B/ B+ Trees

Kumkum Saxena

Motivation for B-Trees

- Index structures for large datasets cannot be stored in main memory
- Storing it on disk requires different approach to efficiency
- Assuming that a disk spins at 3600 RPM, one revolution occurs in 1/60 of a second, or 16.7ms
- Crudely speaking, one disk access takes about the same time as 200,000 instructions

Motivation (cont.)

- Assume that we use an AVL tree to store about 20 million records
- We end up with a **very** deep binary tree with lots of different disk accesses; $\log_2 20{,}000{,}000$ is about 24, so this takes about 0.2 seconds
- We know we can't improve on the log *n* lower bound on search for a binary tree
- But, the solution is to use more branches and thus reduce the height of the tree!
  - As branching increases, depth decreases

Definition of a B-tree

- A B-tree of order *m* is an *m*-way tree (i.e., a tree where each node may have up to *m* children) in which:
  1. the number of keys in each non-leaf node is one less than the number of its children and these keys partition the keys in the children in the fashion of a search tree
  2. all leaves are on the same level
  3. all non-leaf nodes except the root have at least $\lceil m / 2 \rceil$ children
  4. the root is either a leaf node, or it has from two to *m* children
  5. a leaf node contains no more than *m* – 1 keys
- The number *m* should always be odd

Structure of binary search tree node

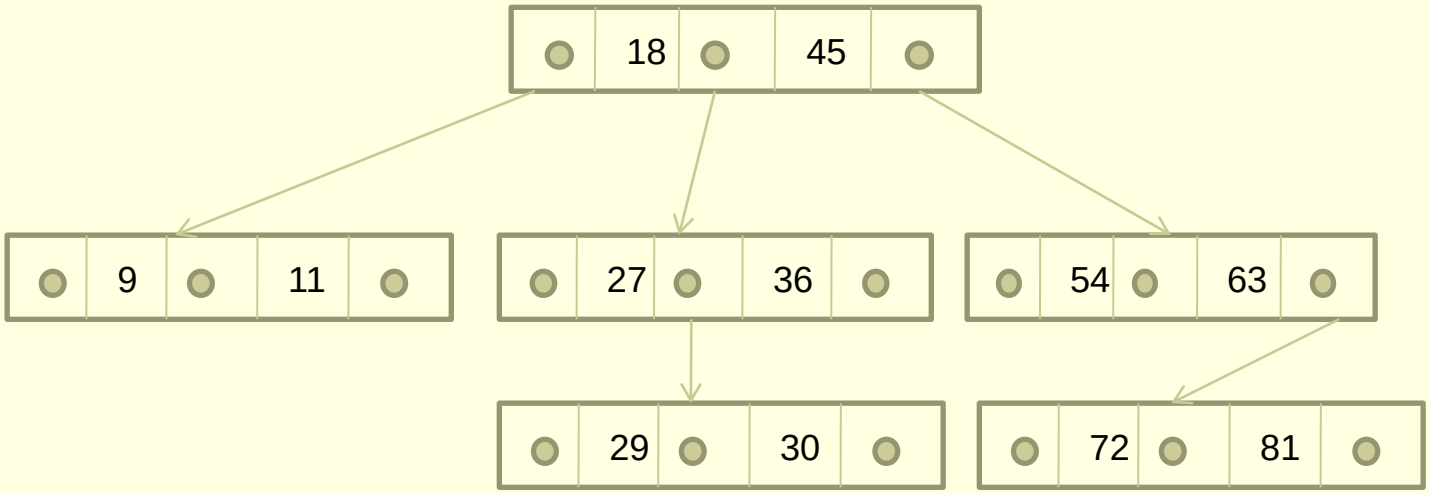
POINTER TO LEFT SUB TREE	VALUE OR KEY OF THE NODE	POINTER TO RIGHT SUB TREE

B tree of order **m** can have a maximum **m-1** keys and **m** pointers to its sub-tree. Storing a large number of keys in the single node keeps the height of tree relatively small. In addition it has the following properties:

- Every node in the B-tree has atmost **m** children
- Every node in B-tree except the root node and leaf node has atleast **m/2** children. This condition helps to keep the tree bushy so that path from root node to leaf node is very short, even in a tree that stores lot of data.
- All leaf nodes are at the same level.

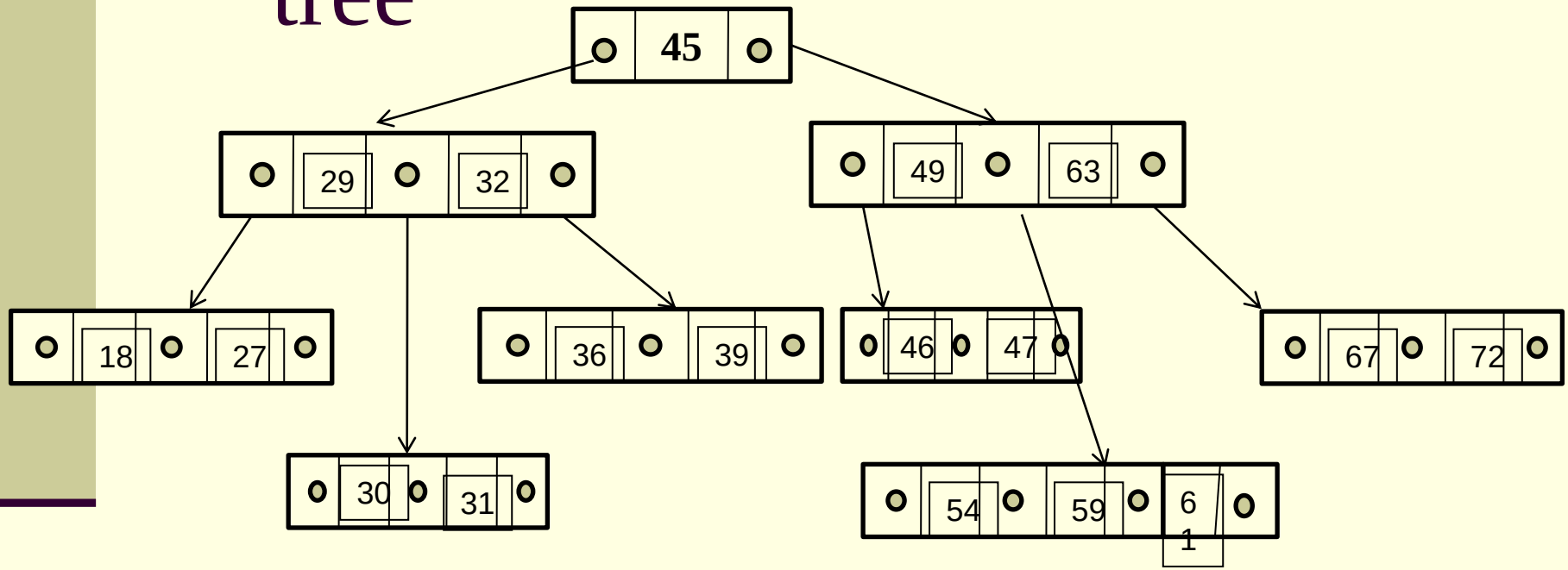
Structure of M-way search tree

- In the structure shown P0,P1,….PN are the pointers to the node subtree and k0,k1,k2…k_{n-1} are key values of the node. All key values are stored in ascending order. i.e. ki<k_{i+1} for 0<=i<=n-2



M-way search tree of order 3

Searching for element in B-tree



Search value 59 and 9

Searching for element in B-tree

- Searching for element in B-tree is similar to that in BST.
- To search for 59, we begin at the root node.
- Root has value 45 which is less than 59. so we traverse to right sub-tree. Right sub tree of root node has two key values 49 and 63.
- Since 49<=59<=63. now we traverse the right sub tree of 49 that is the left sub-tree of 63.
- This sub-tree has three values 54 59 61. on finding value 59 search is successful

Inserting new node in B-tree

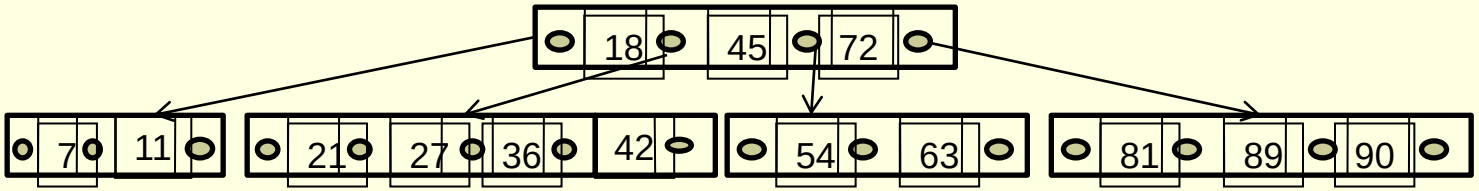
- In a B-tree, all insertions are done at leaf level node. A new value is inserted in B-tree using algorithm given below.
- Search the B-tree to find the leaf node where new key value should be inserted.
- If leaf node is not full that contains less than m-1 key values, then insert the new element in the node keeping the node's element ordered
- If node is full then

  A) insert new values in order into existing set of values.

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

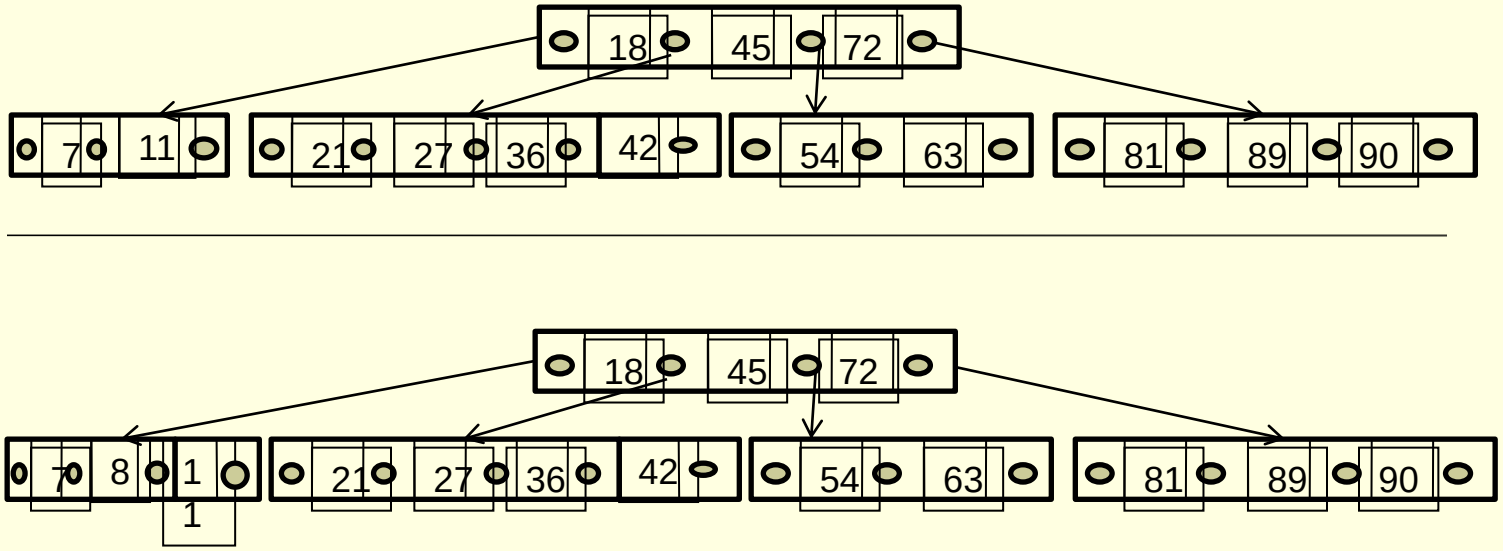
  C)push median at its parent node. If the parent node is already full then split parent node by following same steps

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



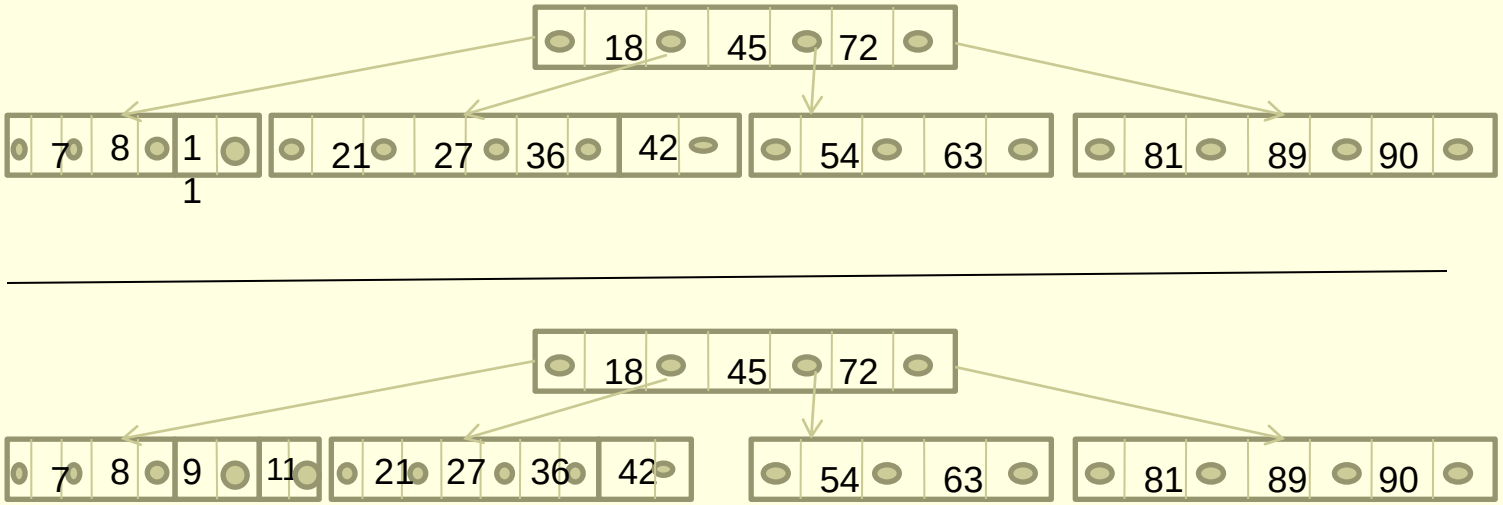
B- tree with order 5

Step 1
Insert 8



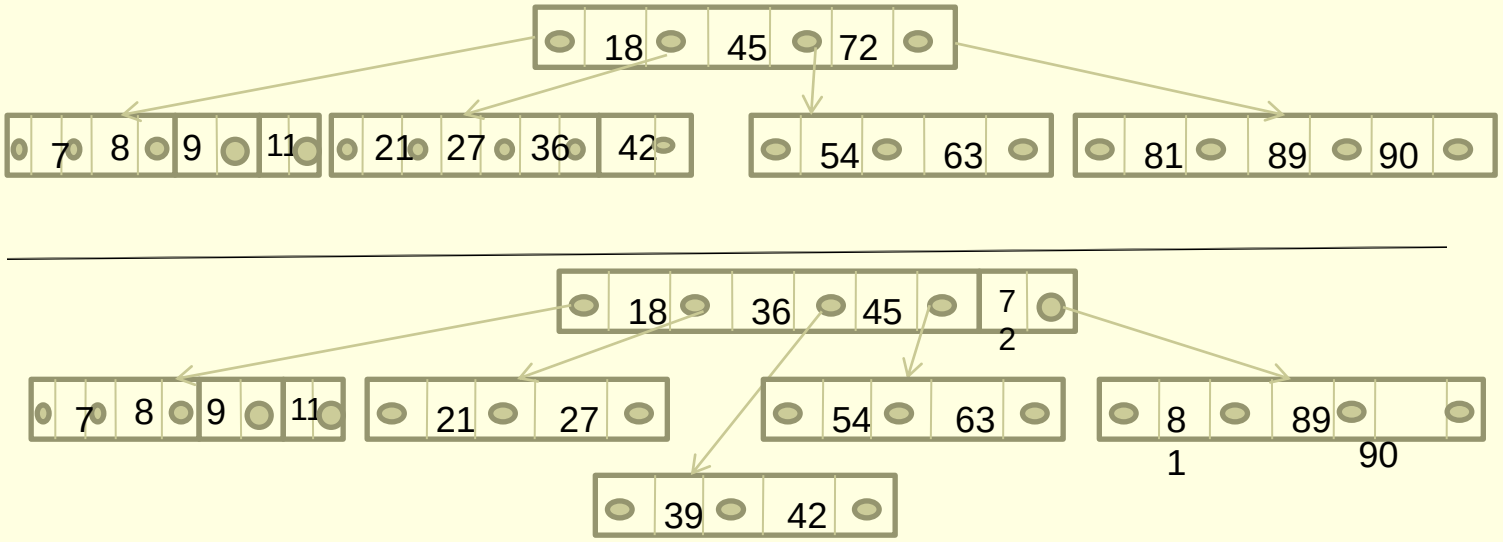
After inserting new value 8

Step 2
Insert 9



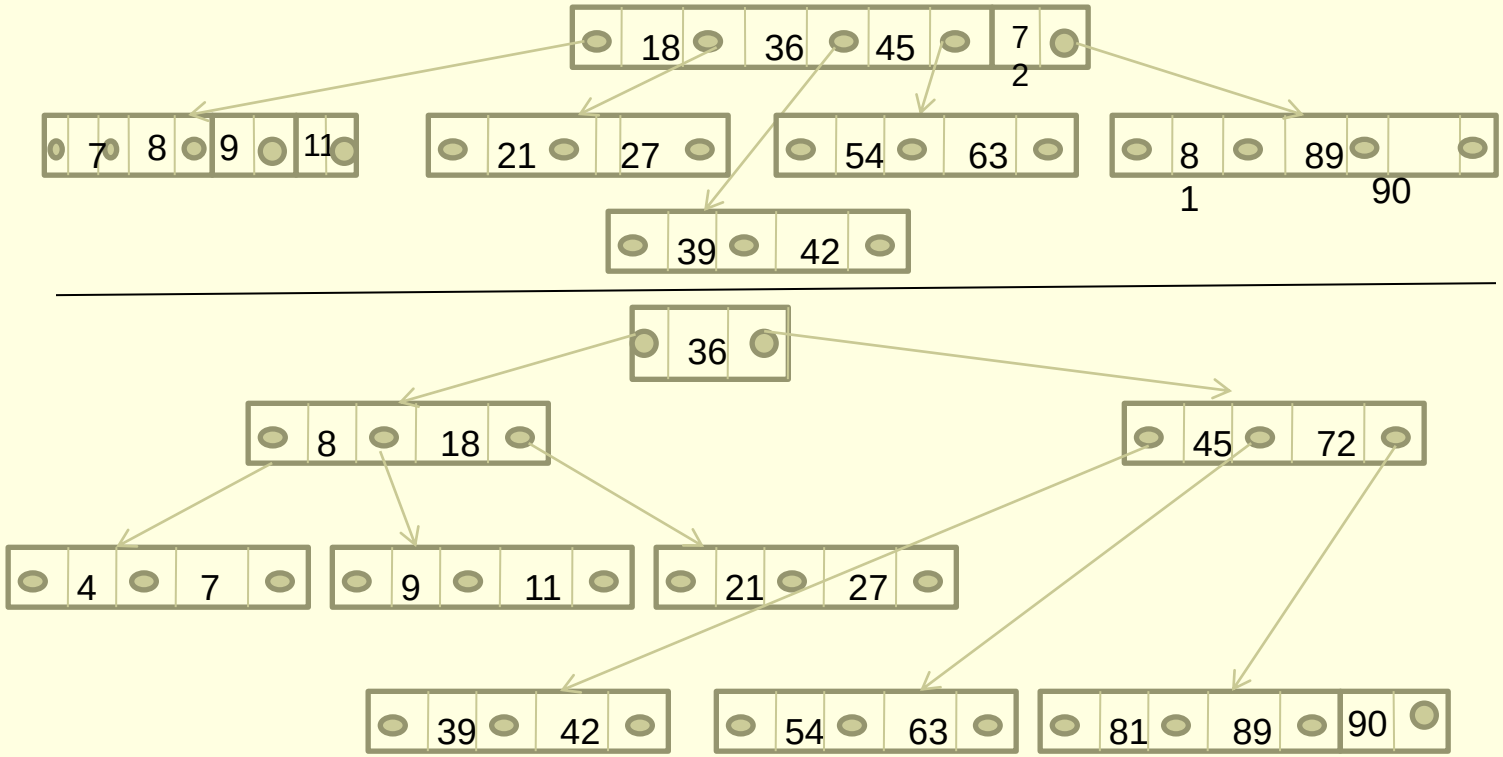
After inserting new value 9

Step 3
Insert 39



After inserting new value 39

Step 4
Insert 4



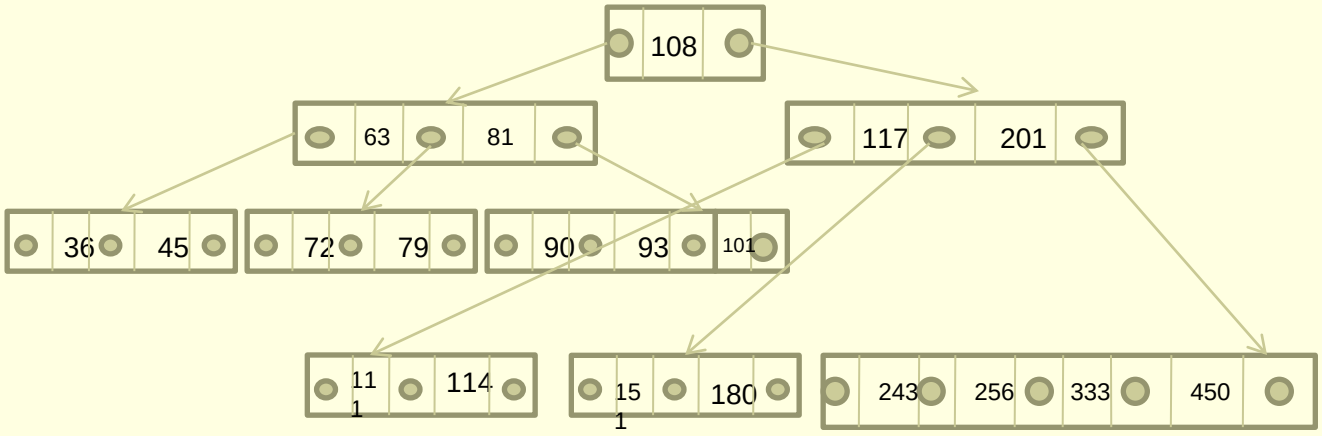
Deleting node from B-tree

- Locate the leaf node has to be deleted
- If leaf node contains more than the minimum number of key values(i.e. m/2) then delete the value.
- Else if leaf node does not contain m/2 elements then fill the node by taking an element either from left or from right siblings.
  - If left sibling has more than minimum number of key values , then push its largest key into parents node and pull down the intervening element from parent to leaf node where the key is deleted
  - Else if right sibling has more than minimum number of key values , then push its smallest key into parents node and pull down the intervening element from parent to leaf node where the key is deleted
- Else if both left and right siblings contain only the minimum number of element then create new leaf node by combining two leaf nodes and intervening elements of parent node

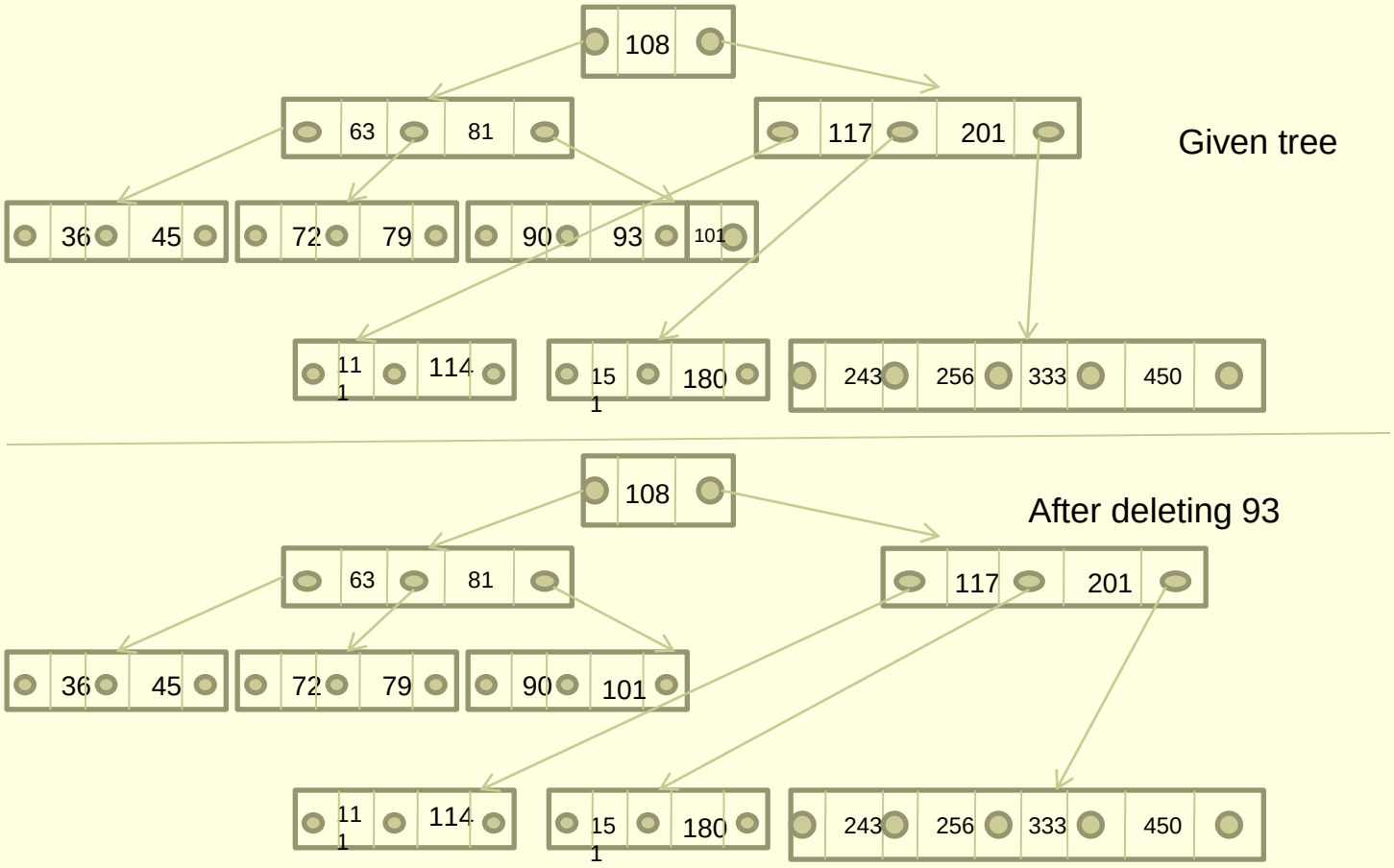
Deleting node from B-tree continue…..

- To delete an internal node, promote the successor or predecessor of the key to be deleted to occupy the position of deleted key. This predecessor and 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

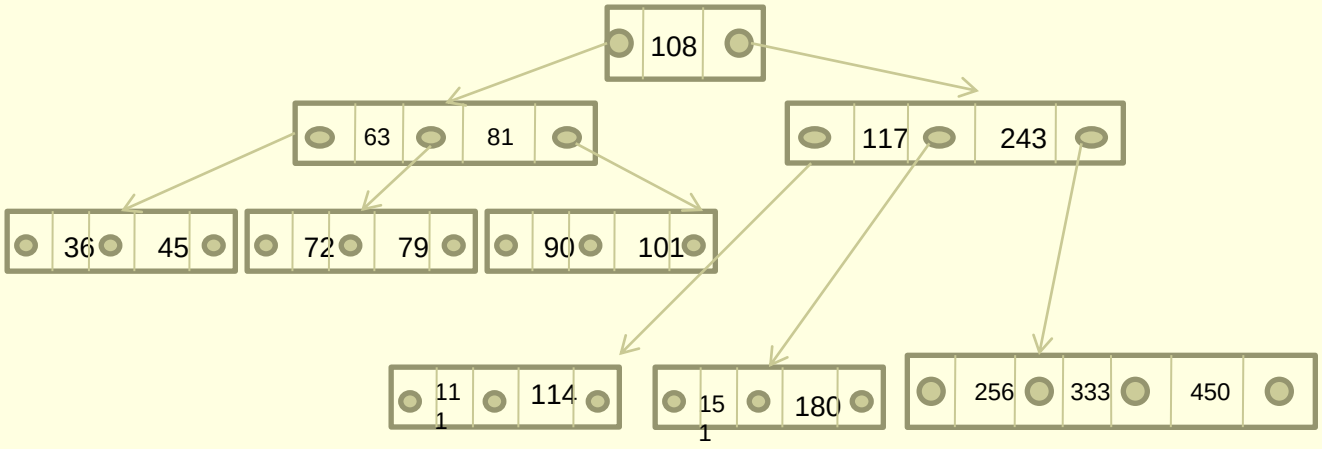
Consider the following B-tree of order 5 and delete values 93,201,180 and 72 from it.



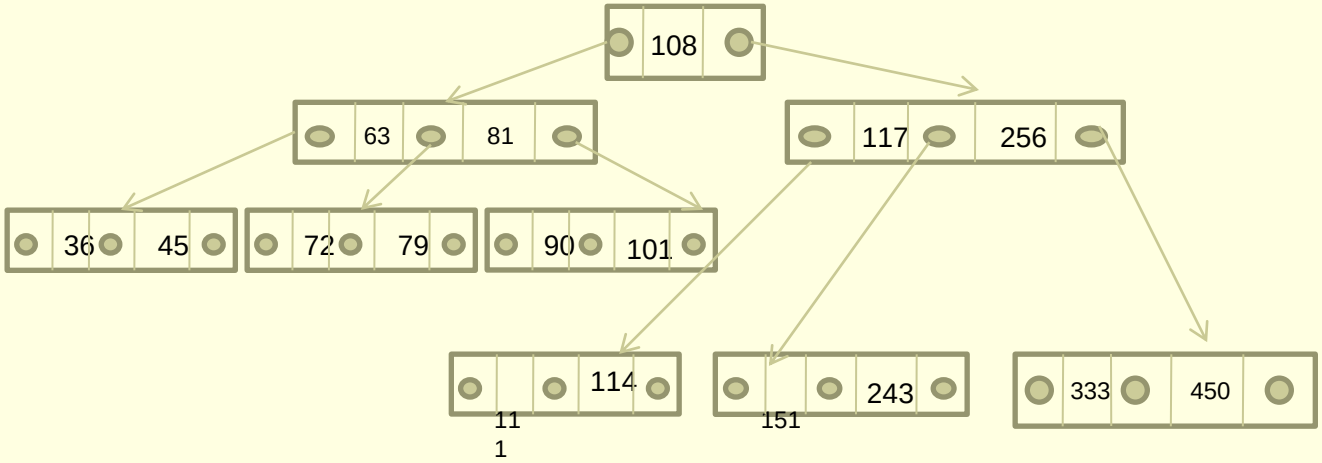
Delete 93


108
63
81
117
201
36
45
72
79
90
93
101
11
1
114
15
1
180
243
256
333
450
Given tree
108
After deleting 93
63
81
117
201
36
45
72
79
90
101
11
1
114
15
1
180
243
256
333
450

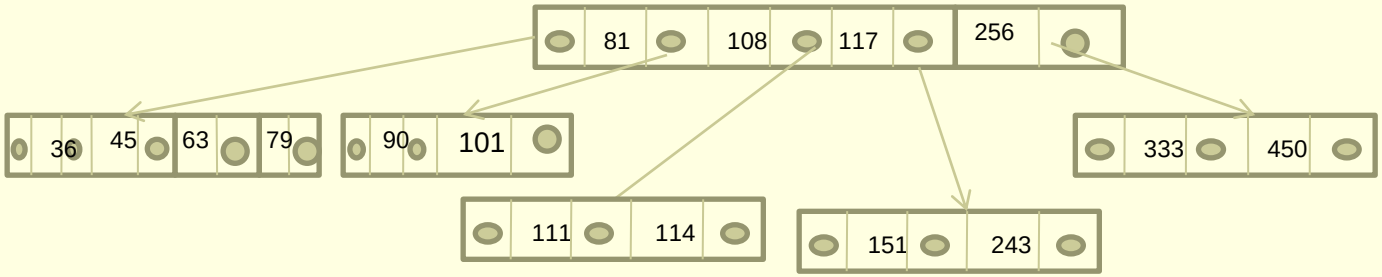
Delete 201



Delete 180



Delete 72



2.B+ Trees

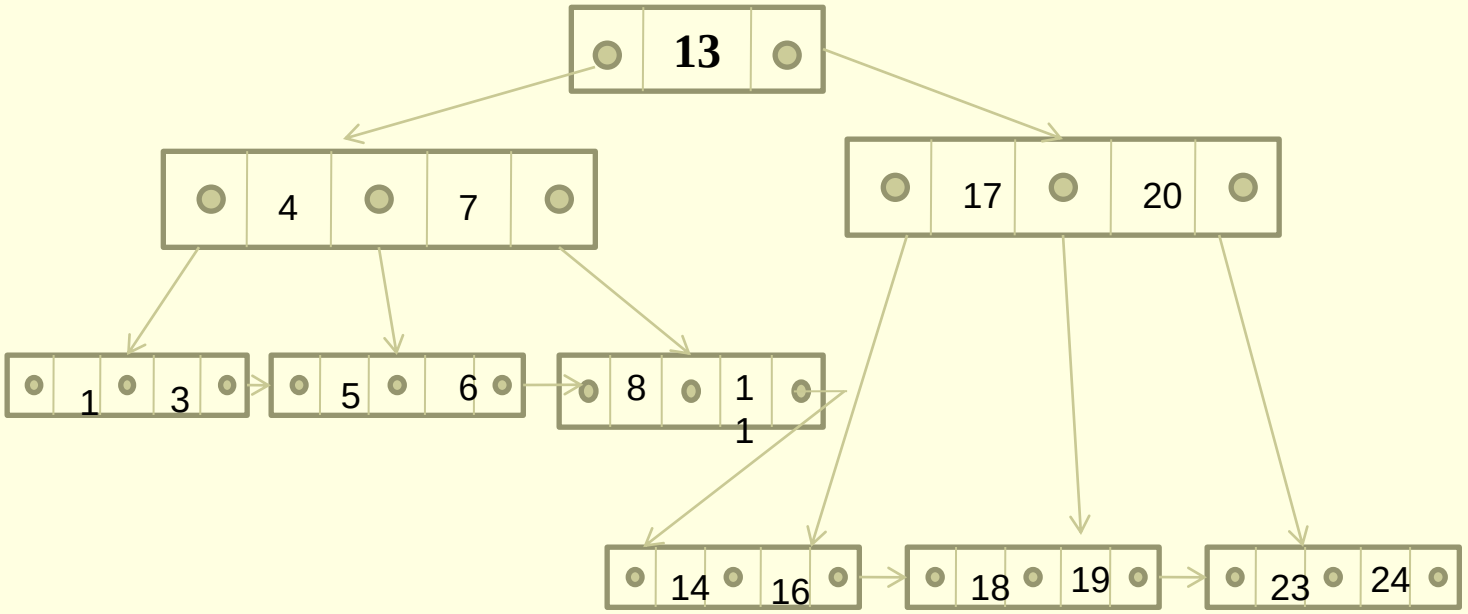
B+ tree

- B+ tree is variant of B-tree which stores sorted data in a way that allows for efficient insertion retrieval and deletion of records, each of which is identified by a *key.*
- while B-tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree. Only keys are stored in its interior nodes
- Leaf nodes of B+ tree are often linked to one another in linked list. This has an added advantage of making queries simpler and more efficient

- typically, B+ trees are used to store large amount of data that can not be stored in main memory.
- With B+ tree the secondary storage is used to store the leaf nodes of trees and internal nodes of trees are stored in the main memory
- Records are stored only at the leaf node and all other nodes are called as index-nodes or i-nodes. This nodes allows us to traverse the tree from root down to leaf node that stores desire data items

- Many database systems are implemented using b+ tree because of its simplicity, since all the data appears in the leaf node and are ordered. The tree is always balanced and makes searching for data efficient.
- A B+ tree can be thought of as a multi-level index in which the leaves make up a dense index and the non leaf node make up the sparse index. The advantage of B+ trees can be given as follows:
  - Record can be fetched in equal number of disk accesses
  - It can be used to perform a wide range of queries easily as leaves are linked to nodes at the upper level.
  - Height of tree is less and balanced
  - Support both random and sequential access to records
  - Keys are used for indexing

B+ tree of Order 3



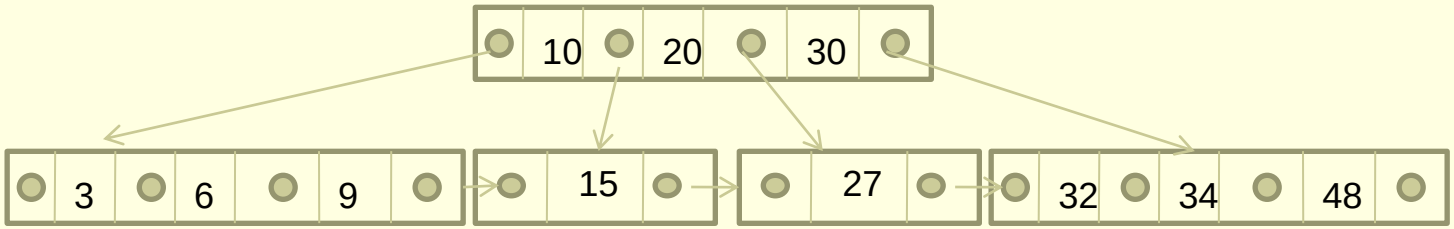
Operations that we can perform on B+ tree

- Search
- Insert
- Delete

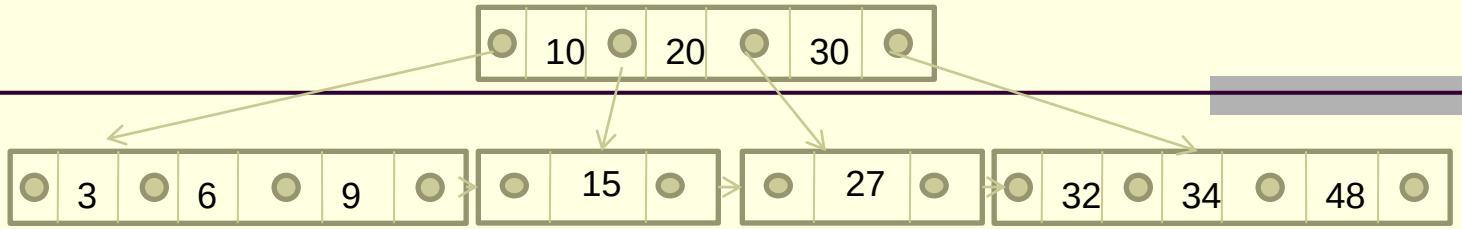
Inserting data into B+ tree

- Insert new node as leaf node
- If the leaf node overflows, split the node and copy the middle element to next index node
- If the index node overflows split that node and move middle element to next index page.

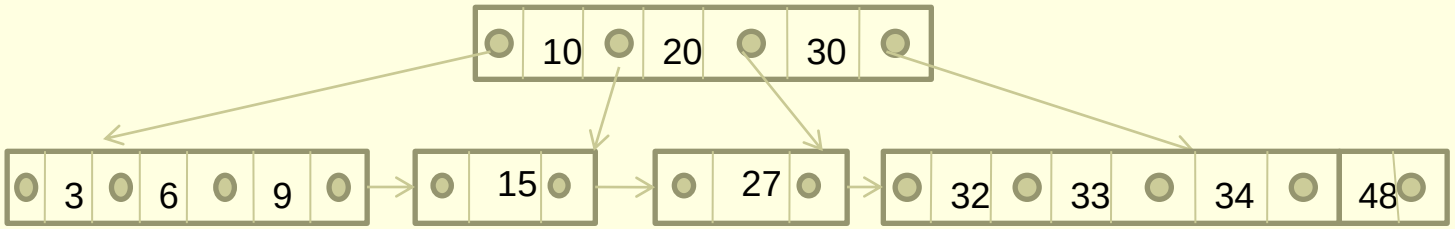
Given B+ tree of order 4



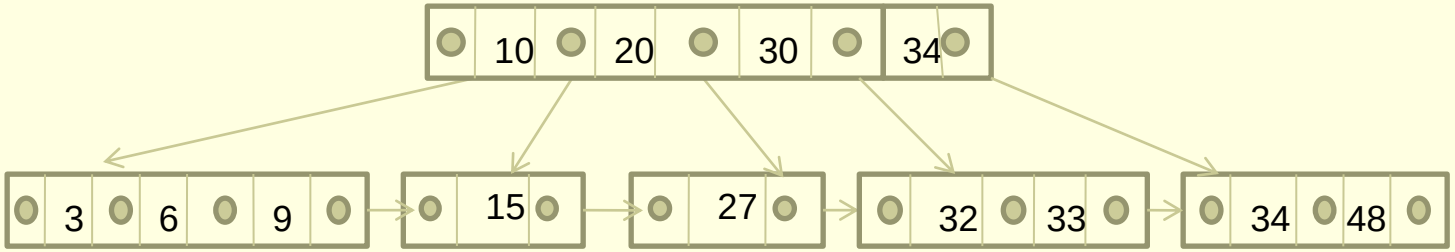
Insert 33 in given B+ tree of order 4



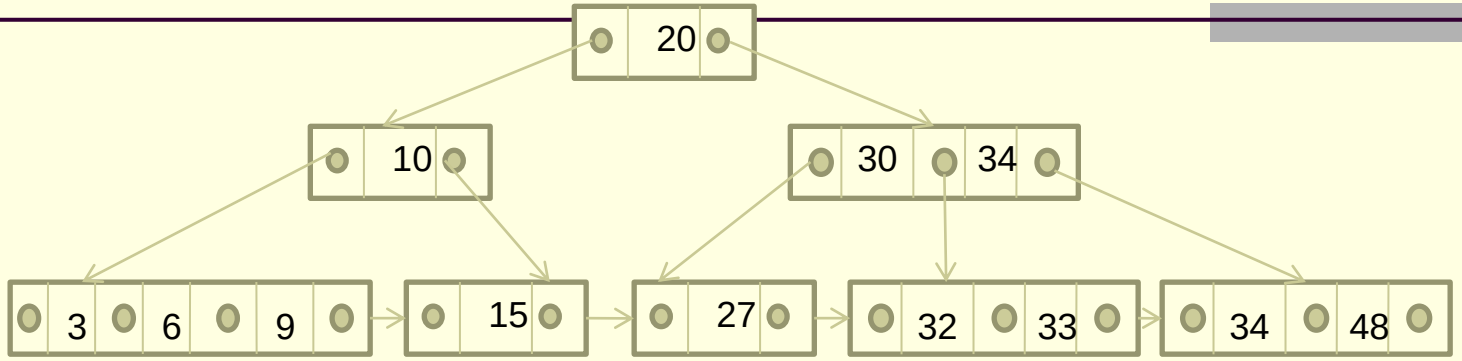
Insert 33



Split the leaf node



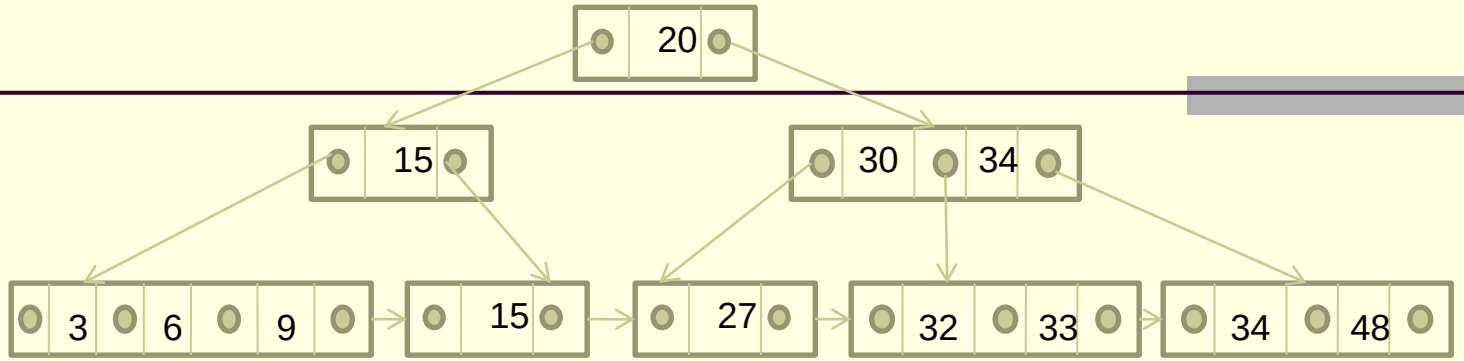
Split the leaf node



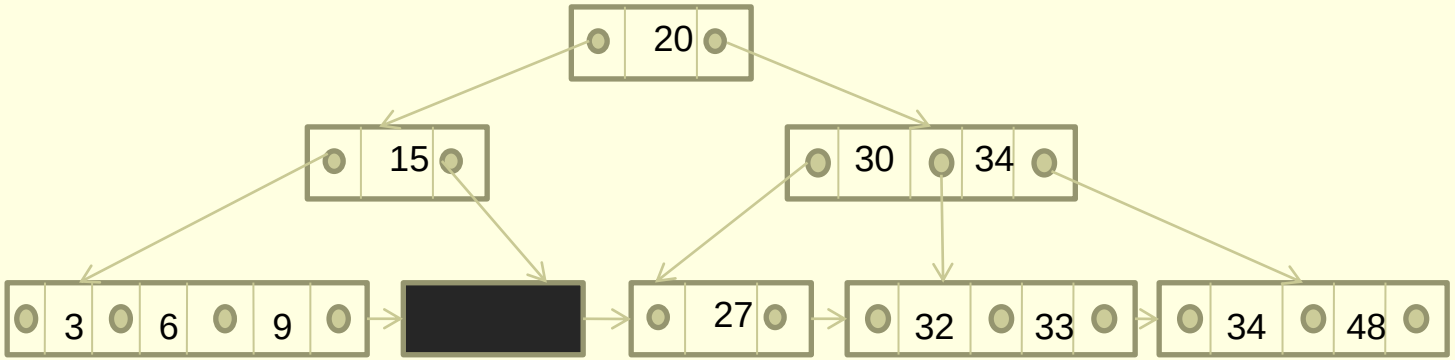
Inserting node 33 in given B+ tree

Deleting data from B+ tree

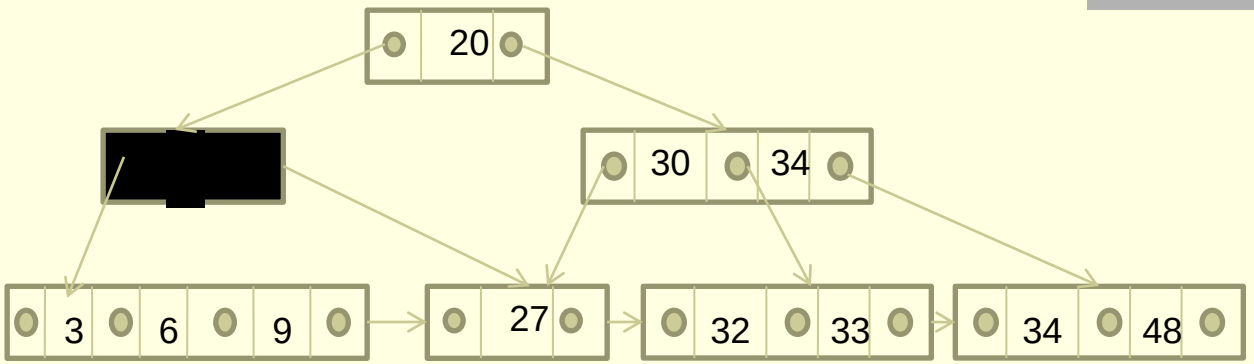
- Delete the key and data from leaves
- If leaf node underflows, merge that node with sibling and delete the key in between them
- If index node underflows, merge that node with sibling and move down the key in between them.



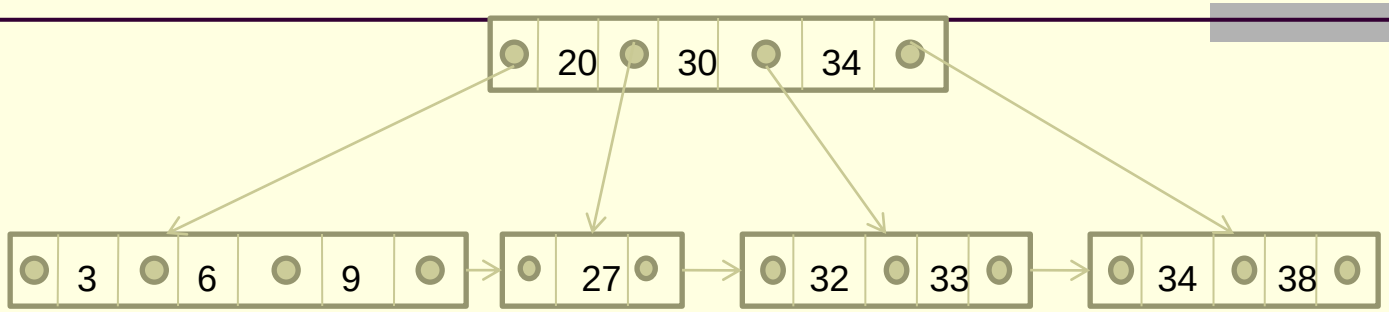
Deleting node 15 in given B+ tree



Leaf node underflows so merge with left sibling and remove key 15



Index node underflows so merge with sibling and delete the node.



Deleting node 15 from given B+ tree

Insertion and deletion operations are recursive in nature and
can cascade up and down B+ tree , thereby affecting its shape dramatically

Compare B and B+ tree

B tree	B+ tree
Data is stored in internal or leaf node	Data is stored in leaf node only
Searching takes more time as data may be found in leaf and non-leaf node	Searching data is very easy as the data can be found in leaf node only
Deletion of leaf node is very complicated	Deletion is very easy because data is in the leaf node
The structured and operations are complicated	The structure and operations are simple

APPLICATION OF TREES

- Trees are used to store simple as well as complex data.
- Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
- Another variation of tree, the B-trees are eminently used to store tree structures on disc. They are used to index a large number of records.
- Trees are an important data structures used for compiler construction.
- Trees are also used in database design.