitting an Internal Node

An example of inserting *M* into a B+ tree of order M= 4 and L = 3

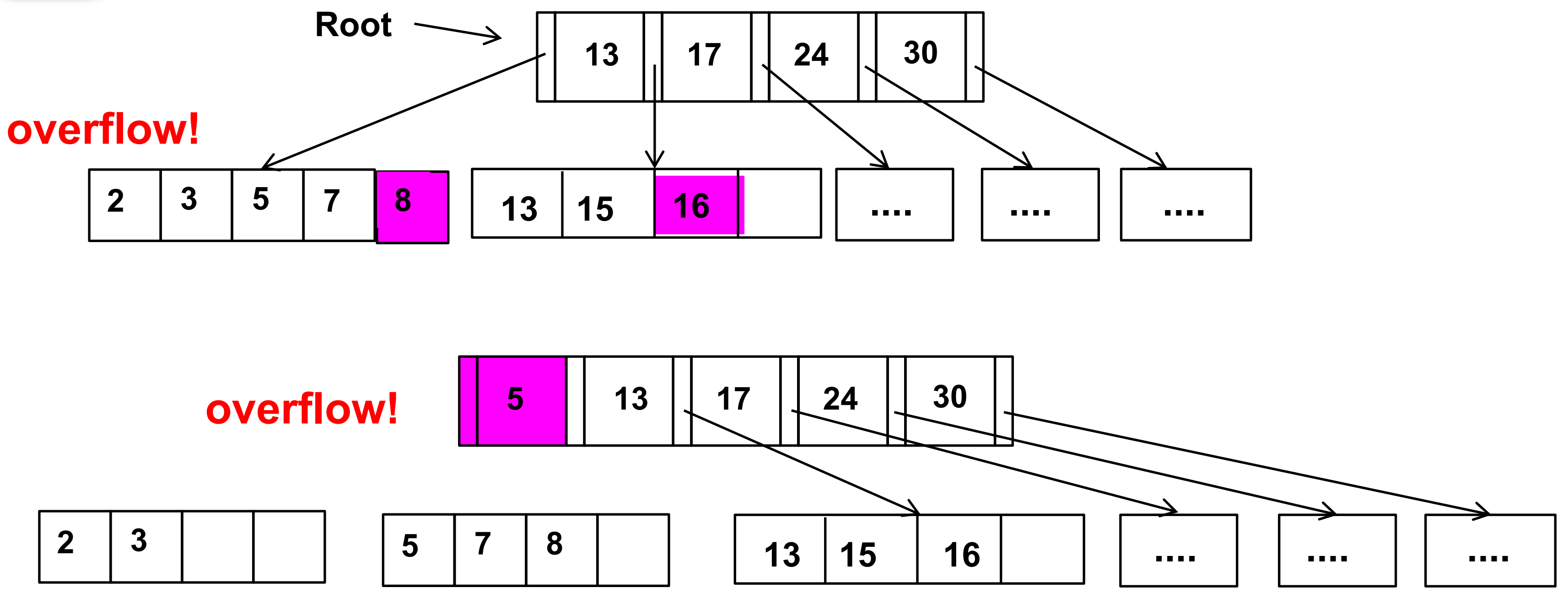


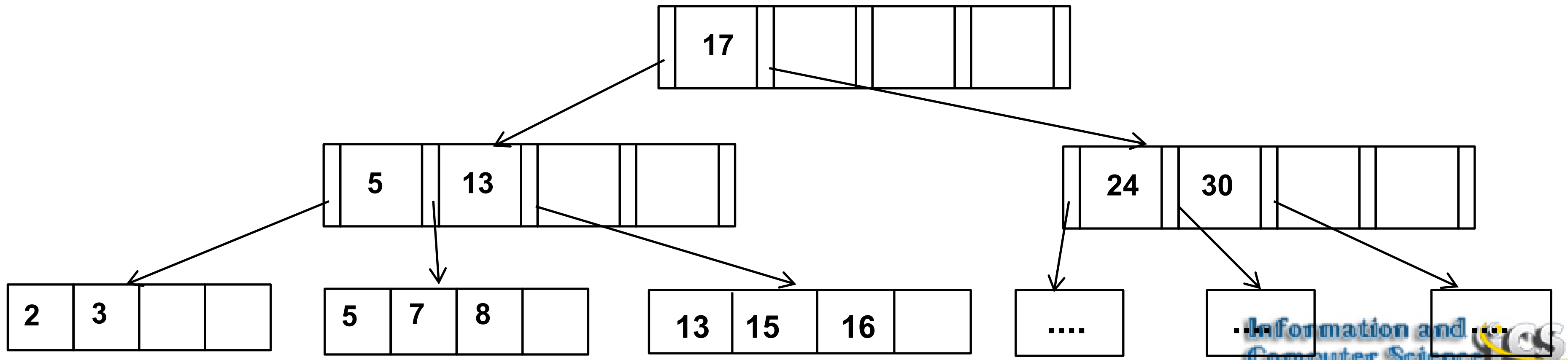
## Insertion in B+ Trees: Splitting an Internal Node (Cont.)



# Insertion in B+ Trees (Increasing the Height)

Example: Insert 16 then 8 in the following B+ tree of order M = 5, L = 4:





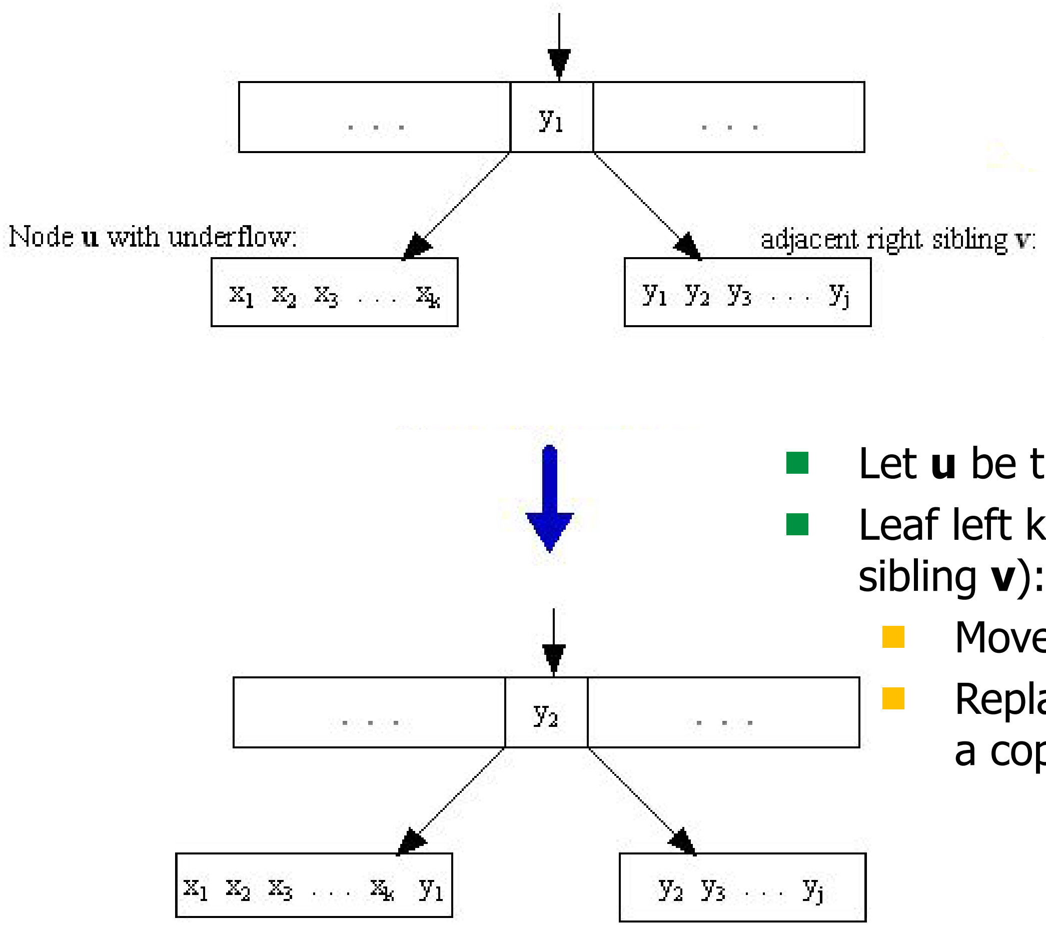
## Deletion in B+ Trees

- Since the internal nodes of a B+ tree (index keys) are a B-Tree, the deletion algorithm for B-Trees applies to it whenever a key is removed from the internal node.
- Hence, we will concentrate on what to do when deleting at the leaf node level.
- A B+ tree has two UNDEFLOW conditions:
  - Leaf underflow Condition:
    - A leaf underflows if after deleting a key from it, it contains $\lceil L/2 \rceil$ - 1 keys
  - Internal node underflow Condition:
    - An internal node (excluding the root node) underflows if in the key deletion process it contains $\lceil M/2 \rceil$ - 2 keys

## B+ Tree Deletion Algorithm

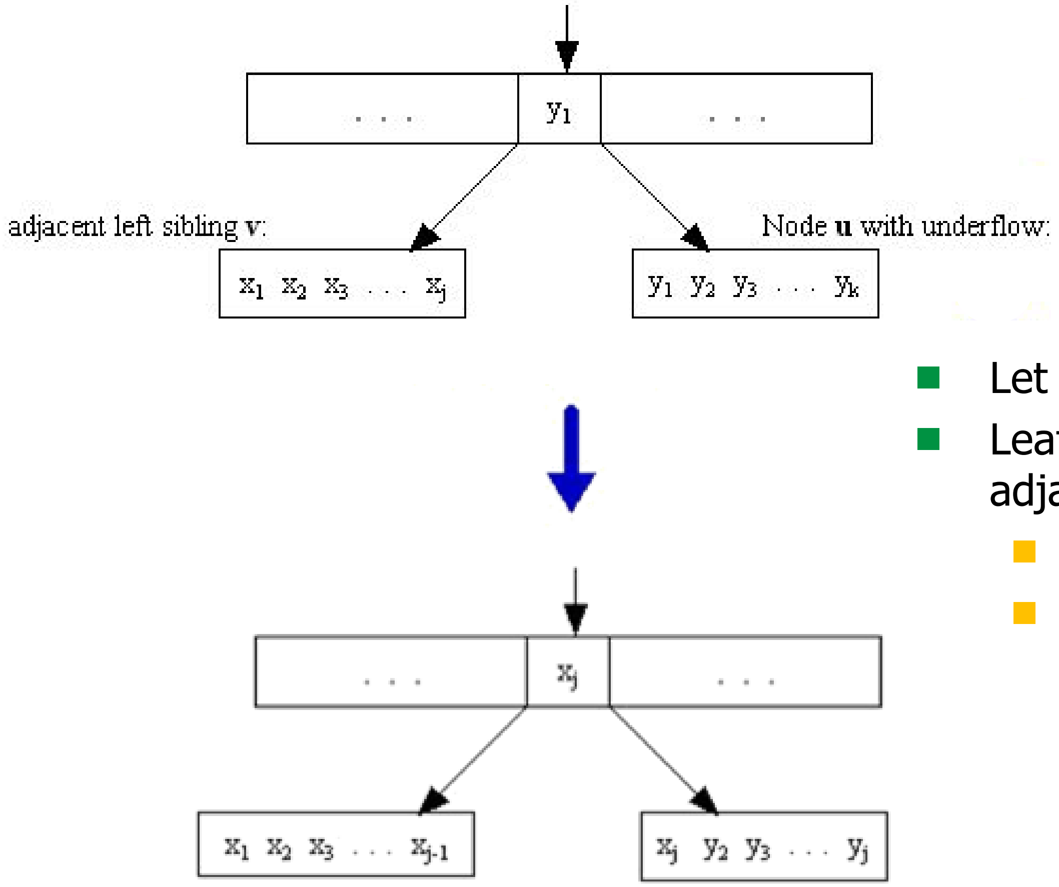
```
search for key targetKey;
if (targetKey is not found in a leaf) report an error;
else { // assume it is found in node x
  remove targetKey and its data reference from node x;
  if (node x did not underflow)  return; // deletion is complete
  else { // there is a leaf underflow (i.e. after deletion
         // node x contains ⌈L/2⌉-1 keys)
     if (there is an adjacent sibling with at least ⌈L/2⌉+1 keys)
       borrow the appropriate key from the adjacent sibling;
       // called Leaf Key Rotation where the appropriate key is the
      // minimum (if right sibling) or the maximum (if left sibling)
     else { // there is no adjacent sibling leaf with at least
            // ⌈L/2⌉+1 keys
        merge the two leaves;
        apply the B-Tree deletion algorithm on the key that was
        separating the two leaves in the previous level of the
        B+ Tree;
     }}}
```

## Deletion in B+ Tree: Leaf Key Rotation


. . .
$y_1$
. . .
Node **u** with underflow:
$x_1$ $x_2$ $x_3$ . . . $x_k$
adjacent right sibling **v**:
$y_1$ $y_2$ $y_3$ . . . $y_j$
. . .
$y_2$
. . .
$x_1$ $x_2$ $x_3$ . . . $x_k$ $y_1$
$y_2$ $y_3$ . . . $y_j$

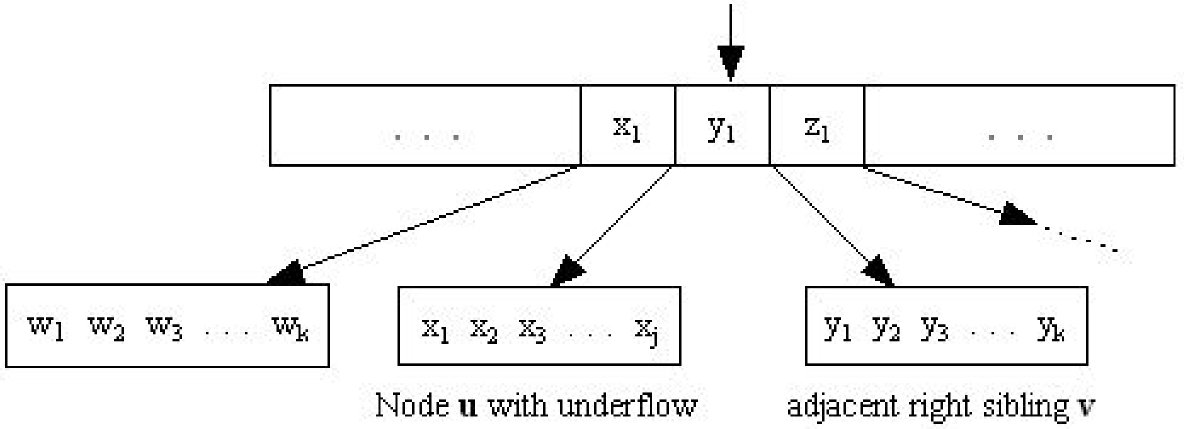
- Let **u** be the node with leaf underflow.
- Leaf left key rotation (borrowing from adjacent right sibling **v**):
  - Move the minimum key of **v** to **u**
  - Replace the separating key between **u** and **v** with a copy of the new minimum in **v**

## Deletion in B+ Tree: Leaf Key Rotation (cont'd)


. . .
$y_1$
. . .
adjacent left sibling v:
Node u with underflow:
$x_1$ $x_2$ $x_3$ . . . $x_j$
$y_1$ $y_2$ $y_3$ . . . $y_k$
. . .
$x_j$
. . .
$x_1$ $x_2$ $x_3$ . . . $x_{j-1}$
$x_j$ $y_2$ $y_3$ . . . $y_j$

- Let **u** be the node with leaf underflow.
- Leaf right key rotation (borrowing from adjacent left sibling **v**)
  - Move the maximum key of **v** to **u**
  - Replace the separating key between **u** and **v** with a copy of the new minimum in **u**

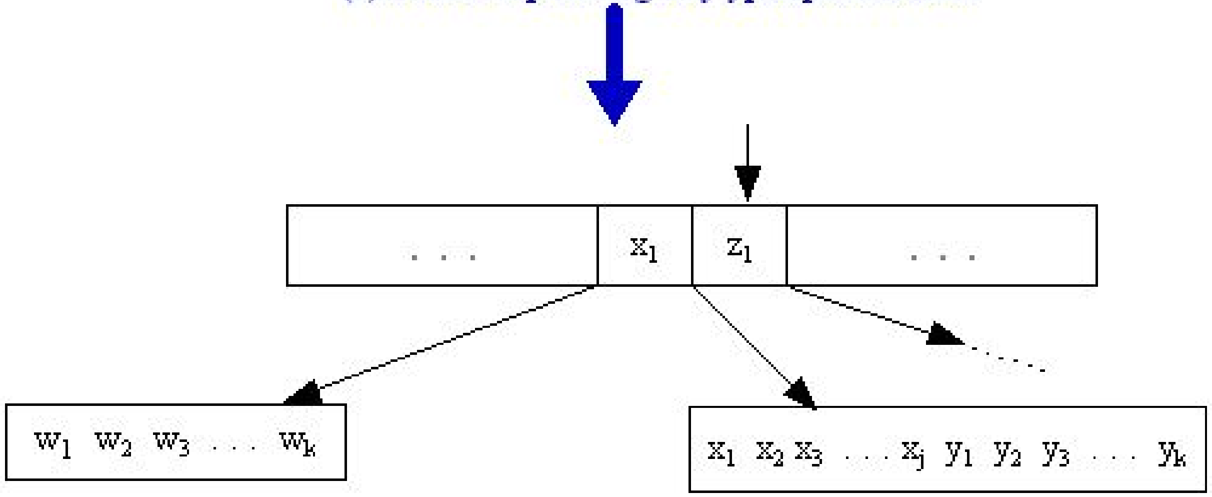
## Leaf keys Merging (with Adjacent Right Sibling)


. . .
$x_1$
$y_1$
$z_1$
. . .
$w_1$ $w_2$ $w_3$ . . . $w_k$
$x_1$ $x_2$ $x_3$ . . . $x_j$
$y_1$ $y_2$ $y_3$ . . . $y_k$
Node **u** with underflow
adjacent right sibling **v**

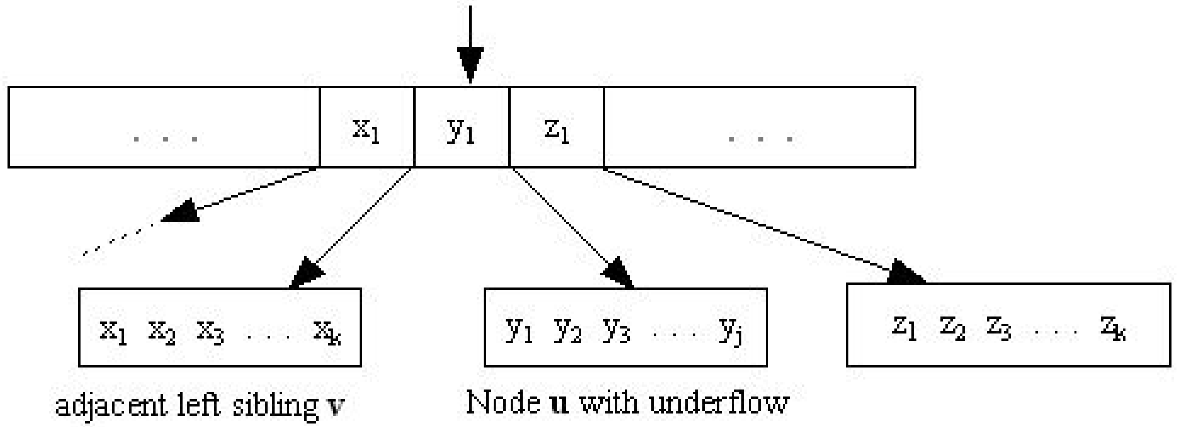
**(1) Move keys from node u to node v**

**(2) Delete node u and its reference in parent node**

**(3) Delete separating key $y_1$ in parent node**


. . .
$x_1$
$z_1$
. . .
$w_1$ $w_2$ $w_3$ . . . $w_k$
$x_1$ $x_2$ $x_3$ . . . $x_j$ $y_1$ $y_2$ $y_3$ . . . $y_k$

## Leaf keys Merging (with Adjacent Left Sibling)

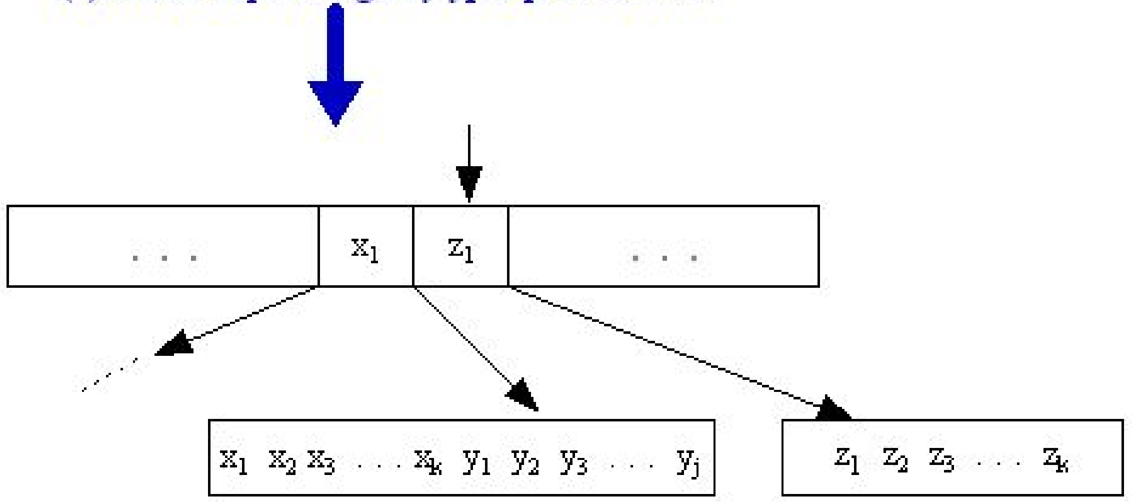


. . .
$x_1$
$y_1$
$z_1$
. . .
$x_1$ $x_2$ $x_3$ . . . $x_k$
$y_1$ $y_2$ $y_3$ . . . $y_j$
$z_1$ $z_2$ $z_3$ . . . $z_k$
adjacent left sibling **v**
Node **u** with underflow

**(1) Move keys from node u to node v**

**(2) Delete node u and its reference in parent node**

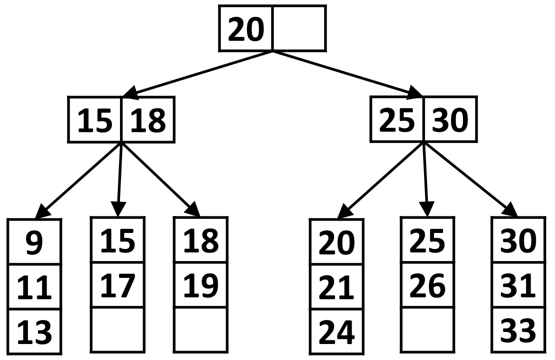
**(3) Delete separating key $y_1$ in parent node**

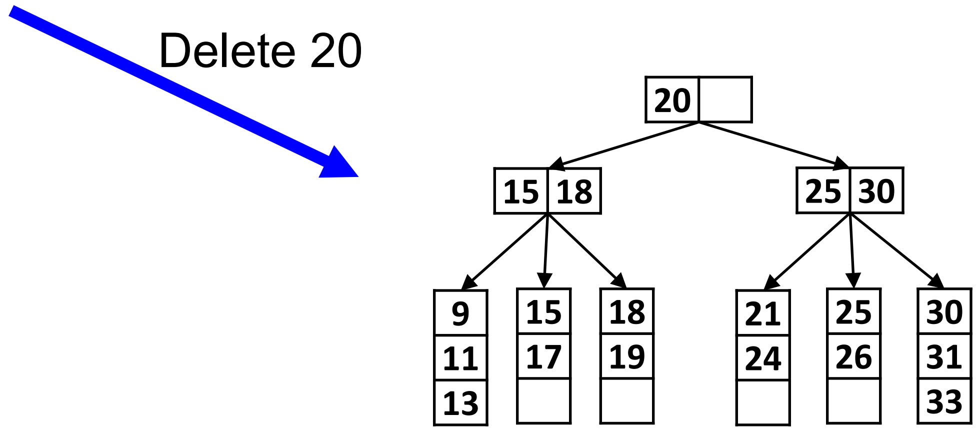


. . .
$x_1$
$z_1$
. . .
$x_1$ $x_2$ $x_3$ . . . $x_k$ $y_1$ $y_2$ $y_3$ . . . $y_j$
$z_1$ $z_2$ $z_3$ . . . $z_k$

# Deletion in B+ Tree - Case1: No underflow

Example: Delete 20 from the following B+ tree of order M = 3 and L = 3

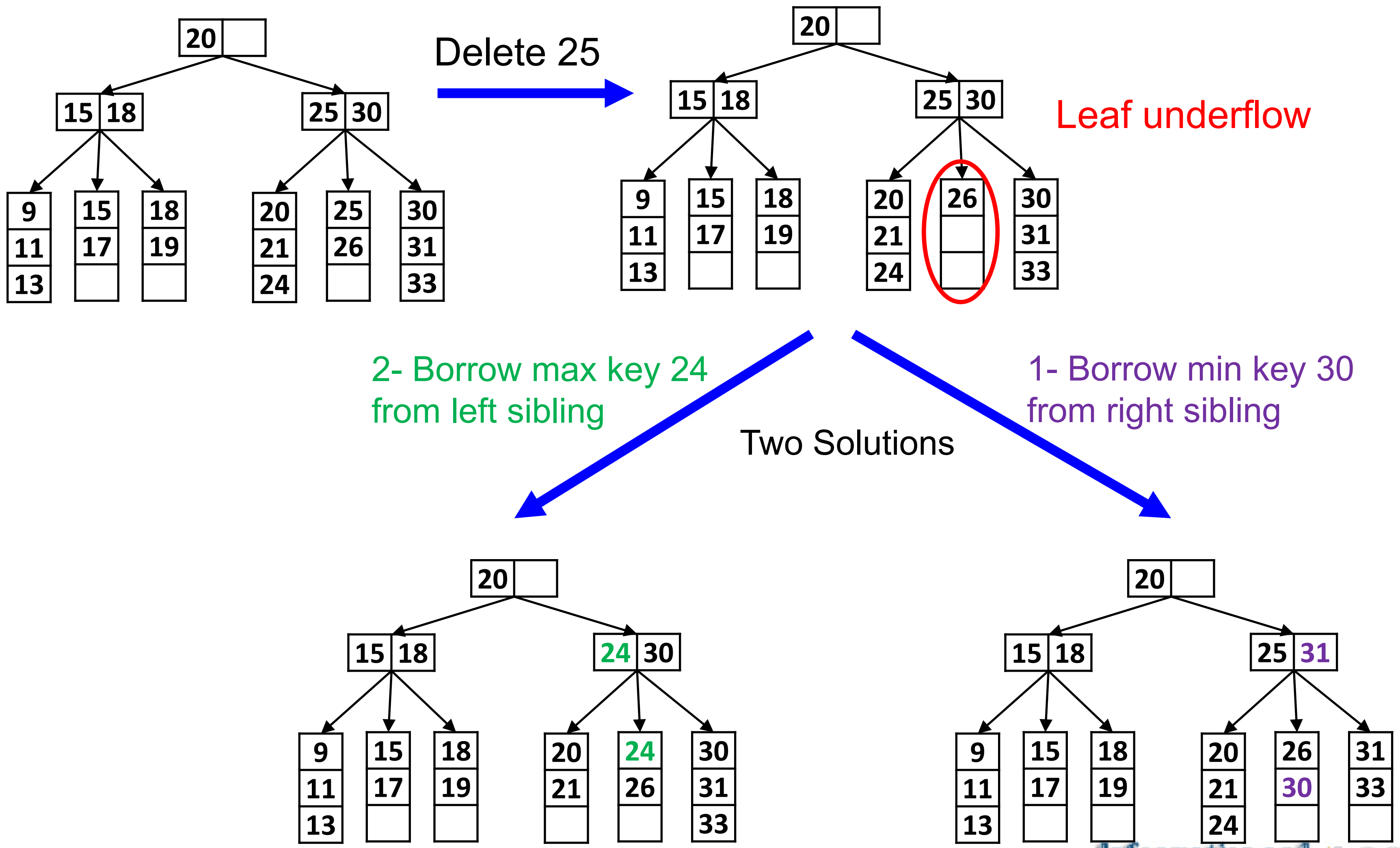




No leaf underflow

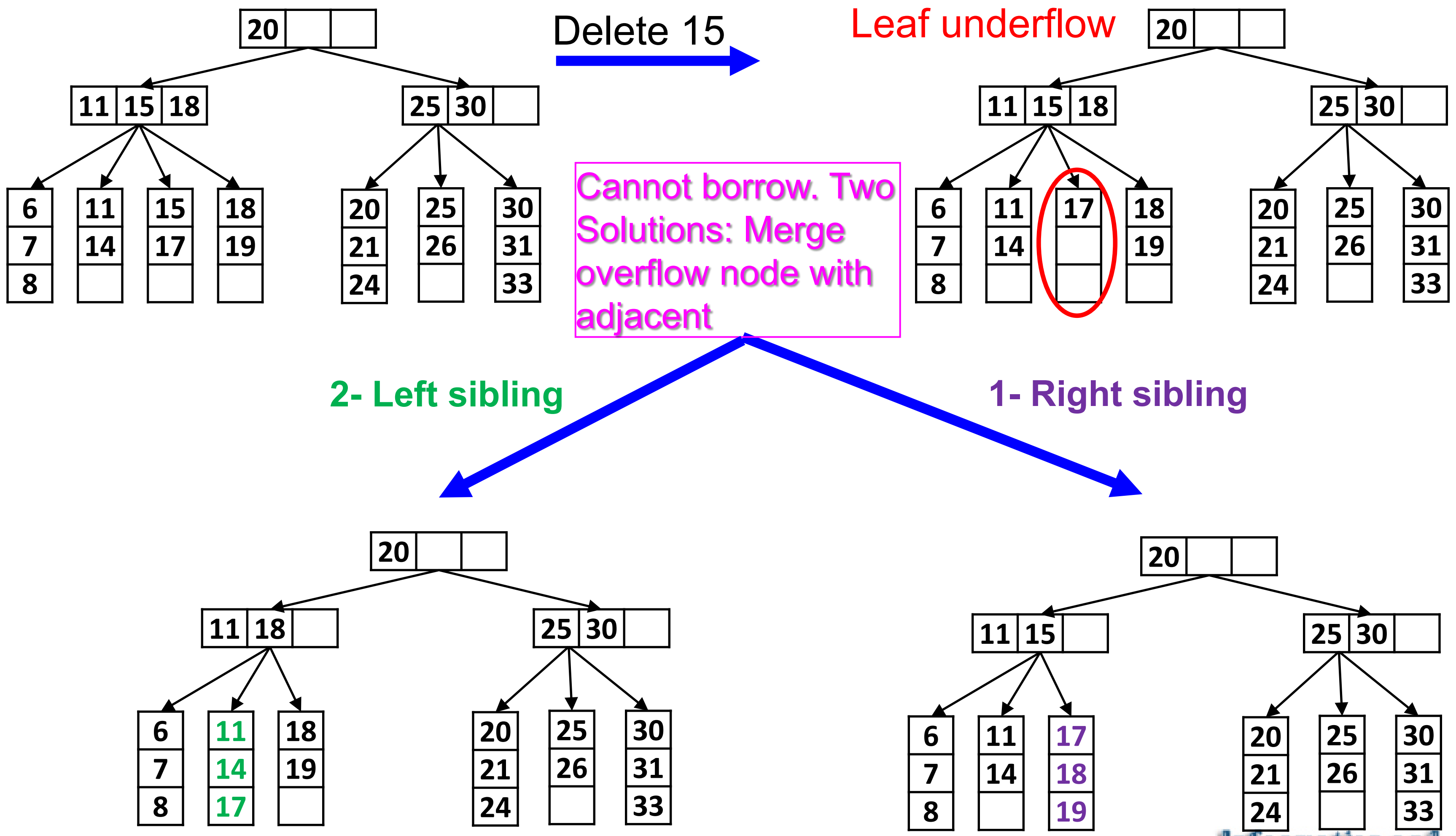
## Deletion in B+ Tree- Case 2: Leaf Key Borrowing

Example: Delete 25 from the following B+ tree of order M = 3 and L = 3



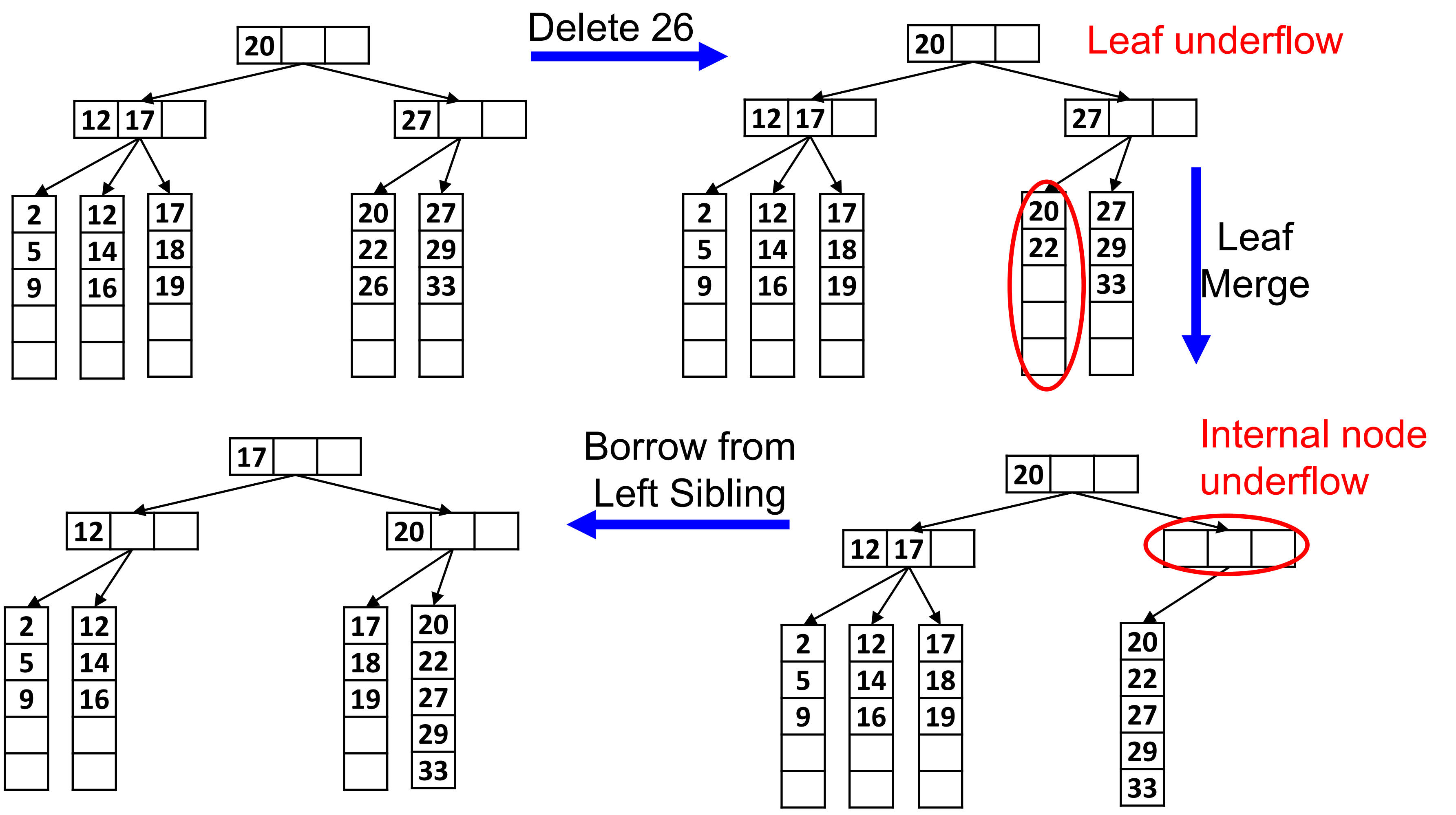
# Deletion in B+ Tree – Case 3: Leaf Merging

Example: Delete 15 from the following B+ tree of order M = 4 and L = 3



## Deletion in B+ Tree – Case 4: Applying Rest of Cases in B-Tree Deletion Algorithm on the Internal Nodes

- Example: Delete 26 from the following B+ tree of order M = 4 and L = 5



Delete 26
Leaf underflow
Leaf Merge
Internal node underflow
Borrow from Left Sibling
20
12 17
27
2 5 9
12 14 16
17 18 19
20 22 26
27 29 33
20 22
20 22 27 29 33
17
12
20