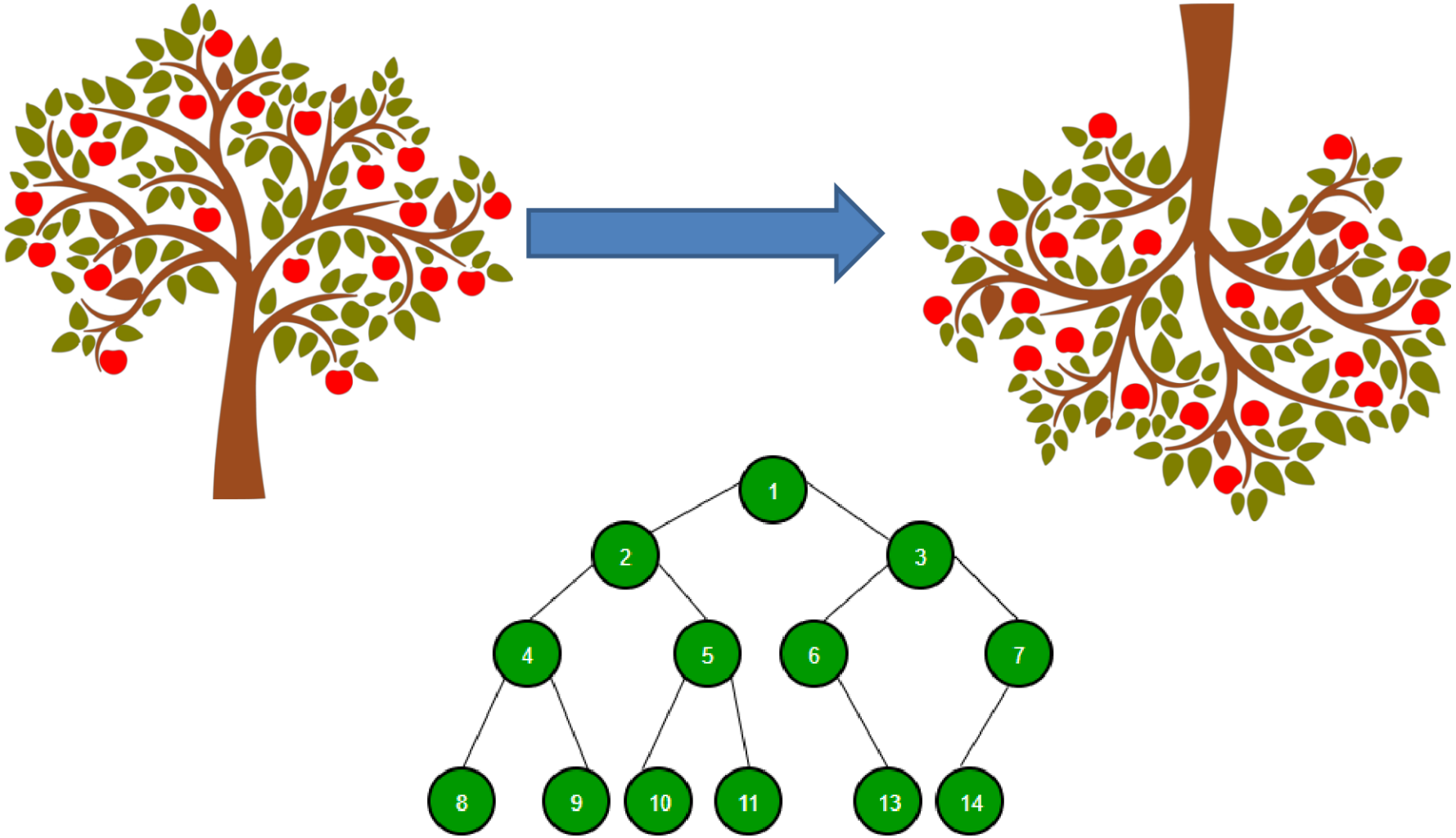
MCSE501L/Data Structures and Algorithms
Module 4: Trees

Syllabus: Module 4

Binary trees- Properties of Binary trees, B-tree, B-Tree definition- Operations on B-Tree: Searching a B-tree, Creating, Splitting, Inserting and Deleting, B+-tree.

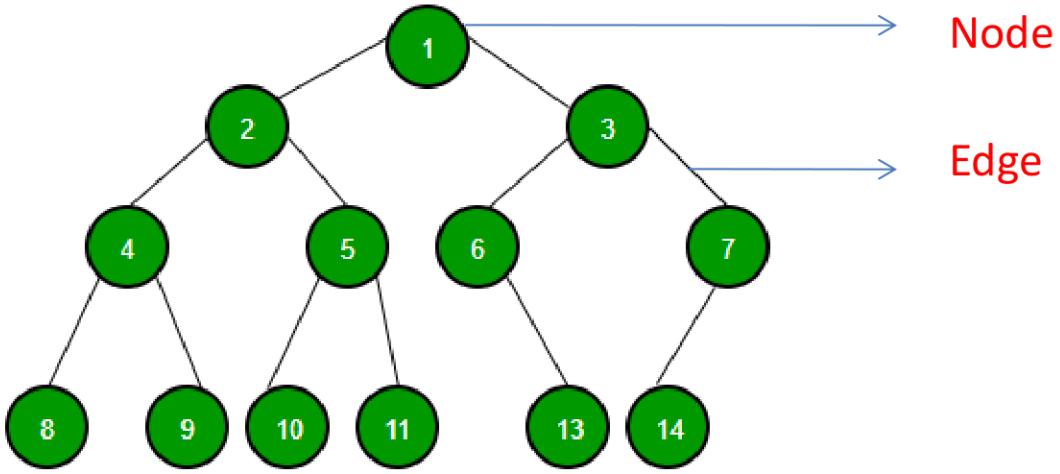
Real Tree vs. DS Tree



Tree

- ADT
- Non-Linear data structure
- Faster than linear data structures
- More natural fit for Hierarchical data

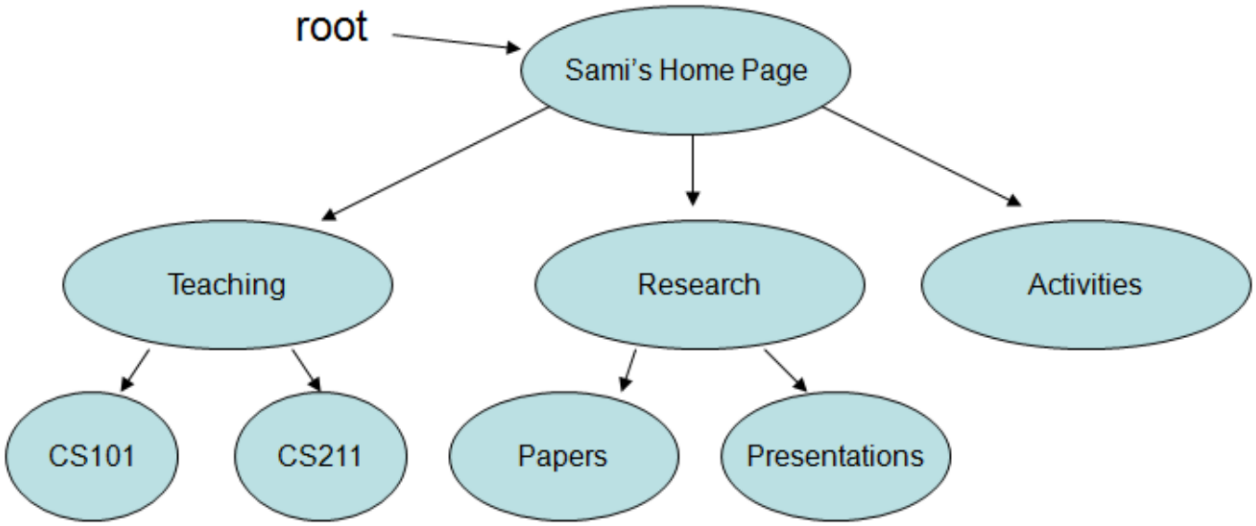
Tree



- Tree consists of collection of nodes and Edges

  T=(N,E)
- Node is represented in circle with data/key
- Edge is represented by a link between two nodes

Example Tree



A web page has been modeled as a Tree data structure

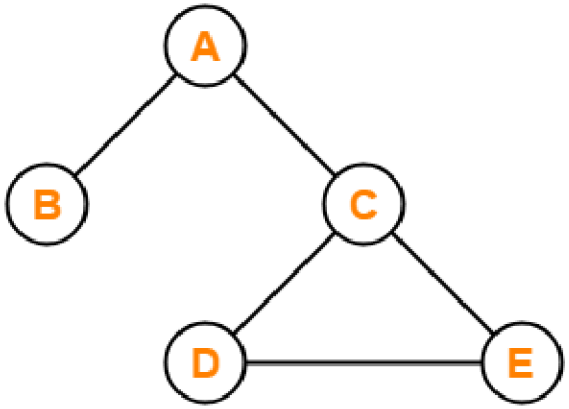
Why Tree Data Structure?

- Other data structures such as arrays, linked list, stack, and queue are linear data structures that store data sequentially.
- In order to perform any operation in a linear data structure, the time complexity increases with the increase in the data size.
- But, it is not acceptable in today's computational world.
- Different tree data structures allow quicker and easier access to the data as it is a non-linear data structure.

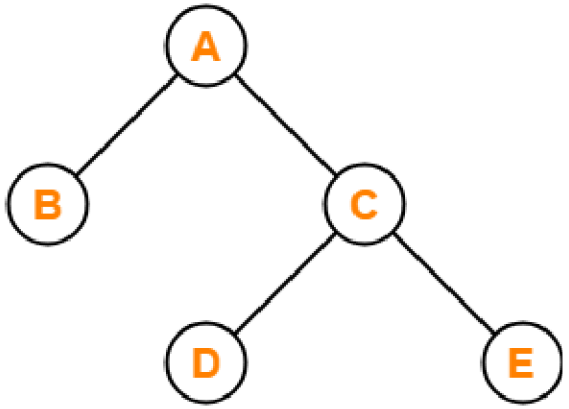
Tree Applications

- Represent organization
- Represent computer file systems (to store information that naturally forms a hierarchy)
- Networks to find best path in the Internet
- Encoding
- Decision tree
- Chemical formulas representation (in Non-CS field too)

Tree vs. Graph

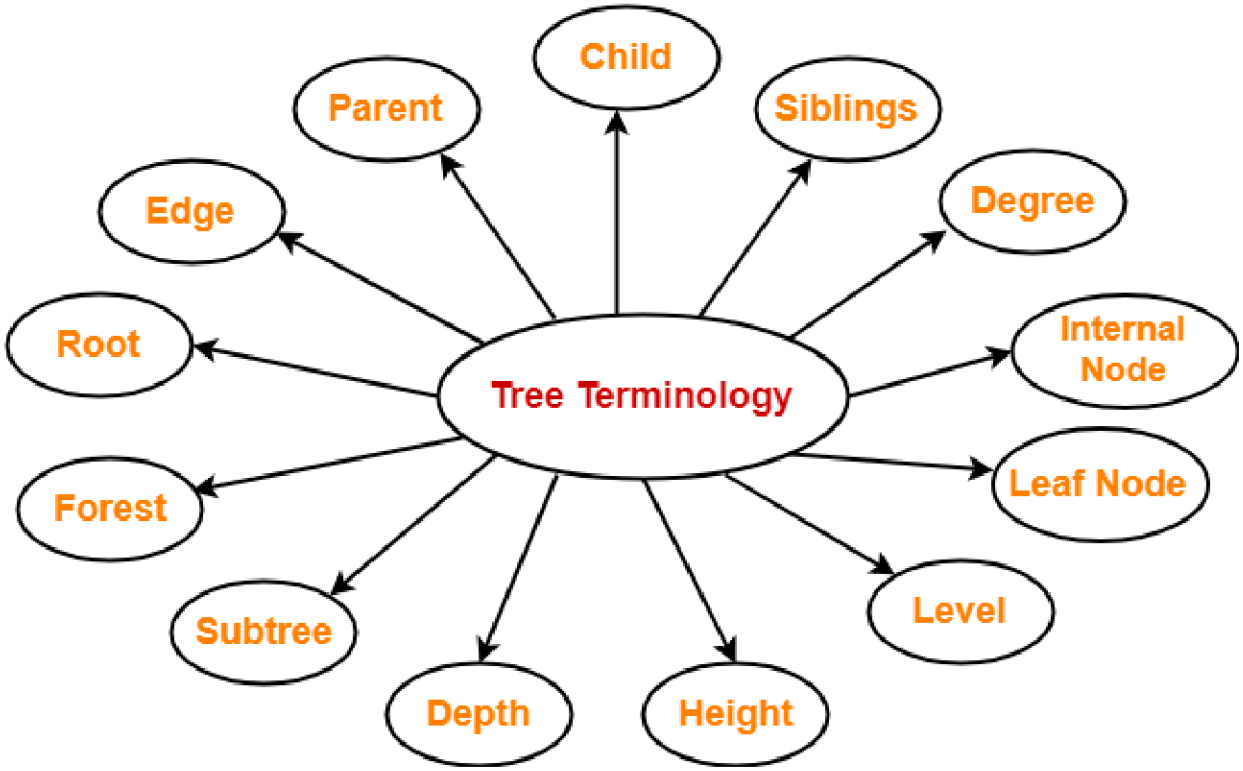


This graph is not a Tree



This graph is a Tree

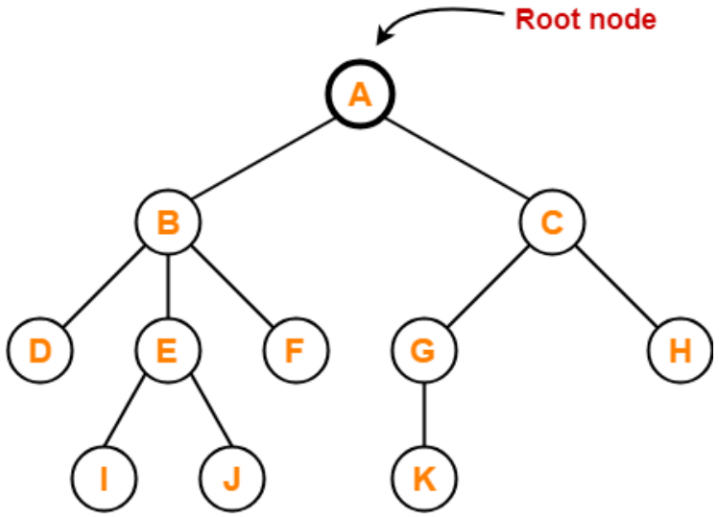
Tree terminology



Tree terminology

Root-

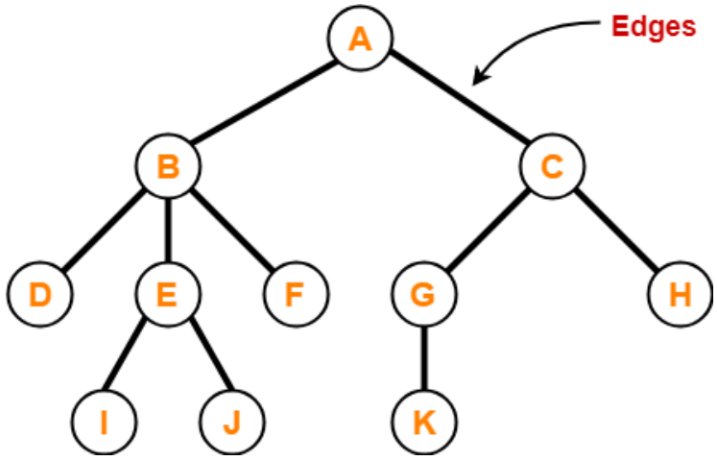
- The first node from where the tree originates is called as a **root node**.
- In any tree, there must be only one root node.



Tree terminology

Edge-

- The connecting link between any two nodes is called as an **edge**.



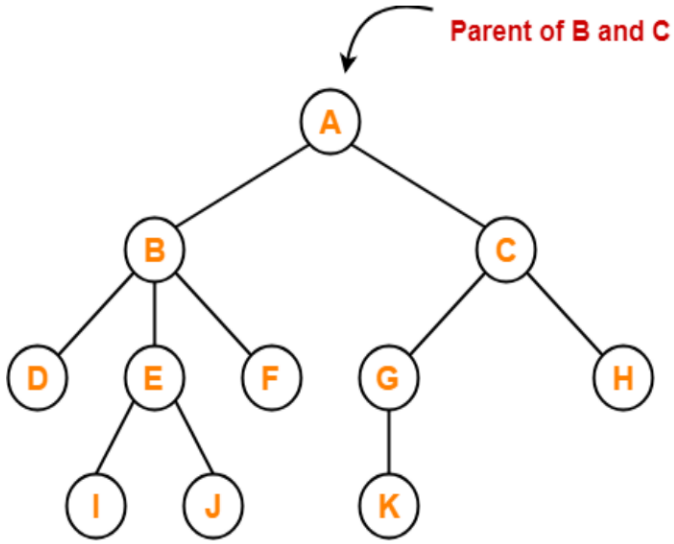
Tree terminology

Parent-

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has one or more children is called as a parent node.
- In a tree, a parent node can have any number of child nodes.

Here,

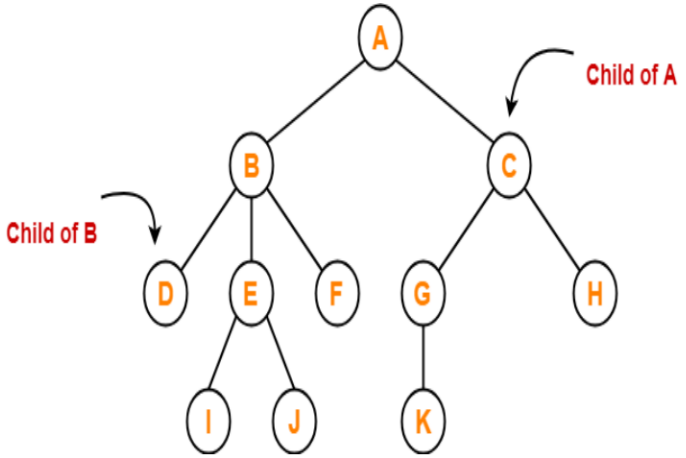
- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K



Tree terminology

Child-

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.



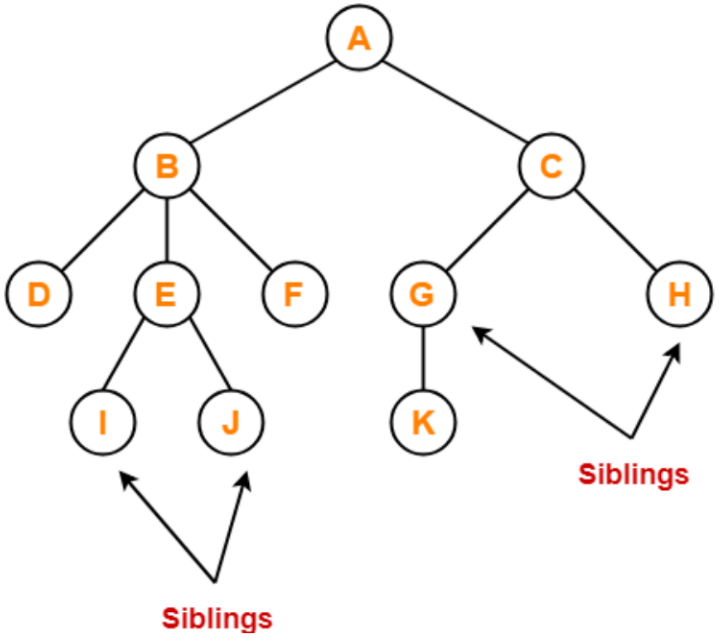
Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

Tree terminology

Siblings-

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.



Here,

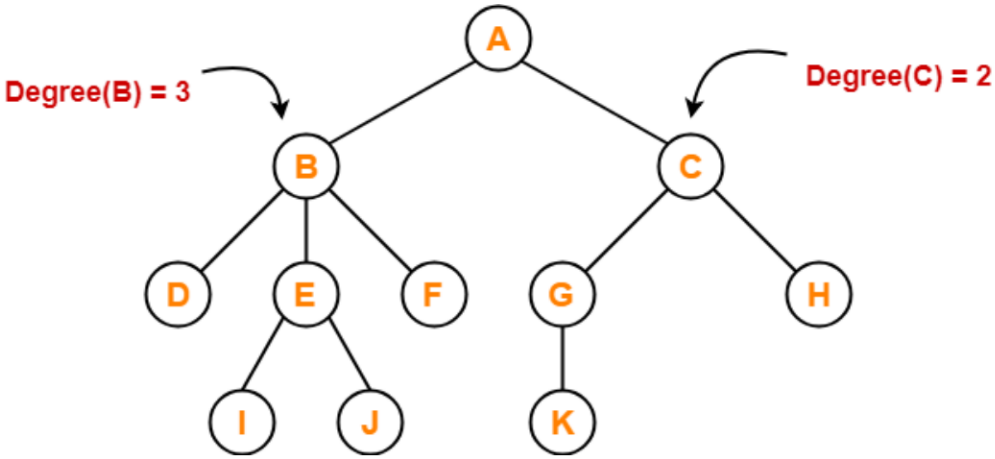
- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

Tree terminology

Degree-

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

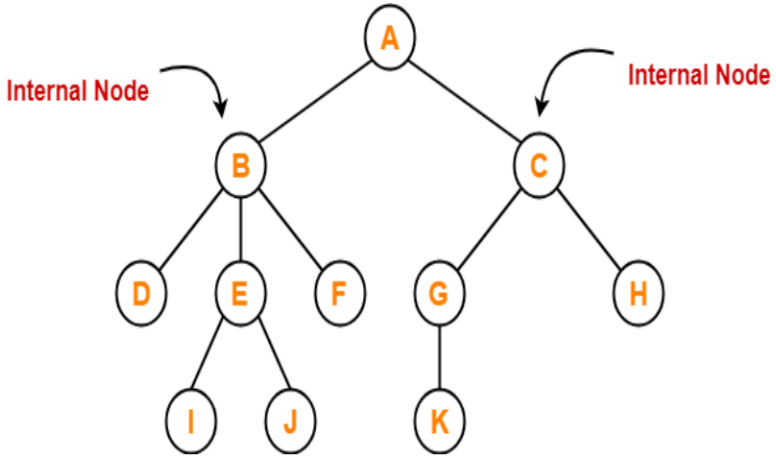
- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0



Tree terminology

Internal Node-(intermediate nodes)

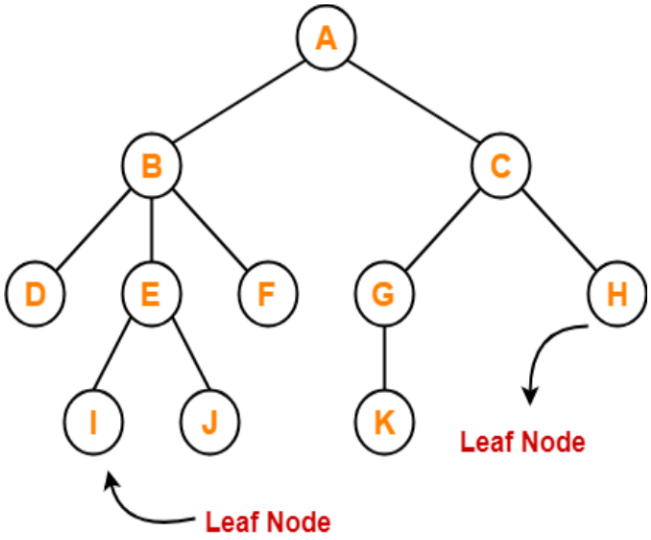
- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.
- Here, nodes A, B, C, E and G are internal nodes.



Tree terminology

Leaf Node-

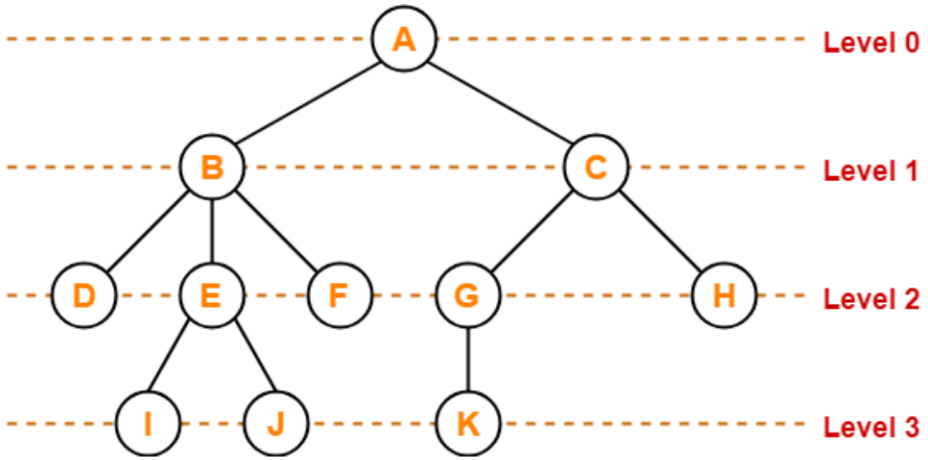
- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.
- Here, nodes D, I, J, F, K and H are leaf nodes.



Tree terminology

Level-

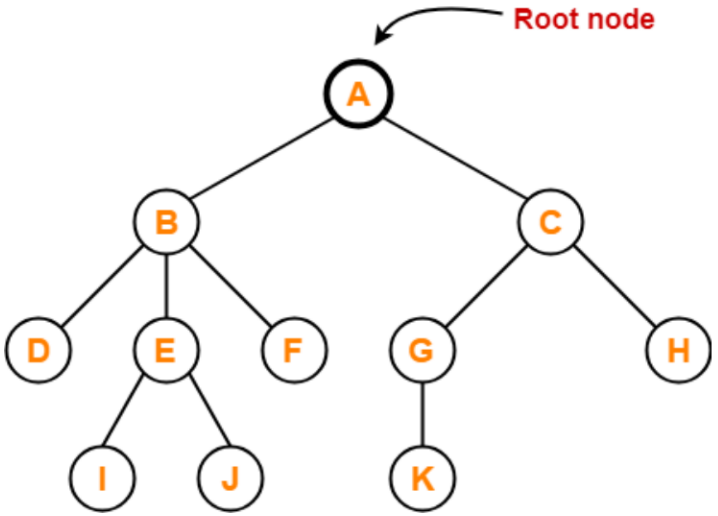
- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.



Tree terminology

Path-

- Sequence of nodes along the edges of a tree
- A,B,D forms a path

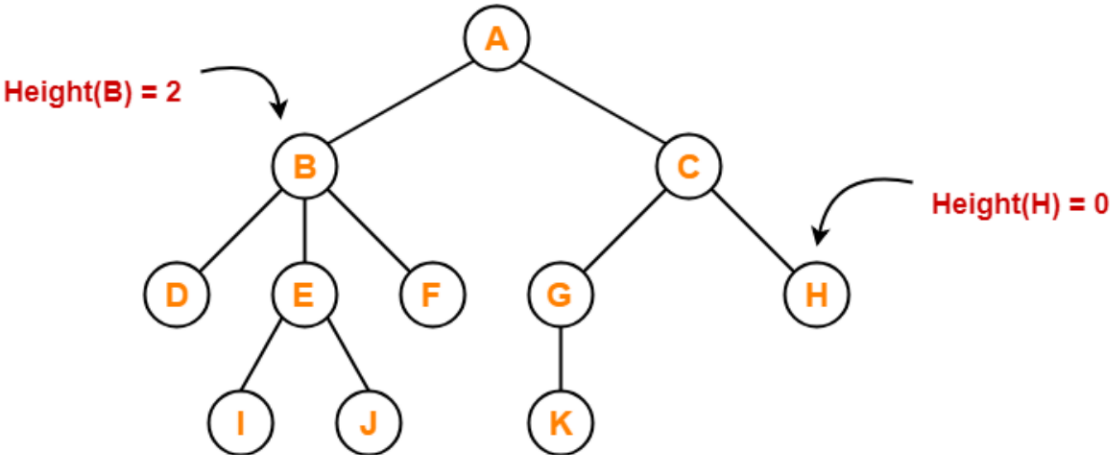


Tree terminology

Height-

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- (Bottom to Top)

- Height of all leaf nodes = 0
- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

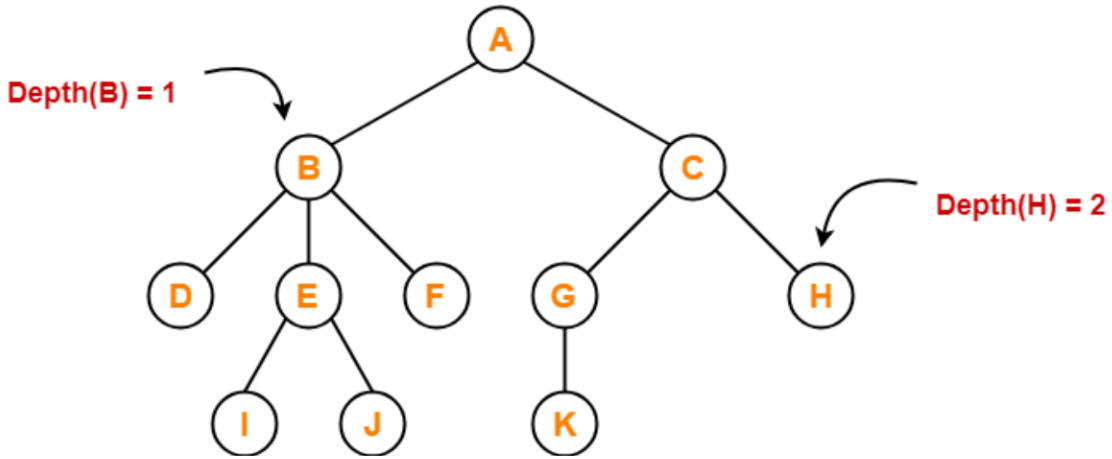


Tree terminology

Depth-

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- (Top to bottom)

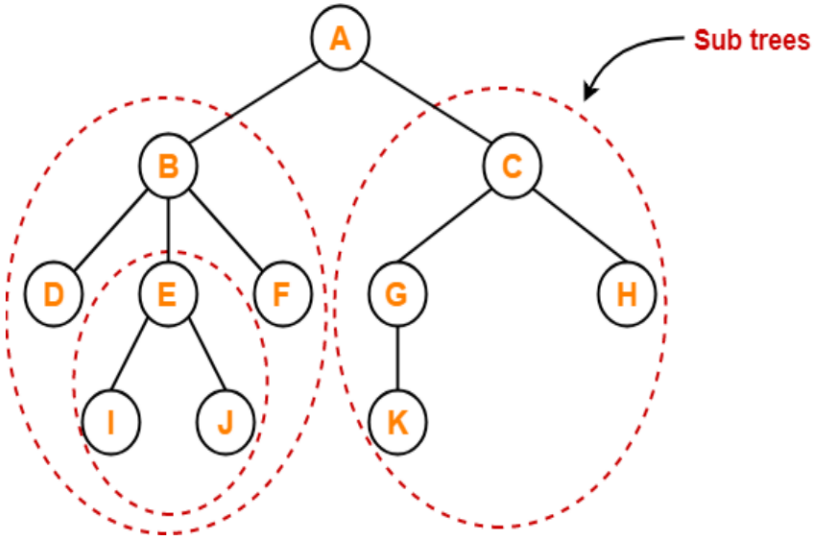
- Depth of the root node = 0
- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3



Tree terminology

Subtree-

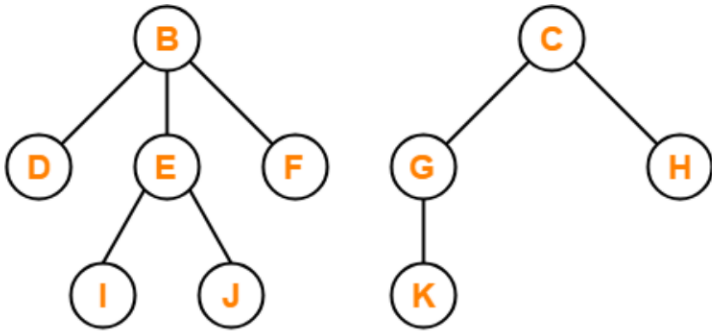
- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.



Tree terminology

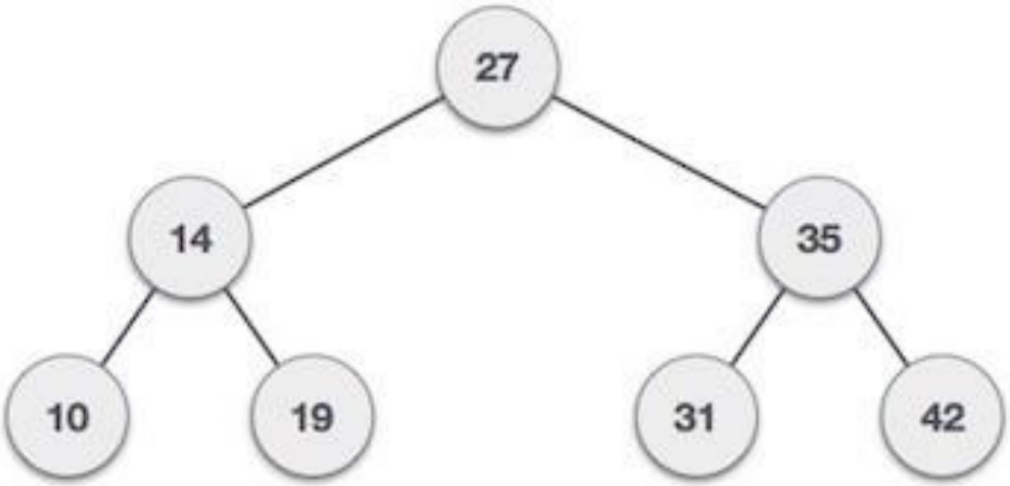
Forest-

- A forest is a set of disjoint trees.

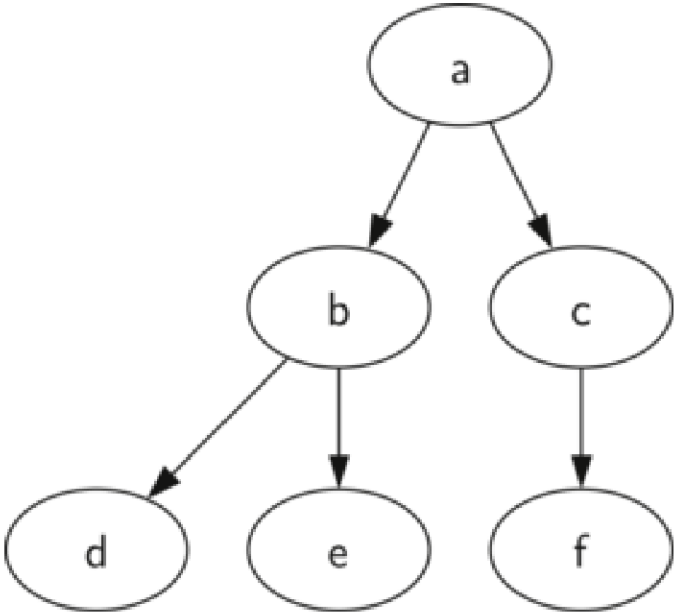


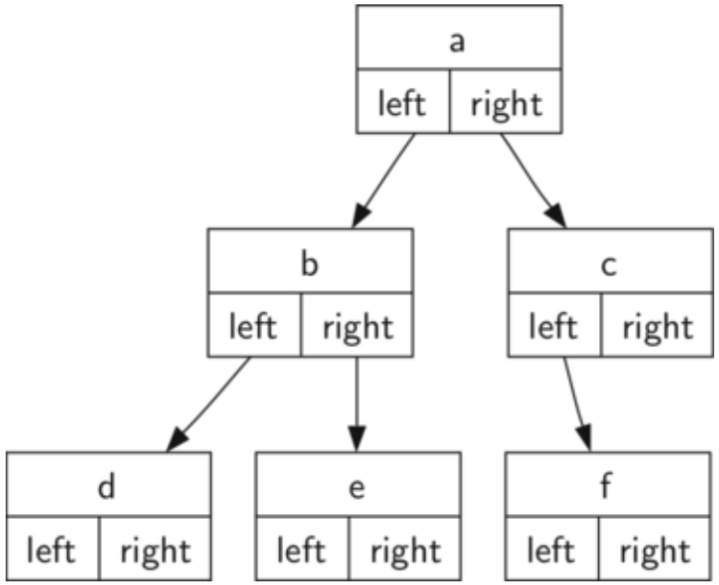
Forest

Tree Terminology-Exercise



Representation of Trees






```
struct node
{
int data;
struct node* left;
struct node* right;
 }
```

Tree Applications

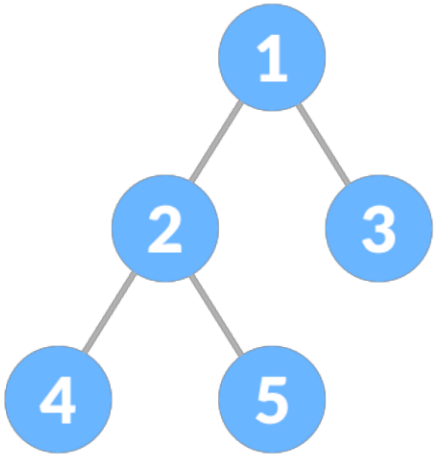
- Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
- Heap is a kind of tree that is used for heap sort.
- A modified version of a tree called Tries is used in modern routers to store routing information.
- Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data
- Compilers use a syntax tree to validate the syntax of every program you write.

Types of Tree

- Binary Tree
- Binary Search Tree
- AVL Tree
- B-Tree
- B+ Tree

Binary Trees

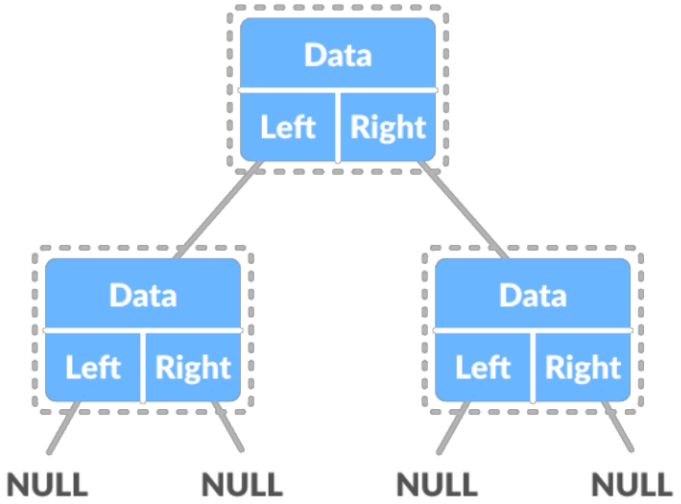
- A binary tree is a tree data structure in which each parent node can have at most two children.



Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

```
struct node
{ int data;
 struct node *left;
struct node *right;
 };
```



Binary Tree Representation

- Array Representation of Binary Tree (sequential representation)
- Linked List Representation of Binary Tree (Dynamic node representation)

Binary Tree Representation- Array

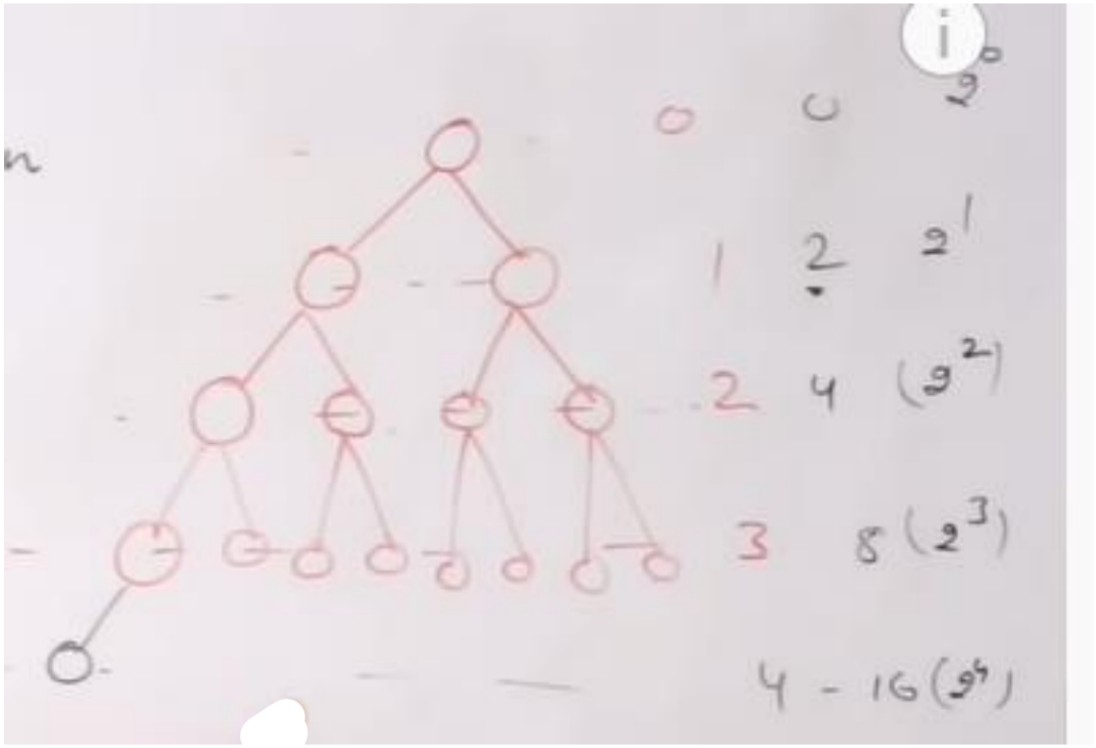
- Array Representation of Binary Tree
- In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

Binary Tree Representation- Array

- For eg: If the array size is a[100]
- Root at a[0]
- If a node occupies a[i]
  - Left child is in a[2i+1]
  - Right child is in a[2i+2]
  - Parent is in a[(i-1)/2]

Binary Tree Representation- Array

- The maximum number of nodes present in a level i of a binary tree is 2^i



Binary Tree Representation- Array

- The maximum number of nodes present in a tree of height h of a binary tree is $2^{h+1}-1$
- Eg: height h

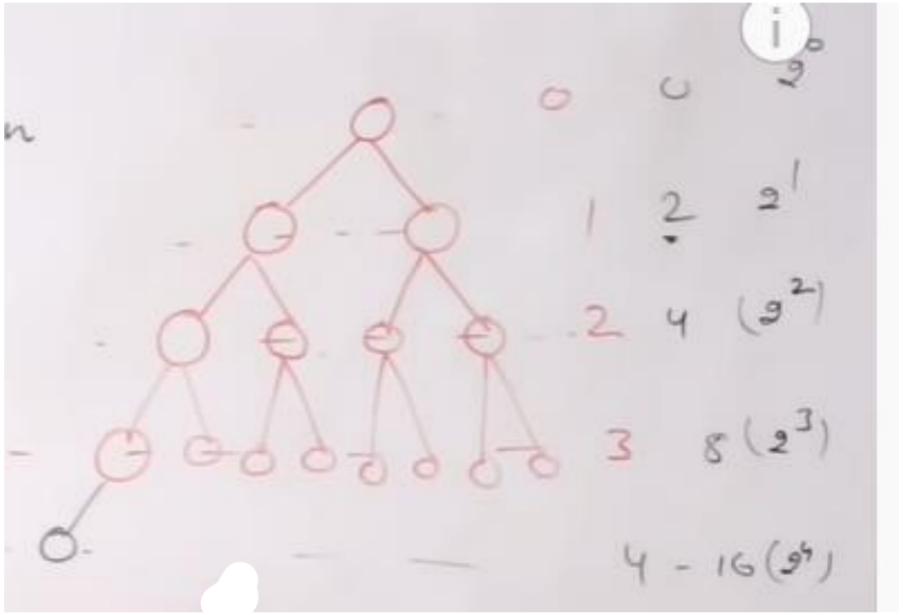
$=2^0+2^1+2^2+.....2^h$

- Eg: height h=2

$=2^{2+1}-1$

$=2^3-1$

$=7$



Binary Tree Representation- Array

- The minimum number of nodes present in a tree of height h of a binary tree is h+1
- Eg: height h

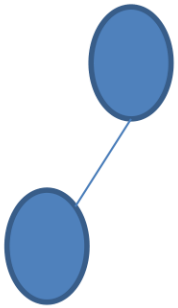
= h+1

- Eg: height h=0

Min.no of nodes=1

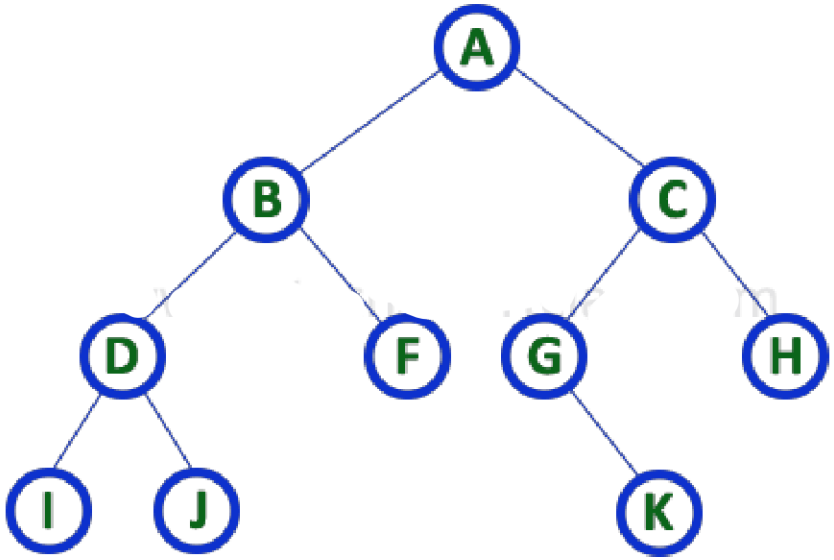
- Eg: height h=1

Min.no of nodes=h+1=>1+1=2



Binary Tree Representation-Array

- Array Representation of Binary Tree



A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

Binary Tree Representation-Linked List

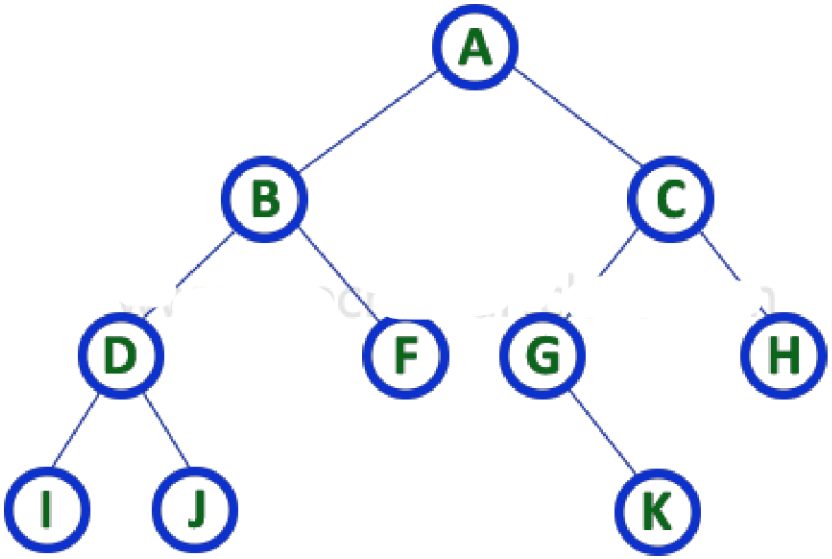
- Linked List Representation of Binary Tree
- We use a doubly linked list to represent a binary tree.
- In a doubly linked list, every node consists of three fields.
- First field for storing left child address, second for storing actual data and third for storing right child address.

Binary Tree Representation-Linked List

- In this linked list representation, a node has the following structure...

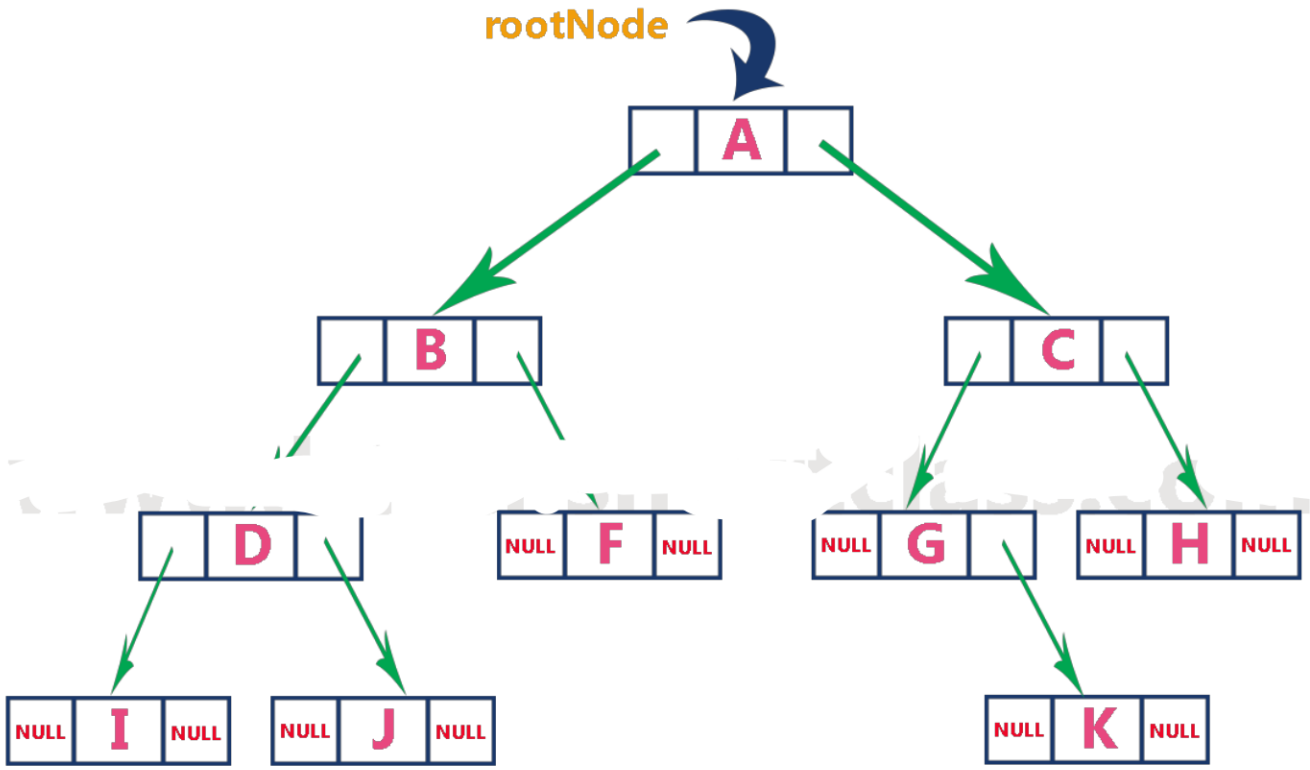
Binary Tree Representation

Example:



In a binary tree with n number of nodes, there are exactly (n-1) number of edges.

Binary Tree Representation- Linked List

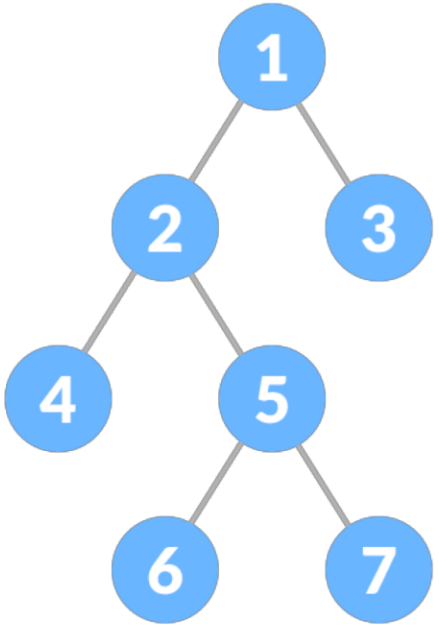


Types of Binary Trees

- Full Binary Tree
- Complete Binary Tree
- Perfect Binary Tree

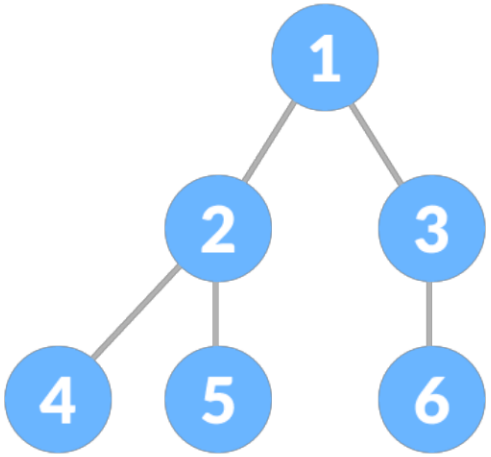
Types of Binary Trees

- Full Binary Tree
- A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. (ie.,no single child)
- Every node other than leaf has two children



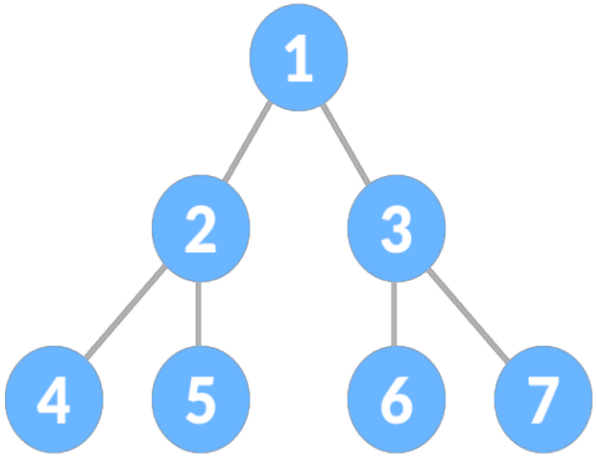
Types of Binary Trees

- **Complete Binary Tree**
- A complete binary tree is just like a full binary tree, but with two major differences
- Every level (except the last)must be completely filled
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



Types of Binary Trees

- **Perfect Binary Tree**
- A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.
- Full+complete binary tree



Binary Tree Applications

- For easy and quick access to data
- In router algorithms
- To implement heap data structure
- Syntax tree

Binary Tree Traversal

- Traversing a tree means visiting every node in the tree.
- Eg: you want to add all the values in the tree or find the largest one.
- For all these operations, you will need to visit each node of the tree.
- Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data.
- But a hierarchical data structure like a tree can be traversed in different ways.

Binary Tree Traversal Types

Depending on the order in which we do this, there can be three types of traversal.

- Inorder
- Preorder
- Postorder

Binary Tree Traversal Types

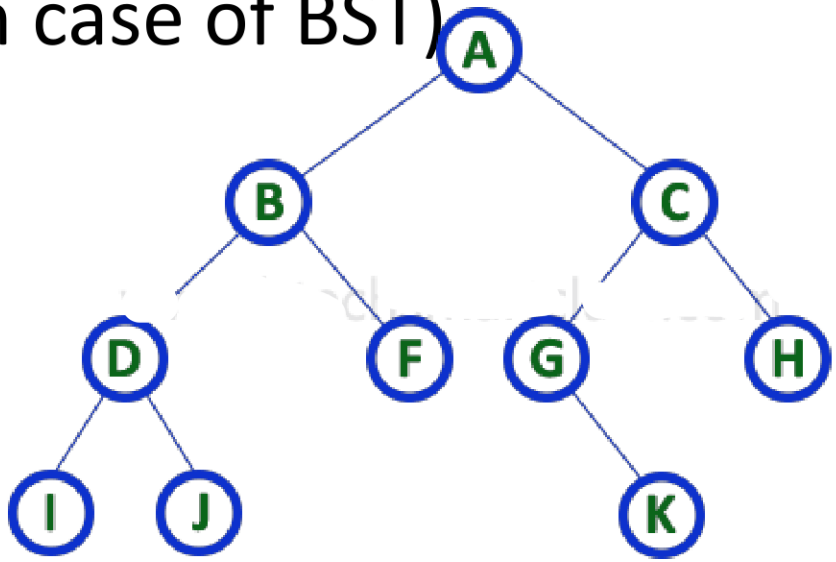
- Inorder - Traverse the left sub-tree first, and then traverse the root and the right sub-tree respectively.
- Preorder - Traverse the root first then traverse into the left sub-tree and right sub-tree respectively.
- Postorder - Traverse the left sub-tree and then traverse the right sub-tree and root respectively.

Binary Tree Traversal Types

- Inorder – left –root- right
- Preorder - root- left -right
- Postorder – left-right-root

Binary Tree Traversal

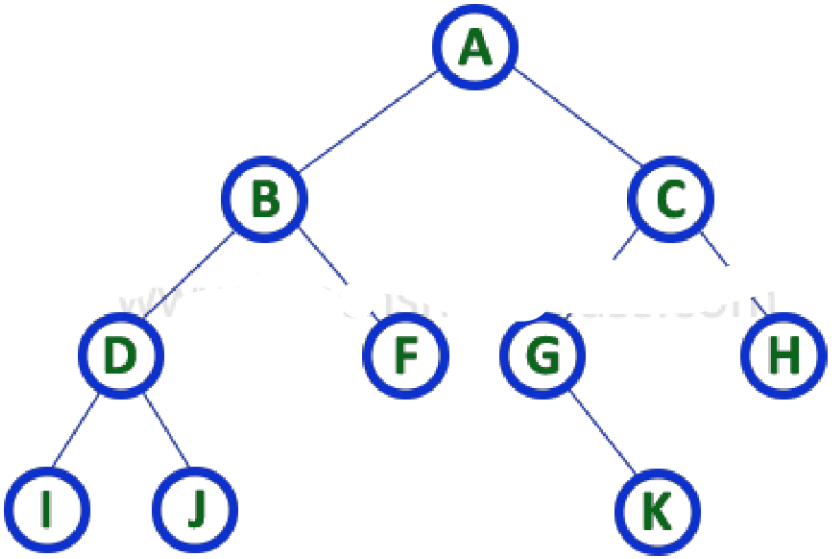
Example: Inorder (always displays in increasing order in case of BST)



I - D - J - B - F - A - G - K - C - H

Binary Tree Traversal

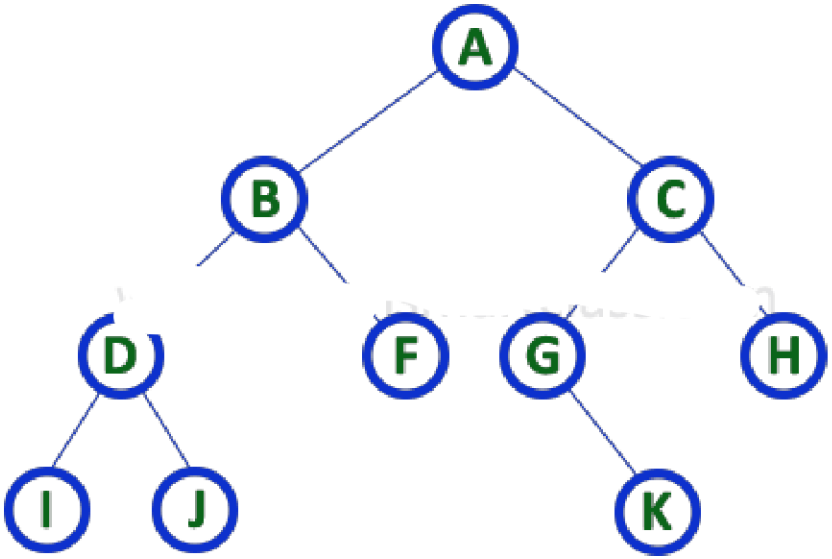
Example: Preorder



A - B - D - I - J - F - C - G - K - H

Binary Tree Traversal

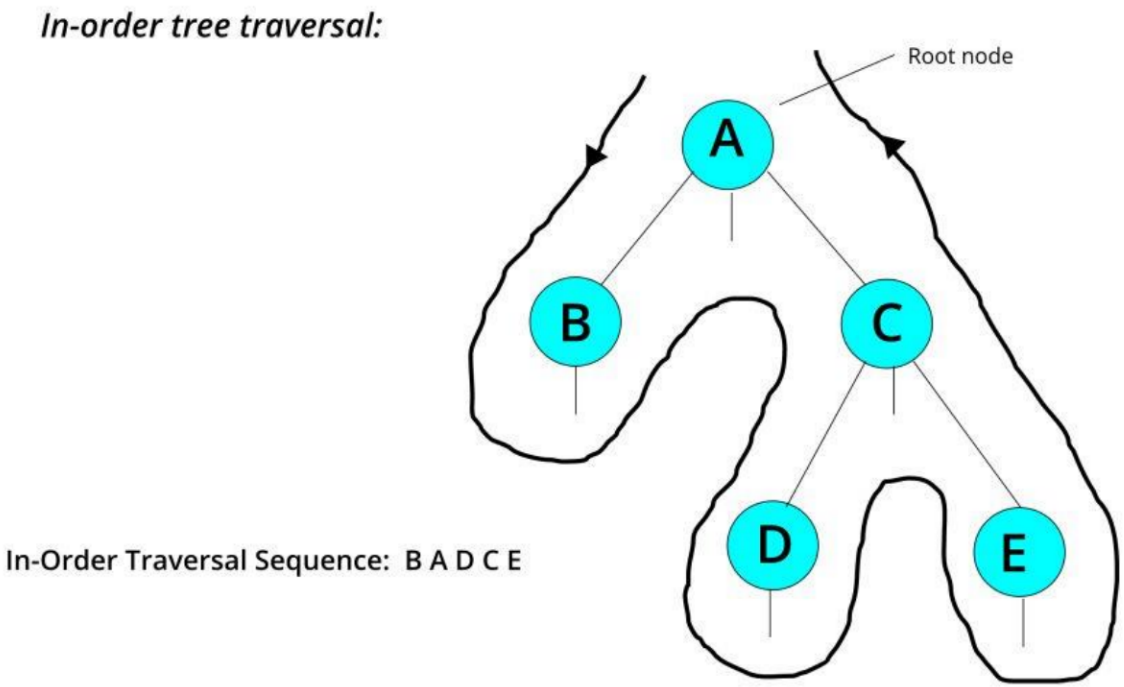
Example: Postorder



I - J - D - F - B - K - G - H - C - A

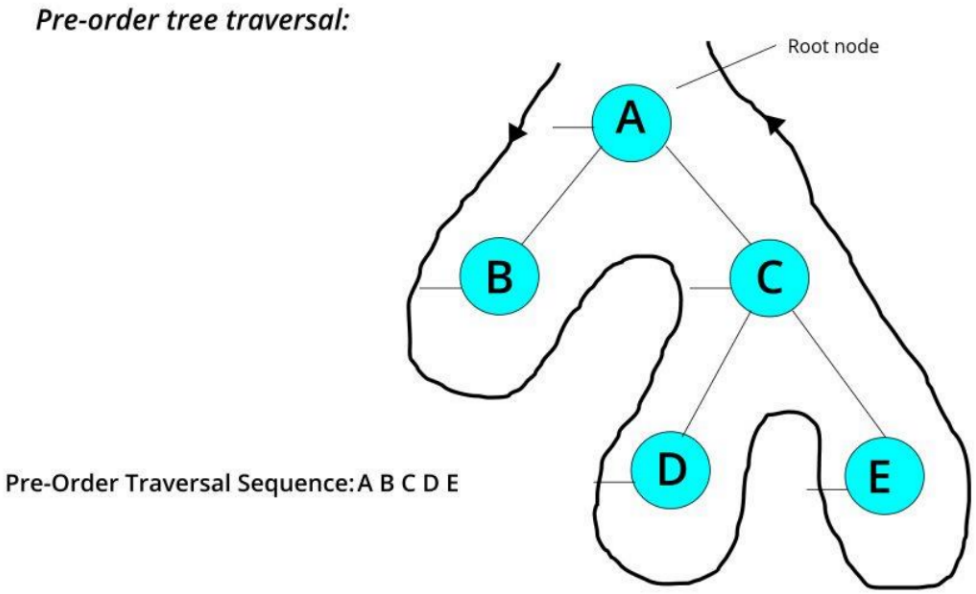
Binary Tree Traversal (Inorder trick)

‘In’ means ‘down’ of the node. So, put the linedown of each node. And, traverse it from root to end.



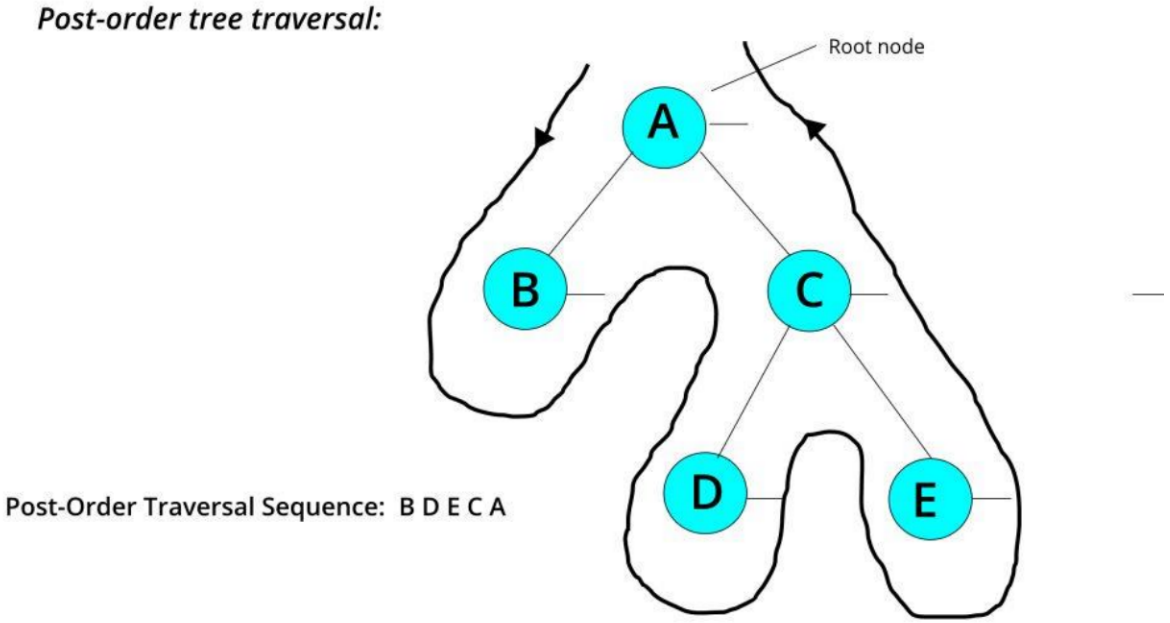
Binary Tree Traversal-Preorder trick

'Pre' means 'before i.e. left' of the node. So, put the line on the left side of each node. And, traverse it from root to end.

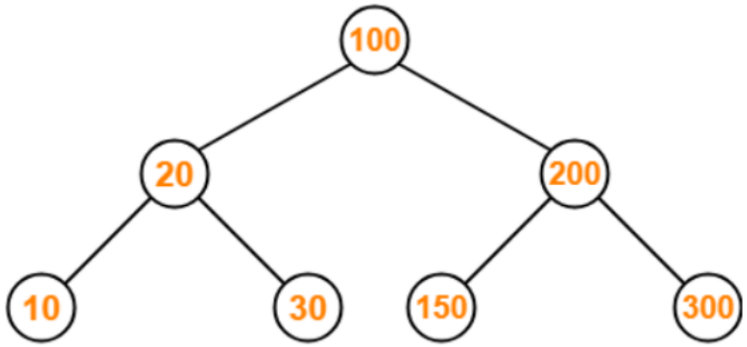


Binary Tree Traversal-post order trick

‘Post’ means ‘after i.e. right’ of the node. So, put the line on the right side of each node. And, traverse it from root to end.



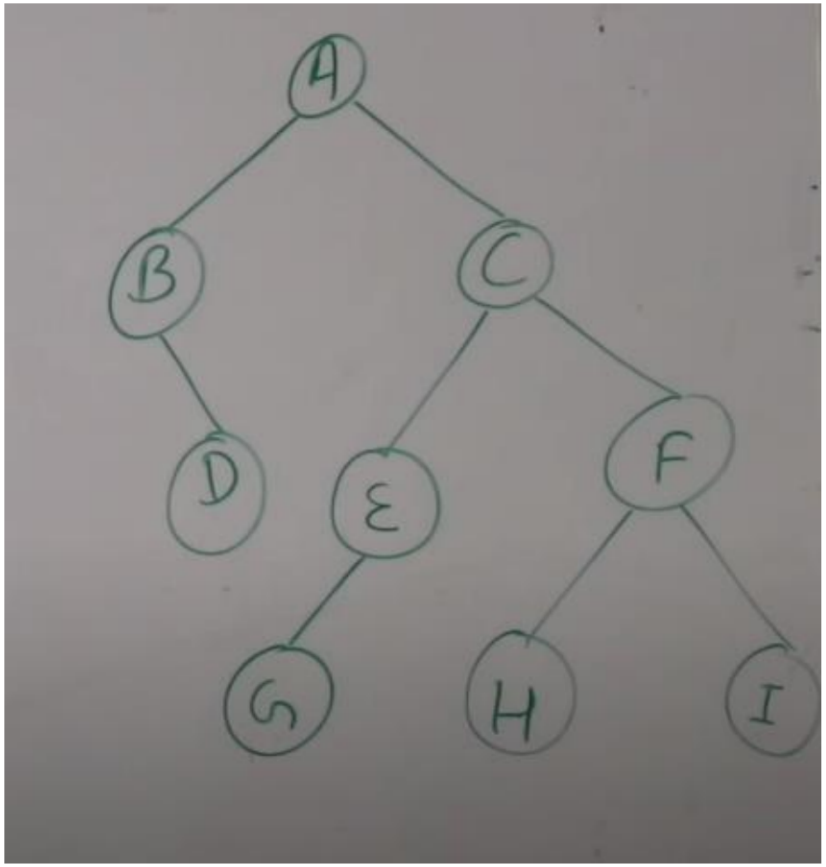
Exercise- Binary Tree Traversal



Answer

- Inorder : 10 , 20 , 30 , 100 , 150 , 200 , 300
- Preorder : 100 , 20 , 10 , 30 , 200 , 150 , 300
- Post order : 10 , 30 , 20 , 150 , 300 , 200 , 100

Exercise- Binary Tree Traversal

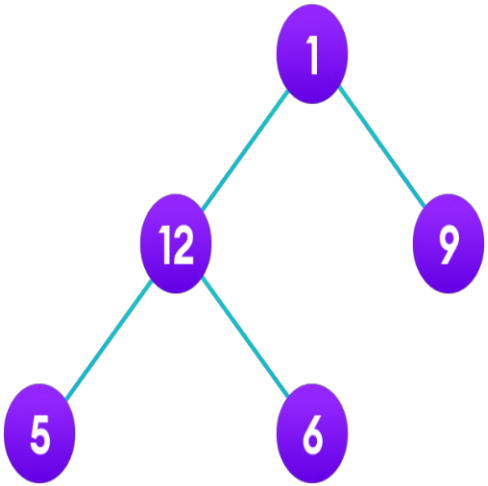


Answer

- Inorder : BDA GEC HFI
- Preorder : ABD CEG FHI
- Post order : DB GE HI FC A

Binary Tree Traversal

Example:



Binary Tree creation

Creation of Binary Tree Using Recursion

A binary tree can be created recursively. The program will work as follow:

- Read a data in x.
- Allocate memory for a new node and store the address in pointer p.
- Store the data x in the node p.
- Recursively create the left subtree of p and make it the left child of p.
- Recursively create the right subtree of p and make it the right child of p.

Binary Tree Creation

```
struct Node
{
int data;
struct Node *left;
struct Node *right;
 };
```

Binary Tree Creation

```
#include<stdio.h>
#include<malloc.h>
 struct node
{
int data;
struct node *left;
struct node *right;
};
```

```
struct node * create()
{
int x;
struct node *newnode;
newnode=(struct node *) malloc(sizeof(struct node));
printf("enter data (-1 for no node)");
scanf("%d",&x);
if(x==-1)
  return 0;
newnode->data=x;
printf("\n enter left child of %d", x);
newnode->left=create();
printf("\n enter right child of %d", x);
newnode->right=create();
return newnode;
}
```

Binary Tree Creation

```
void preorder(struct node *root)
{
if (root==0)
{
return;
}
printf("%d  ",root->data);
preorder(root->left);
preorder(root->right);
}
```

```
void main()
{
struct node *root;
root=0;
root=create();
printf("\n the preorder traversal is");
preorder(root);
}
```

Binary Tree display (inorder)

```
void display(struct Node *root)
{
if(root != NULL)
{
display(root->left);
printf("%d",root->data);
display(root->right);
}
```

Binary Tree display (preorder)

```
void display(struct Node *root)
{
if(root != NULL)
{
printf("%d",root->data);
display(root->left);
display(root->right);
}
```

Binary Tree display (postorder)

```
void display(struct Node *root)
{
if(root != NULL)
{
display(root->left);
display(root->right);
printf("%d",root->data);
}
```

Binary Tree –Another program

```
#include <stdio.h>
#include <stdlib.h>

// Define the structure for a node in the binary tree
struct Node {
    int data;
    struct Node* left;
    struct Node* right;
};

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

```
// Function to insert a node into the binary tree
struct Node* insertNode(struct Node* root, int data) {
    if (root == NULL) {
        return createNode(data);
    }

    char direction;
    printf("Insert %d to the left or right of %d? (l/r): ", data,
root->data);
    scanf(" %c", &direction);

    if (direction == 'l' || direction == 'L') {
        root->left = insertNode(root->left, data);
    } else if (direction == 'r' || direction == 'R') {
        root->right = insertNode(root->right, data);
    } else {
        printf("Invalid input. Node not inserted.\n");
    }
        return root;
}
```

```
// In-order traversal (Left, Root, Right)
void inorderTraversal(struct Node* root) {
    if (root == NULL) return;
    inorderTraversal(root->left);
    printf("%d ", root->data);
    inorderTraversal(root->right);
}
 // Pre-order traversal (Root, Left, Right)
void preorderTraversal(struct Node* root) {
    if (root == NULL) return;
    printf("%d ", root->data);
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}
 // Post-order traversal (Left, Right, Root)
void postorderTraversal(struct Node* root) {
    if (root == NULL) return;
    postorderTraversal(root->left);
    postorderTraversal(root->right);
    printf("%d ", root->data);
}
```

```
// Main function
int main() {
    struct Node* root = NULL;
    int choice, value;

    while (1) {
        printf("\n1. Insert a node\n");
        printf("2. In-order Traversal\n");
        printf("3. Pre-order Traversal\n");
        printf("4. Post-order Traversal\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter value to insert: ");
                scanf("%d", &value);
                root = insertNode(root, value);
                break;
            case 2:
                printf("In-order Traversal: ");
                inorderTraversal(root);
                printf("\n");
                break;
            case 3:
                printf("Pre-order Traversal: ");
                preorderTraversal(root);
                printf("\n");
                break;
            case 4:
                printf("Post-order Traversal: ");
                postorderTraversal(root);
                printf("\n");
                break;
            case 5:
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

Binary Tree –Time Complexity

Binary Tree Creation:

Insertion Time Complexity: $O(h)$

- For each node insertion, the time complexity depends on the height h of the binary tree.
- In the worst case (e.g., a skewed tree where all nodes are either to the left or the right), the height of the tree is O(n), where n is the number of nodes.

Binary Tree –Time Complexity

Binary Tree Traversal:

- **Traversal Time Complexity:** O(n)
  - Whether using in-order, pre-order, or post-order traversal, each node is visited exactly once.
  - Therefore, the time complexity for traversal is O(n), where n is the total number of nodes in the tree.

Binary Tree –Time Complexity

Binary Tree Search:

- **Search Time Complexity:** O(h)
  - Similar to insertion, searching for a node in a binary tree has a time complexity that depends on the height h of the tree.
  - In the worst case, the time complexity is O(n) for an unbalanced tree

Binary Tree –Time Complexity

Binary Tree Deletion:

- **Deletion Time Complexity:** O(h)
  - Deleting a node from a binary tree also has a time complexity that depends on the height h of the tree.
  - In the worst case, the time complexity is O(n) (for a skewed tree)

B-Tree

B-Tree

- **B Tree** is a Balancing Tree , not a Binary Tree
- Balanced m-way tree
- m – means order of a tree (For eg: create a B tree of order m)
- B-Tree was named ***Height Balanced m-way Search Tree***. Later it was named as B-Tree.

B-Tree

- In search trees, like binary search tree and AVL Tree, every node contains only one value (key) and a maximum of two children.
- In B-Tree a node contains more than one value (key) and more than two children which keeps the height of the tree relatively small.
- It allows operations like Insertion, searching, and deletion in less time.
- widely used for disk access

B-Tree

- The number of keys in a node and number of children for a node depends on the order of B-Tree (m).
- The value of “m” depends upon the block size on the disk on which data is primarily located.

B-Tree

Properties:

- All leaves will be at the same level.
- The left subtree of the node will have lesser values than the right side of the subtree. (Generalization of BST)
- Nodes maintain sorted data (in ascending order from left to right)
- Data/key can be present in all the nodes

Rules for B-Tree

- For a B-Tree of order m

Maximum :

- The maximum number of children for any node: m
- The maximum number of keys m-1
- For example:

  m = 4

  max children = 4

  max keys = 4 – 1 = 3

Minimum:

- The minimum children a node can have is half of the order, which is m/2 (the ceiling value is taken).
- Minimum children

  : leaf -> 0

  : root -> 2

  : internal node->m/2 (the ceiling value)

- Every node, except root, must contain minimum keys of [m/2]-1
- Root contains minimum key of 1

For example:

m = 3

min keys = (3/2)-1 = 1.5-1= 2-1=1

B-Tree

The ceil function returns the smallest integer that is greater than or equal to a given number.

Example:

- ceil(4.2)=5
- ceil(−3.7)=−3
- ceil(7.0)=7

B-Tree

Summary for a B-Tree of Order m:

- Minimum Keys per Node: $\lceil \frac{m}{2} \rceil - 1$
- Maximum Keys per Node: $m - 1$
- Minimum Children per Node: $\lceil \frac{m}{2} \rceil$
- Maximum Children per Node: m

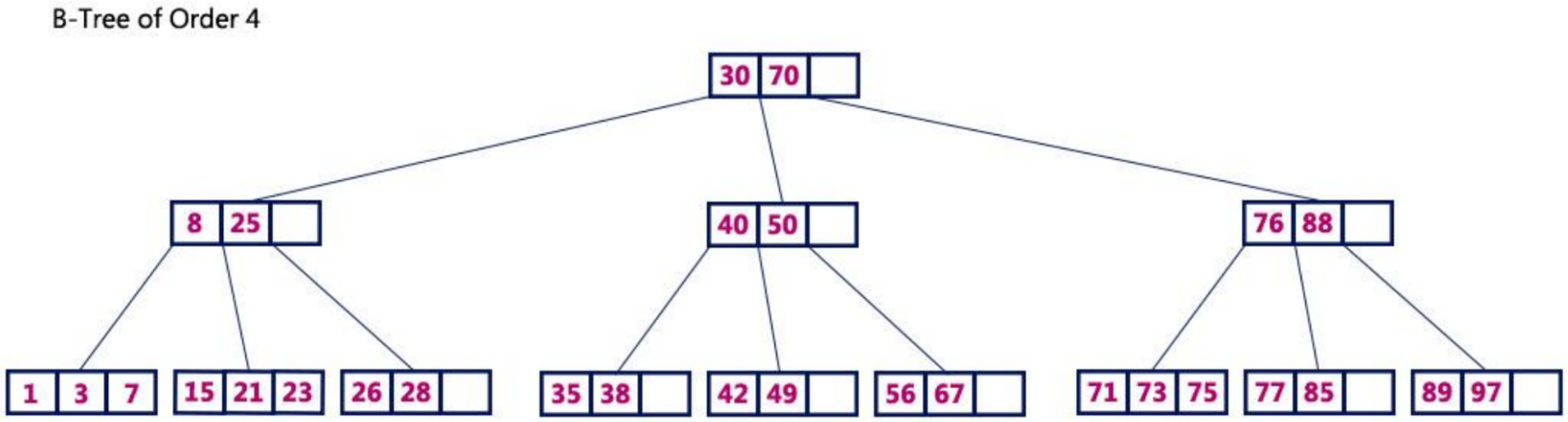
Example:

For a B-Tree of order $m = 5$:

- Minimum Keys per Node: $\lceil \frac{5}{2} \rceil - 1 = 3 - 1 = 2$
- Maximum Keys per Node: $5 - 1 = 4$
- Minimum Children per Node: $\lceil \frac{5}{2} \rceil = 3$
- Maximum Children per Node: 5

Example- B-Tree

- For example, B-Tree of Order 4 (m=4) contains a maximum of 3 key values in a node and maximum of 4 children for a node.



Why use B-Tree

- Data is stored on the disk in blocks, this data, when brought into main memory (or RAM) is called data structure.
- In-case of huge data, searching a record in the disk requires reading the entire disk;
- This increases time and main memory consumption due to high disk access frequency and data size.
- To overcome this, index tables are created that saves the record reference of the records based on the blocks they reside in. This drastically reduces the time and memory consumption.
- Since we have huge data, we can create multi-level index tables.
- Multi-level index can be designed by using B Tree for keeping the data sorted in a self-balancing fashion.

Why use B-Tree

- Reduces the number of reads made on the disk
- It is a useful algorithm for databases and file systems.

B-Tree –Insertion

- In a B-Tree, a new element must be added only at the leaf node. That means, the new keyValue is always attached to the leaf node only.

- **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 Construction

- Construct a **B-Tree of Order 3** (m=3) by inserting numbers from 1 to 10.

B-Tree Construction

insert(1)

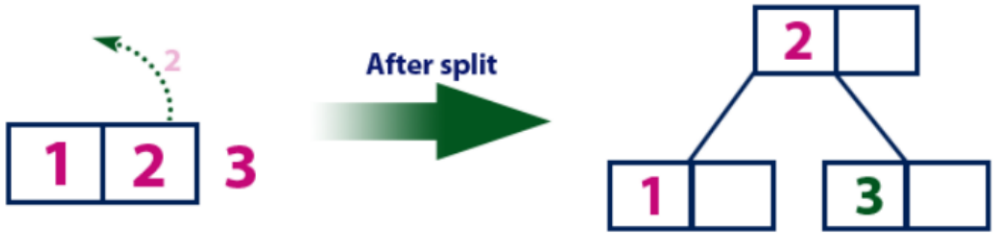
Since '1' is the first element into the tree that is inserted into a new node. It acts as the root node.

insert(2)

Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element (2) can be inserted at that empty position.

insert(3)

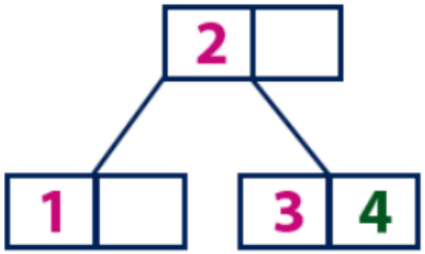
Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't has an empty position. So, we split that node by sending middle value (2) to its parent node. But here, this node doesn't has parent. So, this middle value becomes a new root node for the tree.



B-Tree Construction

insert(4)

Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So, new element (4) can be inserted at that empty position.



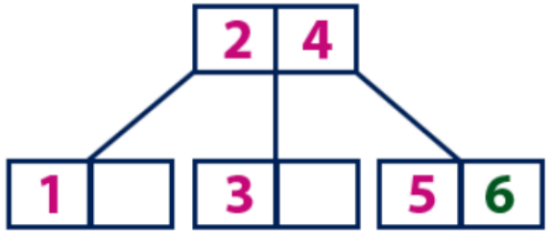
insert(5)

Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (4) to its parent node (2). There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element '5' added as new leaf node.

B-Tree Construction

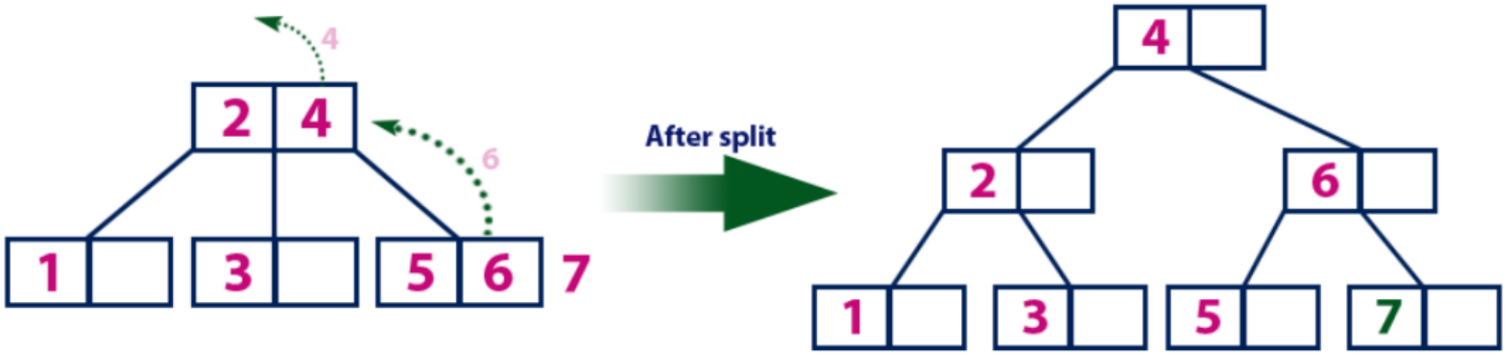
insert(6)

Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element (6) can be inserted at that empty position.



insert(7)

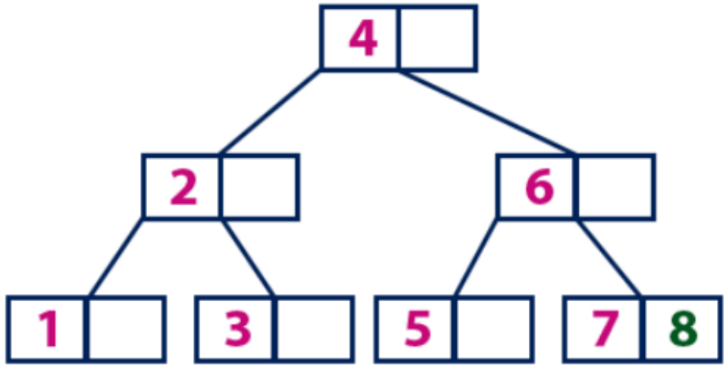
Element '7' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending middle value (6) to its parent node (2&4). But the parent (2&4) is also full. So, again we split the node (2&4) by sending middle value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for the tree.



B-Tree Construction

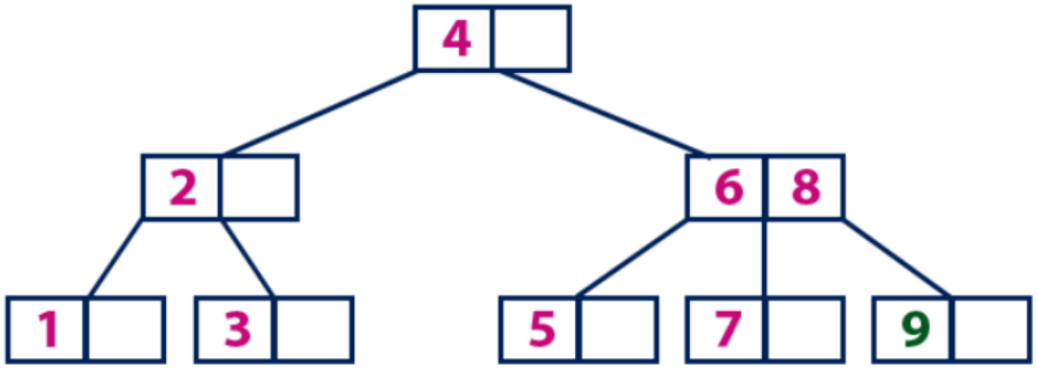
insert(8)

Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '8' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7) and it has an empty position. So, new element (8) can be inserted at that empty position.



insert(9)

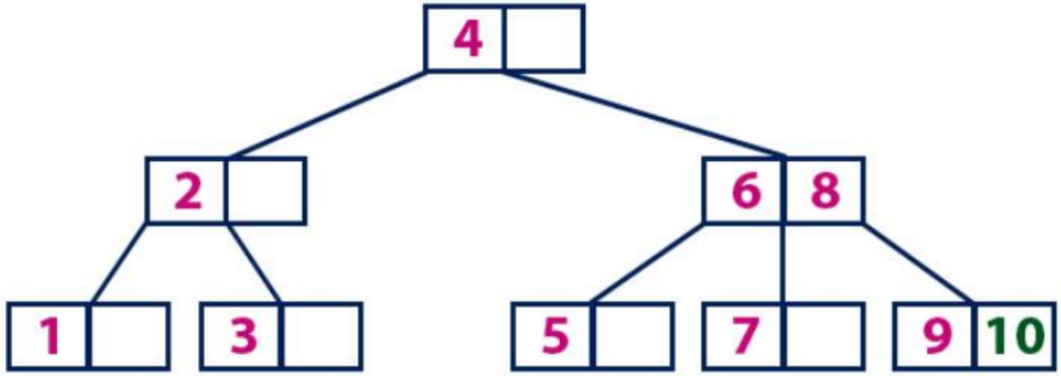
Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. '9' is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node (7 & 8). This leaf node is already full. So, we split this node by sending middle value (8) to its parent node. The parent node (6) has an empty position. So, '8' is added at that position. And new element is added as a new leaf node.



B-Tree Construction

insert(10)

Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with values '6 & 8 '. '10' is larger than '6 & 8' and it is also not a leaf node. So, we move to the right of '8'. We reach to a leaf node (9). This leaf node has an empty position. So, new element '10' is added at that empty position.



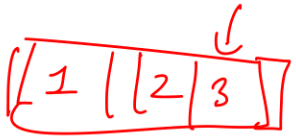
Construct a B-Tree of order m = 3 [max key = 2, max children = 3]

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

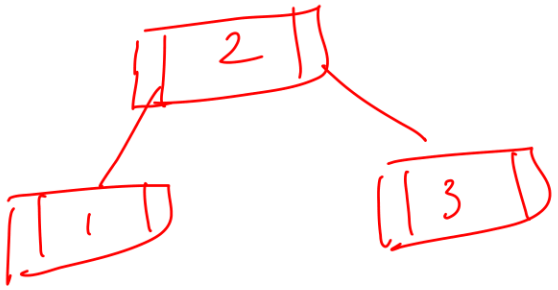
1 ==

2 ==

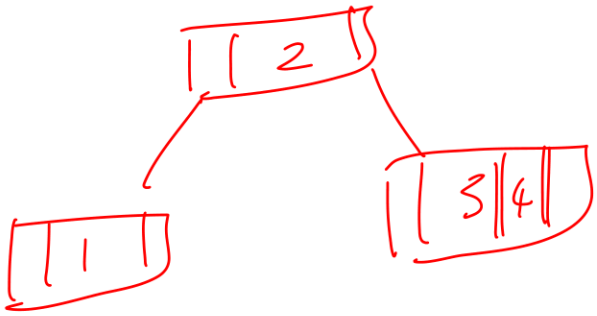
3 ==



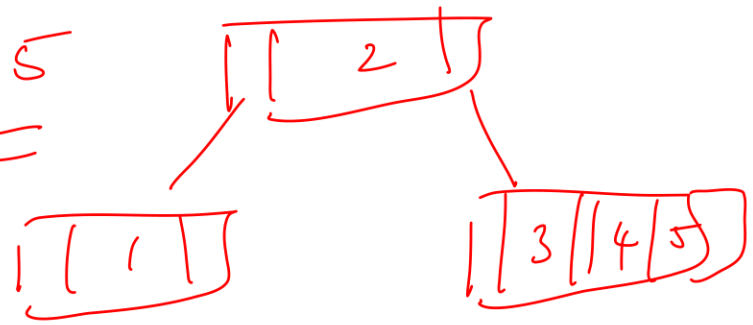
It violates B-Tree property where max key = 2. So split the node into 2, move middle element one level up



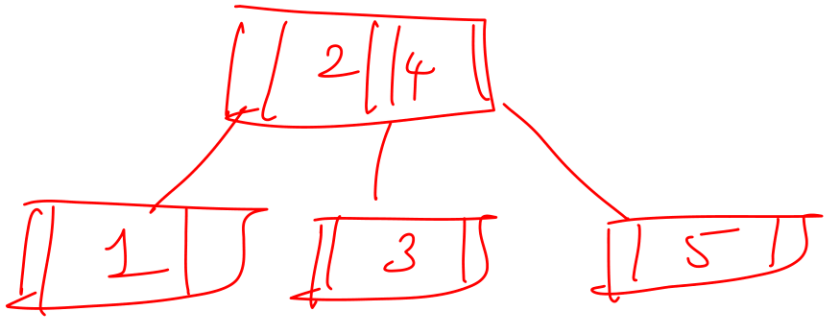
4 ==



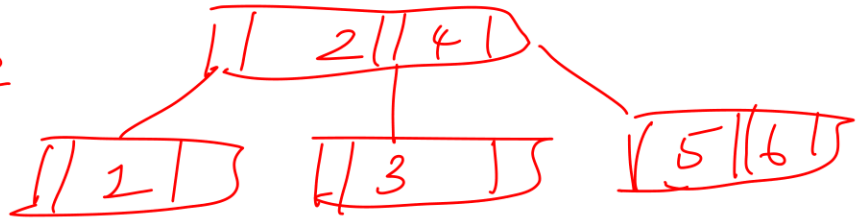
5



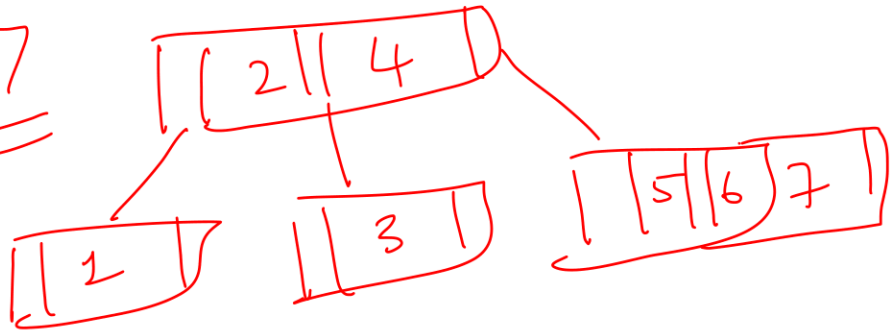
← violates B-tree property



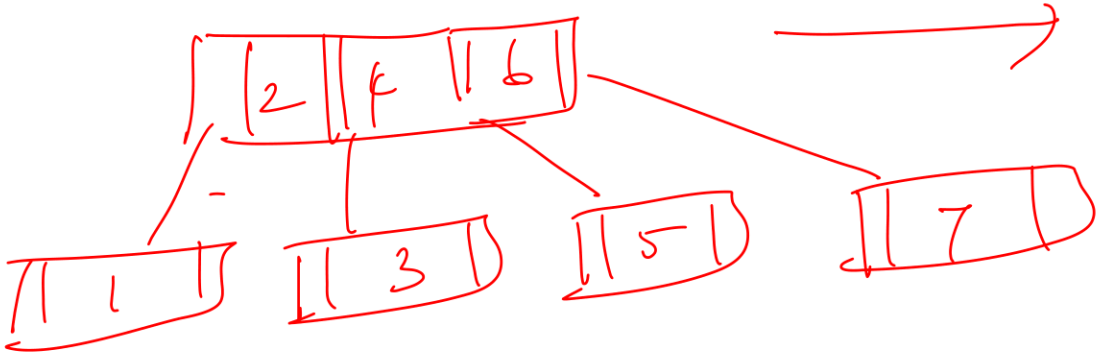
6



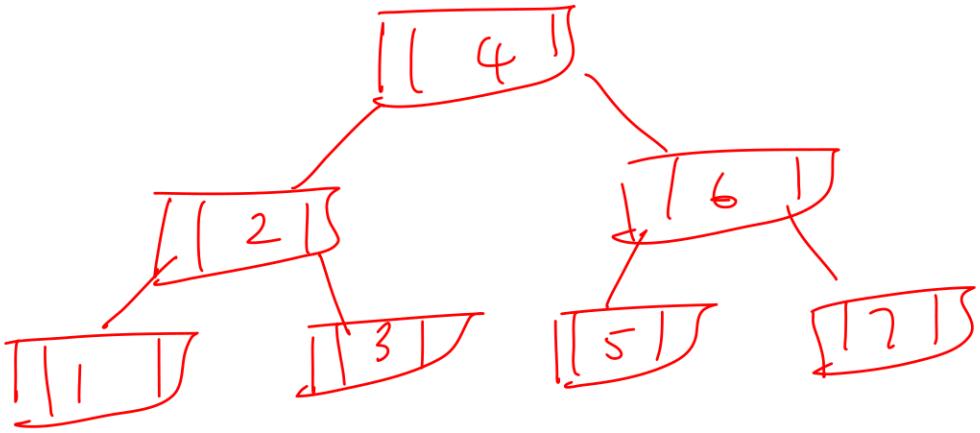
7



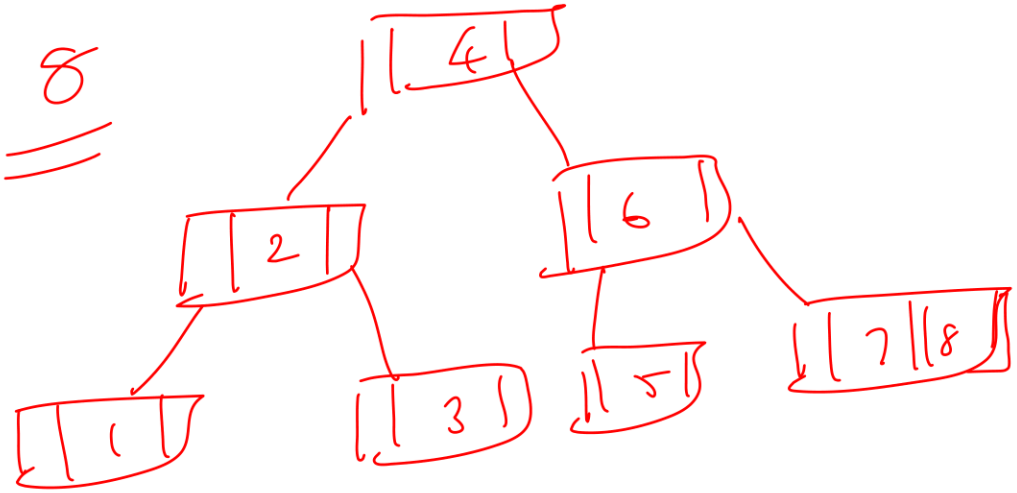
It violates B-tree property



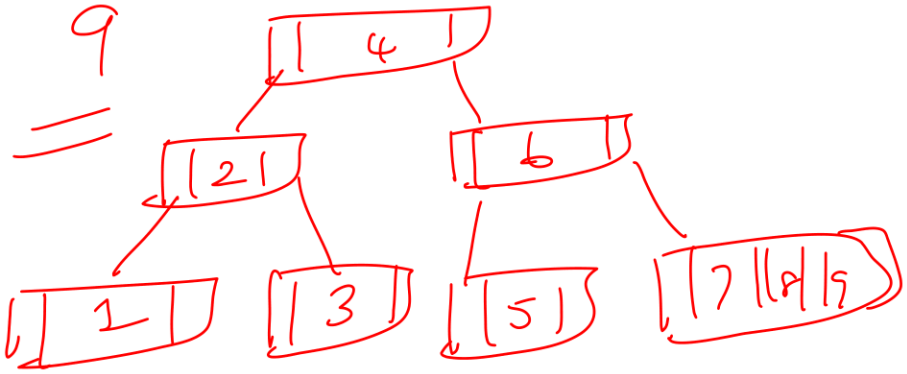
still it violates B-tree property



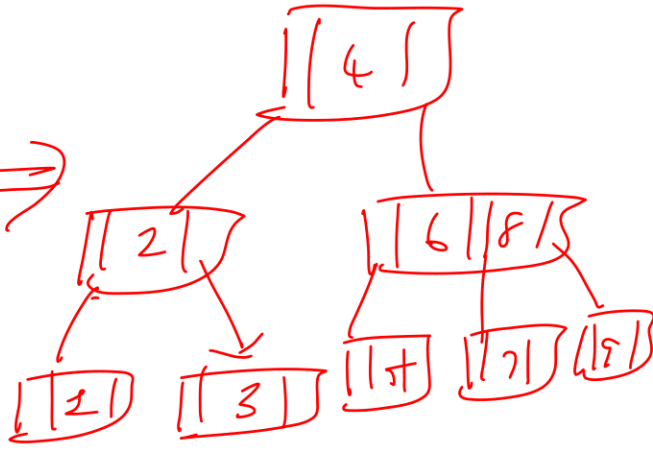
8



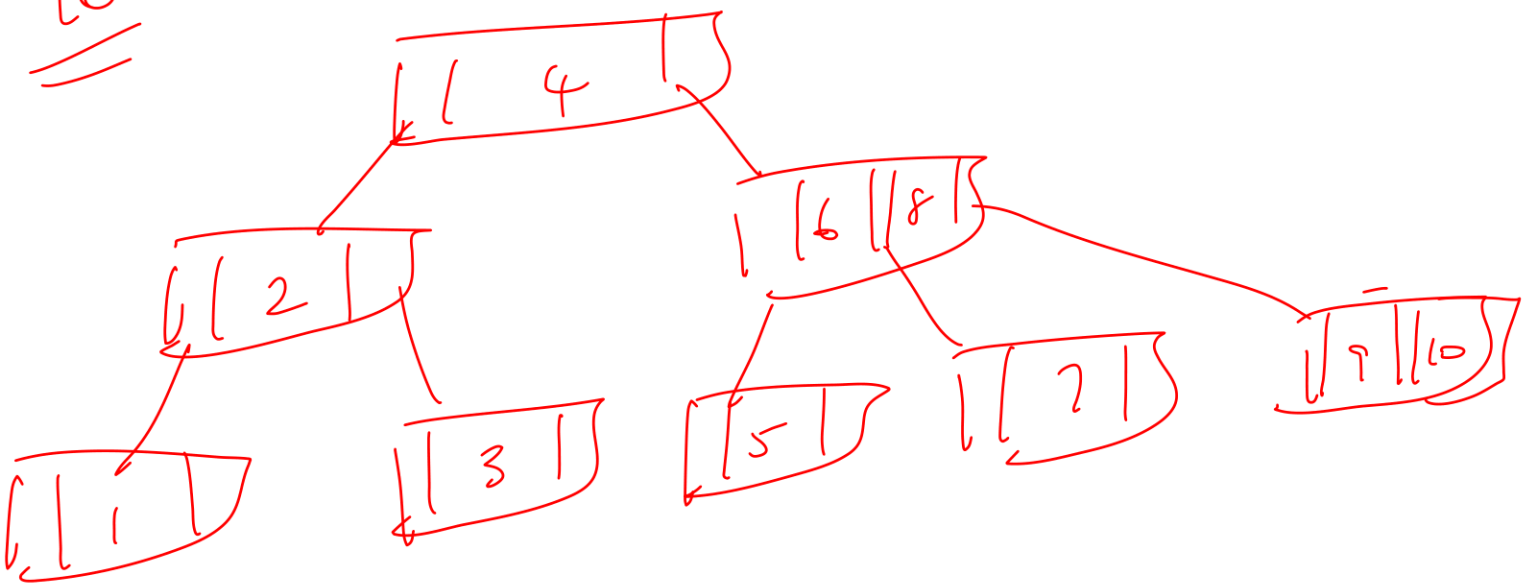
9



It yields a B-tree property



10



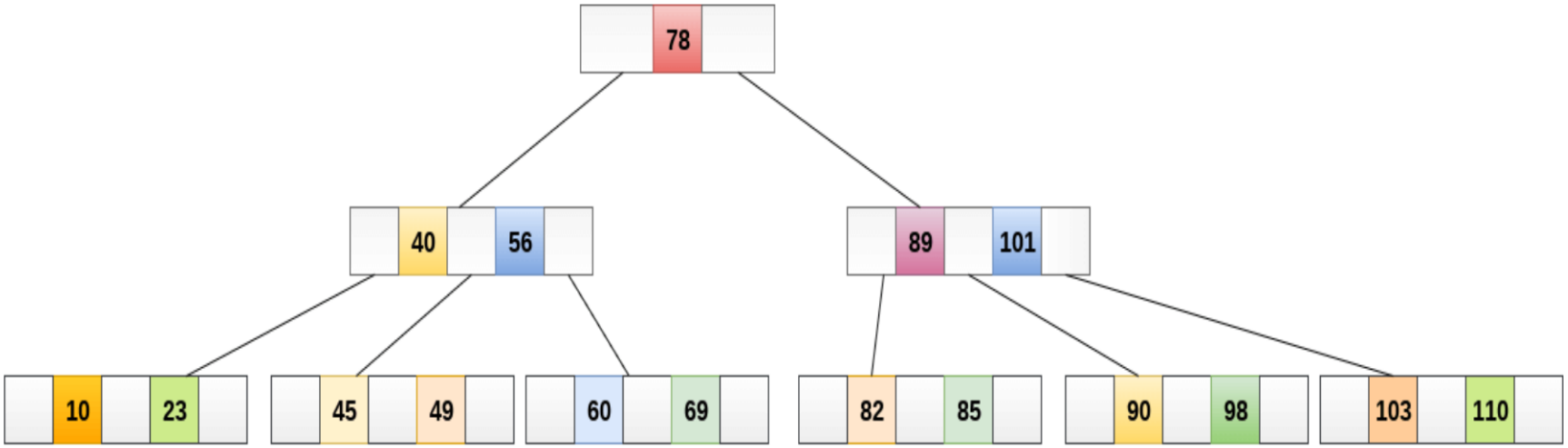
B-Tree –Search Operation

- The search operation in B-Tree is similar to the search operation in Binary Search Tree.
- In a Binary search tree, the search process starts from the root node and we make a 2-way decision every time (we go to either left subtree or right subtree).
- In B-Tree also search process starts from the root node but here we make an m-way decision every time, where ‘m' is the total number of children the node has.

B-Tree –Search Operation

- **Step 1 -** Read the search element from the user.
- **Step 2 -** Compare the search element with first key value of root node in the tree.
- **Step 3 -** If both are matched, then display "Given node is found!!!" and terminate the function
- **Step 4 -** If both are not matched, then check whether search element is smaller or larger than that key value.
- **Step 5 -** If search element is smaller, then continue the search process in left subtree.
- **Step 6 -** If search element is larger, then compare the search element with next key value in the same node and repeat steps 3, 4, 5 and 6 until we find the exact match or until the search element is compared with last key value in the leaf node.
- **Step 7 -** If the last key value in the leaf node is also not matched then display "Element is not found" and terminate the function.

B-Tree –Search Operation- Example



B-Tree –Search Operation- Example

- For example, if we search for an item 49 in the B Tree.
- The process will be :
  - Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
  - Since, 40<49<56, traverse right sub-tree of 40.
  - 49>45, move to right. Compare 49.
  - match found, return.
- Searching in a B tree depends upon the height of the tree.

B-Tree –Deletion Operation- Cases

- **Deleting from a Leaf Node:** Simple removal of the key. If the removal causes the node to have fewer than the minimum number of keys, you may need to merge or borrow from siblings.

- **Deleting from an Internal Node:** Replace the key with the in-order predecessor or successor and delete the predecessor or successor.

- **Underflow Handling:**
  - **Borrowing:** If a sibling has more than the minimum number of keys, borrow a key from it. (either from left or right subling)
  - **Merging:** If a sibling has the minimum number of keys, merge the nodes. (else, with parent)

- **Root Adjustments:** If the root becomes empty after a deletion and the tree height decreases, the tree will merge or adjust accordingly.

B-Tree –Deletion Operation

1. **Inorder Predecessor**
   The largest key on the left child of a node is called its inorder predecessor.
2. **Inorder Successor**
   The smallest key on the right child of a node is called its inorder successor.

B-Tree –Deletion Operation-Example

B-Tree –Deletion Operation- Example

While inserting a key, we had to ensure that the number of keys should not exceed MAX. Similarly, while deleting, we must watch that the number of keys in a node should not become less than MIN.

While inserting, when the keys exceed MAX, we split the node into two nodes, and the median key went to the parent node; while deleting when the keys will become less than MIN, we will combine two nodes into one.

Deletion in a B-Tree can be classified into two cases:
1. Deletion from the leaf node.
2. Deletion from the non-leaf node.

B-Tree –Deletion Operation

B tree of degree m: (if m=3)

- A node can have a maximum of m children. (i.e. 3)
- A node can contain a maximum of m - 1 keys. (i.e. 2)
- A node should have a minimum of [m/2] children. (i.e. 2)
- A node (except root node) should contain a minimum of [m/2] - 1 keys. (i.e. 1)

B-Tree –Deletion Operation- Example

Case I. Deletion From Leaf Node.

If Node Has More Than MIN Keys,

In this case, deletion is very straightforward, and the key can be very easily deleted from the node by shifting other keys of the node.

If Node Has MIN Keys

After the deletion of a key, the node will have less than MIN keys and will become an underflow node. In such a case, we can borrow a key from the left or right sibling if any one of them has more than MIN keys.

Which sibling to start from?

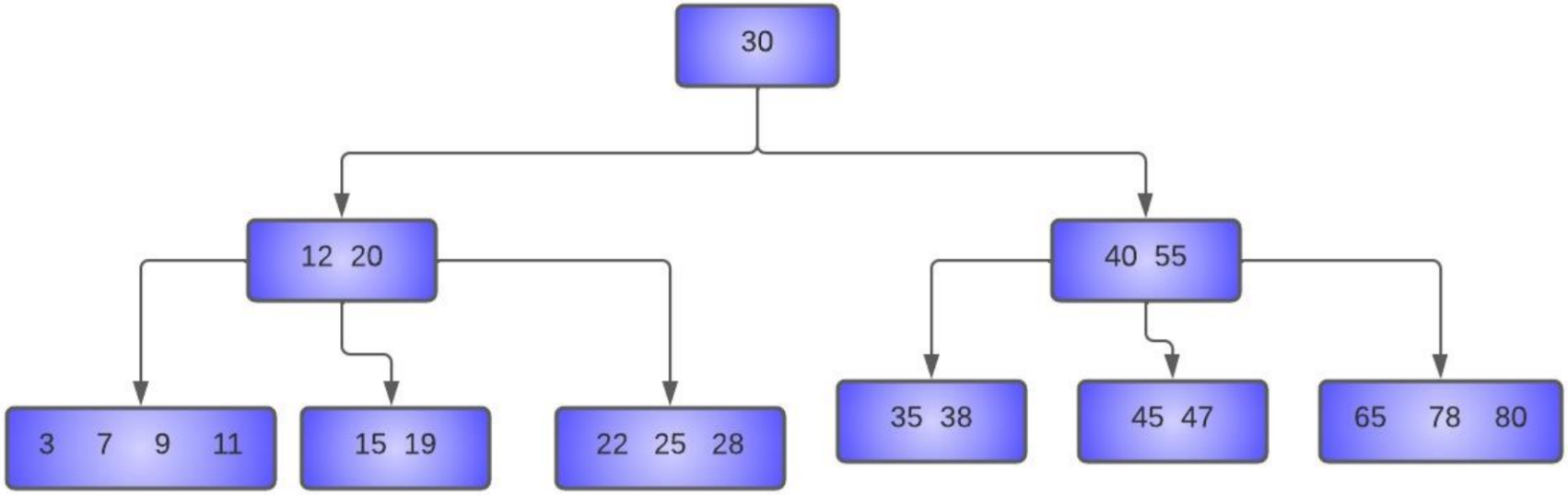
In our algorithm, we'll first try to borrow a key from the left sibling; if the left sibling has only MIN keys, then we'll try to borrow from the right sibling.

B-Tree –Deletion Operation- Example

•When a key is borrowed from the **left** sibling, the separator key in the parent is moved to the underflow node, and the **last** key from the left sibling is moved to the parent.

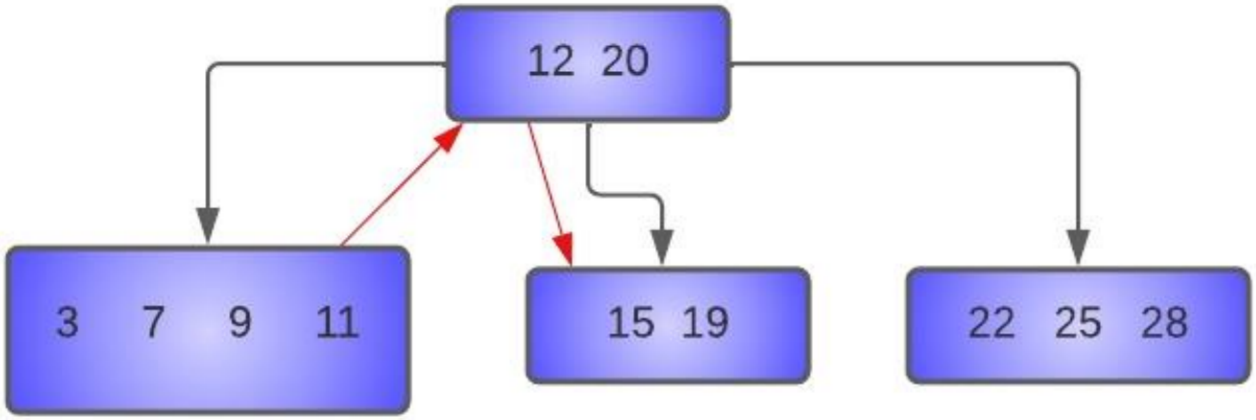
•When a key is borrowed from the **right** sibling, the separator key in the parent is moved to the underflow node, and the **first** key from the right sibling is moved to the parent. All the remaining keys in the right sibling are moved from one position to the left.

B-Tree –Deletion Operation- Example

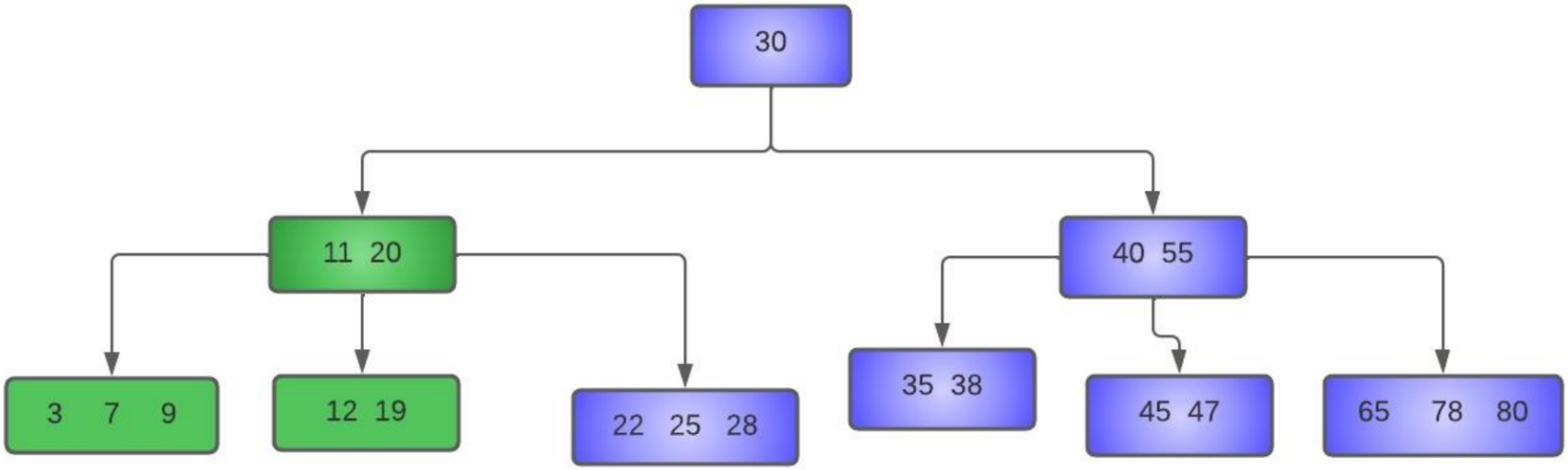


- Delete 15 from the tree:

B-Tree –Deletion Operation- Example

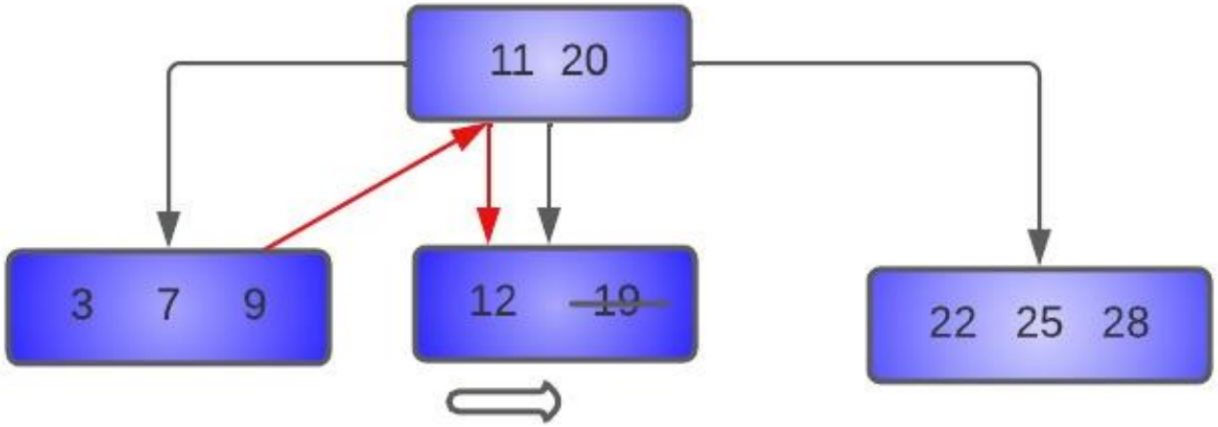


The resulting tree after deletion of 15 will be:-

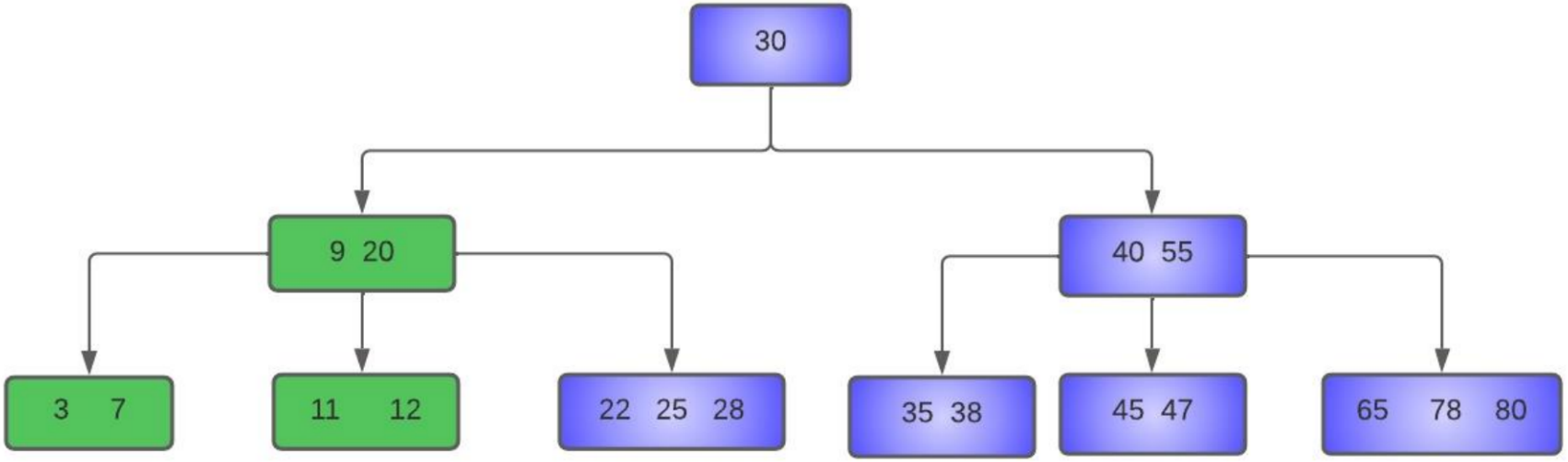


B-Tree –Deletion Operation- Example

Delete 19 from the tree

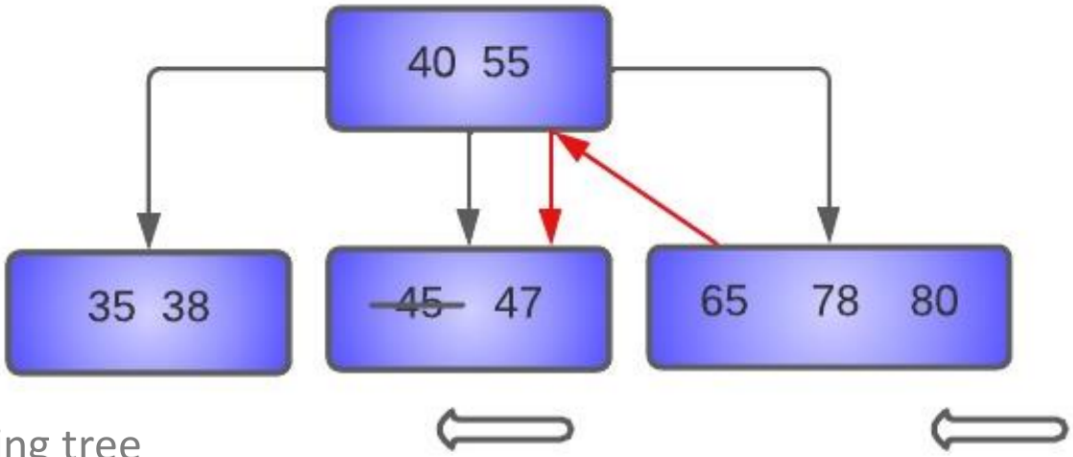


The resultant tree

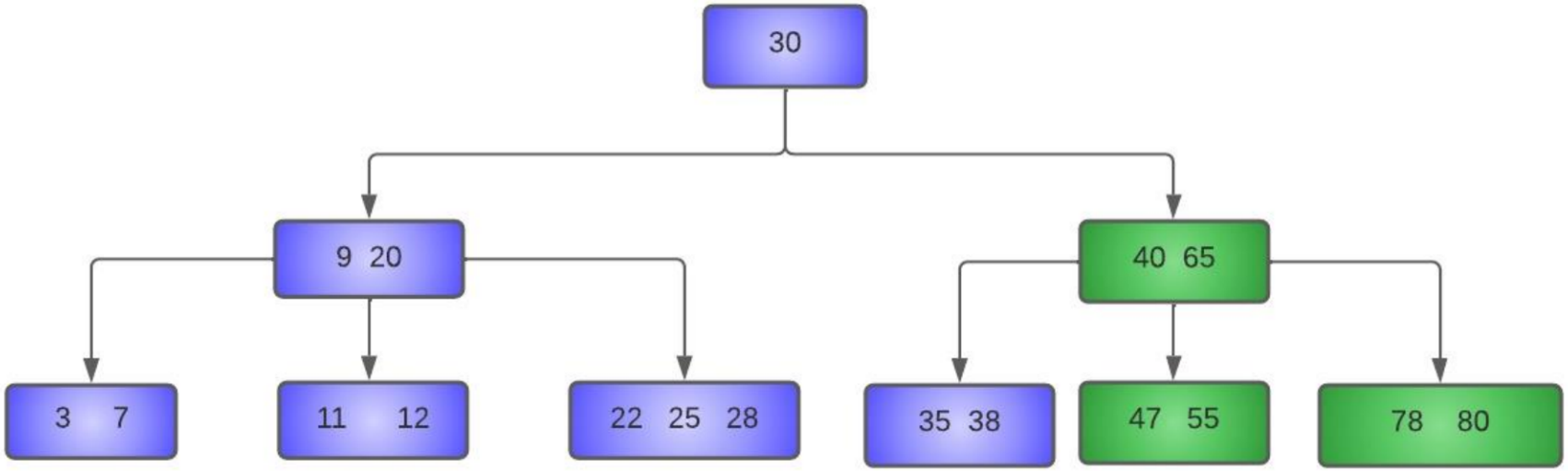


B-Tree –Deletion Operation- Example

•Delete 45 from the tree:

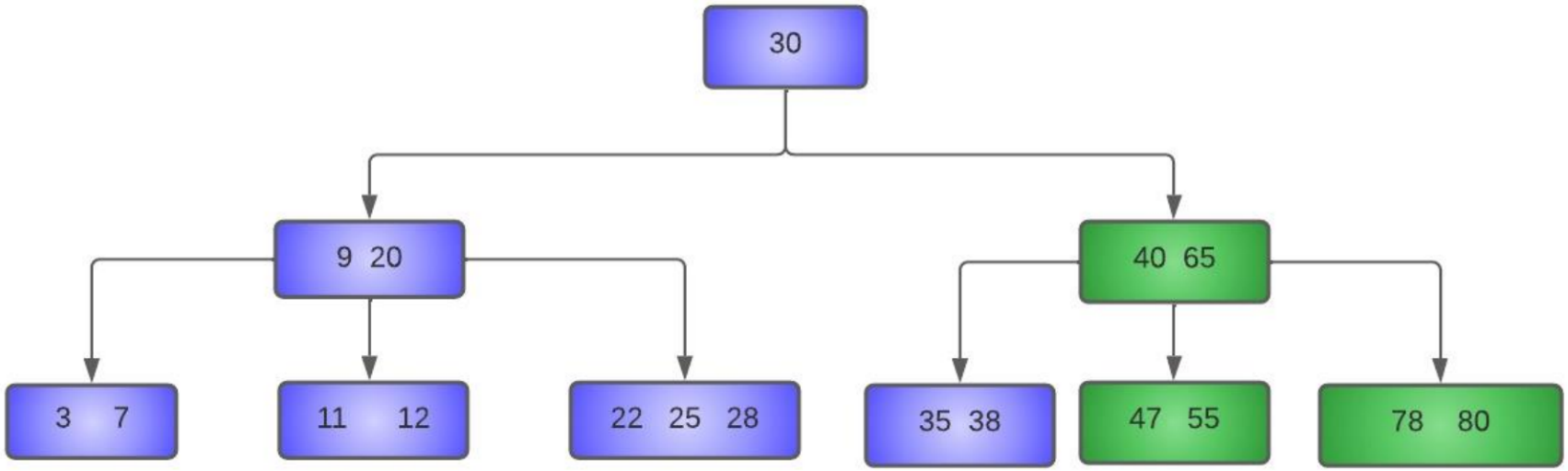


The resulting tree



B-Tree –Deletion Operation- Example

•Delete 47 from the tree:



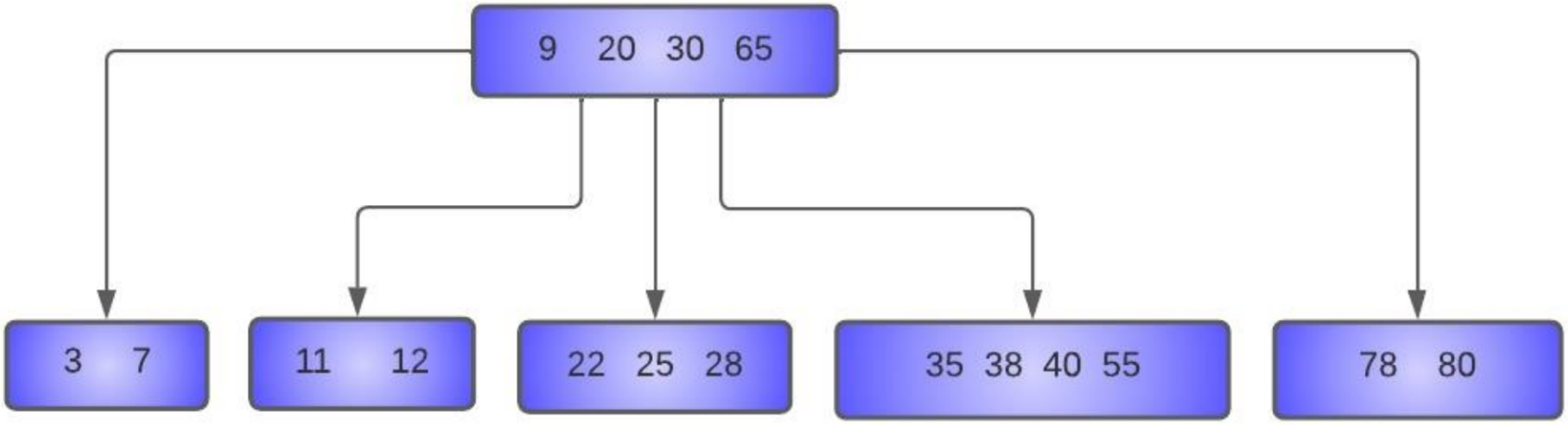
B-Tree –Deletion Operation- Example

If we try to delete 47 from the tree which has only MIN keys and if we delete it, it leads to the underflow condition, so we'll try to borrow from the left sibling [35, 38], which also has only MIN keys, then we try to borrow from the right sibling [78, 80] which has again MIN keys only. So, what should we do now?

We'll combine the left sibling node with the key left in the current node. For combining these two nodes, the separator key(40) from the parent node will move down in the combined node. But the parent node also has only MIN keys.

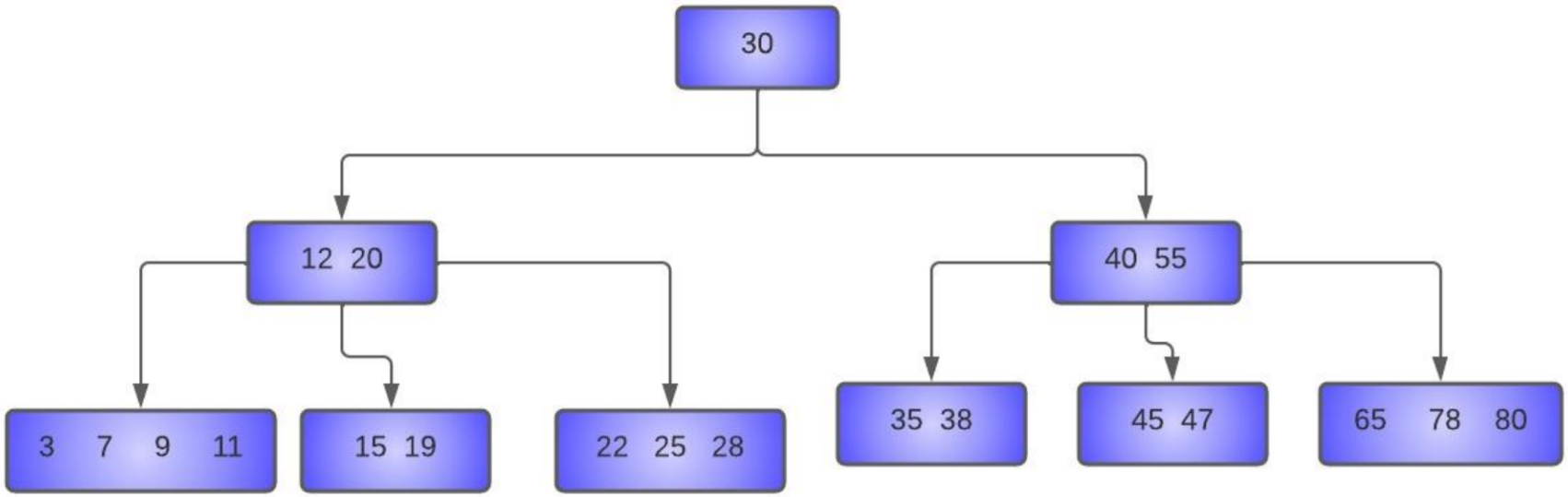
In this case, when the parent node becomes underflow, we will borrow a key from its left sibling. But if we look at its left sibling, it also has only MIN keys.

Here, the separator key (30) the root node comes down in the combined node, which becomes the new root of the tree, and the height of the tree decreases by one.



B-Tree –Deletion Operation- Example

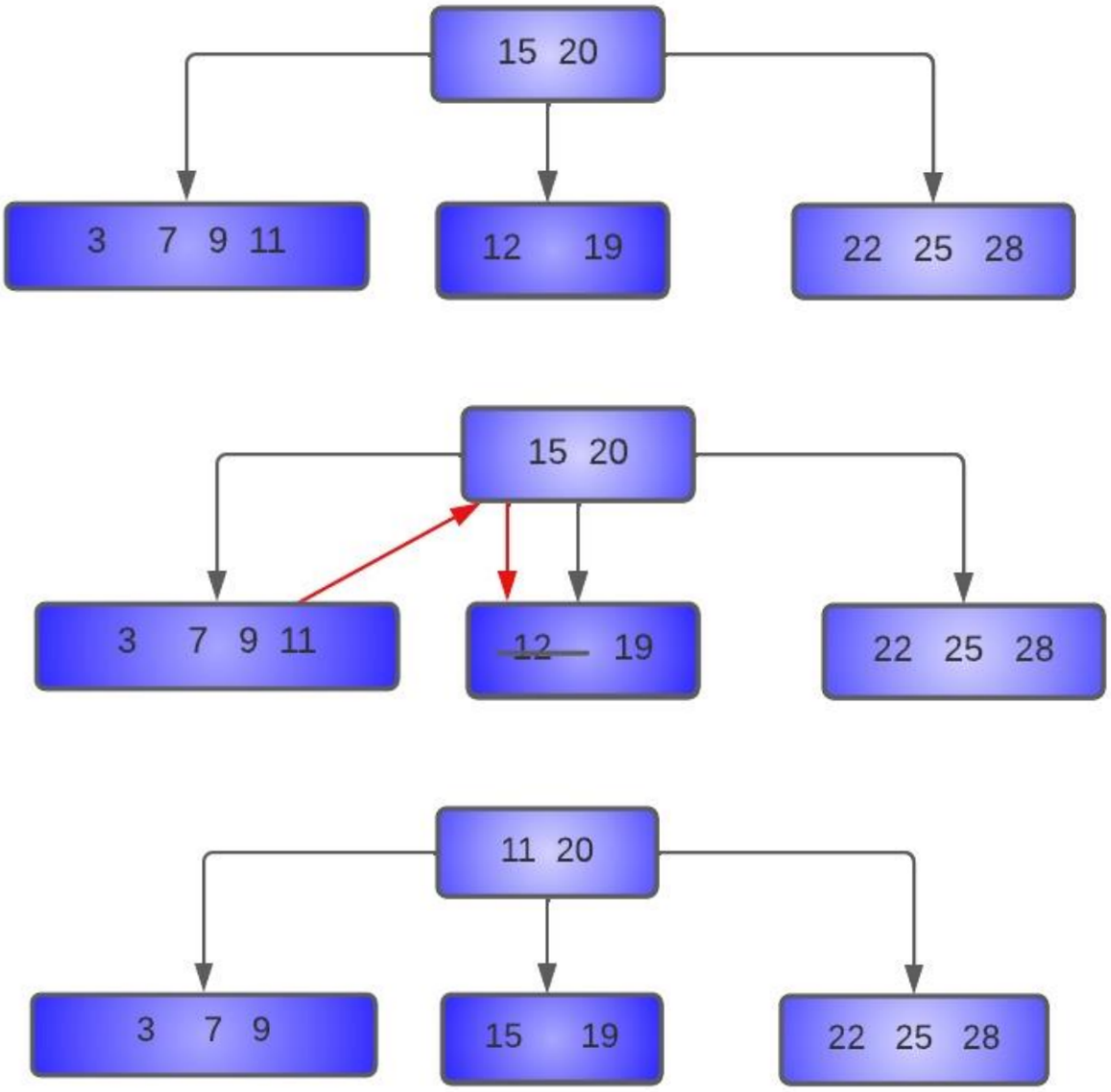
Case 2: Deletion From Non-Leaf Node or internal node



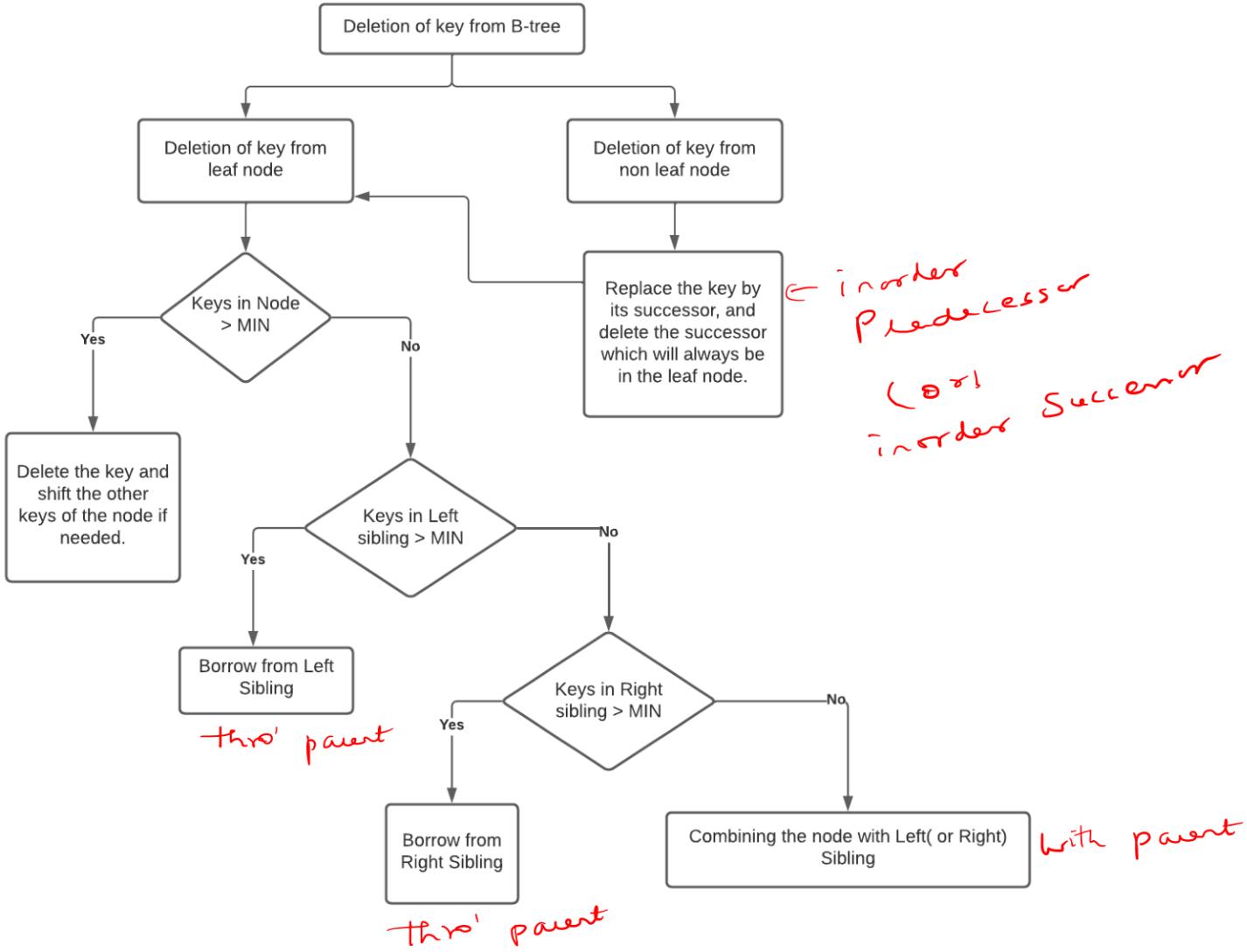
•Delete 12 from the tree:

The successor key of 12 is 15, this we'll copy 15 at the place of 12, and now our task reduces to deletion of 12 from the leaf node. This deletion is performed by borrowing a key from the left sibling.

B-Tree –Deletion Operation- Example



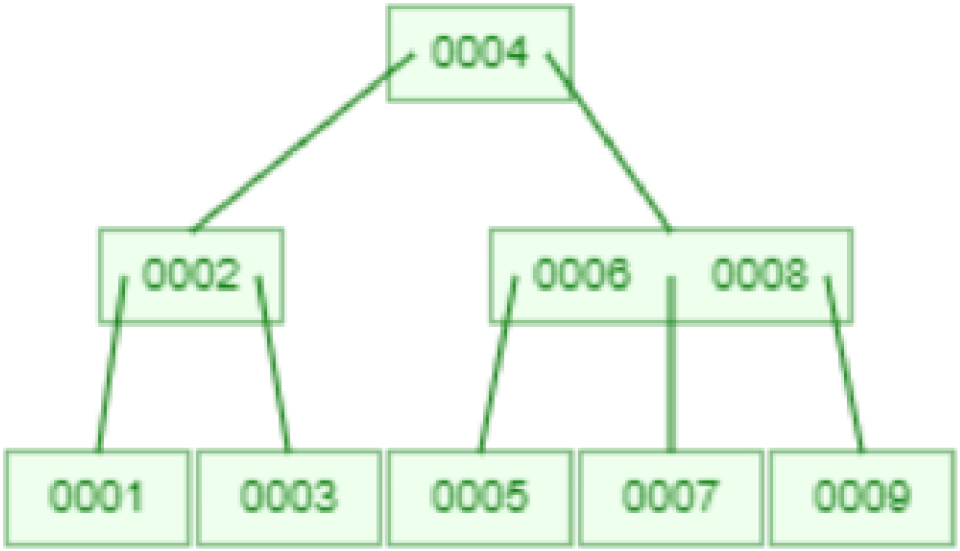
B-Tree –Deletion Operation- Example



B-Tree –Deletion Operation-Another Example

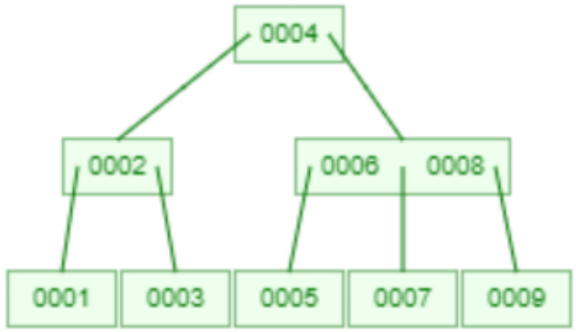
B-Tree –Deletion Operation-Another Example

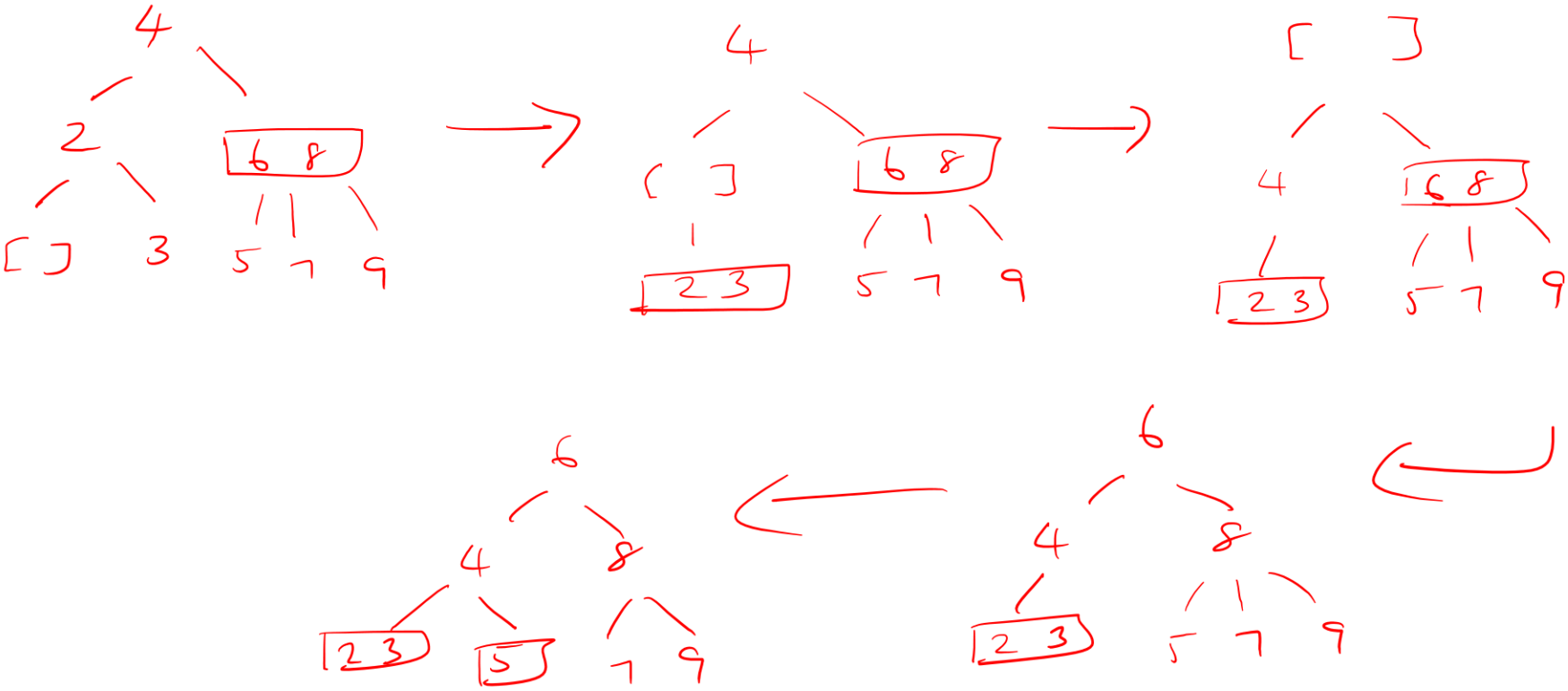
- Construct a B-Tree of order m=3 for the data 1,2,3,4,5,6,7,8,9
- Max key: 2, Min key:1



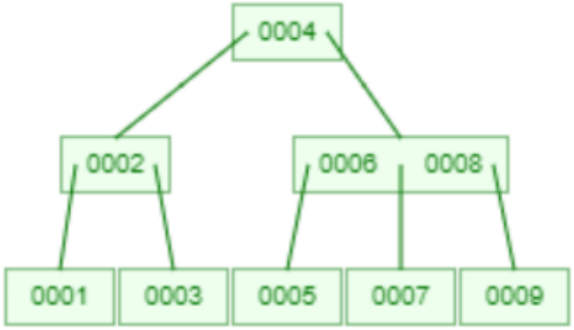
B-Tree –Deletion Operation-Another Example

- Delete 1 from the original tree
- Leaf node
- No borrowing from siblings possible
- Parent comes down to merge & restructure it

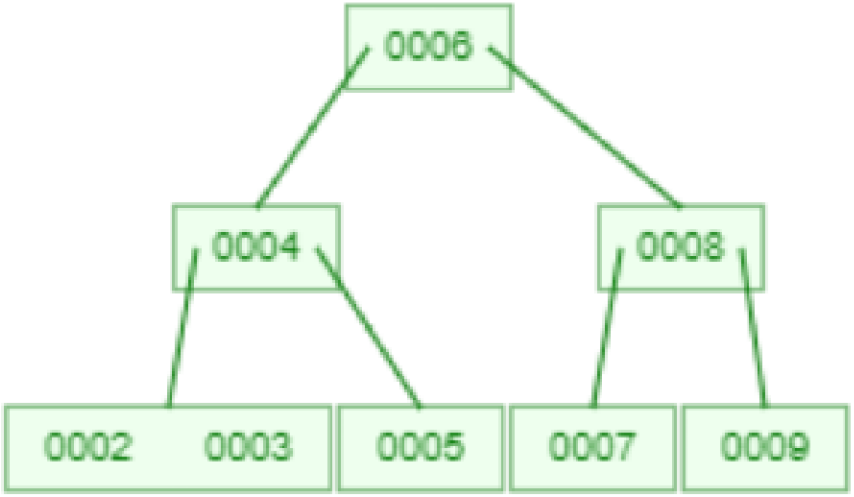




B-Tree –Deletion Operation-Another Example

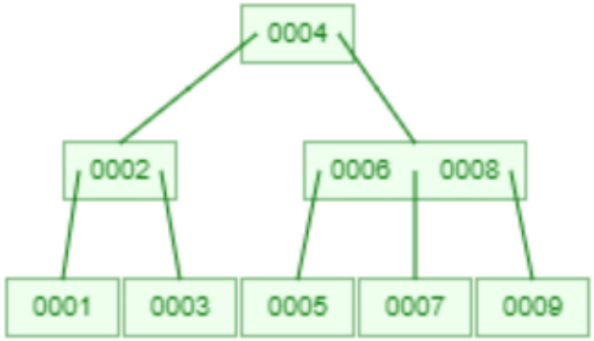


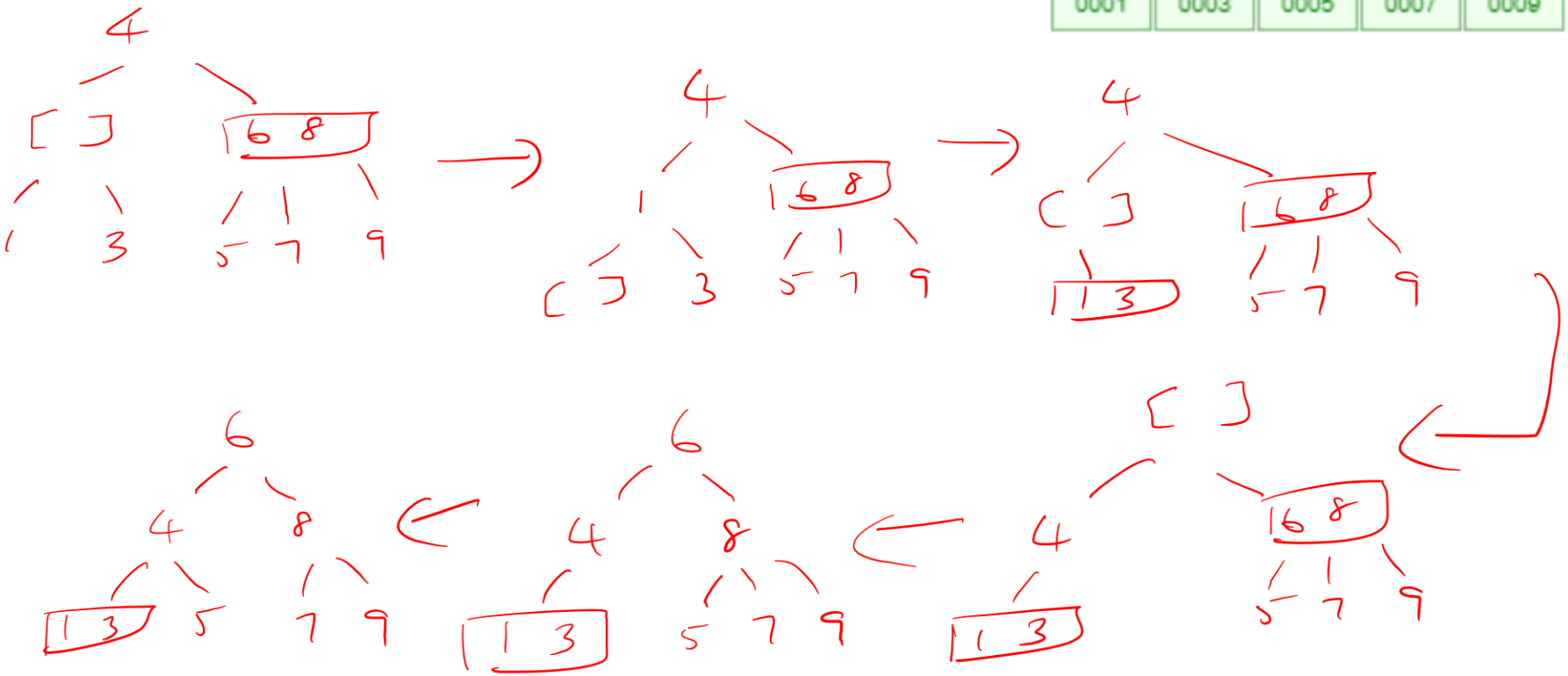
- Delete 1 from the original tree
- Leaf node
- No borrowing from siblings possible
- Parent comes down to merge & restructure it



B-Tree –Deletion Operation-Another Example

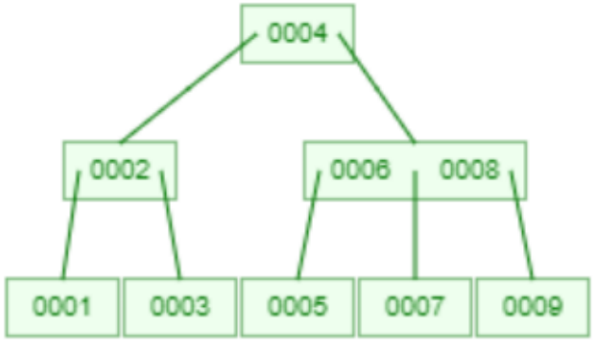
- Delete 2 from the original tree
- Internal node
- Replace it with inorder predecessor (1) and restructure

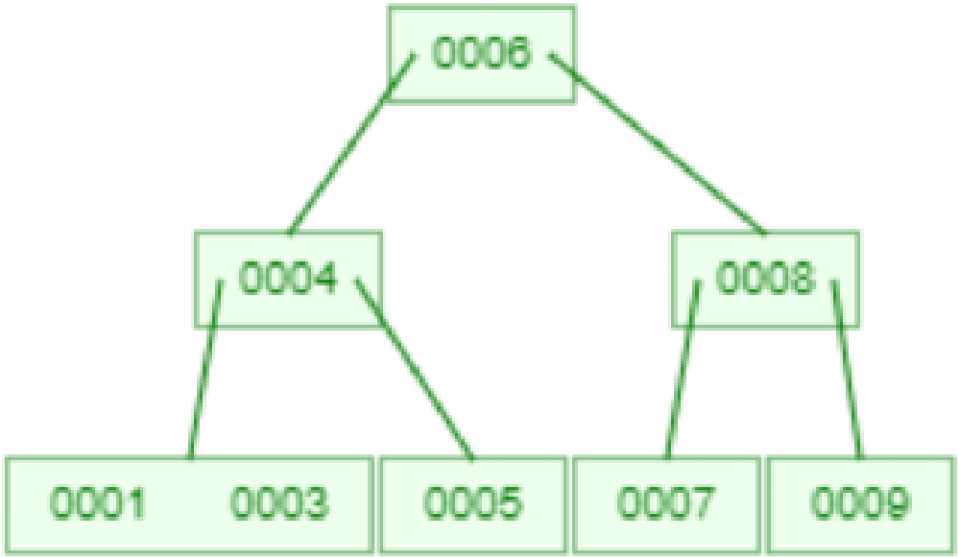




B-Tree –Deletion Operation-Another Example

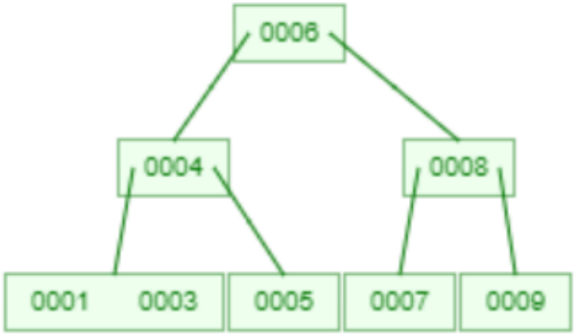
- Delete 2 from the original tree
- Internal node
- Replace it with inorder predecessor (1) and restructure

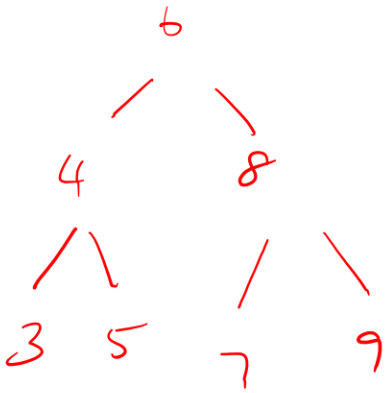




B-Tree –Deletion Operation-Another Example

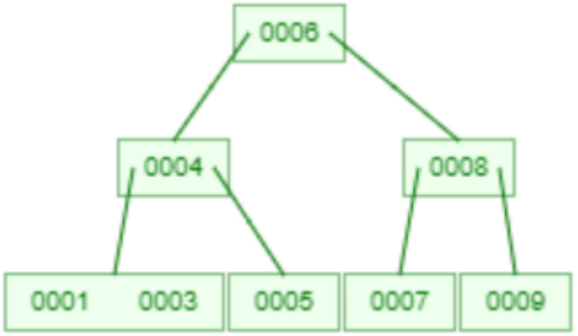
- Delete 1 from the given tree
- Leaf node
- Sufficient keys, so no restructuring

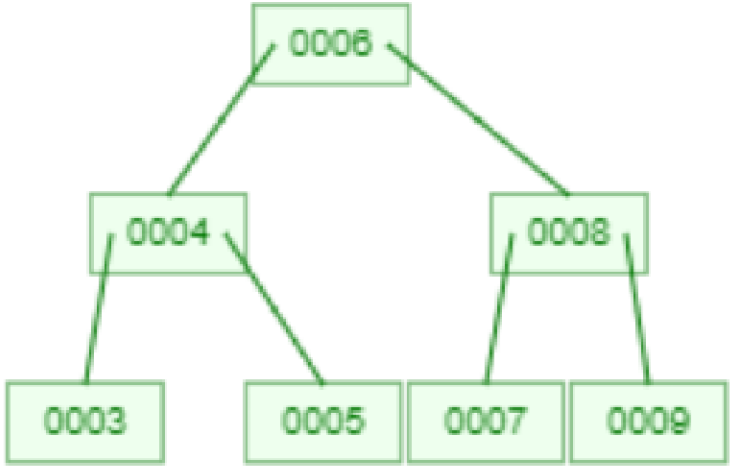




B-Tree –Deletion Operation-Another Example

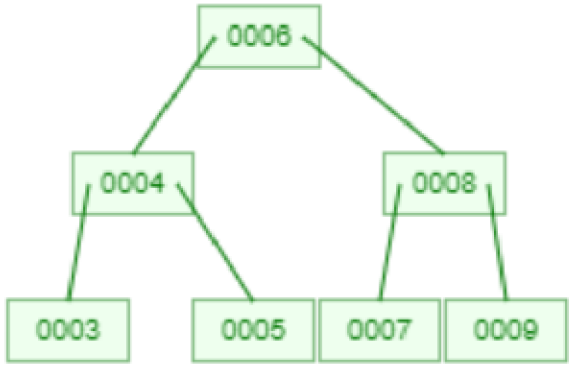
- Delete 1 from the given tree
- Leaf node
- Sufficient keys, so no restructuring

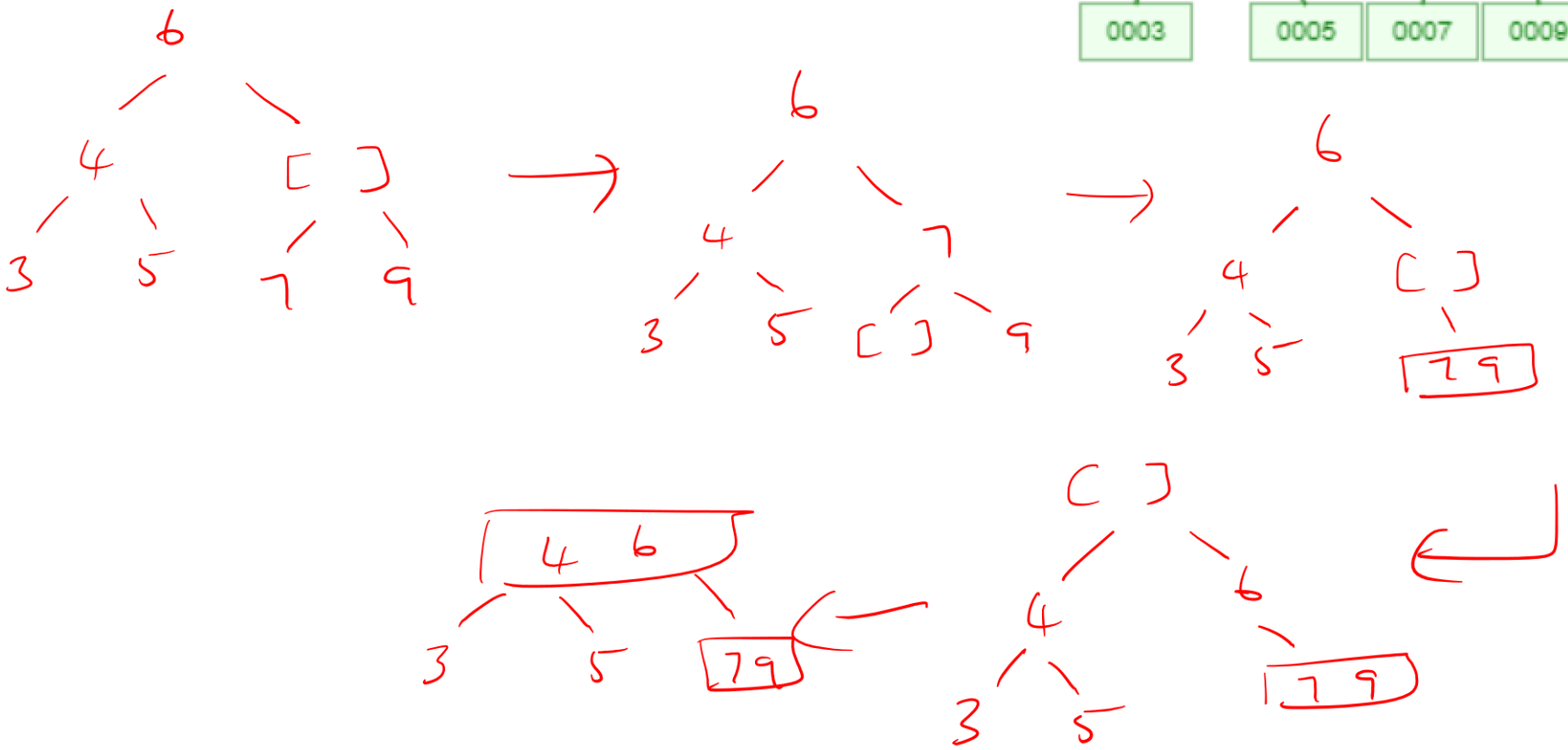




B-Tree –Deletion Operation-Another Example

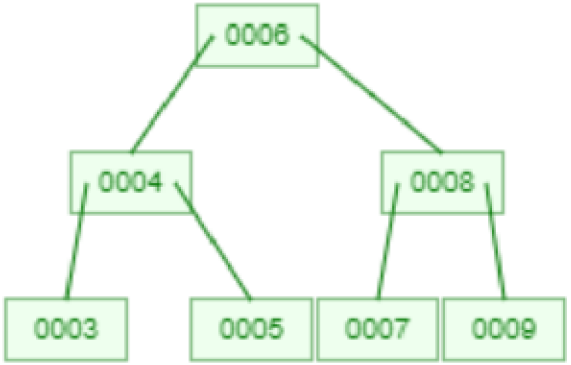
- Delete 8 from the given tree
- Internal node,
- hence replace by inorder predecessor (7) , and restructure it

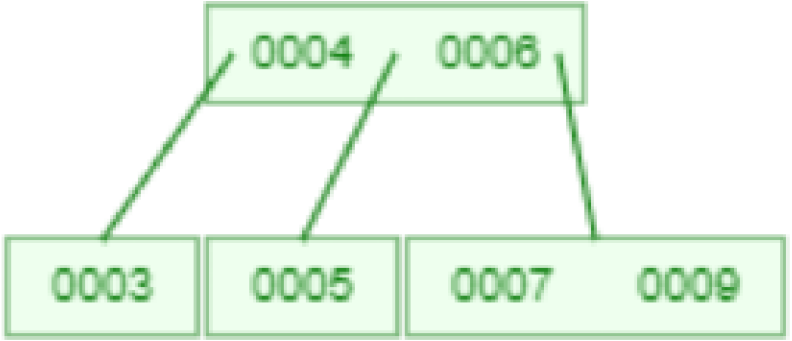




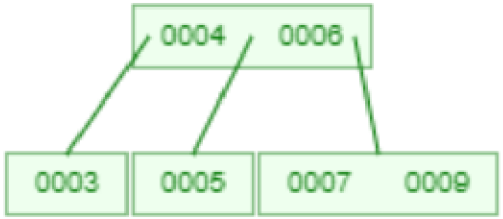
B-Tree –Deletion Operation-Another Example

- Delete 8 from the given tree
- Internal node,
- hence replace by inorder predecessor (7) , and restructure it

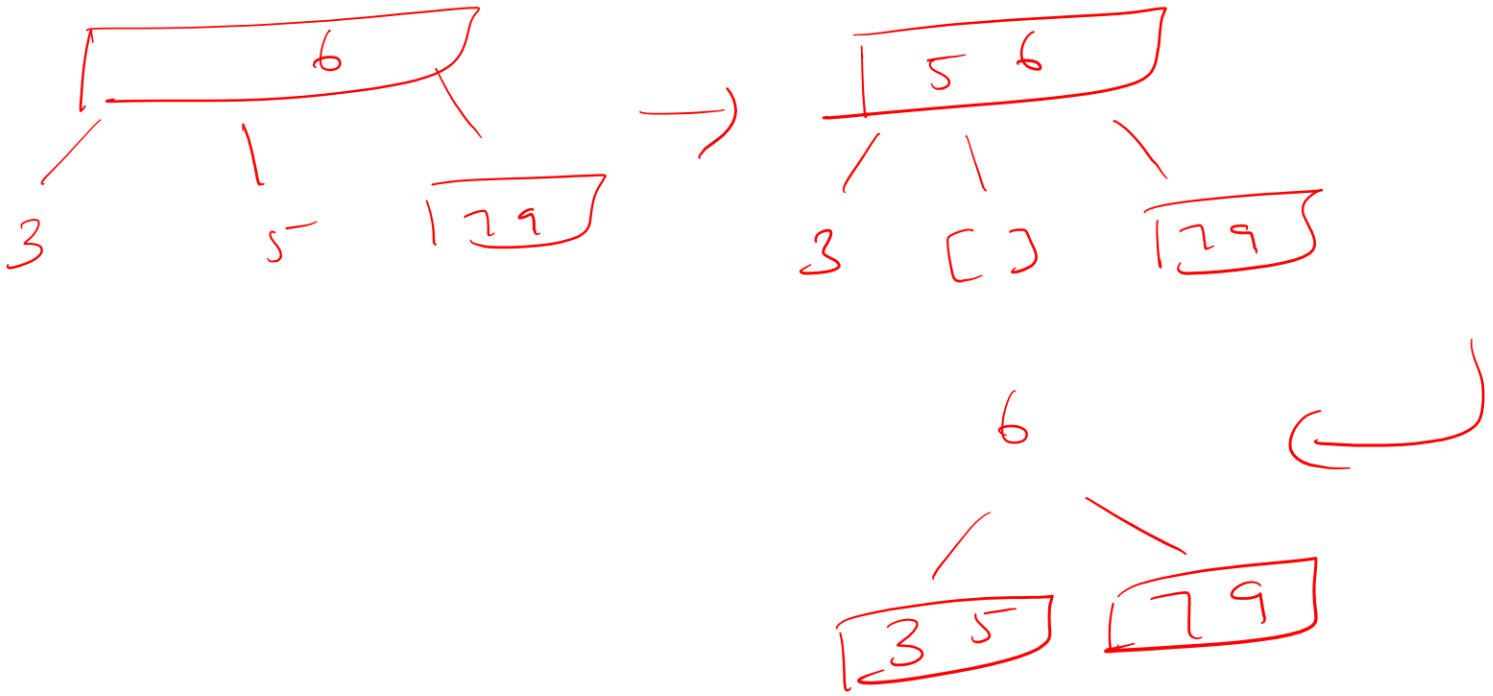




B-Tree –Deletion Operation-Another Example

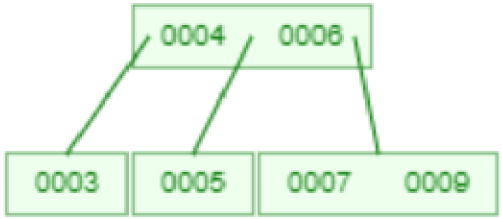


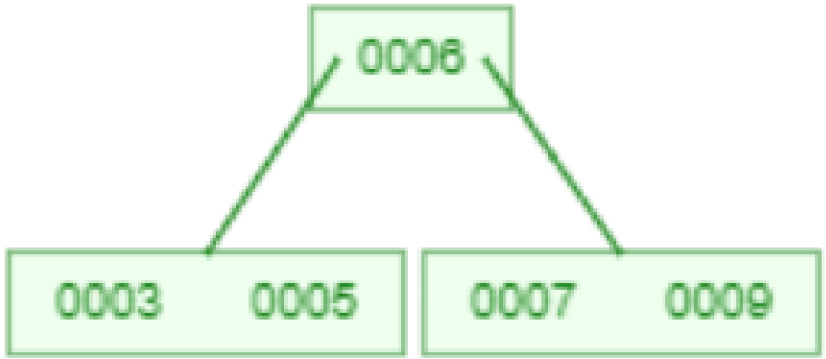
- Delete 4 from the given tree
- Internal node
- Inorder successor (5) & restructure it



B-Tree –Deletion Operation-Another Example

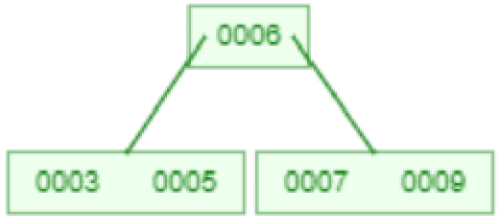
- Delete 4 from the given tree
- Internal node
- Inorder successor (5) & restructure it

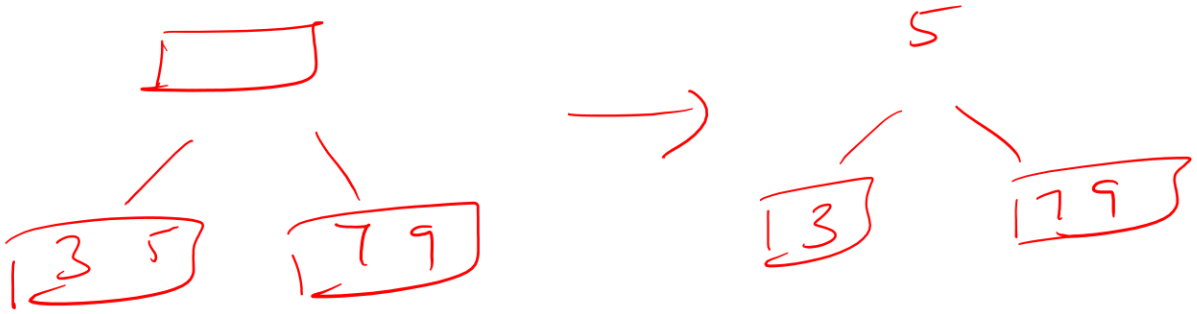




B-Tree –Deletion Operation-Another Example

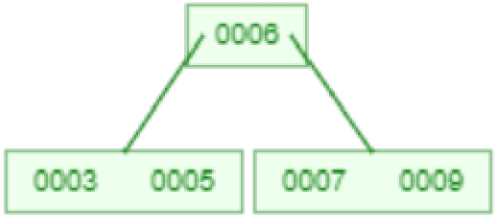
- Delete 6 from the given tree
- Internal node
- Inorder predecessor (5) & no restructuring is required

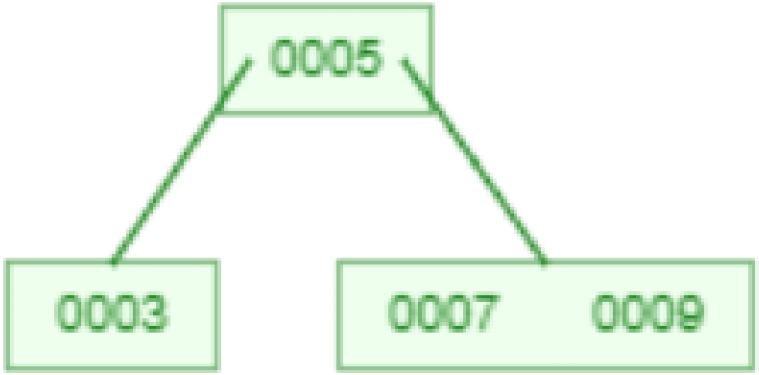




B-Tree –Deletion Operation-Another Example

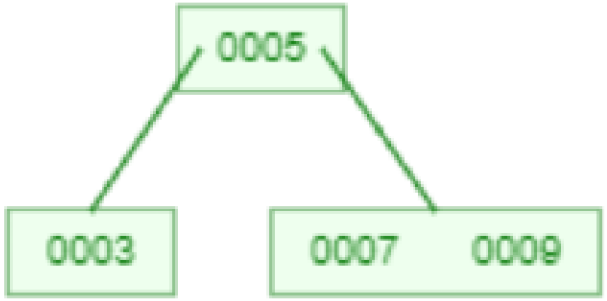
- Delete 6 from the given tree
- Internal node
- Inorder predecessor (5) & no restructuring is required

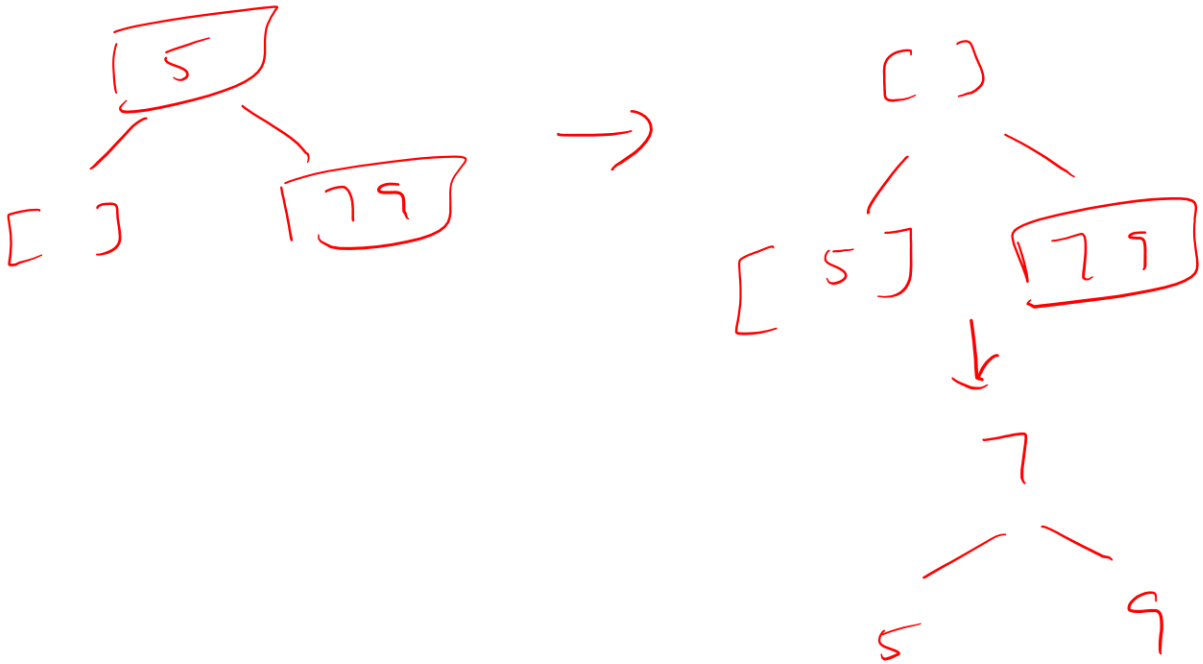




B-Tree –Deletion Operation-Another Example

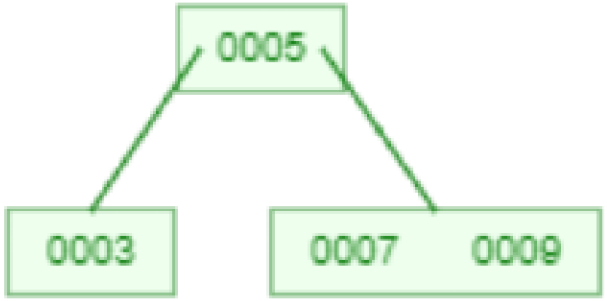
- Delete 3 from the given tree
- Leaf
- borrow from right sibling thro' parent

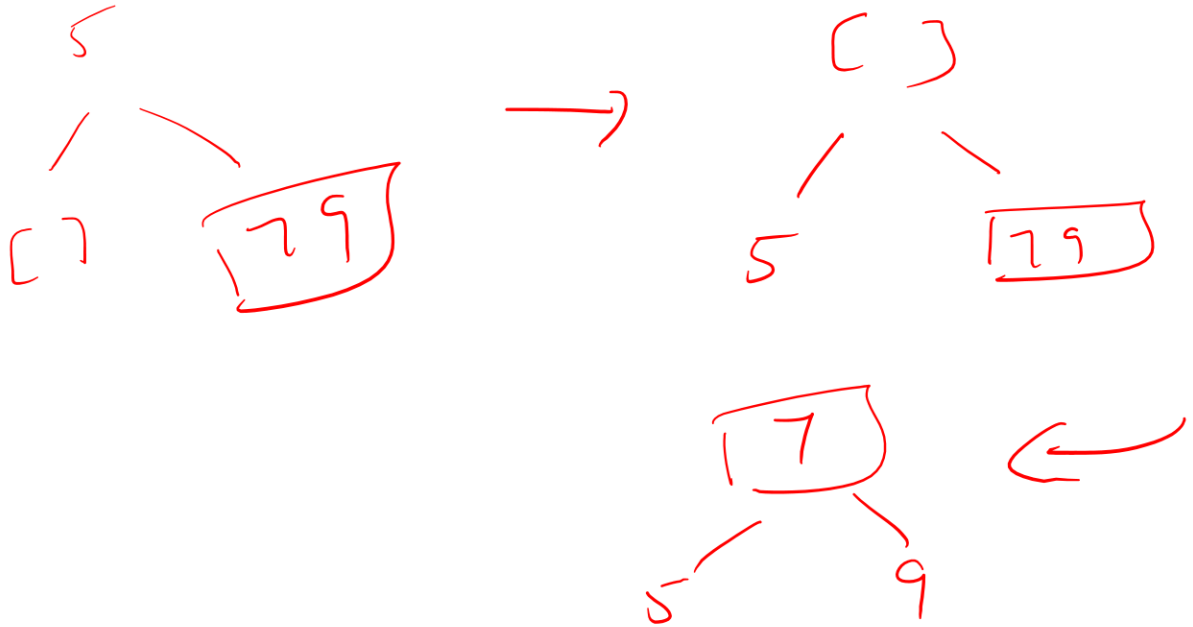




B-Tree –Deletion Operation-Another Example

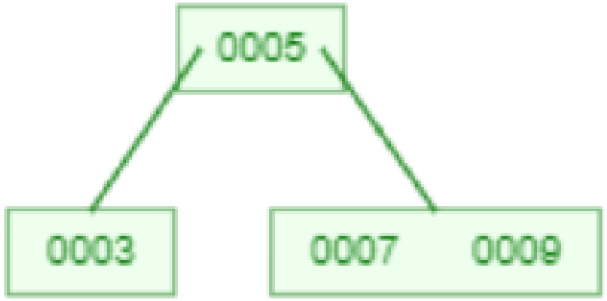
- Delete 3 from the given tree
- Leaf
- borrow from right sibling thro' parent

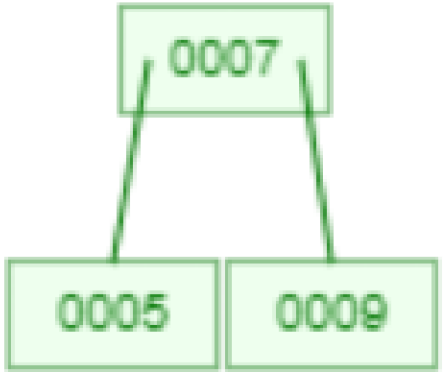




B-Tree –Deletion Operation-Another Example

- Delete 3 from the given tree
- Leaf
- borrow from right sibling thro' parent

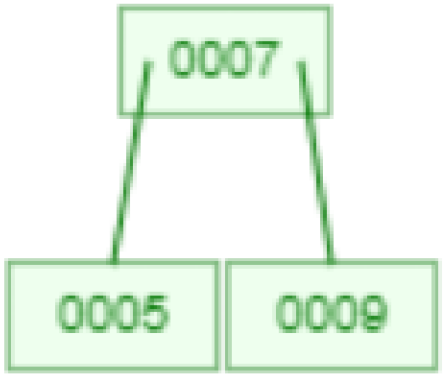




B-Tree –Deletion Operation-Another Example

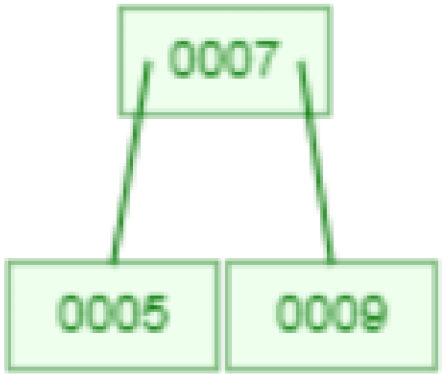
- Delete 7 from the given tree
- Internal node
- Inorder predecessor(5) , parent comes down

to merge



B-Tree –Deletion Operation-Another Example

- Delete 7 from the given tree
- Internal node
- Inorder predecessor(5) , parent comes down to merge



B-Tree –Time Complexity

Operation	Time Complexity	Remark
Search	O(log n)	
Insert	O(log n)	
Delete	O(log n)	
Create	O(nlogn)	If you start with an empty B-Tree and insert n elements one by one, each insertion takes O(logn) time. Thus, the overall time complexity for creating a B-Tree from n elements is O(nlogn)

B+ Tree

B+ Tree

- A **B+ Tree** is a balanced tree
- Same height for paths from root to leaf
- Given a search key K, nearly same access time for different K values
- Key values are maintained in sorted order
- Same rules as that of B-Tree
- Compared to B- Tree, the B+ Tree stores the data only at the leaf nodes of the Tree, which makes search process more accurate and faster.

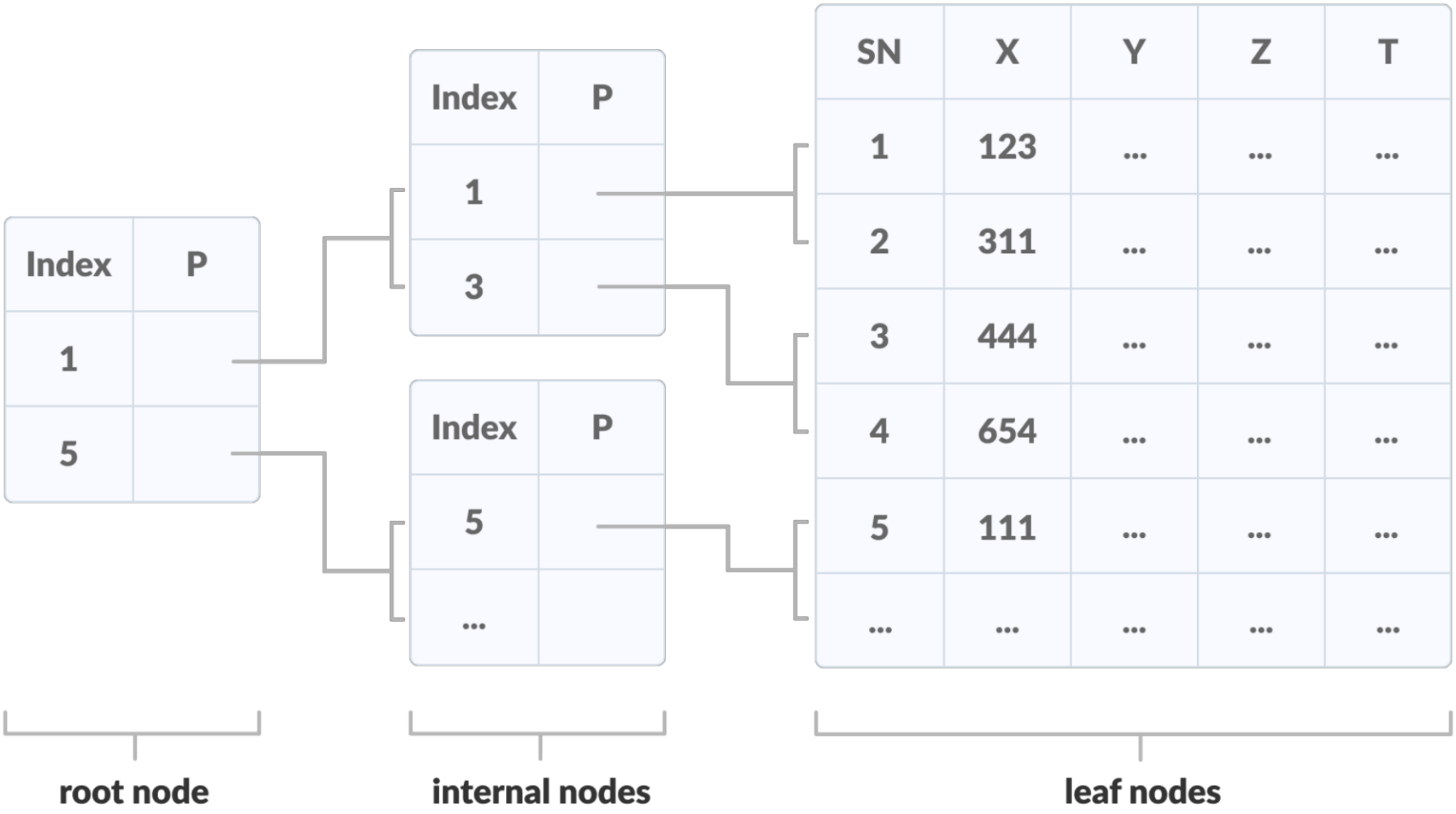
B+ Tree

- A **B+ Tree** is primarily used for implementing multi-level indexing.
- Compared to B- Tree, the B+ Tree stores the data only at the leaf nodes of the Tree, which makes search process more accurate and faster.

B+ Tree

- A B+ tree is an advanced form of a self-balancing tree in which all the values(keys/data) are present in the leaf level.
- An important concept to be understood before learning B+ tree is multilevel indexing.
- In multilevel indexing, the index of indices is created as in figure. It makes accessing the data easier and faster.

B+ Tree


Index
P
1
5
Index
P
1
3
Index
P
5
...
SN
X
Y
Z
T
1
123
...
...
...
2
311
...
...
...
3
444
...
...
...
4
654
...
...
...
5
111
...
...
...
...
...
...
...
...
root node
internal nodes
leaf nodes

Rules of B+ Tree

- Leaves are used to store data records.
- The left subtree of the node will have lesser values than the right side of the subtree. (Generalization of BST)

Rules of B+ Tree

Same rule as that of B-tree for min and max no of children and key of order m

Comparison between a B-tree and a B+ Tree

- The data are present only at the leaf nodes on a B+ tree whereas the data are present in the internal, leaf or root nodes on a B-tree.
- The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.
- Operations on a B+ tree are faster than on a B-tree.

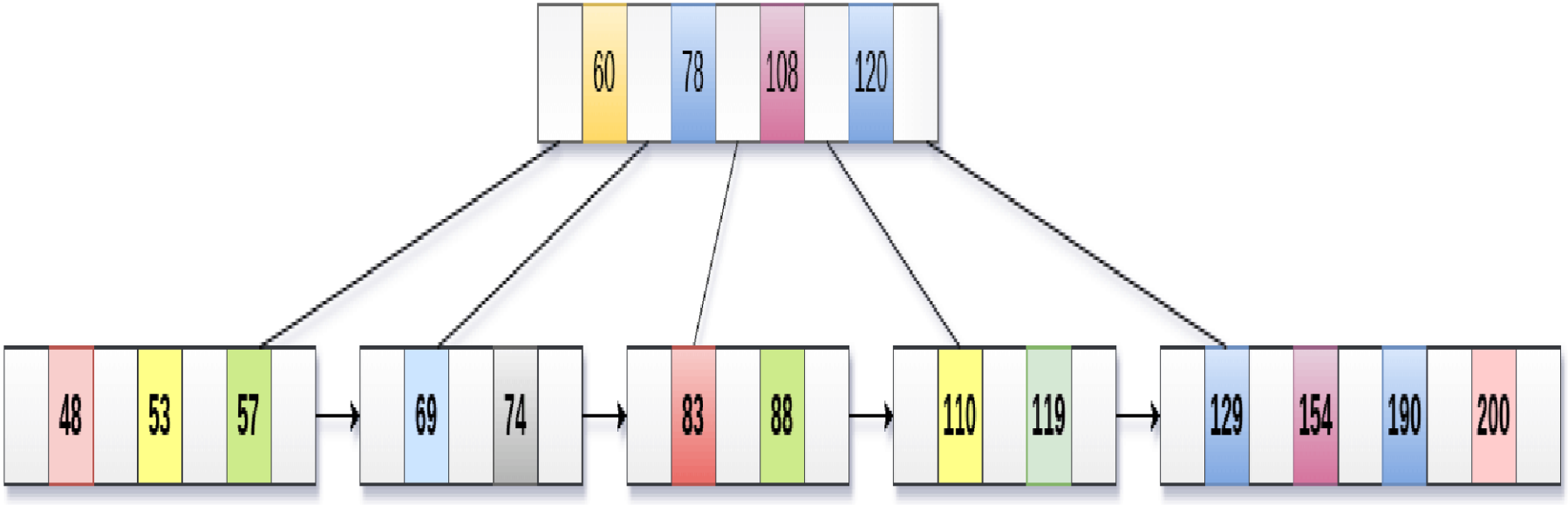
B+ Tree Insertion

- **Step 1:** Insert the new node as a leaf node
- **Step 2:** If the leaf doesn't have required space, split the node and copy the middle node to the next index node.
- **Step 3:** If the index node doesn't have required space, split the node and copy the middle element to the next index page.

B+ Tree Insertion

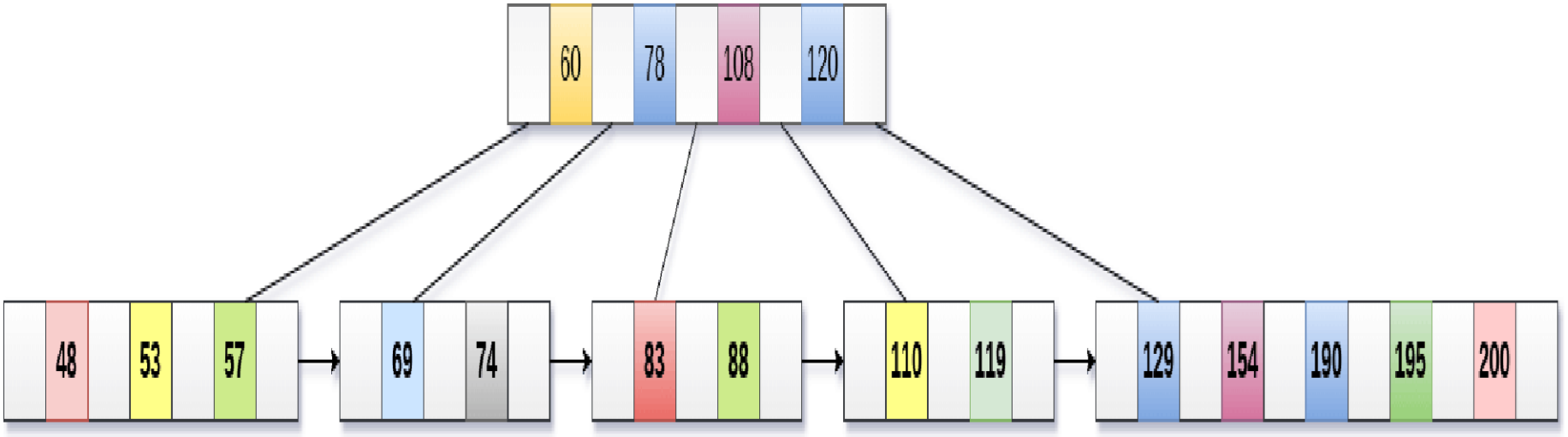
Example :

- Insert the value 195 into the B+ tree of order 5 shown in the following figure.



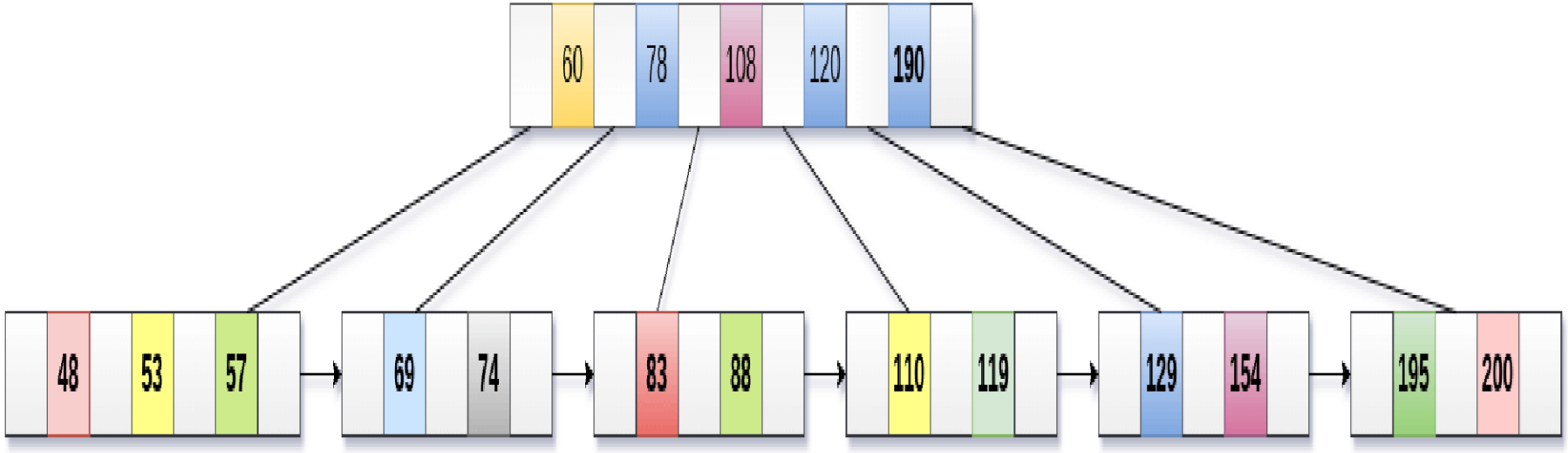
B+ Tree Insertion

195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



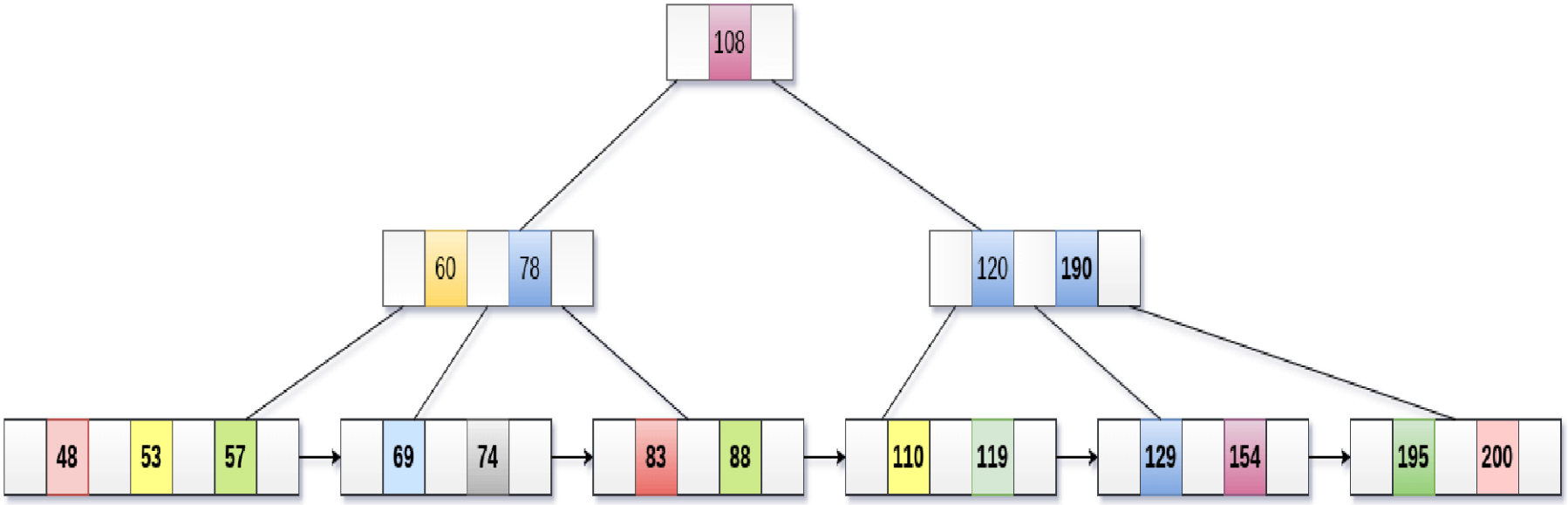
B+ Tree Insertion

- The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



B+ Tree Insertion

- Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.

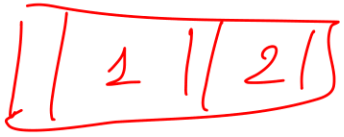


Create a B+-Tree of order $m = 4$ (4 children & 3 → key)

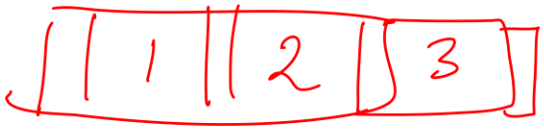
1, 2, 3, 4, 5, 6, 7, 8, 9, 0

1

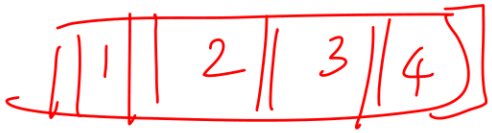
2



3

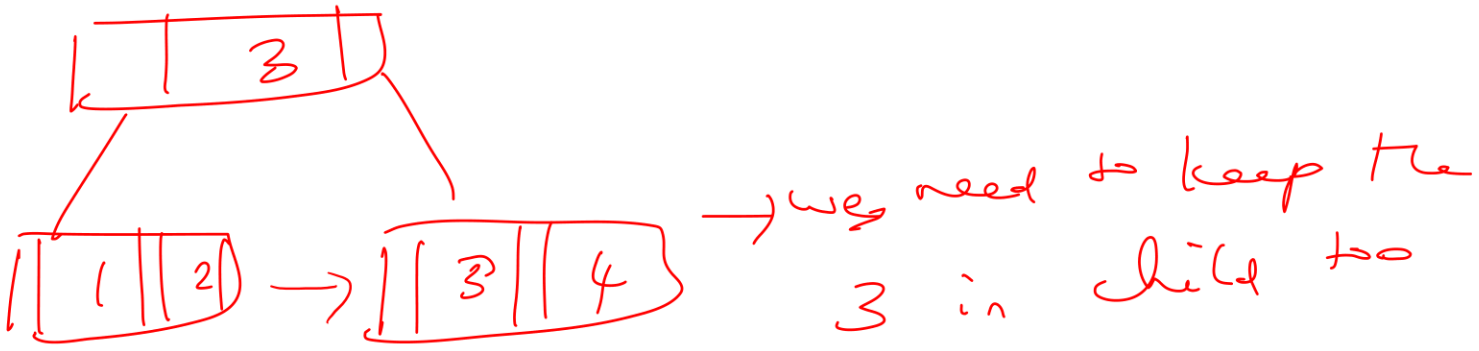


4



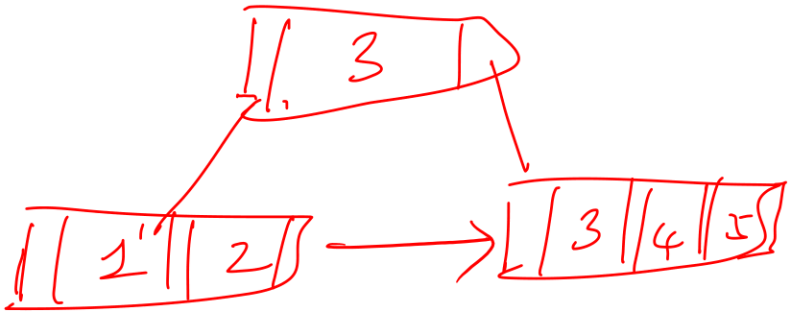
→ violates B+-Tree property,

So, split the node and move middle element one level up

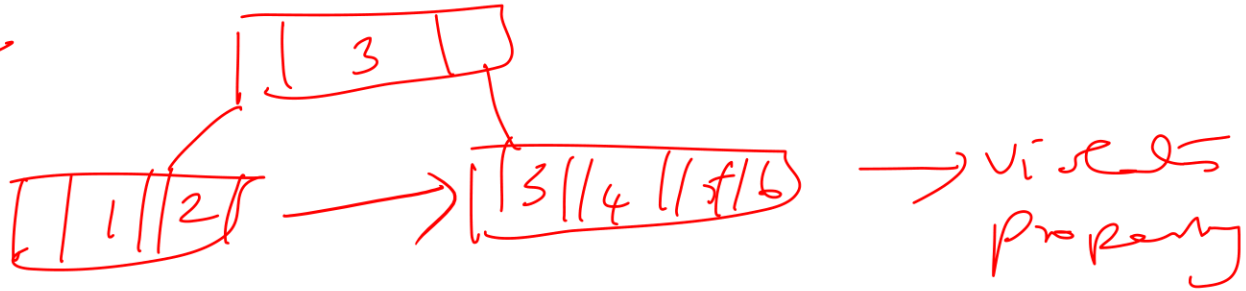


→ we need to keep the 3 in child too

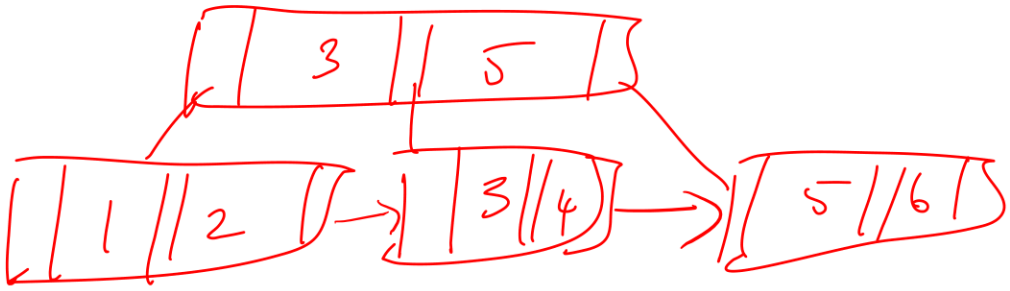
5



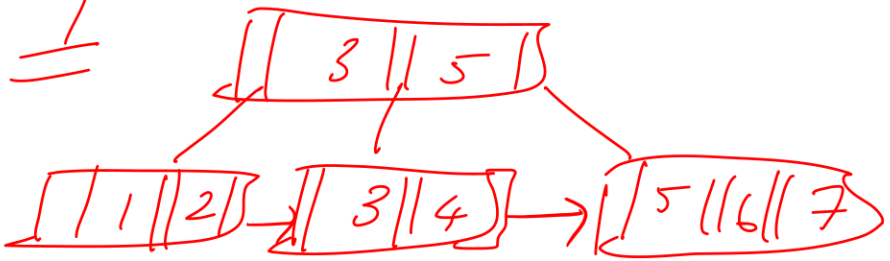
6



→ violates property

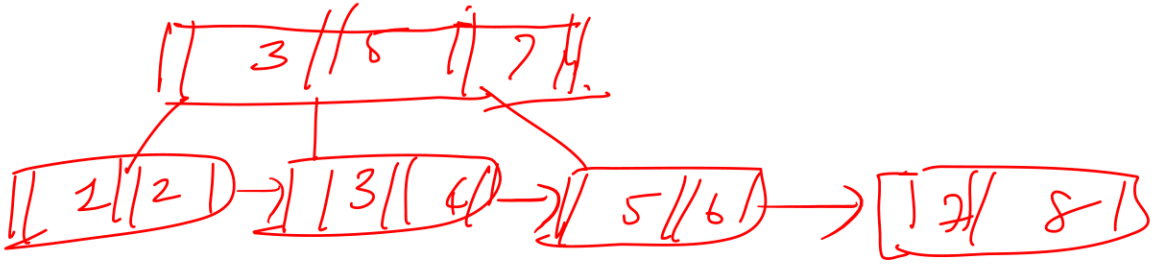


7

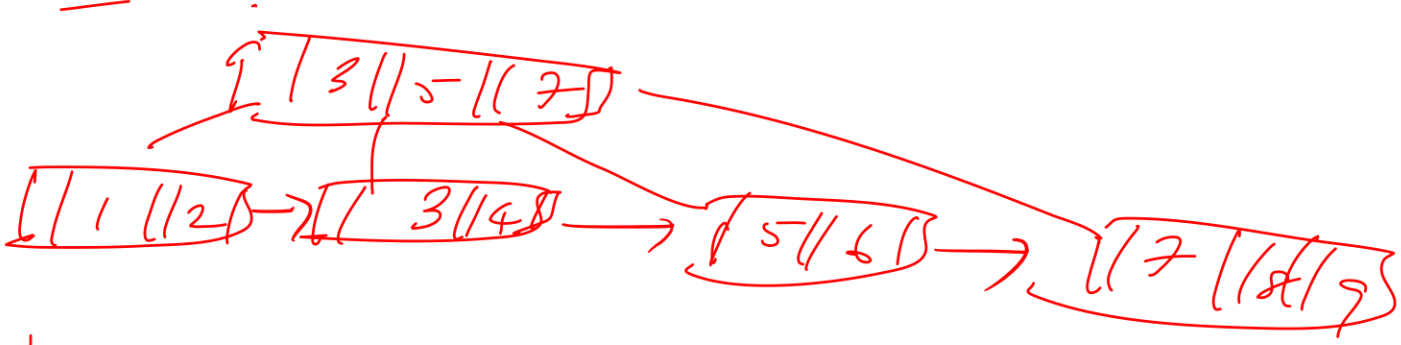


8

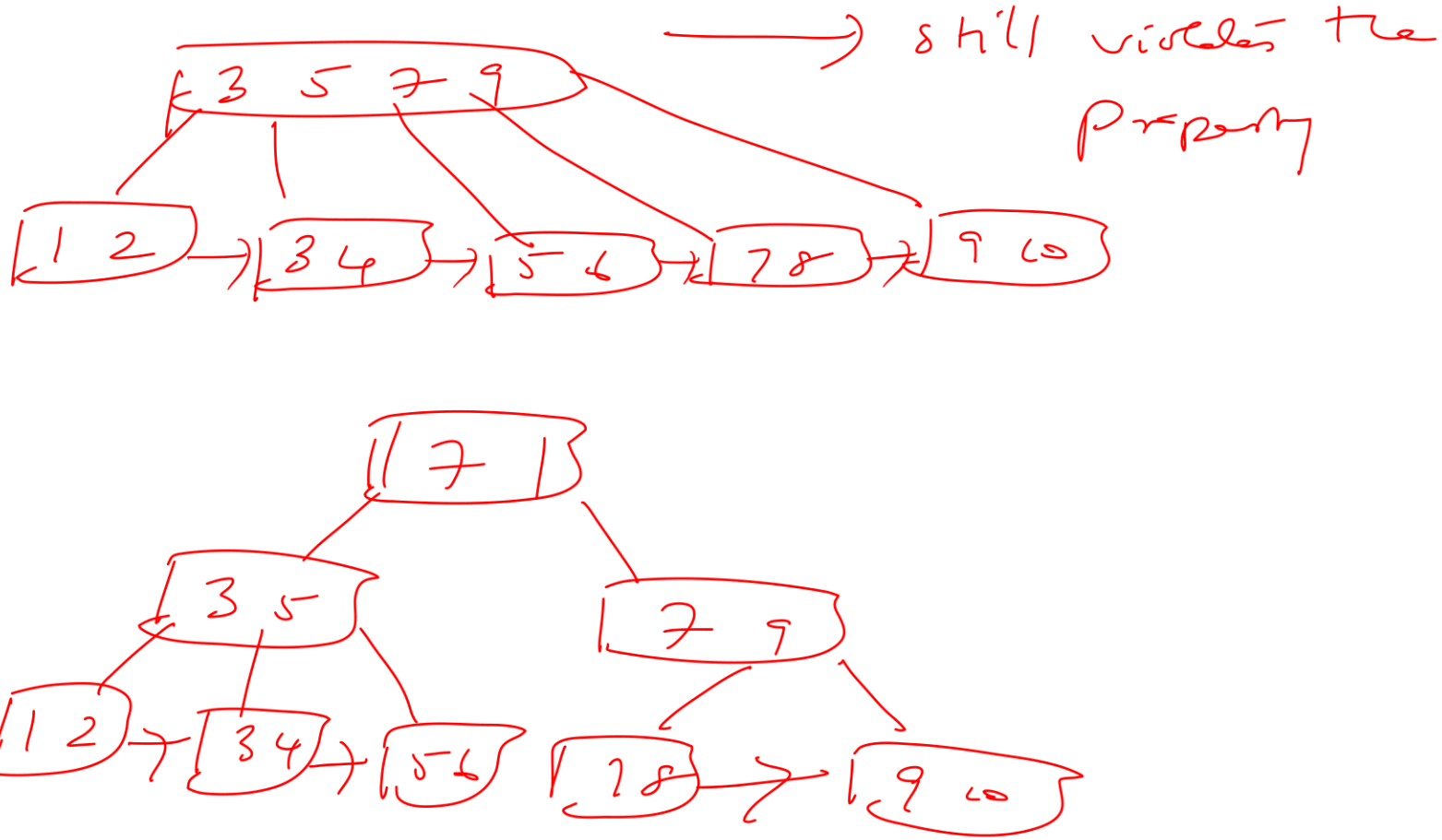
→ violate property



9

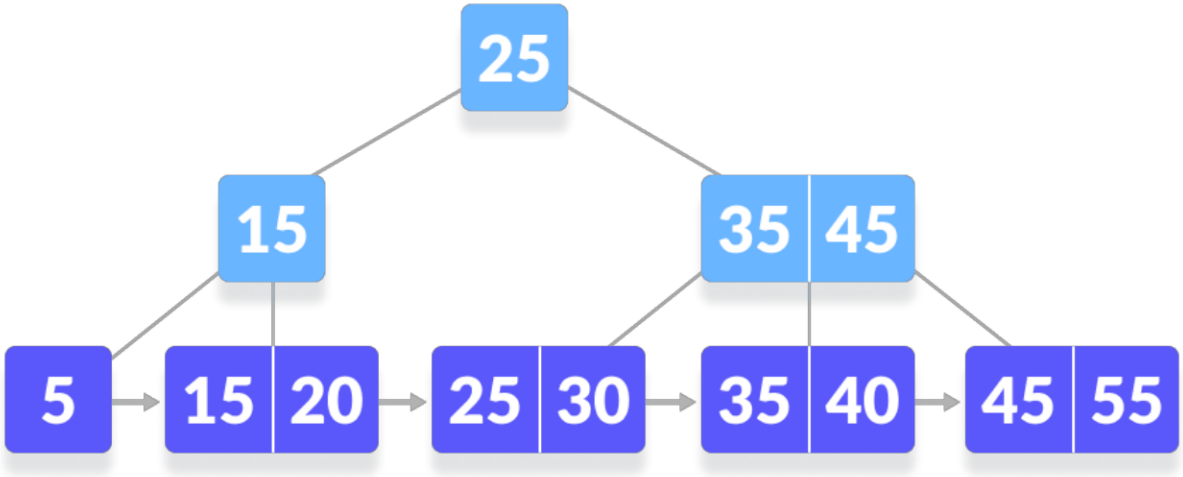


10


→ still violates the property
3 5 7 9
1 2
3 4
5 6
7 8
9 10
7
3 5
7 9
1 2
3 4
5 6
7 8
9 10

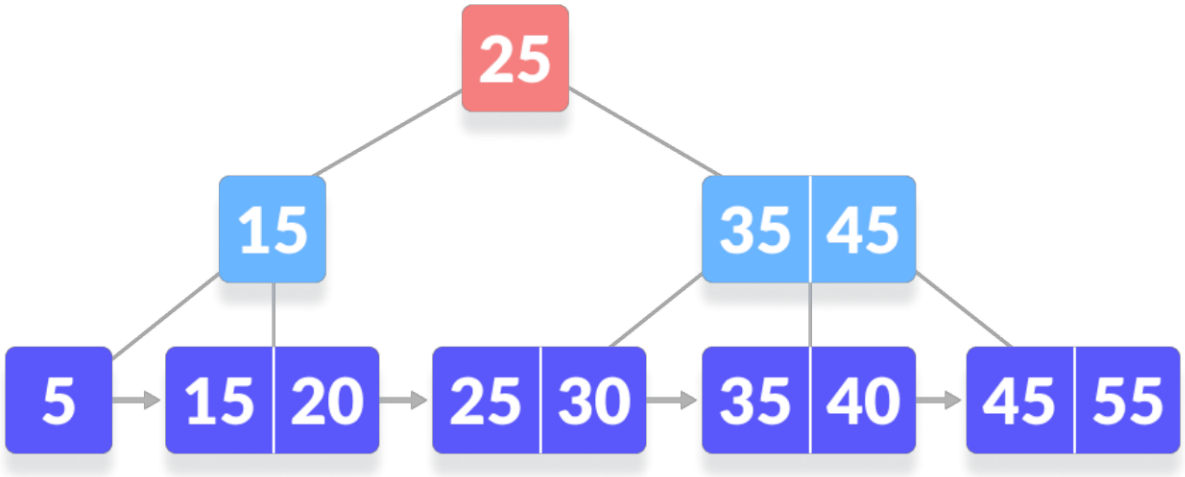
B+ Tree Searching

- **Searching Example on a B+ Tree**
- Let us search k = 45 on the following B+ tree.



B+ Tree Searching

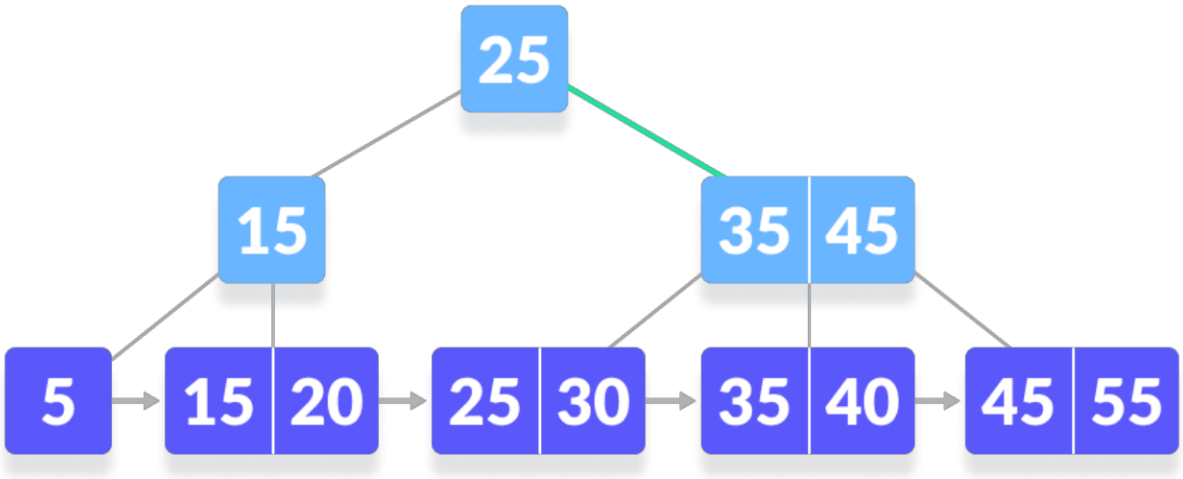
Compare k with the root node, k is not found at the root



B+ Tree Searching

Since k > 25, go to the right child.

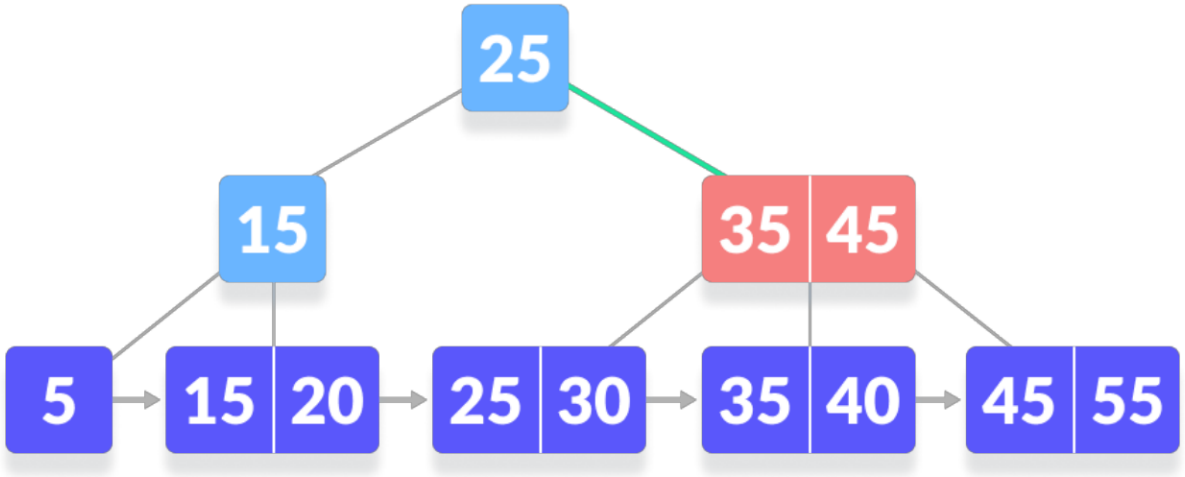
Go to right of the root



B+ Tree Searching

Compare k with 35. Since k > 30, compare k with 45.

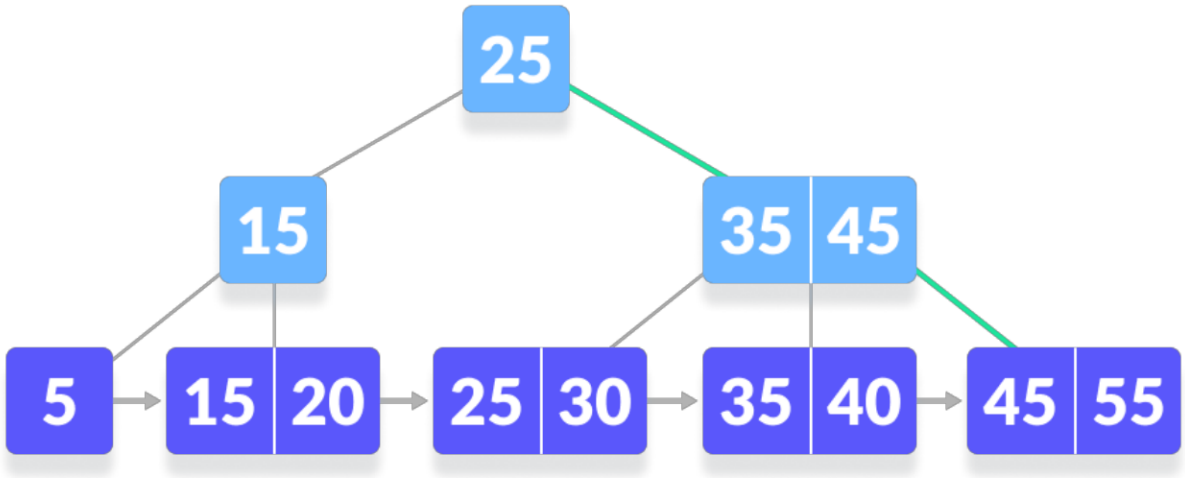
k not found



B+ Tree Searching

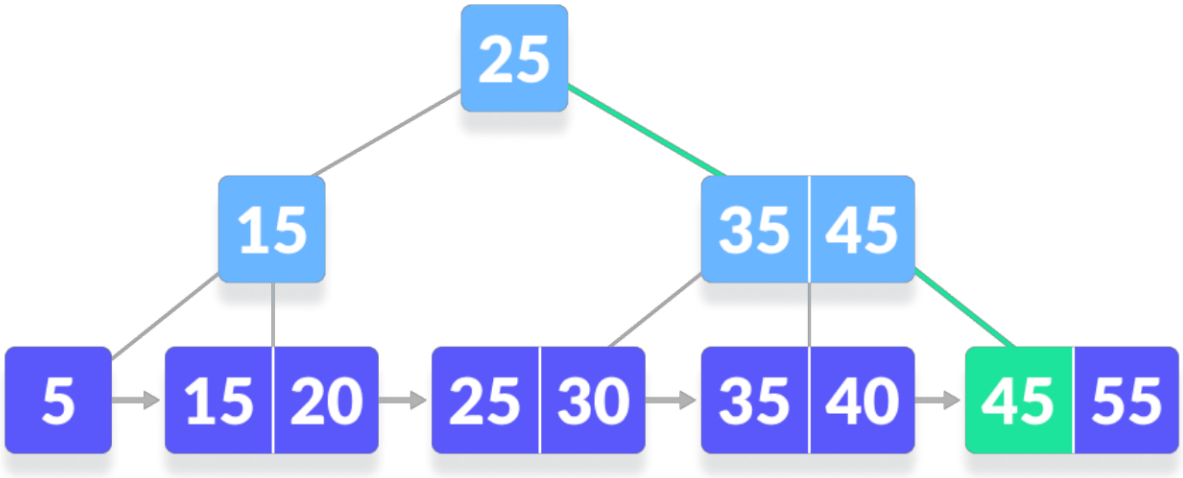
Since k ≥ 45, so go to the right child.

go to the right



B+ Tree Searching

k is found



B+Tree –Time Complexity

Operation	Time Complexity
Search	O(log n)
Insert	O(log n)
Delete	O(log n)
Create	O(nlogn).

B-Tree vs.B+ Tree

B + Tree	B Tree
Search keys can be repeated.	Search keys cannot be redundant.
Data is only saved on the leaf nodes.	Both leaf nodes and internal nodes can store data
Data stored on the leaf node makes the search more accurate and faster.	Searching is slow due to data stored on Leaf and internal nodes.
Linked leaf nodes make the search efficient and quick.	You cannot link leaf nodes.