SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Multiway Trees

swatimali@somaiya.edu

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

Multiway Trees

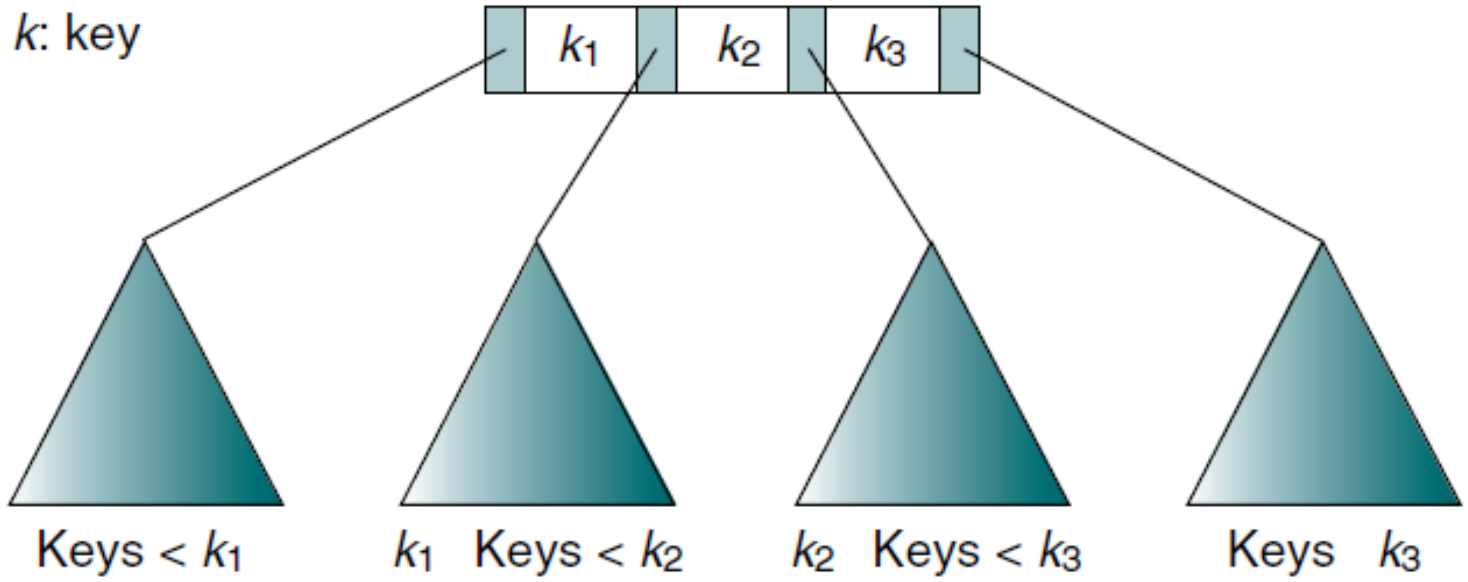
- A tree that can have more than two children
- A multiway tree of order m (or an m-way tree) is one in which a tree can have m children.
- Nodes in an m-way tree will be made up of key fields, in this case m-1 key fields, and pointers to children.

Multiway Tree Definition

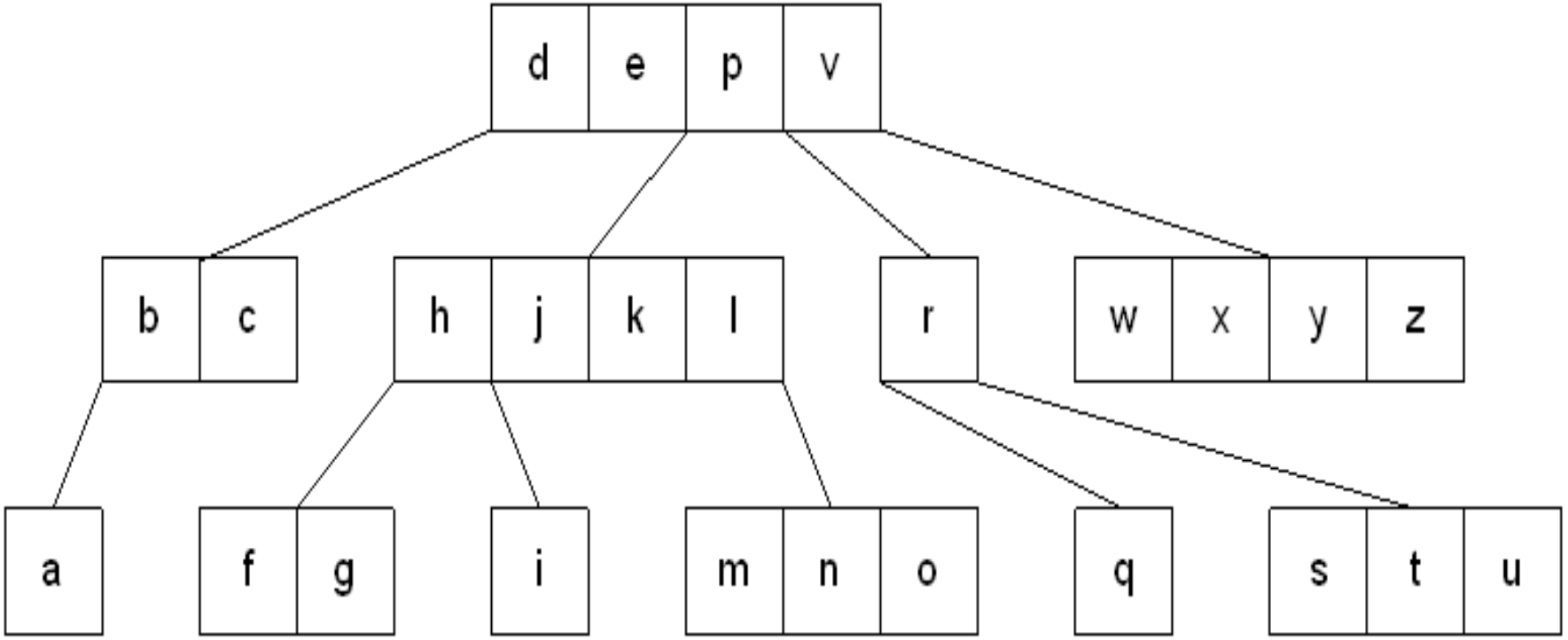
By definition an m-way search tree is a tree in which:

- Each node has m children and m-1 key fields
- The keys in each node are in ascending order.
- The keys in the first i children are smaller than the ith key
- The keys in the last m-i children are larger than the ith key

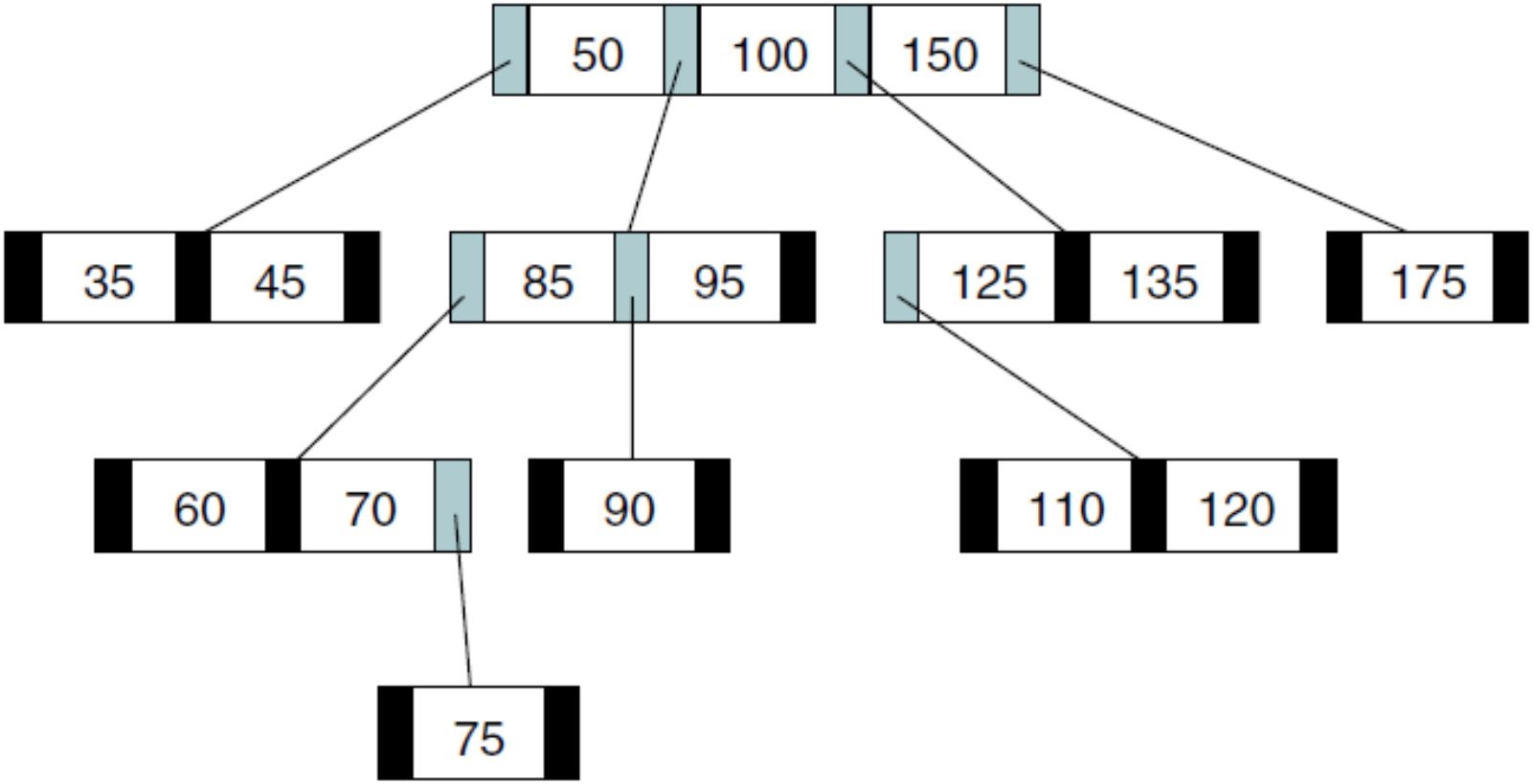
A Multiway tree



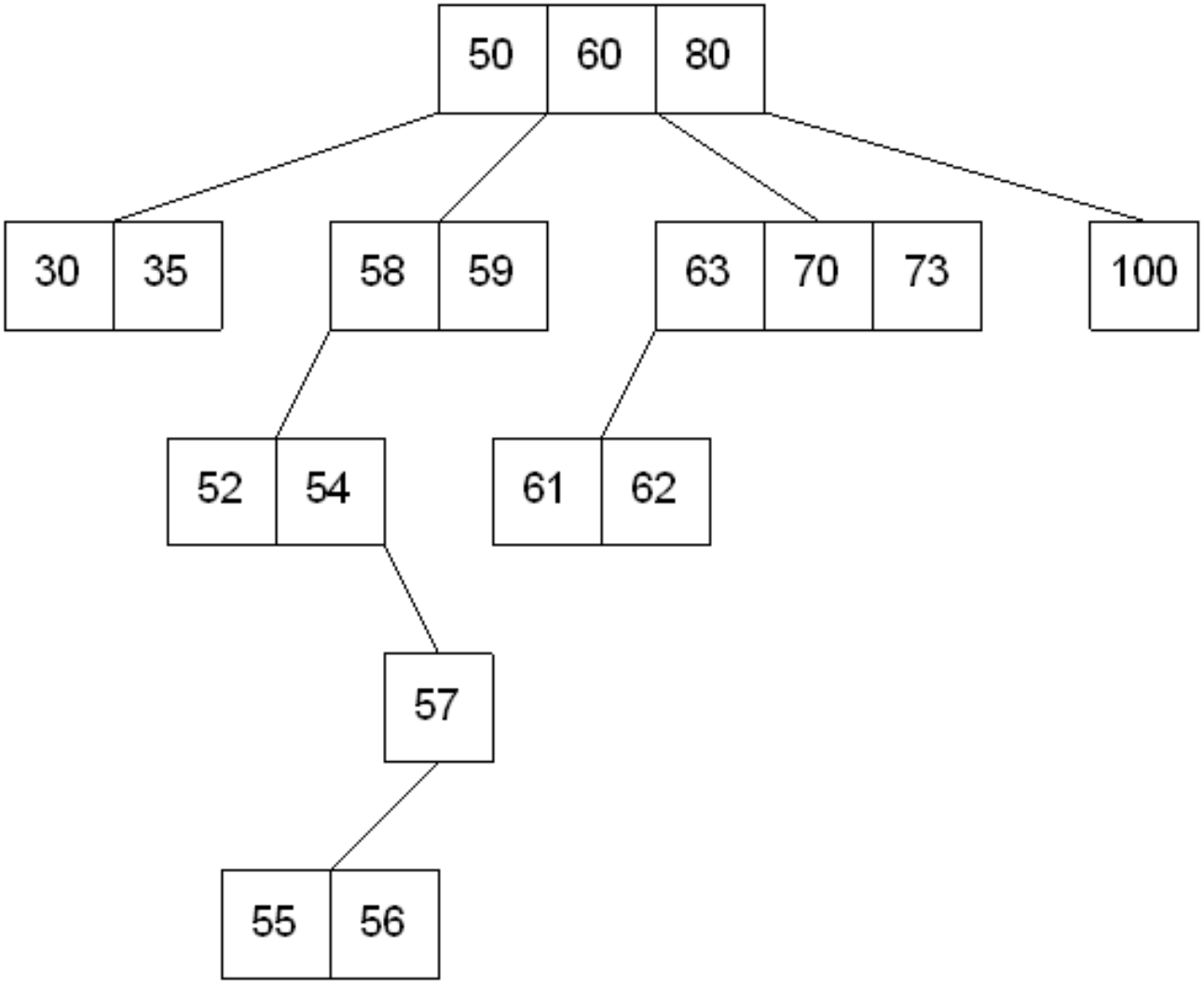
A Multiway Tree



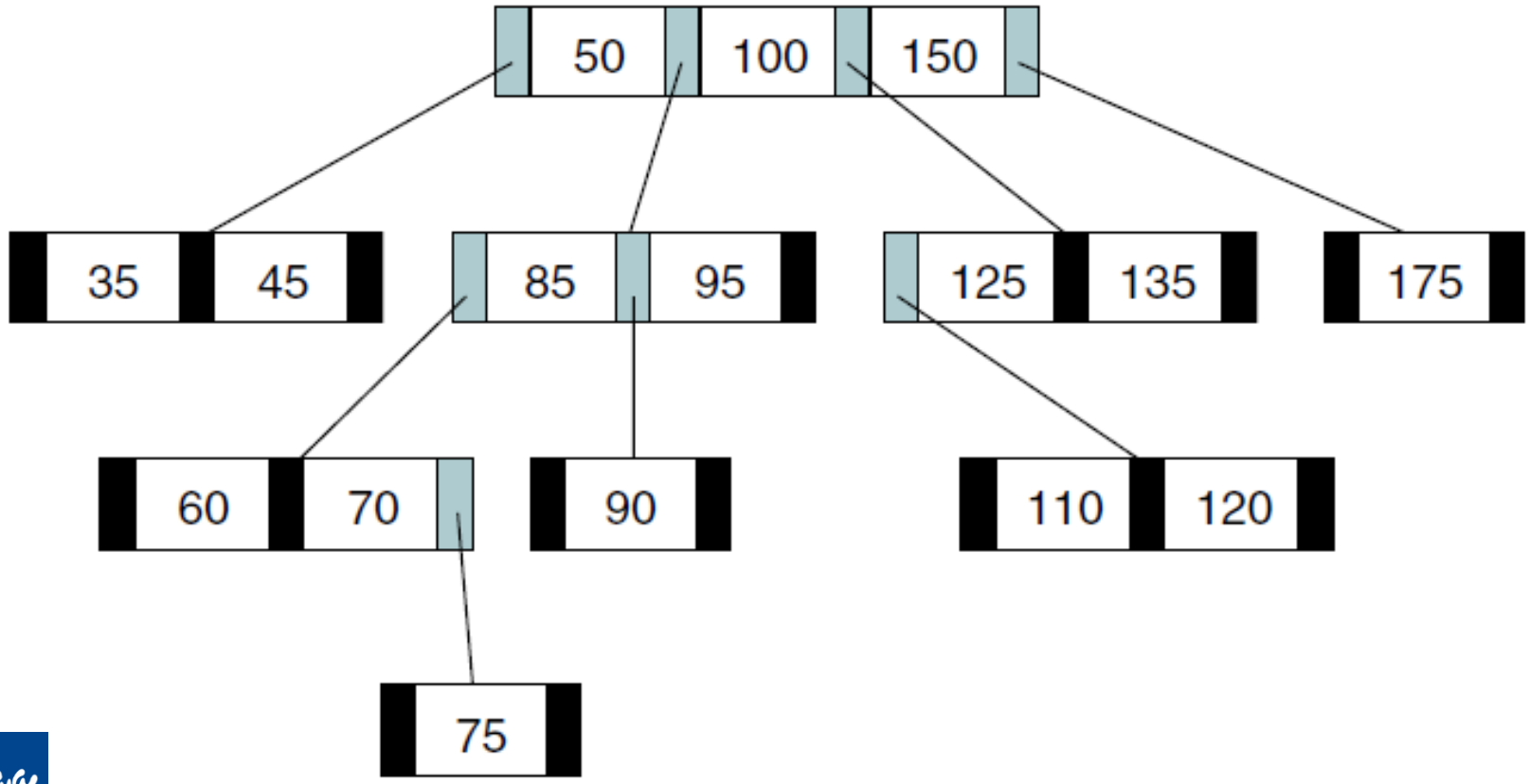
A Multiway Tree



4-way tree



4-way tree



Multiway tree

- Advantages: Provide fast information retrieval and update.
- Disadvantages: Can become unbalanced, which means that the construction of the tree becomes of vital importance.

B Trees

- An extension of a multiway search tree of order m is a B-tree of order m.
- Used when the data to be accessed/stored is located on secondary storage devices because they allow for large amounts of data to be stored in a node.
- Self balancing tree

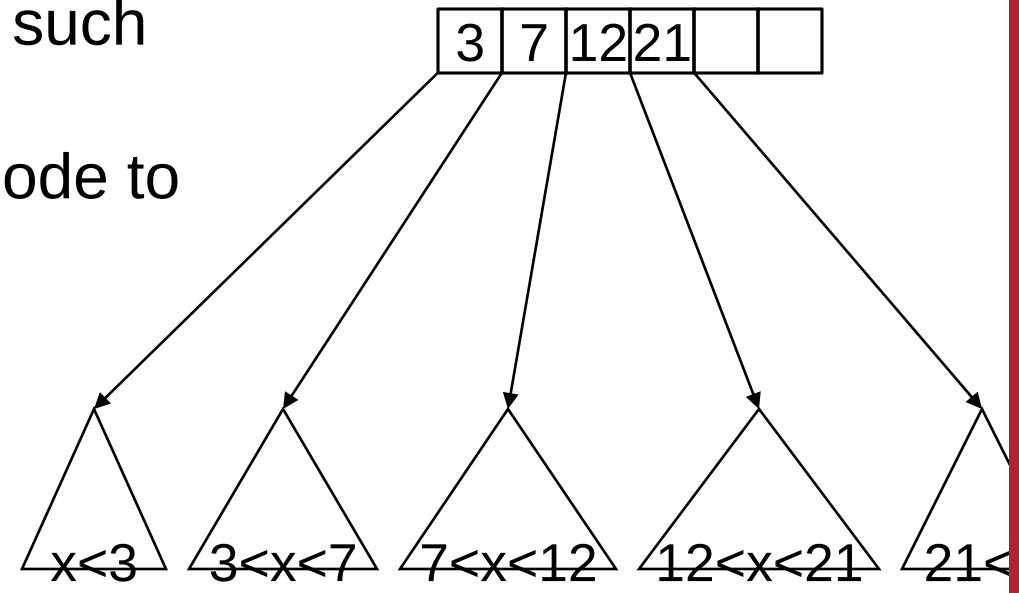
B-Tree

- A B-tree of order m is a multiway search tree in which:
  - The root has at least two subtrees unless it is the only node in the tree.
  - Each non-root and each non-leaf node have at most m nonempty children and at least m/2 nonempty children.
  - The number of keys in each non-root and each non-leaf node is one less than the number of its nonempty children.
  - All leaves are on the same level.

These restrictions make B-trees always at least half full, have few levels, and remain perfectly balanced.

B Trees

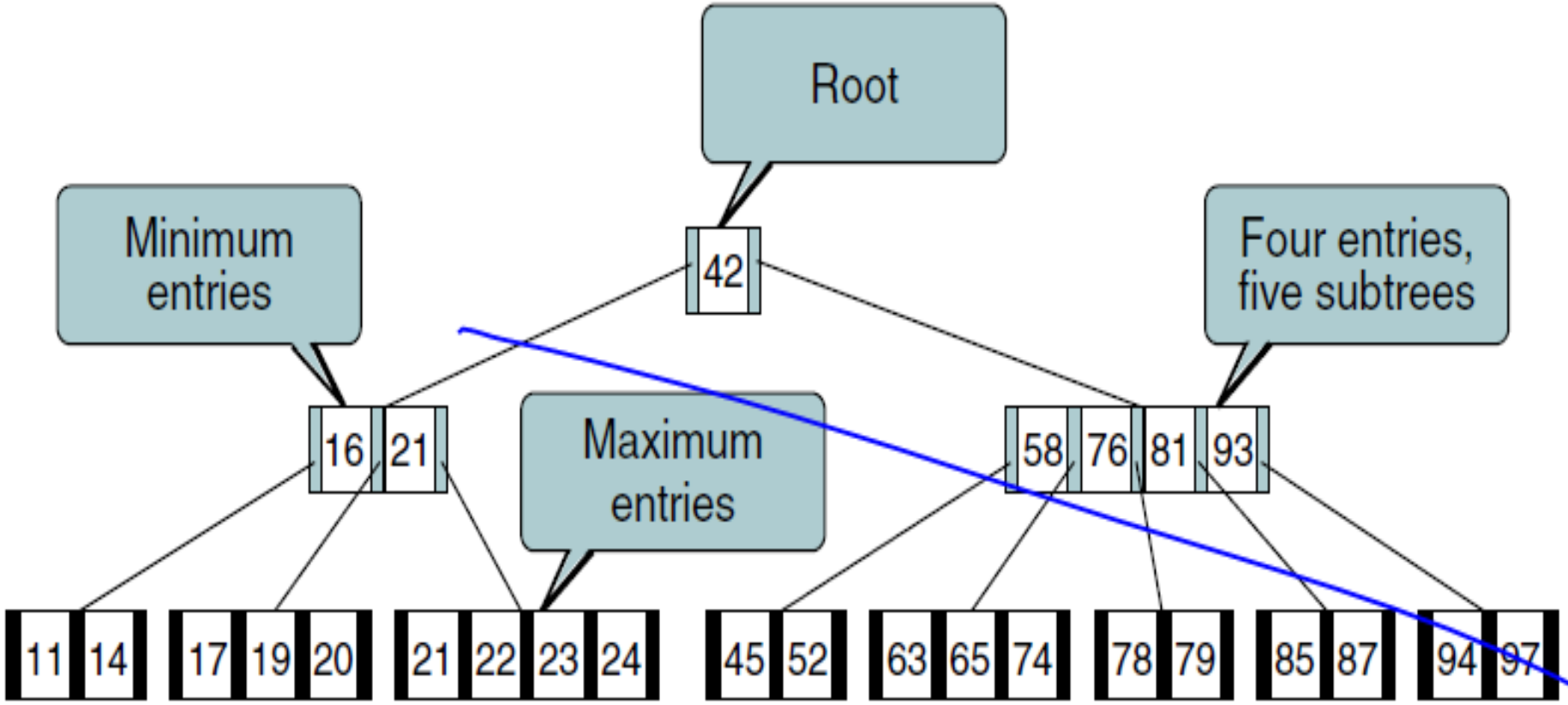
- B-Trees are specialized *M*-ary search trees
- Each node has many keys
- subtree between two keys x and y contains values v such that $x \leq v < y$
- binary search within a node to find correct subtree
- Each node takes one full {*page, block, line*} of memory (disk)



B-Tree Properties

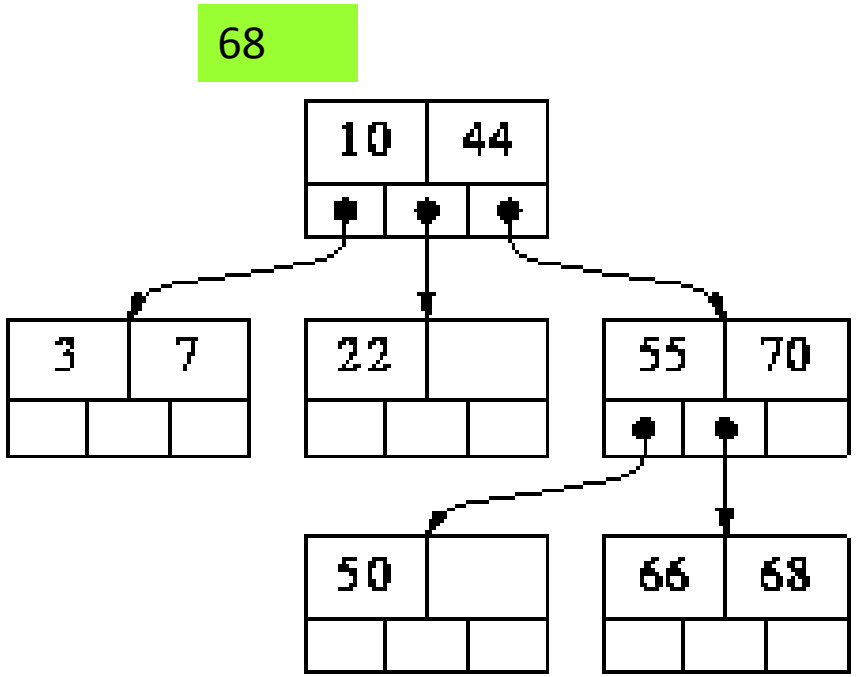
- Properties
  - maximum branching factor of *M*
  - the root has between 2 and *M* children *or* at most *M-1* keys
  - All other nodes have between $\lceil M/2 \rceil$ and M records
    - Keys+data
- Result
  - tree is O(log M) deep
  - all operations run in O(log M) time
  - operations pull in about *M* items at a time

B-Tree of order 5



Example: 3-way search tree

Try: search 68



Search for X

At a node consisting of values $V_1 \ldots V_k$, there are four possible cases:

- If $X < V_1$, recursively search for X in the subtree that is left of $V1$
- If $X > V_k$, recursively search for X in the subtree that is right of V_k
- If $X=V_i$, for some i, then we are done (X has been found)
- Else, for some i, $V_i < X < V_{i+1}$. In this case recursively search for X in the subtree that is between V_i and V_{i+1}

- Time Complexity: $O((M-1)*h)=O(h)$ [M is a constant]

Insertion into a B-tree

- The condition that all leaves must be on the same level forces B-trees to not to grow at the their leaves; instead they are forced to grow at the root.
- A value is inserted directly into a leaf.
- Common insertion cases:
  - A key is placed into a leaf that still has room.
  - The leaf in which a key is to be placed is full.
  - The root of the B-tree is full.

Insertion into a B-tree

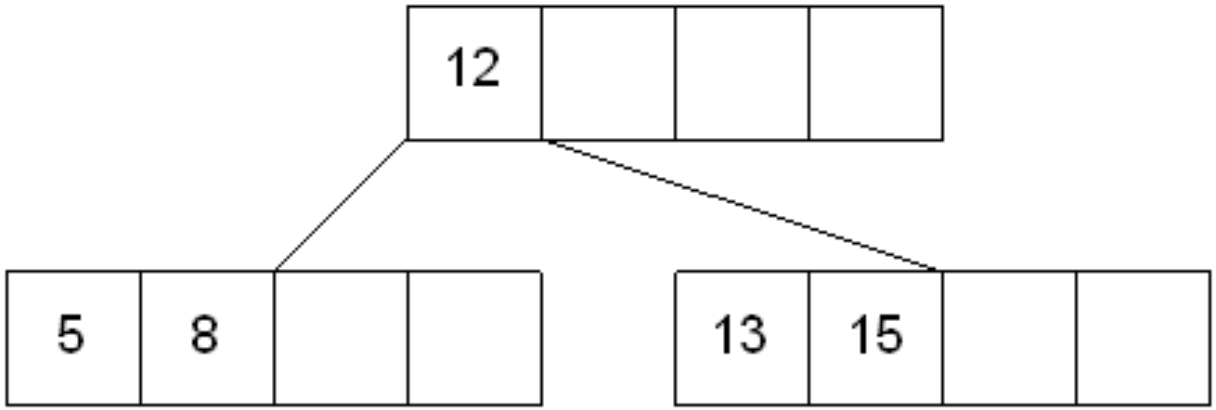
- Since B Tree is a self-balancing tree, you cannot force insert a key into just any node.
- The following algorithm applies:
  - Run the search operation and find the appropriate place of insertion.
  - Insert the new key at the proper location, but if the node has a maximum number of keys already:
  - The node, along with a newly inserted key, will split from the middle element.
  - The middle element will become the parent for the other two child nodes.
  - The nodes must re-arrange keys in ascending order.

Insertion in B-Tree

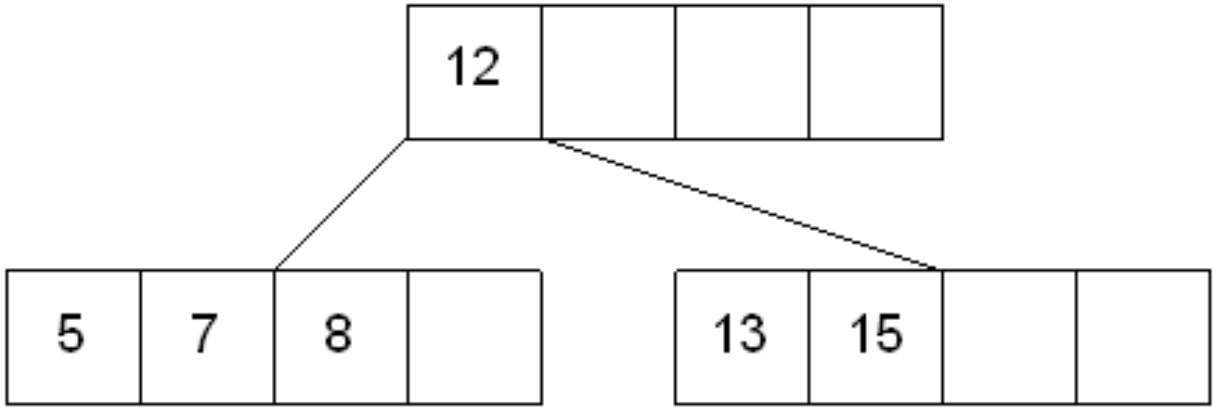
Case 1: A key is placed into a leaf that still has room

This is the easiest of the cases to solve because the value is simply inserted into the correct sorted position in the leaf node.

Insertion into a B-tree



Inserting the number 7 results in:



Insertion into a B-tree

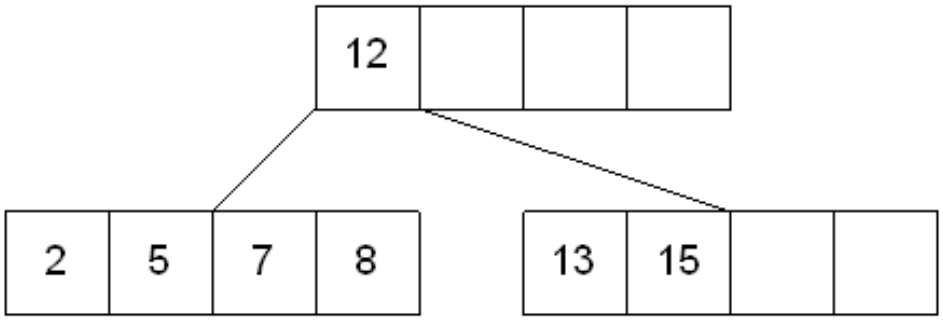
- Case 2: The leaf in which a key is to be placed is full

- In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree.

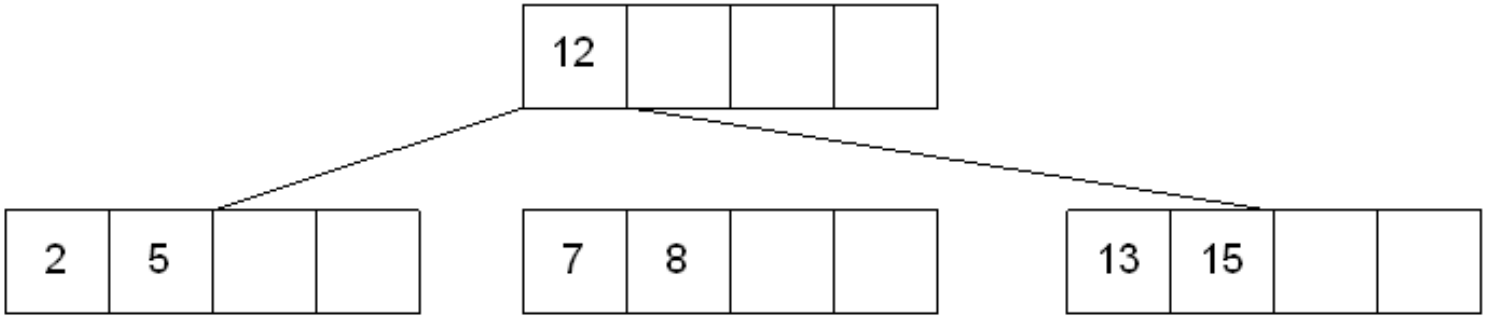
- The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process is continues up the tree until all of the values have "found" a location.

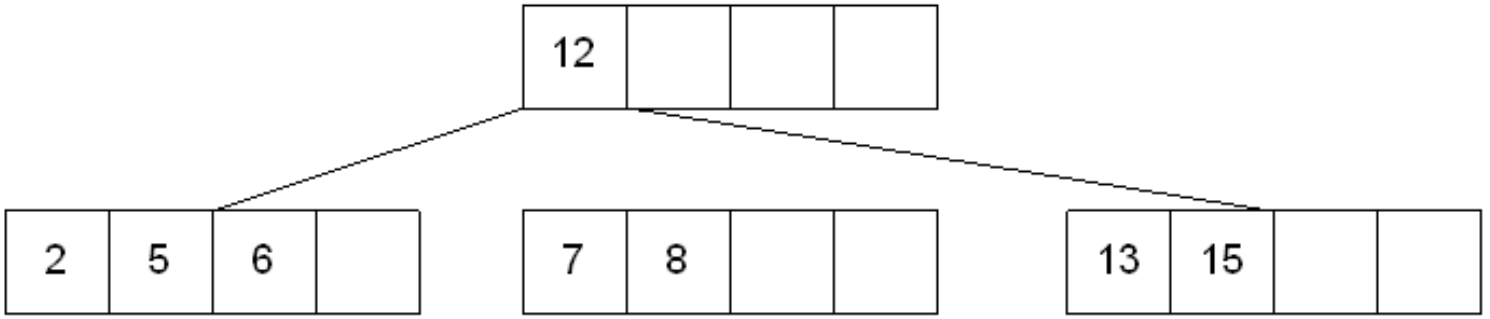
Insertion into a B-tree

Insert 6 into the following B-tree:

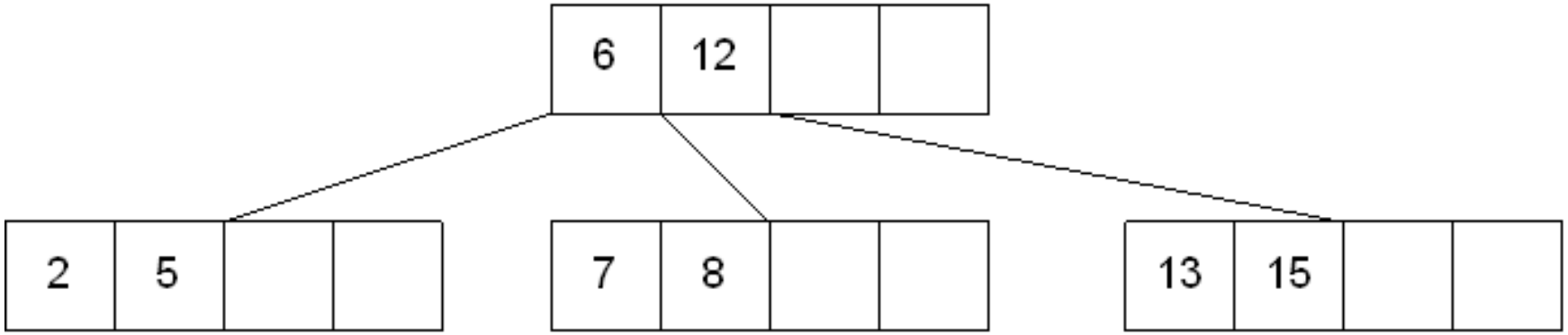


results in a split of the first leaf node:





Insertion into a B-tree

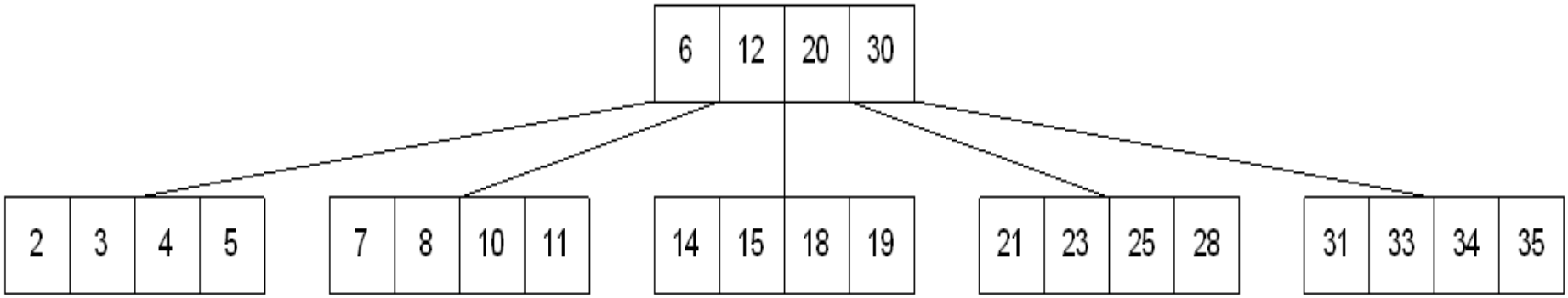


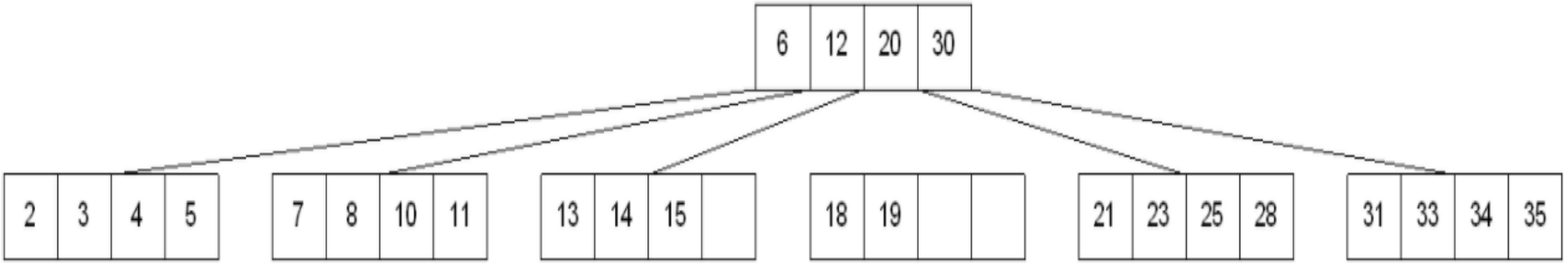
Insertion into a B-tree

- Case 3: The root of the B-tree is full
- The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree.
- If the root is full, the same basic process from case 2 will be applied and a new root will be created.
- This type of split results in 2 new nodes being added to the B-tree.

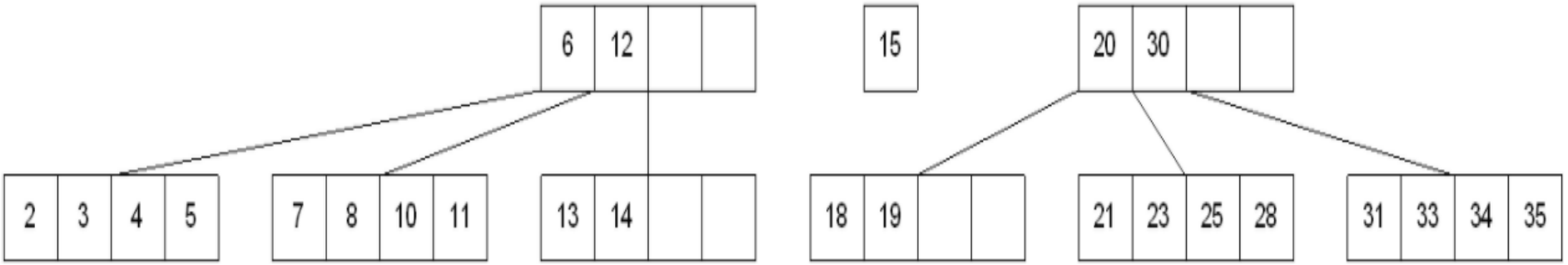
Insertion into a B-tree

Inserting 13 into the following tree:

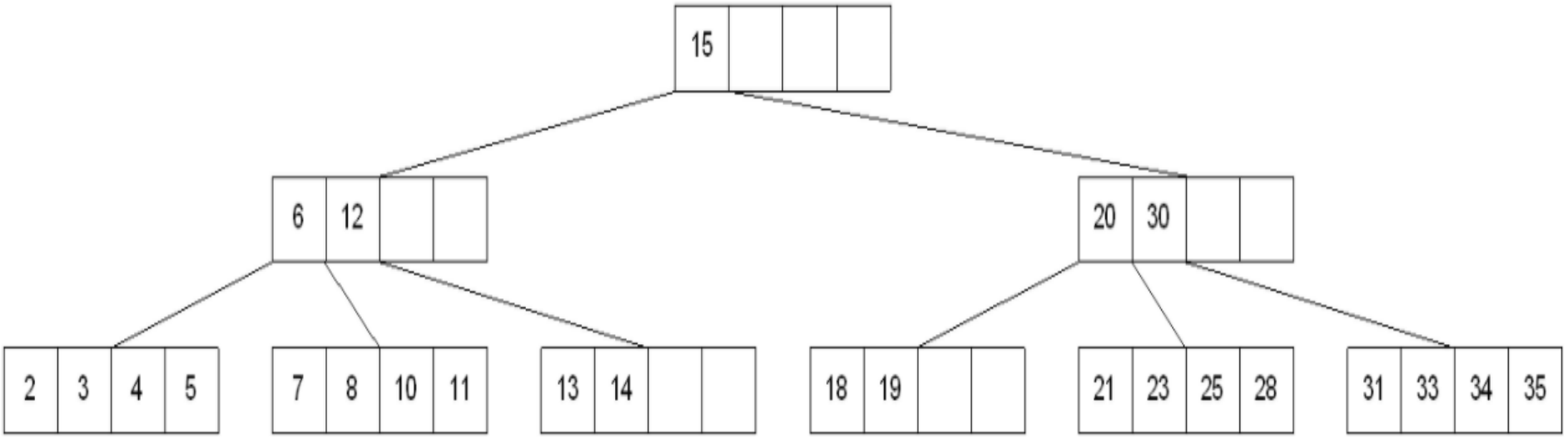




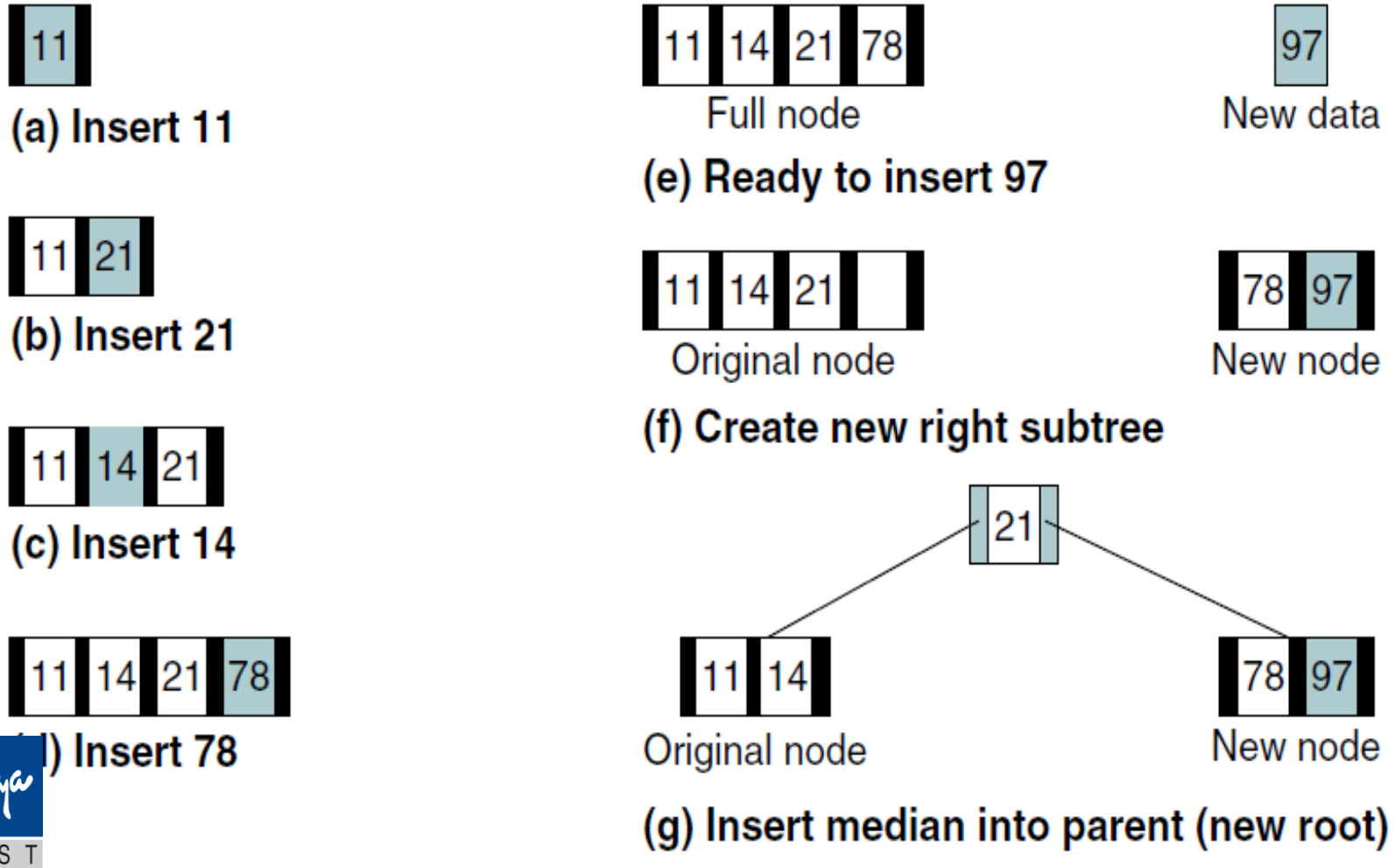
The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:



Insertion into a B-tree



Deleting from a B-tree

- The deletion process will basically be a reversal of the insertion process –
  - rather than splitting nodes, it's possible that nodes will be merged.

- There are two main cases to be considered:
  - Deletion from a leaf
  - Deletion from a non-leaf

Deletion- If the target key is in the leaf node

- Target is in the leaf node, more than min keys.
  - Deleting this will not violate the property of B Tree
- Target is in leaf node, it has min key nodes
  - Deleting this will violate the property of B Tree
  - Target node can borrow key from immediate left node, or immediate right node (sibling)
  - The sibling will say **yes** if it has more than minimum number of keys
  - The key will be borrowed from the parent node, the max value will be transferred to a parent, the max value of the parent node will be transferred to the target node, and remove the target value
- Target is in the leaf node, but no siblings have more than min number of keys
  - Search for key
  - Merge with siblings and the minimum of parent nodes
  - Total keys will be now more than min
  - The target key will be replaced with the minimum of a parent node

Deletion: **If the target key is in an internal node**

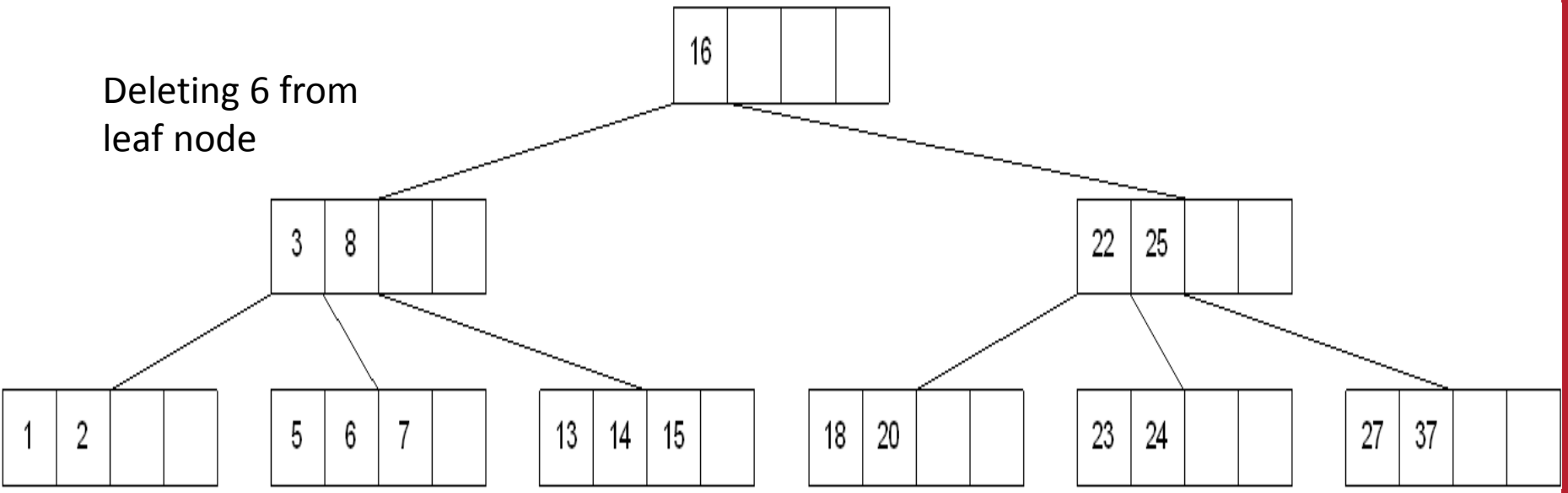
- Either choose, in- order predecessor or in-order successor
- In case the of in-order predecessor, the maximum key from its left subtree will be selected
- In case of in-order successor, the minimum key from its right subtree will be selected
- If the target key's in-order predecessor has more than the min keys, only then it can replace the target key with the max of the in-order predecessor
- If the target key's in-order predecessor does not have more than min keys, look for in-order successor's minimum key.
- If the target key's in-order predecessor and successor both have less than min keys, then merge the predecessor and successor.

Deletion: If the target key is in a root node

- Replace with the maximum element of the in-order predecessor subtree
- If, after deletion, the target has less than min keys, then the target node will borrow max value from its sibling via sibling's parent.
- The max value of the parent will be taken by a target, but with the nodes of the max value of the sibling.

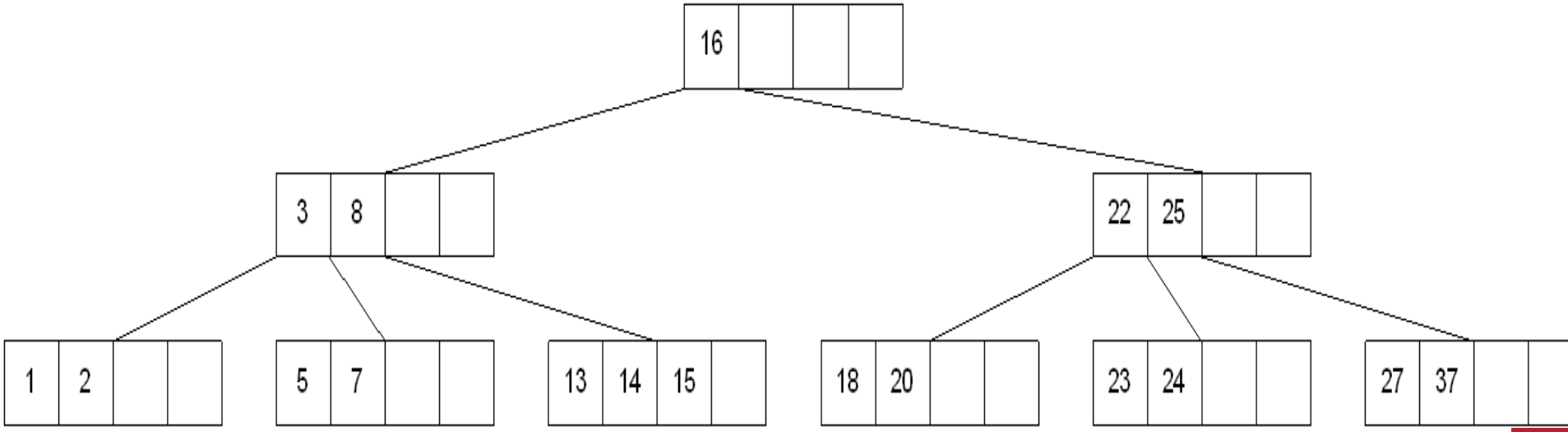
Deletion from a leaf

Deleting 6 from leaf node



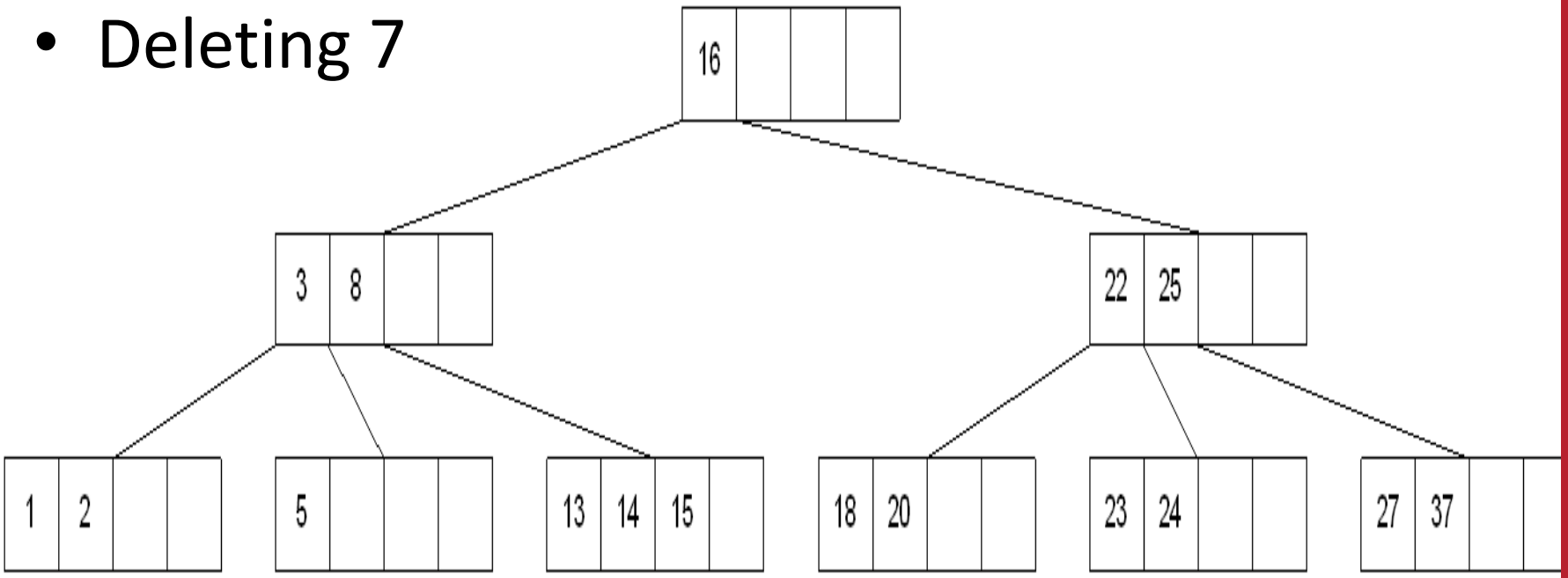
Deletion from a leaf

- After deleting 6



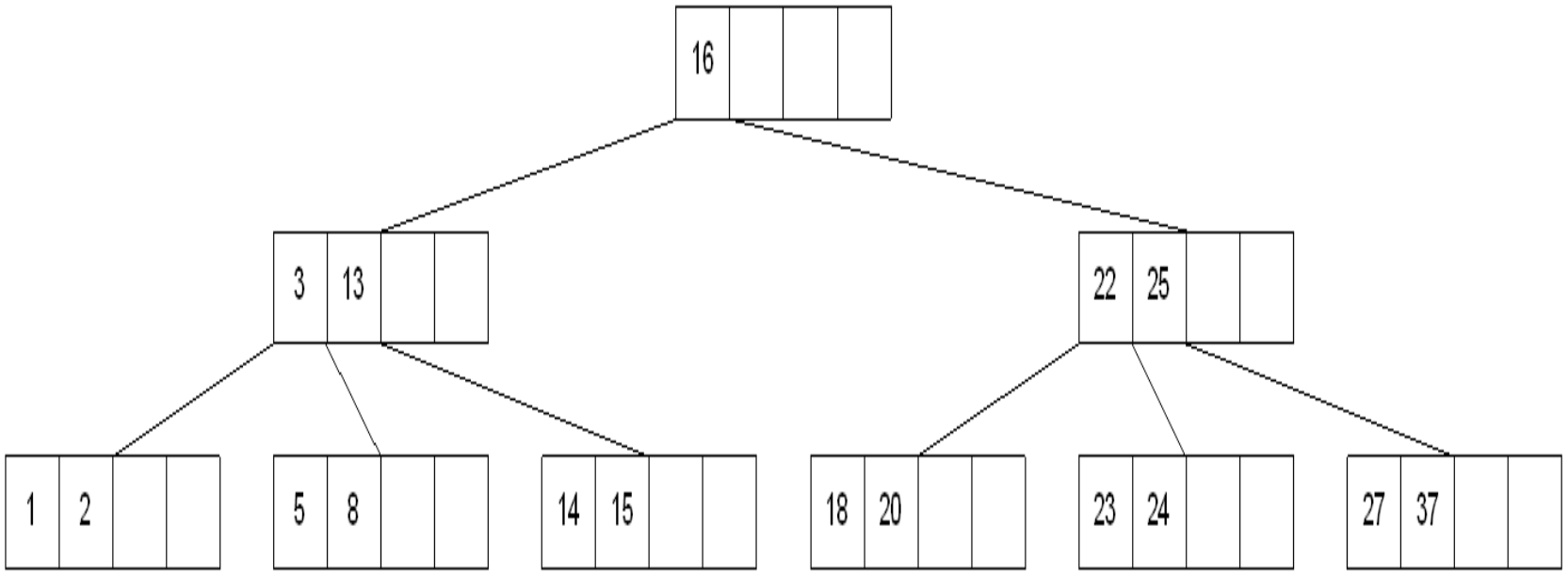
Deletion from a leaf

- Deleting 7



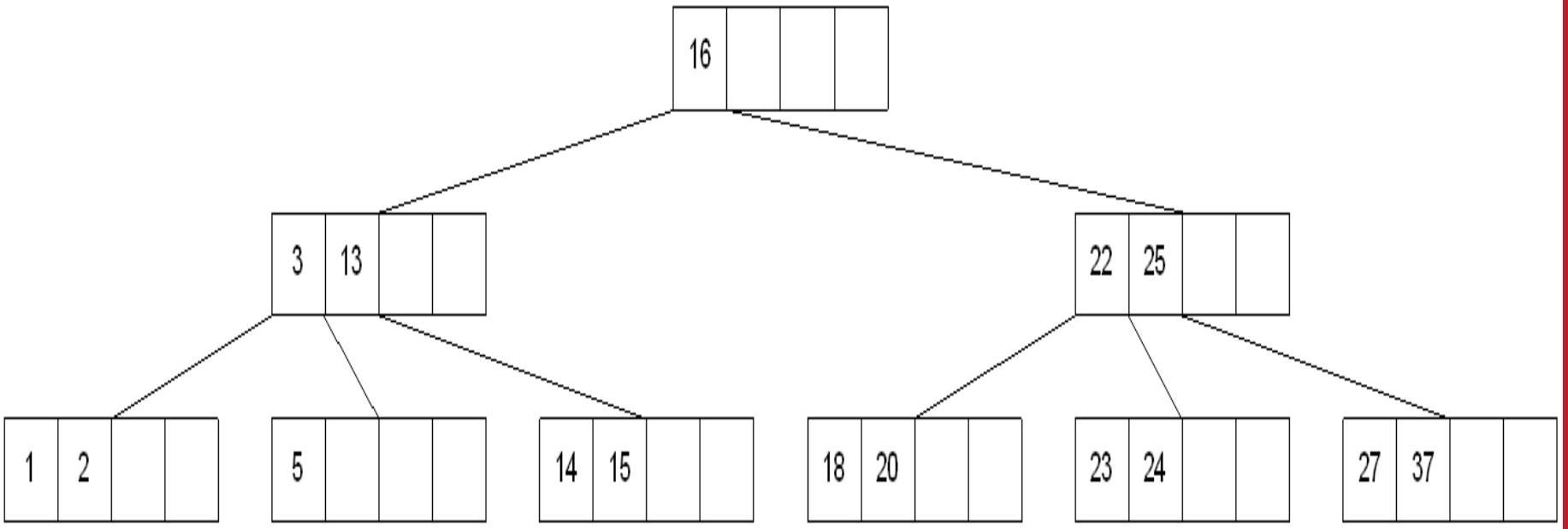
Deletion from a leaf

- Deleting 7, Borrow a key from right sibling



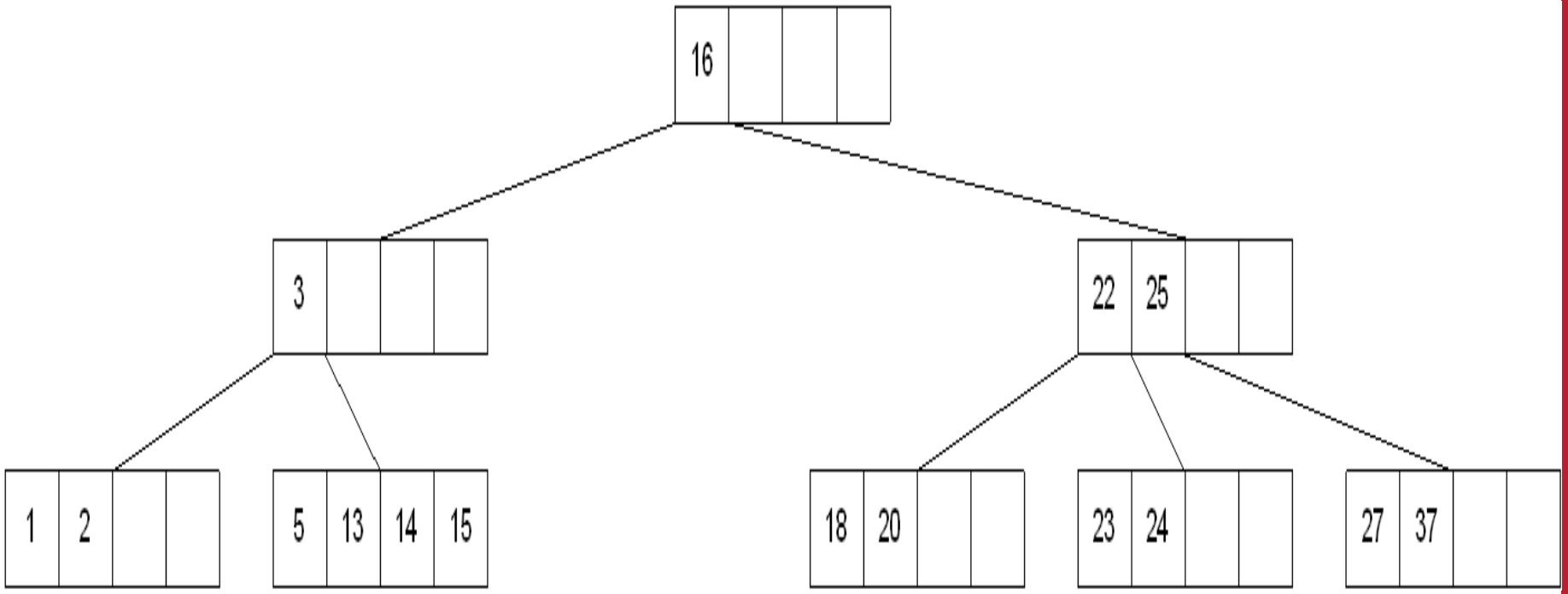
Deletion from a leaf

- Deleting 8



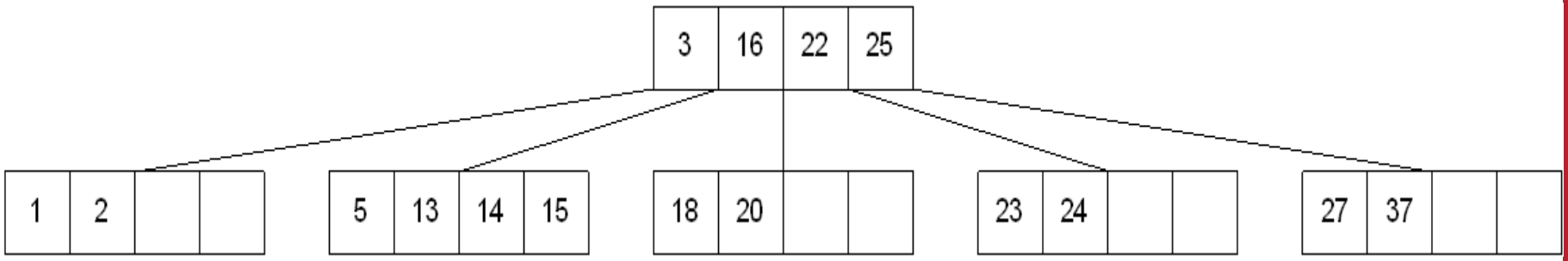
Deletion from a leaf

Deleting 8 and merging with right sibling



Deletion from a leaf

- Deleting 8 needs further merging as some nodes have less than m/2 no of keys

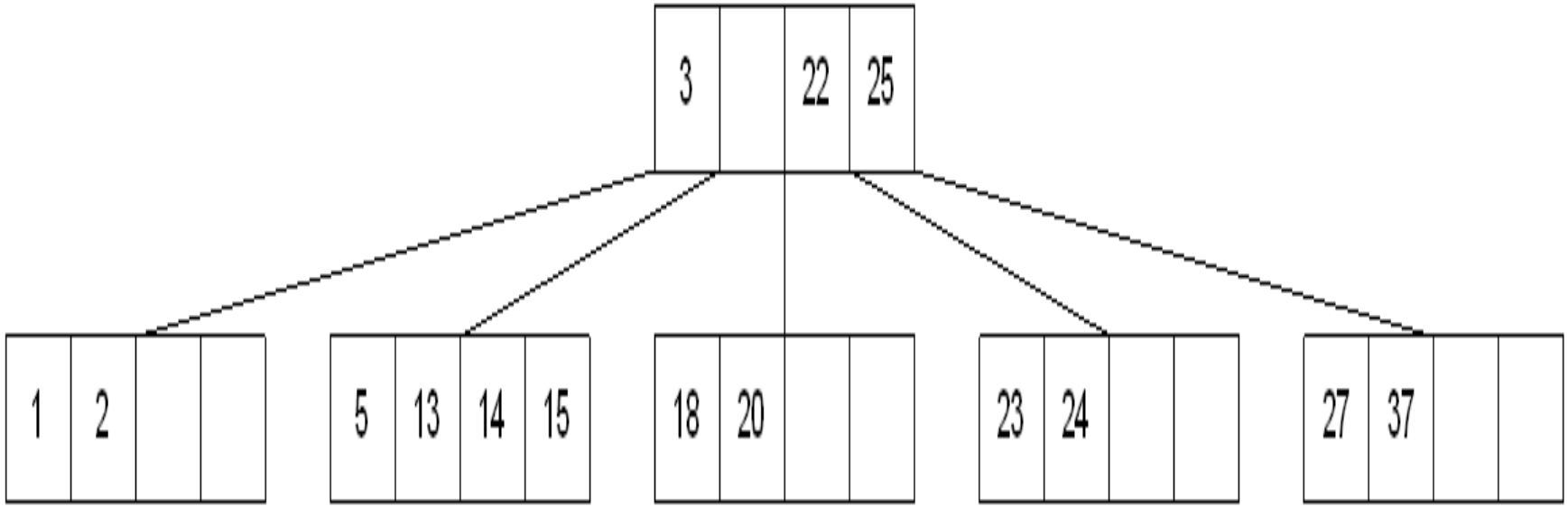


Case 2: Deletion from a non-leaf

- This case can lead to problems with tree reorganization but it will be solved in a manner similar to deletion from a binary search tree.
- The key to be deleted will be replaced by its immediate predecessor (or successor) and then the predecessor (or successor) will be deleted since it can only be found in a leaf node.

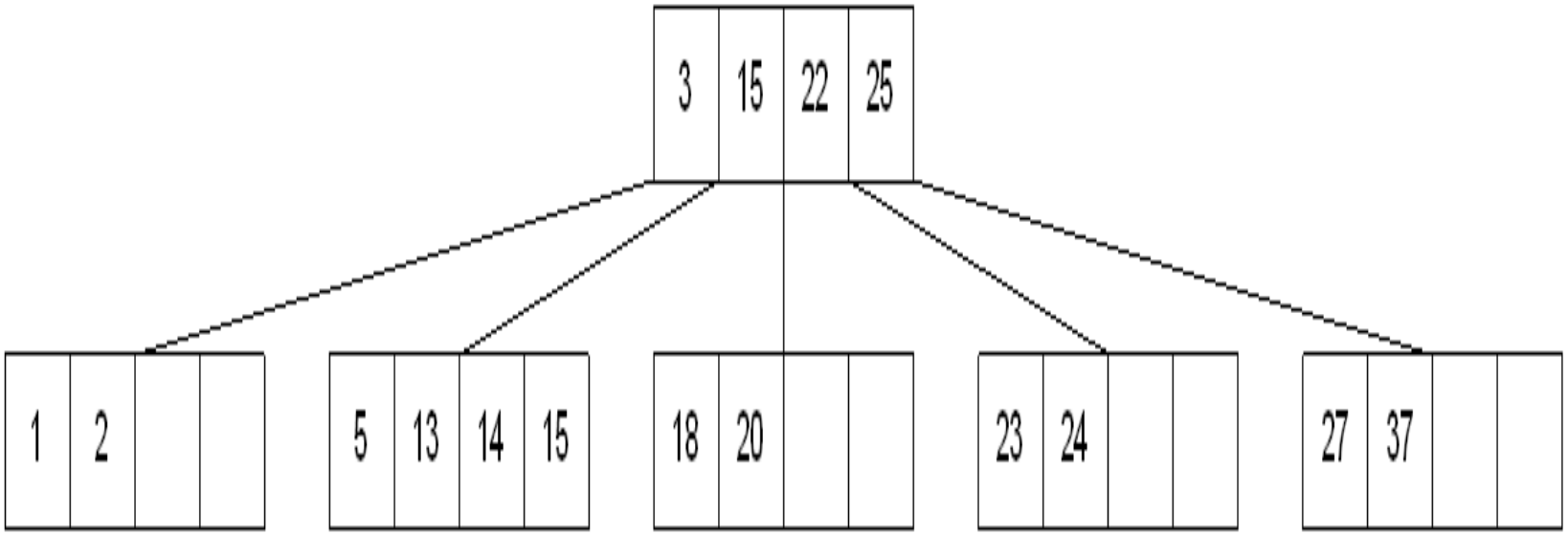
Case 2. Deletion from a non-leaf

- Deleting 16



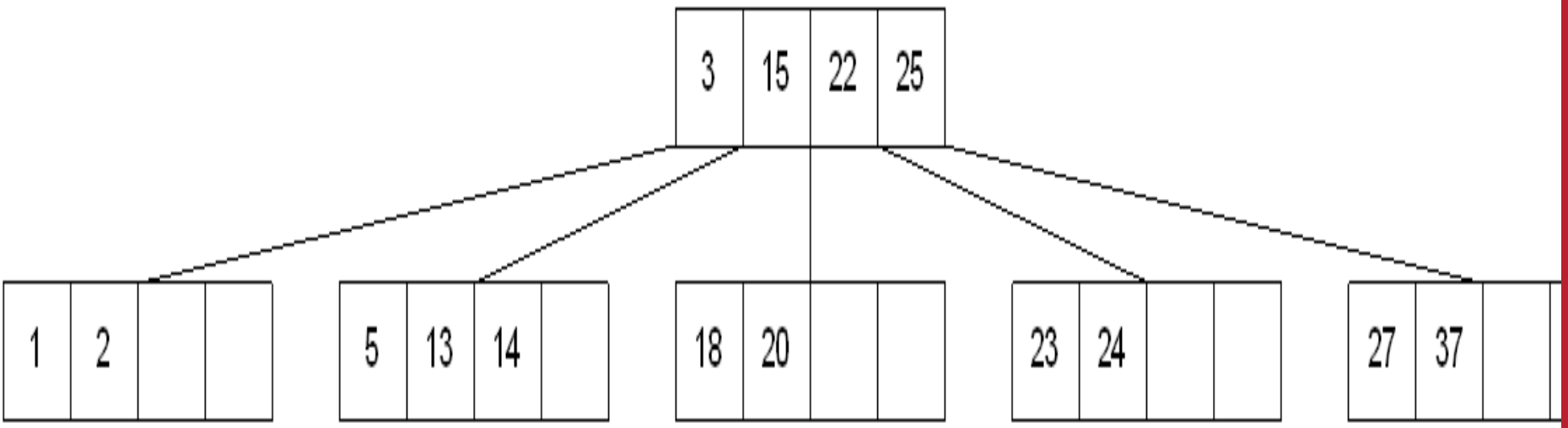
Case 2. Deletion from a non-leaf

- The "gap" is filled in with the immediate predecessor:



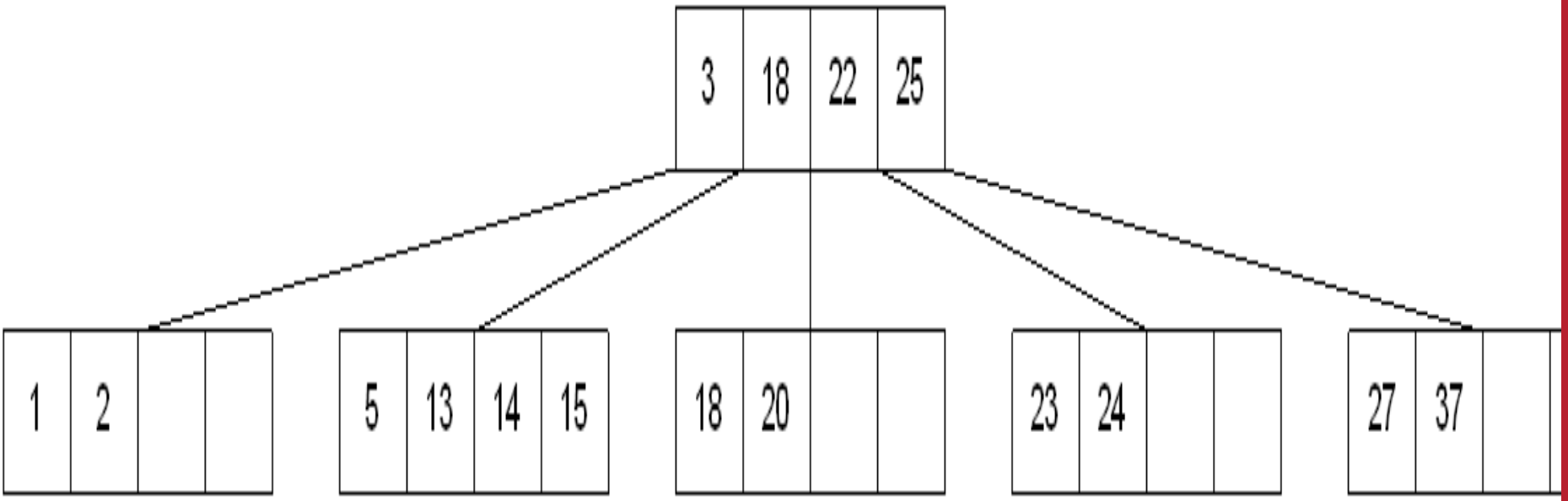
Case 2. Deletion from a non-leaf

- the immediate predecessor is deleted:



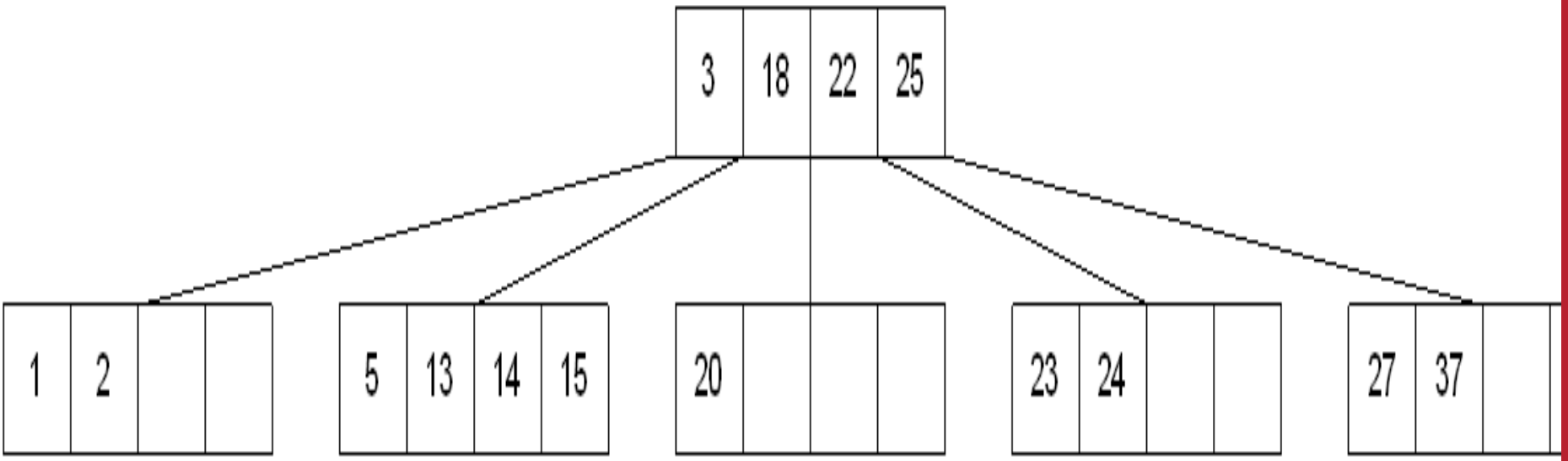
Case 2. Deletion from a non-leaf

- If the immediate successor had been chosen as the replacement:



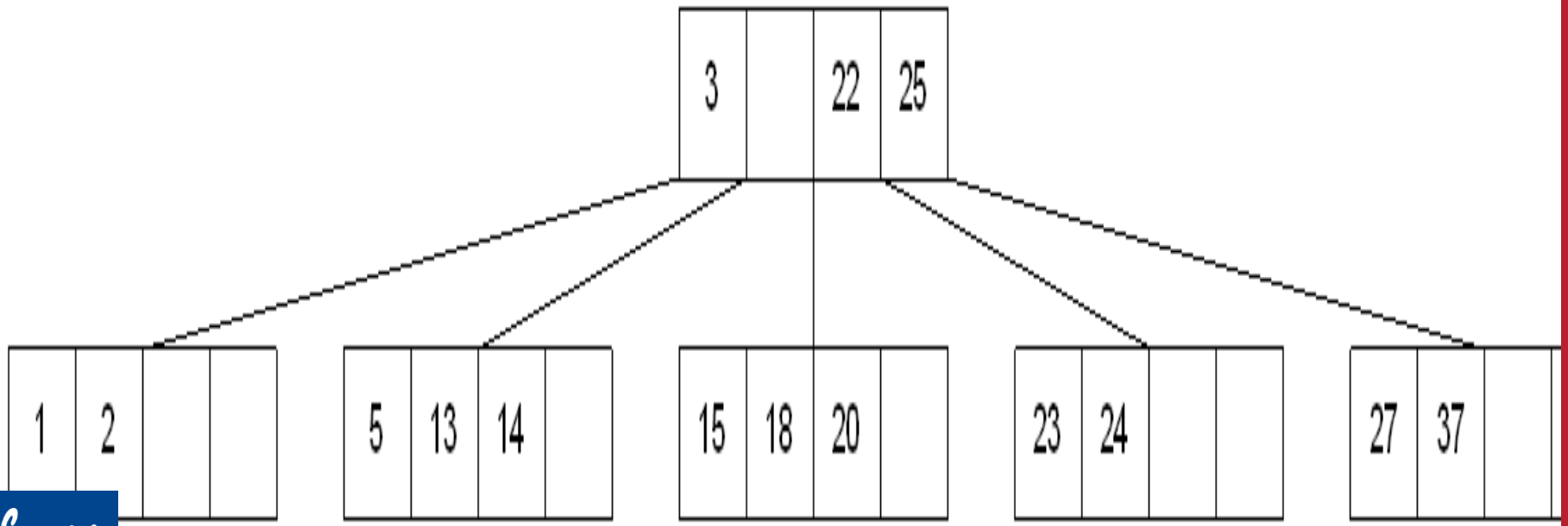
Case 2. Deletion from a non-leaf

- Deleting the successor results in:



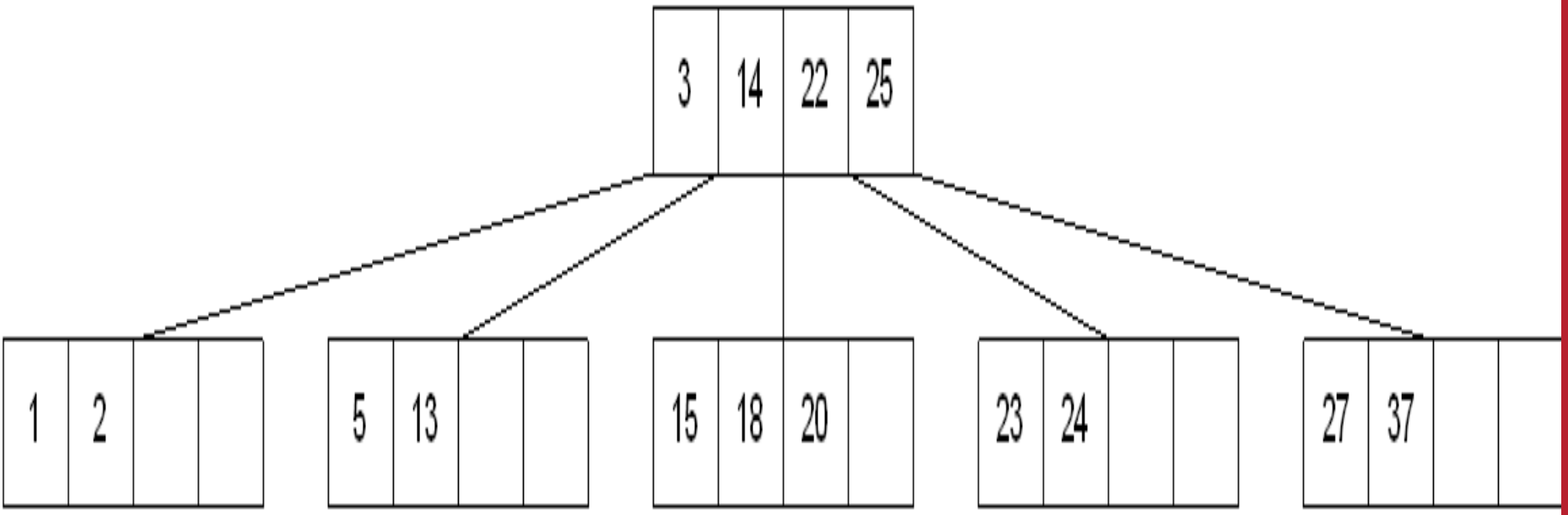
Case 2. Deletion from a non-leaf

- The vales in the left sibling are combined with the separator key (18) and the remaining values. They are divided between the 2 nodes:

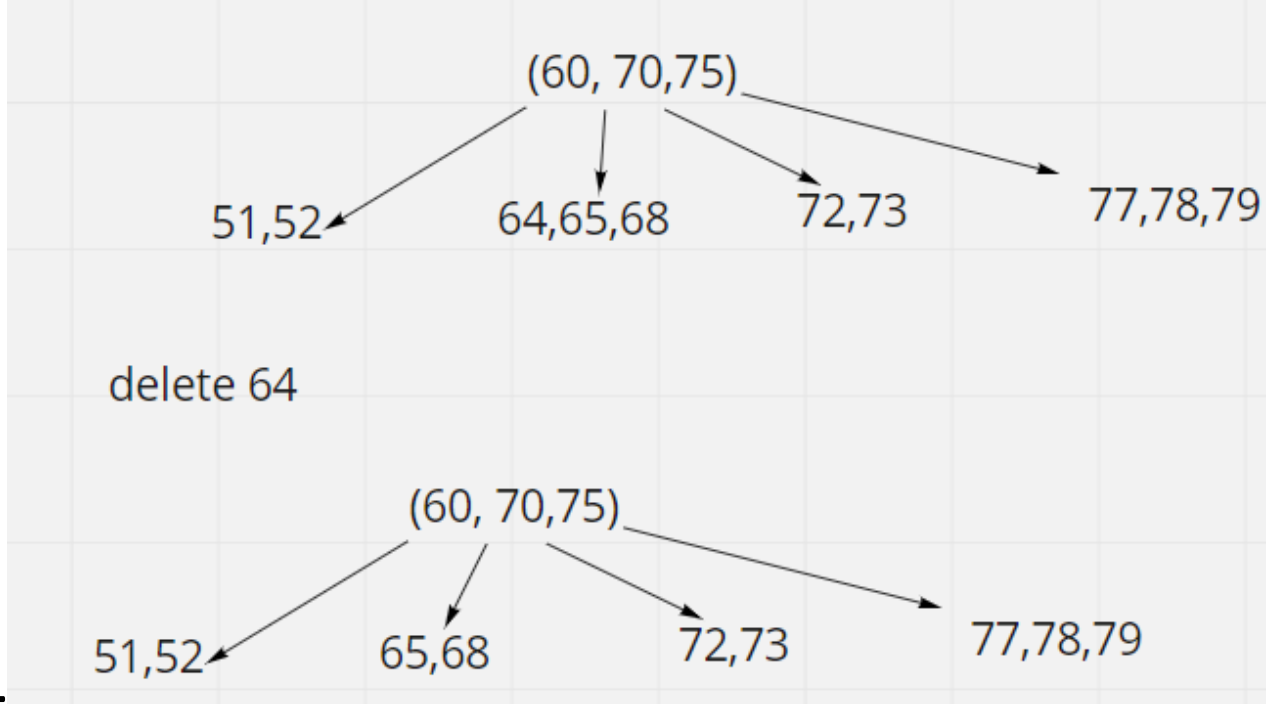


Case 2. Deletion from a non-leaf

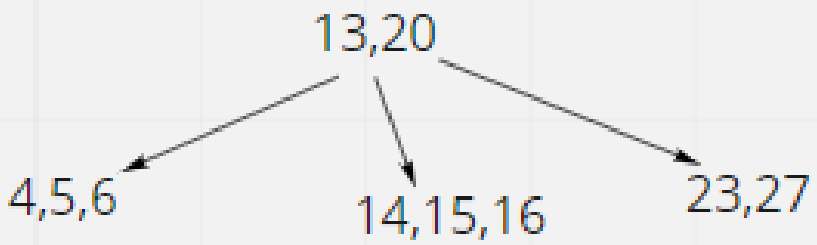
- and then the middle value is moved to the parent:



Some more deletion examples

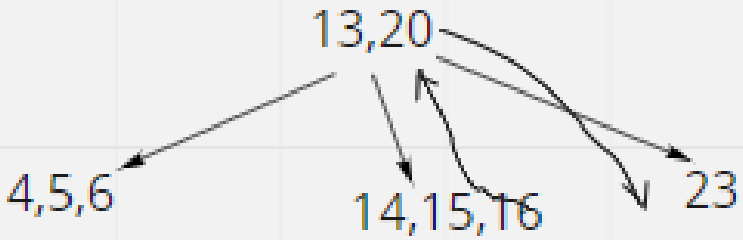


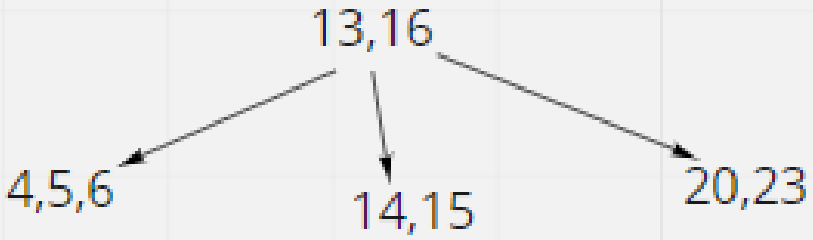
- 5-way tree.
- 5-way tree
- min keys=2
- max keys= 4
- max children node=5
- min children node= 3

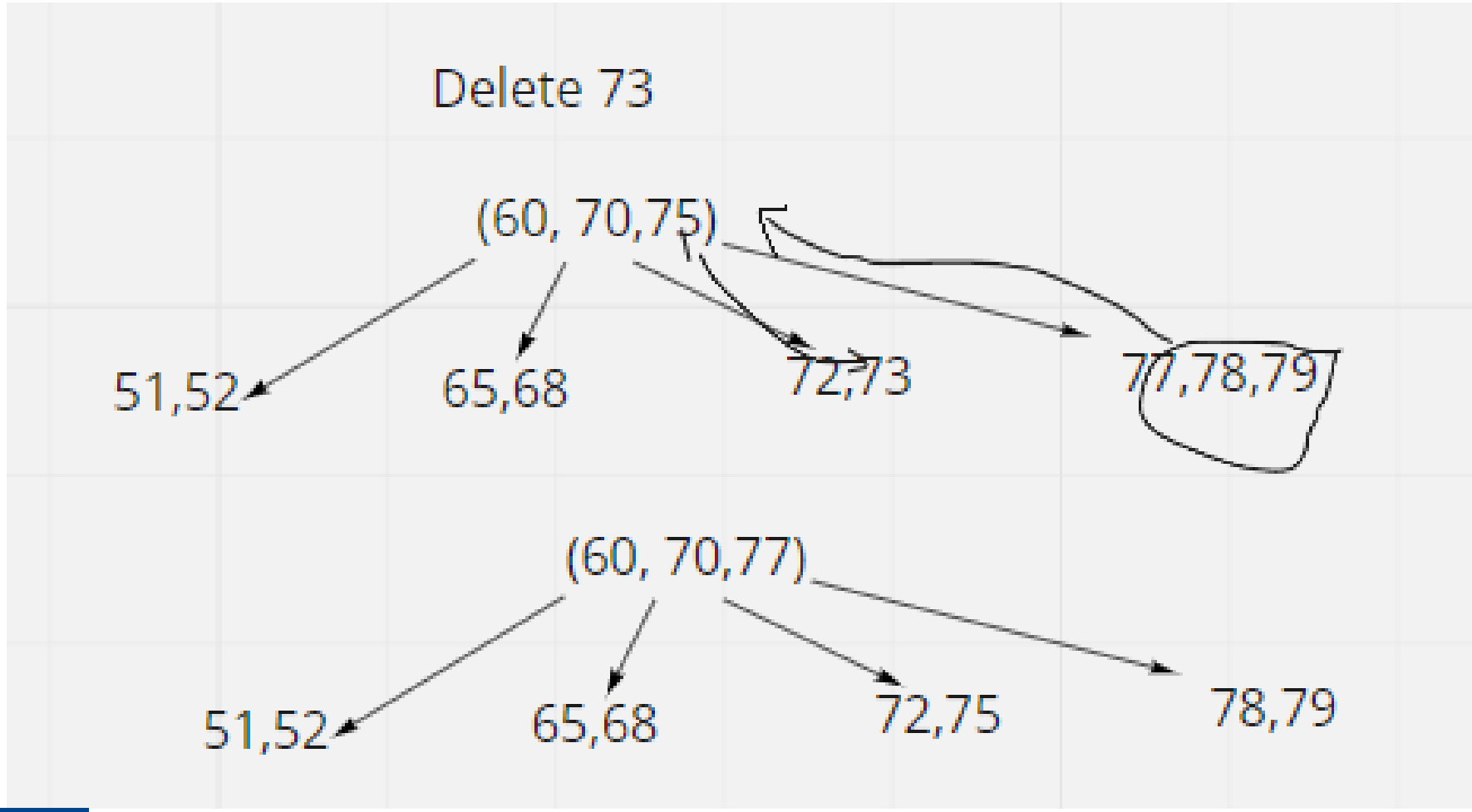


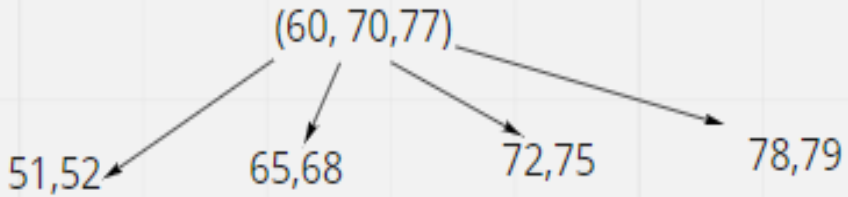
delete 27

target node has keys=m/2
solution: borrow from left or right immediate sibling if they have keys= >m/2



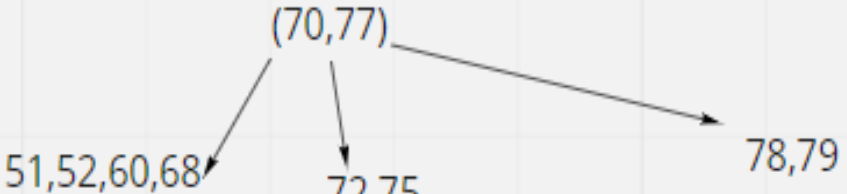



Delete 73
(60, 70,75)
51,52
65,68
72,73
77,78,79
(60, 70,77)
51,52
65,68
72,75
78,79

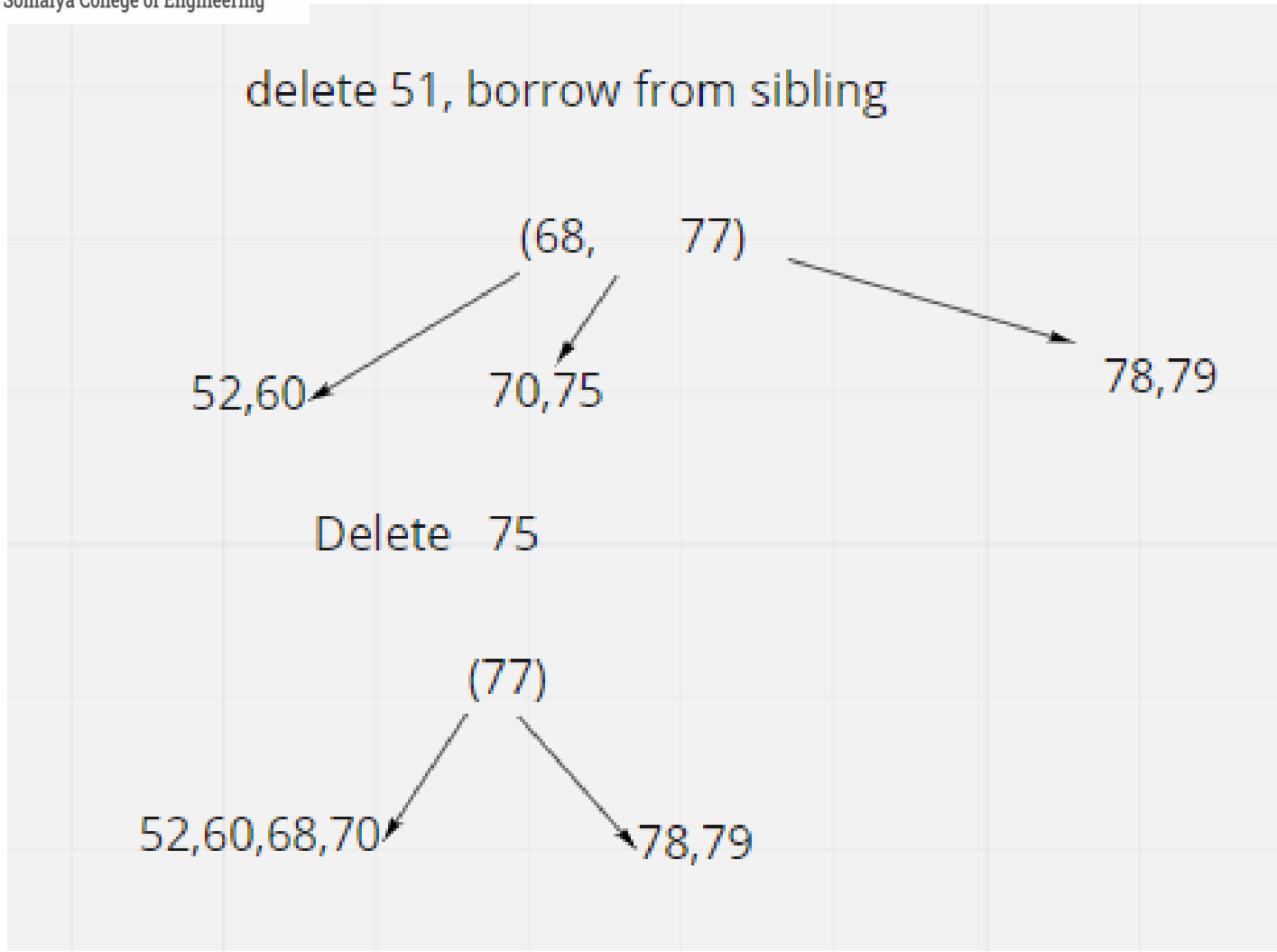


case: Delete from leaf node, none of the sibling have >m/2 no of keys
solution: Delete and merge with either of the sibling along with parent

Delete 65 merge with left sibling

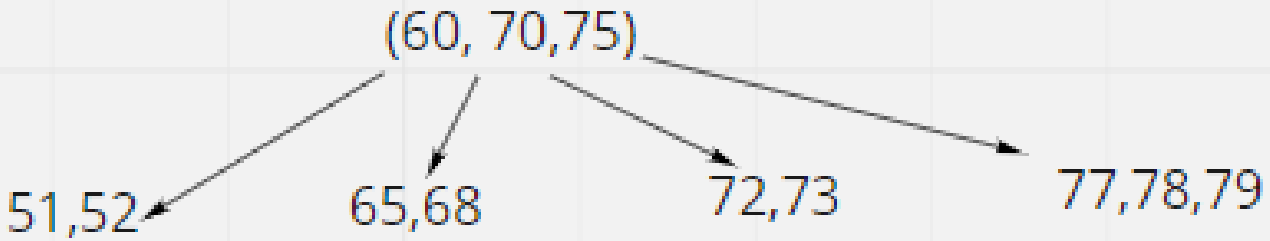


Delete 65, merge with right sibling

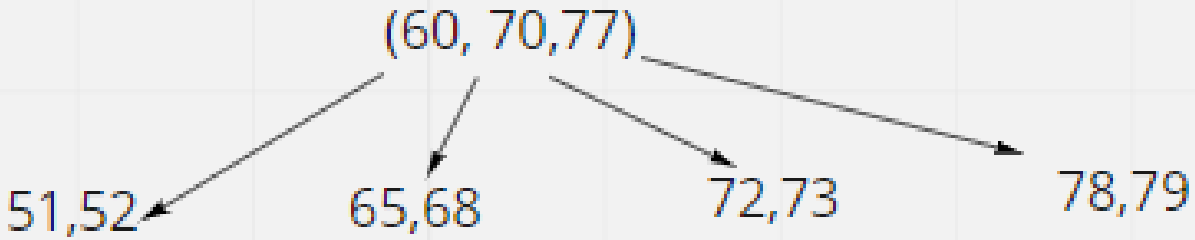

delete 51, borrow from sibling
(68, 77)
52,60
70,75
78,79
Delete 75
(77)
52,60,68,70
78,79

Deletion from non-leaf node

solution: Replace with inorder predecessor or successor



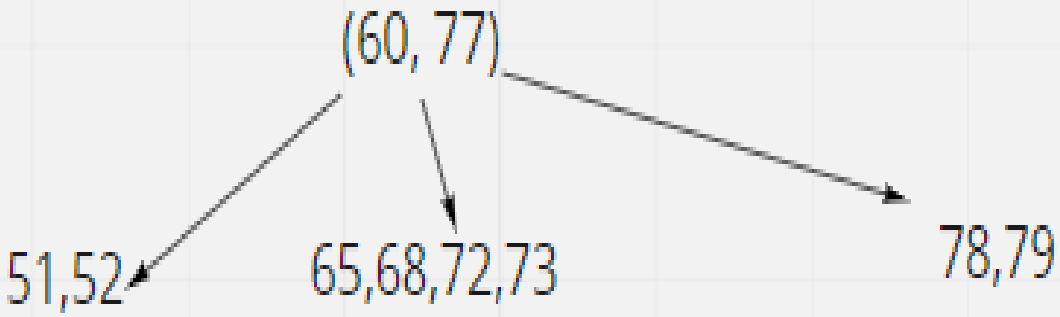
Delete 75 replce with inorder successor



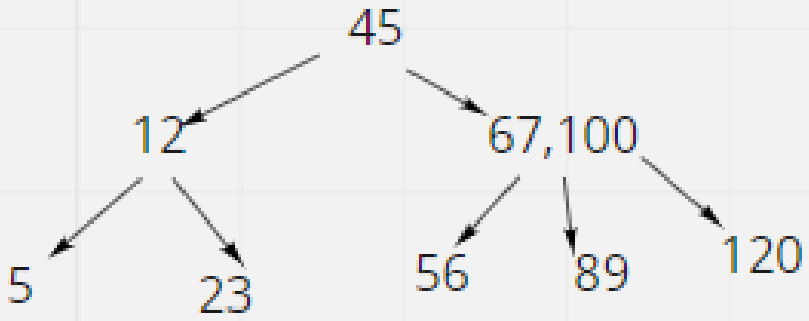
Delete 70

Replace with inorder successor/predecessor,

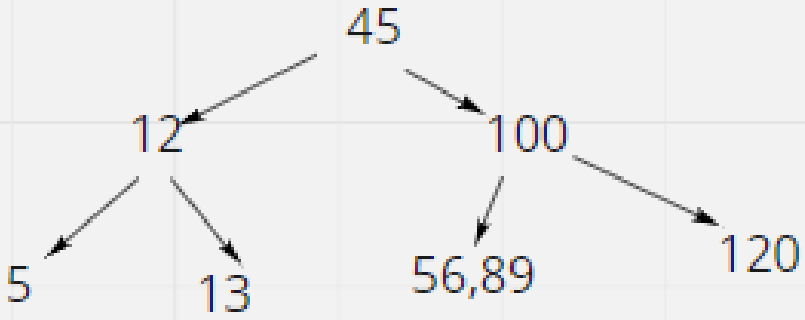
if no of keys in the node from where new key is borrowed = m/2 then merge



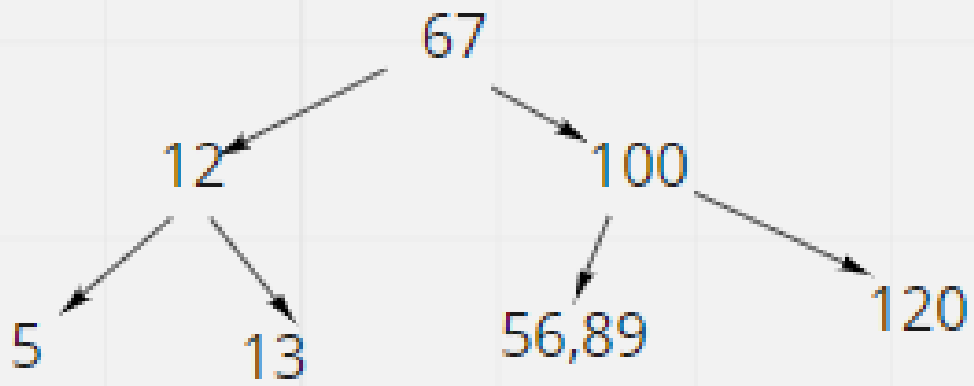
example



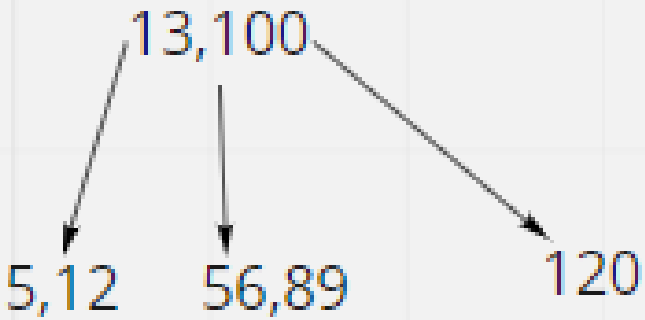
Delete 67



delete 45



delete 67



REFERENCES

- http://faculty.cs.niu.edu/~freedman/340/340notes/340multi.htm, last retrieved on Oct 27,2020

B-Tree Properties

- Properties
  - maximum branching factor of *M*
  - the root has between 2 and *M* children *or* at most *M-1* keys
  - All other nodes have between $\lceil M/2 \rceil$ and M records
    - Keys+data
- Result
  - tree is O(log M) deep
  - all operations run in O(log M) time
  - operations pull in about *M* items at a time

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

What is a B+ Tree?

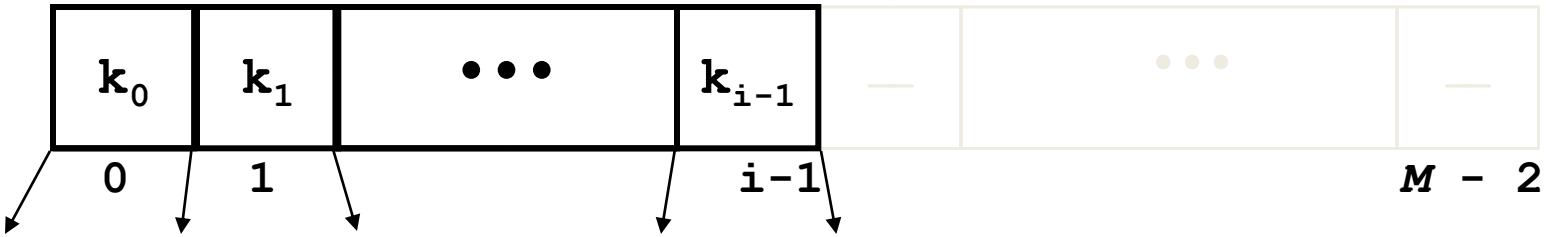
- A variation of B trees in which
  - internal nodes contain only search keys (no data)
  - Leaf nodes contain pointers to data records
  - Data records are in sorted order by the search key
  - All leaves are at the same depth

Definition of a B+Tree

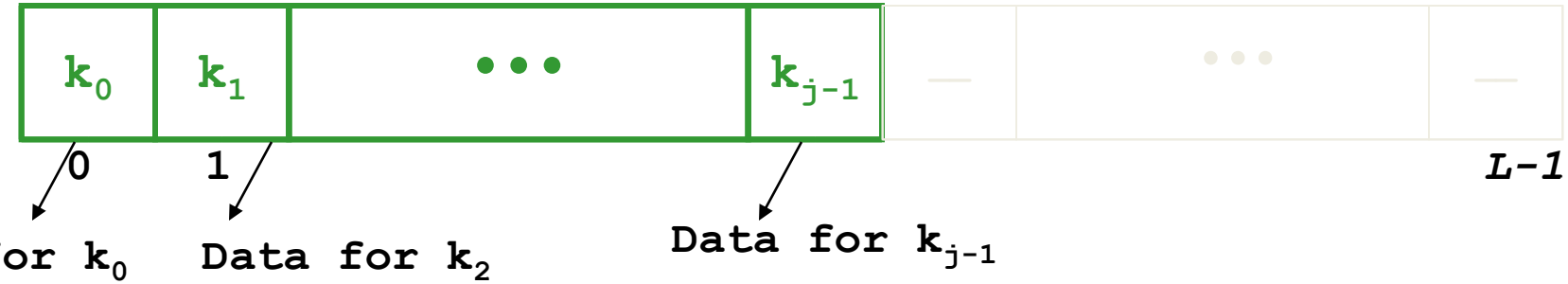
A B+ tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length, and each non-leaf node of the tree has between [M/2] and [M] children, where n is fixed for a particular tree.

B+ Tree Nodes

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



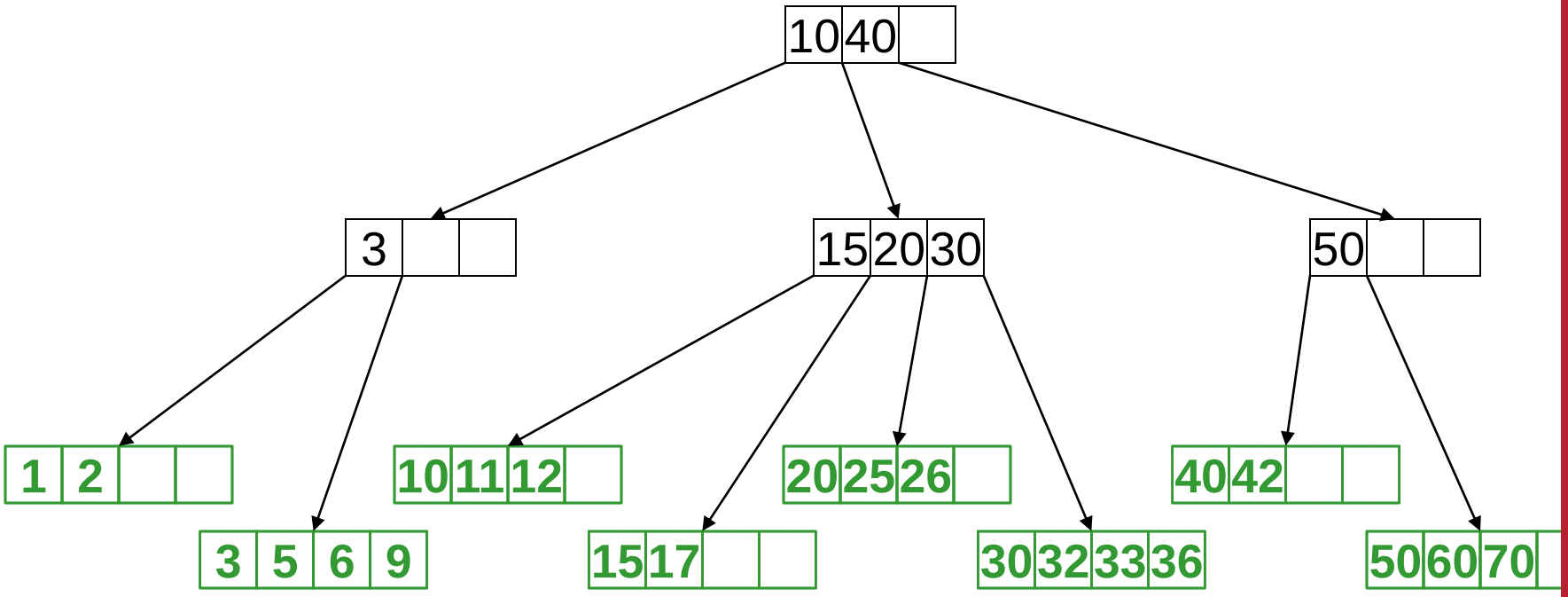
- Leaf
  - (Key, DataPointer)* L in each node
  - first j Keys currently in use



Example

B+ Tree with $M = 4$

Often, leaf nodes linked together



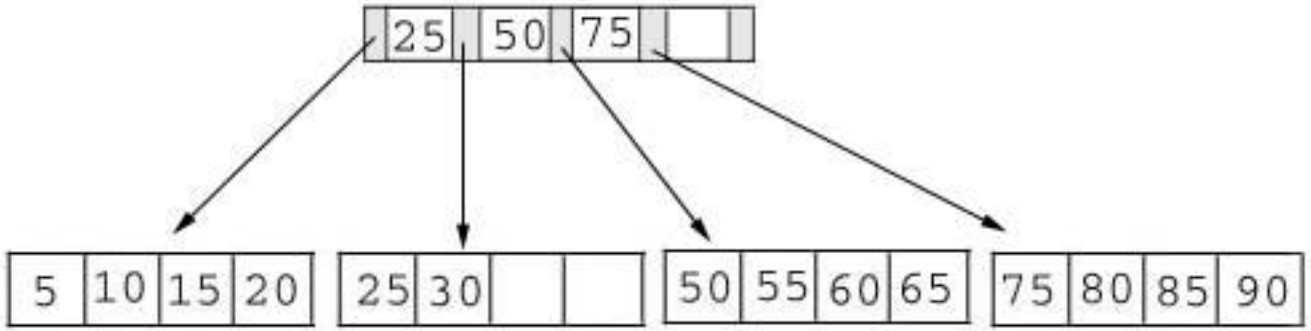
Advantages of B+ tree usage for databases

- 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.

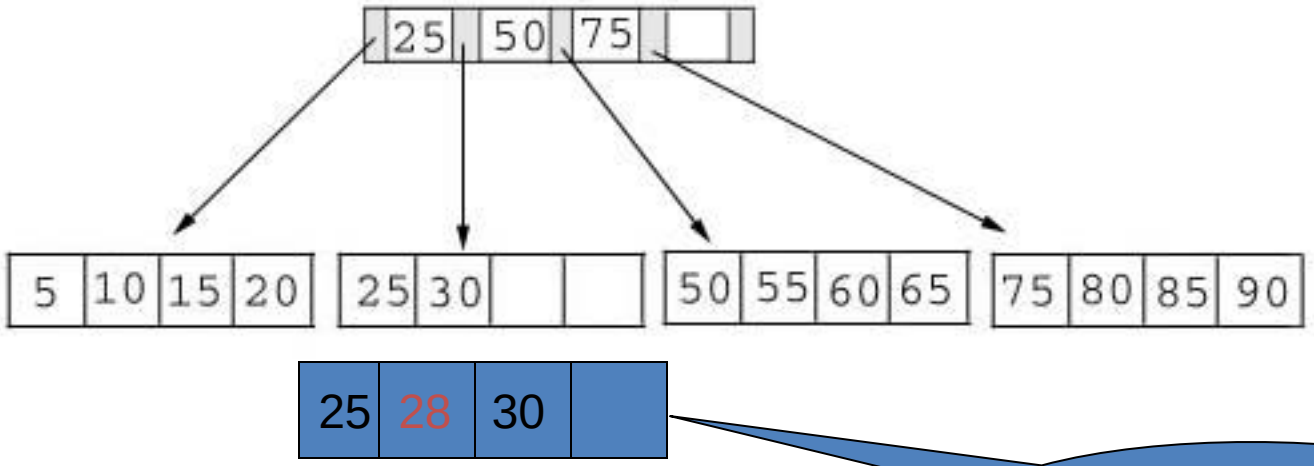


Searching

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

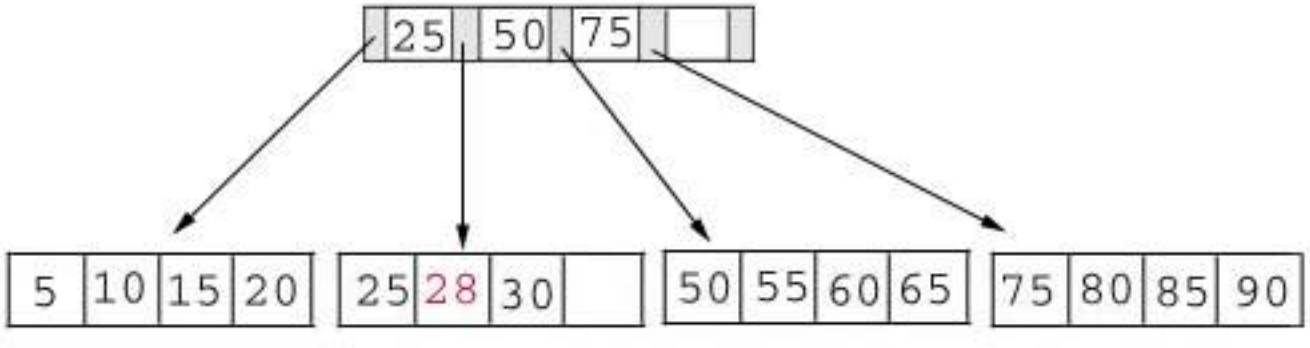
Insertion

- 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.



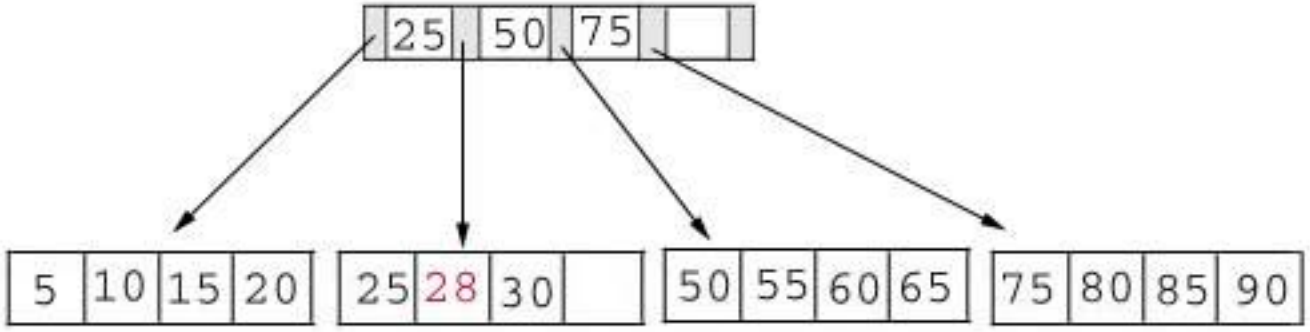
Insertion

- Result:



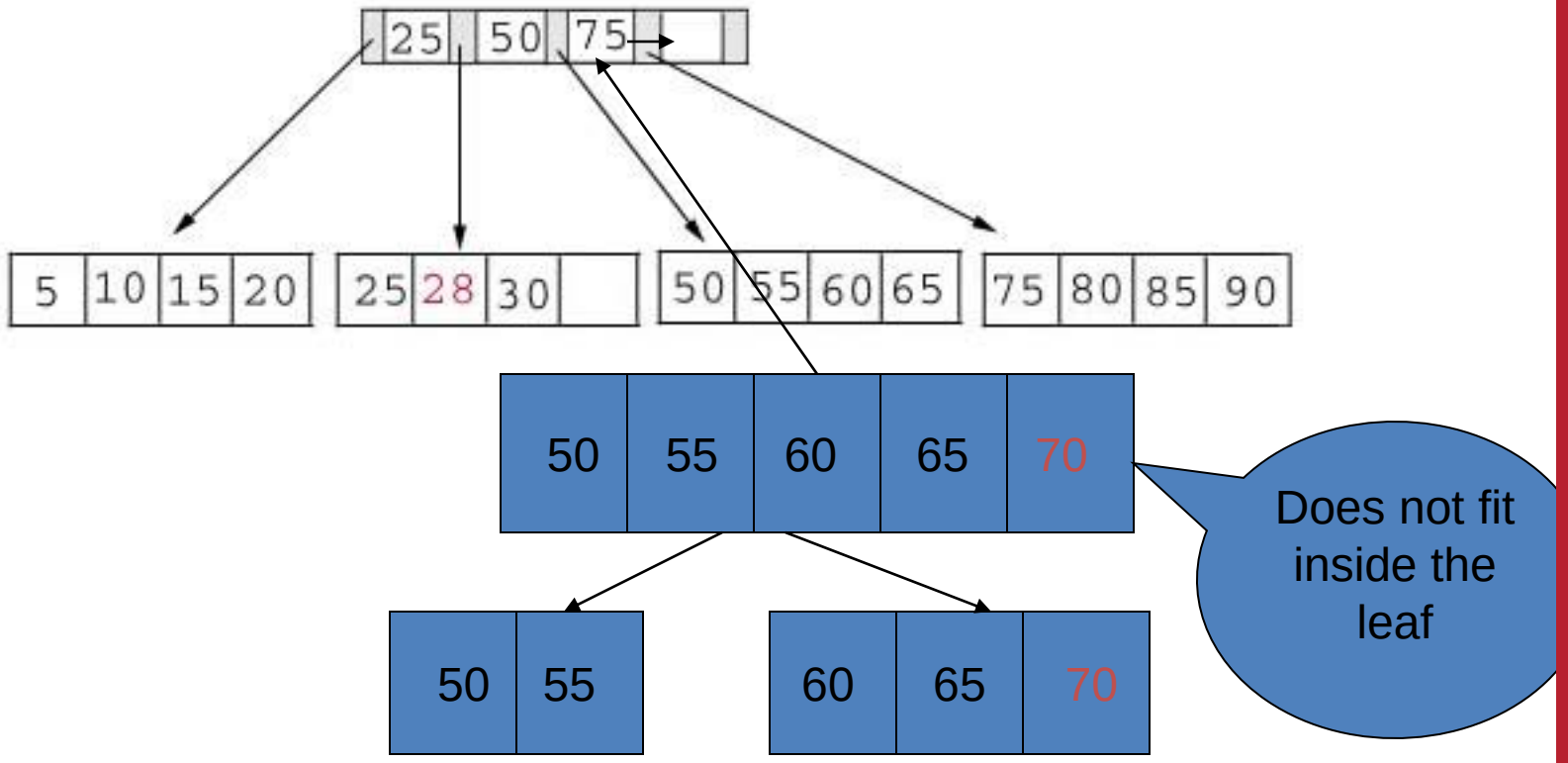
Insertion

- Example #2: insert 70 into below tree



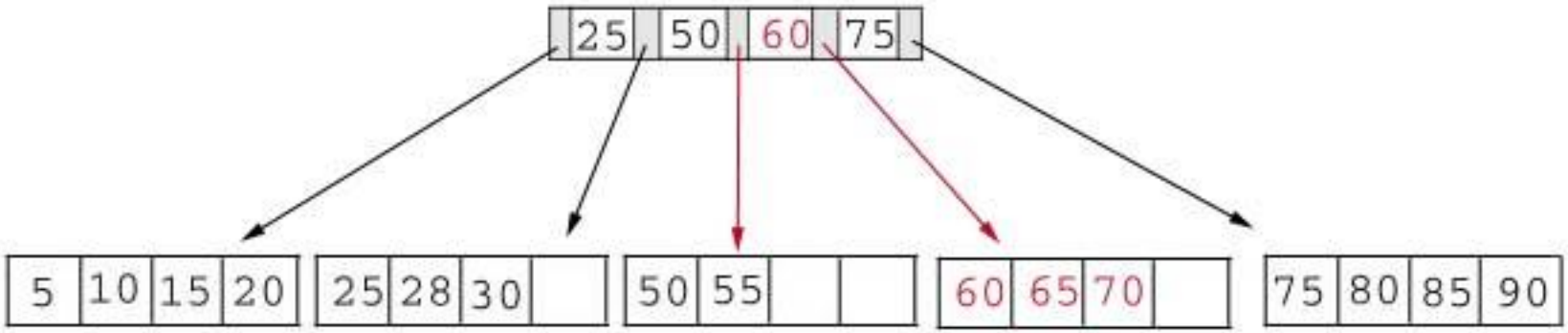
Insertion

- 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.



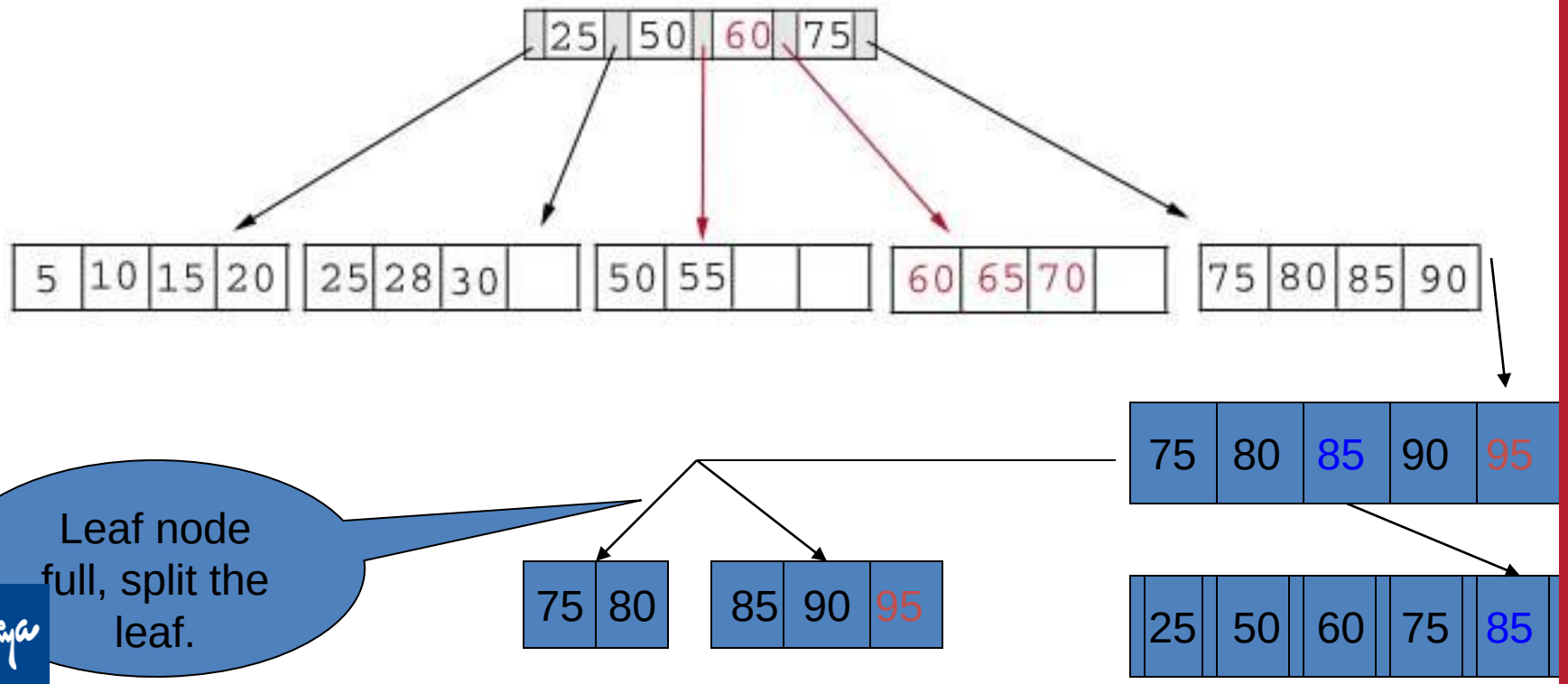
Insertion

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.

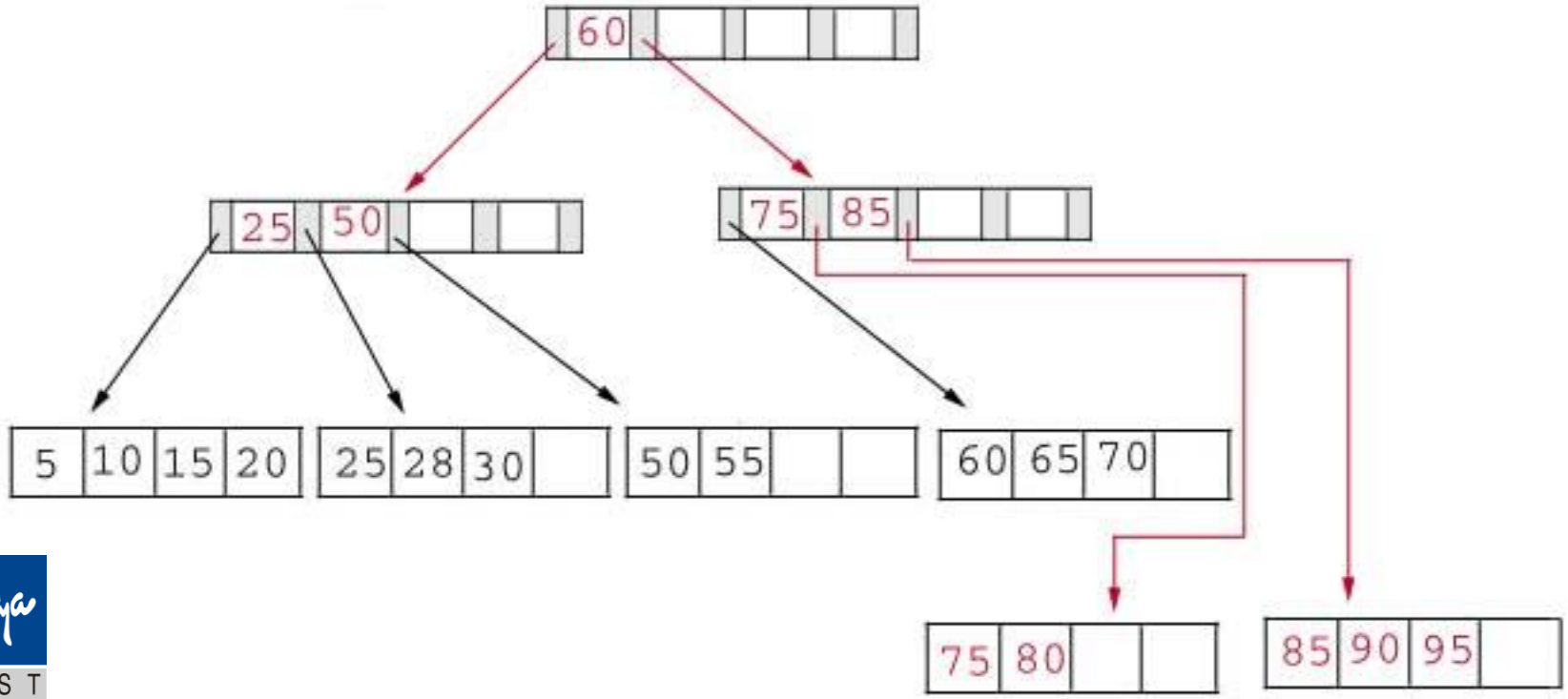
Insertion

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



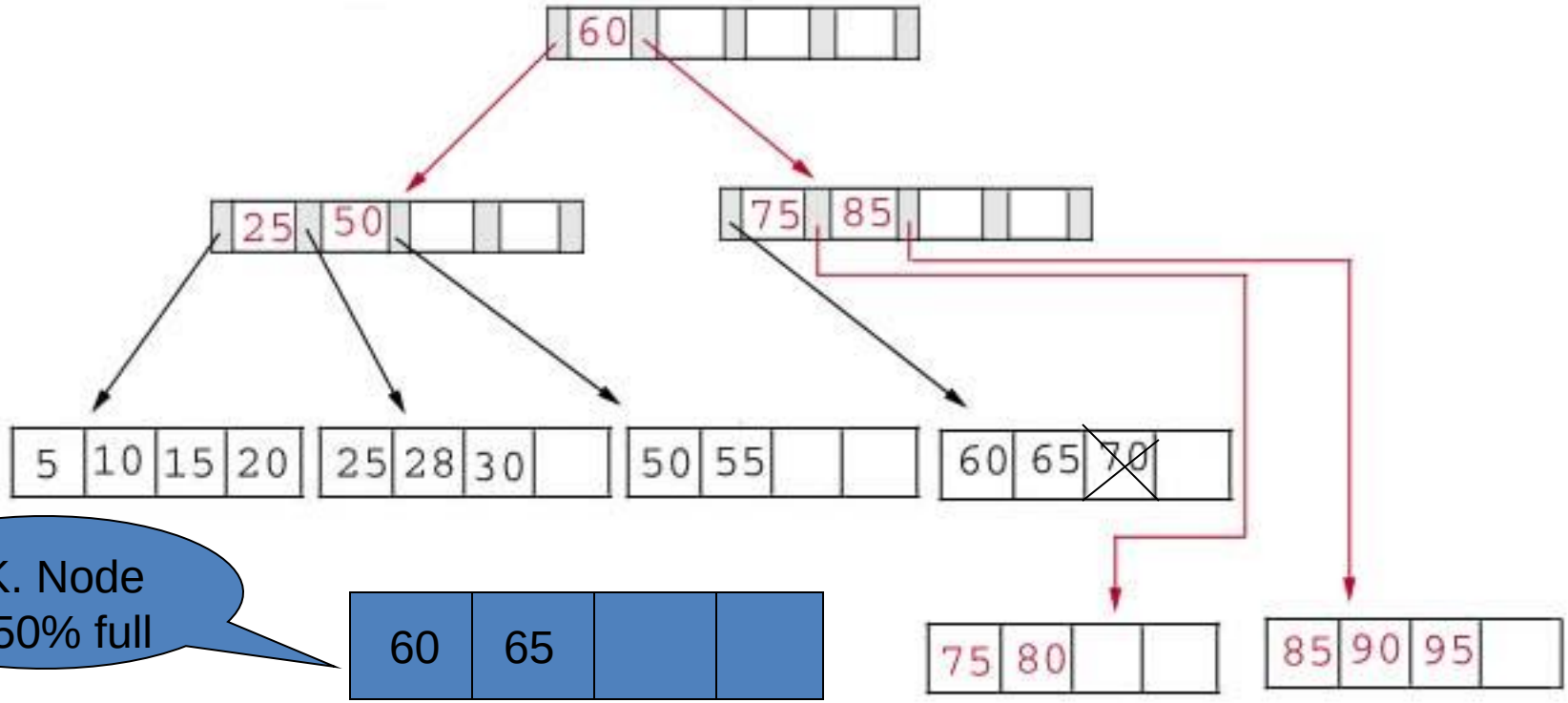
Insertion

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



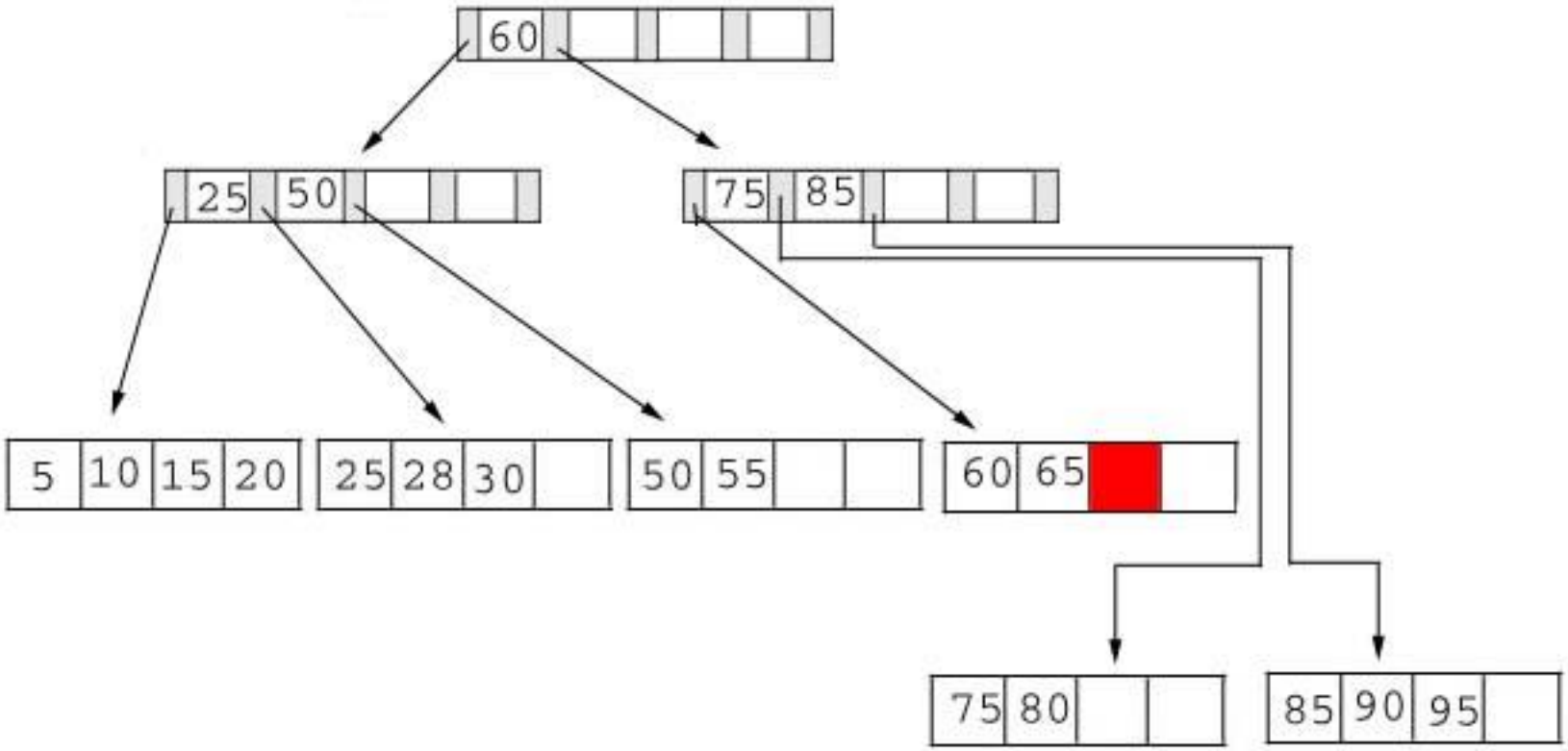
Deletion

- 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



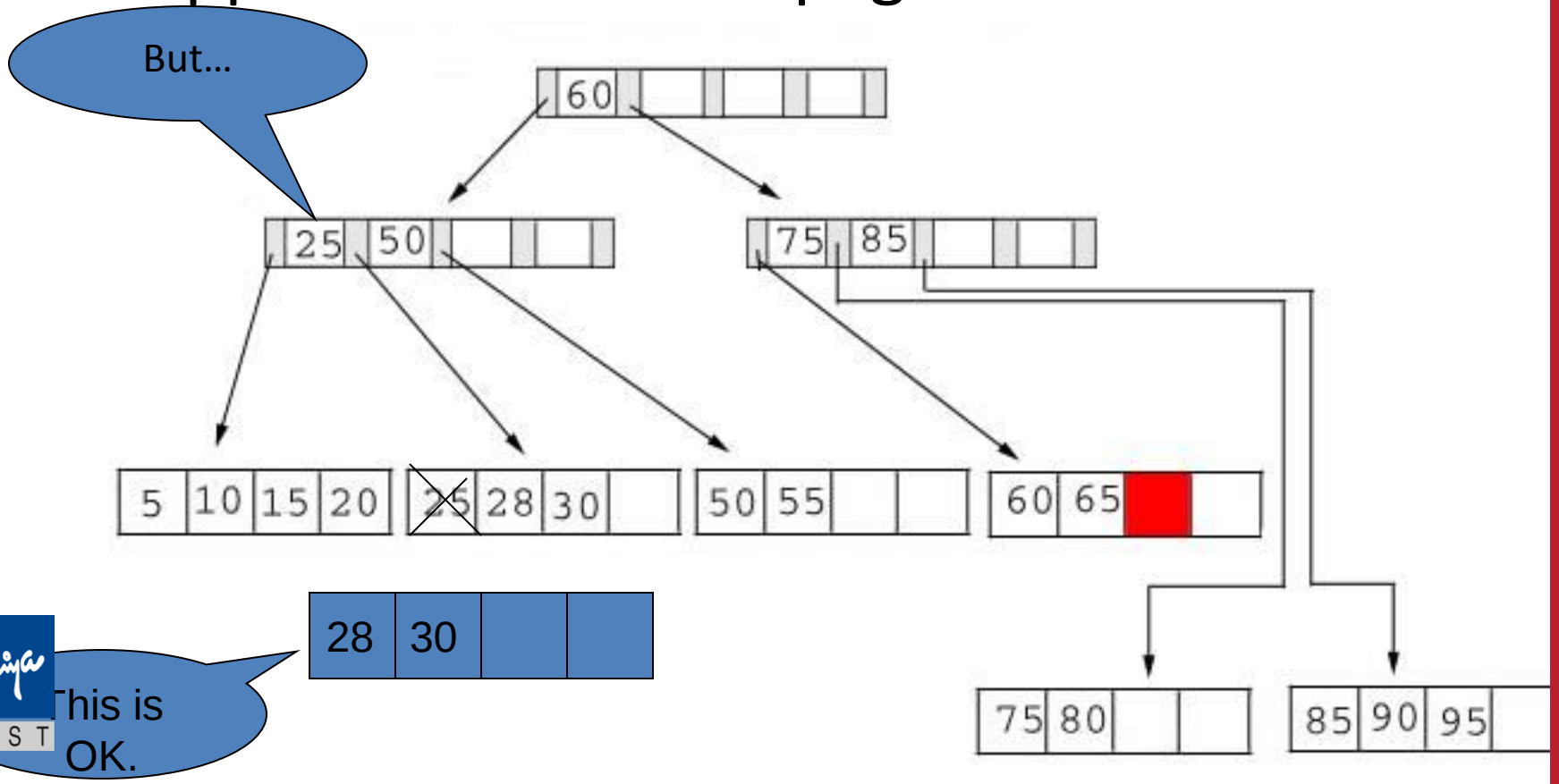
Deletion

- Result:



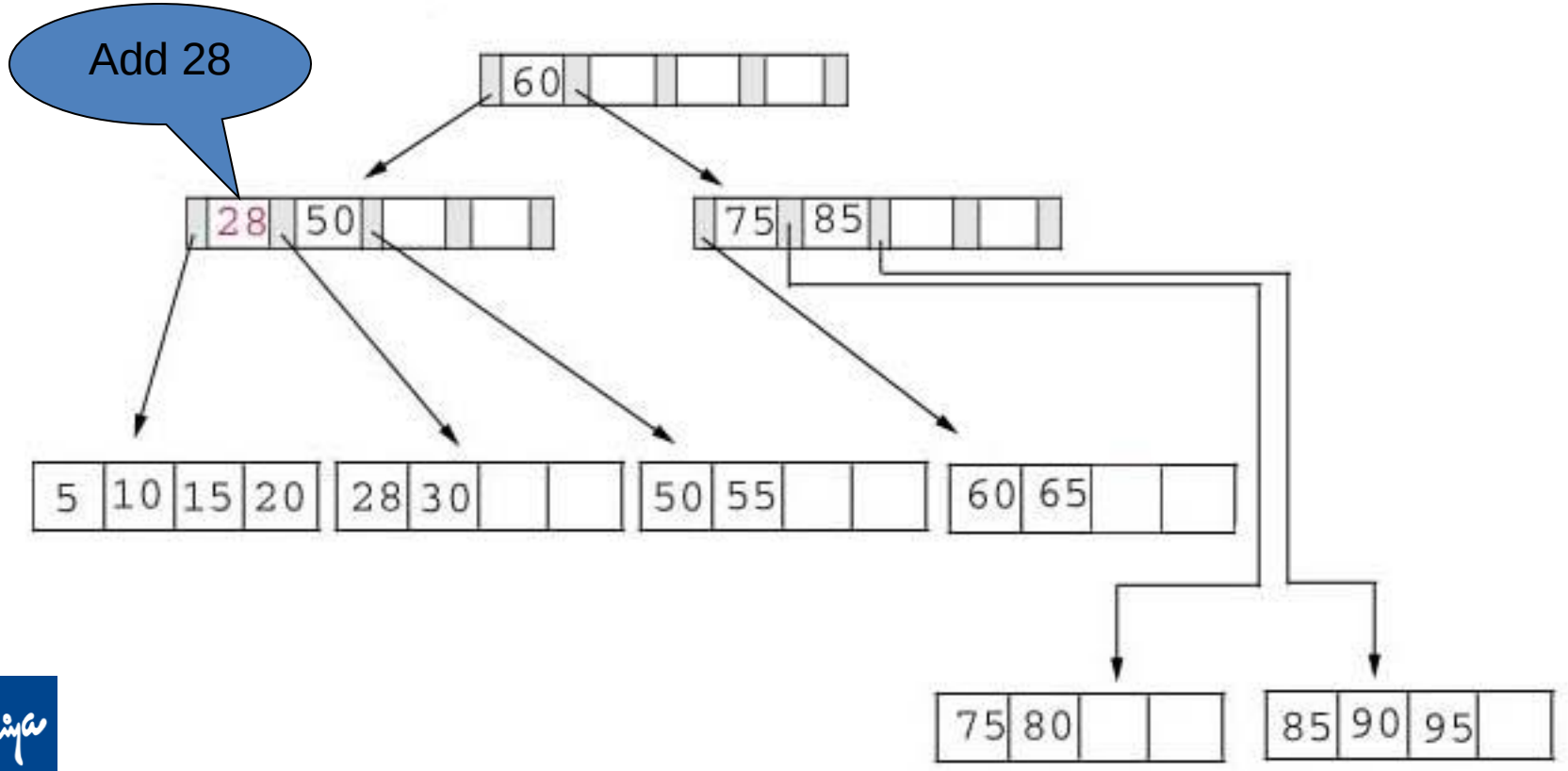
Deletion

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



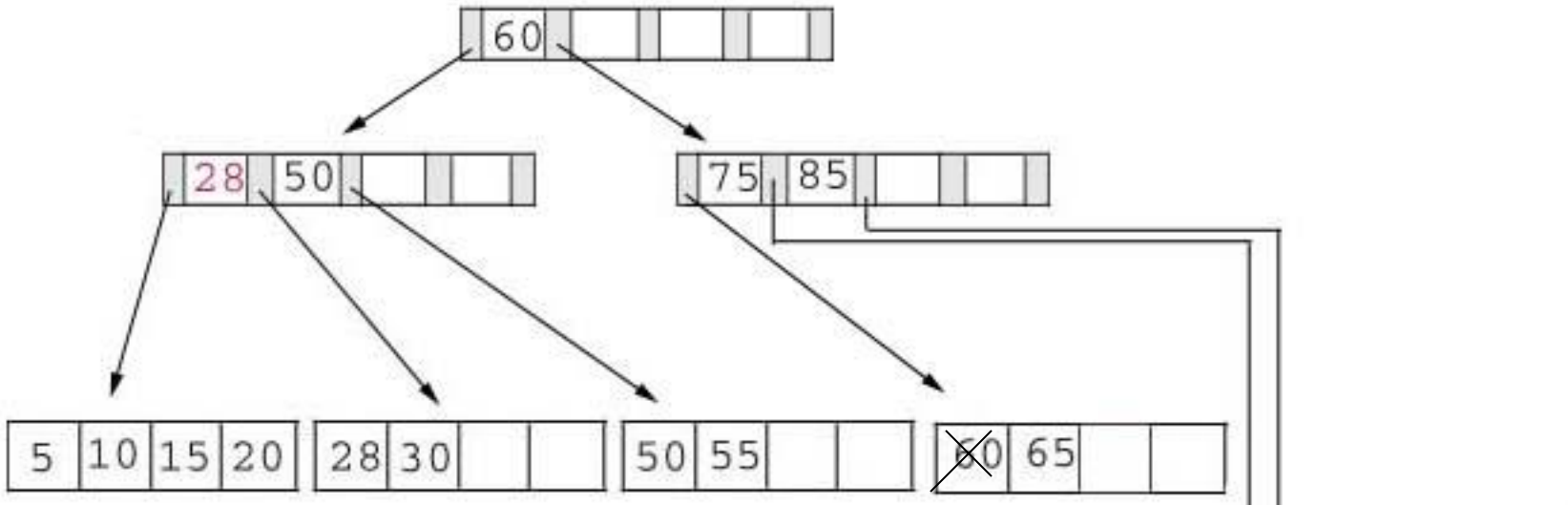
Deletion

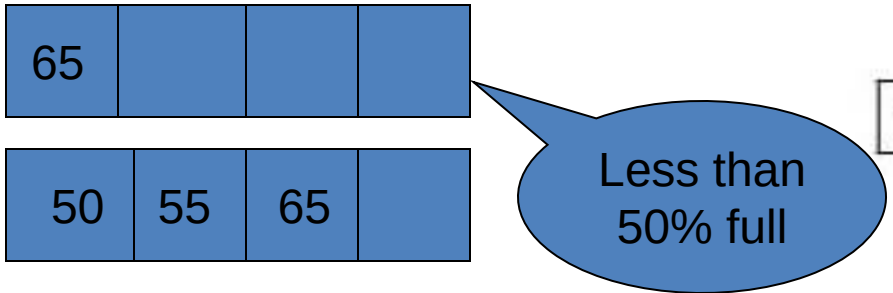
- Result: replace 28 in the index page.



Deletion

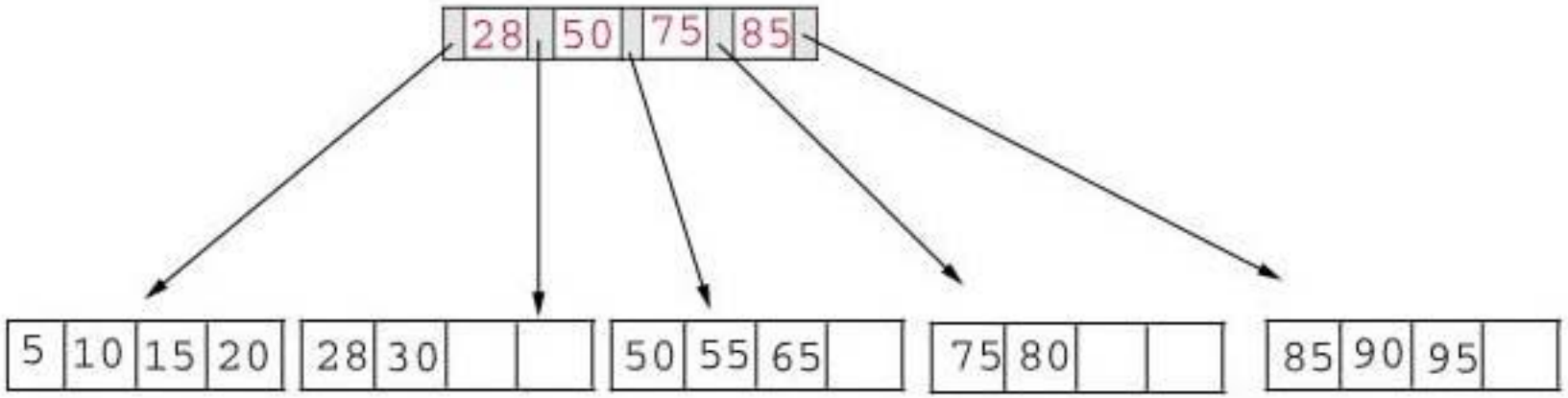
- Example #3: delete 60 from the below tree





Deletion

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



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.

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

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Queries?

Somaiya TRUST

SOMAIYA VIDYAVIHAR UNIVERSITY

K J Somaiya College of Engineering

Thank you!

Somaiya TRUST