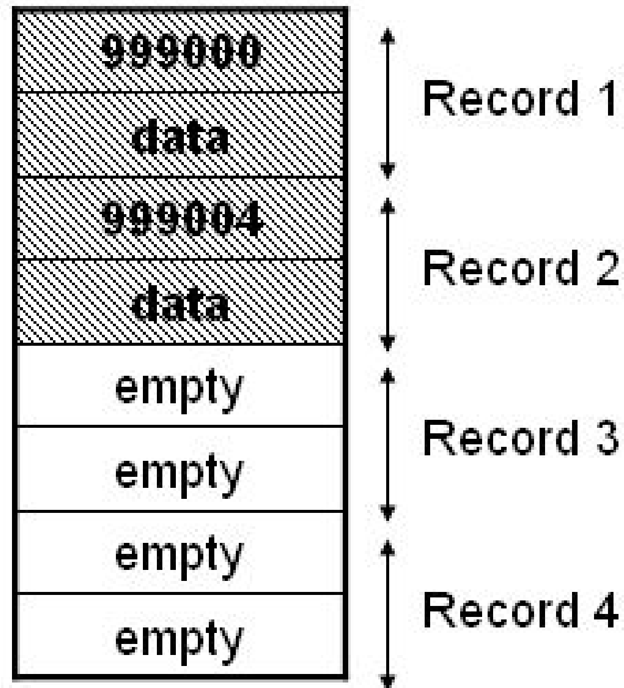
# B-Trees

- Disk Storage
- What is a multiway tree?
- What is a B-tree?
- Why B-trees?
- Insertion in a B-tree
- Deletion in a B-tree

# Disk Storage

- 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.

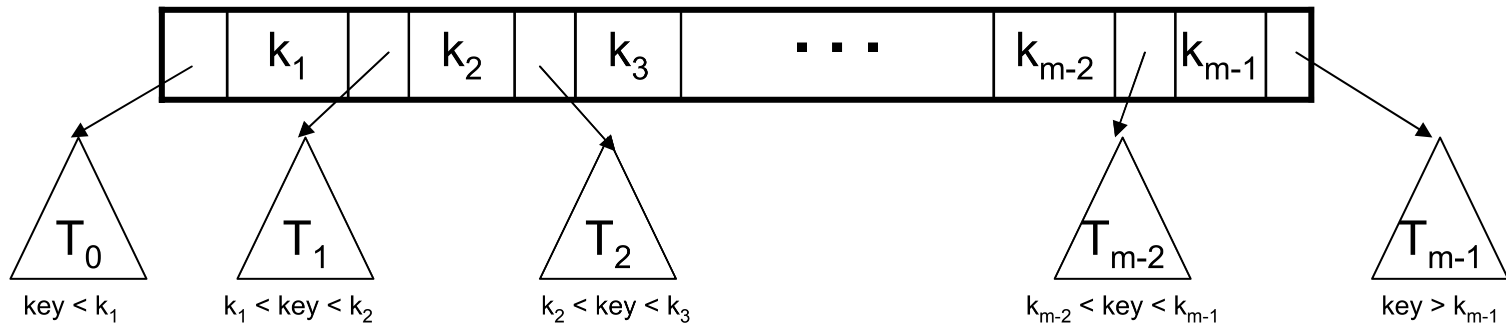


# Motivation for studying Multi-way and B-trees

- 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 **may be empty**.
  - Each node consists of at least **1** and at most **m-1** distinct keys
  - The keys in each node are sorted.

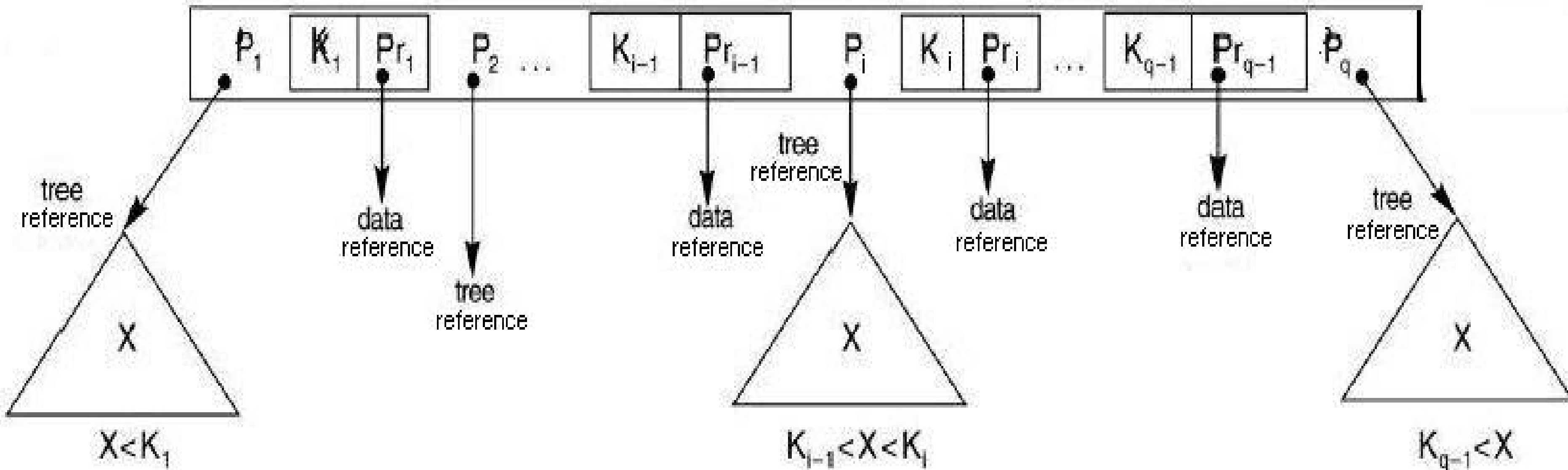



- The keys and subtrees of a non-leaf node are ordered as:

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

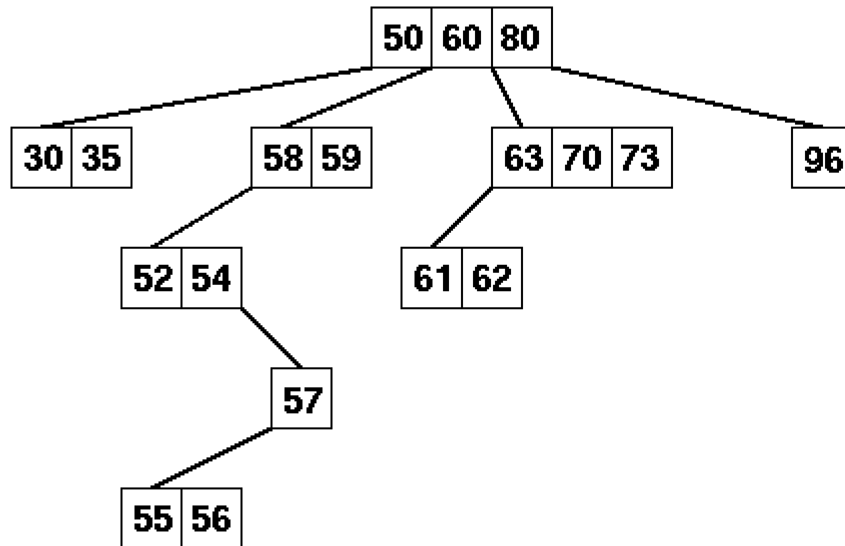
  - All keys in subtree T0 are less than k1.
  - All keys in subtree $T_i$ , $1 <= i <= m - 2$, are greater than $k_i$ but less than $k_{i+1}$.
  - All keys in subtree $T_{m-1}$ are greater than $k_{m-1}$

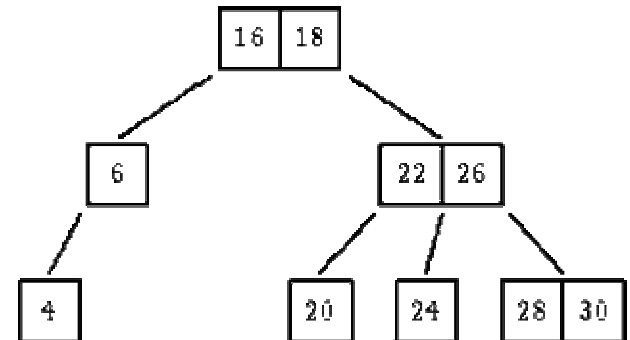
# The node structure of a Multi-way tree




- 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.

# 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:**

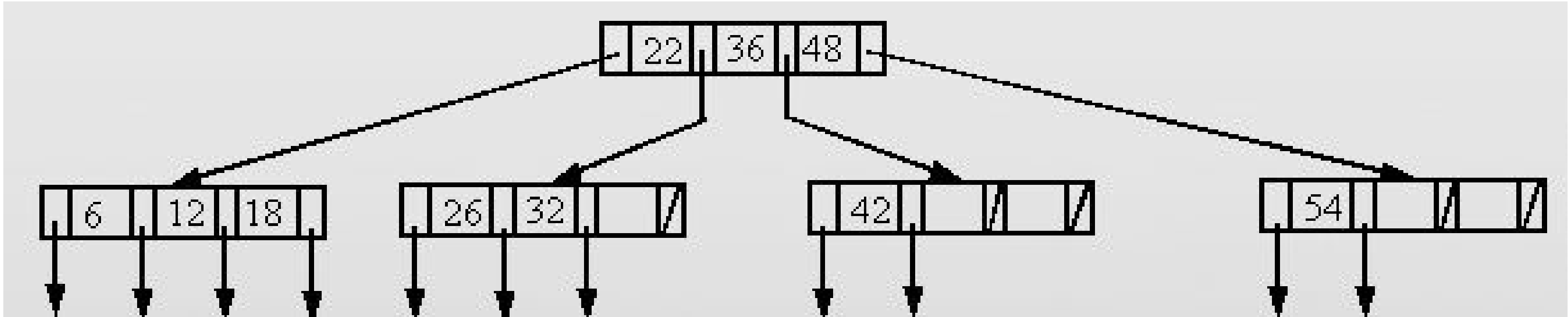
This may be zero, if the node is a leaf as well

| | 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 |

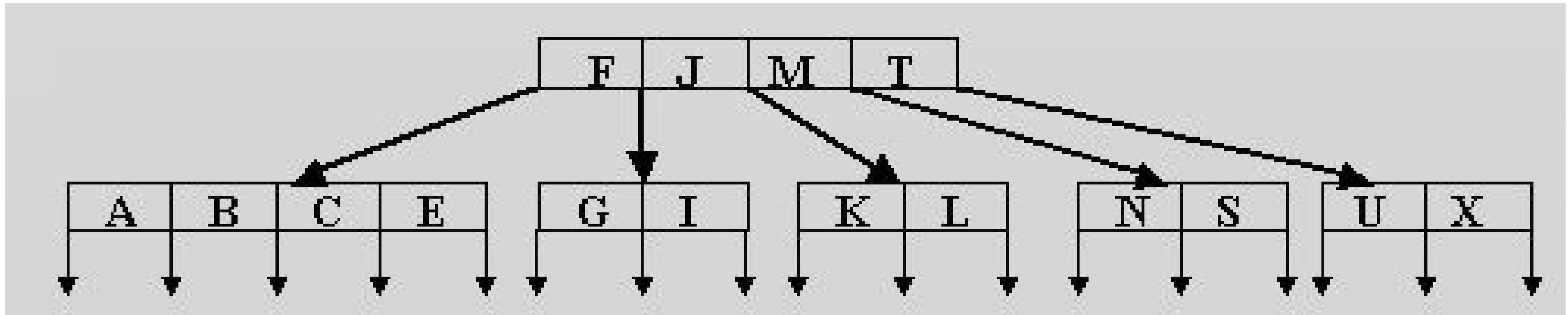
These will be zero if the node is a leaf as well

# 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

# 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:

$$h <= 1 + \log_{\lceil m/2 \rceil} ((n + 1) / 2)$$

- 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.

# 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:**

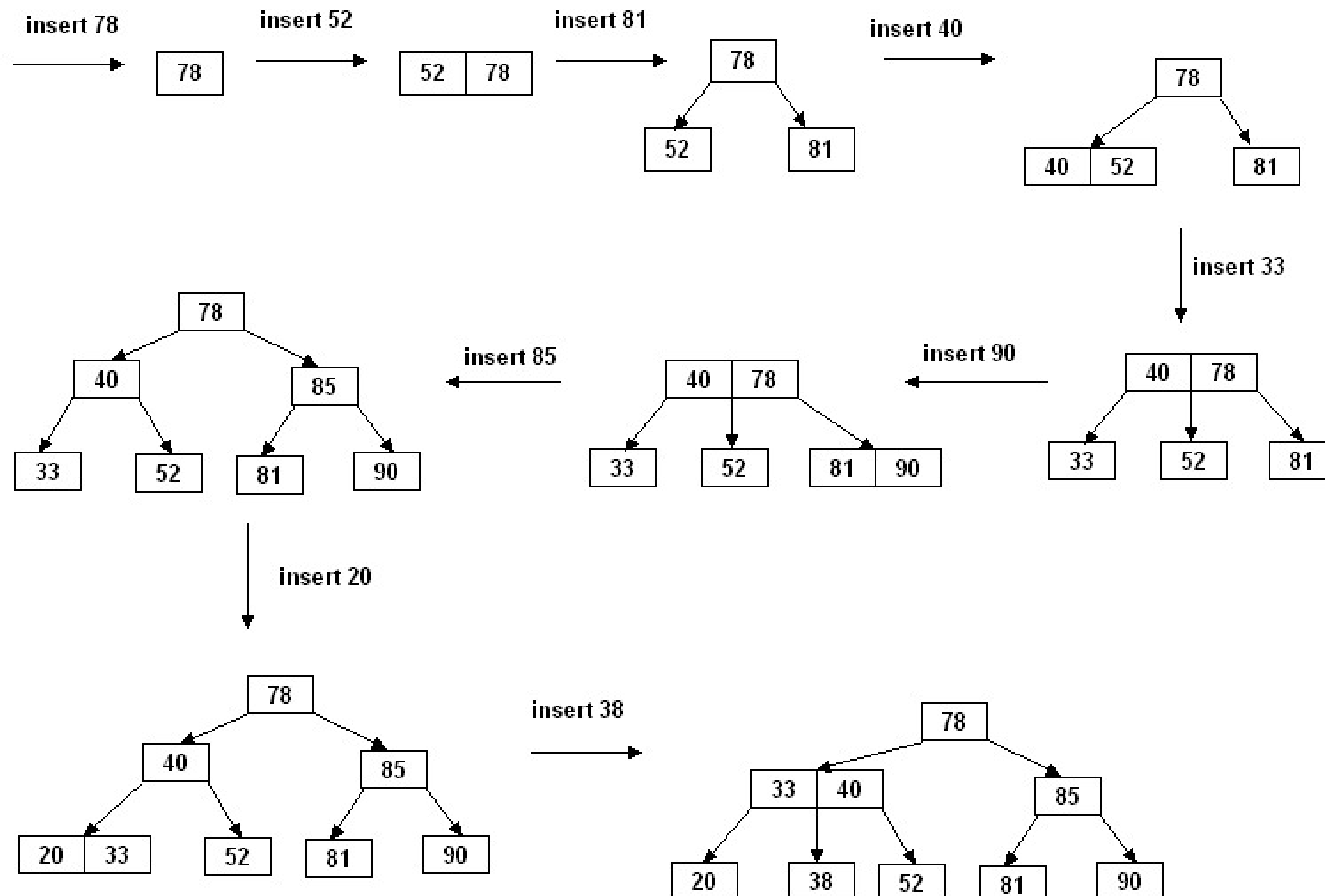
  **If a node overflows, split it into two, propagate the "middle" key to the parent of the node. If the parent overflows the process propagates upward. If the node has no parent, create a new root node.**

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

# Insertion in B-Trees

- **Insertion in a B-tree of odd 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



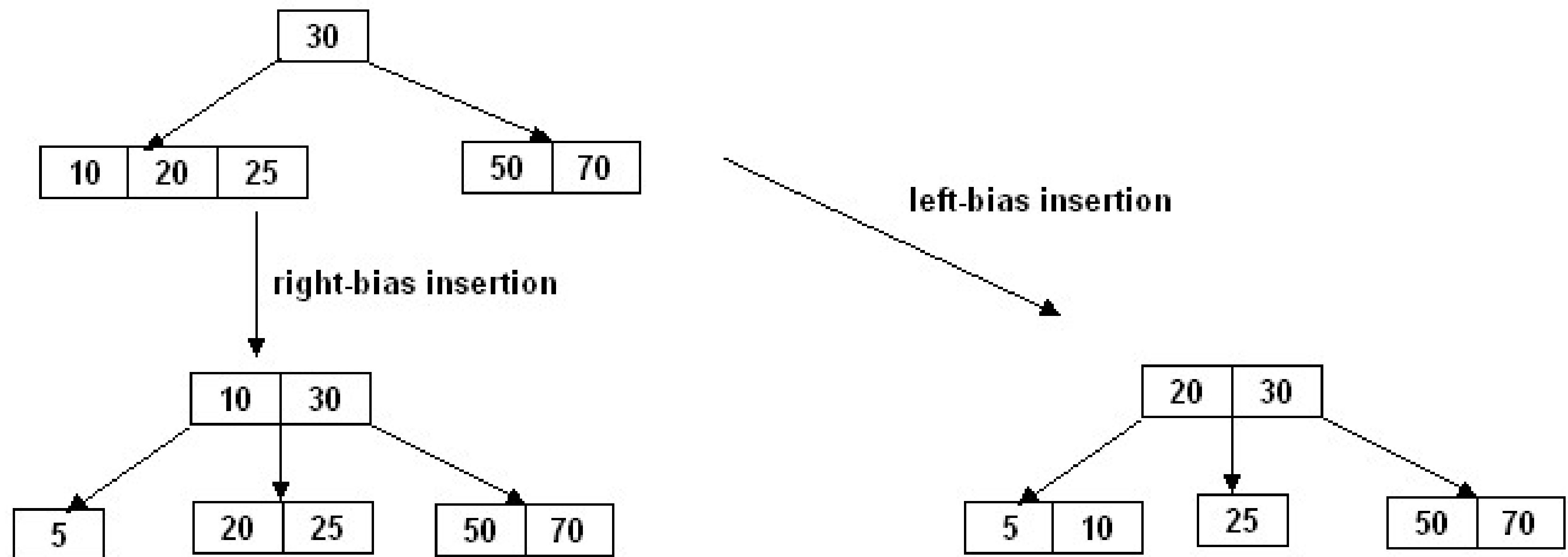
# Insertion in B-Trees

- **Insertion in a B-tree of <u>even</u> 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;
}
```

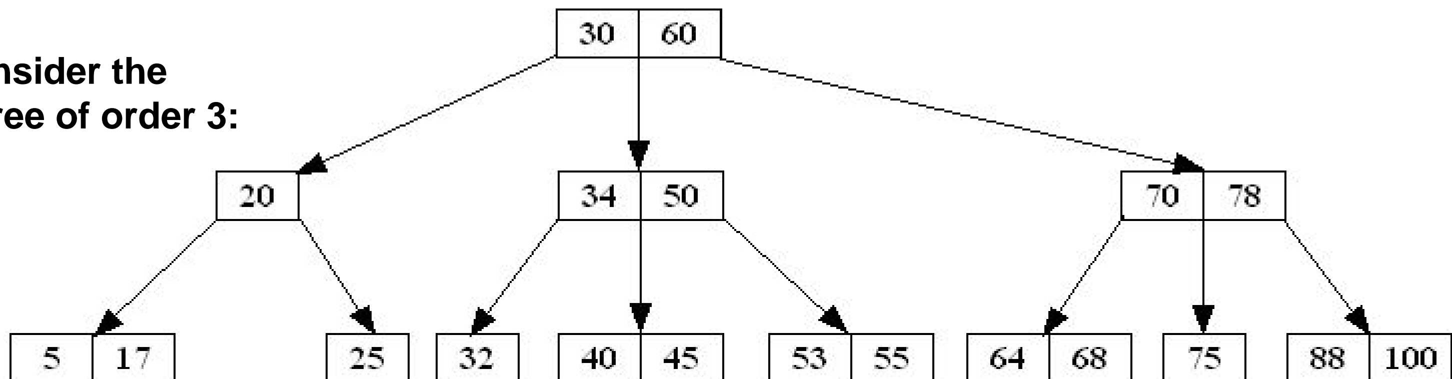
# Deletion in B-Tree

- **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 $\lceil \mathbf{m / 2} \rceil$ 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.

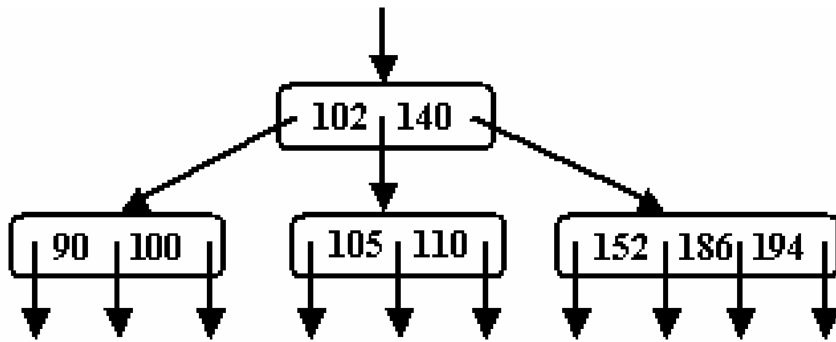
     perform a merging

# Deletion in B-Tree

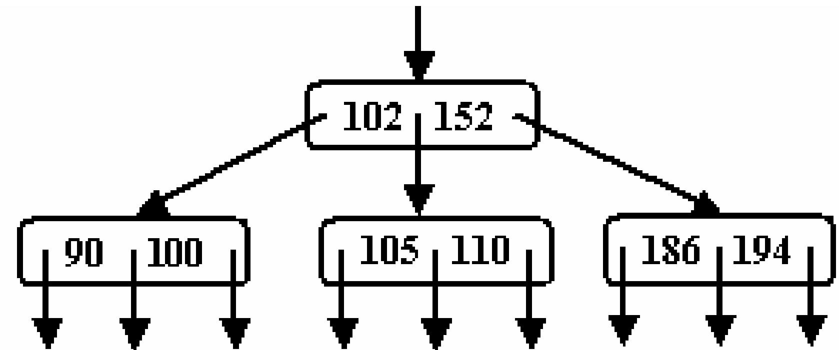
Case1: The leaf does not underflow.

Example:

**B-tree of order 4**




Delete 140

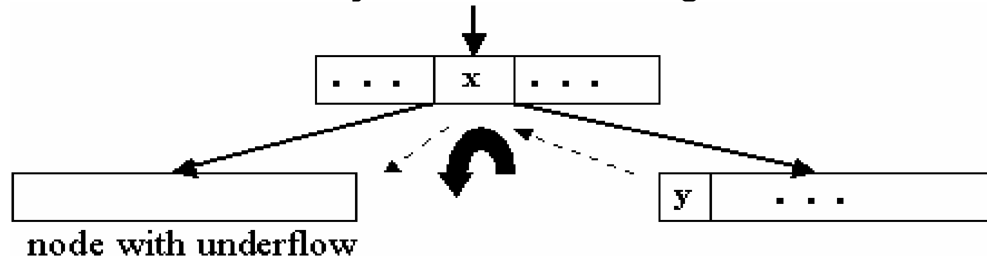


# Deletion in B-Tree (cont'd)

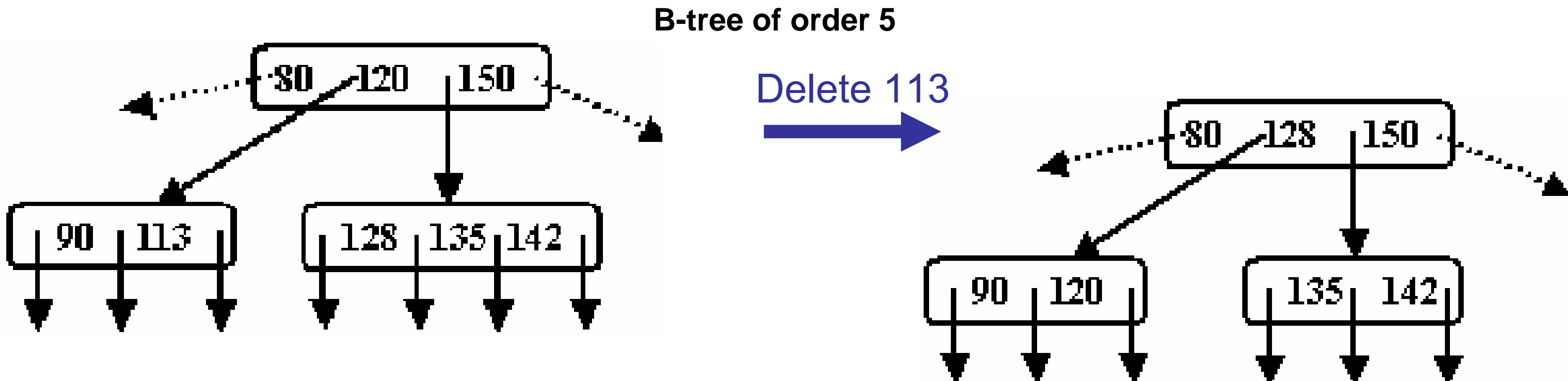
Case2: The leaf underflows and the adjacent right sibling has at least $\lceil m / 2 \rceil$ keys.

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:

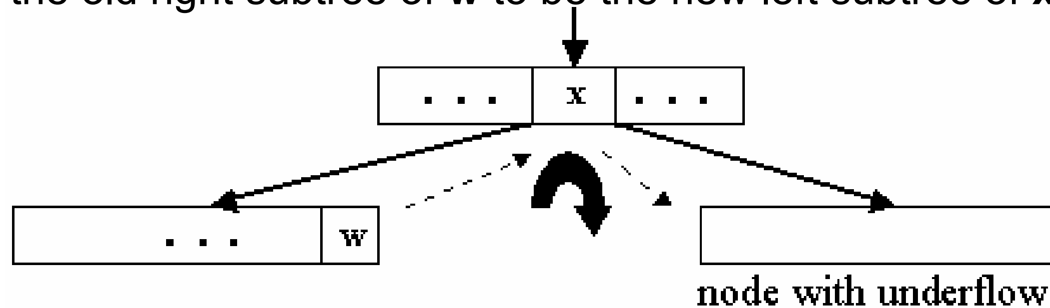


# Deletion in B-Tree (cont'd)

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

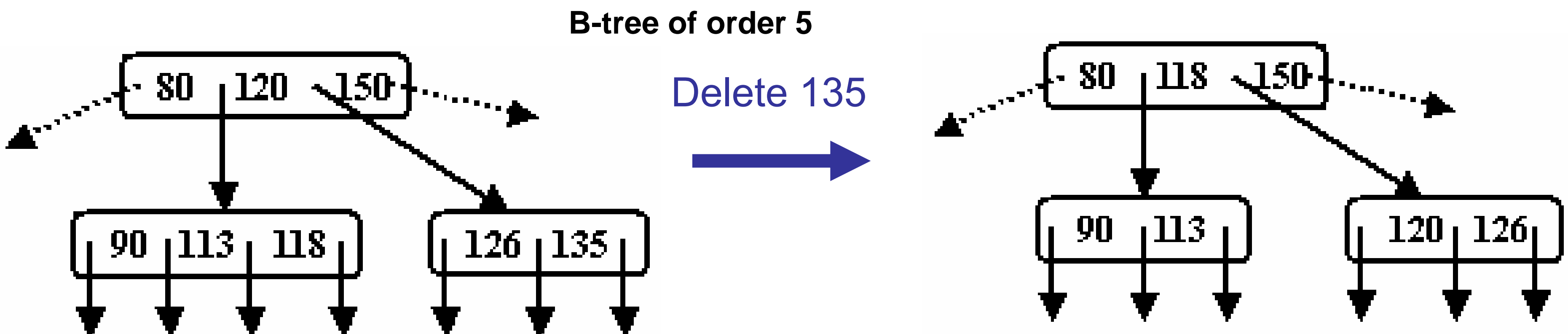
Perform a right key-rotation:

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**



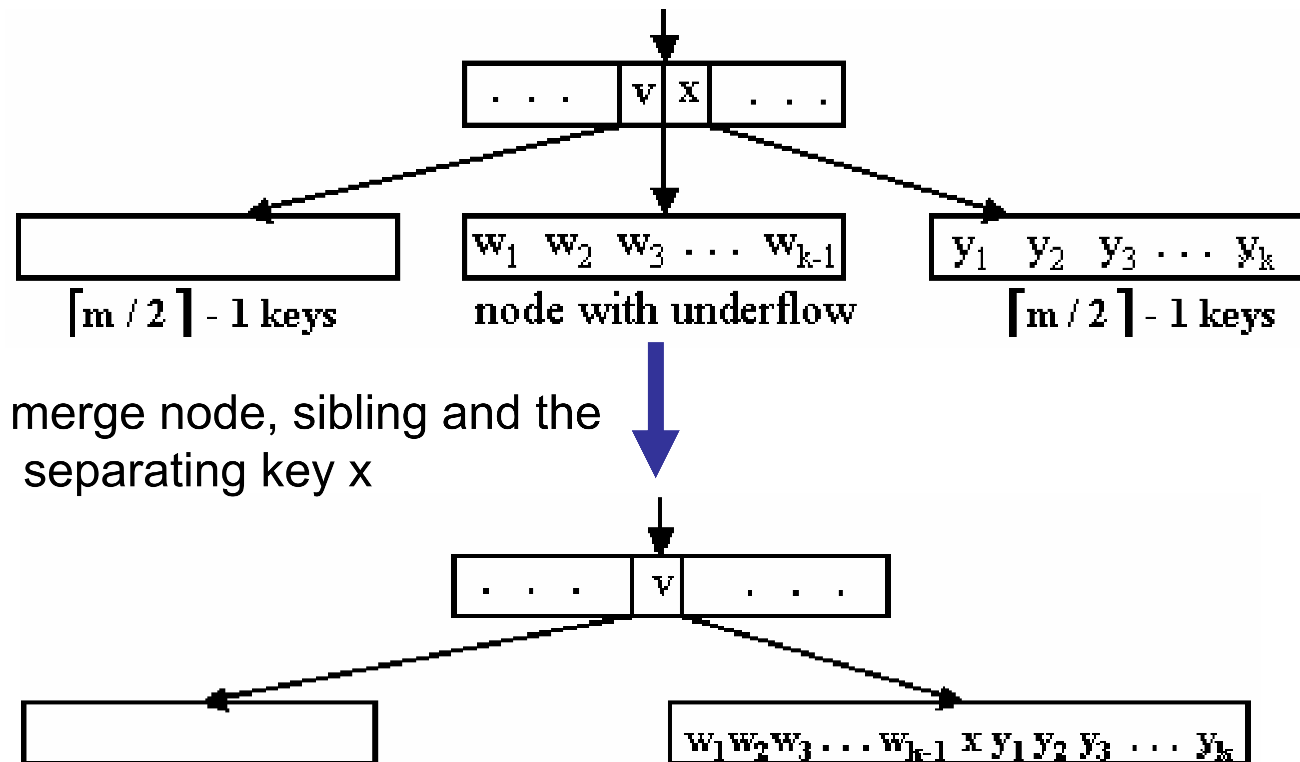

Example:

**B-tree of order 5**



# Deletion in B-Tree (cont'd)

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

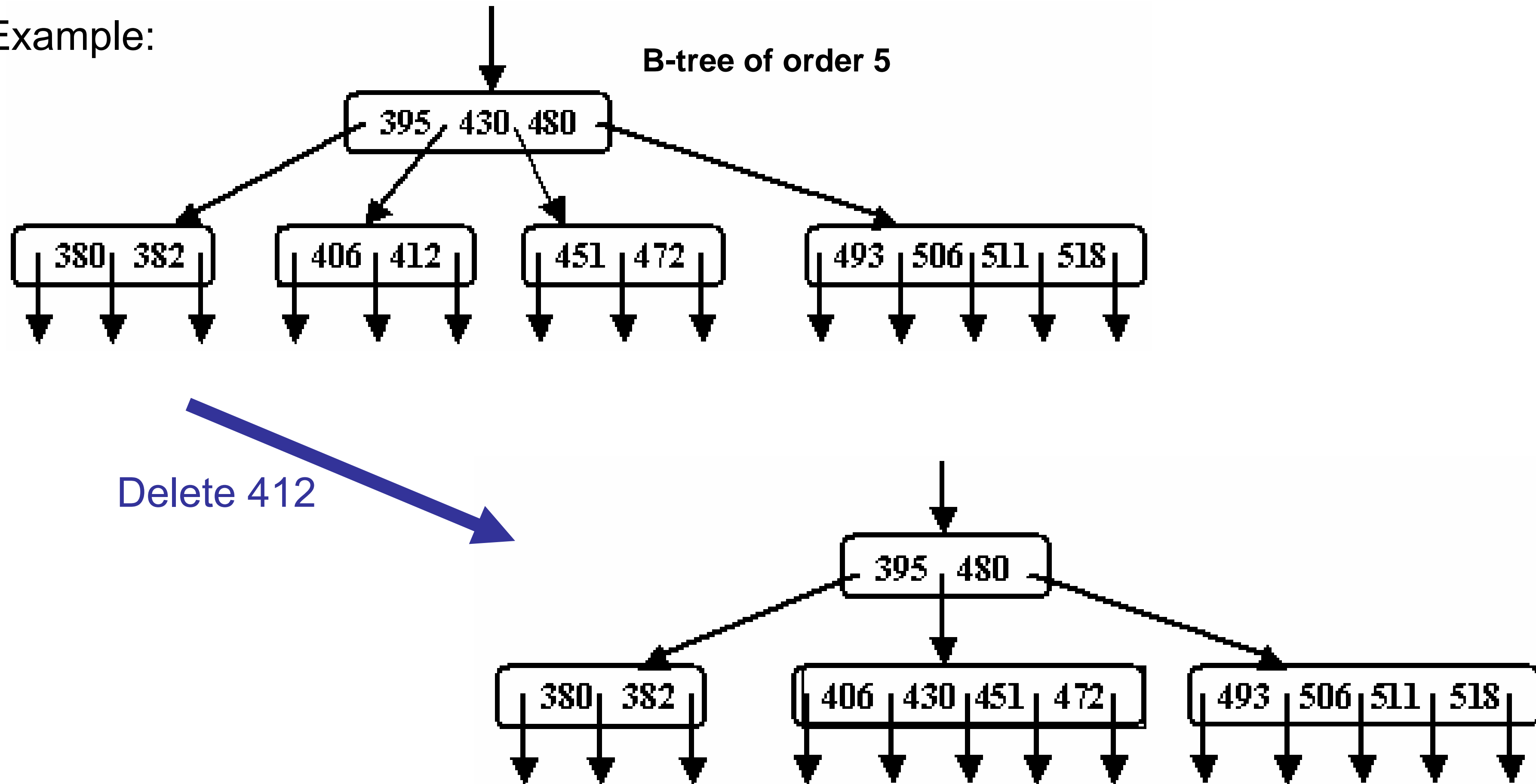



If the parent of the merged node underflows, the merging process propagates upward. In the limit, a root with one key is deleted and the height decreases by one.

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

# Deletion in B-Tree (cont'd)

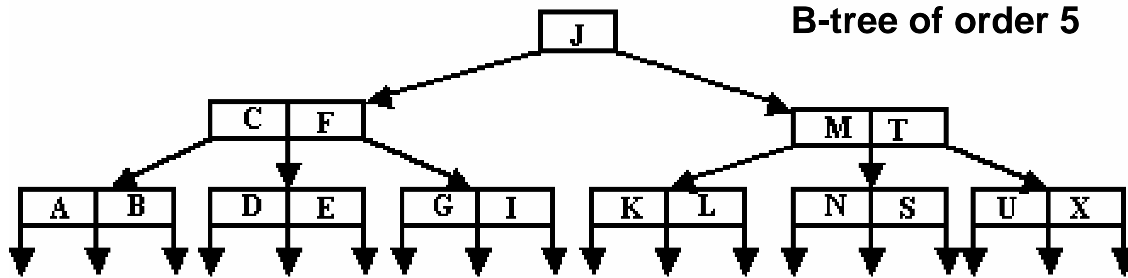
Example:



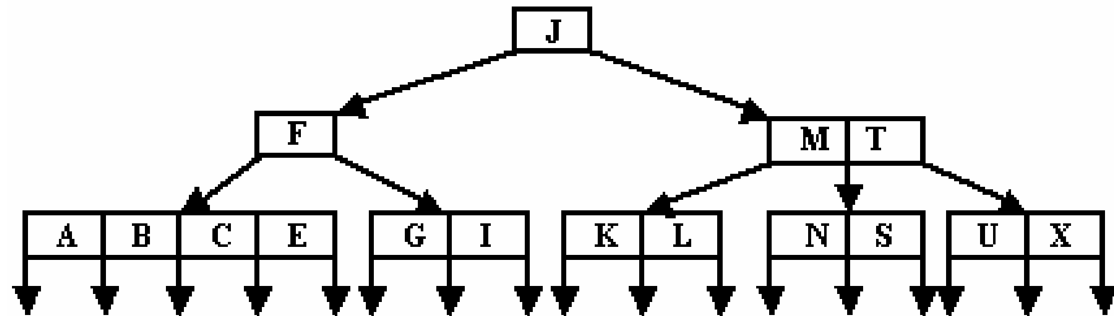

The parent of the merged node does not underflow. The merging process does not propagate upward.

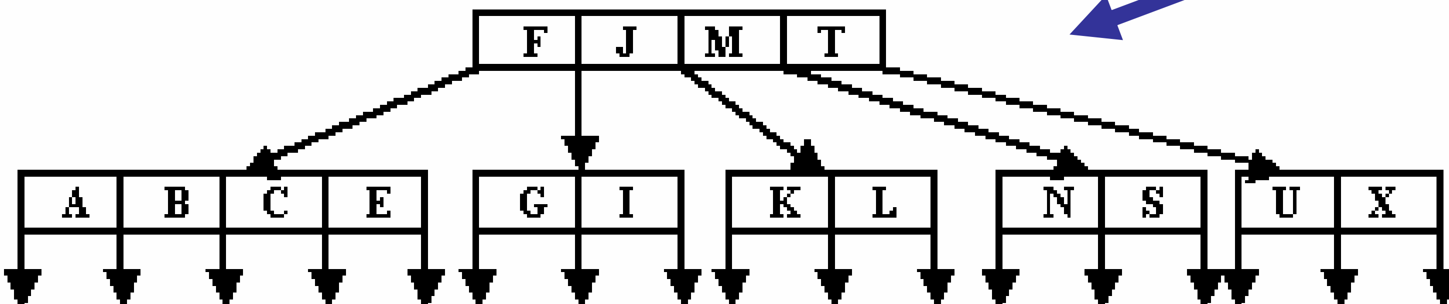
# Deletion in B-Tree (cont'd)

Example:



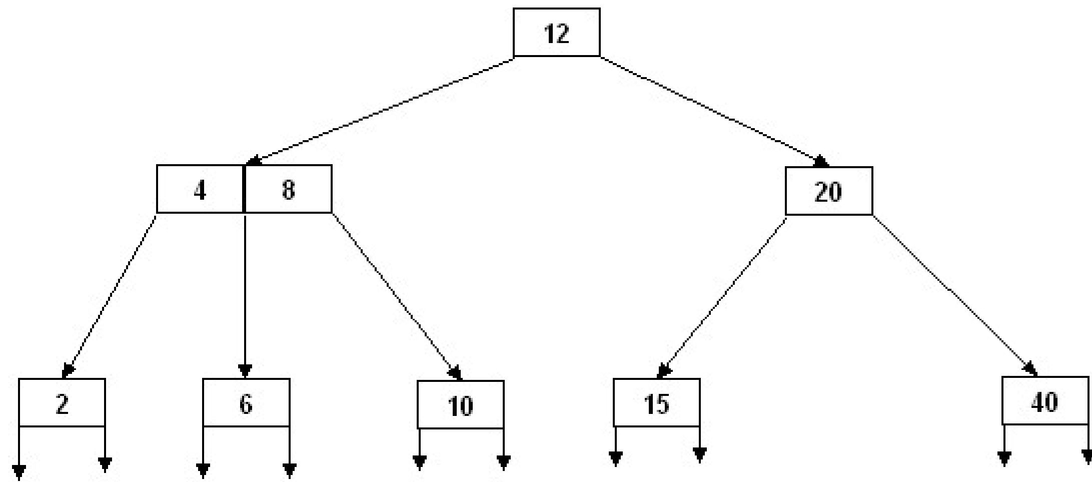

Delete D



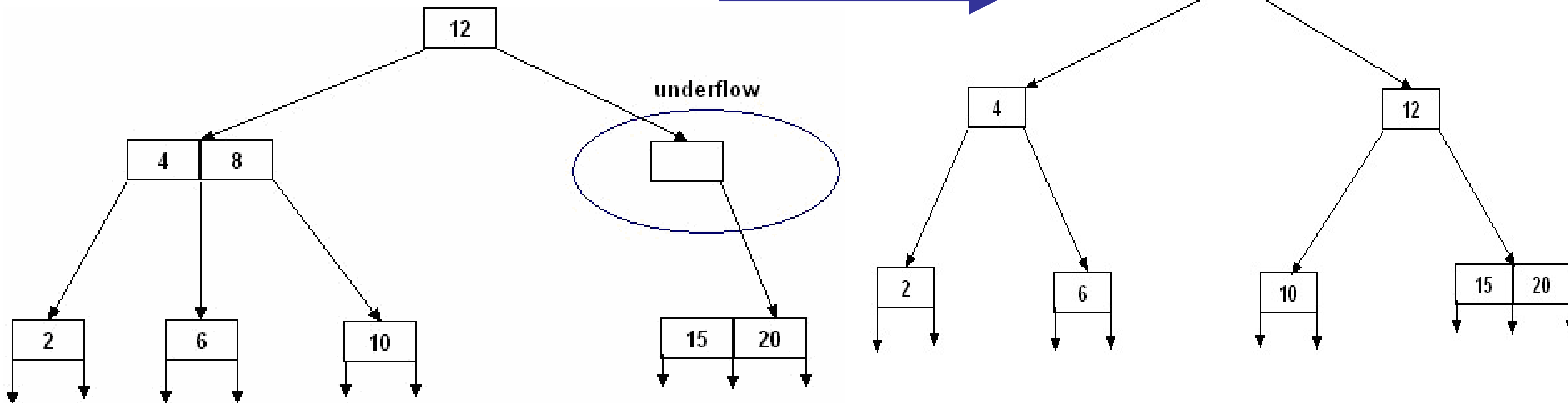
# Deletion : Special Case, involves rotation and merging

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




merge 15 and 20

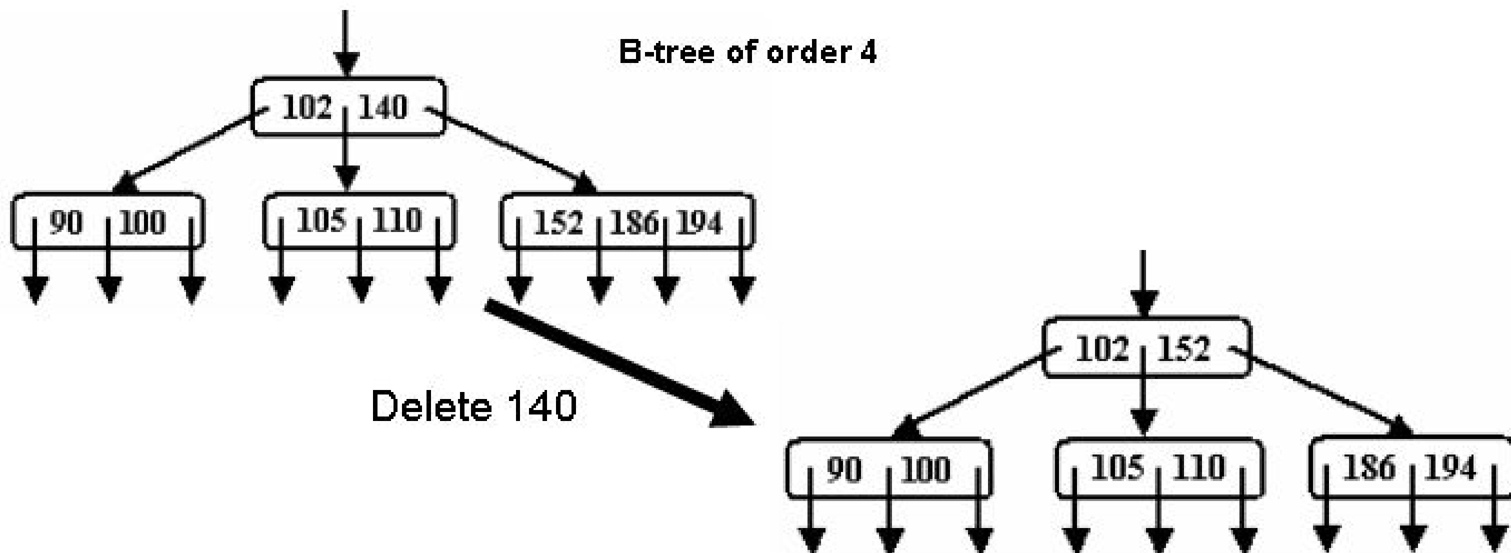
rotate 8 and its right subtree



# Deletion of a non-leaf node

Deletion of a non-leaf key can always be done in two different ways: by first swapping the key with its successor or predecessor. The resulting trees may be similar or they may be different.

Example: Delete the key 140 in the following partial B-tree of order **4:**



# 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;
}
```