MultiwayTrees

sushmakadge@somaiya.edu

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Outline

- Tree – concept
- General tree
- Types of trees
- Binary tree: representation, operation
- Binary tree traversal
- Binary search tree
- BST- The data structure and implementation
- Threaded binary trees.
- Search Trees –
  - AVL tree, Multiway Search Tree, B Tree, B+ Tree, and Trie,
- Applications/Case study of trees.
- Summary
- Queries?

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Outline

- Multiway Trees – concept
- B trees
- B+ trees (Introduction)
- Applications of trees
- Summary
- Queries?

Somaiya TRUST

Structure of a binary search tree node

- Every node in a binary search tree contains one value and two pointers, *left* and *right*, which point to the node's left and right sub-trees, respectively.

Pointer to left sub-tree	Value or Key of the node	Pointer to right sub-tree

- Binary search tree M = 2, so it has one value and two sub-trees.
- In other words, every internal node of an M-way search tree consists of pointers to M sub-trees and contains M – 1 keys, where M > 2.

Multiway Tree Definition

- A multiway tree of order m (or an m-way tree) is one in which a tree can have m children.
- Nodes in an m-way tree will be made up of key fields, and pointers to children. The structure of an M-way search tree node is shown in Fig.

P_0	K_0	P_1	K_1	P_2	K_2	.....	P_{n-1}	K_{n-1}	P_n

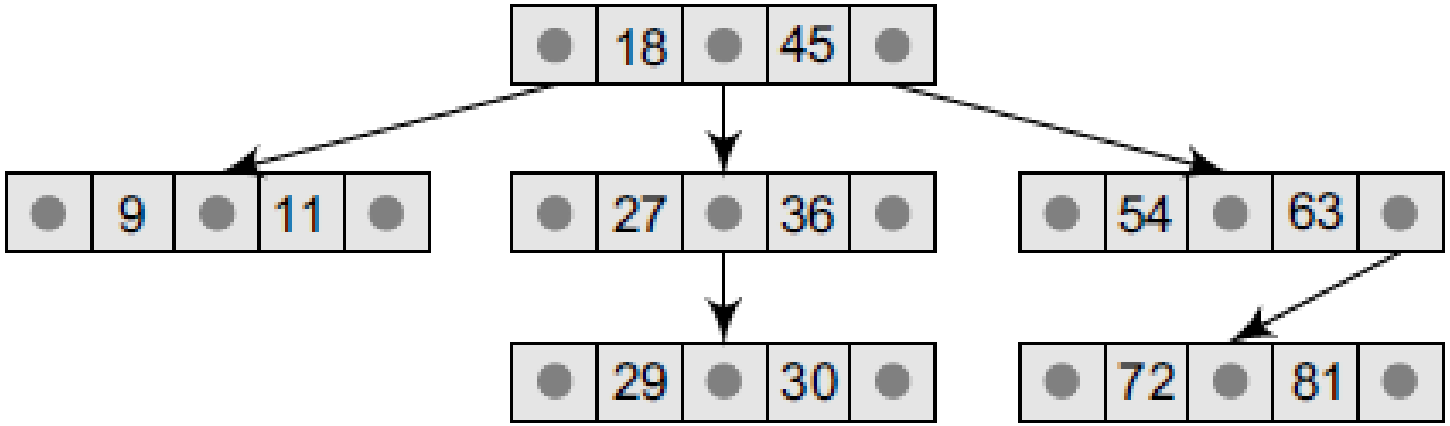
- P0, P1, P2, ..., Pn are pointers to the node's sub-trees and K0, K1, K2, ..., Kn–1are the key values of the node. i.e. Ki < Ki+1.
- All the key values are stored in ascending order.

Multiway Tree

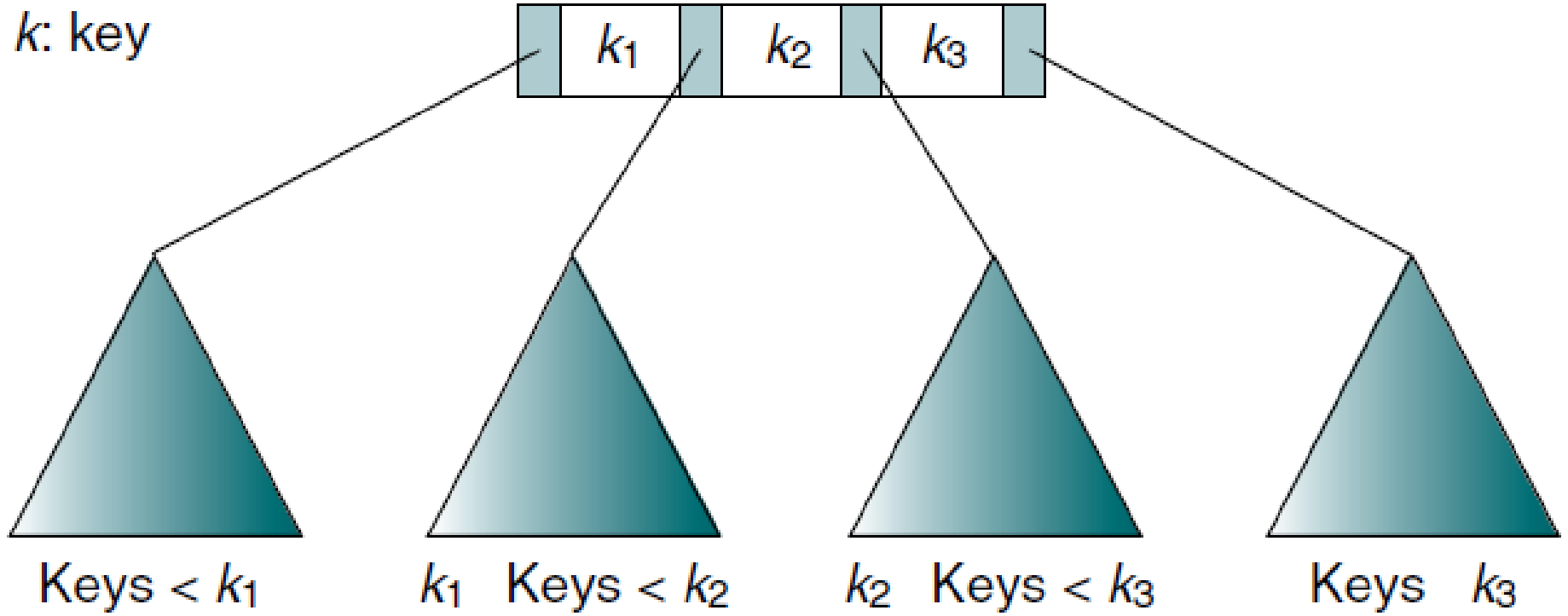
- In an M-way search tree, it is not compulsory that every node has exactly M–1 values and M subtrees.
- The node can have anywhere from 1 to M–1 values, and the number of sub-trees can vary from 0 (for a leaf node) to i + 1, where i is the number of key values in the node.
- M is thus a fixed upper limit that defines how many key values can be stored in the node

Multiway Tree

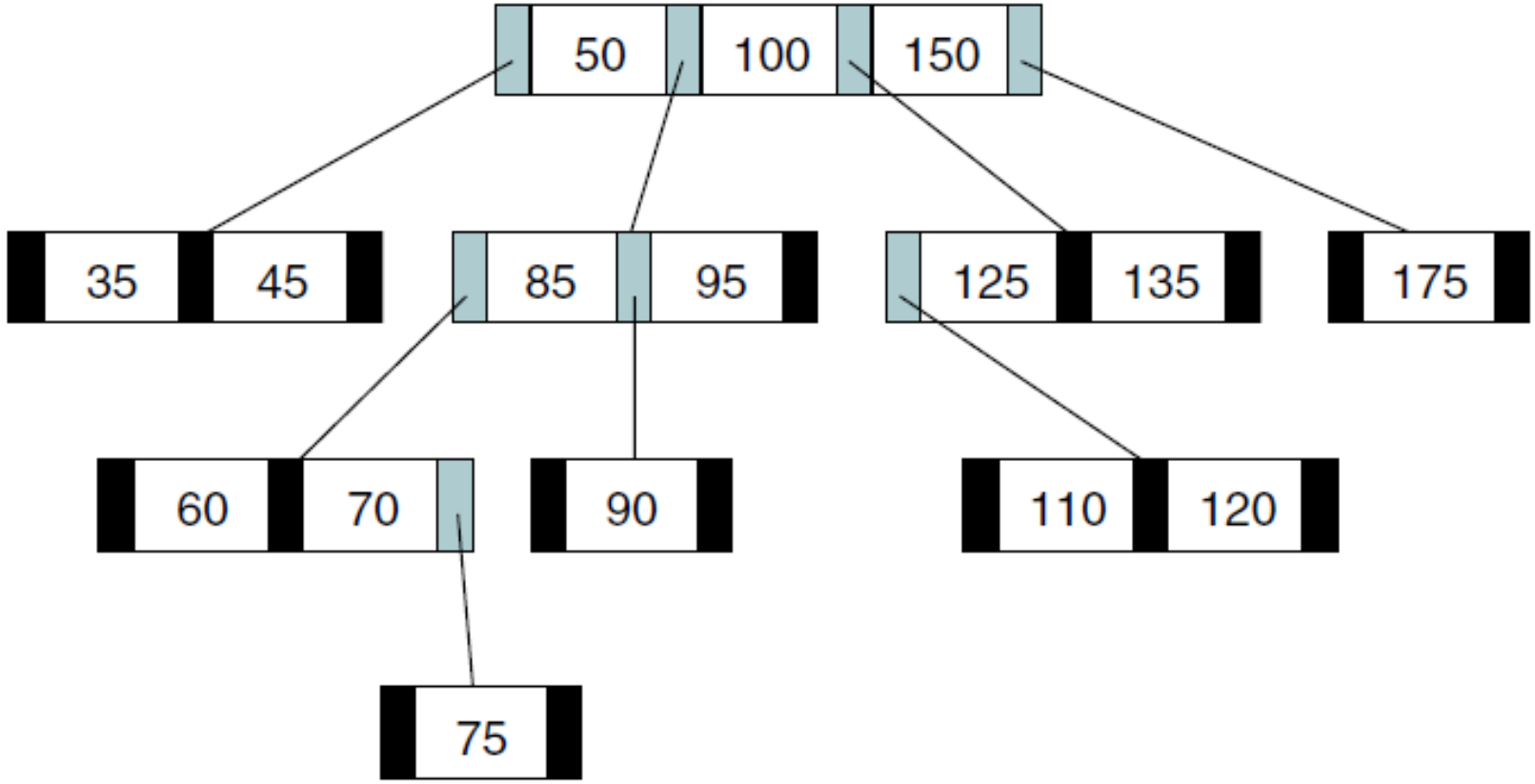
- Consider the M-way search tree shown in Fig. where, M = 3. So a node can store a maximum of two key values and can contain pointers to three sub-trees.
- In our example, we have taken a very small value of M so that the concept becomes easier, but in practice, M is usually very large.
- Using a 3-way search tree, let us lay down some of the basic properties of an M-way search tree.



Multiway Tree



4way Tree



Basic properties of an M-way search tree

- Note that the key values in the sub-tree pointed by P0 are less than the key value K0.
- Similarly, all the key values in the sub-tree pointed by P1 are less than K1, so on and so forth.
- Thus, the generalized rule is that all the key values in the sub-tree pointed by Pi are less than Ki, where $0 \leq i \leq n-1$.
- Note that the key values in the sub-tree pointed by P1 are greater than the key value K0.
- Similarly, all the key values in the sub-tree pointed by P2 are greater than K1, so on and so forth.
- Thus, the generalized rule is that all the key values in the sub-tree pointed by Pi are greater than Ki–1, where $0 \leq i \leq n-1$.
- In an M-way search tree, every sub-tree is also an M-way search tree and follows the same rules.

B Tree

- B tree is a specialized M-way tree.
- B tree of order m can have a maximum of m–1 keys and m pointers to its sub-trees.
- Storing a large number of keys in a single node keeps the height of the tree relatively small.
- B tree is designed to store sorted data and allows search, insertion, and deletion operations to be performed.
- B tree of order m (the maximum number of children that each node can have) is a tree with all the properties of an M-way search tree.

B Tree properties

1. Every node in the B tree has at most (maximum) m children.

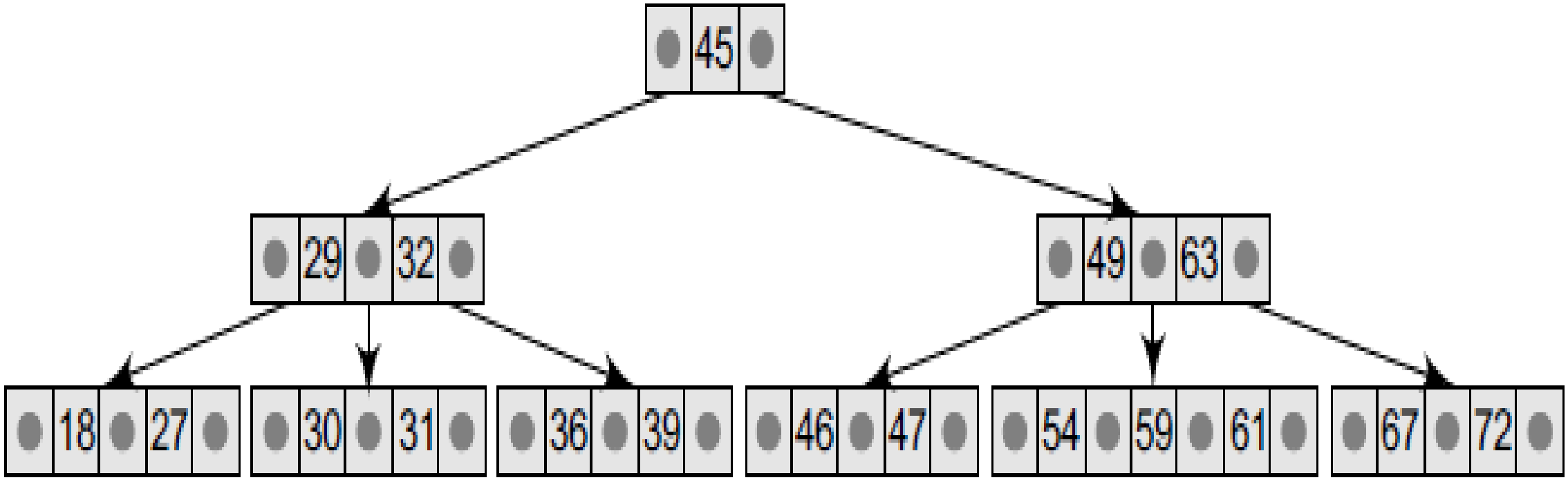
2. Every node in the B tree except the root node and leaf nodes has at least (minimum) m/2 children. This condition helps to keep the tree bushy so that the path from the root node to the leaf is very short, even in a tree that stores a lot of data.

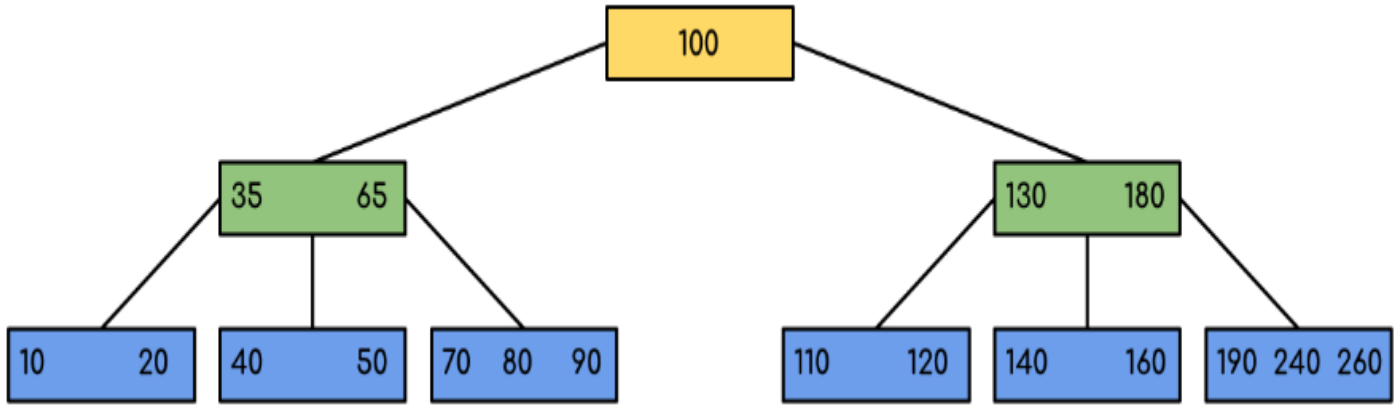
3. The root node has at least two children if it is not a terminal (leaf) node.

4. All leaf nodes are at the same level.

- An internal node in the B tree can have n number of children, where $0 \leq n \leq m$. It is not necessary that every node has the same number of children, but the only restriction is that the node should have at least m/2 children.

B Tree of order 4



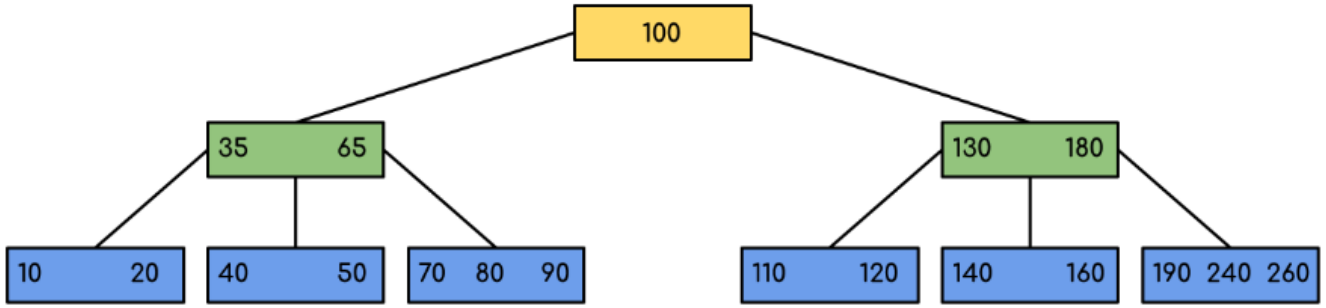


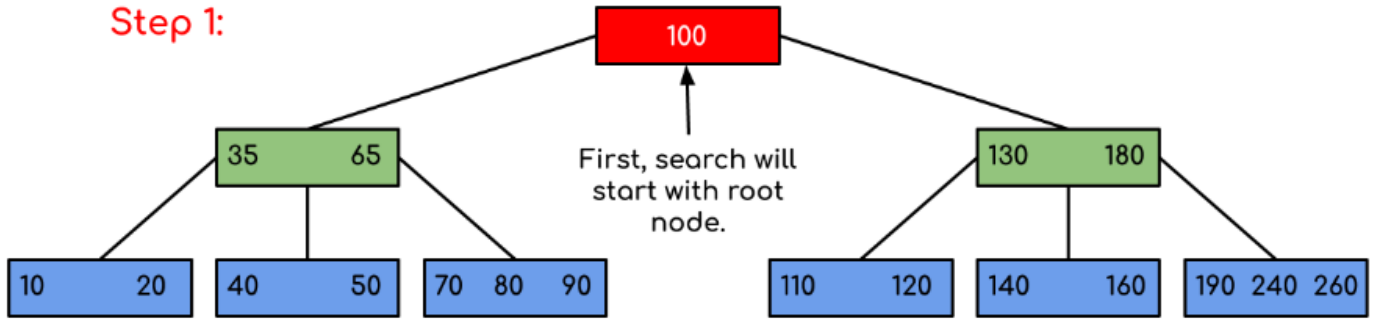
Searching for an Element in a B Tree

- Searching for an element in a B tree is similar to that in binary search trees. Consider the B tree given in Fig. To search for 59, we begin at the root node.
- The root node has a value 45 which is less than 59. So, we traverse in the right sub-tree. The right sub-tree of the root node has two key values, 49 and 63. Since 49 ≤ 59 ≤ 63, we traverse the right sub-tree of 49, that is, the left sub-tree of 63. This sub-tree has three values, 54, 59, and 61. On finding the value 59, the search is successful.
- Take another example. If you want to search for 9, then we traverse the left sub-tree of the root node. The left sub-tree has two key values, 29 and 32. Again, we traverse the left sub-tree of 29. We find that it has two key values, 18 and 27. There is no left sub-tree of 18, hence the value 9 is not stored in the tree.

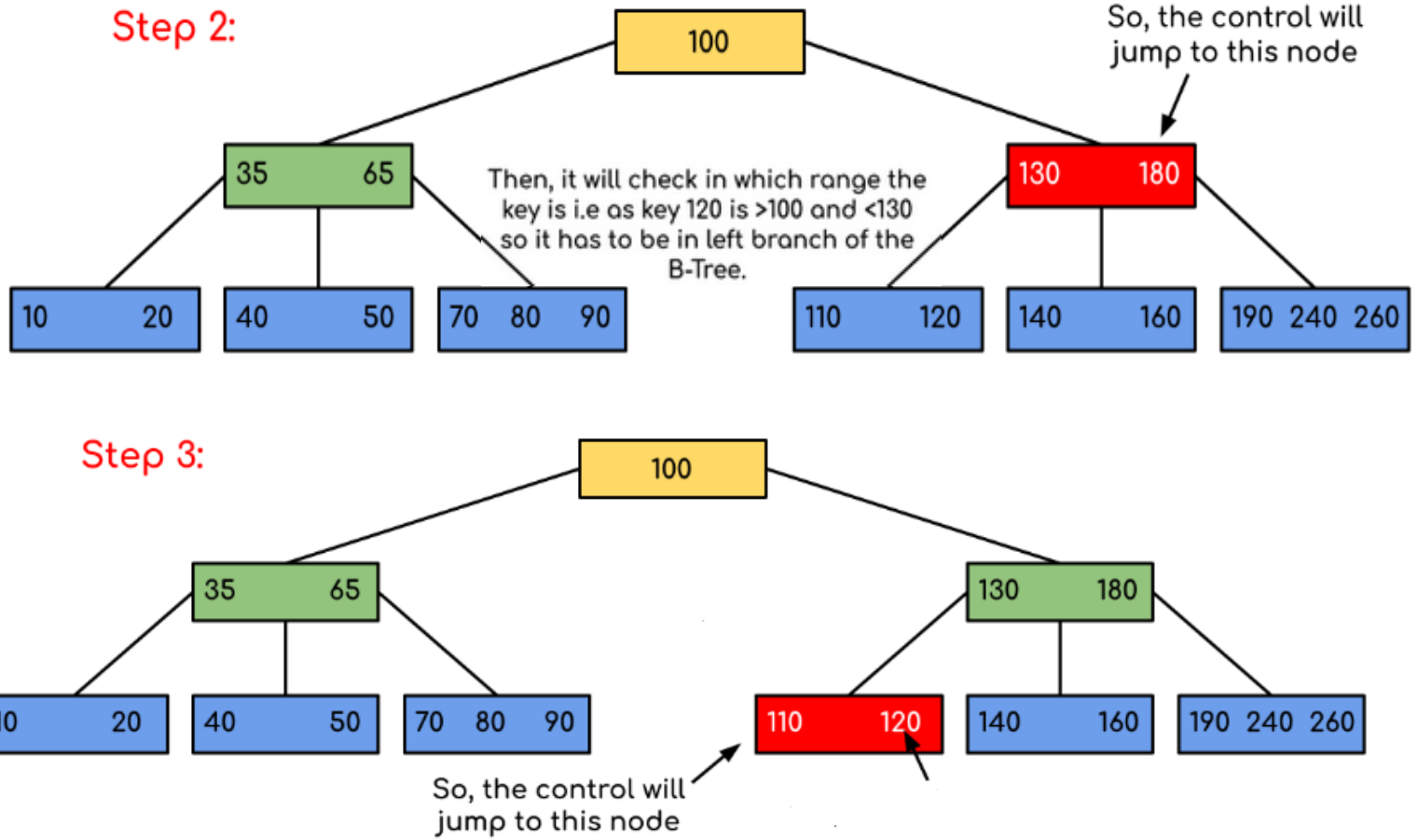
Searching for an Element in a B Tree

- Search 120





Searching for an Element in a B Tree



Inserting a New Element in a B Tree

- In a B tree, all insertions are done at the leaf node level. A new value is inserted in the B tree using the algorithm given below.

1. Search the B tree to find the leaf node where the new key value should be inserted.

2. If the leaf node is not full, that is, it contains less than m–1 key values, then insert the new element in the node keeping the node's elements ordered.

3. If the leaf node is full, that is, the leaf node already contains m–1 key values, then

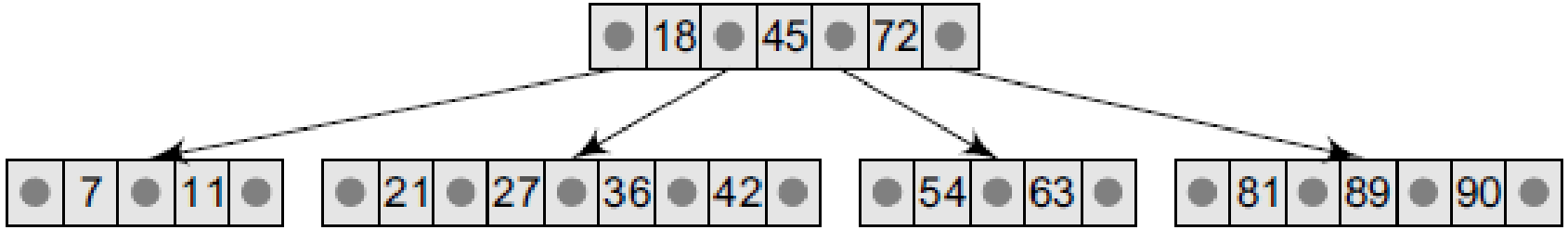
(a) insert the new value in order into the existing set of keys,

(b) split the node at its median into two nodes (note that the split nodes are half full), and

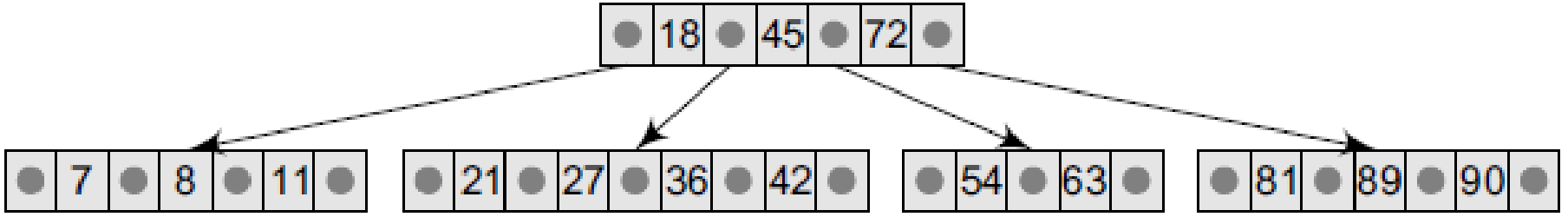
(c) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

Inserting a New Element in a B Tree

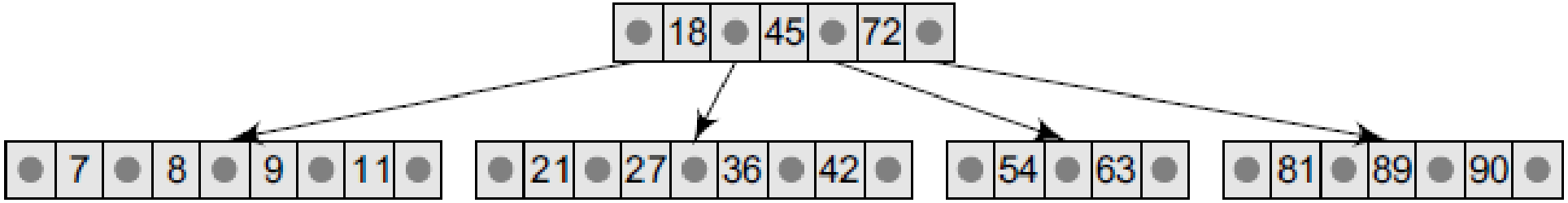
- Look at the B tree of order 5 given below and insert 8, 9, 39, and 4 into it.



Step 1: Insert 8

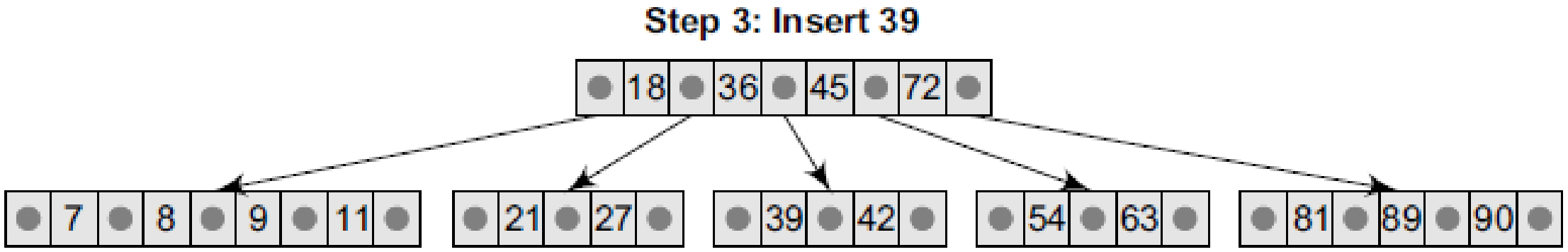


Step 2: Insert 9



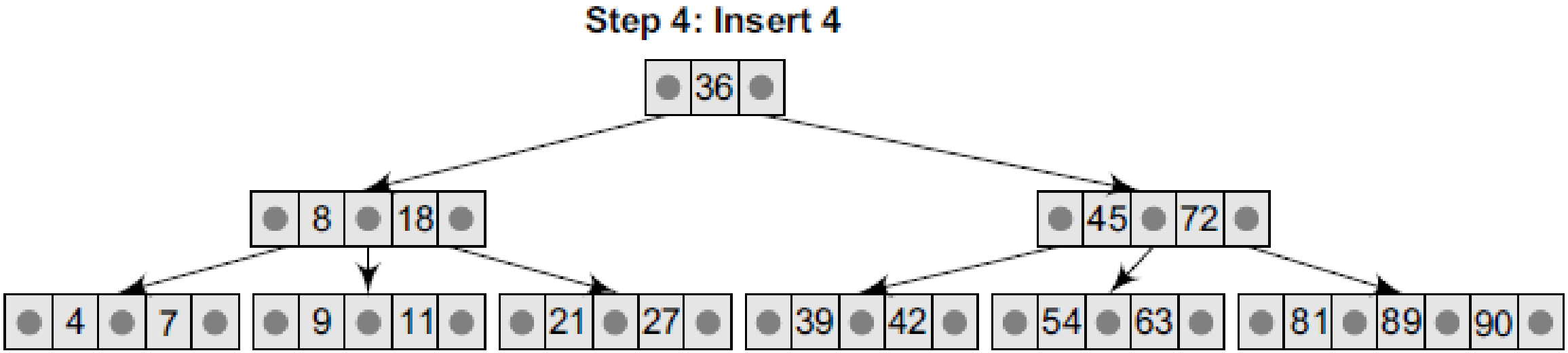
Inserting a New Element in a B Tree

- The node in which 39 should be inserted is already full as it contains four values. Here we split the nodes to form two separate nodes. But before splitting, arrange the key values in order (including the new value). The ordered set of values is given as 21, 27, 36, 39, and 42.
- The median value is 36, so push 36 into its parent's node and split the leaf nodes.



Inserting a New Element in a B Tree

- The ordered set of values is given as 4, 7, 8, 9, and 11. The median value is 8, so we push 8 into its parent's node and split the leaf nodes. But again, we see that the parent's node is already full, so we split the parent node using the same procedure.



Inserting a New Element in a B Tree

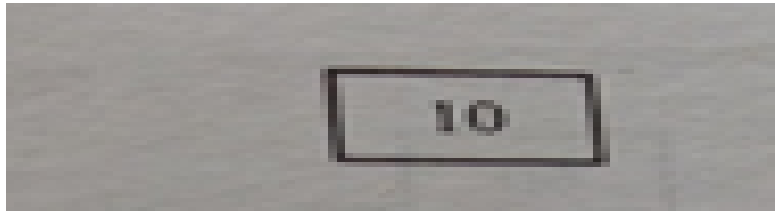
- Create a B Tree of Order 5

List of Keys

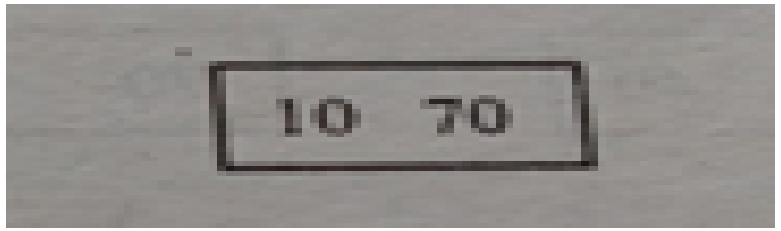
10, 70, 60, 20, 110, 40,80,130,100,50,190,90,180,240,30,120,140,160

Inserting a New Element in a B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

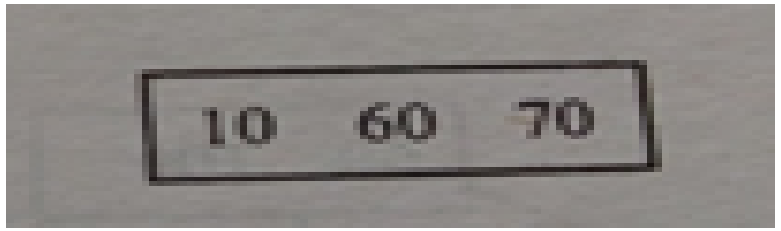


Insert 10



Insert 70

After Inserting 70, the keys in the node will be sorted

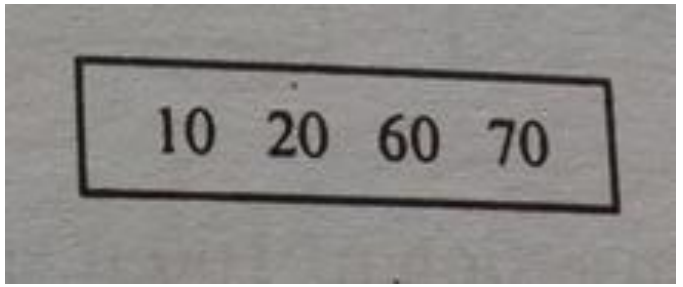


Insert 60

After Inserting 60, the keys in the node will be sorted

Inserting a New Element in a B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160



Insert 20

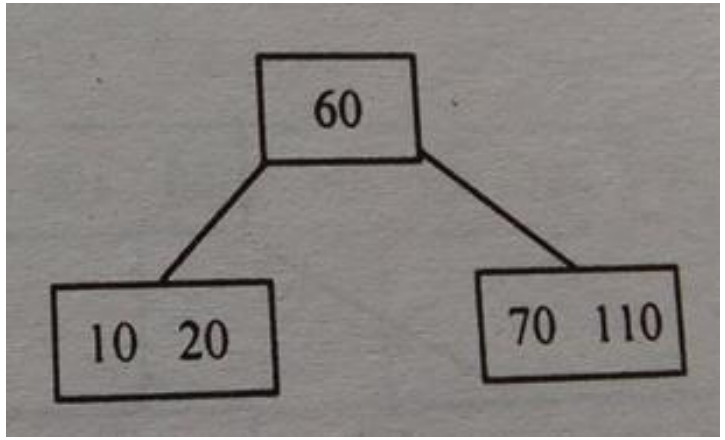
After Inserting 20, the keys in the node will be sorted

Insert 110

Node was already full,

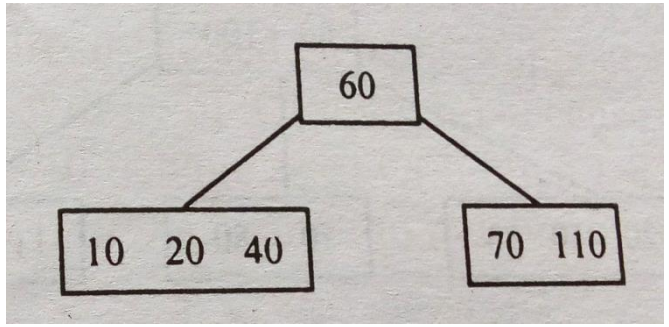
After insertion of 110 , It splits into 2 nodes

60 is the median key, so it goes to parent or becomes root



Inserting a New Element in a B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

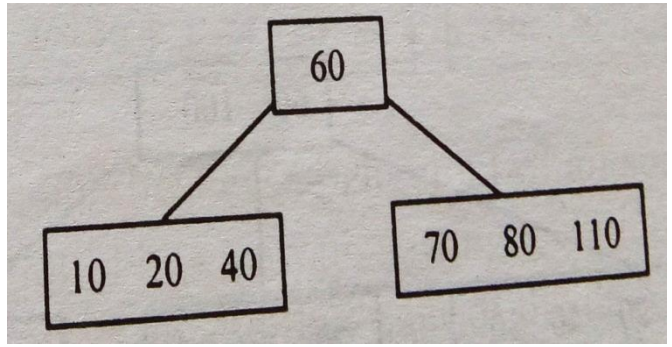


Insert 40

After Inserting 40, the keys in the node will be sorted

Insert 80

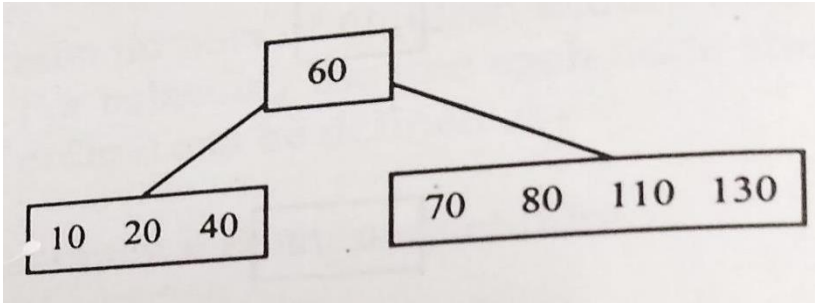
After Inserting 80, the keys in the node will be sorted



Inserting a New Element in a B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

Insert 130

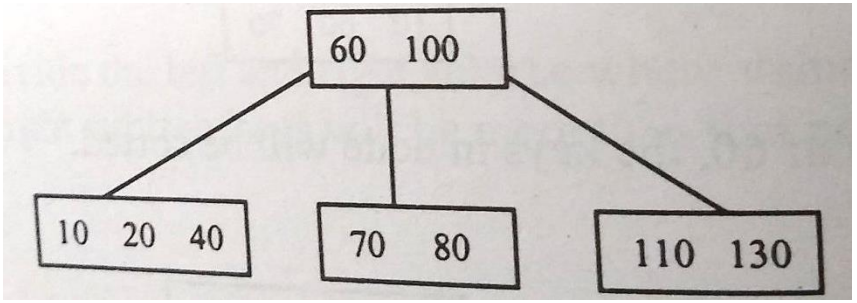


Insert 100

Node was already full

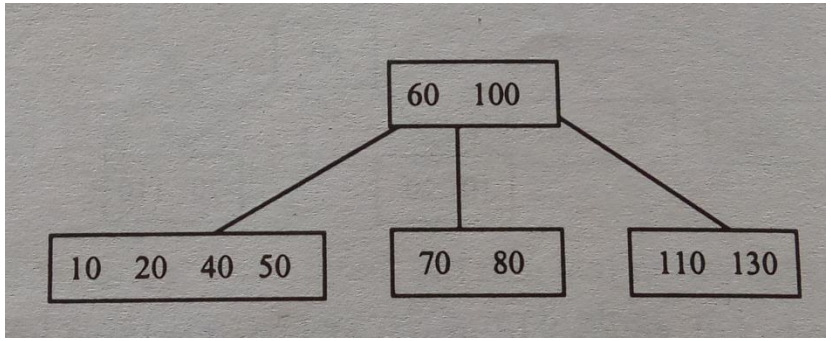
After insertion of 100, it splits in 2 nodes, 100 is the median key ,

100 goes up to the parent node



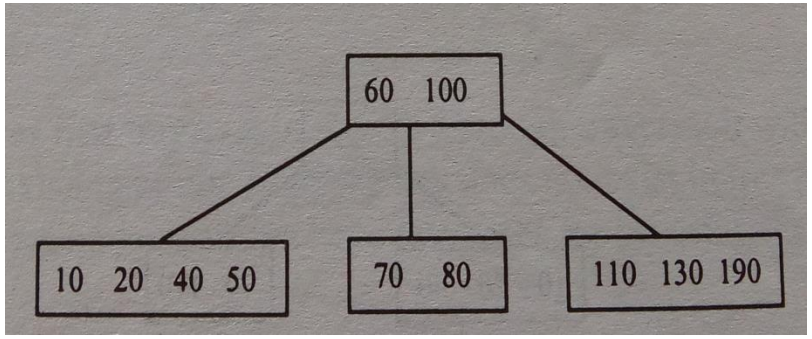
Inserting a New Element in a B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160



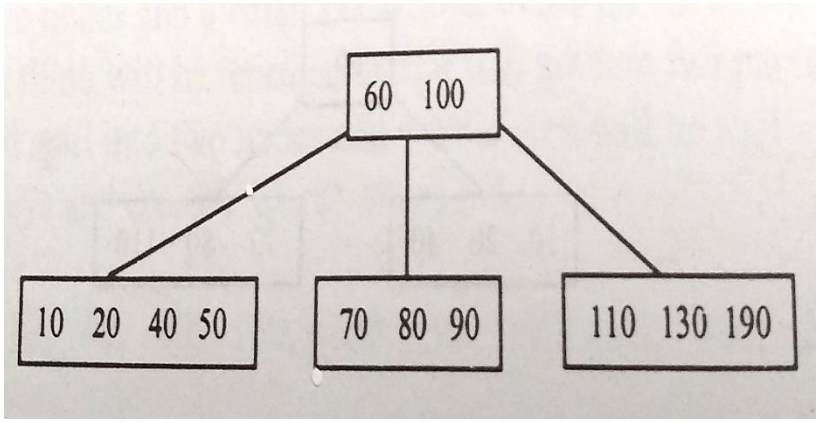
Insert 50

Insert 190



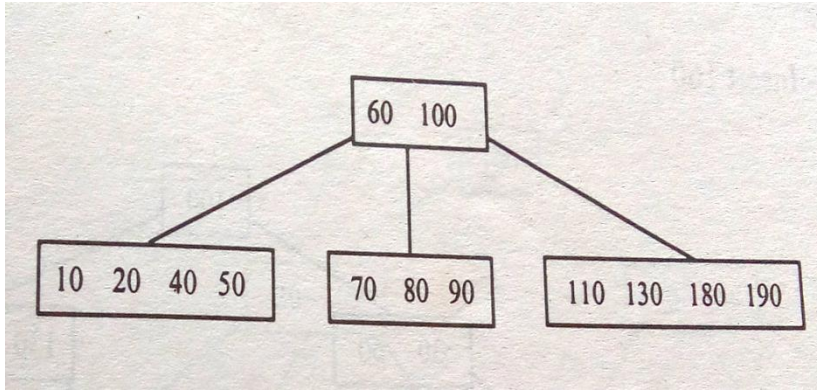
Inserting a New Element in a B Tree

List of
Keys=10,70,60,20,110,40,80,13
0,100,50,190,90,180,240,30,12
0,140,160



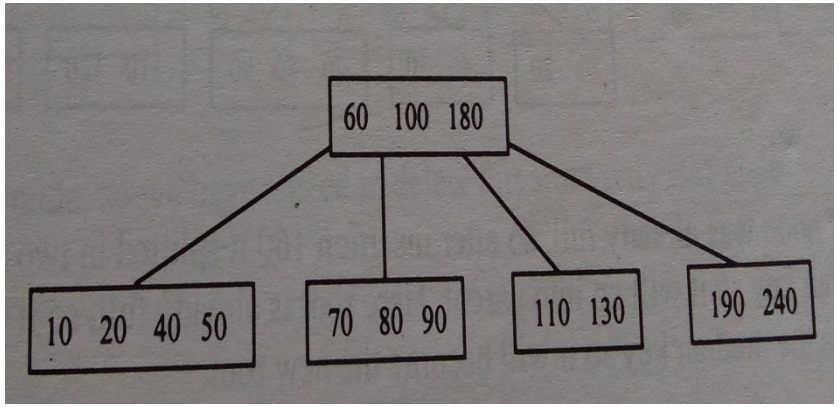
Insert 90

Insert 180



Inserting a New Element in a B Tree

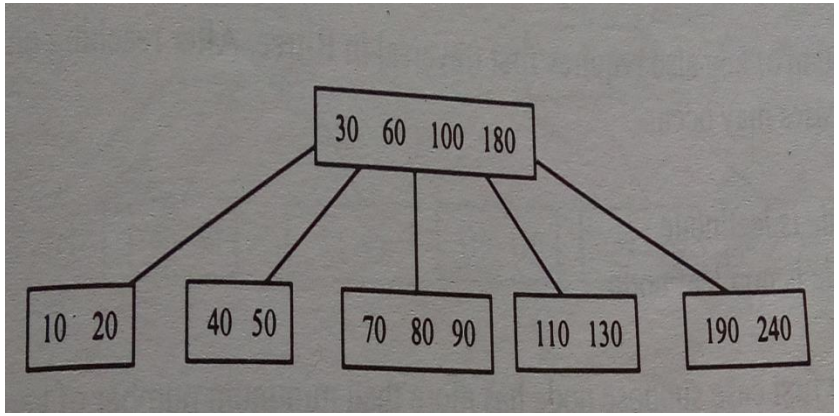
List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160



Insert 240

Insert 30

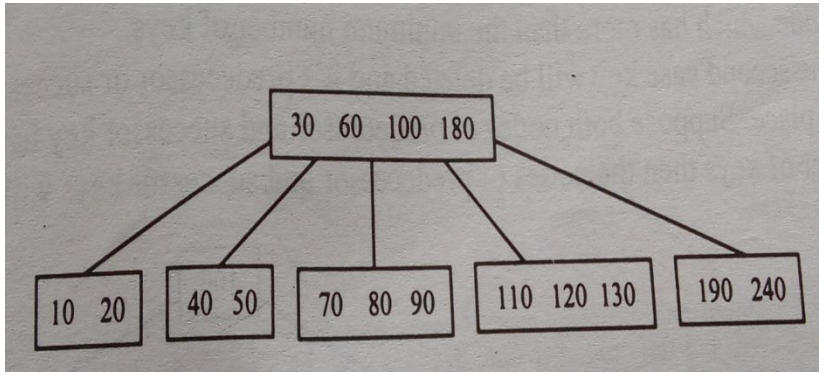
Node was already full, so after insertion of 30, splits in 2 nodes, 30 is the median key so it will go to the parent



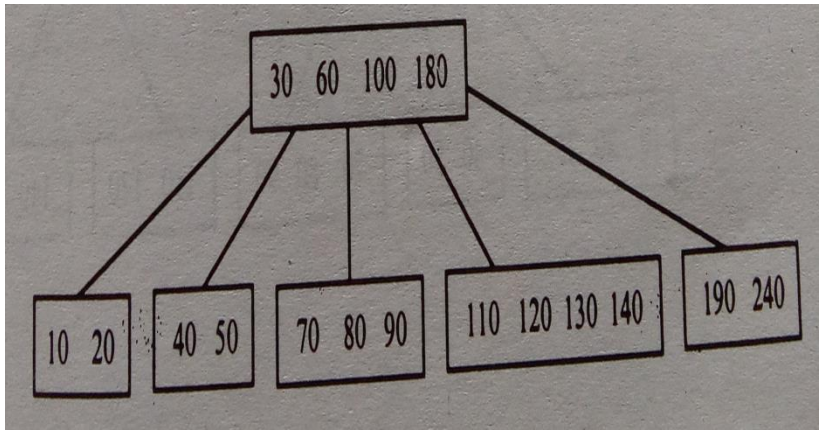
Inserting a New Element in a B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

Insert 120



Insert 140



Inserting a New Element in a B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

Insert 160

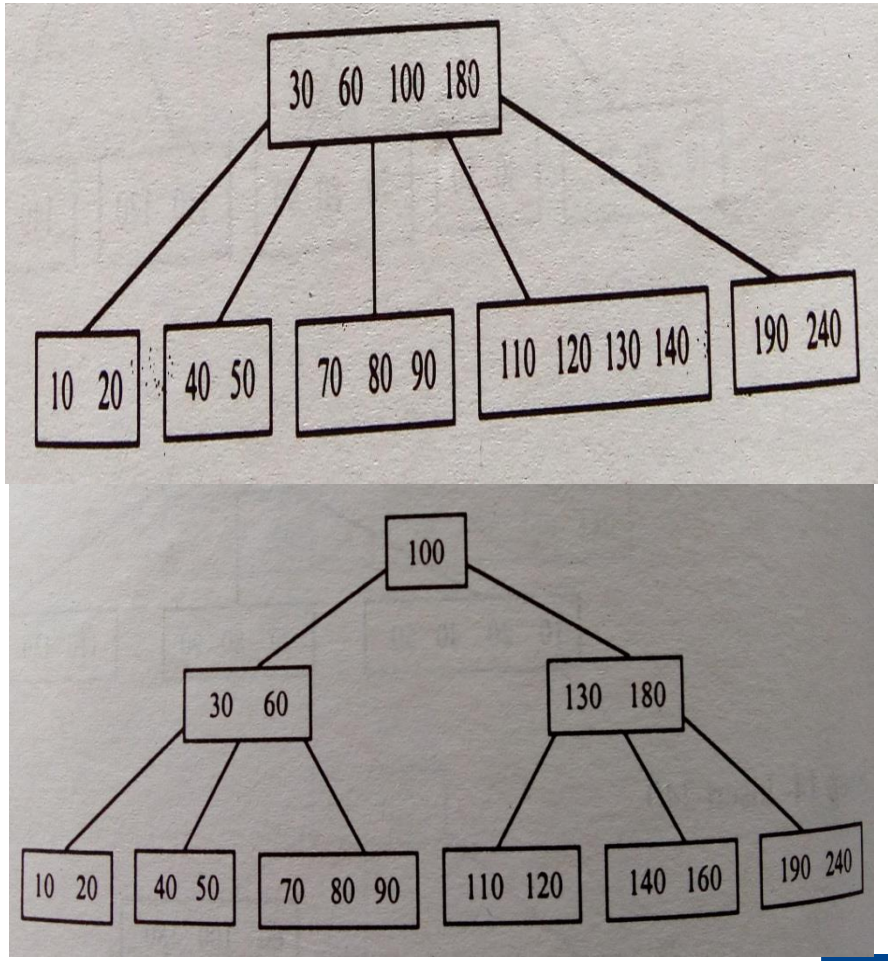
Node was already full,

After insertion of 160

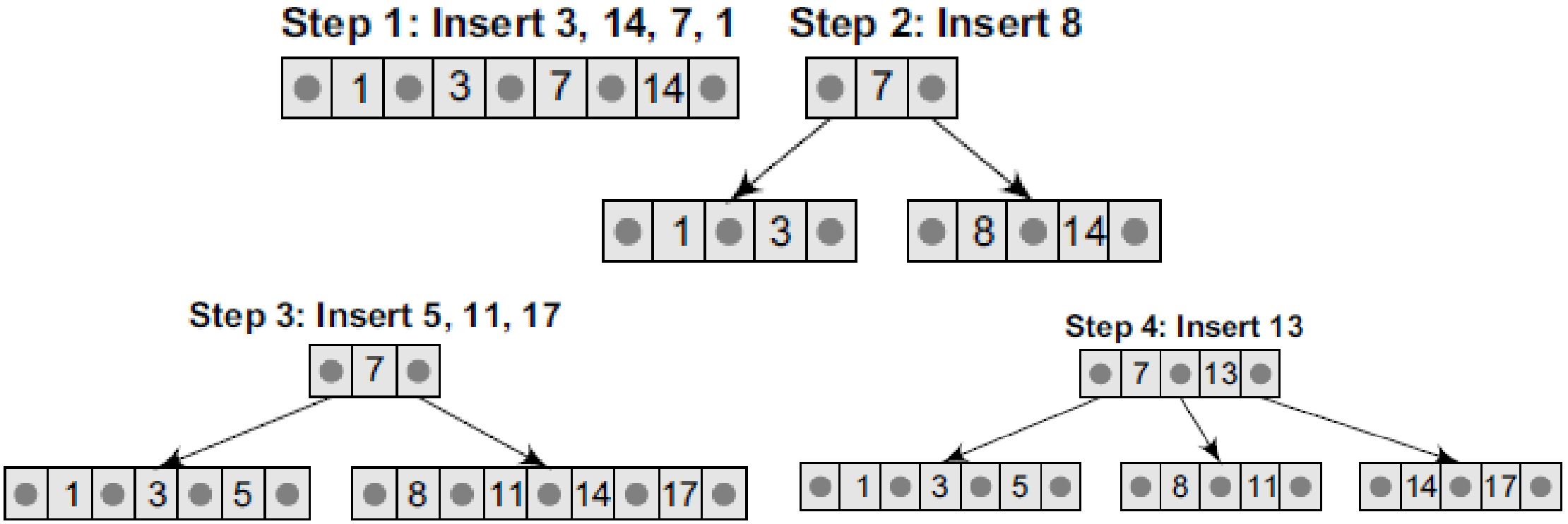
Splits into 2 nodes

130 is the median so it goes up

Root is already full, so it splits in 2 nodes , 100 is the median so it becomes new root

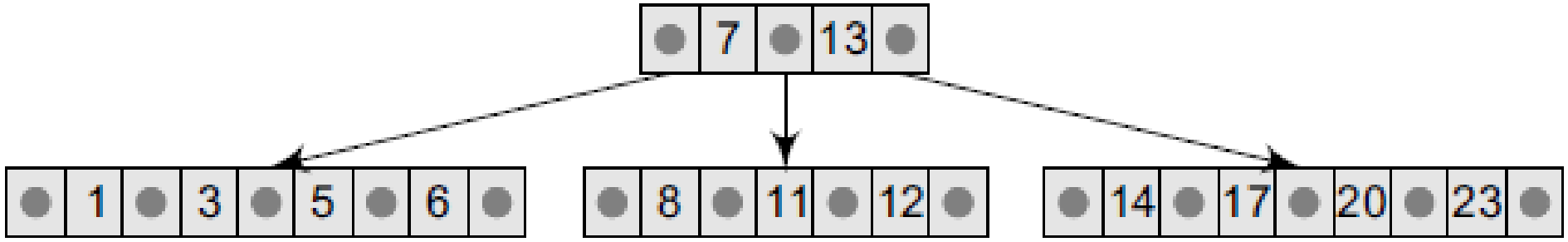


Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25,19.

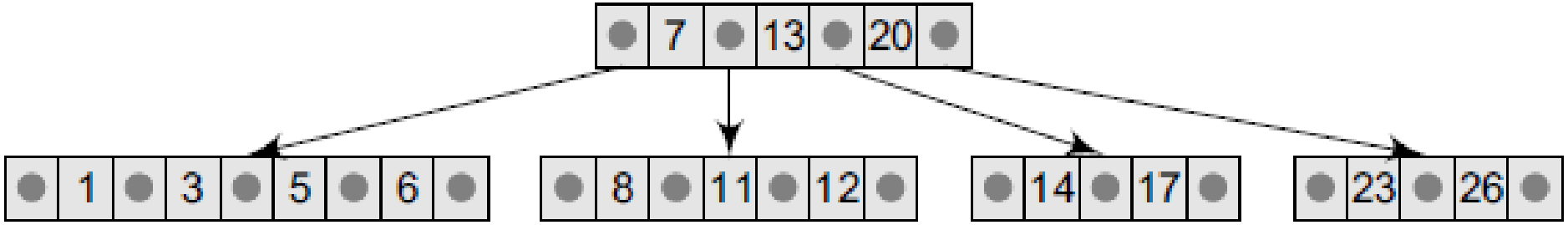


Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.

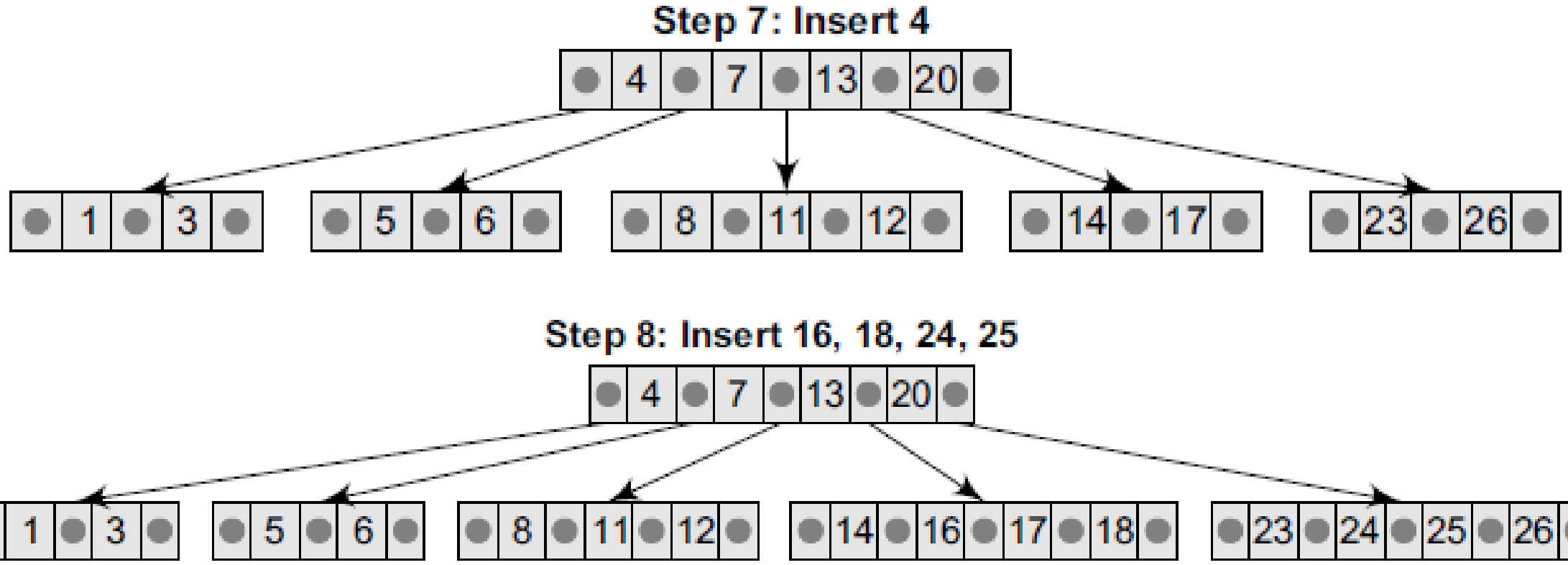
Step 5: Insert 6, 23, 12, 20



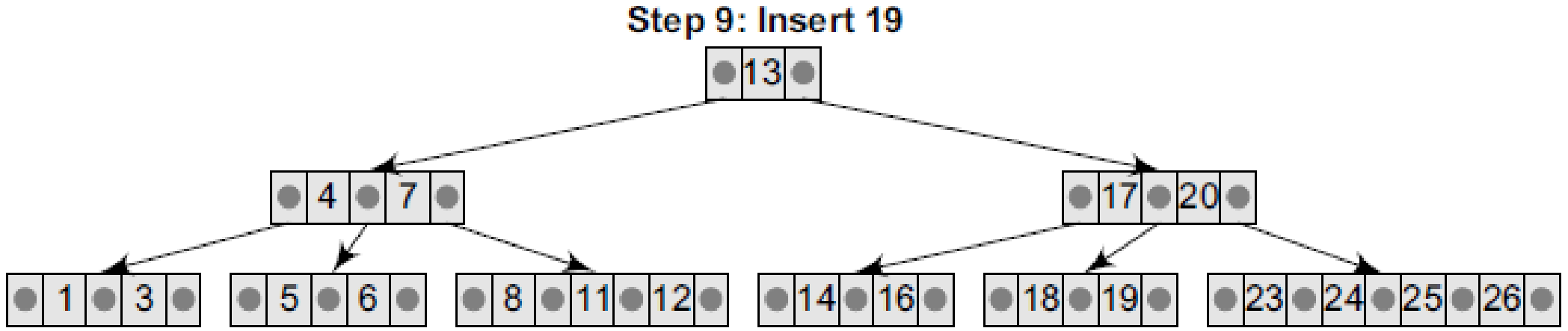
Step 6: Insert 26



Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.



Create a B tree of order 5 by inserting the following elements: 3, 14, 7, 1, 8, 5, 11, 17, 13, 6, 23, 12, 20, 26, 4, 16, 18, 24, 25, and 19.



Deleting an Element from a B Tree

- Deletion is also done from the leaf nodes.
- There are two cases of deletion.
- First case, a leaf node has to be deleted.
- Second case, an internal node has to be deleted.

Deleting an Element from a B Tree

1. Locate the leaf node which has to be deleted.

2. If the leaf node contains the minimum number of key values (m/2 elements), then delete the value.

3. Else if the leaf node does not contain m/2 elements, then fill the node by taking an element either from the left or from the right sibling.

(a) If the left sibling has more than the minimum number of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

(b) Else, if the right sibling has more than the minimum number of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

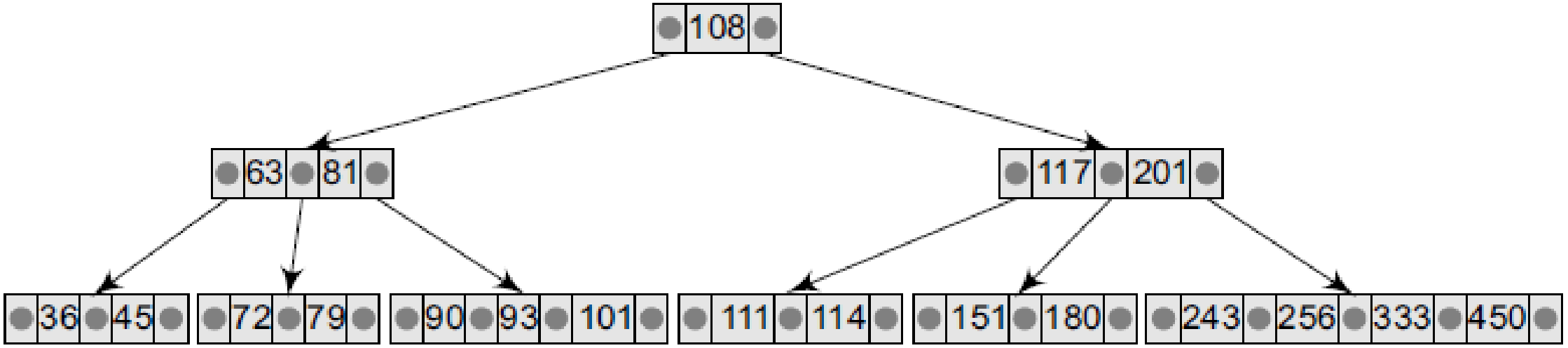
Deleting an Element from a B Tree

4.Else, if both left and right siblings contain only the minimum number of elements, then create a new leaf node by combining the two leaf nodes and the intervening element of the parent node (ensuring that the number of elements does not exceed the maximum number of elements a node can have, that is, m). If pulling the intervening element from the parent node leaves it with less than the minimum number of keys in the node, then propagate the process upwards, thereby reducing the height of the B tree.

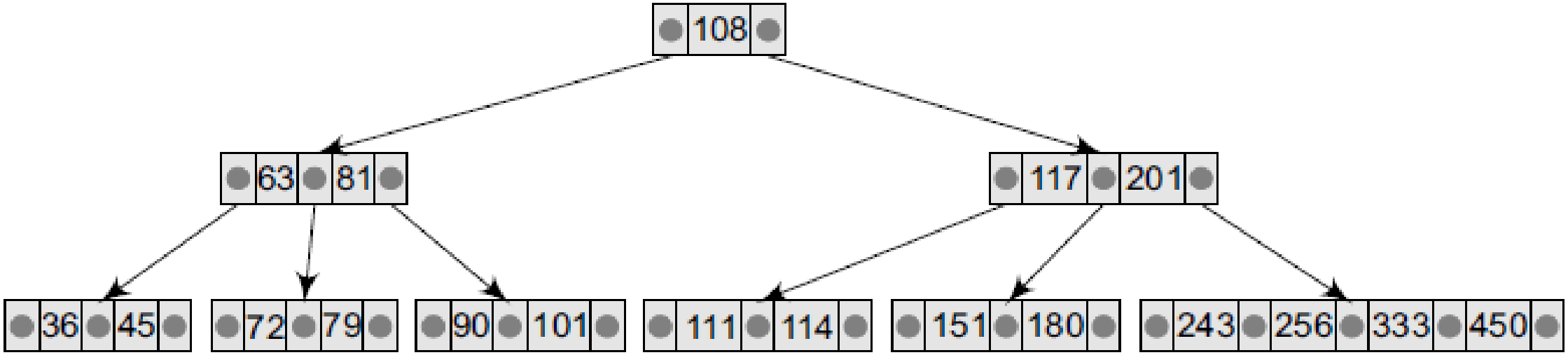
- To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of the deleted key. This predecessor or successor will always be in the leaf node. So the processing will be done as if a value from the leaf node has been deleted.

Deleting an Element from a B Tree

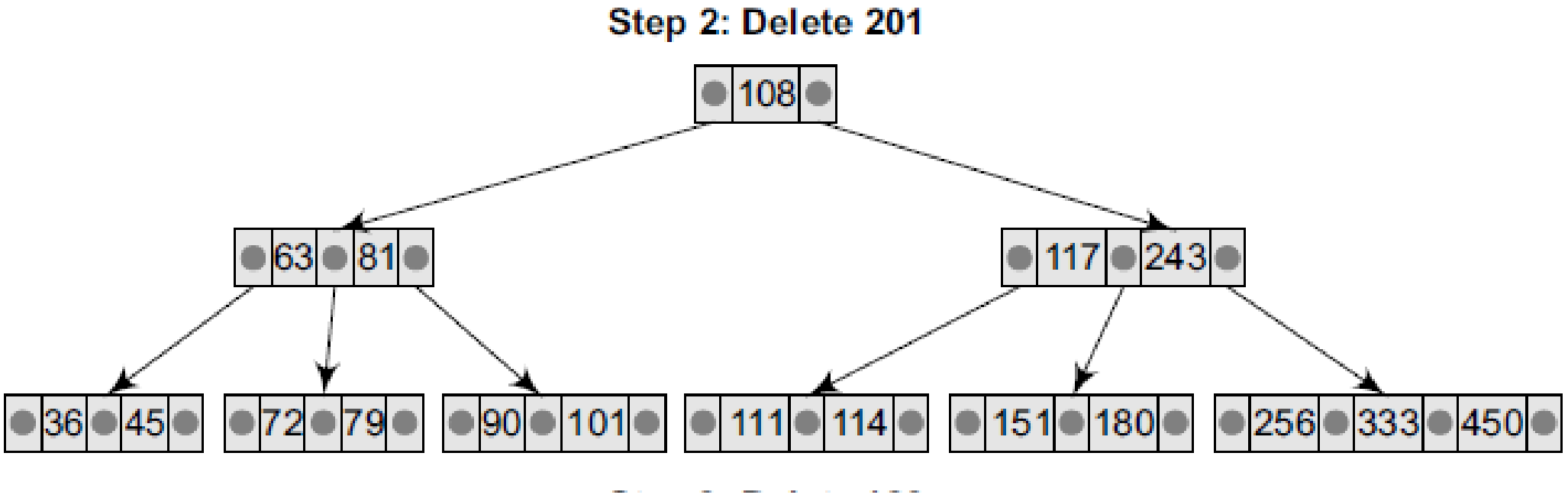
Consider the B tree of order 5 and delete values 93, 201, 180, and 72



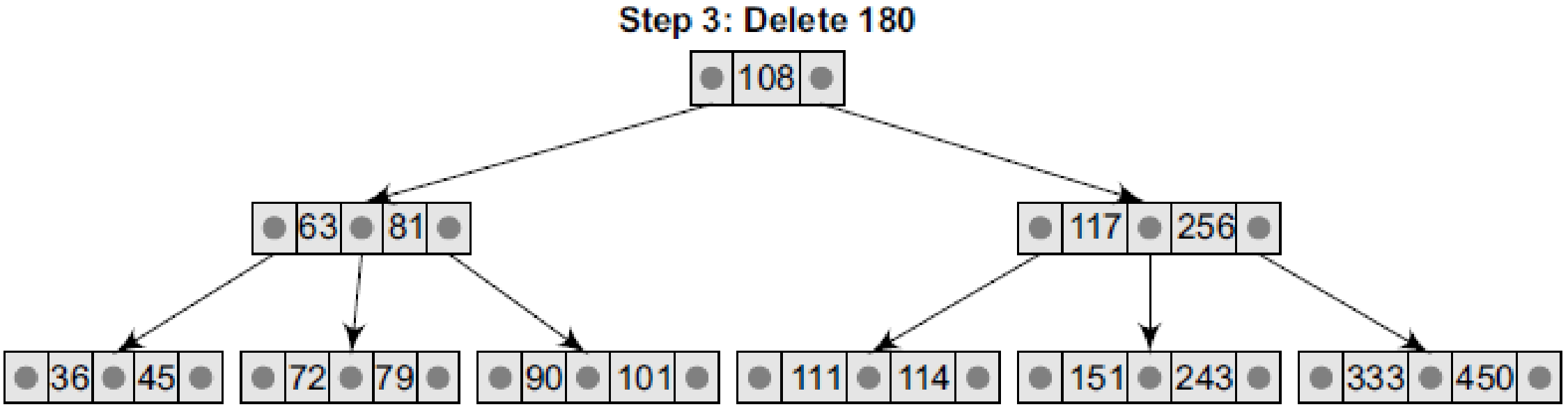
Step 1: Delete 93



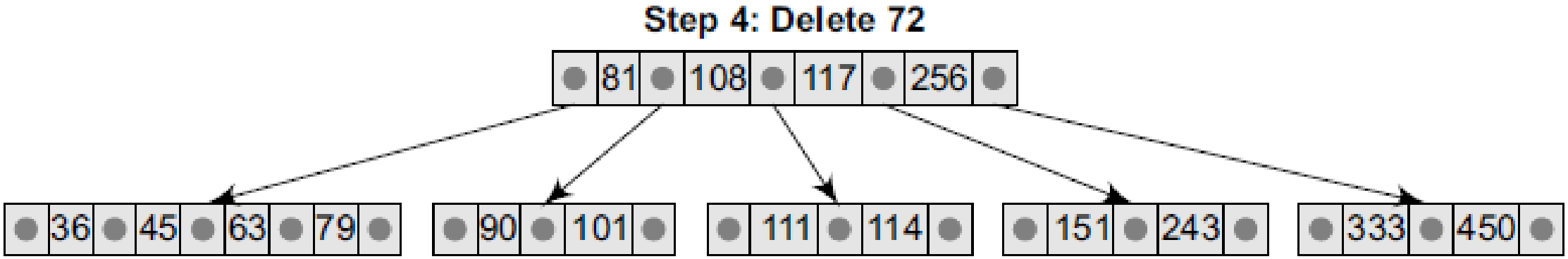
Deleting an Element from a B Tree



Deleting an Element from a B Tree

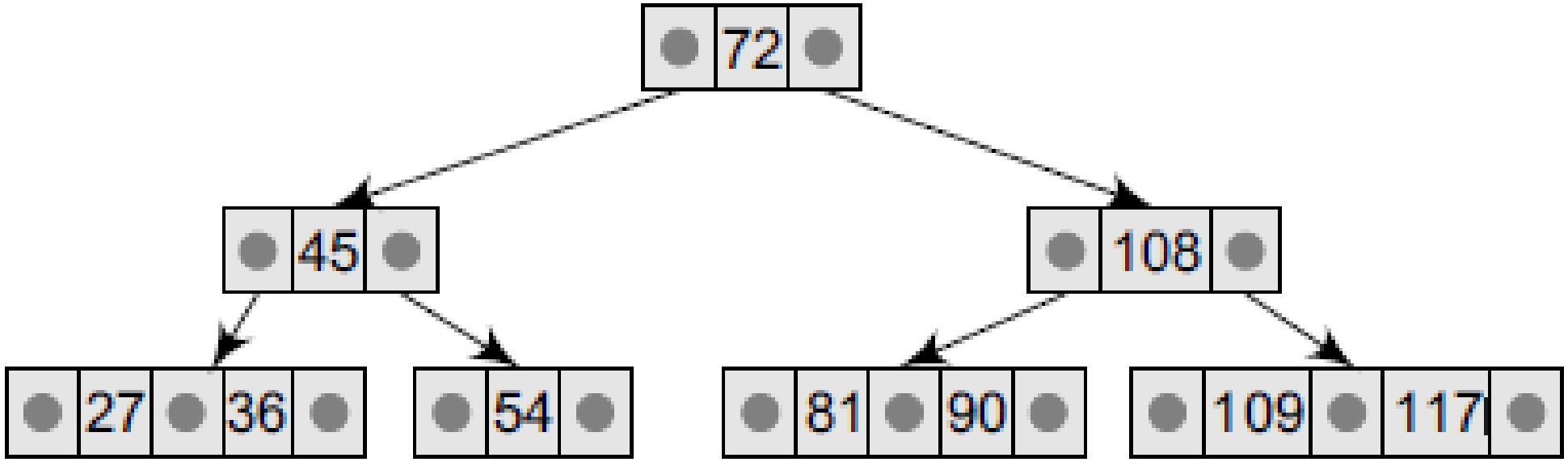


Deleting an Element from a B Tree

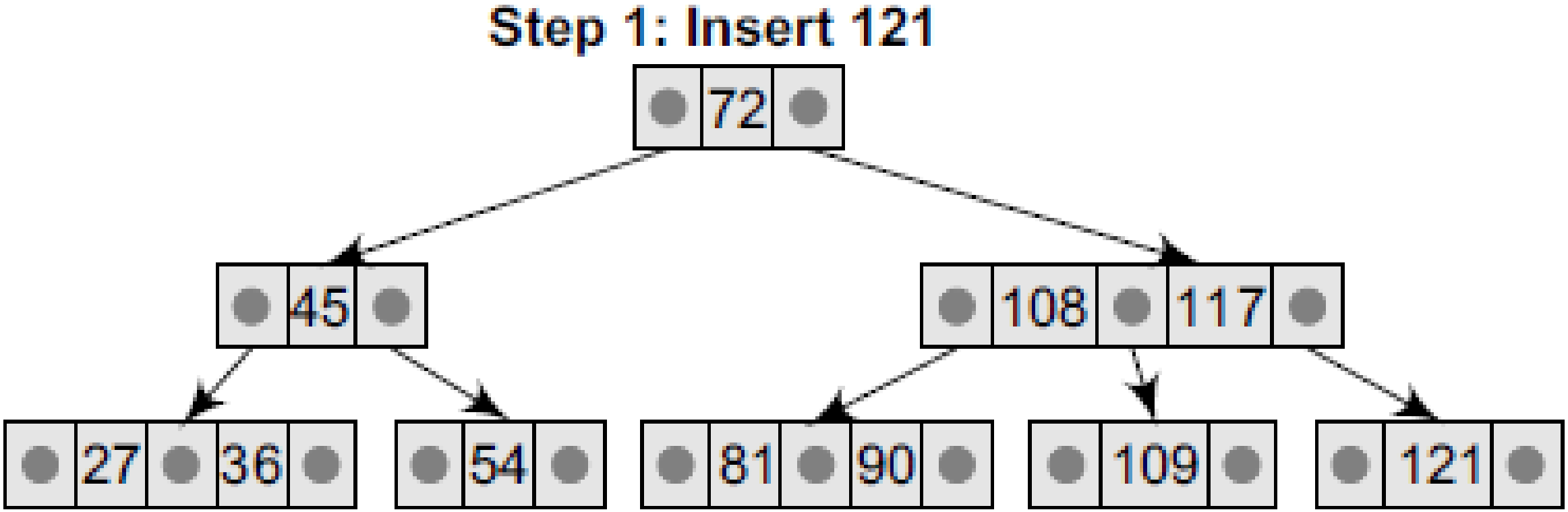


Insert and delete an element from a B Tree

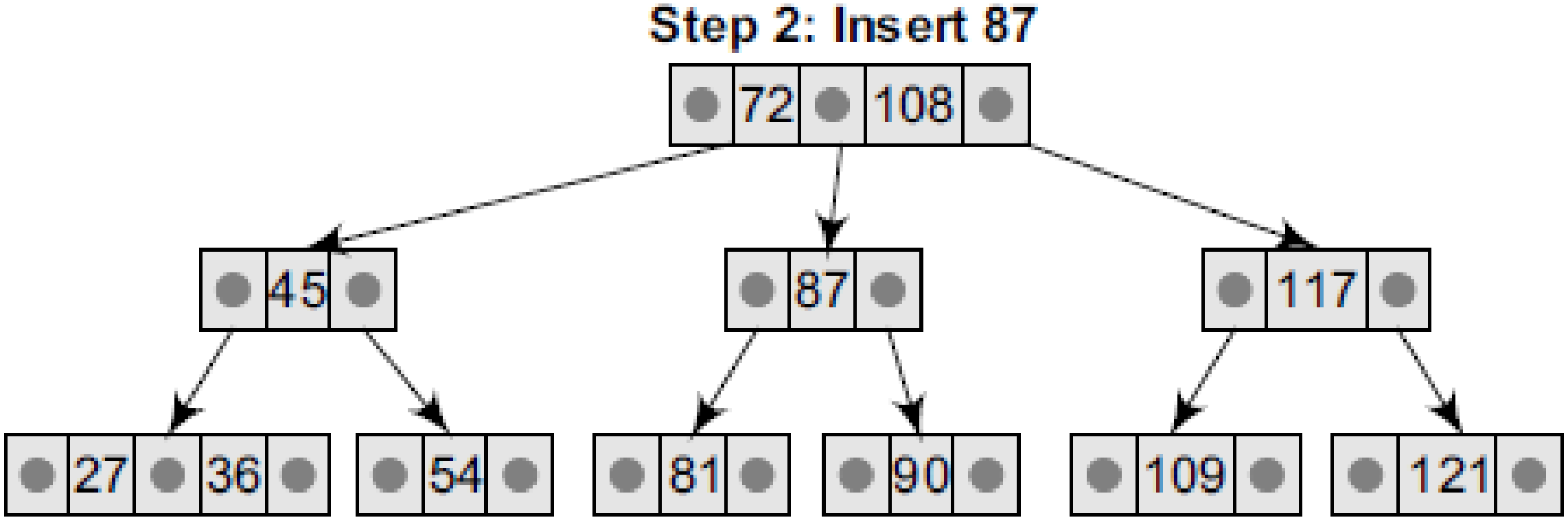
- Consider the B tree of order 3 given below and perform the following operations: (a) insert 121, 87 and then (b) delete 36



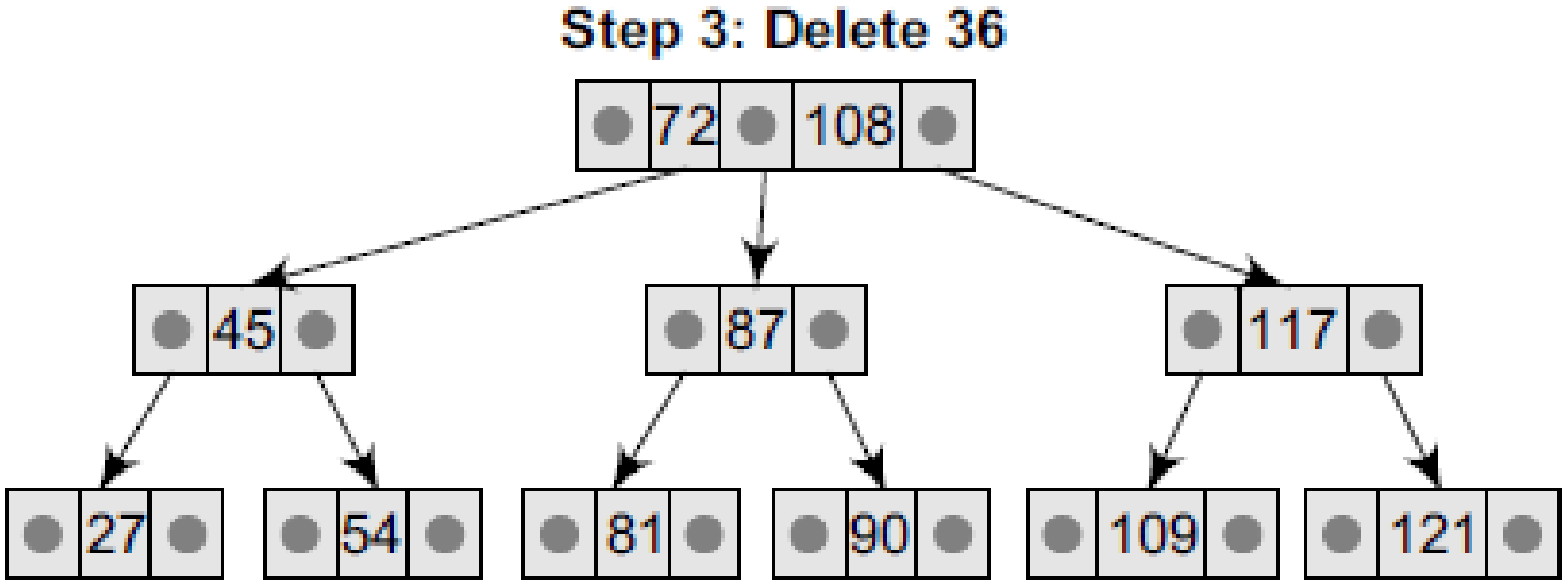
Insert and delete an element from a B Tree



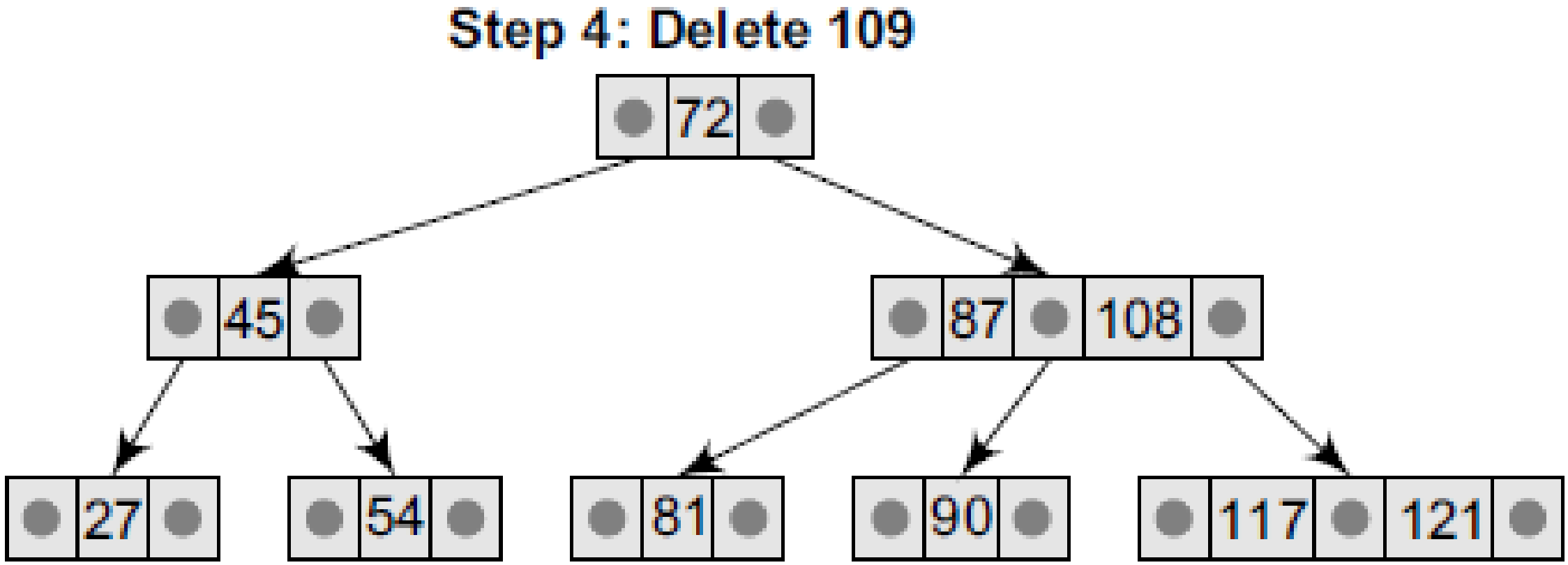
Insert and delete an element from a B Tree



Insert and delete an element from a B Tree



Insert and delete an element from a B Tree



Application of B tree

- We take a large value of m mainly because of three reasons:

1. Disk access is very slow. We should be able to fetch a large amount of data in one disk access.

2. Disk is a block-oriented device. That is, data is organized and retrieved in terms of blocks. So while using a B tree (generalized M-way search tree), a large value of m is used so that one single node of the tree can occupy the entire block. In other words, m represents the maximum number of data items that can be stored in a single block. m is maximized to speed up processing. More the data stored in a block, lesser the time needed to move it into the main memory.

3. A large value minimizes the height of the tree. So, search operation becomes really fast.

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

B+ Trees

- What are B+ Trees used for
- What is a B Tree
- What is a B+ Tree
- Searching
- Insertion
- Deletion

Somaiya TRUST

What are B+ Trees Used For?

- When we store data in a table in a DBMS we want
  - Fast lookup by primary key
    - Just this – hashtable O(c)
  - Ability to add/remove records on the fly
    - Some kind of dynamic tree **on disk**
  - Sequential access to records (physically sorted by primary key on disk)
    - Tree structured keys (hierarchical index for searching)
    - Records all at leaves in sorted order

B+ Tree?

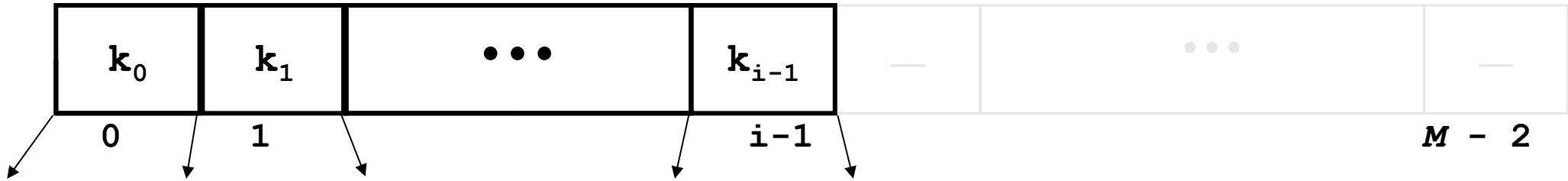
- A variation of B trees in which
  - internal nodes contain only search keys (no data)
  - Leaf nodes contain pointers to data records
  - Data records are in sorted order by the search key
  - All leaves are at the same depth

Definition of a B+Tree

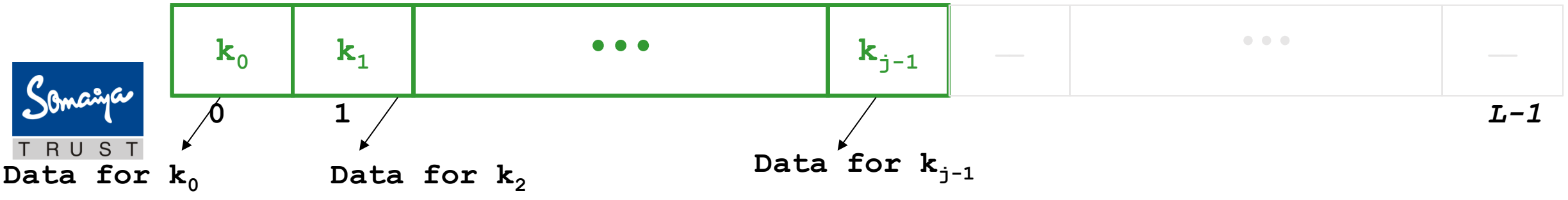
A B+ tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length, and each non-leaf node of the tree has between [M/2] and [M] children, where n is fixed for a particular tree.

B+ Tree Nodes

- Internal node
  - Pointer (Key, NodePointer)*M-1 in each node
  - First i keys are currently in use



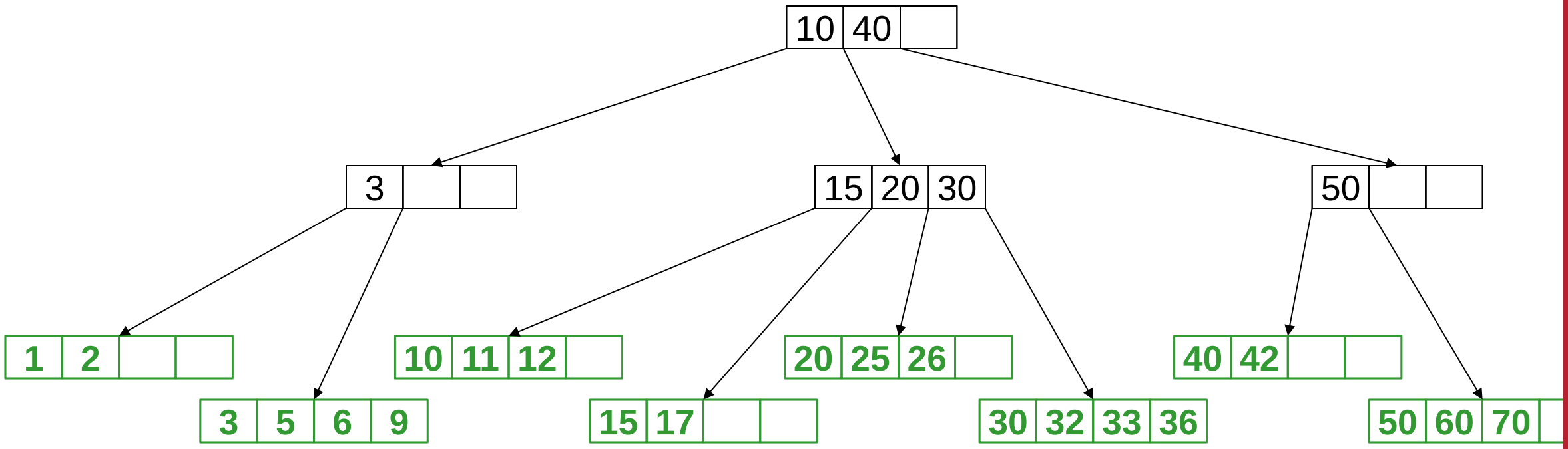
- Leaf
  - (Key, DataPointer)* L in each node
  - first j Keys currently in use



Example

B+ Tree with $M = 4$

Often, leaf nodes linked together



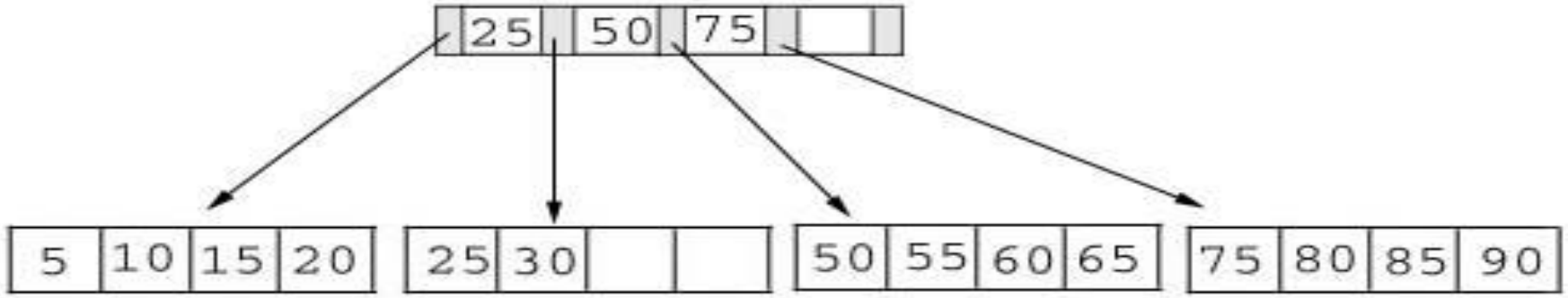
Advantages of B+ tree usage for databases

- keeps keys in sorted order for sequential traversing
- uses a hierarchical index to minimize the number of disk reads
- uses partially full blocks to speed insertions and deletions
- keeps the index balanced with a recursive algorithm
- In addition, a B+ tree minimizes waste by making sure the interior nodes are at least half full. A B+ tree can handle an arbitrary number of insertions and deletions.

Searching

- Just compare the key value with the data in the tree, then return the result.

  For example: find the value 45, and 15 in below tree.

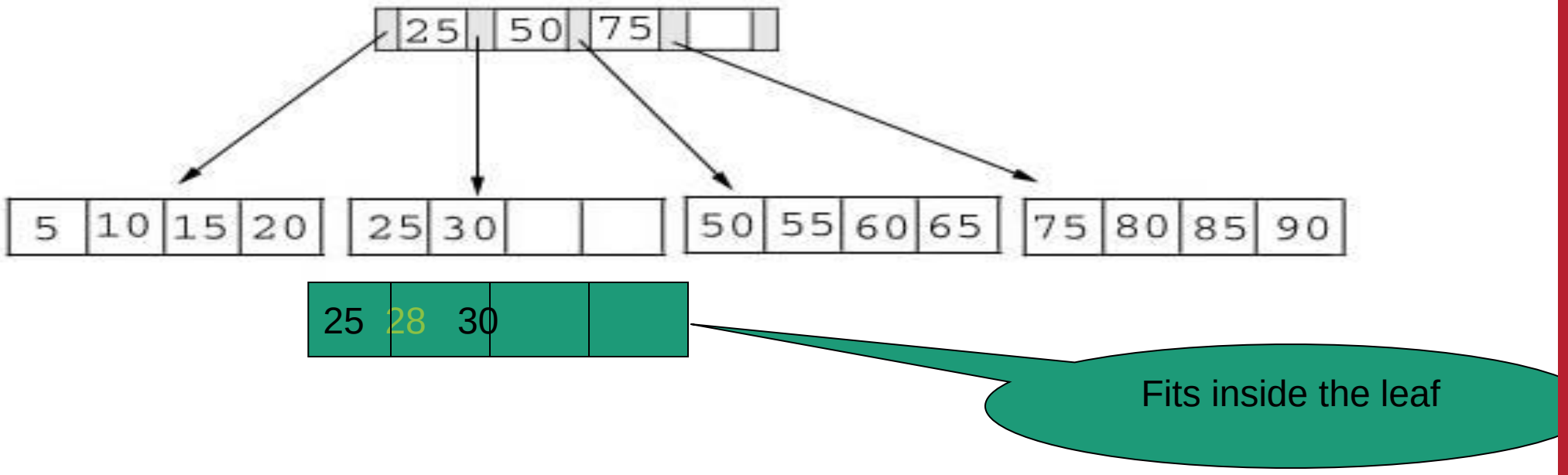


Searching

- Result:
  1. For the value of 45, not found.
  2. For the value of 15, return the position where the pointer located.

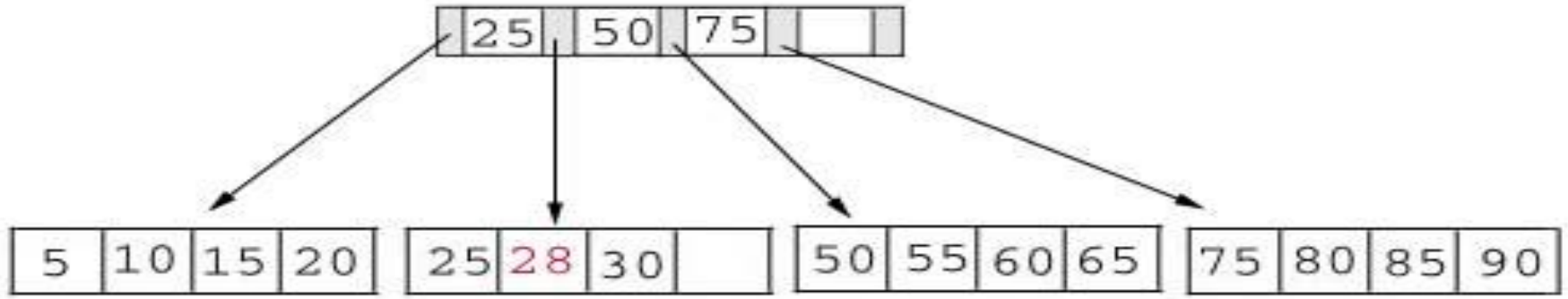
Insertion

- inserting a value into a B+ tree may unbalance the tree, so rearrange the tree if needed.
- Example #1: insert 28 into the below tree.



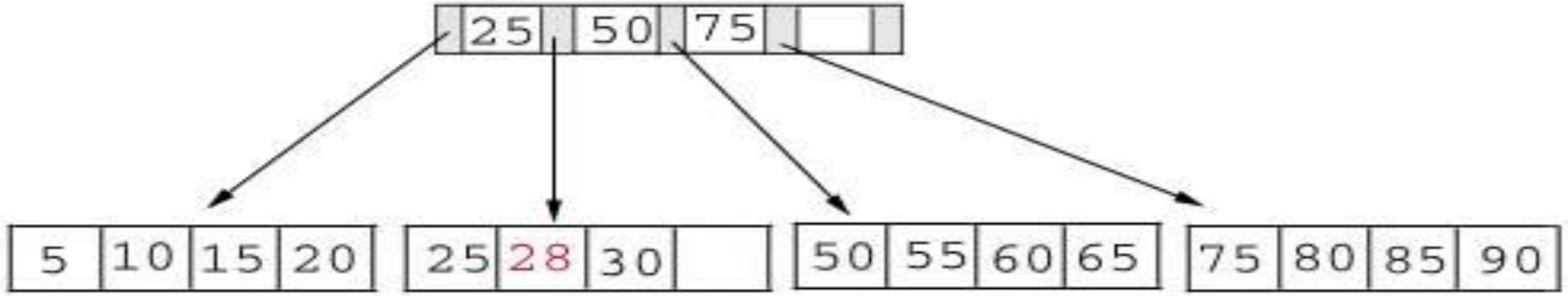
Insertion

- Result:



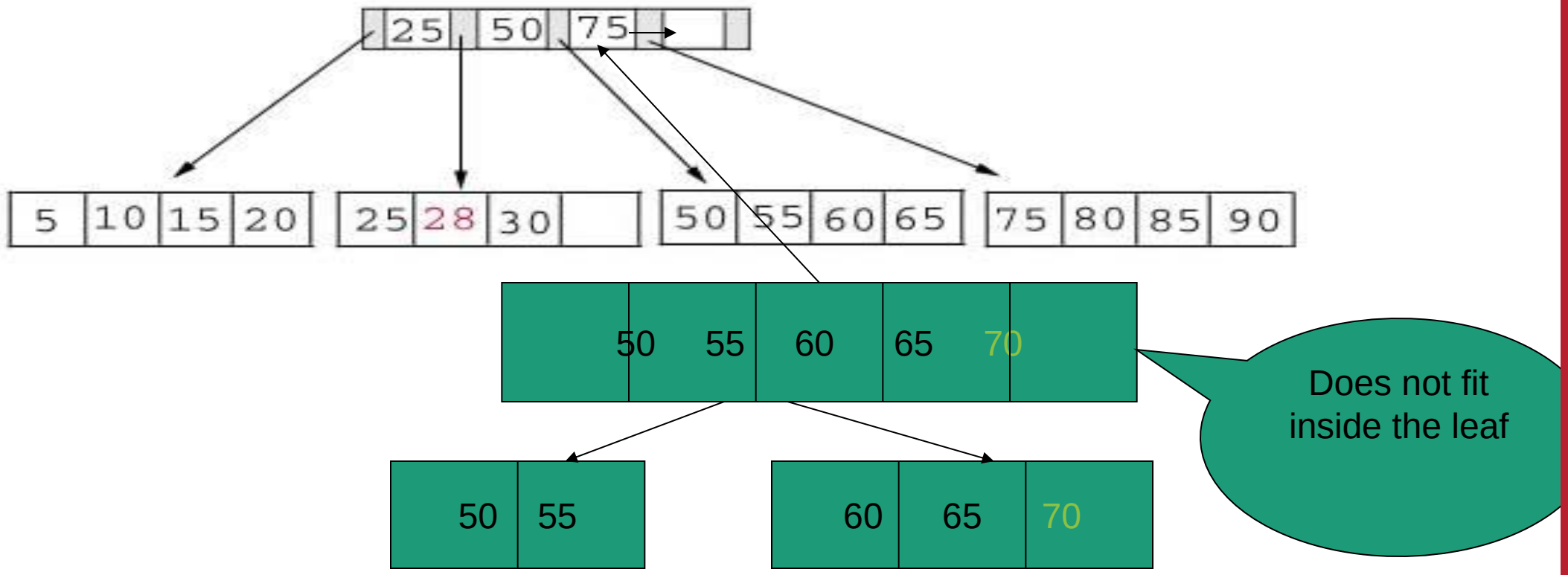
Insertion

- Example #2: insert 70 into below tree



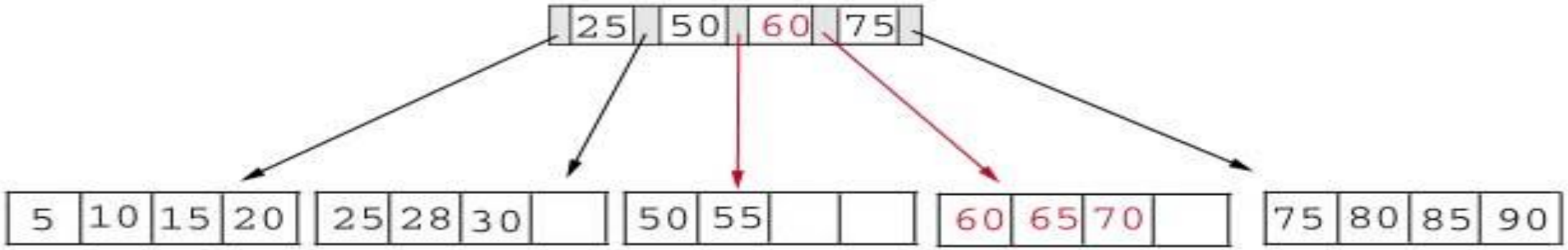
Insertion

- Process: split the leaf and propagate middle key up the tree



Insertion

- Result: chose the middle key 60, and place it in the index page between 50 and 75.



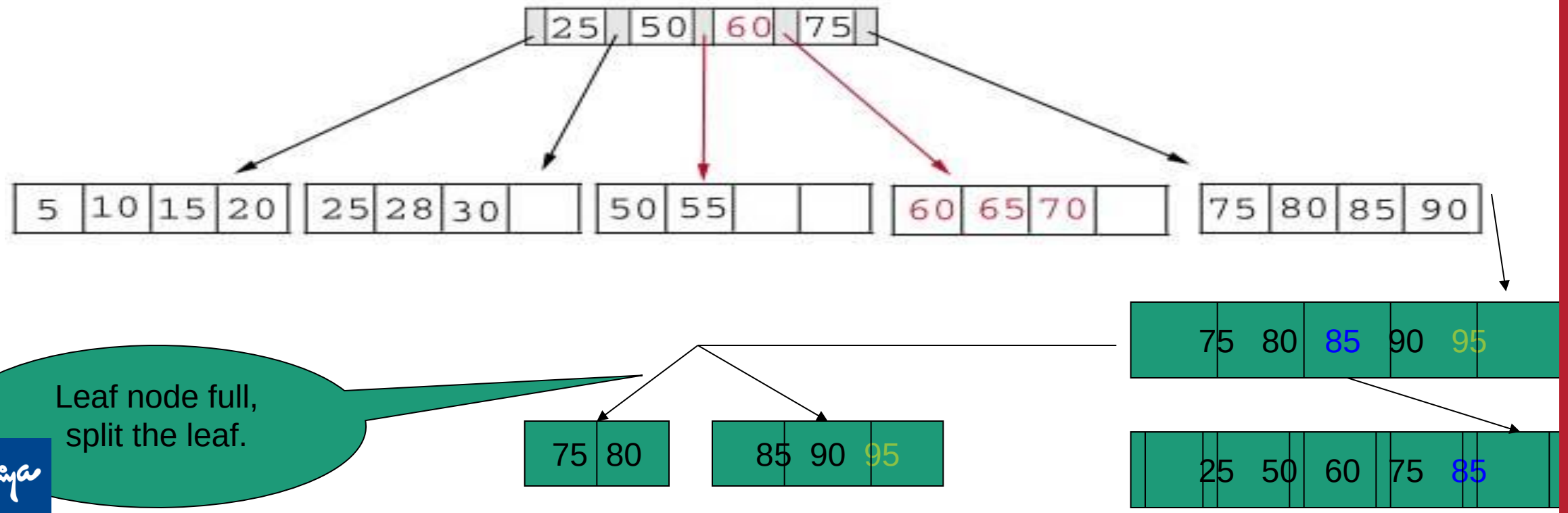
Insertion

The insert algorithm for B+ Tree

Leaf Node Full	Index Node Full	Action
NO	NO	Place the record in sorted position in the appropriate leaf page
YES	NO	1. Split the leaf node 2. Place Middle Key in the index node in sorted order. 3. Left leaf node contains records with keys below the middle key. 4. Right leaf node contains records with keys equal to or greater than the middle key.
YES	YES	1. Split the leaf node. 2. Records with keys < middle key go to the left leaf node. 3. Records with keys >= middle key go to the right leaf node. Split the index node. 4. Keys < middle key go to the left index node. 5. Keys > middle key go to the right index node. 6. The middle key goes to the next (higher level) index node. IF the next level index node is full, continue splitting the index nodes.

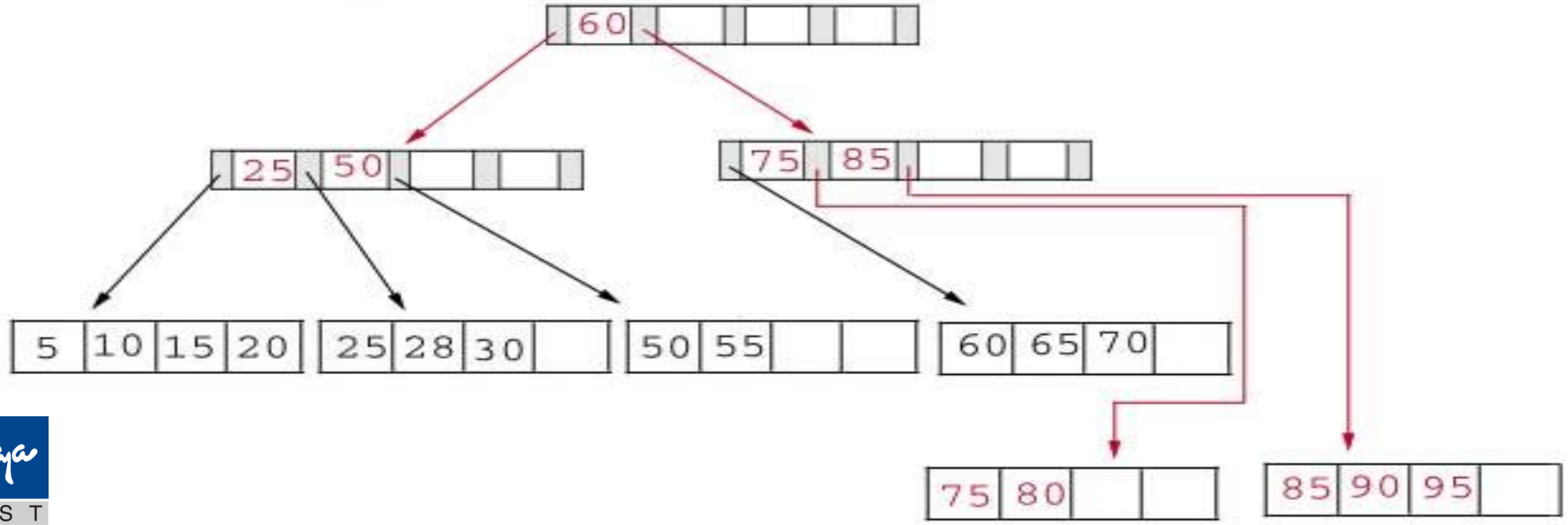
Insertion

- Exercise: add a key value 95 to the below tree.



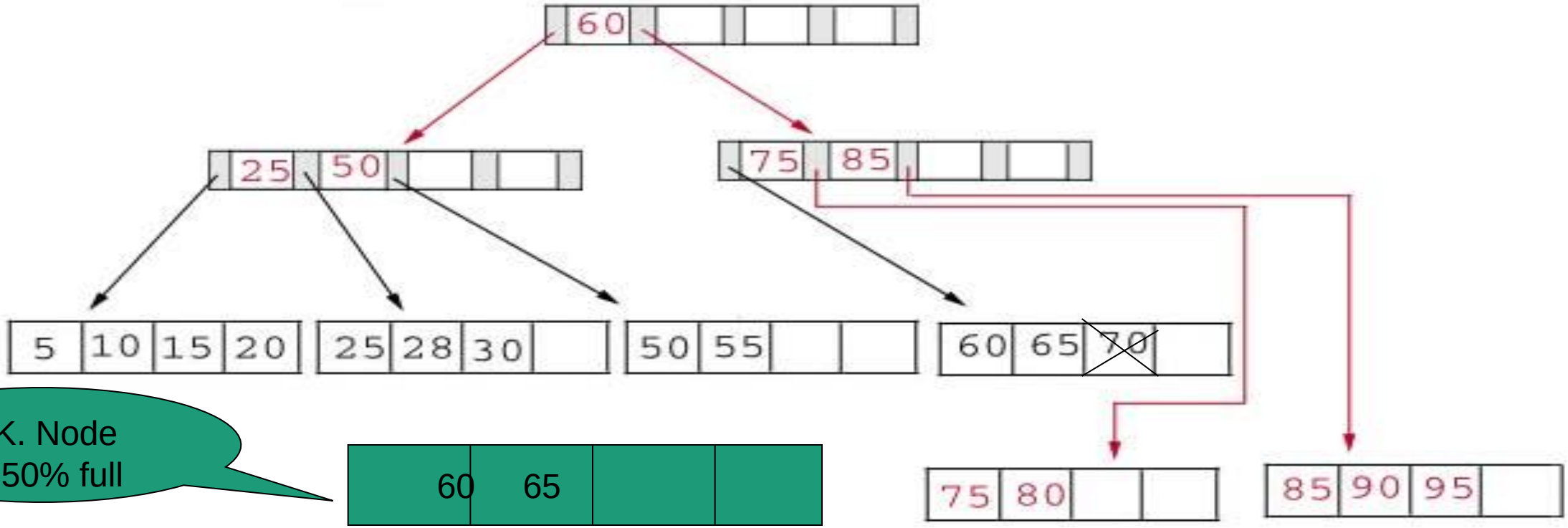
Insertion

- Result: again put the middle key 60 to the index page and rearrange the tree.



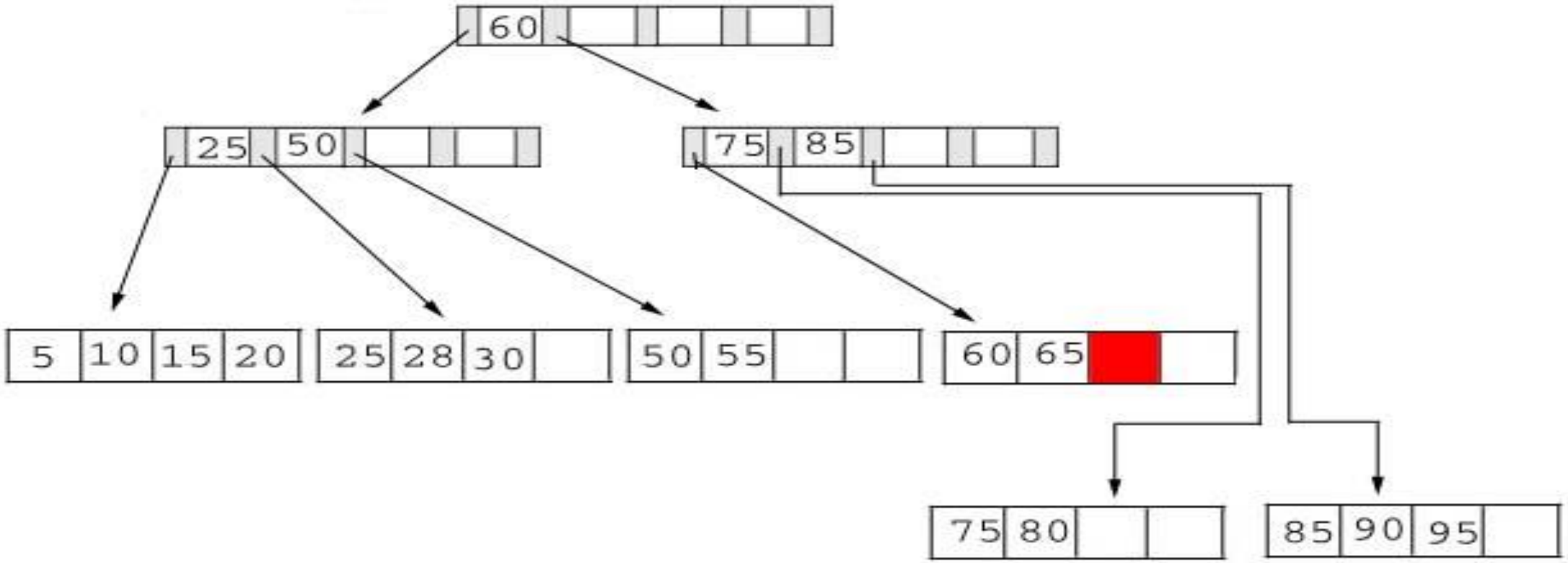
Deletion

- Same as insertion, the tree has to be rebuild if the deletion result violate the rule of B+ tree.
- Example #1: delete 70 from the tree



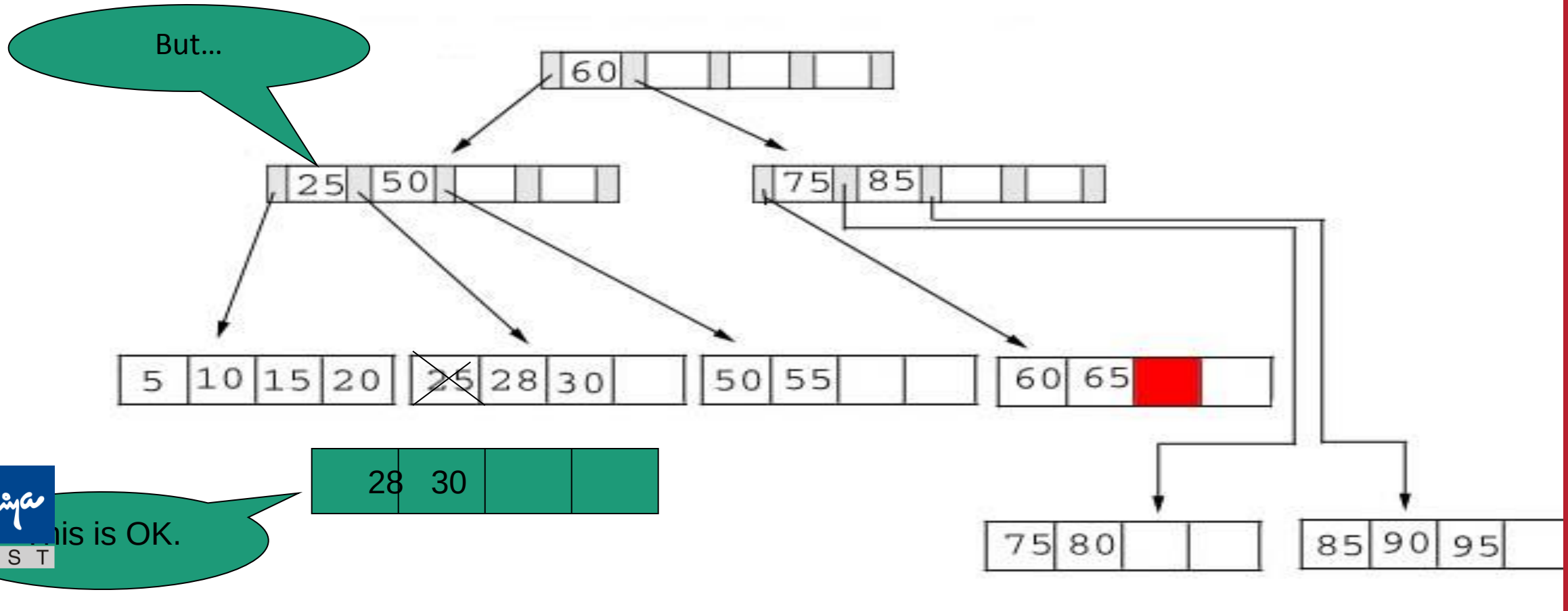
Deletion

- Result:



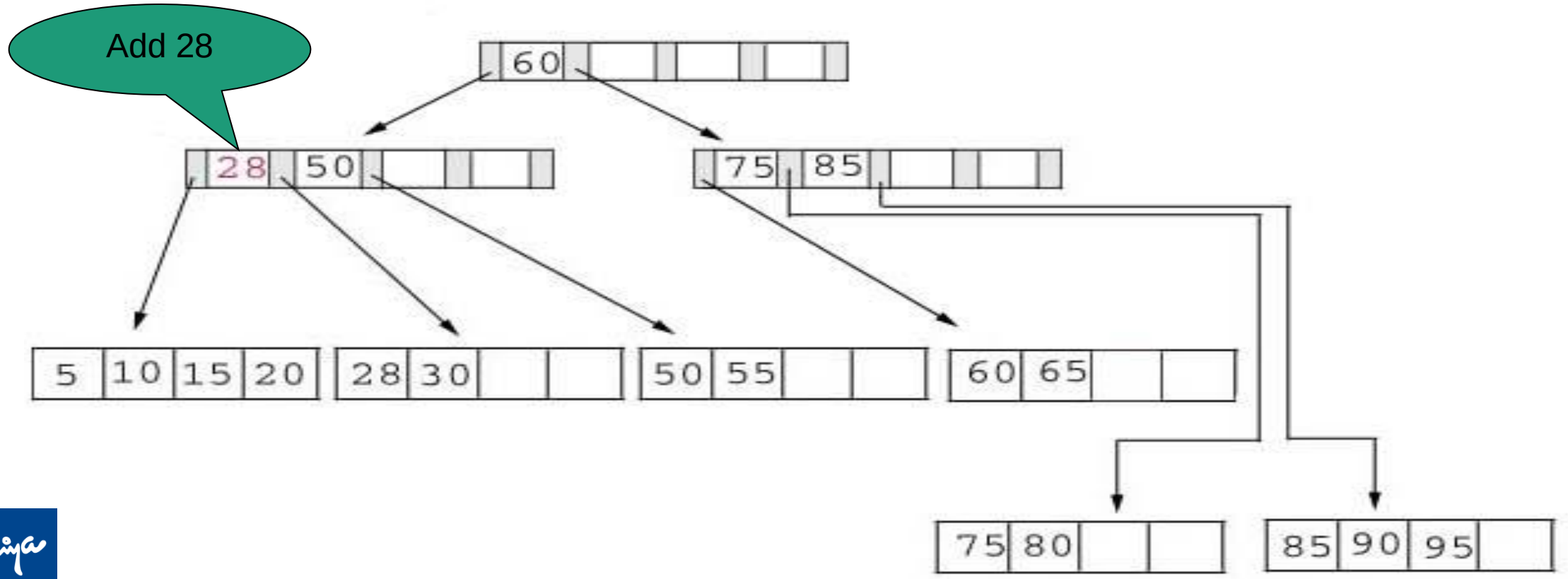
Deletion

Example #2: delete 25 from below tree, but 25 appears in the index page



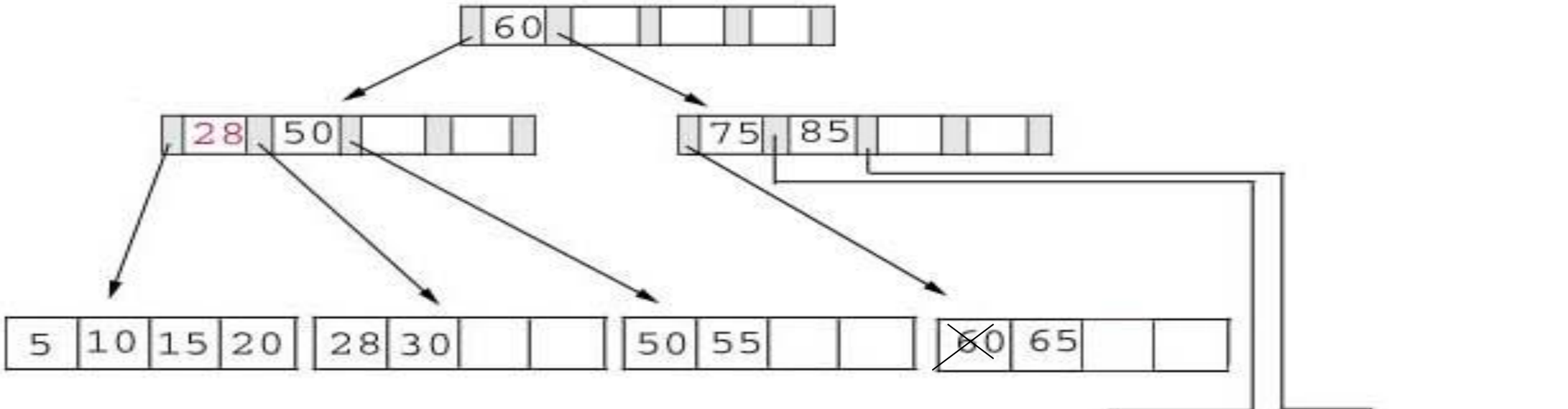
Deletion

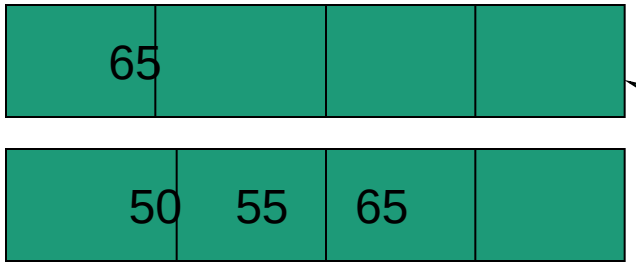
- Result: replace 28 in the index page.



Deletion

- Example #3: delete 60 from the below tree

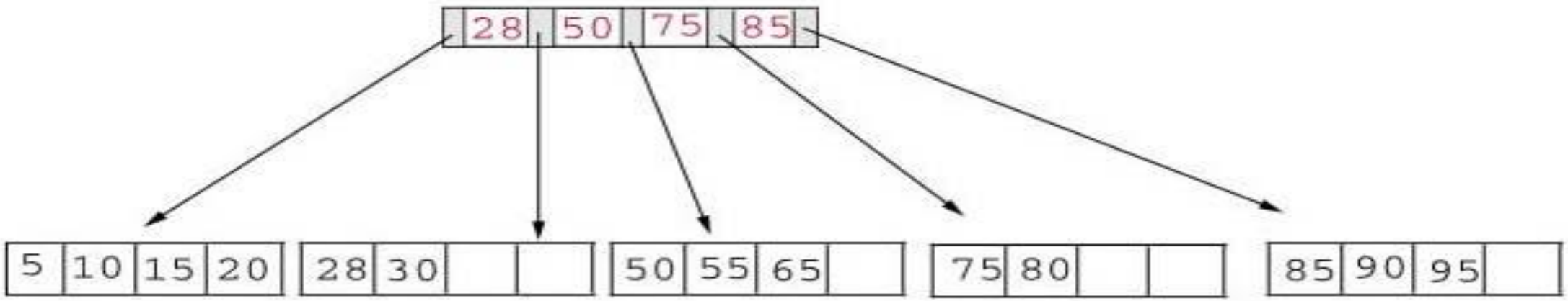




Less than 50% full

Deletion

- Result: delete 60 from the index page and combine the rest of index pages.



SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Deletion

- Delete algorithm for B+ trees

Data Page Below Fill Factor	Index Page Below Fill Factor	Action
NO	NO	Delete the record from the leaf page. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index page, use the next key to replace it.
YES	NO	Combine the leaf page and its sibling. Change the index page to reflect the change.
YES	YES	1. Combine the leaf page and its sibling. 2. Adjust the index page to reflect the change. 3. Combine the index page with its sibling. Continue combining index pages until you reach a page with the correct fill factor or you reach the root page.

Somaiya TRUST

Conclusion

- For a B+ Tree:
- It is “easy” to maintain its balance
  - Insert/Deletion complexity $O(\log_{M/2})$
- The searching time is shorter than most of other types of trees because branching factor is high

B+Trees and DBMS

- Used to index primary keys
- Can access records in $O(\log_{M/2})$ traversals (height of the tree)
- Interior nodes contain Keys only
  - Set node sizes so that the M-1 keys and M pointers fits inside a single block on disk
    - E.g., block size 4096B, keys 10B, pointers 8 bytes
    - (8+ (10+8)*M-1) = 4096
    - M = 228; 2.7 billion nodes in 4 levels
  - One block read per node visited

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Reference

Li Wen & Sin-Min Lee, San Jose State University

Somaiya TRUST

Queries?

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Thank you!

Somaiya TRUST