Unit 3

Non Linear Data Structures - Trees

3.1 PRELIMINARIES :

TREE : A tree is a finite set of one or more nodes such that there is a specially designated node called the Root, and zero or more non empty sub trees T_1, $T_{2 \ldots\ldots}T_k$, each of whose roots are connected by a directed edge from Root R.

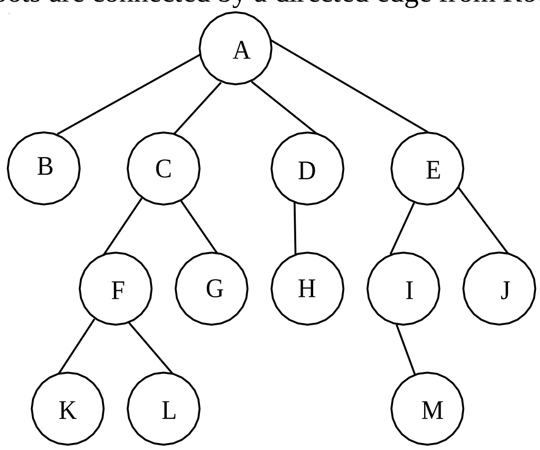


Fig. 3.1.1 Tree

ROOT : A node which doesn't have a parent. In the above tree. The Root is A.

NODE : Item of Information.

LEAF : A node which doesn't have children is called leaf or Terminal node. Here B, K, L, G, H, M, J are leafs.

SIBLINGS : Children of the same parents are said to be siblings, Here B, C, D, E are siblings, F, G are siblings. Similarly I, J & K, L are siblings.

PATH : A path from node n_1 to n_k is defined as a sequence of nodes n_1, n_2,$n_{3 \ldots\ldots}n_k$ such that n_i is the parent of n_{i+1}. for $1 \le i < k$. There is exactly only one path from each node to root.

In fig 3.1.1 path from A to L is A, C, F, L. where A is the parent for C, C is the parent of F and F is the parent of L.

LENGTH : The length is defined as the number of edges on the path.

In fig 3.1.1 the length for the path A to L is 3.

DEGREE : The number of subtrees of a node is called its degree.

In fig 3.1.1

Degree of A is 4

Degree of C is 2

Degree of D is 1

Degree of H is 0.

* The degree of the tree is the maximum degree of any node in the tree.

  In fig 3.1.1 the degree of the tree is 4.

LEVEL : The level of a node is defined by initially letting the root be at level one, if a node is at level L then its children are at level L + 1.

Level of A is 1.

Level of B, C, D, is 2.

Level of F, G, H, I, J is 3

Level of K, L, M is 4.

DEPTH : For any node n, the depth of n is the length of the unique path from root to n.

The depth of the root is zero.

In fig 3.1.1 Depth of node F is 2.

Depth of node L is 3.

HEIGHT : For any node n, the height of the node n is the length of the longest path from n to the leaf.

The height of the leaf is zero

In fig 3.1.1 Height of node F is 1.

Height of L is 0.

Note :	The height of the tree is equal to the height of the root Depth of the tree is equal to the height of the tree.

3.2 TREE TRAVERSALS

Traversing means visiting each node only once. Tree traversal is a method for visiting all the nodes in the tree exactly once. There are three types of tree traversal techniques, namely

1. Inorder Traversal

2. Preorder Traversal

3. Postorder Traversal

Inorder Traversal

The inorder traversal of a binary tree is performed as

- *	Traverse the left subtree in inorder
- *	Visit the root
- *	Traverse the right subtree in inorder.

Example :

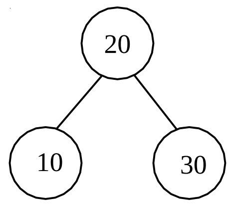


Fig. 3.2.1 Inorder 10, 20, 30

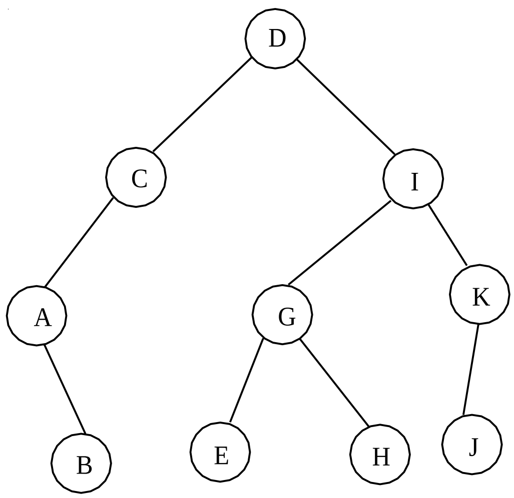


Fig. 3.2.2 A B C D E G H I J K

The inorder traversal of the binary tree for an arithmetic expression gives the expression in an infix form.

RECURSIVE ROUTINE FOR INORDER TRAVERSAL

```
void Inorder (Tree T)
{
    if (T ! = NULL)
    {
        Inorder (T → left);
        printElement (T → Element);
        Inorder (T → right);
    }
}
```

Preorder Traversal

The preorder traversal of a binary tree is performed as follows,

* Visit the root
* Traverse the left subtree in preorder
* Traverse the right subtree in preorder.

Example 1 :

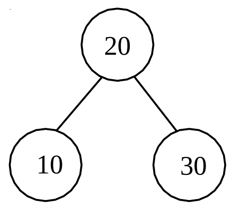


Fig. 3.2.3 Preorder : 20, 10, 30

Example 2 :

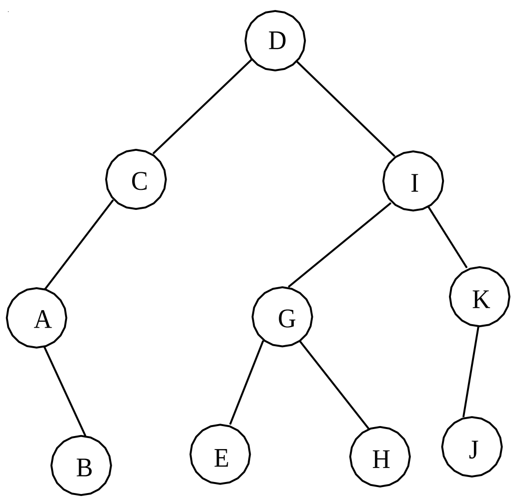


Fig. 3.3.4 Preorder D C A B I G E H K J

the preorder traversal of the binary tree for the given expression gives in prefix form.

Recursive Routine For Preorder Traversal

```
void Preorder (Tree T)
{
        if (T ! = NULL)
        {
                printElement (T → Element);
                Preorder (T →  left);
                Preorder (T → right);
        }
```

Postorder Traversal

The postorder traversal of a binary tree is performed by the following steps.

- Traverse the left subtree in postorder.
- Traverse the right subtree in postorder.
- Visit the root.

Example : 1

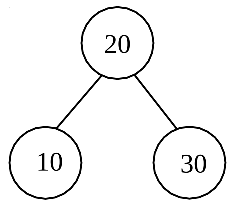


Fig. 3.2.5 Postorder : - 10, 30, 20

Example : 2

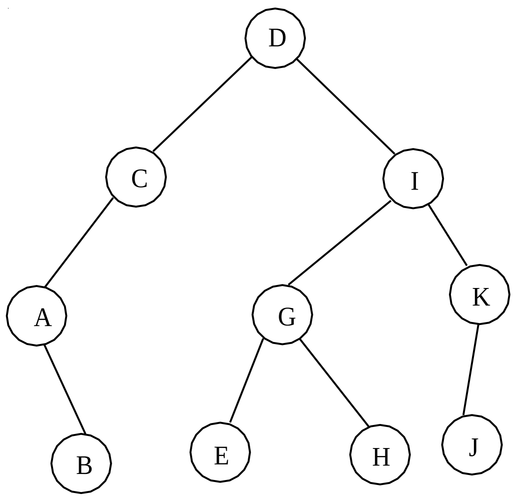


Fig. 3.2.6 Post order : - B A C E H G J K I D

The postorder traversal of the binary tree for the given expression gives in postfix form.

Recursive Routine For Postorder Traversal

```
void Postorder (Tree T)
{
    if (T ! = NULL)
    {
        Postorder (T → Left);
        Postorder (T → Right);
        PrintElement (T → Element);
    }
```

Example : -

Traverse the given tree using inorder, preorder and postorder traversals.

(1)

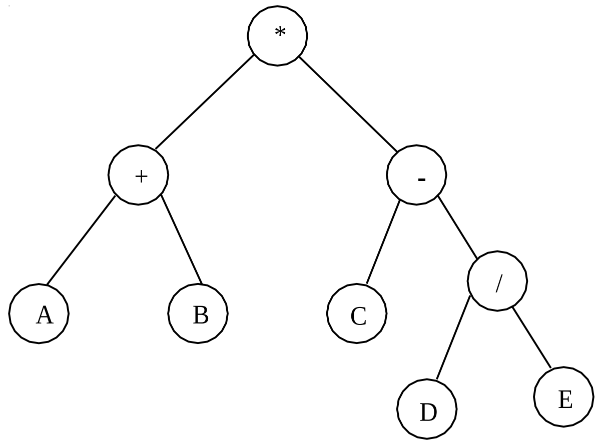


Fig. 3.2.7

Inorder : A + B * C - D / E

Preorder : * + A B - C / D E

Postorder : A B + C D E / -*

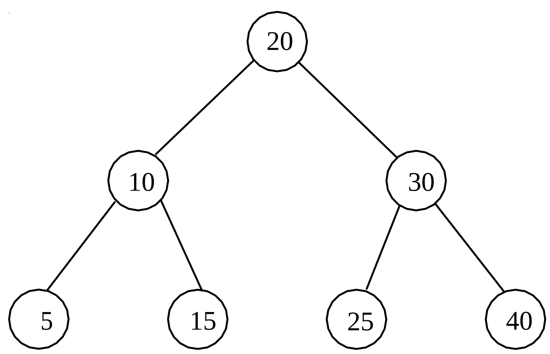


Fig. 3.2.8

Inorder	: 5	10	15	20	25	30	40
Preorder	: 20	10	5	15	30	25	40
Postorder	: 5	15	10	25	40	30	20

3.3 BINARY TREE

Definition :-

Binary Tree is a tree in which no node can have more than two children.

Maximum number of nodes at level i of a binary tree is 2^{i+1}.

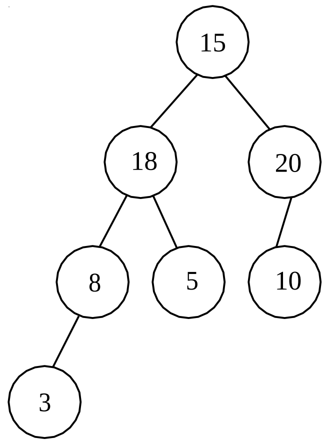
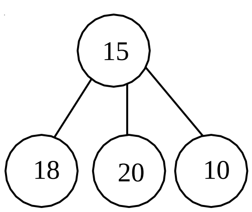
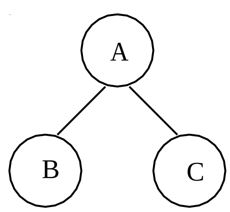


Fig. 3.3.1 Binary Tree

Binary Tree Node Declarations

```
Struct TreeNode
{
        int Element;
        Struct TreeNode *Left ;
        Struct TreeNode *Right;
};
```

COMPARISON BETWEEN GENERAL TREE & BINARY TREE

General Tree	Binary Tree
* General Tree has any number of children.  	* A Binary Tree has not more than two children. 

Full Binary Tree :-

A full binary tree of height h has $2^{h+1} - 1$ nodes.

Here height is 3 $\therefore$ No. of nodes in full binary tree is = $2^{3+1}-1$

= 15 nodes.

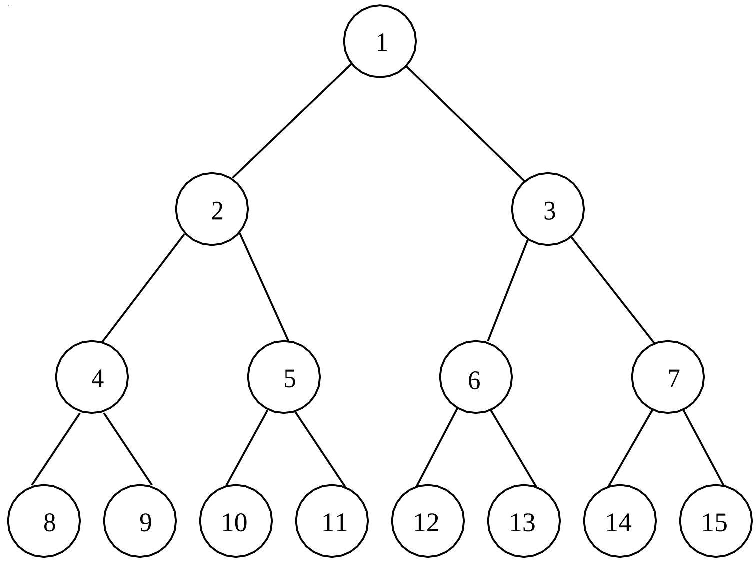


Fig. 3.3.2 A Full Binary Tree

Complete Binary Tree :

A complete binary tree of height h has between 2^h and 2^{h+1} - 1 nodes. In the bottom level the elements should be filled from left to right.

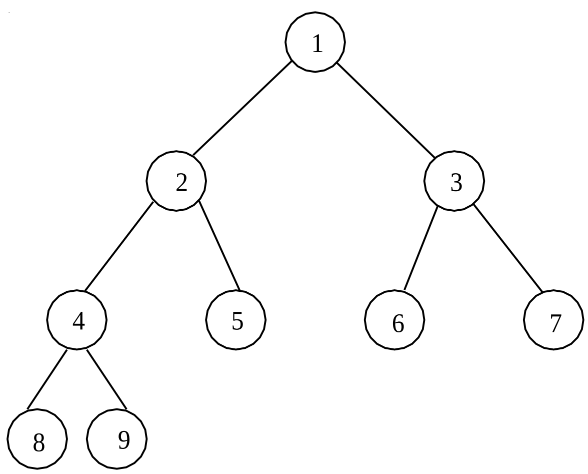


Fig. 3.3.3 A Complete Binary Tree.

Note : A full binary tree can be a complete binary tree, but all complete binary tree is not a full binary tree.

3.3.1 Representation of a Binary Tree

There are two ways for representing binary tree, they are

* Linear Representation
* Linked Representation

Linear Representation

The elements are represented using arrays. For any element in 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).

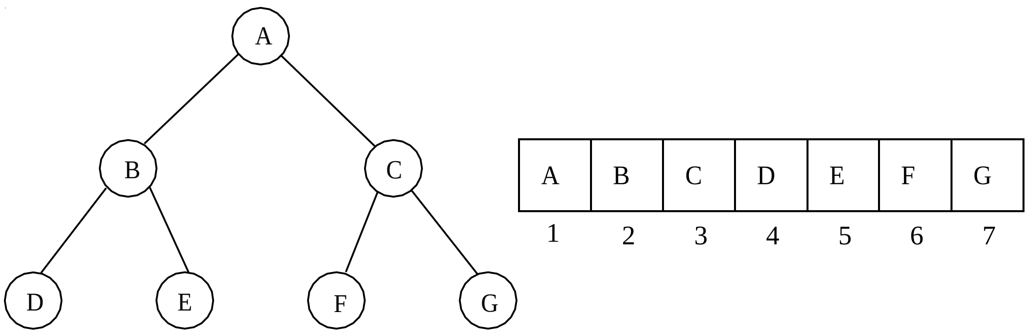


Fig. 3.3.4 Linear Representation

Linked Representation

The elements are represented using pointers. Each node in linked representation has three fields, namely,

* Pointer to the left subtree
* Data field
* Pointer to the right subtree

In leaf nodes, both the pointer fields are assigned as NULL.

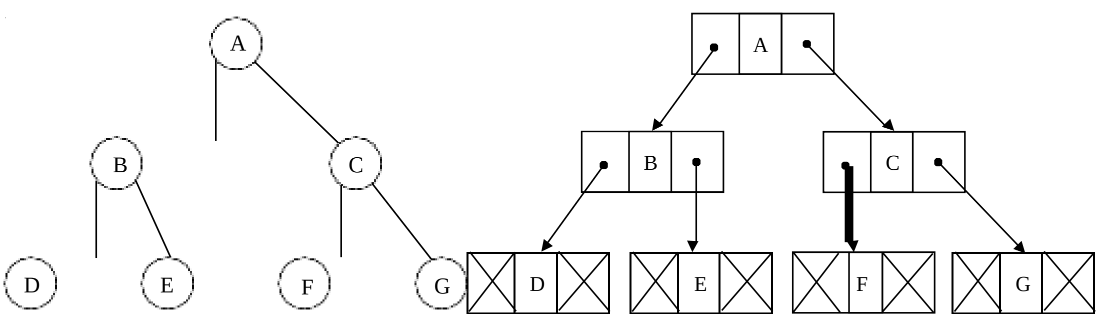


Fig. 3.3.5 Linked Representation

3.3.2 The Leftmost-child, Right-sibling Data Structures

In this representation, cellspace contains three fields namely, leftmost child, label and right sibling. A node is identified with the index of the cell in cellspace that represents it as a child. Then, next pointers of cellspace point to right siblings, and the information contained in the nodespace array can be held by introducing a field leftmost-child in cellspace.

Declaration of cellspace in leftmost-child right-sibling data structure

```
Typedef struct cellspace * ptrtonode;
    Struct cellspace
    {
        Element type    label;
        ptrtonode       leftmost-child;
        ptrtonode       right sibling;
    }
```

Example 1:

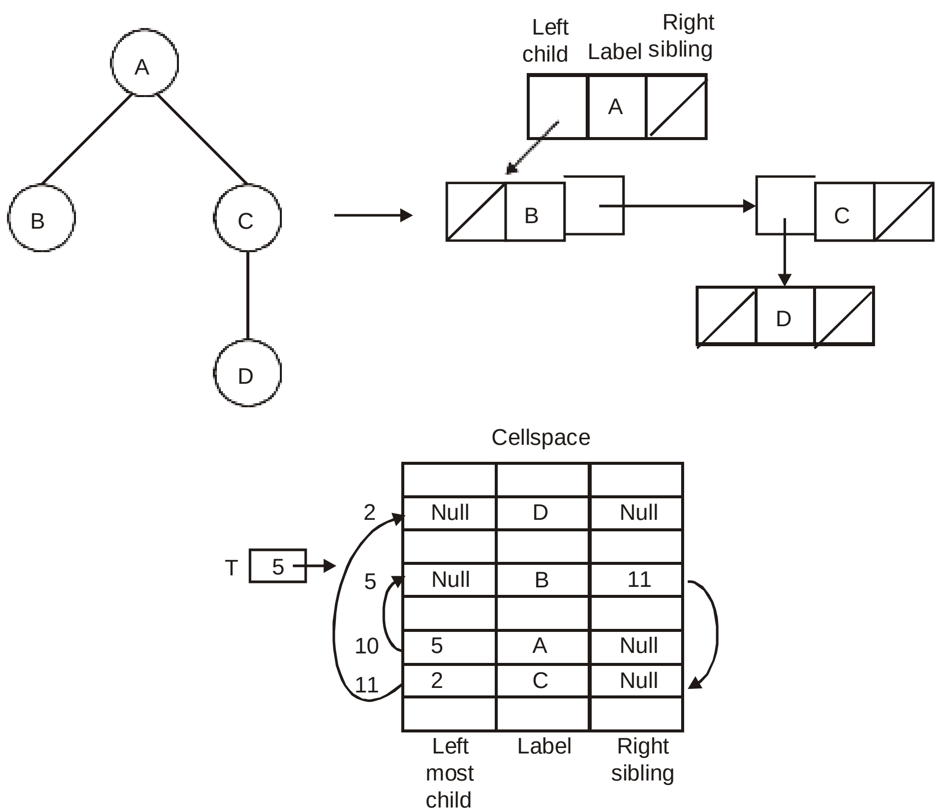
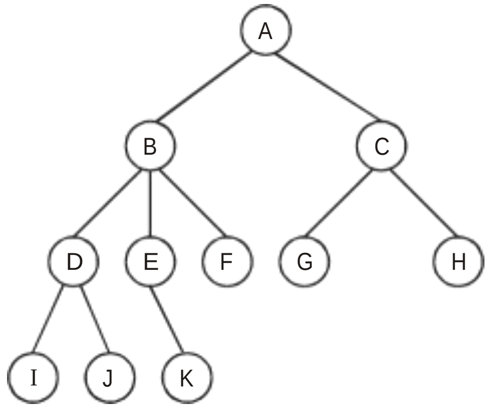
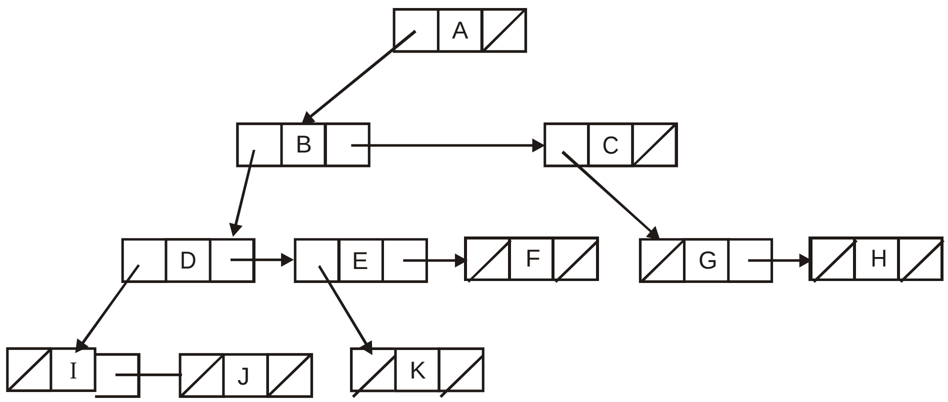


Figure 3.3.6 : Leftmost-child, right-sibling representation of a tree

Example 2:





1	2	B	10
2	15	D	3
3	14	E	11
7	1	A	Null
T 7			
10	13	C	Null
11	Null	F	Null
12	Null	H	Null
13	Null	G	12l
14	Null	K	Null
15	Null	I	16
16	Null	J	Null

Figure 3.3.7: Leftmost-child, right-sibling representation of the above tree

3.4 EXPRESSION TREE

Expression Tree is a binary tree in which the leaf nodes are operands and the interior nodes are operators. Like binary tree, expression tree can also be travesed by inorder, preorder and postorder traversal.

Constructing an Expression Tree

Let us consider postfix expression given as an input for constructing an expression tree by performing the following steps :

1. Read one symbol at a time from the postfix expression.
2. Check whether the symbol is an operand or operator.
   (a) If the symbol is an operand, create a one - node tree and push a pointer on to the stack.
   (b) If the symbol is an operator pop two pointers from the stack namely T_1 and T_2 and form a new tree with root as the operator and T_2 as a left child and T_1 as a right child. A pointer to this new tree is then pushed onto the stack.

Example : -

ab + c *

The first two symbols are operand, so create a one node tree and push the pointer on to the stack.

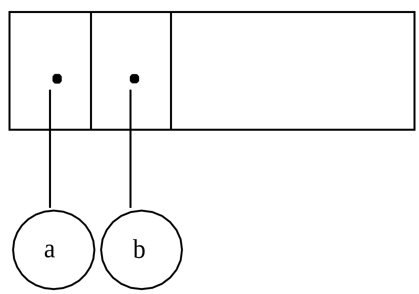


Fig. 3.4.1 (a)

Next ‗+‘ symbol is read, so two pointers are popped, a new tree is formed and a pointer to this is pushed on to the stack.

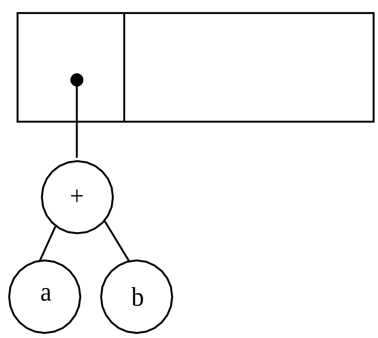


Fig. 3.4.2 (b)

Next the operand C is read, so a one node tree is created and the pointer to it is pushed onto the stack.

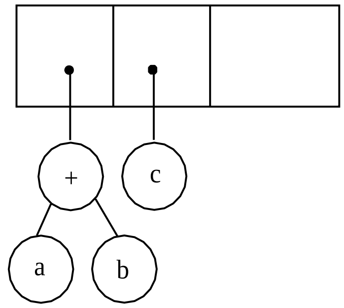


Fig. 3.4.3 (c)

Now _*' is read, so two trees are merged and the pointer to the final tree is pushed onto the stack.

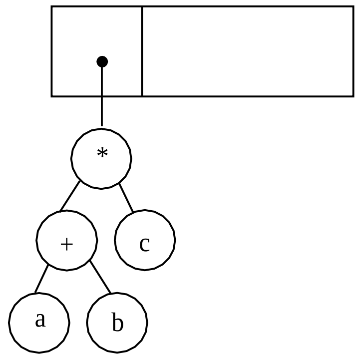


Fig. 3.4.3 (d)

3.5. APPLICATIONS OF TREE

- **Binary Search Tree** - Used in *many* search applications where data is constantly entering/ leaving, such as the map and set objects in many languages' libraries.
- **Binary Space Partition** - Used in almost every 3D video game to determine what objects need to be rendered.
- **Binary Tries** - Used in almost every high-bandwidth router for storing router-tables.
- **Hash Trees** - used in p2p programs and specialized image-signatures in which a hash needs to be verified, but the whole file is not available.
- **Heaps** - Used in implementing efficient priority-queues, which in turn are used for scheduling processes in many operating systems, Quality-of-Service in routers, and A* *(path-finding algorithm used in AI applications, including robotics and video games)*. Also used in heap-sort.
- **Huffman Coding Tree (Chip Uni)** - used in compression algorithms, such as those used by the .jpeg and .mp3 file-formats.

- **GGM Trees -** Used in cryptographic applications to generate a tree of pseudo-random numbers.
- **Syntax Tree -** Constructed by compilers and (implicitly) calculators to parse expressions.
- **Treap -** Randomized data structure used in wireless networking and memory allocation.
- **T-tree -** Though most databases use some form of B-tree to store data on the drive, databases which keep all (most) their data in memory often use T-trees to do so.

BTree :

We use BTree in indexing large records in database to improve search.

3.6 THE SEARCH TREE ADT : - BINARY SEARCH TREE

Definition : -

Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

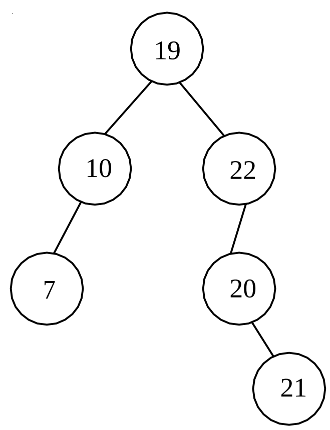
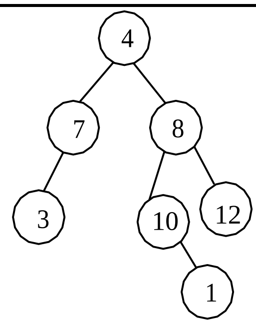
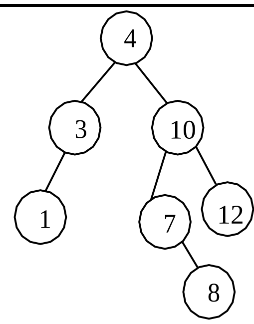


Fig. 3.6.1 Binary Search Tree

Comparision Between Binary Tree & Binary Search Tree

Binary Tree	Binary Search Tree
* A tree is said to be a binary tree if it has atmost two childrens. * It doesn't have any order. * Example	* A binary search tree is a binary tree in which the key values in the left node is less than the root and the keyvalues in the right node is greater than the root.





Note : * Every binary search tree is a binary tree.
* All binary trees need not be a binary search tree.

Declaration Routine for Binary Search Tree

```
Struct TreeNode;
typedef struct TreeNode * SearchTree;
SearchTree Insert (int X, SearchTree T);
SearchTree Delete (int X, SearchTree T);
int Find (int X, SearchTree T);
int FindMin (Search Tree T);
int FindMax (SearchTree T);
SearchTree MakeEmpty (SearchTree T);
Struct TreeNode
{
    int Element ;
    SearchTree Left;
    SearchTree Right;
};
```

Make Empty :-

This operation is mainly for initialization when the programmer prefer to initialize the first element as a one - node tree.

Routine to Make an Empty Tree :-

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

Insert : -

To insert the element X into the tree,

* Check with the root node T
* If it is less than the root,

Traverse the left subtree recursively until it reaches the T → left equals to NULL. Then X is placed in T → left.

* If X is greater than the root.

Traverse the right subtree recursively until it reaches the T → right equals to NULL. Then X is placed in T → Right.

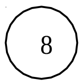
Routine to Insert Into a Binary Search Tree

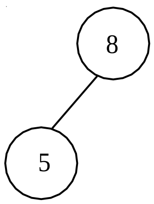
```
SearchTree Insert (int X, searchTree T)
{
    if (T = = NULL)
    {
        T = malloc (size of (Struct TreeNode));
        if (T! = NULL) // First element is placed in the root.
        {
            T → Element = X;
            T → left    = NULL;
            T → Right   = NULL;
        }
    }
    else
        if (X < T → Element)
            T → left = Insert (X, T → left);
        else
        if (X > T → Element)
            T → Right = Insert (X, T → Right);
    // Else X is in the tree already.
    return T;
}
```

Example : -

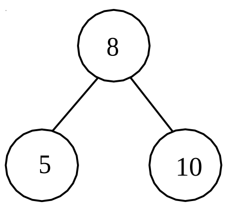
To insert 8, 5, 10, 15, 20, 18, 3

* First element 8 is considered as Root.

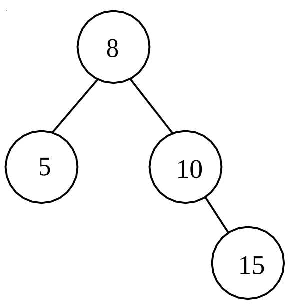




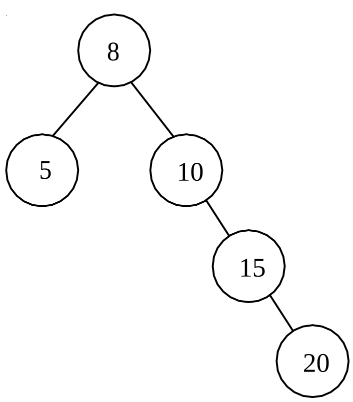
As 5 < 8, Traverse towards left



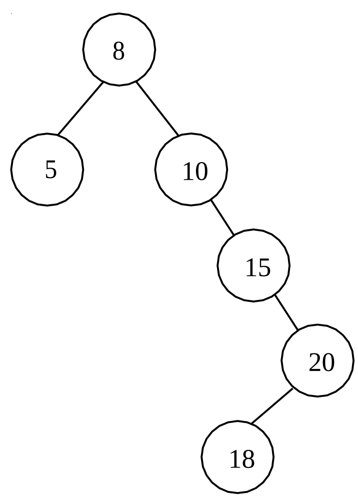
10 > 8, Traverse towards Right.



Similarly the rest of the elements are traversed.



After 20



After 18

Find : -

* Check whether the root is NULL if so then return NULL.
* Otherwise, Check the value X with the root node value (i.e. T → data)

    (1) If X is equal to T → data, return T.

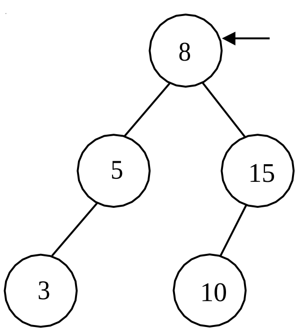
    (2) If X is less than T → data, Traverse the left of T recursively.

    (3) If X is greater than T → data, traverse the right of T recursively.

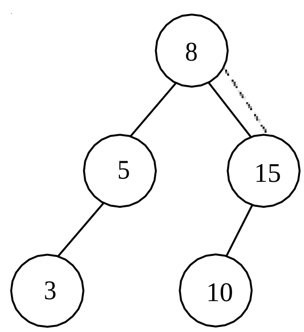
Routine for find Operation

```
Int Find (int 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; // returns the position of the search element.
}
```

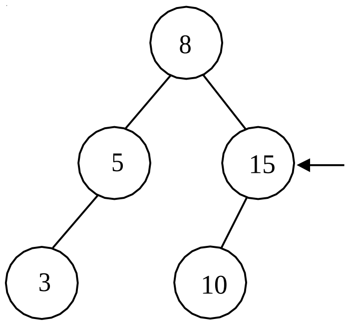
Example : - To Find an element 10 (consider, X = 10)



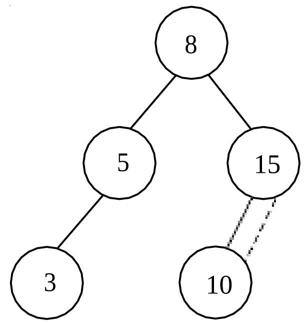
10 is checked with the Root



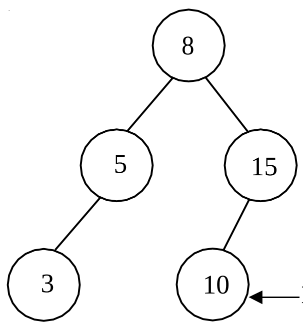
10 > 8, Go to the right child of 8



10 is checked with Root 15



10 < 15, Go to the left child of 15.



10 is checked with root 10 (Found)

Find Min :

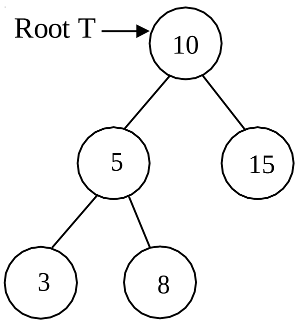
This operation returns the position of the smallest element in the tree.

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

Recurisve Routine For Findmin

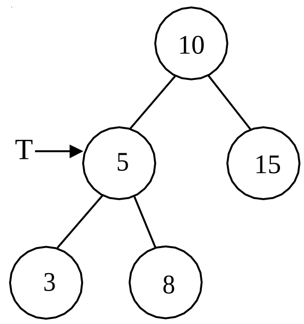
```
int FindMin (SearchTree T)
{
      if (T = = NULL);
            return NULL ;
      else if (T → left = = NULL)
            return T;
      else
            return FindMin (T → left);
}
```

Example : -



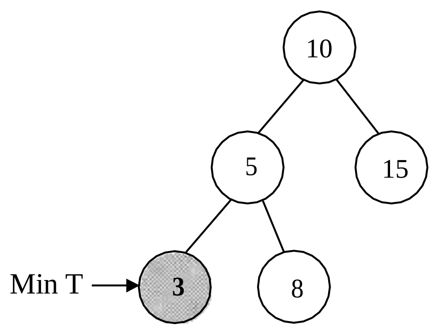
(a) T! = NULL and T →left!=NULL,

Traverse left



(b) T! = NULL and T →left!=NULL,

Traverse left



(c) Since T →left is Null, return T as a minimum element.

Non - Recursive Routine For Findmin

```
int FindMin (SearchTree T)
{
    if (T! = NULL)
        while (T → Left ! = NULL)
            T = T → Left ;
    return T;
}
```

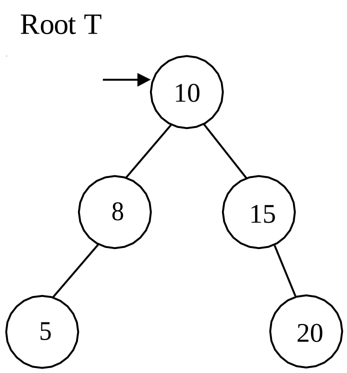
FindMax

FindMax routine return the position of largest elements in the tree. To perform a FindMax, start at the root and go right as long as there is a right child. The stopping point is the largest element.

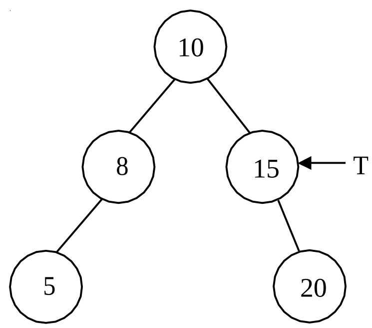
Recursive Routine for Findmax

```
int FindMax (SearchTree T)
{
    if (T = = NULL)
        return NULL ;
    else if (T → Right = = NULL)
        return T;
    else FindMax (T → Right);
}
```

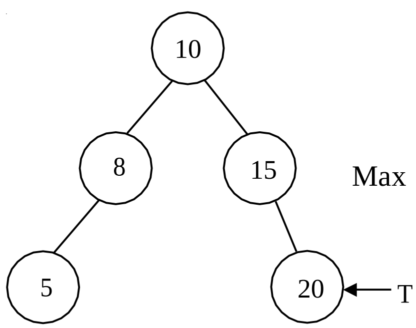
Example :-



(a) T! = NULL and T →Right!=NULL, Traverse Right



(b) T! = NULL and T →Right!=NULL, Traverse Right



(c) Since T → Right is NULL, return T as a Maximum element.

Non - Recursive Routine for Findmax

```
int FindMax (SearchTree T)
{
    if (T! = NULL)
        while (T →Right ! = NULL)
            T = T → Right ;
    return T ;
}
```

Delete :

Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

CASE 1 → Node to be deleted is a leaf node (ie) No children.

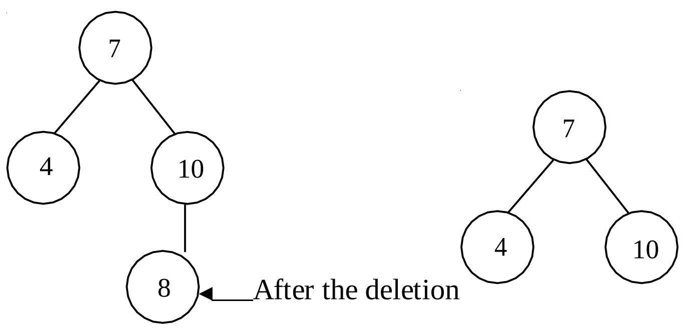
CASE 2 → Node with one child.

CASE 3 → Node with two children.

CASE 1 → Node with no children (Leaf node)

If the node is a leaf node, it can be deleted immediately.

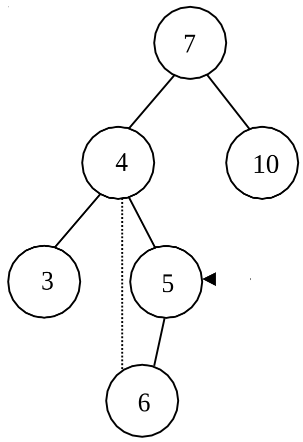
Delete : 8



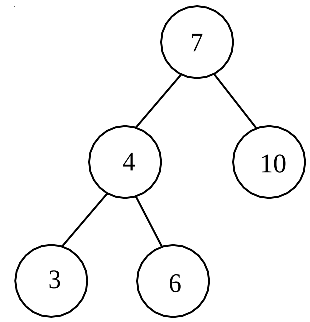
CASE 2 : - Node with one child

If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

To Delete 5



before deletion



After deletion

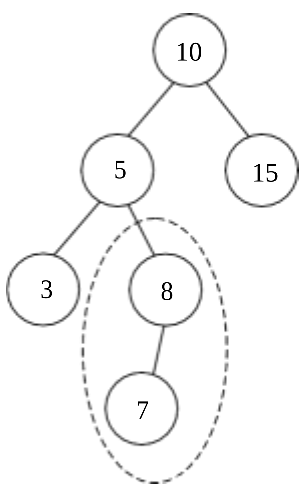
To delete 5, the pointer currently pointing the node 5 is now made to its child node 6.

Case 3 : Node with two children

It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.

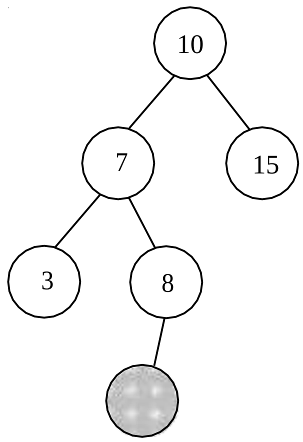
Example 1 :

To Delete 5 :



* The minimum element at the right subtree is 7.

(a)

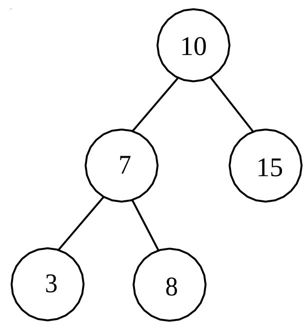


(b)

* Now the value 7 is replaced in the position of 5.

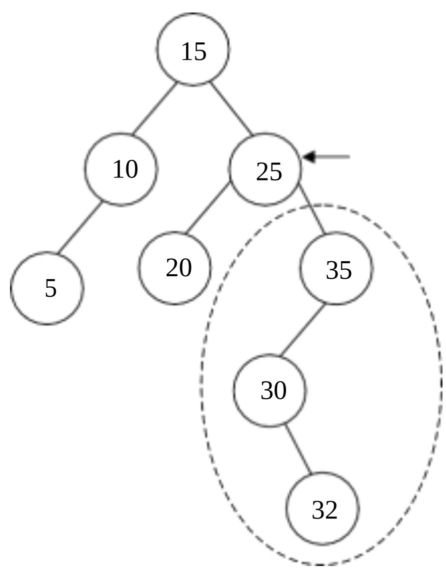
* Since the position of 7 is the leaf node delete immediately.

After deleting the node 5



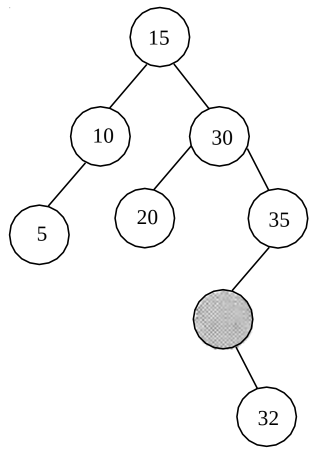
(c)

Example 2 : - To Delete 25



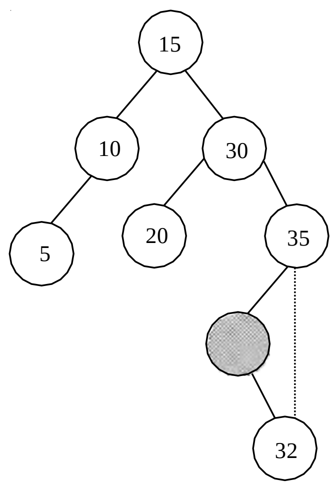
* The minimum element at the right subtree of 25 is 30

(a)



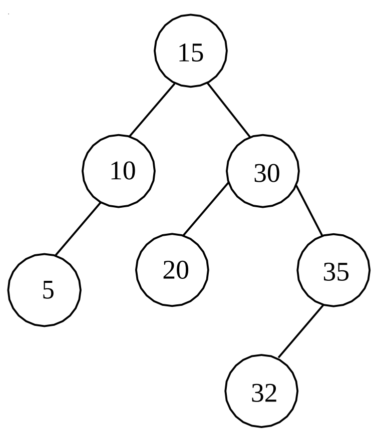
(b)

* The minimum value 30 is replaced in the position of 25



(c)

* Since this node has one child, the pointer currently pointing to this node is made to points to its child node 32



(d)

Binary Search Tree after deleting 25

Deletion Routine for Binary Search Trees

```
SearchTree Delete (int X, searchTree T)
{
    int Tmpcell ;
    if (T = = NULL)
        Error (―Element not found‖);
    else
    if (X < T → Element) // Traverse towards left
        T → Left = Delete (X, T → Left);
    else
    if (X > T → Element) // Traverse towards right
        T → Right = Delete (X, T → Right);
        // Found Element tobe deleted
    else
        // Two children
    if (T → Left && T → Right)
    { // Replace with smallest data 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;
}
```

3.7 THREADED BINARY TREES

Need for threaded binary tree

A binary tree with _n' nodes need 2n pointers out of which (n + 1) are null pointers.

A.J. Perlis and C.Thomton devised a method to utilise these (n + 1) null pointers. These null pointers are now called as threads, which could be effectively used to point to significant nodes choosen according to a traversal scheme to be used for the tree.

Threads that take the place of a left child pointer indicate the inorder predecessor, whereas those taking the place of a right child pointer lead to the inorder successor.

Threaded binary tree

Threaded binary tree is the left subtree of a root node whose right child pointer points to itself. Inorder to keep track of which pointers are threads two more additional br fields TLPOINT and TRPOINT are required in each node.

Linked representation of a threaded binary tree

TLPOINT	LLINK	DATA	RLINK	TRPOINT

Fig. 3.7.1

- For a node _N', if TRPOINT (N) is false then RLINK (N) is a normal pointer.
- If TRPOINT (N) = TRUE then RLINK (N) is a thread pointer, which points to the node which would occur after the node _N' during inorder traversal.

Similarly if TLPOINT (N) = False then LLINK (N) is a normal pointer, else LLINK(N) is a thread pointer, which points to the node that would immediately precede the node N, when the binary tree is traversed in inorder.

Example:(A – B) + C * (E/F)

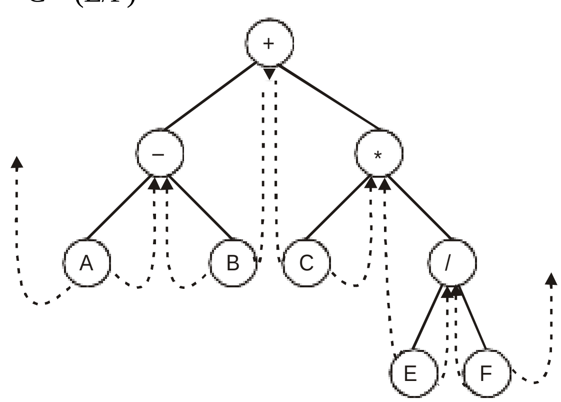


Fig. 3.7.2: Threaded binary tree for the expression (A – B) + C * (E/F)

Two ways of threading

- One way threading
- Two way threading

One way threading: Thread appears only on the RLINK of a node, pointing to the inorder successor of the node.

Two way threading: Threads are appear in both the links LLINK and RLINK and points to the inorder predecessor and inorder successor respectively.

Fig. 3.7.3

An empty threaded binary tree with only.

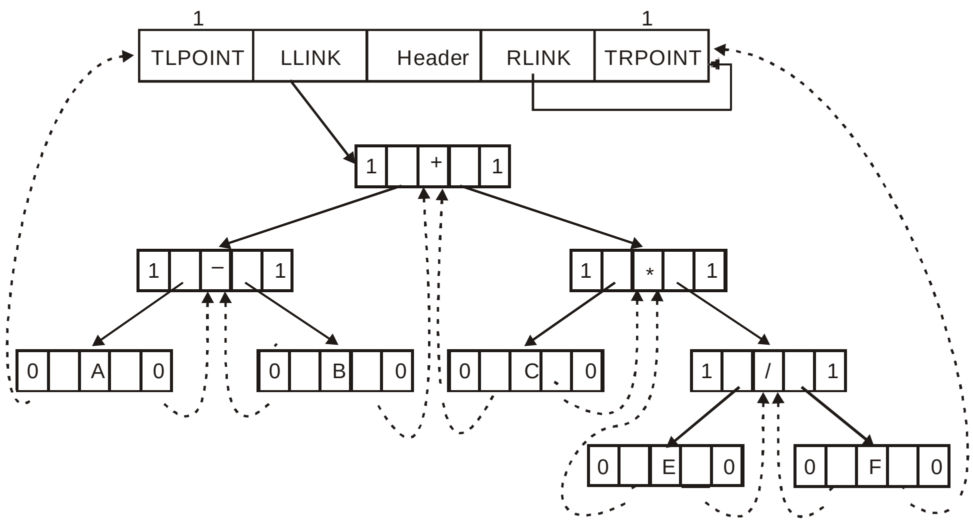


Fig. 3.7.4

Inorder traversal of threaded binary tree

An inorder traversal of a given tree is achieved as follows.

1. Start with the root node.
2. Check whether the right child pointer to the node is a thread or a normal pointer.
3. If it is a thread then it leads directly to the inorder successor.

Else follow the right child pointer to the node it references and from there follow the left most child pointer.

4. Repeat the steps 2 to 3 until a left thread encounters.

Routine for inorder traversal of threaded binary tree

```
Void threaded-inorder (Root P)
   {
      do
         {
            if  P  → TRPOINT   = = True then
                P = P → RLINK;
```

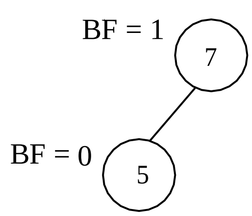
```
Else
                P = P → RLINK;
                while (P  → TLPOINT   !   = TRUE)
                      P = P → LLINK;
            if (P! = ROOT)
                Print f    P   → data;
        }   while (P  =   =   ROOT);
   {
```

3.8 AVL TREE (ADELSON - VELSKILL AND LANDIS)

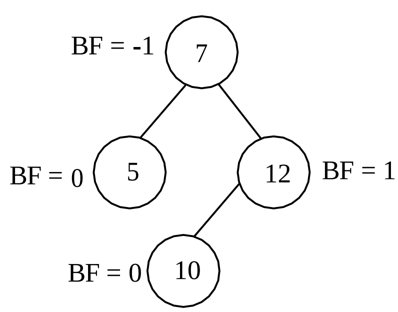
An AVL tree is a binary search tree except that for every node in the tree, the height of the left and right subtrees can differ by atmost 1.

The height of the empty tree is defined to be - 1.

A balance factor is the height of the left subtree minus height of the right subtree. For an AVL tree all balance factor should be +1, 0, or -1. If the balance factor of any node in an AVL tree becomes less than -1 or greater than 1, the tree has to be balanced by making either single or double rotations.



(a)



(b)

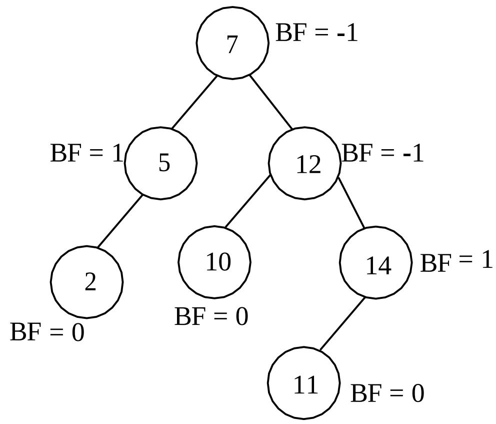


Fig. 3.8.1 AVL Tree

(c)

An AVL tree causes imbalance, when any one of the following conditions occur.

Case 1 : An insertion into the left subtree of the left child of node α .

Case 2 : An insertion into the right subtree of the left child of node α .

Case 3 : An insertion into the left subtree of the right child of node α .

Case 4 : An insertion into the right subtree of the right child of node α .

These imbalances can be overcome by

1. Single Rotation
2. Double Rotation.

Single Rotation

Single Rotation is performed to fix case 1 and case 4.

Case 1. An insertion into the left subtree of the left child of K_2.

Single Rotation to fix Case 1.

General Representation

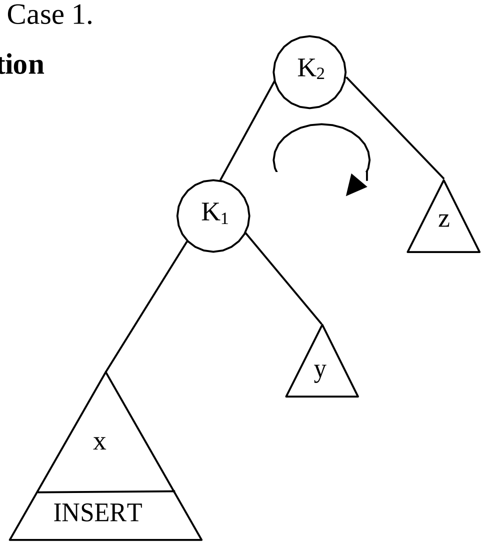


Fig. 3.8.2 (a) Before rotation

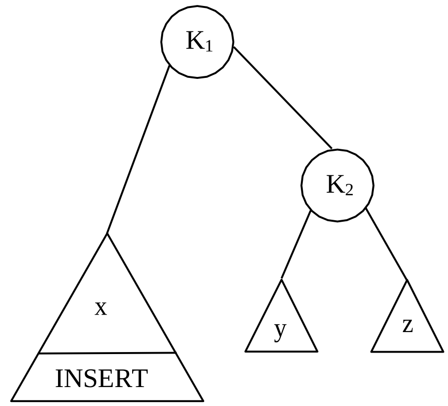


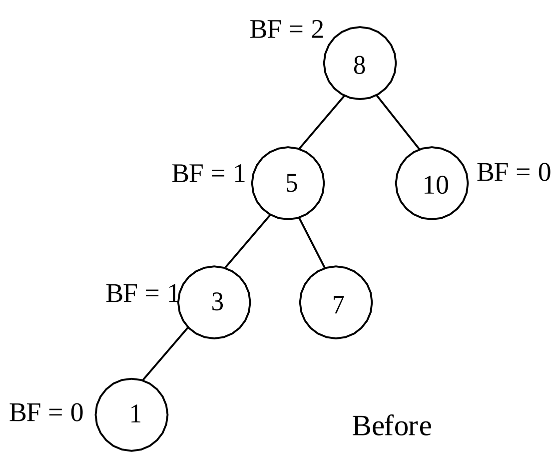
Fig. 3.8.2 (b) After rotation

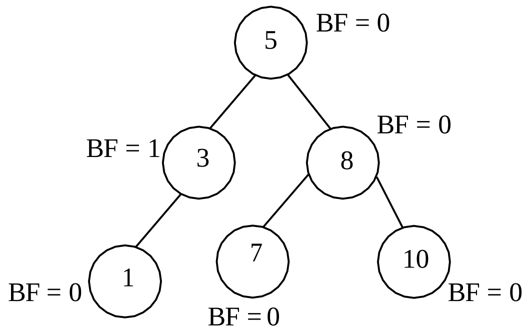
Routine to Perform Single Rotation with Left

SingleRotatewithLeft (Position K_2)
{
Position K_1;
$K_1 = K_2 \rightarrow$ Left ;
$K_2 \rightarrow$ left = $K_1 \rightarrow$ Right ;
$K_1 \rightarrow$ Right = K_2;
$K_2 \rightarrow$ Height = Max (Height ($K_2 \rightarrow$ Left), Height ($K_2 \rightarrow$ Right)) + 1 ;
$K_1 \rightarrow$ Height = Max (Height ($K_1 \rightarrow$ left), Height ($K_1 \rightarrow$ Right)) + 1;
return K_1;
}

Example :

Inserting the value ‗1‘ in the following AVL Tree makes AVL Tree imbalance





After

Fig. 3.8.3

Single Rotation to fix Case 4 :-

Case 4 : - An insertion into the right subtree of the right child of K_1.

General Representation

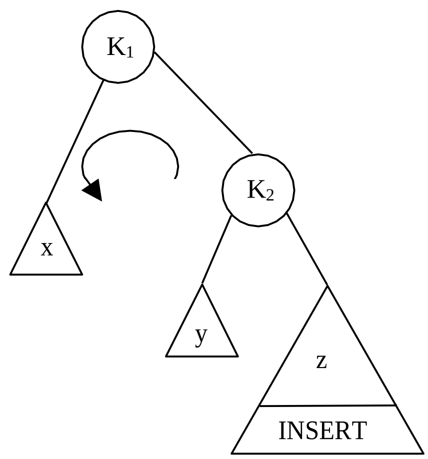


Fig. 3.8.4 (a) Before rotation

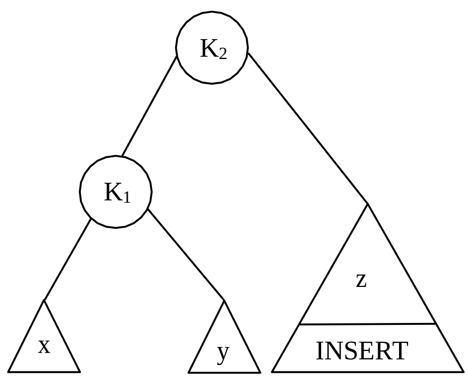


Fig. 3.8.4 (b) After Rotation

Routine to Perform Single Rotation with Right :-

```
Single Rotation With Right (Position K1)
{
        Position K2 ;
        K2 = K1 → Right;
        K1 → Right = K2 → Left ;
        K2 → Left = K1 ;
        K2 → Height = Max (Height (K2 → Left), Height (K2 → Right)) +1 ;
        K1 → Height = Max (Height (K1 → Left), Height (K1 → Right)) +1 ;
        Return K2 ;
}
```

example : -

inserting the value ‗10‘ in the following AVL Tree.

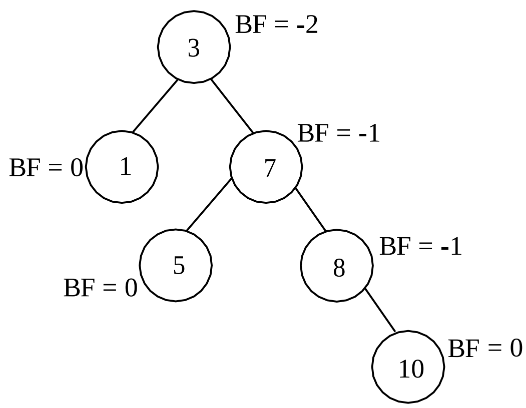


Fig. 3.8.5 (a) AVL Tree with Imbalance

rotate with right

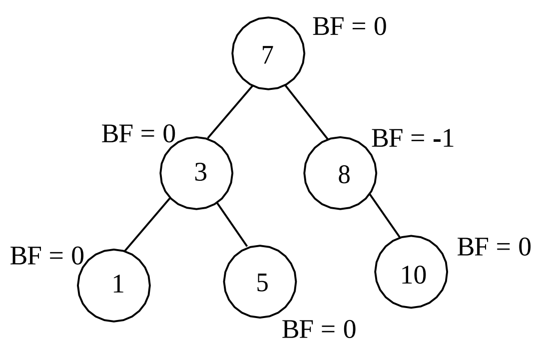


Fig. 3.8.5 (b) Balanced AVL Tree

Double Rotation

Double Rotation is performed to fix case 2 and case 3.

Case 2 :

An insertion into the right subtree of the left child.

General Representation

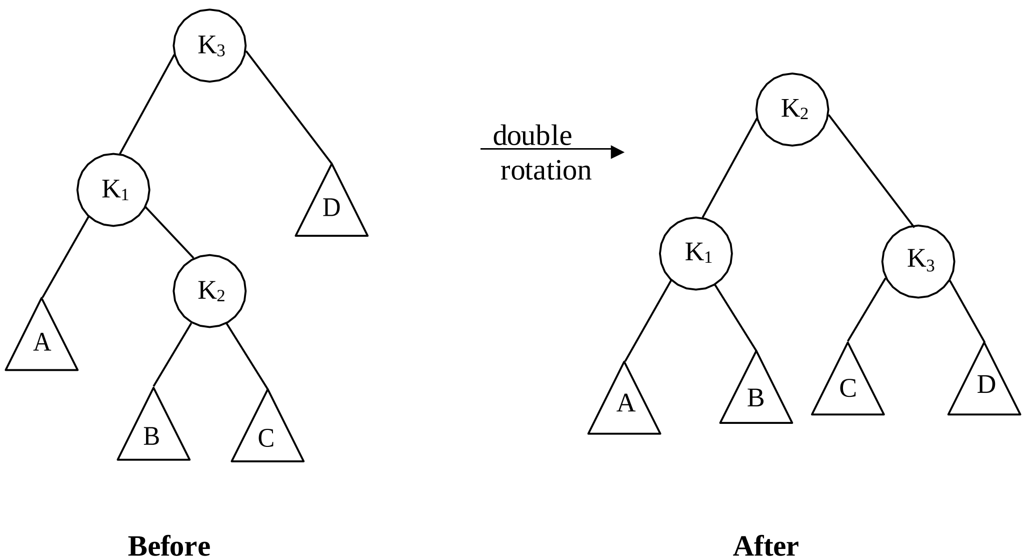
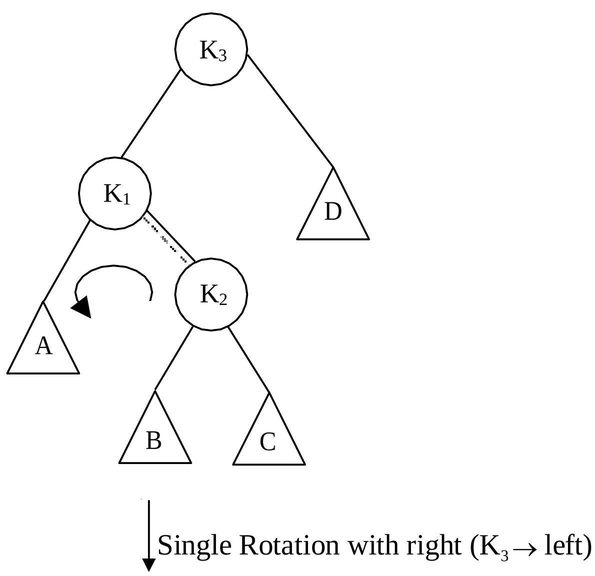


Fig. 3.6.6

This can be performed by 2 single rotations.



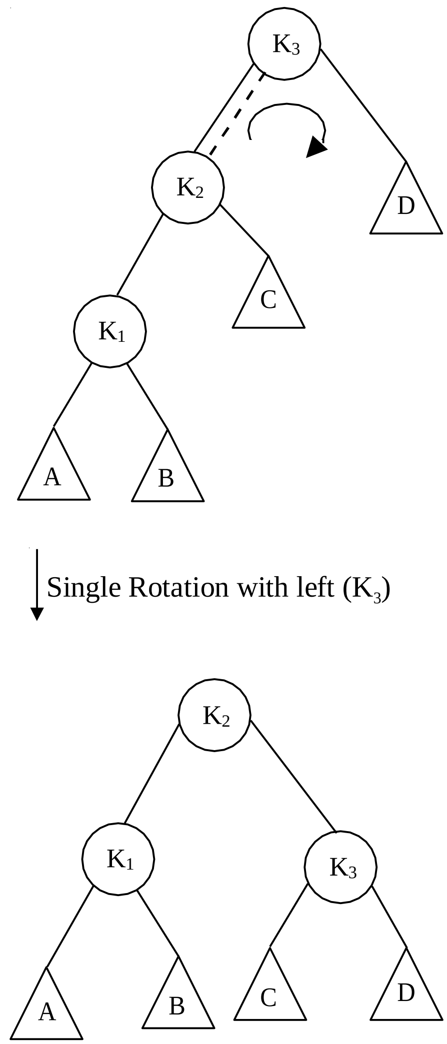
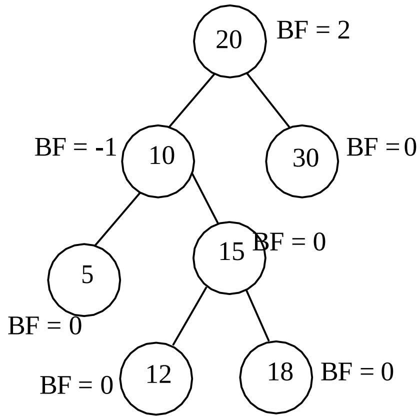


Fig. 3.8.7 Balanced AVL Tree

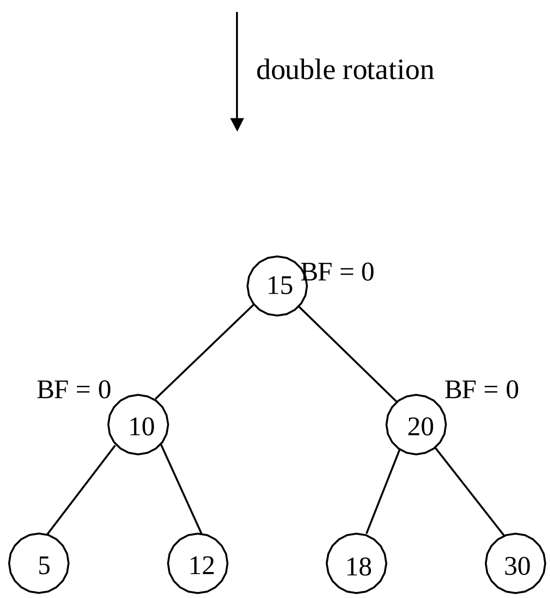
Routine to Perform Double Rotation with Left :

```
Double Rotate with left (Position K3)
{       /* Rotation Between K1 & K2 */
        K3 → Left = Single Rotate with Right (K3 → Left);
                /* Rotation Between K3 & K2 */
        Return Single Rotate With Left (K3);
}
```

Example : Insertion of either _12‘ or _18‘ makes AVL Tree imbalance



(a) before rotation



(b) After rotation

This can be done by performing single rotation with right of _10‘ & then perform the single rotation with left of 20 as shown below.

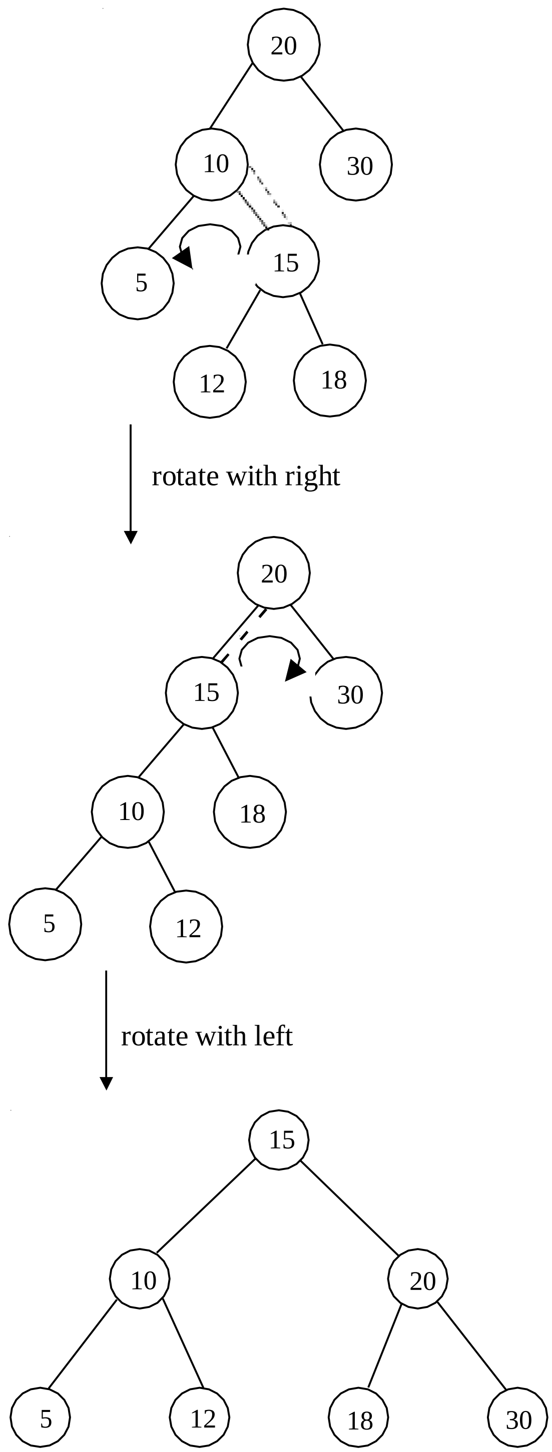


Fig. 3.8.7 Balanced AVL Tree

Case 4 :

An Insertion into the left subtree of the right child of K_1.

General Representation :-

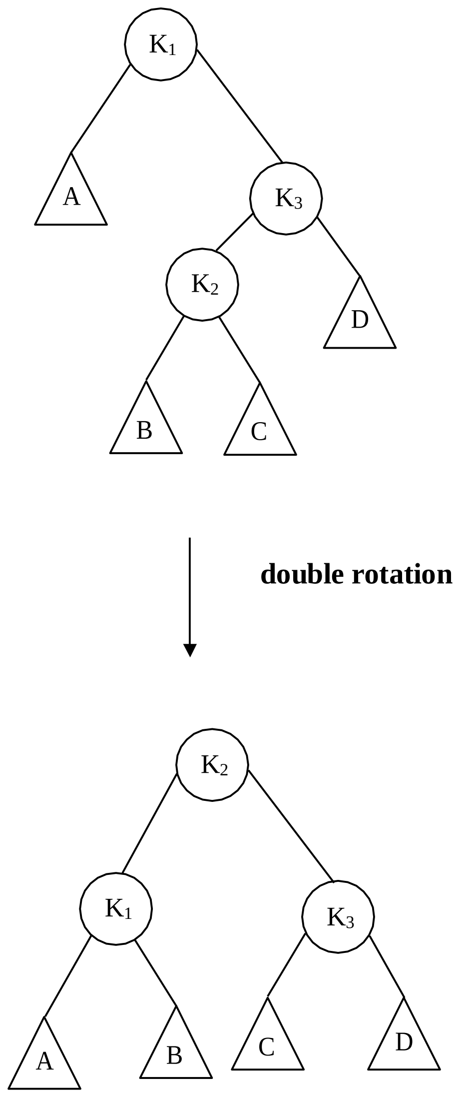


Fig. 3.8.8

This can also be done by performing single rotation with left and then single rotation with right.

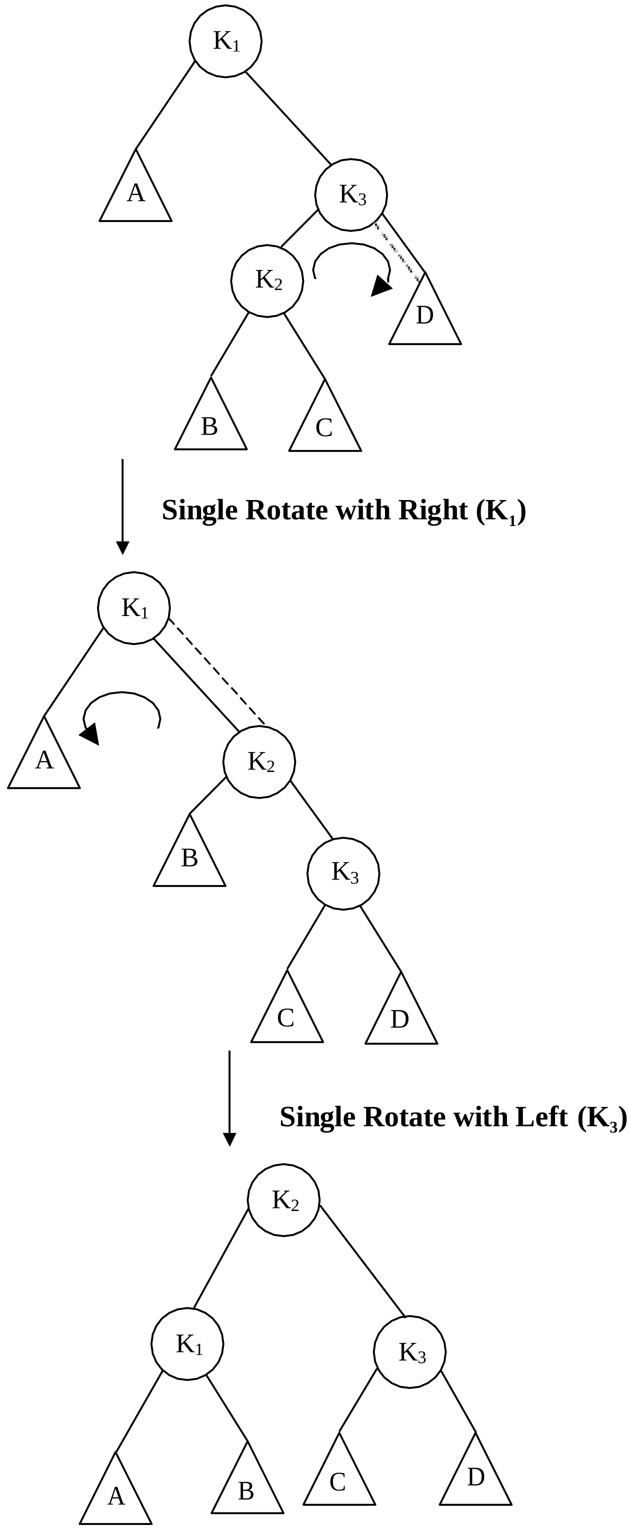


Fig. 3.8.9 Balanced AVL Tree After double rotation.

Routine to Perform Double Rotation with Right :

Double Rotate with Right (Position K_1)

{

/* Rotation Between K_2 & K_3 */

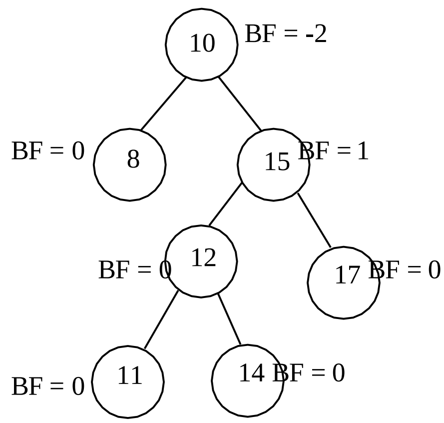
$K_1 \rightarrow$ Right = Single Rotate With Left (K1 $\rightarrow$ Right);

/* Rotation Between K_1 & K_2 */

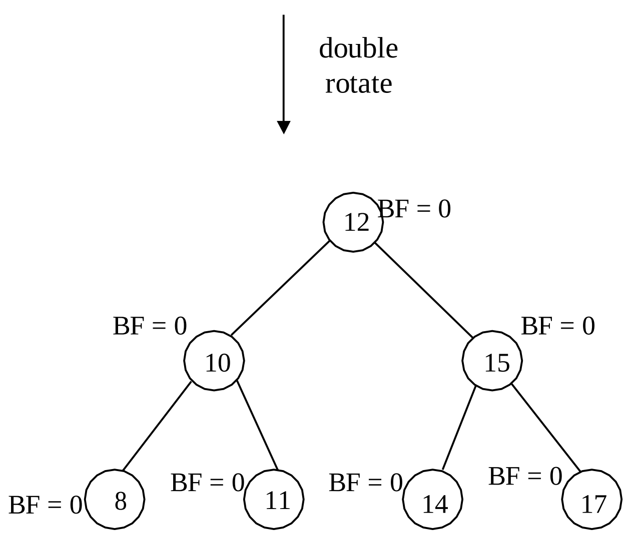
return Single Rotate With Right (K_1);

}

Example :



(a) before



(b) After

This can be done by performing single rotation with left of _15‗ and then performing the single rotation with right of _10‗ as shown below.

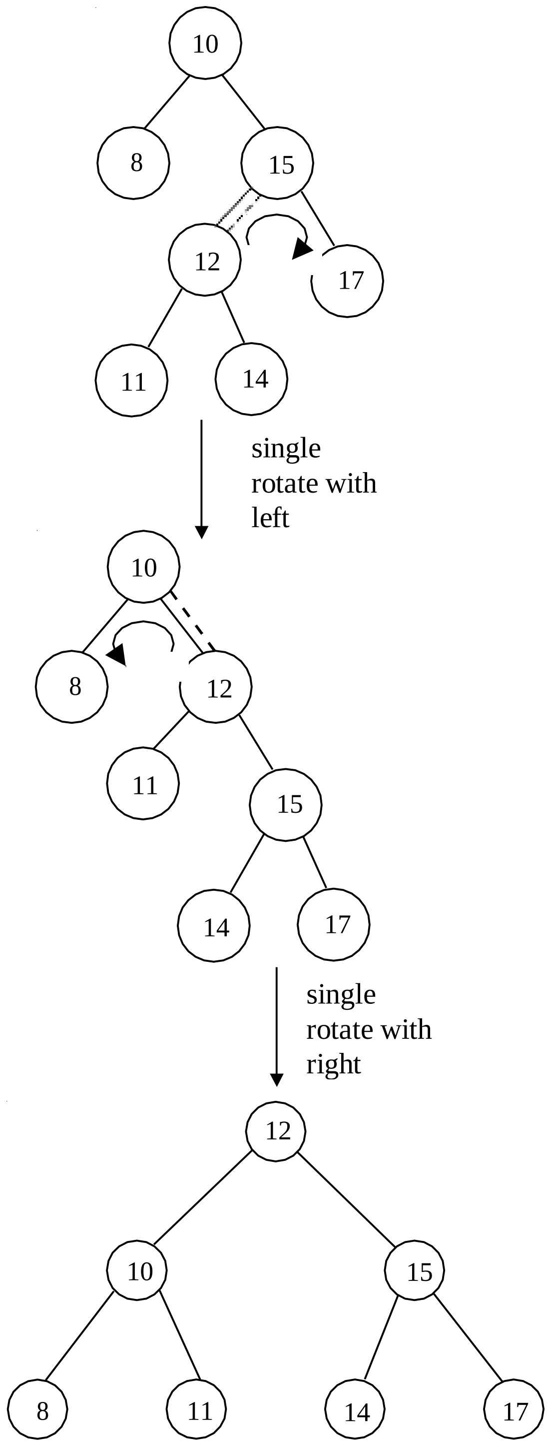


Fig. 3.8.9 BALANCE AVL Tree

Routine to Insert in an Avl Tree : -

```
AVLTree Insert (AVL tree T, int X)
{
    if (T = = NULL)
    {
        T = malloc (size of (Struct AVLnode));
        if (T = = NULL)
            Error (—out of space‖);
        else
        {
            T → data =X;
            T → Height = 0;
            T → Left = NULL;
            T → Right = NULL;
        }
    }
    else
    if (X < T → data)
    {
        T  → left = Insert (T → left,  X);
        if (Height (T → left) - Height (T → Right) = = 2)
            if (X < T → left → data)
                T = Single Rotate With left (T);
            else
                T = Double Rotate is the left (T);
    }
    else
    if (X > T → data)
    {
        T → Right = Insert (T → Right, X);
```

```
                if (Height (T → Right) - Height (T → left) = = 2)
                        if(X > T → Right → Element)
                                T = Single Rotate with Right (T);
                        else
                                T = Double Rotate with Right (T);
        }
        T → Height = Max (Height (T → left), Height (T → Right))+1;
        return T;
}
```

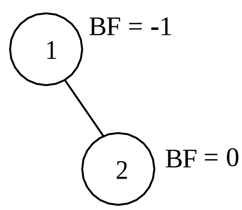
Example :

Let us consider how to balance a tree while inserting the numbers from 1 to 10.

Insert the value 1.

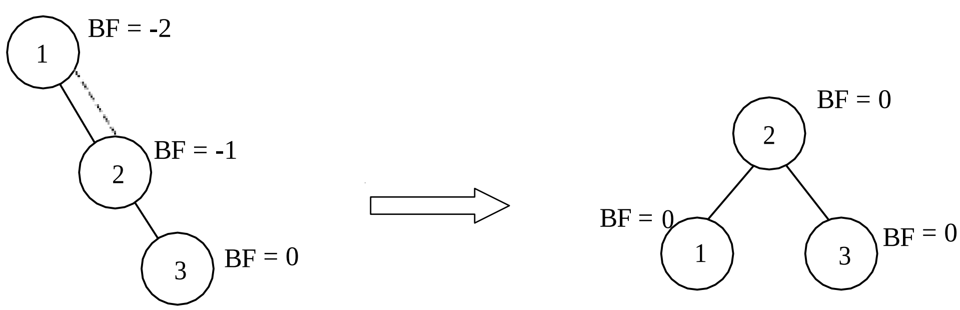
1 BF = 0

Insert the value 2



Balanced Tree

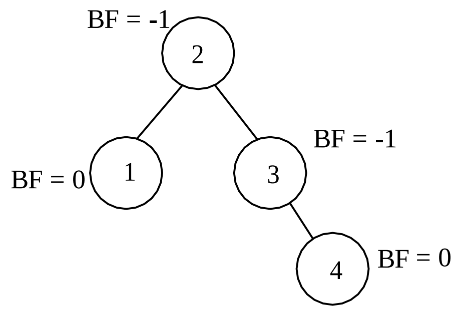
Insert the value 3



Imbalanced Tree **Balanced Tree**

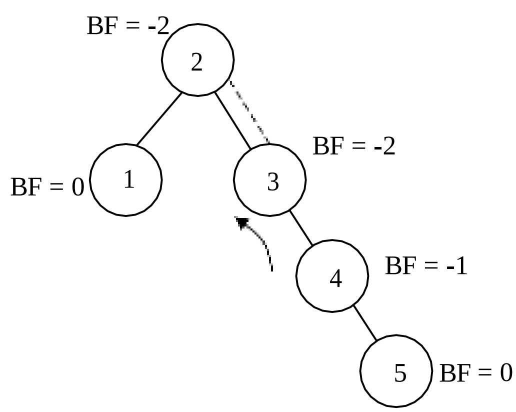
Here the tree imbalances at the node 1. so the single rotation with left is performed.

Insert the value 4

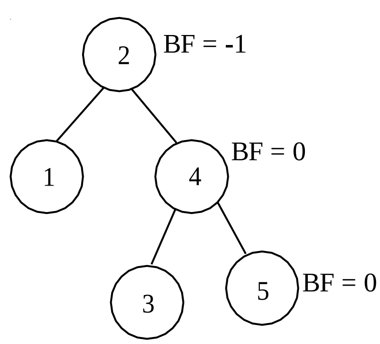


Balanced AVL Tree

Insert the value 5



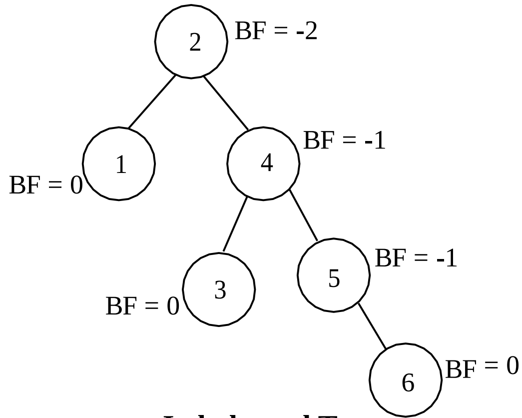
Imbalanced Tree



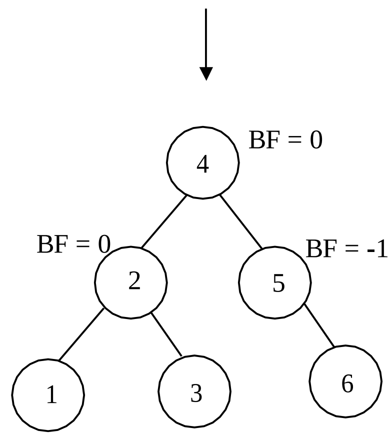
Balanced Tree

Tree is imbalanced at node ‗3', perform the single rotation with left to balance it.

Insert the value 6

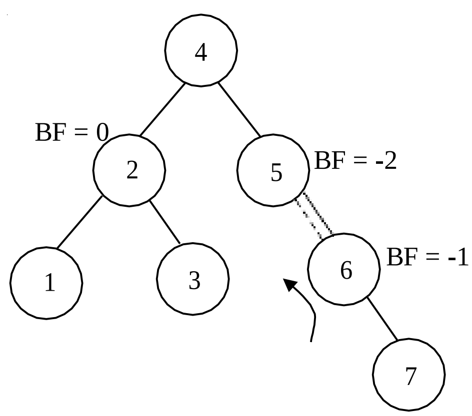


Imbalanced Tree

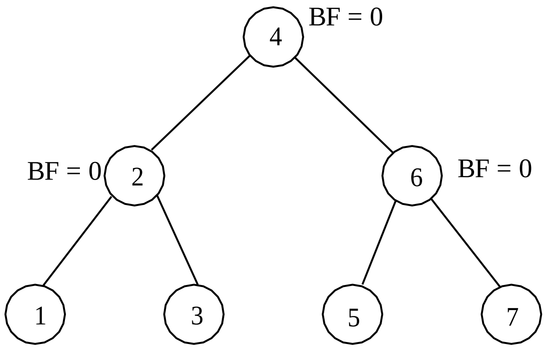


Balanced Tree

Insert the value 7

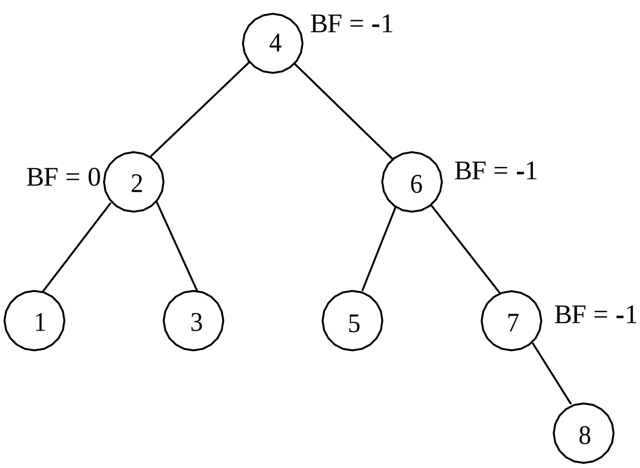


Imbalanced Tree



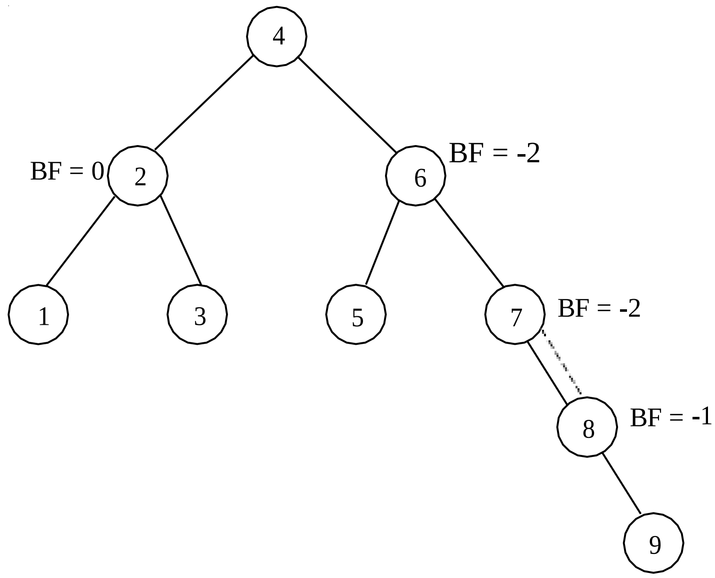
Balanced Tree

Insert the value 8

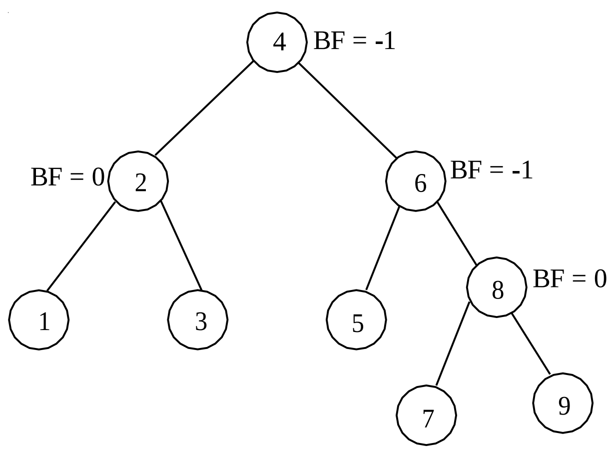


Balanced Tree

insert the value 9

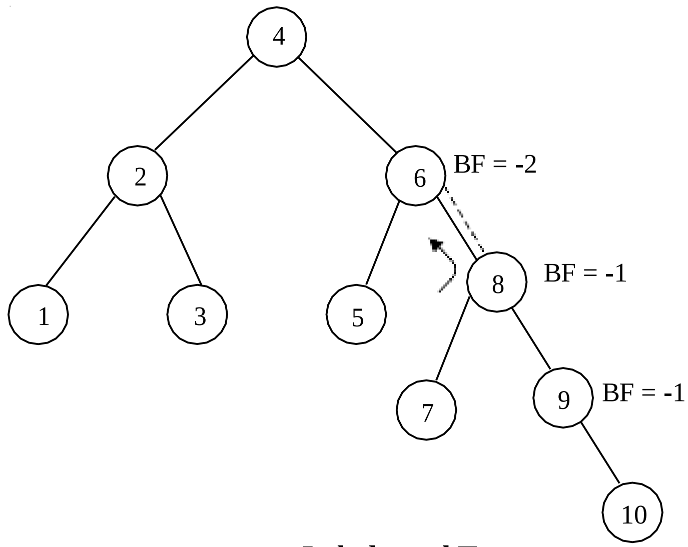


Imbalanced Tree

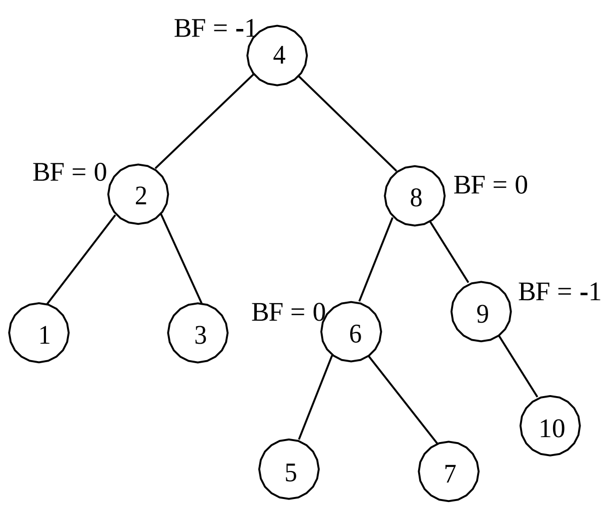


Balanced Tree

Insert the value 10



Imbalanced Tree



Balanced Tree

3.9 B-TREE

A B-Tree of order m is an m-way search tree with the following properties.

- The root node must have atleast two child nodes and atmost m child nodes.
- All internal nodes other than the root node must have atleast m/2 to m non-empty child nodes.
- The number of keys in each internal node is one less than its number of child nodes, which will partition the keys of the tree into subtree.
- All internal nodes are at the same level.

General representation of B-Tree

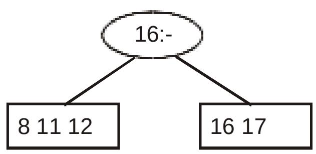


Fig. 3.9.1

Here the non leaf nodes are represented as ellipses, which contain the two pieces of data for each node.

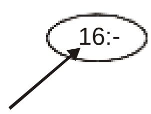


Fig. 3.9.1(a)

1. Represents the key value, which can be the largest element of the left sibling or the smallest element of the right sibling. In this example we considered the smallest element in the right sibling as the key element in the parent node.

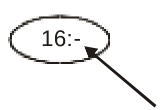
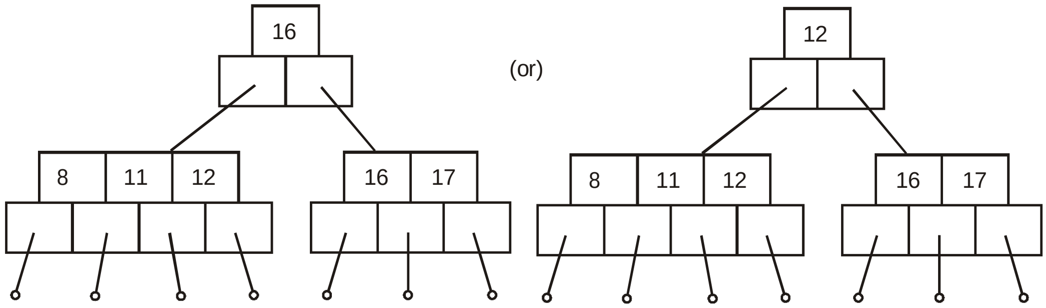


Fig. 3.9.1(b)

2. The dash line indicates that the node has only two children.

B-Tree can also be represented as



Key element in the parent node has smallest element in the right sibling

Key element in the parent node has smalles element in the left sibling

Fig. 3.9.2

Operations on B-Trees

i) Insertion

ii) Deletion

Insertion

To insert a key k in the node X of the B-tree of order m can proceed in one of the two ways.

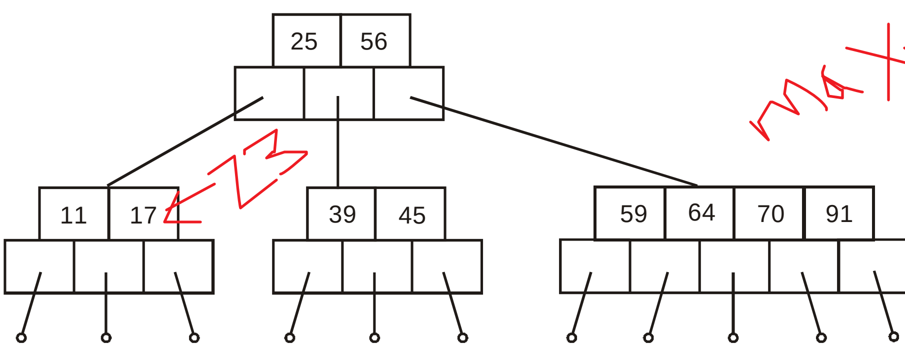
Case 1:

When the node X of the B-tree of order m can accommodate the key K, then it is inserted in that node and the number of child pointer fields are appropriately upgraded.

Example:

m - 1

max - 4



Min

Fig

Fig. 3.9.3 B-Tree of order 5 before insertion

To insert 23

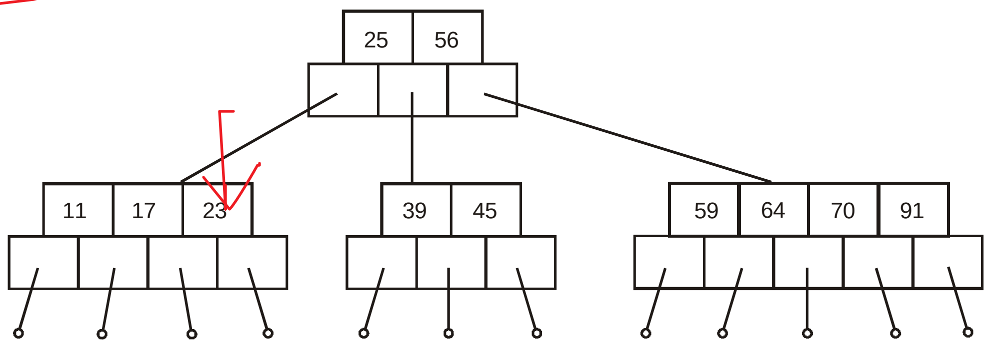


Fig. 3.9.3(a): B-Tree of order 5 after inserting 23

Case 2:

If the node is full, then the key K is apparently inserted into the list of elements and the list is splitted into two on the same level at its median (K_{median}). The keys which are less than K_{median} are placed in the X_{left} and those greater than K_{median} are placed at X_{right}.

The median key is not placed into either of the two new nodes, but is instead moved up the tree to be inserted into the parent node of X. This insertion inturn will call case 1 and 2 depending upon whether the parent node can accommodate or not.

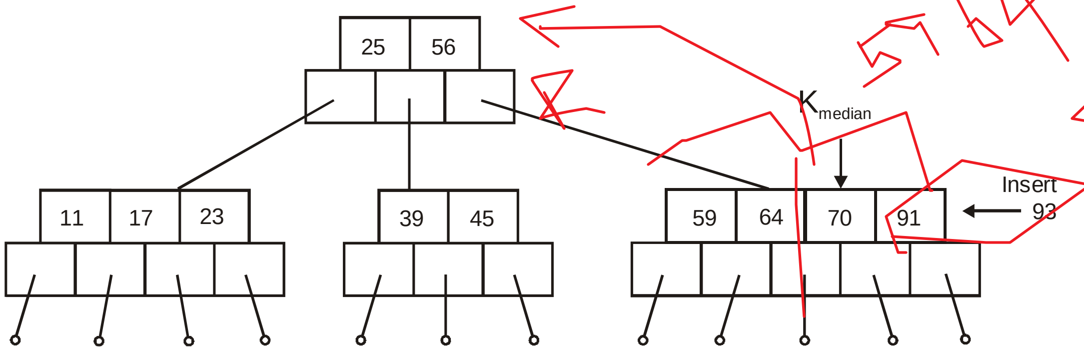


Fig. 3.9.3 (b): B-Tree of order 5 before insertion

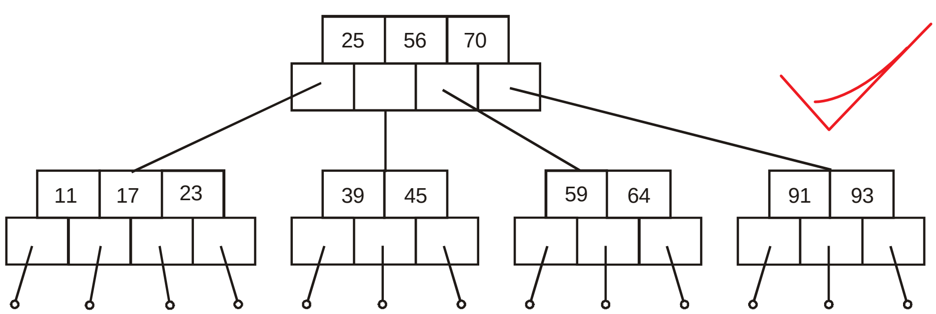


Fig. 3.9.3 (c): B-Tree of order 5 after inserting 93

Deletion

The deletion of a key K for km a B-Tree order m may trigger many cases.

Case 1:

If the key K to be deleted belongs to a leaf node and its deletion does not result in the node having less than its minimum number of elements. Then delete the key from the leaf and adjust the child pointers.

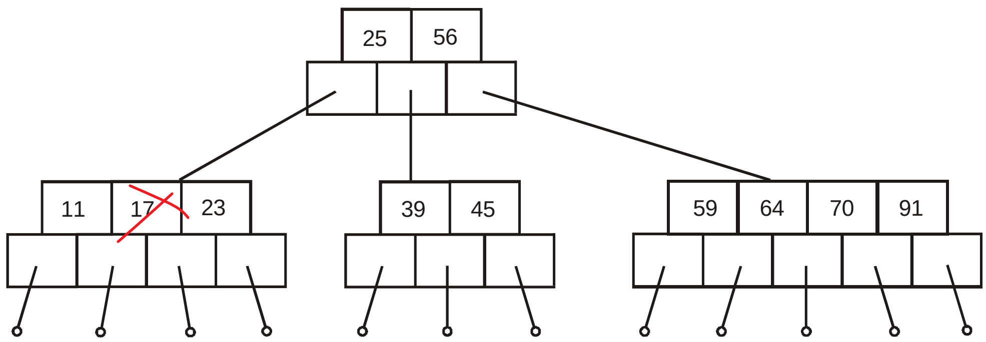


Fig. 3.9.4: B-tree before deletion

To delete 17

The key 17 belongs to a leaf node, so it is deleted immediately.

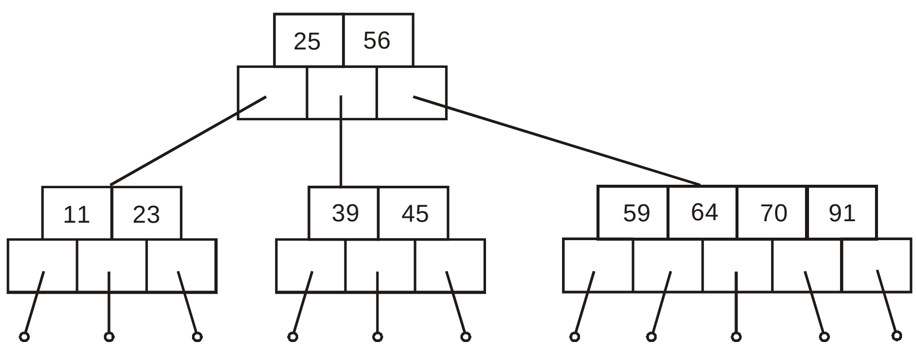


Fig. 3.9.4 (a): B-Tree after deleting the element 17

Case 2:

If the key K belongs to a non leaf node. Then replace K with the largest key K_{Lmax} in the left subtree of K or the smallest key K_{Rmin} from the right subtree of K and then delete K_{Rmin} or K_{Lmax} from the node, which in turn will trigger case 1 or 2.

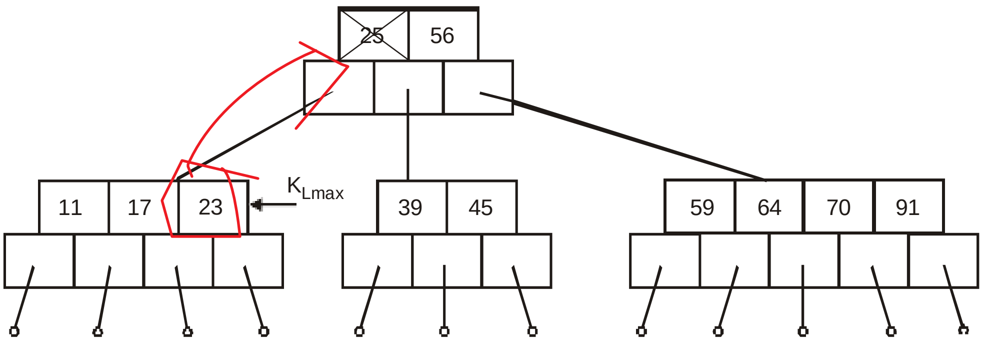


Fig. 3.9.4 (b): B-Tree of order 5 before deletion

To delete 25

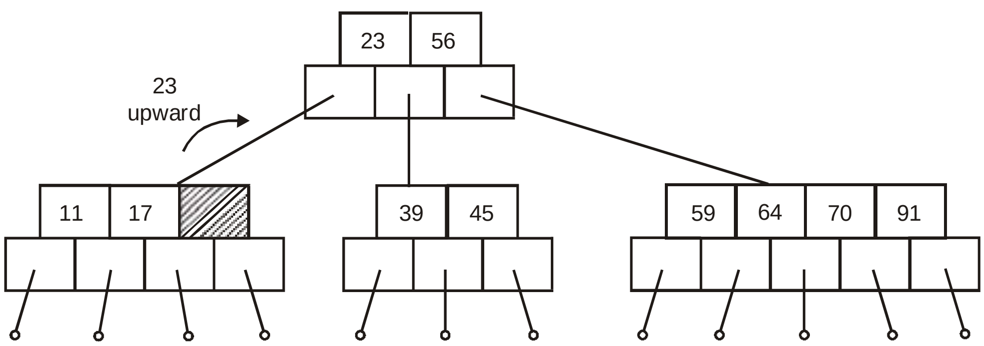


Fig. 3.9.4 (c)

The largest key K_{Lmax} in the left subtree of 25 is replaced and then the key 23 is deleted immediately since it is a leaf node.

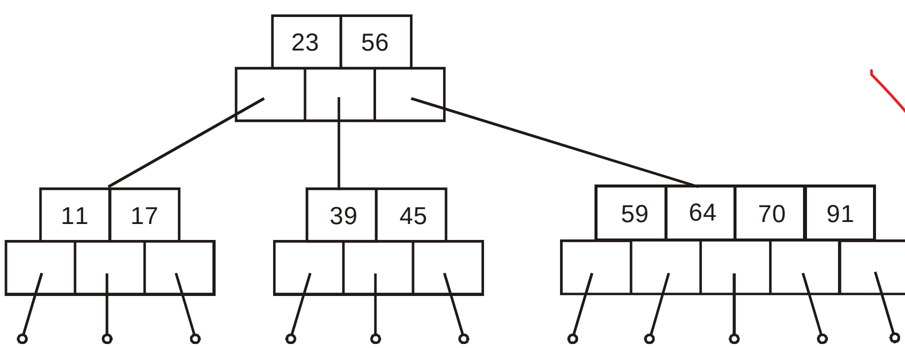


Fig. 3.9.4 (d): B-Tree after deletion of 25

Case 3:

If the key K to be deleted from a node leaves it with less than its minimum number of elements, then the elements may be borrowed either from left or right sibling.

If the left sibling node has an element to spare, then move the largest key K_{Lmax} in the left sibling node to the parent node and the element P in the parent node is moved down to set the vacancy created by the deletion of K in node X.

If the left sibling node has no element to spare then move to case 4.

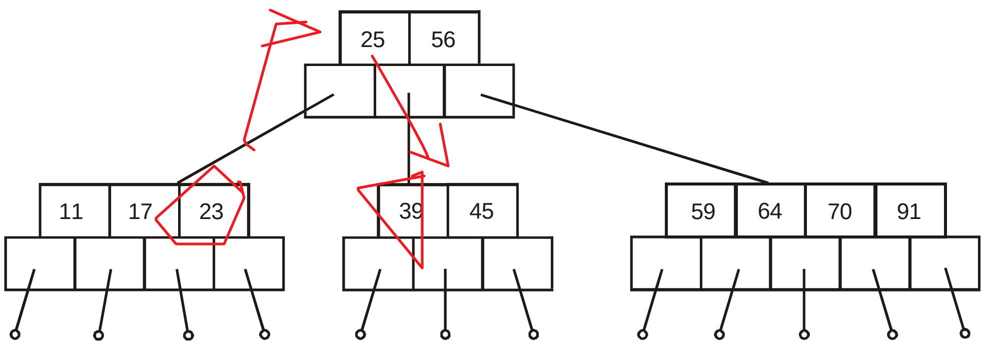


Fig. 3.9.4 (e): B-Tree before deletion

To delete 39

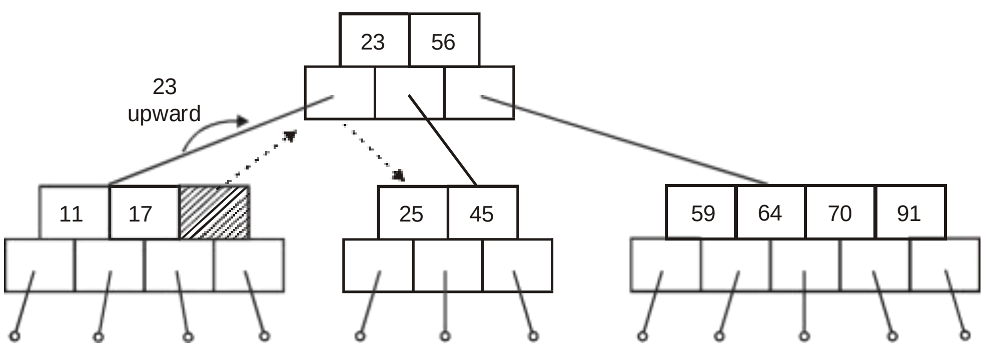


Fig. 3.9.5 (f)

Deleting the key 39 leaves the node less than its minimum number of elements

Here so the largest key 23 from the left sibling is moved to the parent node and the element 25 in the parent node is moved down to set the vacancy created by deleting 39.

Case 4:

If the key K to be deleted from a node X leaves it with less than its minimum number of elements and both the sibling nodes are unable to spare an element. Then the node X is merged with one of the sibling nodes along with intervening element P in the parent node.

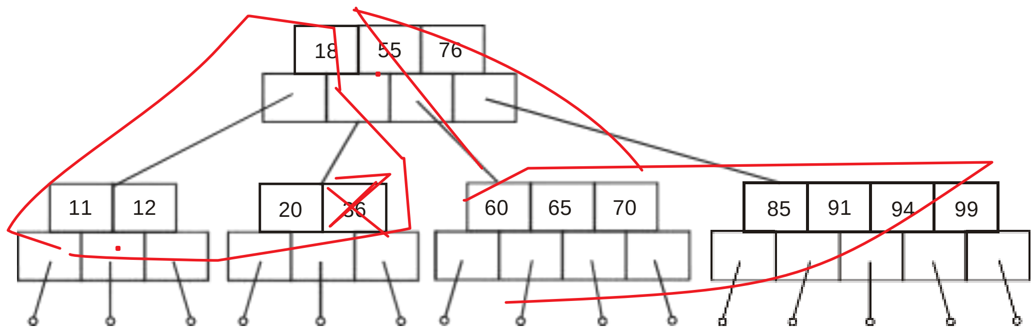


Fig. 3.9.4 (g)

To delete 36

Deleting the key 36 leaves the nodes less than its minimum number of elements and both the siblings are unable to spare. So the node containing key 36 is merged with the left sibling and the intervening parent element 18.

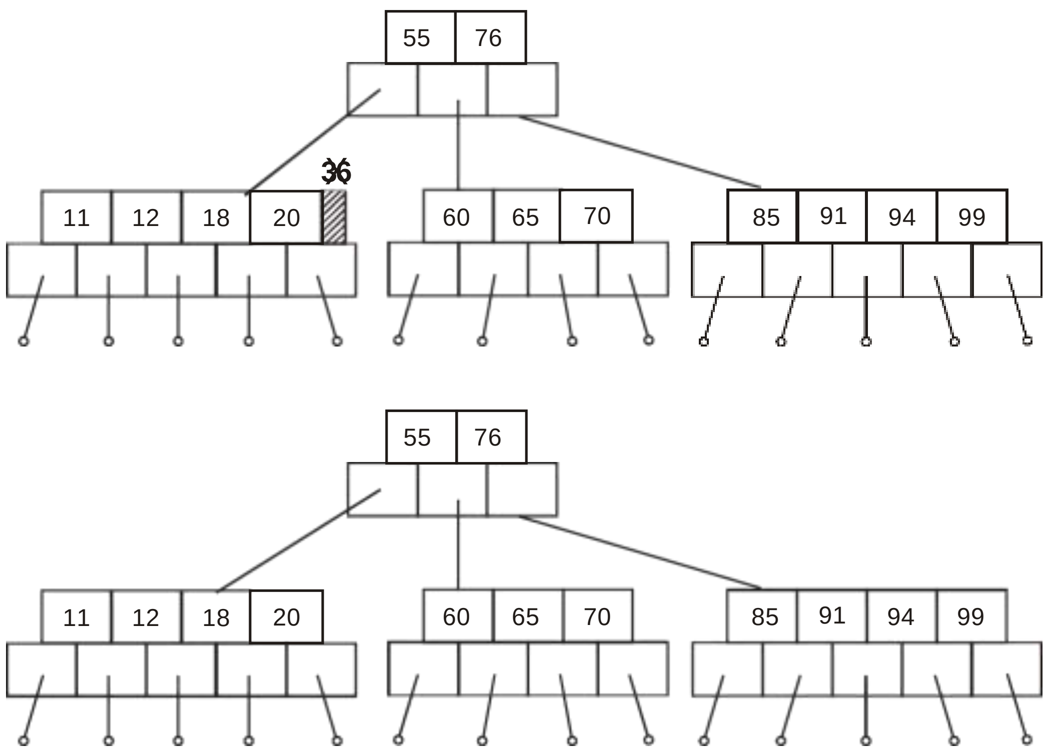


Fig. 3.9.4 (h): B-Tree after deleting 36

Program for B Tree

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 4
#define MIN 2
struct btreeNode {
      int val[MAX + 1], count;
      struct btreeNode *link[MAX + 1];
};
struct btreeNode *root;
/* creating new node */
struct btreeNode * createNode(int val, struct btreeNode *child) {
      struct btreeNode *newNode;
      newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
      newNode->val[1] = val;
      newNode->count = 1;
      newNode->link[0] = root;
      newNode->link[1] = child;
      return newNode;
}
/* Places the value in appropriate position */
void addValToNode(int val, int pos, struct btreeNode *node,
                  struct btreeNode *child) {
      int j = node->count;
      while (j > pos) {
            node->val[j + 1] = node->val[j];
            node->link[j + 1] = node->link[j];
            j—;
      }
      node->val[j + 1] = val;
```

```
        node->link[j + 1] = child;
        node->count++;
    }
    /* split the node */
    void splitNode (int val, int *pval, int pos, struct btreeNode *node,
      struct btreeNode *child, struct btreeNode **newNode) {
        int median, j;
        if (pos > MIN)
                median = MIN + 1;
        else
                median = MIN;
        *newNode = (struct btreeNode *)malloc(sizeof(struct btreeNode));
        j = median + 1;
        while (j <= MAX) {
                (*newNode)->val[j - median] = node->val[j];
                (*newNode)->link[j - median] = node->link[j];
                j++;
        }
        node->count = median;
        (*newNode)->count = MAX - median;
        if (pos <= MIN) {
                addValToNode(val, pos, node, child);
        } else {
                addValToNode(val, pos - median, *newNode, child);
        }
        *pval = node->val[node->count];
        (*newNode)->link[0] = node->link[node->count];
        node->count—;
    }
```

```
/* sets the value val in the node */
int setValueInNode(int val, int *pval,
  struct btreeNode *node, struct btreeNode **child) {
    int pos;
    if (!node) {
        *pval = val;
        *child = NULL;
        return 1;
    }
    if (val < node->val[1]) {
        pos = 0;
    } else {
        for (pos = node->count;
            (val < node->val[pos] && pos > 1); pos—);
        if (val == node->val[pos]) {
            printf(―Duplicates not allowed\n‖);
            return 0;
        }
    }
    if (setValueInNode(val, pval, node->link[pos], child)) {
        if (node->count < MAX) {
            addValToNode(*pval, pos, node, *child);
        } else {
            splitNode(*pval, pval, pos, node, *child, child);
            return 1;
        }
    }
    return 0;
}
```

```
/* insert val in B-Tree */
void insertion(int val) {
    int flag, i;
    struct btreeNode *child;
    flag = setValueInNode(val, &i, root, &child);
    if (flag)
        root = createNode(i, child);
}
/* copy successor for the value to be deleted */
void copySuccessor(struct btreeNode *myNode, int pos) {
    struct btreeNode *dummy;
    dummy = myNode->link[pos];
    for (;dummy->link[0] != NULL;)
        dummy = dummy->link[0];
    myNode->val[pos] = dummy->val[1];
}
/* removes the value from the given node and rearrange values */
void removeVal(struct btreeNode *myNode, int pos) {
    int i = pos + 1;
    while (i <= myNode->count) {
        myNode->val[i - 1] = myNode->val[i];
        myNode->link[i - 1] = myNode->link[i];
        i++;
    }
    myNode->count—;
}
/* shifts value from parent to right child */
void doRightShift(struct btreeNode *myNode, int pos) {
    struct btreeNode *x = myNode->link[pos];
    int j = x->count;
```

```
        while (j > 0) {
                x->val[j + 1] = x->val[j];
                x->link[j + 1] = x->link[j];
        }
        x->val[1] = myNode->val[pos];
        x->link[1] = x->link[0];
        x->count++;
        x = myNode->link[pos - 1];
        myNode->val[pos] = x->val[x->count];
        myNode->link[pos] = x->link[x->count];
        x->count—;
        return;
}
/* shifts value from parent to left child */
void doLeftShift(struct btreeNode *myNode, int pos) {
        int j = 1;
        struct btreeNode *x = myNode->link[pos - 1];
        x->count++;
        x->val[x->count] = myNode->val[pos];
        x->link[x->count] = myNode->link[pos]->link[0];
        x = myNode->link[pos];
        myNode->val[pos] = x->val[1];
        x->link[0] = x->link[1];
        x->count—;
        while (j <= x->count) {
                x->val[j] = x->val[j + 1];
                x->link[j] = x->link[j + 1];
                j++;
        }
        return;
}
```

```
/* merge nodes */
void mergeNodes(struct btreeNode *myNode, int pos) {
    int j = 1;
    struct btreeNode *x1 = myNode->link[pos], *x2 = myNode->link[pos - 1];
    x2->count++;
    x2->val[x2->count] = myNode->val[pos];
    x2->link[x2->count] = myNode->link[0];
    while (j <= x1->count) {
        x2->count++;
        x2->val[x2->count] = x1->val[j];
        x2->link[x2->count] = x1->link[j];
        j++;
    }
    j = pos;
    while (j < myNode->count) {
        myNode->val[j] = myNode->val[j + 1];
        myNode->link[j] = myNode->link[j + 1];
        j++;
    }
    myNode->count—;
    free(x1);
}
/* adjusts the given node */
void adjustNode(struct btreeNode *myNode, int pos) {
    if (!pos) {
        if (myNode->link[1]->count > MIN) {
            doLeftShift(myNode, 1);
        } else {
            mergeNodes(myNode, 1);
        }
```

```
    } else {
        if (myNode->count != pos) {
            if(myNode->link[pos - 1]->count > MIN) {
                doRightShift(myNode, pos);
            } else {
                if (myNode->link[pos + 1]->count > MIN) {
                    doLeftShift(myNode, pos + 1);
                } else {
                    mergeNodes(myNode, pos);
                }
            }
        } else {
            if (myNode->link[pos - 1]->count > MIN)
                doRightShift(myNode, pos);
            else
                mergeNodes(myNode, pos);
        }
    }
}
/* delete val from the node */
int delValFromNode(int val, struct btreeNode *myNode) {
    int pos, flag = 0;
    if (myNode) {
        if (val < myNode->val[1]) {
            pos = 0;
            flag = 0;
        } else {
            for (pos = myNode->count;
                (val < myNode->val[pos] && pos > 1); pos—);
            if (val == myNode->val[pos]) {
```

```
                    flag = 1;
                } else {
                    flag = 0;
                }
            }
            if (flag) {
                if (myNode->link[pos - 1]) {
                    copySuccessor(myNode, pos);
                    flag = delValFromNode(myNode->val[pos], myNode->link[pos]);
                    if (flag == 0) {
                        printf(―Given data is not present in B-Tree\n‖);
                    }
                } else {
                    removeVal(myNode, pos);
                }
            } else {
                flag = delValFromNode(val, myNode->link[pos]);
            }
            if (myNode->link[pos]) {
                if (myNode->link[pos]->count < MIN)
                    adjustNode(myNode, pos);
            }
        }
        return flag;
}
/* delete val from B-tree */
void deletion(int val, struct btreeNode *myNode) {
        struct btreeNode *tmp;
        if (!delValFromNode(val, myNode)) {
                printf(―Given value is not present in B-Tree\n‖);
```

```
            return;
        } else {
            if (myNode->count == 0) {
                tmp = myNode;
                myNode = myNode->link[0];
                free(tmp);
            }
        }
        root = myNode;
        return;
    }

    /* search val in B-Tree */
    void searching(int val, int *pos, struct btreeNode *myNode) {
        if (!myNode) {
            return;
        }
        if (val < myNode->val[1]) {
            *pos = 0;
        } else {
            for (*pos = myNode->count;
                (val < myNode->val[*pos] && *pos > 1); (*pos)—);
            if (val == myNode->val[*pos]) {
                printf(—Given data %d is present in B-Tree‖, val);
                return;
            }
        }
        searching(val, pos, myNode->link[*pos]);
        return;
    }
```

```
/* B-Tree Traversal */
void traversal(struct btreeNode *myNode) {
    int i;
    if (myNode) {
        for (i = 0; i < myNode->count; i++) {
            traversal(myNode->link[i]);
            printf(–%d –, myNode->val[i + 1]);
        }
        traversal(myNode->link[i]);
    }
}
int main() {
    int val, ch;
    while (1) {
        printf(—1. Insertion\t2. Deletion\n‖);
        printf(—3. Searching\t4. Traversal\n‖);
        printf(—5. Exit\nEnter your choice:‖);
        scanf(–%d‖, &ch);
        switch (ch) {
            case 1:
                printf(—Enter your input:‖);
                scanf(–%d‖, &val);
                insertion(val);
                break;
            case 2:
                printf(—Enter the element to delete:‖);
                scanf(–%d‖, &val);
                deletion(val, root);
                break;
```

```
                case 3:
                        printf(―Enter the element to search:‖);
                        scanf(-%d‖, &val);
                        searching(val, &ch, root);
                        break;
                case 4:
                        traversal(root);
                        break;
                case 5:
                        exit(0);
                default:
                        printf(―U have entered wrong option!!\n‖);
                        break;
        }
        printf(-\n‖);
    }
}
```

3.10 B+ Tree Definition:

B+ tree has one root, any number of intermediary nodes (usually one) and a leaf node. Here all leaf nodes will have the actual records stored. Intermediary nodes will have only pointers to the leaf nodes; it not has any data. Any node will have only two leaves.

The main goal of B+ tree is:

Sorted Intermediary and leaf nodes: Since it is a balanced tree, all nodes should be sorted.

Fast traversal and Quick Search:

One should be able to traverse through the nodes very fast. That means, if we have to search for any particular record, we should be able pass through the intermediary node very easily. This is achieved by sorting the pointers at intermediary nodes and the records in the leaf nodes.

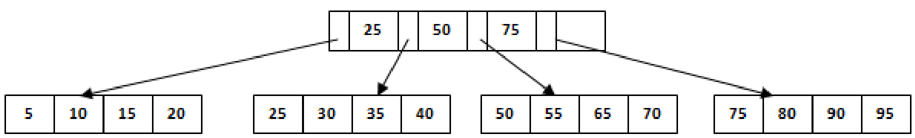
Any record should be fetched very quickly. This is made by maintaining the balance in the tree and keeping all the nodes at same distance.

No overflow pages:

B+ tree allows all the intermediary and leaf nodes to be partially filled – it will have some percentage defined while designing a B+ tree. This percentage up to which nodes are filled is called fill factor. If a node reaches the fill factor limit, then it is called overflow page. If a node is too empty then it is called underflow. In our example above, intermediary node with 108 is underflow. And leaf nodes are not partially filled, hence it is an overflow. In ideal B+ tree, it should not have overflow or underflow except root node.

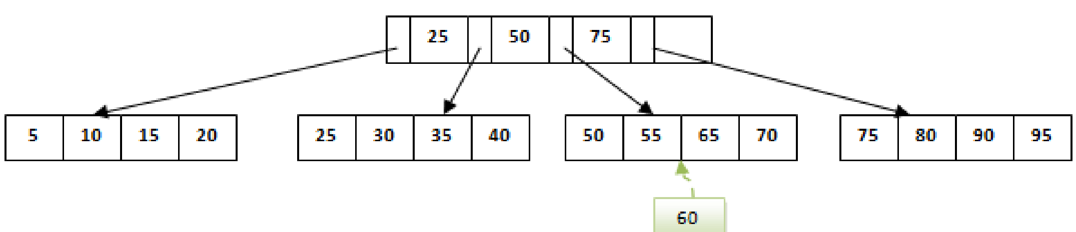
Searching a record in B+ Tree

Suppose we want to search 65 in the below B+ tree structure. First we will fetch for the intermediary node which will direct to the leaf node that can contain record for 65. So we find branch between 50 and 75 nodes in the intermediary node. Then we will be redirected to the third leaf node at the end. Here DBMS will perform sequential search to find 65. Suppose, instead of 65, we have to search for 60. What will happen in this case? We will not be able to find in the leaf node. No insertions/update/delete is allowed during the search in B+ tree.

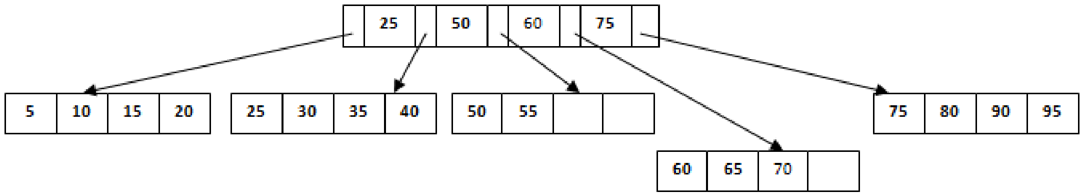


Insertion in B+ tree

Suppose we have to insert a record 60 in below structure. It will go to 3rd leaf node after 55. Since it is a balanced tree and that leaf node is already full, we cannot insert the record there. But it should be inserted there without affecting the fill factor, balance and order. So the only option here is to split the leaf node. But how do we split the nodes?



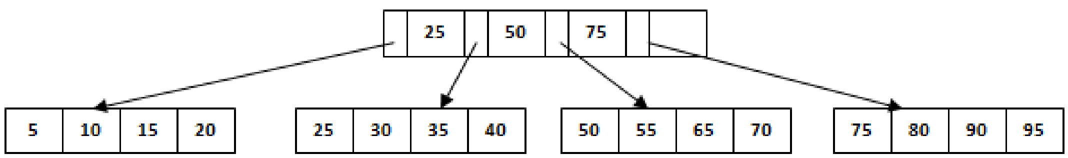
The 3rd leaf node should have values (50, 55, 60, 65, 70) and its current root node is 50. We will split the leaf node in the middle so that its balance is not altered. So we can group (50, 55) and (60, 65, 70) into 2 leaf nodes. If these two has to be leaf nodes, the intermediary node cannot branch from 50. It should have 60 added to it and then we can have pointers to new leaf node.



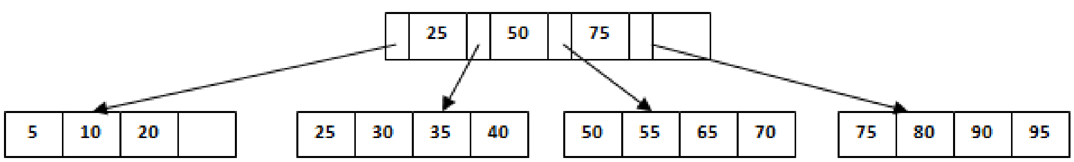
This is how we insert a new entry when there is overflow. In normal scenario, it is simple to find the node where it fits and place it in that leaf node.

Delete in B+ tree

Suppose we have to delete 60 from the above example. What will happen in this case? We have to remove 60 from 4th leaf node as well as from the intermediary node too. If we remove it from intermediary node, the tree will not satisfy B+ tree rules. So we need to modify it have a balanced tree. After deleting 60 from above B+ tree and re-arranging nodes, it will appear as below.



Suppose we have to delete 15 from above tree. We will traverse to the 1st leaf node and simply delete 15 from that node. There is no need for any re-arrangement as the tree is balanced and 15 do not appear in the intermediary node.



Program for B+ Tree

```
C++ Program to Implement B+ Tree
 */
#include<stdio.h>
#include<conio.h>
#include<iostream>
using namespace std;
struct B+TreeNode
```

```
{
   int *data;
   B+TreeNode **child_ptr;
   bool leaf;
   int n;
}*root = NULL, *np = NULL, *x = NULL;
B+TreeNode * init()
{
   int i;
   np = new B+TreeNode;
   np->data = new int[5];
   np->child_ptr = new B+TreeNode *[6];
   np->leaf = true;
   np->n = 0;
   for (i = 0; i < 6; i++)
   {
      np->child_ptr[i] = NULL;
   }
   return np;
}
void traverse(B+TreeNode *p)
{
   cout<<endl;
   int i;
   for (i = 0; i < p->n; i++)
   {
      if (p->leaf == false)
      {
         traverse(p->child_ptr[i]);
      }
```

```
        cout << — — << p->data[i];
      }
      if (p->leaf == false)
      {
        traverse(p->child_ptr[i]);
      }
      cout<<endl;
    }
    void sort(int *p, int n)
    {
      int i, j, temp;
      for (i = 0; i < n; i++)
      {
        for (j = i; j <= n; j++)
        {
          if (p[i] > p[j])
          {
            temp = p[i];
            p[i] = p[j];
            p[j] = temp;
          }
        }
      }
    }
    int split_child(B+TreeNode *x, int i)
    {
      int j, mid;
      B+TreeNode *np1, *np3, *y;
      np3 = init();
      np3->leaf = true;
```

```
if (i == -1)
{
    mid = x->data[2];
    x->data[2] = 0;
    x- >n—;
    np1 = init();
    np1->leaf = false;
    x->leaf = true;
    for (j = 3; j < 5; j++)
    {
        np3->data[j - 3] = x->data[j];
        np3->child_ptr[j - 3] = x->child_ptr[j];
        np3->n++;
        x->data[j] = 0;
        x->n—;
    }
    for(j = 0; j < 6; j++)
    {
        x->child_ptr[j] = NULL;
    }
    np1->data[0] = mid;
    np1->child_ptr[np1->n] = x;
    np1->child_ptr[np1->n + 1] = np3;
    np1->n++;
    root = np1;
}
else
{
    y = x->child_ptr[i];
    mid = y->data[2];
```

```
        y->data[2] = 0;
        y- >n—;
        for (j = 3; j < 5; j++)
        {
            np3->data[j - 3] = y->data[j];
            np3->n++;
            y->data[j] = 0;
            y->n—;
        }
        x->child_ptr[i + 1] = y;
        x->child_ptr[i + 1] = np3;
    }
    return mid;
}
void insert(int a)
{
    int i, temp;
    x = root;
    if (x == NULL)
    {
        root = init();
        x = root;
    }
    else
    {
        if (x->leaf == true && x->n == 5)
        {
            temp = split_child(x, -1);
            x = root;
            for (i = 0; i < (x->n); i++)
```

```
        {
            if ((a > x->data[i]) && (a < x->data[i + 1]))
            {
                i++;
                break;
            }
            else if (a < x->data[0])
            {
                break;
            }
            else
            {
                continue;
            }
        }
        x = x->child_ptr[i];
    }
    else
    {
        while (x->leaf == false)
        {
        for (i = 0; i < (x->n); i++)
        {
            if ((a > x->data[i]) && (a < x->data[i + 1]))
```

```
        {
            i++;
            break;
        }
        else if (a < x->data[0])
        {
            break;
        }
        else
        {
            continue;
        }
    }
        if ((x->child_ptr[i])->n == 5)
        {
            temp = split_child(x, i);
            x->data[x->n] = temp;
            x->n++;
            continue;
        }
        else
        {
            x = x->child_ptr[i];
        }
```

```
            }
        }
    }
    x->data[x->n] = a;
    sort(x->data, x->n);
    x->n++;
}
int main()
{
    int i, n, t;
    cout<<—enter the no of elements to be inserted\n‖;
    cin>>n;
    for(i = 0; i < n; i++)
    {
        cout<<—enter the element\n‖;
        cin>>t;
        insert(t);
    }
    cout<<—traversal of constructed tree\n‖;
    traverse(root);
    getch();
}
```

3.11 BINARY HEAP

The efficient way of implementing priority queue is Binary Heap. Binary heap is merely referred as Heaps, Heap have two properties namely

* Structure property
* Heap order property.

Like AVL trees, an operation on a heap can destroy one of the properties, so a heap operation must not terminate until all heap properties are in order. Both the operations require the average running time as O(log N).

Structure Property

A heap should be complete binary tree, which is a completely filled binary tree with the possible exception 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.

For example if the height is 3. Then the numer of nodes will be between 8 and 15. (ie) (2^3 and 2^4-1).

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 i/2. As it is represented as array it doesn‘t require pointers and also the operations required to traverse the tree are extremely simple and fast. But the only disadvantage is to specify the maximum heap size in advance.

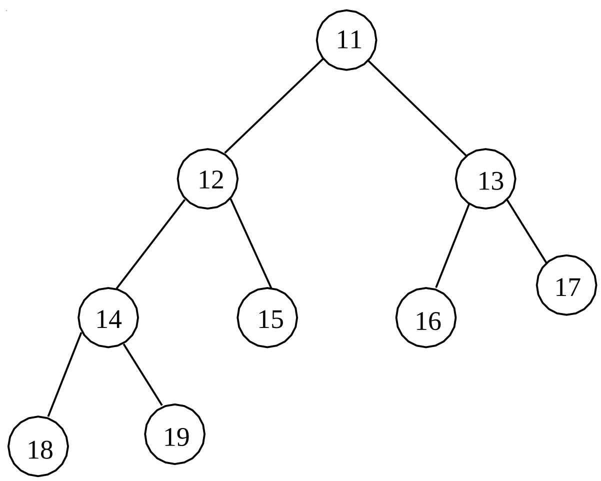


Fig. 3.11.1 A complete Binary Tree

	11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9

Fig. 3.11.2 Array implementation of complete binary tree

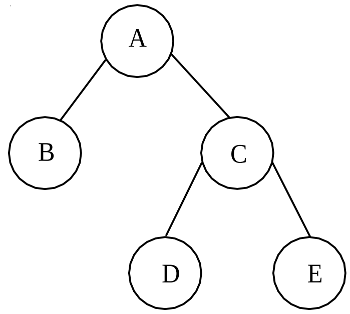


Fig. 3.11.3 Not a Complete Binary Tree

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).

This property allows the deletemin operations to be performed quickly has the minimum element can always be found at the root. Thus, we get the FindMin operation in constant time.

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).

This property allows the deletemin operations to be performed quickly has the minimum element can always be found at the root. Thus, we get the FindMin operation in constant time.

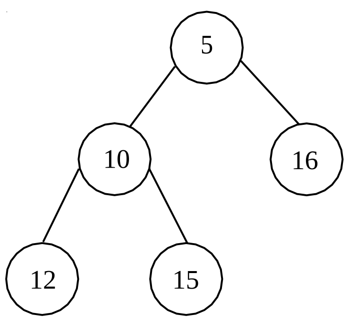


Fig. 3.11.4 (a) Binary tree with structure and heap order property.

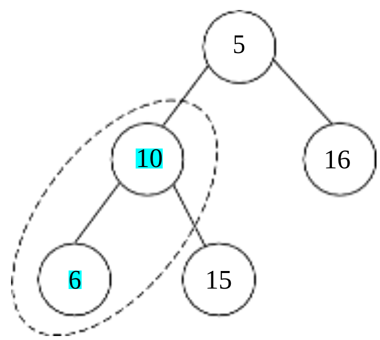


Fig. 3.11.4 (b) Binary tree with structure but violtating heap heap order property

Declaration for priority queue

```
Struct Heapstruct;
typedef struct Heapstruct * priority queue;
PriorityQueue Initialize (int MaxElements);
void insert (int X, PriorityQueue H);
int DeleteMin (PriorityQueue H);
```

```
Struct Heapstruct
{
    int capacity;
    int size;
    int *Elements;
};
```

Initialization

```
PriorityQueue Initialize (int MaxElements)
{
    PriorityQueue H;
    H = malloc (sizeof (Struct Heapstruct));
    H → Capacity = MaxElements;
    H → size = 0;
    H → elements [0] = MinData;
    return H;
}
```

Basic Heap Operations

To perform the insert and DeleteMin operations ensure that the heap order property is maintained.

Insert Operation

To insert an element X into the heap, we create a hole in the next available location, otherwise the tree will not be complete. If X can be placed in the hole without violating heap order, then place the element X there itself. Otherewise, we slide the element that is in the hole‘s parent node into the hole, thus bubbling the hole up toward the root. This process continues until X can be placed in the hole. This general strategy is known as Percolate up, in which the new element is percolated up the heap until the correct location is found.

Routine to insert into a Binary Heap

```
void insert (int X, PriorityQueue H)
{
    int i;
    If (Isfull (H))
    {
        Error (– priority queue is full‖);
        return;
    }
```

```
        for (i = ++H → size; H → Elements [i/2] > X; i/=2)
        /* If the parent value is greater than X, then place the element of parent
              node into the hole */.
              H → Elements [i] = H → Elements [i/2];
              H → elements [i] = X; // otherwise, place it in the hole.
  }
```

Example :

To Insert 10 :

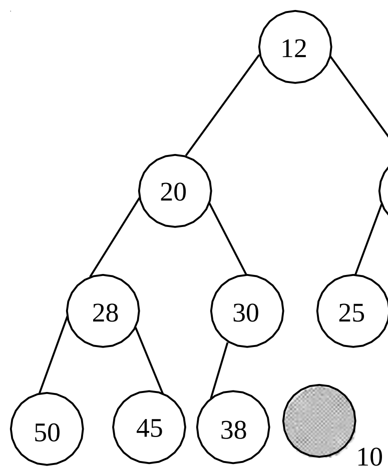


Fig. 3.11.5 (a) A hole is created at the next location

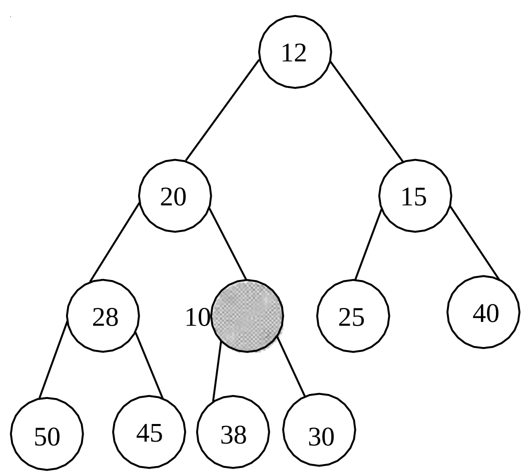


Fig. 3.11.5 (b) Percolate the hole up to satisfy heap order

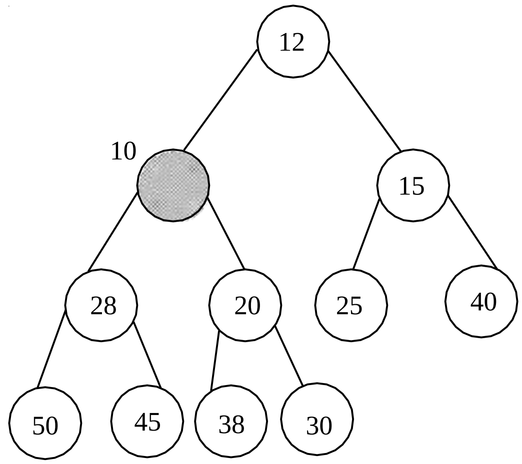


Fig. 3.11.5 (c) Percolate the hole up to satisfy heap order

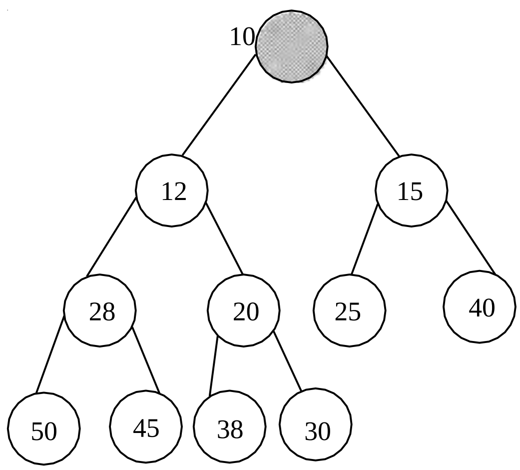


Fig. 3.11.5 (d) Percolate the hole up to satisfy heap order

In Fig 3.10.5 (d) the value 10 is placed in its correct location.

DeleteMin

DeleteMin Operation is deleting the minimum element from the Heap.

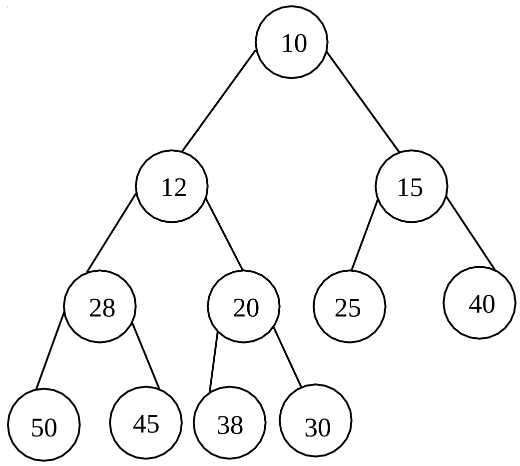
In Binary heap the minimum element is found in the root. When this minimum is removed, a hole is created at the root. Since the heap becomes one smaller, makes the last element X in the heap to move somewhere in the heap.

If X can be placed in hole without violating heaporder property place it.

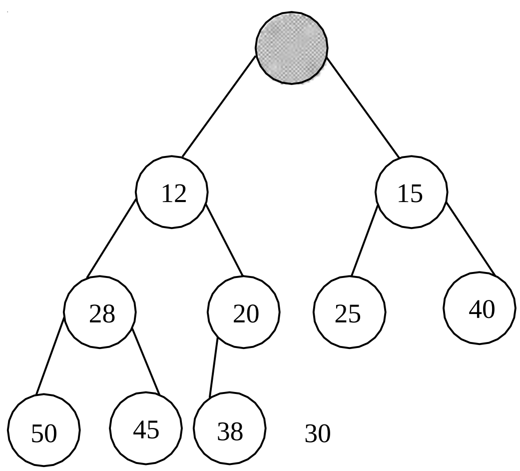
Otherwise, we slide the smaller of the hole's children into the hole, thus pushing the hole down one level.

We repeat until X can be placed in the hole. This general strategy is known as perculate down.

Example: To delete the minimum element 10

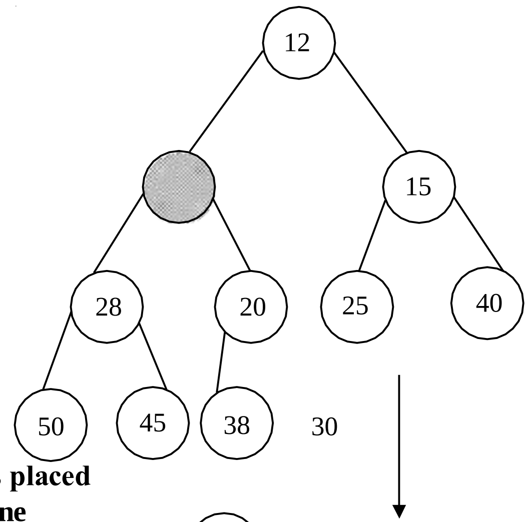


Delete Minimum element 10, creates the hole at the root.

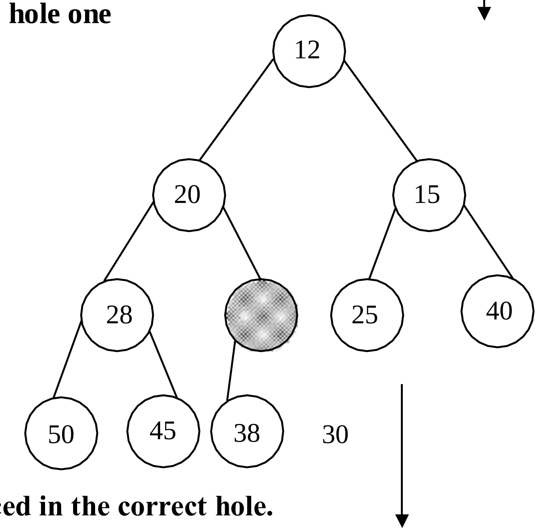


The last element '30' must be moved somewhere in the heap.

The Hole's smallest children (12) is placed into the hole by pushing the hole down one level.



The hole's smallest children (20) is placed into the hole by pushing the hole one level down.



The last element '30' is placed in the correct hole.

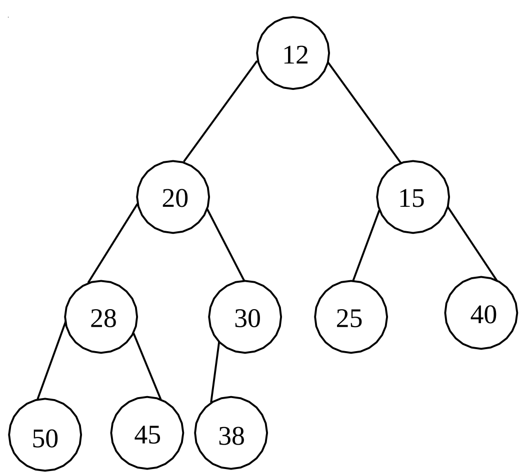


Figure. 3.11.6

Routine to Perform Deletemin in a Binary Heap

```
int Deletemin (PriorityQueue H)
{
    int i, child;
    int MinElement, LastElement;
    if (IsEmpty (H))
    {
        Error (—Priority queue is Empty‖);
        return H → Elements [0];
    }
    MinElement = H → Elements [1];
    LastElement = H → Elements [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 down
        if (LastElement > H → Elements [child])
            H → Elements [i] = H → Elements [child];
        else
            break ;
    }
    H → Elements [i] = LastElement;
    return MinElement;
}
```

Other Heap Operations

The other heap operations are

(i) Decrease - key

(ii) Increase - key

(iii) Delete

(iv) Build Heap

Decrease Key

The Decreasekey (P, Δ, H) operation decreases the value of the key at position P by a positive amount Δ. This may violate the heap order property, which can be fixed by percolate up.

example : Priority Queue H

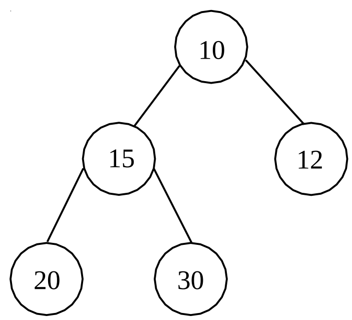
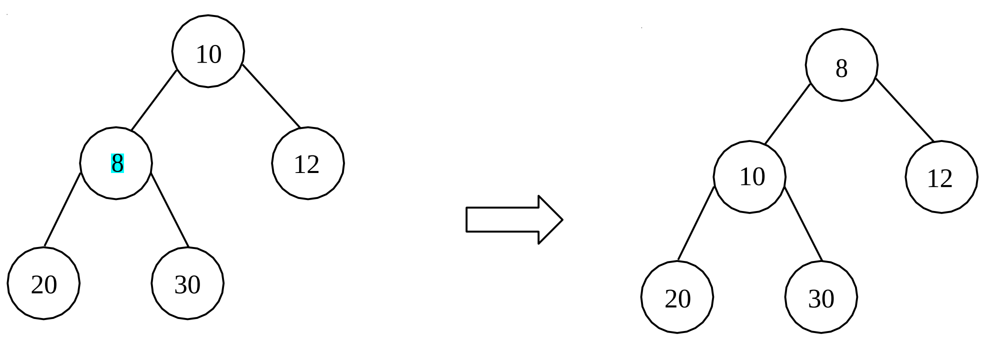


Figure. 3.11.7 (a) Decrease Key (2, 7, H)

Element at position 2 is _15‘. Decrease that element by 7. Now the position 2 has the value _8‘, which violates the heap order property.

This can be fixed by percolating up strategy.



Increase - Key

The increase - key (p, Δ , H) operation increases the value of the key at position p by a positive amount Δ . This may violate heap order property, which can be fixed by percolate down.

example :

Priority Queue H

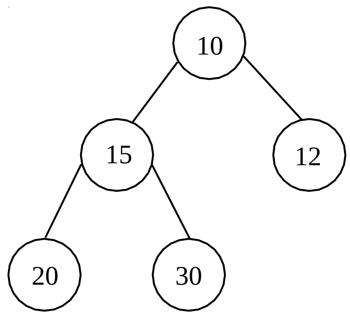
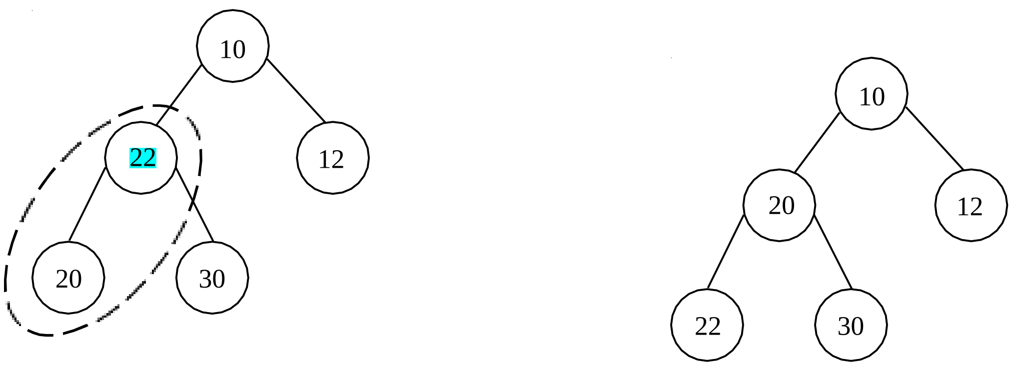


Fig. 3.11.8 (a) Increase Key (2, 7, H)

Here, the Element at position 2 is 15. Increase that value by 7. Now the position 2 has the value 22, which violates the heap order property.

This can be fixed by percolate down.



Delete :

The Delete (P, H) operation removes the node at the position P from the heap H. This can be done by.

(i) Perform the decreasekey operation

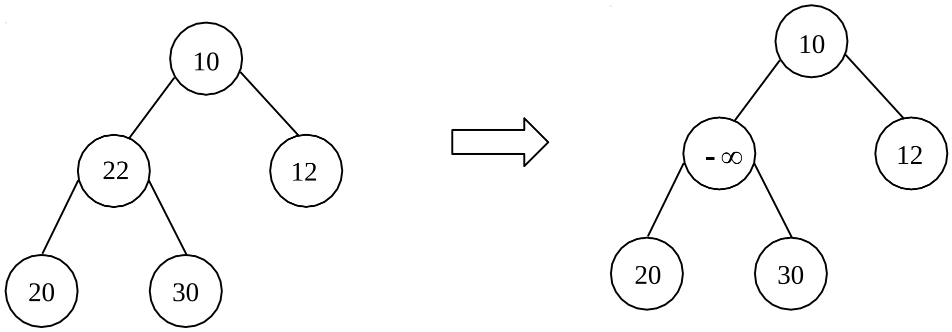
Decreasekey (P, ∞ , H)

(ii) Perform Deletemin operation

DeleteMin (H)

eg : - Delete (2, ∞ , H)

(i) Decreasing by Infinity



After Decreasing the value at position 2 by ∞ . The value changes to - ∞ , which is the least element in heap.

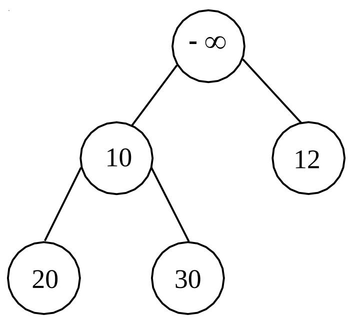
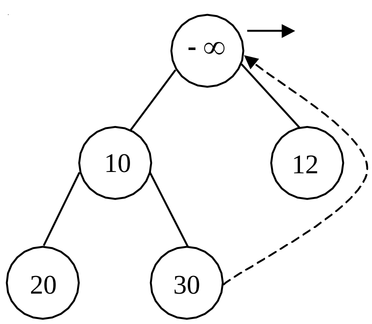


Figure. 3.11.9 Binary heap satisfying heap order property

since ‗- ∞ ‘ occupies the root position, apply DeleteMin operation.

(ii) DeleteMin

After deleting the minimum element, the last element will occupy the hole. Then will occupy the hole. Then rearrange the heap till it satisfies heap order property.



Build Heap

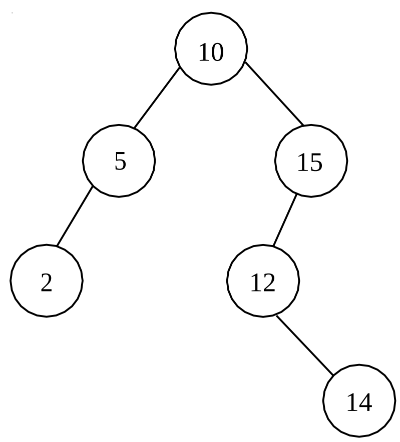
The Build Heap (H) operations takes as input N keys and places them into an empty heap by maintaining structure property and heap order property.

3.12. APPLICATIONS OF HEAP

- To quickly find the smallest and largest element from a collection of items or array.
- In the implementation of Priority queue in graph algorithms like Dijkstra's algorithm (shortest path), Prim's algorithm (minimum spanning tree) and Huffman encoding (data compression).
- In order to overcome the Worst Case Complexity of Quick Sort algorithm from O(n^2) to O(nlog(n)) in Heap Sort.
- For finding the order in statistics.
- Systems concerned with security and embedded system such as Linux Kernel uses Heap Sort because of the O(nlog(n)) .

PART - A

1. Compare General Tree and binary tree?
2. Define the following terminologies in a tree
   (1) Siblings, parent
   (2) Depth, Path
   (3) Height, Degree
3. What is complete binary tree?
4. Define Binary Search Tree.
5. Give the array and linked list representation of tree with an example.
6. Show that the maximum number of nodes in a binary tree of height H as $2^{H+1}-1$.
7. Define Tree Traversal.
8. Give the preorder form for the following Tree.



9. Write a routine to find the minimum element in a given tree.
10. Write the recursive procedure for inorder traversals.
11. Draw a binary search tree for the following input lists. 60, 25, 75, 15, 33, 44
12. How is a binary tree represented using an array?
13. Define AVL tree.
14. What are the two properties of a binary heap.
15. Define B-Tree.
16. What do you mean by self adjusting tree?
17. Write a routine to perform single rotate with left.

18. List the operations performed on priority Queue.

19. Differentiate between binary tree and Binary search tree.

20. Differentiate between general tree and binary tree.

21. What is threaded binary tree?

22. Show that the maximum number of nodes in a binary tree of height H is $2^{H+1} - 1$.

23. What is B+ tree.

PART - B

1. (a) Write an algorithm to find an element from binary search tree.

   (b) Write a program to insert and delete an element from binary search tree.

2. Write a routine to generate the AVL tree.

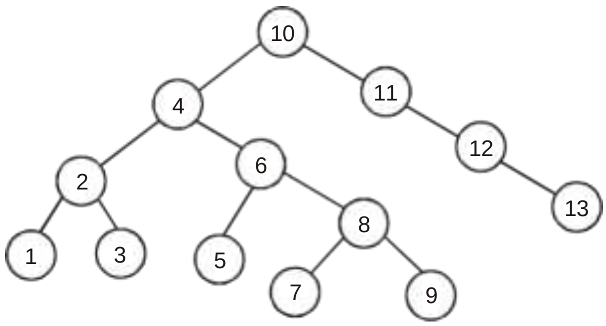
3. What are the different tree traversal techniques? Explain with examples.

4. Write a function to perform insertion and deletemin in a binary heap.

5. Write a routine to perform insertion into a B-tree.

6. Explain the operations performed on threaded binary tree in detail.

7. Show the result of accessing the keys 3,9,1,5 in order in the splay tree in the following figure.



8. Write the function to perform AVL single rotation and double rotation.

9. Construct splay tree for the

   following values:

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

Explain B+ tree in detail.