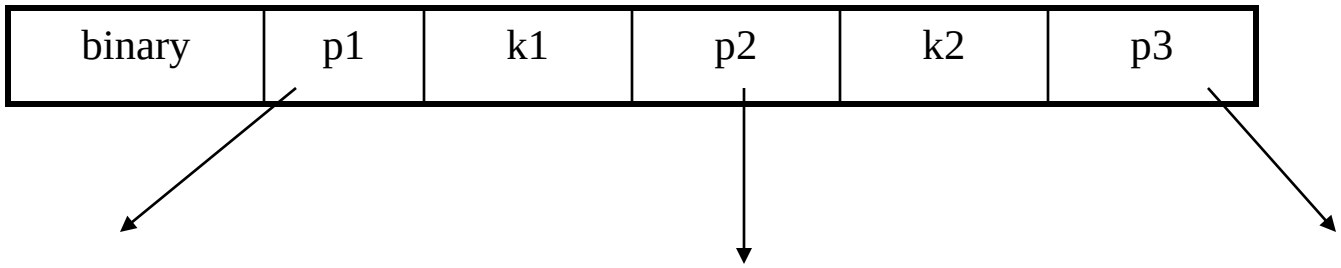
2-3 TREE

2-3 Tree Definition

- All leaf nodes are at the same level.
- All non leaf nodes are either binary or ternary.
- Binary nodes are like BST nodes with single key k.
- Ternary nodes have two keys k1 and k2 and three children pointers p1, p2, p3.
- All keys < k1 reside in sub-tree pointed to by p1. All keys > k2 reside in sub-tree pointed to by p3. Others reside in the sub-tree pointed to by p2. All keys are distinct .

2-3 Tree node

```
typedef struct nt {
        T       k1, k2;
        int     binary;
struct nt       * p1, * p2, * p3;
        }       23treenode;
```



Operations

- If h is the height of a 2-3 tree having n nodes,

then $2^h-1 <= n <= 3^h-1$

Thus, $\log_3(n+1) <= h <= \log_2(n+1)$

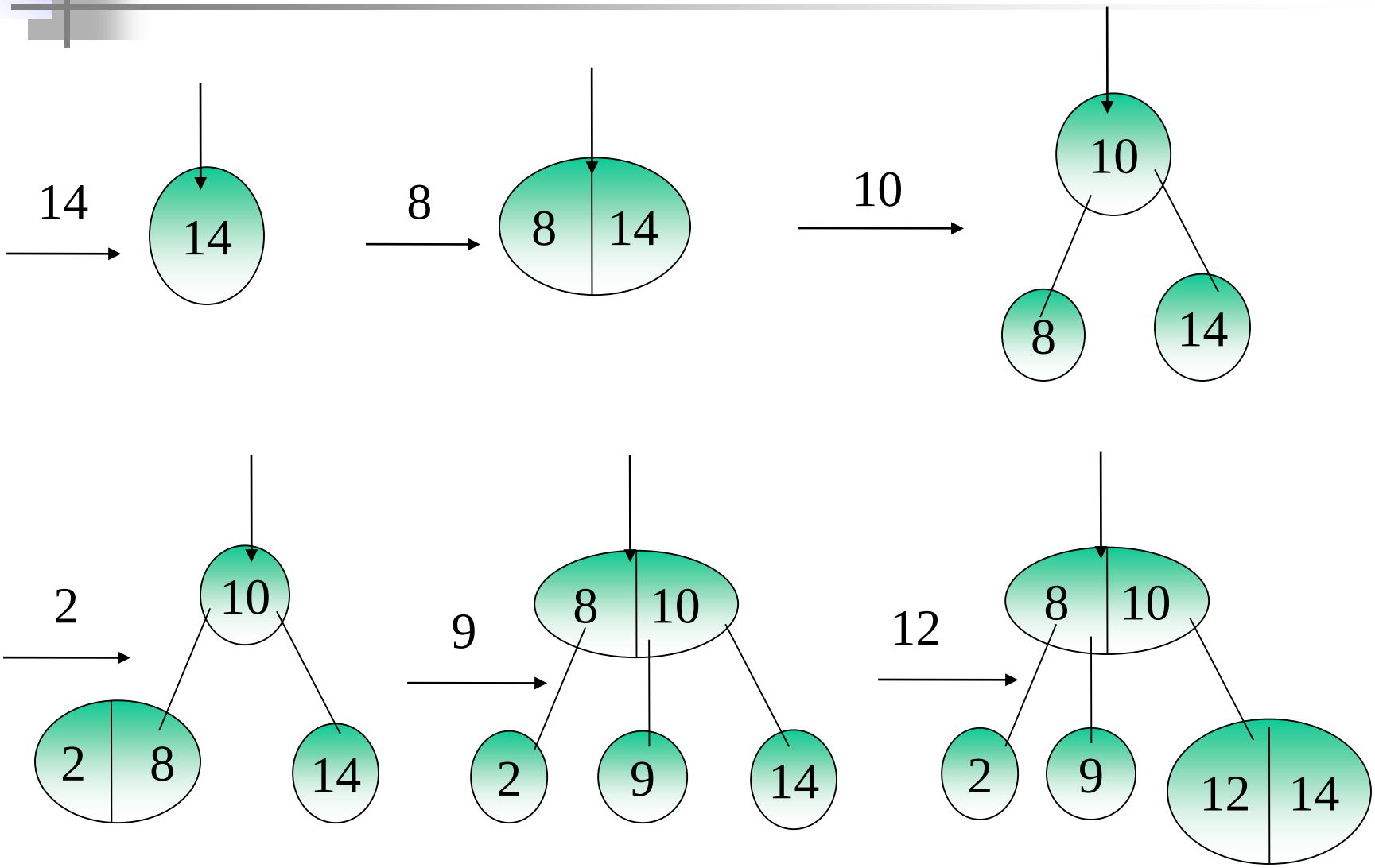
Operations:

Insert node

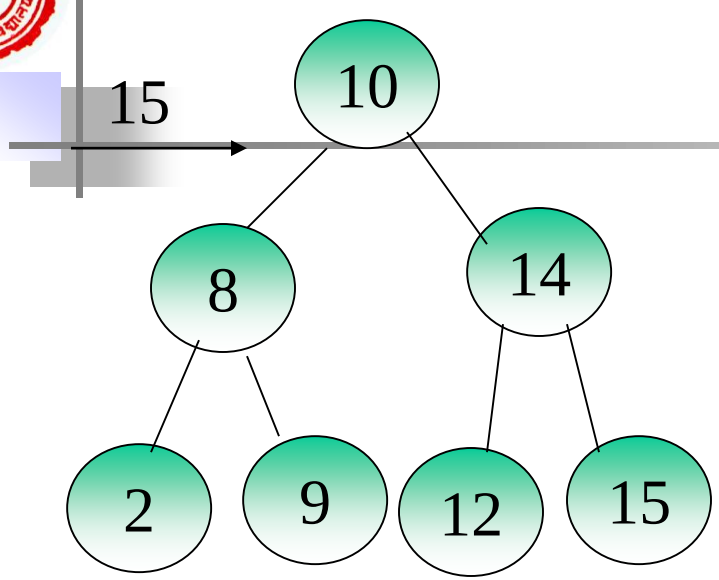
Delete node

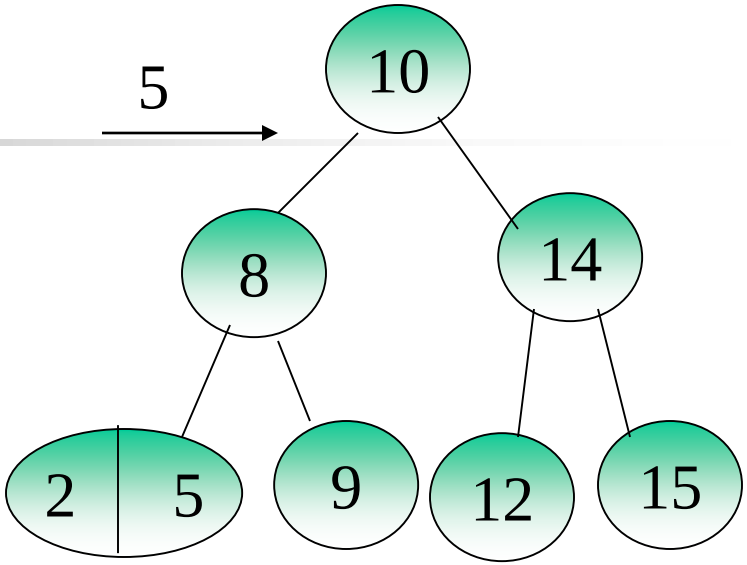
Search node

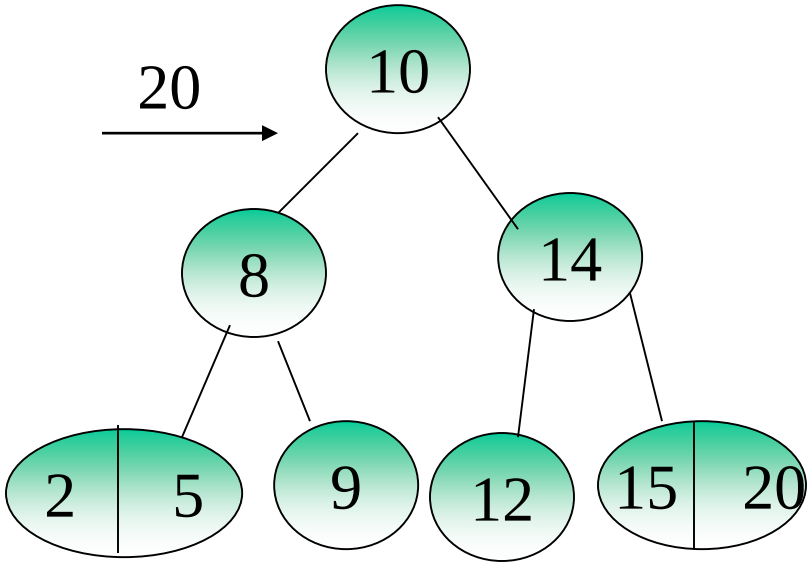
INSERTION EXAMPLES

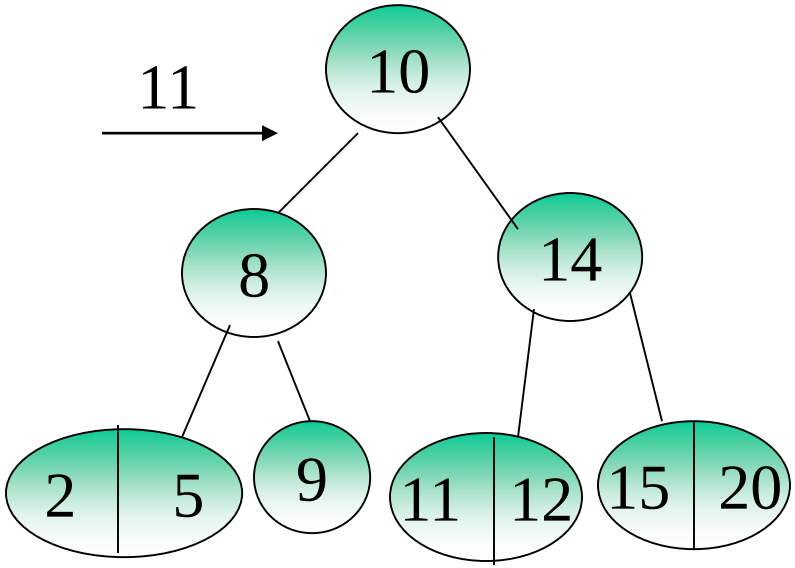


INSERTION EXAMPLES …

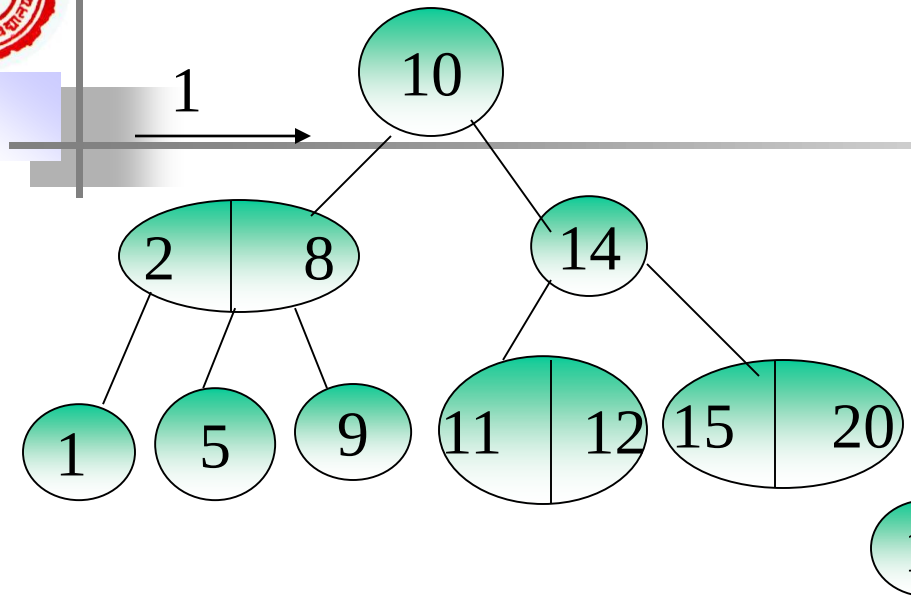


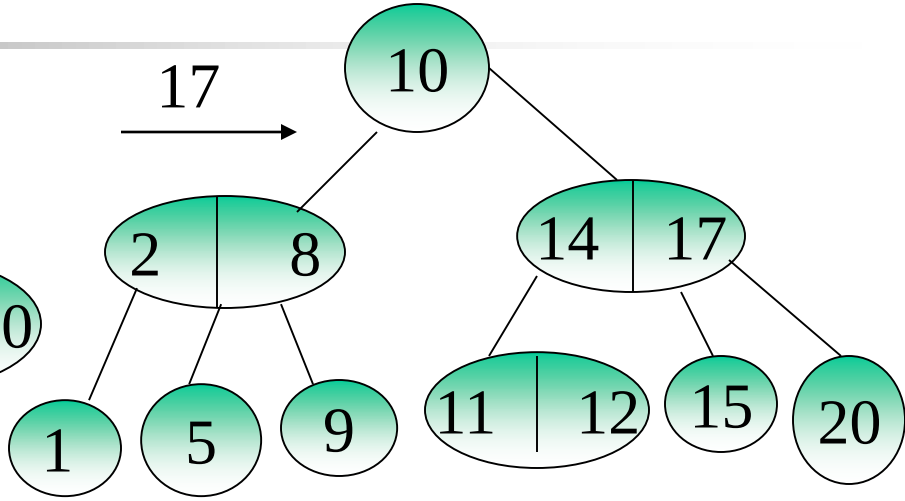


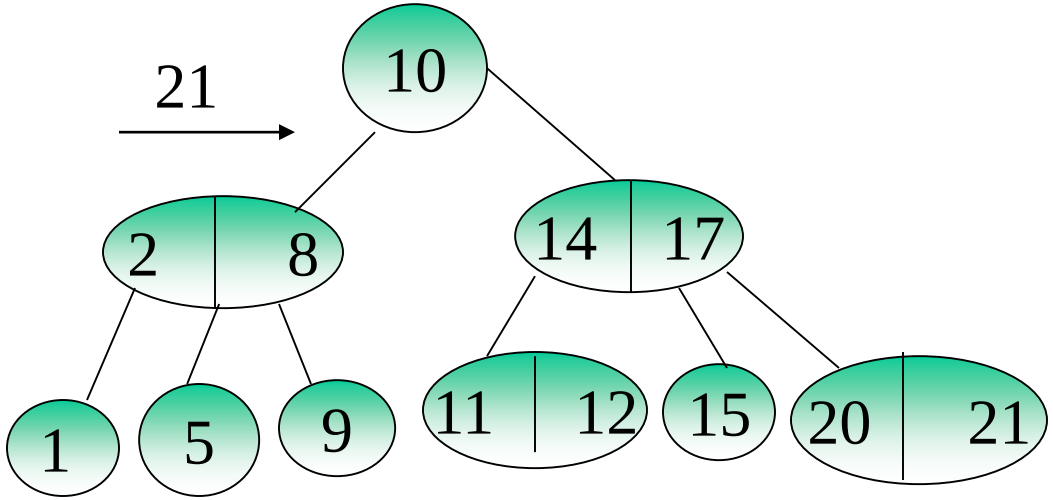




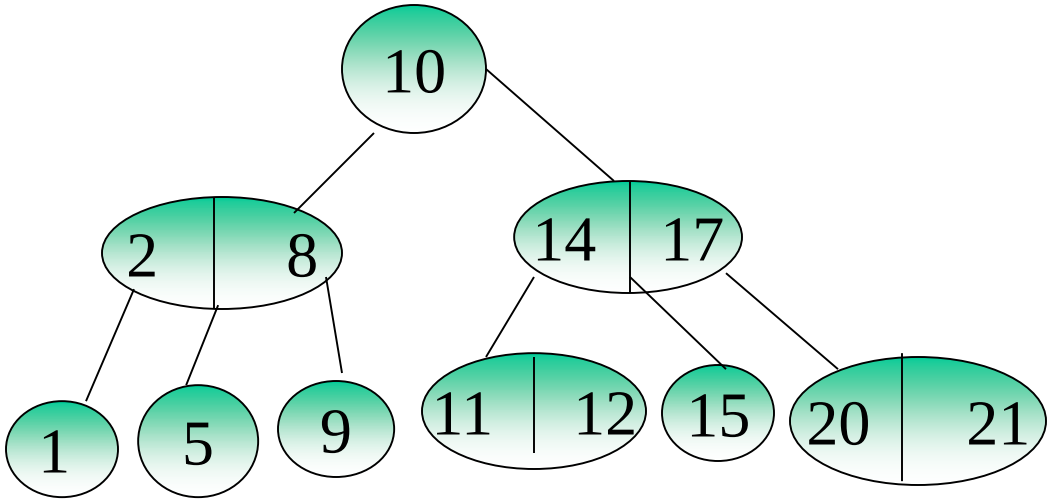
INSERTION EXAMPLES …

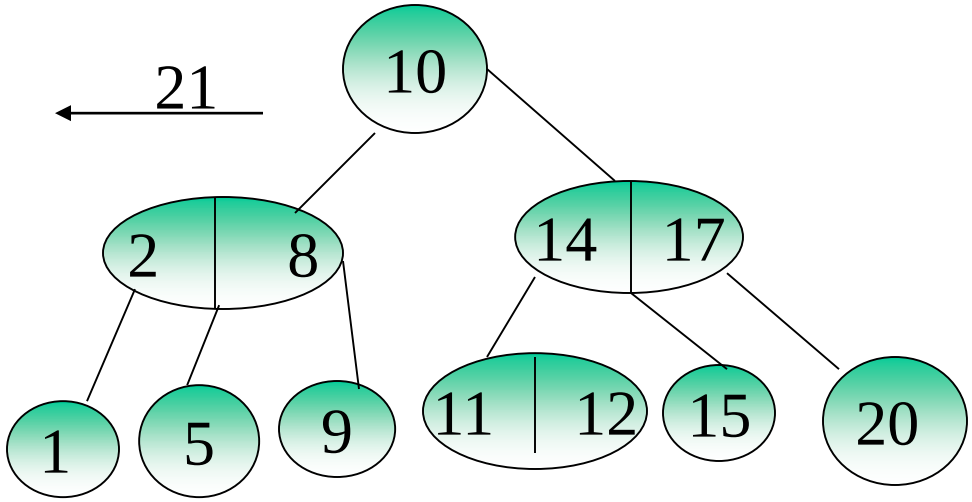




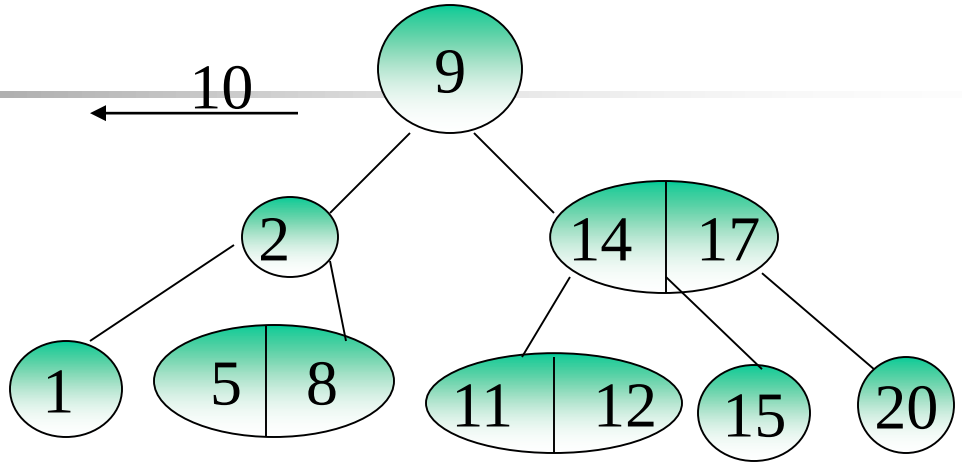


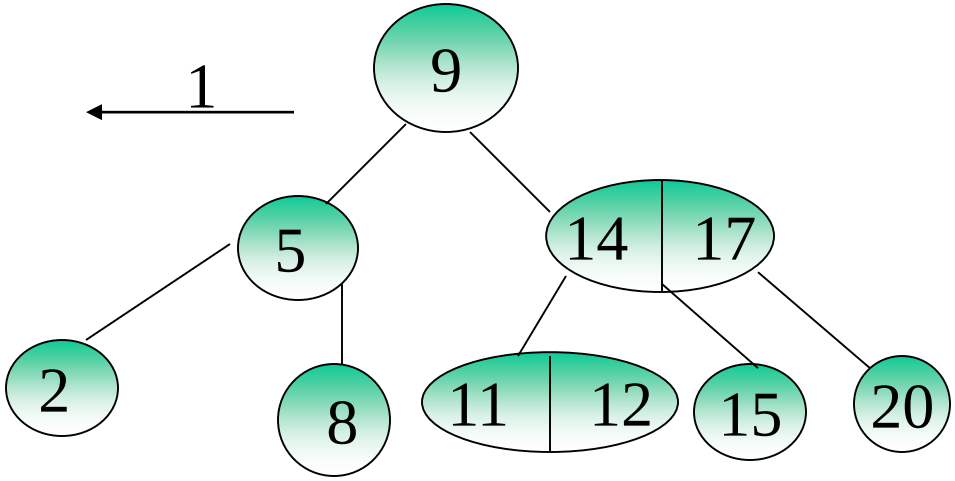
DELETION EXAMPLES



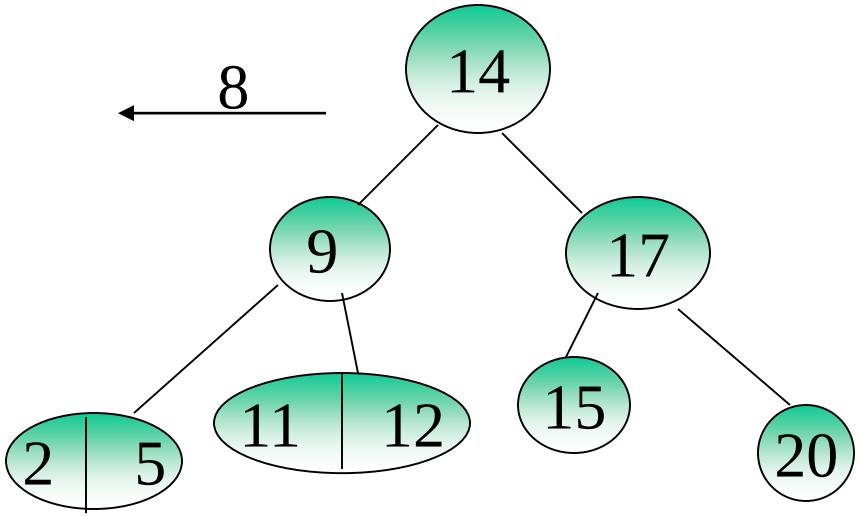


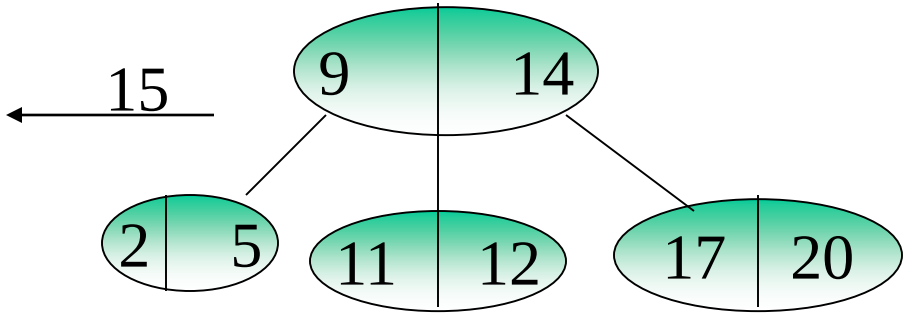
DELETION EXAMPLES ...





DELETION EXAMPLES ...





B TREE

B-Tree Definition

A B-Tree of order d is a tree with following properties:

- Each node (except possibly the root node) contains at most d records and at least $\lfloor d/2 \rfloor$ records.
- The root node can have at most d records and as few as one record.
- An internal node containing m records ($1 <= m <= d$) with key values k_1 to k_m, have pointers to m+1 subintervals of keys stored in sub-trees.
- A leaf node has empty sub-tree below it. All leaf nodes are at the same level.
- All keys are distinct

B-TREE Node

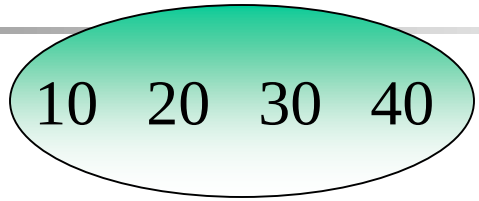
```
typedef struct btnode {
                        int       k;
                        T         datalist[maxkeyno];
    struct btnode       * ptr[maxkeyno + 1];
                        }
```

Height of a B-Tree of order d containing n nodes is given by $\lfloor \log_{\lfloor d/2 \rfloor + 1}(n+1) \rfloor >= h >= \lceil \log_{d+1}(n+1) \rceil$

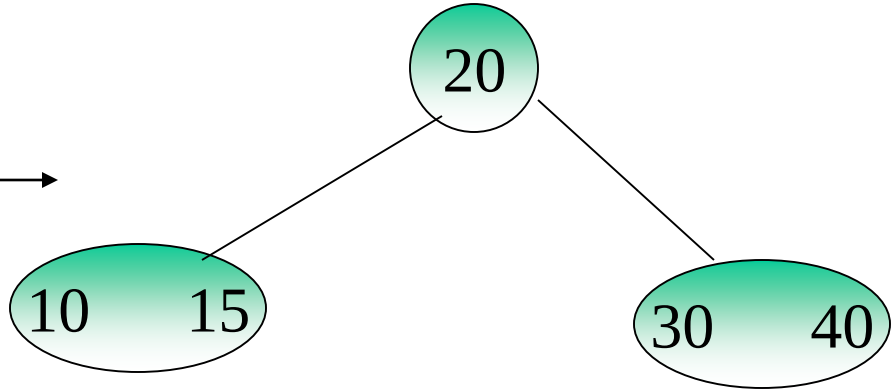
2-3 tree is a B-Tree of order 2

Insertion Examples on a B-Tree of order 4

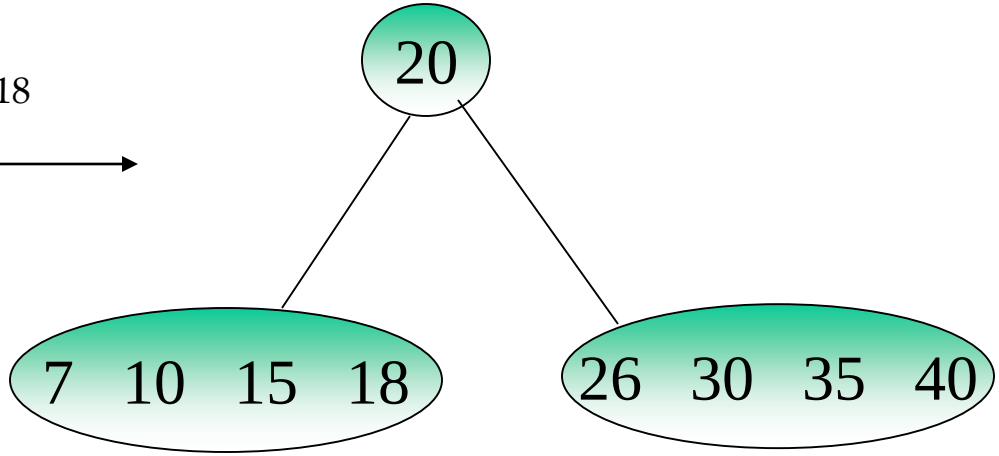
20, 40, 10, 30



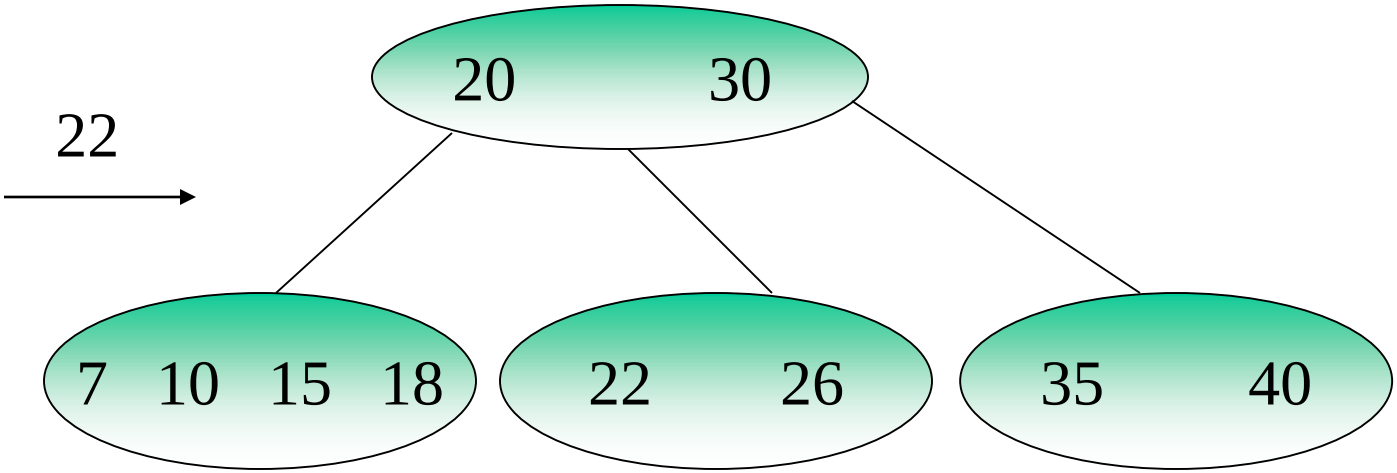
15

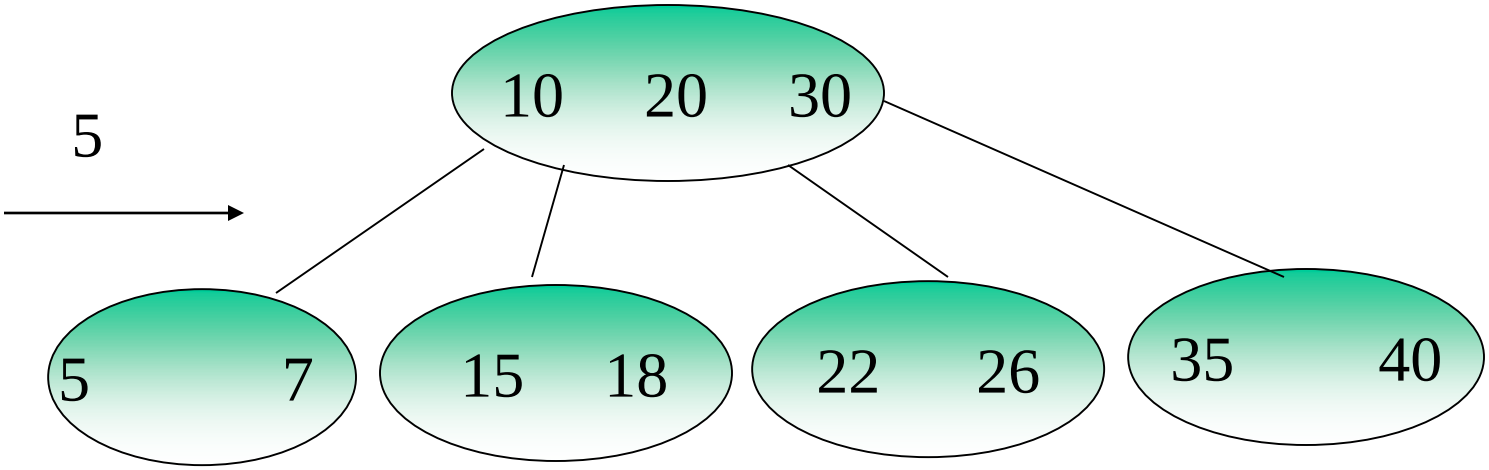


35, 7, 26,18

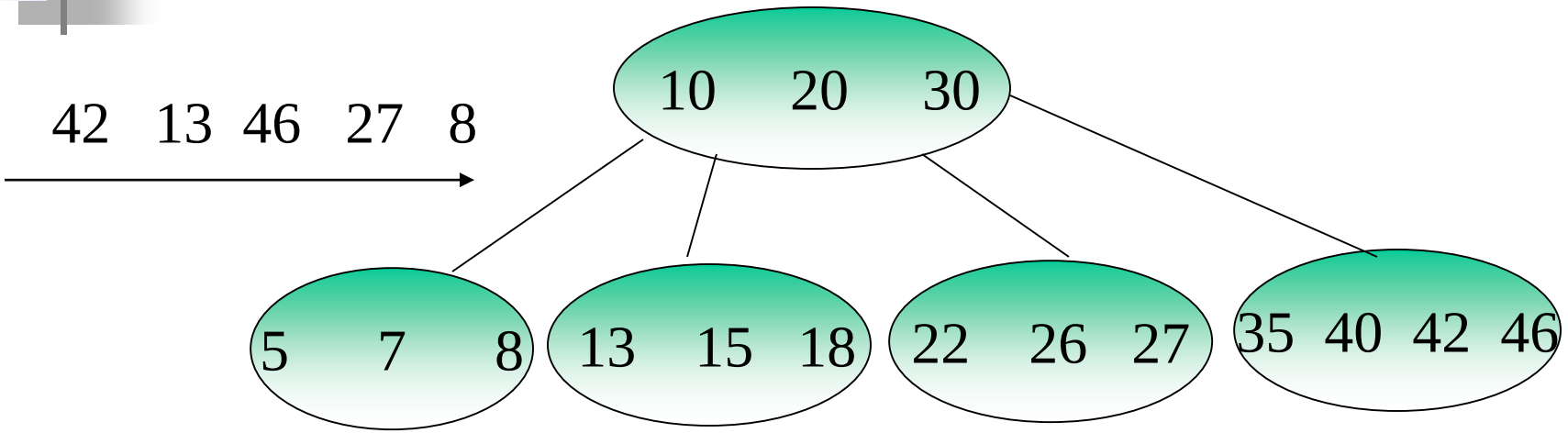


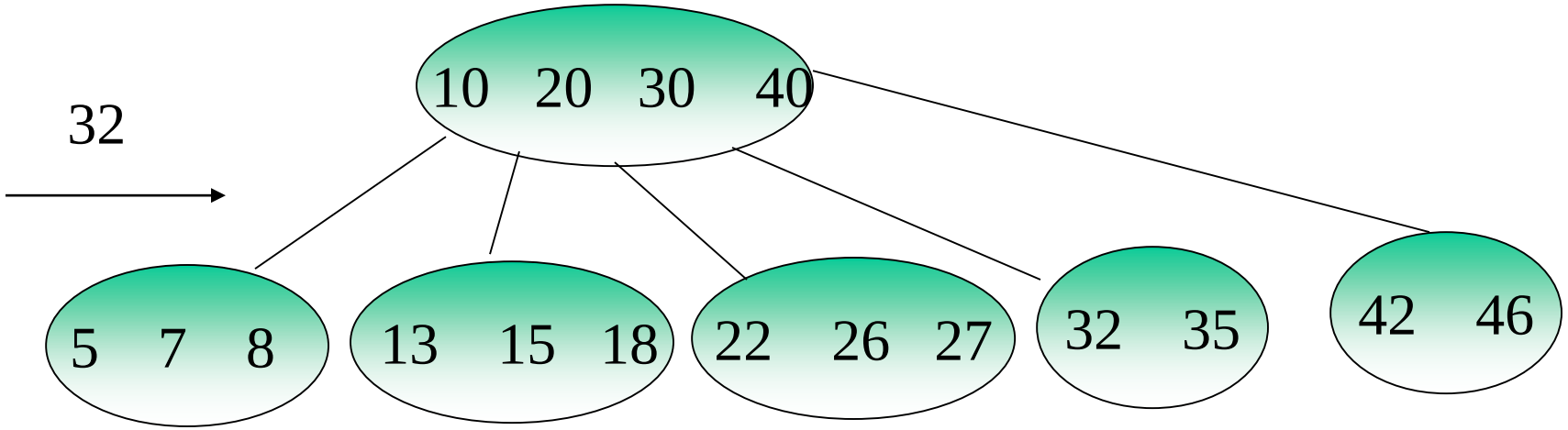
Insertion Examples …



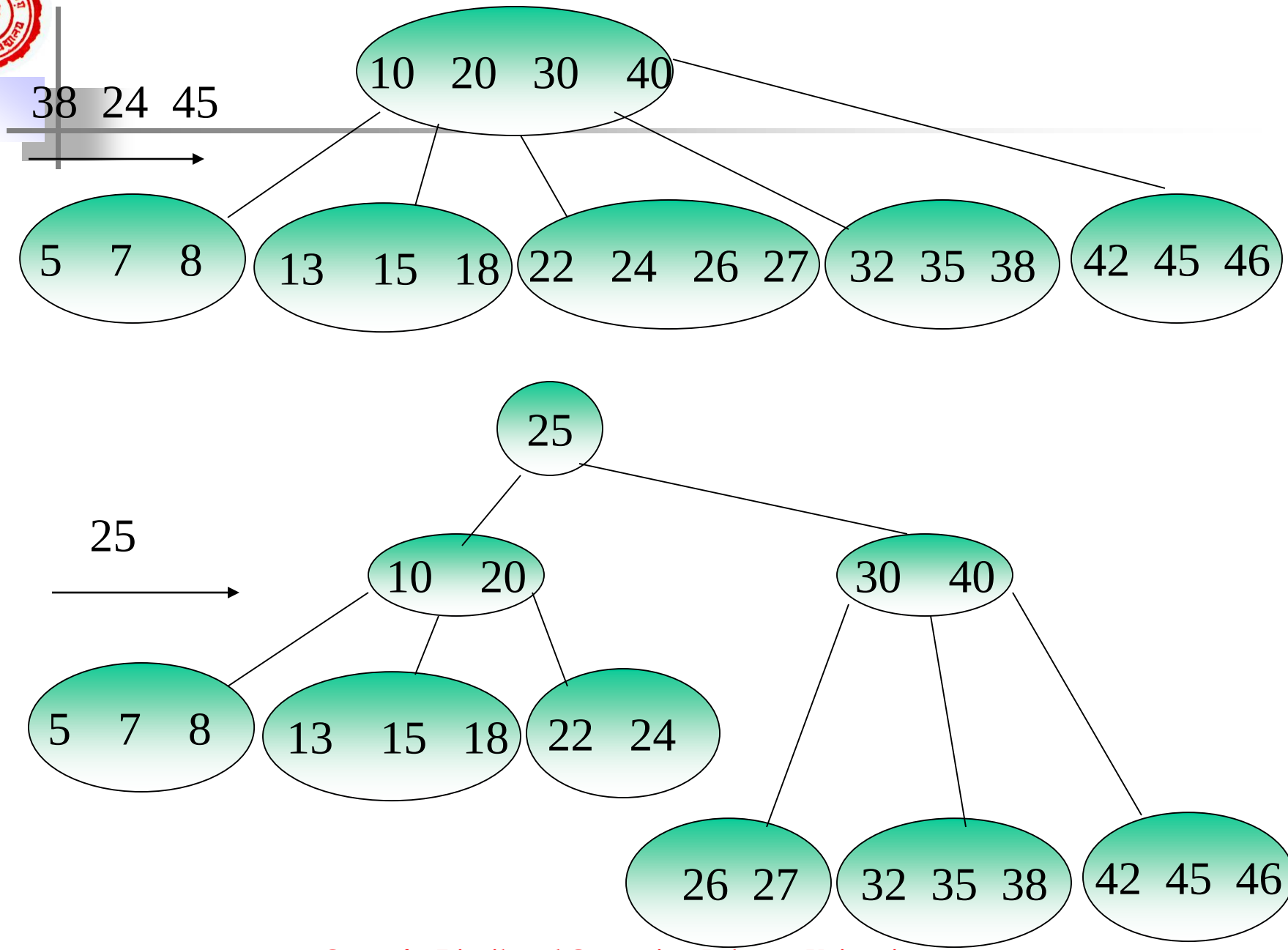


Insertion Examples …

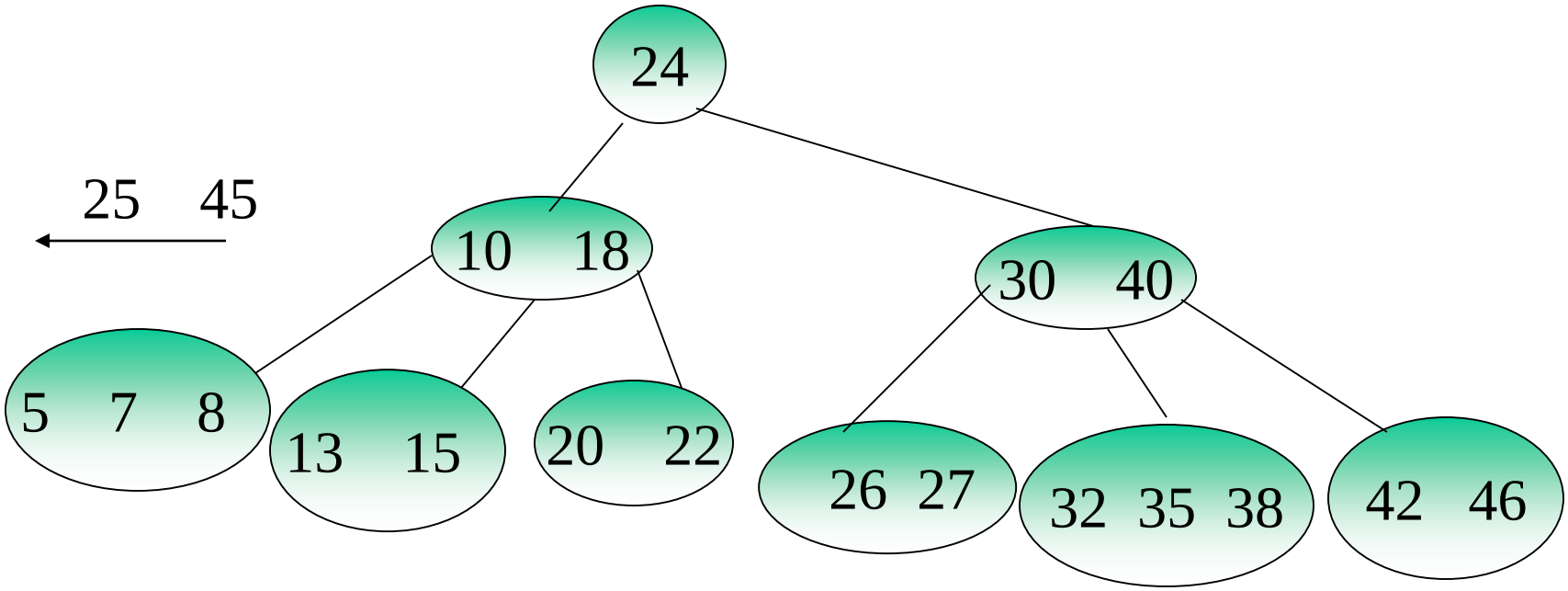




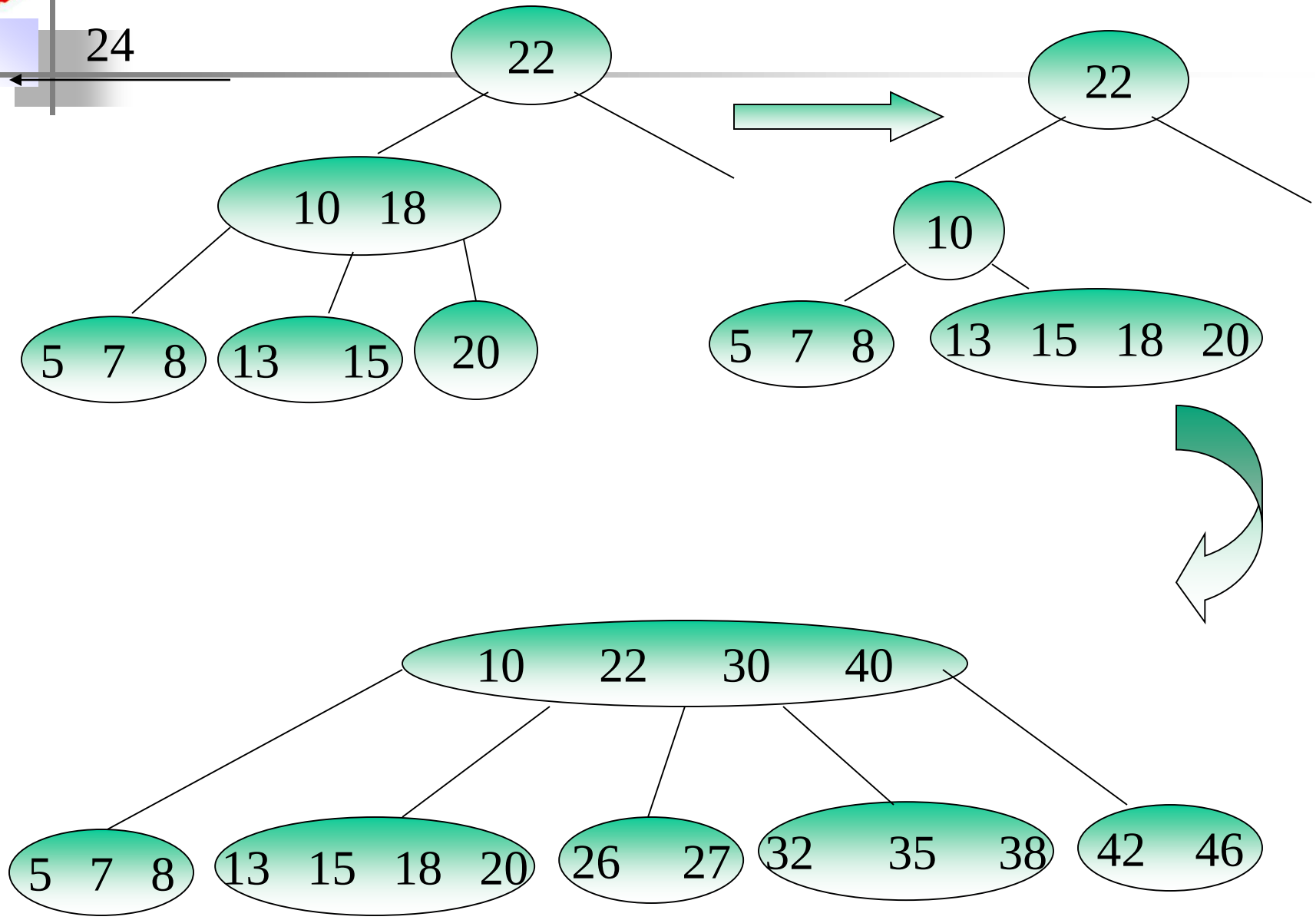
Insertion Examples …



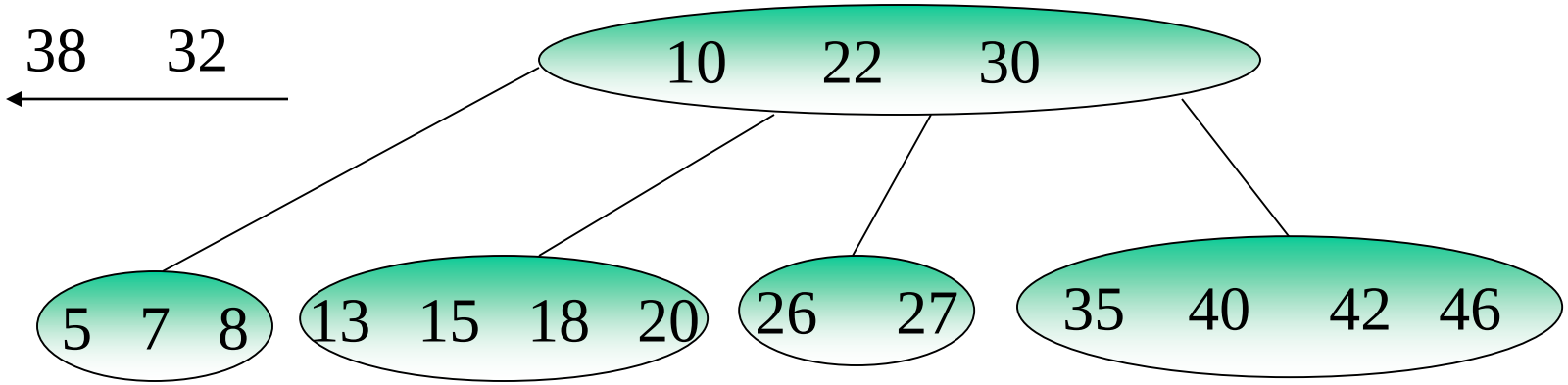
Deletion examples

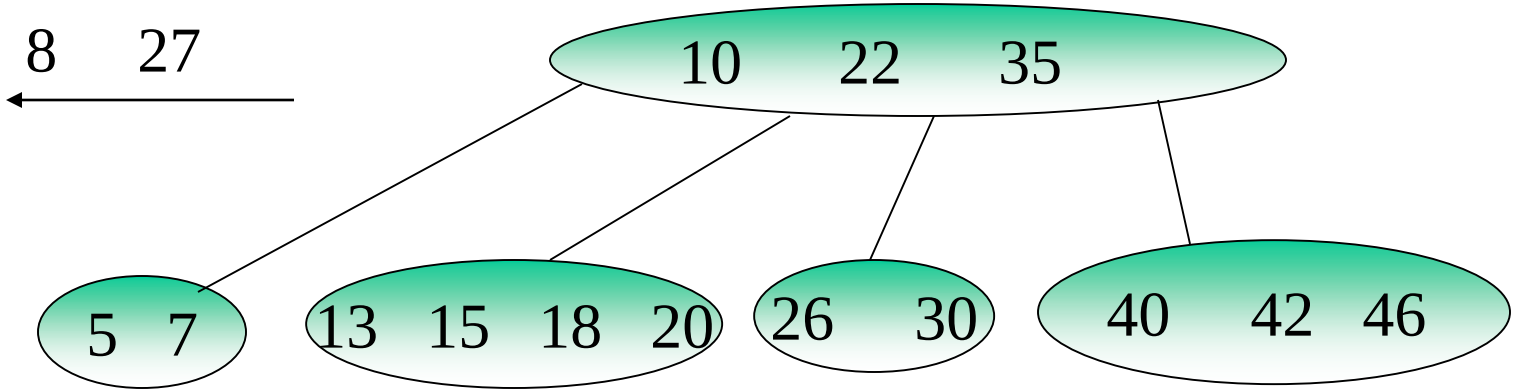


Deletion examples …

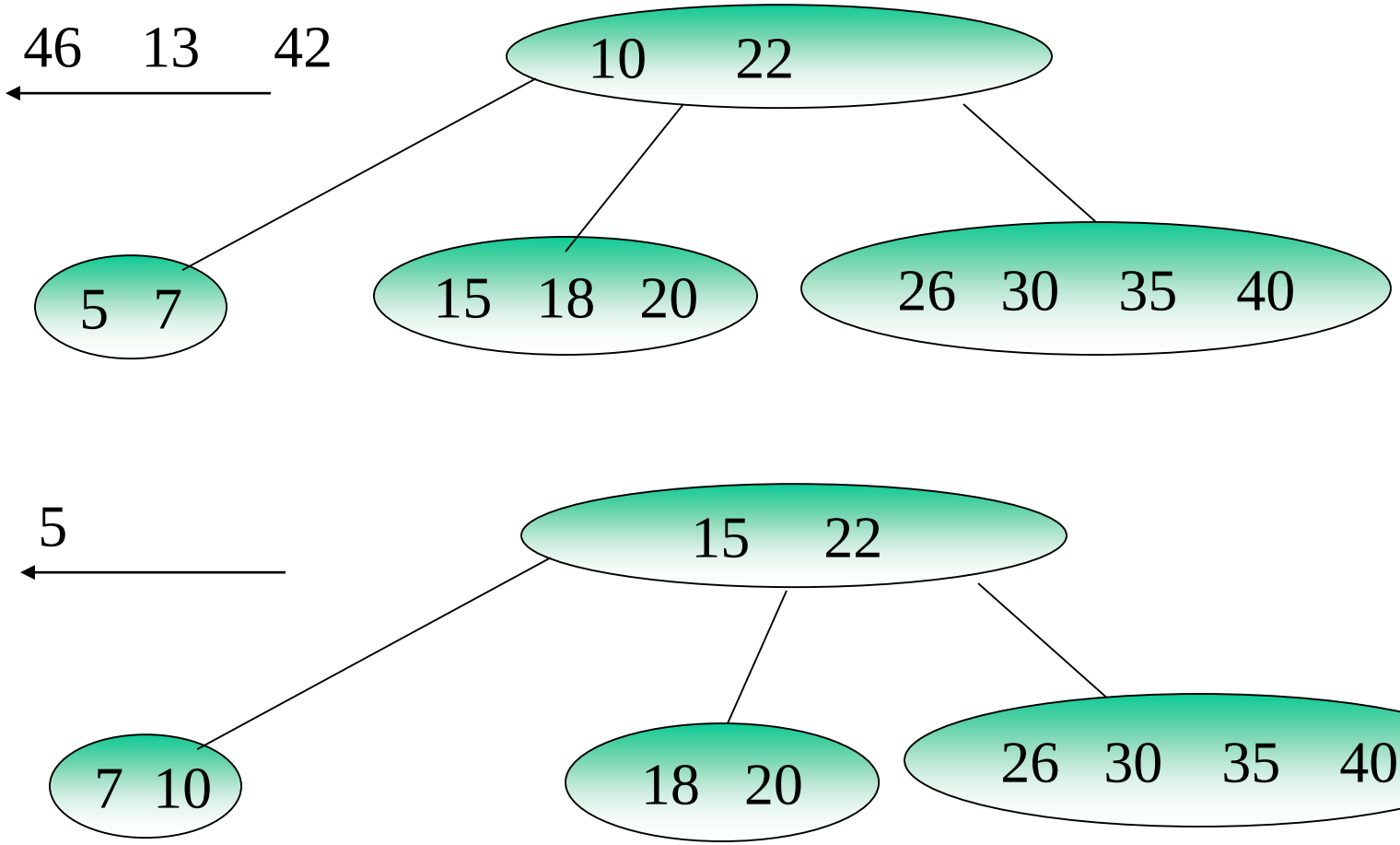


Deletion examples …

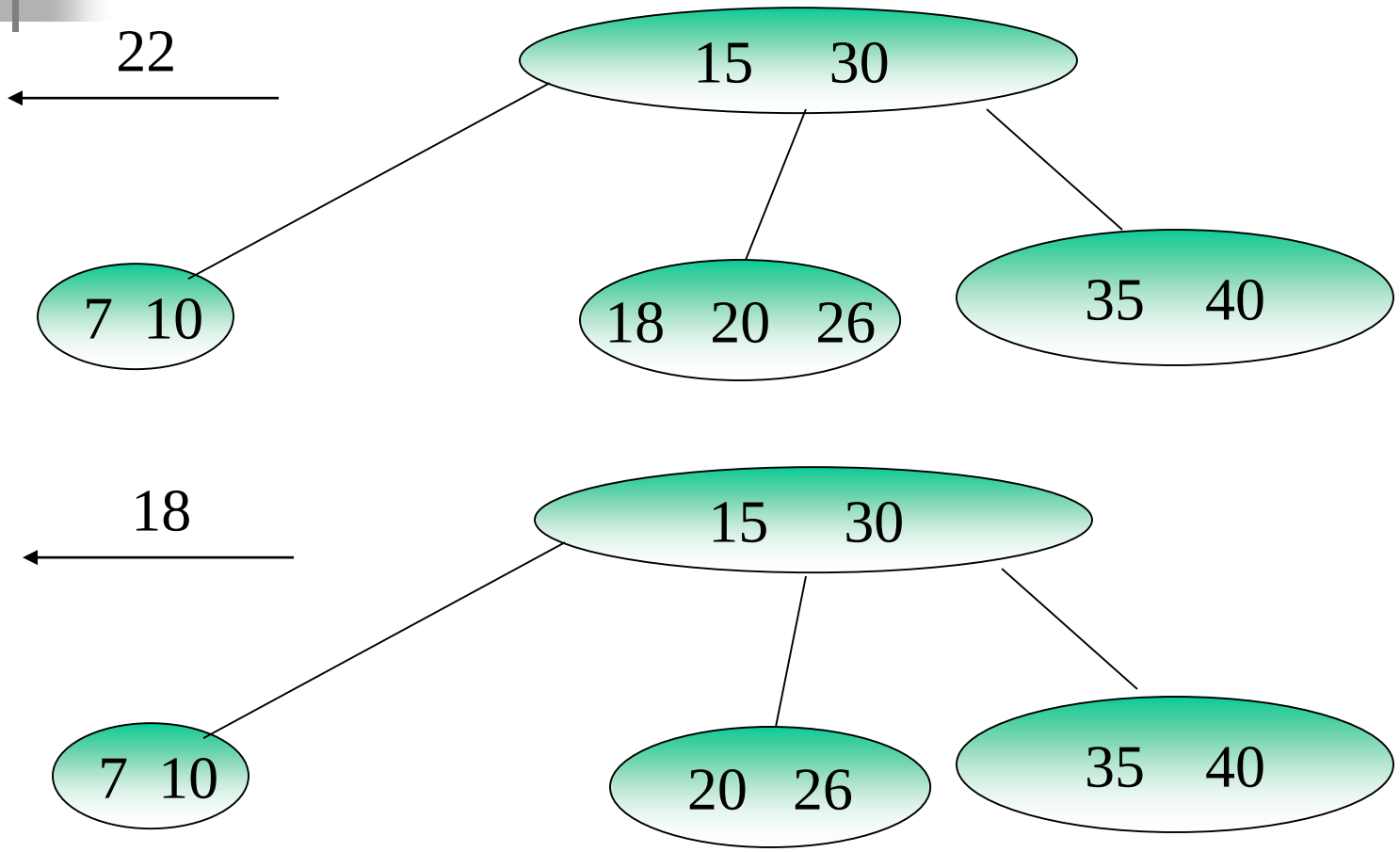




Deletion examples …



Deletion examples …



Delete 26 , 7 , 35 , 15

B+ Tree

What is a B+ Tree?

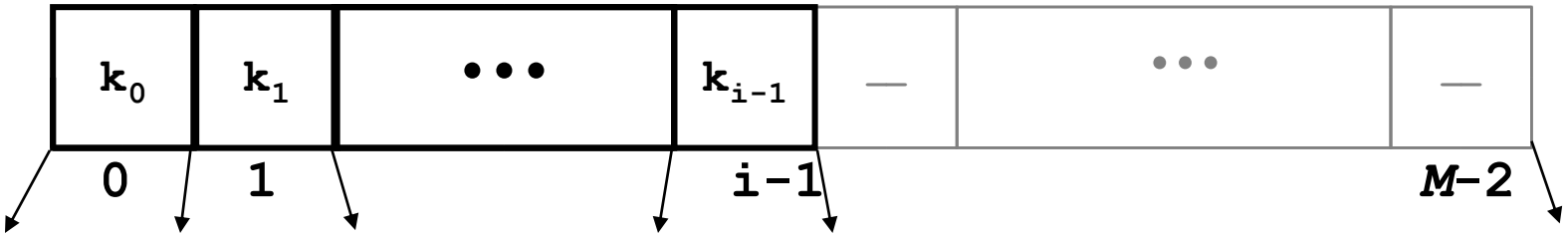
- A variation of B tree in which
  - internal nodes contain only search keys (no data)
  - Leaf nodes contain keys with pointers to data records
  - Data records are in sorted order by the search key
  - All leaves are at the same depth
  - Internal nodes and Leaf nodes may have different orders
  - Keys are distinct

Definition of a B+Tree

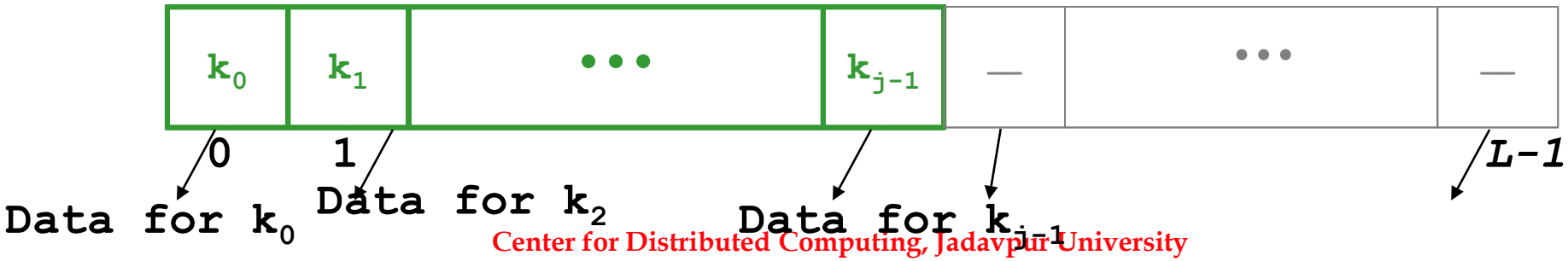
A B+ tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length, and each non-leaf node of the tree has between M/2 and M children, where M is fixed for internal nodes of a particular tree.

B+ Tree Nodes

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



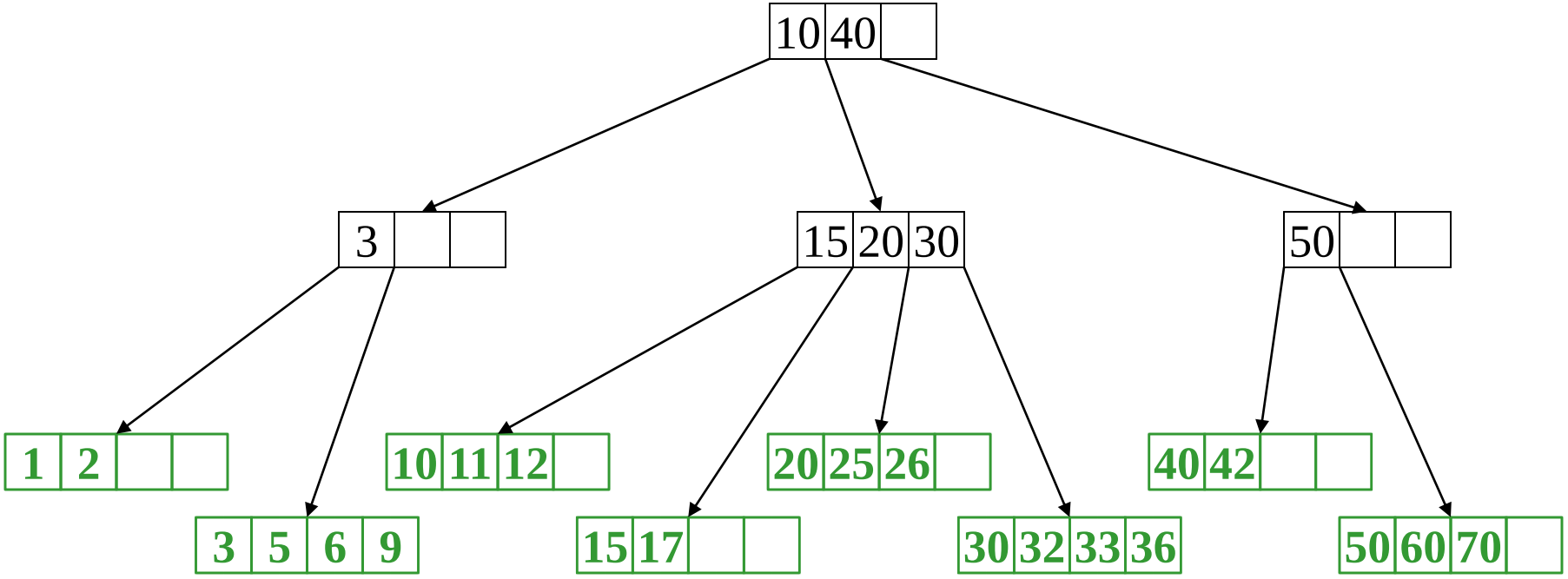
- **Leaf node**
  - (Key, DataPointer)* L in each node
  - first j Keys currently in use



Example

B+ Tree with **M = 4 (3 keys and 4 pointers)**

Leaf nodes (having L <= 4 keys)linked together



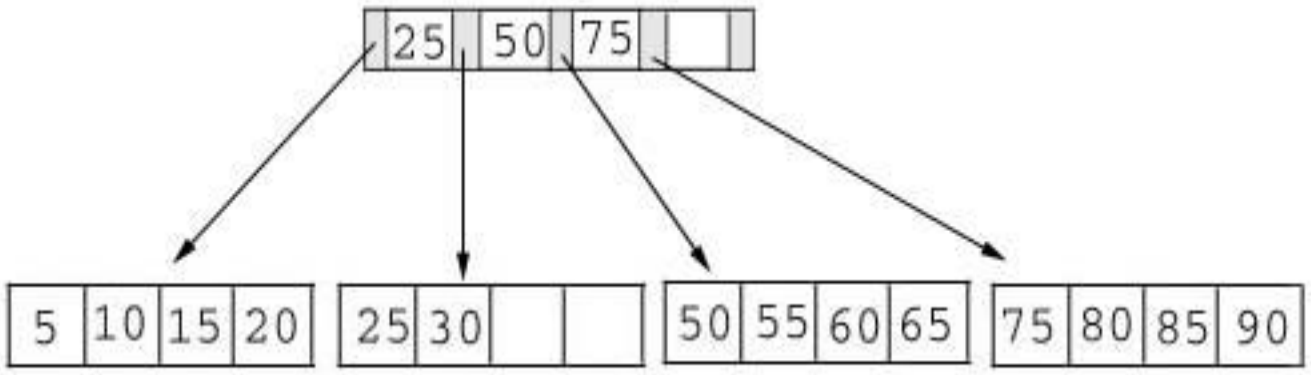
Advantages of B+ tree usage for databases

- 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.
- Special pointers linking the leaf nodes enable very fast range queries

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.



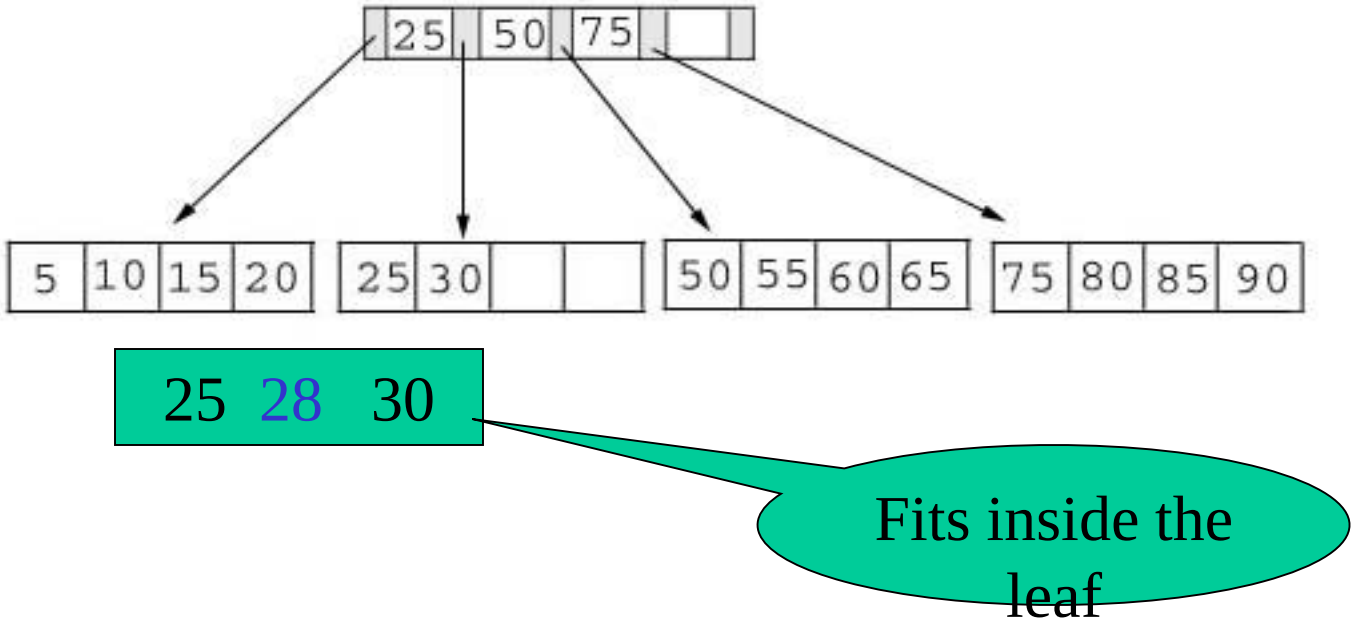
Searching

- Result:

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

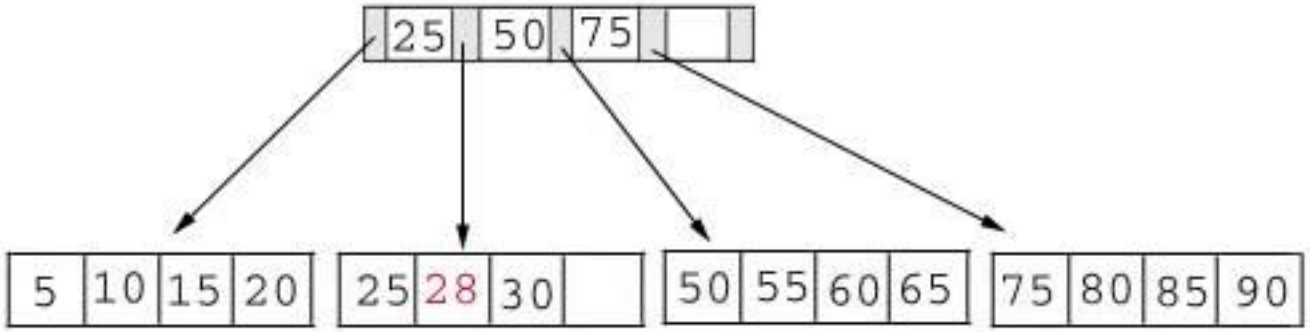
Insertion

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



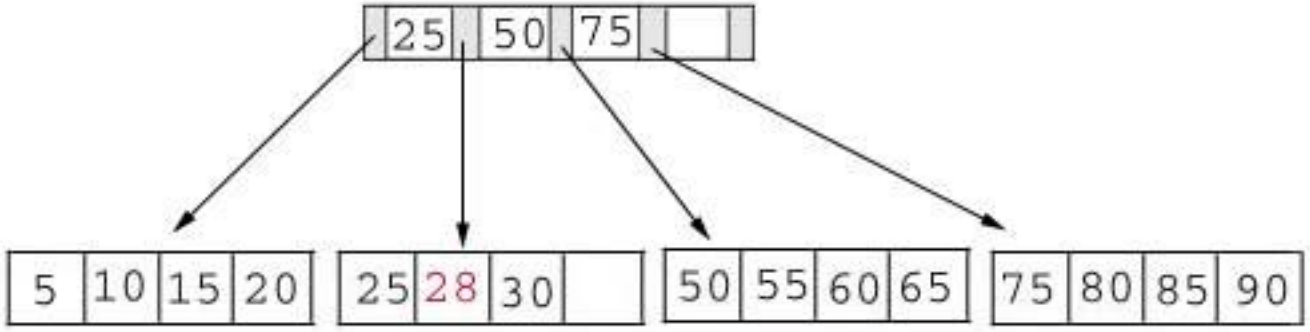
Insertion

- Result:



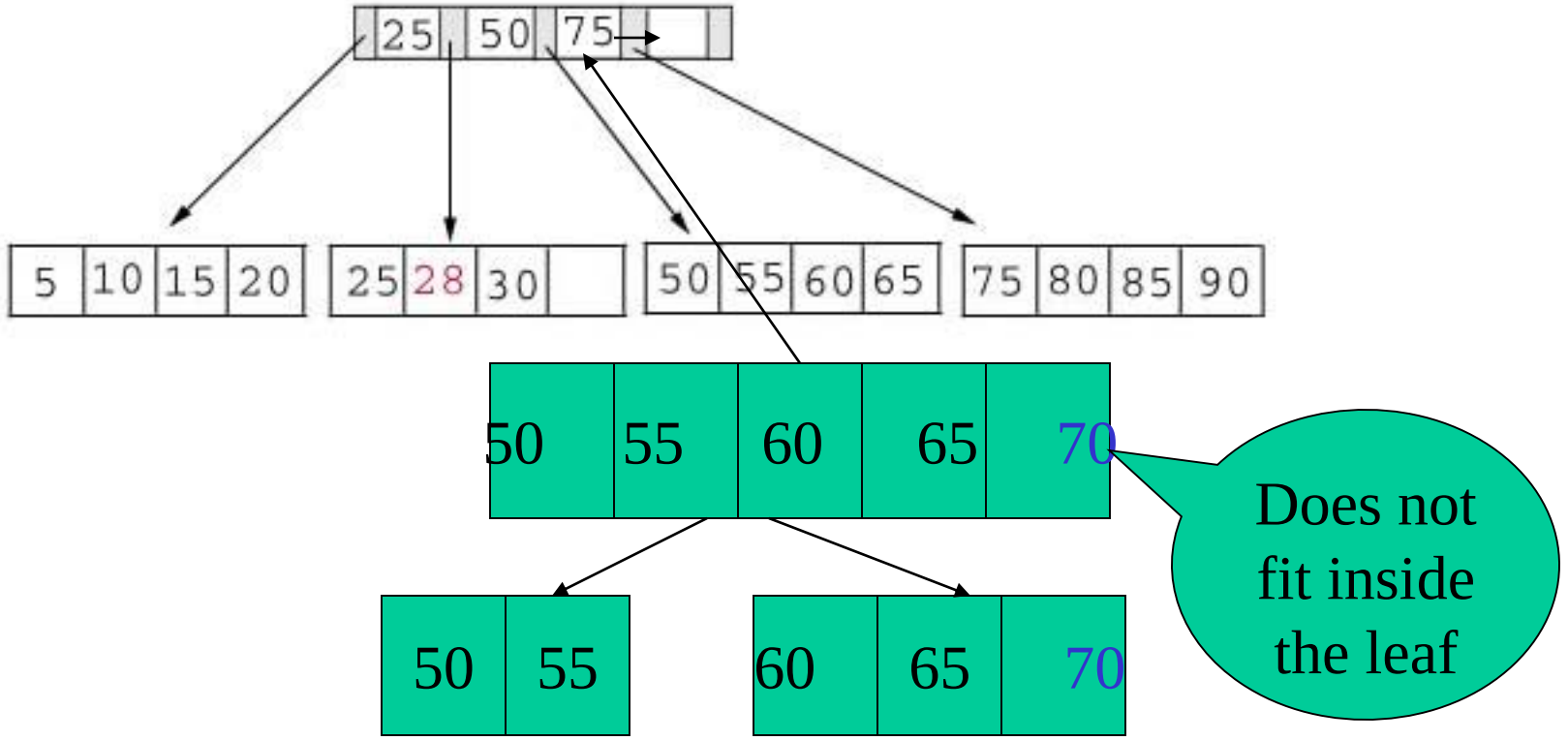
Insertion

- Example #2: insert 70 into the following tree



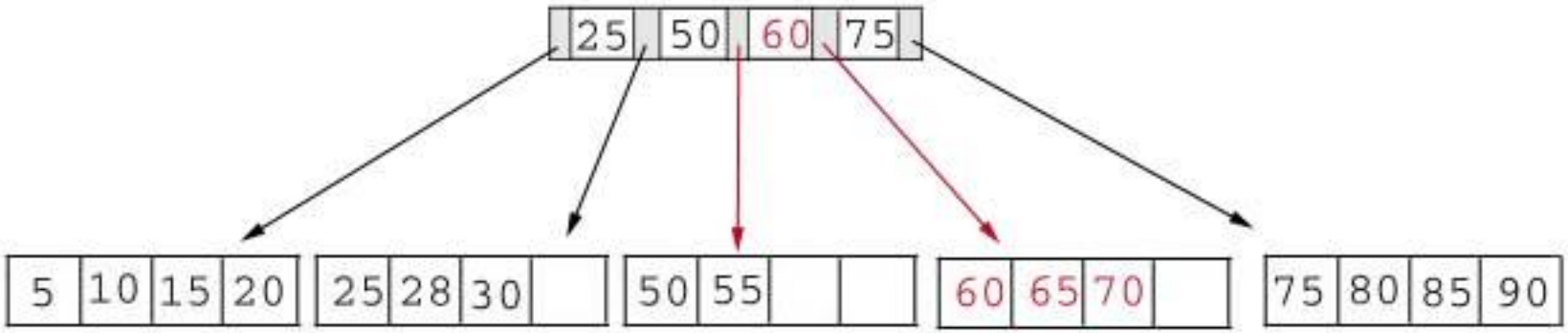
Insertion

- 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.



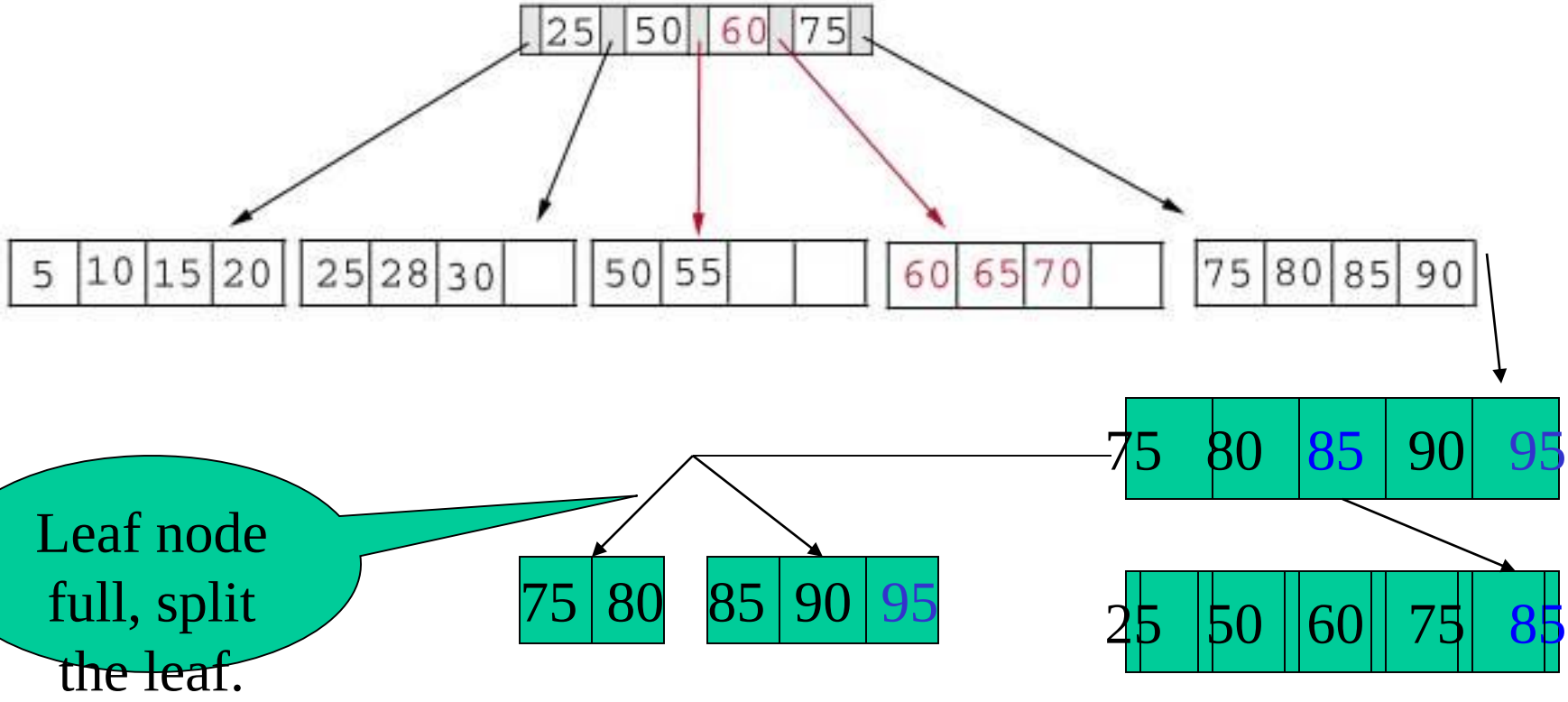
Insertion

The insert algorithm for B+ Tree

Leaf Node Full	Index Node Full	Action
NO	NO	Place the key with the record pointer in sorted position in the appropriate leaf node
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. keys < middle key go to the left leaf node (along with record pointers). 3. keys >= middle key go to the right leaf node (along with record pointers). 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.

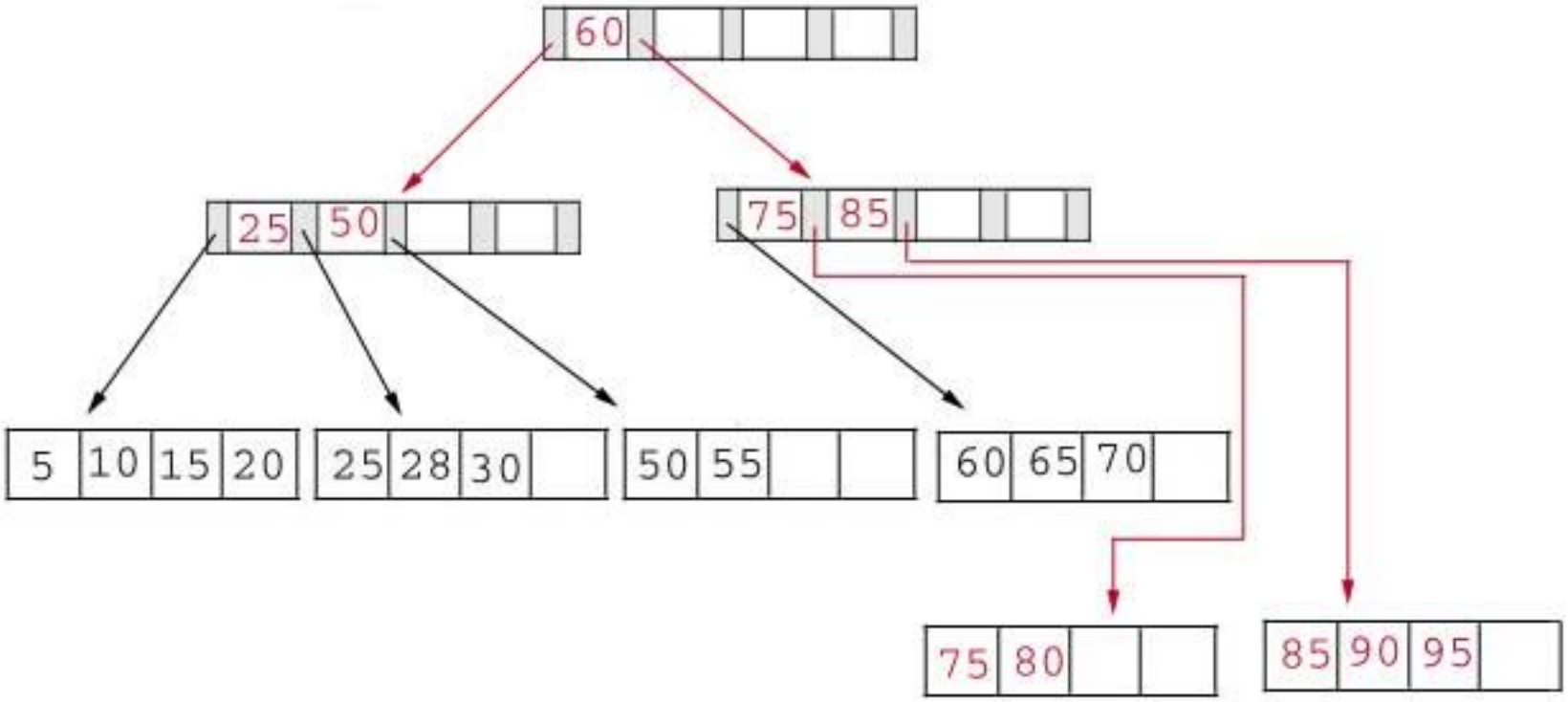
Insertion

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



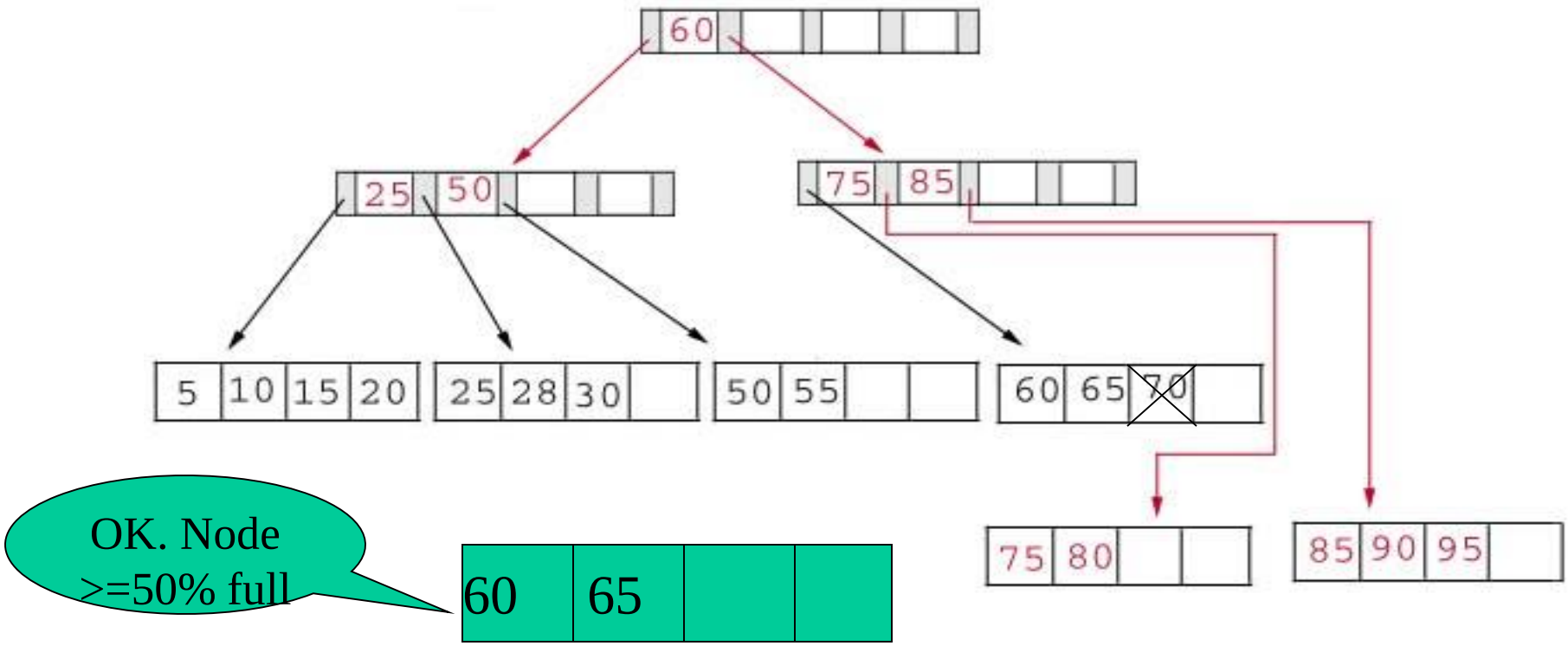
Insertion

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



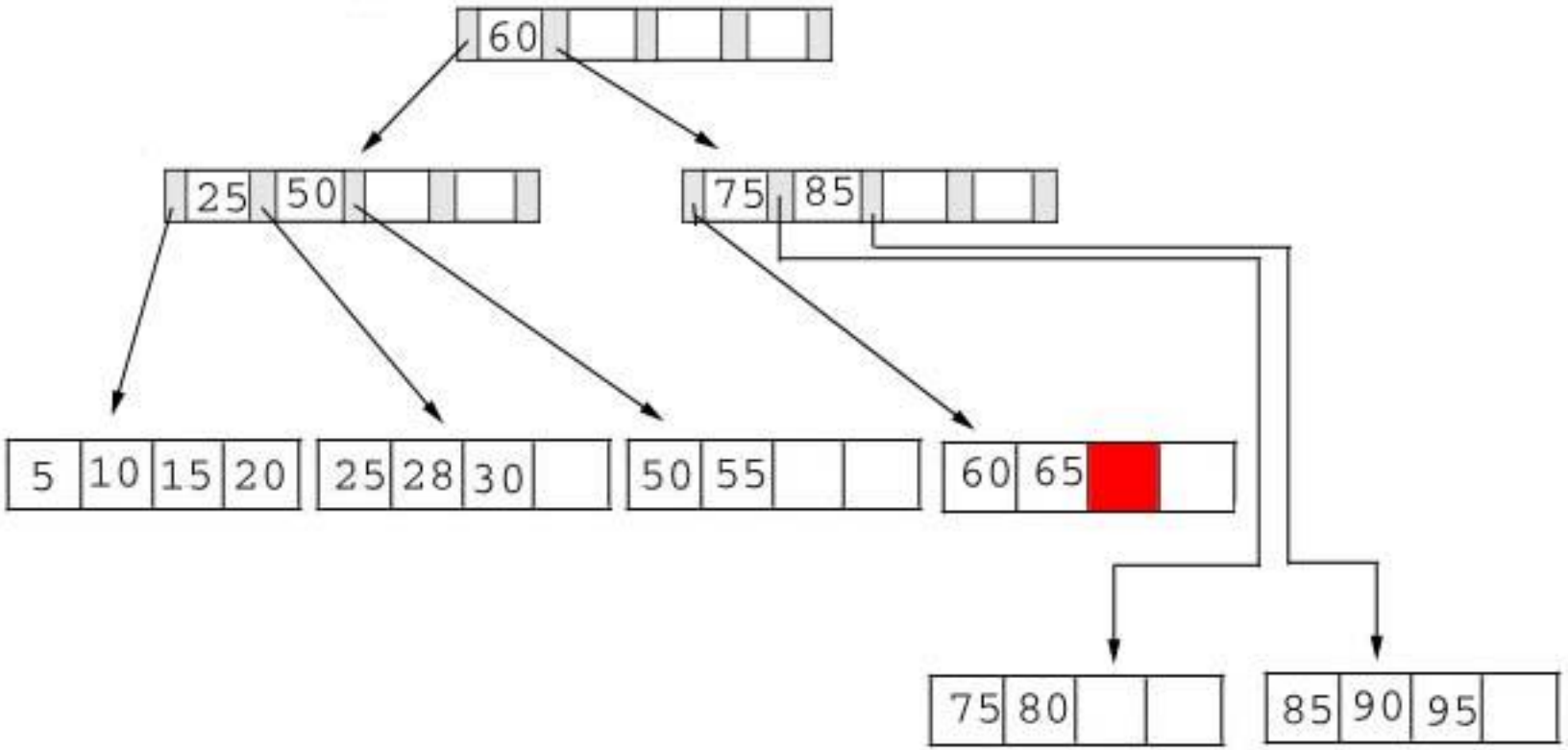
Deletion

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



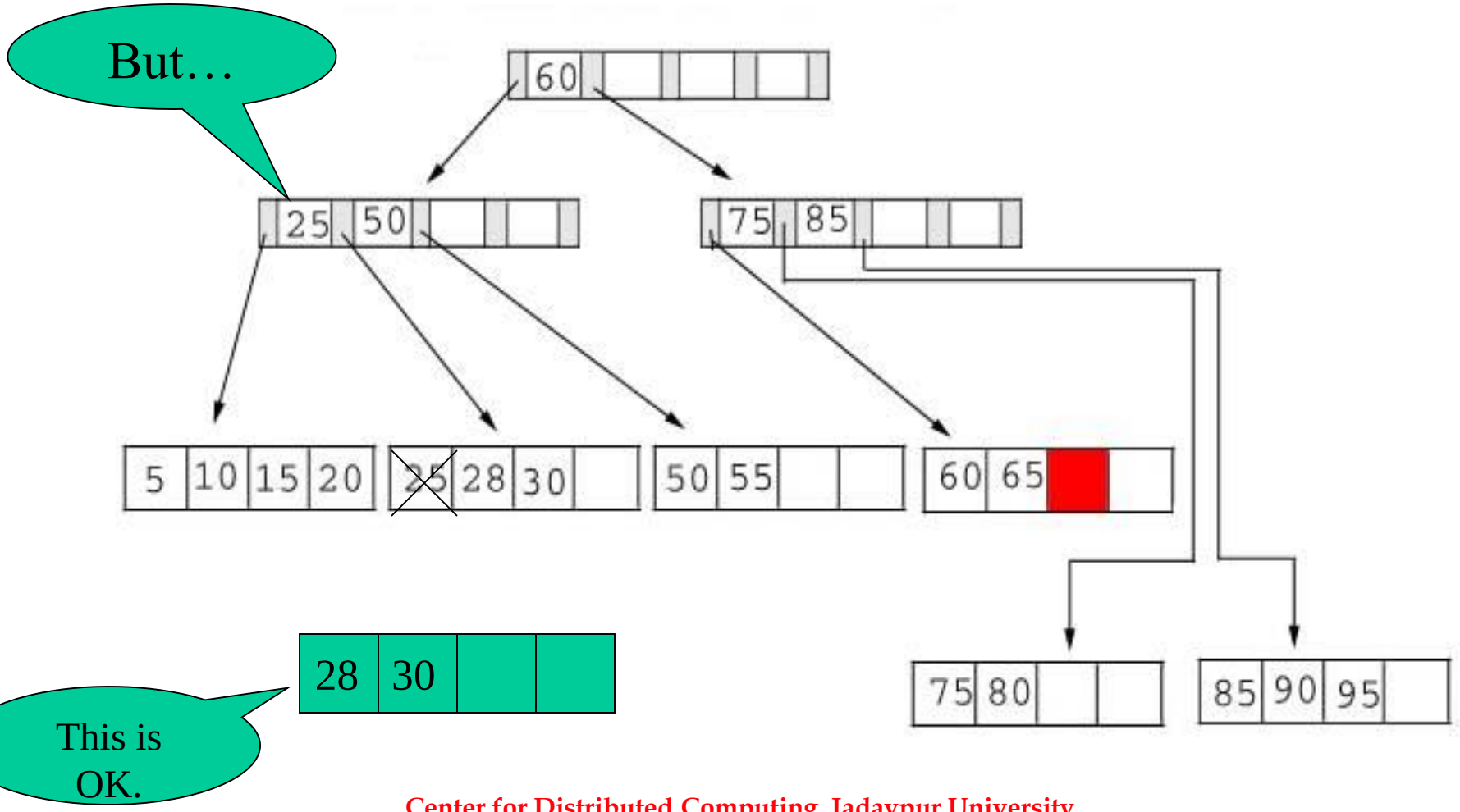
Deletion

- Result:



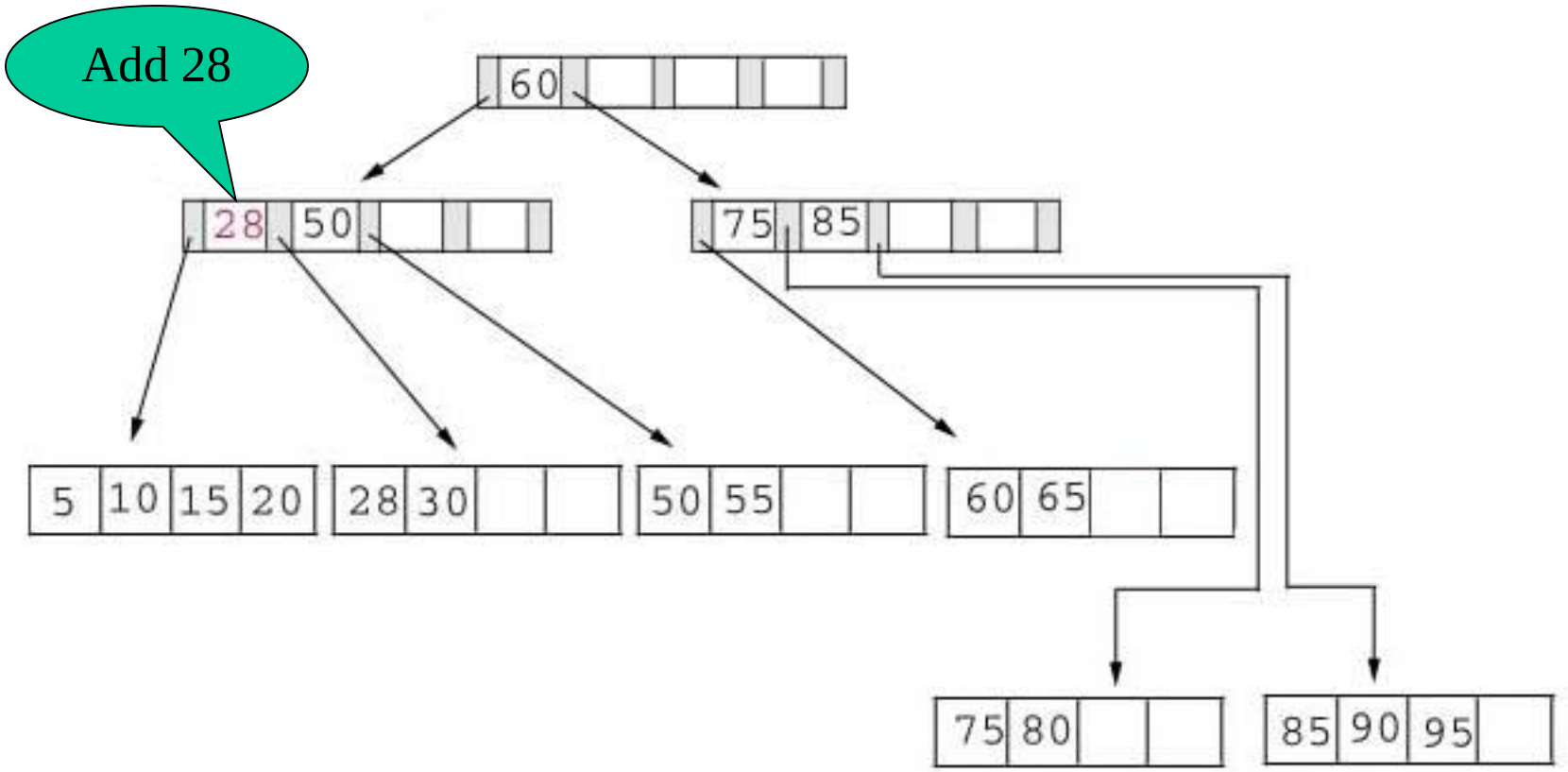
Deletion

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



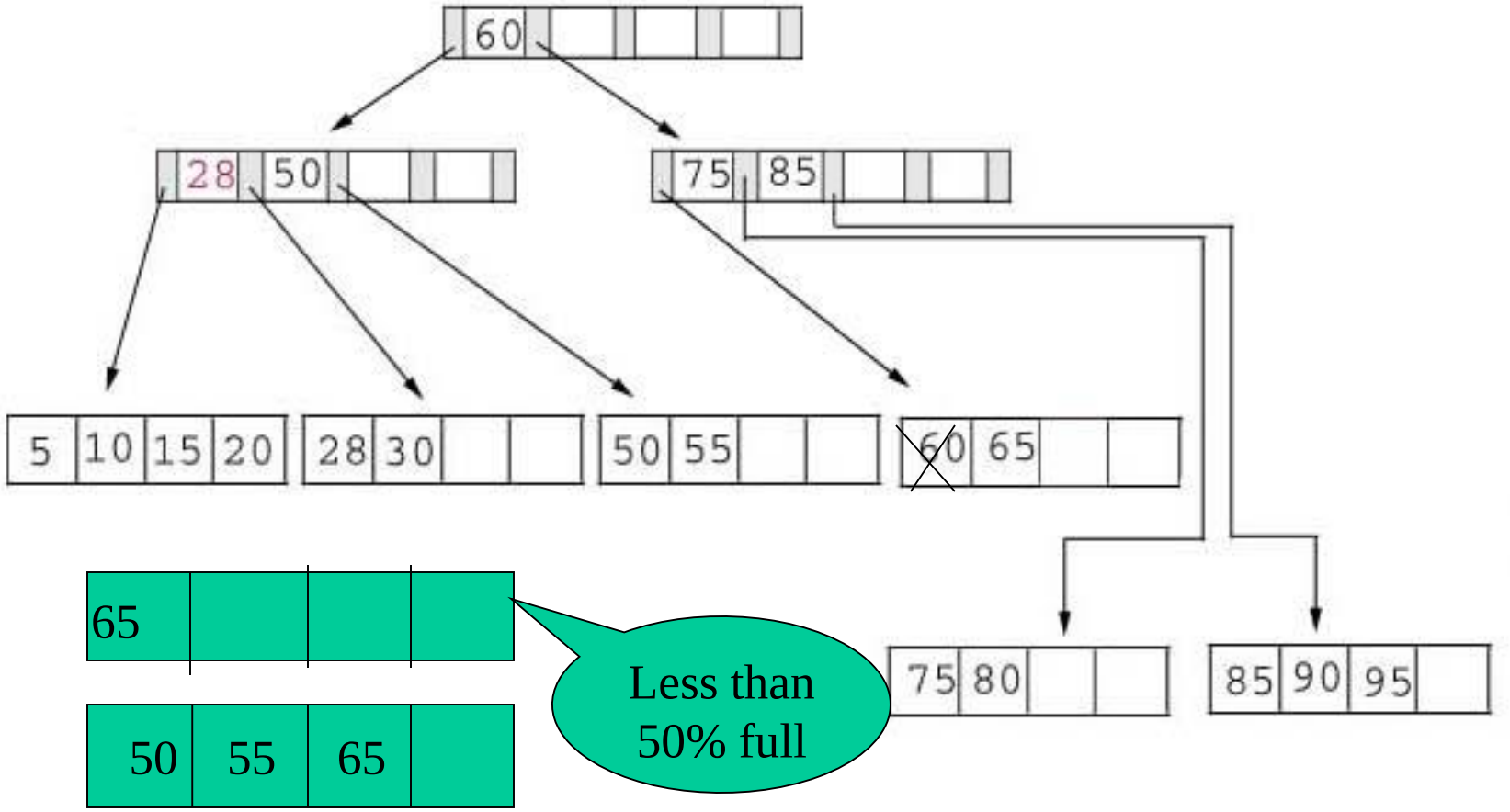
Deletion

- Result: replace 25 with 28 in the index page.



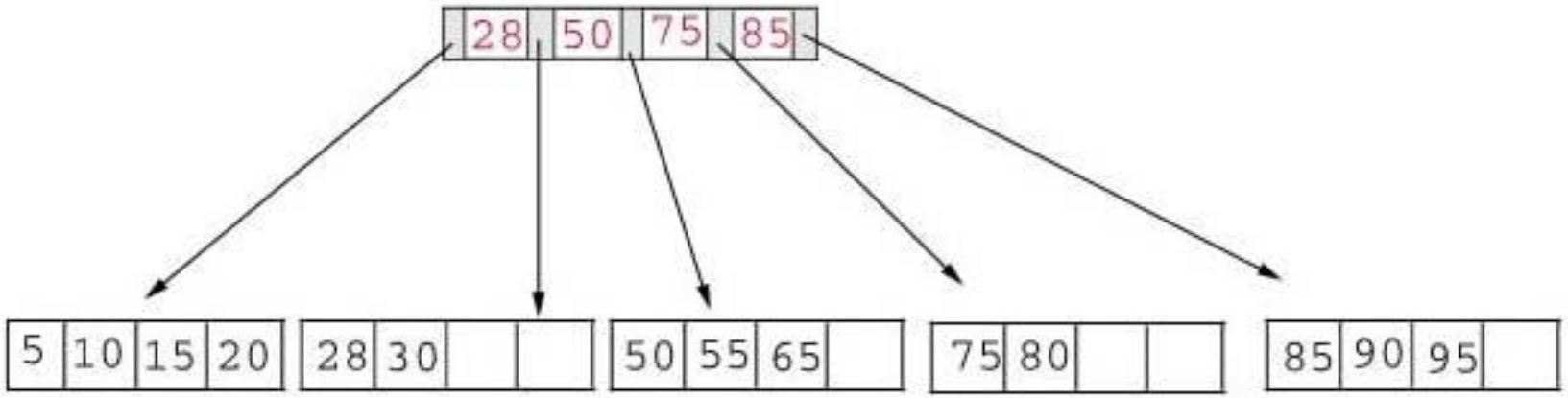
Deletion

- Example #3: delete 60 from the following 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 node. Arrange keys in ascending order to fill void. If the key of the deleted record appears in the index node, use the next key to replace it.
YES	NO	Merge the leaf node and its sibling. Change the index node to reflect the change.
YES	YES	1. Merge the leaf node and its sibling. 2. Adjust the index node to reflect the change. 3. Merge the index node with its sibling. Continue merging index nodes until you reach a node with the correct fill factor or you reach the root node.

Conclusion

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

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 fit 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

Applications

- B+ trees are used by
  - NTFS, ReiserFS, NSS, XFS, JFS, ReFS, and BFS file systems for metadata indexing
  - BFS for storing directories.
  - IBM DB2, Informix, Microsoft SQL Server, Oracle, Sybase ASE, and SQLite for table indexes, MySQL, Postgres, MongoDB, …