

**MET's Institute of Engineering**

**Bhujbal Knowledge City, Adgaon, Nashik.**

**Department of Computer Engineering**

# **“INDEXING AND MULTIWAY TREE”**

**Prepared By**

**Prof. Anand N. Gharu**

**(Assistant Professor)**

**Computer Dept.**

**CLASS : SE COMPUTER 2019**

**SUBJECT : DSA (SEM-II)**

**UNIT : V**

**15 April 2024**

Unit V	Indexing and Multiway Trees	(07 Hours)
<b>Indexing and Multiway Trees-</b> Indexing, indexing techniques-primary, secondary, dense, sparse, Multiway search trees, B-Tree- insertion, deletion, B+Tree - insertion, deletion, use of B+ tree in Indexing, Trie Tree.		
<u>#Exemplar/Case Studies</u>	Heap as a Priority Queue	
<u>*Mapping of Course Outcomes for Unit V</u>	CO2, CO3, CO5	

# Indexing Techniques

- Indexing is used to speed up retrieving of data.
- A Separate sequential file is maintained for indexing.
- Index file contains a key field and a pointer to main file where actual data is stored.

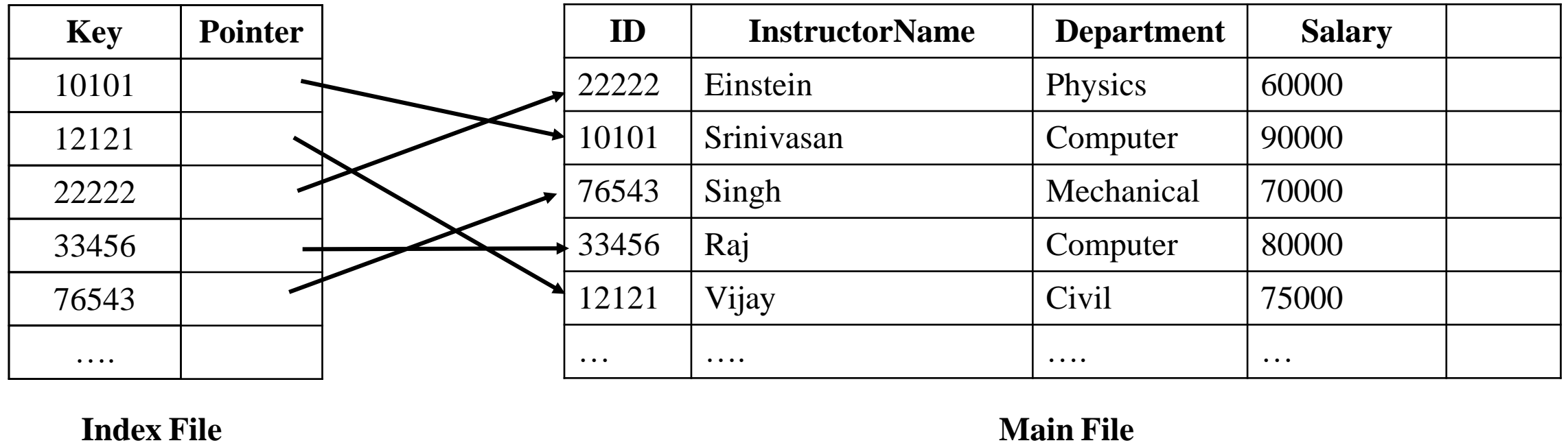


- Address of data from index file is used to retrieve data from main file.
- Index files are typically much smaller than the original file.

## Advantages:

1. Indexing provide better flexibility. Key are ordered.
2. Requires less storage compared to main file. Only Keys are stored and pointers.
3. Inserting new records doesn't affect main file. Need to update index structure.

# Indexing Techniques

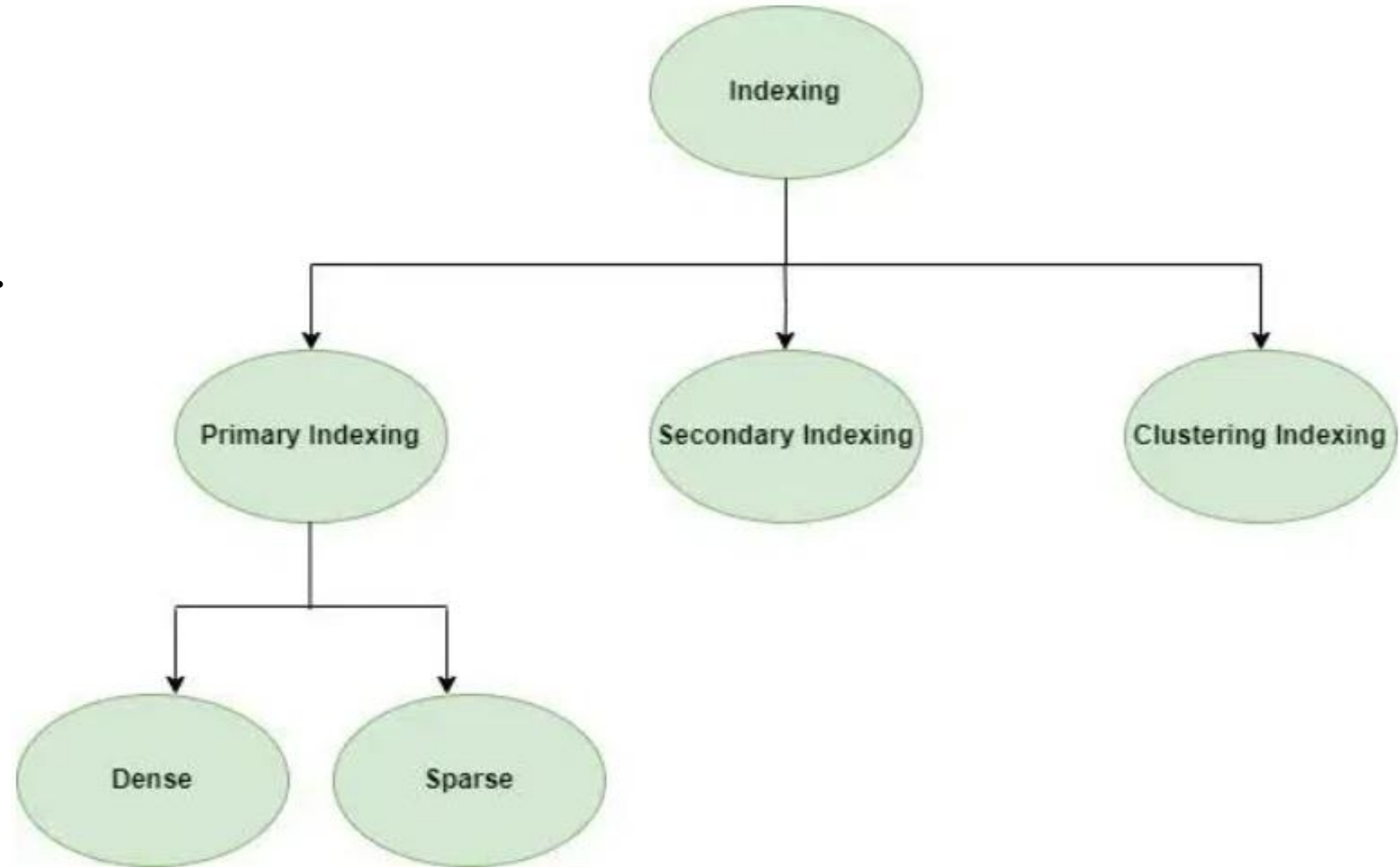


## Disadvantages :

1. **Increased database maintenance overhead:**
2. **Choosing an index can be difficult:**
3. Indexing can reduce insert and update performance since the index data structure must be updated each time data is modified
4. Indexing necessitates more storage space to hold the index data structure, which might increase the total size of the database.

# Indexing Techniques

- **Primary Indexing.**
  - Dense Indexing.
  - Sparse Indexing.
- **Secondary Indexing.**



# Indexing Techniques

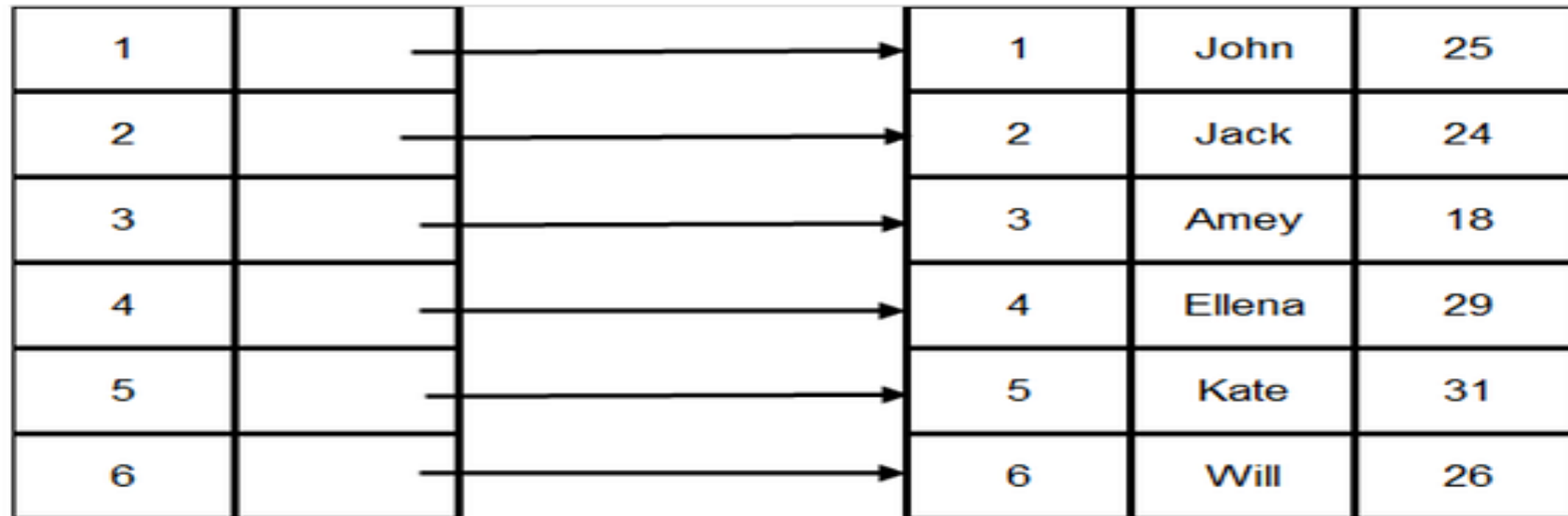
## Primary Indexing :

Primary indexing only has two columns. First column has the primary key values which are the search keys. The second column has the pointers which contain the address to the corresponding data block of the search key value. The table should be ordered and there is a **one-to-one** relationship between the records in the index file and the data blocks. This is a more traditional yet a fast mechanism.

# Indexing Techniques

## Dense Index :

There is an index record that contains a search key and pointer for every search key value in the data file. Though the Dense index is a fast method it requires more memory to store index records for each key value



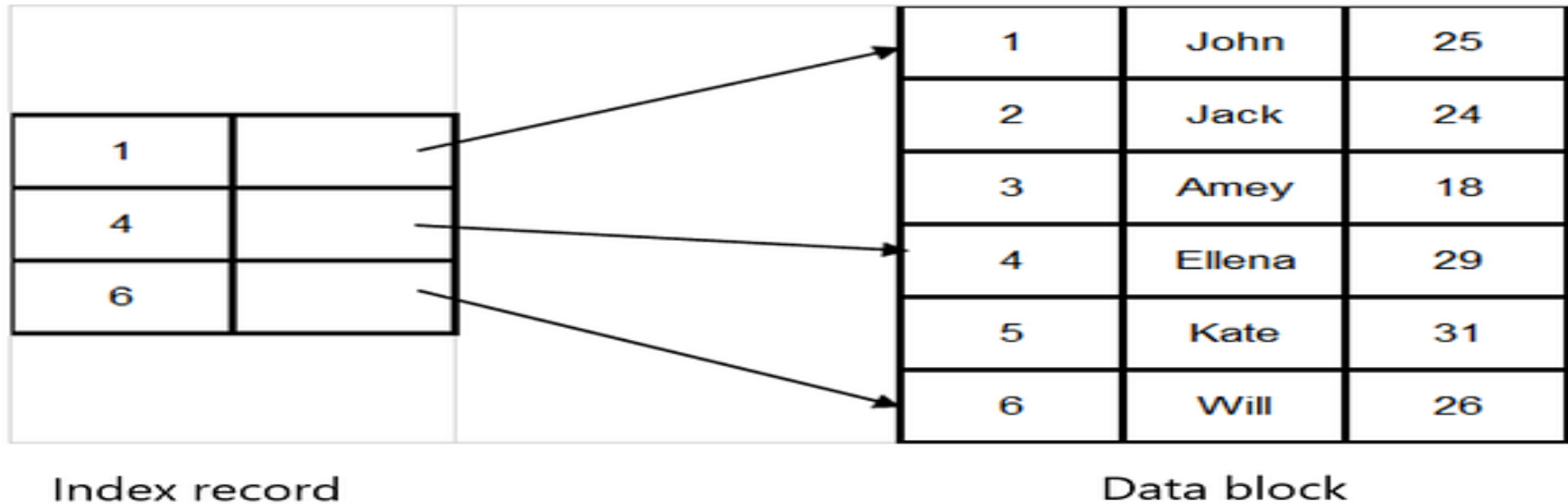
Index record

Data block

# Indexing Techniques

## Sparse Index :

There are only a few index records that point to the search key value. First, the index record starts searching sequentially by pointing to a location of a value in the data file until it finds the actual location of the search key value. Though sparse indexing is time-consuming, it requires less memory to store index records as it has less of them.

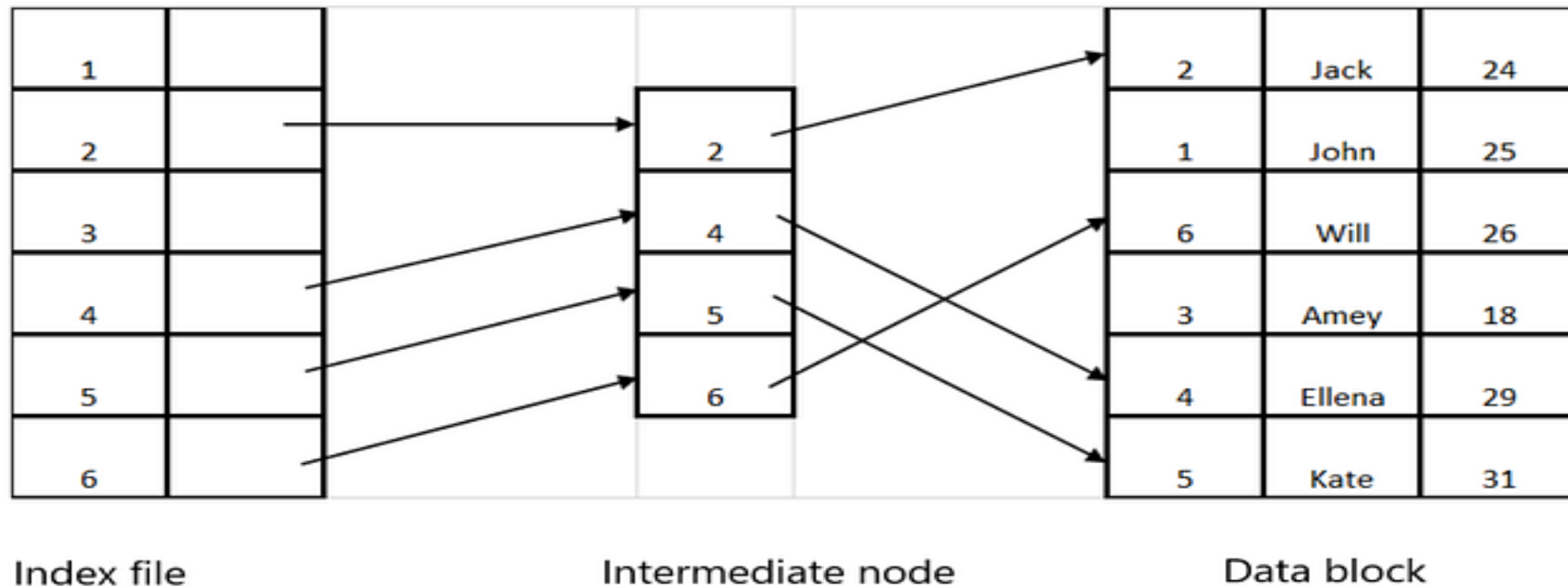




# Indexing Techniques

## Secondary Indexing (Non clustered Indexing) :

The columns in the Secondary indexing hold the values of the candidate key along with the respective pointer which has the address to the location of the values. Index and data are communicated with each other through an intermediate node.

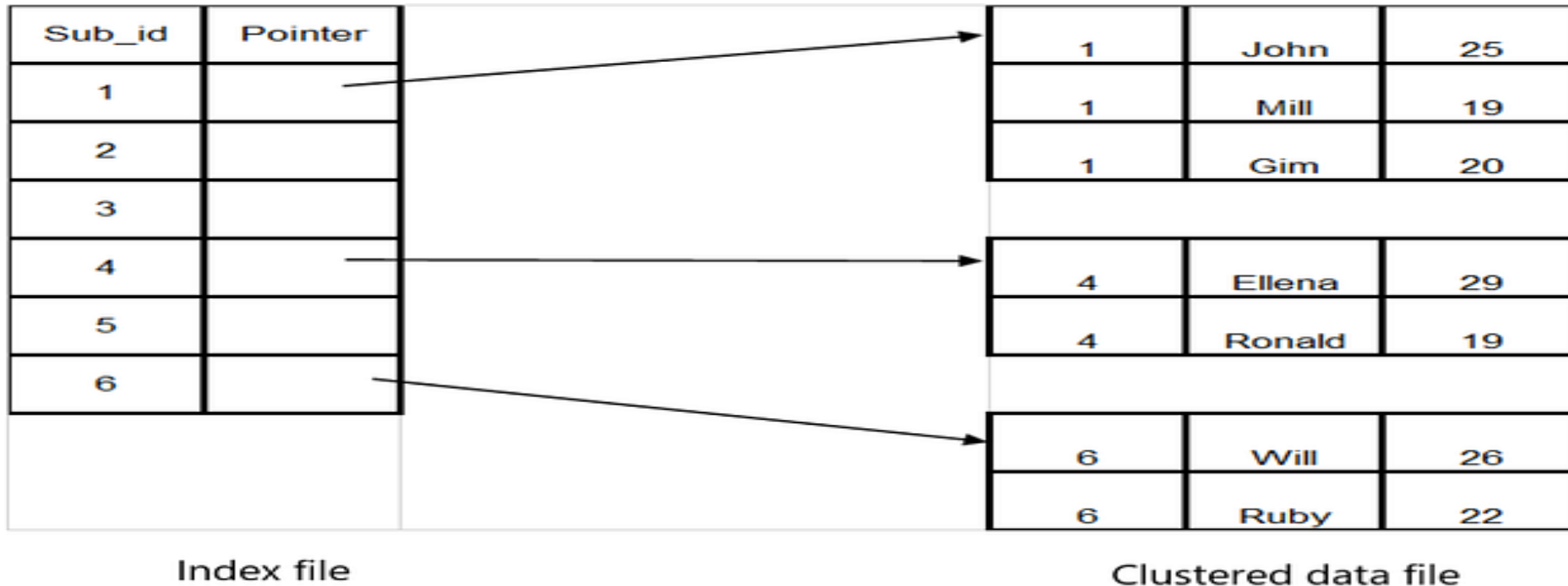


# Indexing Techniques

## Clustered Indexing :

The table is ordered in clustered indexing. At the times when the indexes are created using the non-primary key, we combine two or more columns together to get the unique values to identify data uniquely and use it to create the index.

The pointers are pointed to the cluster as a whole.



# Indexing Techniques

## **Multilevel Indexing :**

If the primary index does not fit in the memory, multilevel indexing is used. When the database increases its size the indices also get increased. A single-level index can be too big to store in the main memory. In multilevel indexing, the main data block breaks down into smaller blocks that can be stored in the main memory.

- 1. B+ Tree Indexing**
- 2. B- Tree Indexing**

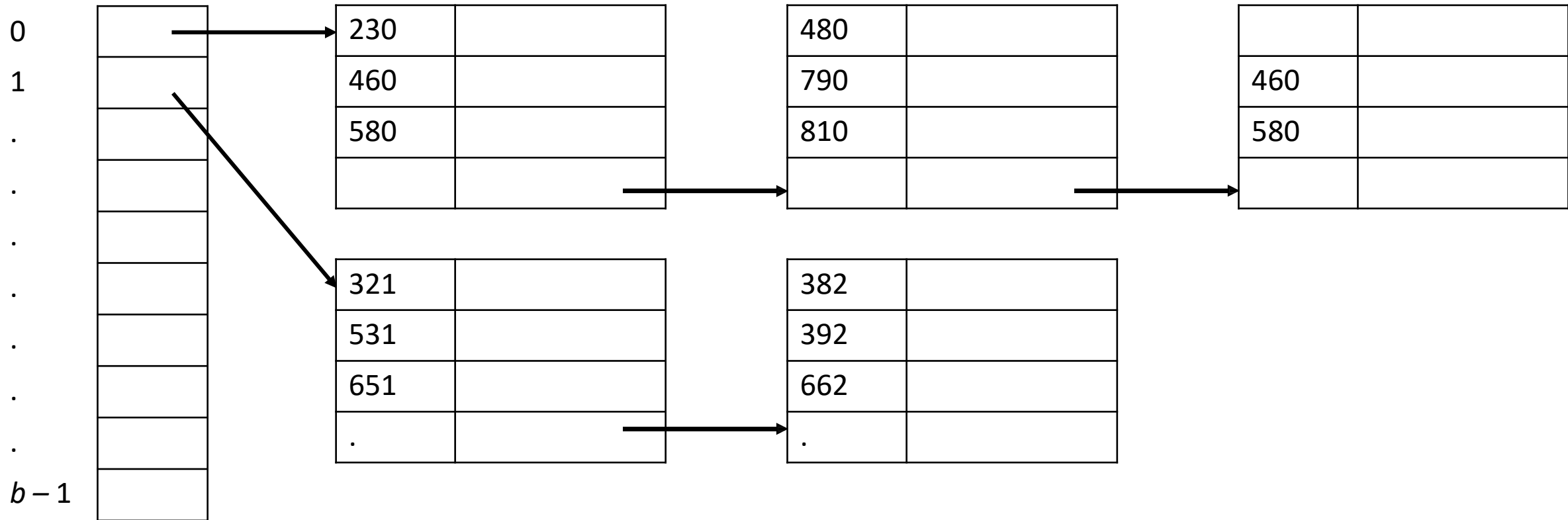
# Indexing Techniques

- Hashed Indexes
- Tree Indexing (B-Tree).
- Trie indexing – (Compact Trie).

# Indexing Techniques

- Hashed Indexes:

- Hash indexes are same as hash tables.
- Record of file are divided among buckets. A bucket is either one block or cluster of block.
- Hashing function maps a key into a bucket number. Buckets are numbered from 0 to  $b - 1$ .
- Translation of bucket number to disk block is done with the help of bucket directory.



Bucket Directory

# Indexing Techniques

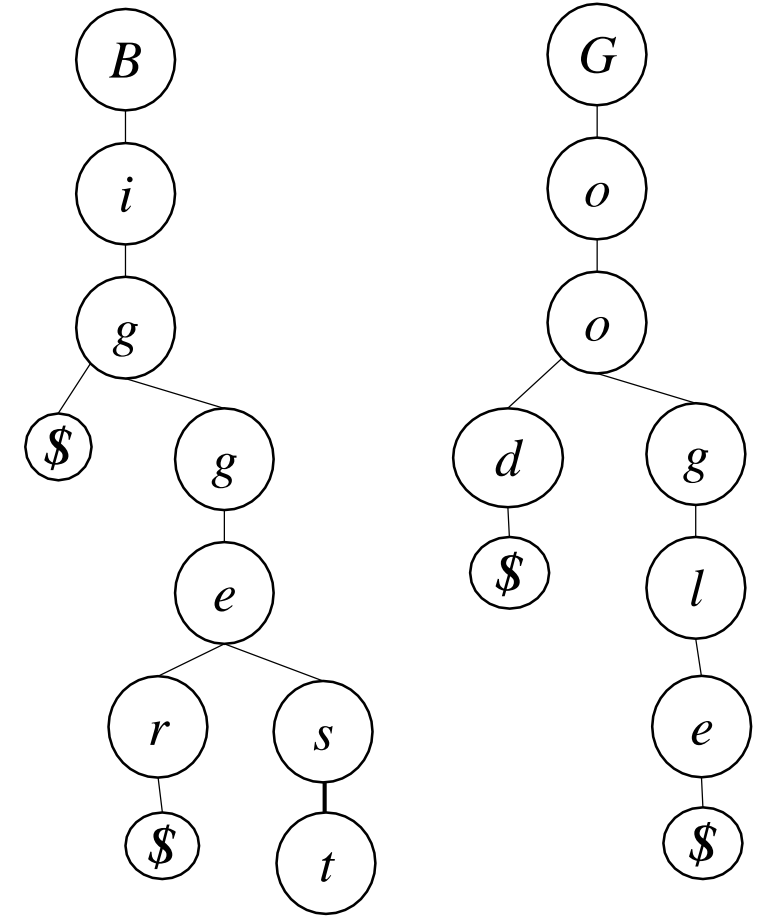
- Tree Indexing:
  - AVL tree is example of Tree Indexing.
  - Indexes can be stored in tree nodes.
  - Search, insertion and deletion can be easy.
  - An m- way search tree reduces the height of a search tree and thus decreases no. of disk access.

# Indexing Techniques

- Tree Indexing:
  - AVL tree is example of Tree Indexing.
  - Indexes can be stored in tree nodes.
  - Search, insertion and deletion can be easy.
  - An m- way search tree reduces the height of a search tree and thus decreases no. of disk access.

# Indexing Techniques

- Trie Indexing:
  - Trie is an efficient information retrieval data structure.
  - It is useful when key values are of varying size.
  - A trie is a tree of degree  $m \geq 2$ .
  - Useful for storing words as sequence of characters.
  - Each path from root to leaf node corresponds to a word.
  - Example : Big, Bigger, Biggest, Good, Google.





# Multiway Search Tree

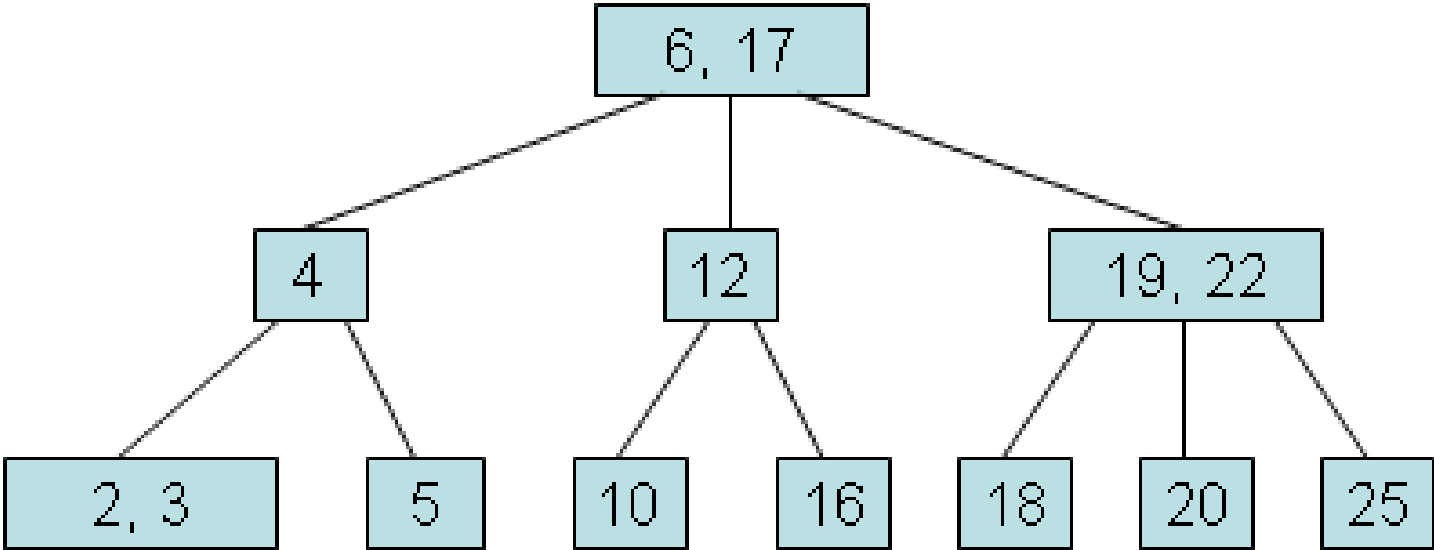
- Allows node to store multiple keys.
- Nodes have more than 2 children.
- All leafs are at same level.
- Each node have N children and N-1 search keys.
- Search, Insert and Delete operations can be performed on the Multiway search tree.
- **Example:** B-Tree, B+-Tree, Splay tree.

# B-Tree

- A B-tree is a tree data structure that keeps data sorted and allows searches, insertions, and deletions in logarithmic amortized time.
- Unlike self-balancing binary search trees, it is optimized for systems that read and write large blocks of data.
- It is most commonly used in database and file systems
- A self balanced search tree with multiple keys in every node and more than two children for every node.
- A B-tree of order  $M$  is a  $M$ -way search tree with following properties:
  - Root can have 1 to  $M-1$  keys.
  - All nodes (except root node) have between  $(M-1)/2$  and  $M-1$  keys.
  - All leaves are at same levels.
  - If node have  $k$  children, then it has  $k-1$  keys.
  - Keys are stored in sorted order.

# B-Tree

- 



# B-Tree

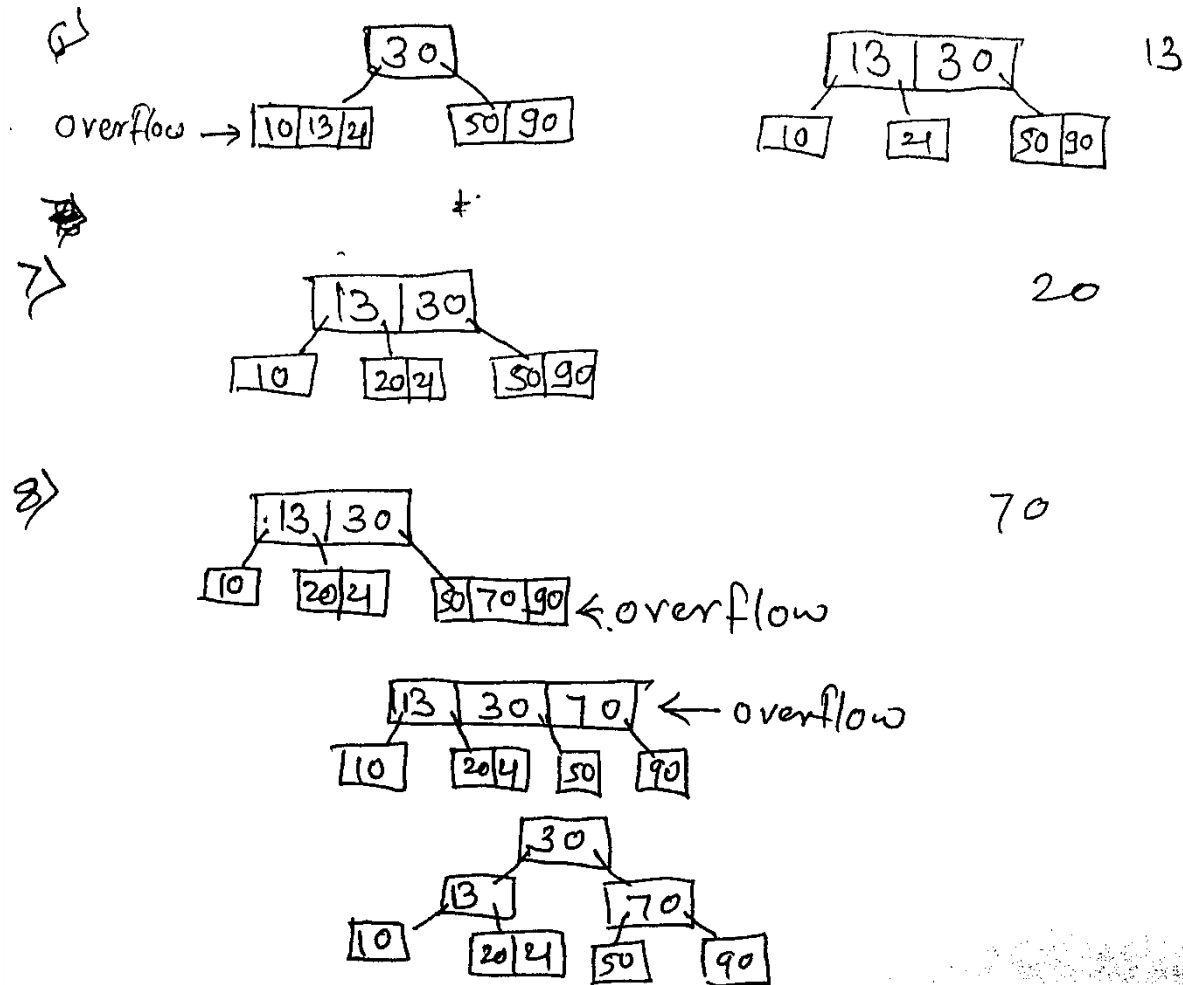
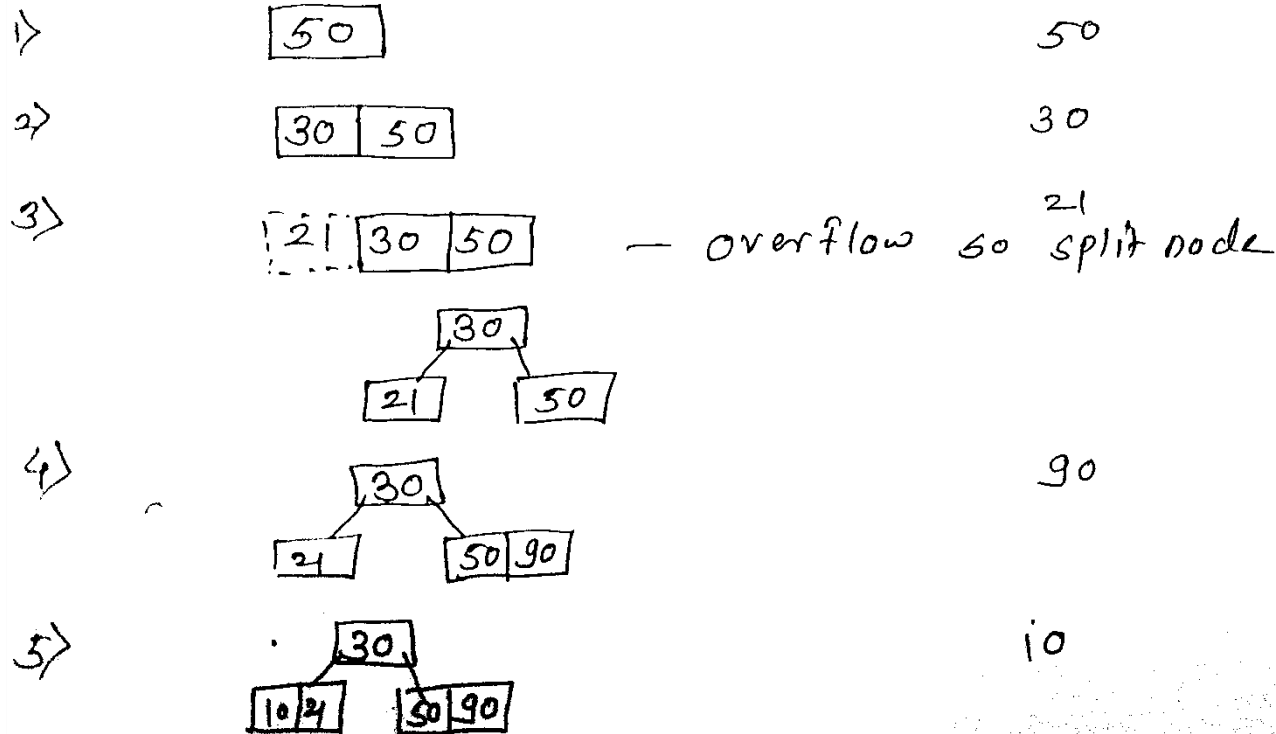
- Insertion of node in B-Tree:
  - **Step 1** - Check whether tree is Empty.
  - **Step 2** - If tree is Empty, then create a new node with new key value and insert it into the tree as a root node.
  - **Step 3** - If tree is Not Empty, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.
  - **Step 4** - If that leaf node has empty position, add the new key value to that leaf node in ascending order of key value within the node.
  - **Step 5** - If that leaf node is already full, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.
  - **Step 6** - If the spilting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

# B-Tree

- Construct a B-Tree of order 3 for following data: 50, 30, 21, 90, 10, 13, 20, 70, 25, 92, 80.

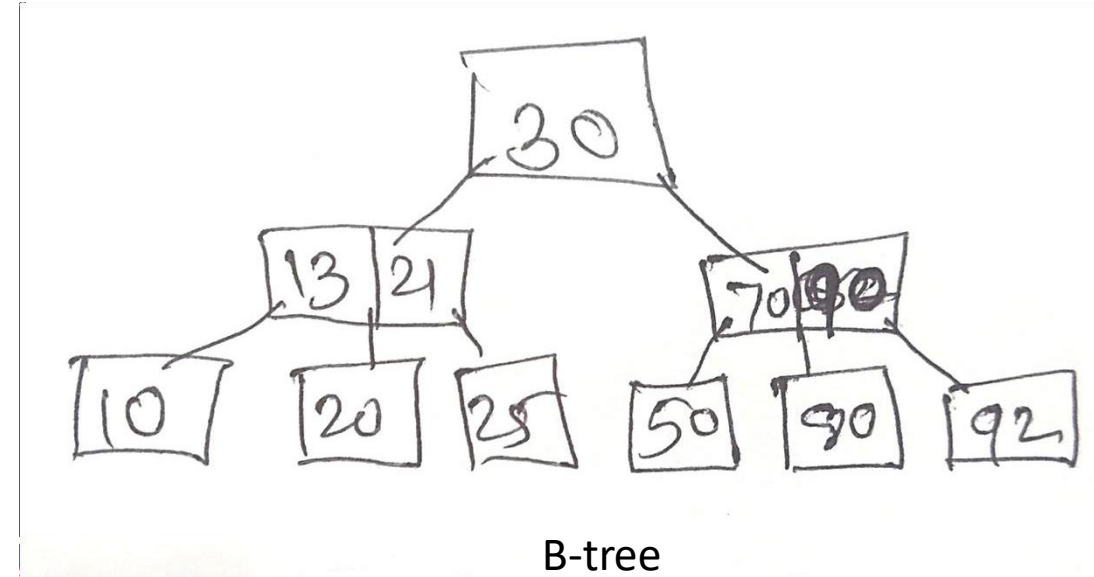
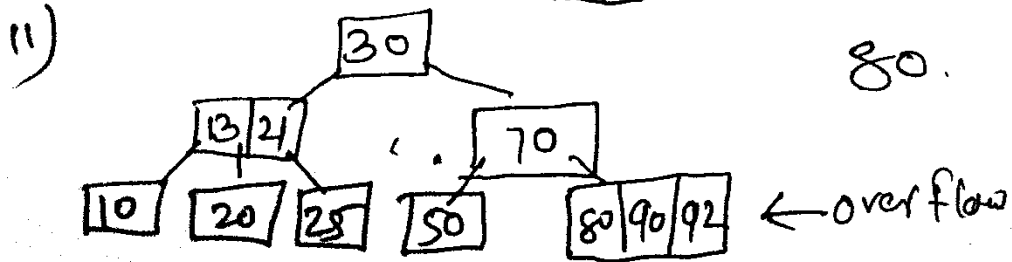
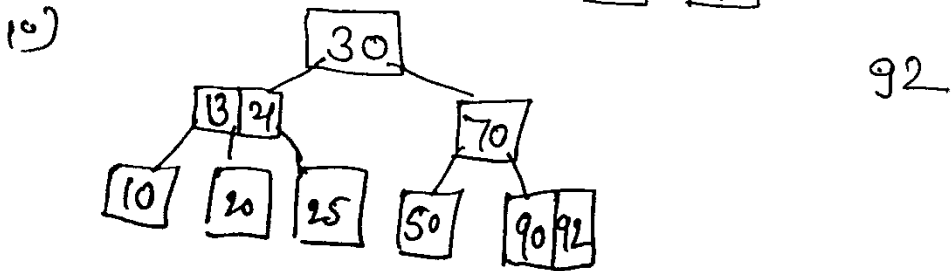
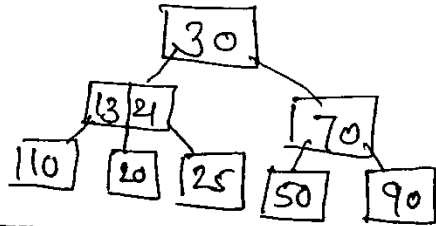
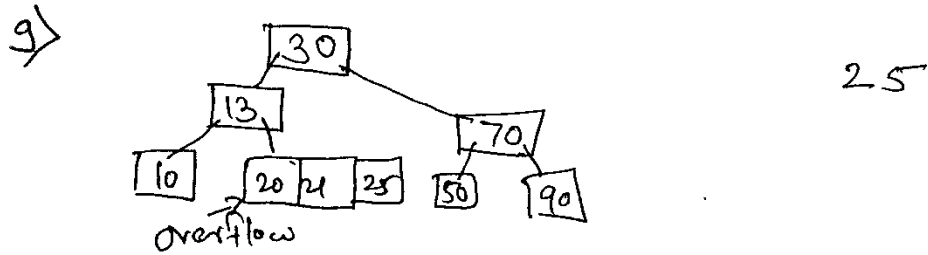
B-Tree - order 3.

So No. of keys for holding - 2  
 Overflow condition when keys in node is 3  
 No. of childrens - 3 for each node possible.



# B-Tree

- Construct a B-Tree of order 3 for following data: 50, 30, 21, 90, 10, 13, 20, 70, 25, 92, 80.

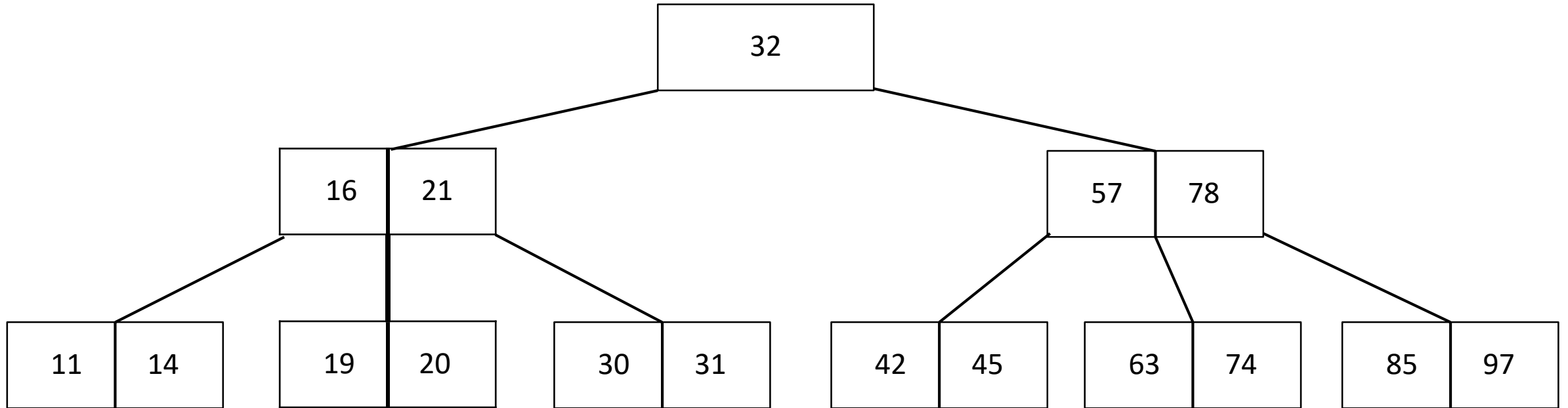


# B-Tree

- Construct a B-Tree of order 5 for following data: 78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 32, 30, 31.

# B-Tree

- Construct a B-Tree of order 5 for following data: 78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 32, 30, 31.

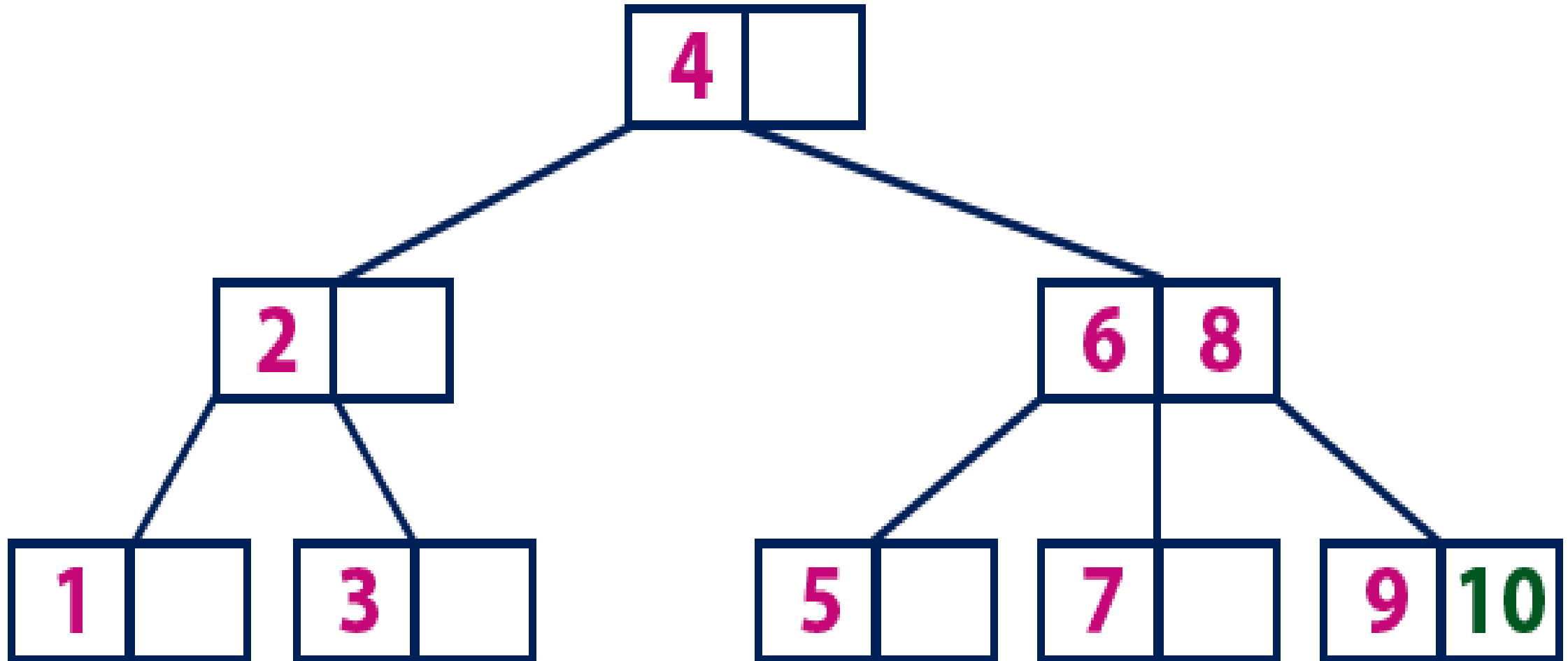


B – Tree.



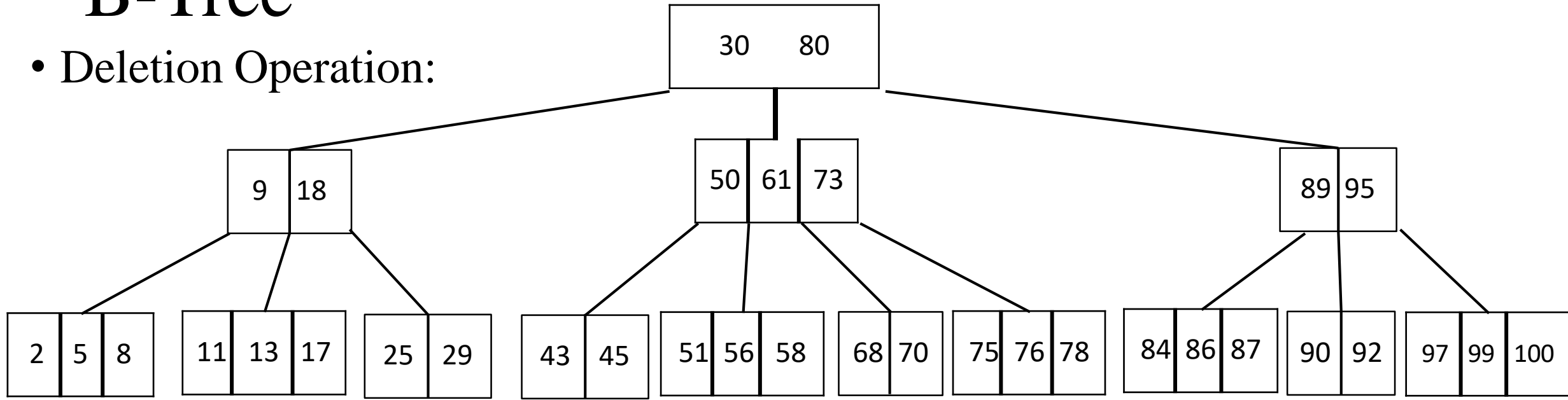
# B-Tree

- Construct a B-Tree of order 3 for following data: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10.



# B-Tree

- Deletion Operation:



- Facts of B-tree of degree  $m$ .

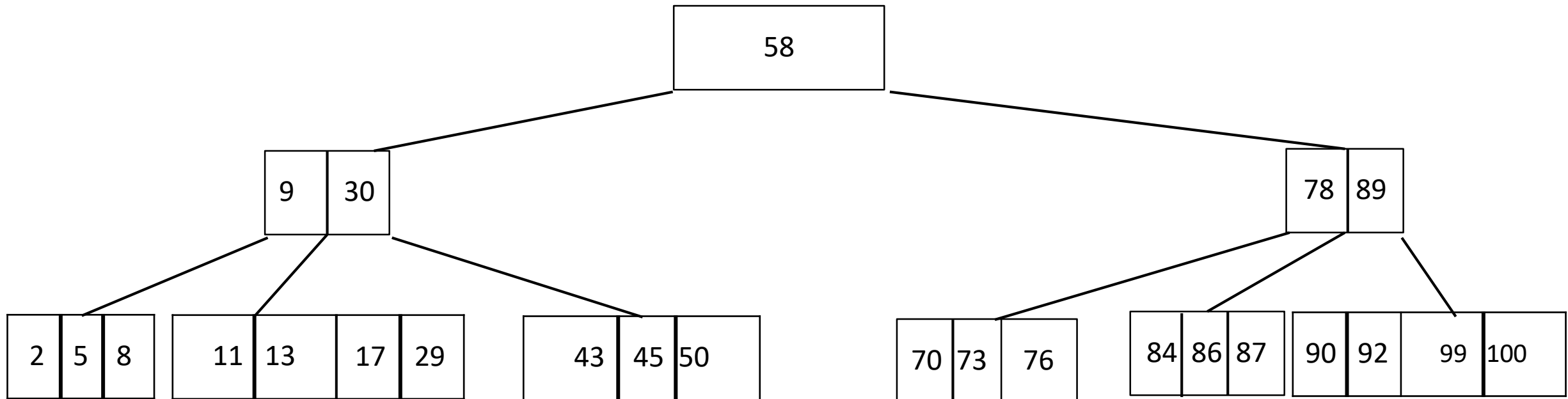
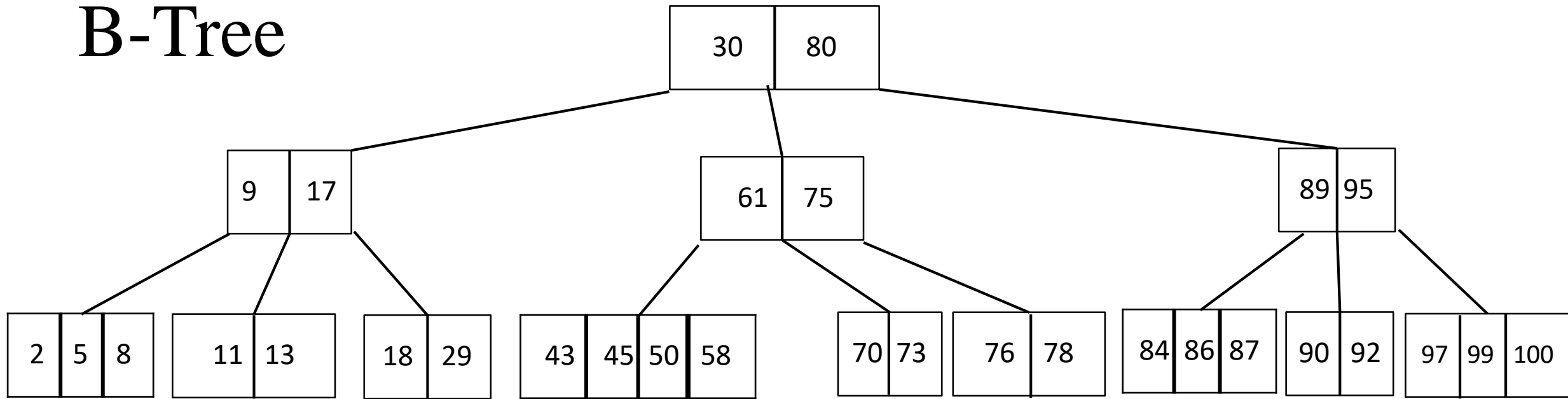
- A node can have a maximum of  $m$  children.
- A node can contain a maximum of  $m - 1$  keys.
- A node should have a minimum of  $\lceil \frac{m}{2} \rceil$  children.
- A node (except root node) should contain a minimum of  $\lceil \frac{m}{2} \rceil - 1$  keys.
- Inorder Predecessor, Inorder Successor

- If Key to be deleted is in leaf node.

- Node contains keys more than min. no. of keys
- Node contains min. no. of keys.
  - Borrow from immediate Left sibling.
  - Borrow from immediate Right sibling.
- Both Left & Right sibling have min. no. of keys.

- If key to be deleted is in internal node.

# B-Tree



# B<sup>+</sup>-Tree

- Insertion Operation:

**During insertion following properties of B<sup>+</sup> Tree must be followed:**

1. Each node except root can have a maximum of  $M$  children and at least  $\text{ceil}(M/2)$  children.
2. Each node can contain a maximum of  $M - 1$  keys and a minimum of  $\text{ceil}(M/2) - 1$  keys.
3. The root has at least two children and atleast one search key.
4. While insertion overflow of the node occurs when it contains more than  $M - 1$  search key values.
5. Here  $M$  is the order of B<sup>+</sup> tree.

# B<sup>+</sup>-Tree

## • Insertion Operation:

### Steps for insertion in B<sup>+</sup> Tree

Every element is inserted into a leaf node. So, go to the appropriate leaf node.

Insert the key into the leaf node in increasing order only if there is no overflow. If there is an overflow go ahead with the following steps mentioned below to deal with overflow while maintaining the B<sup>+</sup> Tree properties.

### Properties for insertion B<sup>+</sup> Tree

#### Case 1: Overflow in leaf node

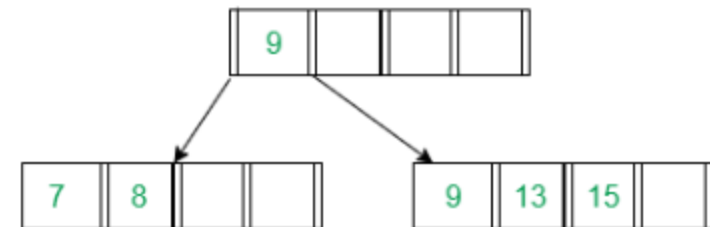
1. Split the leaf node into two nodes.
2. First node contains  $\text{ceil}((m-1)/2)$  values.
3. Second node contains the remaining values.
4. Copy the smallest search key value from second node to the parent node.(Right biased)

order=5



Insert 8

$\lceil(5-1)/2\rceil=2$



# B<sup>+</sup>-Tree

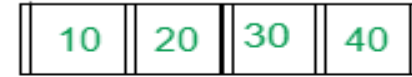
- Insertion Operation:

Below is the illustration of inserting 15 into B+ Tree of order of 5:

## Case 2: Overflow in non-leaf node

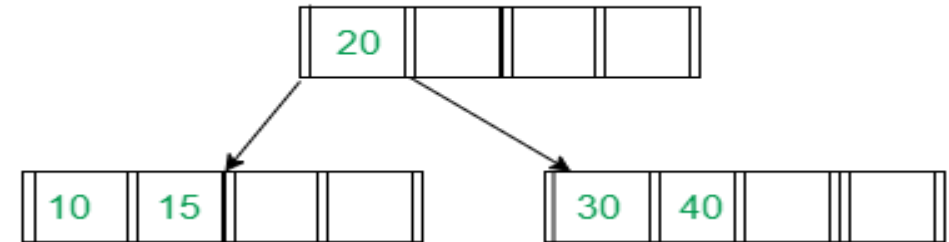
1. Split the non leaf node into two nodes.
2. First node contains  $\text{ceil}(m/2)-1$  values.
3. Move the smallest among remaining to the parent.
4. Second node contains the remaining keys

order=5



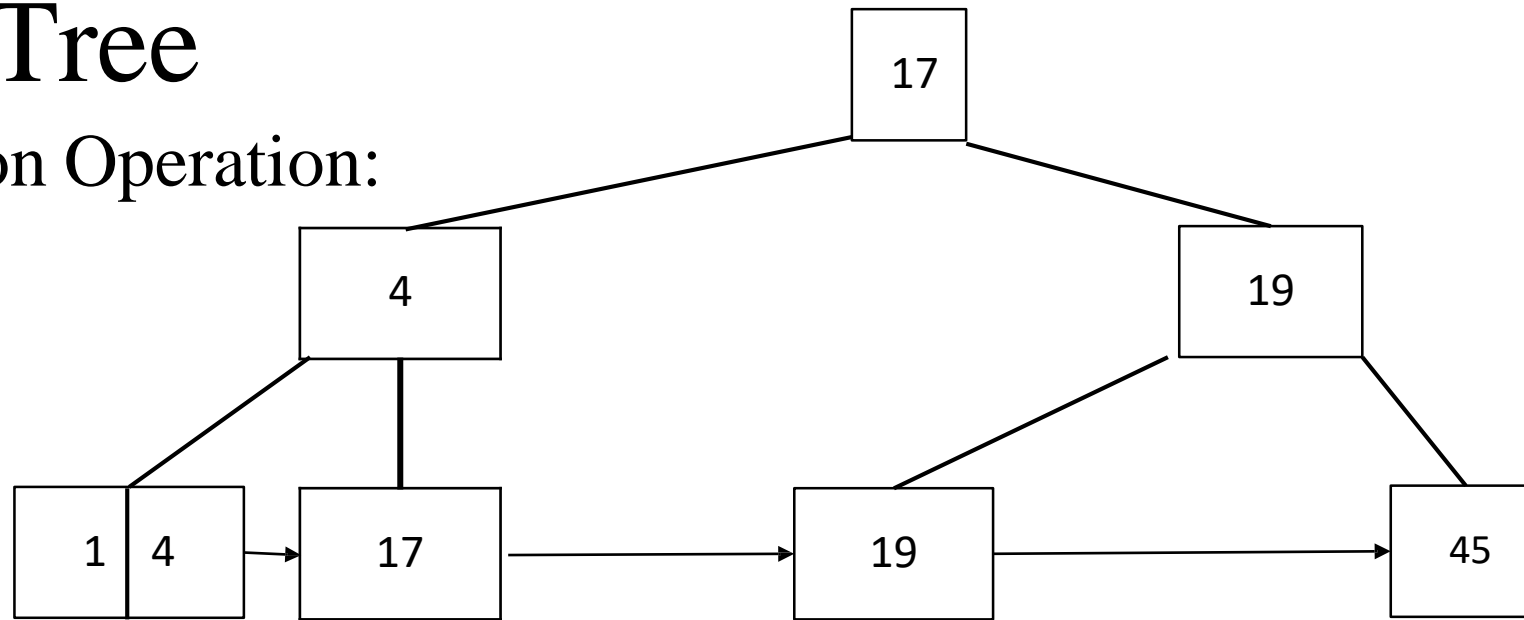
INSERT 15

$\lceil 5/2 \rceil - 1 = 2$



# B<sup>+</sup>-Tree

- Deletion Operation:



- Facts of B<sup>+</sup>-tree of degree  $m$ .

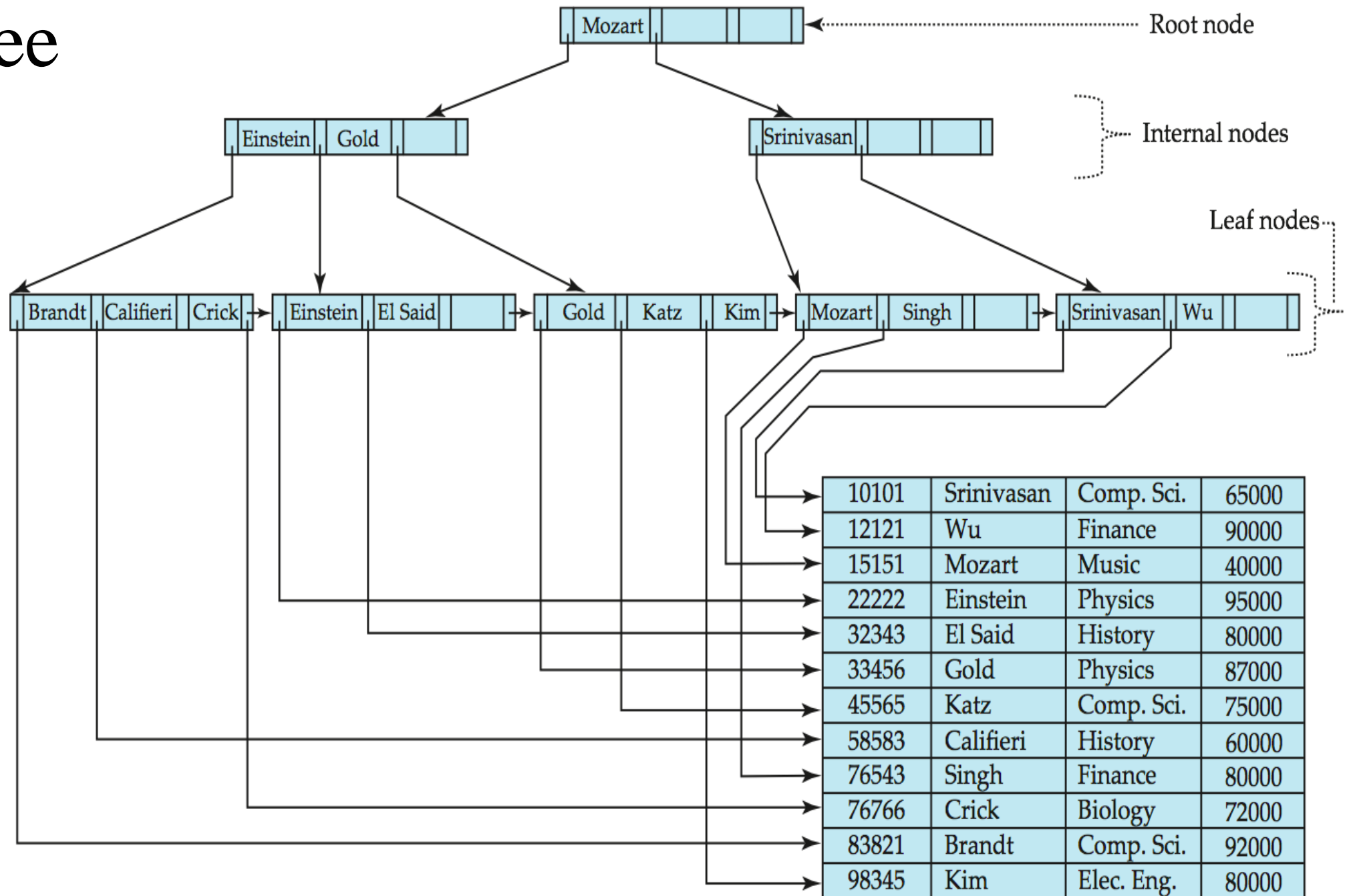
- A node can have a maximum of  $m$  children.
- A node can contain a maximum of  $m - 1$  keys.
- A node should have a minimum of  $\lceil \frac{m}{2} \rceil$  children.
- A node (except root node) should contain a minimum of  $\lceil \frac{m}{2} \rceil - 1$  keys.

# B<sup>+</sup>-Tree

- A B<sup>+</sup>-tree is a variation of B-tree data structure.
- Data pointers in B<sup>+</sup>-tree are stored only at leaf nodes of tree.
- B<sup>+</sup>-tree structure of a leaf node differs from the structure of internal nodes.
- Advantage of B<sup>+</sup>-tree index files:
  - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- Disadvantage of B<sup>+</sup>-trees:
  - Extra insertion and deletion overhead, space overhead.

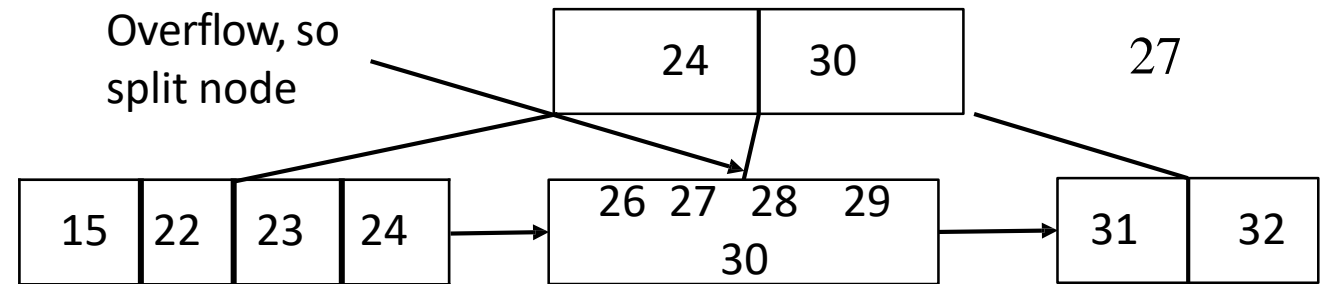
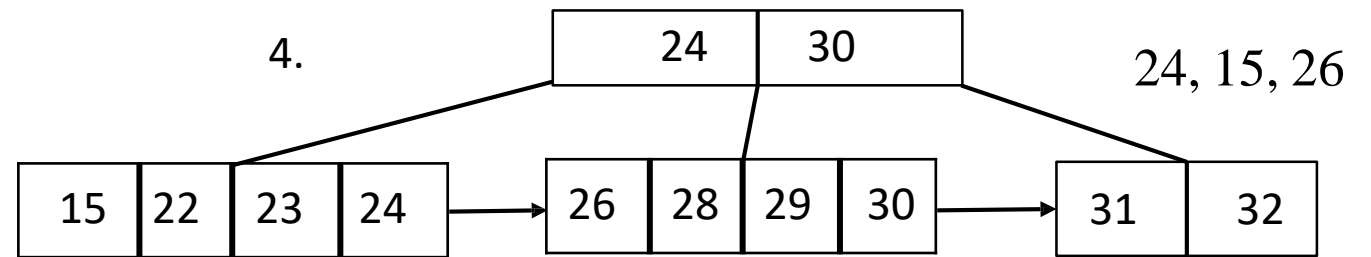
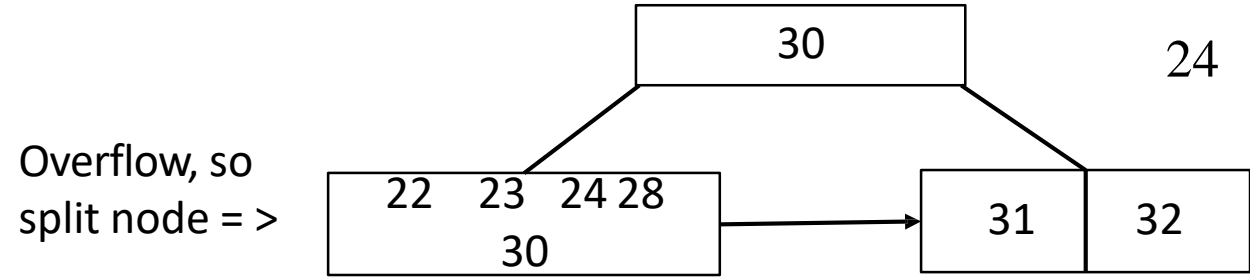
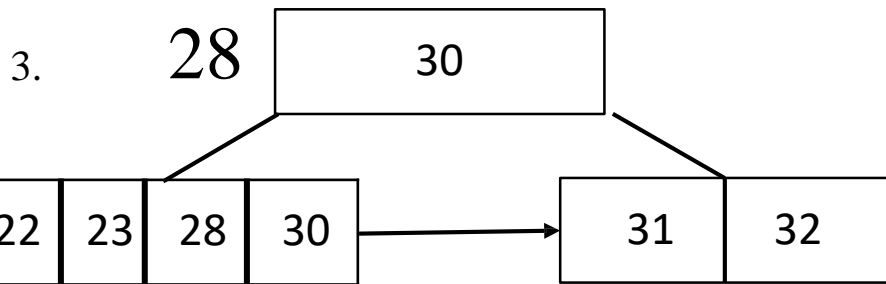
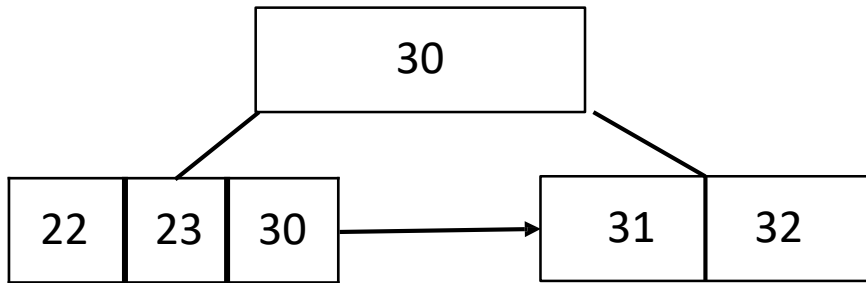
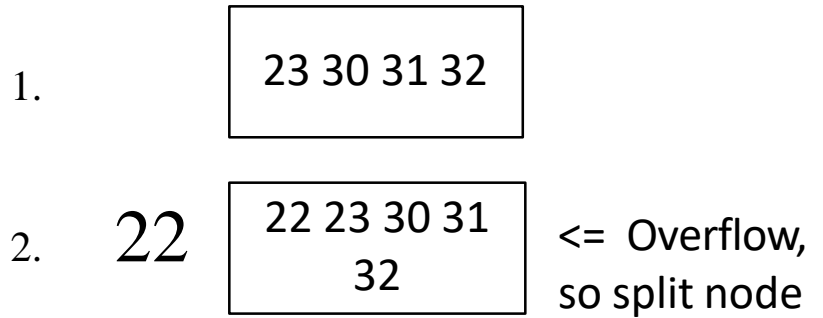


# B<sup>+</sup>-Tree



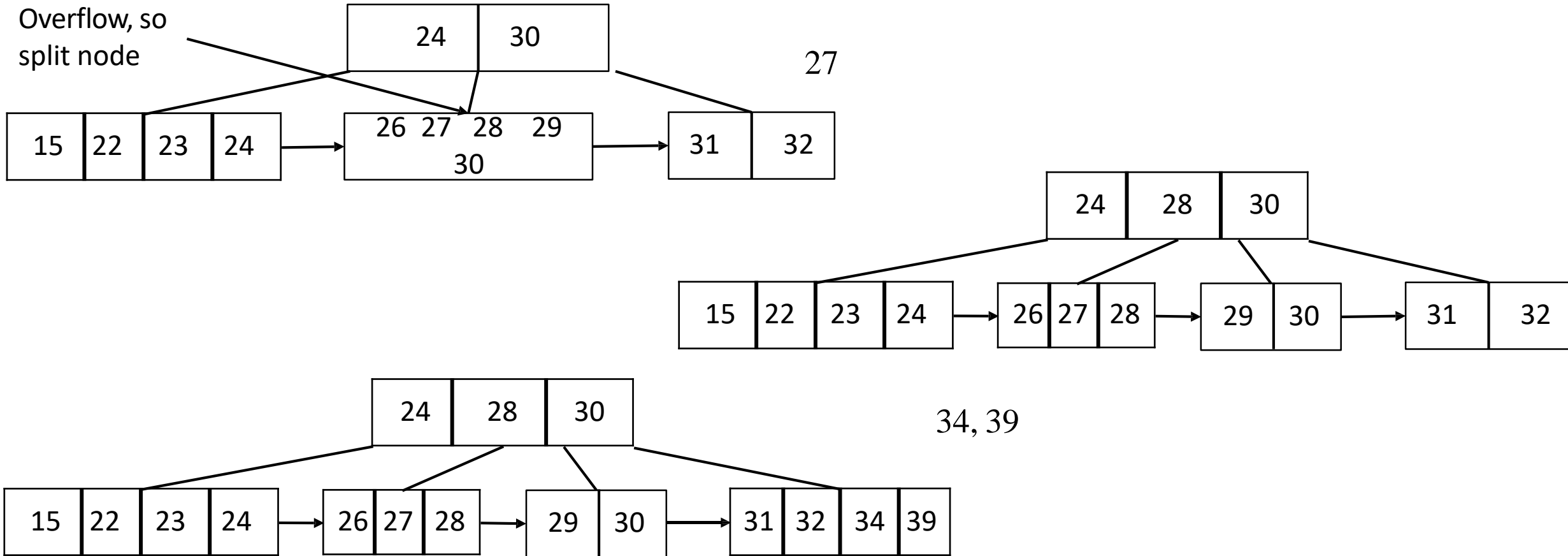
# B+Tree

- Construct a B<sup>+</sup>-Tree of order 5 for following data: 30, 31, 23, 32, 22, 28, 24, 29, 15, 26, 27, 34, 39, 36.



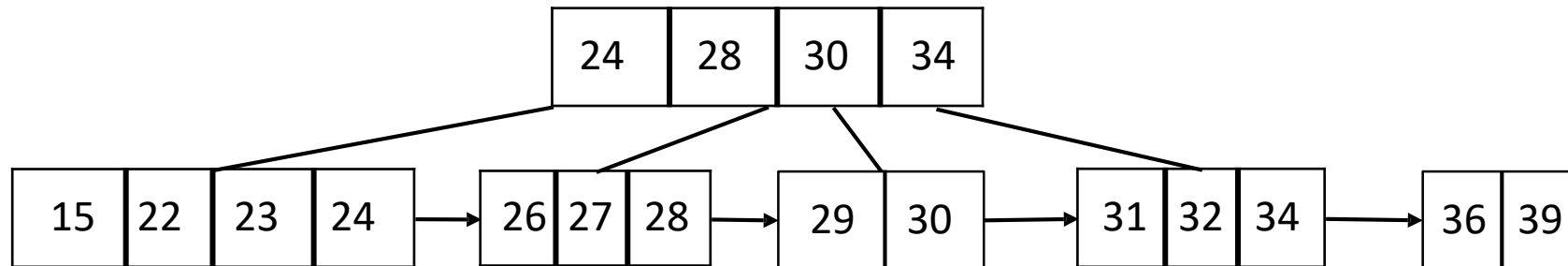
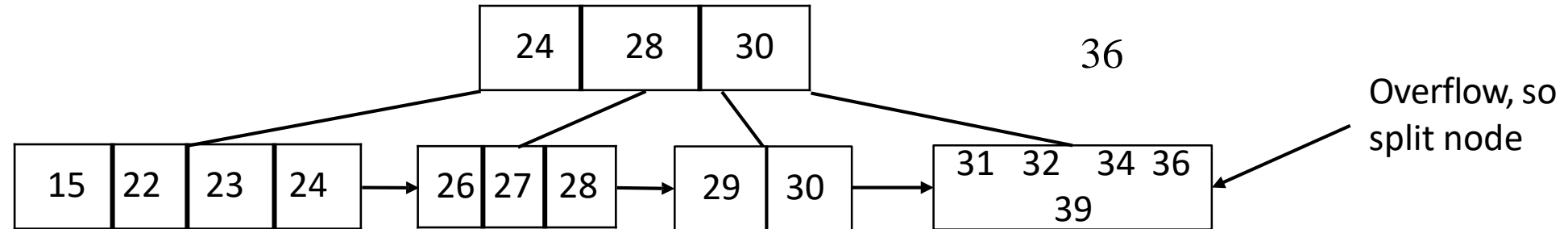
# B-Tree

- Construct a B<sup>+</sup>-Tree of order 5 for following data: 30, 31, 23, 32, 22, 28, 24, 29, 15, 26, 27, 34, 39, 36.



# B+Tree

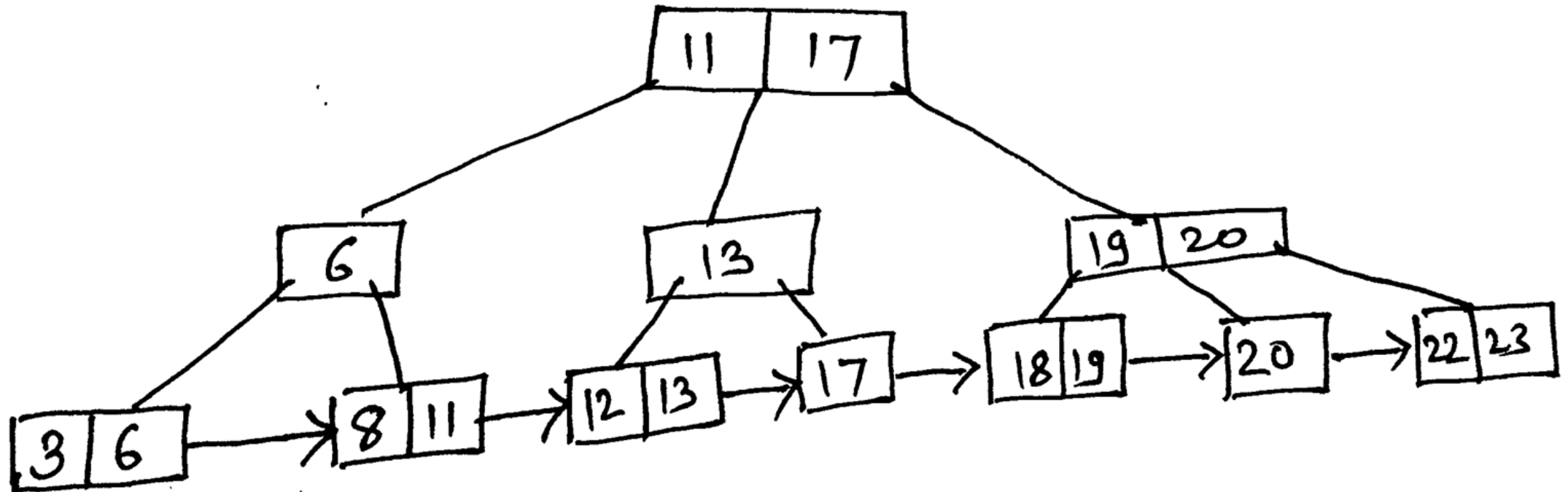
- Construct a B<sup>+</sup>-Tree of order 5 for following data: 30, 31, 23, 32, 22, 28, 24, 29, 15, 26, 27, 34, 39, 36.



B<sup>+</sup> - Tree

# B+Tree

- Construct a B<sup>+</sup>-Tree of order 3 for following data: 6, 19, 17, 11, 3, 12, 8, 20, 22, 23, 13, 18.



B<sup>+</sup> - Tree

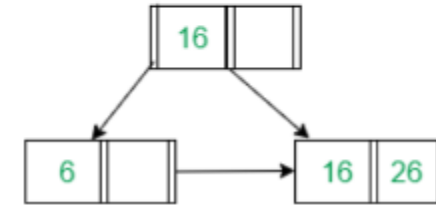
# B+Tree

- Problem: Insert the following key values 6, 16, 26, 36, 46 on a B+ tree with order = 3.

Insert 6, 16, 26, 36, 46 on a B+ tree with order =3

Insert 26

causes overflow 6, 16, 26

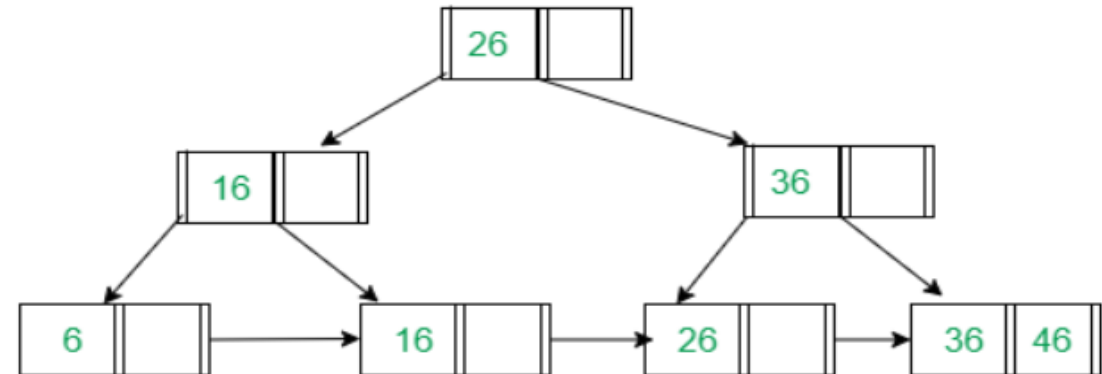
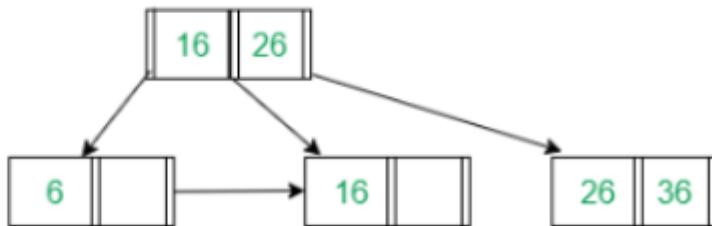


Insert 6, 16



Insert 36

16, 26, 36



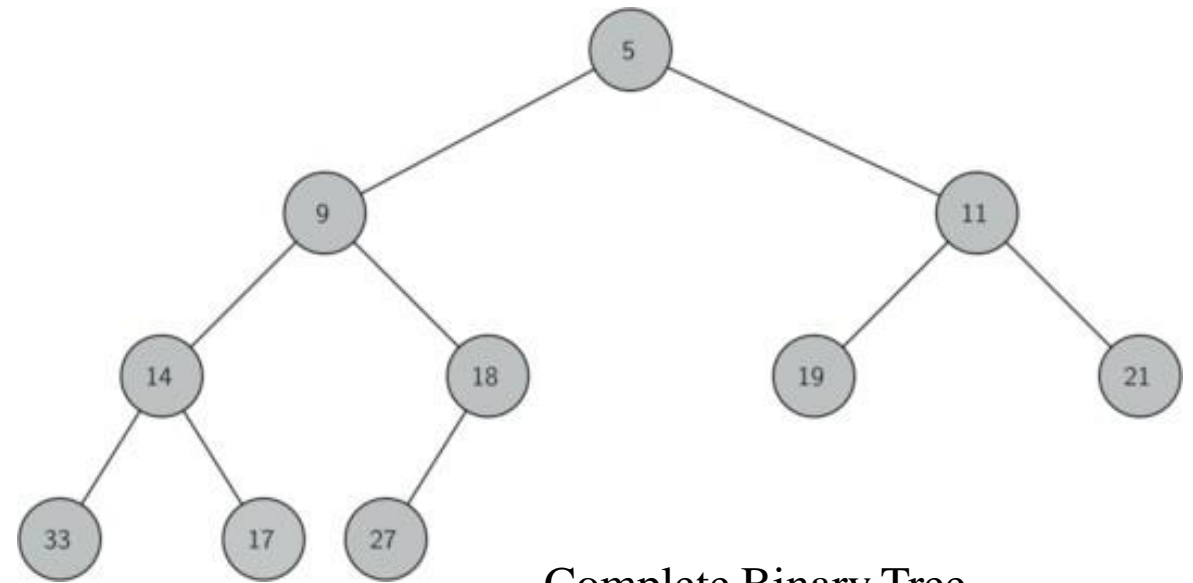
B<sup>+</sup> - Tree

# B-Tree vs B+ Tree

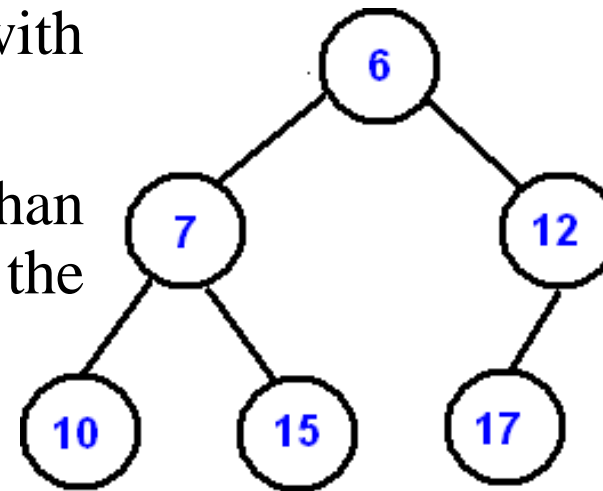
SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

# Binary Heap

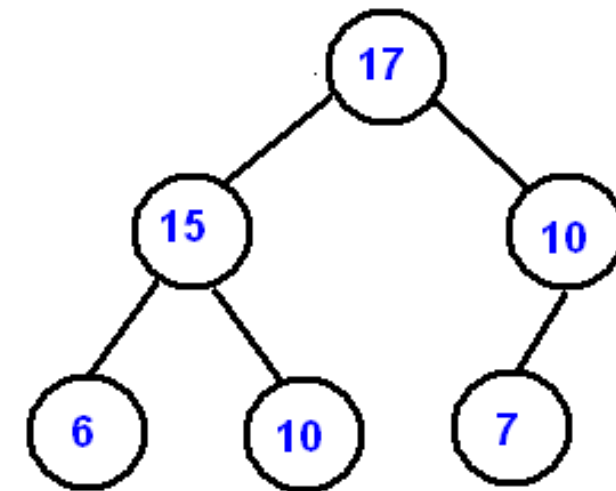
- A Heap is a complete binary tree.
- **In complete binary tree elements are filled level by level from left to right.**
- Bottom level may not be completely filled.
- A Binary Heap is either Min or Max Heap.
- Min Heap: Value of each node is **greater** than or equal to the value of its parent, with the minimum-value element at the root.
- Max Heap: Value of each node is **less** than or equal to the value of its parent, with the maximum-value element at the root.



Complete Binary Tree



Min Heap



Max Heap



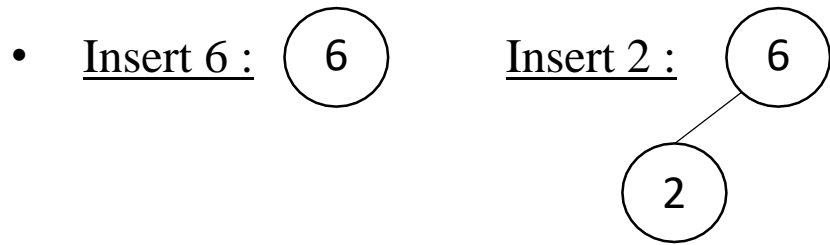
# Binary Heap

- Insertion of node in Min-Heap or Max-Heap tree.
  - Add element to bottom level of heap at the *most left*.
  - Compare added element with its parent if they are in the correct order , stop.
  - If not, swap the element with its parent and return to previous step.

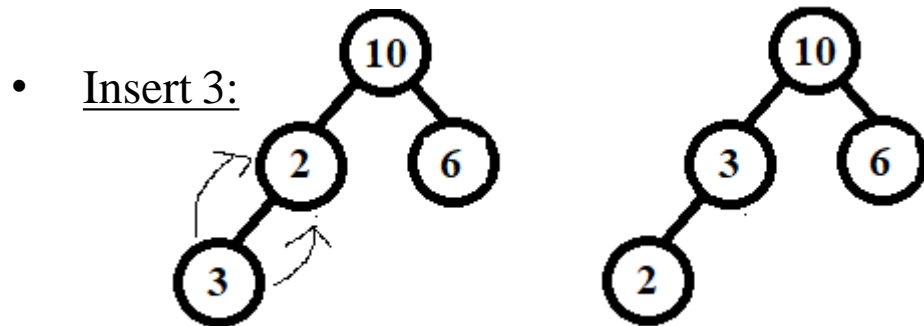
Note: Min-Heap or Max-Heap tree is complete Binary tree. So elements are added at leaf from left to right.

# Binary Heap

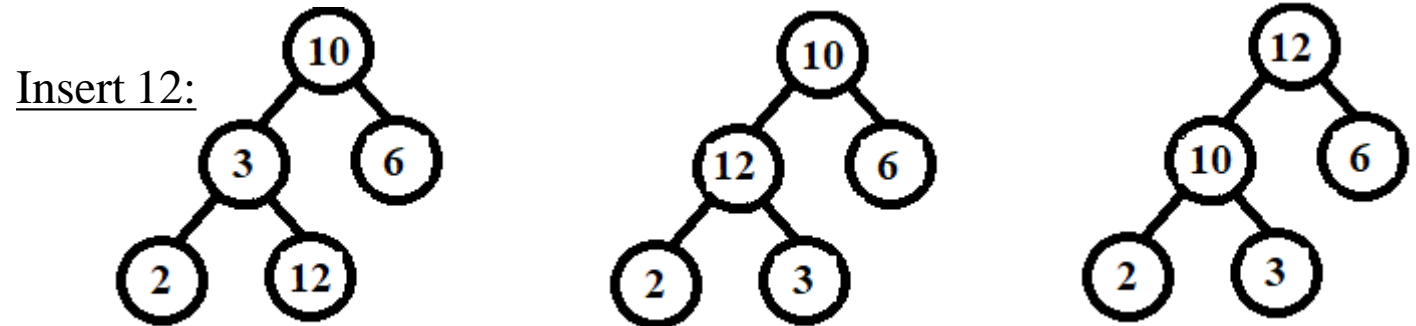
- Create Max Heap with following elements : 6, 2, 10, 3, 12, 60, 7, 200, 8.



Not following Max Heap Property: Parent  $\geq$  Child

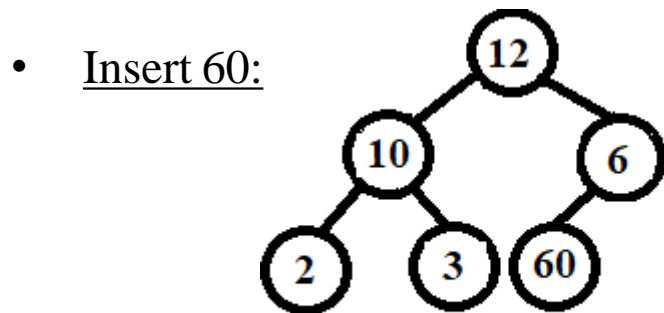


Not following Max Heap Property:  
Parent  $\geq$  Child

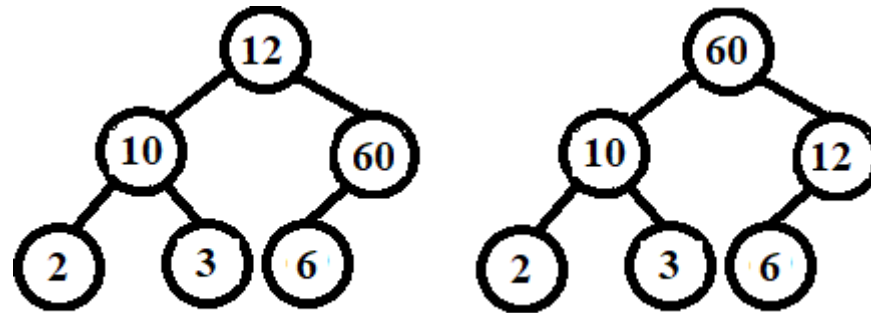


Not following Max Heap Property  
Parent  $\geq$  Child

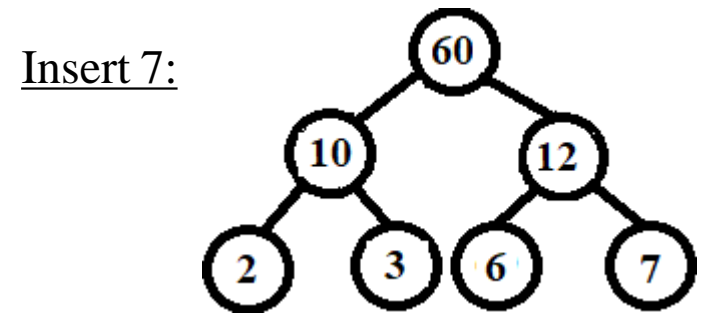
Not following Max Heap Property:  
Parent  $\geq$  Child



Not following Max Heap Property:  
Parent  $\geq$  Child



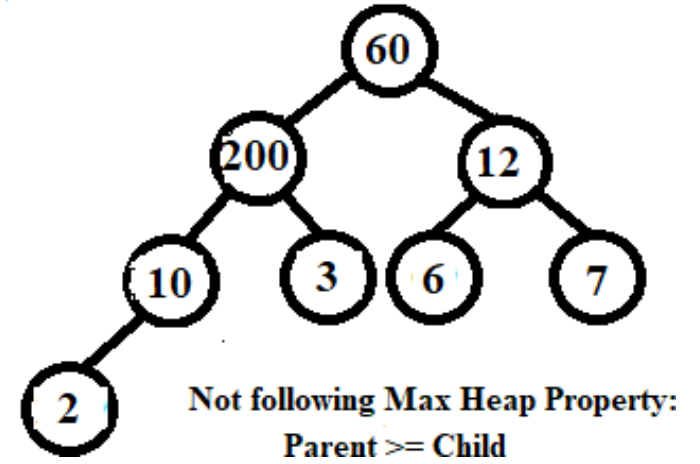
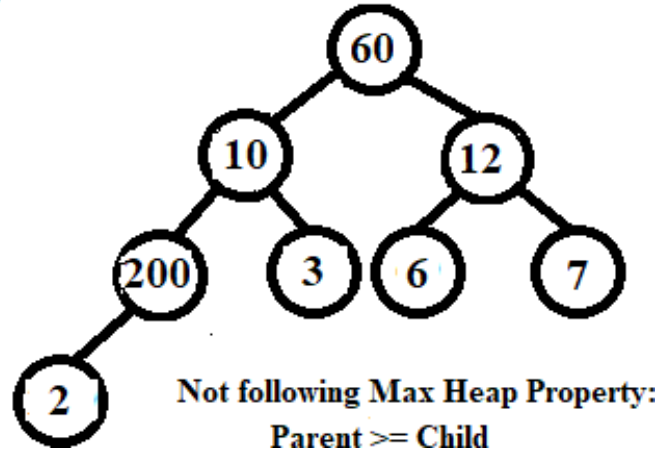
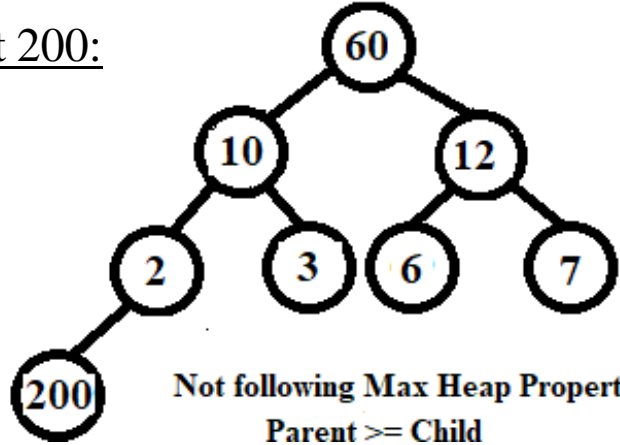
Not following Max Heap Property:  
Parent  $\geq$  Child



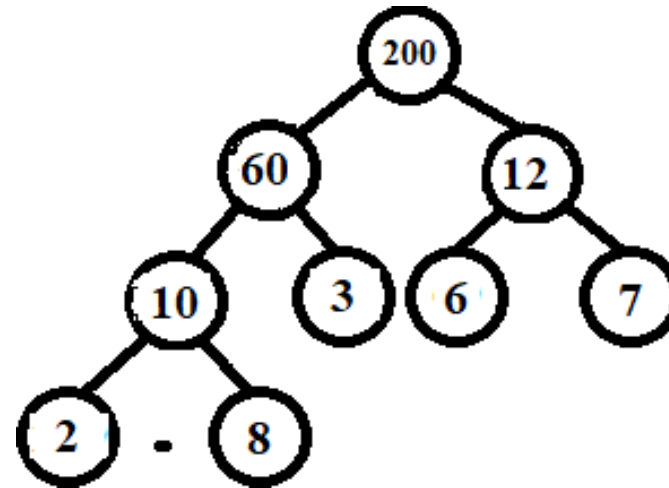
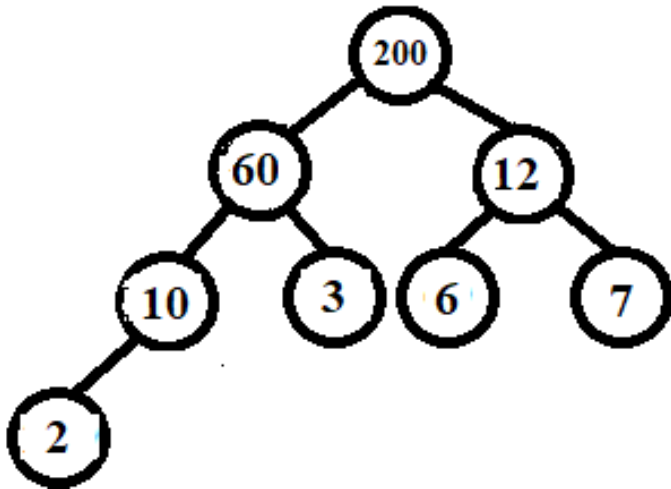
# Binary Heap

- Create Max Heap with following elements : 6, 2, 10, 3, 12, 60, 7, 200, 8.

- Insert 200:




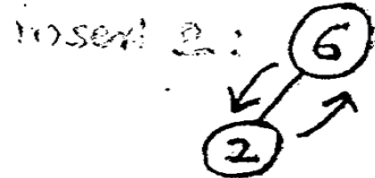
- Insert 8:



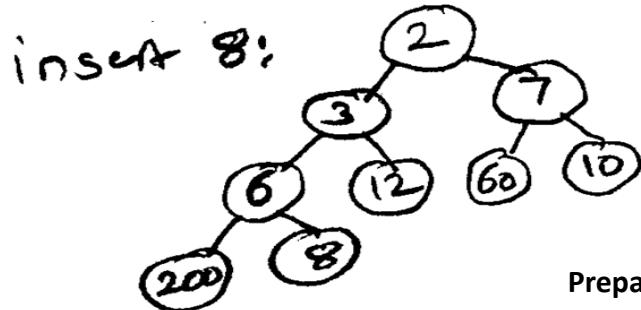
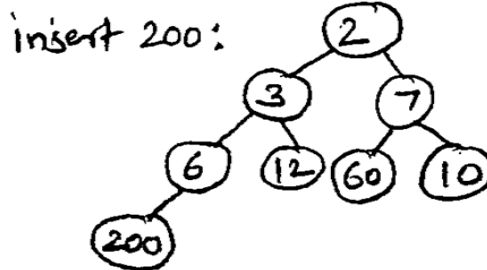
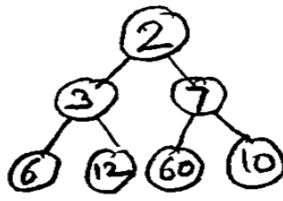
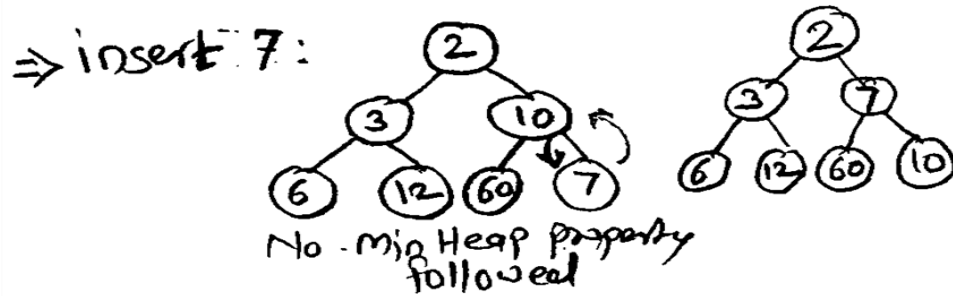
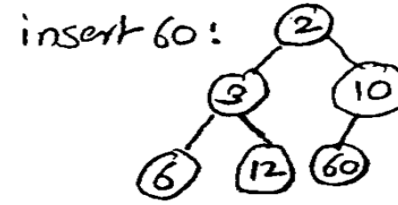
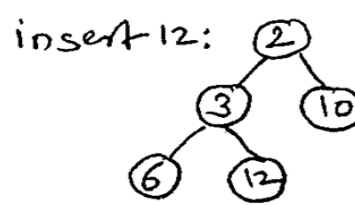
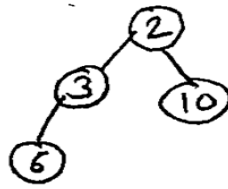
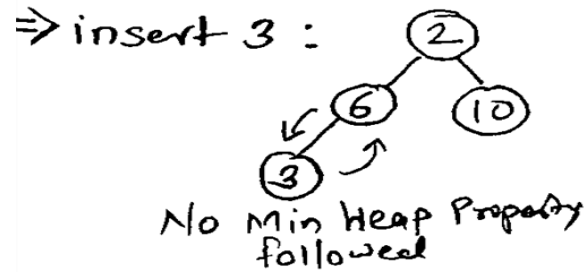
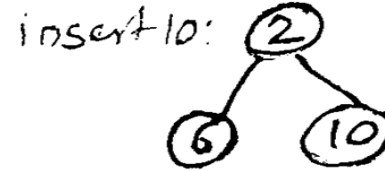
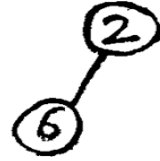
# Binary Heap

- Create Min Heap with following elements : 6, 2, 10, 3, 12, 60, 7, 200, 8.

⇒ insert 6 : 



No Min Heap Property followed



# Binary Heap

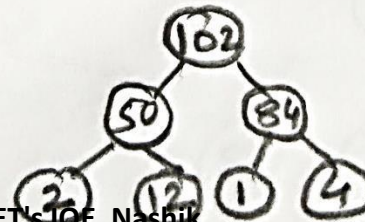
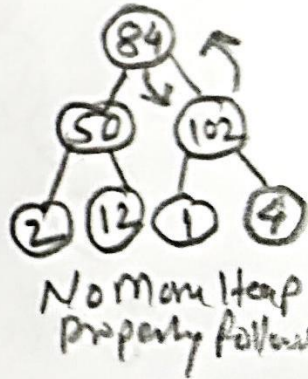
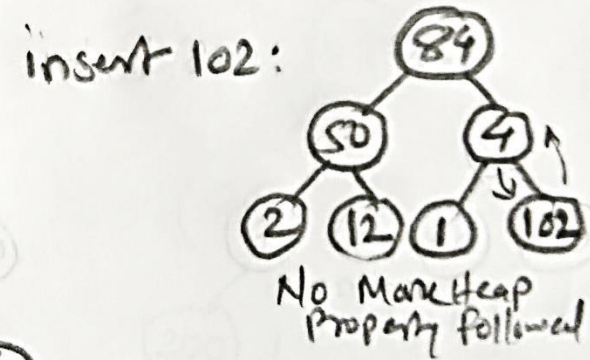
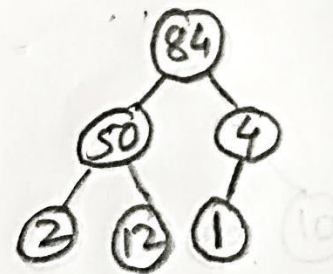
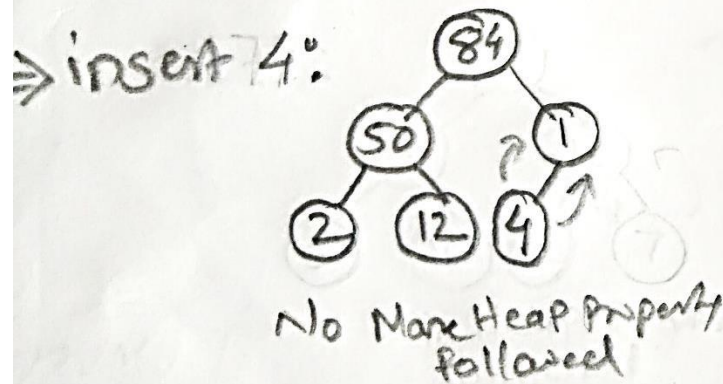
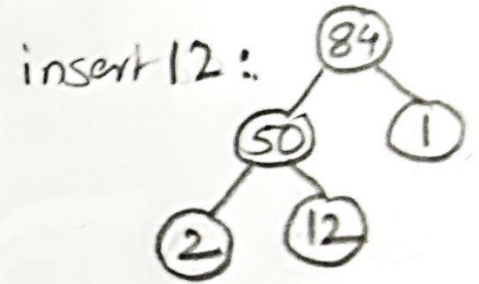
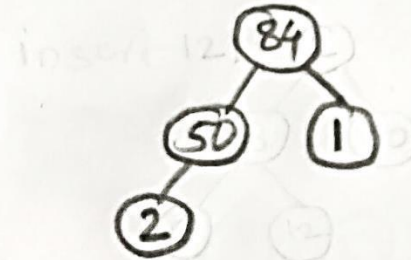
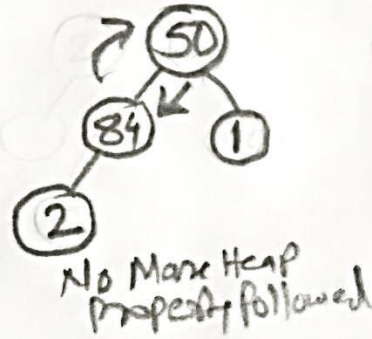
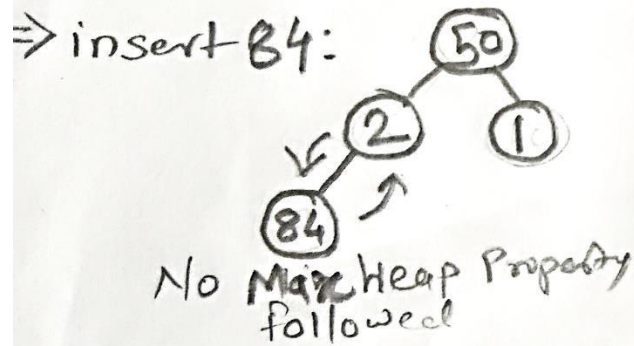
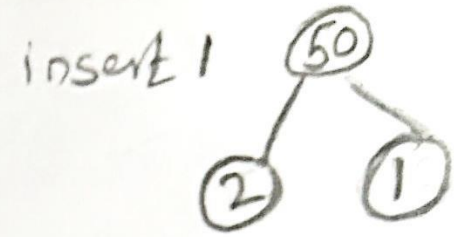
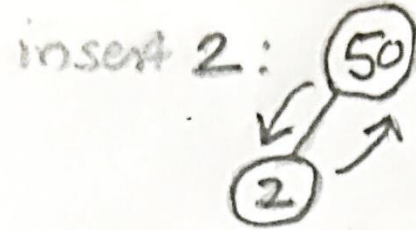
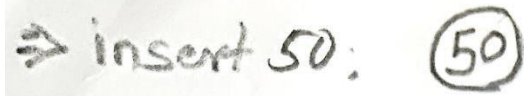
- Deletion of node in Min-Heap or Max-Heap tree.
  - Replace the value of root with the value of rightmost leaf and remove leaf from tree.
  - Now check whether the tree follows the Heap property.
  - If not, swap the element with its parent and return to previous step. For Max-Heap if parent is less than both the children then replace the root with larger child.

# Heap Sort

- From the unsorted list create a Heap tree (Min or Max).
  - Once Heap is created, now select the first element(root) and add it to array.
  - Remove first element and rearrange the Heap and repeat above step.
- Heap sort algorithm:
  1. Create binary tree from the given elements.
  2. Convert binary tree to Min Heap or Max Heap.
  3. Delete the root element from the Heap.
  4. Add deleted element to array.
  5. Rearrange the Heap and repeat the step 3 to 5 till Heap is empty.
  6. If heap is empty , display sorted list from array and stop.

# Heap Sort

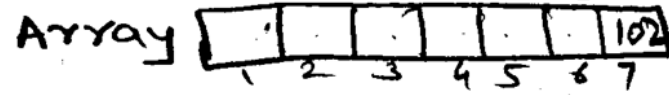
- Sort numbers in ascending order using heapsort : 50, 2, 1, 84, 12, 4, 102.
- Create Max-Heap:



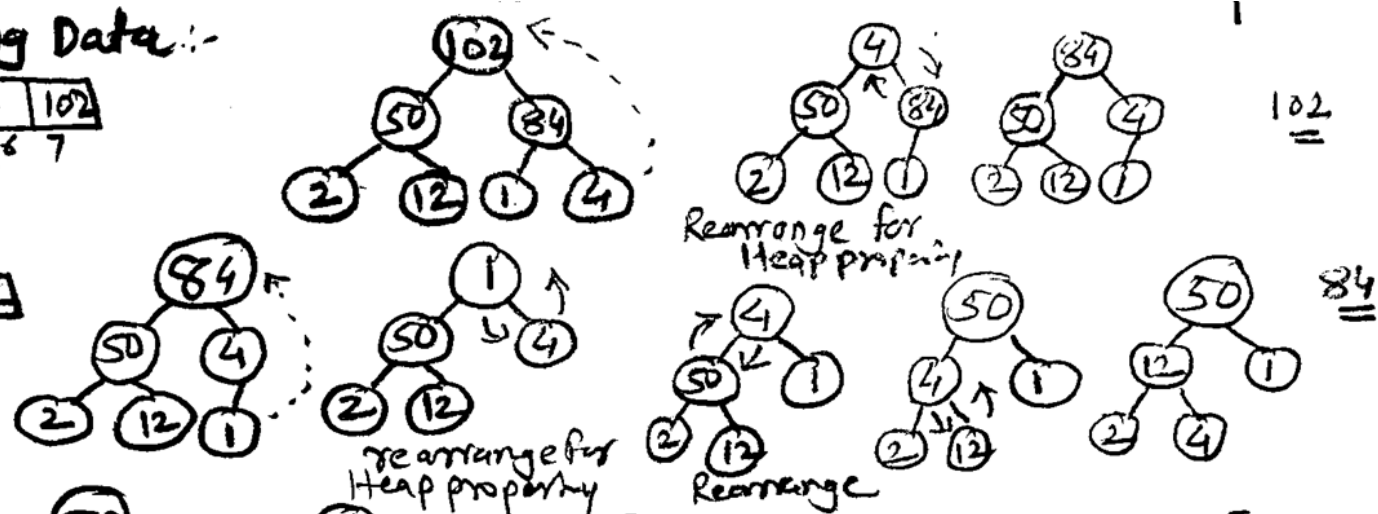
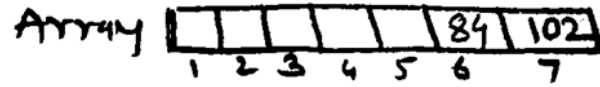
# Heap Sort

- Sort numbers in ascending order using heapsort : 50, 2, 1, 84, 12, 4, 102.
- Sorting Data:

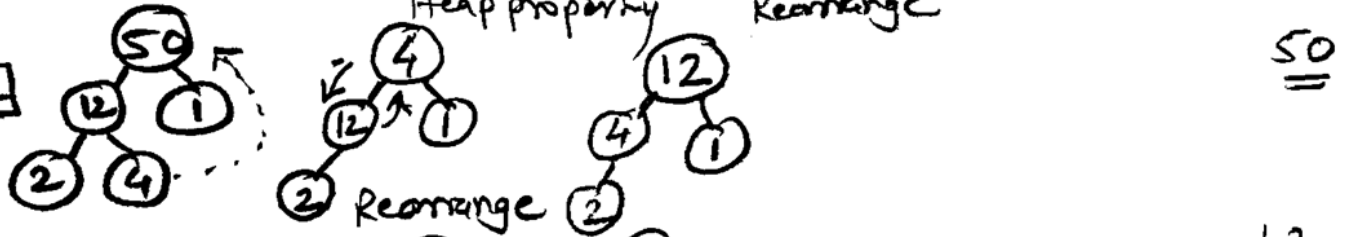
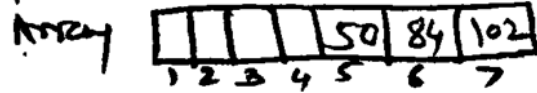
Sorting Data:-



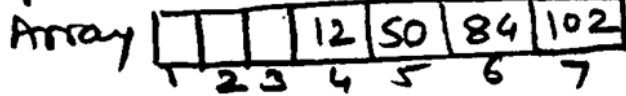
iteration 2:



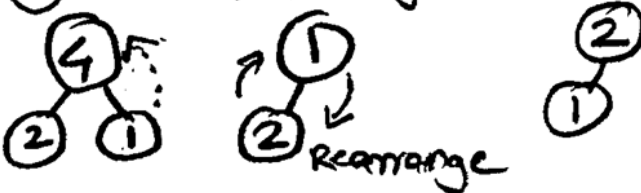
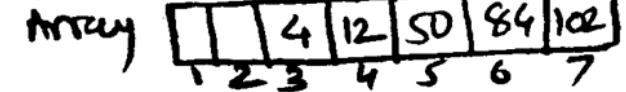
iteration 3:



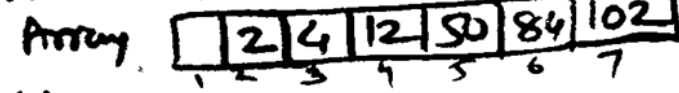
iteration 4:



iteration 5:



iteration 6:



iteration 7:





Sort the numbers using heapsort: 12, 45, 21, 76, 83, 97, 82, 54.

Sort the numbers using heapsort: 25, 4, 70, 1, 60, 10, 85, 11.

# Questions

- Insert keys to a 5-way B-tree: 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56.
- Construct B<sup>+</sup> tree of order 3 : 1, 42, 28, 21, 31, 10, 17, 7, 31, 25, 20, 18.
- Explain Heap tree, Max-Heap & Min-Heap.
- Write Algorithm & Pseudo code for Inserting Element in Min Heap and Max Heap.
- Create Min Heap & Max Heap for given data: 10, 20, 15, 12, 25, 30, 14, 2, 5, 4.
- Delete root node of above Heap tree and Re-arrange Heap tree.
- Write algorithm to sort the numbers in ascending order using heapsort.
- Sort the numbers using heapsort: 18, 13, 12, 22, 15, 24, 10, 16, 19, 14, 30.

# References

<https://www.javatpoint.com/indexing-in-dbms>

# THANK YOU!!!

**My Blog :** <https://anandgharu.wordpress.com/>

**Email :** gharu.anand@gmail.com