UNIT- III

NON LINEAR DATA STRUCTURES - TREES

Tree ADT - tree traversals - Binary Tree ADT- expression trees - application of trees - Binary Search Tree ADT- Threaded Binary Trees - AVL trees - B-Tree - B+ Tree - Heap- Applications of heap.

— * — * — * — * —

Tree ADT:-

A tree is recursively defined as a set of one or more nodes, where one node is designated as the root of the tree. and all the remaining nodes can be partitioned into non-empty sets each of which is a sub-tree of the root.

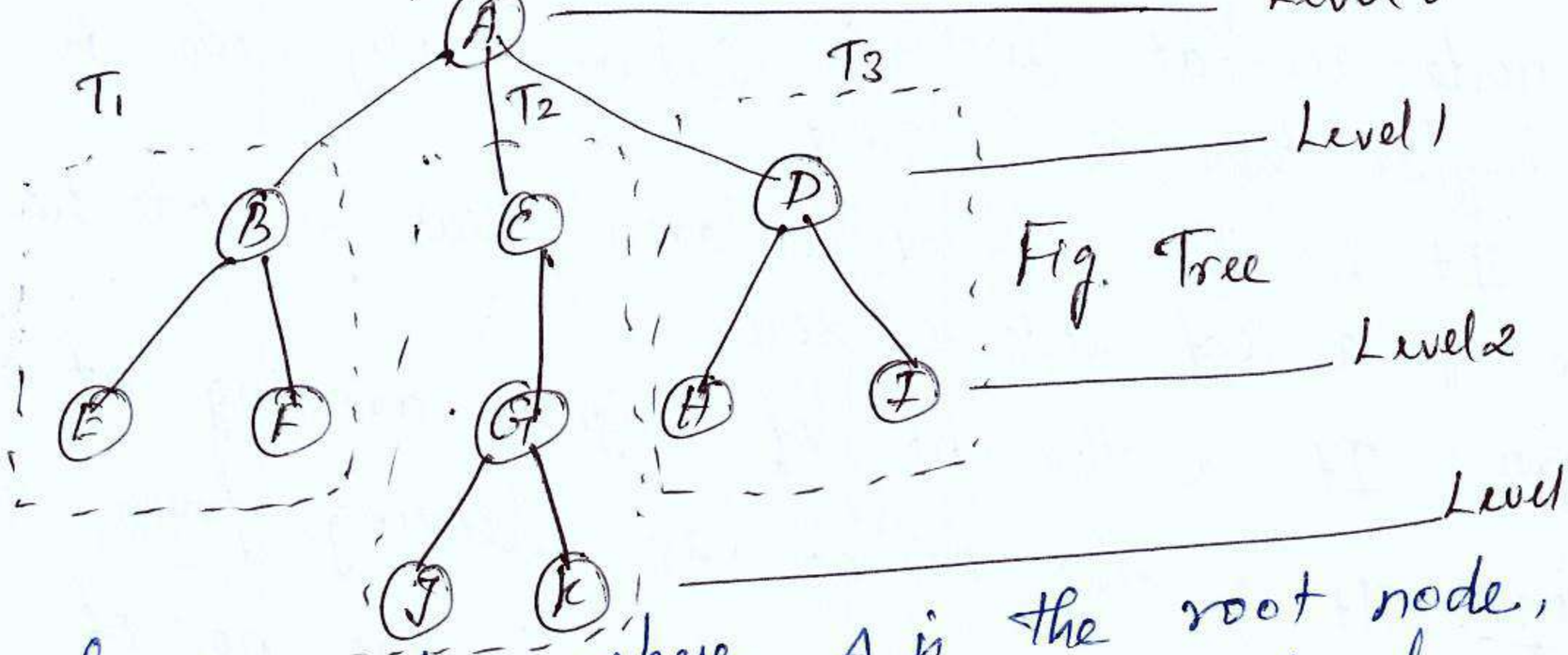


Fig. Tree

Fig. shows a tree, where A is the root node, nodes B, C, and D are children of the root node and form sub-trees of the tree rooted at node A.

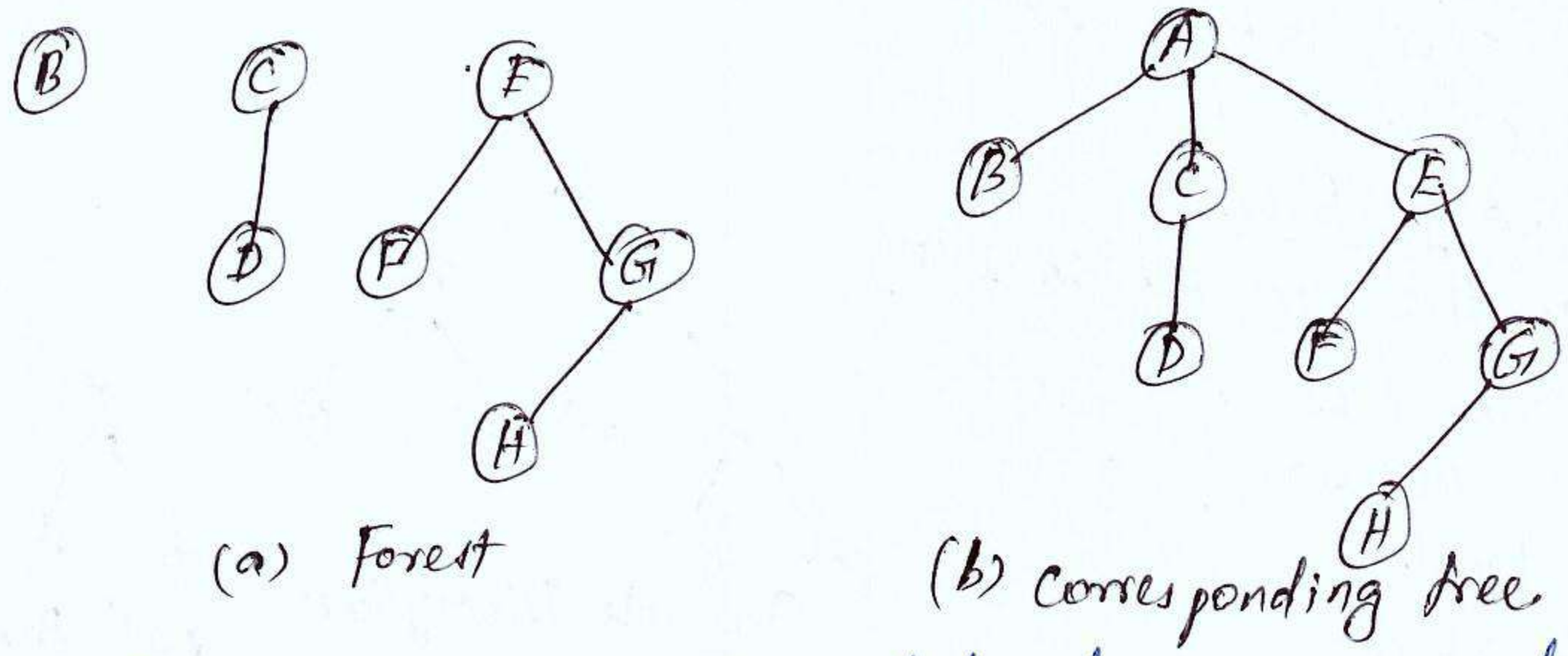
Basic Terminologies:

* **Root Node:** The root node R is the topmost node in the tree. If R = NULL, then the tree is empty.
* **Sub-trees:** If the root node R is not NULL, then the trees. T_1, T_2, and T_3 are called the subtrees of R.

* Leaf node: A node that has no children is called the leaf node or the terminal node.
* Path: A sequence of consecutive edges is called a path. For example, in Fig., the path from A to I is A, D and I.
* Ancestor node: An ancestor of a node is any predecessor node on the path from root to that node. The root node does not have any ancestors. For example, in Fig, nodes A, E and G are the ancestors of node K.
* Descendant node: A descendant node is any successor node on any path from the node to a leaf node. Leaf nodes do not have any descendants. For example, in Fig, nodes C, G, J and K are the descendants of node A.
* Level: Every node in the tree is assigned a level in such a way that the root node is at level 0, children of the root node are at level 1. Thus, every node is at one level higher than its parent.
* Degree: It is the no. of children that a node has. The degree of a leaf node is zero.
* In-degree: It is the no. of edges arriving at a particular node.
* Out-degree: It is the no. of edges leaving from a node.
* Length: The length of the path is the no. of edges on the path. There is a path of length zero from every node to itself.
* Depth: For any node, the depth is the length of the unique path from the root to that node. Thus, the root is at depth 0.
* Height: It is the length of the longest path from a node to a leaf. Thus, all leaves are at height 0. The height of a tree is equal to the height of the root.

Foresh:

A forest is a disjoint union of trees. A set of disjoint trees (or forests) is obtained by deleting the root and the edges connecting the root node to nodes at level 1.



(a) Forest (b) Corresponding tree

A forest can also be defined as an ordered set of zero or more general trees.

We can convert a forest into a tree by adding a single node as the root node of the tree.

Binary Tree ADT;-

A binary tree is a tree in which no node can have more than two children. In a binary tree, the topmost element is called the root node, and each node has 0, 1 or at the most 2 children.

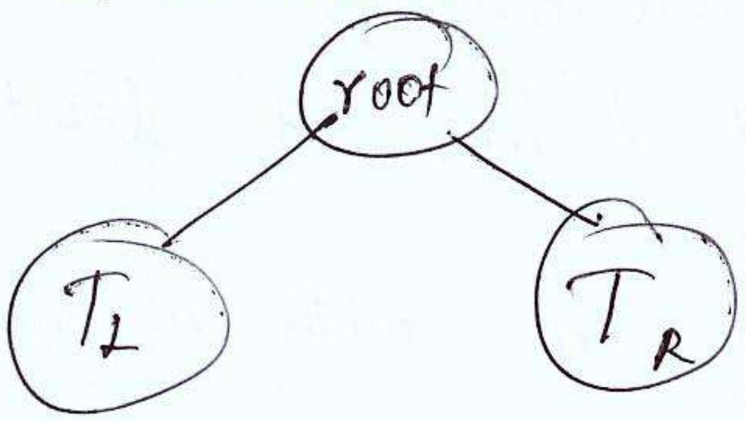


Fig. Generic Binary Tree

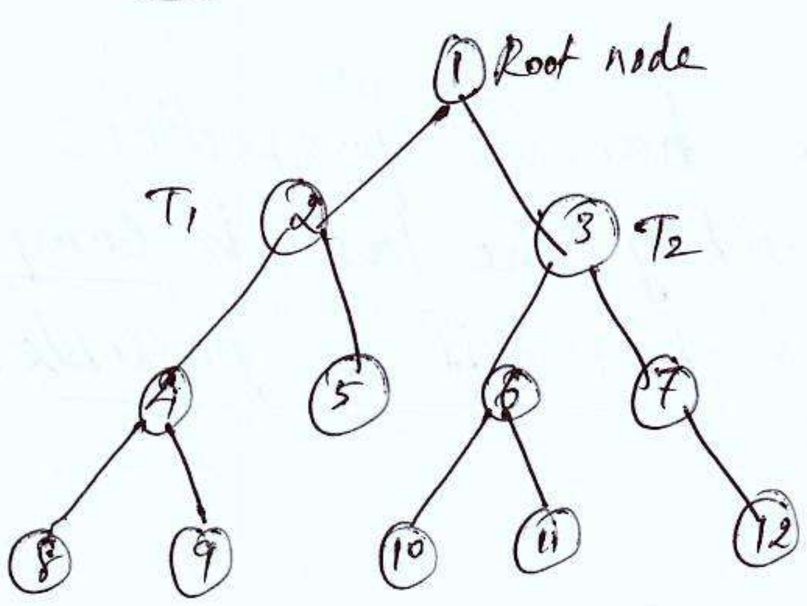
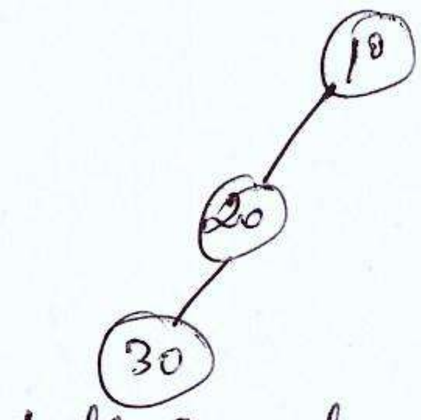


Fig. Binary Tree

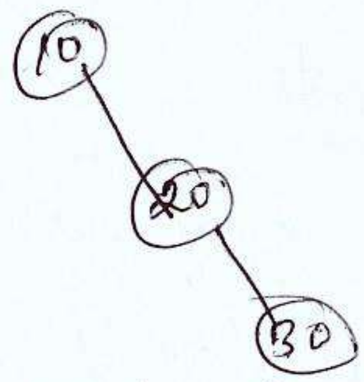
Types of Binary Trees:

1) Left Skewed Binary Tree:
A binary tree, which has only left child nodes



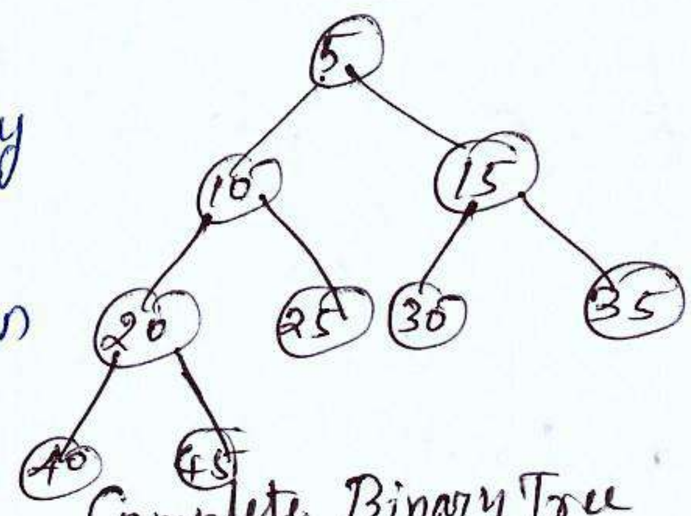
Left Skewed Binary Tree

2) Right Skewed Binary Tree:
A binary tree, which has only right child nodes.



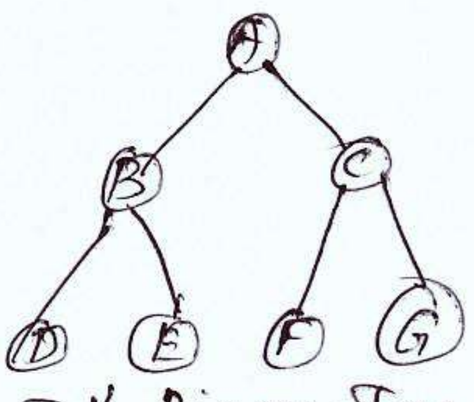
Right Skewed Binary Tree

3) Complete Binary Tree:
A binary tree in which every non-leaf nodes has exactly 2 children not necessarily to be on the same level.



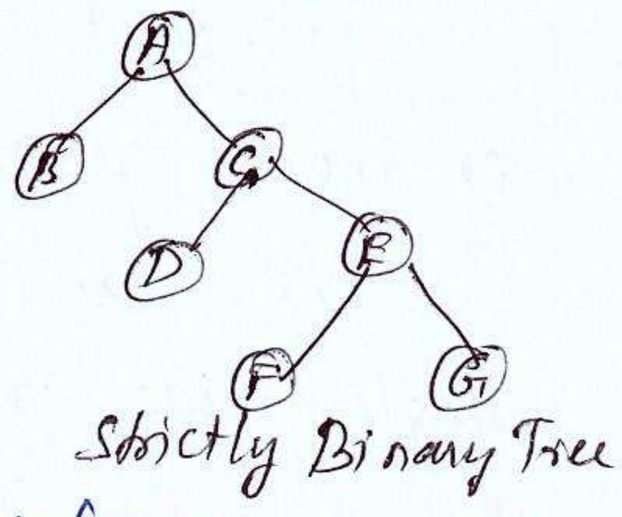
Complete Binary Tree

4) Full Binary Tree:
A binary tree is which all leaves are on the same level and every non-leaf node has exactly 2 children.



Full Binary Tree

* Every node has zero or 2 children.
* The no. of nodes $n = 2^{h+1} - 1$ ⇒ where h is the height of the binary tree



Strictly Binary Tree

5) Strictly Binary Tree:
A strictly binary tree in which every node will have either two children or no child.

* Sibling: All nodes that are at the same level and share the same parent are called siblings.
* A binary tree of height 'h' has at least 'h' nodes and at most $2^h - 1$ nodes.
* The height of a binary tree with 'n' nodes is at least $\log_2(n+1)$ and at most n.
* A complete binary tree has (2) properties:
  1). Every level, except possibly the last, is completely filled.
  2). All nodes appear as far left as possible.

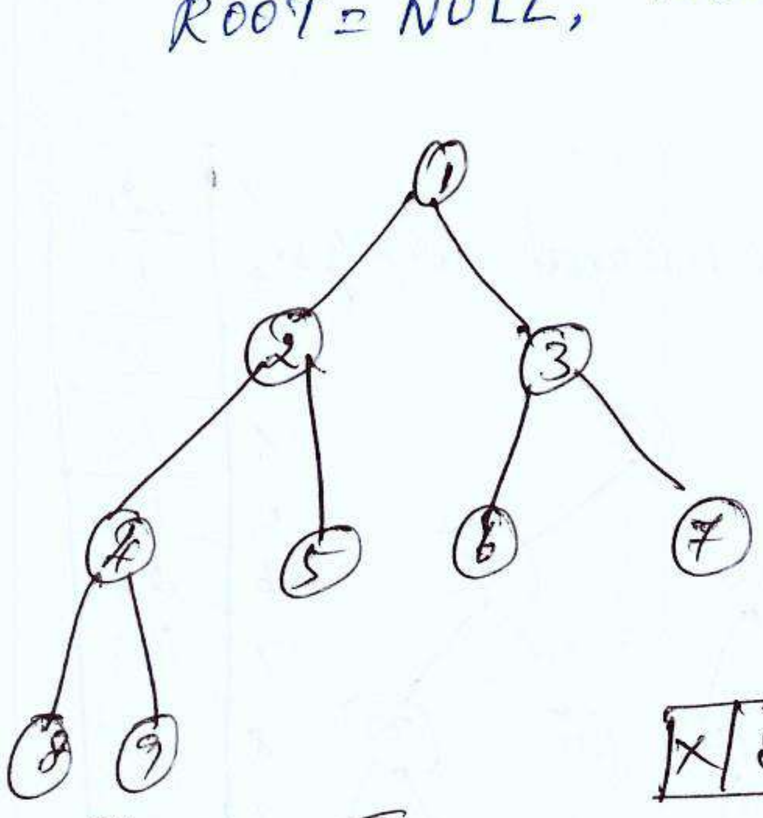
Representation of Binary Trees :-

1) Linked representation:

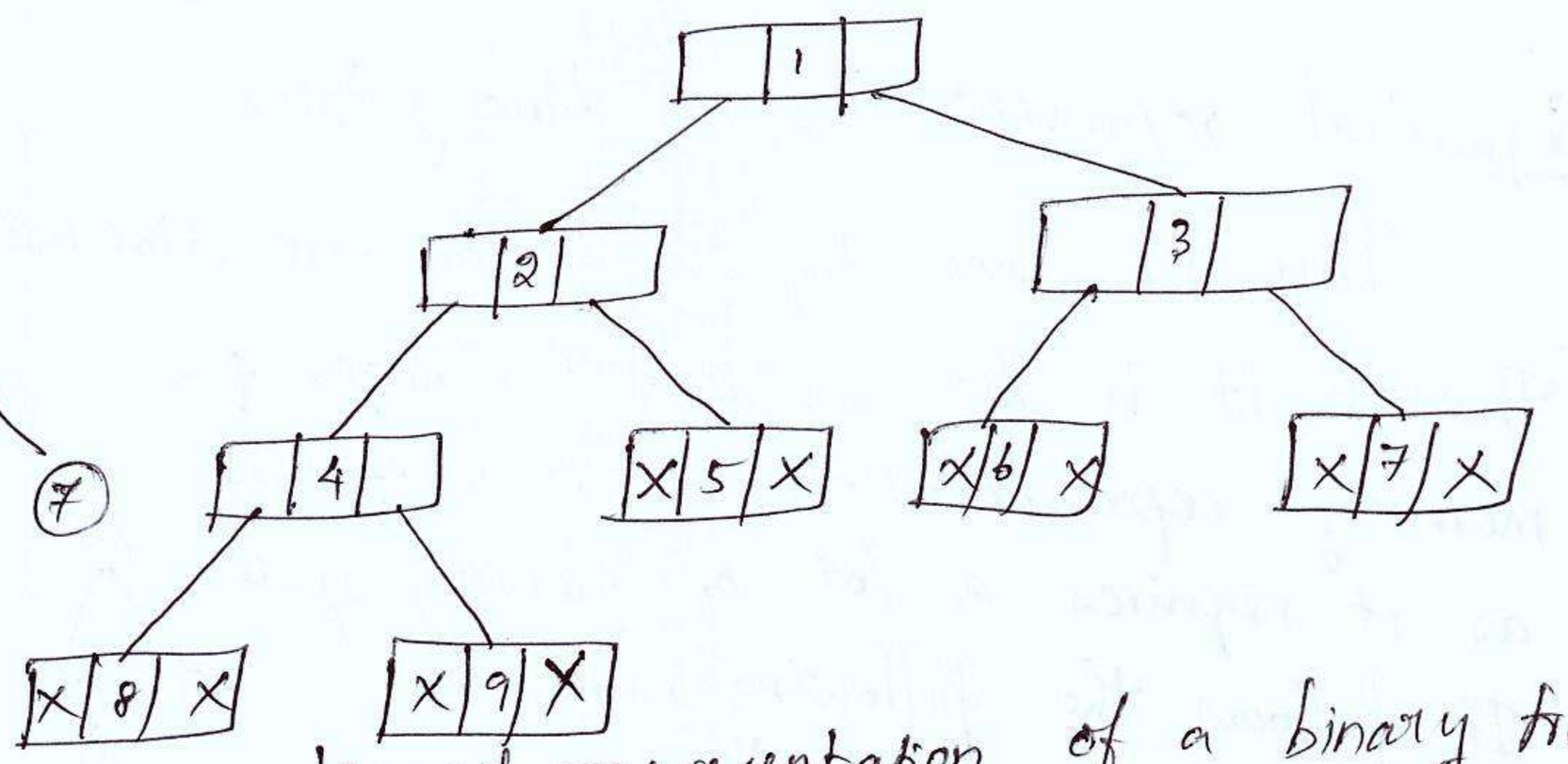
In the linked representation of binary tree, every node consists of (3) parts
- data element
- a pointer to the left node
- a pointer to the right node

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

Every binary tree has a pointer ROOT, which points to the root element (topmost element) of the tree. If ROOT = NULL, then the tree is empty.



Binary Tree



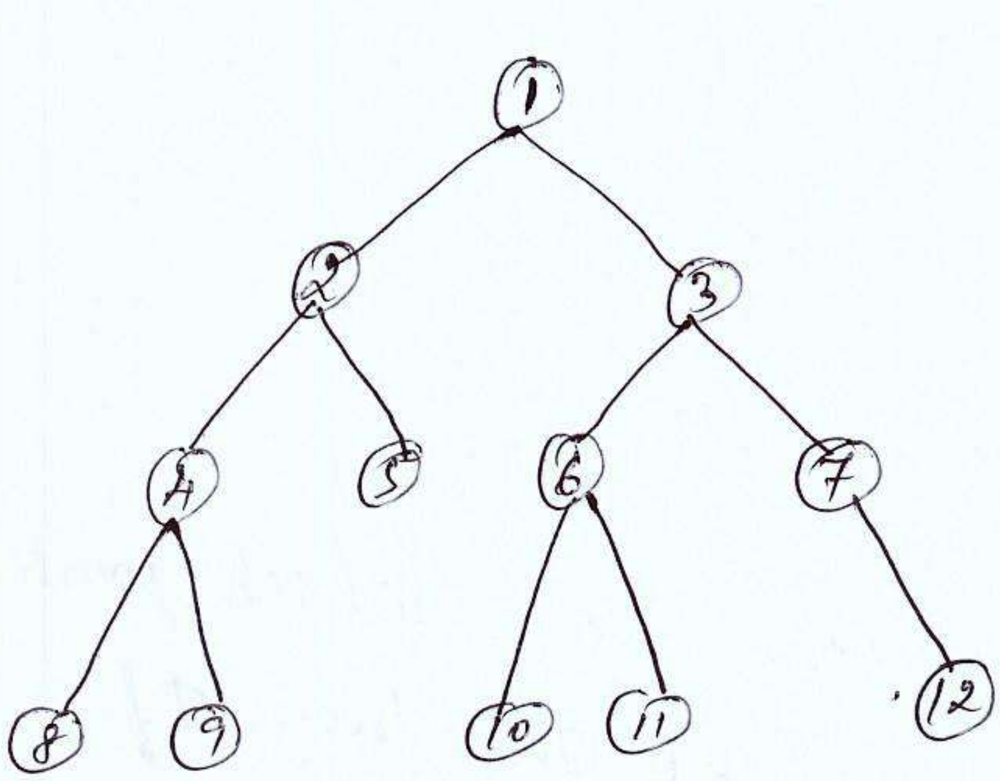
Linked representation of a binary tree

In the above Fig, the left position is used to point to the left child of the node or to store the address of the left child of the node.

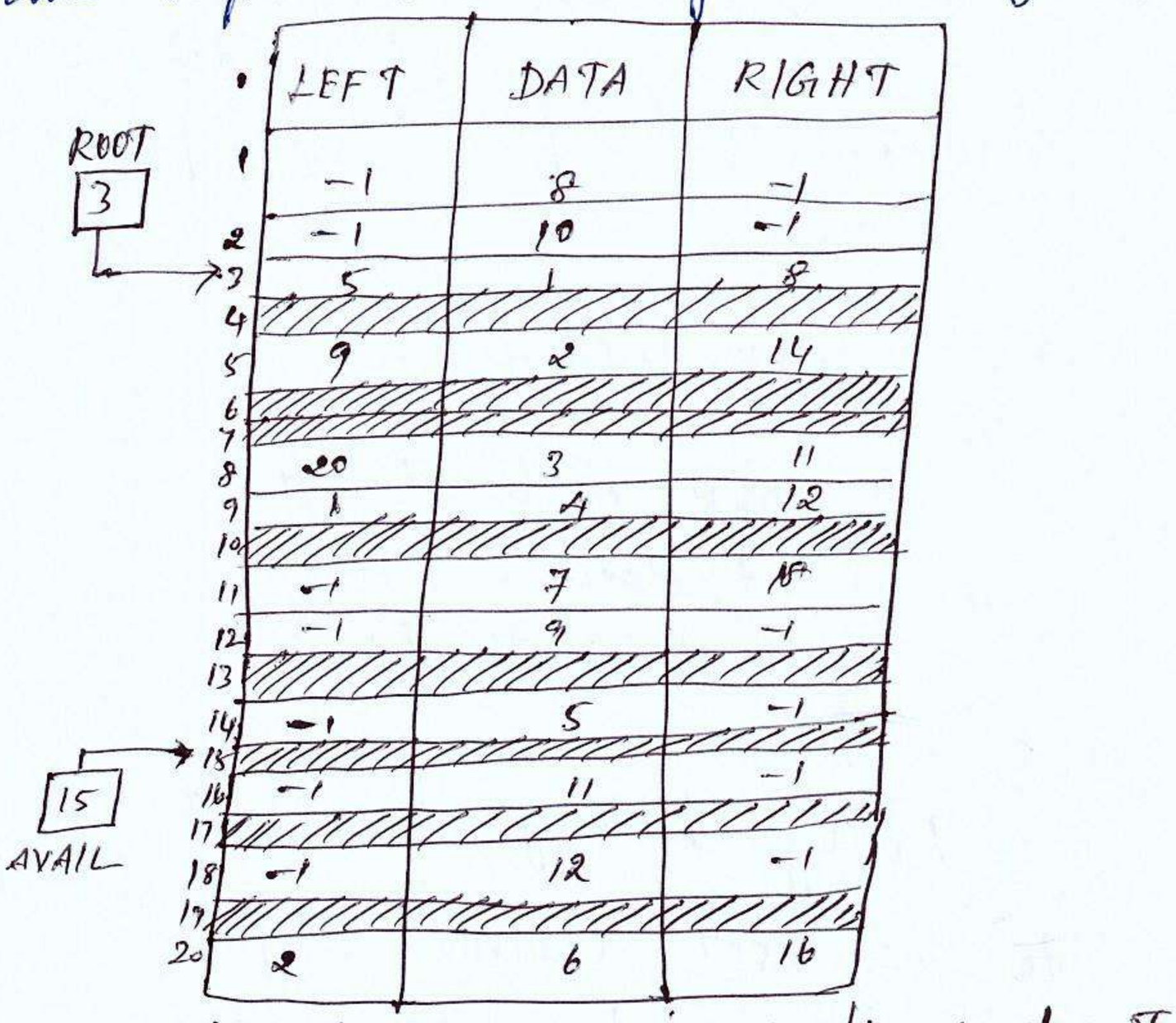
* The middle position is used to store the data.
* Finally, the right position is used to point to the

right child of the node or to store the address of the right child of the node.

* Empty sub-trees are represented using X (meaning NULL).



Binary Tree T

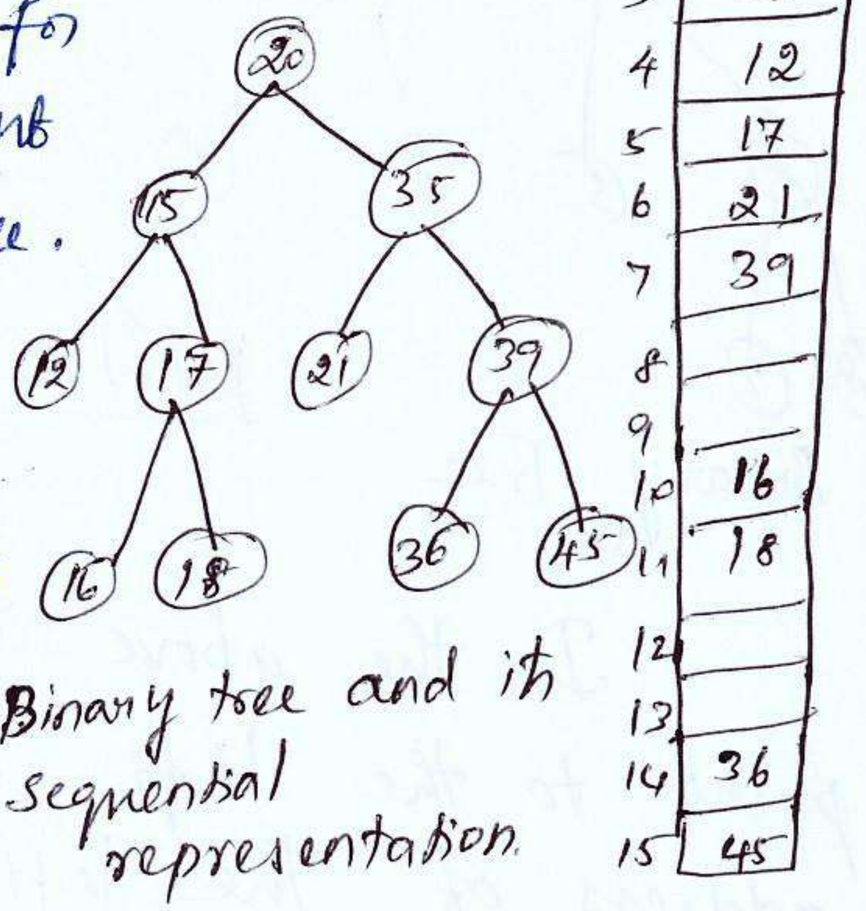


Linked representation of binary tree T

Sequential representation of binary trees:

This is done by single or one-dimensional arrays. Though it is the simplest technique for memory representation, it is inefficient as it requires a lot of memory space. It follows the following rules:

* A one-dimensional array, called TREE, is used to store the elements of tree.
* The root of the tree will be stored in the first location. ie, TREE[1]
* The children of a node stored in location k will be stored in locations (2*k) and (2*k+1).



Binary tree and its sequential representation

* The maximum size of the array TREE is $(2^h - 1)$, where h is the height of the tree.
* An empty tree or sub-tree is specified ming NULL. If TREE[i] = NULL, then the tree is empty.

Tree Traversals:-

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited.

1) In-order Traversal
2) Pre-order Traversal
3) Post-order Traversal.

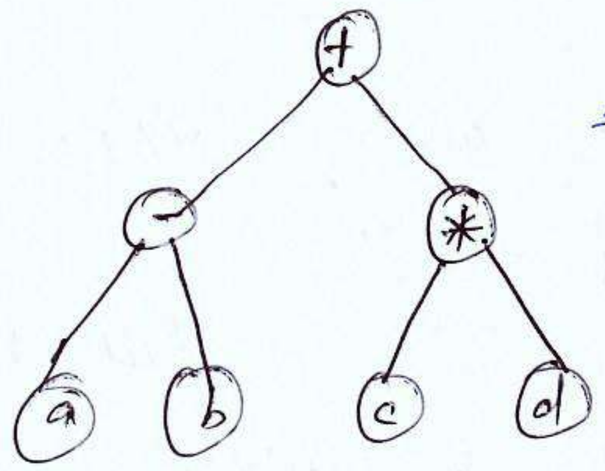
1) Pre-order Traversal:-

To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node.

(i) Visiting the root node
(ii) Traversing the left sub-tree, and finally
(iii) Traversing the right sub-tree.

This pre-order traversal is also called as depth-first traversal.

* In this alg, the left subtree is always traversed before the right subtree.
* This alg is also known as NLR (Node-Left-Right) traversal alg.
* Pre-order traversal algs are used to extract a prefix notation from an expression tree.
* When we traverse the elements of a tree ming the pre-order traversal alg, the expression we get is a prefix expression.



⇒ + - a b * c d ⇒ prefix expression

Alg:

Step 1: Repeat steps 2 to 4 while TREE != NULL

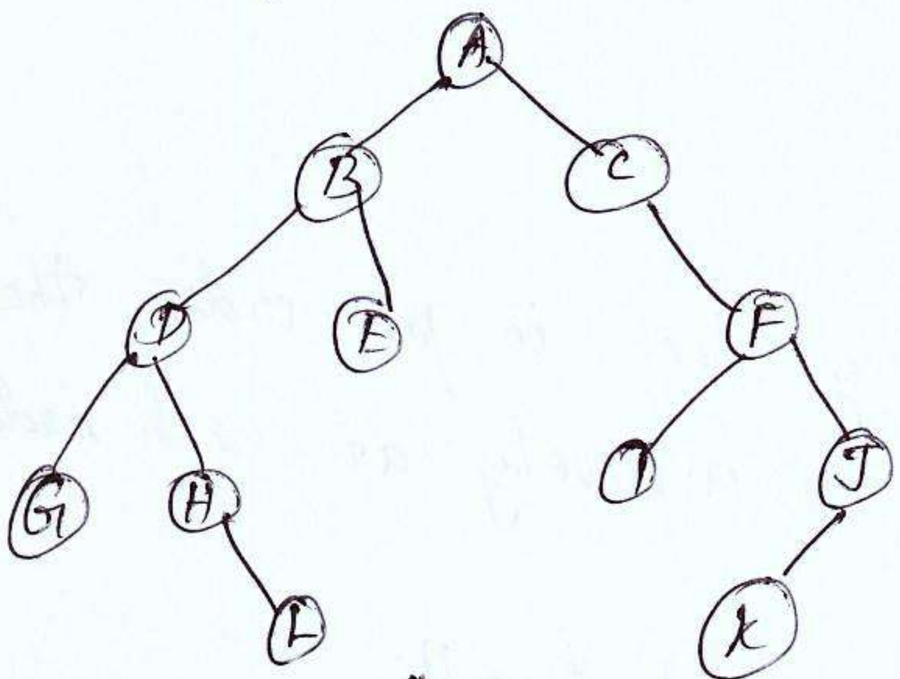
Step 2: write TREE → DATA

Step 3: PREORDER (TREE → LEFT)

Step 4: PREORDER (TREE → RIGHT)

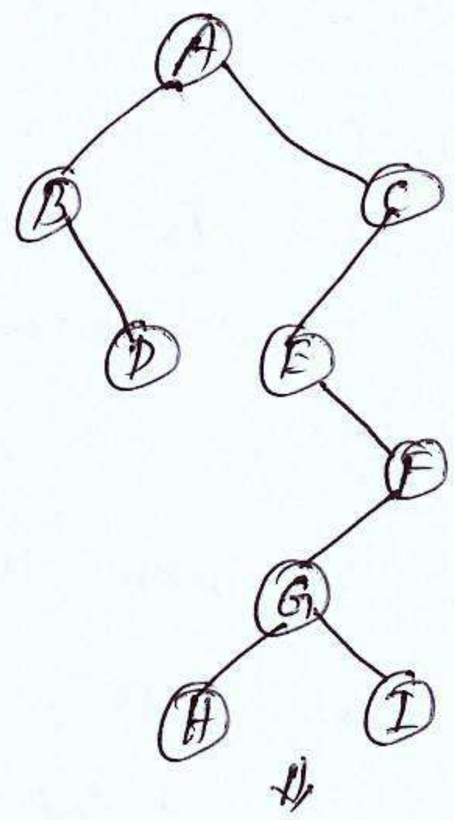
[END OF LOOP]

Step 5: END



⇓

prefix/
pre-order traversal
A, B, D, G, H, L, E, C, F, I, J, K



⇓

prefix/
pre-order travel
A, B, D, C, E, F, G, H, I.

2) In-order Traversal:-

To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node.

(i) Traversing the left sub-tree
(ii) Visiting the root node, and finally
(iii) Traversing the right sub-tree.

* In-order traversal is also called as <u>symmetric traversal</u>.
* In this alg, the left subtree is always traversed before the root node and right subtree.
* The word 'in' specifies that the root node is accessed in between the left and right subtrees.
* This is also known as the LNR traversal alg.

(Left - Node - Right)

Alg:

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: INORDER (TREE -> LEFT)
Step 3: Write TREE -> DATA
Step 4: INORDER (TREE -> RIGHT)
[END OF LOOP]
Step 5: END

In-order traversal alg is mually used to display the elements of a binary search tree.

* Here, all the elements with a ~~lower~~ value lower than a given value are accessed before the elements with a higher value

In-order traversal for previous example.

In-order/Infix => G, D, H, L, B, E, A, C, I, F, K, J
=> B, D, A, E, H, G, I, F, C.

3) <u>Post-order Traversal</u>;-

To traverse a non-empty binary tree in post-order, the following operations are performed. recursively at each node.

(i) Traversing the left subtree
(ii) Traversing the right subtree, and finally
(iii) Visiting the root node.

In this alg, the left subtree is always traversed before the right subtree and the root node.

* The word 'post' specifies that the root node is accessed after the left and right subtrees.

* This is also known as LRN traversal alg.
(Left - Right - Node)

* This is used to extract postfix notation from an expression tree.

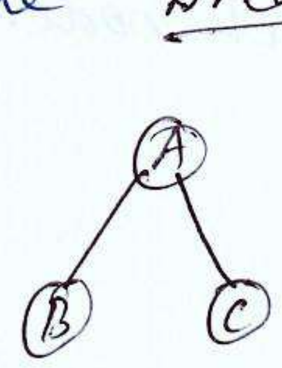
Alg:

Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2: POSTORDER (TREE → LEFT)
Step 3: POSTORDER (TREE → RIGHT)
Step 4: Write TREE → DATA
[END OF LOOP]
Step 5: END

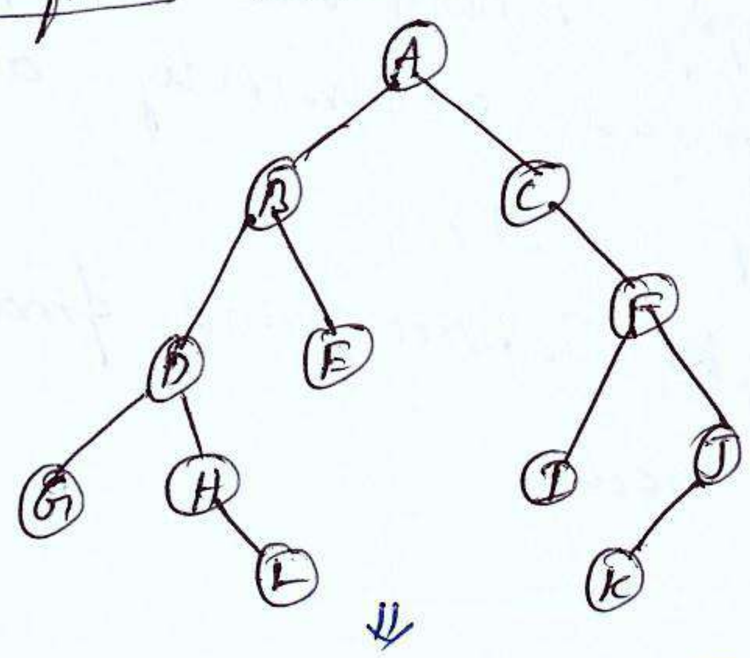
Post order traversal ⇒ G, L, H, D, E, B, I, K, J, F, C, A
⇒ D, B, H, I, G, F, E, C, A.

4) Level-order Traversal:-

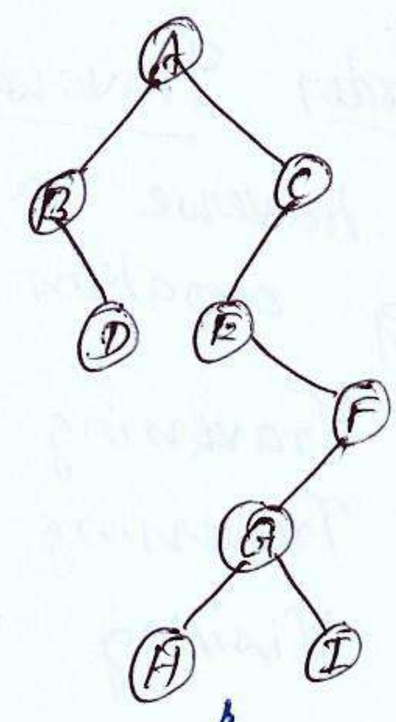
In level-order traversal, all the nodes at a level are accessed before going to the next level. This alg is also called the breadth-first traversal alg.



A, B, C



A, B, C, D, E, F, G, H, I, J, L, K



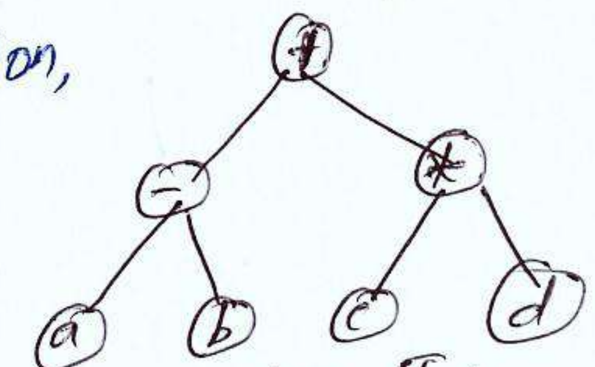
A, B, C, D, E, F, G, H, I

Expression Tree:-

Binary trees are widely used to store algebraic expression. Expression tree is a binary tree in which leaf nodes are operands and the internal nodes are operators.

For example, consider the algebraic expression,

Exp = (a-b) + (c*d)



Expression Tree

Construction of an Expression Tree:

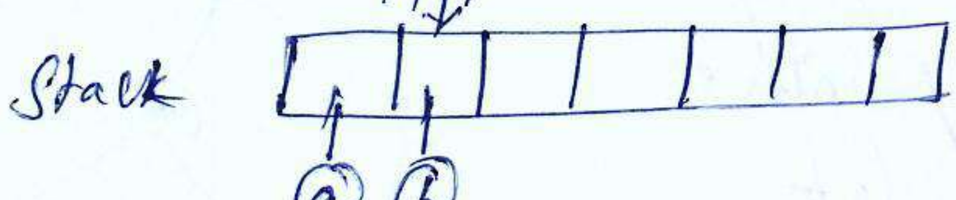
1) Convert the given expression into postfix.
2) Read one symbol at a time from the postfix expression.
3) Check the symbol is an operand or operator
   (i) If the symbol is an operand, create a one node tree and push a point on to the stack.
   (ii) If the symbol is an operator, pop two pointers namely, T_1 and T_2. from the stack. Form a new tree whose root is the operator. T_2 as the left child and T_1 as the right child. A pointer to this new tree is pushed on to the stack.

Example:

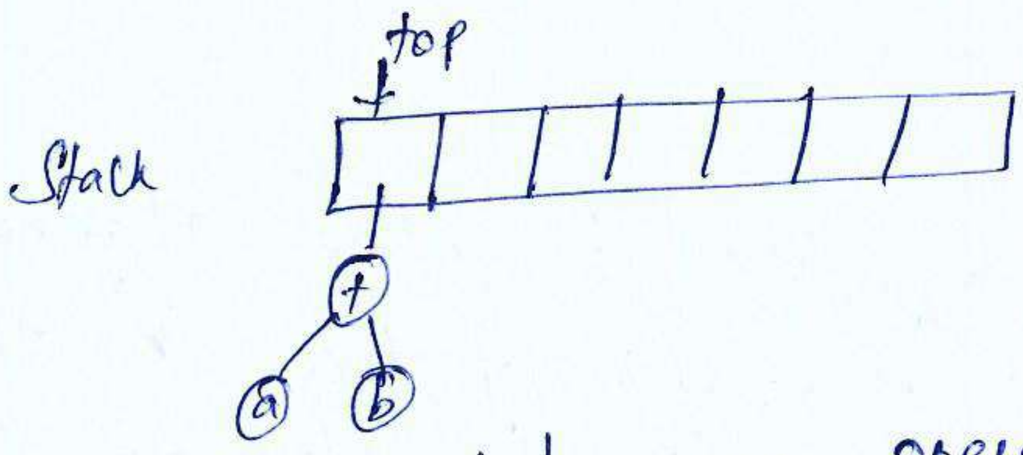
(a+b) * (c * (d+e))

Postfix form ⇒ ab+ cde+**

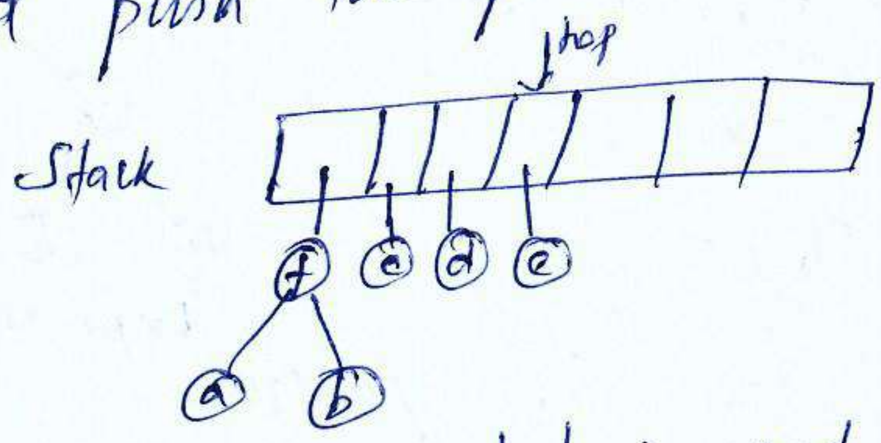
1) The first two symbols are operands, so create one node trees and push the pointers on to the stack.



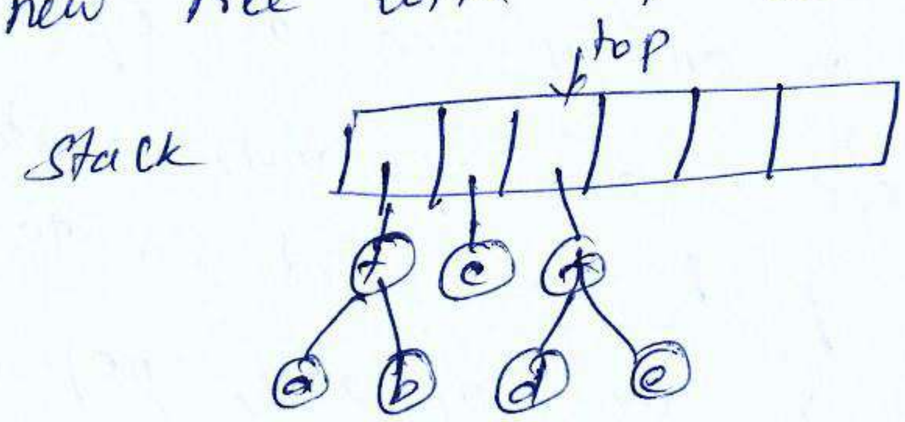
2) Next '+' symbol is read, so the two pointers are popped, a new tree is formed and a pointer to it pushed on to the stack.



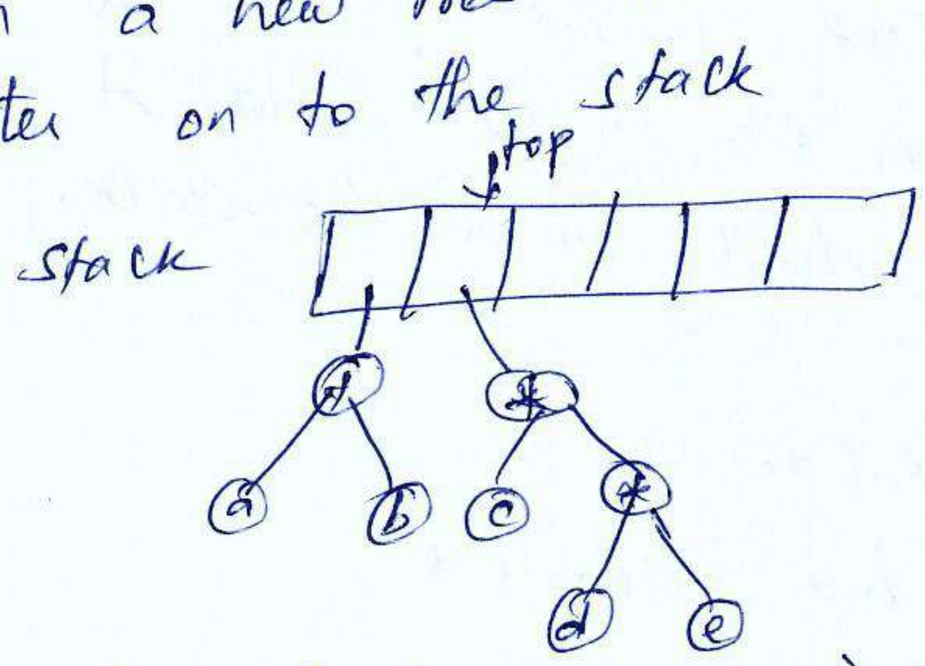
3) Next 3 symbols are operands, so create one node trees and push the pointers on to the stack.



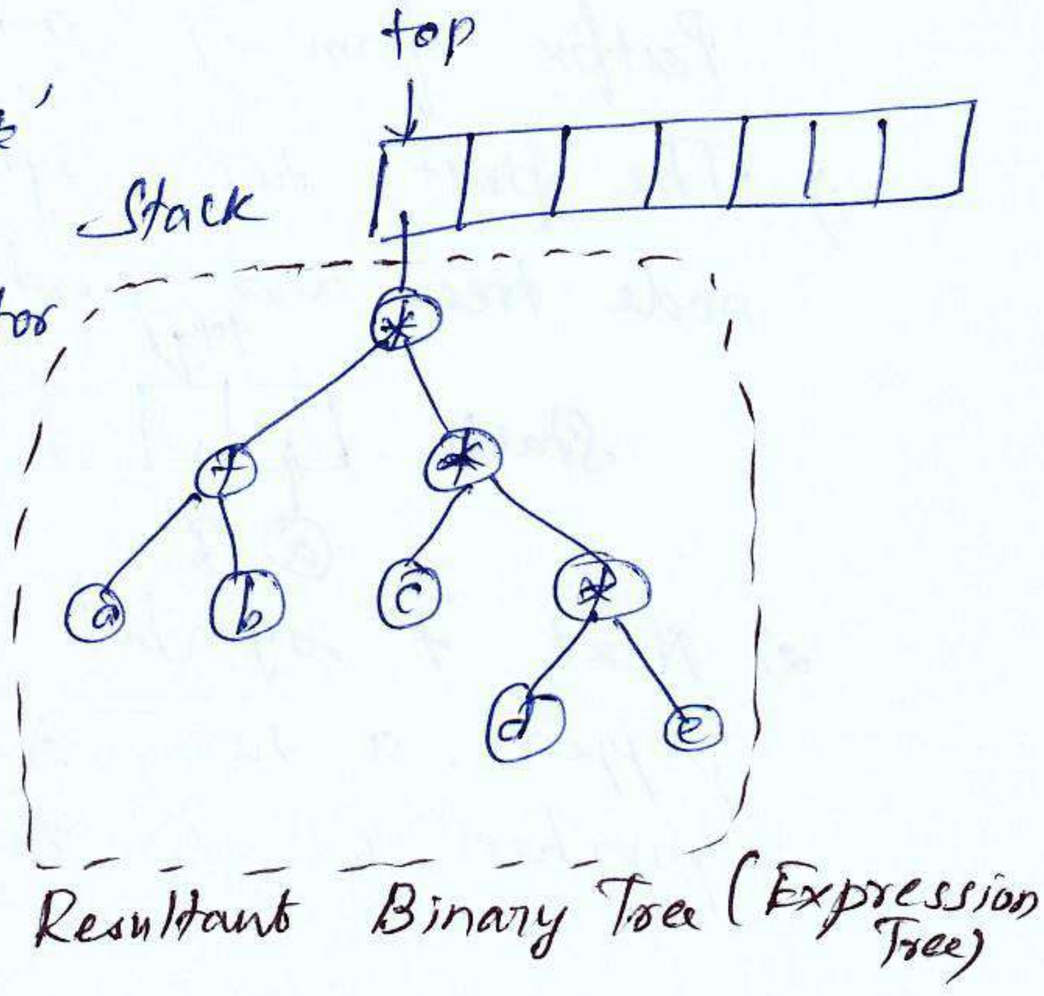
4) Next '+' symbol is read, so pop 2 pointers and form a new tree with '+' as a root node.



5) Next '*' symbol is read, so pop 2 pointers and form a new tree with '*' as root and push the pointer on to the stack



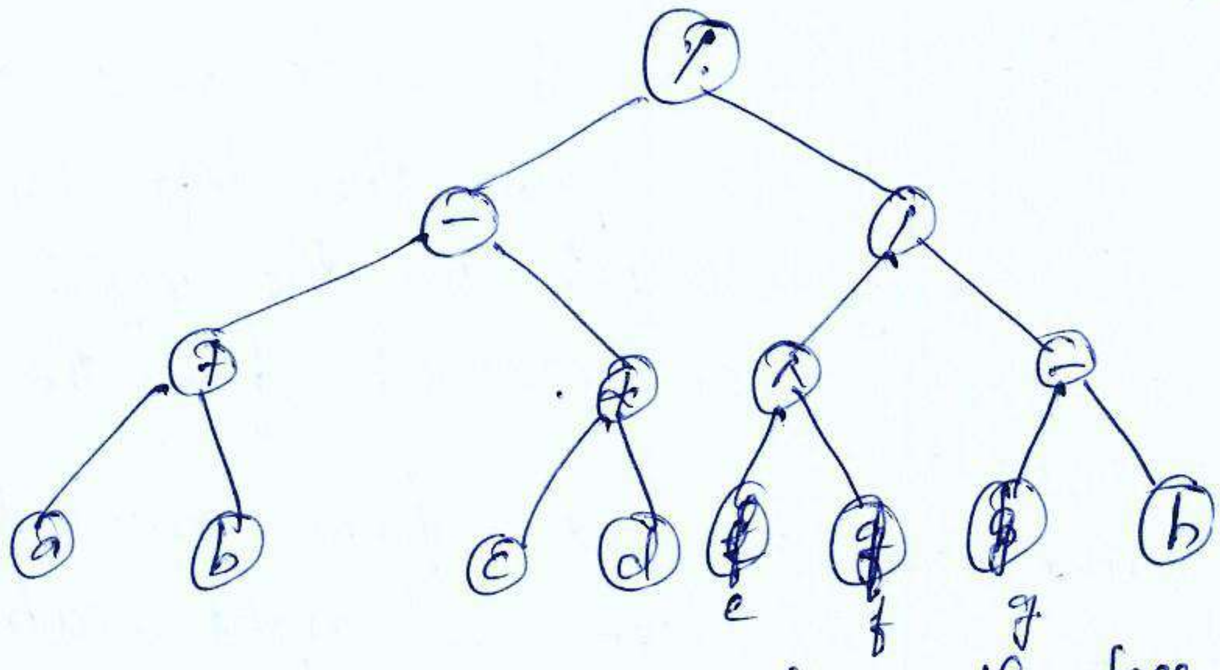
6) Finally, the last symbol '*' is read, therefore, two trees are merged with '*' operator, as root and a pointer to the final tree is left on to the stack.



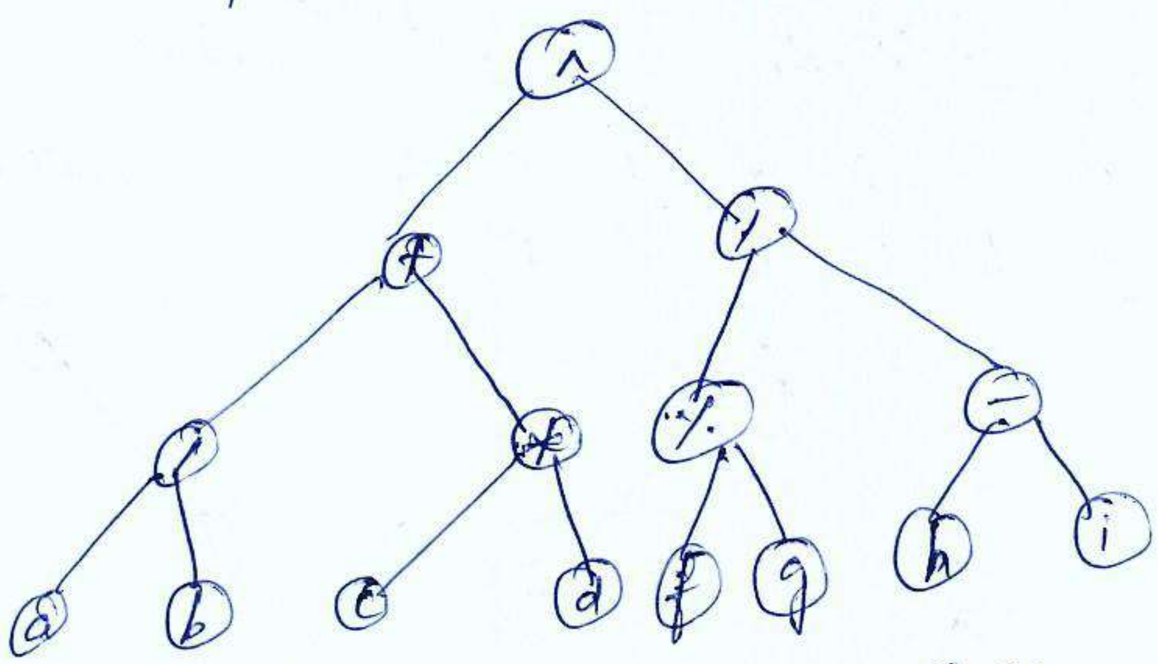
Resultant Binary Tree (Expression Tree)

Examples:

1) Exp ((a+b)-(c*d)) % ((e^f)/(g-h))



2) Find the expression. from the tree



Exp = [{(a/b)+(c*d)} ^ {(f%g)/(h-i)}]

Constructing a Binary Tree from Traversal Results:

We can construct a binary tree if we are given at least two traversal results.

* The first traversal must be the in-order traversal and the second can be either pre-order or post-order traversal.

* The in-order traversal result will be used find the left and right child nodes, and the pre-order/post-order can be used to determine the root node.

For example,

In-order traversal: D B E A F C G Pre-order traversal: A B D E C F G

Follow the steps given to construct the tree:

Step 1: Use the pre-order sequence to determine the root node of the tree. The first element would be the root node.

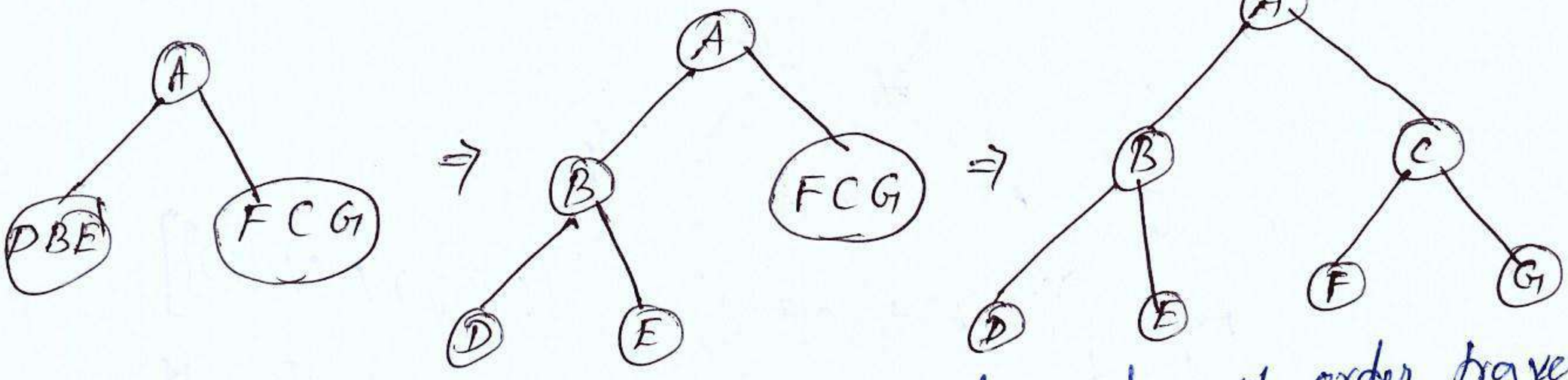
Step 2: Elements on the left side of the root node in the in-order traversal sequence form the left subtree of the root node. Similarly, elements on the right side of the root node in the in-order traversal form the right subtree of the root node.

Step 3: Recursively select each element from pre-order traversal sequence and create its left and right subtrees from the in-order traversal sequence.

Left subtree → | ← Right subtree

In-order Traversal: D B E A F C G A ⇒ root node

Pre-order Traversal: A B D E C F G



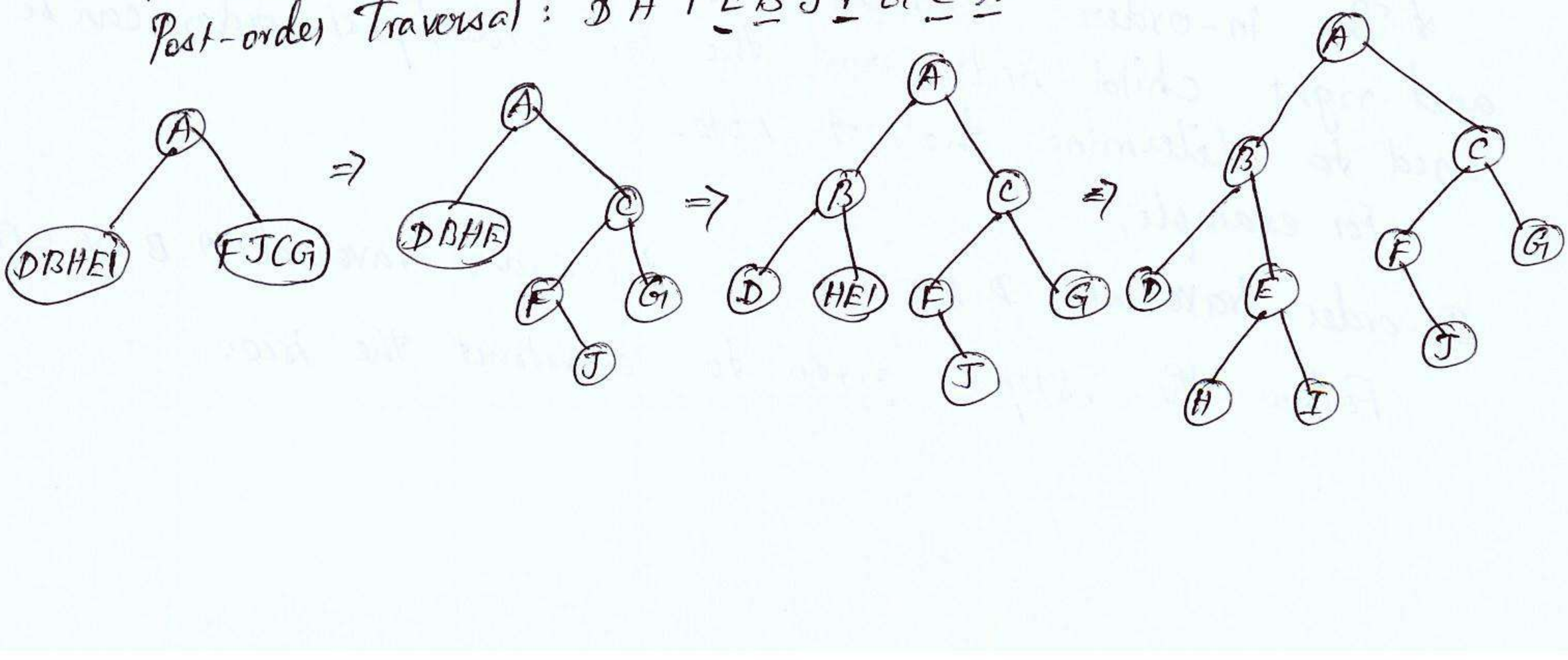
* Now comider the in-order traversal and post-order traversal sequences of a given binary tree.

* Before constructing the binary tree, the last node in post-order traversal becomes the root node. Rest of the steps will be the same as above.

Left subtree → | ← Right subtree

In-order Traversal: D B H E I A F J C G

Post-order Traversal: D H I E B J F G C A



Applications of Trees:-

* Trees are used to store simple as well as complex data.
* Trees are often used for implementing other types of data structures like hash tables, sets, and maps.
* A self-balancing tree, Red-black tree is used in kernel scheduling, to preempt massively multiprocessor computer operating system etc.
* Another variation of tree, B-trees are used to store tree structures on disc. They are used to index a large no. of record
* B-trees are used for secondary indexes in database, where the index facilitates a select operation to answer some range criteria.
* Trees are an important data structure used for compiler construction.
* Trees are also used in database design.
* Trees are also used in file system directories
* Trees are also used for information storage and retrieval in symbol tables.
* Trees are also used in compression algs. such as jpeg and mp3 file formats.
* Trees are also used in cryptographic applns to generate a tree of pseudo-random numbers.
* Expression Trees - arithmetic expressions
* Searching algorithms - searching in external memory, web browsers
* Text processing (dictionary)

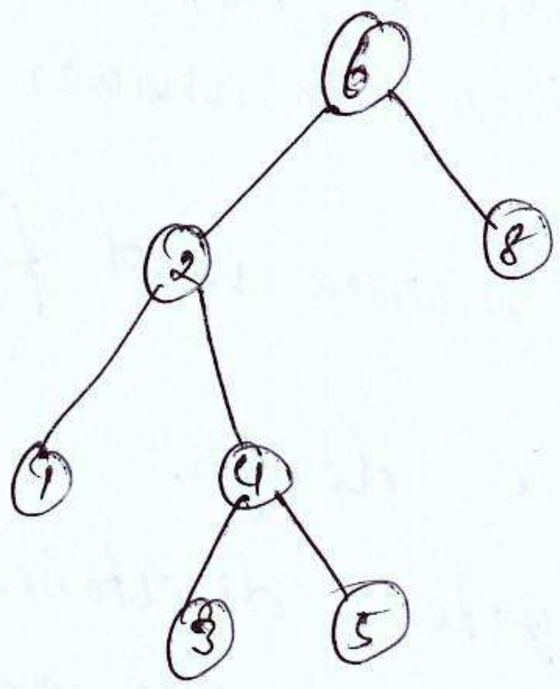
Binary Search Tree ADT:-

A binary search tree is a binary tree, which is either empty or in which each node contains a key that satisfies the following condition:

(i) All keys are distinct

(ii) For every node X, in the tree, the values of all the keys in the left subtree are smaller than the key value in X.

(iii) For every node X, in the tree, the values of all the keys in the right subtree are larger than the key value in X.




```
Struct TreeNode
{
    ElementType Element;
    SearchTree Left;
    SearchTree Right;
};
```

* This is also known as an ordered binary tree, is a variant of binary tree in which the nodes are arranged in an order.

* Since the nodes in a BST (Binary Search Tree) are ordered, the time needed to search an element in the tree is reduced.

* Whenever we search for an element, we do not need to traverse the entire tree.

* At every node, we get a hint regarding which subtree to search in.

* The avg. running time of a search operation is $O(\log_2 n)$, as at every step, we eliminate half of the subtree from the search process.

* BSTs also speed up the insertion and deletion operations. The tree has a speed adv. when the data in the structure changes rapidly.

BSTs are considered to be efficient DS (Data Structures) especially when compared with sorted linear arrays and linked lists.

* In sorted array, searching can be done in $O(\log_2 n)$ time, but insertions and deletions are quite expensive.

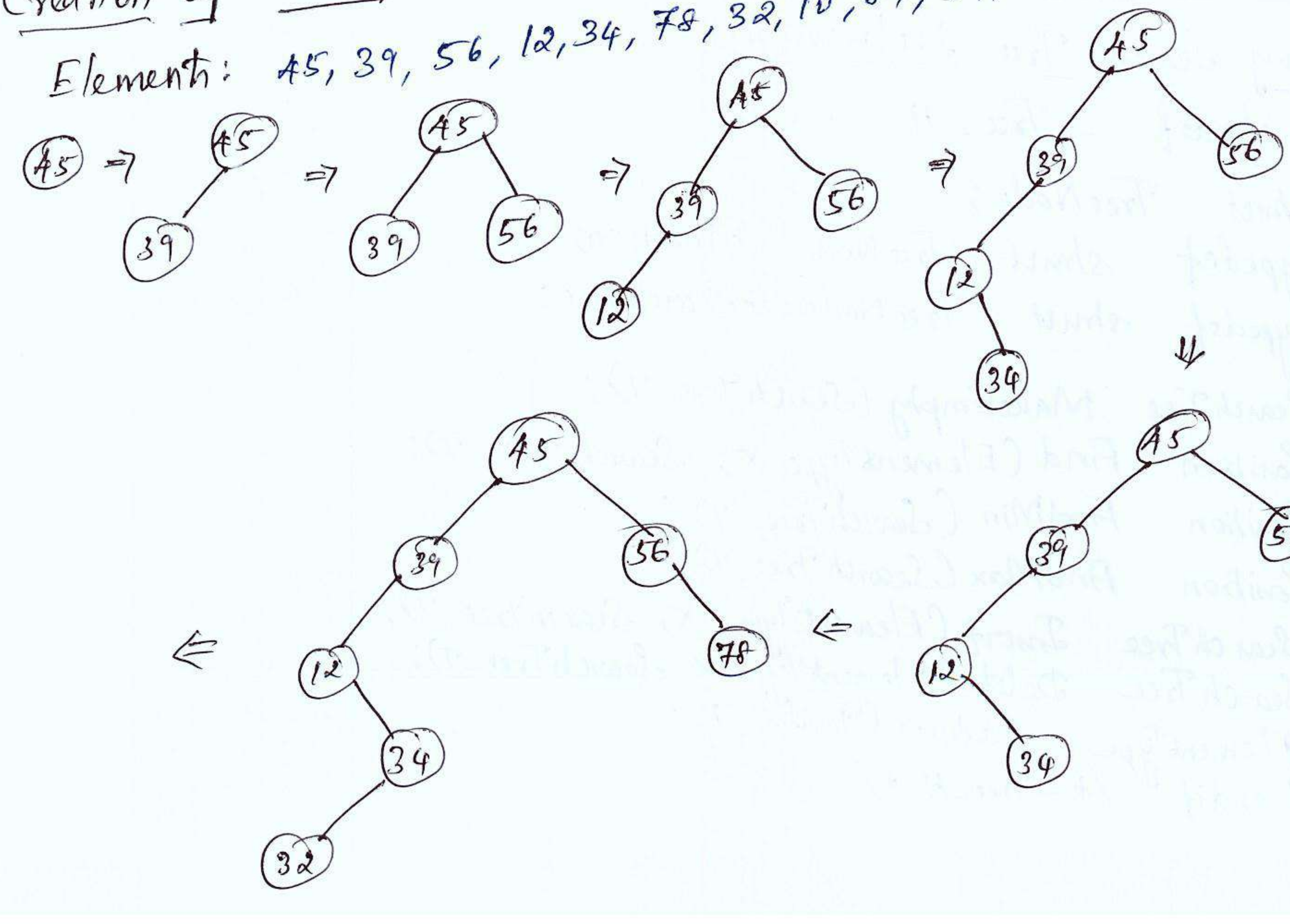
* In contrast to linked list, the insertion and deletion of elements are easy, but searching for an element is done in $O(n)$ time.

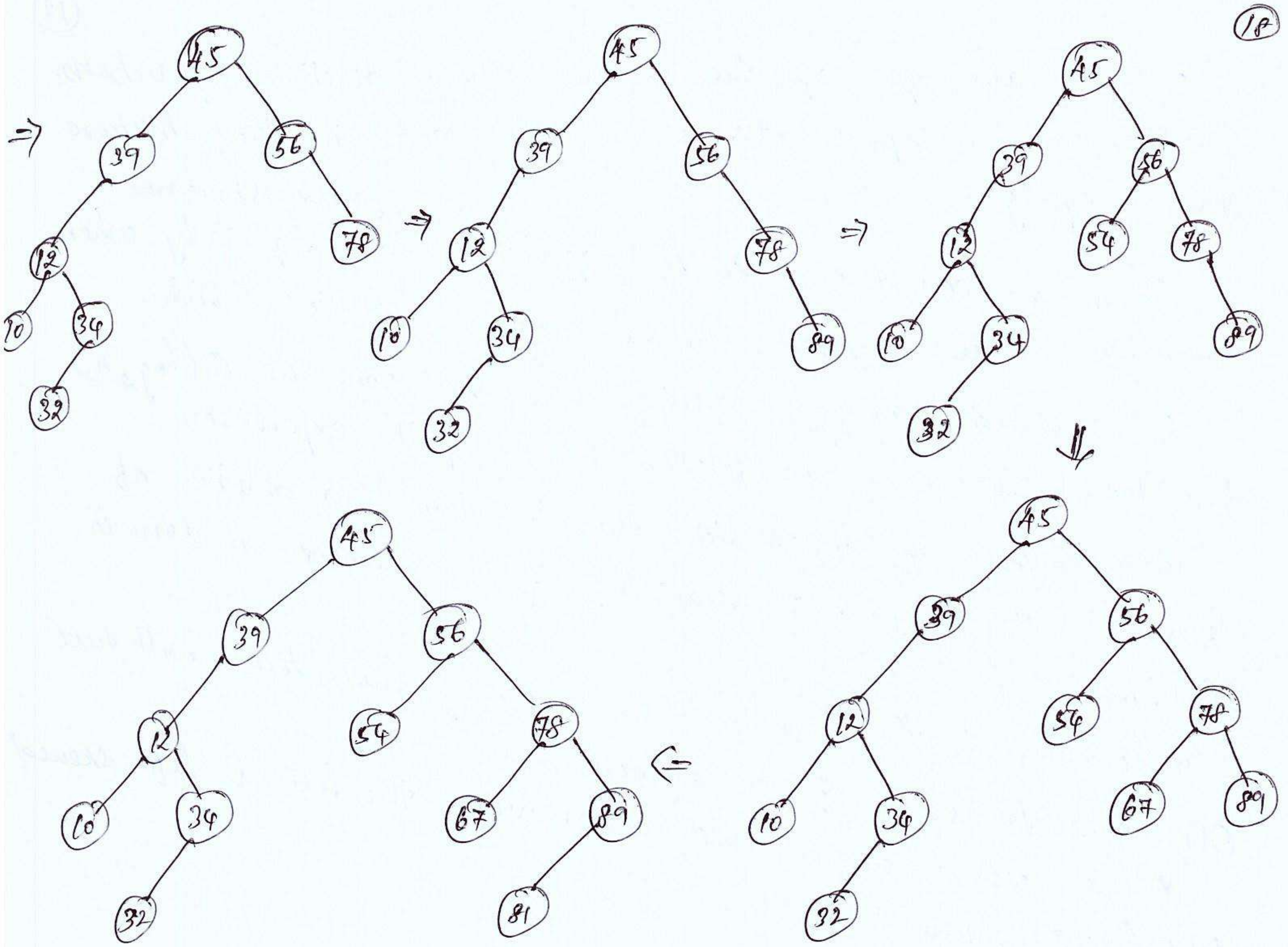
* However, in the worst case, a binary search tree will take $O(n)$ time to search for an element.

* The worst case would occur when the tree is left skewed BST / right skewed BST.

Creation of BST:-

Elements: 45, 39, 56, 12, 34, 78, 32, 10, 89, 54, 67, 81.





Operations on BST :-

Binary Search Tree Declarations

```
#ifndef _Tree_H
struct TreeNode;
typedef struct TreeNode *Position;
typedef struct TreeNode *SearchTree;
SearchTree MakeEmpty (SearchTree T);
Position Find (ElementType X, SearchTree T);
Position FindMin (SearchTree T);
Position FindMax (SearchTree T);
SearchTree Insert (ElementType X, SearchTree T);
SearchTree Delete (ElementType X, SearchTree T);
ElementType Retrieve (Position P);
#endif /* _Tree_H */
```

/* Place in the implementation file */

```
struct TreeNode
{
 ElementType Element;
 SearchTree Left;
 SearchTree Right;
};
```

1) MakeEmpty / Delete a BST;-

```
SearchTree MakeEmpty (SearchTree T)
{
   if (T != NULL)
    {
      MakeEmpty (T→ Left);
      MakeEmpty (T→ Right);
       free (T);
    }
   return NULL;
}
```

2) Find / Search an Element in a BST;-

```
Position Find (ElementType X, SearchTree T)
{
      if (T == NULL)
          return NULL;
      if (X < T→Element)
          return Find (X, T→ Left);
      else if (X > T→Element)
          return Find (X, T→ Right);
      else
          return T;
}
```

3) FindMin ;-

To perform a FindMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element in the BST.

```
Position FindMin (SearchTree T)
{
  if (T == NULL)
      return NULL;
  else if (T->Left == NULL)
      return T;
  else
      return FindMin (T->Left);
}
```

4) FindMax ;-

To perform a FindMax, start at the root node and go right as long as there is a right child. the stopping point is the largest element.

```
Position FindMax (SearchTree T)
{
  if (T != NULL)
      while (T->Right != NULL)
          T = T->Right;
  return T;
}
```

5) Insert :

* Memory is allocated for the newnode.
* The insertion of an element x into the tree
  - Check the root node of the tree is NULL.
  - If the condition is true, then the newnode as the root node.
  - Otherwise follow the next steps
  - Compare the newnode with the root node element. for the following 2 conditions:

(i) If the newnode element is less than the root node element, traverse the left subtree recursively, until it reaches left is NULL. Left is assigned to newnode.

(ii) If the newnode is greater than the root node element, traverse the right subtree recursively, until it reaches right is NULL. Right is assigned to newnode.

```
SearchTree Insert (ElementType X, SearchTree T)
{
  if (T==NULL)
  {
    /* Create and return a one-node tree */
    T= malloc (sizeof (struct TreeNode));
    if (T==NULL)
        FatalError ("Out of Space!!!");
    else
    {
      T→ Element = X;
      T→ Left = T→ Right = NULL;
    }
  }
else if (X < T→Element) /* Traverse left subtree */
      T→Left = Insert (X, T→Left);
else if (X > T→ Element) /* Traverse right subtree */
      T→ Right = Insert (X, T→ Right);
/* Else X is in the tree already, we will do nothing */
  return T;
}
```

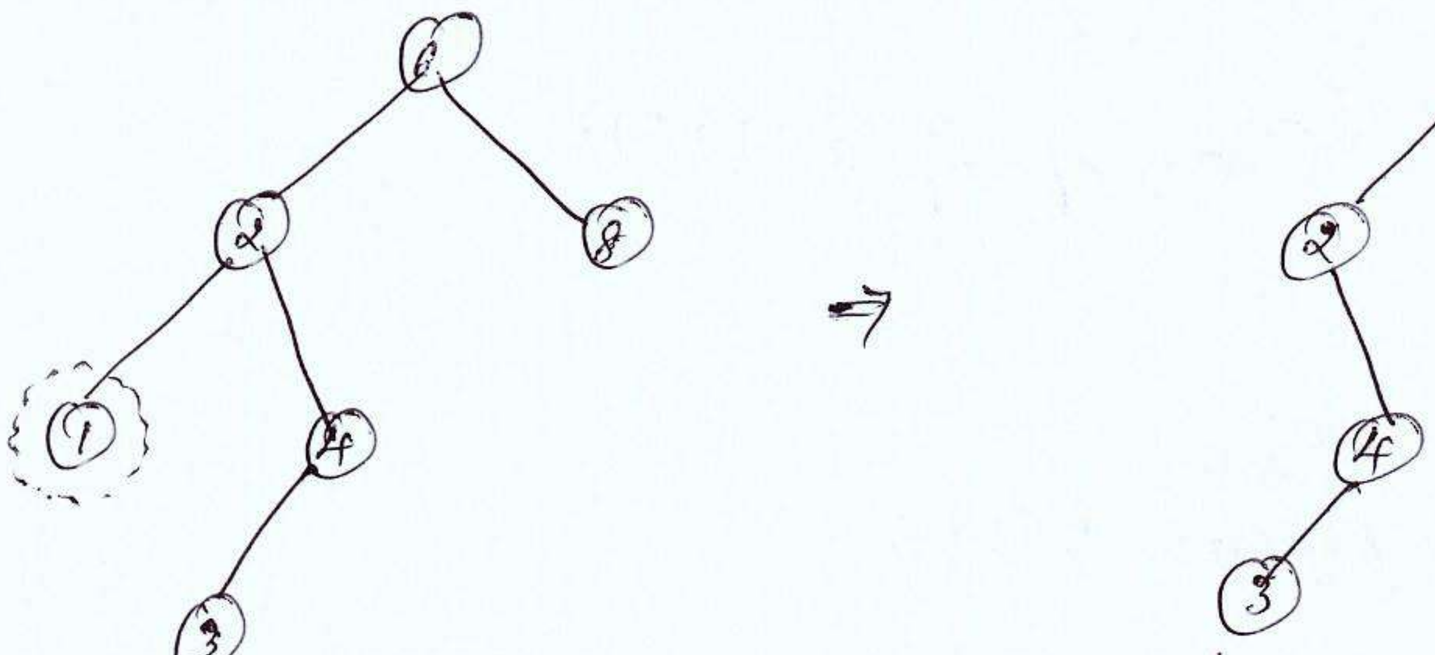
6) Delete:-

Memory is to released for the node to be deleted.
A node can be deleted with 3 possibilities:

(i) Deletion of leaf node
(ii) Deletion of node with one child
(iii) Deletion of node with two children.

(i) Deletion of leaf node:

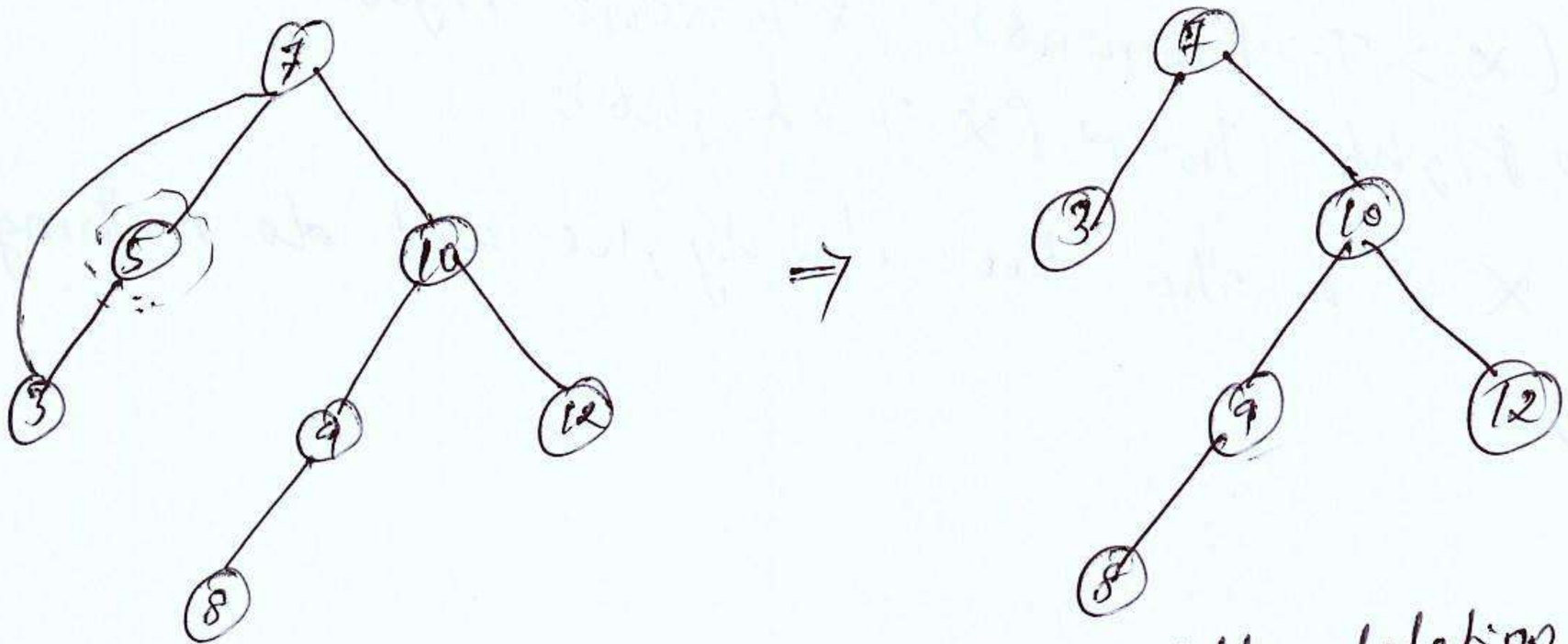
* Search the parent of the leaf node starting from the root node and make the link to the leaf node as NULL
* Release the memory for the deleted node.



Before deletion ⇒ After deletion.

Deletion of leaf node !

(ii) Deletion of a node with one child:

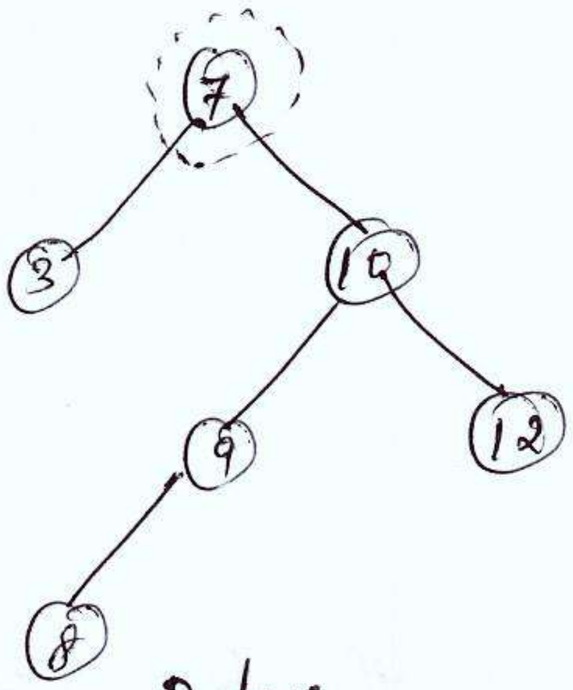


Before deletion ⇒ After deletion

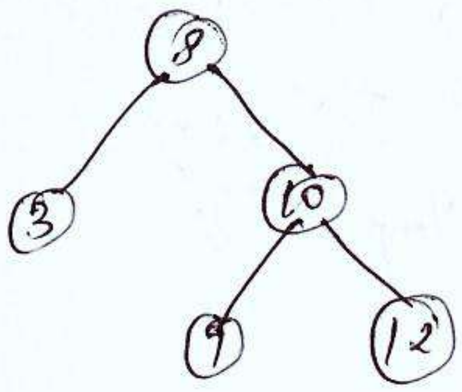
* Search the parent of the node to be deleted.
* Assign the link of the parent node to the child of the node to be deleted.
* Release the memory for the deleted node.

(ii) Deletion of a node with two children:

* Replace the element of the node to be deleted with its smallest element of the right subtree and recursively delete that node.
* Release the memory for the deleted node.



Before



After

```
SearchTree Delete( ElementType X, SearchTree T)
{
  Position TmpCell;
  if(T==NULL)
      Error("Element not found");
  else if (X < T->Element)          /* Go left subtree */
      T->Left = Delete(X, T->Left);
  else if (X > T->Element)          /* Go right subtree */
      T->Right = Delete(X, T->Right);
  else if (T->Left && T->Right)     /* Two children */
  {
  /* Replace with smallest in right subtree */
```

```
        TmpCell = FindMin (T→ Right);
        T→ Element = TmpCell → Element;
        T→ Right = Delete (T→ Element , T→ Right);
    }
else  /* one or zero children */
{
    TmpCell = T;
    if (T→ Left == NULL)
        T= T→ Right;
    else if (T→ Right == NULL)
        T=T→ Left;
    free (TmpCell);
}
return T;
}
```

7) Find Total No. of Nodes in a BST;-

To calculate the total no. of nodes in the tree, we count the no. of nodes in the left subtree and right subtree plus the root node.

Number of nodes = totalNodes(left subtree) + totalNodes(Right subtree) + 1.

* If the tree is empty, then the no. of nodes will be zero.

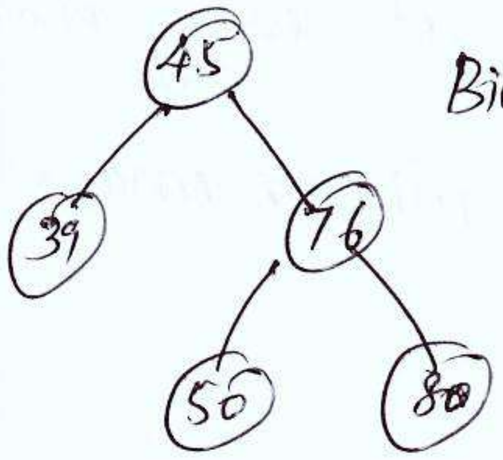
TREE = NULL

```
int totalNodes ( SearchTree T)
{
    if(T == NULL)
        return 0;
```

```
    else
        return (totalNodes(T->Left) + totalNodes(T->Right)+1);
}
```

8) Height of the tree:

In order to find the height of a BST, we calculate the height of the left subtree and the right subtree. Whichever height is greater, 1 is added to it.



Binary Search tree with height = 2

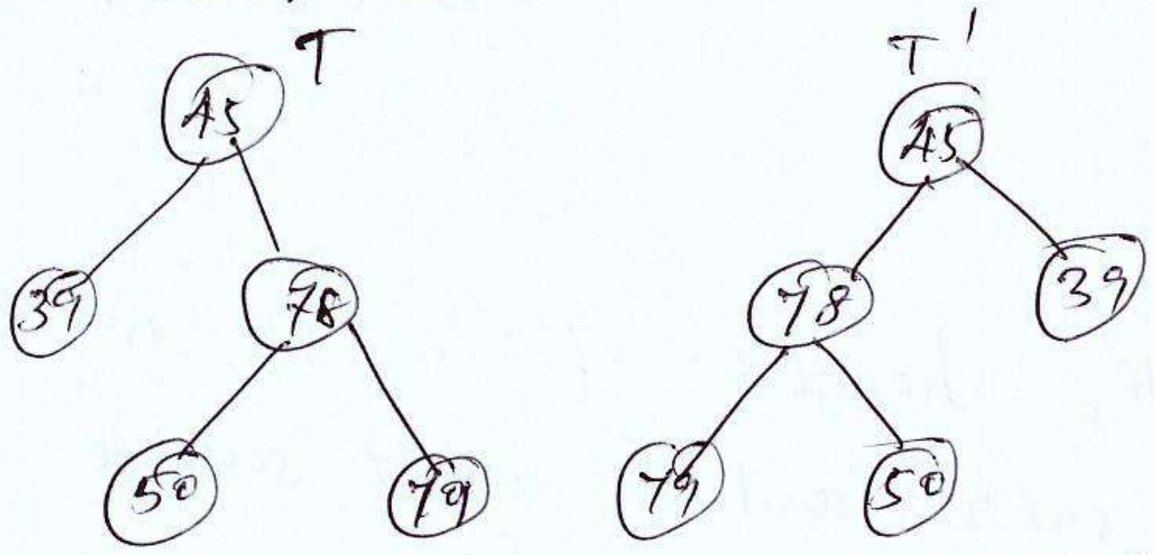
⇓

Height of Left subtree = 1
Height of Right subtree = 2.
} ⇒ Higher value = 2

Height of the tree = 2 + 1 = 3

```
int Height(SearchTree T)
{
    int leftheight, rightheight;
    if (T == NULL)
        return 0;
    else
    {
        leftheight = Height(T->Left);
        rightheight = Height(T->Right);
        if (leftheight > rightheight)
            return (leftheight+1);
        else
            return (rightheight+1);
    }
}
```

9) Mirror Image of a BST :-



BST Mirror image of BST

Mirror image of a BST is obtained by interchanging the left subtree with right subtree at every node of the tree.

& For example, given a Tree T, the mirror image of T can be obtained as T'.

```
struct TreeNode * mirrorImage ( struct TreeNode T)
{
  struct TreeNode *ptr;
  if(T != NULL)
  {
    mirrorImage (T->Left);
    mirrorImage (T->Right);
    ptr = T->Left;
    ptr->Left = ptr->right;
    T-> right = ptr;
  }
}
```

Pgm Code for various operations of BST

```
#include <stdio.h>
#include <conio.h>
#include <malloc.h>
struct node
{
        int data;
        struct node *left;
        struct node *right;
};
struct node *tree;
void create_tree(struct node *);
struct node *insertElement(struct node *, int);
void preorderTraversal(struct node *);
void inorderTraversal(struct node *);
```

```
void postorderTraversal(struct node *);
struct node *findSmallestElement(struct node *);
struct node *findLargestElement(struct node *);
struct node *deleteElement(struct node *, int);
struct node *mirrorImage(struct node *);
int totalNodes(struct node *);
int totalExternalNodes(struct node *);
int totalInternalNodes(struct node *);
int Height(struct node *);
struct node *deleteTree(struct node *);
int main()
{
        int option, val;
        struct node *ptr;
        create_tree(tree);
        clrscr();
        do
        {
                printf("\n ******MAIN MENU******* \n");
                printf("\n 1. Insert Element");
                printf("\n 2. Preorder Traversal");
                printf("\n 3. Inorder Traversal");
                printf("\n 4. Postorder Traversal");
                printf("\n 5. Find the smallest element");
                printf("\n 6. Find the largest element");
                printf("\n 7. Delete an element");
                printf("\n 8. Count the total number of nodes");
                printf("\n 9. Count the total number of external nodes");
                printf("\n 10. Count the total number of internal nodes");
                printf("\n 11. Determine the height of the tree");
                printf("\n 12. Find the mirror image of the tree");
                printf("\n 13. Delete the tree");
                printf("\n 14. Exit");
                printf("\n\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
```

```
        case 1:
             printf("\n Enter the value of the new node : ");
             scanf("%d", &val);
             tree = insertElement(tree, val);
             break;
        case 2:
             printf("\n The elements of the tree are : \n");
             preorderTraversal(tree);
             break;
        case 3:
             printf("\n The elements of the tree are : \n");
             inorderTraversal(tree);
             break;
        case 4:
             printf("\n The elements of the tree are : \n");
             postorderTraversal(tree);
             break;
        case 5:
             ptr = findSmallestElement(tree);
             printf("\n Smallest element is :%d",ptr->data);
             break;
        case 6:
             ptr = findLargestElement(tree);
             printf("\n Largest element is : %d", ptr->data);
             break;
        case 7:
             printf("\n Enter the element to be deleted : ");
             scanf("%d", &val);
```

```
                tree = deleteElement(tree, val);
                break;
            case 8:
                printf("\n Total no. of nodes = %d", totalNodes(tree));
                break;
            case 9:
                printf("\n Total no. of external nodes = %d",
                        totalExternalNodes(tree));
                break;
            case 10:
                printf("\n Total no. of internal nodes = %d",
                        totalInternalNodes(tree));
                break;
            case 11:
                printf("\n The height of the tree = %d",Height(tree));
                break;
            case 12:
                tree = mirrorImage(tree);
                break;
            case 13:
                tree = deleteTree(tree);
                break;
            }
        }while(option!=14);
        getch();
        return 0;
}
void create_tree(struct node *tree)
{
        tree = NULL;
}
struct node *insertElement(struct node *tree, int val)
{
        struct node *ptr, *nodeptr, *parentptr;
        ptr = (struct node*)malloc(sizeof(struct node));
        ptr->data = val;
        ptr->left = NULL;
        ptr->right = NULL;
        if(tree==NULL)
        {
                tree=ptr;
                tree->left=NULL;
                tree->right=NULL;
        }
        else
        {
                parentptr=NULL;
                nodeptr=tree;
                while(nodeptr!=NULL)
                {
                        parentptr=nodeptr;
                        if(val<nodeptr->data)
                                nodeptr=nodeptr->left;
                        else
                                nodeptr = nodeptr->right;
                }
                if(val<parentptr->data)
                        parentptr->left = ptr;
                else
                        parentptr->right = ptr;
        }
        return tree;
}
void preorderTraversal(struct node *tree)
{
```

```
        if(tree != NULL)
        {
                printf("%d\t", tree->data);
                preorderTraversal(tree->left);
                preorderTraversal(tree->right);
        }
}
void inorderTraversal(struct node *tree)
{
        if(tree != NULL)
        {
                inorderTraversal(tree->left);
                printf("%d\t", tree->data);
                inorderTraversal(tree->right);
        }
}
void postorderTraversal(struct node *tree)
{
        if(tree != NULL)
        {
                postorderTraversal(tree->left);
                postorderTraversal(tree->right);
                printf("%d\t", tree->data);
        }
}
struct node *findSmallestElement(struct node *tree)
{
        if( (tree == NULL) || (tree->left == NULL))
                return tree;
        else
                return findSmallestElement(tree->left);
}
struct node *findLargestElement(struct node *tree)
{
        if( (tree == NULL) || (tree->right == NULL))
                return tree;
        else
                return findLargestElement(tree->right);
}
struct node *deleteElement(struct node *tree, int val)
{
        struct node *cur, *parent, *suc, *psuc, *ptr;
        if(tree->left==NULL)
        {
                printf("\n The tree is empty ");
                return(tree);
        }
        parent = tree;
        cur = tree->left;
        while(cur!=NULL && val!= cur->data)
        {
                parent = cur;
                cur = (val<cur->data)? cur->left:cur->right;
        }
        if(cur == NULL)
        {
                printf("\n The value to be deleted is not present in the tree");
                return(tree);
        }
        if(cur->left == NULL)
                ptr = cur->right;
        else if(cur->right == NULL)
```

```
        ptr = cur->left;
else
{
        // Find the in-order successor and its parent
        psuc = cur;
        cur = cur->left;
        while(suc->left!=NULL)
        {
                psuc = suc;
                suc = suc->left;
        }
        if(cur==psuc)
        {
                // Situation 1
                suc->left = cur->right;
        }
        else
        {
                // Situation 2
                suc->left = cur->left;
                psuc->left = suc->right;
                suc->right = cur->right;
        }
        ptr = suc;
}
// Attach ptr to the parent node
if(parent->left == cur)
        parent->left=ptr;
else
        parent->right=ptr;
free(cur);
return tree;
}
int totalNodes(struct node *tree)
{
        if(tree==NULL)
                return 0;
        else
                return(totalNodes(tree->left) + totalNodes(tree->right) + 1);
}
int totalExternalNodes(struct node *tree)
{
        if(tree==NULL)
                return 0;
        else if((tree->left==NULL) && (tree->right==NULL))
                return 1;
        else
                return (totalExternalNodes(tree->left) +
                totalExternalNodes(tree->right));
}
int totalInternalNodes(struct node *tree)
{
        if( (tree==NULL) || ((tree->left==NULL) && (tree->right==NULL)))
                return 0;
        else
                return (totalInternalNodes(tree->left)
                + totalInternalNodes(tree->right) + 1);
}
int Height(struct node *tree)
{
        int leftheight, rightheight;
```

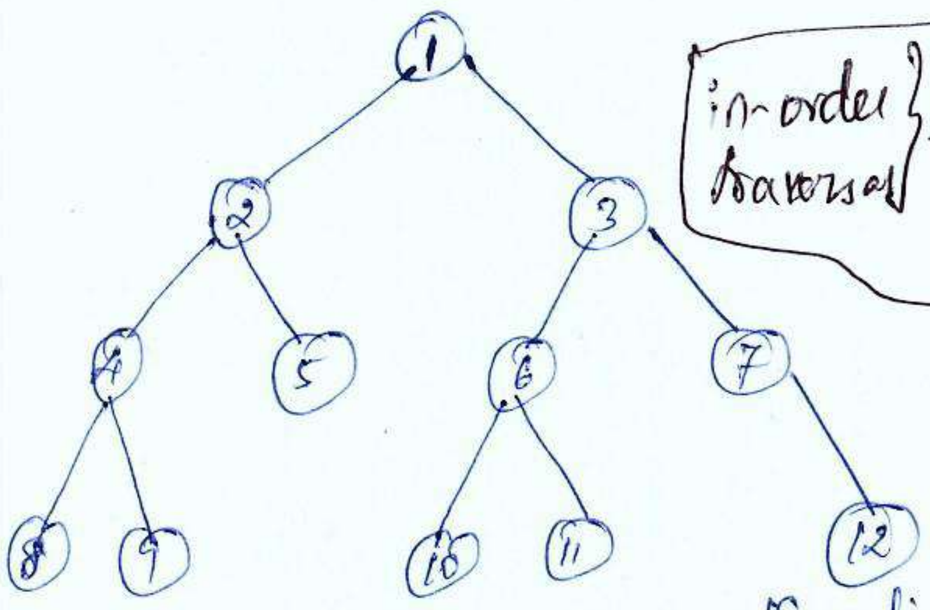
```
if(tree==NULL)
        return 0;
else
{
        leftheight = Height(tree->left);
        rightheight = Height(tree->right);
        if(leftheight > rightheight)
                return (leftheight + 1);
        else
                return (rightheight + 1);
        }
}
struct node *mirrorImage(struct node *tree)
{
        struct node *ptr;
        if(tree!=NULL)
        {
                mirrorImage(tree->left);
                mirrorImage(tree->right);
                ptr=tree->left;
                ptr->left = ptr->right;
                tree->right = ptr;
        }
}
struct node *deleteTree(struct node *tree)
{
        if(tree!=NULL)
        {
                deleteTree(tree->left);
                deleteTree(tree->right);
                free(tree);
        }
}
```

Output

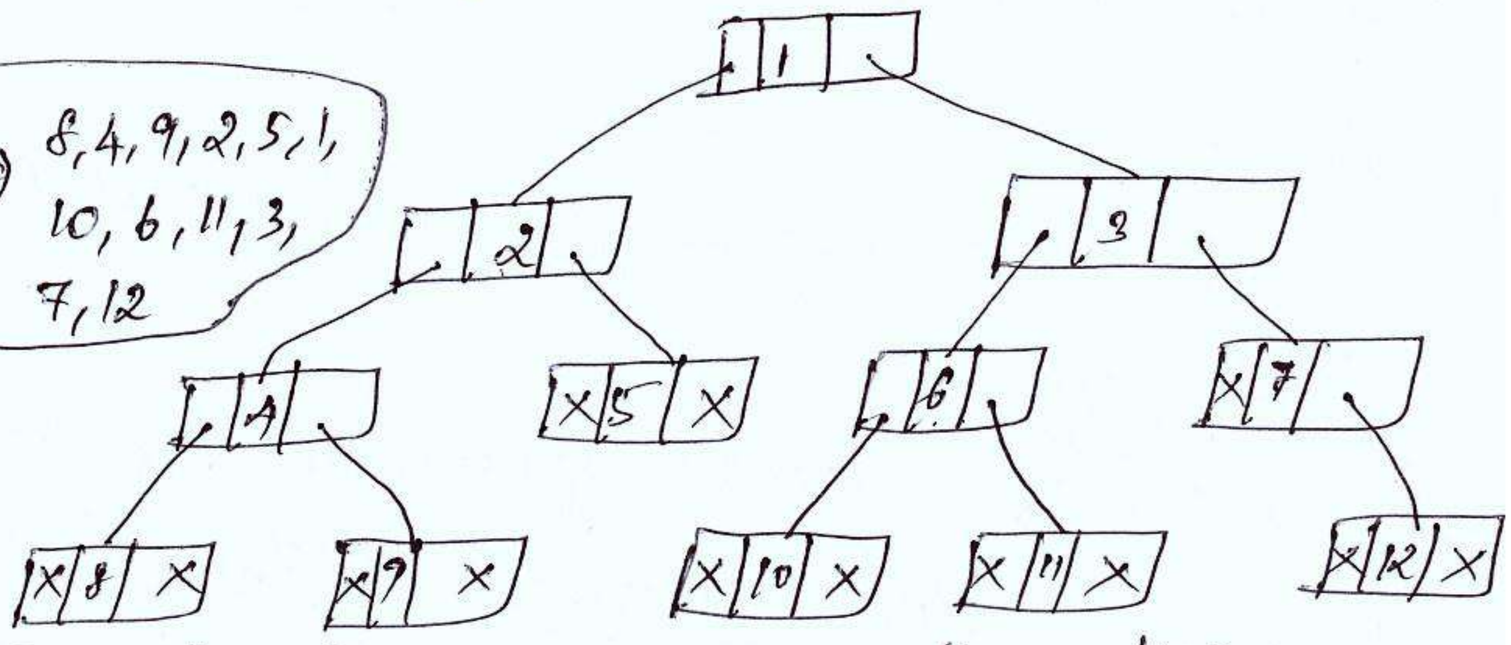
```
*******MAIN   MENU*******
1. Insert Element
2. Preorder Traversal
3. Inorder Traversal
4. Postorder Traversal
5. Find the smallest element
6. Find the largest element
7. Delete an element
8. Count the total number of nodes
9. Count the total number of external nodes
10. Count the total number of internal nodes
11. Determine the height of the tree
12. Find the mirror image of the tree
13. Delete the tree
14. Exit
Enter your option : 1
Enter the value of the new node : 1
Enter the value of the new node : 2
Enter the value of the new node : 4
Enter your option : 3
2    1    4
Enter your option : 14
```

Threaded Binary Trees :- (TBT).

A threaded binary tree is the same as of binary tree but with a difference in storing the NULL pointers. Consider the linked representation of a binary tree as in Fig.



Binary tree without threading



Linked representation of binary tree (without threading)

In the linked representation, a no. of nodes contain a NULL pointer, either in their left or right fields or in both.

* This space is wasted in storing a NULL pointer can be efficiently used to store some other useful piece of information.

* For example, the NULL entries can be replaced to store a pointer to the in-order predecessor or the in-order successor of the node.

* These special pointers are called <u>threads</u> and binary trees containing threads are called <u>threaded trees</u>.

* In the linked representation of a TBT, threads will be denoted using arrows.

* A threaded binary tree may correspond to <u>one-way</u> threading or a <u>two-way</u> threading.

In one-way threading, a thread will appear either in the right field or the left field of the node.

* A one-way threaded tree is also called a single-threaded tree.

* If the thread appears in the left field, then the left field will be made to point to the in-order predecessor of the node.

* Such a one-way threaded tree is called a left-threaded binary tree.

* If the thread appears in the right field, then it will point to the in-order successor of the node.

* Such a one-way threaded tree is called a right-threaded binary tree.

In a two-way threaded tree, also called a double-threaded tree, threads will appear in both the left and the right field of the node.

* While the left field will point to the in-order predecessor of the node, the right field will point to its successor.

* A two-way threaded binary tree is called a fully threaded binary tree.

One-way Threading:

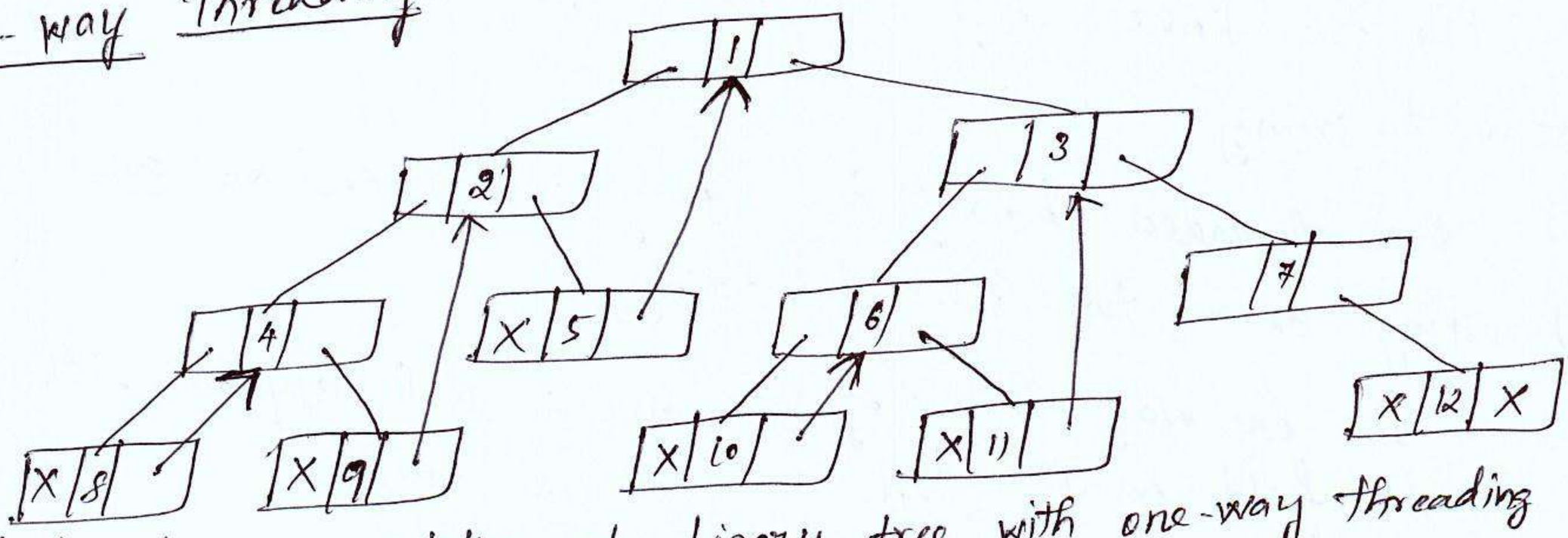


Fig.(a) Linked representation of binary tree with one-way threading

In-order traversal => 8, 4, 9, 2, 5, 1, 10, 6, 11, 3, 7, 12

Fig. shows a binary tree with one-way threading and its corresponding linked representation.

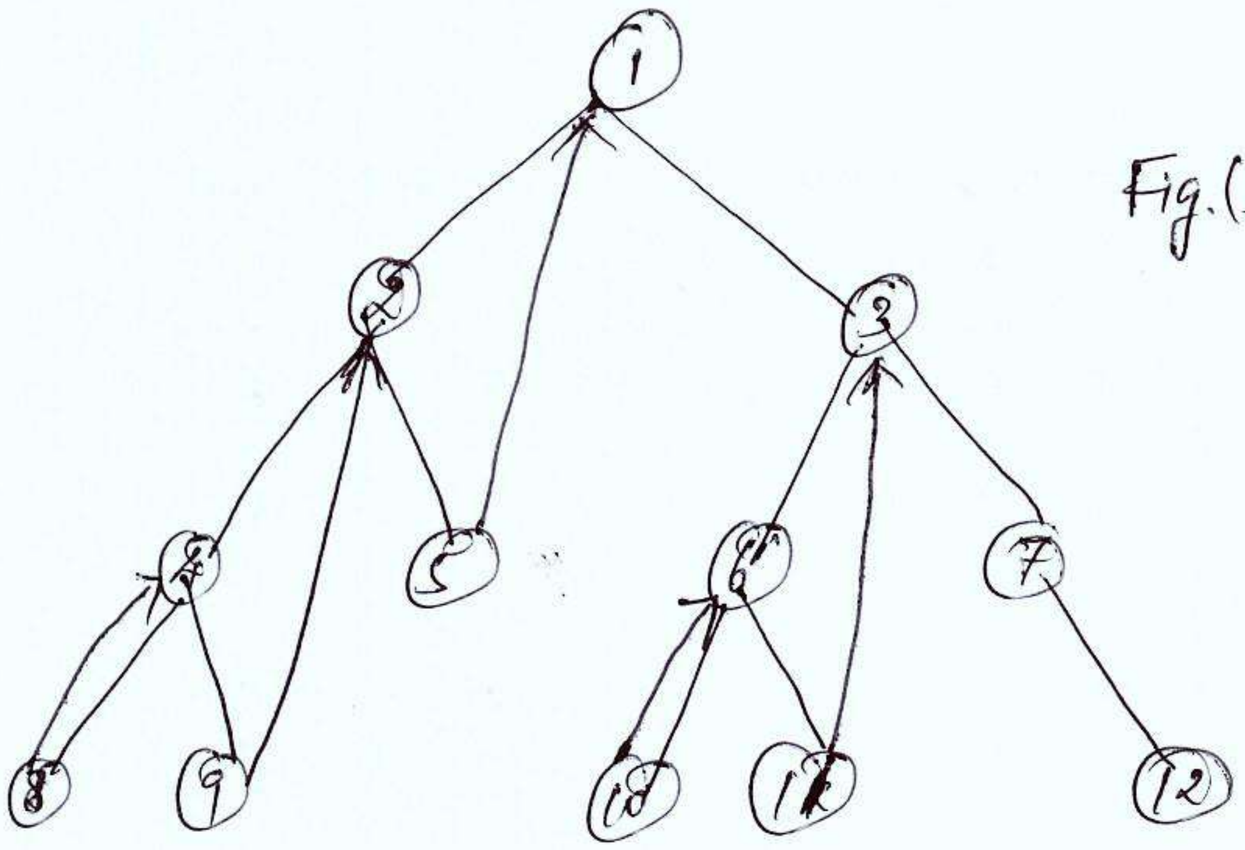
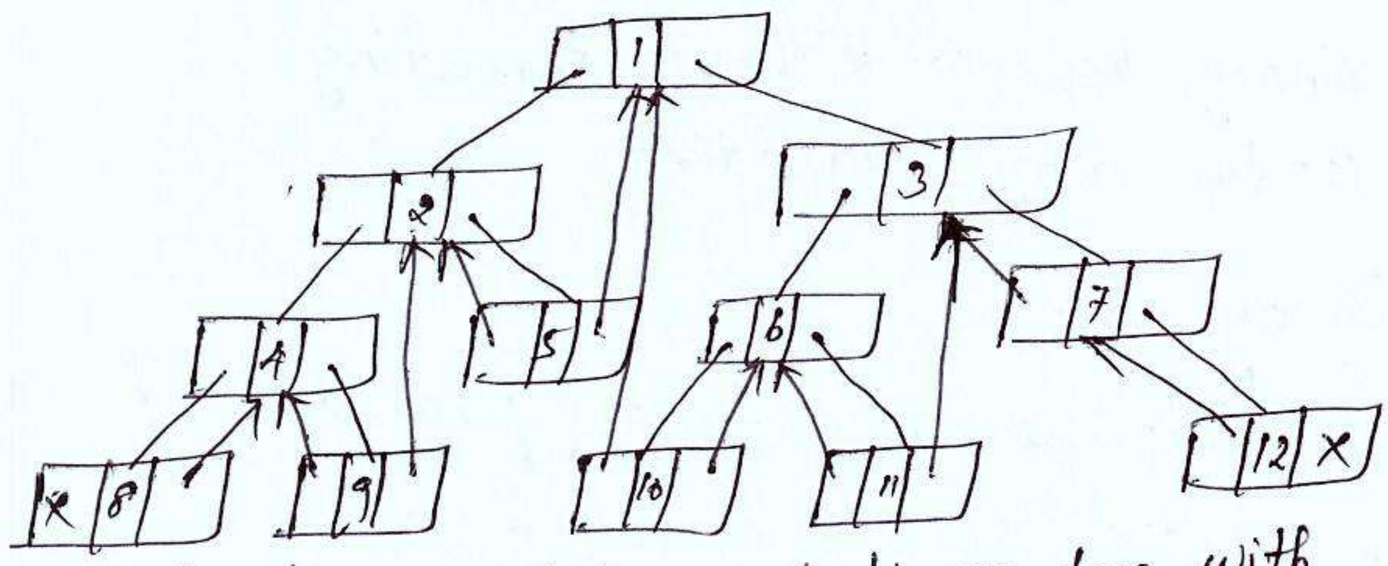


Fig.(b) Binary tree with one-way threading.

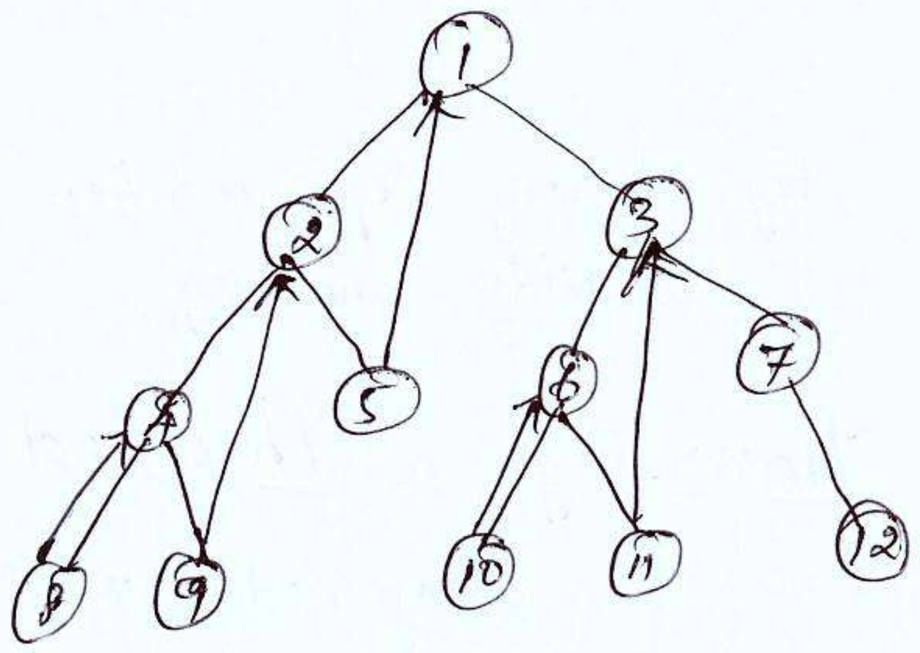
Node 5 contains a NULL pointer in its RIGHT field, so it will be replaced to point to node 1, which is a successor in In-order traversal.

Two-way Threading:

Fig. shows a binary tree with two-way threading and its Corresponding linked representation.



Fig(a) Linked representation of binary tree with threading



(b) binary tree with two-way threading

Node 5 contains a NULL pointer in its LEFT field, so it will be replaced to point to node 2, which is its in-order predecessor.

* Similarly, the LEFT field of node 8 will contain NULL because it has no in-order predecessor.

Now, let us look at the memory representation of a binary tree without threading, with one-way threading and with two-way threading. This is illustrated in Fig below.

(a)

ROOT = 3

	LEFT	DATA	RIGHT
1	-1	8	-1
2	-1	10	-1
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	-1	7	18
12	-1	9	-1
13			
14	-1	5	-1
15			
16	-1	11	-1
17			
18	-1	12	-1
19			
20	2	6	16

AVAIL = 15

(b)

ROOT = 3

	LEFT	DATA	RIGHT
1	-1	8	9
2	-1	10	20
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12.
10			
11	-1	7	18
12	-1	9	5
13			
14	-1	5	3
15			
16	-1	11	8
17			
18	-1	12	-1
19			
20	2	6	16

AVAIL = 15

(c)

ROOT = 3

	LEFT	DATA	RIGHT
1	-1	8	9
2	3	10	20
3	5	1	8
4			
5	9	2	14
6			
7			
8	20	3	11
9	1	4	12
10			
11	8	7	18
12	9	9	5
13			
14	5	5	3
15			
16	20	11	8
17			
18	11	12	-1
19			
20	2	6	16

AVAIL = 15

Fig. Memory representation of binary trees (a) without threading (b) with one-way, and (c) two-way threading

Traversing a Threaded Binary Tree;-

For every node, visit the left subtree first, provided if one exists and has not been visited earlier.

* Then the node (root) itself is followed by visiting its right subtree (if one exists).

* In case there is no right subtree, check for the threaded link and make the threaded node the current node in consideration.

Algorithm:

Step 1: Check if the current node has a left child that has not been visited. If a left child exists that has not been visited, go to Step 2, else go to Step 3.

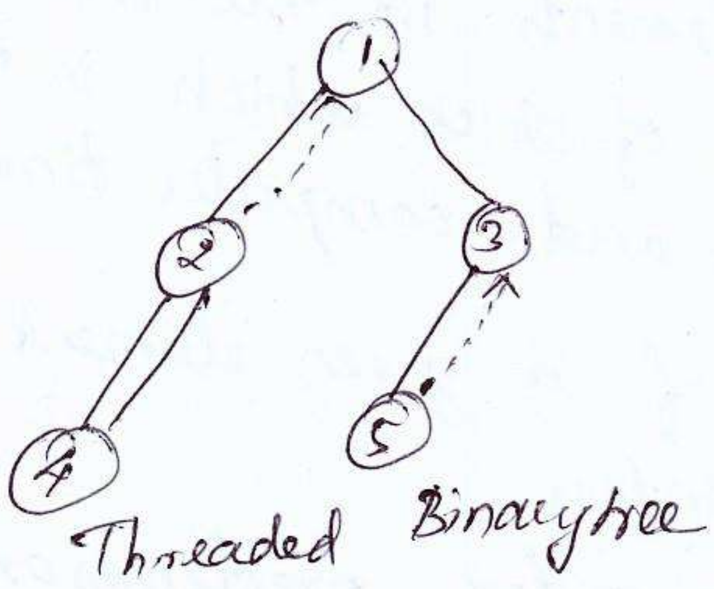
Step 2: Add the left child in the list of visited nodes. Make it as the current node and then go to step 6.

Step 3: If the current node has a right child, go to step 4. else go to step 5.

Step 4: Make that right child as current node and go to Step 6.

Step 5: Print the node and if there is a threaded node make it the current node.

Step 6: If all the nodes have visited then END else go to Step 1.



Threaded Binarytree

Let us consider the threaded binary tree as in Fig.

1). Node 1 has a left child i.e. 2 which has not been visited. So, add 2 ~~in~~ in the list of visited nodes, make it as the current node.

In-order Traversal: 4, 2, 1, 5, 3

2). Node 2 has a left child i.e. 4 which has not been visited, so add 4 in the list of visited nodes, make it as the current node.

3) Node 4 does not have any left or right child, so print 4 and check its threaded link. It has a threaded link to node 2, so make node 2, the current node.

4) Node 2 has a left child which has already been visited. However, it does not have a right child. Now, print 2 and follow its threaded link to node 1. Make node 1 the current node.

5) Node 1 has a left child which has already been visited. ~~However, it does not have a right~~ So print 1. Node 1 has a right child 3 which has not yet been visited, so make it the current node.

6) Node 3 has a left child (node 5) which has not been visited. So make it the current node.

7). Node 5 does not have any left or right child. So print 5. However, it does have a threaded link which points to node 3. Make node 3 the current node.

8). Node 3 has a left child which has already been visited. So print 3.

9). Now there are no nodes left, so we end here. The in-order sequence of nodes printed is - 4 2 1 5 3.

Advantages:

* It enables linear traversal of elements in the tree.
* Linear traversal eliminates the use of stack which in turn consume a lot of memory space and computer time.
* It enables to find the parent of a given element without explicit use of parent pointers.
* Since nodes contain pointers to in-order predecessor and successor, the threaded tree enables forward and backward traversal of the nodes as given by in-order fashion.

Disadvantages:

* Insertion and deletion becomes more difficult.
* Tree traversal algs become difficult.
* Memory required to store a node increases. Each node has to store the information whether the links are normal links or thread links.

Write a program to implement simple right in-threaded binary trees.

```
#include <stdio.h>
#include <conio.h>
struct tree
{
        int val;
        struct tree *right;
        struct tree *left;
        int thread;
};
struct tree *root = NULL;
struct tree* insert_node(struct tree *root, struct tree *ptr, struct tree *rt)
{
        if(root == NULL)
        {
                root = ptr;
                if(rt != NULL)
                {
                        root->right = rt;
                        root->thread = 1;
                }
        }
        else if(ptr->val < root->val)
                root->left = insert_node(root->left, ptr, root);
        else
                if(root->thread == 1)
                {
                        root->right = insert_node(NULL, ptr, rt);
                        root->thread=0;
                }
                else
                        root->right = insert_node(root->right, ptr, rt);
        return root;
}
struct tree* create_threaded_tree()
{
        struct tree *ptr;
        int num;
        printf("\n Enter the elements, press -1 to terminate ");
        scanf("%d", &num);
        while(num != -1)
        {
        ptr = (struct tree*)malloc(sizeof(struct tree));
                ptr->val = num;
                ptr->left = ptr->right = NULL;
                ptr->thread = 0;
                root = insert_node(root, ptr, NULL);
                printf(" \n Enter the next element ");
                fflush(stdin);
                scanf("%d", &num);
        }
        return root;
}
void inorder(struct tree *root)
{
        struct tree *ptr = root, *prev;
        do
        {
```

```
while(ptr != NULL)
{
        prev = ptr;
        ptr = ptr->left;
}
if(prev != NULL)
{
        printf(" %d", prev->val);
        ptr = prev->right;
        while(prev != NULL && prev->thread)
        {
                printf(" %d", ptr->val);
                prev = ptr;
                ptr = ptr->right;
        }
}
}while(ptr != NULL);
}
void main()
{
    //   struct tree *root=NULL;
        clrscr();
        create_threaded_tree();
        printf(" \n The in-order traversal of the tree can be given as : ");
        inorder(root);
        getch();
}
```

Output

```
Enter the elements, press -1 to terminate 5
Enter the next element 8
Enter the next element 2
Enter the next element 3
Enter the next element 7
Enter the next element -1
The in-order traversal of the tree can be given as:
2       3       5       7       8
```

AVL Trees :-

An AVL (Adelson - Velskii and Landis) tree is a binary search tree with a balanced condition. An AVL tree is identical to a binary search tree, except that for every node in the tree, the height of the left and right subtrees can differ by atmost 1.

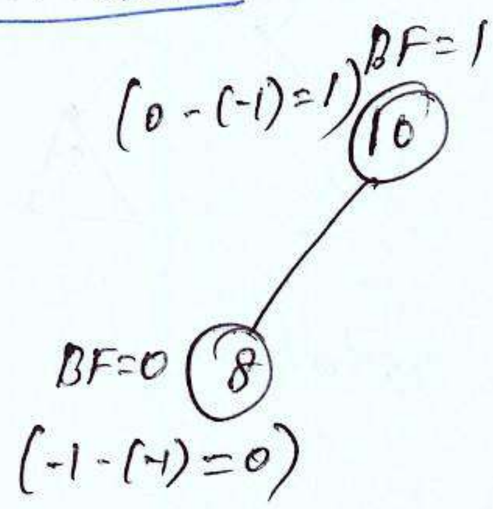
The height of an empty tree is defined to be -1. The balance condition ensures that the depth of the tree is O(log N).

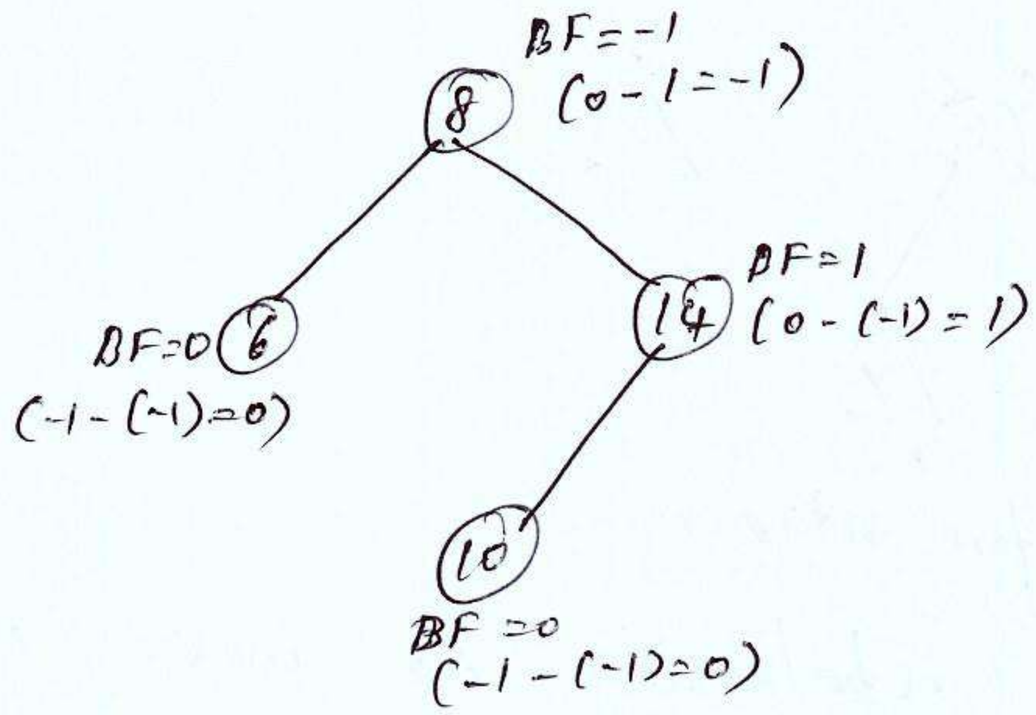
The balance factor of a node is calculated by subtracting the height of its right subtree from the height of its left subtree.

* A BST in which every node has a balance factor of -1, 0, 1 is said to be height balanced.

Balance factor = Height (left subtree) - Height (right subtree).

If the balanced factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced by making a simple modifications in the tree called rotation.





An AVL tree causes imbalance, when any one of the following conditions occurs:

1) An insertion into the left subtree of the left child
2) " " " right " " " " "
3) " " " left " " " right "
4) " " " right " " " " "

These rotations can be solved by

(i) Single rotation
(ii) Double rotation.

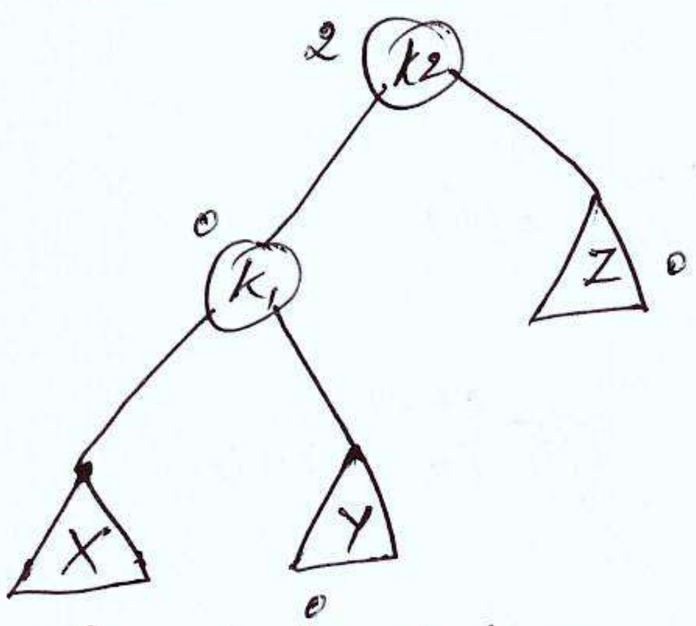
Single Rotation;-

Insertion occurs on the outside is fixed by a single rotation of the tree. It is performed to fix case 1 & 4.

Case 1: (LL)

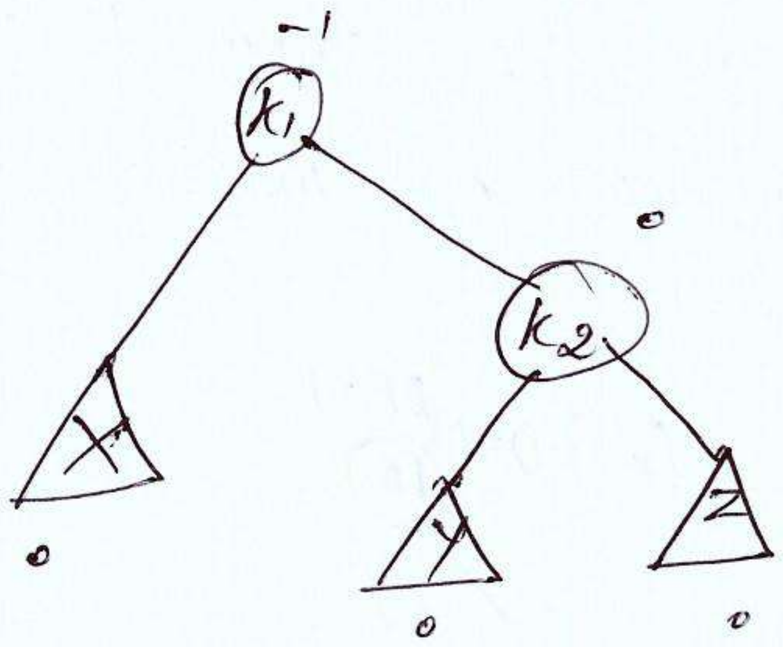
An insertion into the left subtree of the left child of k_2. (LL)

General Representation:



Before rotation

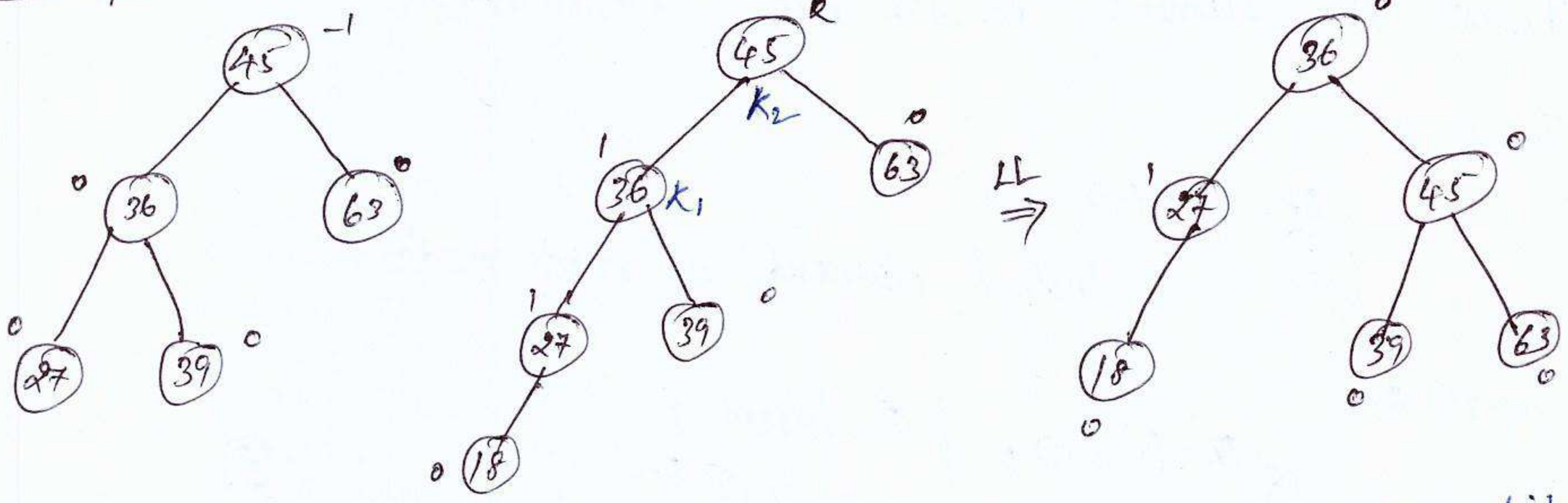
⇒



After rotation

For rebalancing k_2 moves one level down and k_1 one level up. So k_1 becomes new root and k_2 in the right child of k_1.

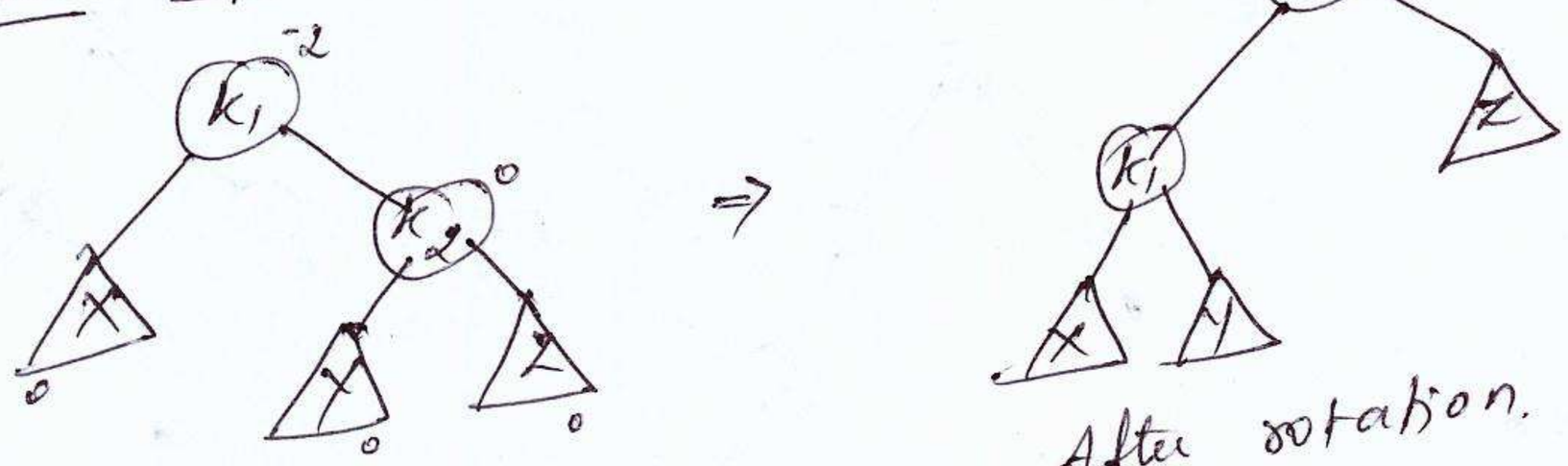
Example1:



Case 4: (RR)

An insertion into the right subtree of the right child of K_1 (RR).

General Representation:

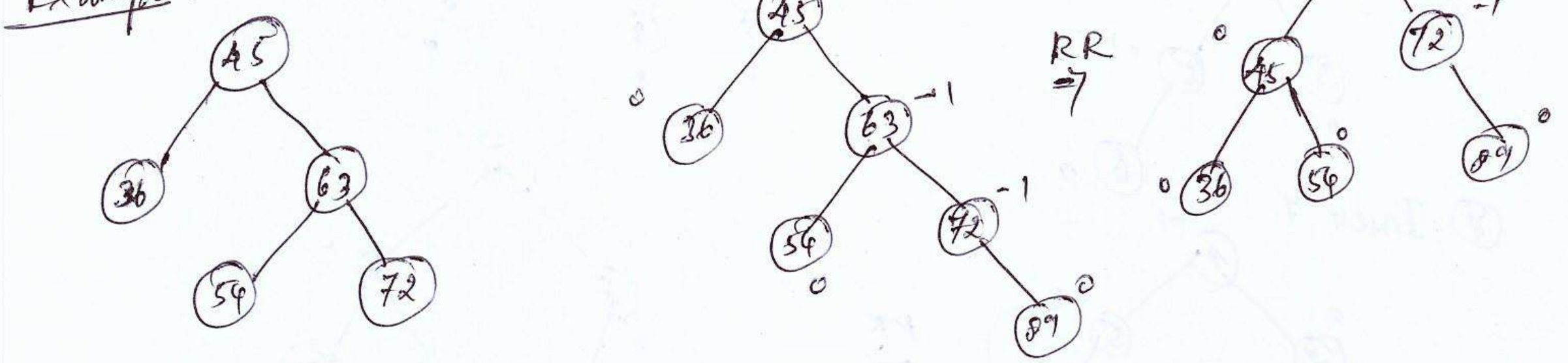


Before rotation

After rotation.

For rebalancing K2 moves one level up and K1 moves one level down. So K2 becomes new root and K1 is the left child of K2.

Example2:



Example: 3

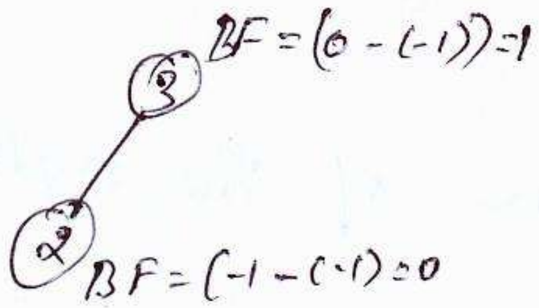
Insert the elements to AVL tree: 3, 2, 1, 4, 5, 6, 7

① Insert 3

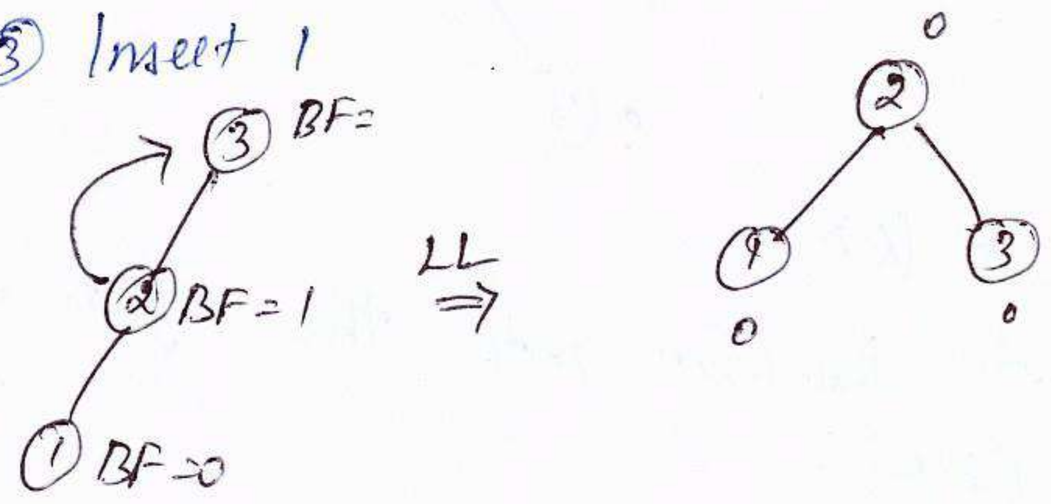
$BF = (-1 - (-1)) = 0$

(3) Initial element as root node

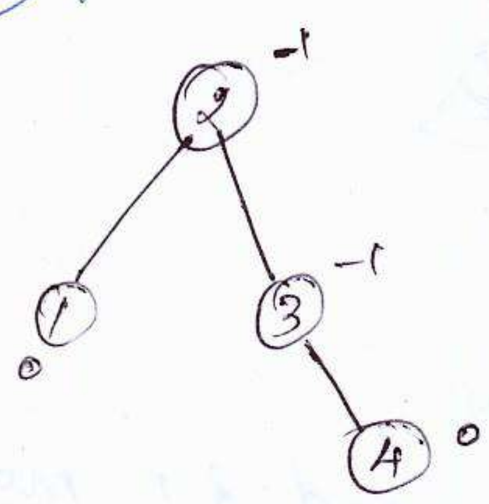
② Insert 2



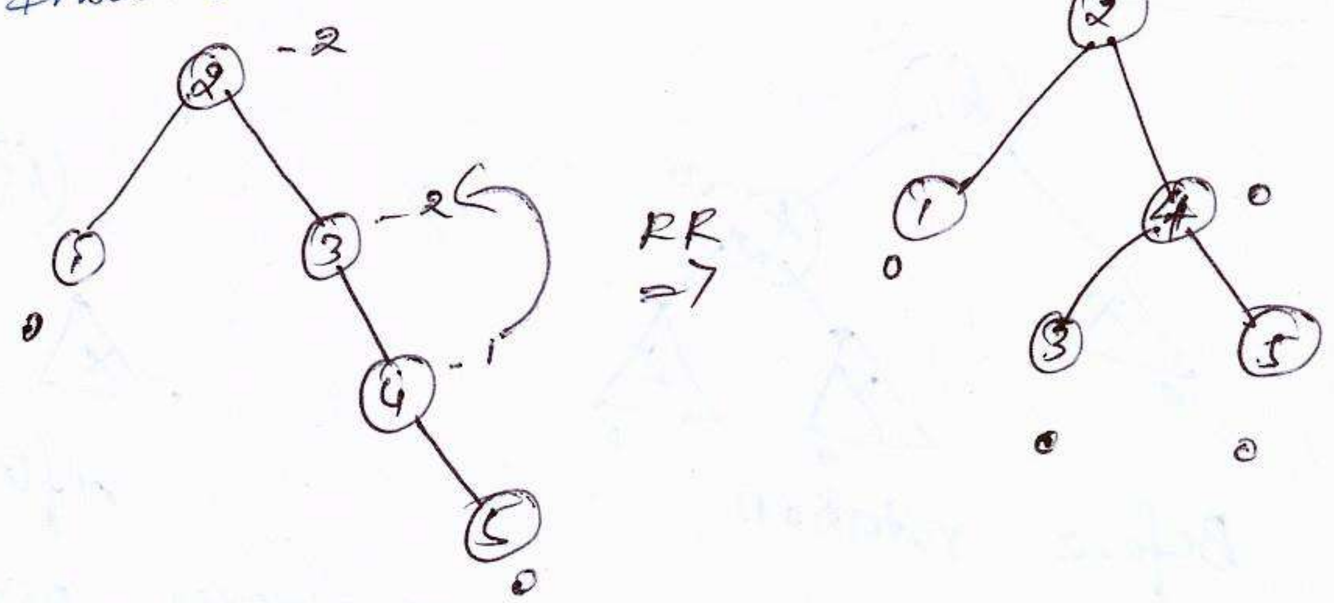
③ Insert 1



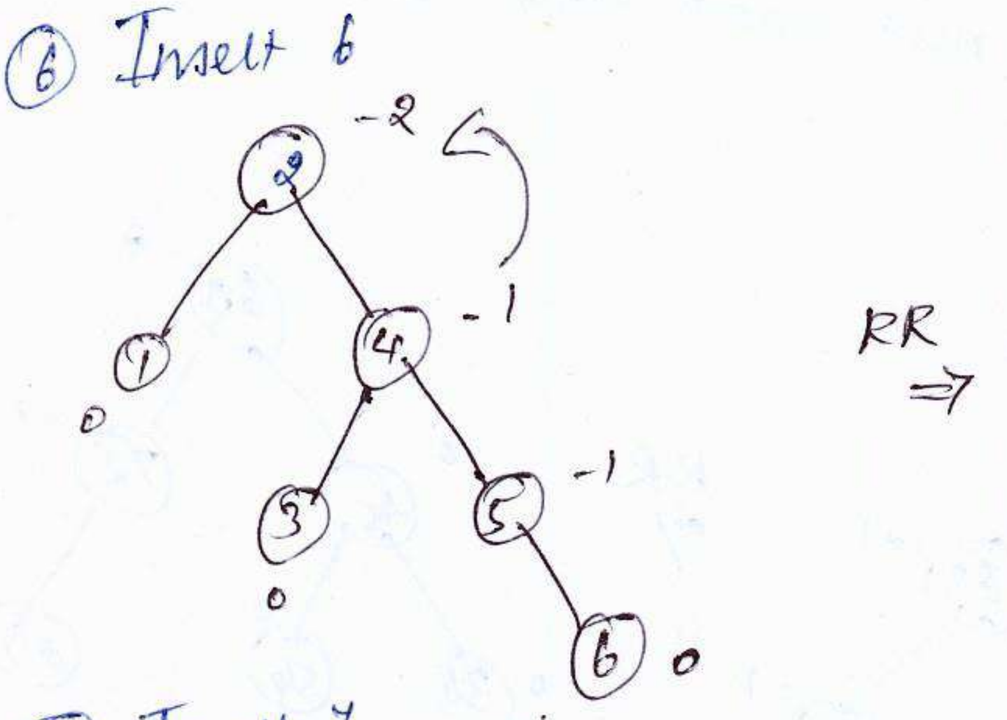
④ Insert 4



⑤ Insert 5



⑥ Insert 6



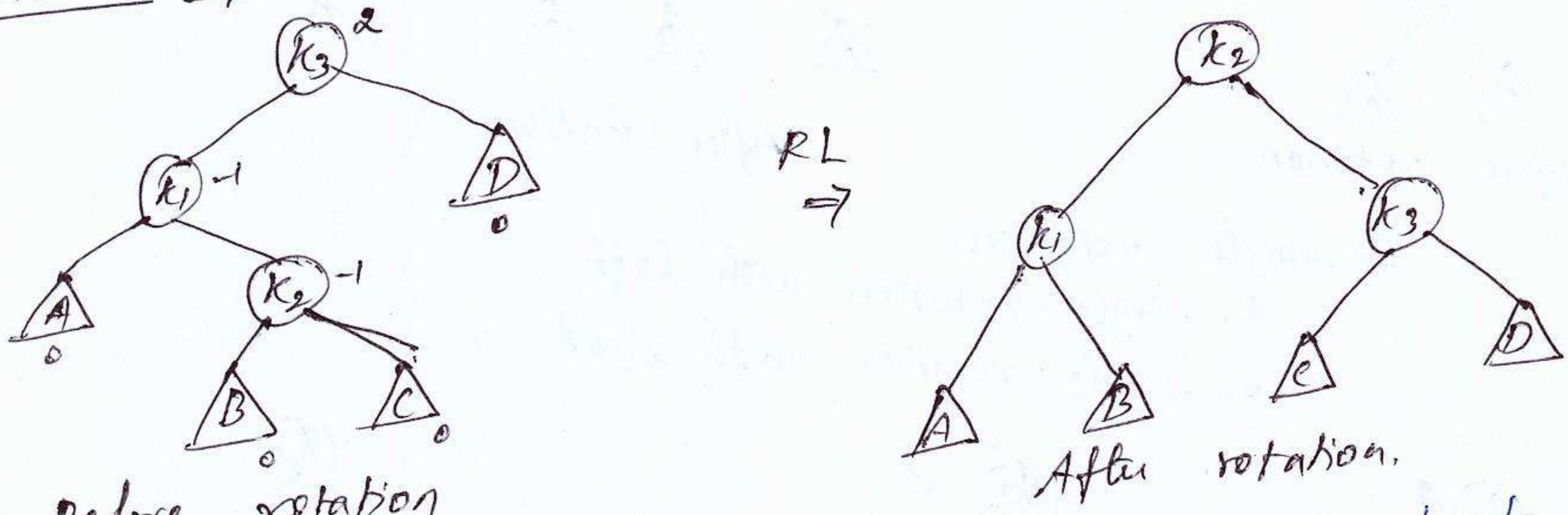
⑦ Insert 7

Double Rotation:

Insertion occurs on the inside of the tree is handled by the double rotation. It fixes the case 2 and 3.

Case 2: RL
An insertion into right subtree of the left child (RL).

General Representation:



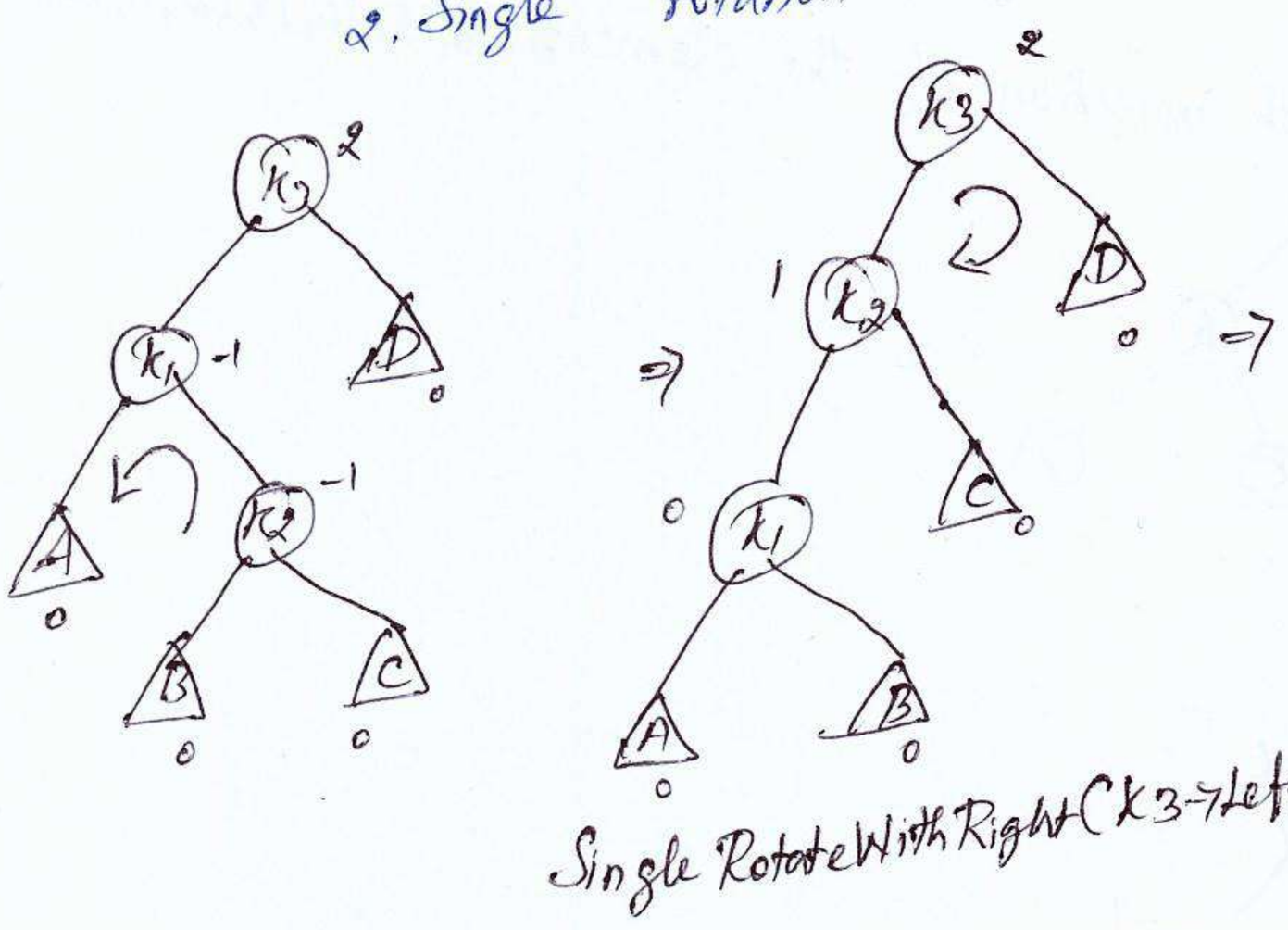
Before rotation

After rotation.

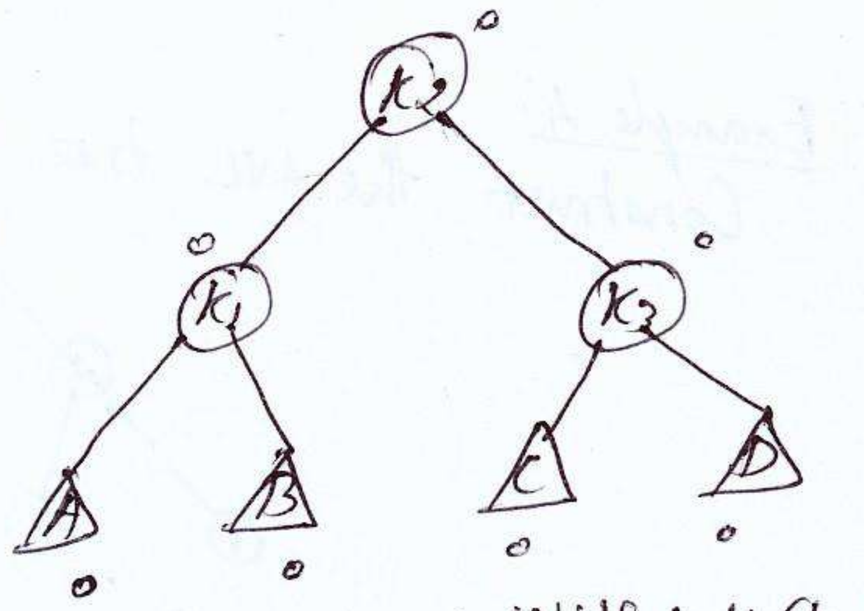
Place k2 as the new root node and make k1 to be k2's left child and k3 to its right child.

The above double rotation can be done by using the following two single rotations:

1. Single rotation with right
2. Single rotation with left



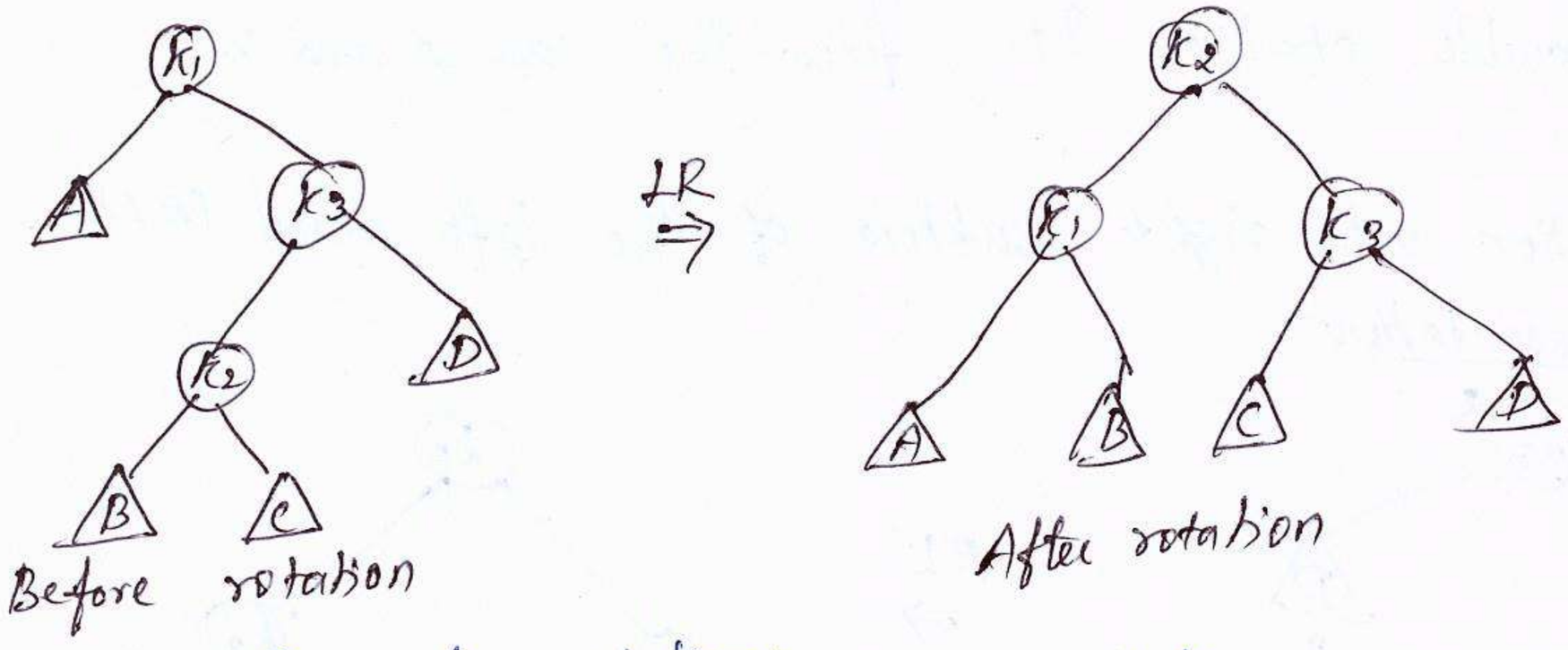
Single Rotate With Right (k3 → left)



Single Rotate With Left (k3)

Case 3: An insertion into the left subtree of the right child (LR)

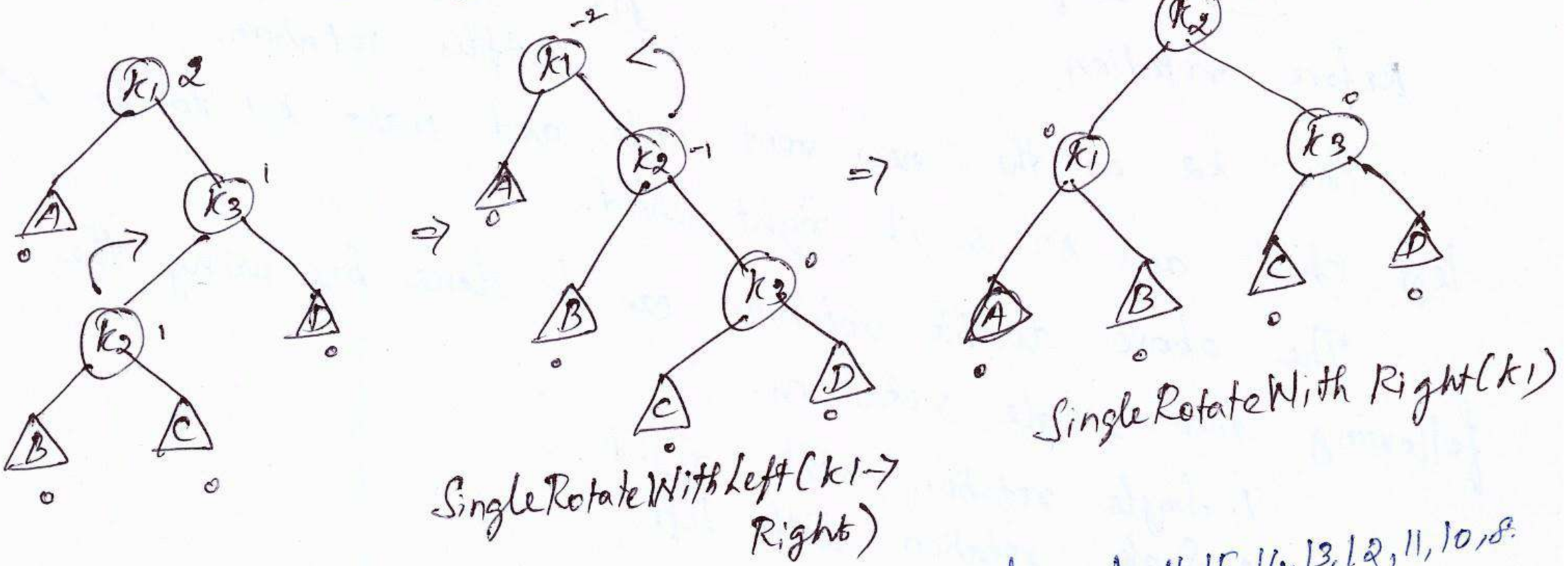
General Representation:



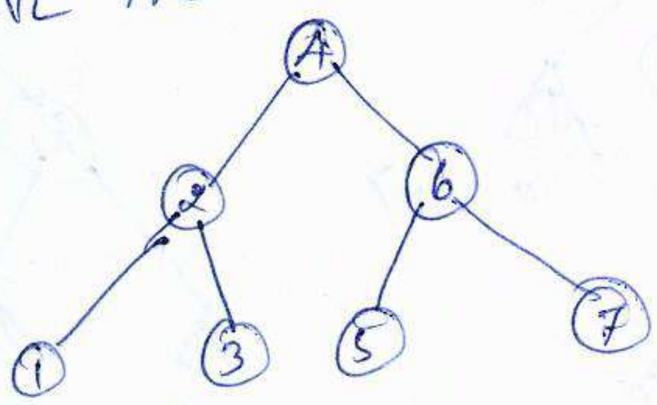
Before rotation

After rotation

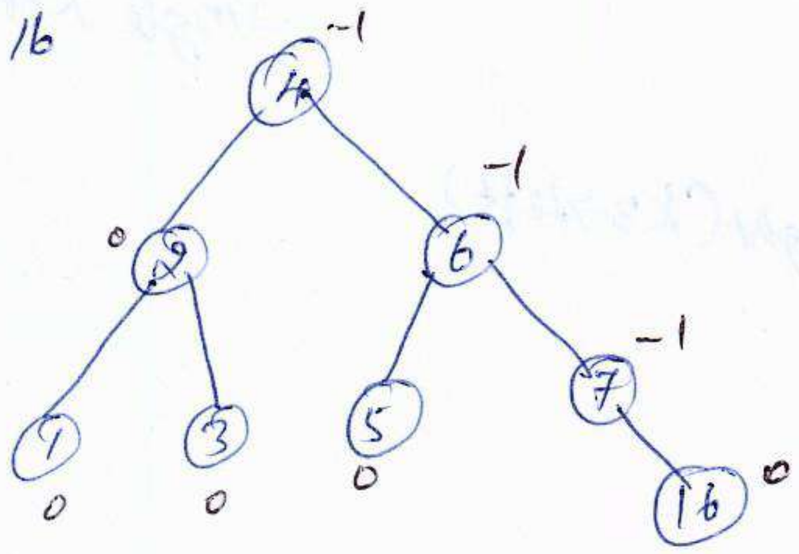
② single rotations
1. Single rotation with left
2. Single rotation with right



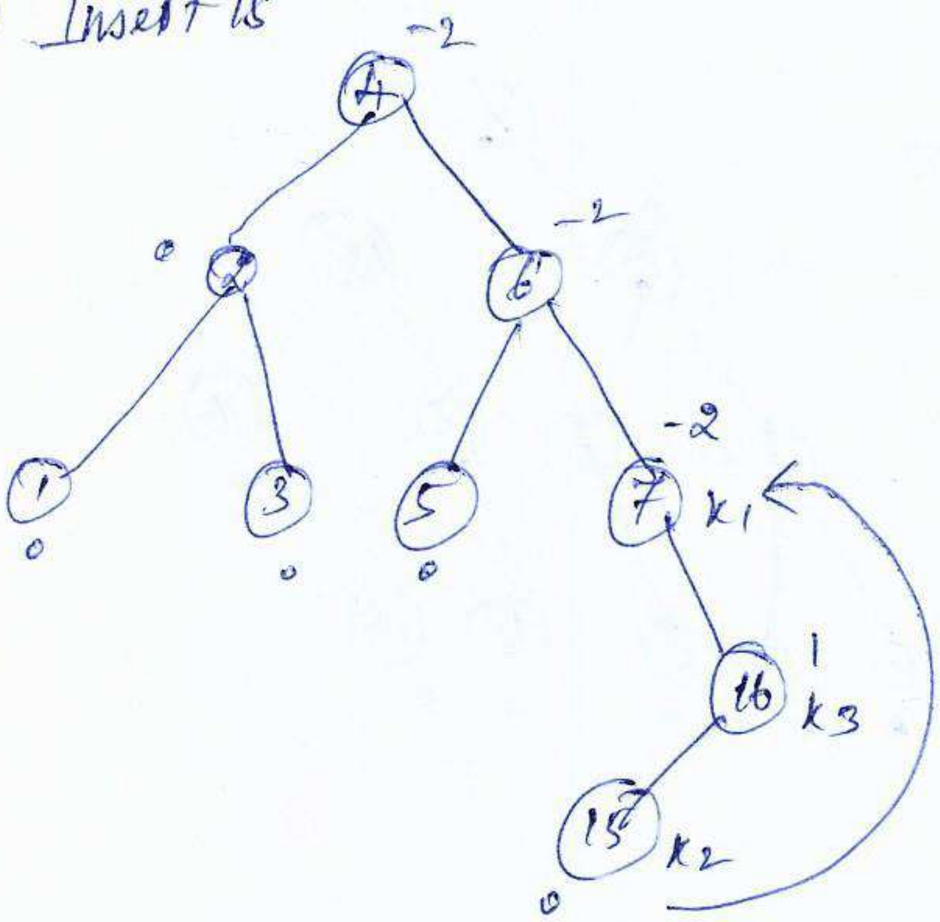
Example 4:
Construct the AVL tree with insertion of the elements: 16, 15, 14, 13, 12, 11, 10, 8.



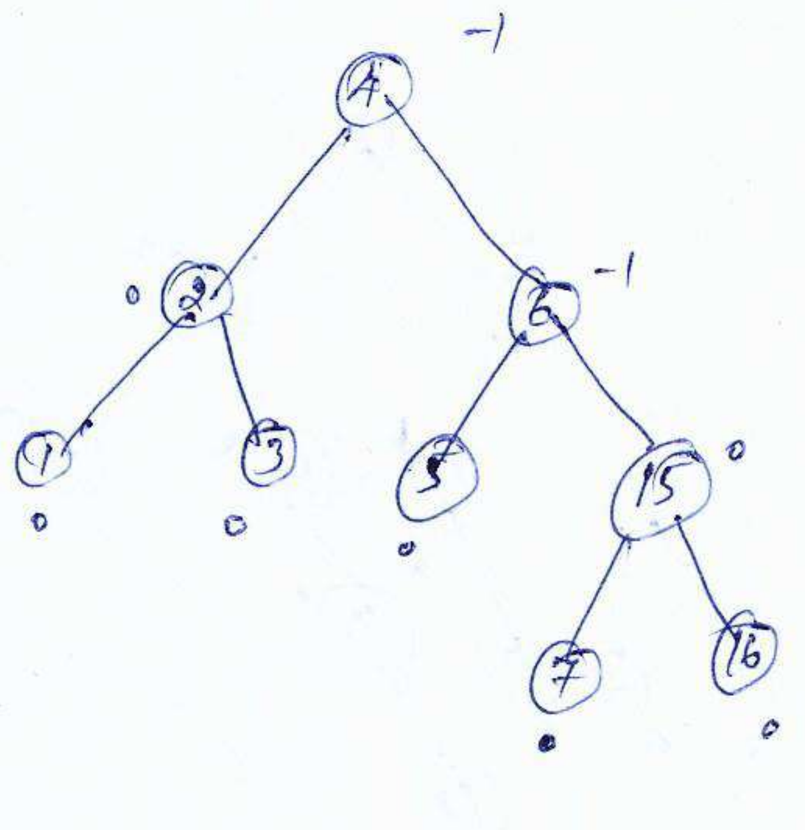
① Insert 16



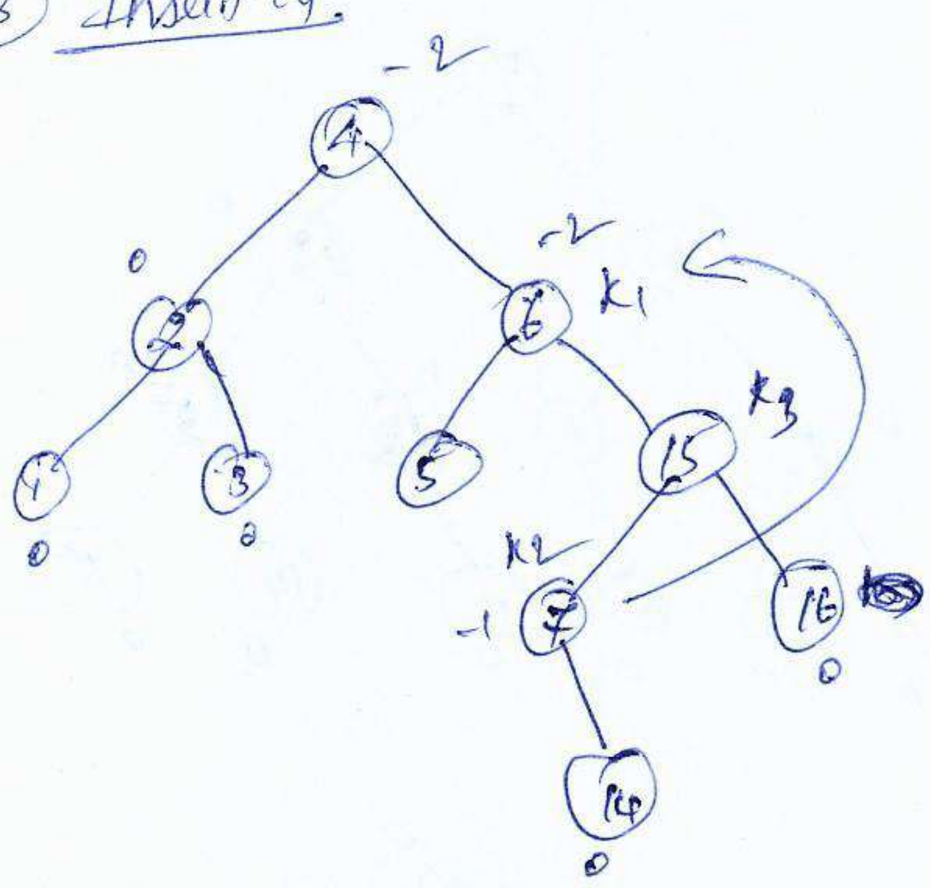
② Insert 15



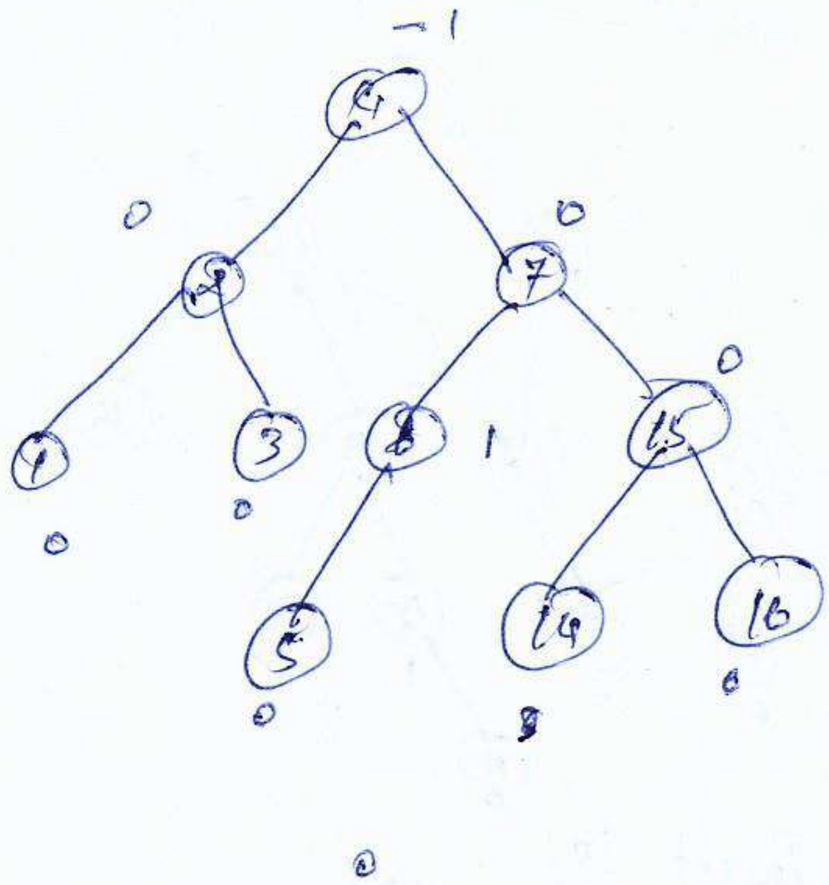
Double Rotation LR ⟹ Case 3



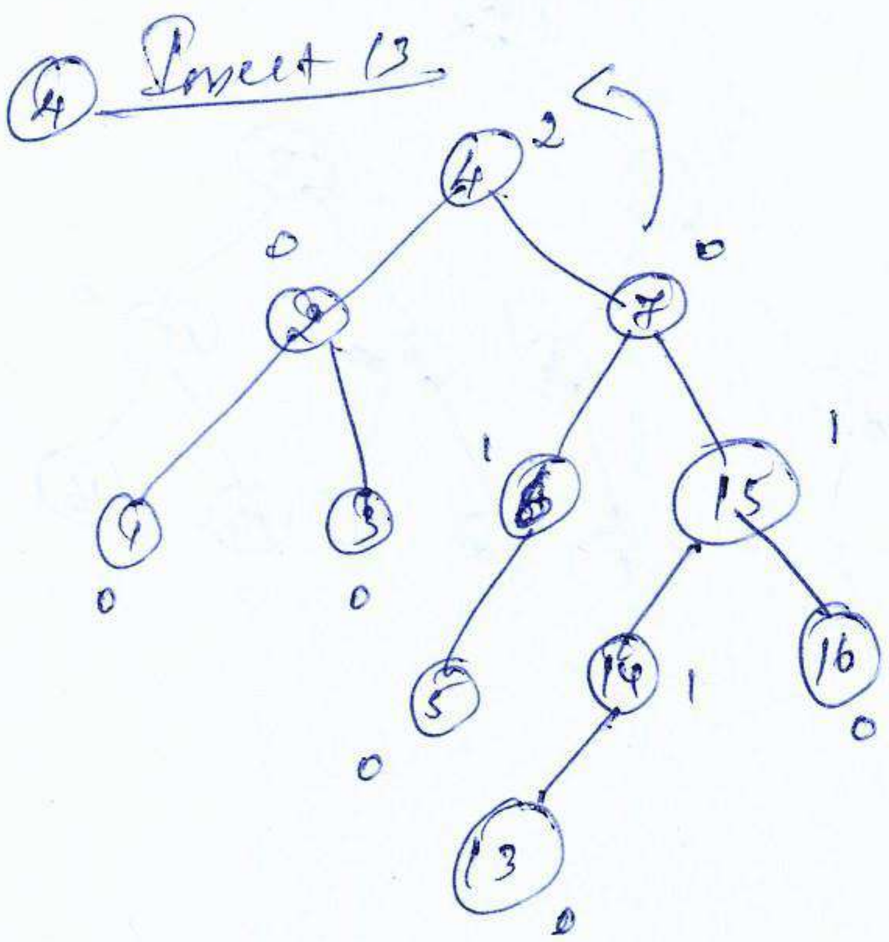
③ Insert 14.



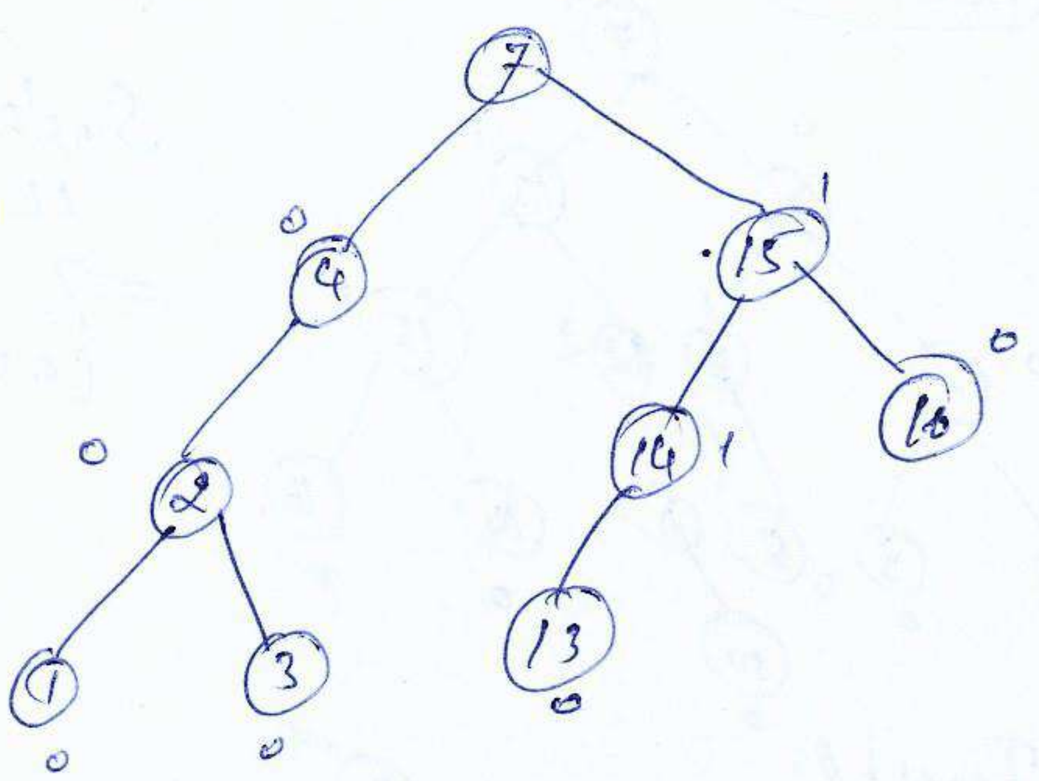
Double Rotation LR ⟹ Case 3



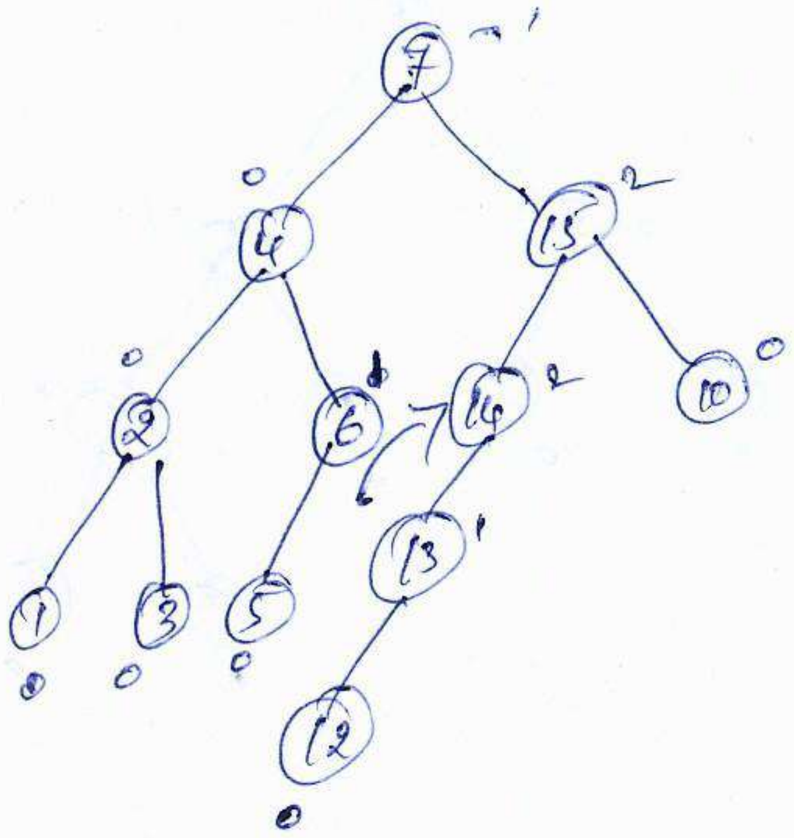
④ Insert 13.



Single Rotation RR ⟹ Case 4



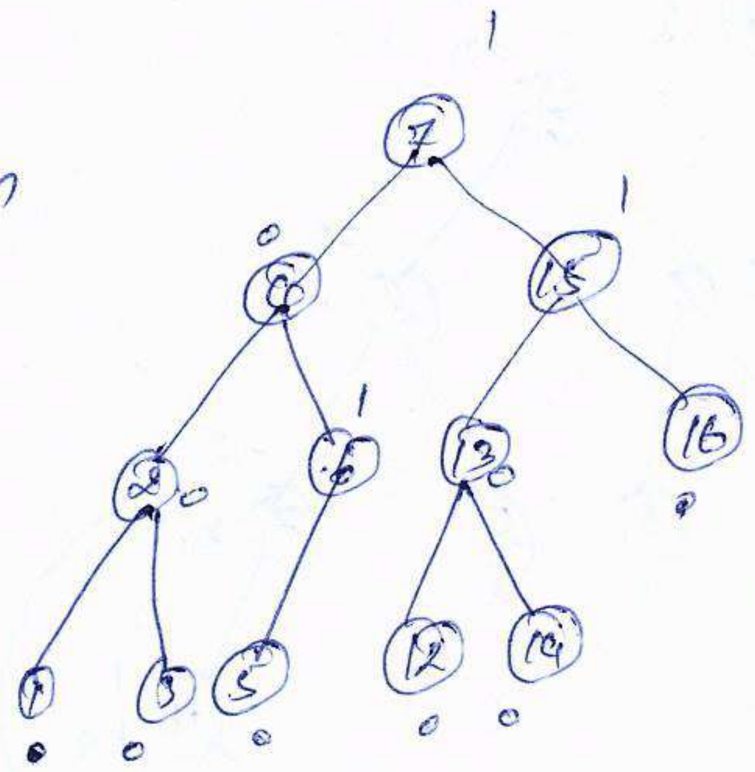
(5) Insert 12



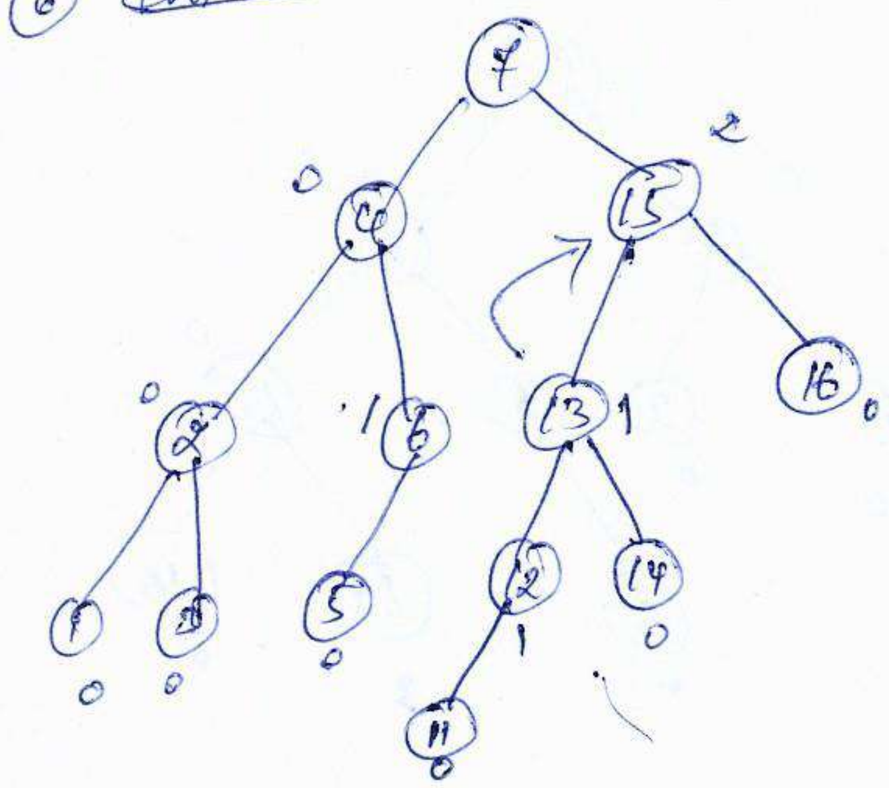
Single Rotation

LL

Case 1



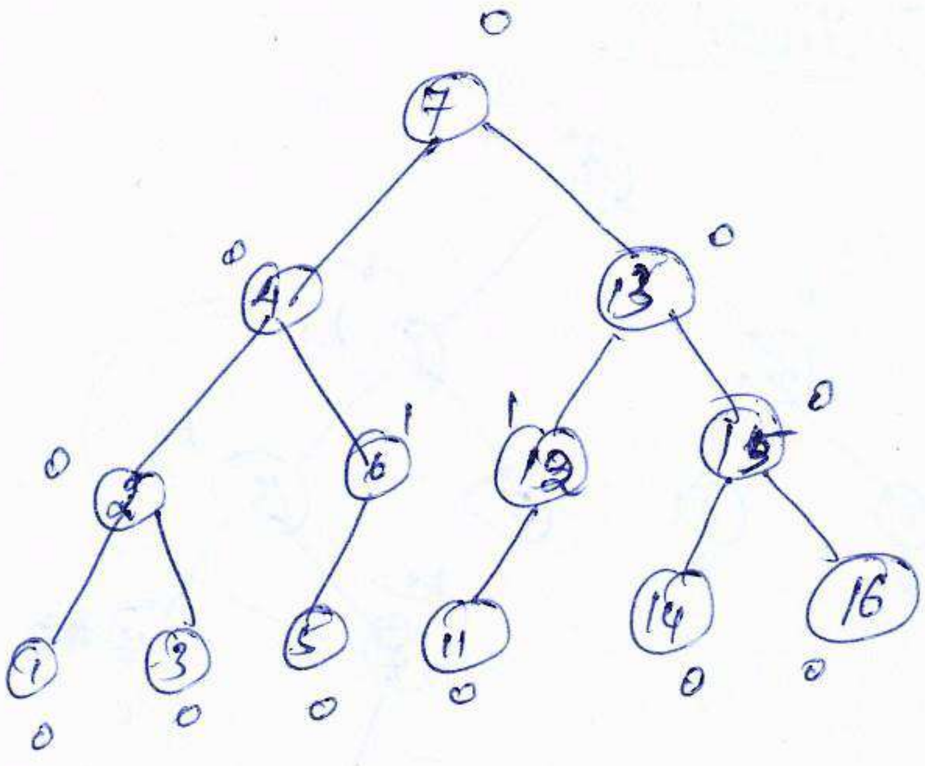
(6) Insert 11



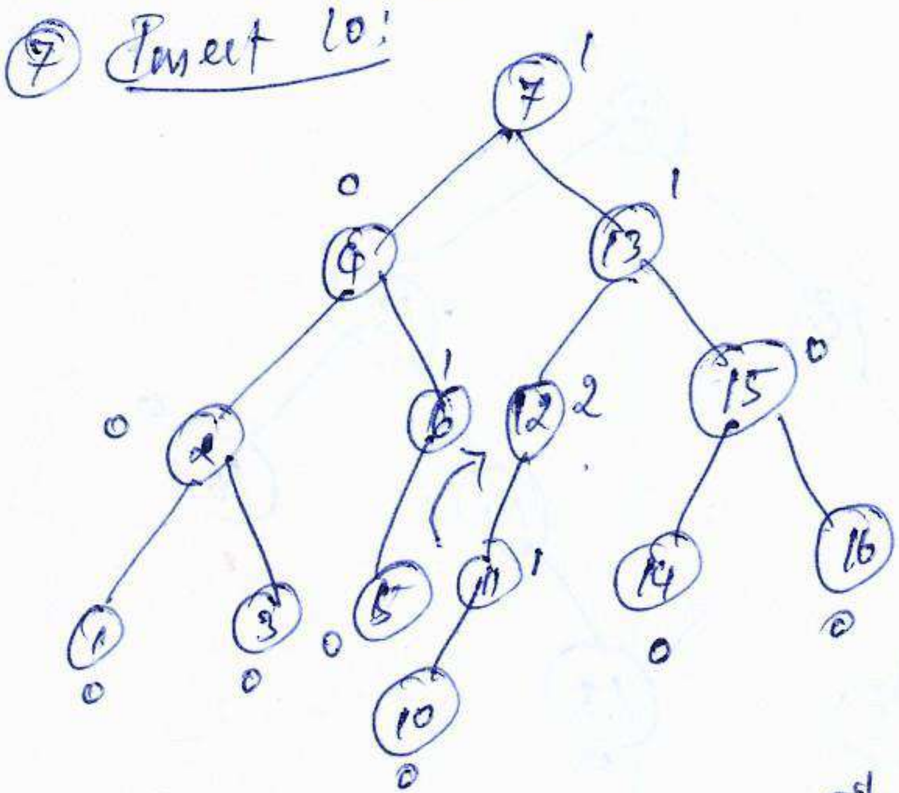
Single Rotation

LL

Case 1



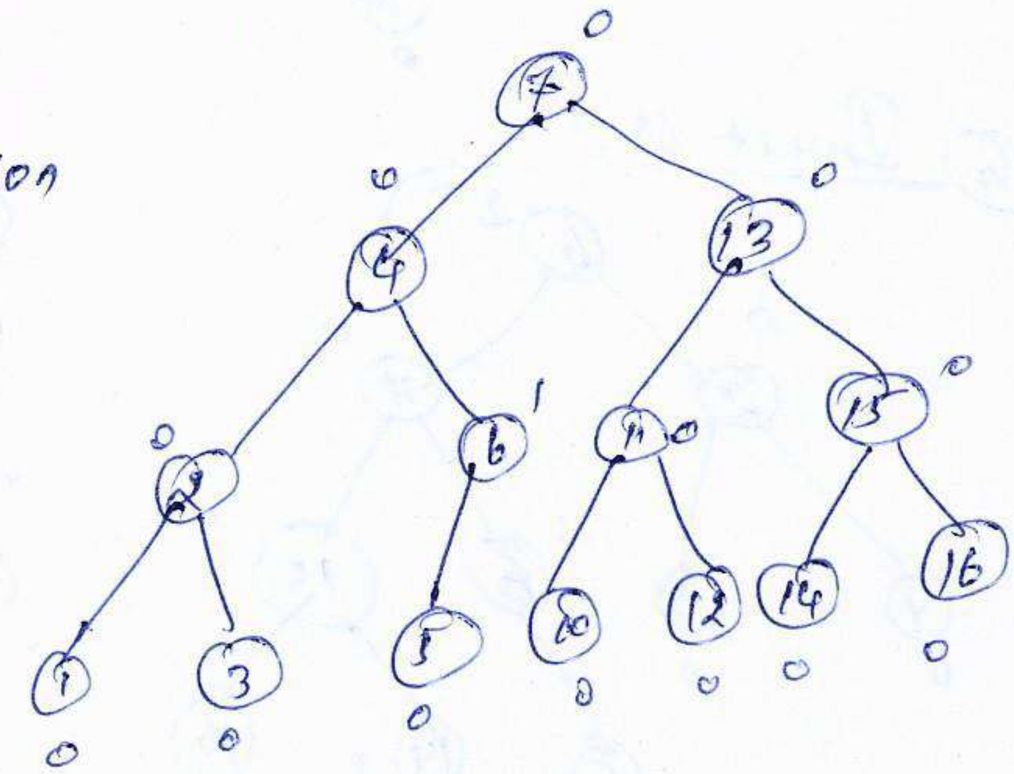
(7) Insert 10:



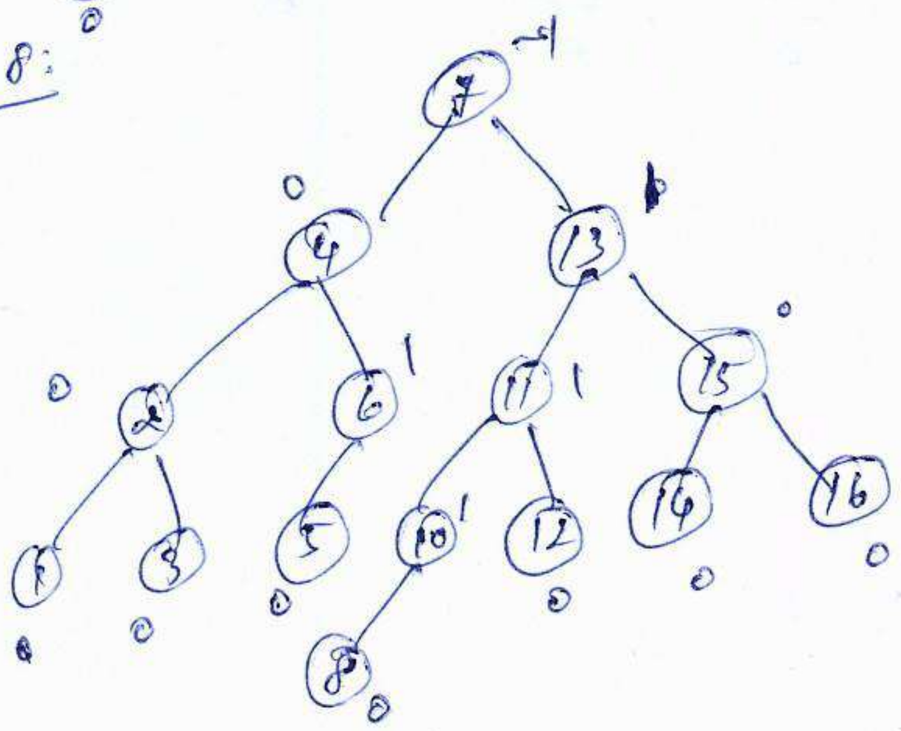
Single Rotation

LL

Case 1



(8) Insert 8:



Node Declaration:

```
struct AvlNode
{
  ElementType Element;
  AvlTree Left;
  AvlTree Right;
  int Height;
};
```

Height:

```
static int Height ( Position P )
{
  if ( P == NULL )
    return -1;
  else
    return P -> Height;
}
```

Insertion:

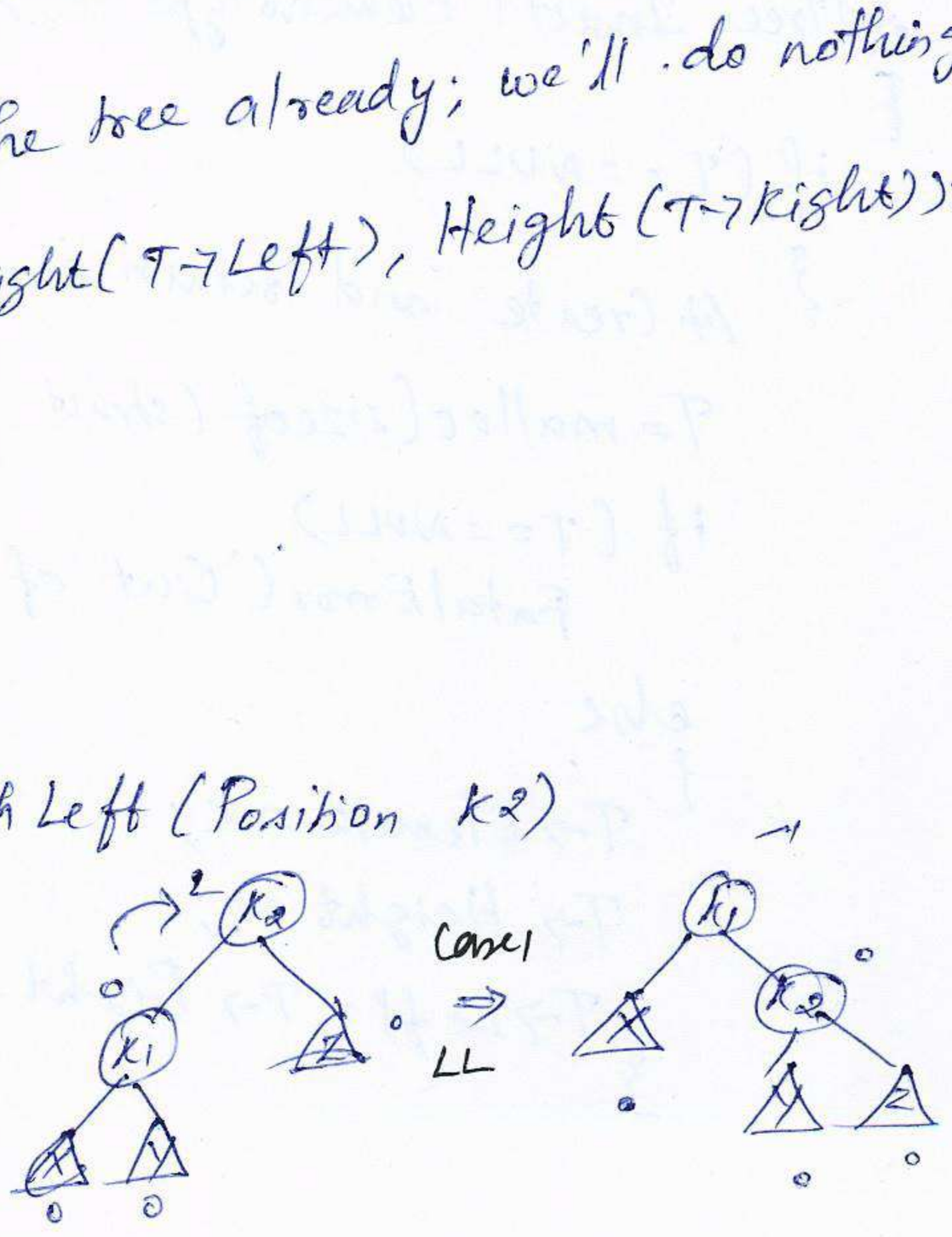
```
AvlTree Insert ( ElementType X, AvlTree T )
{
  if ( T == NULL )
  {
    /* Create and return a one-node tree */
    T = malloc ( sizeof ( struct AvlTNode ));
    if ( T == NULL )
      FatalError ("Out of Space !!!");
    else
    {
      T -> Element = X;
      T -> Height = 0;
      T -> Left = T -> Right = NULL;
    }
```

```
else if ( X < T->Element)
{
    T->Left = Insert( X, T->Left);
    if ( Height(T->Left) - Height(T->Right) == 2)
        if ( X < T->Left->Element)
            T = SingleRotateWithLeft(T);
        else
            T = DoubleRotateWithLeft(T);
}
else if( X > T->Element)
{
    T->Right = Insert (X, T->Right);
    if ( Height(T->Right) - Height(T->Left) == 2)
        if ( X > T->Right->Element)
            T = SingleRotateWithRight(T);
        else
            T = DoubleRotateWithRight(T);
}
/* Else X is in the tree already; we'll do nothing */
T->Height = Max( Height(T->Left), Height(T->Right)) + 1;
return T;
}
```

Single Rotation:

Case 1:

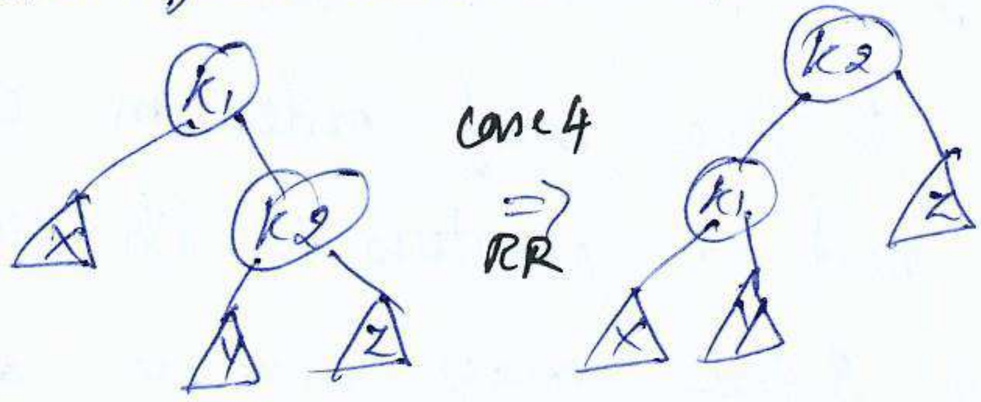
```
static Position SingleRotateWithLeft (Position k2)
{
    Position k1;
    k1 = k2->Left;
    k2->Left = k1->Right;
    k1->Right = k2;
```



```
K2 -> Height = Max (Height (K2 -> Left), Height (K2 -> Right)) +1;
K1 -> Height = Max (Height (K1 -> Left), K2 -> Height) +1;
return K1; /* New root */
}
```

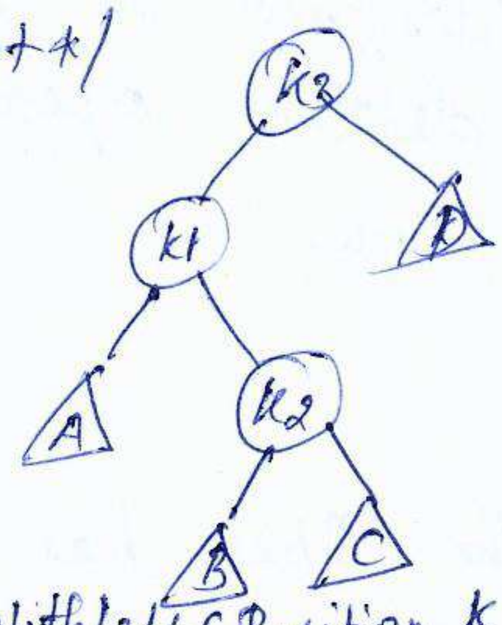
Case 4:

```
static Position SingleRotateWithRight (Position K1)
{
Position K2;
K2 = K1 -> Right;
K1 -> Right = K2 -> Left;
K2 -> Left = K1;
K1 -> Height = Max (Height (K1 -> Right), Height (K1 -> Left)) +1;
K2 -> Height = Max (Height (K2 -> Right), K1 -> Height) +1;
return K2; /* New root */
}
```

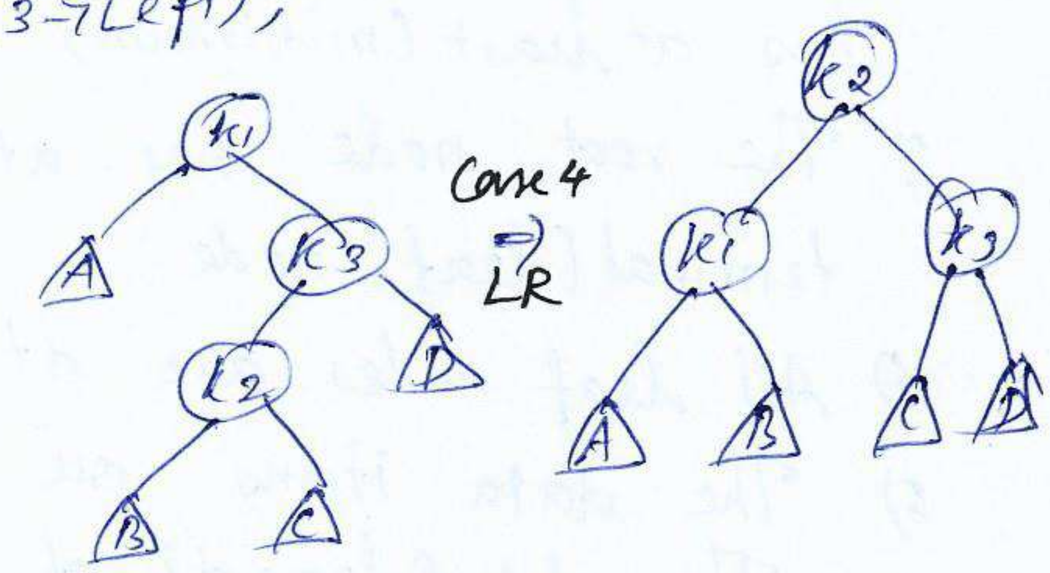


Double Rotation:

Case 2:




```
Static Position DoubleRotateWithLeft (Position K3)
{
/* Rotate between K1 and K2 */
K3 -> Left = SingleRotateWithRight (K3 -> Left);
/* Rotate between K3 and K2 */
return SingleRotateWithLeft (K3);
}
```



Case 3:

```
static Position DoubleRotateWithRight
                (Position K1)
{
/* Rotate between K3 and K2 */
K1 -> Right = SingleRotateWithLeft (K1 -> Right);
/* Rotate between K1 and K2 */
return SingleRotateWithRight (K1);
}
```

B-Trees:-

A B-tree is a specialized M-way tree that is widely used for disk access (secondary storage).

* In database applications, where huge volumes of data is handled, the search tree cannot be accomodated in primary memory. B-trees are primarily meant for secondary storage.

A B-tree of order m can have a maximum of m-1 keys and m pointers to its subtrees.

* A B tree may contain a large no. of key values and pointers to subtrees.
* Storing the large no. of keys in a single node keeps the height of the tree relatively small.

A B-tree is designed to store sorted data and allows search, insertion and deletion operations to be performed in logarithmic amortized time.

Properties:-

1) Every node in the Btree has at most (maximum) m children.
2) Every node in the B tree except the root node and leaf nodes has at least (minimum) m/2 children.
3) The root node has at least two children if it is not a terminal (leaf) node.
4) All leaf nodes are at same level.
5) The data items are stored at leaves.
6) The data (elements) of a node are stored in ascending order.

An internal node in the Btree can have 'n' no. of children, where, $0 \leq n \leq m$.

* It is not necessary that every node has the same no. of children, but the only restriction is that the node should have at least m/2 children.

An example of a B-tree of order 5 is shown in Fig.

* All nonleaf nodes have between 3 and 5 children.
* Here, we have L=5. Since L is 5, each leaf has between 3 and 5 data items.

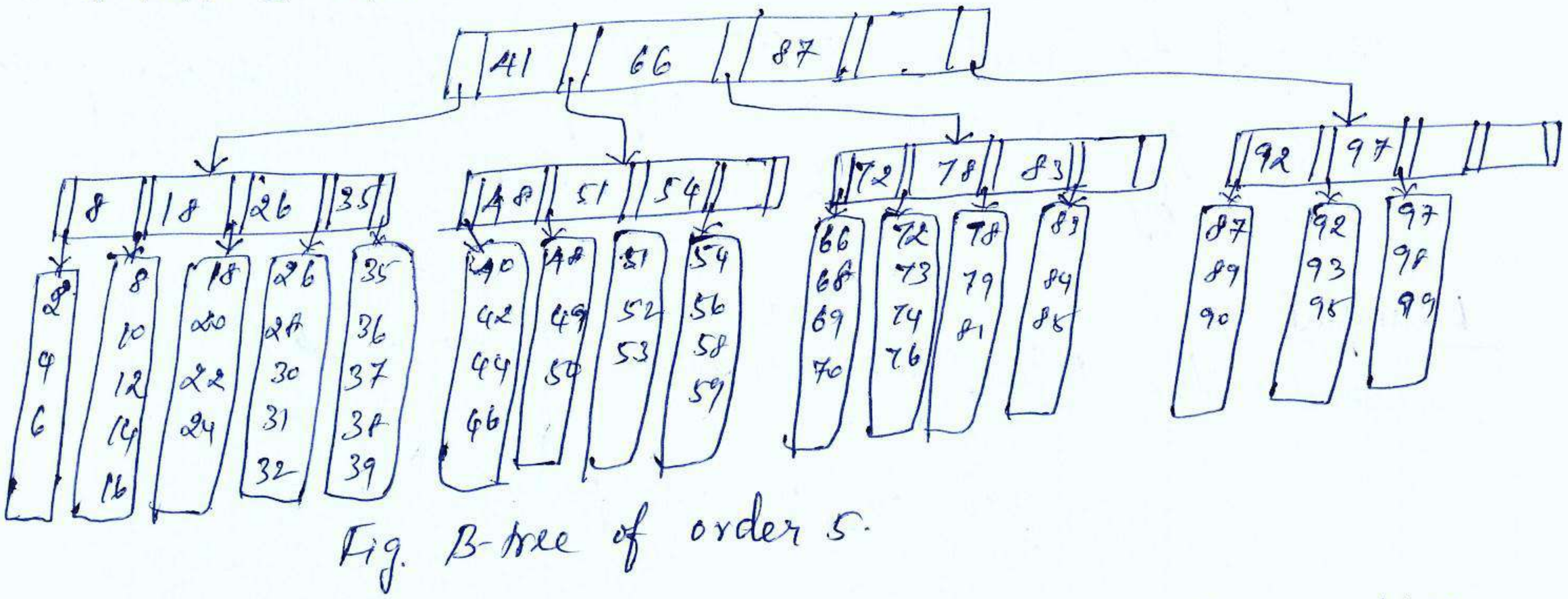


Fig. B-tree of order 5.

1) Search:

Searching for an element in a B-tree is similar to binary search trees.

* Comider the B-tree given in above Fig. To search an data item 70, we begin at the root node.
* The root node has three values 41, 66, 87. Since $66 \leq 70 \leq 87$, we traverse the 3rd subtree in the next level.
* The subtree has three values 72, 78, 83. Since $70 \leq 72$.
* So we found the element here as the 4th value, then the search is successful.

2) Insertion:

In a B-tree, all insertions are done at the leaf node level.

1) Search the B-tree to find the leaf node where the new key value should be inserted.

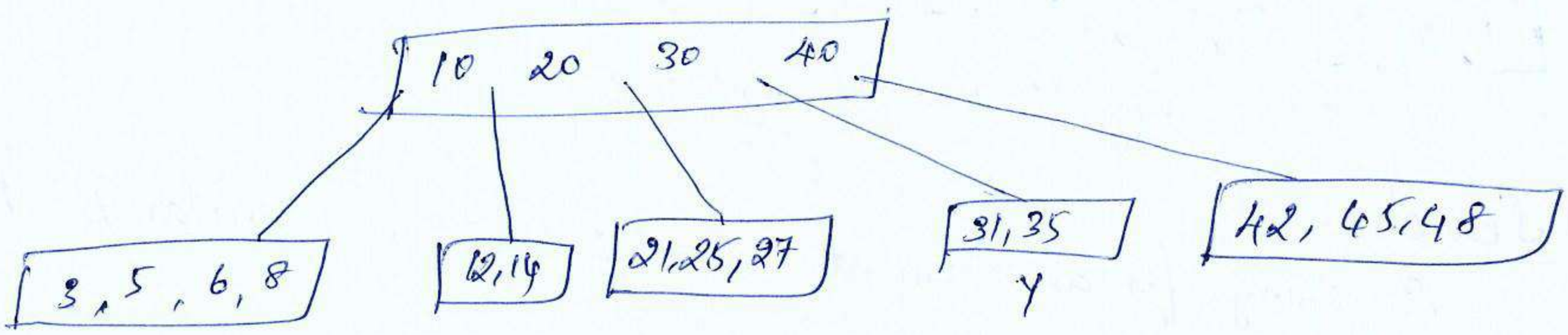
2) If the leaf node is null, i.e. it contains m-1 key values, then insert the new element in the node keeping the node's elements ordered.

3). If the leaf node is full i.e, the leaf node already contains m-1 key values, then,

(i) insert the new value in order into the existing set of keys,
(ii) split the node at its median into two nodes
(iii) push the median element up to its parent's node. If the parent's node is already full, then split the parent node by following the same steps.

Example①

Comider the B-tree of order 5 given below. Insert 38, 7



Insert 38:

Since 38 lies between 30 and 40, the subtree shown as 'y' is selected for insertion. As there is space in the node, 38 is inserted and insertion process completes.

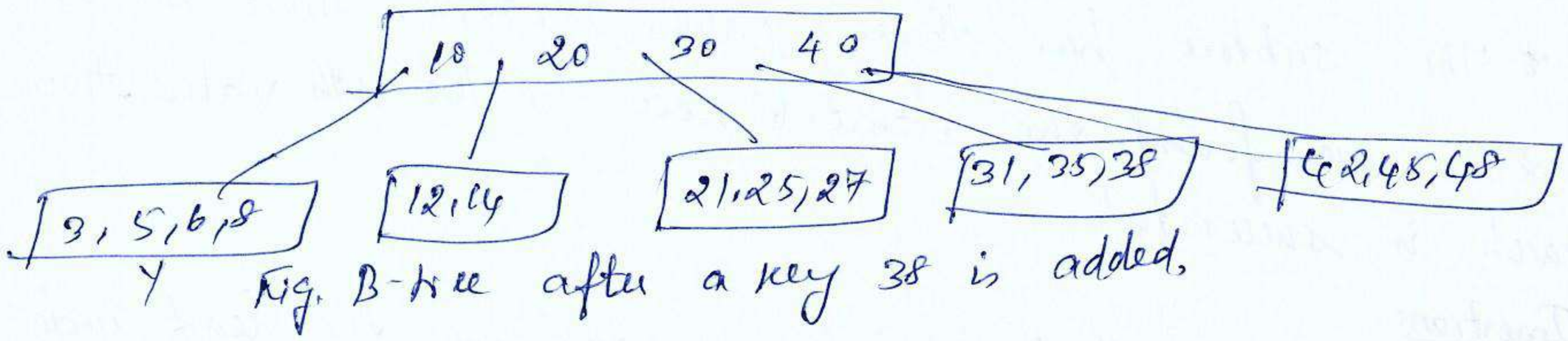
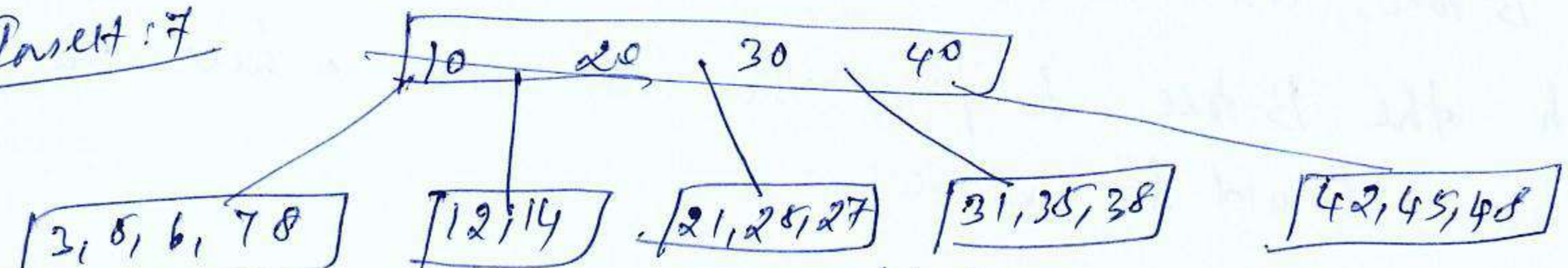


Fig. B-tree after a key 38 is added.

Insert: 7



B-tree after a key 7 is added.

After the insertion of the new key with value 7, an overflow in node Y has been occurred. Here the leaf node is full, the no. of keys in B-tree of order 5 cannot be more than 4. The leaf node is split into two half as below.

6

3,5,6,7,8 ⇒ 3,5 7,8

Now the center key 6 moves up to the parent's node.

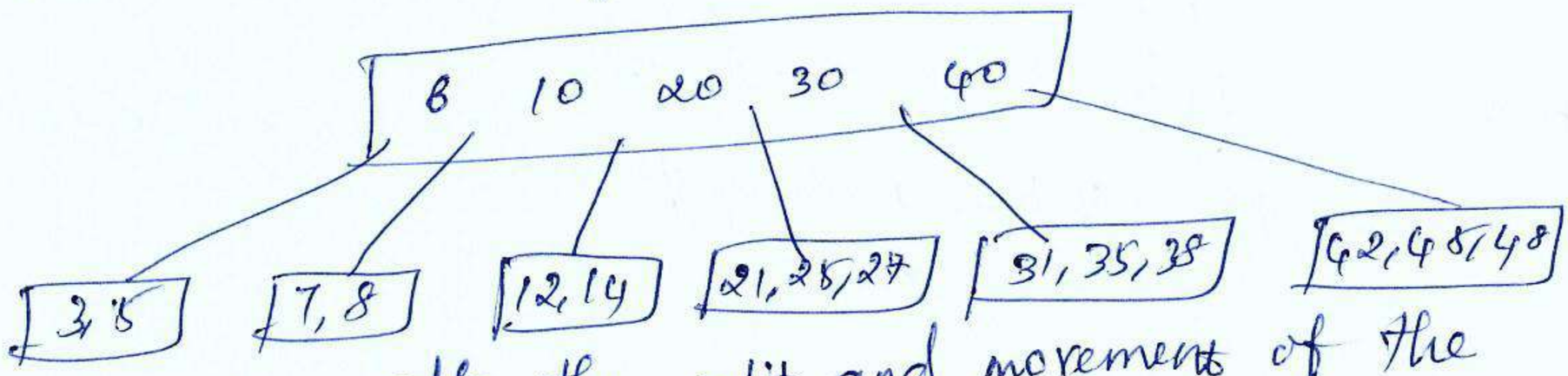


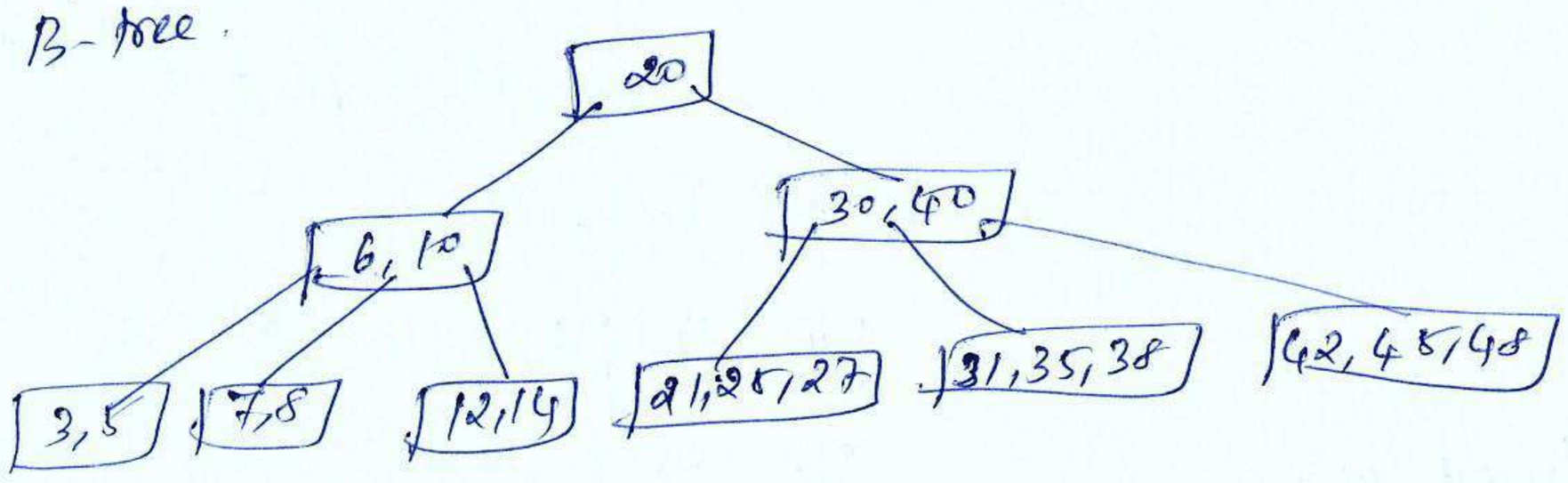
Fig. B-tree after the split and movement of the center key in the root node.

Now, the root node contains 5 keys and the root node is full, therefore split it into two halves as

20

6,10,20,30,40 ⇒ 6,10 30,40

Center key with value 20 is stored in a new node which gives the final B-tree after insertion of a key with value 7 in the B-tree.



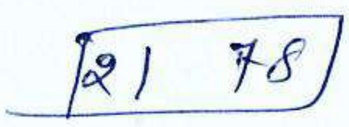
Example ②:

Insert the elements into a B-tree of order 5.

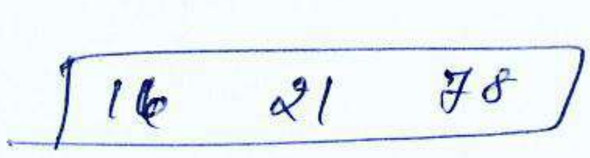
78, 21, 16, 11, 97, 85, 74, 63, 45, 42, 57, 20, 14, 19, 32, 30, 31.

Insert 78

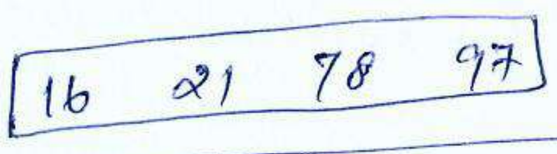
Insert 21



Insert 16

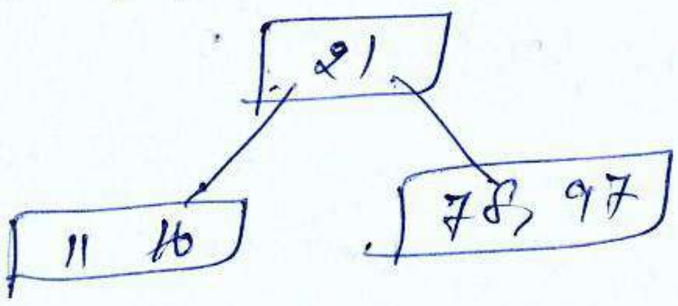


Insert 97

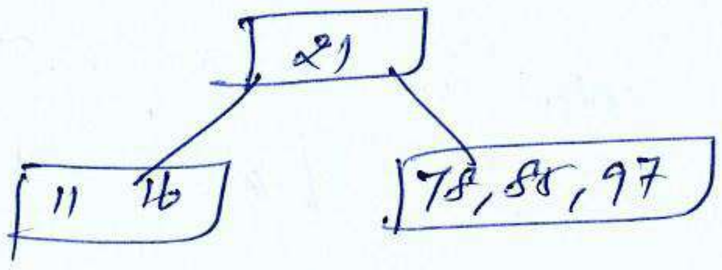


Insert 11

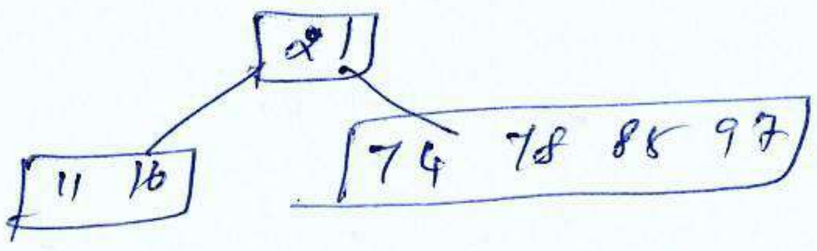
Now the B-tree node is full. Split it into two halves



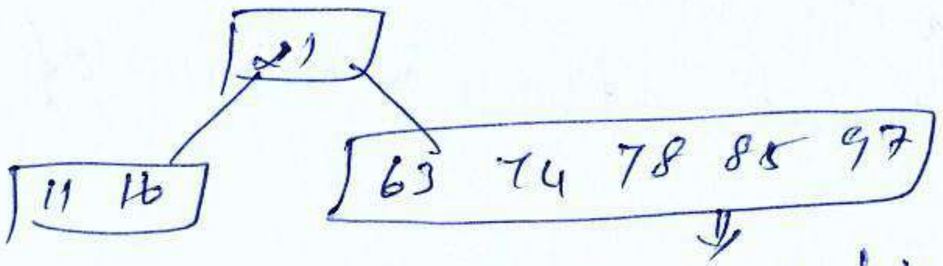
Insert 85



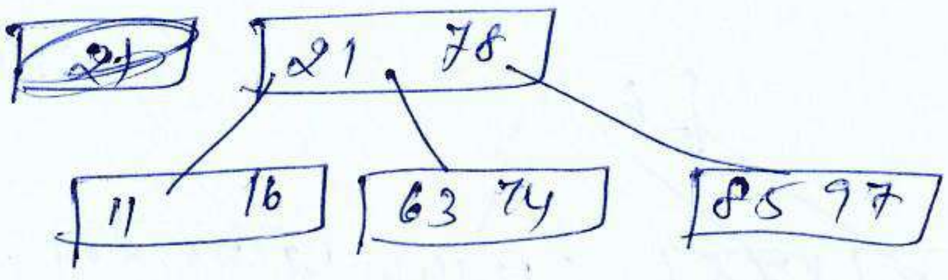
Insert 74



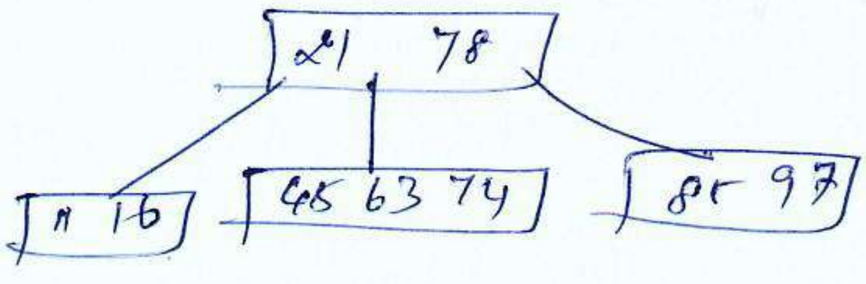
Insert 63



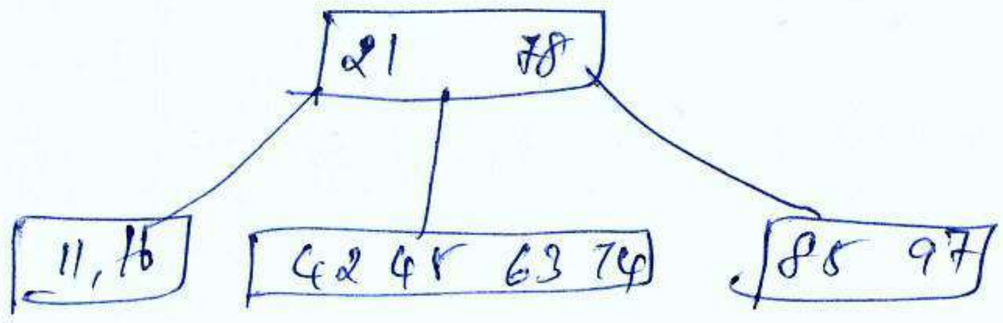
this node is full. Split it into two halves



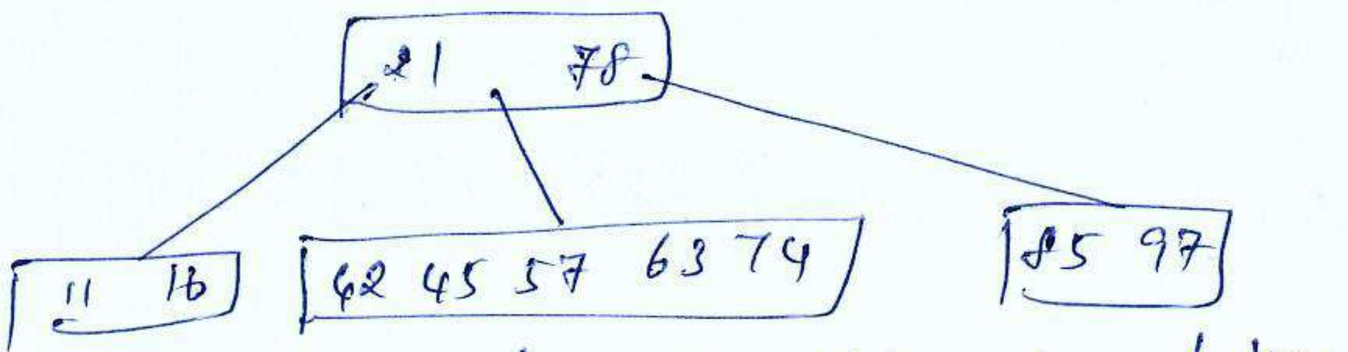
Insert 45



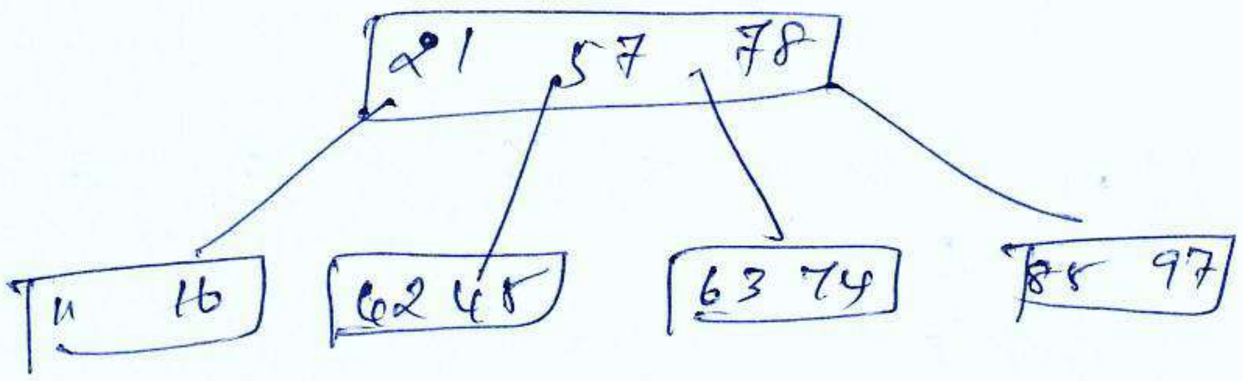
Insert 42



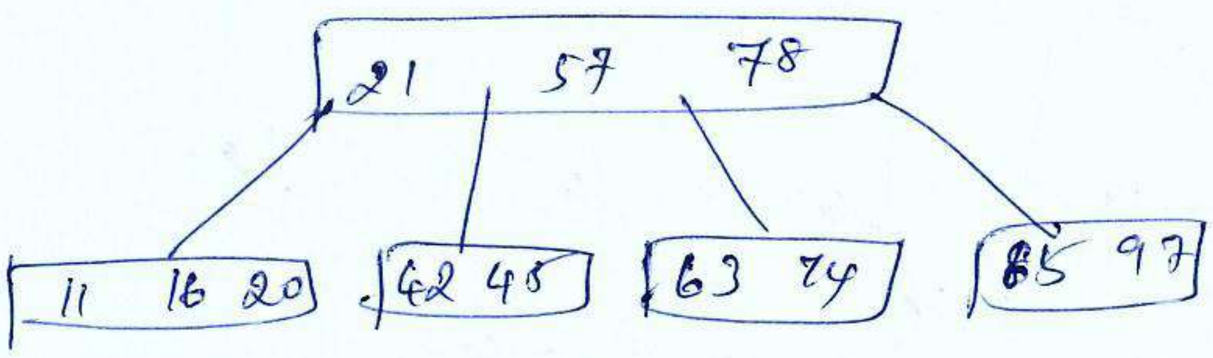
Insert 57



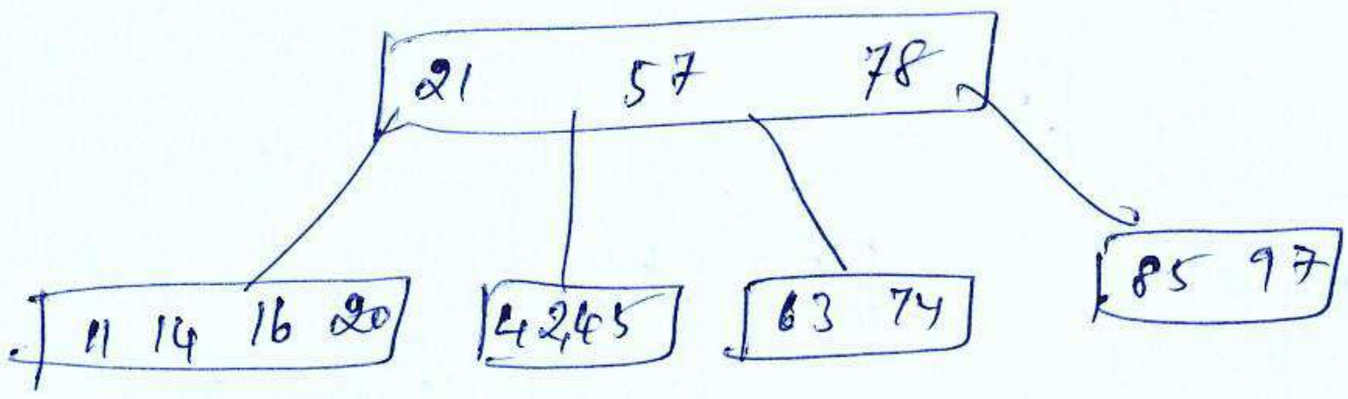
↓ Split it into two halves



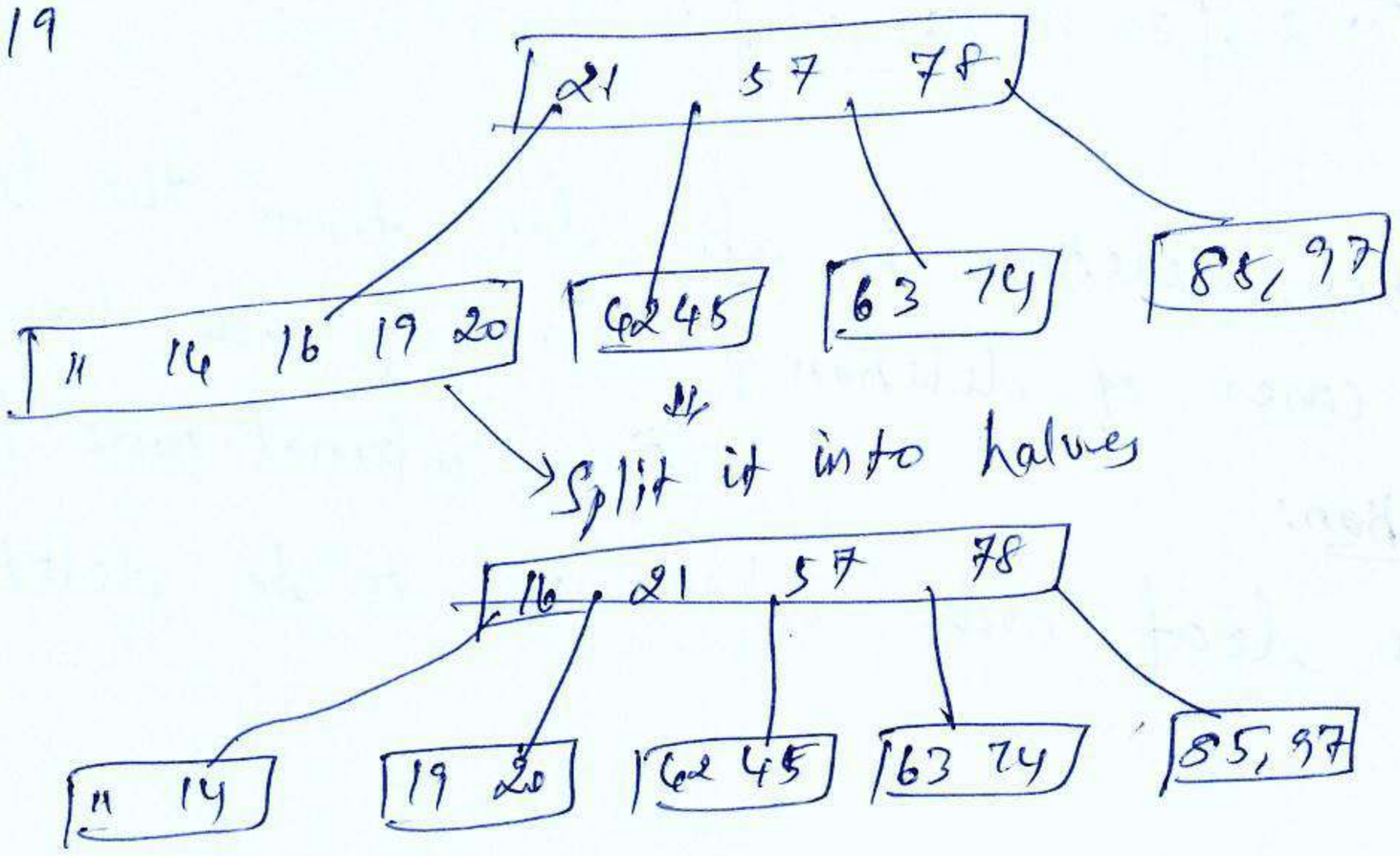
Insert 20



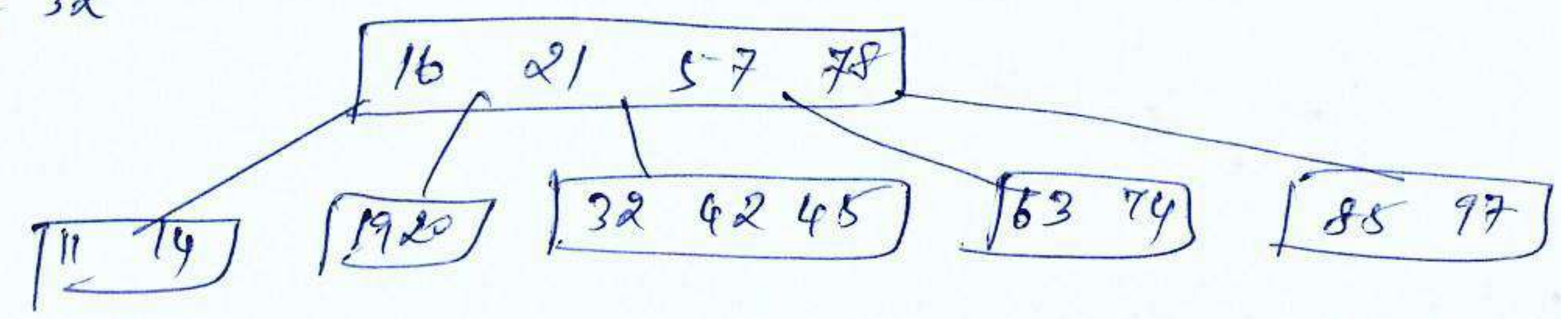
Insert 14



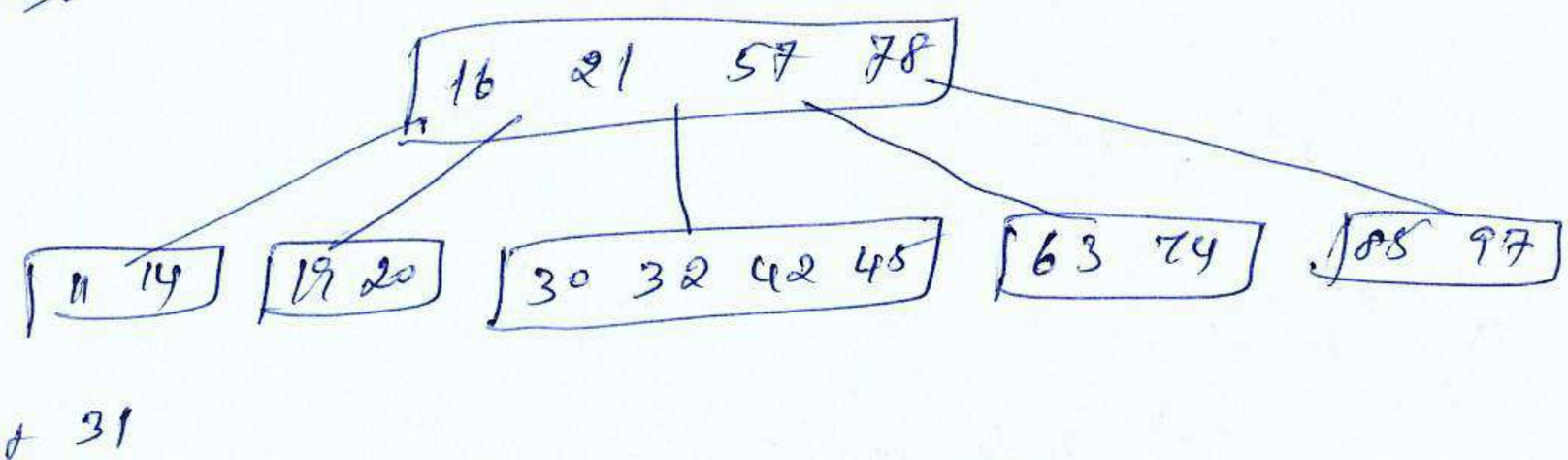
Insert 19



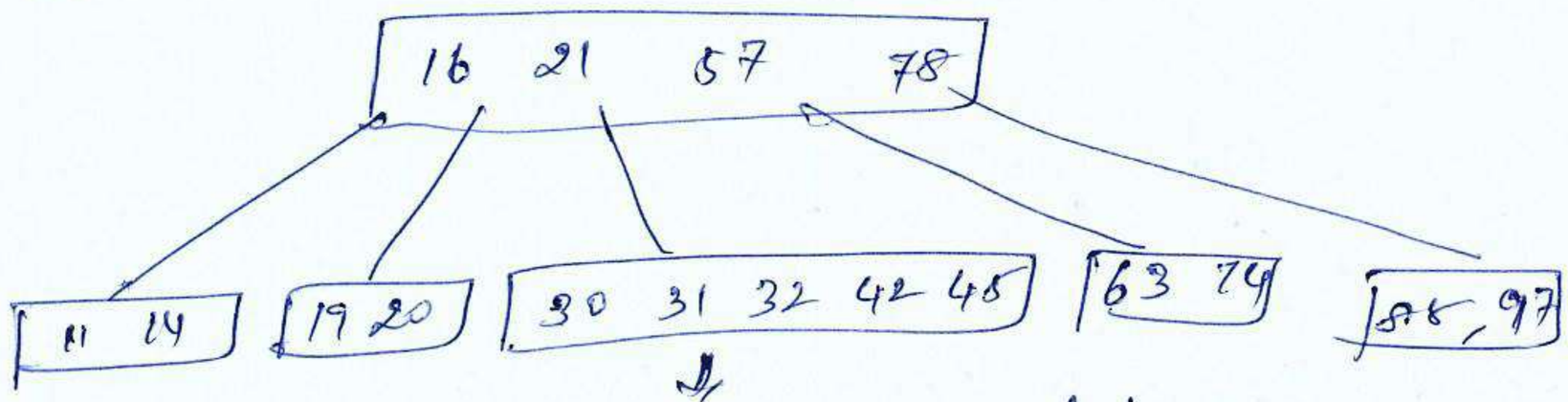
Insert 32



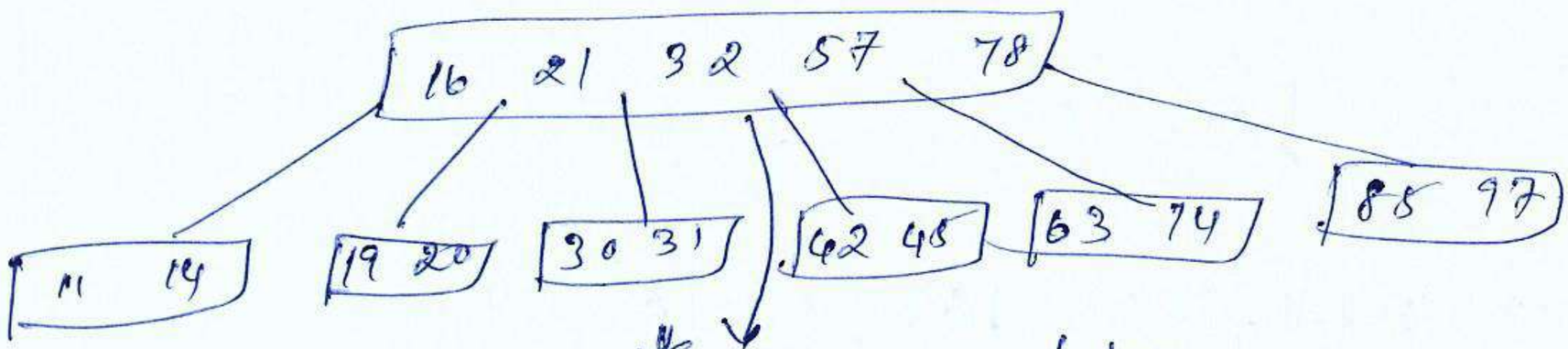
Insert 30



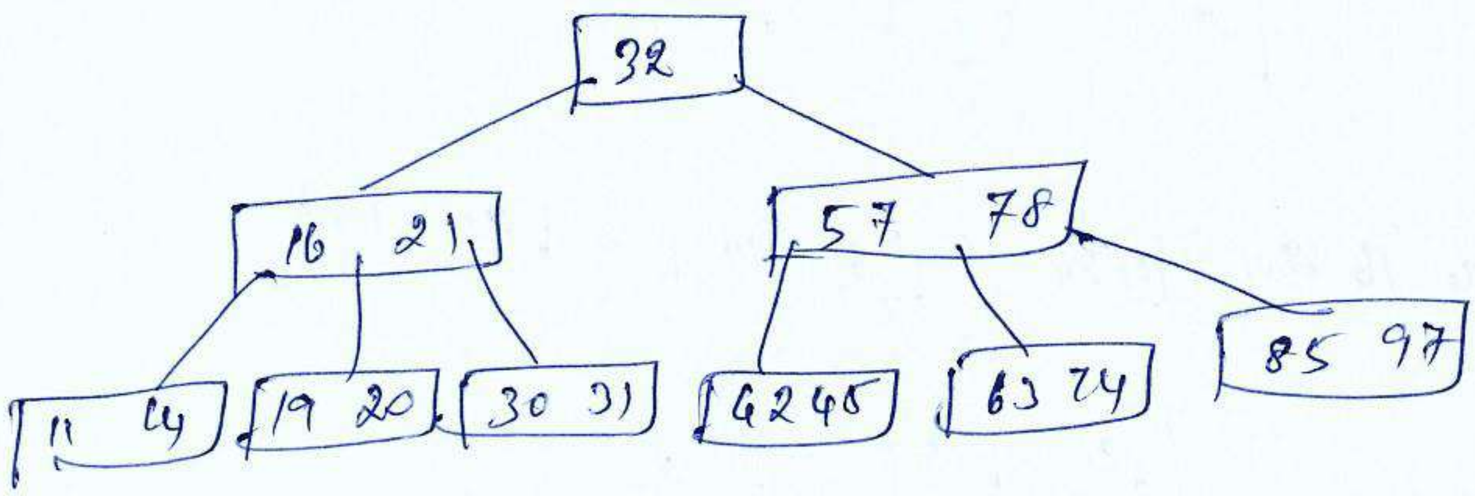
Insert 31



Split it into two halves



Split it into two halves



3) Deletion:

Like Insertion, deletion is also done from the leaf nodes.
There are two cases of deletion ⇒ ① a leaf node has to be deleted.
② an internal node has to be deleted.

Leaf Node Deletion:

1) Locate the leaf node which has to be deleted.

2) If the leaf node contains more than the minimum no. of key values (more than m/2 elements), then delete the value.

3). Else if the leaf node does not contain m/2 elements, then fill the node by taking an element either from the left or from the right sibling.

(a) If the left sibling has more than the minimum no. of key values, push its largest key into its parent's node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

(b) Else, if the right sibling has more than the minimum no. of key values, push its smallest key into its parent node and pull down the intervening element from the parent node to the leaf node where the key is deleted.

4. Else, if both the left and right siblings contain only the minimum no. of elements, then create a new leaf node by combining the two leaf nodes and the intervening elements of the parent node. If pulling the intervening element from the parent node leaves it with less than the minimum no. of keys in the node, then propagate the process upwards, thereby reducing the height of B-tree.

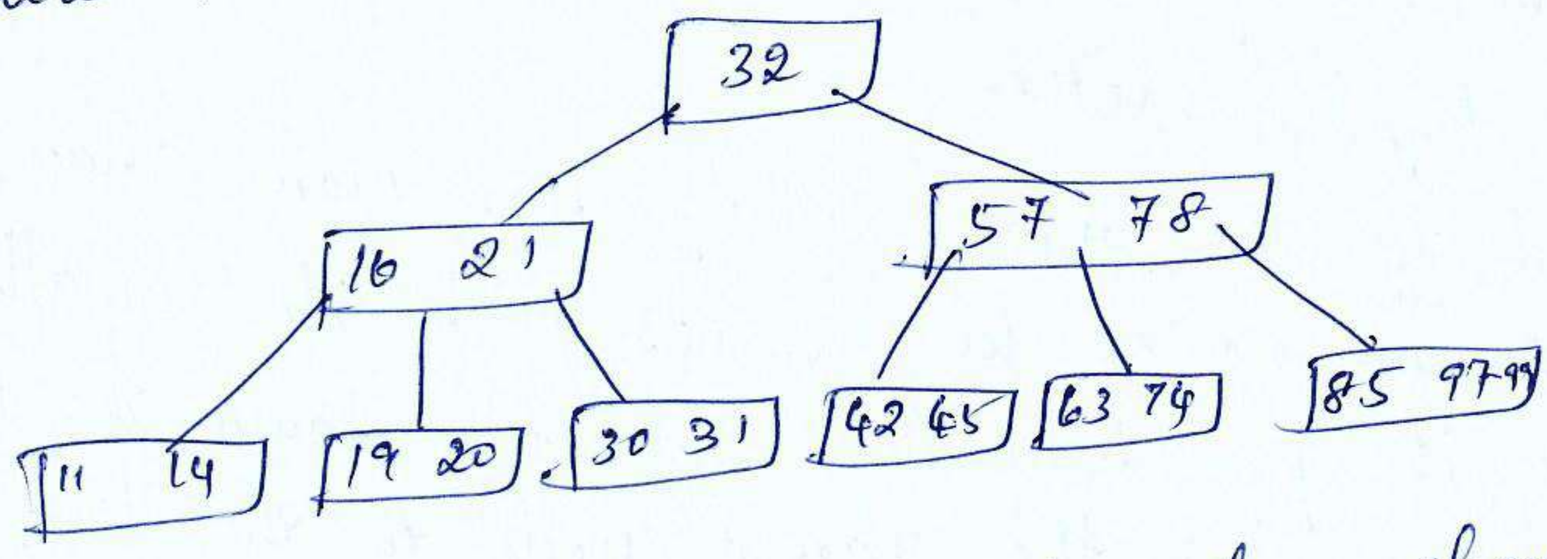
Internal node deletion:

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

* This predecessor or successor will always be in the leaf node.
* So the processing will be done as if a value from the leaf node has been deleted.

Example①:

Delete the nodes 74, 32 from a B-tree of order 5.

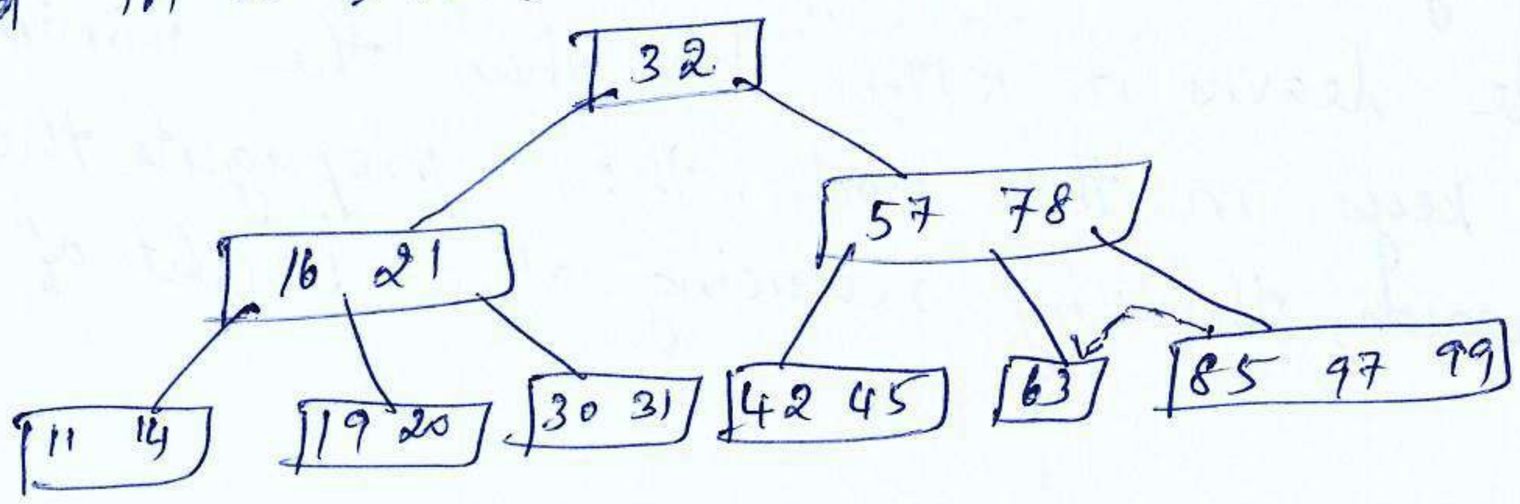


Every node except the root node should have at least

$\frac{5-1}{2} = 2$ keys.

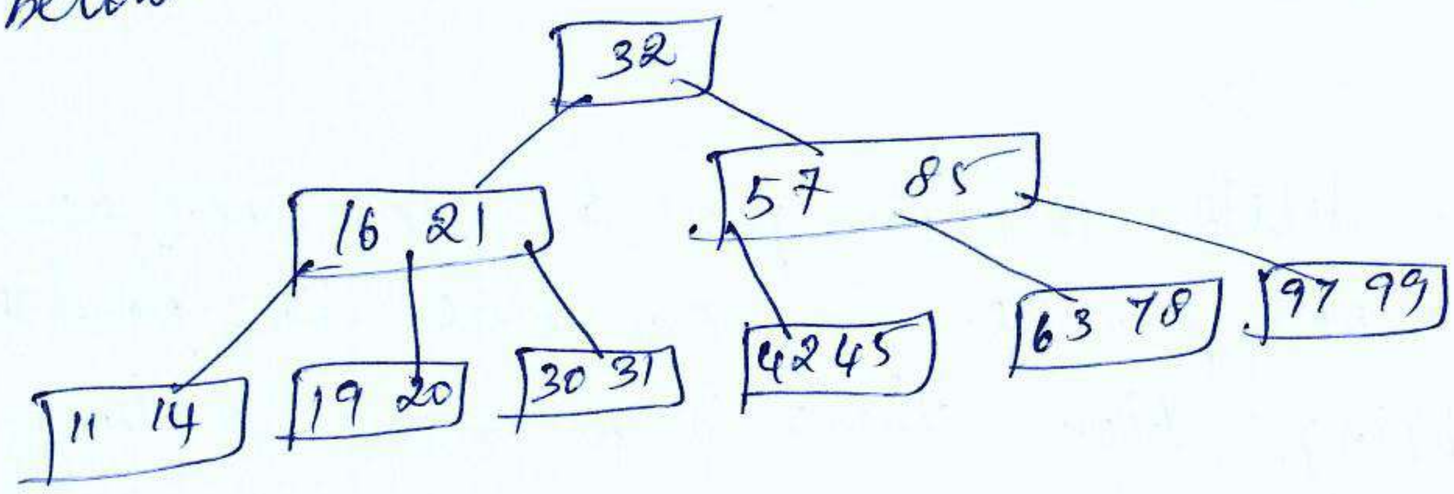
Deletion of 74:

Step 1: Locate 74, since it is in a leaf node, it can be deleted in a straight way.



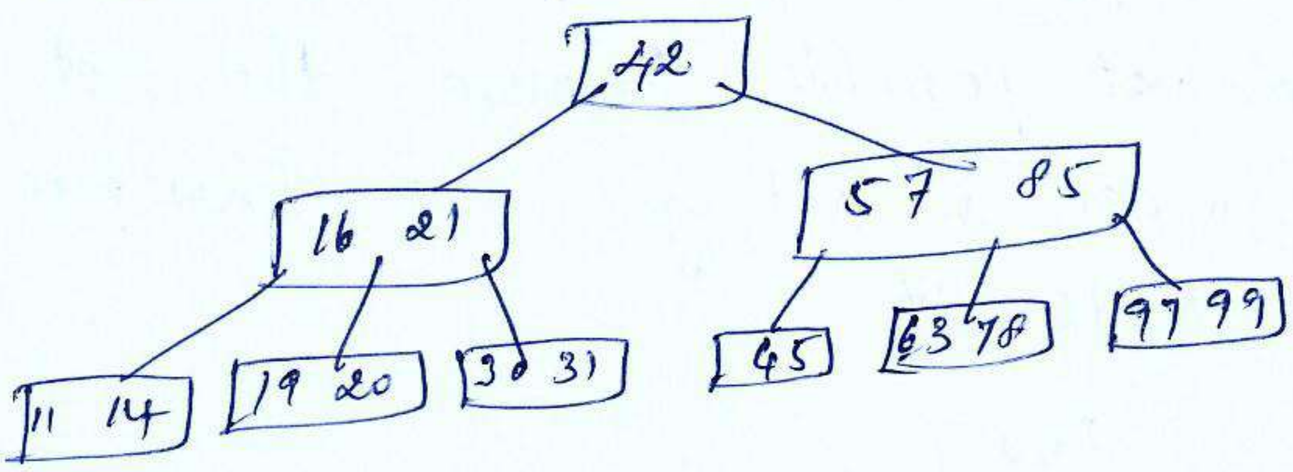
Step 2:

After deletion of 74, the node is left with only one key [63]. It can borrow a key from its right neighbor as the right neighbor has one extra key. The final tree is shown below:



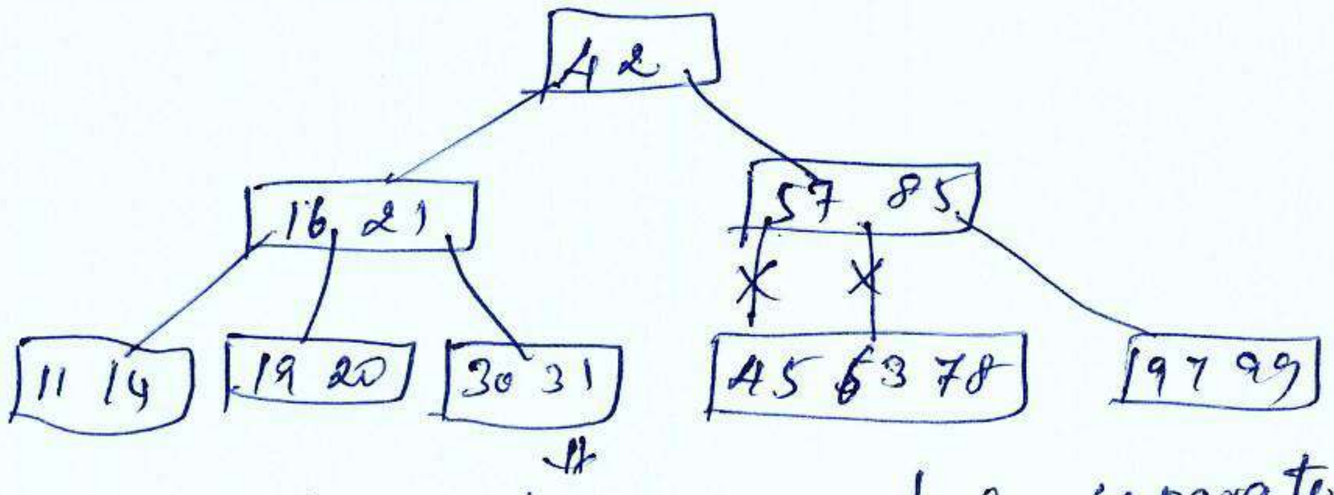
Deletion of 32:

Step 1: Since 32 is not a leaf node, it is replaced with its successor. Successor of 32 is 42.

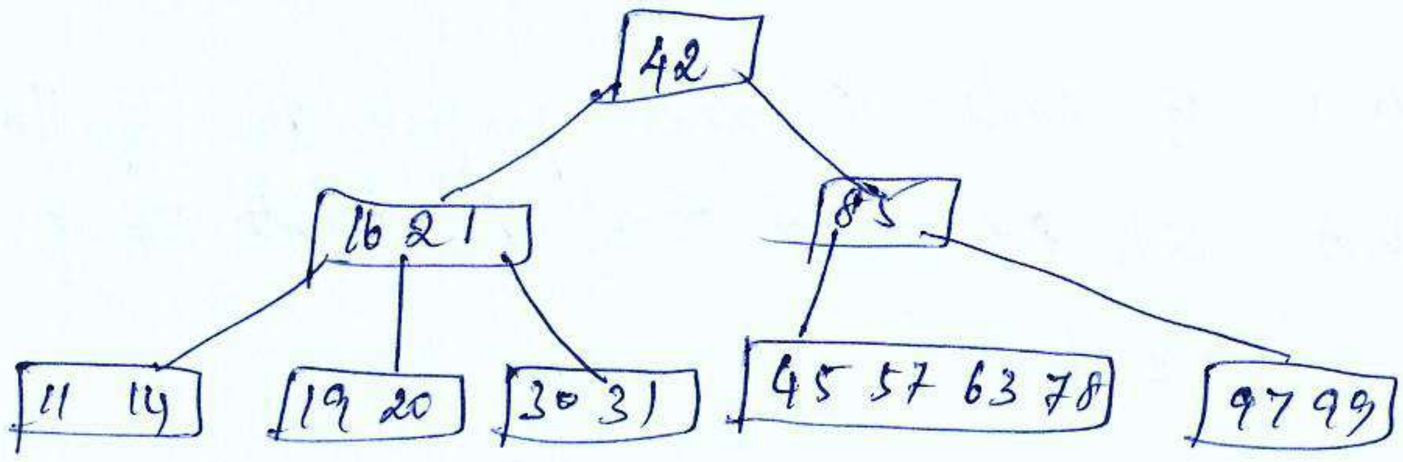


Step 2:

Node 45 does not satisfy the minimum key requirement condition. It does not have a left sibling. Its right sibling does not have an extra key. So Node [45] is merged with [63 78] along with their common parent 57.

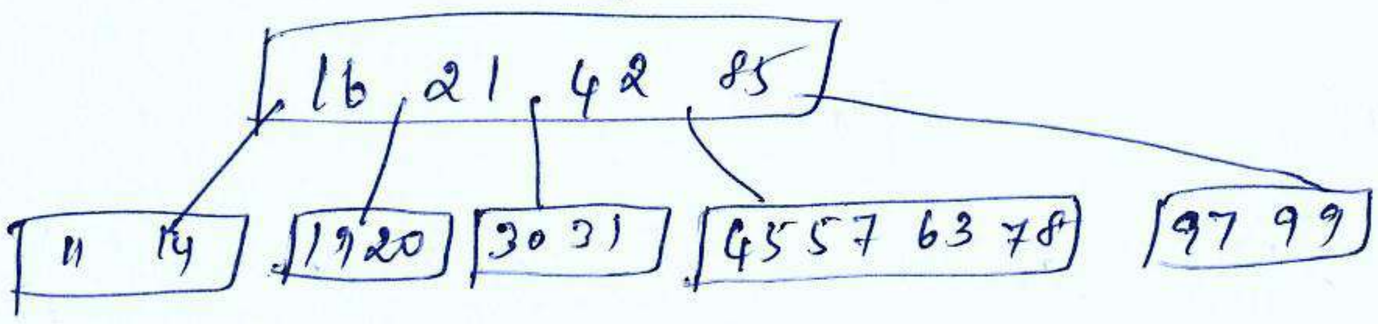


Now we have to give two separate siblings, this is not possible. Therefore Node 57 is combined with his children [45 63 78]. Now the B-tree becomes.



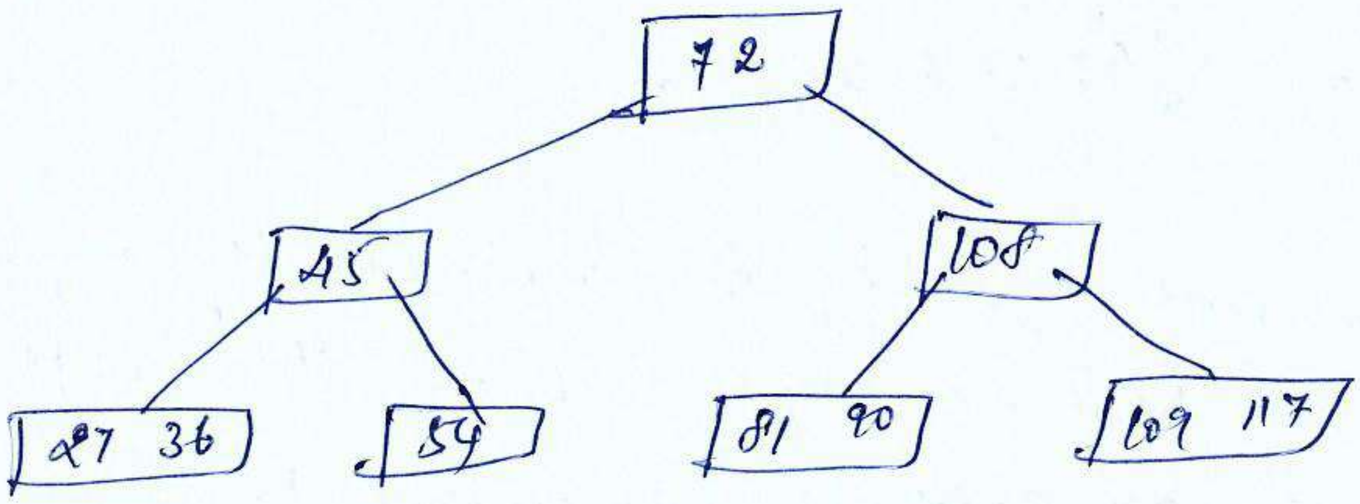
Step 3:

Node 85 does not satisfy the minimum key requirement condition. It does not have a right sibling. Its left sibling [16 21] does not have an extra key. Therefore, Node 85 is merged with [16 21] along with their common parent 42.

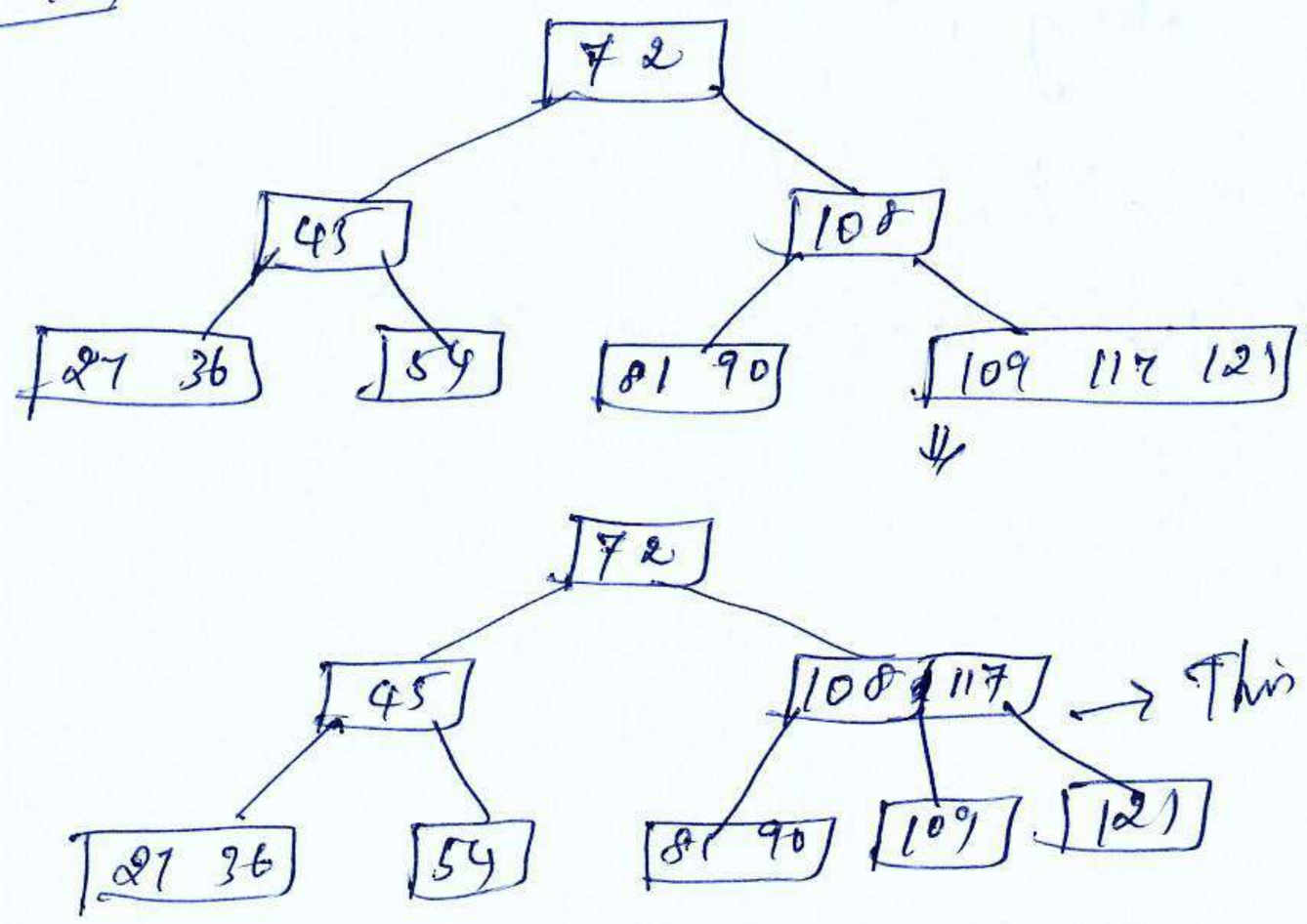


Example ②:

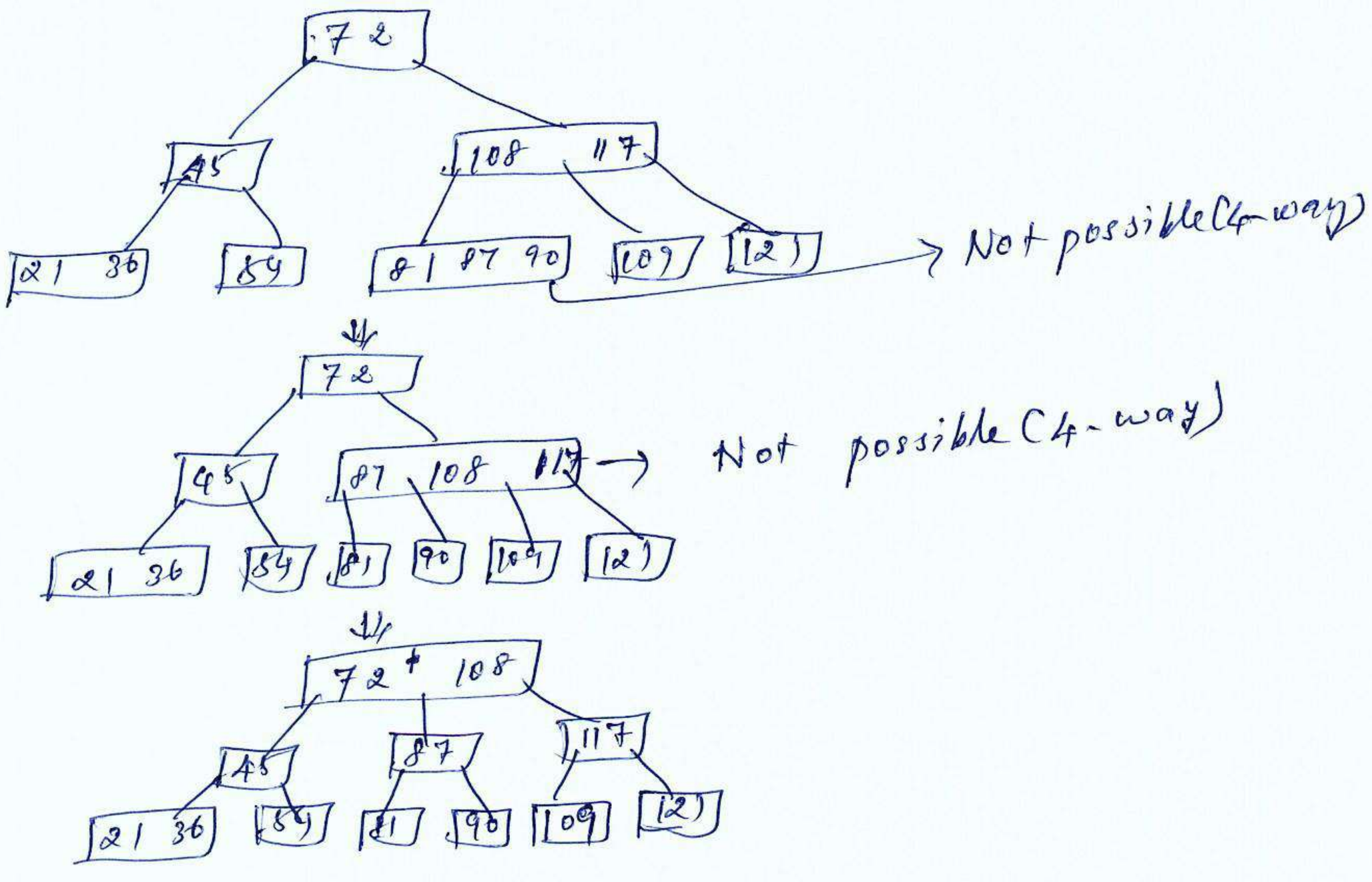
Consider the B-tree of order 3 and perform the following operations: (a) insert 121, 87 and then (b) delete 36, 109.



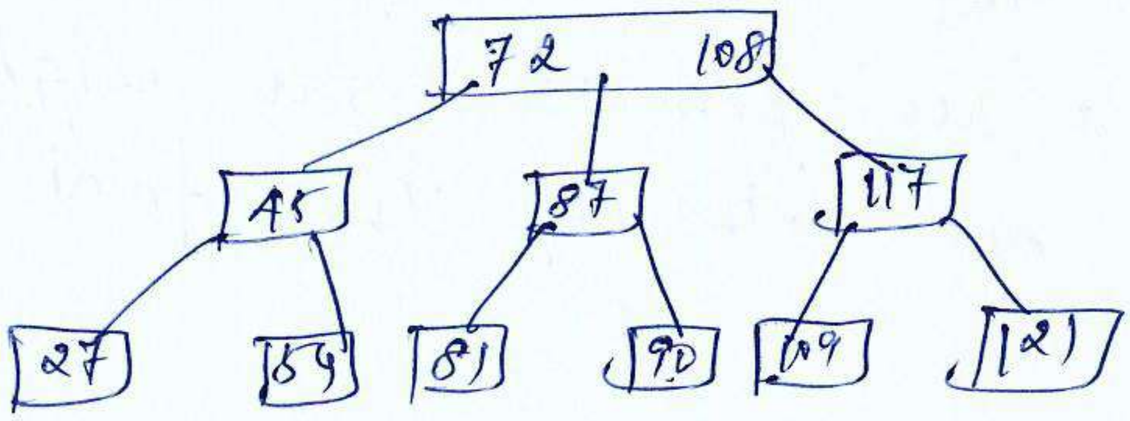
Insert 121



Insert 87

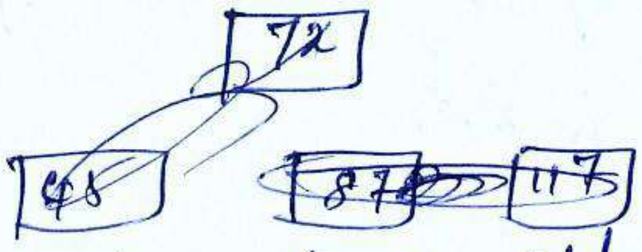


Delete 36:

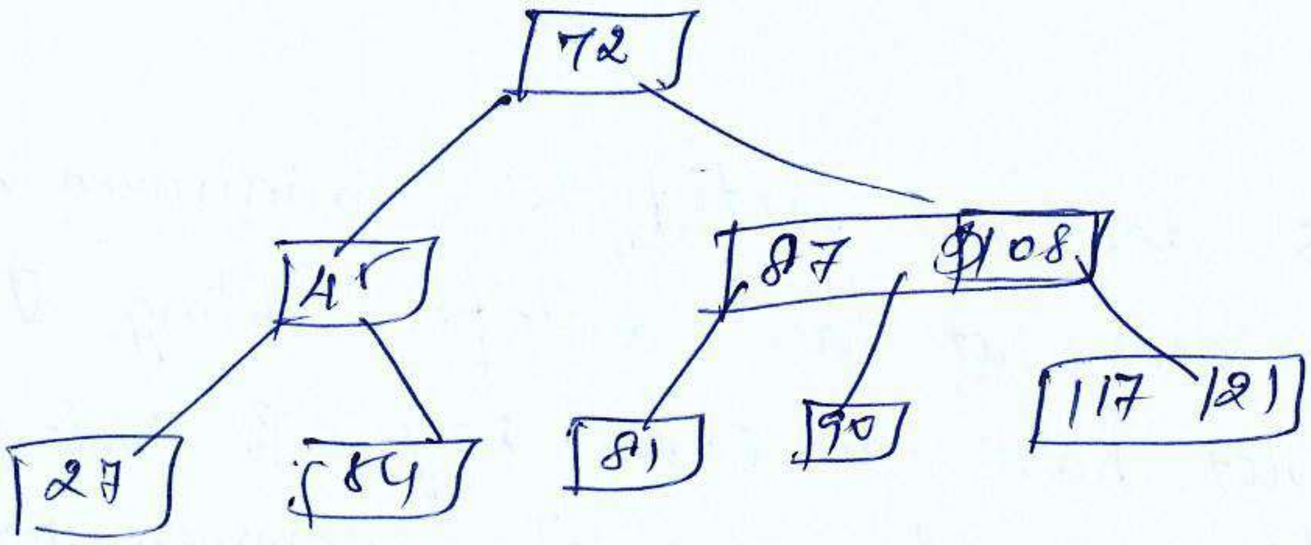


Delete 109

If we delete 109, its parent does not have one children. This is not possible. So combine it with a predecessor or successor sibling. Here, there is no right sibling (successor) and there is a left sibling 87 then combined.



Again this is not possible. Because the root node key is 108. The sorted order is not yet met. Therefore Node 108 is combined with Node 87.



B+ Trees :-

A B+ tree is a variant of a B tree which stores sorted data in a way that allows for efficient insertion, retrieval, and removal of records, each of which is identified by a key.

* While a B tree can store both keys and records in its interior nodes, a B+ tree, in contrast, stores all the records at the leaf level of the tree; only keys are stored in interior nodes.

* The leaf nodes of a B+ tree are often linked to one another in a linked list. This has an added advantage of making the queries simpler and more efficient.

* B+ trees are used to store large amounts of data that cannot be stored in main memory. With B+ trees, the secondary storage (magnetic disk) is used to store the leaf nodes of trees and the internal nodes of trees are stored in main memory.

* B+ trees store data only in the leaf nodes.

* All other nodes (internal nodes) are called index nodes or i-nodes and store index values.

* This allows us to traverse the tree from the root down to the leaf node.

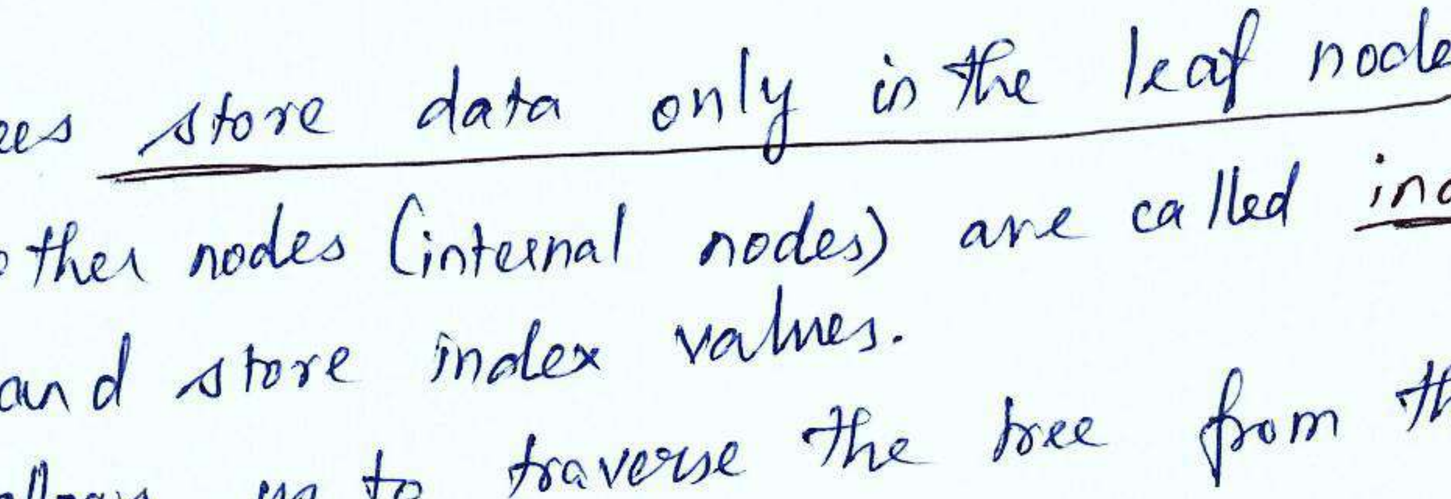


Fig. B+ tree of order 3

Many database systems are implemented using B+ tree structure because of its simplicity.

A B+ tree can be thought of as a multi-level index in which the leaves make up a dense index and the non-leaf nodes make up a sparse index.

Adv:-

1) Records can be fetched in equal no. of disk accesses.
2) It can be used to perform a wide range of queries easily.
3) Height of the tree is less and balanced.
4) Supports both random and sequential access to records.
5) Keys are used for indexing.

Comparison between B Trees and B+ Trees:-

B Tree	B+ Tree
1. Search keys are not repeated	1. Stores redundant search key
2. Data is stored in internal or leaf nodes	2. Data is stored only in leaf nodes
3. Searching takes more time as data may be found in a leaf or non-leaf node	3. Searching data is very easy as the data can be found in leaf nodes only
4. Deletion of non-leaf nodes is very complicated	4. Deletion is very simple because data will be in the leaf node
5. Leaf nodes cannot be stored using linked lists	5. Leaf node data are ordered using sequential linked lists
6. The structure and operations are complicated	6. The structure and operations are simple

Insertion:

A new element is simply added in the leaf node if there is enough space for it. Otherwise, the node is split into two nodes.

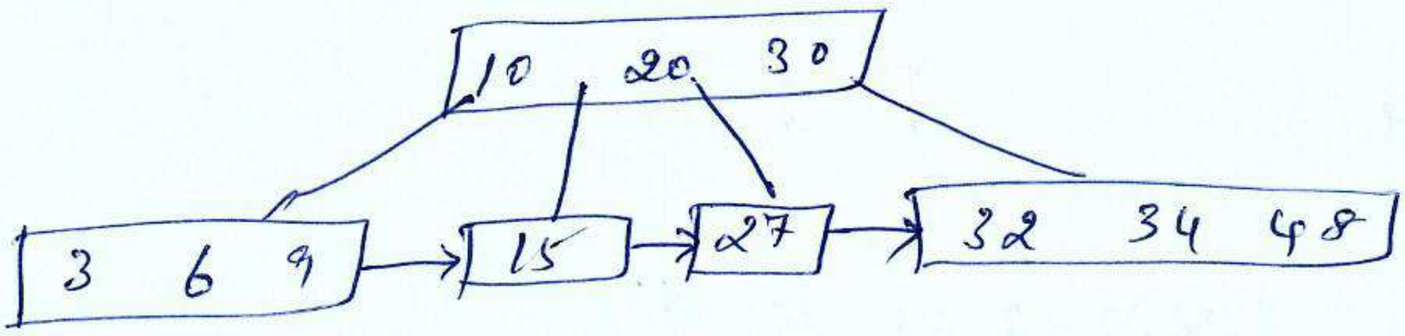
* This calls for adding a new index value in the parent index node. ~~in the parent node~~
* However, adding a new index value in the parent node may cause it, in turn, to split.
* In fact, all the nodes on the path from a leaf to the root may split when a new value is added to a leaf node.
* If the node splits, a new leaf node is created and the tree grows by one level.

Steps:

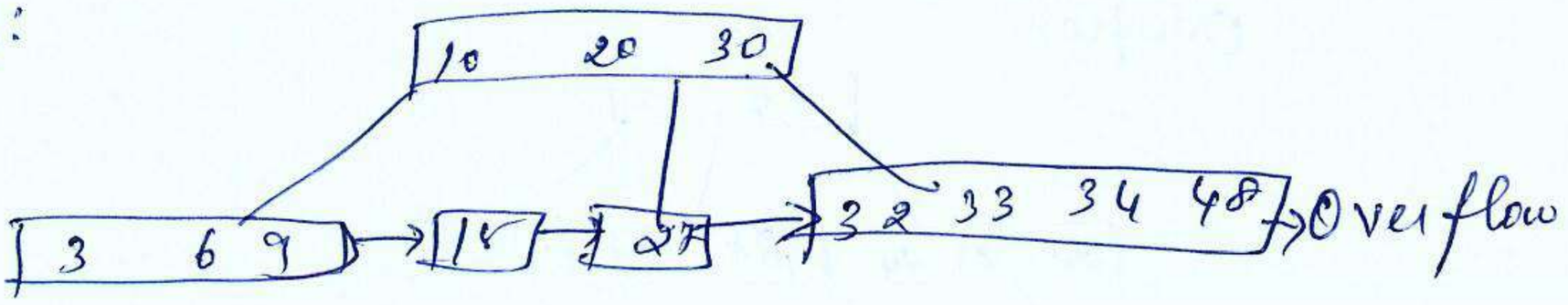
Step 1 : Insert the new node as the leaf node
Step 2 : If the leaf node overflows, split the node and copy the middle element to next index node.
Step 3 : If the index node ovesflows, split that node and move the middle element to next index page.

Example ①:

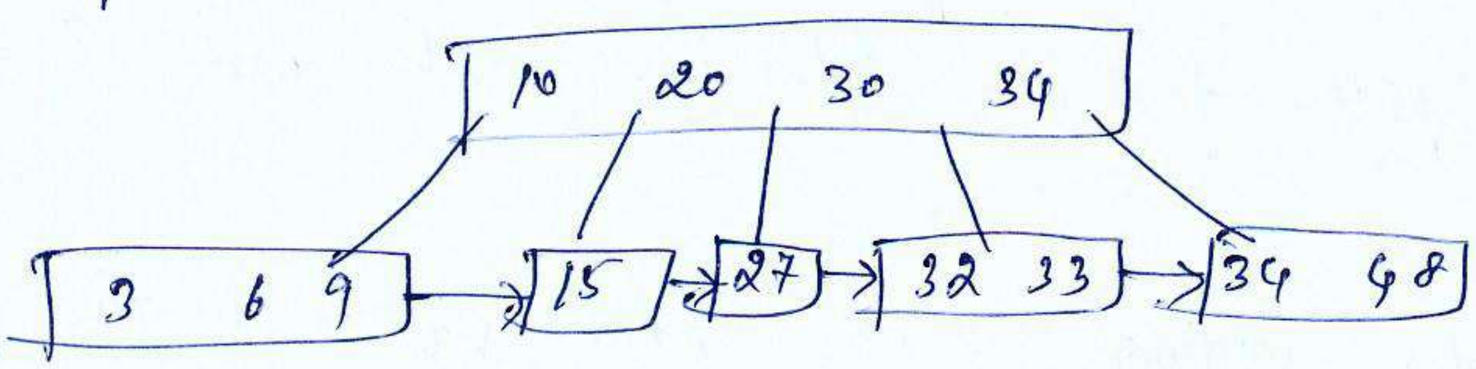
Comider the B+ tree and insert 33. Order of B+ tree is 4.



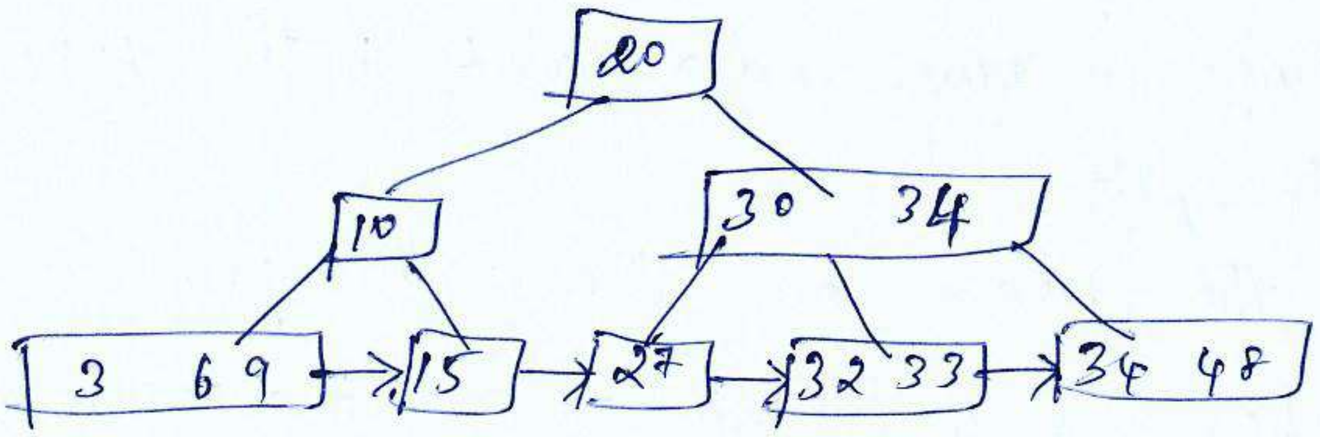
Insert 33:



Split the leaf node

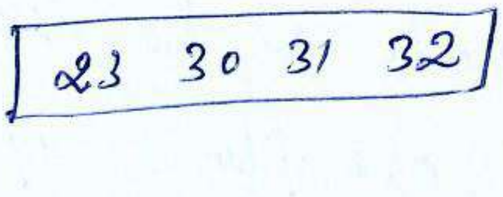


Split the index value

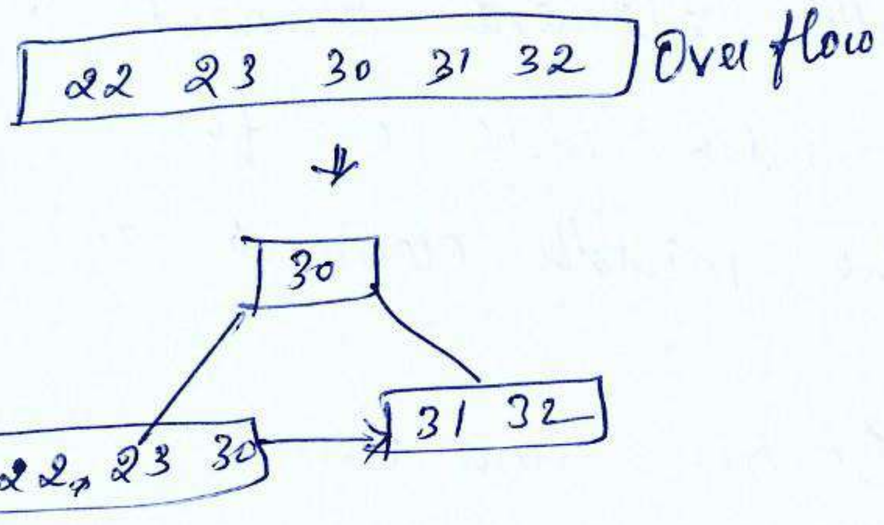


Example②:
Construct a B+tree of order 5 with the following data:
30, 31, 23, 32, 22, 28, 24, 29, 15, 26, 27, 34, 39, 36

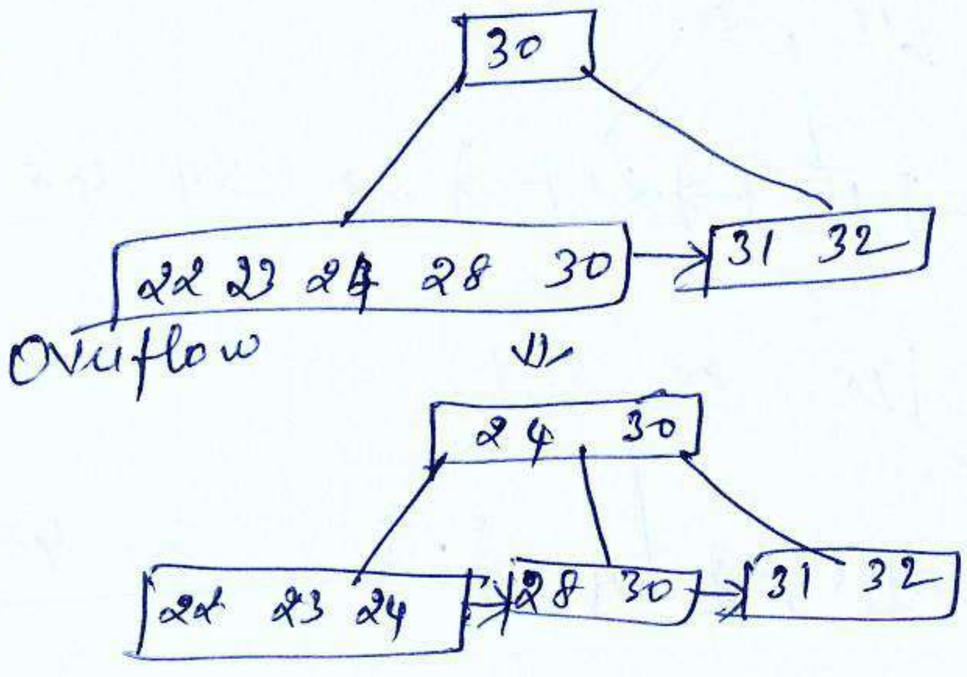
Insert 30, 31, 23, 32.



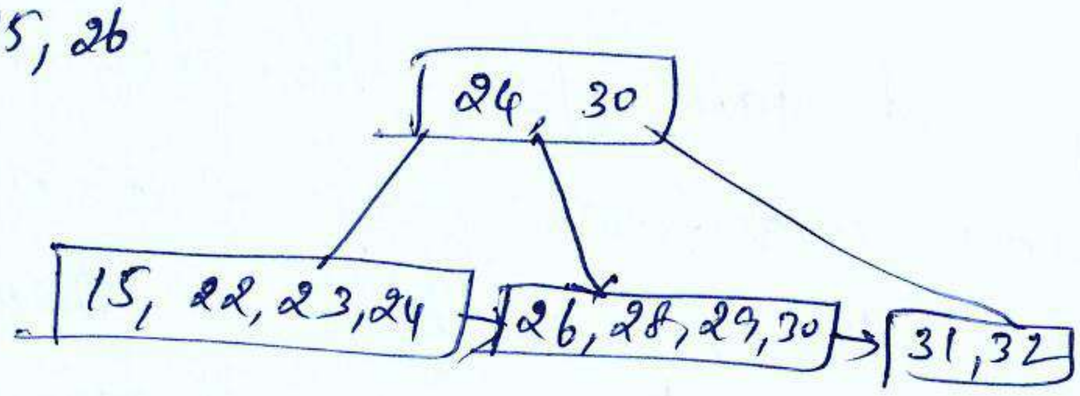
Insert 22



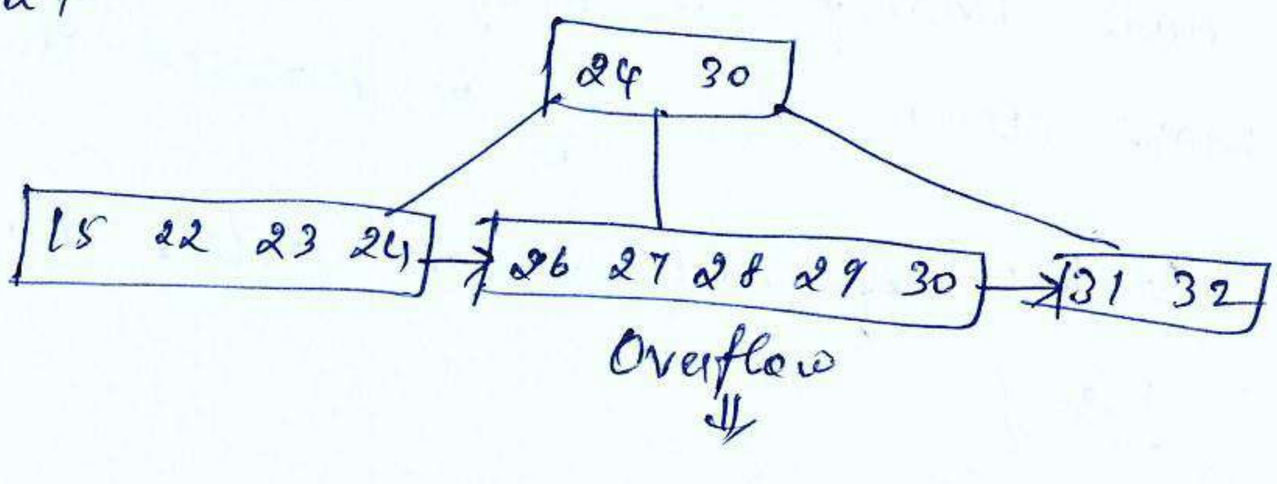
Insert 28, 24

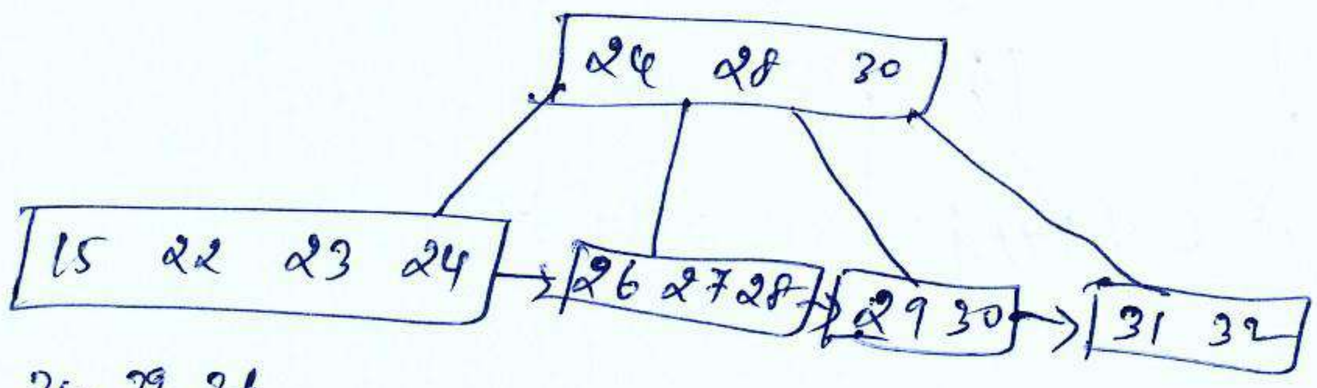


Insert 29, 15, 26

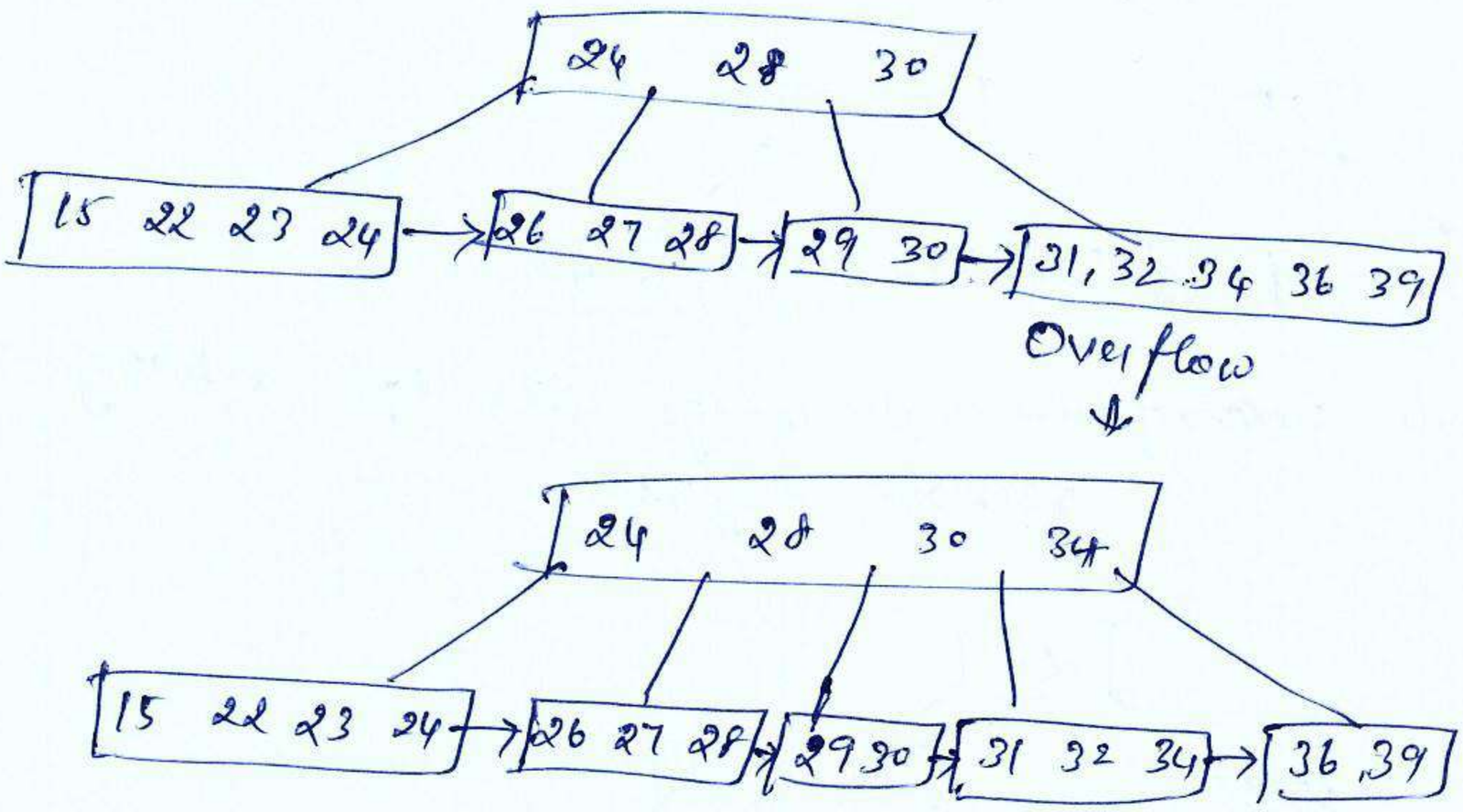


Insert 27





Insert 34, 39, 36



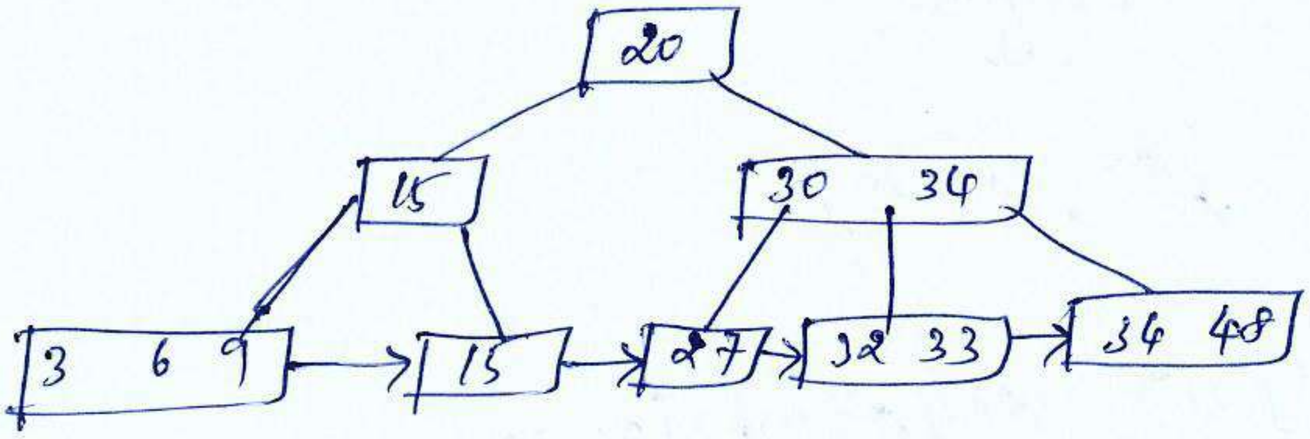
Deletion:

As in B trees, deletion is done from a leaf node. If deleting a data leaves that node empty, then the neighboring nodes are examined and merged with the underfull node. This leads to shrinking of the tree by one level.

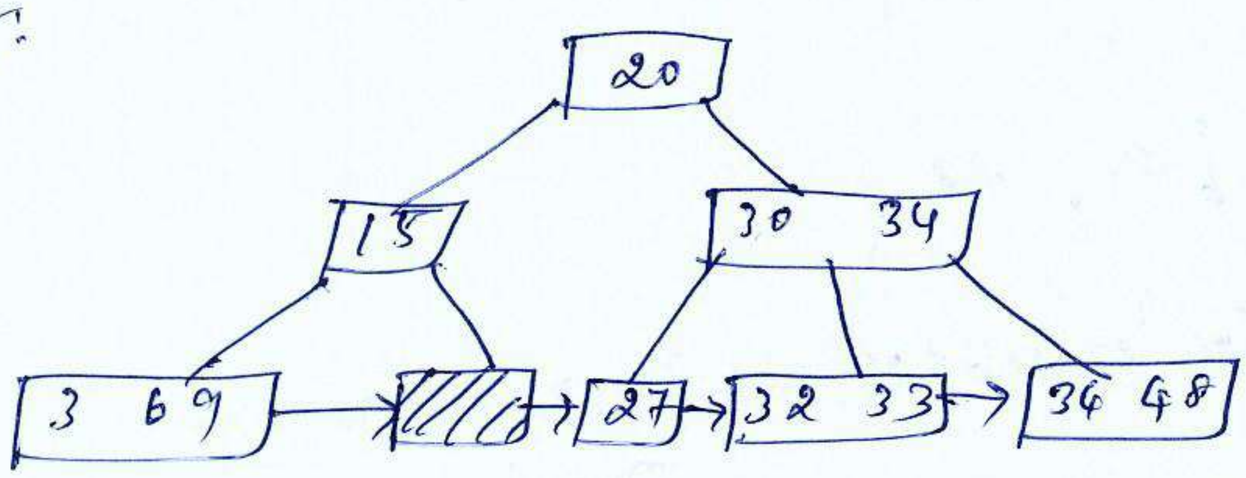
Steps:

Step 1: Delete the key and data from the leaves
Step 2: If the leaf node underflows, merge that node with the sibling and delete the key in between them.
Step 3: If the index node underflows, merge that node with the sibling and move down the key in between them.

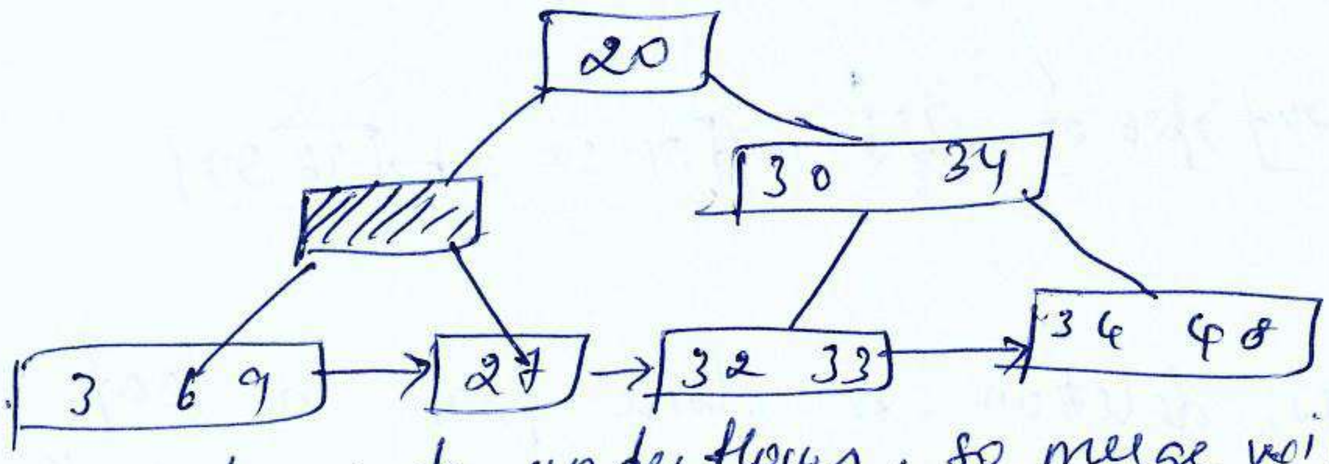
Example ③: Consider the B+ tree of order 4 and delete node 15.



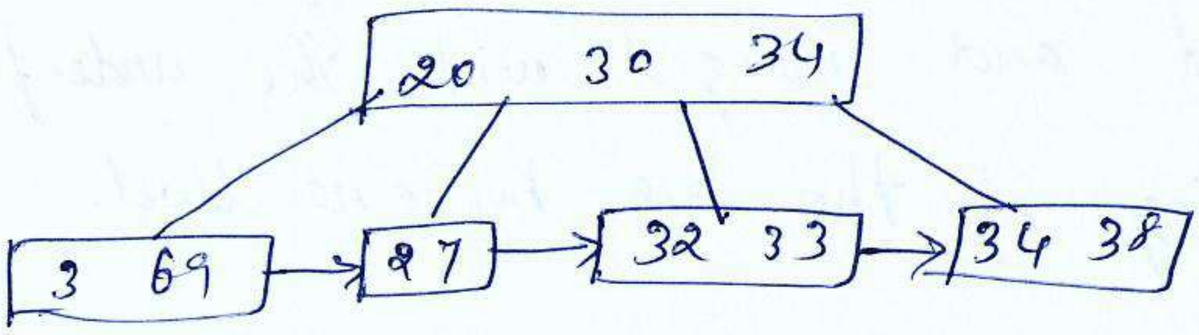
Delete 15:



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



Now index node underflows, so merge with sibling and delete the node



Heaps:-

Heap is a complete binary tree or almost a complete binary tree in which every parent node be either greater or lesser than its child nodes.

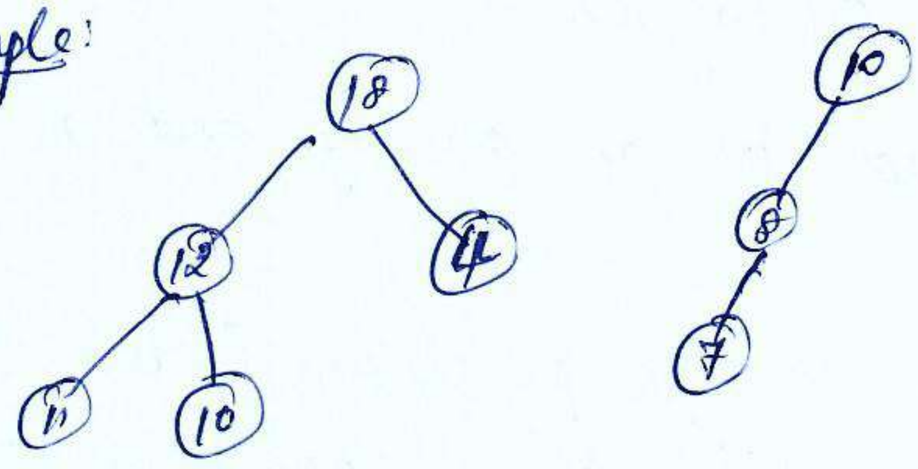
Types of Heap:

② types — min heap
         — max heap

Max Heap:

Max heap is a tree in which the value of each node is greater than or equal to the value of its children node.

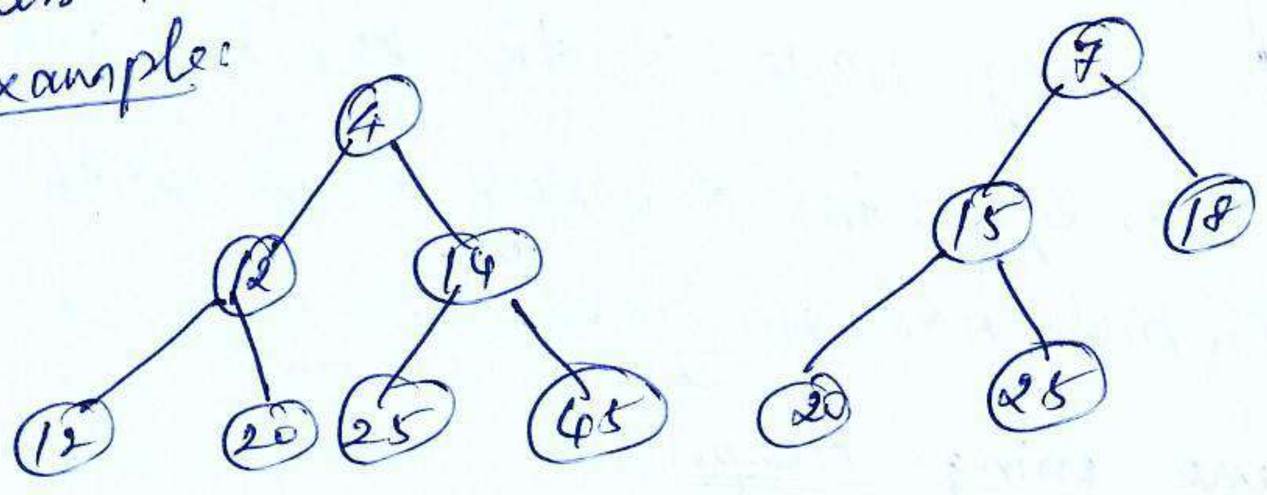
Example:



Min Heap:

Min heap is a tree in which the value of each node is less than or equal to the value of its children node.

Example:



Application:

1) Sorting an array using heapsort algorithm.
2) Implementing priority queues.
3) Heap implemented priority queues are used in graph algs like Prim's and Dijkstra's alg.
4) Order Statistics - The heap can be used to efficiently find the kth smallest (or largest) element in an array.

Two properties of heap:

* It should be a complete binary tree
* It should satisfy parental property.

1) Structure Property:

* A heap should be a complete binary tree which is completely filled binary tree exception with the possible of the bottom level, which is filled from left to right.
* A complete binary tree of height h has between 2^h and $2^{h+1}-1$ nodes. This implies that the height of a complete binary tree is O(log N).
* It can be represented in an array and no pointers are necessary.
* For any element in array position i, the left child is in position 2i, the right child is in position 2i+1, and the parent is in position i/2.

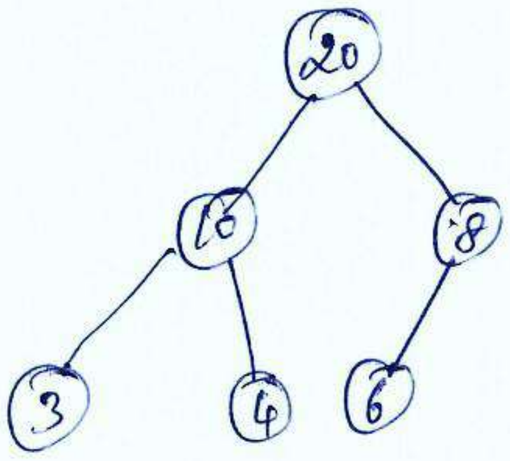
2) Heap-order Property:

In a heap for every node X, the key in the parent of X is smaller than (or equal to) the key in X with the exception of the root (which has no parent).

Realizing Priority Queue ming Heaps:-

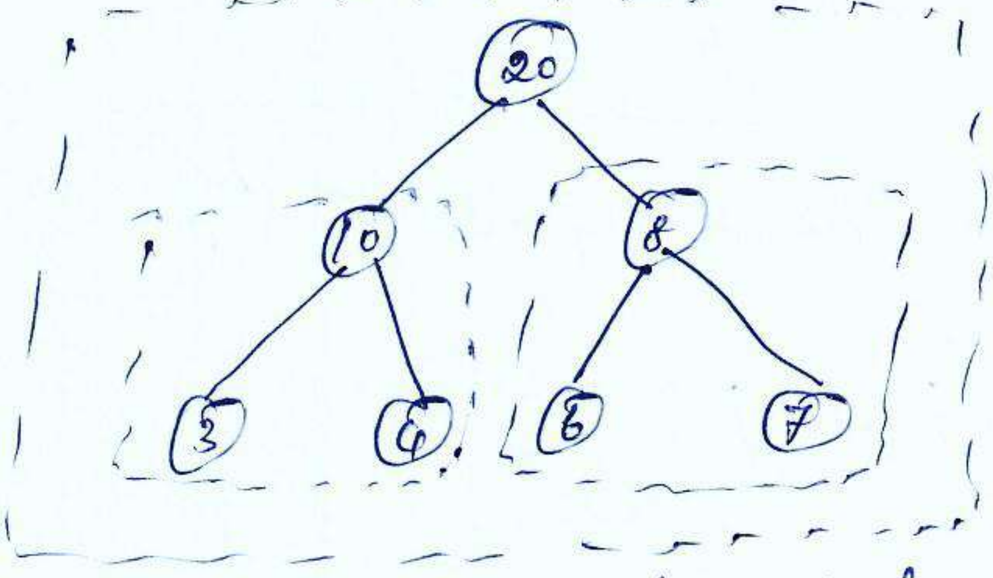
There are some important properties that should be followed by heap.

1). There should be either "complete binary tree" or almost complete binary tree.
2). The root of a heap always contains its largest element



This is heap because 20 is largest among all node's values.

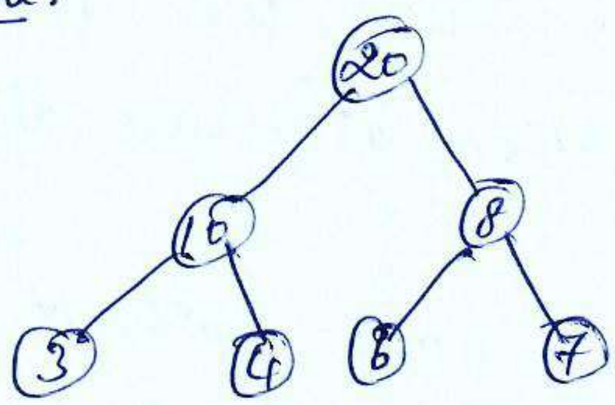
3). Each subtree is a heap is also a heap.



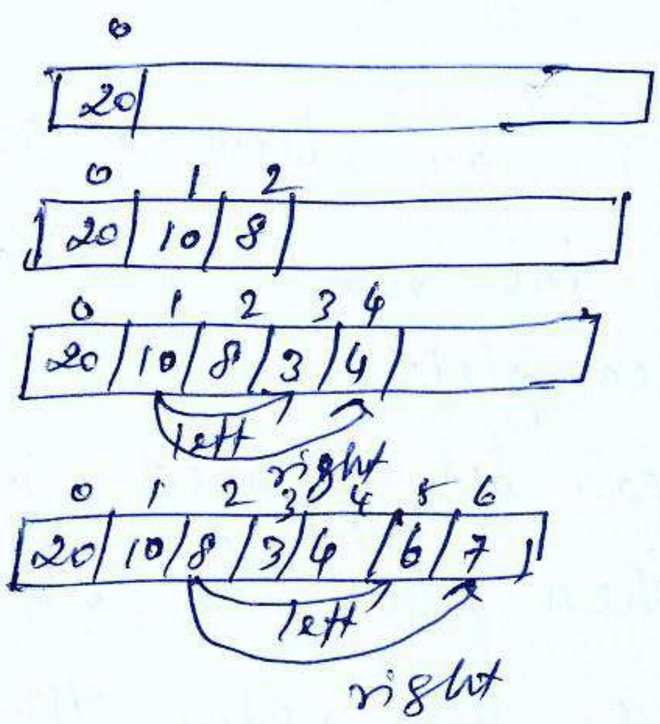
4). Heap can be implemented as array. by recording its elements in the top-down and left-to-right fashion.

For array A[size]
- → Root node is stored at A[0]
- → Left child is at 2i+1 position in array A
- → Right child is at 2i+2 " " " "
- → Parent node " " i " " " "

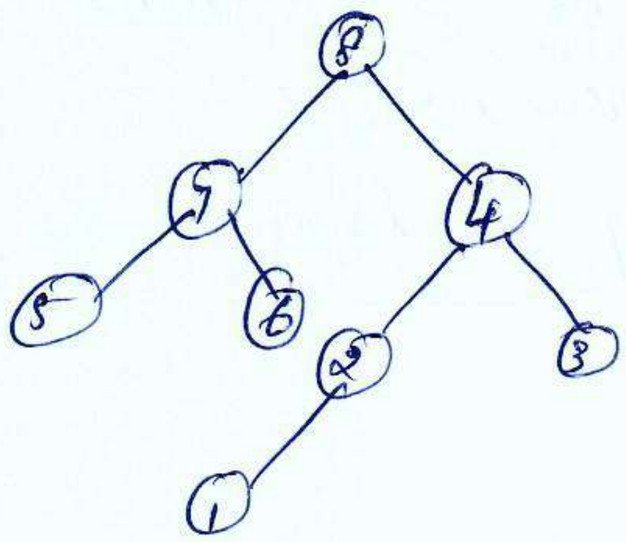
Example:

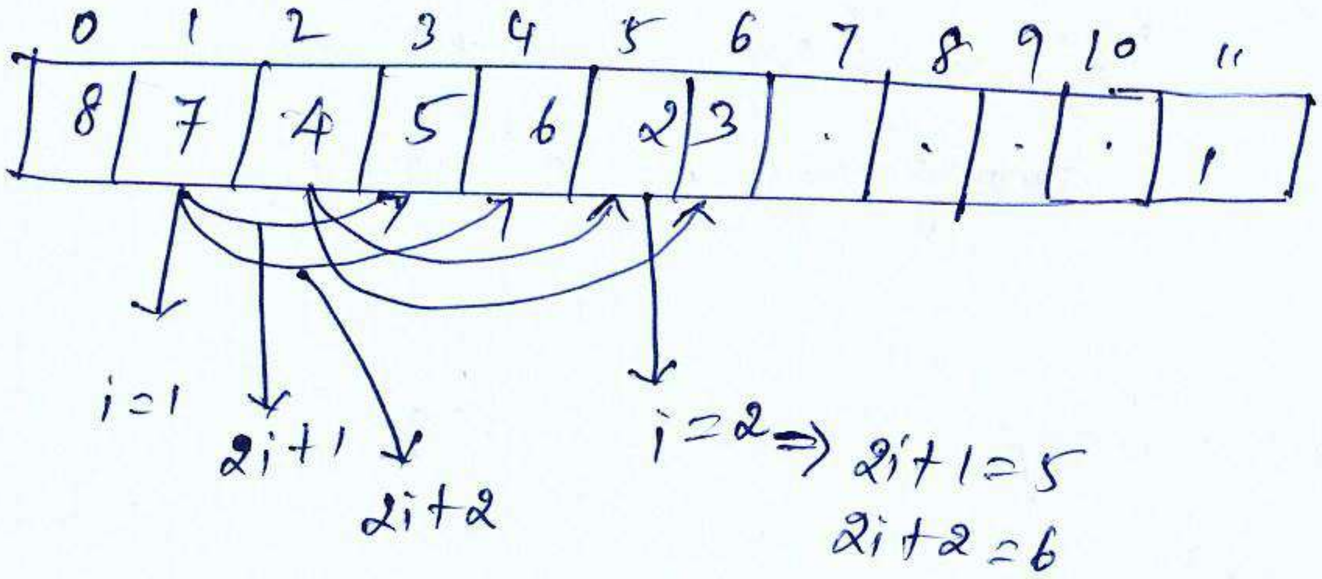


⇒

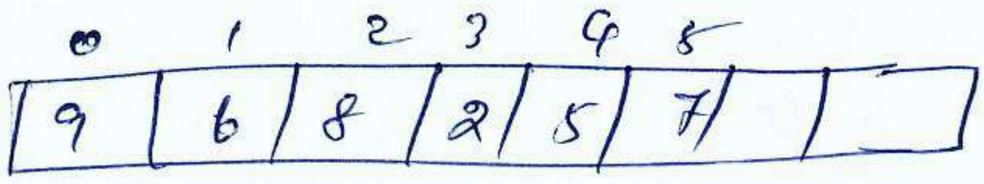


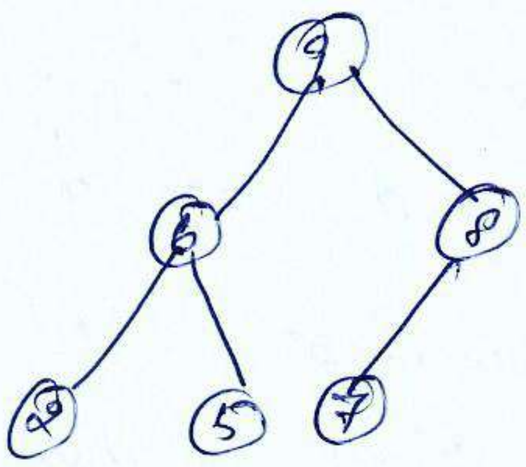
For a given heap, give the array representation.





Draw the tree for a given heap represented by an array.





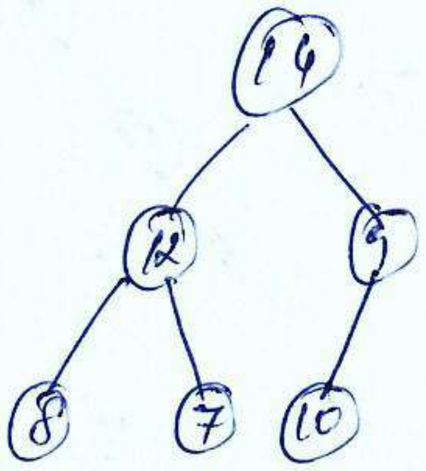
Operations of Binary Heap:

1) Insertion:

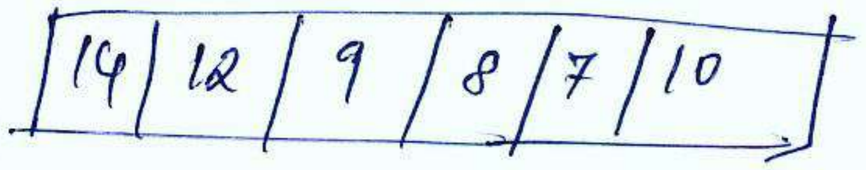
* To insert an element x into the heap, we create a one node in the next available location, otherwise the tree will not be complete.
* If x can be placed in the new node without violating heap order, then place the element x there itself
* Otherwise, we swap the element that is in the new node's parent node into the new node, thus bubbling the one node up towards the root. We continue this process until x can be placed in the new node.
* This general strategy is known as <u>percolate up.</u>

Example ①:

Insert 18 into the following heap

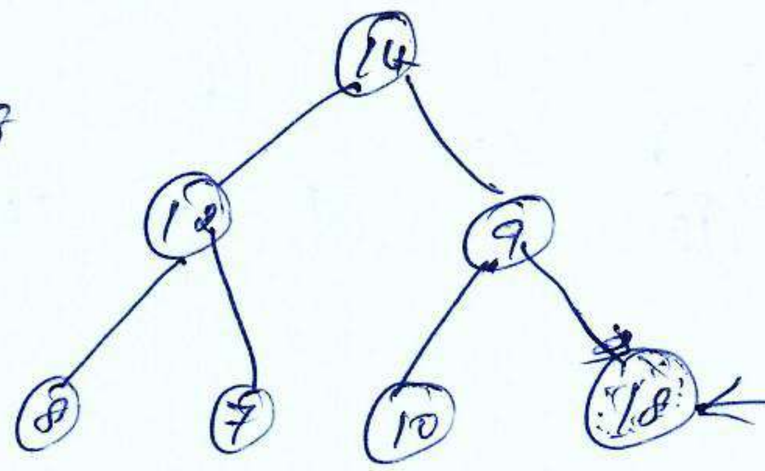


⇒



Step 1:

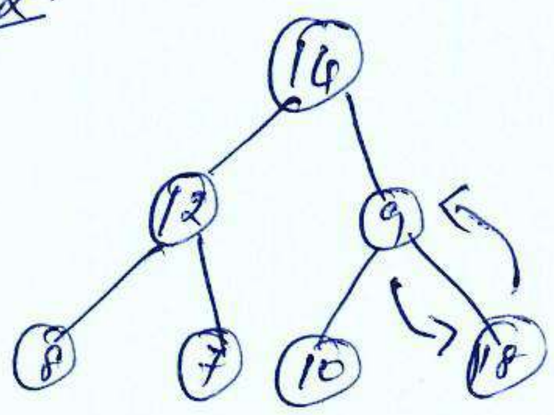
Insert 18



← Attach this node as a last node

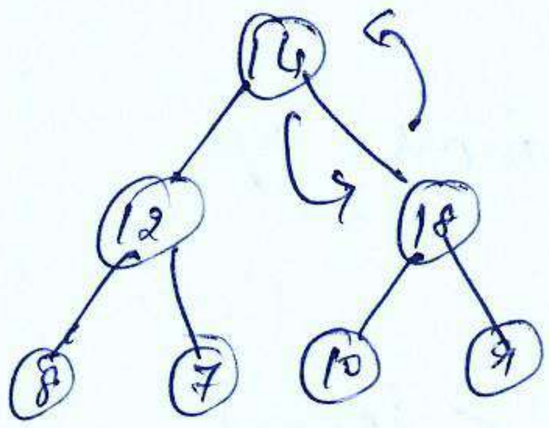
We always fill the heap from left to right order.

Step 2:



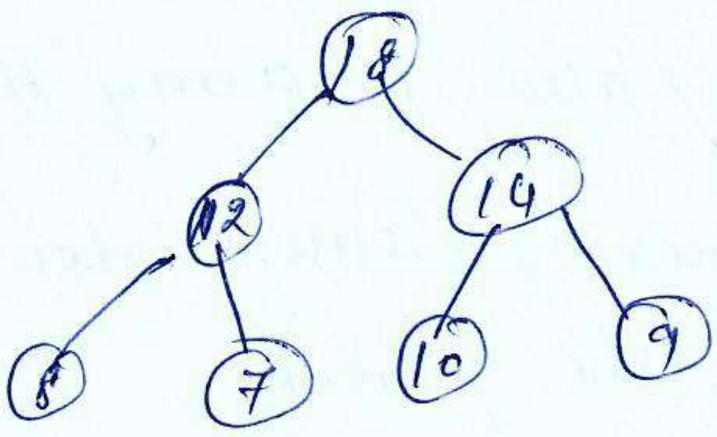
Parent dominance property is violated (9 < 18), we swap two values

Step 3:

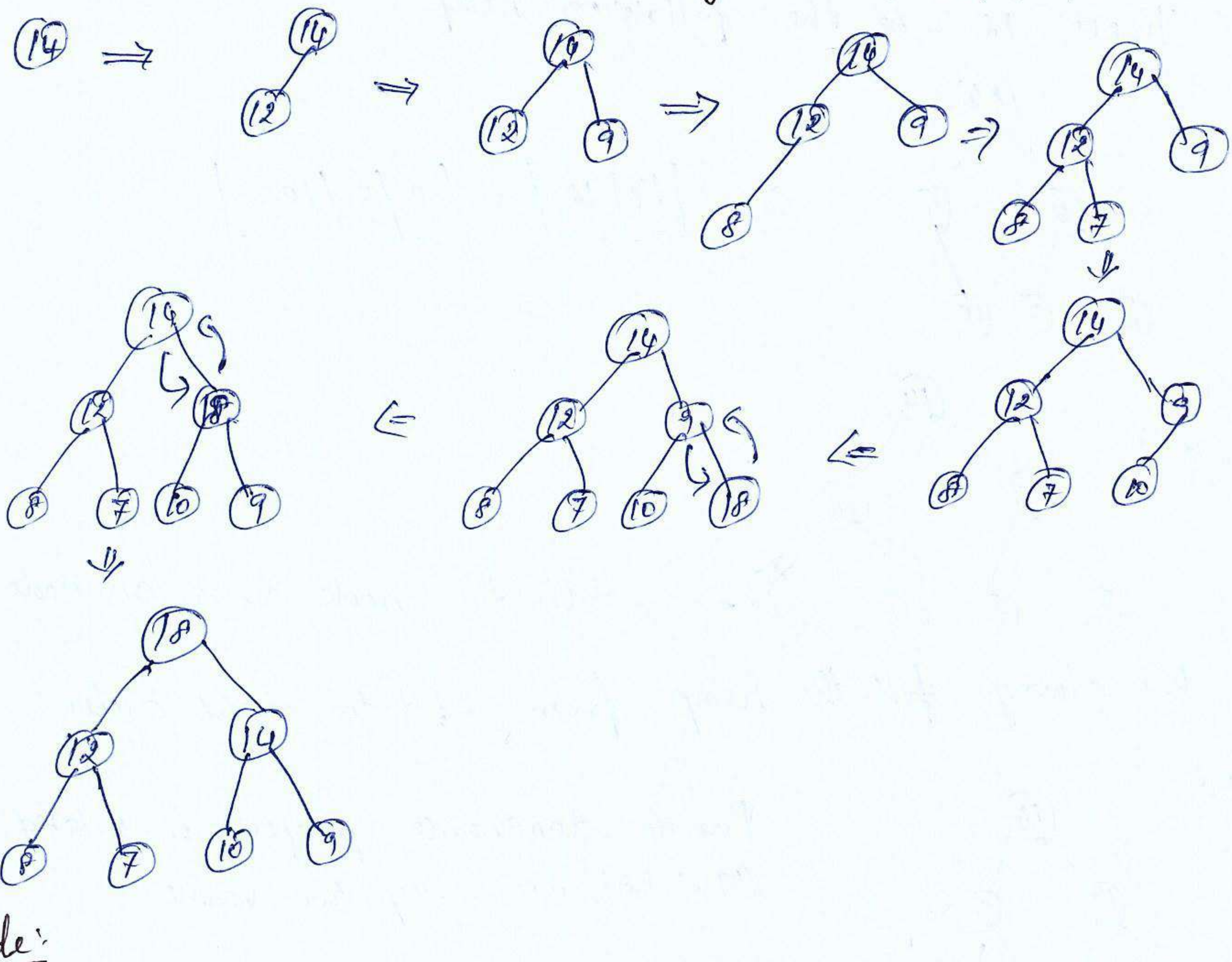


Parent dominance property is violated (14 < 18), we swap two values.

Step 4:



Example (2): Construct a heap using 14, 12, 9, 8, 7, 10, 18 elements.



Code:

```
void heapinsert (int item)
{
   int temp;
   //temp node starts at leaf and moves up
   temp = ++size;
   while (temp!= 1 && heap [temp/2] < item)
   {
     // the element cannot be placed in array H
   H[temp] = H [temp/2]; // moving element down
   temp = temp/2;    //finding the parent
   }
   H[temp] = item; // placing the element in its position.
}
```

2) DeleteMin :-

DeleteMin operation is deleting the minimum element from the heap.

* The minimum element is found in the root.
* After the deletion of minimum element, a hole is created at the root. [i.e, empty root]
* The last element 'x' in the heap to move some where in the heap.
* If x can be placed in the hole without violating heap-order property.
* Otherwise we slide the smaller of the hole's children into the hole, thus pushing the hole down one level.
* This process is repeated, until x can be placed in the hole.
* This general strategy is known as percolate down.

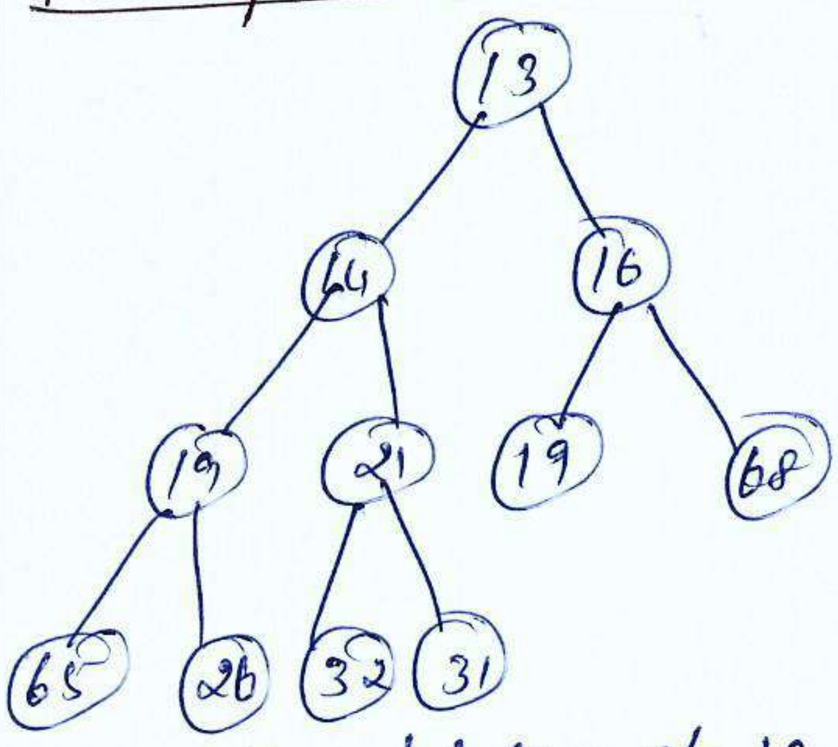
Routine:

```
int DeleteMin(PriorityQueue H)
{
    int i, child;
    int minelement, lastelement;
    if (IsEmpty(H))
    {
        printf("\n Priority Queue is Empty");
        return H->element[0];
    }
minelement = H->element[i];
lastelement = H->element[H->size--];
```

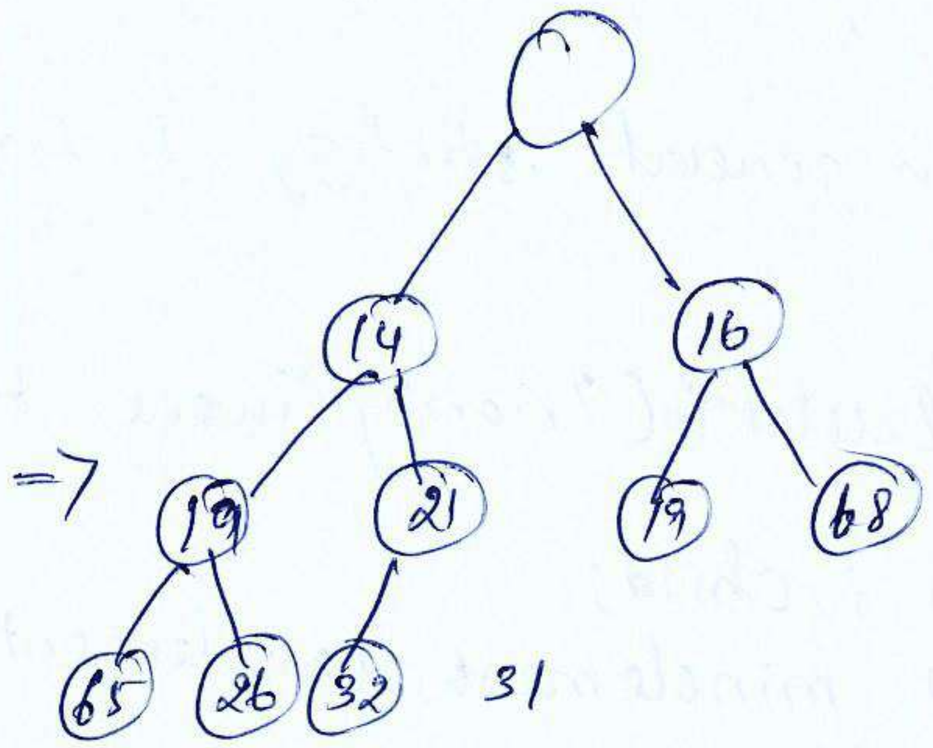
```
for (i=1; i*2 <= H->size; i=child)
{
    // Find smaller child
    child = i*2;
    if((child != H->size && H->elements[child+1]) <
                            H->elements[child])

        child++;
    // Percolate one level
    if (lastelement > H->element[child])
        H->element[i] = H->element[child];
    else
        break;
    H->element[i] = lastelement;
    return minelement;
}
```

Example (3):

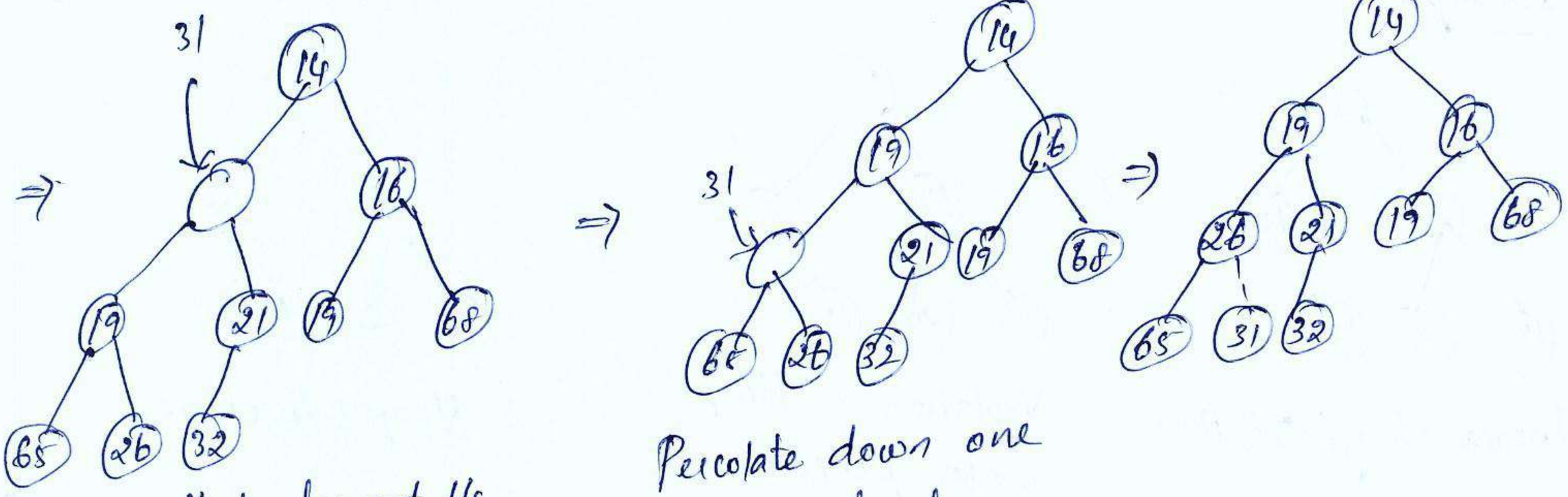


After deletion of 13 create a hole at root



31

The last element 31 move some where in the heap.



The smallest element 14 is placed at root

Percolate down one level.

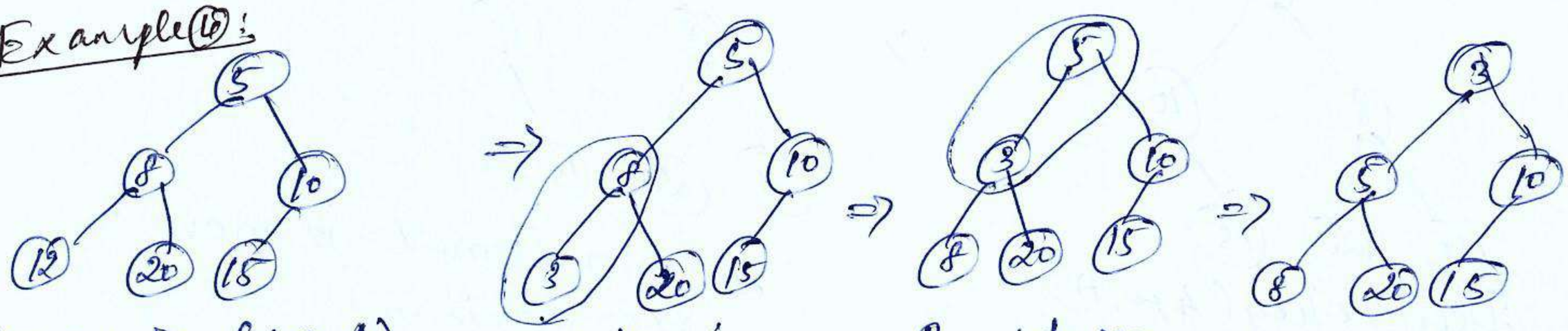
2) Decrease key :-

* The decrease key (P, Δ, H) operation lowers the value of the key at position P by a positive amount Δ.
* It violates the heap-order property.
* It must be fixed by percolate up.

Benefit:

* This operation could be useful to system administrators.
* This can make their pgms run with highest priority.

Example (16):



Decrease key (4, 9, Δ)

Violation in heap order.

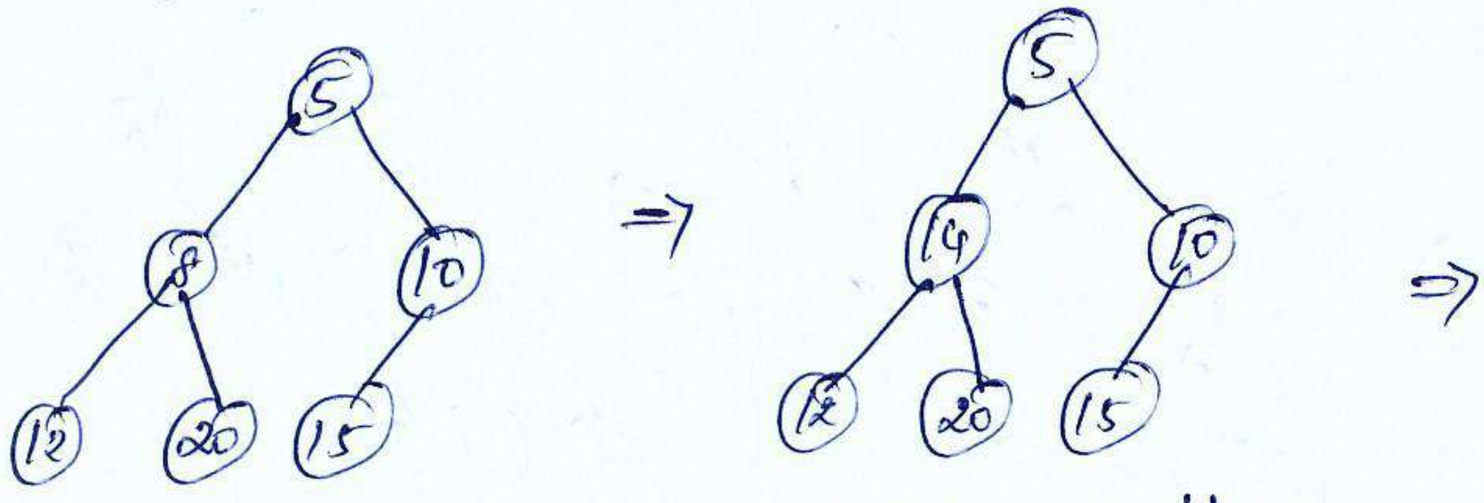
Percolate up

4) Increase key :-

* The increase key (P, Δ, H) increases the value of the key at position P by a positive amount Δ.
* This can be done with a percolate down.

Benefit: Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

Example (4):



Increasekey(2,6,H) ⇒ Violation in Heap order, percolate down. ⇒ Heap structure.

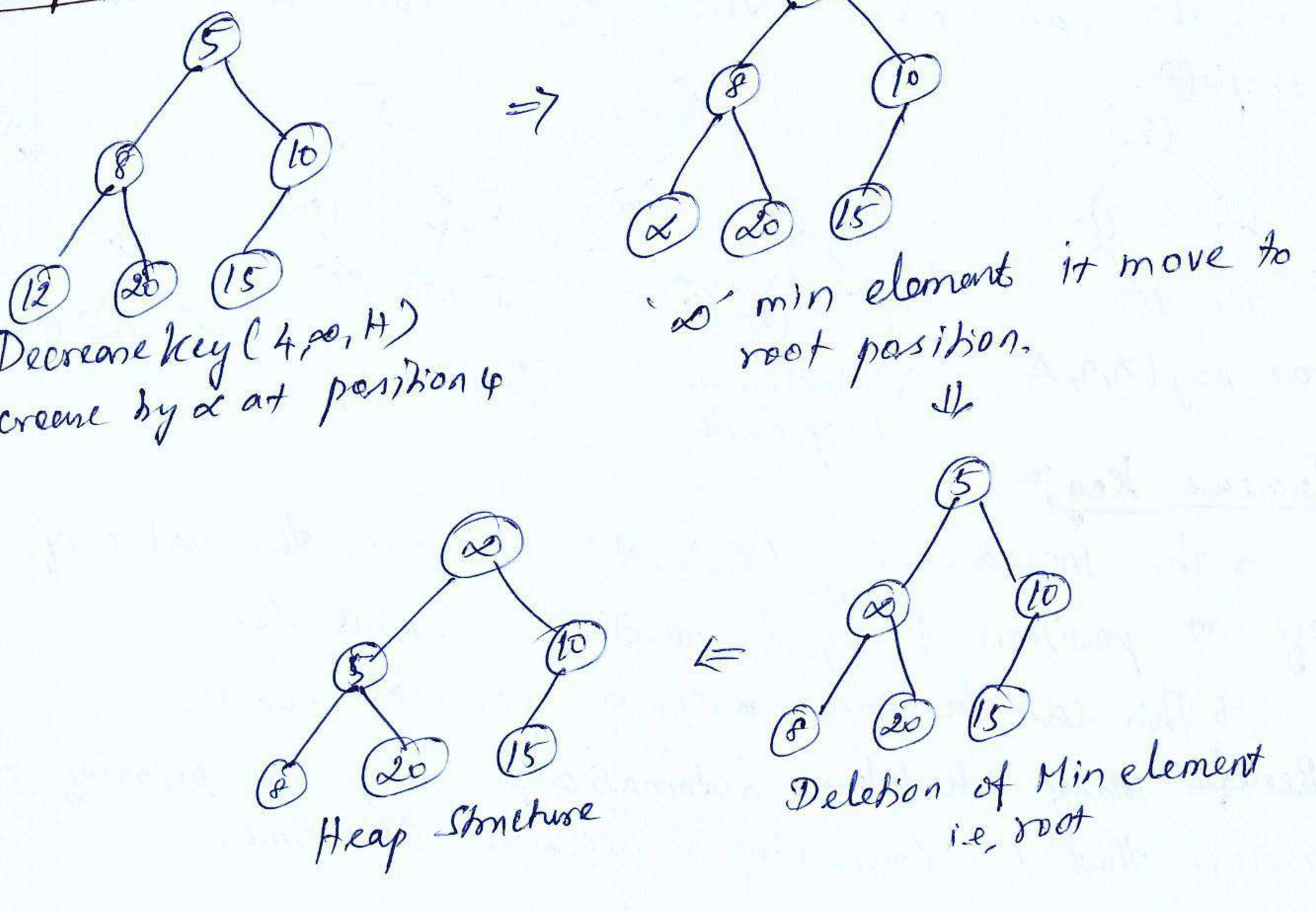
5) Delete:-

The delete (P, H) operation removes the node at position P from the heap. It involves 2 operations — Decreasekey(P, ∞, H), DeleteMin

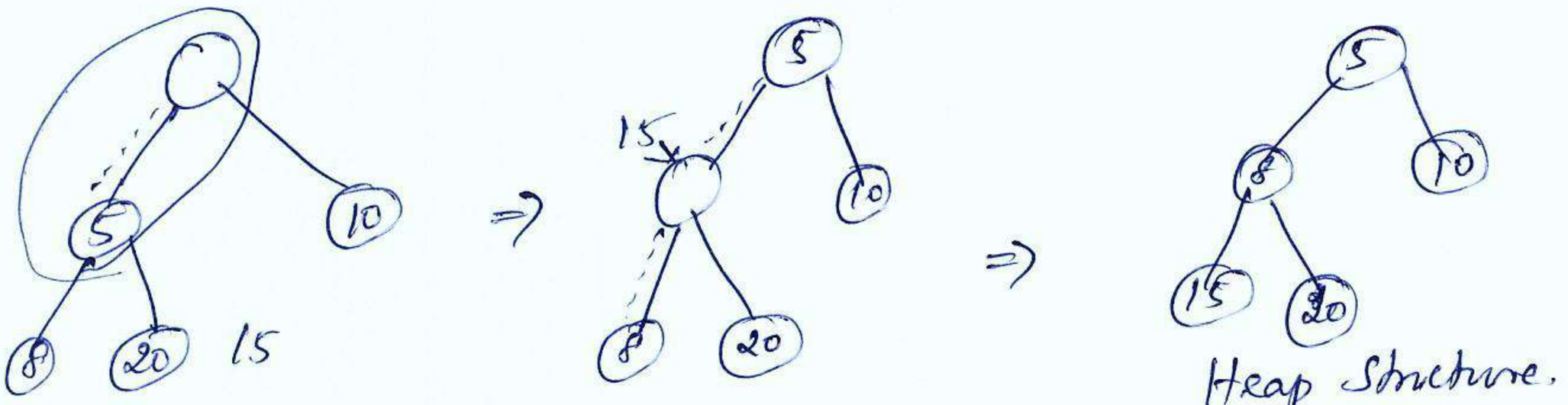
Benefit:

When a process is terminated by a user (instead of finishing normally), it must be removed from the priority queue.

Example (5):



Decreasekey(4, ∞, H) decreese by ∞ at position 4 ⇒ '∞' min element it move to root position. ⇓ Deletion of Min element i.e. root ⇐ Heap Structure



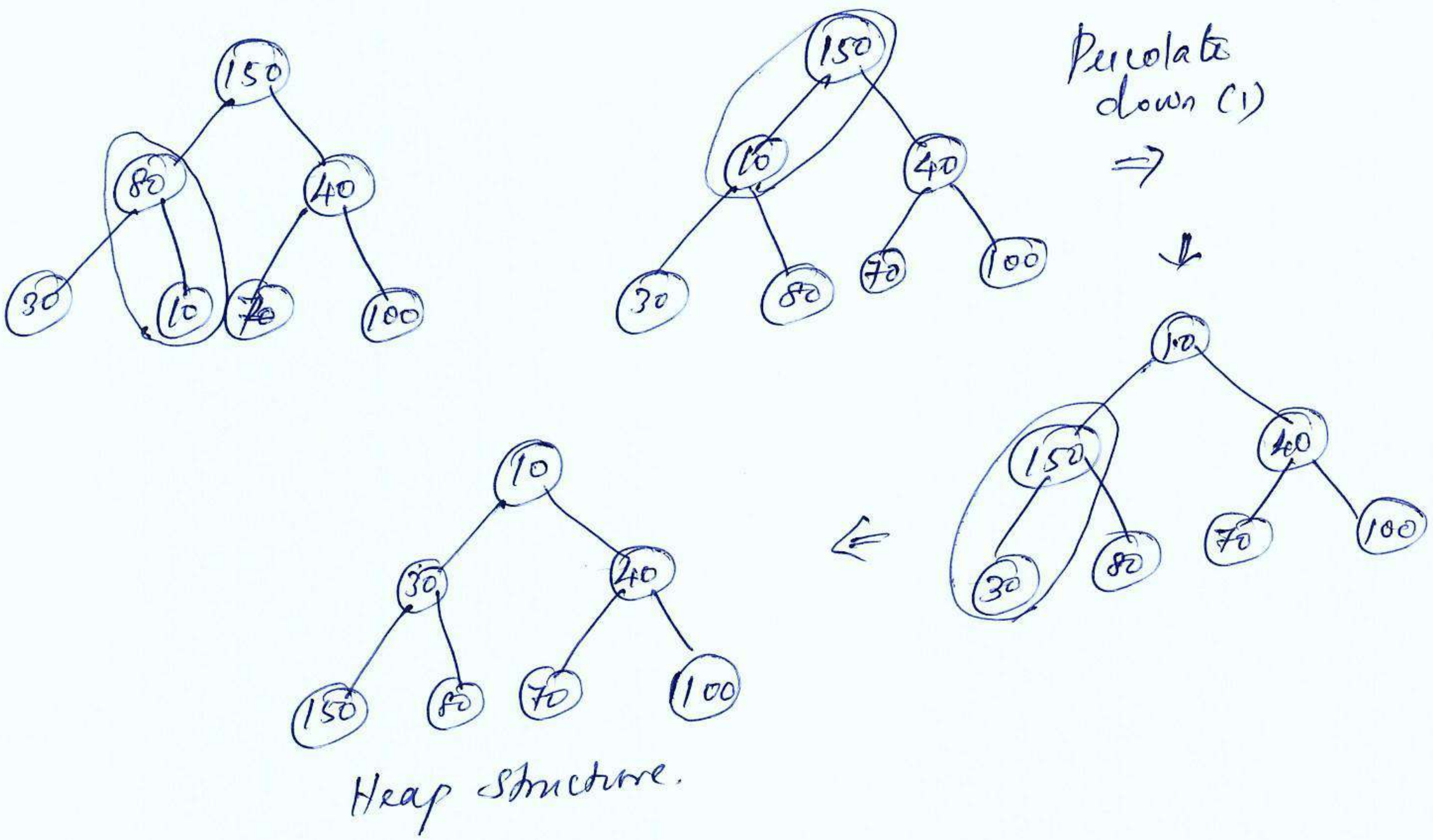
Heap Structure.

6) Build Heap:

* It takes as i/p N keys and places them into an empty heap.
* This can be done with N successive inserts.
* Each insert will take O(1) average and O(log N) worst case time.

Another way to build heap:

* To place N keys into the tree any order maintaining the structure property.
* Percolate down from node i to create heap-ordered tree.



Heap Structure.