BBM371- Data Management

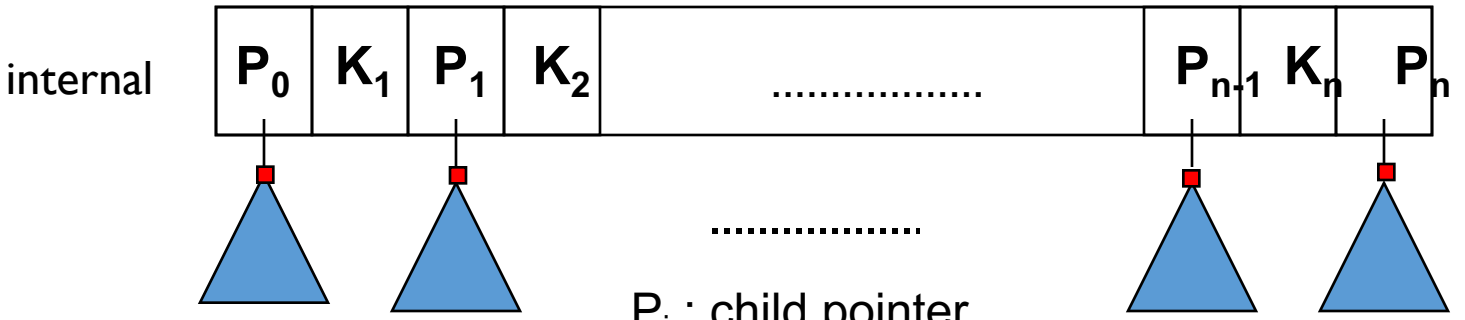
Lecture 9 – B+Trees

6.12.2018

Main Characteristics of a B+ tree

- B+ tree is always balanced.
- A minimum occupancy of 50 percent is guaranteed for each node except the root node.
- Searching for a record requires just a traversal from the root to the appropriate leaf.

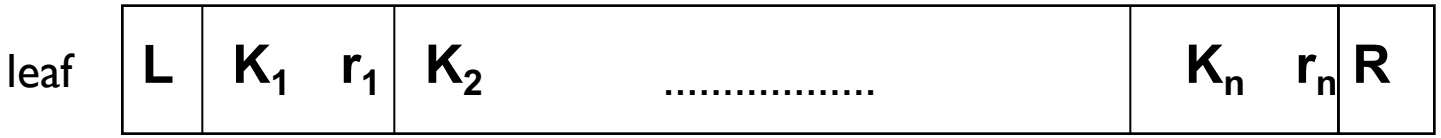
B+ Tree: node structure



P_i : child pointer

K_i : search key

# of search key = n, # of pointers= n+1

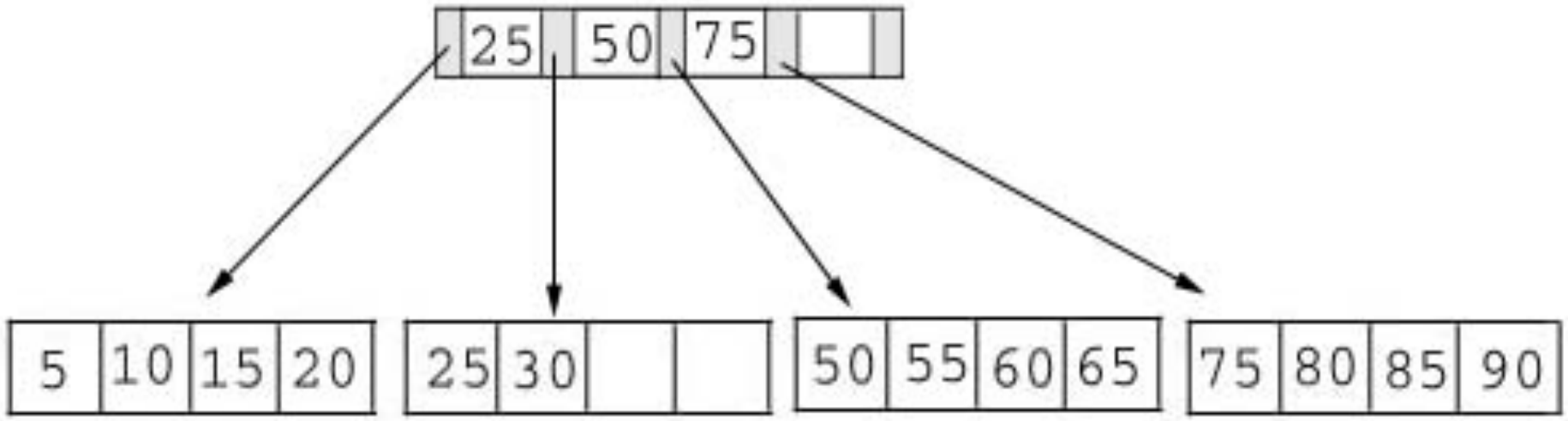


L, R : pointers to left and rights neighbours

More about B+ Tree

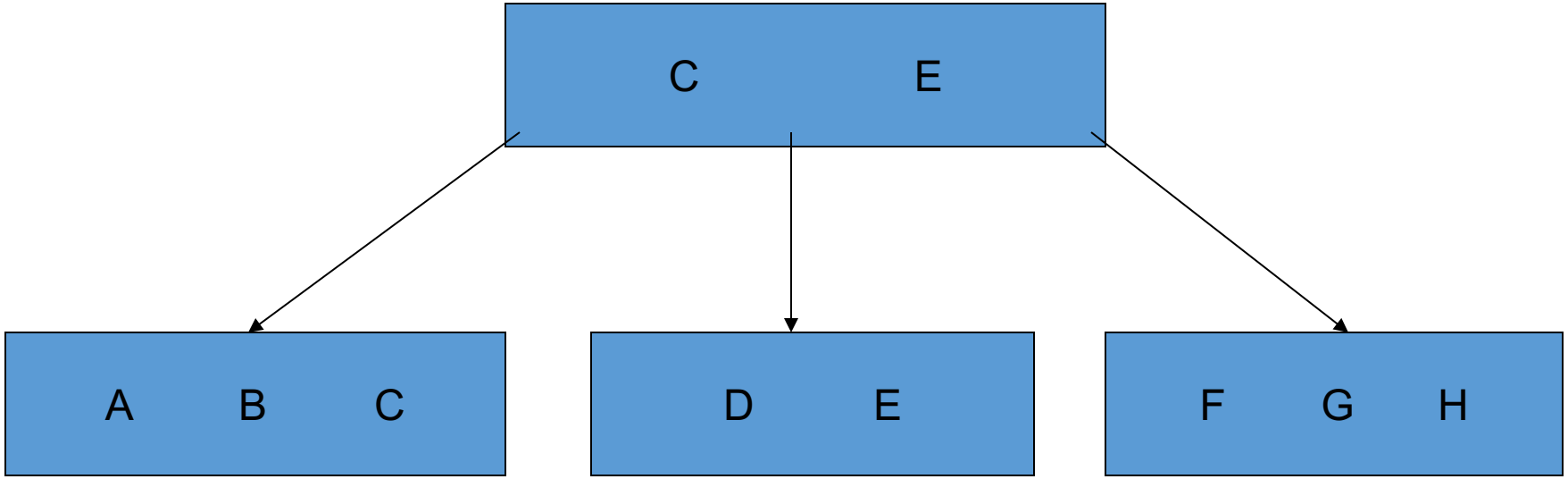
- Only leaf nodes contain data entries.
- Internal nodes contain indexes.
- Every data entry must appear in a leaf page.
- Leaf nodes are doubly linked to each other.

What is a B+ Tree



What is a B+ Tree

- Question: Is this a valid B+ Tree?



What is a B+ Tree

Answer:

1. Both trees in slide 3 and slide 4 are valid; how you store data in B+ Tree depends on your algorithm when it is implemented

2. As long as the number of data in each leaf are balanced, it doesn't matter how much data you stored in the leaves.

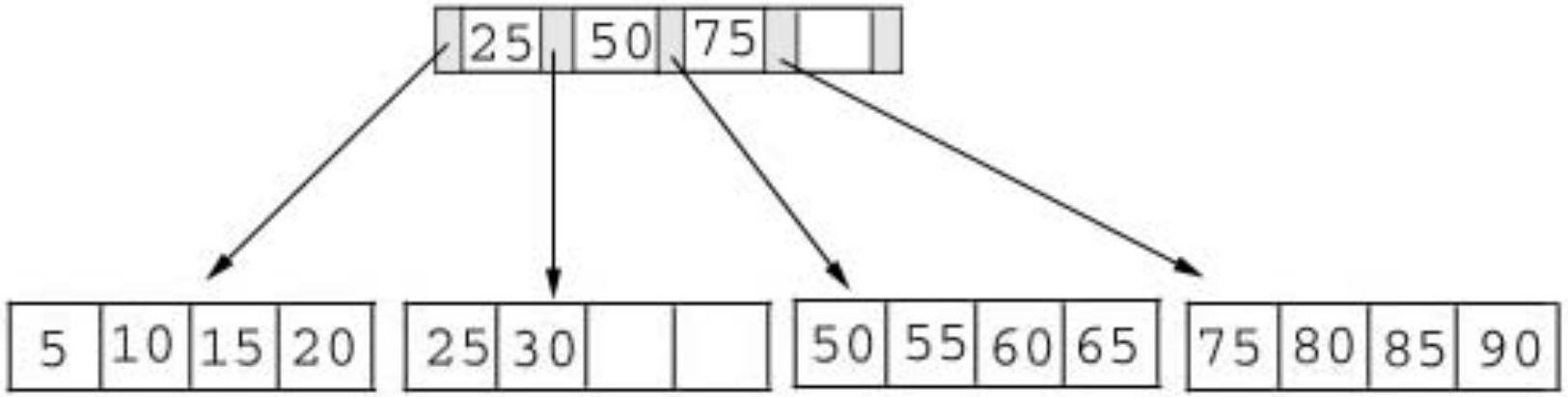
Searching

func *tree_search* (nodepointer, search key value K) **returns** nodepointer
// Searches tree for entry
if *nodepointer is a leaf, return nodepointer;
else,
    if $K < K_1$ then return tree_search(P_0, K);
    else,
        if $K \geq K_m$ then return tree_search(P_m, K);   // m = # entries
        else,
            find i such that $K_i \leq K < K_{i+1}$;
            return tree_search(P_i, K)
endfunc

Searching

- Since no structure change in a B+ tree during a searching process, so just compare the key value with the data in the tree, then give the result back.

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



Searching

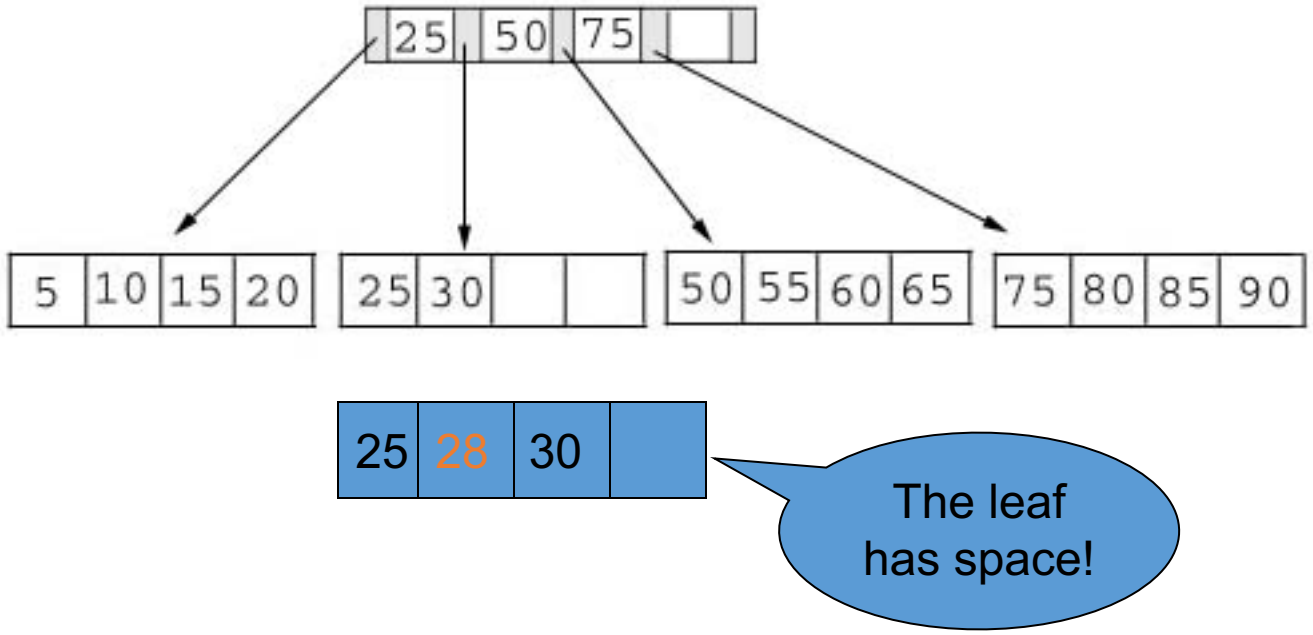
- Result:
  1. For the value of 45, not found.
  2. For the value of 15, return the position where the pointer located.

Insertion

- An entry is inserted at a leaf node that is searched from the root to the leaf node for an appropriate position.
- If there is not enough space in the leaf node, the node is split and the middle entry is copied into its parent.
- When an internal node is split, the middle entry is moved to the parent without copying it to the current node.

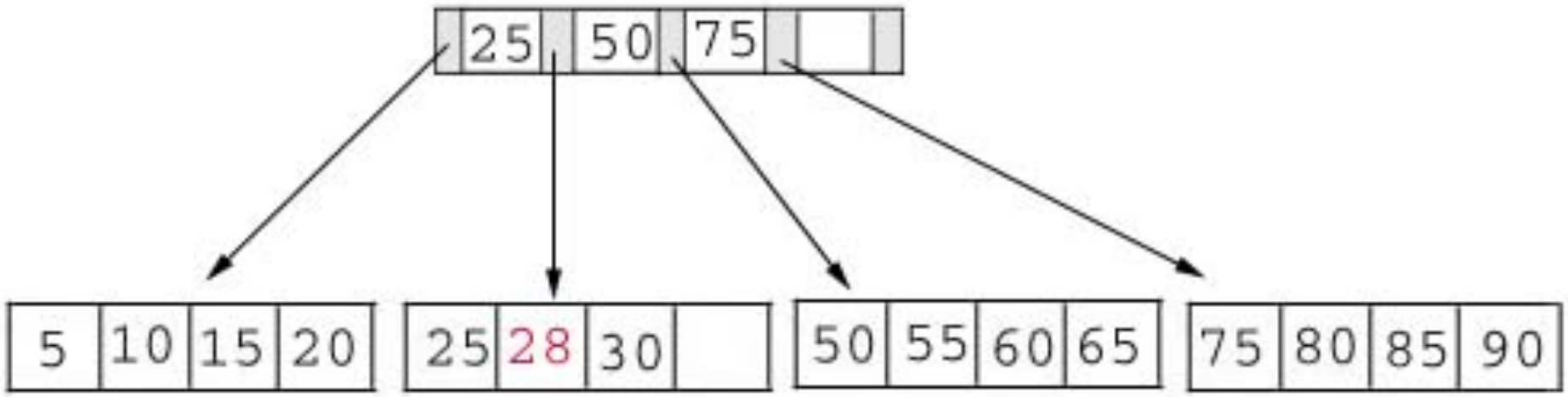
Insertion

- Since inserting a value into a B+ tree may cause the tree to be unbalanced, so rearrange the tree if needed.
- Example #1: insert 28 into the below tree.



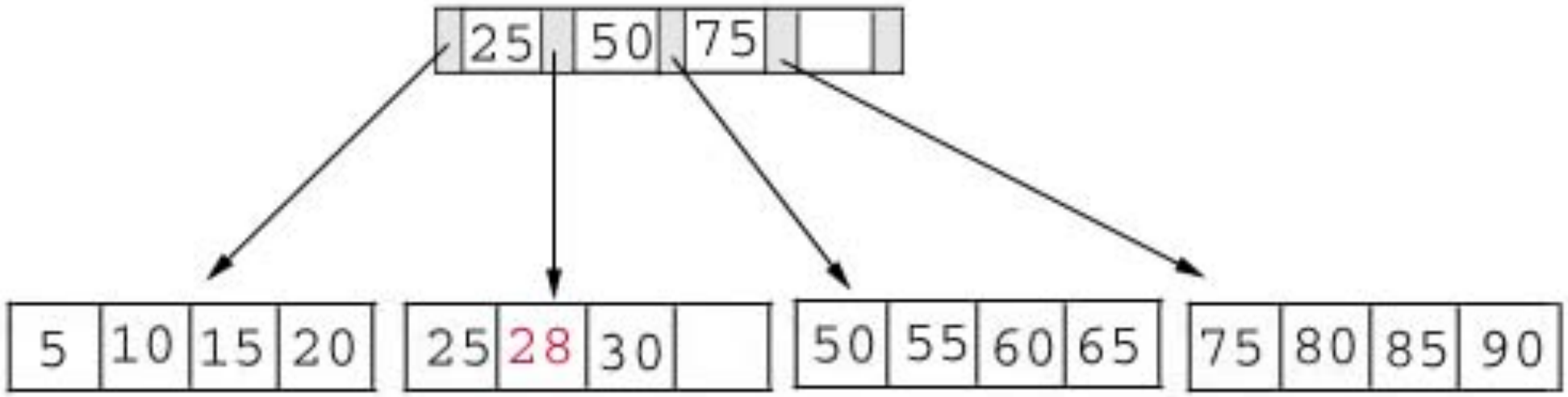
Insertion

- Result:



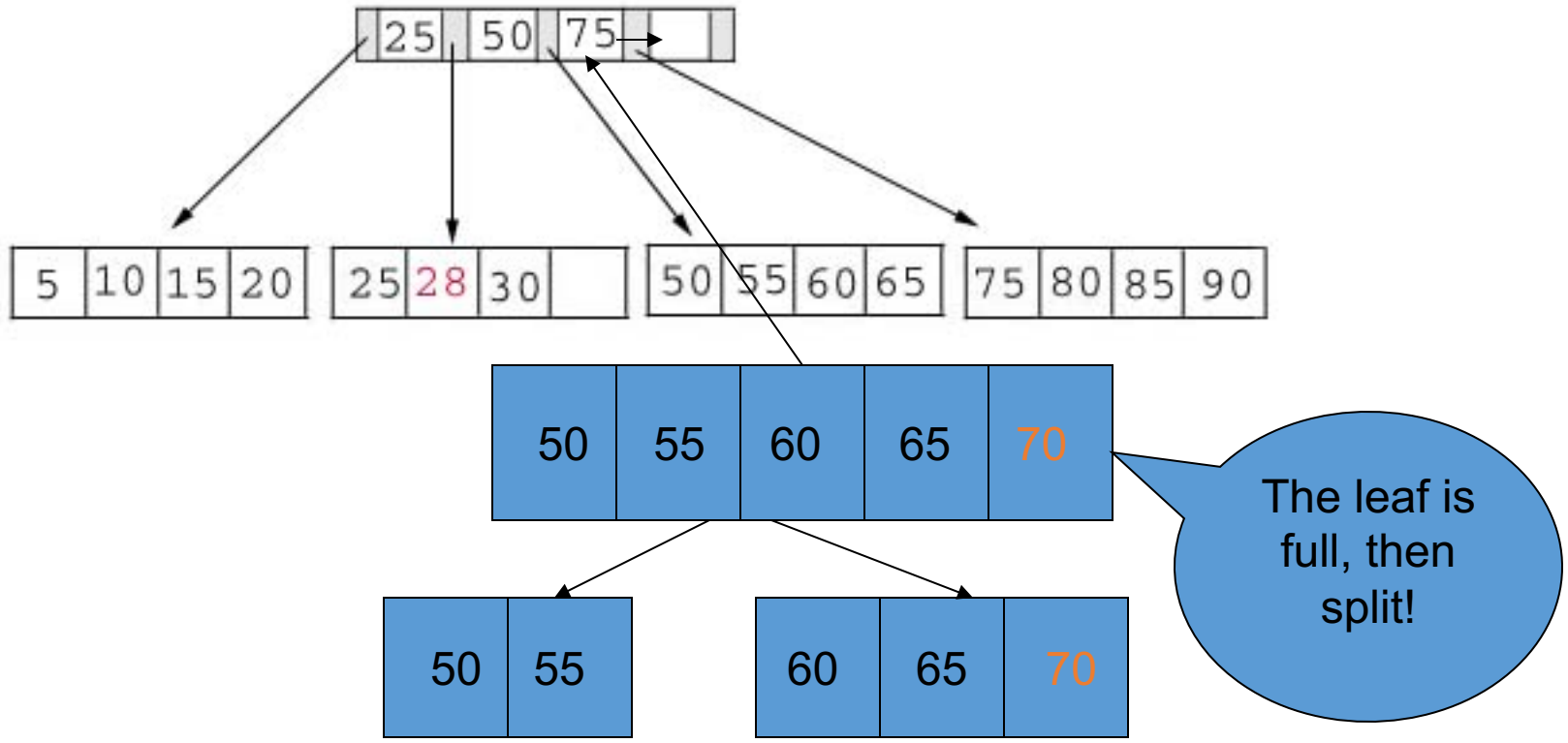
Insertion

- Example #2: insert 70 into below tree



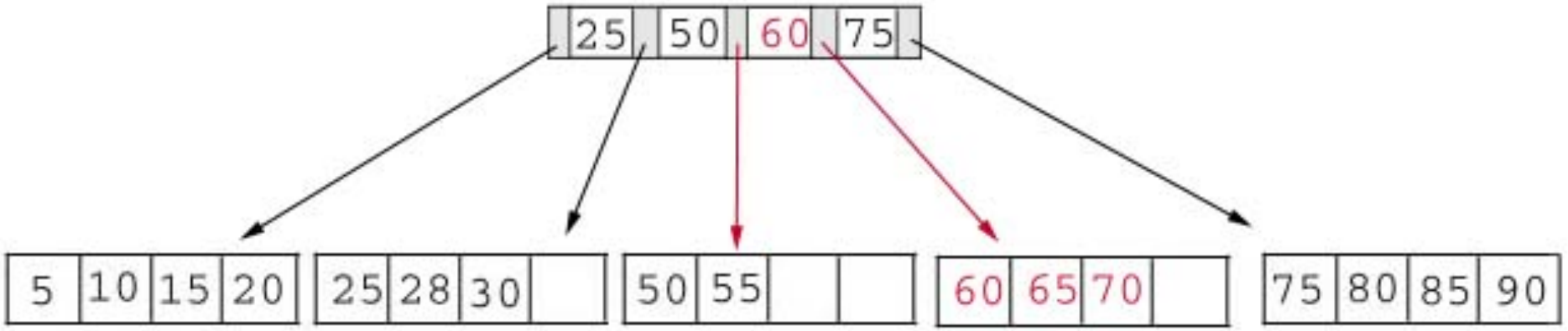
Insertion

- Process: split the tree



Insertion

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



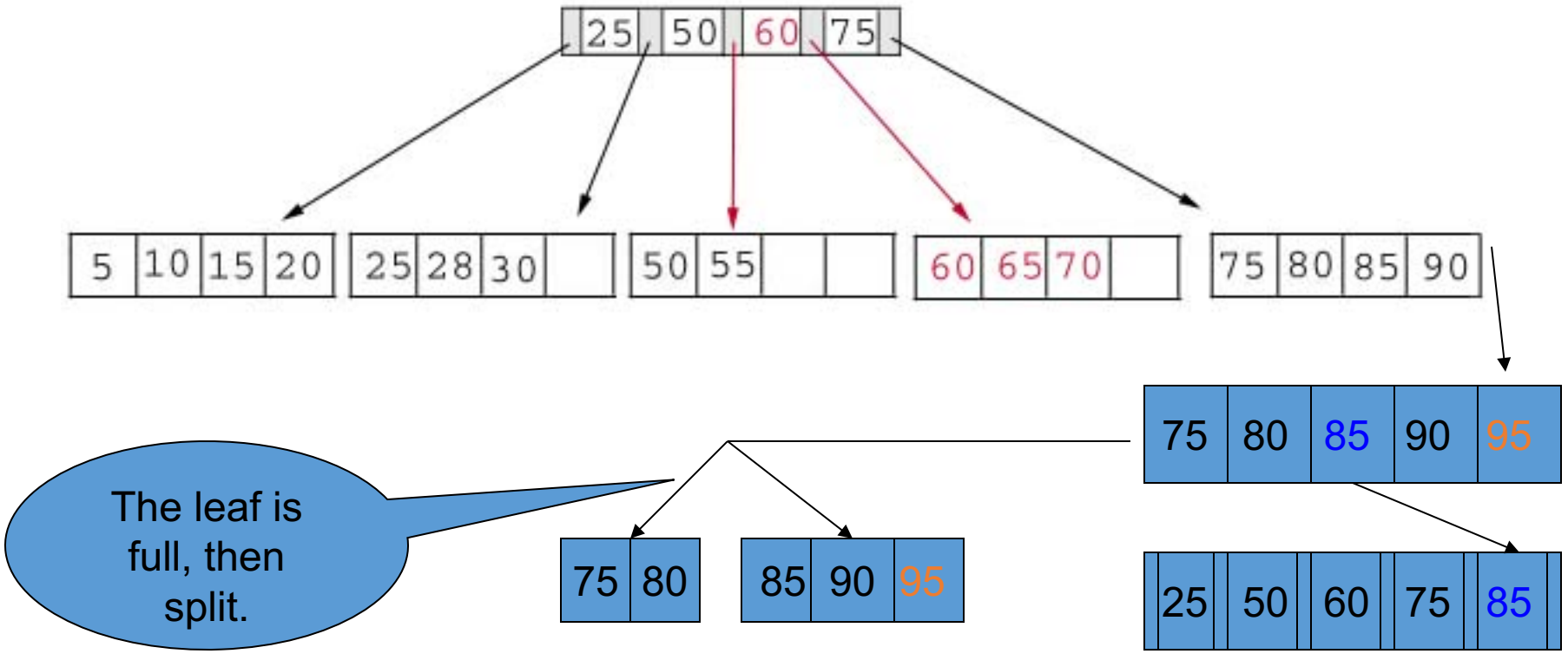
Insertion

The insert algorithm for B+ Tree

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

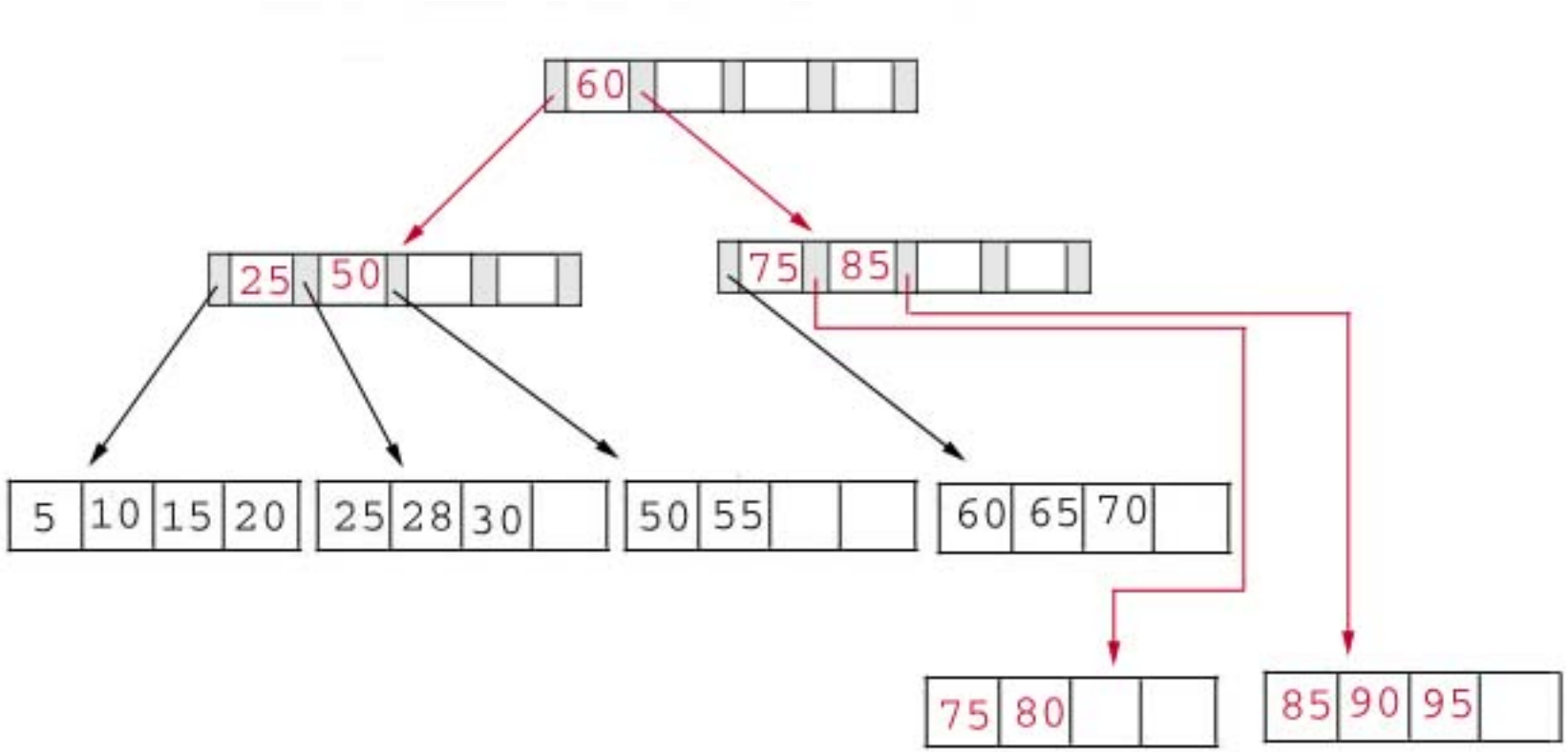
Insertion

- 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.

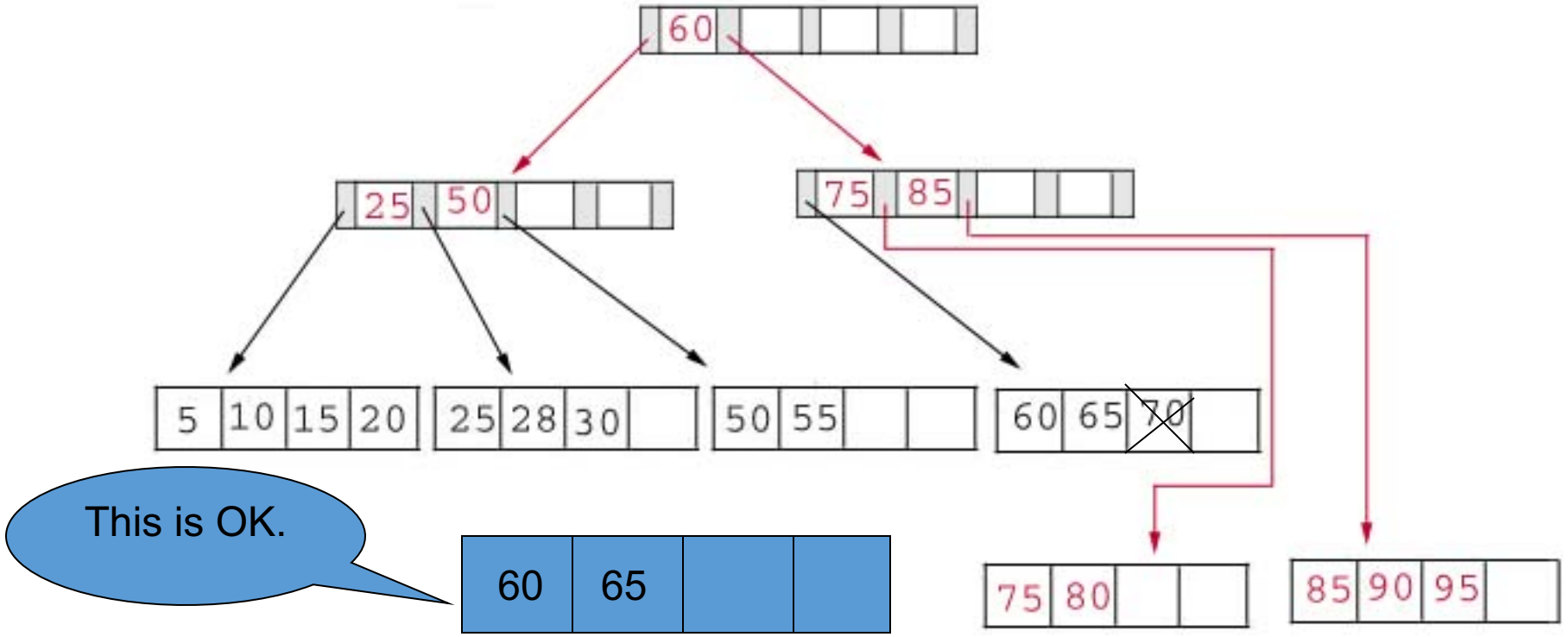


Deletion

- We go down to the leaf node where the entry belongs.
- Delete the entry. If it has too few entries in the leaf node, either redistribute the entries to the sibling nodes or merge the node with a sibling node.
- If entries are redistributed, their parent node must be updated to reflect this; the key value in the index entry pointing to the second node must be changed to the lowest search key in the second node.
- If two nodes are merged, their parent must be updated to reflect this by deleting the index entry for the second node.

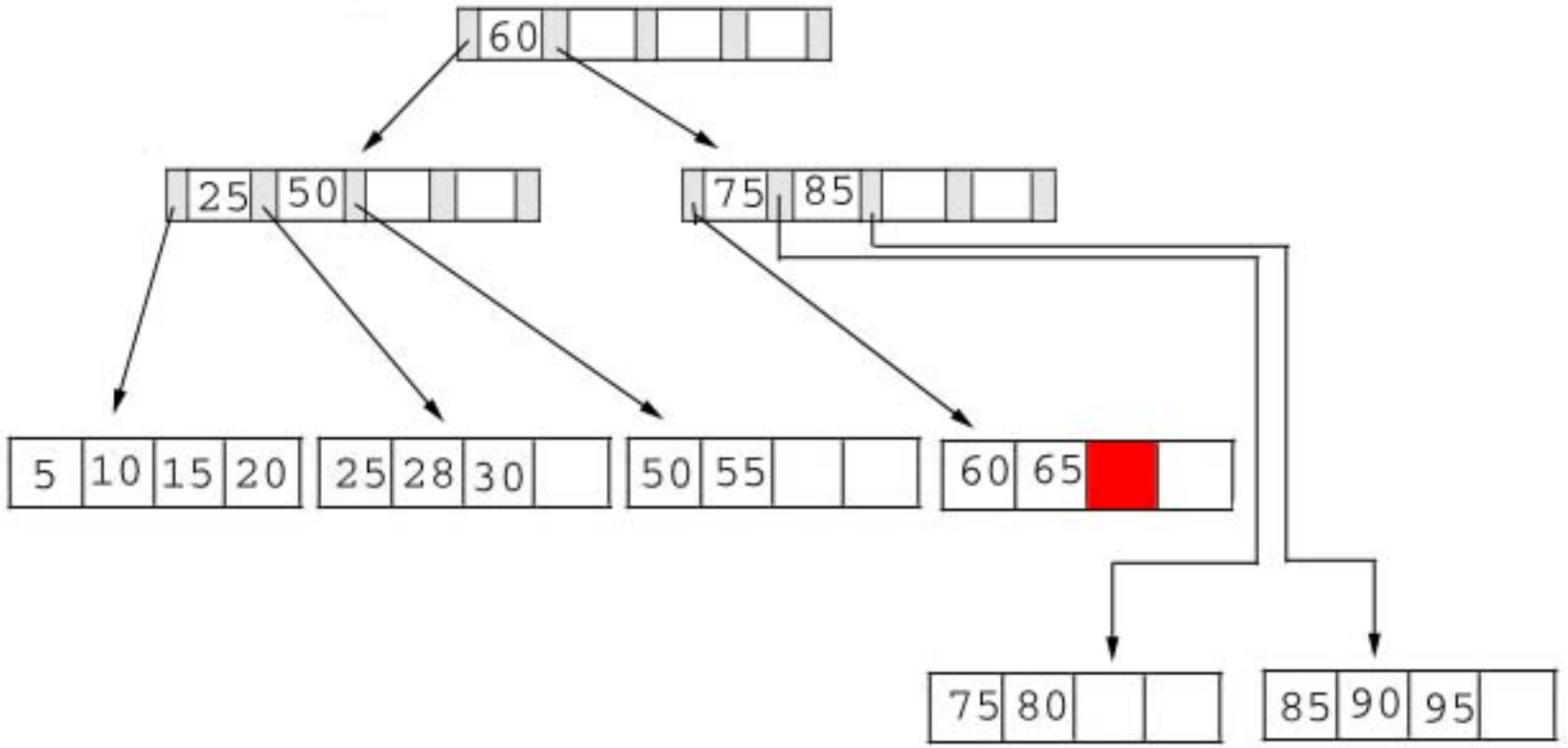
Deletion

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



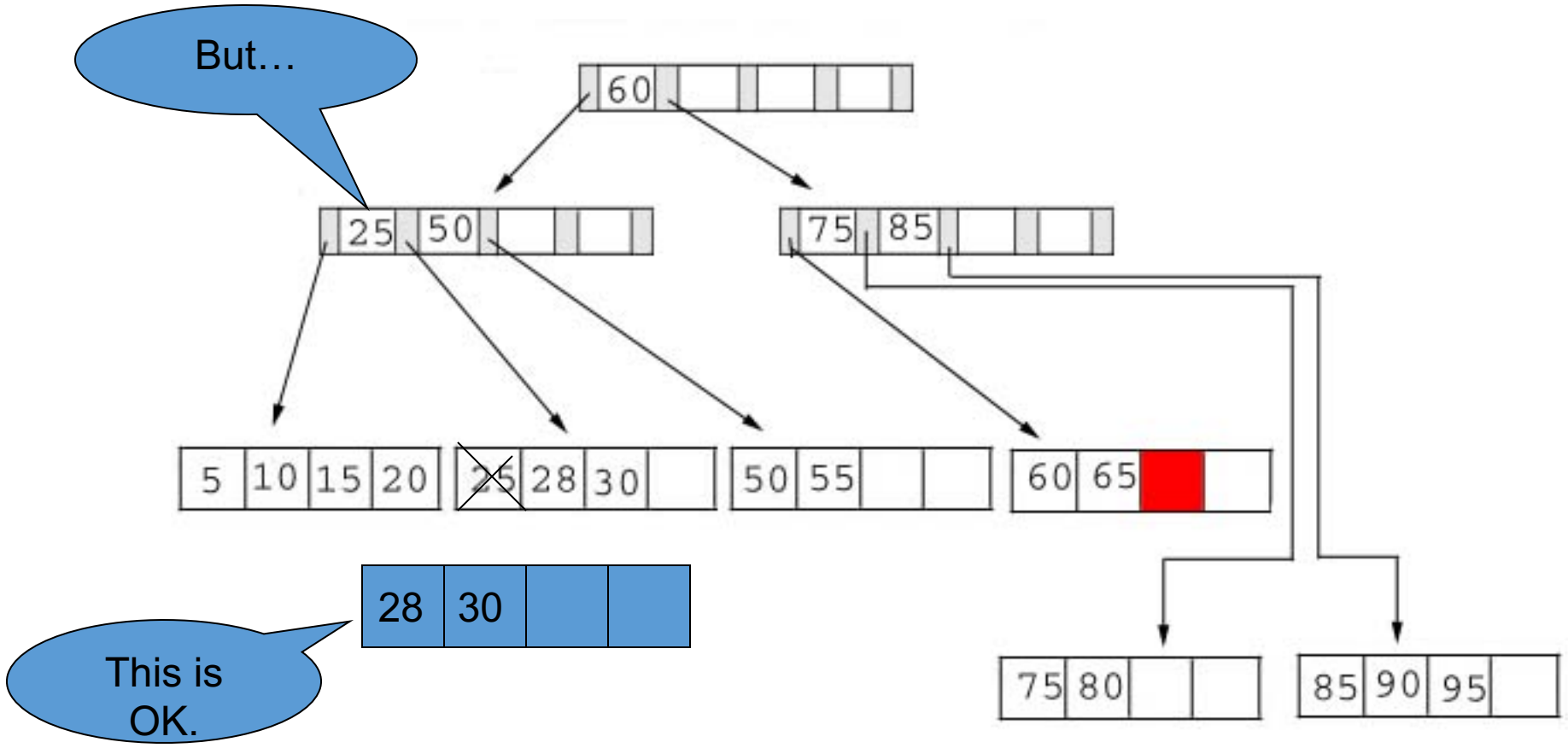
Deletion

- Result:



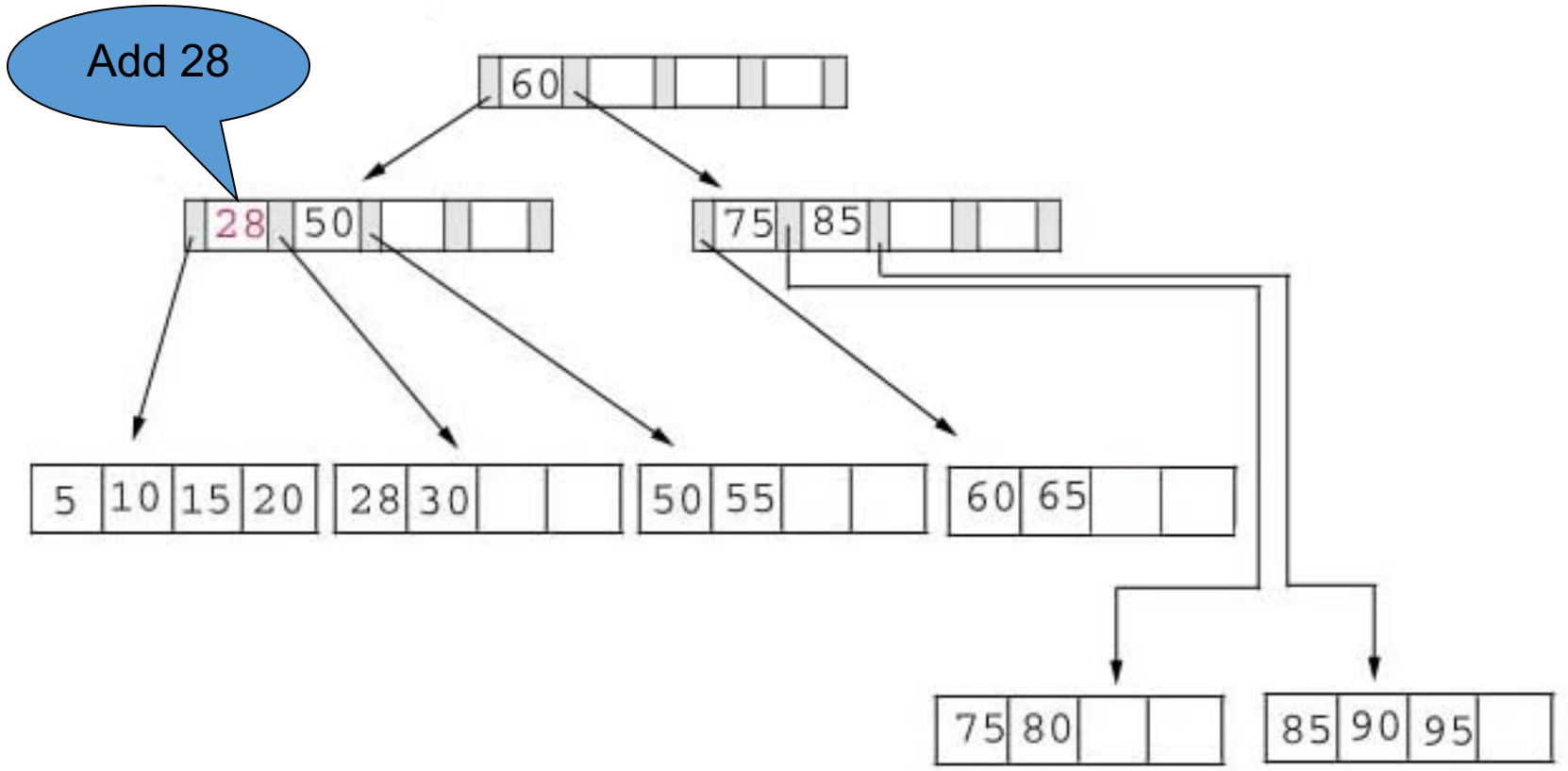
Deletion

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



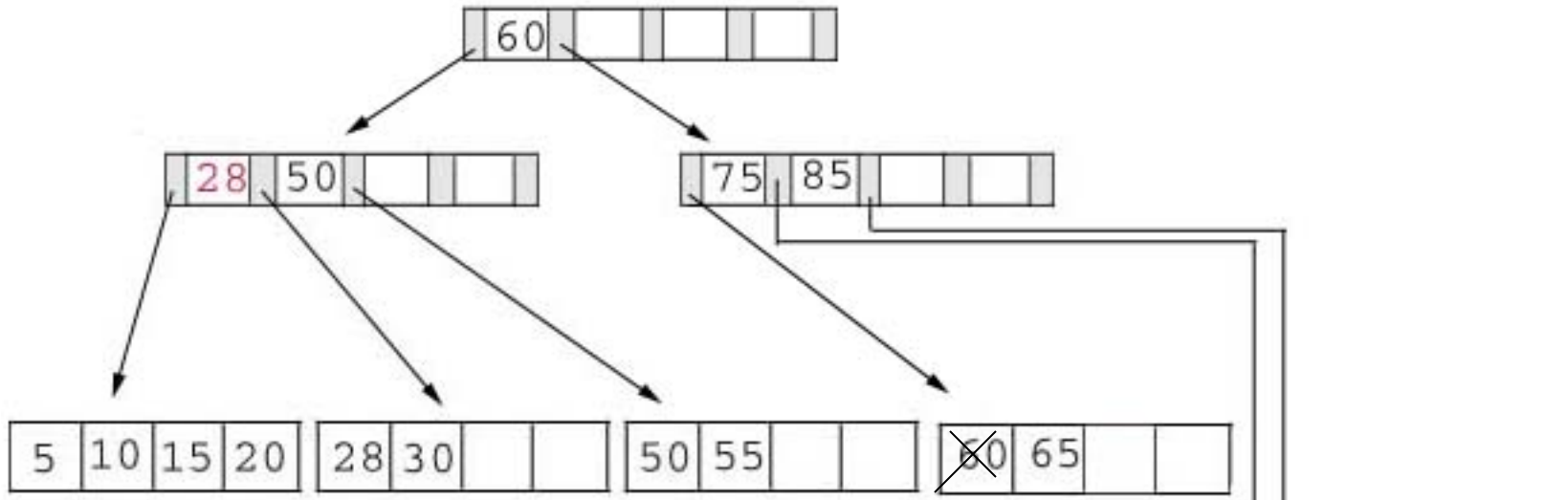
Deletion

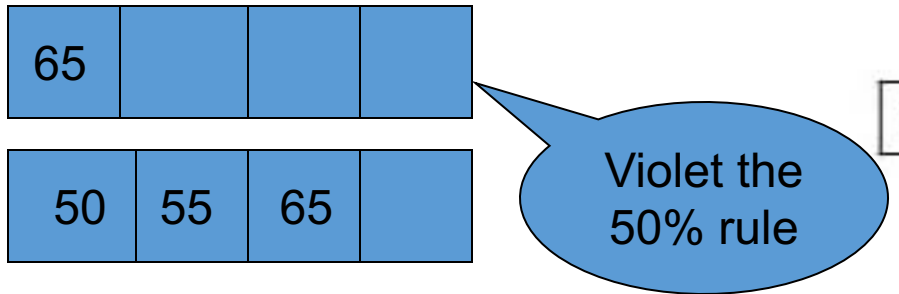
- Result: replace 28 in the index page.



Deletion

- Example #3: delete 60 from the below tree





Deletion

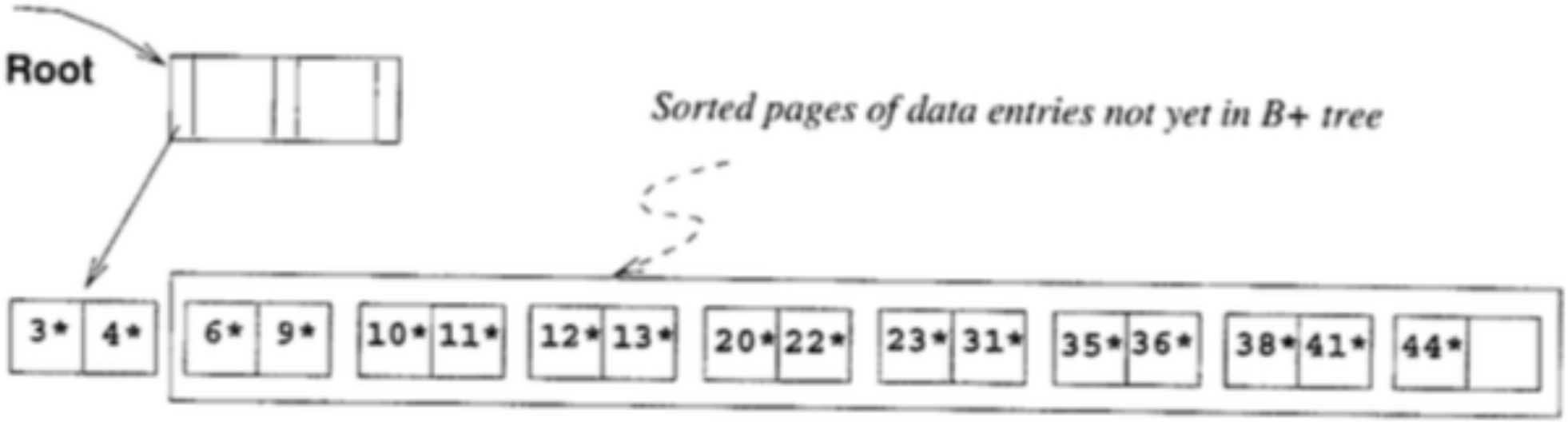
- Delete algorithm for B+ trees

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.

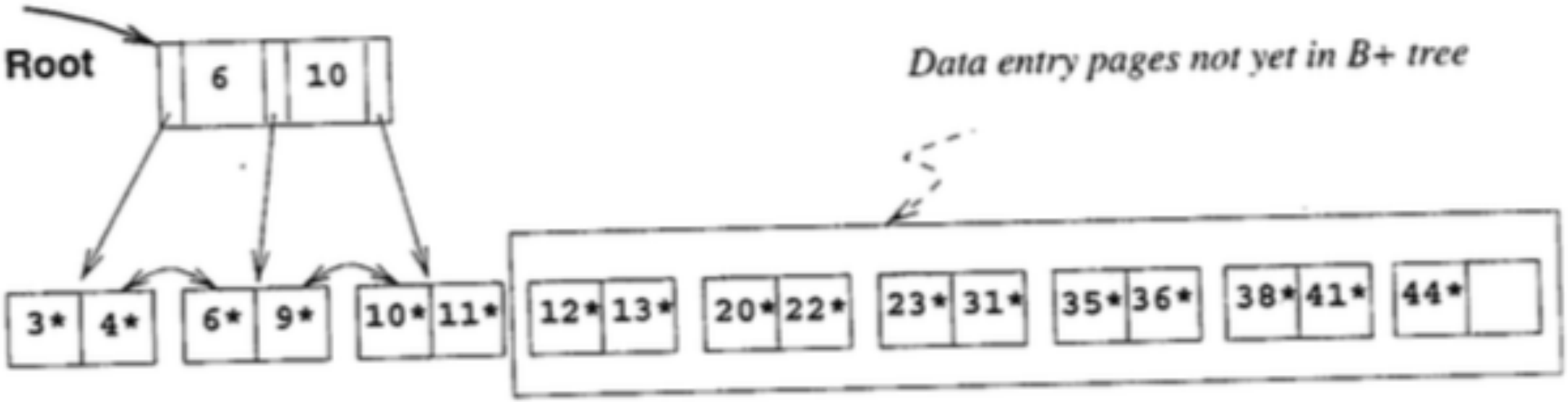
Bulk Loading in B+ Tree

- Inserting entries one by one is expensive because it requires searching from the root till the leaf node at each insertion.
- Instead, all the data to be inserted is sorted and inserted as a collection of data entries.

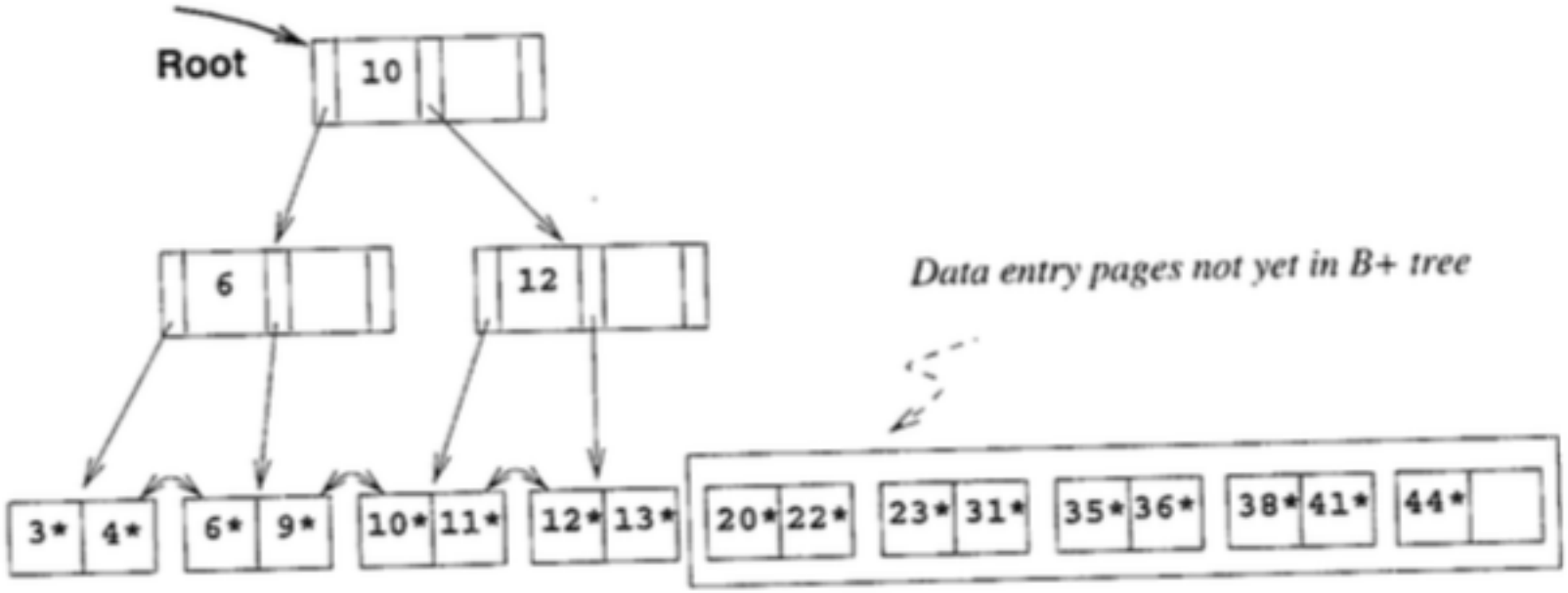
Bulk Loading in B+ Tree



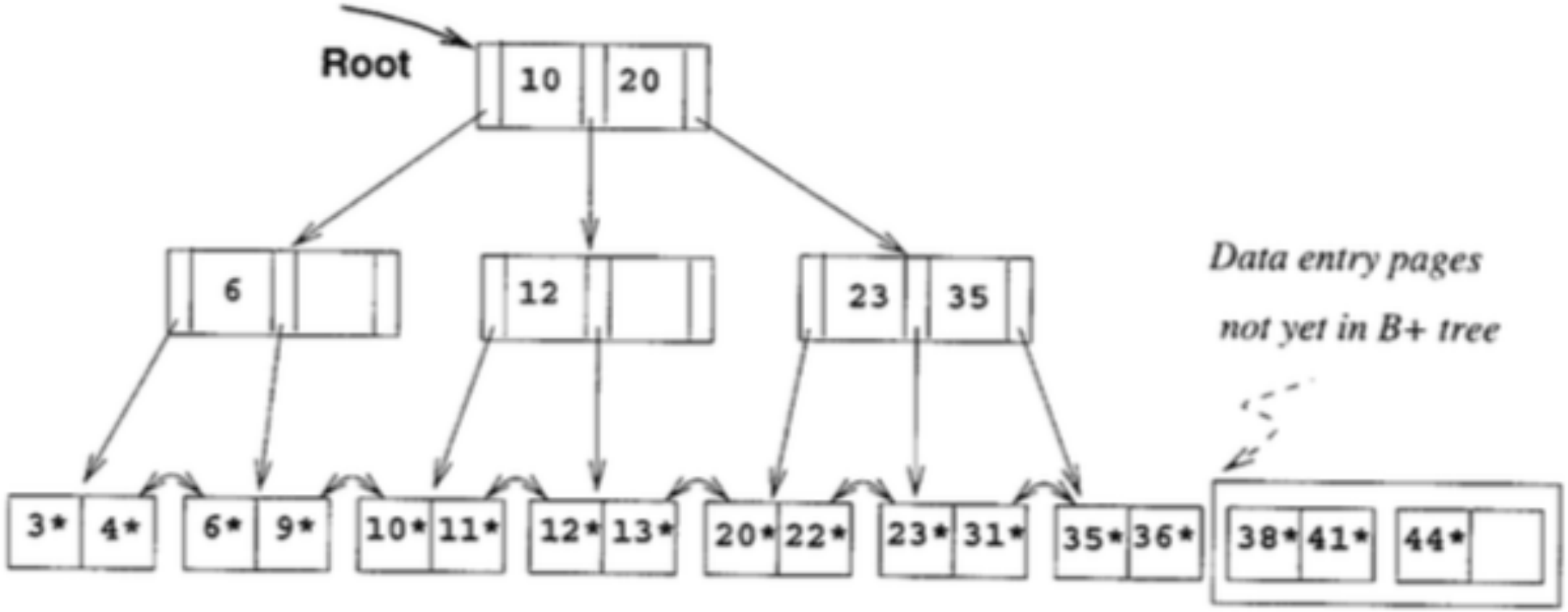
Bulk Loading in B+ Tree



Bulk Loading in B+ Tree



Bulk Loading in B+ Tree



Bulk Loading in B+ Tree