3.1 Multiway Search Tree

BTree

Multiway Search Tree

- Generalised versions of binary trees where each node contains multiple elements

- A Multiway Search Tree of **order n** is a general tree in which
  - each node has **n or fewer subtrees** and
  - contains no of keys as **one less than no of subtrees**

- i.e. In an **m-Way tree of order n**,
  - each node contains a **maximum of n – 1 elements and n children.**

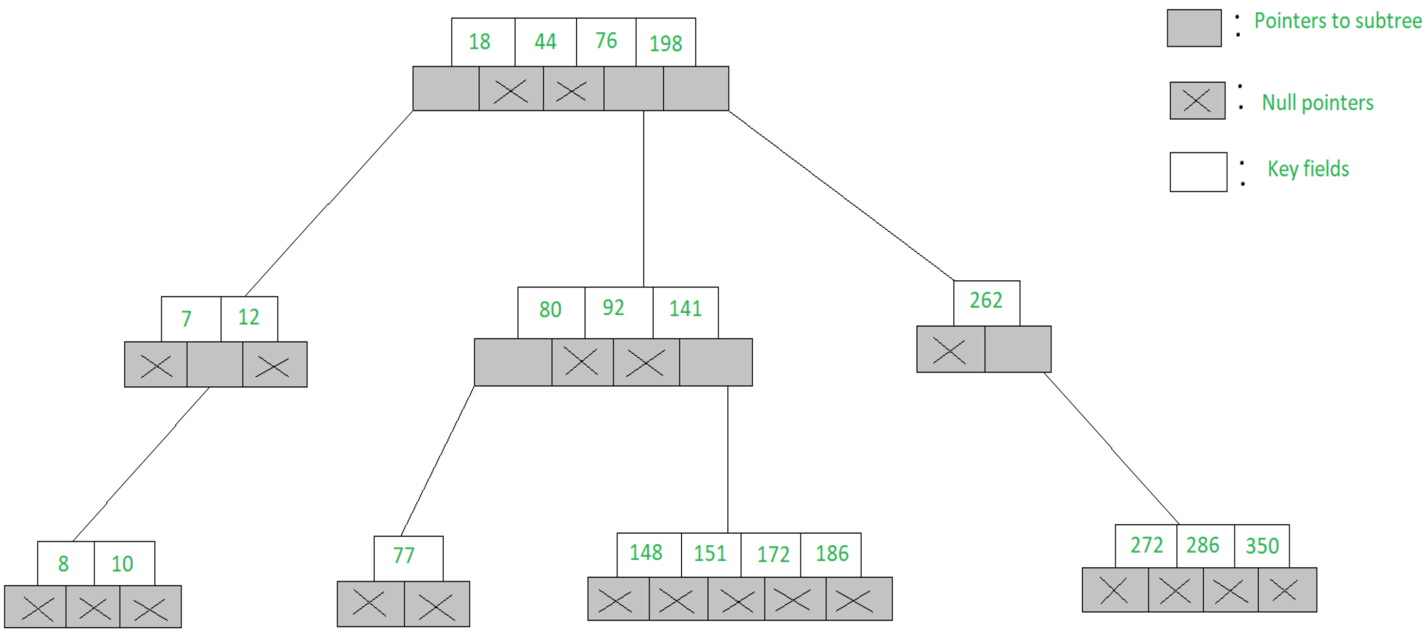
- If the Node has **4 subtrees, it contains 3 keys**

Multiway Search Tree

- The goal of m-Way search tree of **height h** calls for **O(h) no. of accesses for an insert/delete/retrieval operation.**

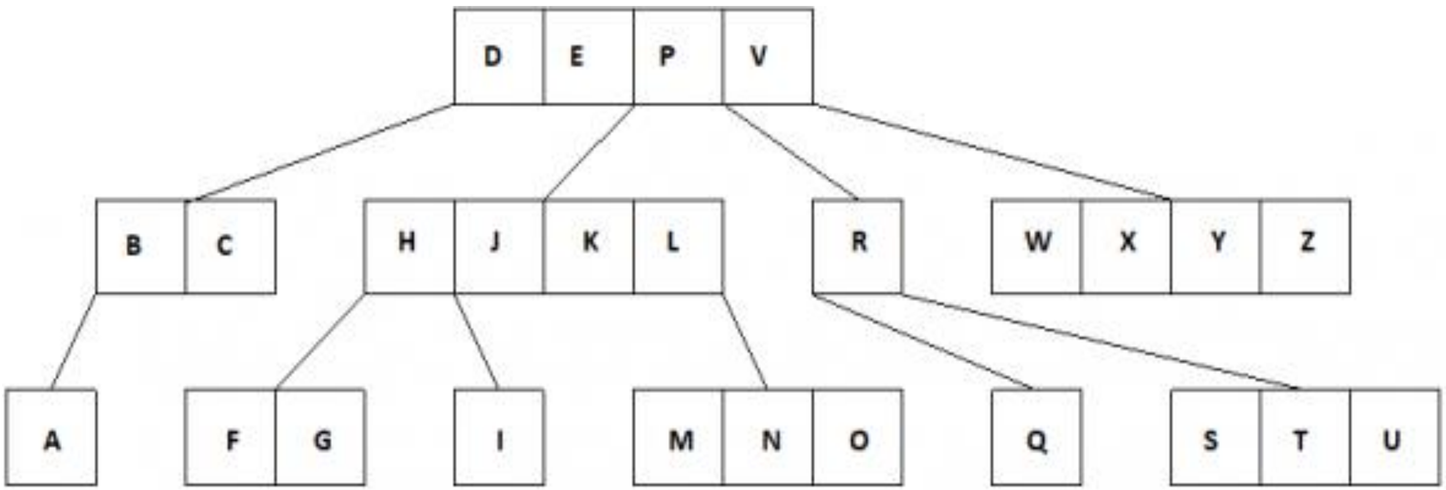
Multiway Search Tree

- 5-Way search tree
- Each node has at most **5 child nodes and at most 4 keys**



Multiway Search Tree

- 5-Way search tree
- Multiway tree of **order 5**
- Each node has at most **5 child nodes and at most 4 keys**



Multiway Search Tree

- Structure of a node of an m-Way tree

```
struct node {
        int count;
        int value[MAX];
        struct node* child[MAX + 1];
        };
```

- **count** represents the **number of children that a particular node has**
- The **values of a node** stored in the array **value**
- The **addresses of child** nodes are stored in the **child** array
- The **MAX** macro signifies the **maximum number of values that a particular node can contain**

Structure of M-Way Search Tree Node

P_0	K_0	P_1	K_1	P_2	K_2	P_3	K_3	P_4	K_4	P_5	K_5	••••••	P_{n-1}	K_{n-1}	P_n

- P0, P1 …, Pn are the **pointers to the node's sub-trees**
- K0, k1, …, Kn-1 are the **key values of the node.**

- All the key values are stored in **ascending order.**

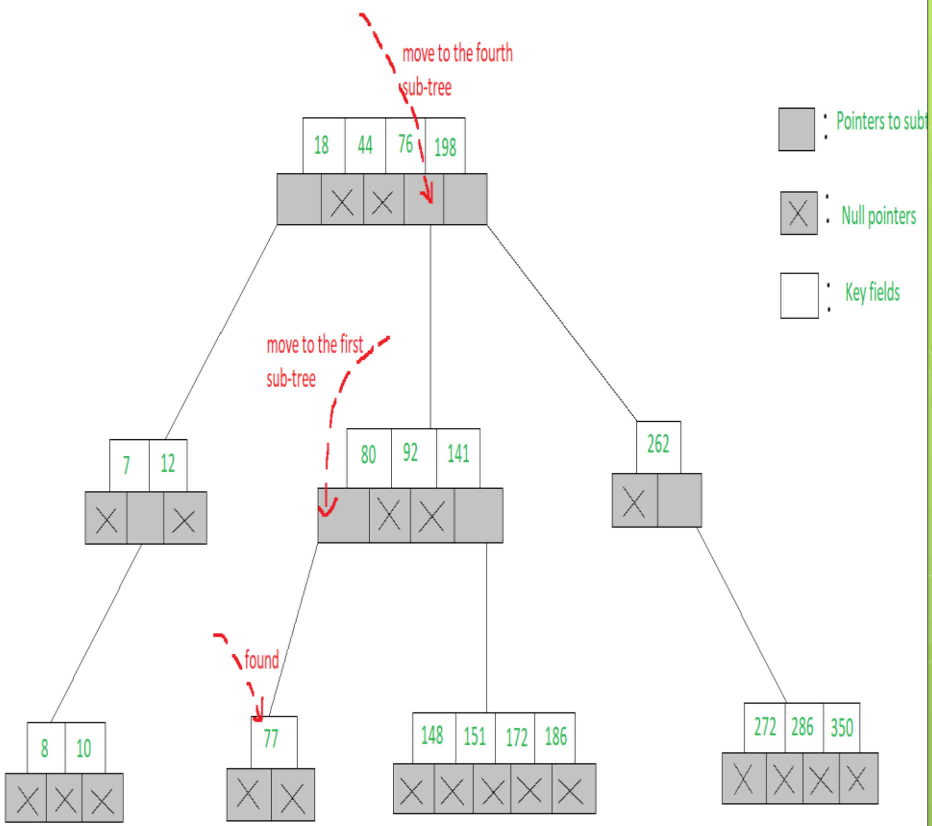
- The basic properties of M-way search trees:

1) **Key vales** in the sub tree pointed by **P0 < key value K0**.
2) **Key values** in the sub-tree pointed by **P1 > key value K0**

Similar pattern for the rest of the P's and K's.

Searching in an m-Way search tree

- **Similar to that of binary search tree**
- To search for **77 in the 5-Way search tree,**
- **Begin at the root**
- As 77> 76> 44> 18, move to the **fourth sub-tree**
- In the root node of the fourth sub-tree, **77< 80** & therefore we move to the **first sub-tree of the node.**
- Since 77 is available in the only node of this sub-tree



11/14/2023

B Tree

B Tree

- A **balanced order n multiway search tree** in which each **non-root node contains atleast (n-1)/2 keys** is called B Tree of Order n
- **B Tree of O(n)**
- **Max no of keys in each node (root/non-root)=n-1**
- **Min** no of keys in **each non-root node=(n-1)/2**

Properties of B Tree

1) All leaf nodes will be at same level
2) Every node except root must contain at least ([n-1]/2) keys. The root may contain minimum 1 key.
3) All nodes (including root) may contain at most n – 1 keys.
4) Number of children of a node is equal to the number of keys in it plus 1.

Properties of B Tree

5) All keys of a node are sorted in increasing order.
6) The child between two keys k1 and k2 contains all keys in the range from k1 and k2.
7) B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

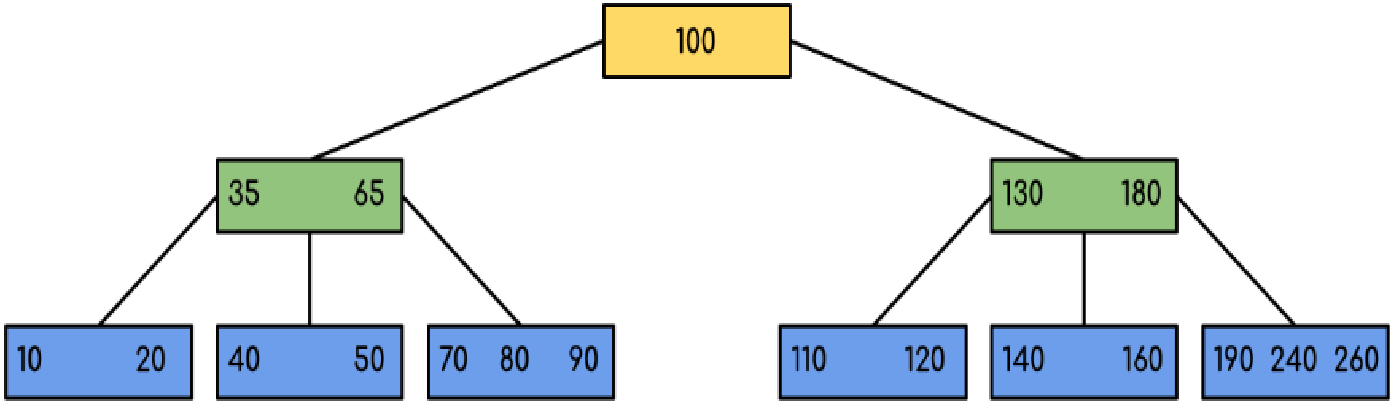
Properties of B Tree

1) The B stands for balanced,

2) In a B-tree the left and right side of each node is roughly kept to the same size (number of subnodes)

Example

1) B-Tree of minimum order 5.
2) All the leaf nodes are at the same level



Animation for Searching in B Tree

- https://condor.depaul.edu/ichu/csc383/notes/notes7/B-Trees_files/tree-search.gif

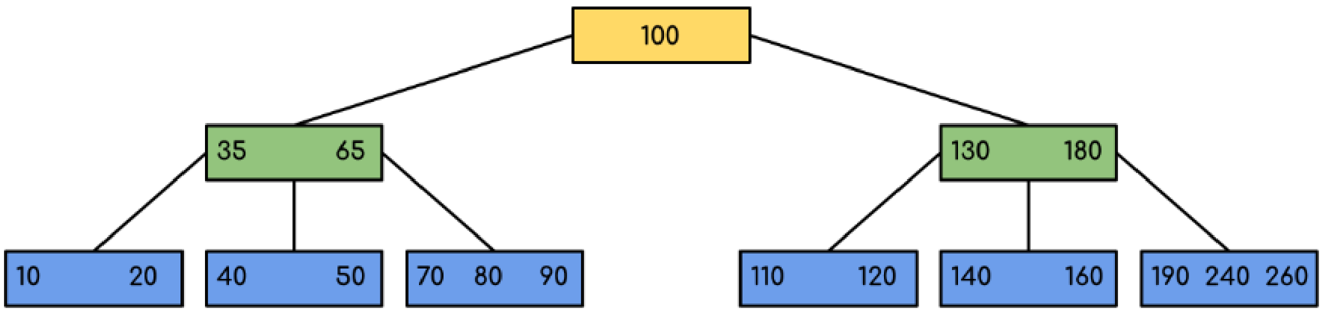
Courtesy: https://condor.depaul.edu/ichu/csc383/notes/notes7/btree.html

Searching in B Tree

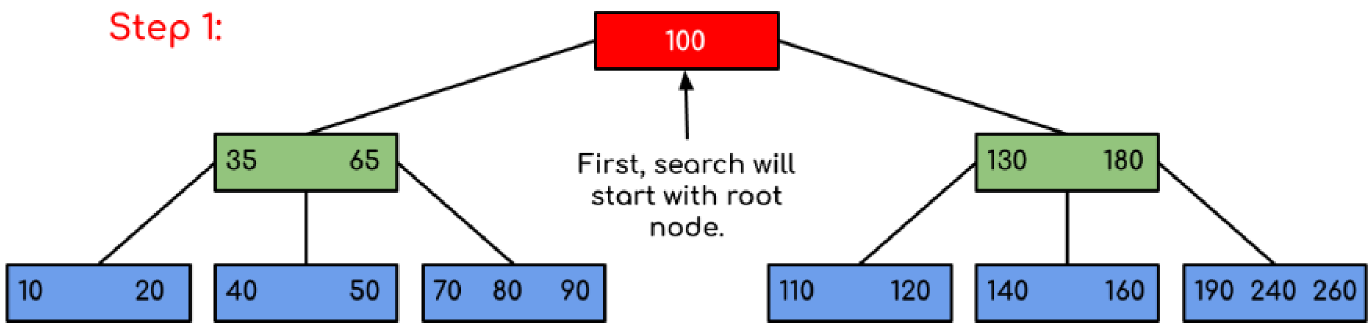
- Searching in B Trees is similar to that in Binary search tree.
  - Let the key to be searched be k.
  - Start from the root
  - Recursively traverse down.
  - For every visited non-leaf node, if the node has the key, we simply return the node.
  - Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.
  - If we reach a leaf node and don't find k in the leaf node, we return NULL.

Searching in B Tree

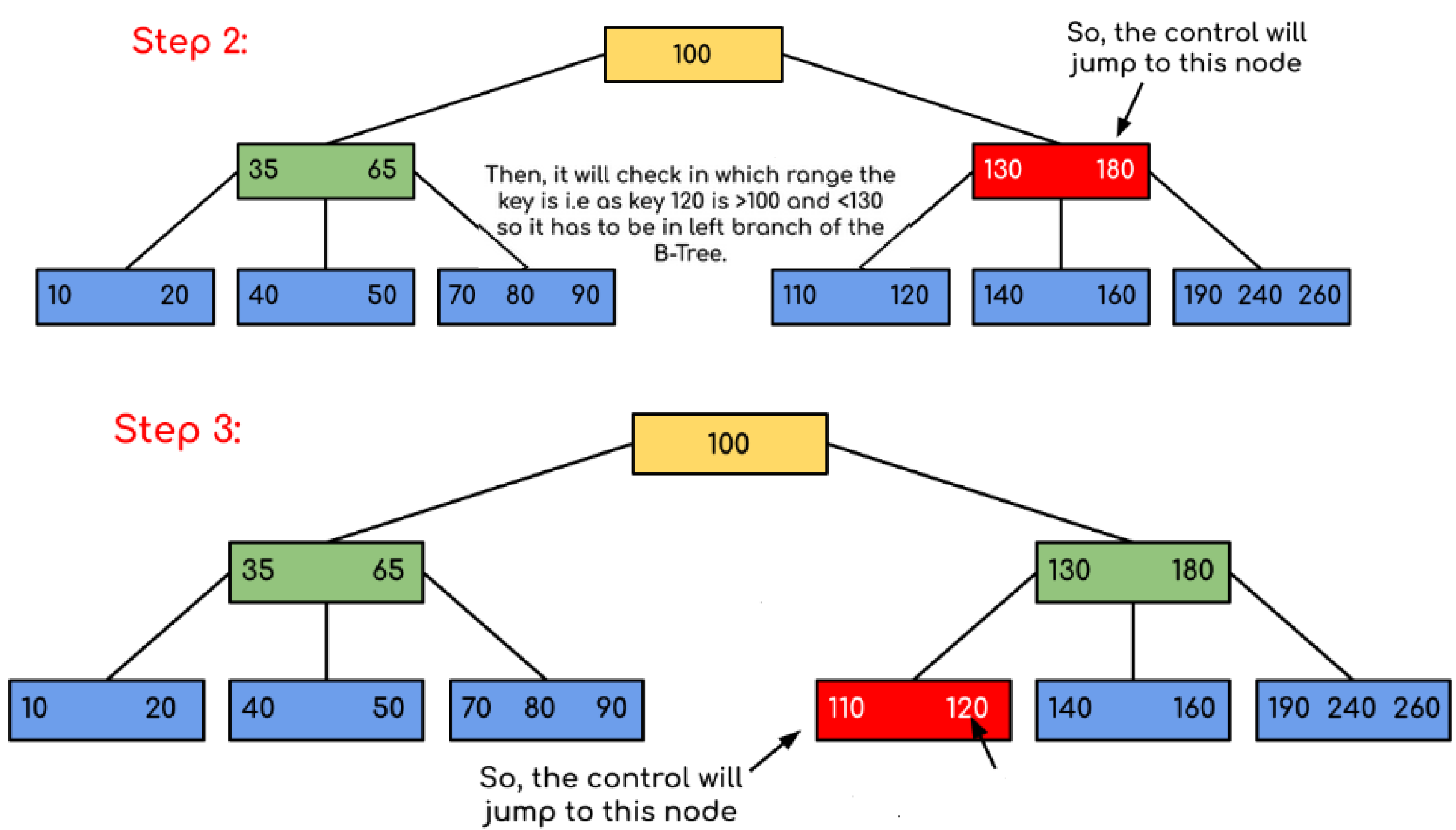
Search for 120



Step 1:



Searching in B Tree



Insertion in B Tree

Insertion in B Tree

- Insertion requires first traversal in B Tree
- **Check key to be inserted is already existing or not**, through traversal
- Suppose the key does not exist in tree then through traversal , it will reach the leaf node
- **We will have 2 cases for inserting the keys**

Insertion in B Tree

The 2 cases are-

- **Case 1: Node is Not Full**
- **Case 2: Node is already full**

Insertion in B Tree

- **Case 1: Node is Not Full-**
  - We simply add the key in the Node
- **Case 2: Node is already full-**
  - **Split the Node in 2 nodes**
  - **Median key goes to the parent of that node**
  - **If parent is also full then same process will be repeated until it will get non-full parent node**

Algorithm for Insertion in B Tree

- The following algorithm applies:

1) Run the search operation and find the appropriate place of insertion.
2) **Insert the new key at the proper location, but if the node has a maximum number of keys** already:
3) **The node, along with a newly inserted key, will split from the middle element.**
4) The **middle element will become the parent** for the other two child nodes.
5) **The nodes must re-arrange keys in ascending order.**

Courtesy-https://www.guru99.com/b-tree-example.html

Algorithm for Insertion in B Tree

TIP

The following is not true about the insertion algorithm:

- Since the node is full, therefore it will split, and then a new value will be inserted

CORRECT METHOD-

- **The node, along with a newly inserted key, will split from the middle element.**

Courtesy-https://www.guru99.com/b-tree-example.html

11/14/2023

Insertion with Odd Order

Insertion in B Tree

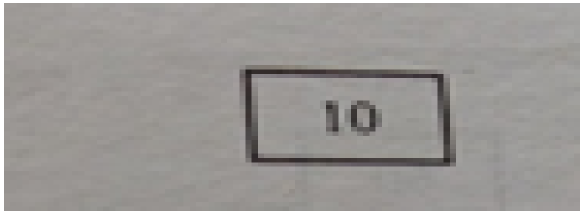
- Create a B Tree of Order 5
- n=5
- List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

- **Max no of keys in each node (root/non-root)=n-1=4**
- **Min no of keys in each non-root node=(n-1)/2=2**

Insertion in B Tree

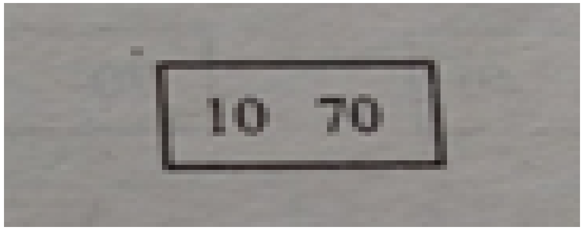
List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

Insert 10



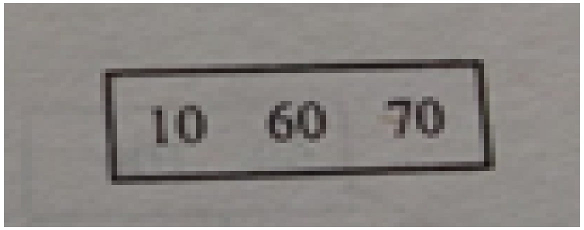
Insert 70

After Inserting 70, the keys in the node will be sorted



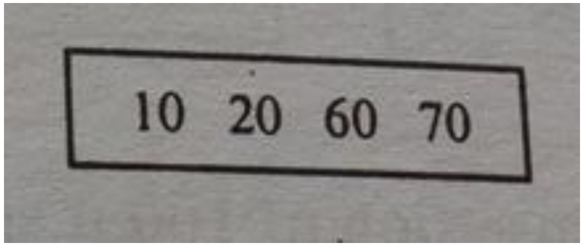
Insert 60

After Inserting 60, the keys in the node will be sorted



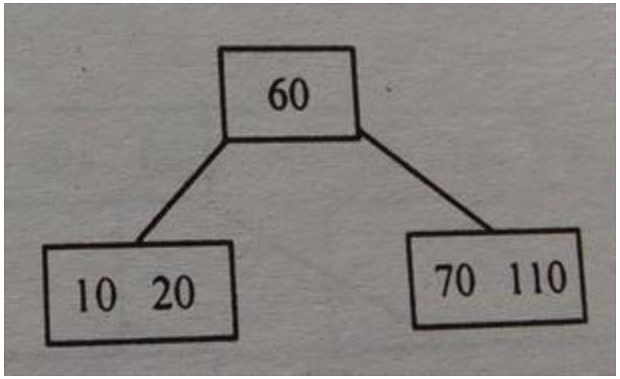
Insertion in B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160



Insert 20

After Inserting 20, the keys in the node will be sorted



Insert 110

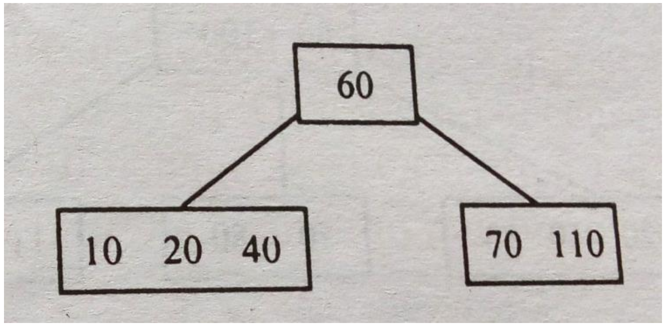
Node was already full,

After insertion of 110 , It splits into 2 nodes

60 is the median key, so it goes to parent or becomes root

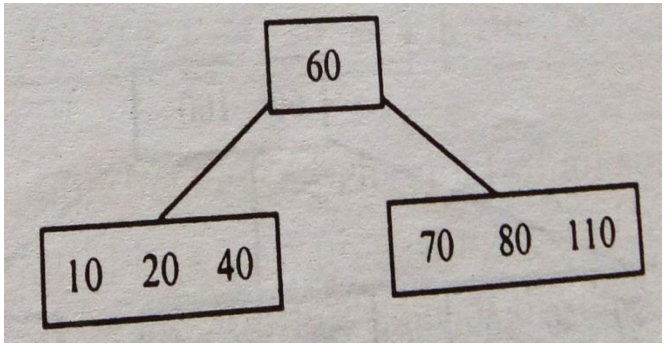
Insertion in B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160



Insert 40

After Inserting 40, the keys in the node will be sorted

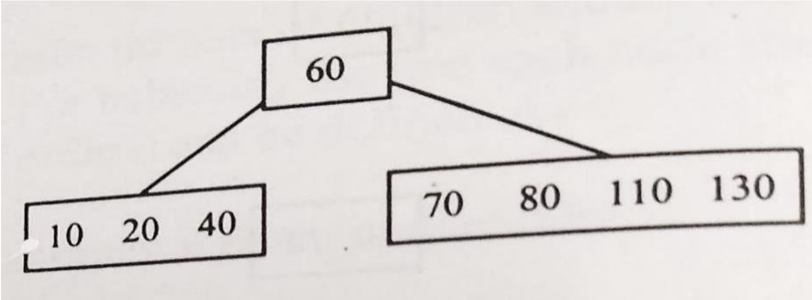


Insert 80

After Inserting 80, the keys in the node will be sorted

Insertion in B Tree

List of Keys=10,70,60,20,110,40,80,130, 100,50,190,90,180,240,30,120,140,160



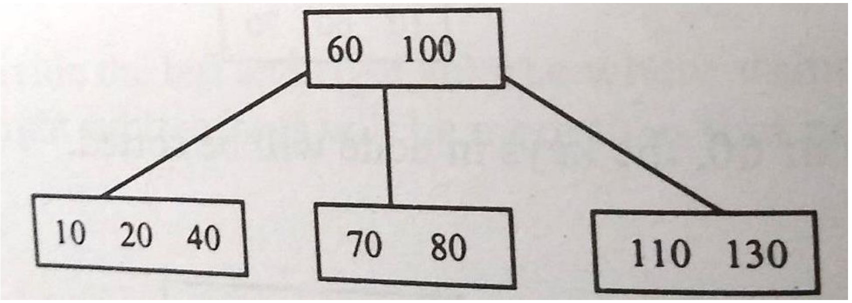
Insert 130

Insert 100

Node was already full

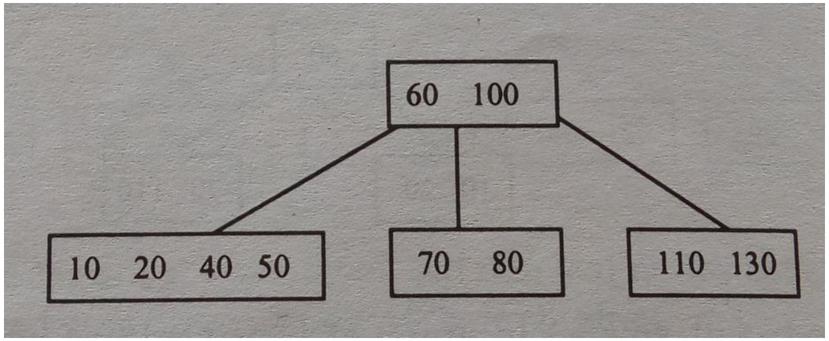
After insertion of 100, it splits in 2 nodes, 100 is the median key ,

100 goes up to the parent node



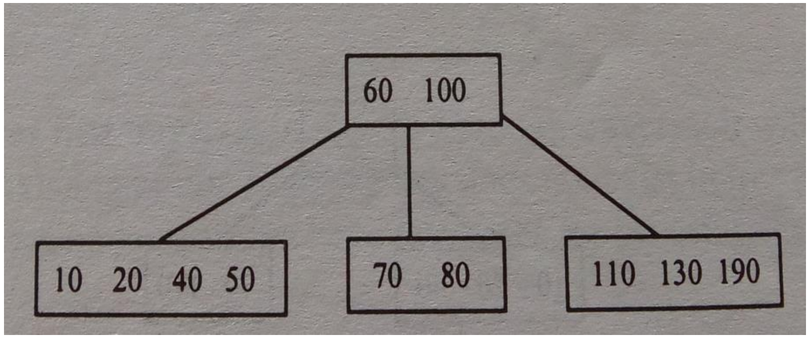
Insertion in B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160



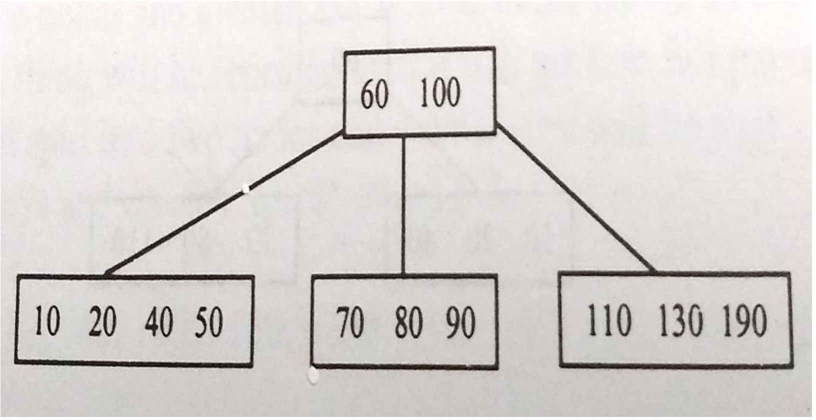
Insert 50

Insert 190



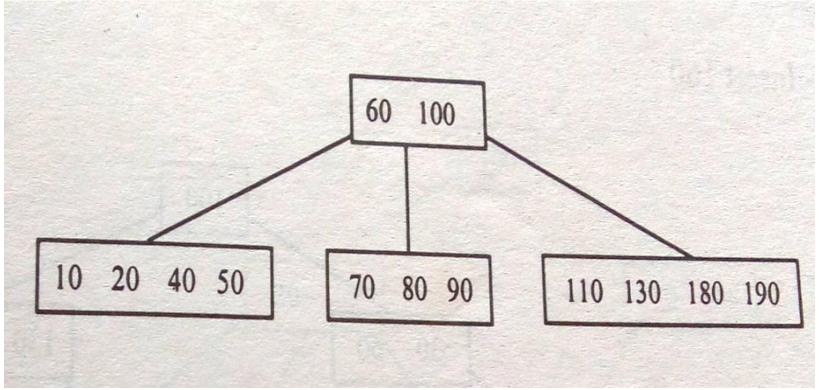
Insertion in B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160



Insert 90

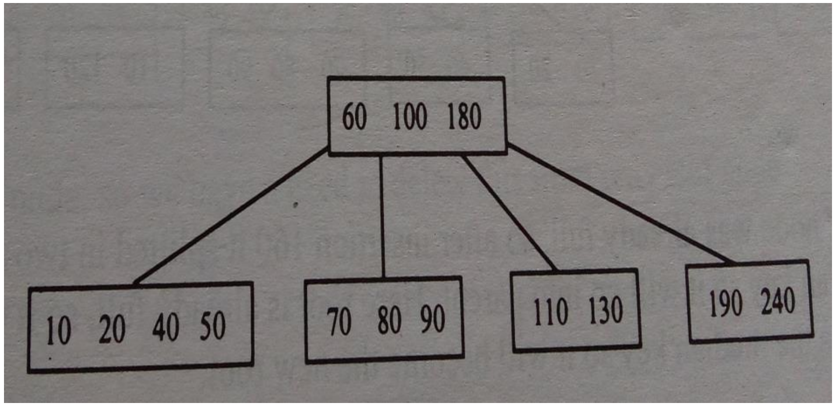
Insert 180



Insertion in B Tree

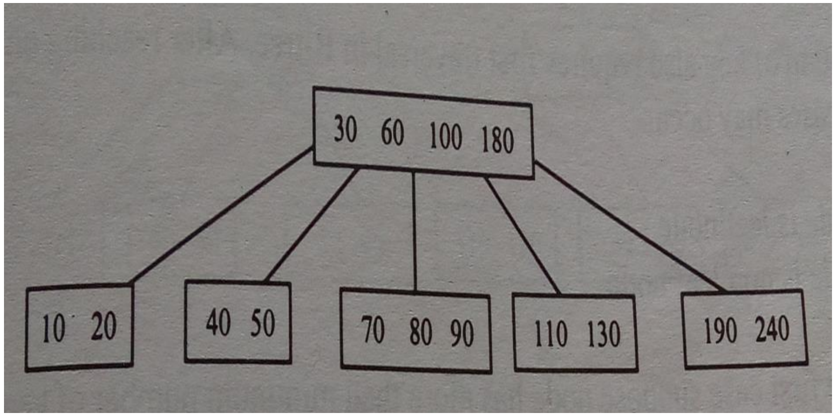
List of Keys=10,70,60,20,110,40,80, 130,100,50,190,90,180,240,30,120,140,160

Insert 240



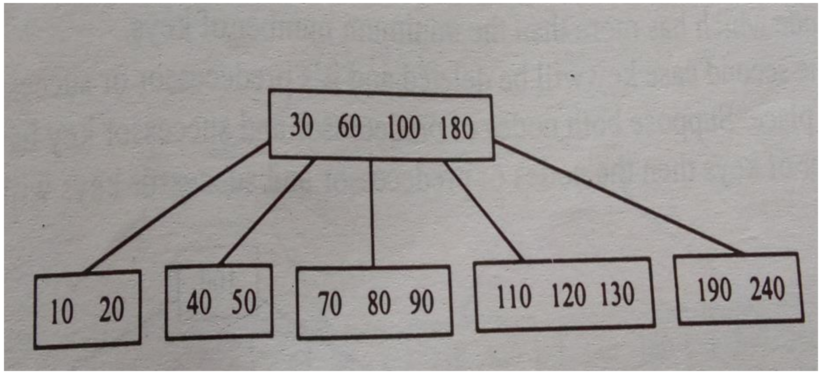
Insert 30

Node was already full, so after insertion of 30, splits in 2 nodes, 30 is the median key so it will go to the parent

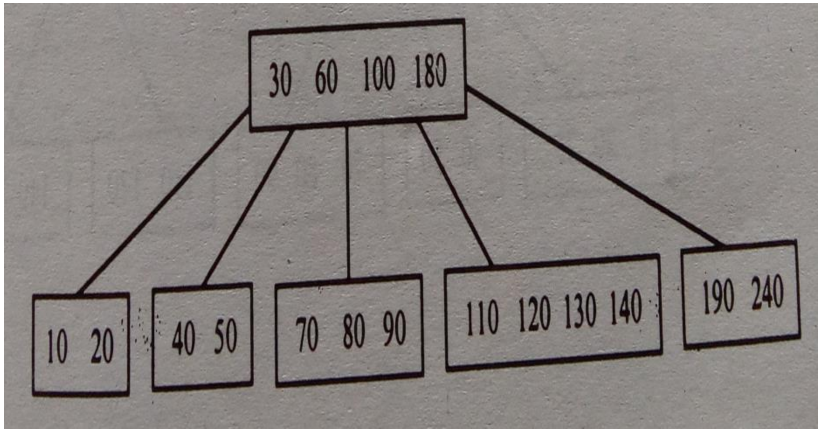


Insertion in B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160



Insert 120

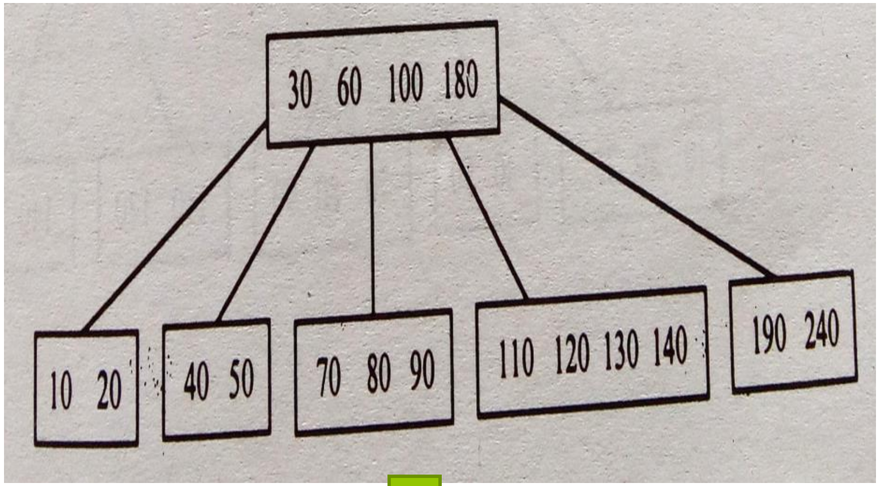


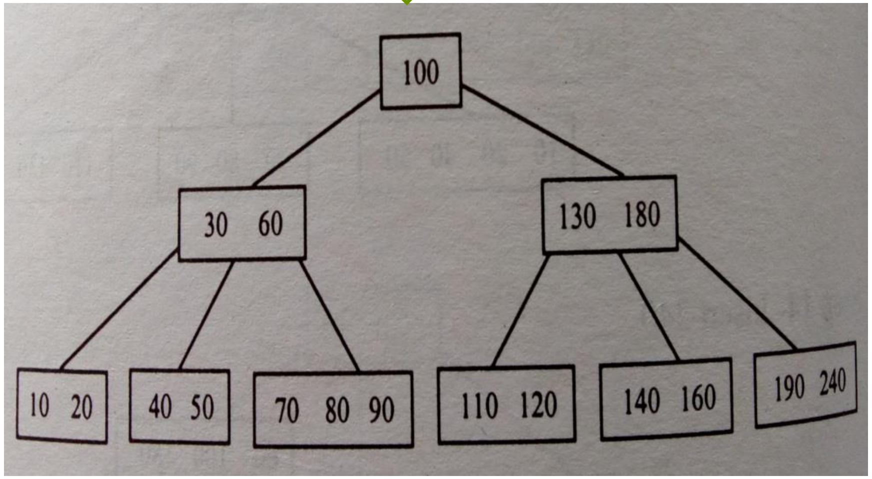
Insert 140

Insertion in B Tree

List of Keys=10,70,60,20,110,40,80,130,100,50,190,90,180,240,30,120,140,160

Insert 160
Node was already full,
After insertion of 160
Splits into 2 nodes
130 is the median so it goes up
Root is already full, so it splits in 2 nodes , 100 is the median so it becomes new root





Example

- Illustrate the steps to build a B Tree of order 5 for the following data 78, 20, 10, 93, 82, 75, 69, 45, 42, 13, 36 Show all the intermediate steps.

Insertion with Even Order

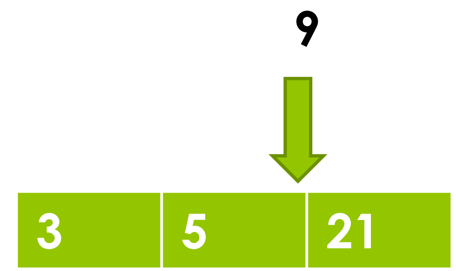
Insertion in B Tree

- In case of **even number of keys,**

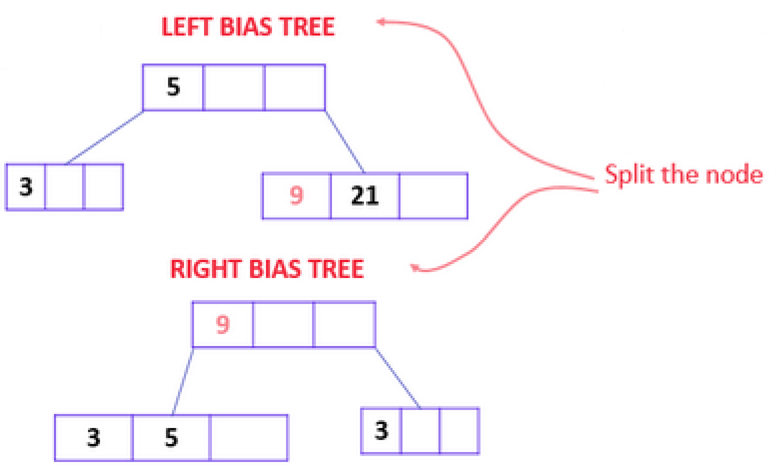
- The middle node will be selected by
  - **Left bias or**
  - **Right bias**

Insertion in B Tree

- Order =4
- List=5,3,21,9,1,13,2,7
- Min no of nodes(non-root)=(n-1)/2=3/2=1



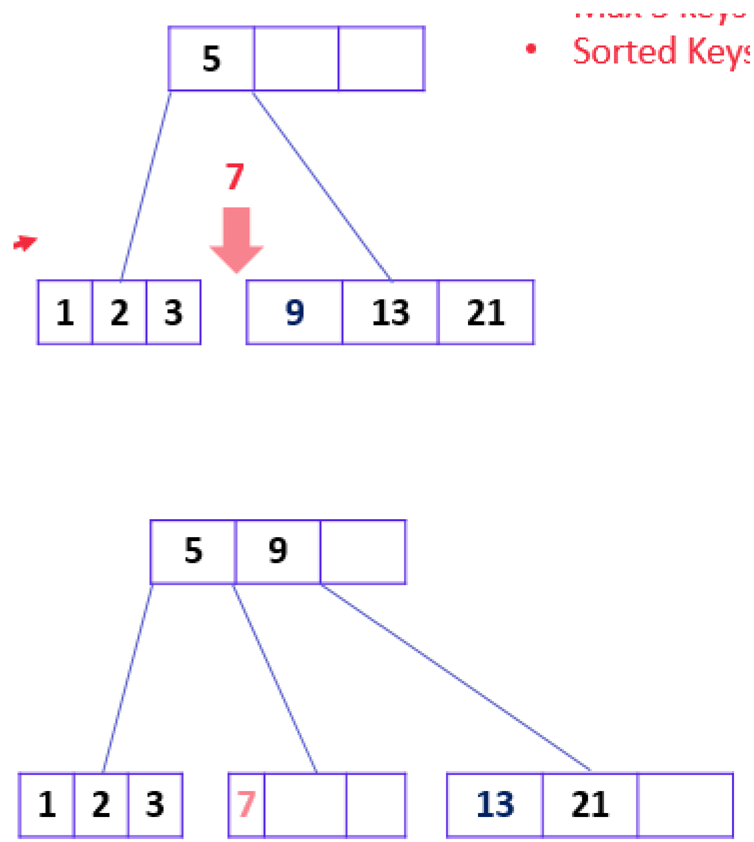
- Insert 9
- Even number of keys,
- The middle node will be selected by
  - **Left bias or**
  - **Right bias**



Insertion in B Tree

- Order =4
- List=5,3,21,9,1,13,2,7

- Middle key by **Left bias**



Splitting in B Tree

List=10, 20, 30, 40, 50, 60, 70, 80 and 90

Insert in an initially empty B-Tree of **Order 6**

n=6

- **Max no of keys in each node (root/non-root)=n-1=5**
- **Min no of keys in each non-root node=(n-1)/2**
  =(6-1)/2
  =5/2
  =2

Splitting in B Tree

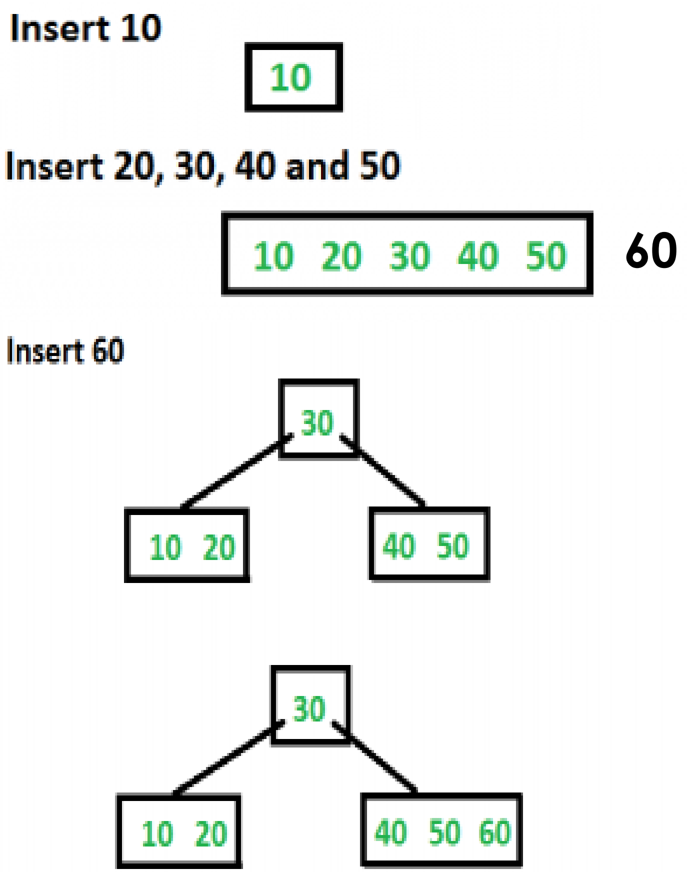
List=10, 20, 30, 40, 50, 60, 70, 80 and 90

Insert 10

Insert 20,30,40

Insert 60

1) Since root node is full,
2) **Node will split into two nodes**
3) **Median=30, using Left Bias so 30 goes to parent or becomes root,**



Splitting in B Tree

Construct a B-Tree with data 75,52,81,40,33,90,85,20,38

Insert in an initially empty B-Tree of Order 4

n=4

- Max no of keys in each node (root/non-root)=n-1=3
- Min no of keys in each non-root node=(n-1)/2=(3)/2=1

Order 4

Minimum no of keys in a node $= \frac{n-1}{2} = \frac{4-1}{2} = \frac{3}{2} = 1$

Maximum no of Keys in a node $= n-1 = 4-1 = 3$

Insert 75

(75)

Insert 52

(52 75)

Insert 81

(52 75 81)

Insert 40

Goes to parent

(40 52 75 81) splitting

52

40 | 75 81

Insert 33

52

33 40 | 75 81

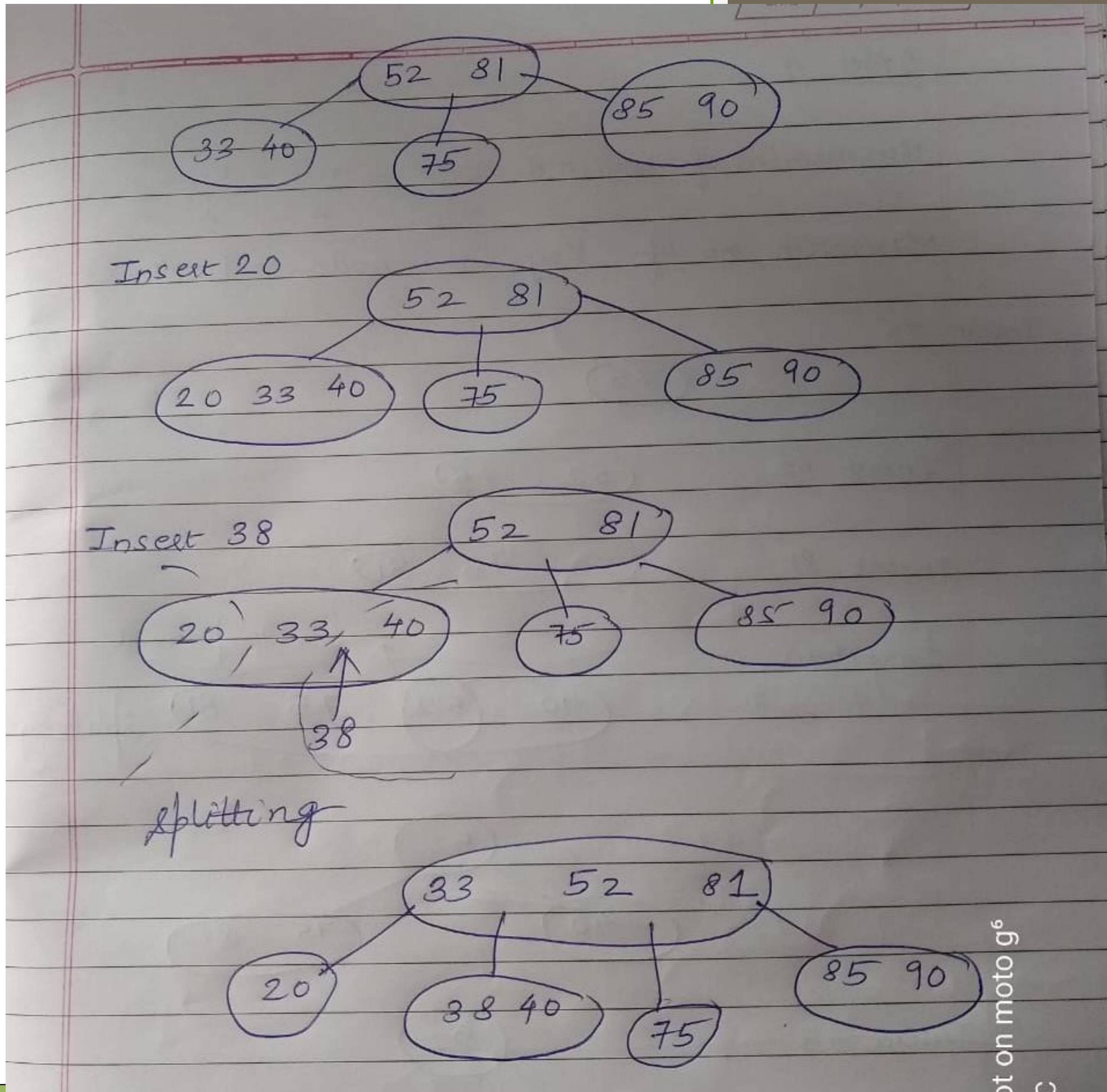
Insert 90

Insert 85

52

33 40 | 75, 81 90 85 splitting

Insert 85


52 81
33 40
75
85 90
Insert 20
52 81
20 33 40
75
85 90
Insert 38
52 81
20 33 40
75
85 90
38
Splitting
33 52 81
20
38 40
75
85 90

Deletion in B Tree

Deletion in B Tree

- **Deletion also requires traversal**
- After reaching a particular node
- 2 cases may occur-
  - **Node is a leaf node**
  - **Node is non leaf node**

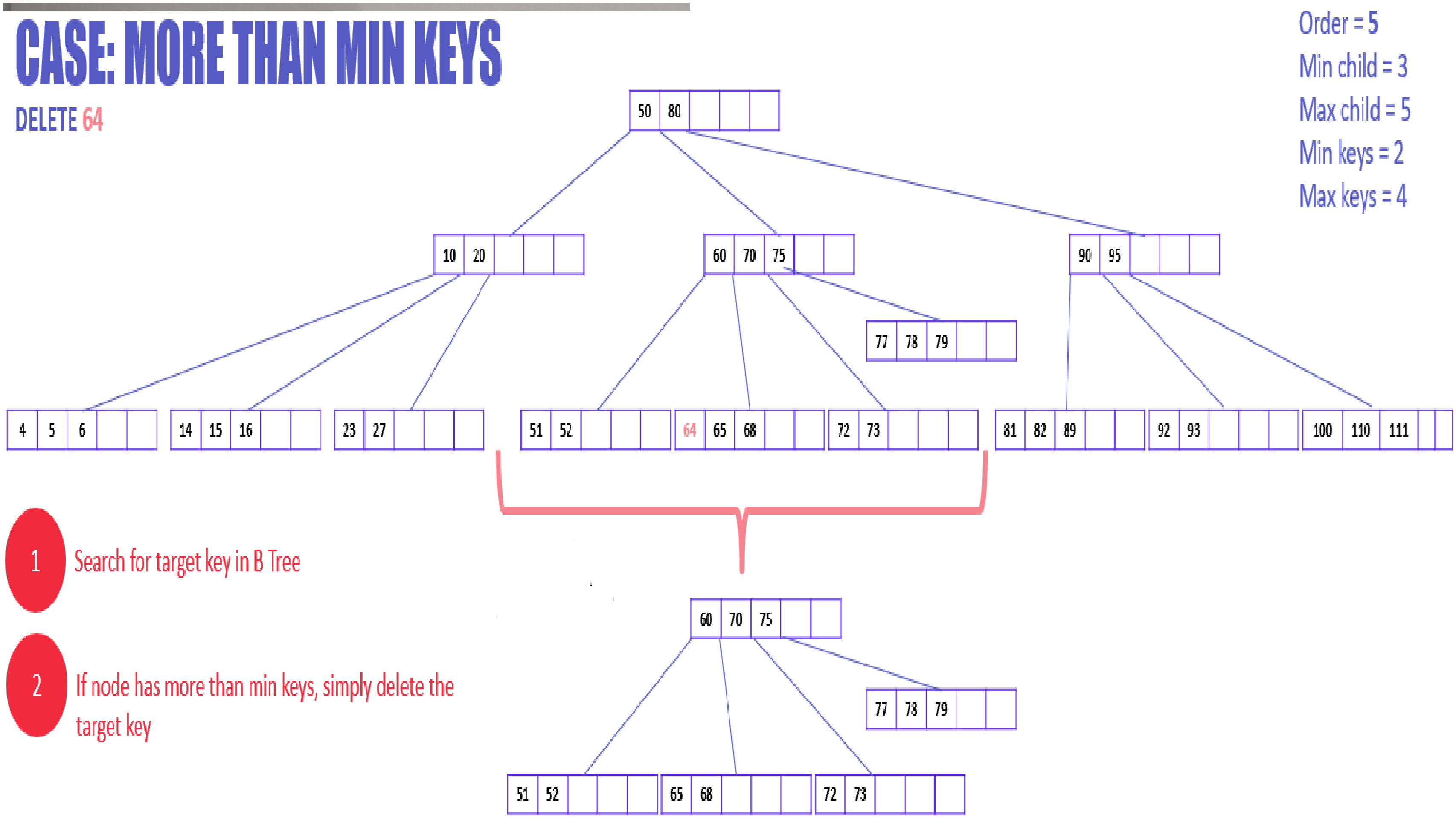
Deletion in B Tree

Case 1-Node is a leaf node

- **If node has more than minimum of keys then it can be easily deleted**
  - **Just delete it**
  - Deleting this will not violate the property of B Tree

Deletion in B Tree

Case 1-Node is a leaf node



Deletion in B Tree

Case 1-Node is a leaf node

- **If it has minimum no of keys**
  - 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 of sibling will be transferred to a parent [Left Sibling]**
  - **The max value of the parent node will be transferred to the target node, and remove the target value**

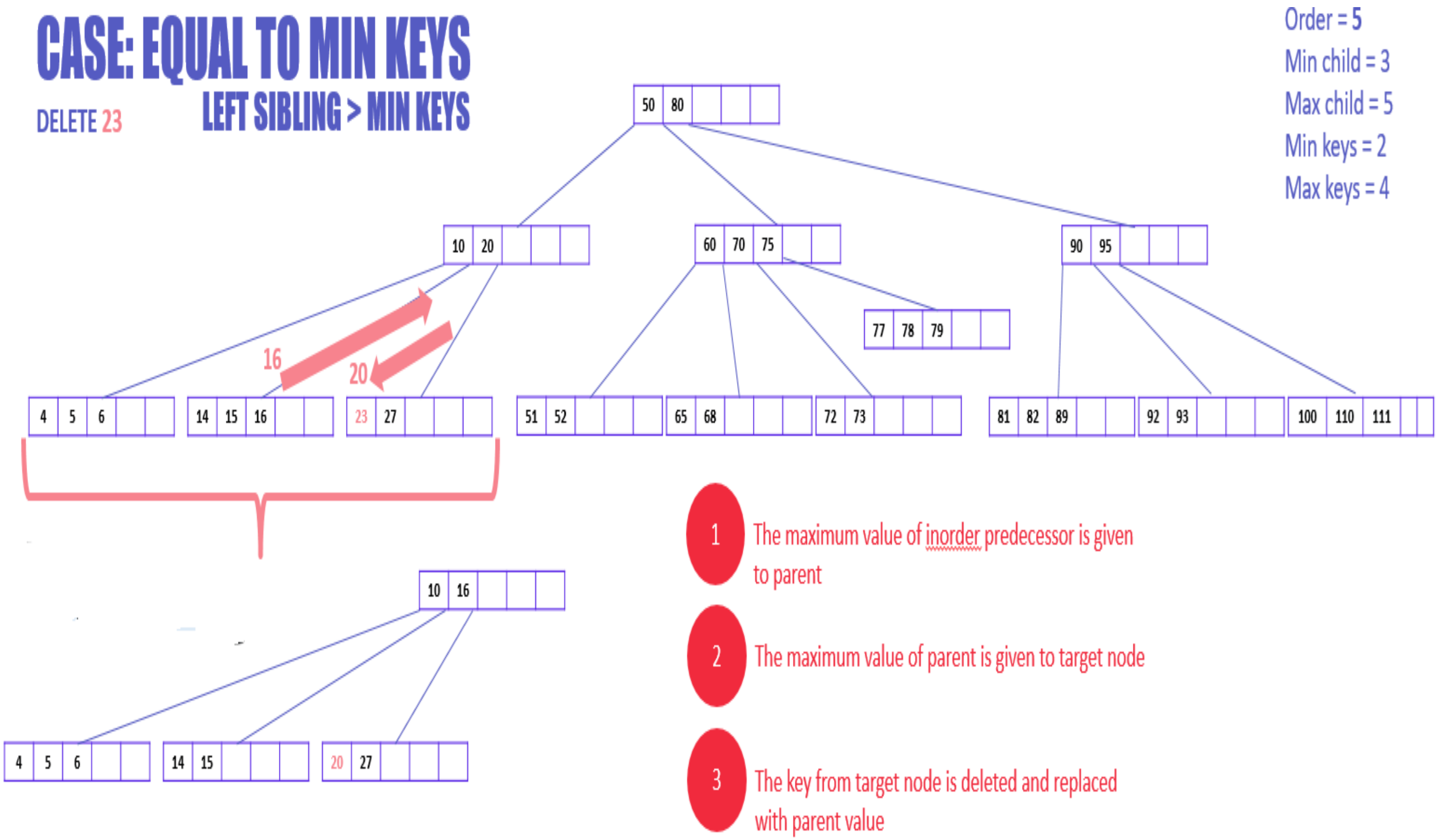
Deletion in B Tree

Case 1-Node is a leaf node

- If it has minimum no of keys
  - Pull up one key from adjacent node to father and pull down the father

Deletion in B Tree

Case 1-Node is a leaf node



Deletion in B Tree

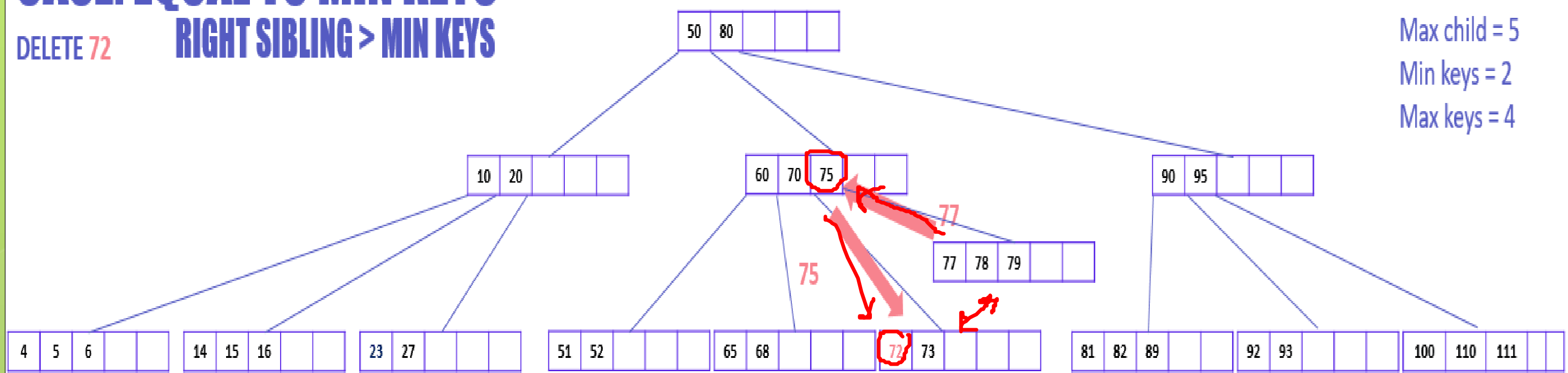
Case 1-Node is a leaf node

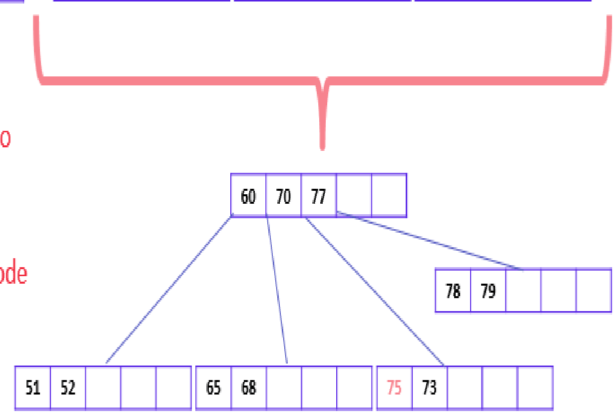
CASE: EQUAL TO MIN KEYS

DELETE 72

RIGHT SIBLING > MIN KEYS

Order = 5
Min child = 3
Max child = 5
Min keys = 2
Max keys = 4





1. The minimum value of inorder successor is given to parent
2. The maximum value of parent is given to target node
3. The key from target node is deleted and replaced with parent value

Deletion in B Tree

Case 1-Node is a leaf node

- **If it has minimum no of keys**
- **Key cannot be borrowed from Sibling**
- **Merge the target node with any of its immediate siblings along with parent key**
  - That key from the parent node is merged which comes in between the two merging sibling
- Delete the target key from the merged node

Deletion in B Tree

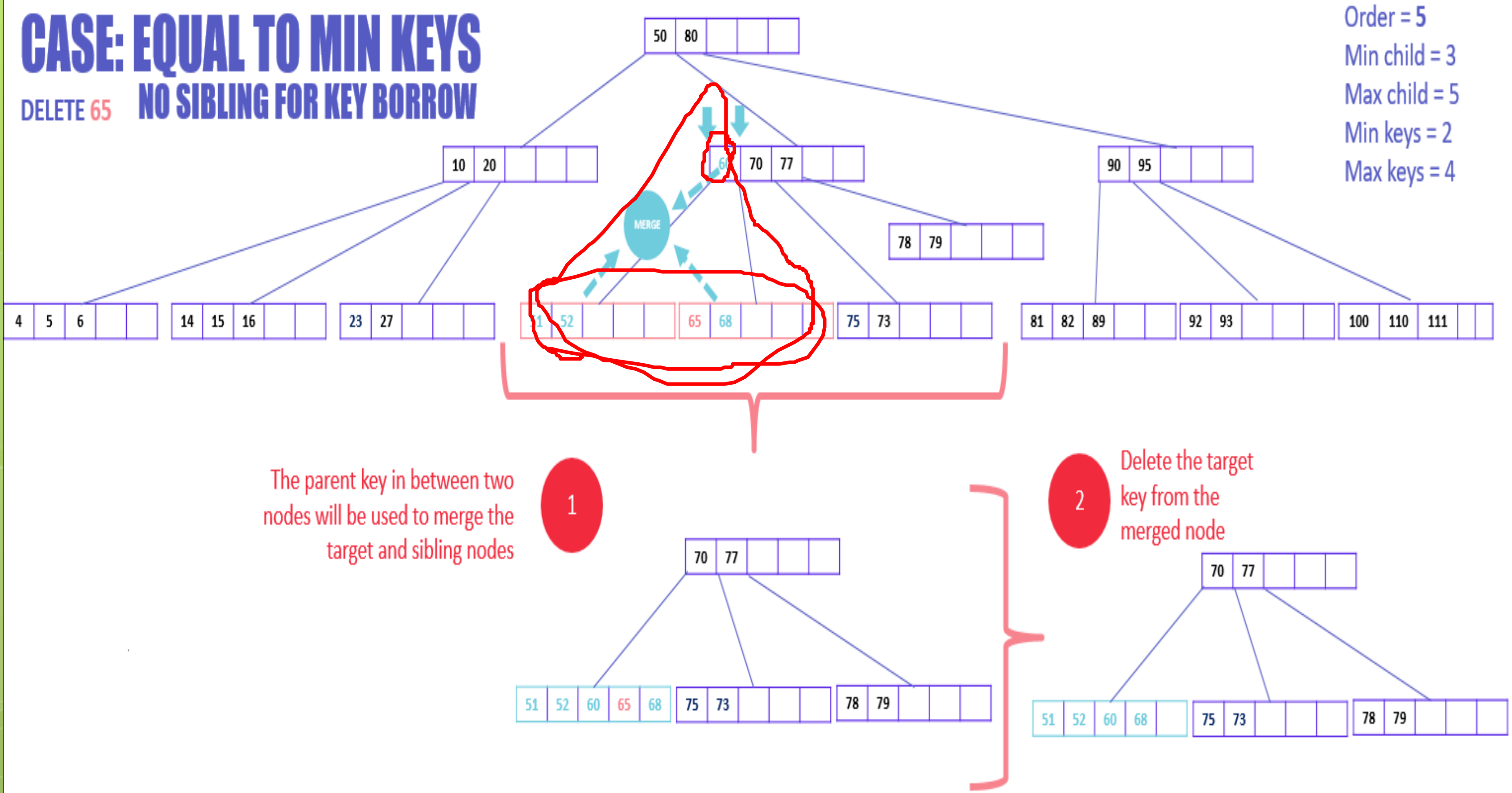
Case 1-Node is a leaf node

- If it has minimum no of keys
- If adjacent node also has minimum no of keys, then the two adjacent leaves and the median key from the parent can be combined as one new leaf

Deletion in B Tree

Case 1-Node is a leaf node

- If it has minimum no of keys
- Key cannot be borrowed from Sibling



Deletion in B Tree

Case 2-Node is a non leaf node

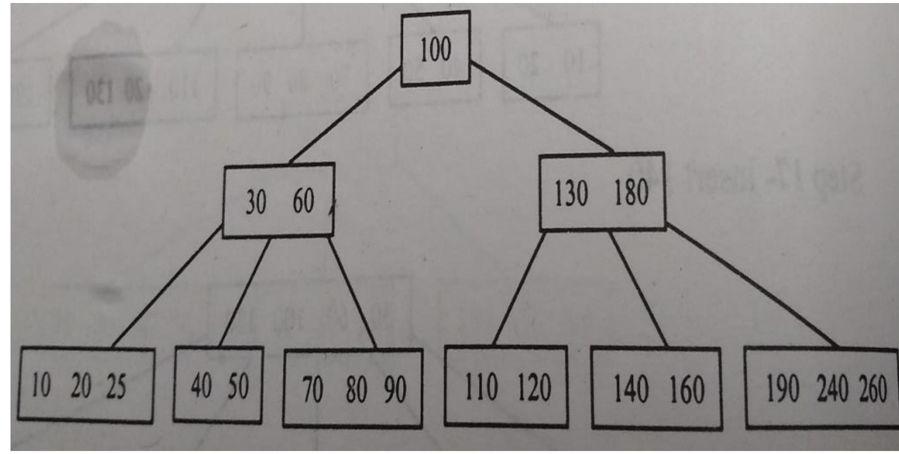
- The key will be deleted and its predecessor or successor key will come on its place
- (Inorder Predecessor or Inorder Successor, depending on which child node has extra keys)
- If suppose both nodes of predecessor and successor have minimum keys then the nodes of predecessor and successor will be combined

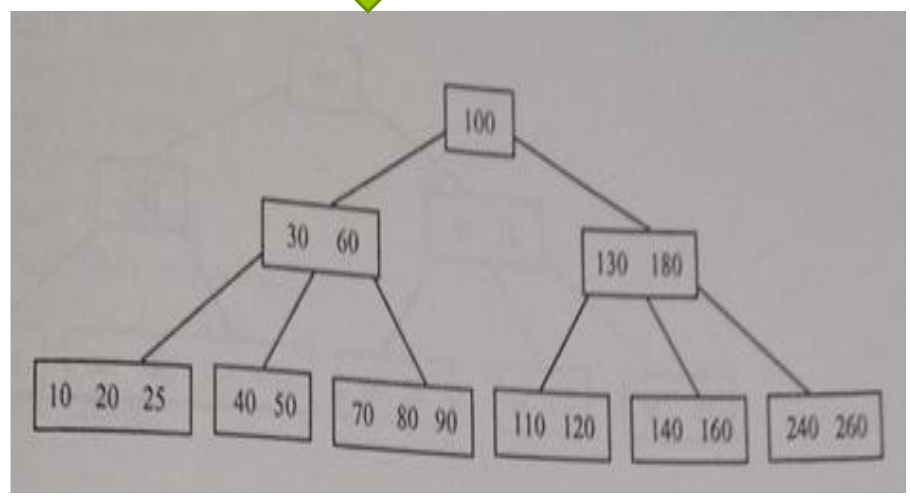
Deletion in B Tree

- Delete 190

Case 1-Node is a leaf node

- **If node has more than minimum of keys**
- **Just delete it**



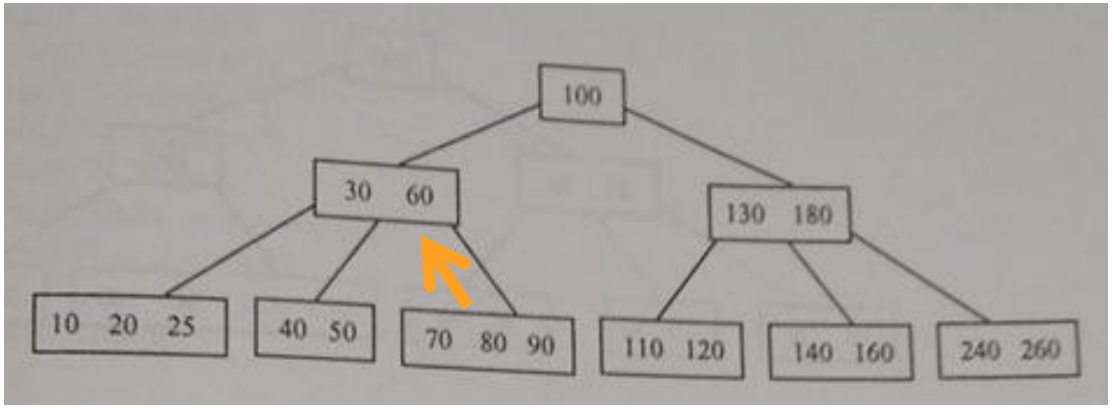


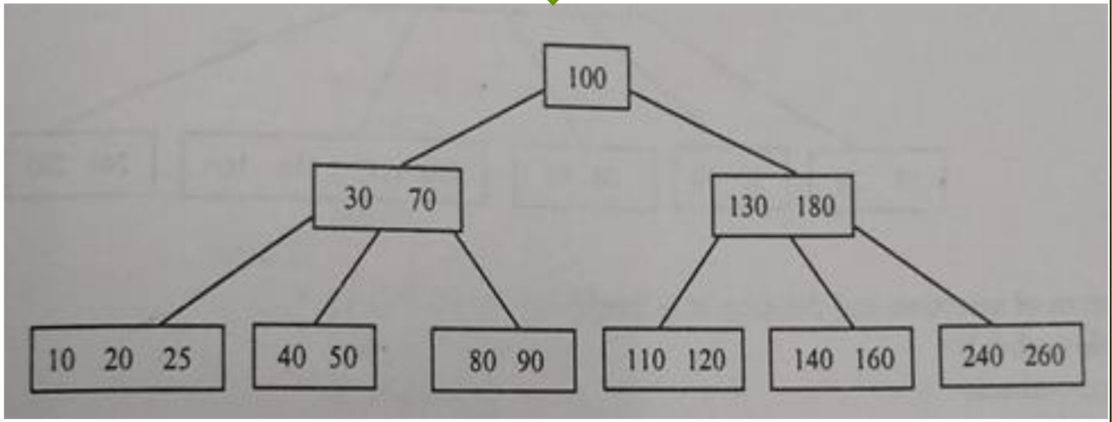
Deletion in B Tree

- Delete 60

Case 2-Node is a non leaf node

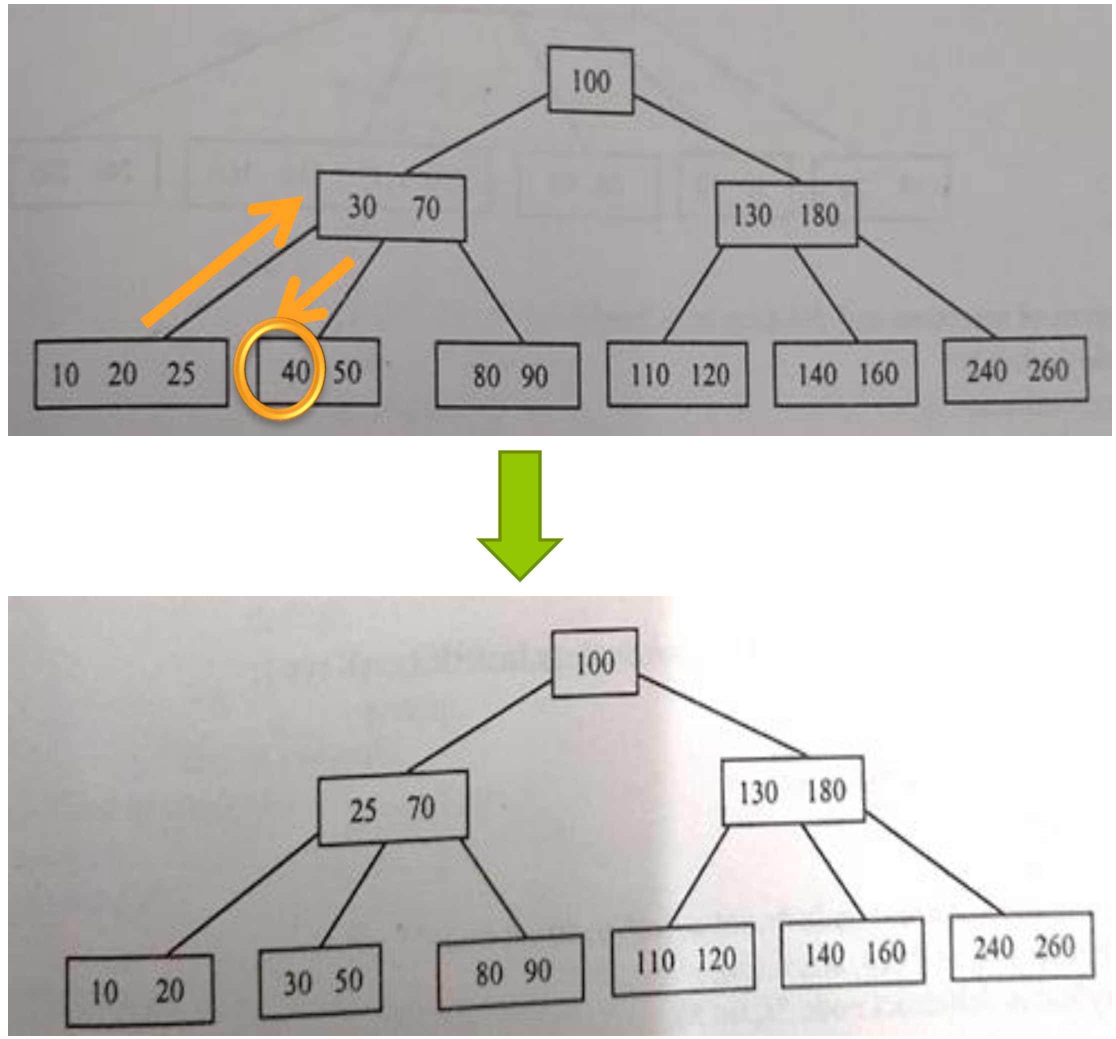
- **The key will be deleted and its predecessor or successor key will come on its place**





Deletion in B Tree

- Delete 40
- **If it has minimum no of keys**
  - **Pull up one key from adjacent node to father and pull down the father**

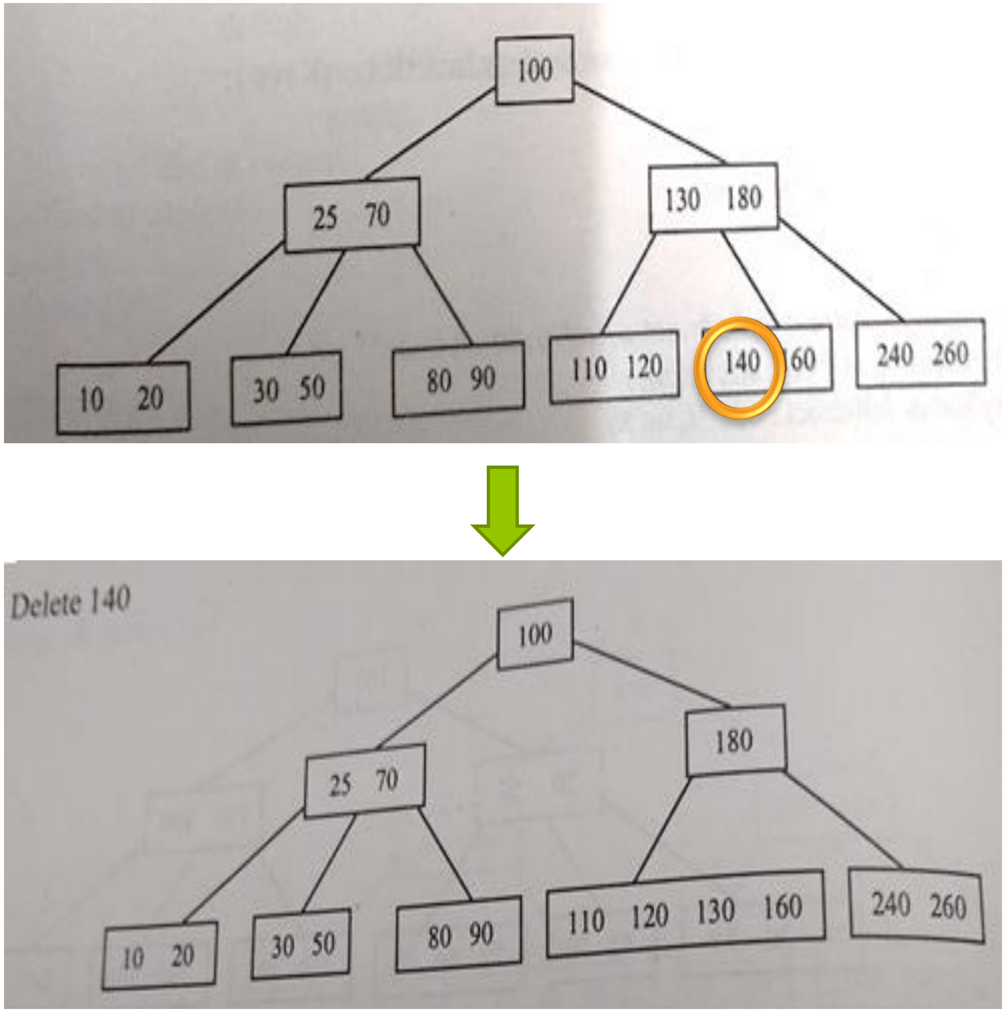


Deletion in B Tree

- Delete 140

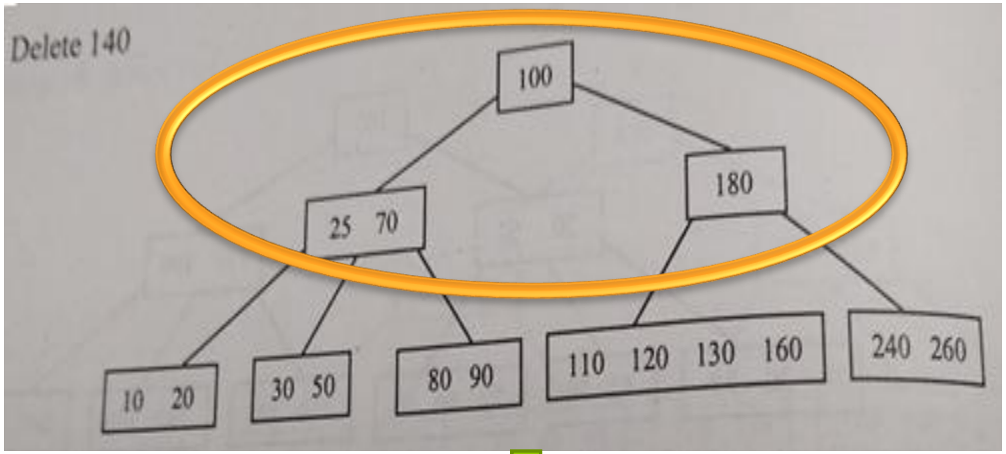
Case 1-Node is a leaf node

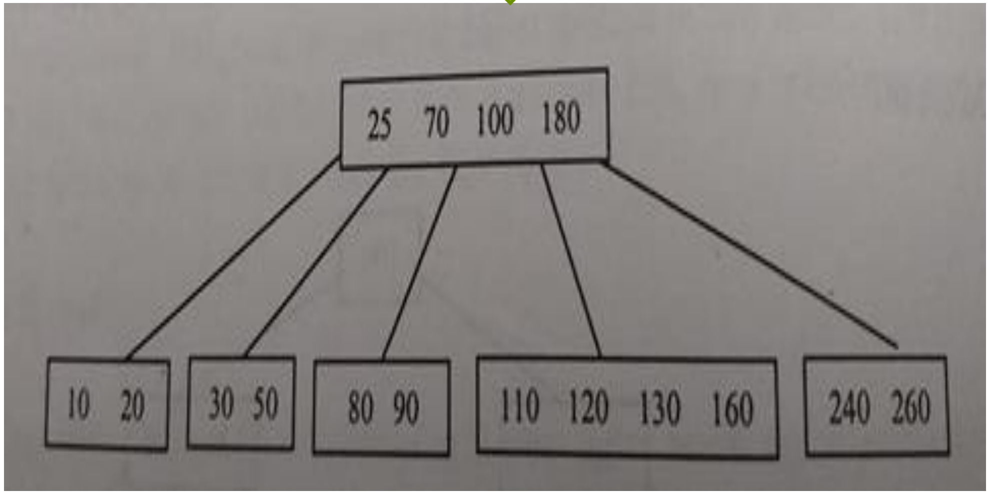
- **If it has minimum no of keys**
- **If adjacent node also has minimum no of keys, then the leaves and the median key from the parent can be combined as one new leaf**



Deletion in B Tree

- Merging at Level 0
- Final Tree





11/14/2023

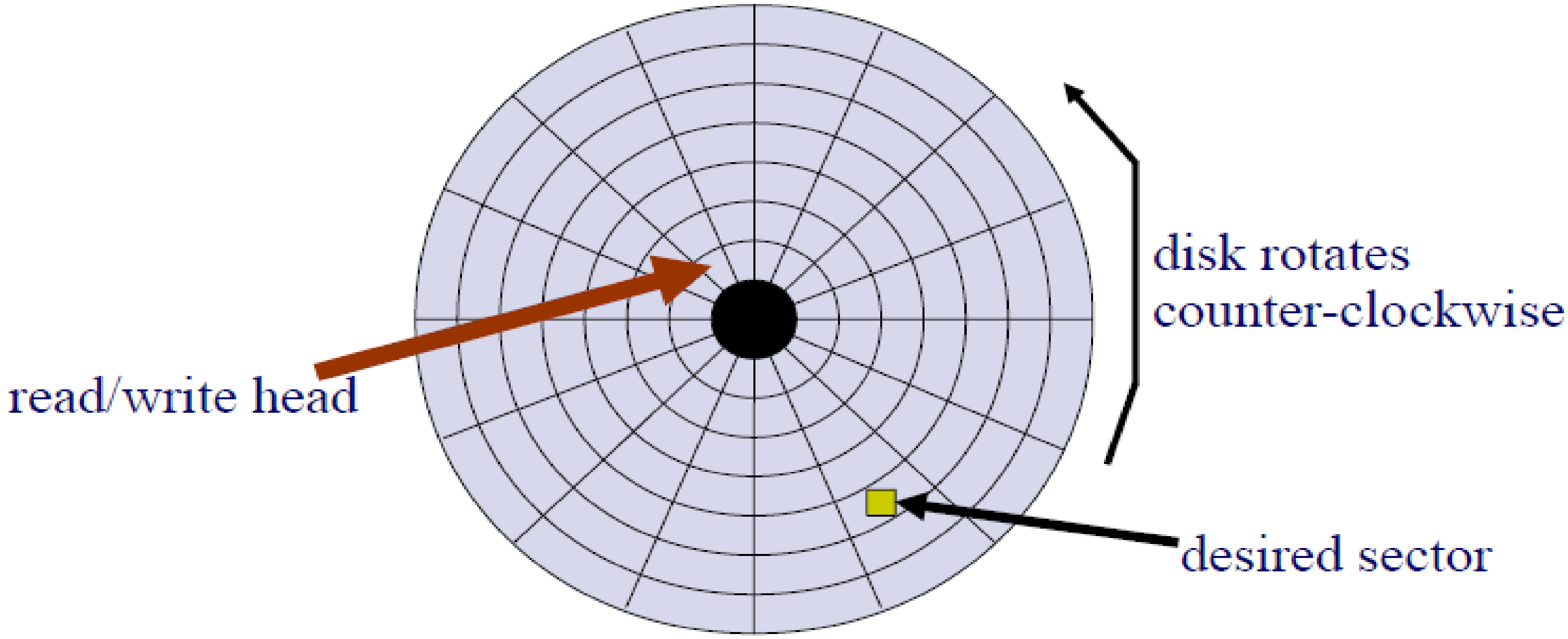
Uses/Application of B Tree

Use of B Tree

1) B-trees are balanced search trees designed to work well **on magnetic disks or other direct-access secondary storage devices**
2) Better at **minimizing disk I/O operations**
3) There is huge amount of data that cannot fit in main memory.
   - When the number of keys is high, **the data is read from disk in the form of blocks.**
   - **Disk access time is very high compared to the main memory access time.**

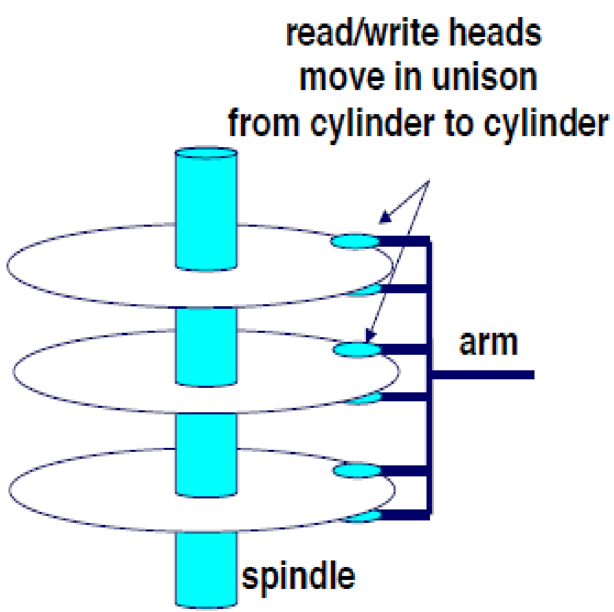
Anatomy of a Hard Drive

Let's read in a sector from the disk



Use of B Tree

- The time it takes to access data on secondary storage is a function of three variables:
  - The time it takes for the **arm to move to the track** where the requested sector lies. Usually around 10 milliseconds.
  - The time it takes for the **right sector to spin under the arm**. For a 7200 RPM drive, this is 4.1 milliseconds.
  - The time it takes **to read or write the data.** Depending on the density of the data, this time is negligible compared to the other two.



Use of B Tree

- Data can be **arranged into "blocks" that are these adjacent multi-sector aggregates.**

- Contrast this to access times to RAM.
  - A typical non-sequential RAM access took **about 5 microseconds.**
  - This is 3000 times faster; we can do **at least 3000 memory accesses in the time it takes to do one disk access,**

Use of B Tree

1) One of the main reason of using B tree is
   - its capability to **store large number of keys in a single node and**
   - by **keeping the height of the tree relatively small.**

2) The main idea of using B-Trees is to reduce the number of disk accesses.

Use of B Tree

1) Most of the tree operations **(search, insert, delete, max, min, ..etc) require O(h) disk accesses**
   - where h is the height of the tree.

2) The height of B-Trees is **kept low by putting maximum possible keys in a B-Tree node.**

3) **Generally, the B-Tree node size is kept equal to the disk block size.**

Use of B Tree

1) In a typical B-tree application, the amount of data handled is so large that all the data do not fit into main memory at once.

2) **The B-tree algorithms copy selected blocks from disk into main memory as needed and write back onto disk pages that have changed.**

3) Since the B-tree algorithms only **need a constant number of blocks in main memory at any time, the size of main memory does not limit the size of B-trees that can be handled.**

Use of B Tree

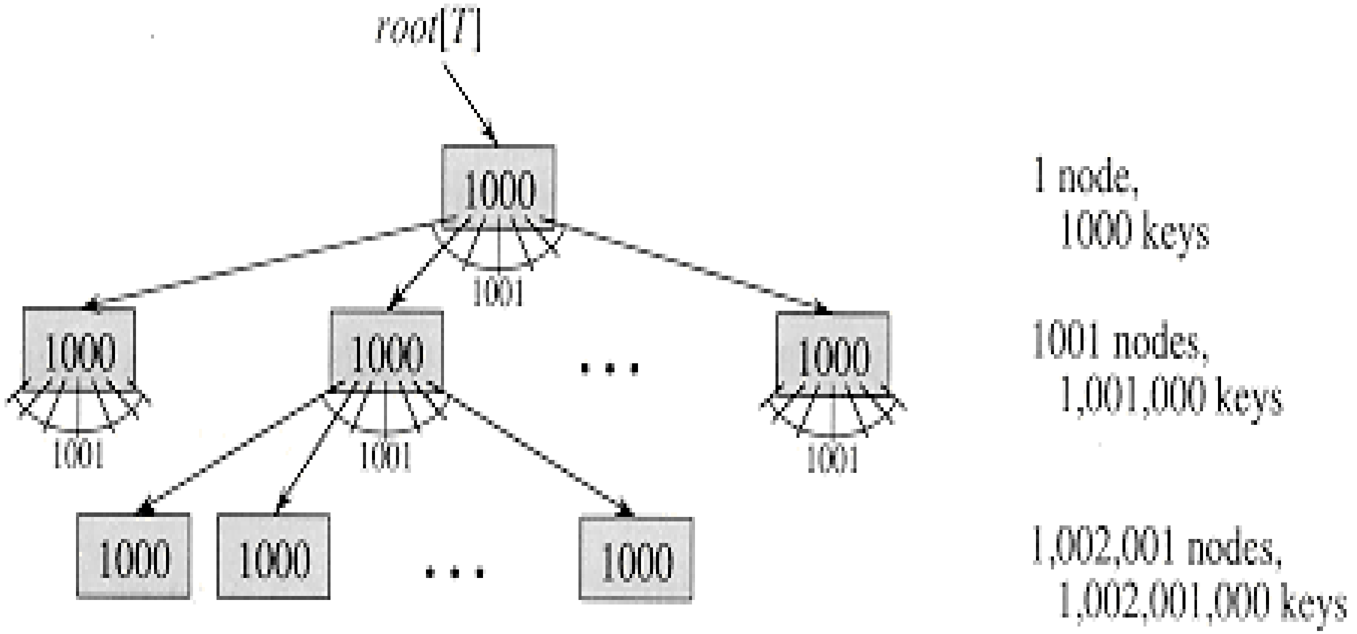
Pseudocode for disk operations-

1) . . .
2) **x a pointer to some object**
3) **DISK-READ(x)**
4) **operations that access and/or modify the fields of x**
5) **DISK-WRITE(x) Omitted if no fields of x were changed.**
6) **other operations that access but do not modify fields of x**
7) **...**

Use of B Tree

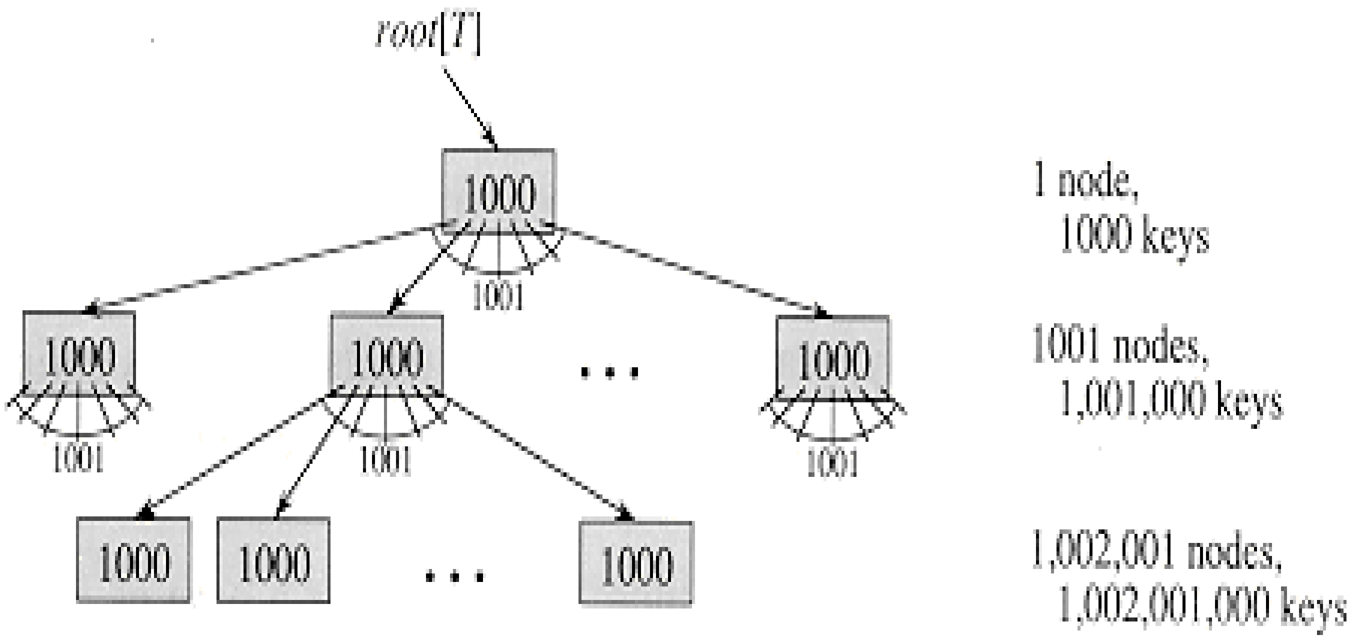
- B-trees are balanced trees that are optimized for situations
  - when part or all of the tree must be maintained in secondary storage such as a magnetic disk.
- Since disk accesses are expensive (time consuming) operations, a b-tree tries to minimize the number of disk accesses.

Use of B Tree



- A B-tree of height 2 containing over one billion keys.
- Each internal node and leaf contains 1000 keys.
- There are 1001 nodes at depth 1 and over one million leaves at depth 2.

Use of B Tree



- A b-tree with a **height of 2 and a branching factor of 1001 can store over one billion keys**
- **But requires at most two disk accesses to search for any node**

.