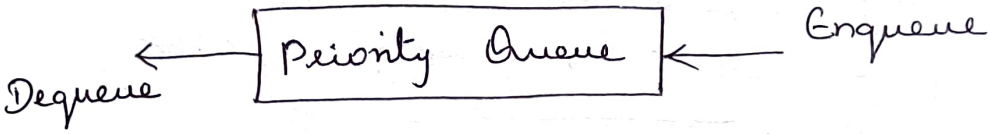
Priority Queue :

- Priority Queue are special kind of queue, where each element has a priority associated with it.



Types :

→ Ascending Priority Queue (Min Priority Queue)
→ Descending Priority Queue (Max Priority Queue)

→ In this, element is enqueued and dequeued based on its priority.
→ The elements in the priority queue have some priority.

Rules for processing elements in a Priority Queue

→ The element with highest priority is processed first.
→ The two elements with equal priority are processed based on a First come First Serve (FCFS) basis.

Array Implementation of Priority Queue.

→ In the function enqueue(), initially check if the queue is full. If it is, then print the result as "Queue Overflow"

→ If not, take the number to be inserted as input and enqueue a new element into the queue.

→ In the function, dequeue(), firstly check if the queue is empty. If it is, print the result as "Queue underflow".

→ Otherwise, remove the element and the priority from the front of the queue.

Create an empty Queue:

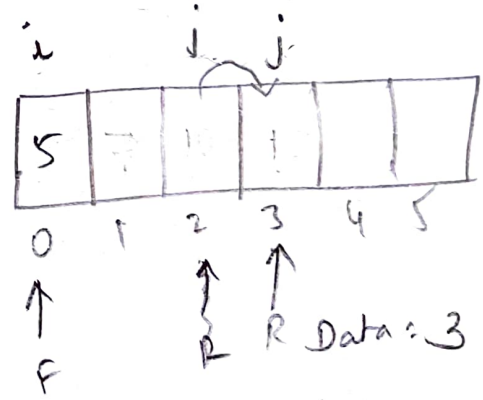
```
void create_queue()
{
    rear = front = -1;
}
```

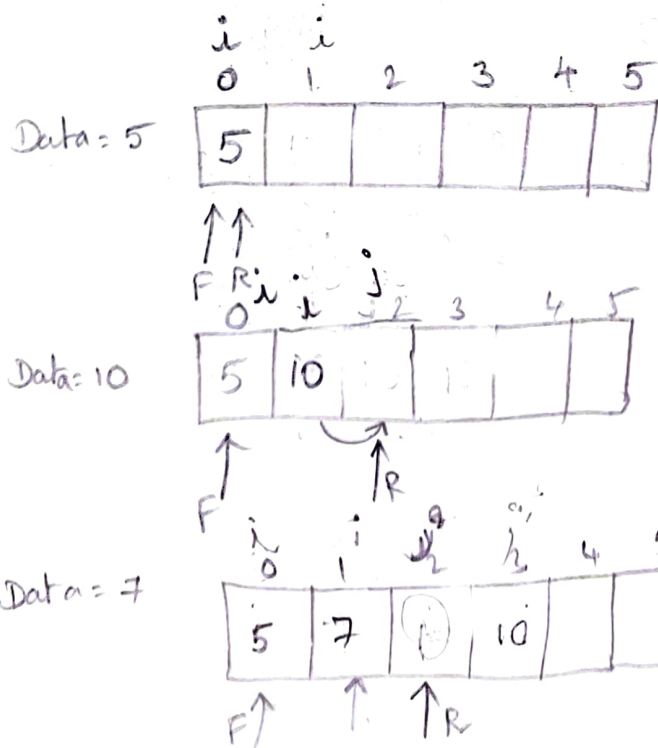
Enqueue( ):

```
Void enqueue (int data)
{
  if (rear >= Max - 1)
  {
    Printf ("\n Queue Overflow");
    return ;
  }
  if ((front == -1) && (rear == -1))
  {
    front ++;
    rear ++;
    Priority Queue
    P.queue [rear] = data;
    return ;
  }
  else
    check - priority (ddta);
  rear ++;
}
```

```
void check_priority (int data)
{
    int i, j;
    for(i=0; i<=rear; i++)
    {                    3 >= 5
      if(data >= pqueue[i])   // if condition is False, increment i. (i++)
      {
        for (j= rear+1; j > i; j--)
        {
          pqueue[j] = pqueue[j-1];
        }
        pqueue[i] = data;
        return;
      }
    }
    pqueue[i] = data;
}
```

3 >= 5 ✓





10 >= 5
F

7

7 >= 5 X

7 >= 10 ✓

7 >= 5
F

7 >= 10 ✓

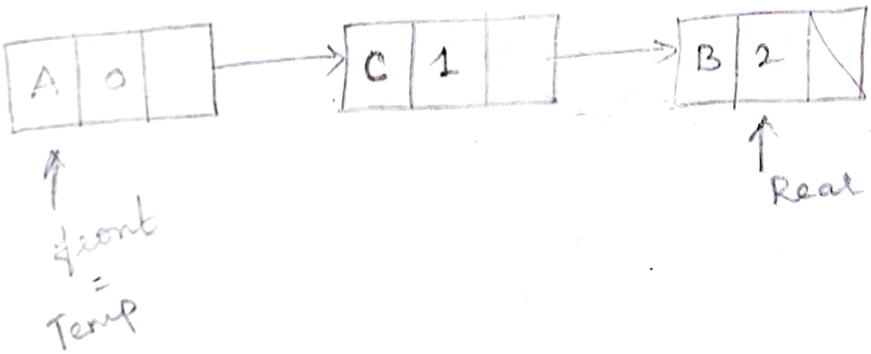
Linked List Implementation of Priority Queue

```
struct node
{
  int data;
  int priority;
  struct node * next;
};
```

Enqueue ( )

```
void enqueue (int element, int priority)
{
 struct node * newnode;
 newnode -> data = element;
 newnode -> priority = priority;
 newnode -> next = 0;
}
 if (front == 0 && rear == 0)
   {
    front = rear = newnode;
   }
 else if (priority < front -> priority)
   {
     newnode -> next = front;
     front = newnode;
   }
```

```
else    // Traverse and find the position of the
        new node
{
  temp = front ;
 while (temp -> next != NULL && temp -> next -> priority <=
                                                  priority)
 {
  temp = temp -> next ;
 }
  newnode -> next = temp -> next ;  // Enqueue at the middle
  temp -> next = newnode ;
 }
```



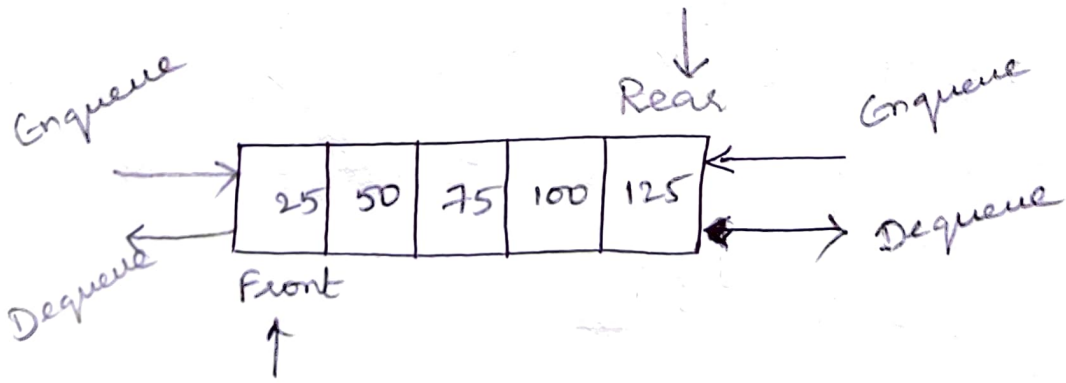
Dequeue :

→ A Double Ended Queue is also known as Deque, is a queue data structure in which insertion and deletion can be done from both left and right ends.

→ Deque is divided into two types.

Input Restricted Deque : In this, input is blocked at a single end but allows deletion at both the ends.

Output Restricted Deque : In this, output is blocked at a single end but allows insertion at both the ends.



Application of Queue ADT :

→ Managing requests on a single shared resource such as CPU scheduling and disk scheduling.

→ Handling hardware (or) real-time systems interrupts.

→ Handling website traffic.

→ Routers and switches in networking.

→ Maintaining the playlist in media players (MP3)

→ Spooling in printers.

→ Buffer for devices like keyboard.

→ Mail Queues.

→ dequeue, DEqueue, Priority Queue implementation

→ Whatsapp when we send messages to our friends and they don't have an internet then these messages are queued on the server of whatsapp.

Real-time Examples of Queue.

→ A queue of people at ticket window.

→ Vehicles on toll-tax bridge.

→ Phone answering system.

→ Luggage checking machine.

→ Patients waiting outside the doctor's clinic.

→ People on an escalator.

→ cashier line in a store.

→ Car wash line.

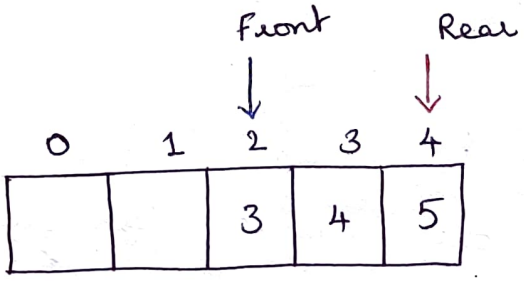
→ One way exits

Circular Queue :

- A circular queue is the extended version of a normal linear queue where the last element is connected to the first element.

- The circular queue solves the major limitation of the linear queue.

- In a linear queue, after insertion and deletion, there will be non-usable empty space.



- Here, indexes 0 and 1 can only be used after resetting the queue (deletion of all elements). This reduces the actual size of the queue.

How Circular Queue Works :

- Circular Queue works by the process of circular increment (i.e., when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.

- Here, the circular increment is performed by modulo division with the queue size.

(i.e.), Rear = (Rear + 1) % 5 = 0 (Start of Queue).

Circular Queue Operations :

- Two pointers Front and Rear.
- Front track the first element of the queue.
- Rear track the last elements of the queue.
- Initially, set value of Front and Rear to -1.

Enqueue Operation :

- Check if the queue is full.
- For the first element, set value of Front to 0.
- Circularly increase the Rear index by 1 (ie)., if the rear reaches the end, next it would be at the start of the queue.
- Add the new element in the position pointed to by Rear.

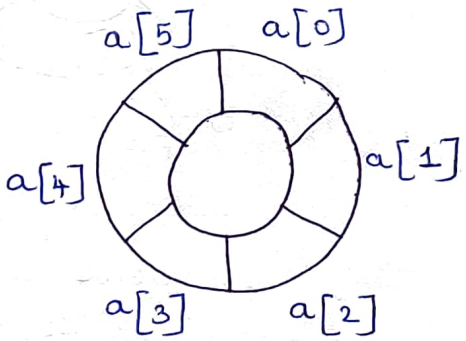
Dequeue Operation :

- Check if the queue is empty.
- Return the value pointed by Front.
- Circularly increase the Front index by 1.
- For the last element, reset the values of Front and Rear to -1.

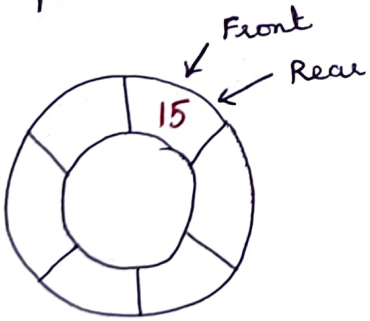
Circular Queue Implementation using Arrays :

Size = 6

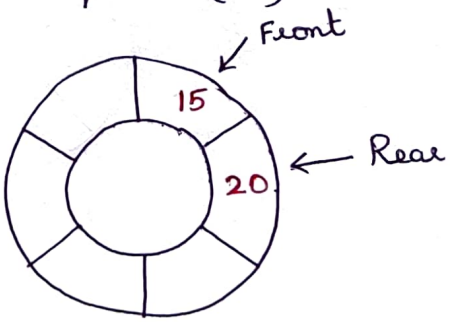
Front, rear = -1



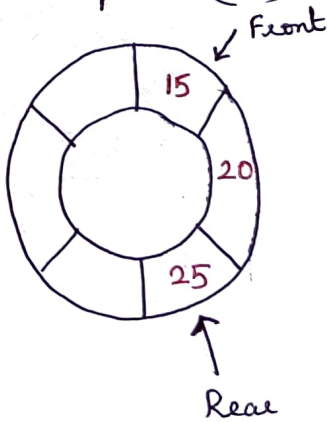
Enqueue (15)



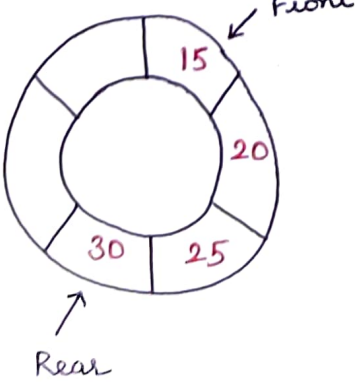
Enqueue (20)



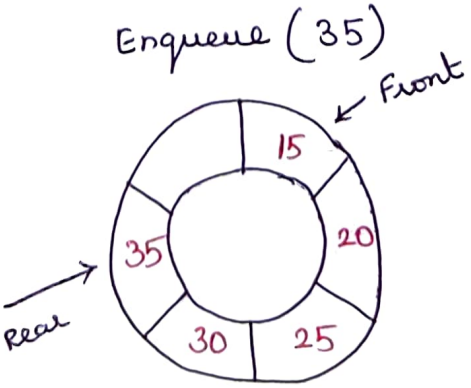
Enqueue (25)



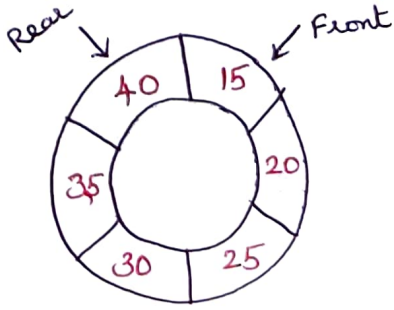
Enqueue (30)



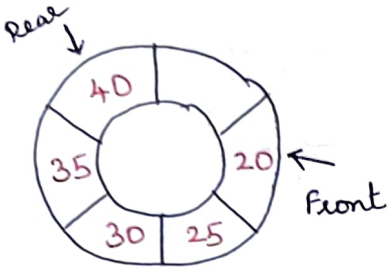
Enqueue (35)



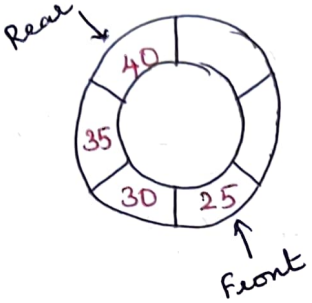
Enqueue (40)



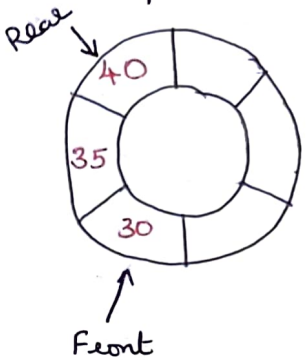
Dequeue ( )



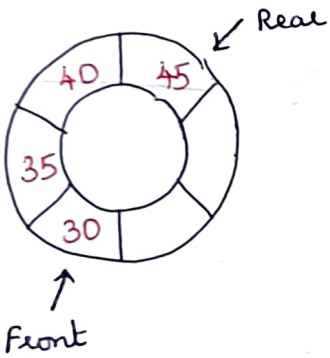
Dequeue ( )



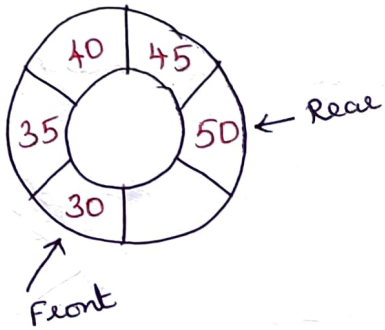
Dequeue ( )



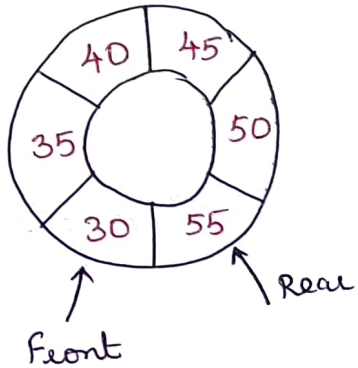
Enqueue (45)



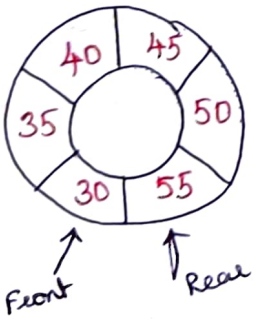
Enqueue (50)



Enqueue (55)



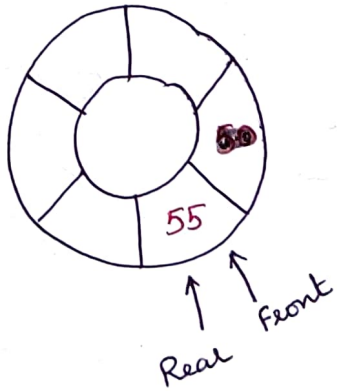
Enqueue (60)



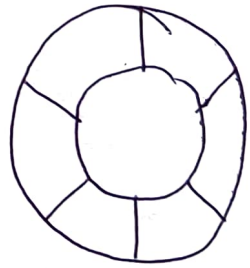
Can't Enqueue.

Overflow!

Dequeue - 5 items



Dequeue ( )



Front = Rear = -1

Can't Dequeue.

Underflow!

```
#define SIZE 6

int queue[SIZE];
int front = -1, rear = -1;

// Check whether the Circular Queue is full or not.

int checkFull()
{
  if ((front == rear + 1) || (front == 0 &&
                              rear == SIZE - 1))
  {
    return 1;
  }
  return 0;
}

// Check whether the C. Queue is empty or not.

int checkEmpty()
{
  if (front == -1)
  {
    return 1;
  }
  return 0;
}
```

// Enqueue in the Circular Queue

```
Void enqueue (int value)
{
  if (checkFull ())
   { Printf (" Overflow condition \n");
   }
  else
    {
      if (front == -1)
      { front = 0 ; }

      rear = (rear + 1) % SIZE ;
      queue [rear] = value;
      Printf (" Enqueued %d ", value) ;
    }
  }
```

// Dequeue from the Circular Queue

```
int dequeue ()
{
  int val ;
  if (checkEmpty ())
  {
    Printf (" Underflow condition ");
    return -1 ;
  }
```

```
else
{
    val = queue[front];
    if (front == rear)          // If queue has single element
    {
        front = rear = -1;
    }
    else
    {
        front = (front + 1) % SIZE;
    }
    printf("Dequeued %d", val);
    return 1;
  }
}

// Display the Queue.

void Print()
{
  int i;
  if (checkEmpty())
  {
    printf("Nothing to dequeue");
  }
```

```
else
  {
   Printf("The queue looks like : \n");
  for(i = front ; i != rear ; i = (i+1) % SIZE)
   {
     Printf ("%d ", queue[i]);
   }
   Printf("%d \n", queue[rear]);
   }
  }
```

Linked list Implementation of Circular Queue.

- Using Circular Queue is better than normal queue because there is no memory wastage.

- Linked list provide the facility of dynamic memory allocation, so it is easy to create.

- When we implement circular queue using linked, it is similar to circular linked list except there is two pointers front and rear in circular queue where as circular linked list has only one pointer head.

Enqueue (data) :

- Create a struct node type node.
- Insert the given data in the new node data section and NULL in address section.
- If Queue is empty then initialize front and rear from new node.
- Queue is not empty then initialize rear next and rear from new node.
- New node next initialize from front.

dequeue ( ) :

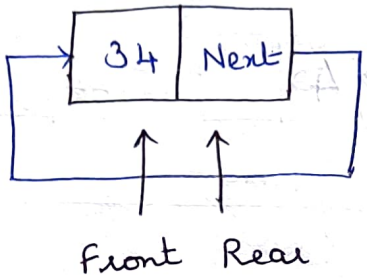
- Check if queue is empty or not.
- If queue is empty then dequeue is not possible.
- Else initialize temp from front.
- If front is equal to the rear then initialize front and rear from null.
- Print data of temp and free temp memory.
- If there is more than one node in Queue then make front next to front then initialize rear next from front.
- Print temp and free temp.

Print ( ) :

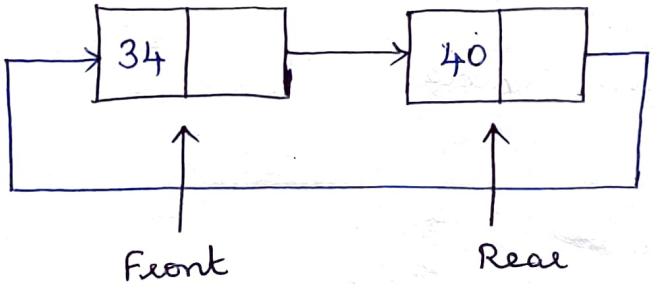
- Check if there is some data in the queue or not.
- If the queue is empty print "No data in the queue".
- Else define a node pointer and initialize it with front.
- Print data of node pointer until the next of node pointer becomes NULL.

Adding the elements into Queue.

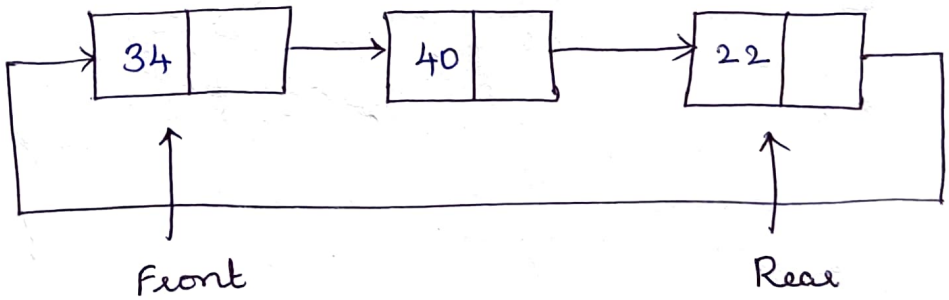
Enqueue (34)



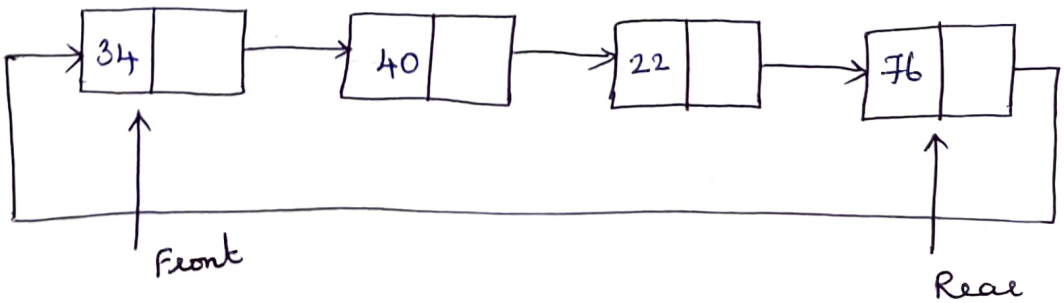
Enqueue (40)



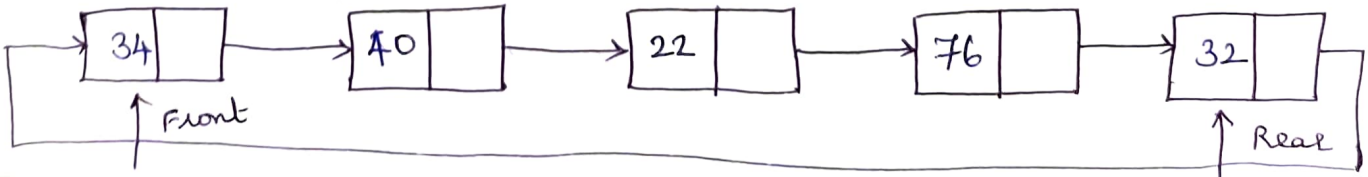
Enqueue (22)



Enqueue (76)

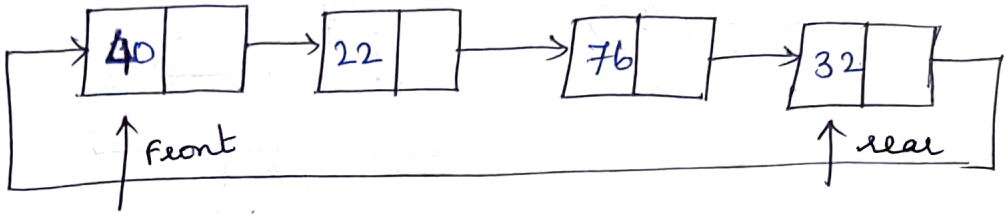


Enqueue (32)



Removing the elements from Queue.

dequeue ()



Node Structure :

```
struct node
{
  int data ;
  struct node * next ;
};

struct node * front = NULL ;
struct node * rear = NULL ;
void enqueue (int data)  // Insert elements in Queue
{
  struct node * newnode ;
  newnode = (struct node *) malloc (sizeof (struct node));
  newnode -> data = data ;
  newnode -> next = NULL ;
```

```
if((rear == NULL) && (front == NULL))
{
    front = rear = newnode;
    rear -> next = front;
}
else
  {
    rear -> next = newnode;
    rear = newnode;
    newnode -> next = front;
  }
}
```

Dequeue ():

```
void dequeue()  // Delete an element from Queue
{
    struct node * temp;
    temp = front;
   if((front == NULL) && (rear == NULL))
    {
     Printf("Queue is Empty");
    }
```

```
else if (front == rear)
  {
    front = rear = NULL;
    free(temp);
  } else { front = front -> next;
           rear -> next = front;
           free(temp);
         } }
```

Display ( ) :

```
void Print()  // Print the elements of Queue
{
  struct node* temp;
  temp = front;
  if ((front == NULL) && (rear == NULL));
    {
      Printf("Queue is Empty");
    }
  else
    {
      do
        {
          Printf("\n %d", temp -> data);
          temp = temp -> next;
        } while (temp != front);
    } }
```

UNIT - III

Non - Linear Data Structures

TREE STRUCTURES

Tree :

- Tree defined as a collection of nodes / elements which

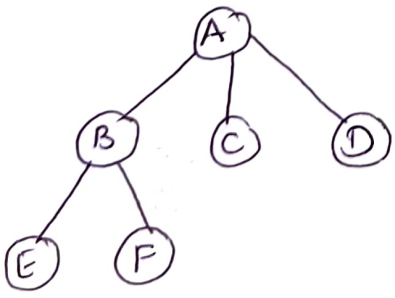
(i). can be empty

(ii). has a node designated as root from which zero (or) more subtrees / branch.

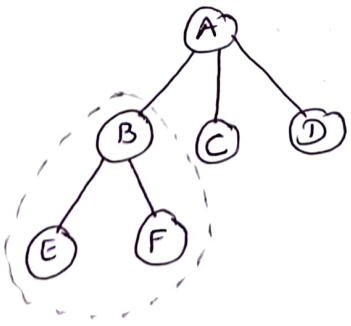
Subtree :

- A Subtree is a tree on its own. Each subtree is disjoint (A node cannot be a part of two subtrees.

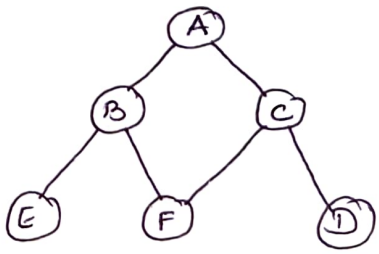
Tree



Subtree



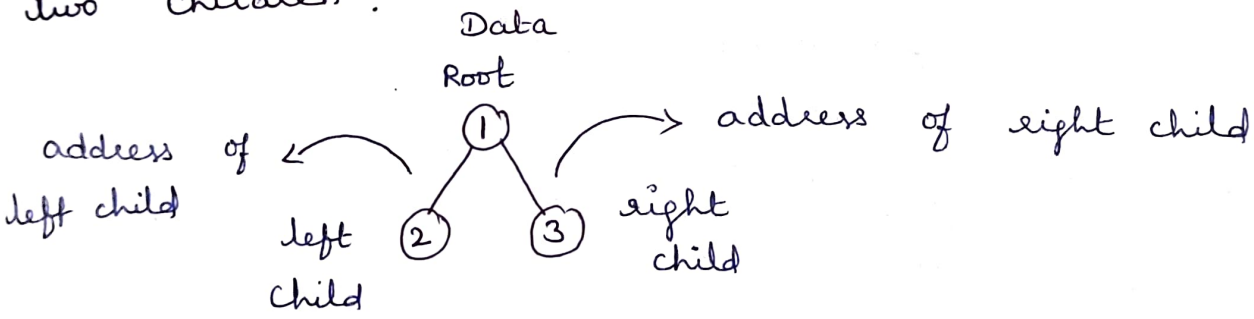
A node cannot be part of 2 subtrees



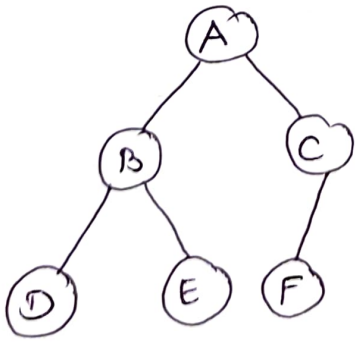
Binary Tree :

- A binary tree is a finite ordered collection of elements in which one element is designated as root and remaining elements are partitioned into two disjoint sets, called left and right subtrees.

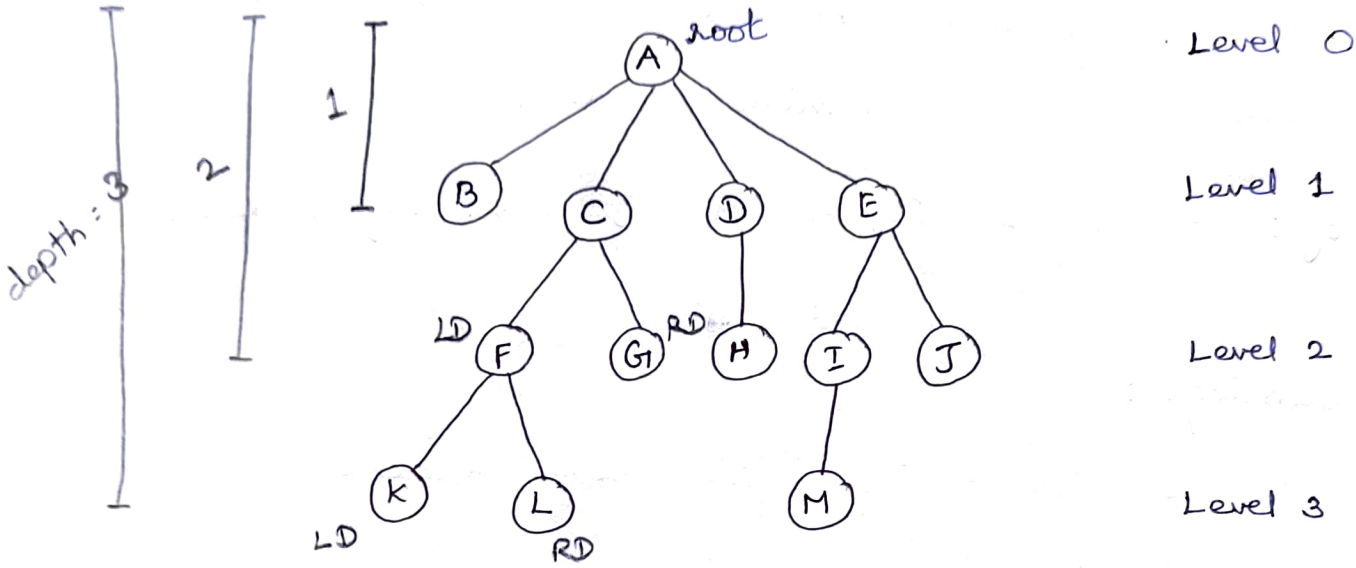
- It is a non-linear data structure in which each parent node can have at most two children.



Left and Right Subtrees :



Terminologies & Representation :



Root :

- A node which has the top hierarchy of the tree, which doesn't have a parent.
- A tree has only one root. Eg : A in the above diagram.

Node :

- A node is a data structure that stores a value that can be of any data type and has a pointer to another node.

Eg : B, C, D, ... (in the above diagram)

Parent Node :

- A node can have any no. of children.
- A node node one level up in the hierarchy of the tree with which it is connected.

Eg : Parent of nodes B, C, D and E is A
K and L is F

Leaf Node / Terminal Node :

- A node that doesn't have children is called leaf (or) terminal node.

Eg : B, K, L, G, H, M, J → leaf nodes.

Siblings :

- Children of the same parents are called siblings.

Eg : B, C, D, E are siblings

F, G are siblings

I, J are siblings.

Interior Nodes :

- Nodes other than leaves.

Eg : C, D, E.

Ancestor Node :

- A node (n_1) is ancestor of another node (n_2), if n_2 is the child of n_1 (or) to any of its children.

Eg : A, C, F are ancestors of K.

E is ancestor of J.

Descendant Node :

- A node (n_1) is descendant to node (n_2), if n_1 is the child of n_2 (or) to any of its children.

Eg : ~~A node (n_1) is descendant to~~

C and G are descendant of A.

I and M are descendant of E.

Left Descendant Node :

- A node (n_1) is left descendant to node (n_2) if n_1 is the left child of n_2, (or) to any of its left descendants.

Eg : B is L.D of A.

F, K are L.D of C.

I, M are L.D of E

H is L.D of D.

Right Descendant Node :

- A node n_1 is right descendant to n_2 if n_1 is the right child of n_2 (or) to any of its right descendants.

Eg : L is RD of F

E, J are RD of A

G is RD of C.

Path :

- Nodes that follow a branch of a tree, representing a linear subset of the tree.
- It is the path from one node to another.

Eg : A - C - F - L → Path to L.

A - E - I - M → Path to M.

A - D - H → Path to H.

A - B → Path to B.

Length of a Path :

- Number of edges in a path.

Eg : Length of A - C - F - L is 3

Length of A - E - I - M is 3

Length of A - D - H is 2

Length of A - B is 1

Level of a node :

- Root is at level 0.
- The level of any node is one plus the level of its parent. (or)

- The level of a node is the number of edges along the unique path between it and the root node.
- Therefore, the root node has a level of 0.
- If it has children, both of them have a level of 1.

Eg: A → 0 (root node)

B, C, D, E → 1

F, G, H, I, J → 2

K, L, M → 3

Depth of the Tree:

- Maximum ~~level~~ length of any leaf in the tree.

Eg: Depth of the above tree is 3.

Degree:

- The number of subtrees of a node is called its degree.

Eg: Degree (A) → 4 Degree (D) → 1

Degree (C) → 2 Degree (H) → 0

- Degree of the tree is the maximum degree of any node in the tree.

Degree (Tree) → 4.

Height :

- The height of the node n is the length of the longest path from n to leaf.

Height (F) → 1

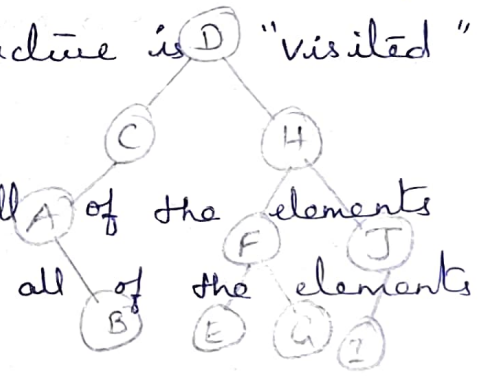
Height (L) → 0

Height (C) → 2

TREE TRAVERSAL : (Binary Tree ADT)

- It is the process in which each and every element present in a data structure is "visited" (or accessed) at least once.

- This may be done to display all of the elements (or) to perform an operation on all of the elements.



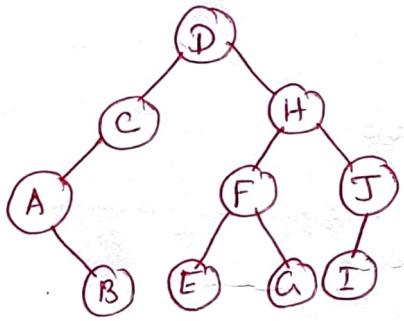
Types of Tree Traversals :

* In-Order Traversal
* Pre-Order Traversal
* Post-Order Traversal

In-Order Traversal

Routine for Inorder Traversal

```
void Inorder (Tree T)
{
  if (T != NULL)
  {
    Inorder (T -> Left);
    PrintElement (T -> Element);
    Inorder (T -> Right);
  }
}
```



Inorder : L (R) R

A B C D E F G H I J

Algorithm : Left (Root) Right

1). Traverse the Left-subtree. (ie)., call inorder (left-subtree). (Traverse till left-most node is reached).

2). Visit the root.

3). Traverse the right subtree. (ie)., call inorder (right subtree).

Pre-order Traversal : (R) L R

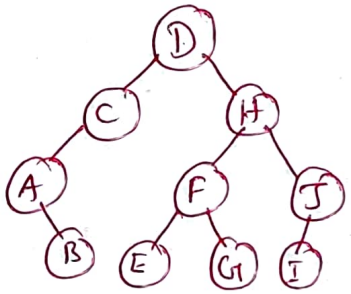
```
Void  Preorder (Tree T)
{
  if (T != NULL)
  {
    Printf ("%d", T → Element);
    Preorder (T → Left);
    Preorder (T → Right);
  }
}
```

Algorithm :

1). Visit the root.

2). Traverse the left subtree. (ie). call Preorder (left-subtree).

3). Traverse the right subtree (ie)., call Preorder (right subtree).



Preorder : (R) L R : D C A B H F E G J I

Post-order Traversal : L R (R)

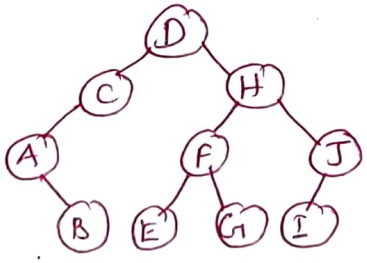
```
Void Postorder (Tree T)
{
  if (T != Null)
  {
    Postorder (T -> Left);
    Postorder (T -> Right);
    Printf ("%d", T -> Element);
  }
}
```

Algorithm :

1). Traverse the left subtree (i.e., call Postorder. (left - subtree).

2). Traverse the right subtree (i.e., call Postorder. (right - subtree).

3). Visit the root.

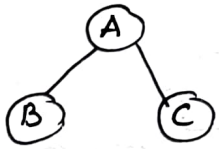


Post Order : L R (R) : B A C E G F I J H D

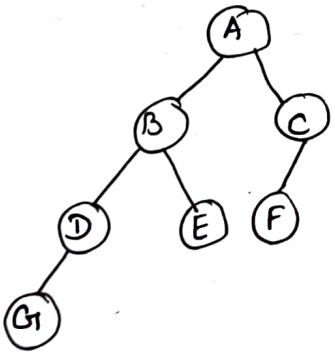
Binary Tree ADT :

- A binary tree is a finite ordered collection of elements in which one element is designated as root and remaining elements are partitioned into two disjoint sets called left and right subtrees.

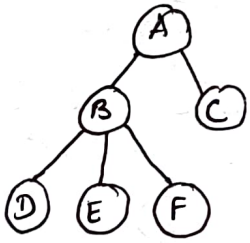
- A binary tree has not more than two children (or) Each parent node in a binary tree can have at-most two children.

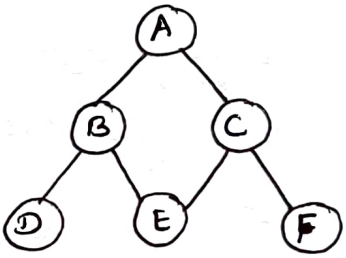


Binary Tree



Non-Binary Tree





- Each node of a binary tree consists of three items.

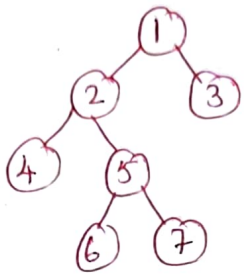
* data item
* address of left child
* address of right child

Types of Binary Trees :

1). Strict / Full binary tree :

- A full binary tree is a special type of binary tree in which every parent node / internal node has either two or no children.

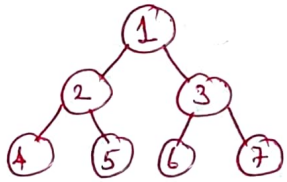
Eg :



2). Perfect Binary Tree :

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

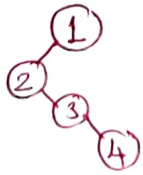
Eg :



3. Degenerate or Pathological tree :

- A degenerate tree is the tree having a single child either left or right.

Eg :



4. Complete binary Tree / Almost complete binary Tree :

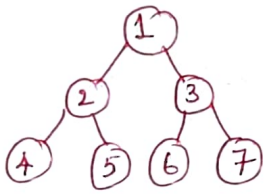
- A complete binary tree is a full binary tree but with two major differences.

1). Every level must be completely filled.

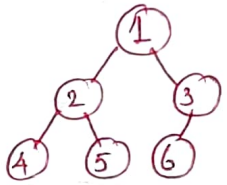
2). All the leaf elements must lean towards the left.

3). The last leaf element might not have a right sibling (ie)., a complete binary tree doesn't have to be a full binary tree.

Eg:



Complete binary Tree



Almost complete binary tree

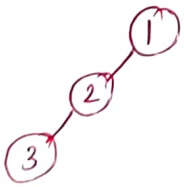
5. Skewed Binary tree :

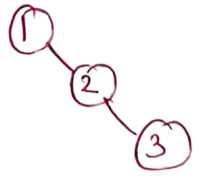
- A skewed binary tree is a degenerate tree in which the tree is either dominated by the left nodes (or) the right nodes.

- There two types of skewed binary tree.

* Left skewed binary tree
* Right skewed binary tree.

Eg:

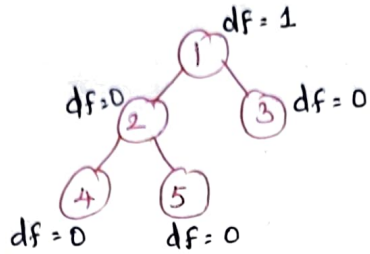




6. Balanced Binary Tree :

- It is a type of binary tree in which the difference between the height of the left and the right subtree for each node is either 0 (or) 1.

eg :



Applications of Binary Tree :

* for easy and quick access to data.
* In router algorithms.
* To implement heap data structure.
* Syntax tree.

Expression trees :

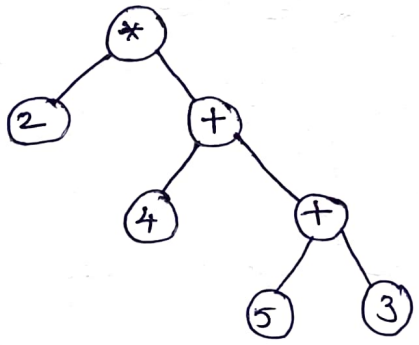
- Expression trees are binary trees in which the leaf nodes are operands and the interior nodes are operators.

- It can be traversed by,

  → Inorder
  → Preorder
  → Postorder

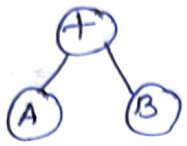
- These trees are referred to as heterogeneous trees.

Eg: (2 * (4 + (5+3)))



- In this expression tree, the parent nodes are the operators and the children are the operands.

① Expression A + B



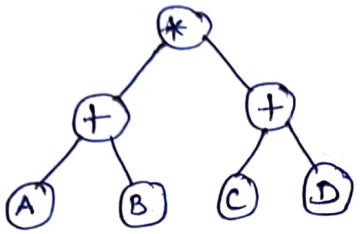
Traversing the expression tree.

Inorder : A + B

Preorder : + AB

Postorder : AB+

② (A+B) * (C+D)



Inorder : A + B * C + D

Preorder : * + AB + CD

Postorder : AB + CD + *

③ (A+B+C) * (D+E+F)

④ (A > B) && (C == D)

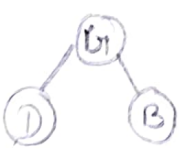
⑤ A - B / (C ^ D) + (E * F)

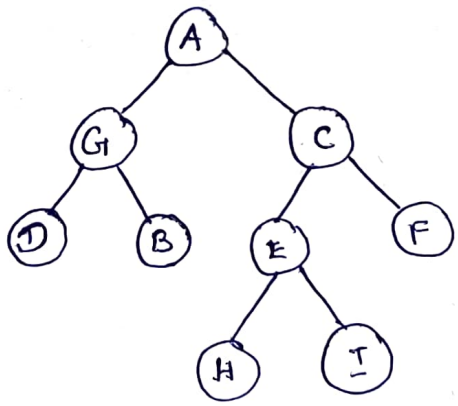
Convert the sequence into tree Inorder.

L R A R R

D G B A H E I C F — Inorder L R R

L R R





D G B A H E I C F

Construction of an Expression tree.

Consider a postfix expression a b + c *

1). Read one symbol at a time from the postfix expression.

2). Check whether the symbol is an operand (or) operator,

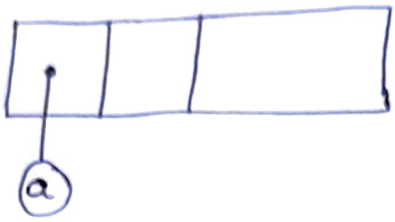
(i). If the symbol is an operand, create one-node tree and push a pointer on to the stack.

(ii). 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, T_2 as a left child and T_1 as a right child.

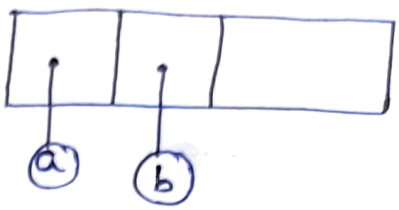
(iii) A pointer to this new tree is then pushed onto the stack.

Eg: a b + c *

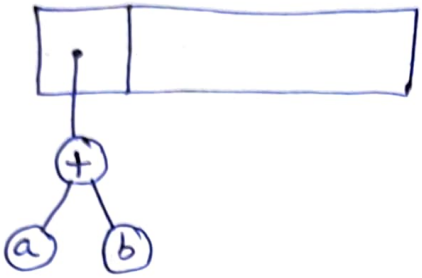
(i)



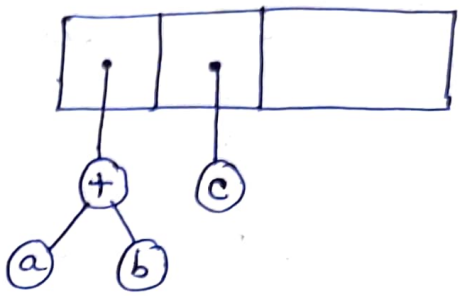
(ii)



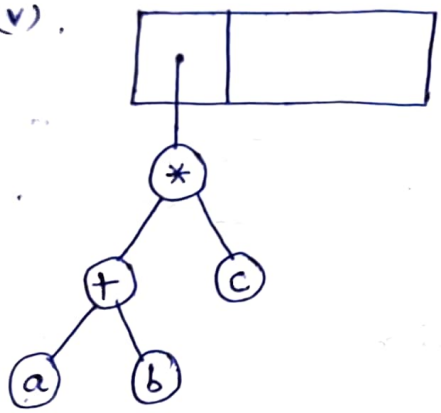
(iii)



(iv)



(v)



* / -

+ - -

Prefix :

$a \overset{1}{*} b \overset{2}{/} c \overset{6}{+} e \overset{3}{/} f \overset{4}{*} g \overset{7}{+} k \overset{8}{-} x \overset{5}{*} y$

Postfix :

a b * c / e f / g * + k + x y * -

(1) A binary tree has 8 nodes. The Inorder and post order traversal of the tree are given below. Draw the tree and find preorder.

post order : F E C H G D B A

Inorder : F C E A B H D G

(Answer → Next Page)

a + b / c + e / f * g + k − x * y − Infix expression

↓

Post fix expression

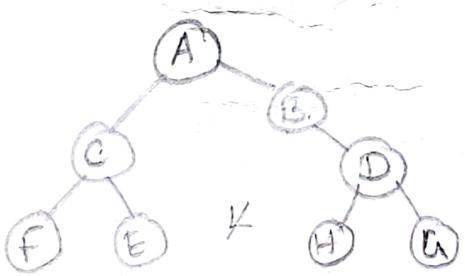
ab * c / ef / g * + k + xy * −

L X R

① Post order : F/L E/R C/R H/L G/R D/R B/R A/R

Inorder : F/L C/R E/R A/R B H/L D/R G/R

Draw the Tree & find pre order.



Inorder : F C E A B H D G

Post order : F E C H G D B A

Pre order : A/R C/R F/L E/R B D/R H/L G/R

R R L R

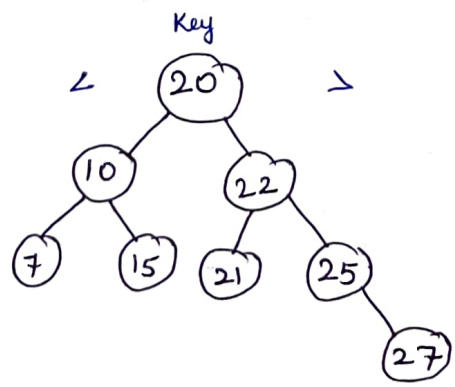
Binary Search Tree ADT

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

Operations :

→ Node Declarations
→ Make an empty tree
→ Insertion
→ Find
→ Find Min
→ Find Max
→ Delete
  → Node with no children = leaf node
  → Node with one child
  → Node with two children

Eg:



Node Structure

```
Struct BSTnode
{
  int element;
Struct BSTnode * left;
Struct BSTnode * right;
};
```

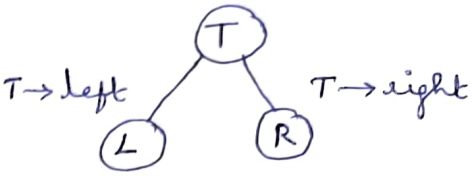
Declaration routine for BST :

```
Struct BSTnode;
typedef struct BSTnode * SearchTree;
SearchTree Insert (int x, SearchTree T);
SearchTree Delete (int x, SearchTree T);
int Find (int x, SearchTree T);
int FindMin (SearchTree T);
int FindMax (SearchTree T);
SearchTree MakeEmpty (SearchTree T);
```

```
SearchTree Insert (int X , SearchTree T)
{
  if (T == NULL)
  {
  T = malloc (sizeof (struct BSTnode));
  if (T != NULL)      // one element .
  {
     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);
   return T ;
}
```

MakeEmpty :

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



Insert an element X into the tree.

1). Check with the root node T.

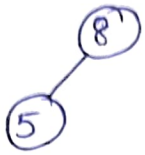
2). If it is less than root, Traverse the left subtree recursively until it reaches the T→left equals to NULL. Then X is Placed in T→left.

3). If it is 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.

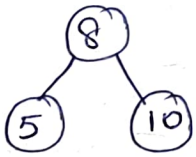
Eg: 8, 5, 10, 15, 20, 18, 3

→ root node

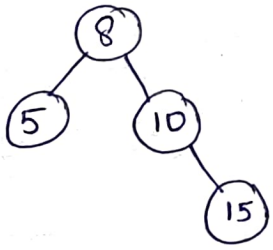
Insert 5, 5 is less than the root element, so insert at the left.



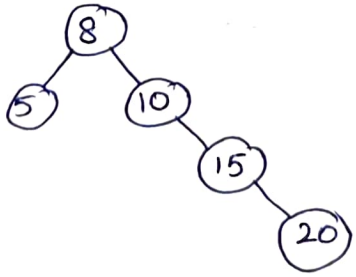
insert 10, 10 is greater than the root element. so insert at the right.



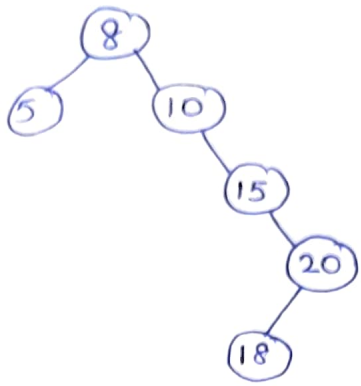
insert 15, 15 > 8, 15 > 10.



insert 20, 20 > 8, 20 > 10, 20 > 15



insert 18 , 18 > 8 , 18 > 10 , 18 > 15 , 18 < 20



Find :

1. Check whether the root is NULL if so then return NULL.
2. Otherwise, check the value X with the root node value (T → data).

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

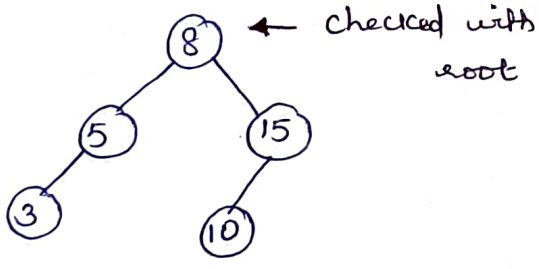
   (ii). If X is less than T → data, traverse the left of T recursively.

   (iii). If X > T → data, traverse the right of T recursively.

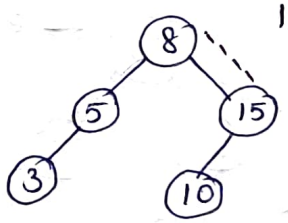
Routine :

```
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 ;
}
```

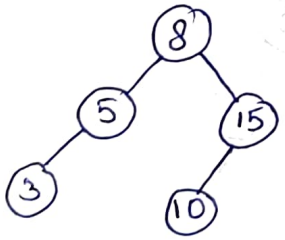
Eg: To find an element 10.



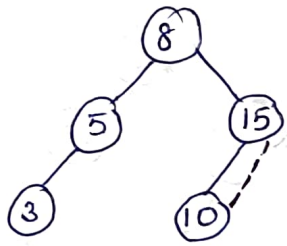
← checked with root



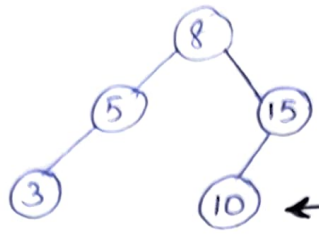
10 > 8
Go to RC (Right child)



← 10 is checked with root, 15



10 < 15
Go to LC of 15

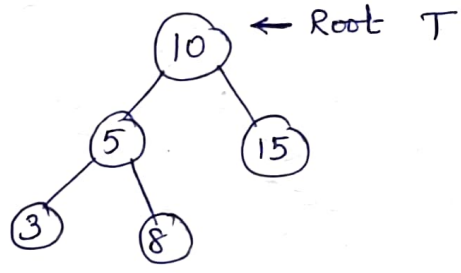


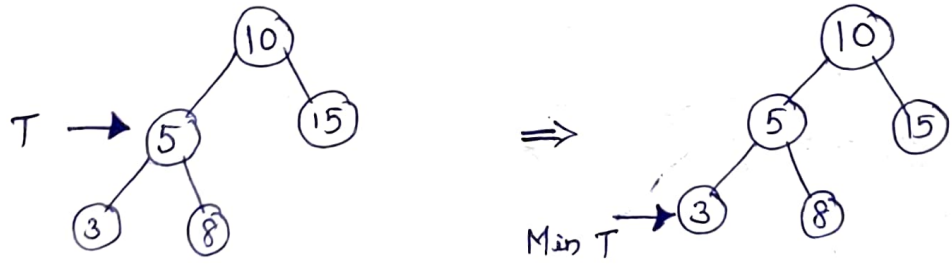
← 10 is checked with root (found)

FindMin :

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

Eg :





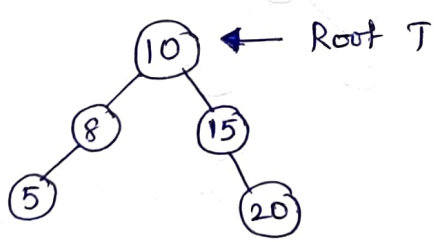
Non - Recursive Routine for FindMin :

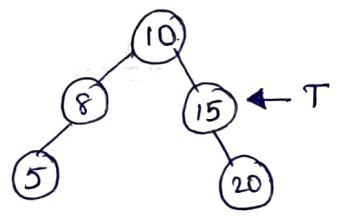
```
int FindMin (SearchTree T)
{
  if (T != NULL)
    while (T->left != NULL)
      T = T->left;
    return T;
}
```

FindMax :

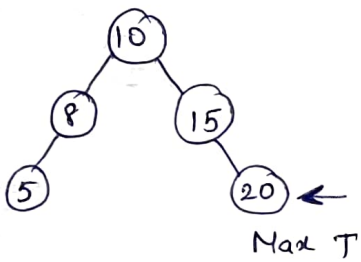
```
int FindMax (SearchTree T)
{
  if (T == NULL)
    return NULL;
  else if (T->right == NULL)
    return T;
  else FindMax (T->right);
}
```

Eg:





=>



Non - Recursive Routine for FindMax :

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

Deletion :

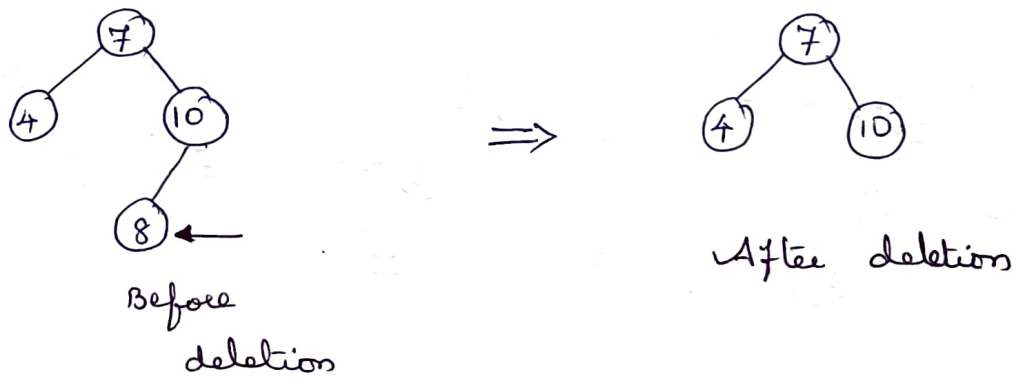
Case 1 : Node with no children

Case 2 : Node with one child

Case 3 : Node with two child

Case 1 : Node with no children

Delete 8 ,



Case 2 : Node with one child.

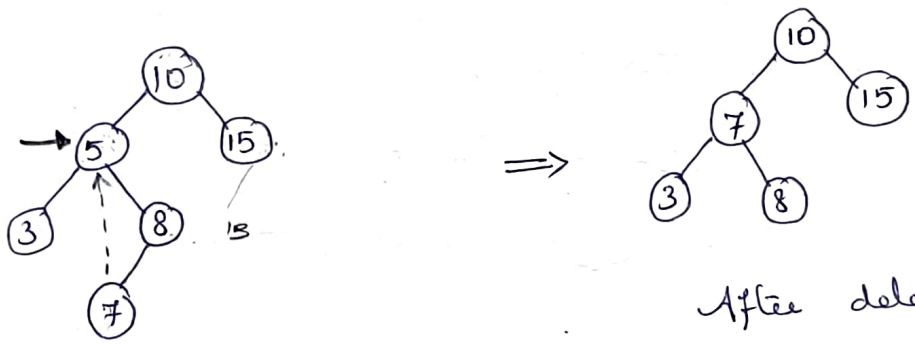
Delete 5 ,

Before deletion

After deletion

Case 3 : Node with two children

Delete 5 ,



Before deletion

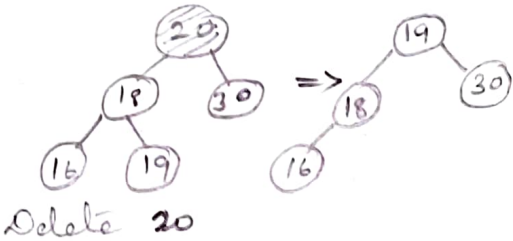
After deletion

(Median should be the root node.

eg: 5 ; (7), 8 (or)

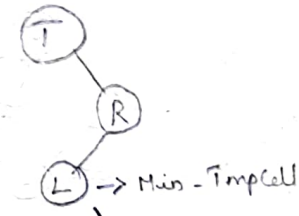
The minimum value of the right sub tree)

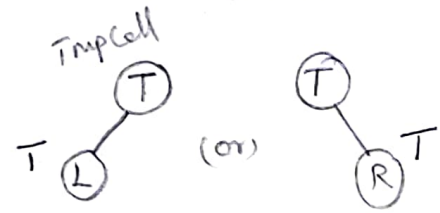
[NOTE : If we delete a node with two children, choose either its minimum of right subtree (or) maximum of left tree.]



Delete 20

```
SearchTree Delete ( int X , SearchTree T)
{
  int TmpCell ;
  if (T == NULL)
    Error ("Element not found");
  else if (X < T -> Element)
    T -> left = Delete (X , T -> left);
  else if (X > T -> Element)
    T -> right = Delete (X , T -> right);
  else if (T -> left && T -> right)   // with two children
  {
    TmpCell = FindMin (T -> right);
    T -> Element = TmpCell -> Element;
    T -> right = Delete (T -> Element , T -> right);
  } // Delete min element in right subtree and
    move the remaining tree as its right child
  else                                // with zero (or) one child
  {
    TmpCell = T ;
    if (T -> left == NULL)
      T = T -> right ;
    else if (T -> right == NULL)
      T = T -> left ;
    free (TmpCell);
  } return T ;
}
```





AVL Trees

→ AVL trees are named after the inventors Adelson, Velskii and Landis.

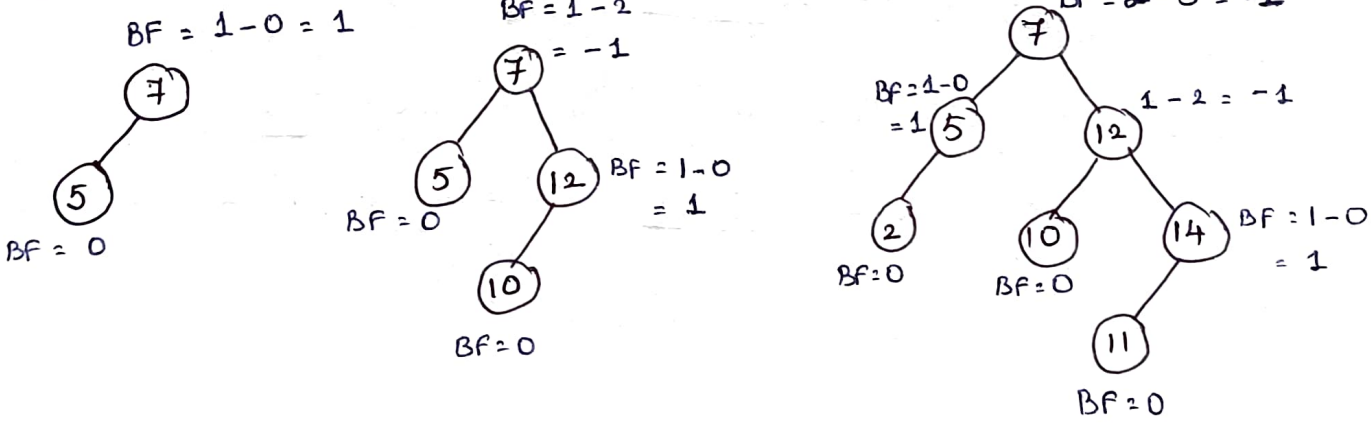
→ An AVL tree is a binary search tree except that for every node in the tree, the height of the left and right subtrees can differ by atmost 1.

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

→ A balance factor is the height of the left sub tree minus (-) height of the right sub tree.

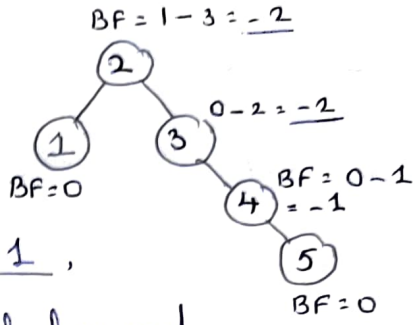
→ For an AVL tree, all balance factor should be +1, 0, -1.

Eg:



Representation of AVL Trees

Imbalance conditions :



- If the balance factor of any node in the AVL tree becomes less than -1 (or) greater than +1, then tree is said to be imbalanced.
- The tree has to be balanced by making either single (or) double rotation.
- The imbalance conditions are,

Case 1 : An insertion into the left subtree of the left child of node x. (LL)

Case 2 : An insertion into the right subtree of the left/right child of node x. (RR/LR)

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

Case 4 : An insertion into the right subtree of the right child of node x. (RR)

Balancing an AVL Tree

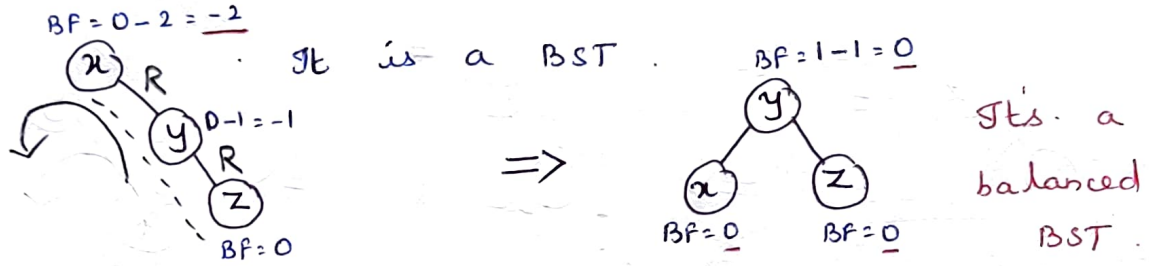
- The imbalances can be overcome by,

1). Single Rotation
   - Left (LL)
   - Right (RR)

2). Double Rotation
   - Left (LR)
   - Right (RL)

x, y, z

Eg: 1

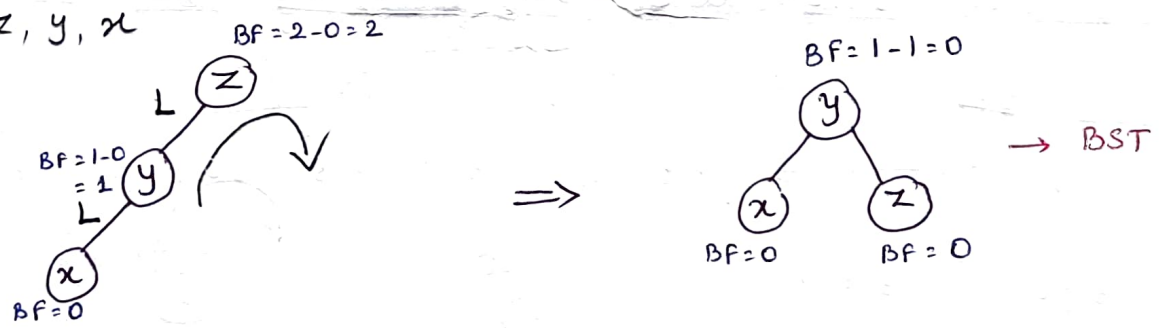


RR - SRR

(Single Right Rotation)

z, y, x

Eg: 2



LL - SLR

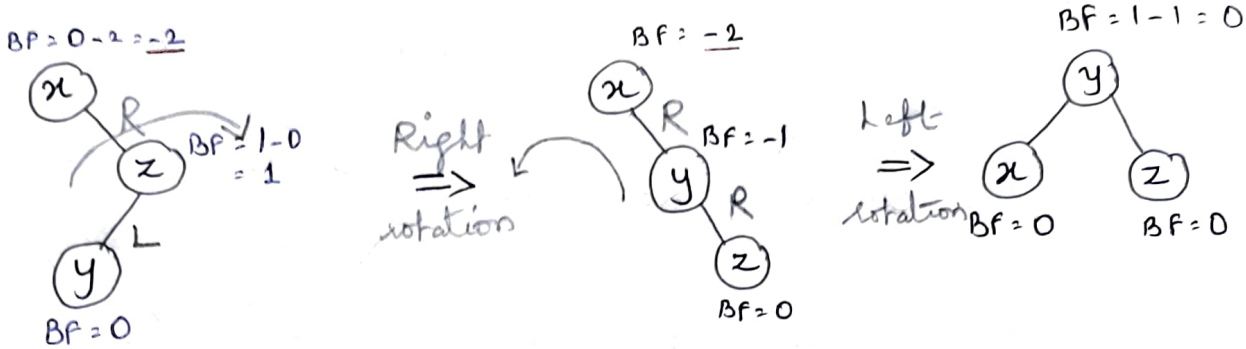
(Single Left Rotation)

NOTE: The median element would be the root.

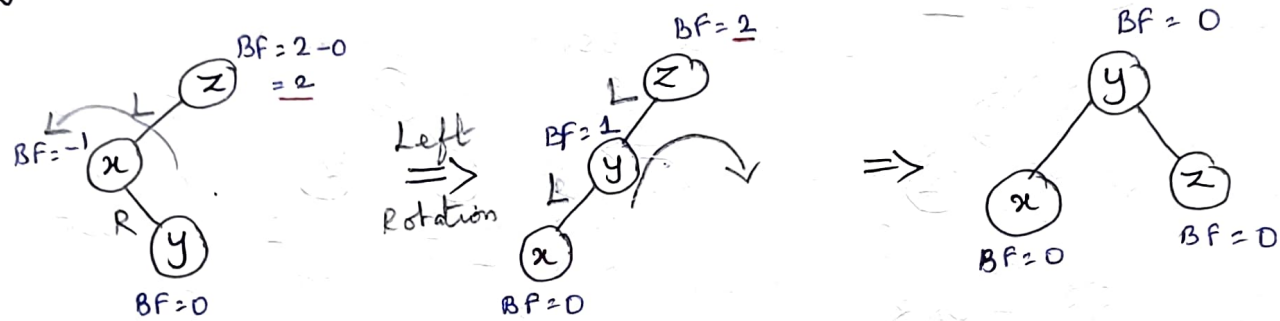
Eg: x, y, z → y is the root (median)

1, 2, 3 → 2 is the median.

Eg: 3 x, z, y

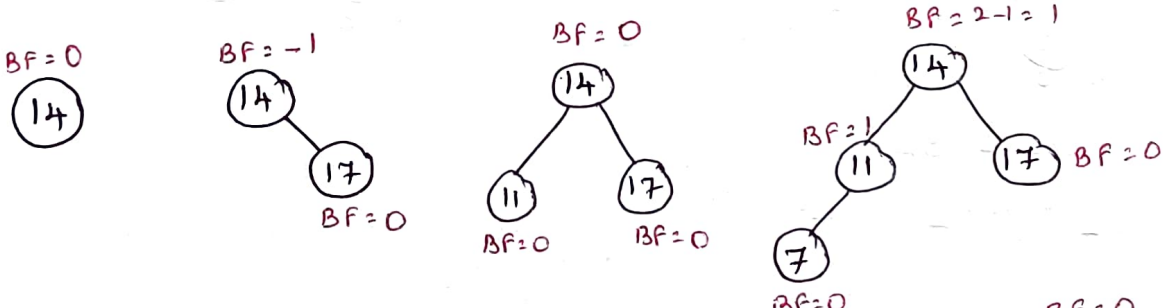


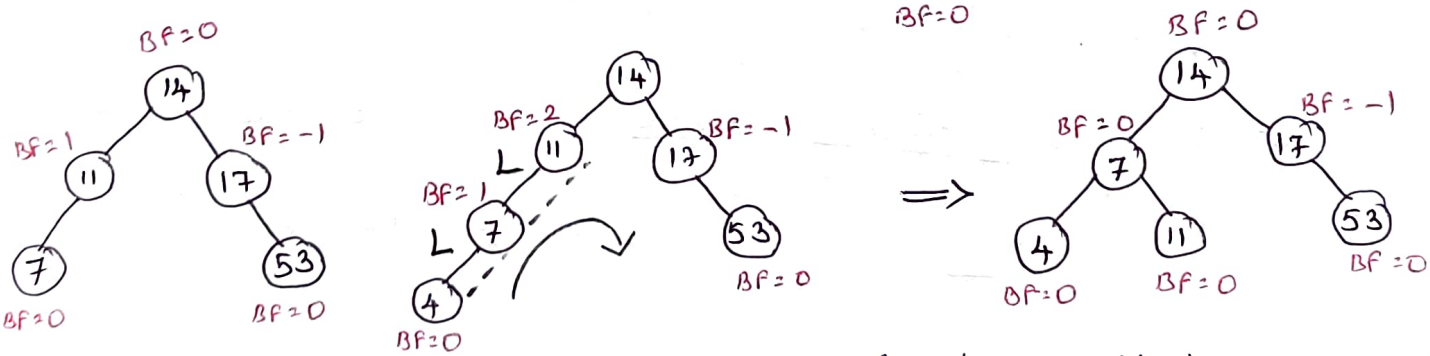
Eg: 4 z, x, y



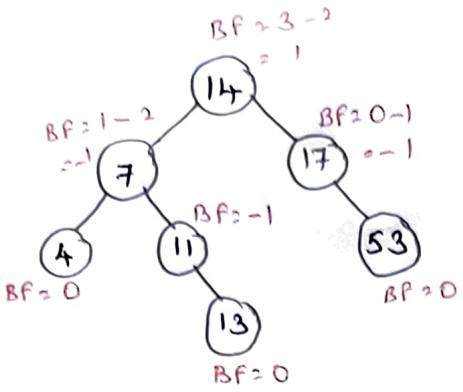
Construct AVL tree by inserting the following elements.

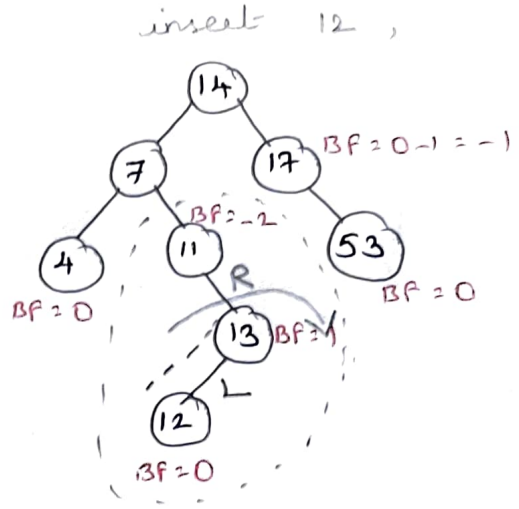
14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

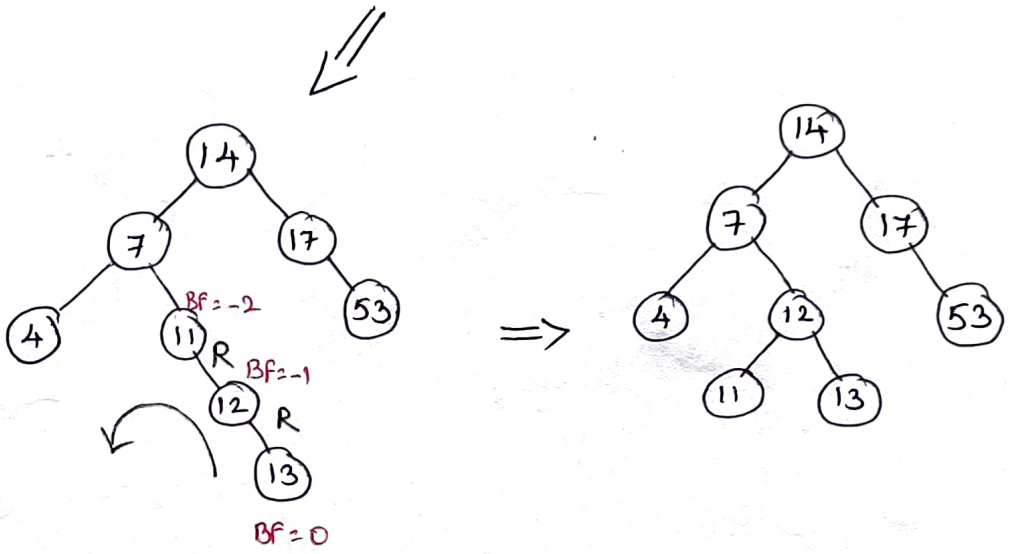




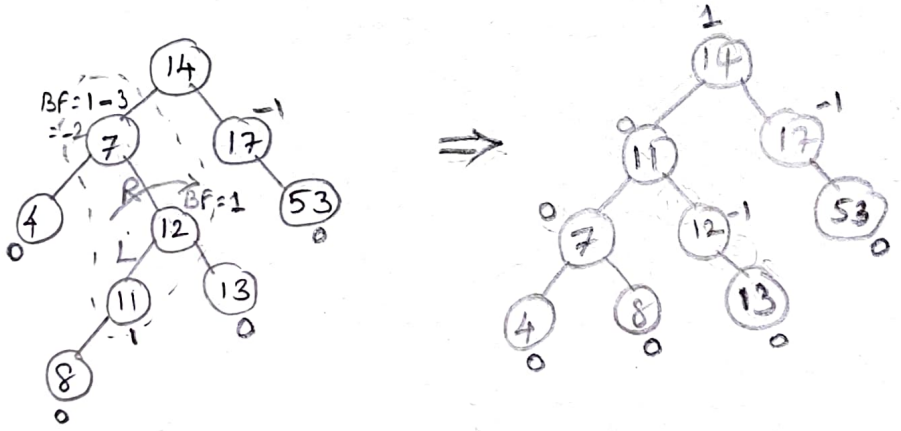
(Medians would be the root)



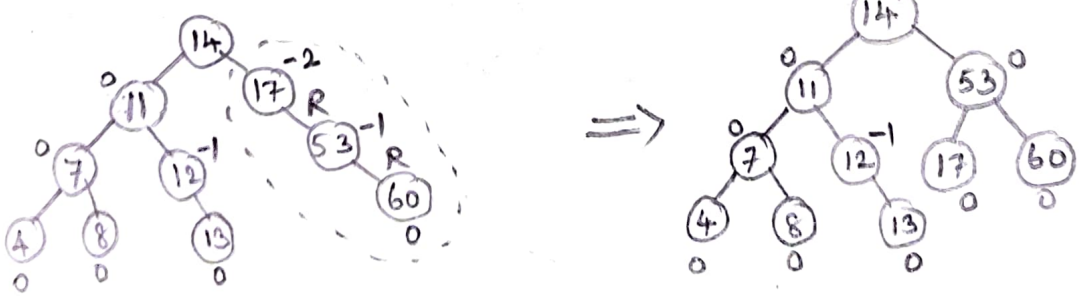




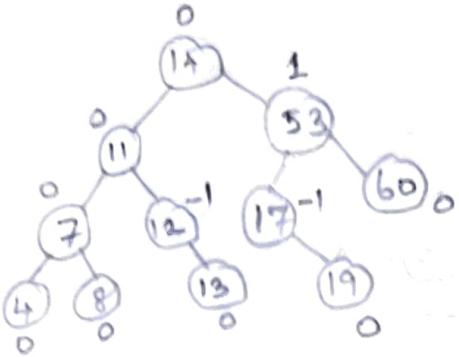
Insert 8



Insert 60



Insert 19

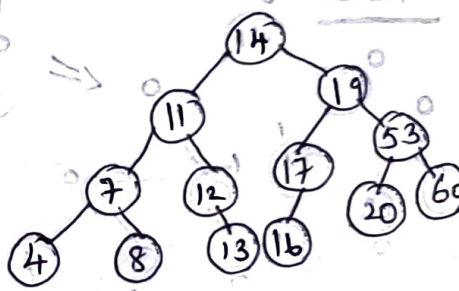


Insert 16

Insert 20

SLR ⟹

⟹ Final BST



(2). Construct an AVL Tree with values:

1, 2, 3, 4, 5, 6, 7, 16, 15, 14, 13, 12, 11, 10, 8.

AVL Tree - Deletion

- Deletion in an AVL Tree is similar to that in a BST.
- Deletion in AVL tree consists of two steps.

  * Removal of the node : The given node is removed from the tree structure. The node to be removed can either be a leaf node (or) an internal node.

  * Re-balancing of the tree : The elimination of a node from the tree can cause imbalance of certain node. Thus it is important to re-balance the balance factor of the nodes.

Deletion :

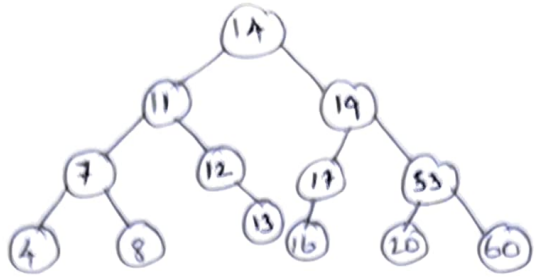
Case 1 : If the node to be deleted is a leaf node, it is simply removed from the tree.

Case 2 : If the node to be deleted has one child, the child node is replaced with the node to be deleted.

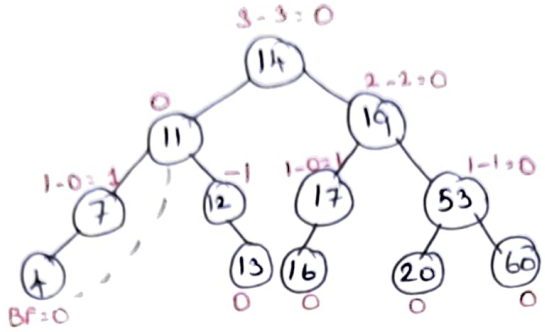
Case 3 : If the node to be deleted has two children,

→ Either replace the node with its inorder predecessor (ie)., the largest element of the left subtree (or) inorder successor (ie)., the smallest element of the right subtree.

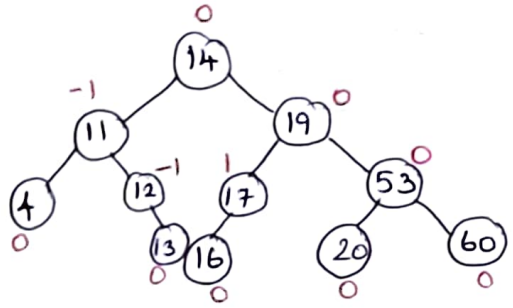
Eg:



Delete 8 ,

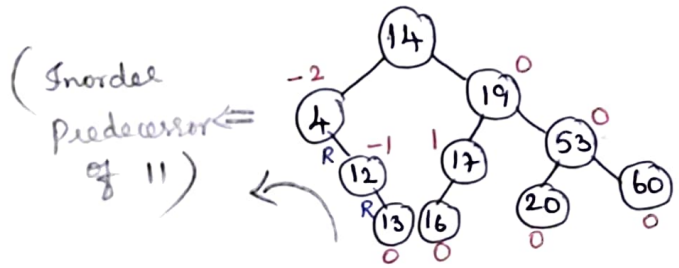


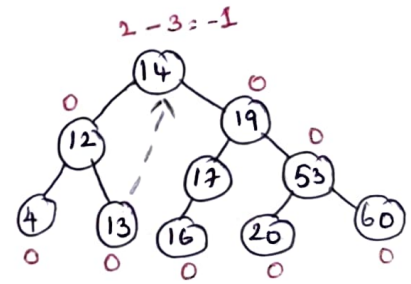
Delete 7 ,



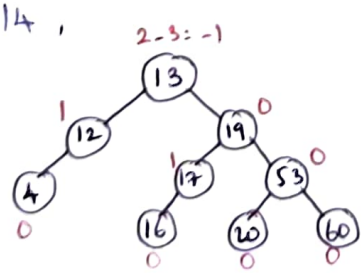
47, 64, 65, 66, 67,
68, 69, ~~70~~, 74, 75,
79, ~~80~~, 82, 84, 85,
86, 87, 88, 90, 92,
95, 96, 97, 100, 101,
103, 104, 105, 111,
112, 113, 115, 117,
119, .

Delete 11 ,

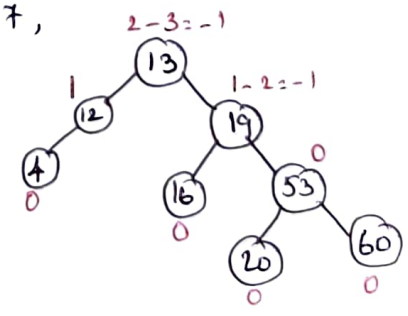




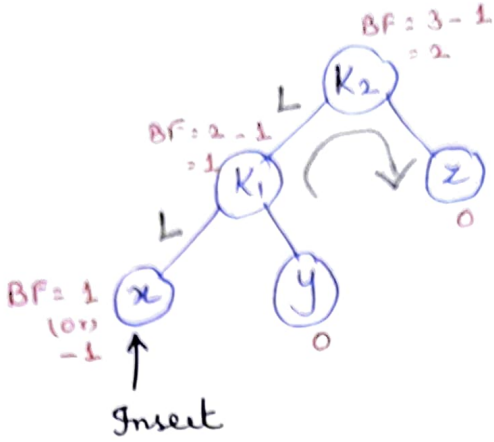
Delete 14 ,

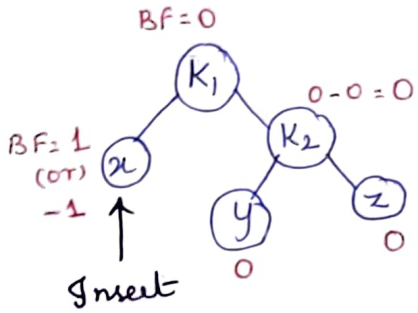


Delete 17 ,

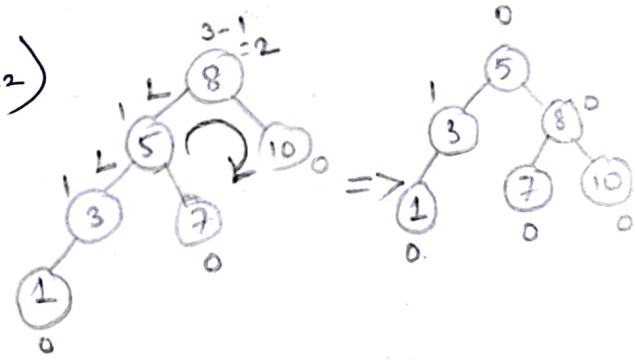


Single Rotation (Left-Left) :



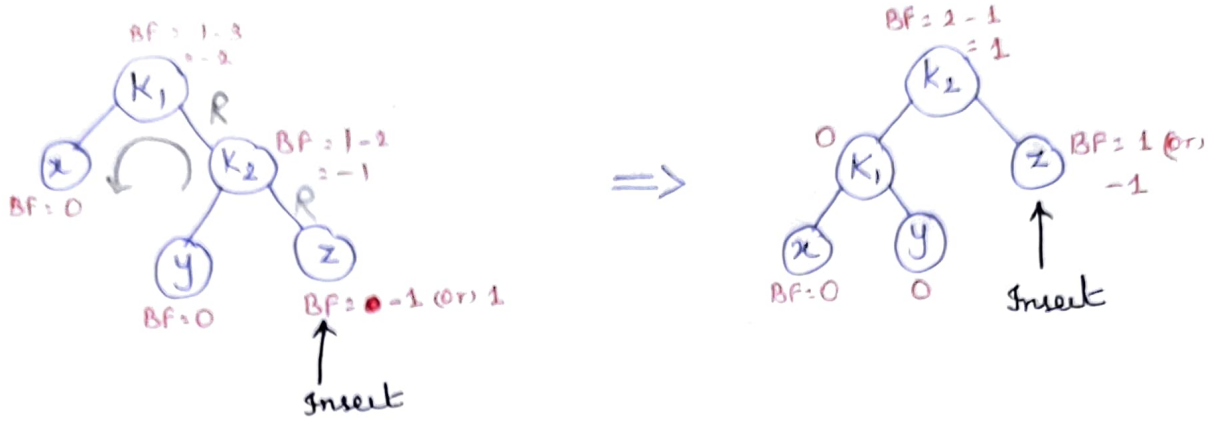


Routine to perform Single Rotation with Left :




```
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),
                        Height (K1 -> Right)) + 1 ;
    return K1 ;
}
```

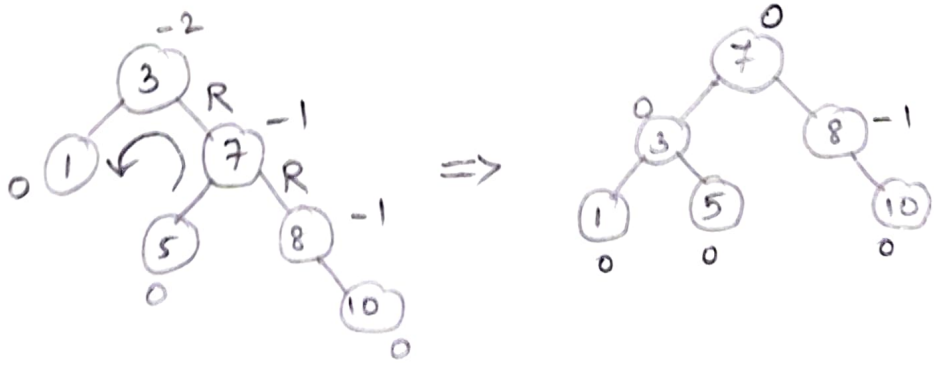
Single Rotation (Right - Right) :



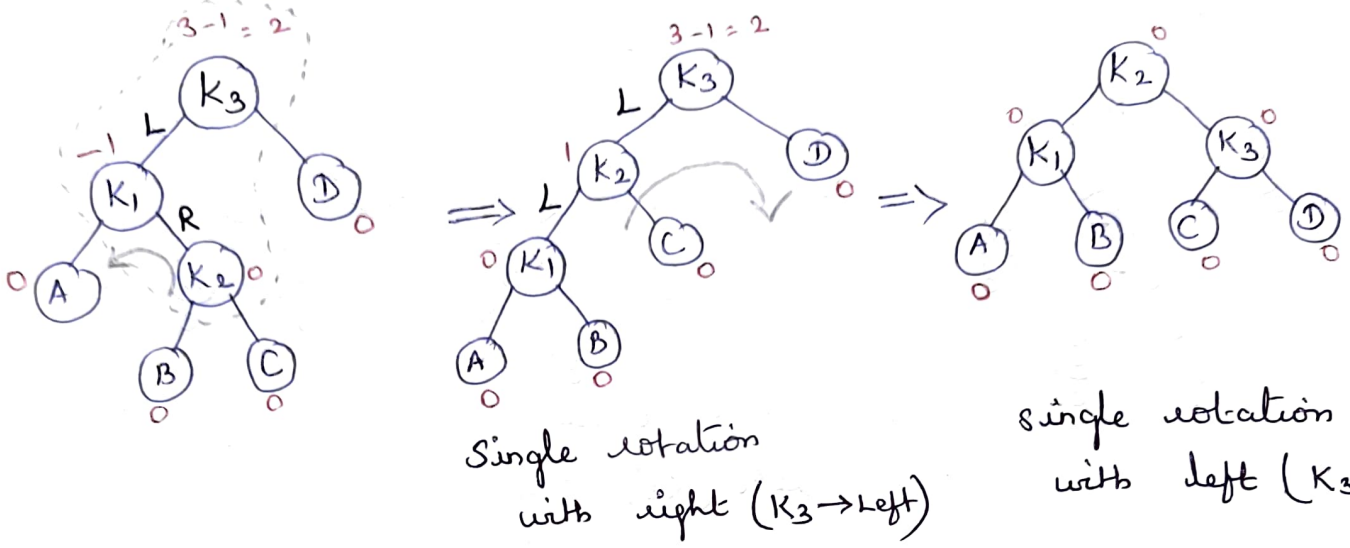
Routine to perform Single Right Rotation :

```
SingleRotationwithRight ( 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 ;
}
```

Eg:



Double Rotation (Left - Right) :



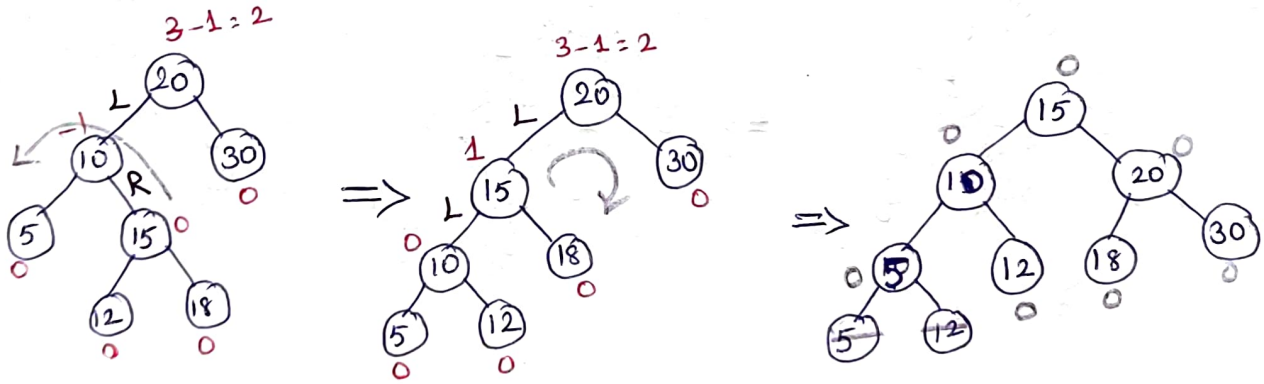
Single rotation with right ($K_3 \rightarrow$ Left)

Single rotation with left (K_3)

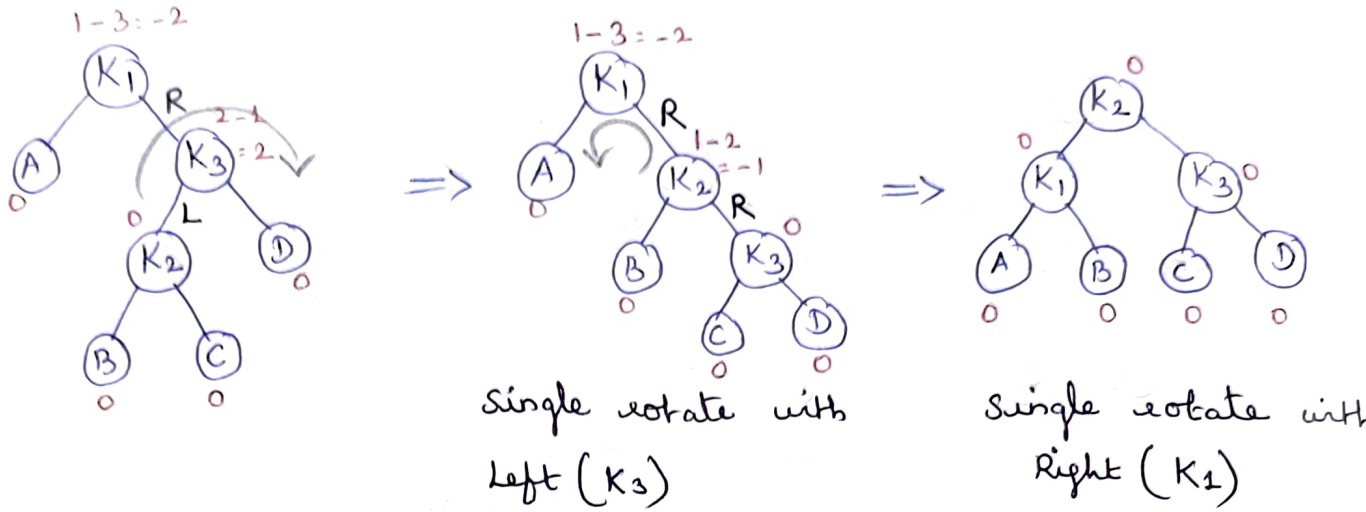
Routine to peeform Double rotation with left :

```
DoubleRotationWithLeft ( Position K3 )
{
  K3 -> Left = SingleRotatewithRight ( K3 -> Left ) ;
  Return SingleRotatewithLeft ( K3 ) ;
}
```

Eg :



Double Rotation with Right :



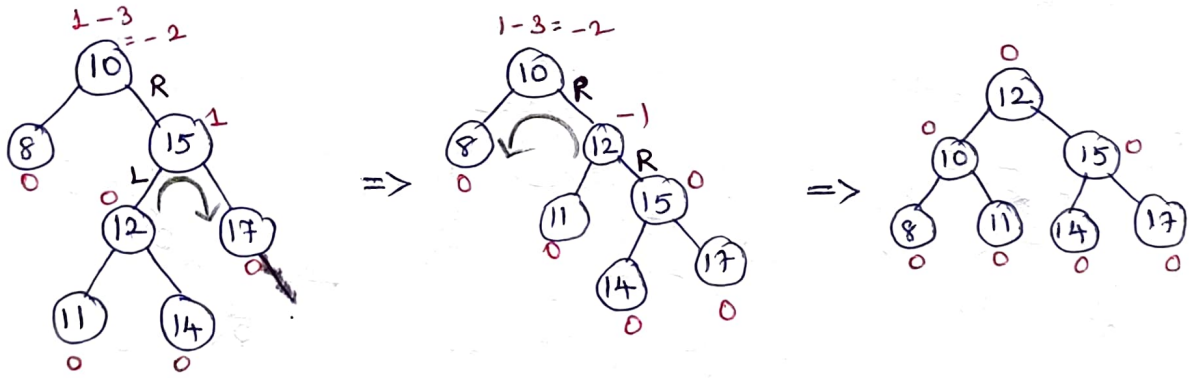
Single rotate with Left (K_3)

Single rotate with Right (K_1)

Routine :

```
DoubleRotatewithRight ( Position K1)
{
  K1 → Right = SingleRotatewithLeft (K1 → Right) ;
  return SingleRotatewithRight (K1) ;
}
```

Eg:



Routine to insert in an AVL Tree.

```
AVL Tree Insert (AVL tree T, int X)
{
  if (T == NULL)
  {
   T = malloc (sizeof (struct AVL node));
   if (T == NULL)
      Error ("Out of space");
   else
    {
     T -> data = X;
     T -> Height = 0;        // single node
     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 = SingleRotateWithLeft (T);
     else
        T = DoubleRotateWithLeft (T);
  }
```

Node structure

```
struct Node
{
 int data;
 struct Node * Left;
 struct Node * right;
 int height;
};
```

```
else if (x > T->data)
{
  T->Right = Insert(T->Right, x);
    if (Height(T->Right) - Height(T->Left) == 2)
      if(x > T->Right->data)
        T = SingleRotatewithRight(T);
      else
        T = DoubleRotatewithRight(T);
    }
  T->Height = Max(Height(T->Left),
                  Height(T->Right)) + 1;
  return T;
}
```

HEAP

- Heap is a special kind of data structure in which the tree is a complete binary tree.

Properties of Heap :

* Structure property - should be a complete binary tree.
* Heap-order property
  - → Min Heap
  - → Max Heap

} - Types.

Min Heap :

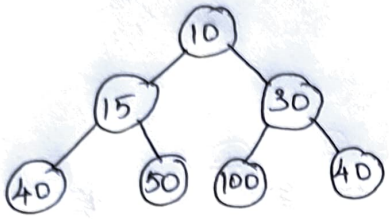
- In Min-Heap, the key present at the root node must be minimum (or) less than (or) equal to the keys present at all of it's children.
- The same property must be recursively true for all sub trees in that Binary tree.

Max Heap :

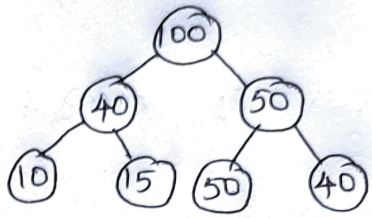
- In Max-Heap, the key present at the root node must be greater than (or) equal to the keys present at all of it's children.
- The same property must be recursively true for all sub trees in that binary tree.

Heap Data structure

Min Heap

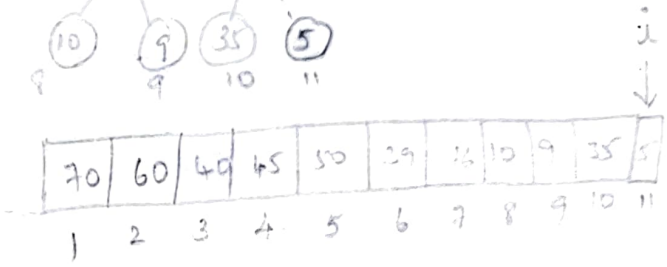


Max Heap



Heap - Insert (Max Heap)

```
insert (A, n, value)
{
  n = n + 1 ;   // increasing the array size
  A[n] = value ;
   i = n;
  while (i > 1)
  {
   Parent = [i/2];
   if (A[Parent] < A[i])
   {
    Swap (A[Parent], A[i])
      i = parent;
   }
  else
    {
     return ;
    }
}
```



Parent = $i/2$

LC = $2i$

RC = $2i+1$

Applications of Heap:

→ Heap is used while implementing a priority queue.

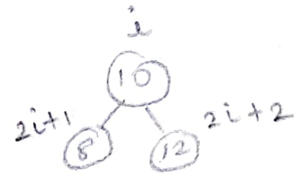
→ Dijkstra's Algorithm, Prim's Algorithm.

→ Heap sort

→ It can be used to efficiently find min (or) max elements in an array.

Heap - DeleteMax (Max heap)

```
deleteRoot (a[], n)
{
 int lastelement = a[n-1];
  a[0] = lastelement;
   n = n-1;
   Heap(a, n, 0);
}
Heap (a, n, i)        // Max Heapify
 {
   int largest = i;
   int l = (2*i)+1;
   int r = (2*i)+2;
   if ((l < n) && a[l] > a[largest])
        largest = l;
    if ((r < n) && a[r] > a[largest])
         largest = r;
      if (largest != i)
       {
         int  swap = a[i];
              a[i] = a[largest];
              a[largest] = swap;
             Heap (a, n, largest);
        }
```



Applications of Trees :

-> Syntax tree - Scanning, parsing, generation of code and evaluation of arithmetic expression in compiler design.

-> Trie - Used to implement dictionaries with prefix look up.

-> Suffix Tree - For quick pattern searching in a fixed text.