SASURIE INSTITUTIONS

SASURIE
COLLEGE OF ENGINEERING

Vijayamangalam, Tiruppur.

Department of Artificial intelligence and Data Science

Regulation 2021

I Year – II Semester

AD3251 – Data Structure Design

AD3251 Data Structure and Design.

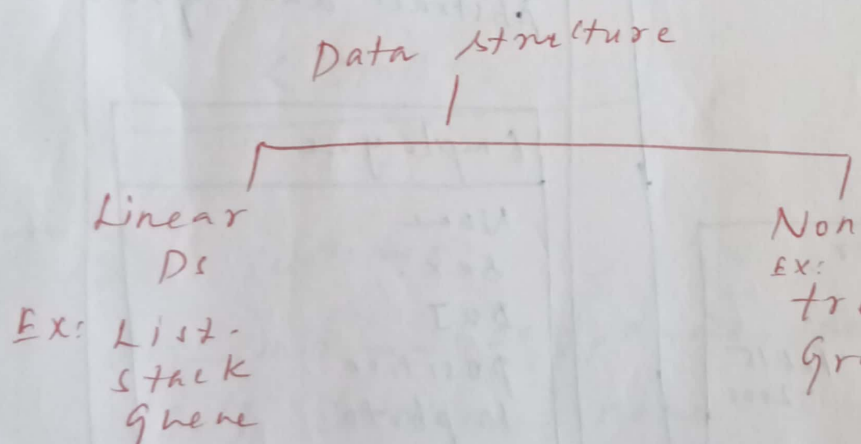
UNIT-1 ABSTRACT DATA TYPE

Abstract Data Type (ADTs) - ADTs and classes - Introduction to OOP - classes in python - Inheritance - namespaces - shallow and deep copying. Introduction to analysis of algorithm - asymptotic Notation - recursion - analyzing recursive algorithm.

Introduction to Data Structure.

Data structure can be defined as the collection of elements and all the possible operations which are required for those set of elements.

Types of Data structure.



ADT

ADT is a mathematical abstraction. It specifies the operation and the semantics of the operations but it does not specify the implementation of the operation.

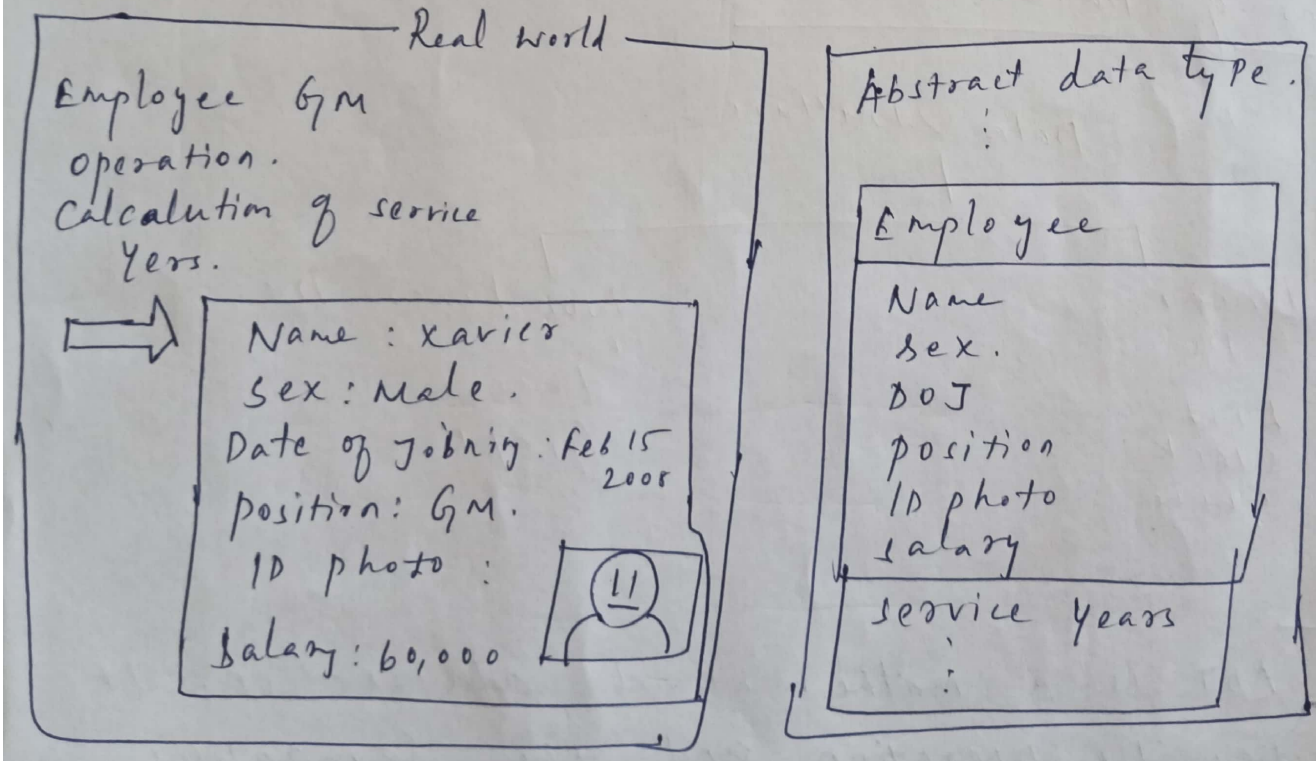
Advantage of ADT:

1. ADTs give the feel of plug and play interface.
2. Reduce the program developing time.
3. Reduce the chance of logical error in code.
4. Increase Understandability and modularity.

ADTs and classes.

A data structure is the implementation for an ADT. In oop; an ADT and its implementation together make up a class. Each operation associated with the ADT is implemented by a member function or method.

EX:



Introduction to OOP

Goals:

1. Robustness
2. Adaptability
3. Reusability.

Principles:

1. Modularity
2. Abstraction.
3. Encapsulation.

Classes in python.

Defining a class.

class is a Collection of data attributes and the methods that can access or manipulate the data attributes.

Syntax:

```
class classname:
    <stmt1>
    :
    <stmtn>
```

EX:

```
class Customer:
    def __init__(self, name, Iden, accno):
        self.CusName = name
        self.CusID = Iden.
        self.CustAccno = accno.

    def display(self):
        print("Customer Name", self.CusName)
        print("Customer ID    ", self.CusID).
        print("Customer Acc.no", self.CusAccno)
c = Customer("Ramesh", 100, 4224)
c.display()
```

Self Identifier & Self parameter.

- Self identifier places a key role.
- Self identifies the instance upon which a method is invoked. While writing function in a class, at least one argument has to be passed, that is called Self parameter.
- Self parameter is a reference to the class itself.

Object Creation.

- An object is the runtime entity used to provide the functionality to the python class.

Syntax:

```
object_name = class_name
```

dot operator (.) is used to call the functions.

```
object_name.function_name()
```

class Variable Vs Instance Variable.

- class variable is defined in the class and can be used by all the instance of that class.
- Instance variable is defined in a method and its scope is only within the object that defines it.

Ex:

```
class order:
    def __init__(self, coffee_name, price):
        self.coffee_name = coffee_name.   } class variable.
        self.price = price.

ram_order = order("Espresso", 210).
print(ram_order.coffee_name)
print(ram_order.price).
```

Instance variable.

Constructor.

Constructor is a special method used to create and initialize an object of a class.

Syntax:

```
def __init__(self):
```

TYPES:

1. Default Constructor.
2. Non-parameterized "
3. Parameterized "

Encapsulation.

It describes the Concept of bundling data and methods within a single unit.

EX:

```
class Employee:
    def __init__(self, name, project):
        self.name=name.          } Data Members.
        self.project = project
    def work(self):
        print(self.name,'is Working on', self.project)
```

Method.

Access Modifiers

3 Types
- Public Member.
- private "
- protected "

Public Member:

Public data member are accessible within and outside of a class. All member variable of the class are by default public.

private Member:

- To define a private member, prefix the variable name with two underscores.
- private members are accessible only within the class.

protected Member:

- protected members are accessible within class and also available to its sub class.
- To define a protected member, prefix the member name with a single underscore.

Ex:

```
class Employee:
    def __init__(self, name, salary):
        self.name = name.      → public
        self._project = project → protected
        self.__salary = salary. → private.
```

Operator Overloading.

- It means giving extended meaning beyond their predefined operational meaning. For example operator + is used to add two integers as well as join two strings and merge two lists.

- To perform operator overloading, python provides some special function or magic function that is automatically invoked when it is associated with the particular object.

Ex:

```
class item:
    def __init__(self, price):
        self.price = price
    def __add__(self, other):
        return self.price + other.price
b1 = item(400)
b2 = item(300)
print("Total price", b1+b2)
```

Inheritance.

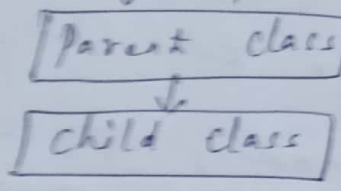
- It is a mechanism through which we can create a class or object based on another class or object.

- It Consists of 5 types.

1. Single Inheritance.
2. Multiple "
3. Multilevel "
4. Hierarchical Inheritance
5. Hybrid "

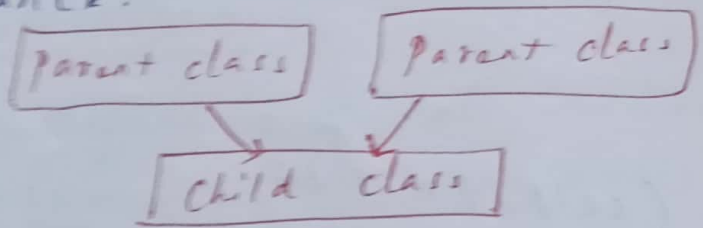
1. Single Inheritance

In Single Inheritance, a child class is inherited from a single-parent class.



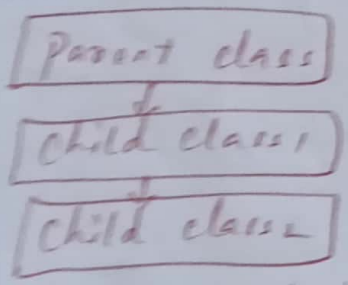
2. Multiple Inheritance.

When child class is derived or inherited from more than one parent class. This is called multiple Inheritance.



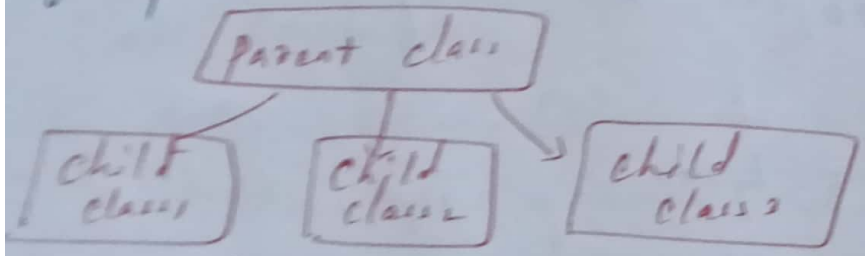
3. Multilevel Inheritance.

In Multilevel Inheritance, a class inherits from a child class or derived class.



4. Hierarchical Inheritance.

More than one child class is derived from a single parent class.



5. Hybrid Inheritance.

Combination of multiple, simple, Multilevel and Hierarchical Inheritance

Built-in Namespace.

- It Contains the names of all of python's builtin objects.

Ex:

```
Name = input ("Enter your name")
print (Name)
```

Shallow and Deep copying.

→ Shallow copying creates a new object that shares the same data as the original object.

→ Deep coying create a new object that has its own copy of the data.

Shallow Copy	Deep Copy
1. Creates a new object. that shares the same data as the original object.	Creates a new object that has its own copy of the data.
2. It faster than shallow copy	It slower than Deep copy.
3. Is used when you need to make a copy of an object that does not Contain any nested objects.	Is used when you need to make a copy of an object that Contains nested objects.

Name Space.

A Namespace is a Collection of Currently defined symbolic names along with information about the object that each name reference.

TYPES:

1. Local Name Space.
2. Global Name Space.
3. Built-in Name Space.

1. Local Name Space.

- Local Name space is a name space for defining the names in a local function.
- It define for a class, function, loop.

EX:

```
def f1():
    y=20  // Local scope of variable.
    print(y)
f1()
```

2. Global Name Space:

It Contains any names defined at the level of main program.

EX:

```
x=10  // global scope.
def f1():
    print(x)
f1()
```

Rate of Growth:

Time Complexity	Name.
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Linear logarithmic
n^2	quadratic
n^3	Cubic
2^n	exponential.

Algorithm analysis
- Best case (lower bound)
- Worst " (upper bound)
- Avg case.

Asymptotic Notations.

- It is a shorthand way to represent the time Complexity.
- Various notations such as Ω, Θ, O.

1. Big oh Notation:

- It is denoted by O.
- It represent the upper bound of algorithm's running time.
- We can give longest amount of time taken by the algorithm to complete.

Example:

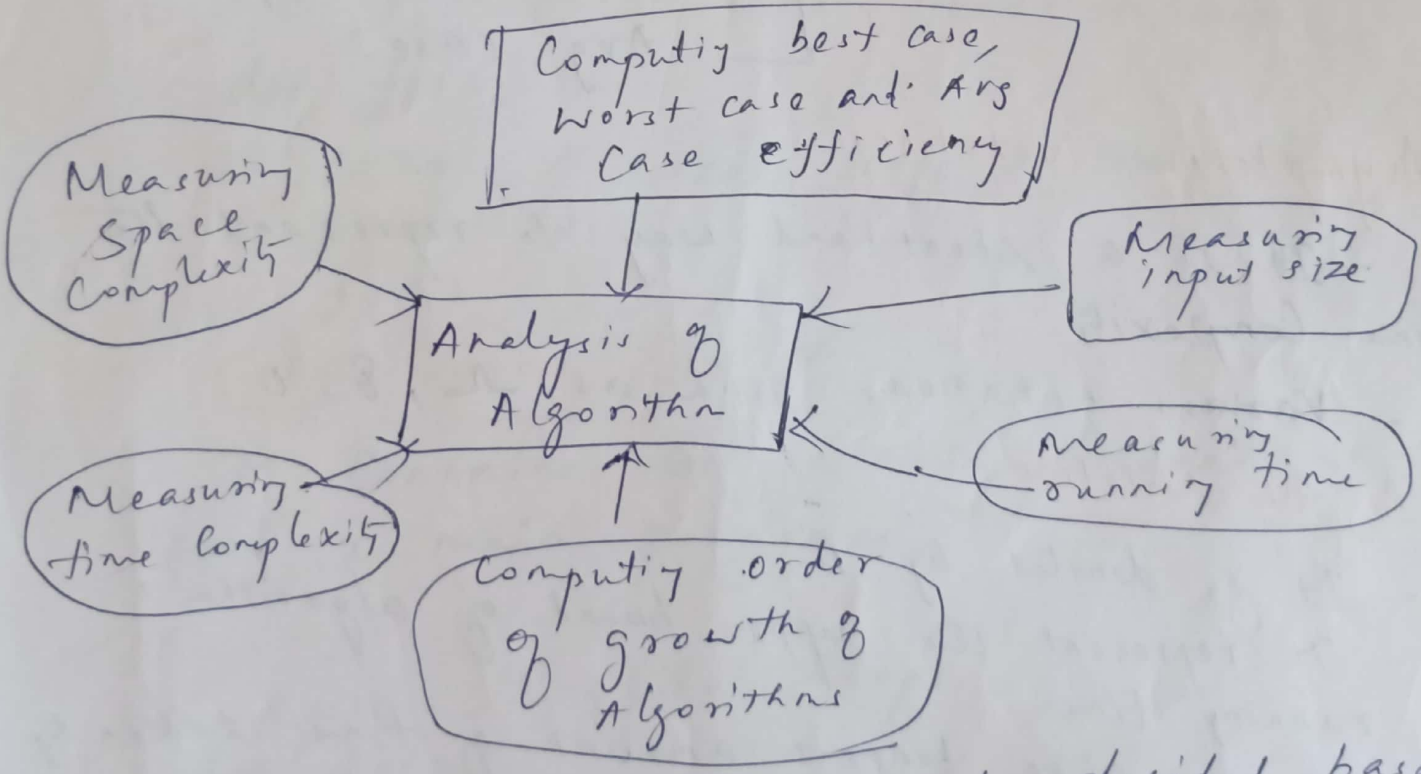
Shallow copy syntax:

```
copy.copy(x)
```

```
lst1 = [1,2,3,4]
lst2 = lst1.
lst2[1] = 1000.
Print(lst2)
print(lst1)
```

```
lst1 = [[1,2,3],[3,4,5
lst2 = copy.deepcopy(lst1)
lst2[1][0] = 100.
print(lst2)
print(lst1).
```

Introduction to Analysis of Algorithm.



efficiency of Algorithm can be decided based on:

1. Amount of time required by an algorithm to execute.
2. Amount of storage required.
3. Size of the i/p set.

Ex:

$f(n) = 2n^2 + 5$ $\quad g(n) = 7n$.

if $n=0$ $\quad f(n) = 5$ $\quad g(n) = 0$. $\quad f(n) > g(n)$

if $n=1$ $\quad f(1) = 7$. $\quad g(n) = 7$ $\quad f(n) \geq g(n)$.

if $n=2$ $\quad f(2) = 13$ $\quad g(n) = 14$ $\quad f(n) < g(n)$.

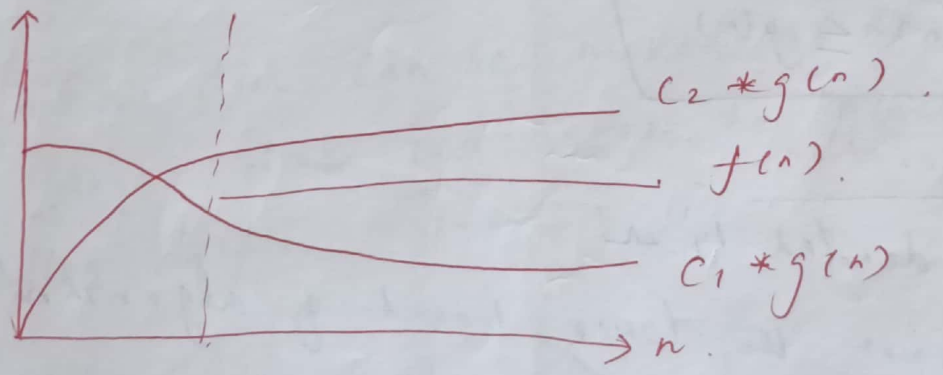
if $n=3$ $\quad f(3) = 23$ $\quad g(n) = 21$ $\quad f(n) > g(n)$.

Hence we conclude $n > 3$ we get $f(n) > g(n)$.

② Notation.

- It is denoted by θ.
- Running time is btw upper bound and lower bound.

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n).$$

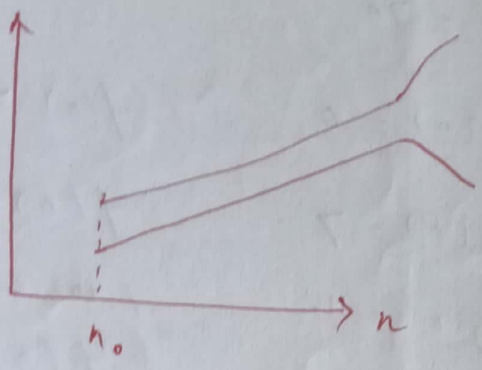


Example:

$f(n) = 2n + 8$ $\quad g(n) = 7n$.

$5n < 2n + 8 < 7n$ $\quad$ for $n \geq 2$.

$\therefore$ $c_1 = 5$, $\quad c_2 = 7$.



$f(n) \in O\, g(n)$

$f(n) \leq c\, g(n)$

Ex:

$f(n) = 2n+2 \qquad g(n) = n^2.$

if $n=1$ then.

$f(n) = 4 \quad g(n) = 1 \quad f(n) > g(n)$

if $n=2$ then

$f(n) = 6 \quad g(n) = 4 \quad f(n) > g(n).$

if $n=3$ Then

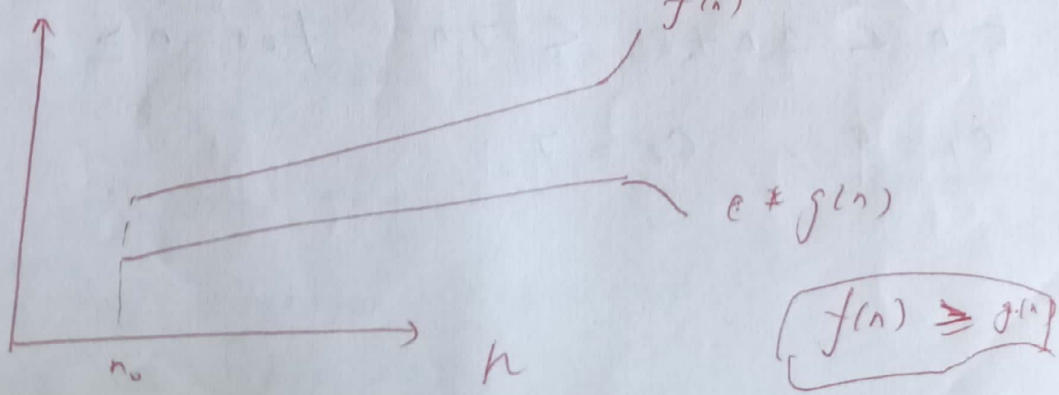
$f(n) = 8 \quad g(n) = 9 \quad f(n) < g(n)$

Hence we can conclude that for $n \geq 2$ we obtain.

$$f(n) \leq g(n)$$

Omega Notation.

- It is denoted by 'Ω'
- Represent the lower bound of algorithm's running time.
- shortest amount of time taken by algorithm.



$f(n) \geq g(n)$

Analyzing Recursive Algorithms.

General plan.

1. Decide the input size based on parameter n.
2. Identify algorithm's basic operation.
3. Check how many times the basic operation is executed.
4. Setup the recurrence relation with some initial condition.
5. Solve the recurrence relation.

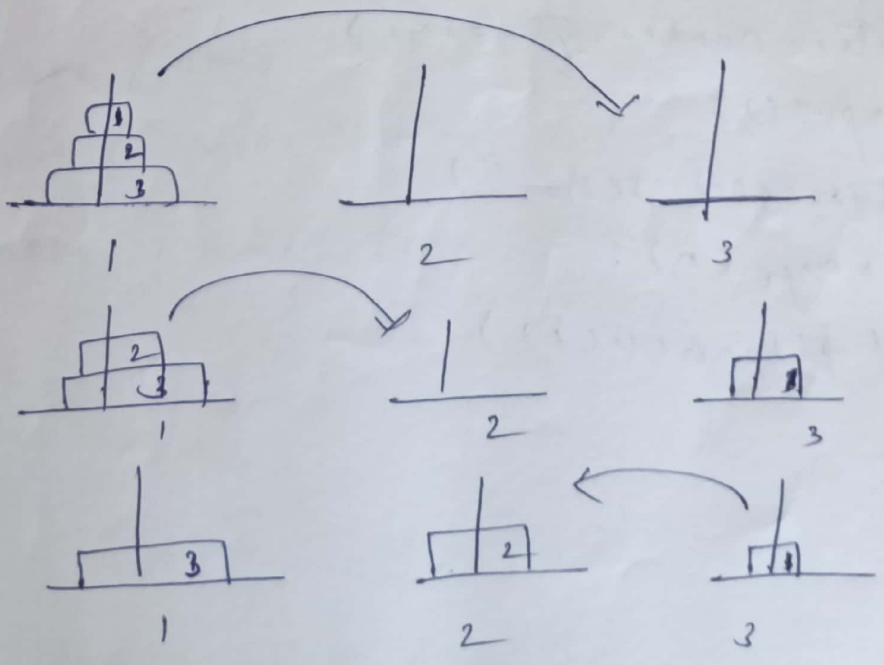
Tower of Hanoi

It Consists of a board with three vertical poles and a stack of disks.

Conditions:

1. Only one disk can be moved at a time.
2. A larger disk can never be placed on top of smaller disk.

Ex:



Recursion:

Recursion is a technique in which a function calls itself directly or indirectly.

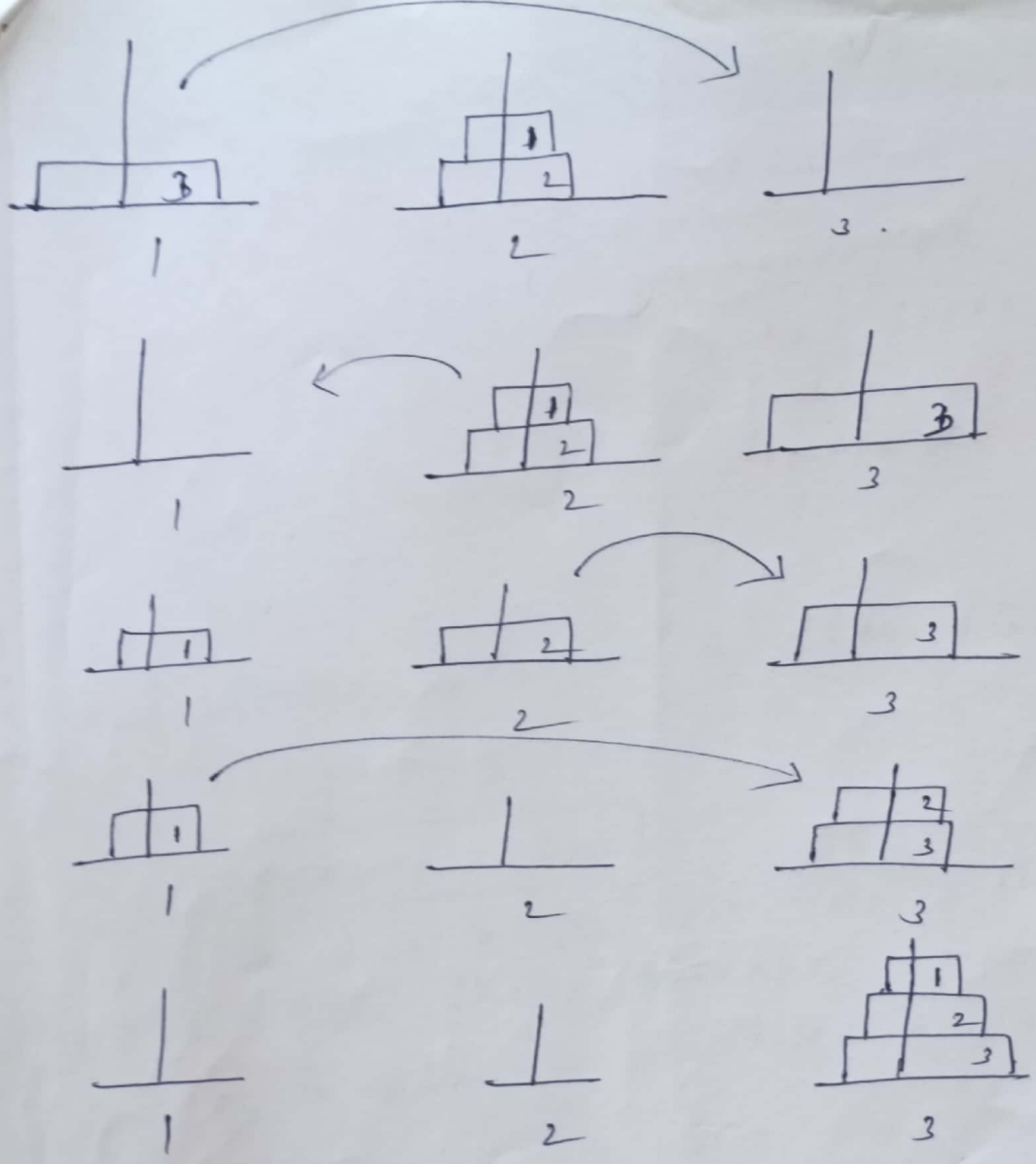
EX:

1) Write a python program for recursive factorial function.

```
def factorial(n):
    if n==0:
        return 1
    else:
        result = n * factorial(n-1)
    return result
```

2) Fibonacci series.

```
def fibonacci(n):
    if(n<=1):
        return n.
    else:
        return (fibonacci(n-1) + fibonacci(n-2))
print("Enter number of terms")
n = int(input())
print("Fibonacci series").
for i in range(n):
    print(fibonacci(i))
```




```
def move(n, src, dest, temp):
    if n >= 1:
        move(n-1, src, ~~dest~~ temp, dest)
        print("move %d → %d", %(src, dest))
        move(n-1, temp, dest, src)
```

UNIT-II LINEAR STRUCTURES.

List ADT – array based implementation – linked list Implementations – singly linked lists – Circularly Linked Lists – doubly linked lists – applications of lists – Stack ADT – Queue ADT – double ended queues.

List ADT

- list is a Linear collection of data item.
- The general form of list is.

$A_1, A_2, A_3 \ldots A_N$.

Where N is the size of the list. A_1 is the first element. A_N is the last elements.

- The list of 0 elements is called as empty lists.

Operations performed on the list are,

1. Insert (X, P) – Insert the element X after the position P.

Ex:
Insert (X, 4).
36, 23, 49, 53, 12, 98 will be 36, 23, 49, 53, X, 12, 98.

2. Delete (X) – Delete the element X from the list.

Ex:
Delete (12): → 36, 23, 49, 53, X, 98.

3. Find (x) – Return the position of x.
4. Next (i) – Return the position of successor element $i+1$
5. Previous (i) – Return the position of predecessor element $i-1$
6. Print List – Print the elements of the list.
7. Make empty – Make the list empty.

Implementation of List ADT

2 Ways. They are,

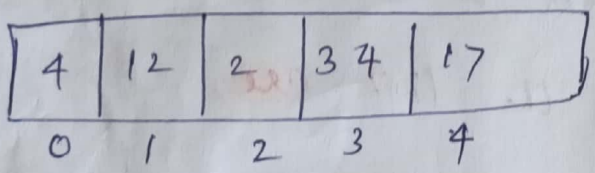
1. Array Based Implementation.
2. Linked List Implementation.

1. Array Based Implementation.

→ In Array Based Implementation of list, a group of releated variable can be stored one after another in a continous memory location.

→ list is mutable.

Ex: 4, 12, 2, 34, 17 can be represented as,



Operations:

1. Creating a python List

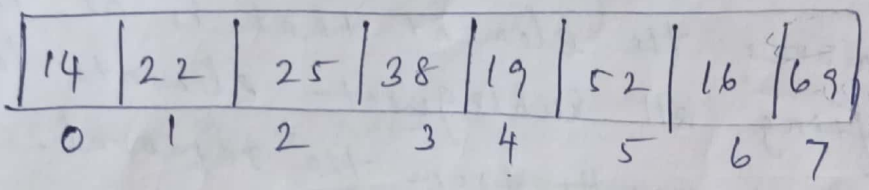
- When the list() Constructor is called, an array structure is created to store the items in the list.
- The length of the list obtained using len().

2. Appending Item:

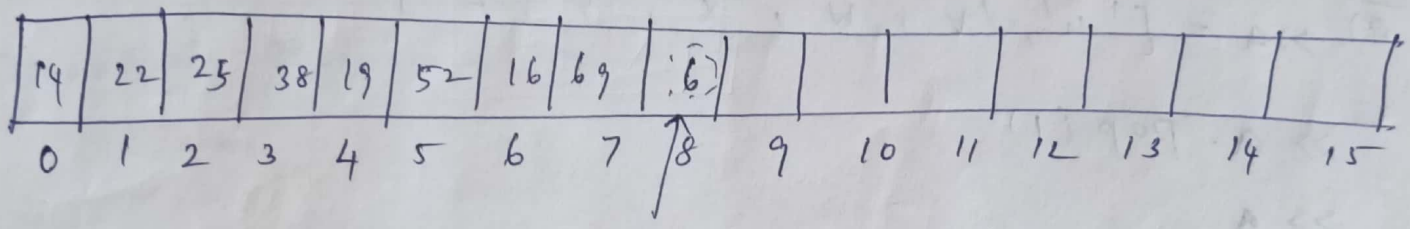
- Append() method is used to insert the item to the end of the list.
- If there is a room in the array, the item is stored in the next available slot of the array and length field is incremented by one.

An array cannot change its size once it has been created. When the array becomes full and there is no free slot to append the item, the list allows expansion of list.

Ex:



To insert 6, the array will have to be expanded.



6 is append to end of list.

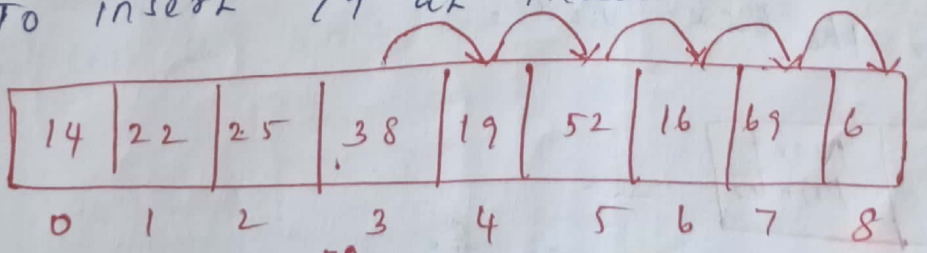
3. Insertion.

- Insertion is the process of adding an element into existing list. It can be done at any position.
- Insert() method is used to insert the element anywhere within the list.

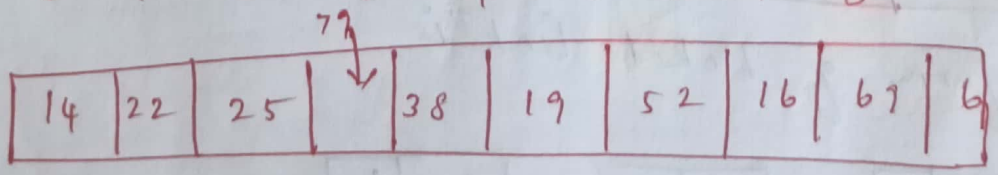
Ex:

To insert 79 at index 3.

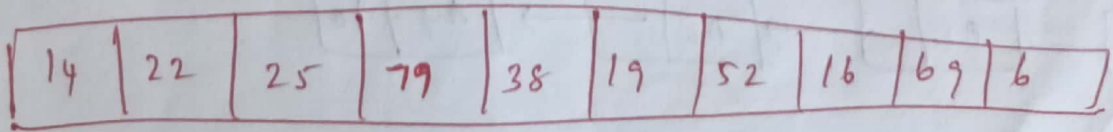
a)



b)



c)



A. Deletion

- Deletion is the process of removing an element from the array at any position.
- pop() removes the last element from a list.
- pop(K), removes the element that is at index K in a list, shifting all subsequent elements leftward to fill the gap that results from the removal.

Ex:

a) >>a = ['u', 'v', 'w', 'x', 'y', 'z']

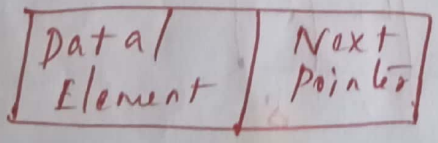
>> a.pop(1)

>> a

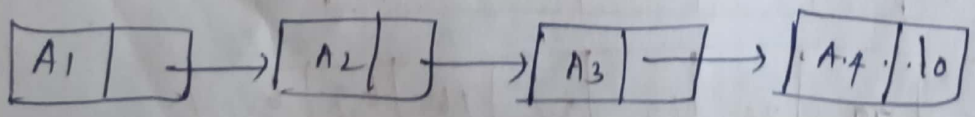
['u', 'w', 'x', 'y', 'z'].

Linked List Implementation

- In Linked list implementation of list, the linked list consists of series of nodes, which are not necessarily adjacent in memory.
- Each structure consists of two parts. ie the element and a pointer to the next node.



The general form of Linked link is,

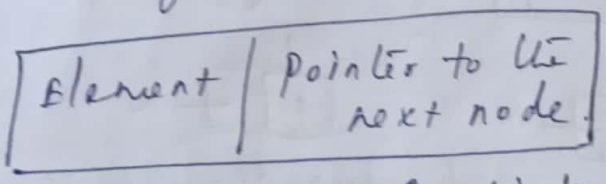


es of Linked list.

1. Singly Linked List
2. Doubly " "
3. Circular " "

1. Singly Linked List

Structure of a node is,



Operations performed in Linked list are,

1. Insertion.
2. Deletion.
3. print List (L), Find (L, key)

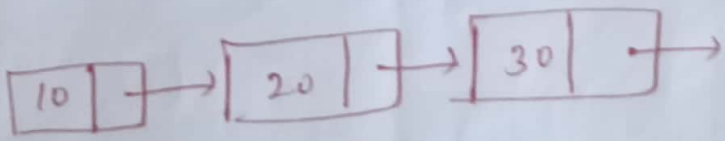
1. Insertion.

a) Insert the element at the Head.

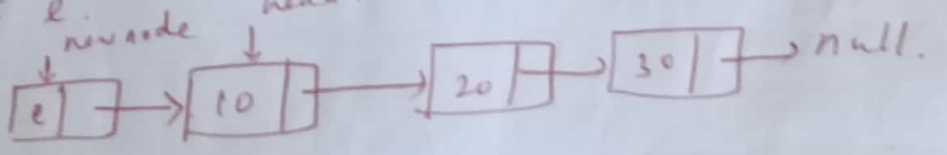
steps:

→ Create a new node, set its element to the new element.
→ Set its next link to refer to the current head.
→ Set the list's head to point to the new node.

Ex:

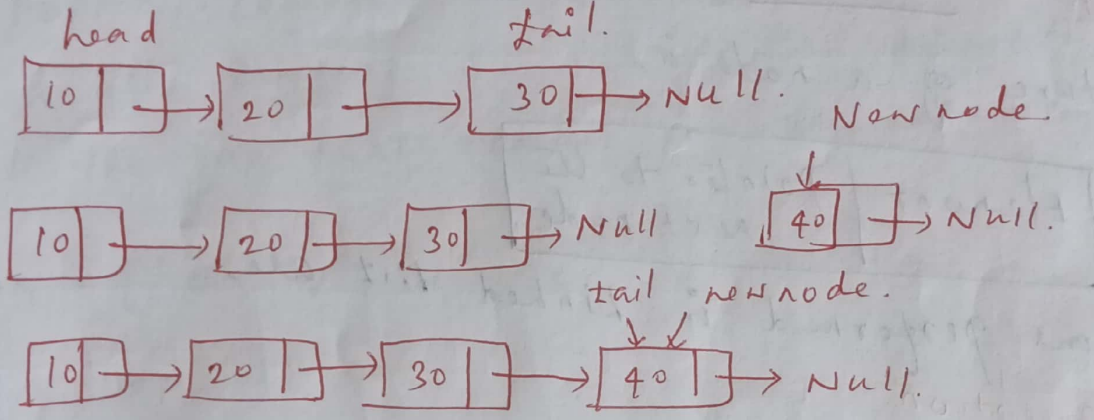


Insert e



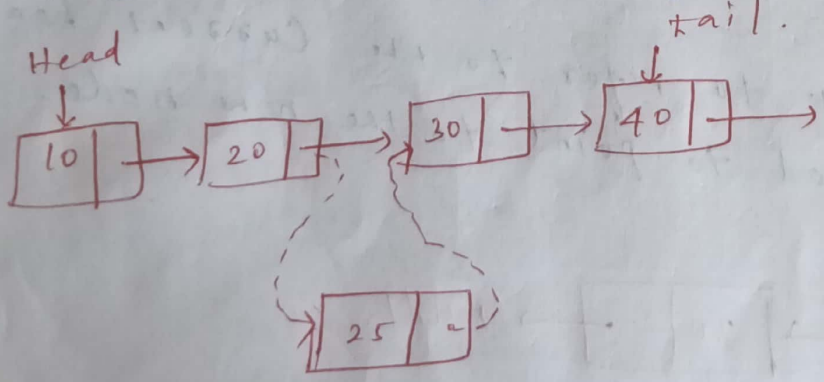
b) Inserting an element at Tail.

→ Create a new node, set its element to the new element assign its next reference to NULL.
→ Set the next reference of the tail to point to this new node and
→ Update the tail reference itself to this new node.



c) Inserting an element at the middle.

→ Create a new node, set its element to the new element.
→ Set the next reference of the newnode as next field of previous node.
→ Set the next field of previous node as newnode.

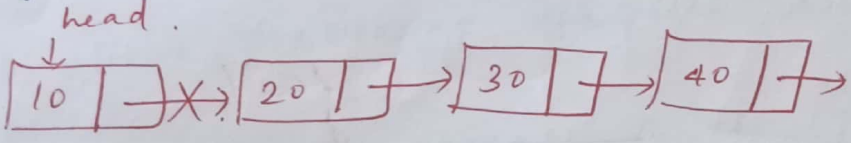


...tion

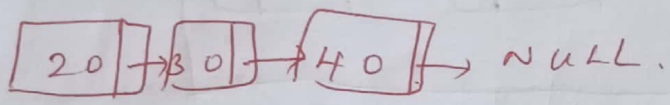
a) Remove first element from a list.

Removing an element from the head of a singly linked list.

Ex:

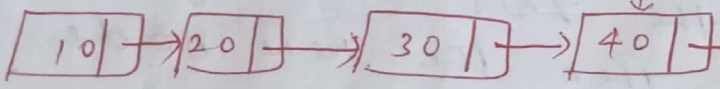


Remove 10

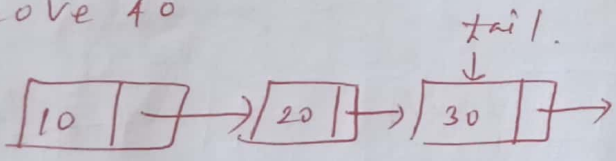


b) Remove last element. To delete the last node from the list, find the node before the last node.

Ex:

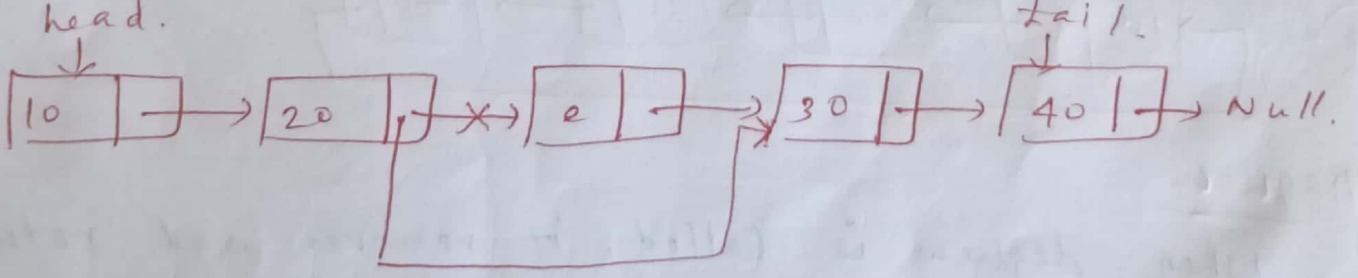


Remove 40



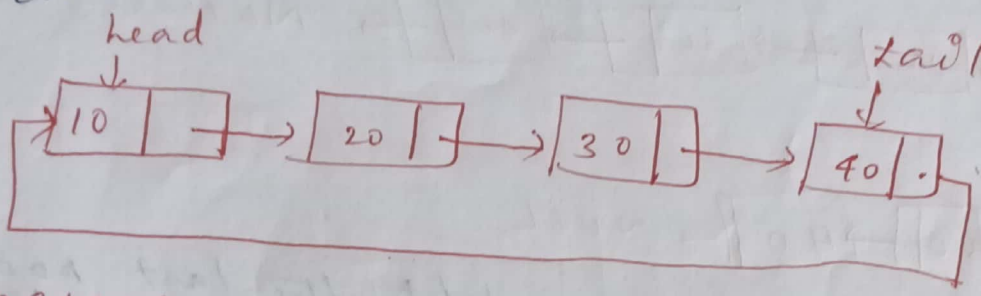
c) Remove Intermediate element

To delete the intermediate node from the list, find the node before the node to be deleted. Then make the following changes.



Circularly Linked List.

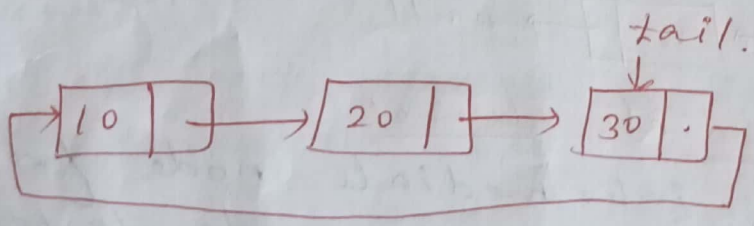
It is a linked list in which the last node of the list points to the first node of the list. Thus nodes form a continous circle.



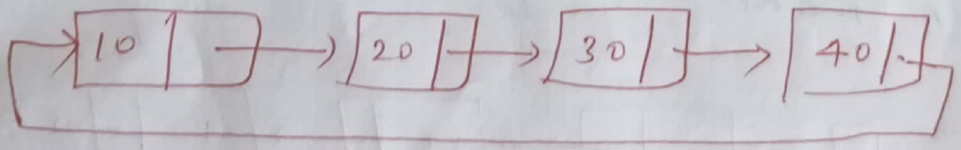
Enqueue.

When enqueue is called, a new node is placed just after the tail but before the current head, and then the new node becomes the tail.

Ex:

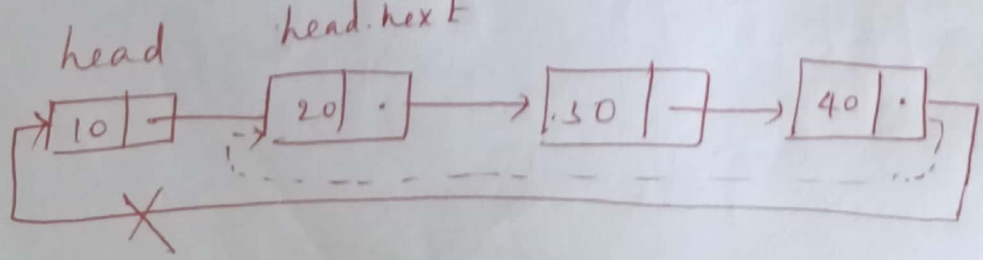


Insert 40



Dequeue:

When dequeue is called, it removes and return the first element of the queue, by bypassing the head node.



Application.

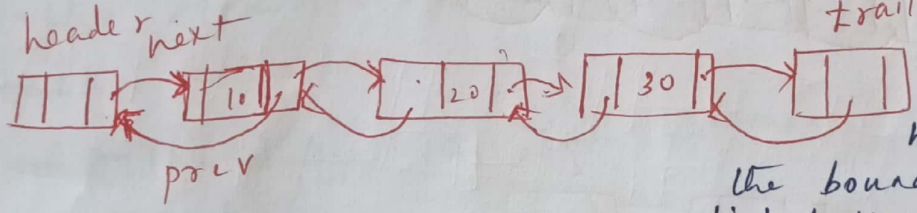
- Data is processed in a round fashion.. for example CPU scheduling.

Advantages:

1. It allows traversing the list starting at any position.
2. It allows quick access to the first and last nodes.

Doubly Linked List:

It Consists of 2 node. i Prev node and next node. Prev node points to the previous node and next node points to the next node.



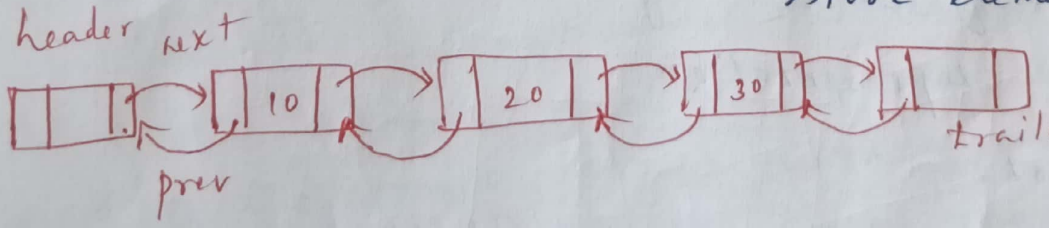
Sentinels:

In order to avoid some special cases when operating near the boundaries of a doubly linked list, a special header node is added at the beginning of the list and trailer node is added at the end of the list. These dummy nodes are called as sentinels and they do not store elements of the primary sequence.

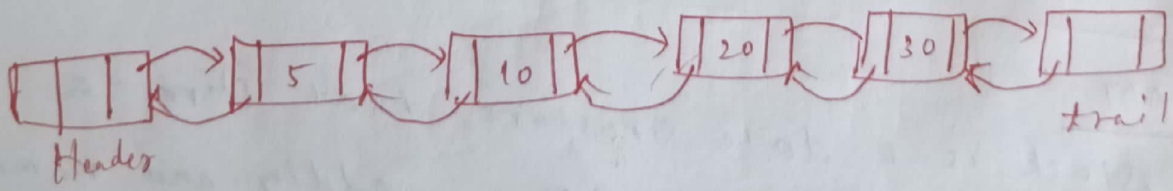
Insertion:

a) Insertion at First:

Ex:

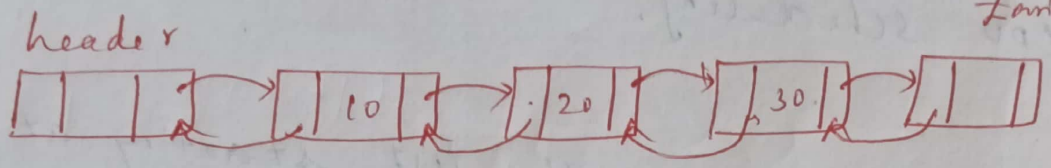


Insert 5.

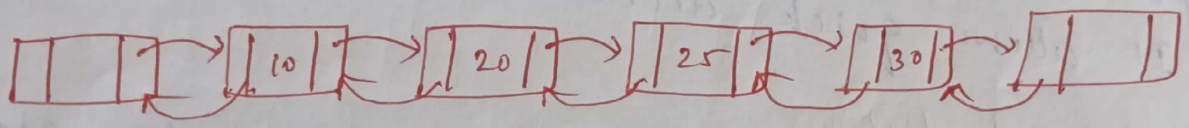


Insert at middle:

Ex:

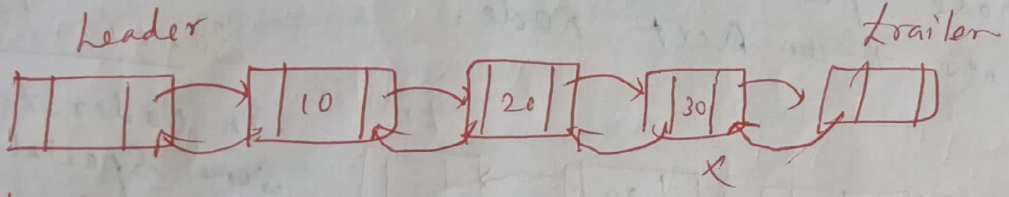


Insert 25

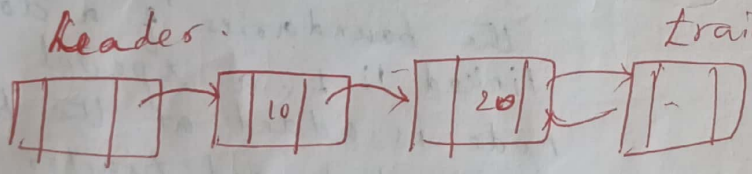


Deletion.

Ex:



delete 30



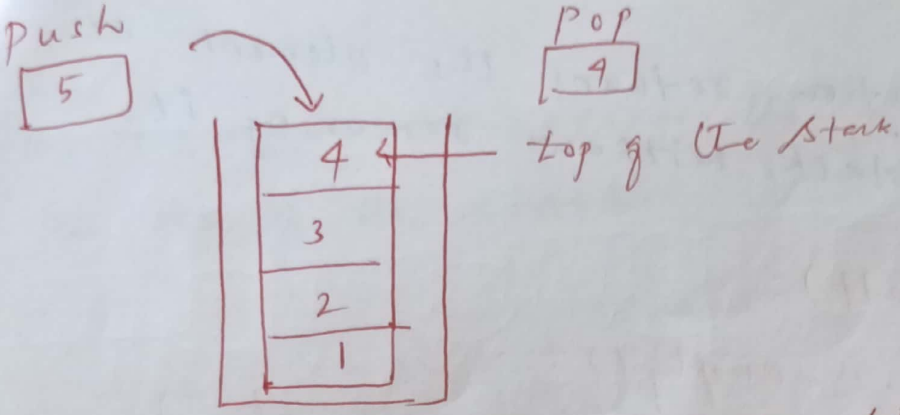
Application of List:

1. Keep track of course registration at a University.
2. polynomial representation.
3. Radix sort.

Linear structures — Stack, Queue

Stack ADT:

A stack is a data structure that stores a linear collection of items. Adding and removing items is restricted to one end known as top of the stak. Stack follows LIFO.



Array Based Implementation of stack using Python List

Stack Operations.

1. push.

- push operation is the insertion operation. In stack, element can be pushed at the top end of the stack.
- push(e) method add element e to the top of ~~the~~ stack s.

```
def push(self, e).
    self.stack.append(e).
```

2. Pop

The Pop operation is the deletion operation. It stack, element can be popped at the top end of the stack. pop() method remove and return the top element from the stack s.

```
def pop(self):
    if self.is_empty():
        raise Empty("Stack is empty")
    return self.stack.pop().
```

Top

The top operation returns the element at the top of the stack, without removing it.

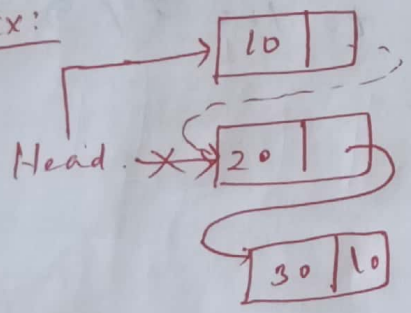
```
def top(self):
    if self.is_empty():
        raise Empty("stack is empty")
    return self.stack[-1]
```

Linked List Implementation of Stack.

push:

push operation is same as insertion at the head of a singly linked list.

Ex:




```
def push(self,e)
    self.head = self.Node(e,
                          self.head)
    self.size = self.size + 1
```

pop:

pop operation is same as deletion at the head of a singly linked list.

```
def pop(self):
    if self.is_empty():
        raise Empty("stack is empty")
    answer = self.head.element
    self.head = self.head.Next
    self.size = self.size - 1
    return answer
```

Top

The top operation returns the element that is at the top of the stack.

```
def top(self):
    if self.is_empty():
        raise Empty("stack is empty")
    return self.head.element
```

Converting infix to postfix.

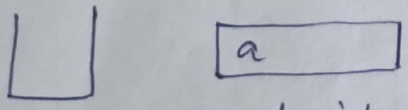
Algorithm.

1. Read one Character at a time.
2. If it is an operand, then it is placed on array.
3. If it is an operator then push it onto stack.
4. Left parentheses also pushed onto the stack.
5. If the input is close symbol then pop the stack until the left parenthesis and write the array.

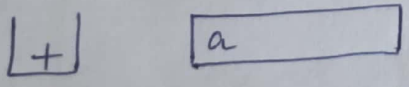
Example:

a + b * c + (d * e + f) * g.

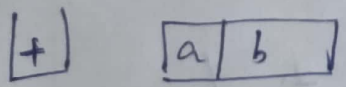
Read a - It is an operand to write an array.



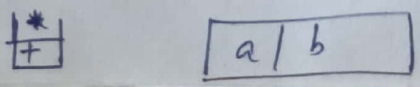
Read + pushed it on the stack.



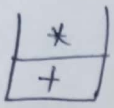
Read b - It is an operand to write an array.

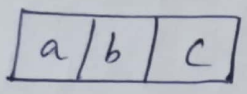


Read * pushed it on the stack.

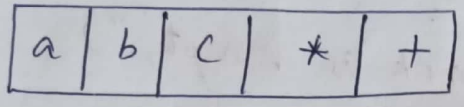


Read c - Write it on to the array.

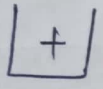


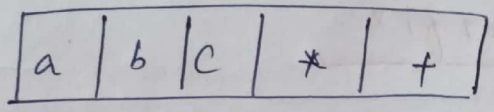


Read +. + is lower priority than *. So pop *, +

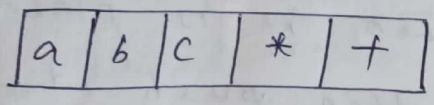


Read +.

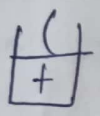


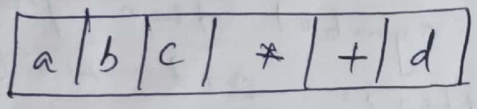


Read (.

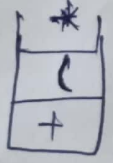


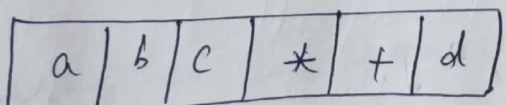
Read d



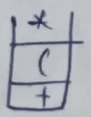


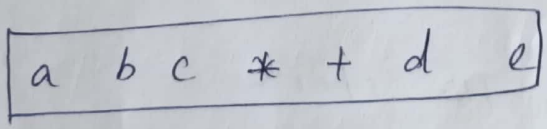
Read *



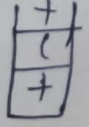


Read e



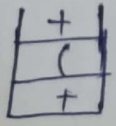


Read +. + is lower priority than *. So pop *.



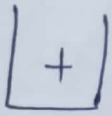
a b c * + d e *.

Read f.



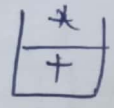
a b c * + d e * f

Read). pop until (.



a b c * + d e * f +

Read *



a b c * + d e * f +.

Read g

a b c * + d e * f + g

a b c * + d e * f + g * +

Evaluating postfix expression.

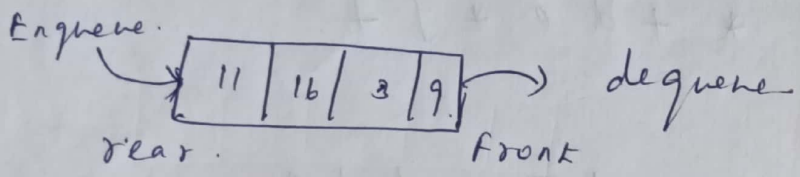
EX:

Evaluate the postfix expression a b c + * d /. Where a:=8, b=2, c=3, d=4.

step	Input	Stack.
1	8	8
2	2	8 2
3	3	8 2 3
4	+	8 5
5	*	40
6	4	40 4
7	/	10

Queue ADT

- Queue is a specialized list in which items can be added to one end and removed from the other.
- It is also known as FIFO.



Queue Methods:

1. Q.enqueue(e)
2. Q.dequeue()
3. Q.first()
4. Q.isempty()
5. len(Q).

Array Implementation of Queue.

Enqueue:

- Enqueue operation is the insertion operation.
- elements can be inserted at the rear end of the queue.

```
def enqueue(self,e):
    self._queue.append(e)
    self._size = self.size+1
```

Dequeue:

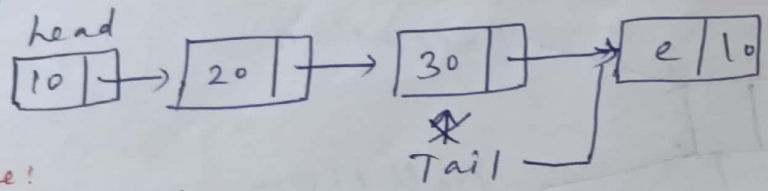
- Dequeue operation is the deletion operation.
- elements can be removed from the front end of the queue.

```
def dequeue(self):
    if self.is_empty():
        raise Empty("queue is empty")
    value = self._queue[self._front]
    self._queue[self._front] = None
    self._front = self._front+1
```

self.size = self.size - 1
return value.

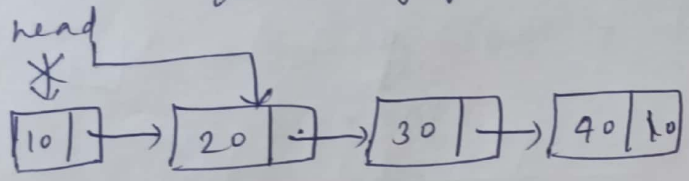
Linked List Implementation of Queue.

Enqueue: It is same as insertion at the tail of a singly Linked List.




```
def enqueue(self, element):
    new = self.node(element, node)
    if self.is_empty():
        self.head = new
    else:
        self.tail.next = new
    self.tail = new
    self.size = self.size + 1
```

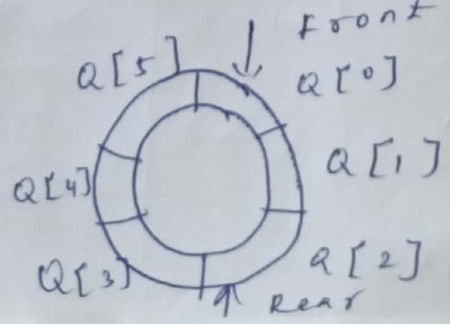
Dequeue: It is same as Deletion at the head of a singly linked list.




```
def dequeue(self):
    if self.is_empty()
        raise isemptyerror("empty queue")
    result = self.head.element
    self.head = self.head.next
    self.size = self.size - 1
    if self.is_empty():
        self.tail = none
    return result.
```

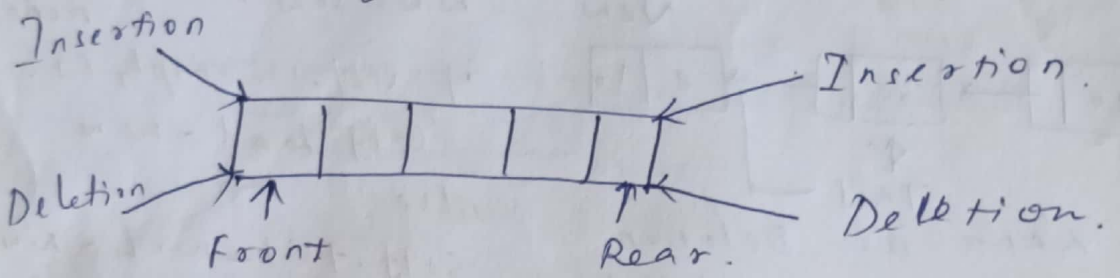
Circular Queue:

Circular Queue is a special Version of Queue. Where last element of the queue is connected to the first element of the queue forming a Circle.



Double Ended queue (or) Dequeue.

It is a data structure that support insertion and deletion at both the front and back of the queue..



UNIT-III SORTING AND SEARCHING.

Bubble Sort - Selection Sort - insertion sort - merge Sort - quick sort - linear search - binary search - hashing - hash function - Collision handling - load factors, rehashing and efficiency.

Sorting Techniques.

Sorting is the process of arranging the elements of a list into either ascending or descending order. There are 2 types of sorting techniques. They are,

1. Internal sorting.
2. External sorting.

Internal sorting:

Sorting is done in main memory of the Computer.

Ex: Bubble sort, quick sort, insertion.

External Sorting.

Sorting is done in secondary memory of the Computer

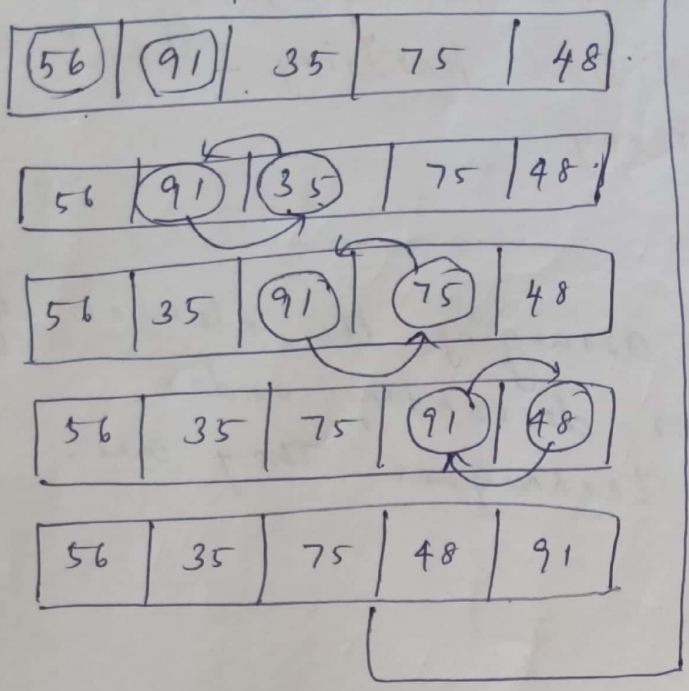
Ex. Multiway merging, Radix sort.

Bubble Sort.

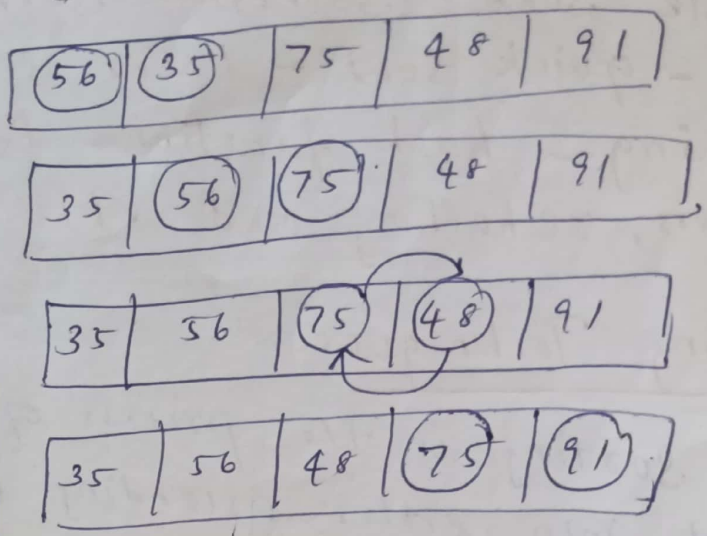
- Bubble sort is the simplest sorting algorithm.
- It works by iterating the input array from the first element to the last element, Comparing each pair of elements and swapping them if needed.
- It continue its iteration until no more swaps are needed.

56 91 35 75 48.

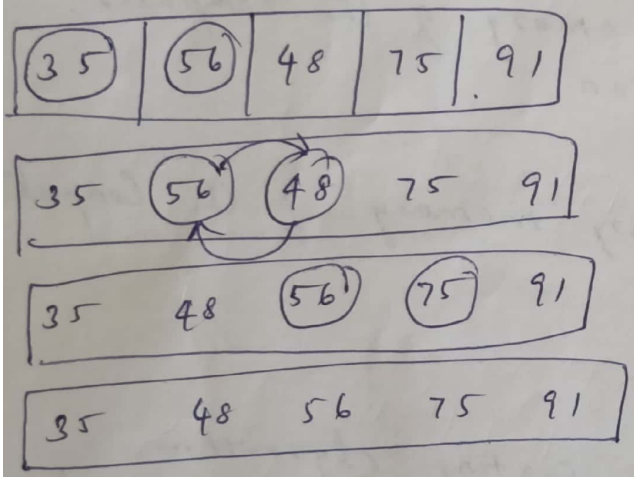
Pass 1



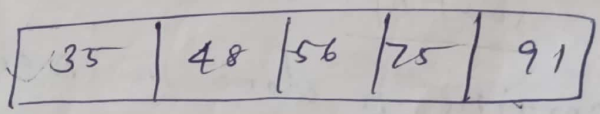
Pass 2



Pass 3.



Pass 4.



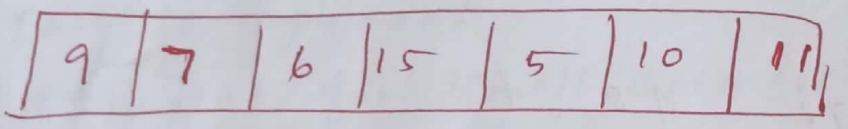

```
def bubble sort(A):
  for n in range (len(A)-1, 0, -1):
    for i in range(n):
      if A[i] > A[i+1]:
        temp = A[i]
        A[i] = A[i+1]
        A[i+1] = temp
A = [56, 91, 35, 75, 48]
bubble sort (A).
print (A).
```

- Simple to implement.
- Sorting the files with very large values.

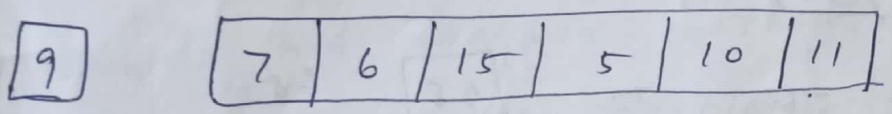
Insertion Sort.

- It is simple and efficient comparision sort.
- It is used to insert the element in appropriate place.

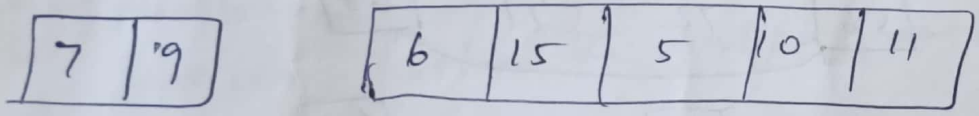
Ex:

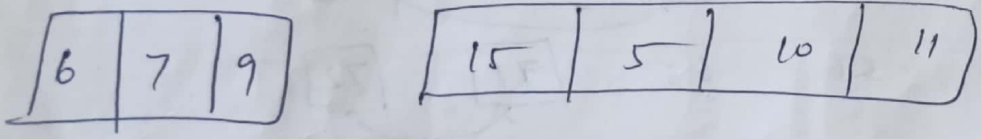


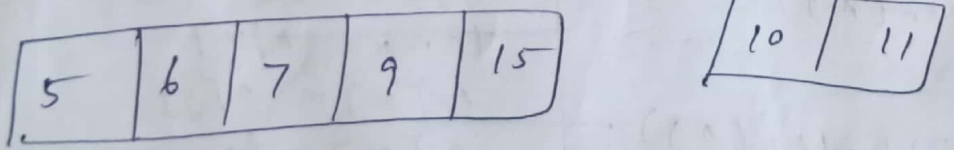
process start with first element.

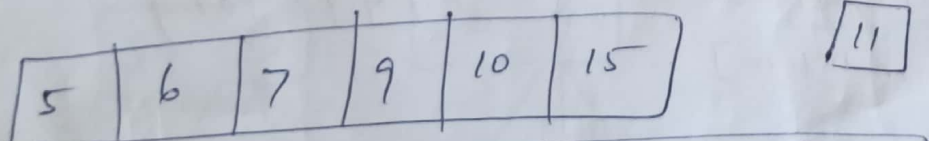


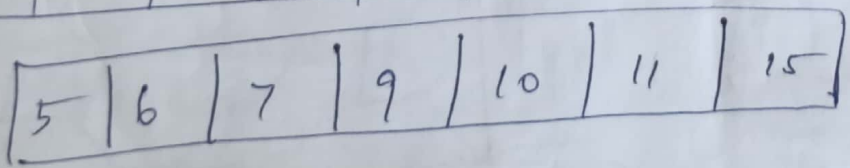
Sorted Unsorted.











Adv

- Used for small Data set.
- occupy less memory.

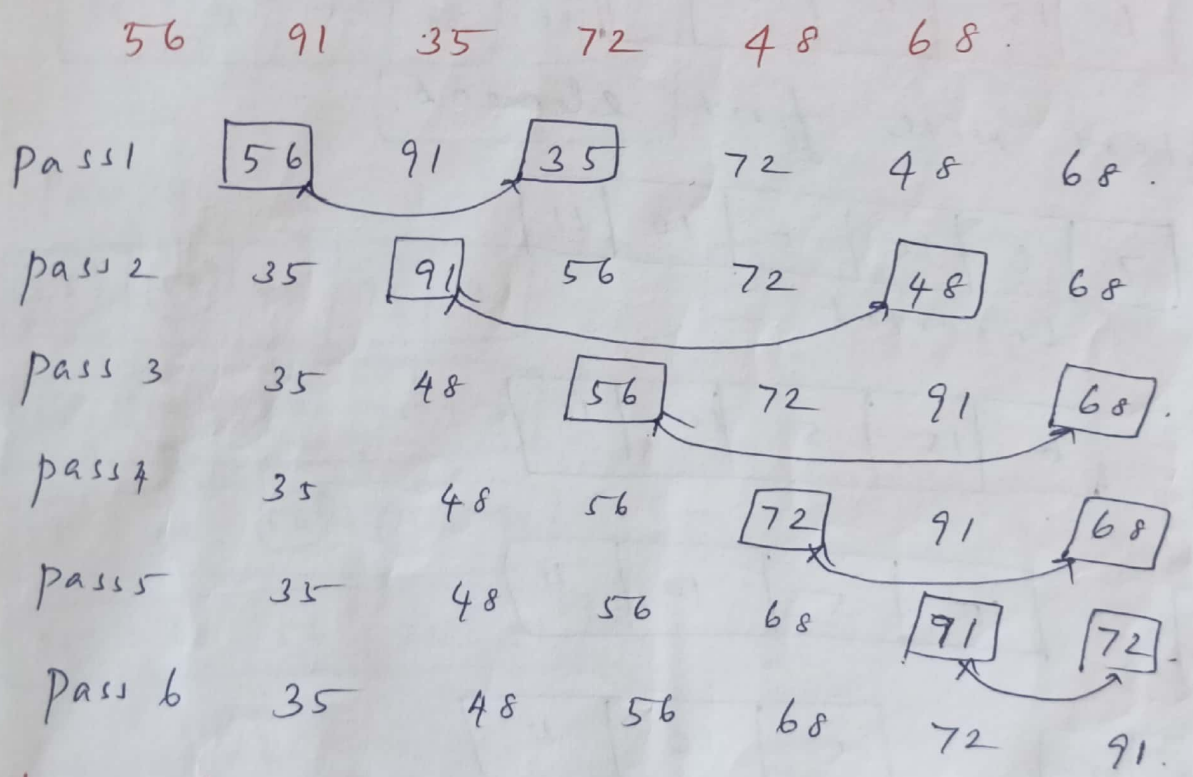
Disadv

- Inefficient for la... datasez.

Selection Sort

- Find the minimum value in the list.
- Swap it with the value in the current position.
- Repeat this process for all the elements until the entire array is sorted.

EX:




```
def SelectionSort(A):
    for i in range(len(A)):
        minimum = i
        for j in range(i+1, len(A)):
            if A[j] < A[minimum]:
                minimum = j
        Swap(A, minimum, i)
```

```
def swap(A, x, y):
    temp = A[x]
    A[x] = A[y]
    A[y] = temp
```

```
def insertion_sort(A):
    for i in range(1, len(A)):
        temp = A[i]
        k = i
        while k>0 and temp < A[k-1]:
            A[k] = A[k-1]
            k -= 1
        A[k] = temp
```

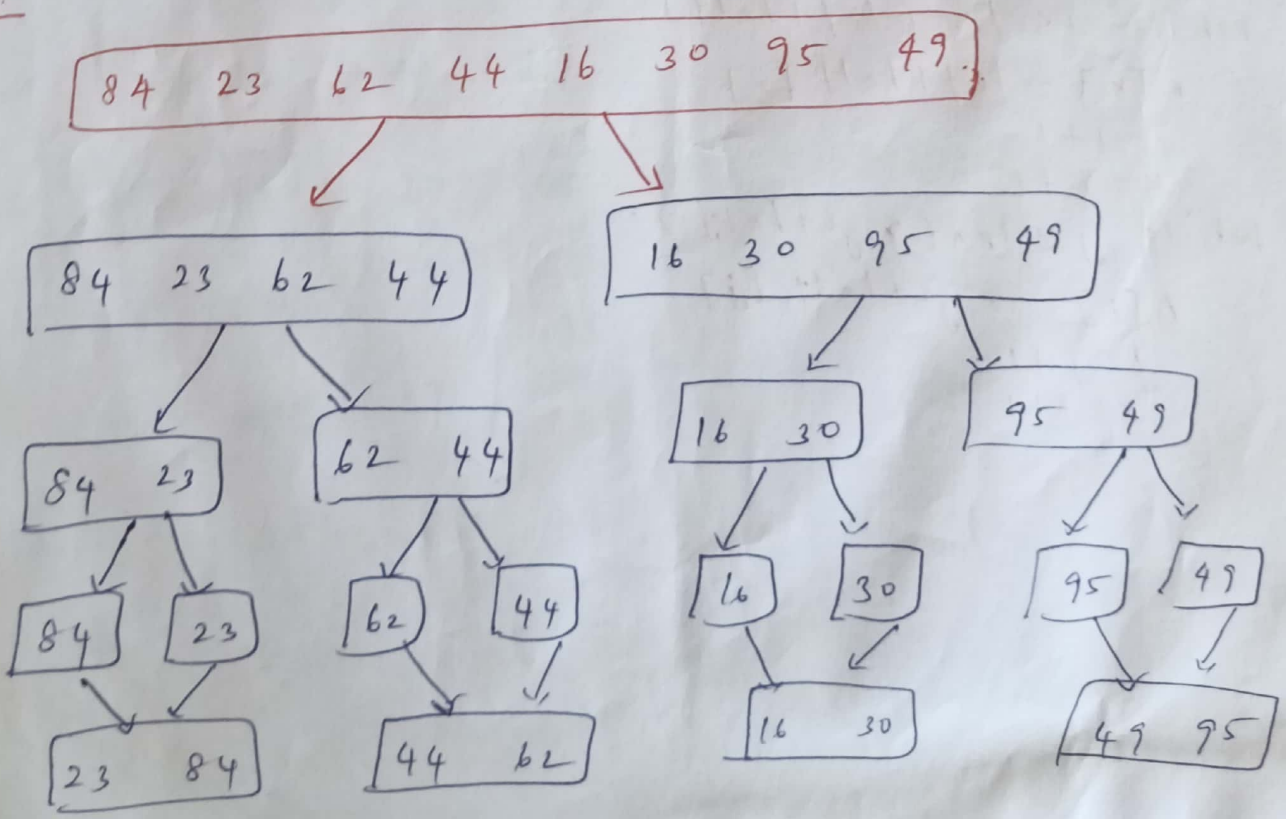
Adv

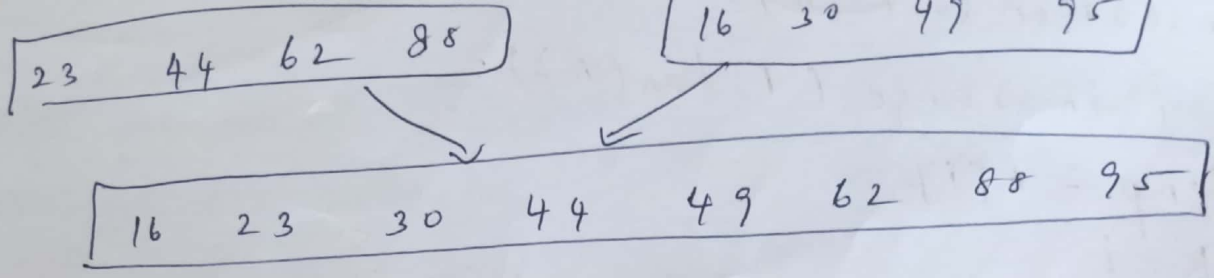
- Easy to implement
- Efficient for small data set.

Merge sort

- Divide and Conquer method.
- Divide - Break the problem into several sub problems.
- Conquer - Solve the subproblem recursively.

Ex:






```
def Mergesort(A):
    if len(A) > 1:
        mid = len(A)//2
        lefthalf = A[:mid]
        righthalf = A[mid:]
        Mergesort(lefthalf)
        Mergesort(righthalf)
        i=j=k=0.
    While i<len(lefthalf) and j<len(righthalf):
        if lefthalf[i] < righthalf[j]:
            A[k] = lefthalf[i]
            i = i+1
        else:
            A[k] = righthalf[j]
            j = j+1
        k = k+1
    While i< len(lefthalf):
        A[k] = lefthalf[i]
        i = i+1
        k = k+1
    While j < len(righthalf):
        A[k] = righthalf[j]
        j = j+1
        k = k+1
```

Quick Sort

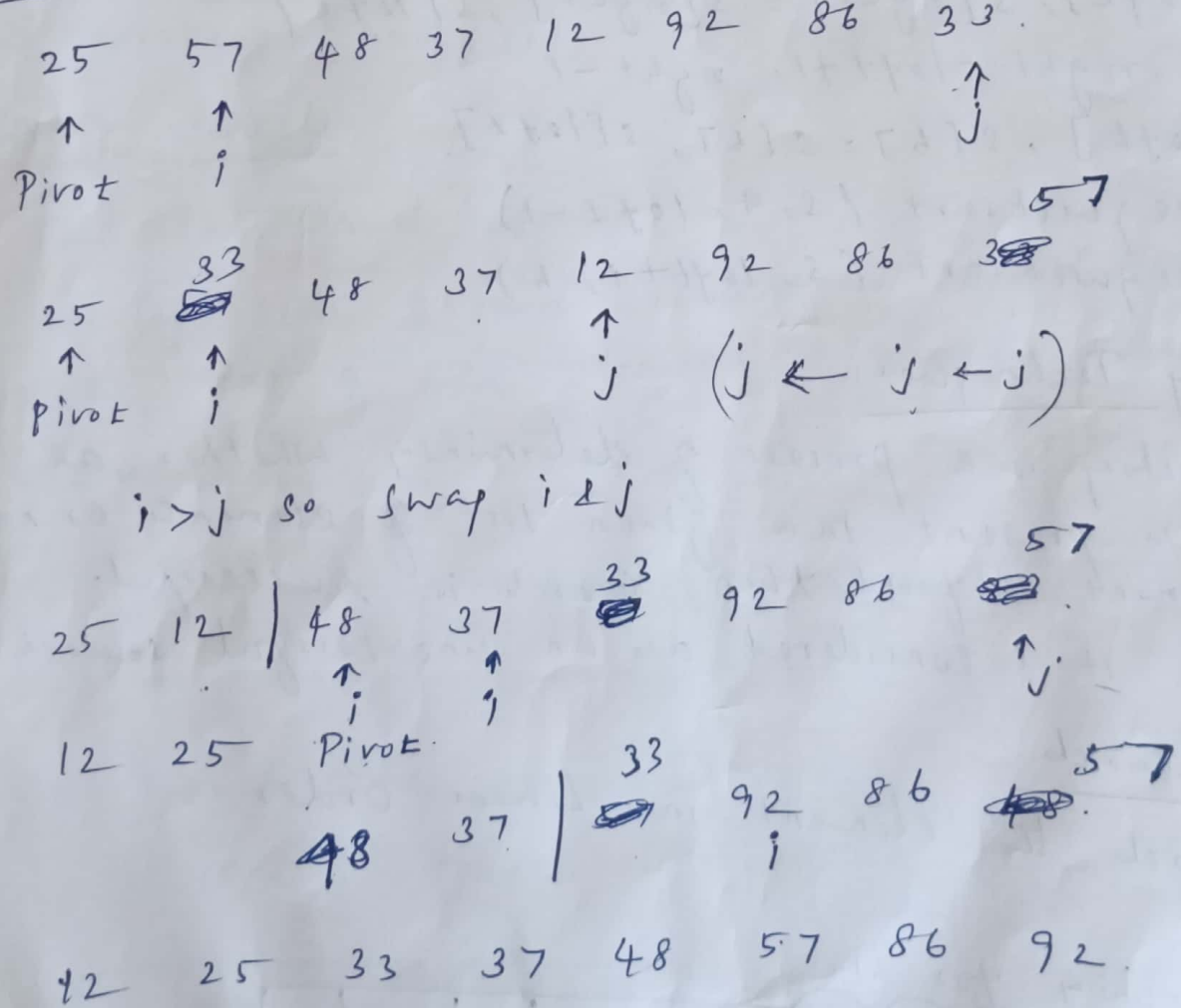
Quick sort is a Sorting algorithm that uses the divide-conquer strategy.

Divide – Split the array into two sub arrays that each element in the left sub array is less than or equal the middle element. right sub array is greater than or equal the middle element.

Conquer: Recursively sort the two sub array.

Combine: Combine all the sorted elements.

EX:



```
def inplace quicksort (S, a, b):
  if a >= b:
    return
  pivot = S[b]
  left = a.
  right = b-1
  While left <= right:
    While left <= right and S[left] < pivot:
      left += 1.
    While left <= right and pivot < S[right]:
      right -= 1.
    if left <= right:
      S[left], S[right] = S[right], S[left]
      left, right = left+1, right-1.
      S[left], S[b] = S[b], S[left].
    inplacequicksort (S, a, left-1)
    inplacequicksort (S, left+1, b)
```

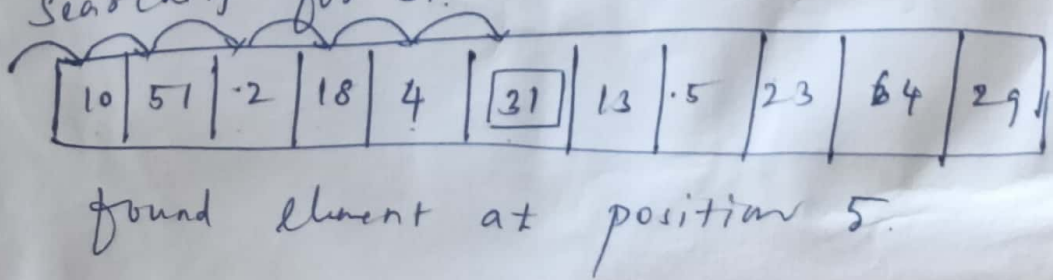
Searching Techniques:

Searching is a process of determining whether an element is present in a given list of elements or not. If the element is found, then search is successful. otherwise it is considered as an unsuccessful search.

Linear Search.

Search the element in Linear Order.

Ex:



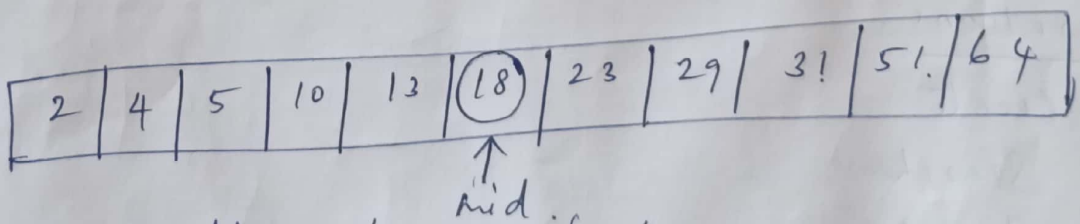
found element at position 5.

```
def linearsearch(A, key):
    found = 0
    for i in range(len(A)):
        if A[i] == key:
            found += 1
        else:
            Continue
    if found == 0:
        print("key not found")
    else:
        print("key found")
num = [10, 51, 2, 18, 4, 31, 13, 5, 23, 64]
linearsearch(num, 31)
```

Binary Search.

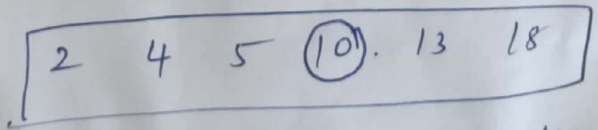
- Calculate mid element.
- ~~to~~ left side of mid element is lesser than. right side of mid element is greater than the element.

Ex:



To search the element is 10.

10 < 18 search left side of element.



10 = 10 found the element.

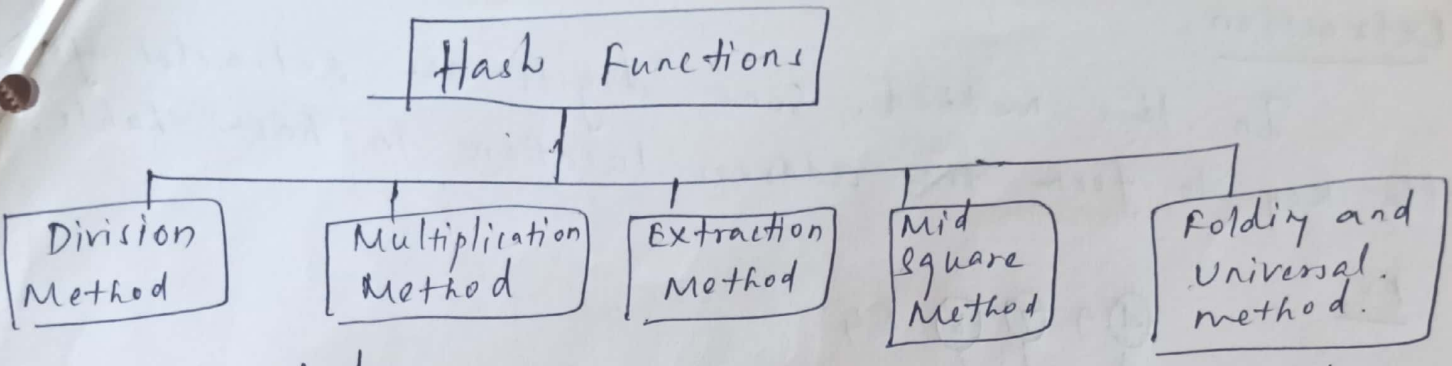
```
def binarysearch ( data, target, low, high):
    if low > high :
        return false.
    else:
        mid = (low + high) /2
    if target == data [mid]:
        return true.
    elif target < data [mid]:
        return binarysearch (data, target, low, mid-1)
    else:
        return binary search (data, target, mid +1, high)
```

Hashing Techniques.

- Hashing is the process of mapping a search key to a limited range of array.
- Hash table data structure is an array of some fixed ~~size~~ Size containing the keys and a hash function is associated with the table.
- Hash function converts or maps the search keys to specific entries in the hash table.

Operations :

1. Create
2. Search.
3. Insert
4. Delete



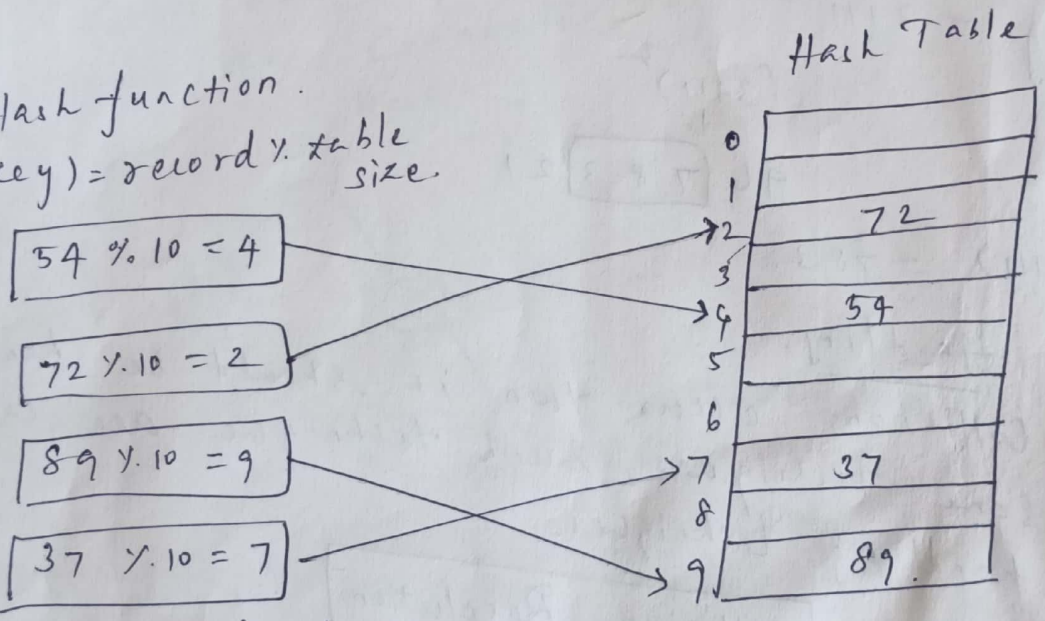
Division Method

- The hash function depends upon the remainder of division.
- Ex:

54, 72, 89, 37 insert into hash table. table size is 10.

Hash function.

h(key) = record % table size.



Multiplicative Hash Function.

Steps:

1. Multiply the key 'k' by a Constant A. Where A is in the range $0 < A < 1$.
2. Multiply this fractional part by m and take the floor.

Ex:

Key k = 107 assume m = 50

A = 0.6180339887

$h(k) = \lfloor m * 107 * 0.6180339887 \rfloor$

66.12

0.12 fractional part

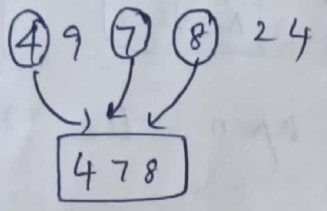
$h(k) = 50 * 0.12$

$h(k) = 6$

Extraction.

In this method, some digits are extracted from the key to form the address location in hash table.

Ex:



Mid Square

- Square the key
- Extract middle part of the result.

Ex: key = 3111

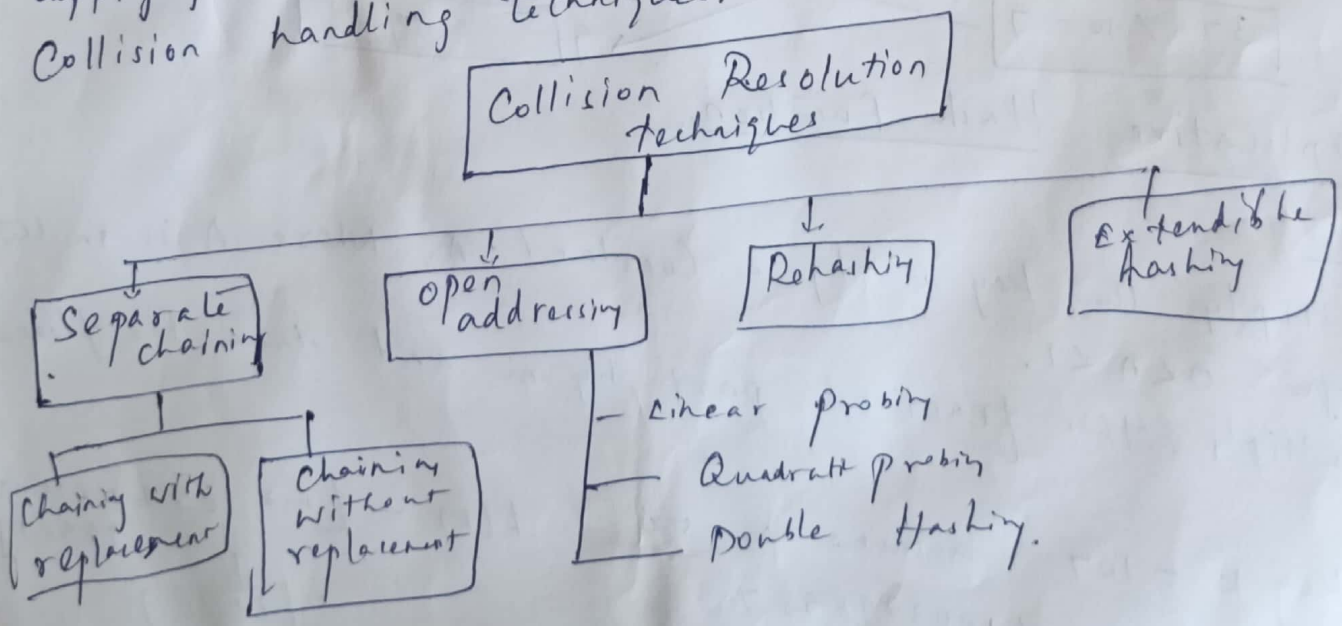
$\therefore (3111)^2$

96 | 7 8 3 | 21

H(3111) = 783.

Collision Handling.

If Collisions occurs then it should be handled by applying some techniques, such technique are called. Collision handling techniques.

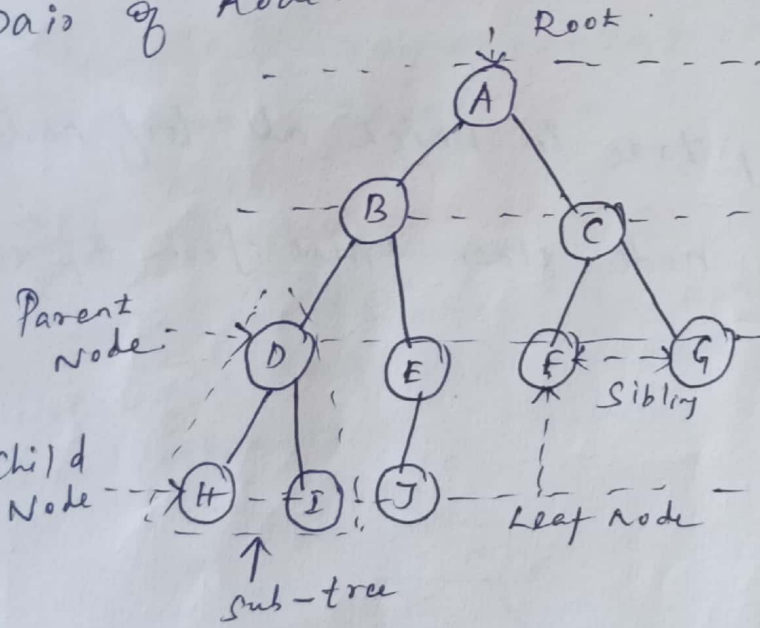


UNIT-IV TREE STRUCTURES.

Tree ADT – Binary Tree ADT – tree traversals – binary Search trees – AVL trees – heaps – multi-way Search tree.

Tree ADT

- A tree abstract data type Consists of nodes and edges that organize data in a hierarchical fashion.
- The data elements are stored in nodes and pair of nodes are Connected by edges.



Binary Tree.

A Binary tree is an ordered tree with the following properties.

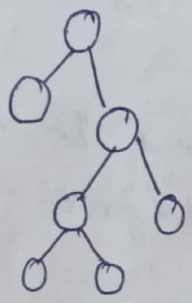
1. Every node has at most two children.
2. Each child node is labelled as being either a left child or a right child.

Types of Binary Tree

1. Proper Binary Tree or Strict Binary Tree

A Binary tree is called proper or strict binary tree, if each node has exactly zero children or two children.

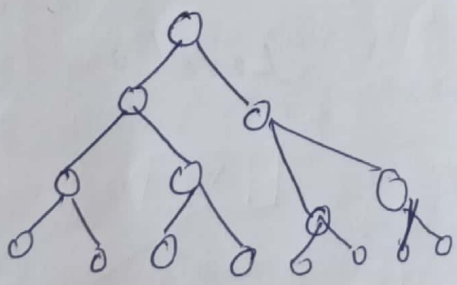
Ex:



2. Perfect Binary Tree.

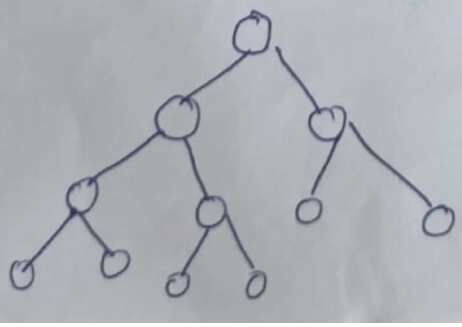
- It is a full binary tree in which all leaf nodes are at the same level.
- It has all possible node slots filled from top to bottom with no gaps.

Ex:



3. Complete Binary Tree

A Complete Binary Tree is a binary Tree in which every level, except possibly the last, is completely filled, from left to right leaving no gaps.



Properties of Binary Tree.

~~1. Binary tree have~~

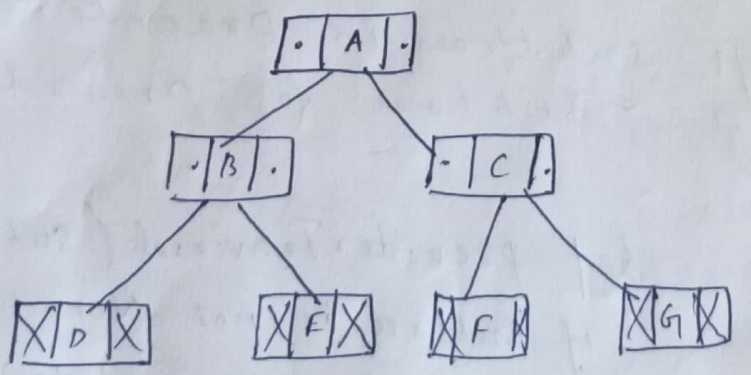
1. The number of nodes in a full binary tree is $2^{h+1} - 1$.
2. Complete Binary tree is between 2^h and $2^{h+1} - 1$.
3. The number of leaf nodes in a full binary tree is 2^h.
4. The number of NULL link in a Completed binary tree of n nodes is $n+1$.

Implementations of Binary Tree.

1. Linked representation of a binary Tree.
2. Array Based " " "

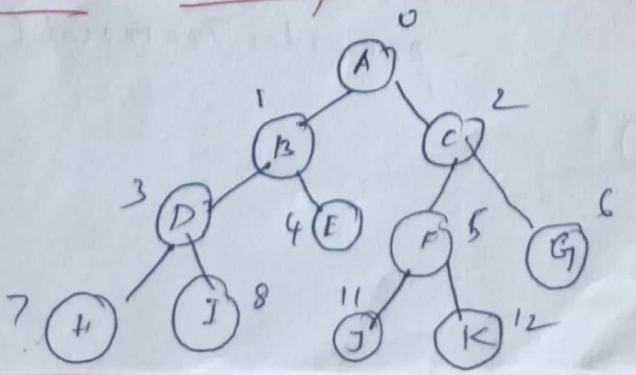
1. Linked Representation.

- It is dynamic, because memory is dynamically allocated ie when it is needed and it is efficient and avoids wastage of memory space.




```
class Node:
    def __init__(self, element):
        self.element = element
        self.left = None
        self.right = None
```

2. Array Based Representation.



A	B	C	D	E	F	G	H	I	J	K
0	1	2	3	4	5	6	7	8	11	12

Applications of Binary Tree

1. Expression trees are used in Compilers.
2. Tree ~~are~~ used in ~~Hiral~~ ~~Hierarchi~~ Hierarchical DBMS.
3. It support search, insertion and deletion.

Tree Traversal Algorithms.

- A traversal of a tree is a systematic way of accessing or visiting all the nodes in a tree. Each node is processed only once but it may be visited more than once.
- There are 3 traversal Algorithm.

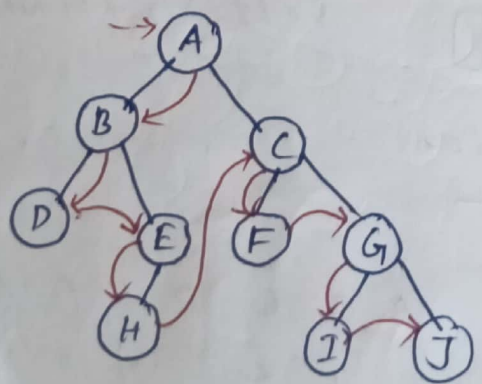
1. preorder.
2. Inorder.
3. post order.

1. preorder.

Algorithm.

1. Visit the root.
2. Traverse the left subtree in preorder.
3. Traverse the right subtree in preorder.

Ex:



A B D E H C F G I J.

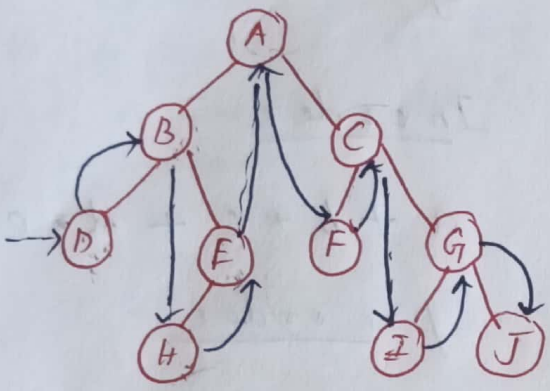
```
def preorderTraversal(subtree):
    if subtree is not None:
        print(subtree.data)
        preorderTraversal(subtree.left)
        preorderTraversal(subtree.right)
```

2. Inorder Traversal.

Algorithm

→ Traverse the left subtree in inorder.
→ Visit the root.
→ Traverse the right subtree in inorder.

Ex:



DBHEAFCIGJ

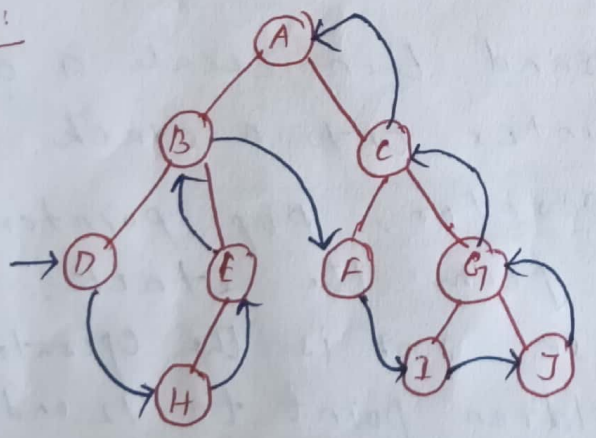
```
def inorderTraversal(subtree):
    if subtree is not None:
        inorderTraversal(subtree.left)
        print(subtree.data)
        inorderTraversal(subtree.right)
```

3. postorder Traversal.

Algorithm

→ Traverse the left subtree.
→ Traverse the right subtree.
→ visit the root.

Ex:



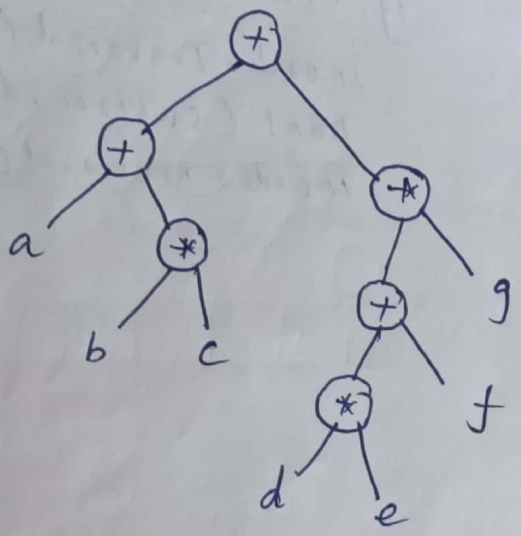
DHEBFIJGCA.

```
def postorderTraversal(subtra):
    if subtree is not None:
        postorderTraversal(subtra.left)
        postorderTraversal(subtra.right)
        print(subtree.data).
```

Expression tree.

Expression tree is a binary tree in which leaf nodes are operands and the interior nodes are operators.

Ex: $(a+b*c)+((d*e+f)*g)$



Inorder.

a + b * c + d * e + f * g

preorder

+ + a * b c * + * d e f g

postorder

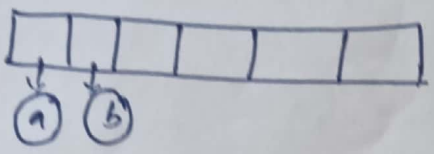
a b c * + d e * f + g * +

Constructing an Expression tree

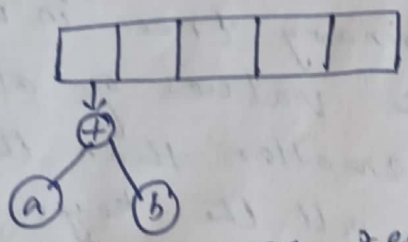
1. Convert Infix to postfix expression.
2. Read one symbol at a time from the postfix expression.
3. If the symbol is an operand then create a one-node tree and push a pointer onto a stack.
4. If the symbol is an operator, pop pointers to two trees T_1 and T_2 from the stack and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1. A pointer to this new tree is then pushed onto the stack.

Example: Input : a b + c d e + * *

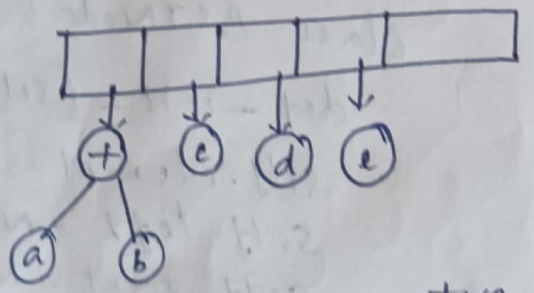
1. The first two symbols are operands, so we create one-node trees and push pointers to them onto a stack.



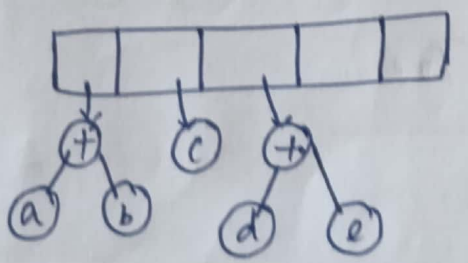
2. Next '+' is read, so two pointers from the trees are popped, a new tree is formed and its pointer is pushed on to the stack.



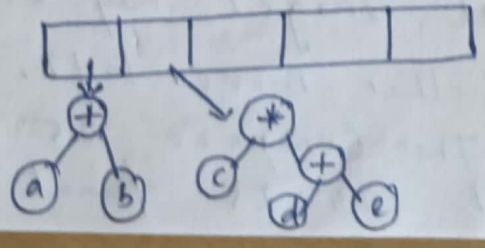
3. Next c, d, e are read and for each operand one node tree is created and a pointer to the corresponding tree is pushed onto the stack.



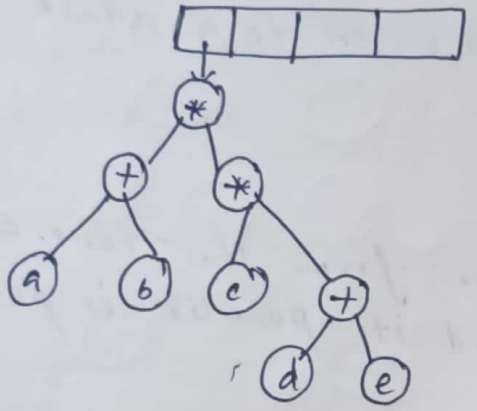
4. Now '+' is read. So two tree are merged.



5. Now '*' is read. So we pop two tree pointers and form a new tree with a * as root.



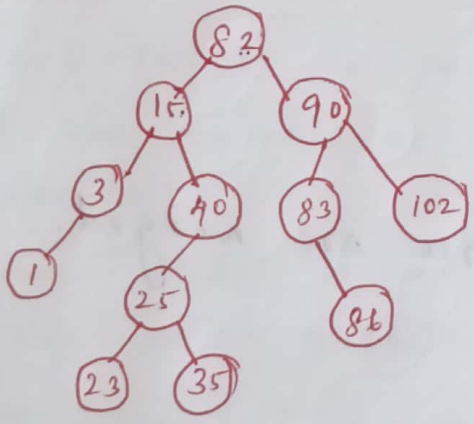
6. Finally, a last symbol is read, two trees are merged and pointer to the final tree is,



Binary Search tree.

Binary Search tree is a binary tree in which for every node x in the tree, the values of all the key 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 greater than the key value in x.

Ex:




```
class BSTNode:
    def __init__(self, element):
        self.element = element
        self.left = None.
        self.right = None.
```

Operations on Binary Search Tree.

1. find.

Ex: To search a key value 25 in the binary search tree. Start Comparing with target key 25 to root node 82. Since 25 < 82, then move left. Then Compare 15 ~~15~~. then move right. Then Compare 25 < 40 move left. 25 = 25 found
25 > 15

Find Min.

In binary search tree the minimum element is the left most node, which does not have any left child.

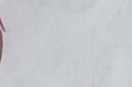
3. Find Max.

In binary search tree the maximum element is the right most node, which does not have any right child.

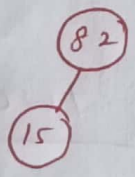
4. Insertion.

Ex: Build a binary search tree for the key list 82, 15, 90, 40, 3, 83.

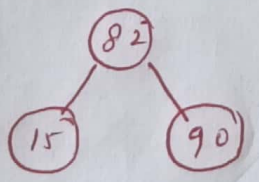
1. Insert 82.



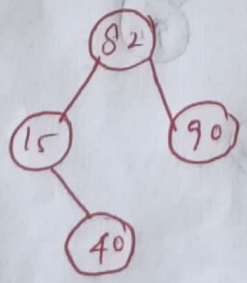
2. Insert 15



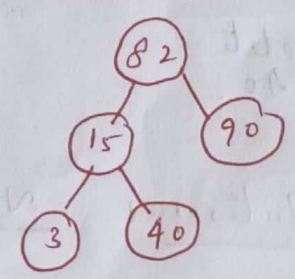
3. Insert 90



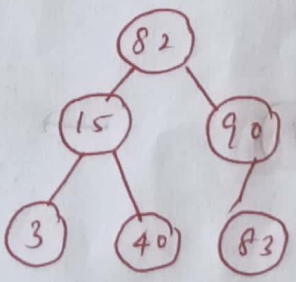
4. Insert 40



5. Insert 3



6. Insert 83.



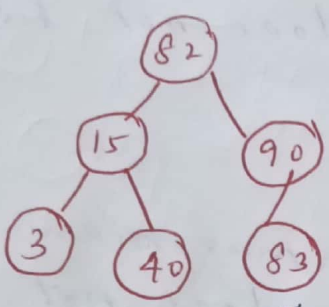
5. Deletion

There are three cases in the deletion.

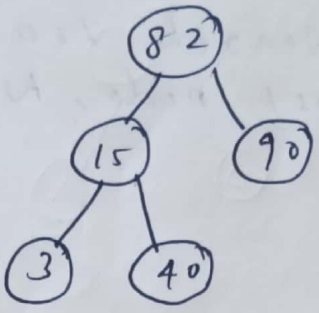
1. The node is a leaf.
2. The node has a single child.
2. The node has a two children.

Deleting Leaf node.

Ex:

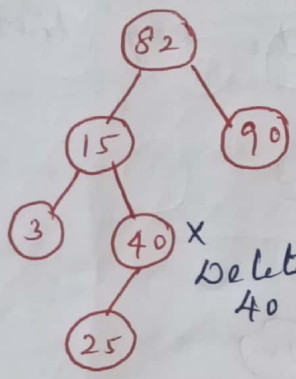


× Delete it.



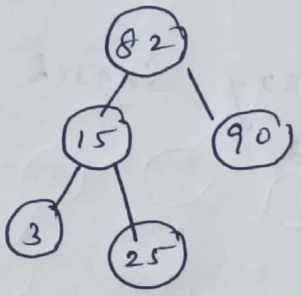
Deleting an Interior Node with one child.

Ex:



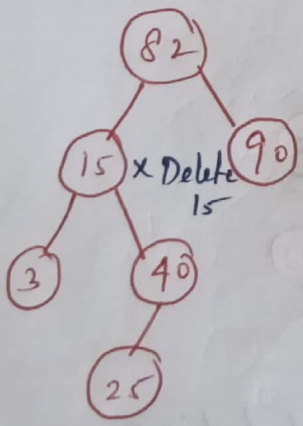
× Delete 40.

→



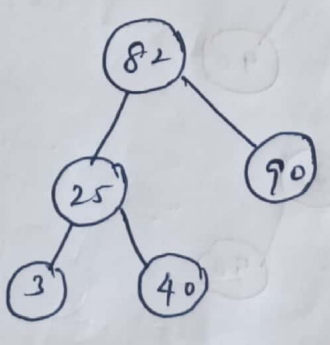
Deleting an Interior Node with two children.

Ex:



× Delete 15

→

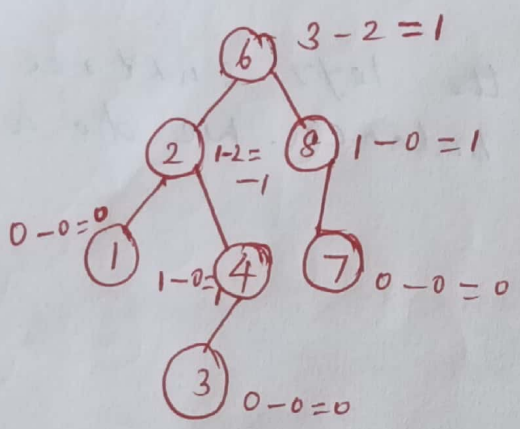


AVL Tree.

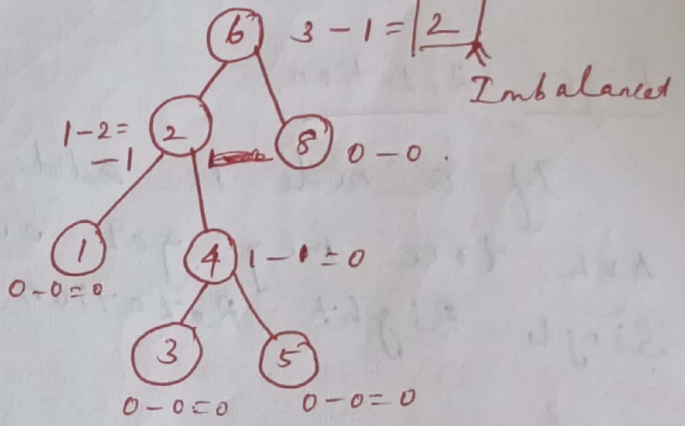
- AVL Tree was invented by G.M. Adelson-Velskii and Y.M. Landis in 1962.

- A binary tree is balanced if the heights of the left and right subtree of every node differ by at most 1.

Ex:



AVL Tree



Not AVL Tree.

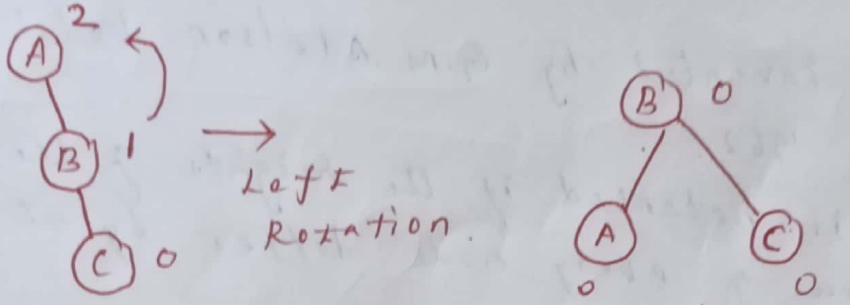
Insertion.

It Consists of four cases.

1. Left Rotation. } - single Rotation.
2. Right Rotation. }
3. Left-Right Rotation } - Double Rotation.
4. Right-Left Rotation }

1. Left Rotation.

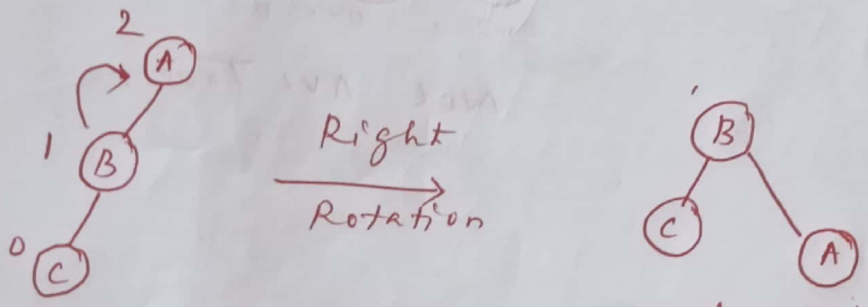
When a node is added into the right sub tree, ~~of the right~~ if the tree gets out of balance, we do a single left rotation.



UnBalanced. Balanced.

Right Rotation.

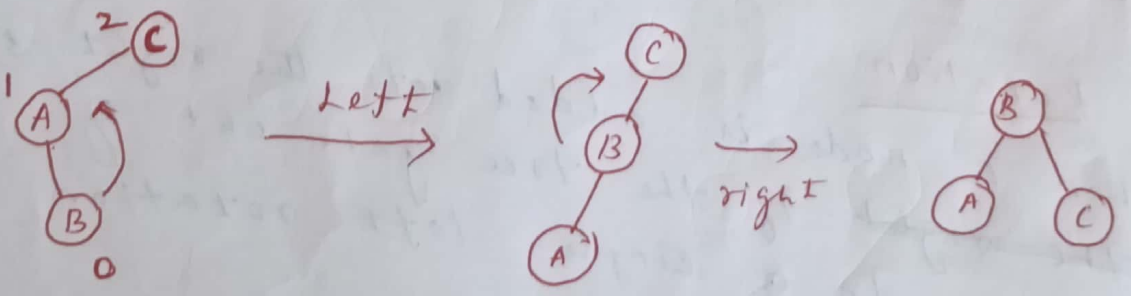
If a node is added to the left subtree, AVL tree may get out of balance. We do a Single Right Rotation.



Unbalanced Balanced.

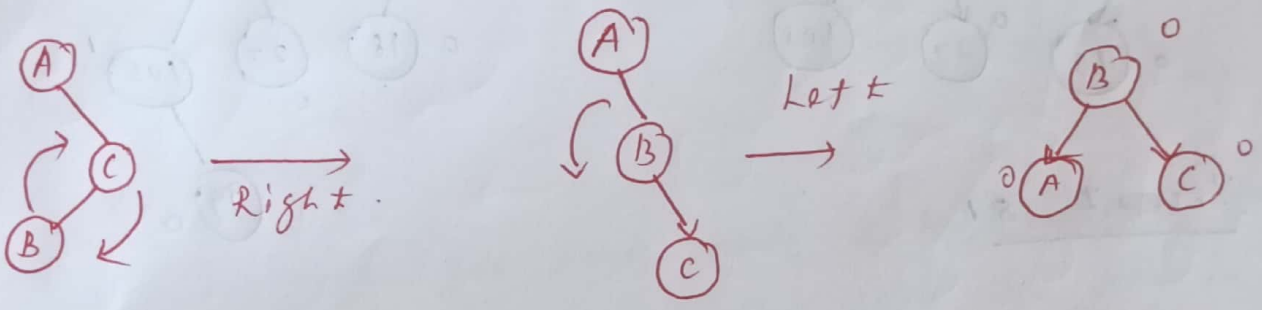
Left-Right Rotation.

A Left-right Rotation is a Combination in which first left rotation takes place after that right rotation executes.



ght - Left Rotation:

Right-left rotation is a combination in which first right rotation takes place after that left rotation executes.

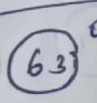


Example:

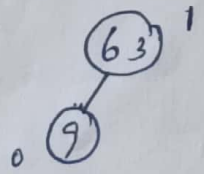
Construct an AVL Tree by inserting the following elements in the given order.

63, 9, 19, 27, 18, 108, 99, 81.

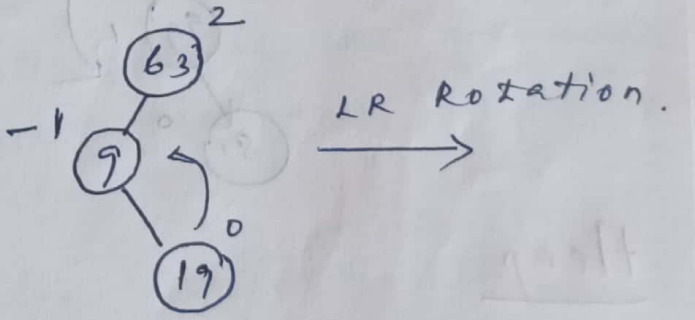
1. Insert 63

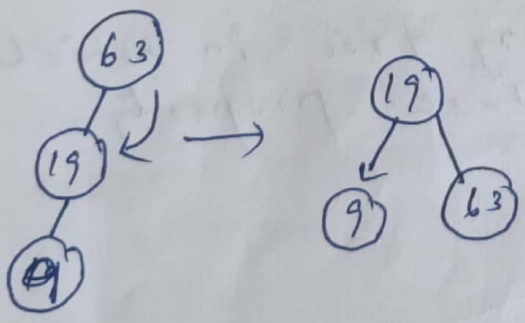


2. Insert 9

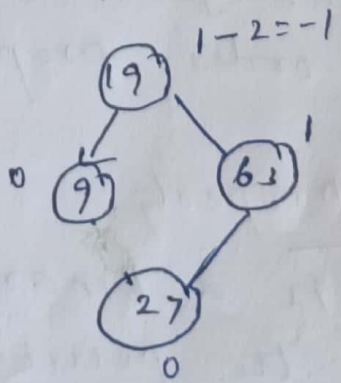


3. Insert 19.

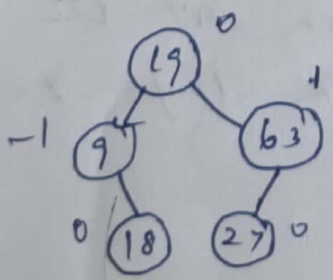




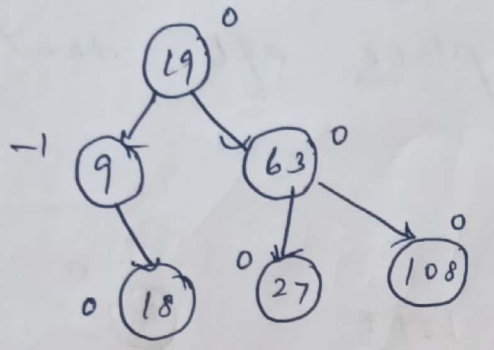
4. Insert 27.



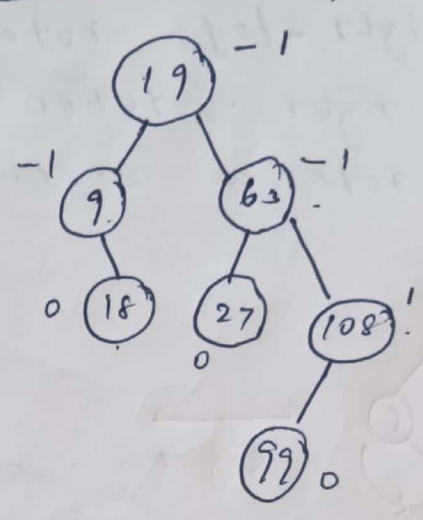
5. Insert 18.



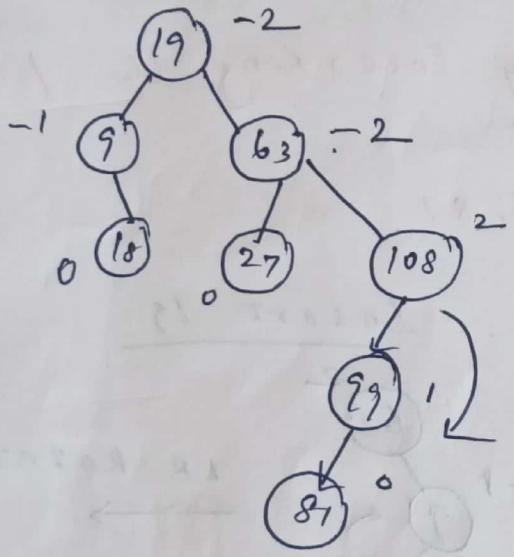
6. Insert 108.



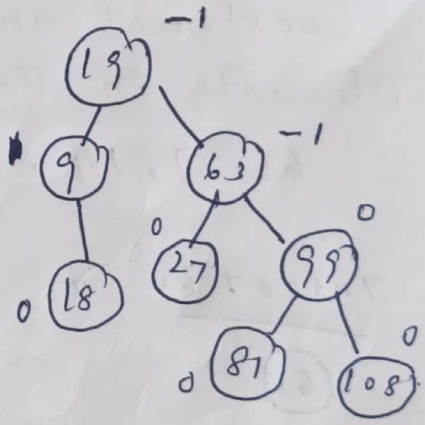
7. Insert 99.



8. Insert 81



LL Rotation



Heap

A Heap is a Complete binary tree in which every node satisfies a structure property and a heap order property.

structure property

A heap is a binary tree that is completely filled with the possible exception at the bottom level, which is filled from left to right.

Heap order property

Heap can be classified as, Max heap and Min heap.

Max heap

In a Max-heap, for every node x in a tree, the key in the parent of x is greater than the key in x.

Min heap

In Min-heap, for every node x in a tree, the key in the parent of x. is lesser than the key in x.

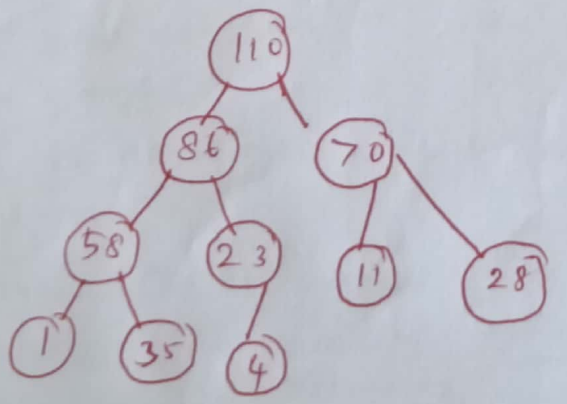
Insertion.

When a new value is inserted into a heap, the heap order property and structure property must be maintained.

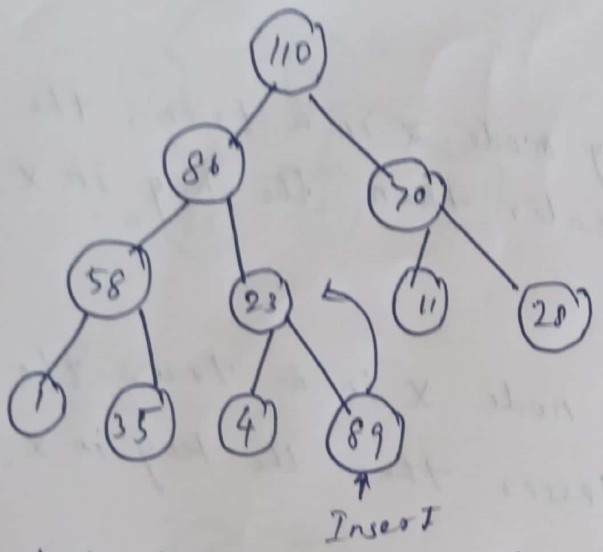
Algorithm:

1. Create a new node and fill it with the new value.
2. The node is then attached as a leaf node at the only spot in the tree.
3. To restore the heap order property, the new value has to move up along the path in reverse order towards root until a node is positioned properly.

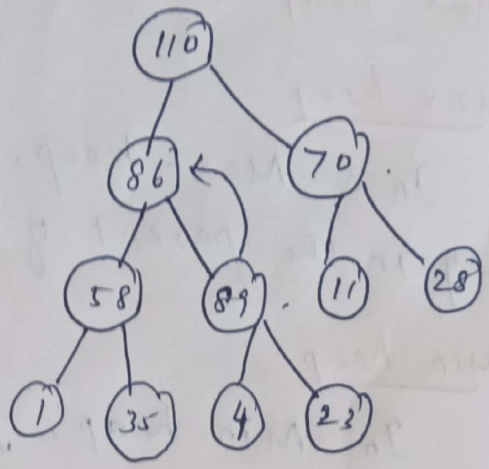
EX:

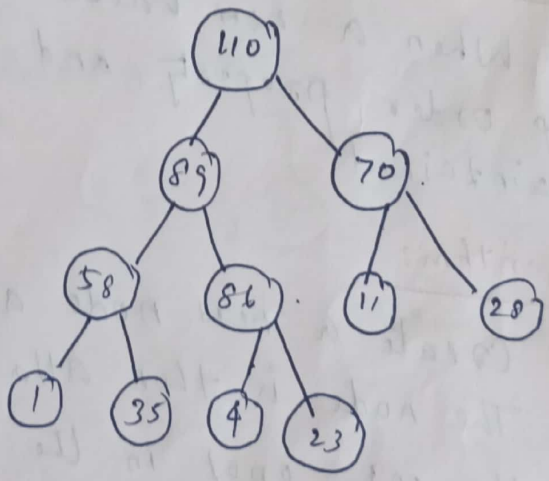


To insert 89 in the above max heap, create a new node and it is inserted in the next available location.



Not Satisfy heap order property





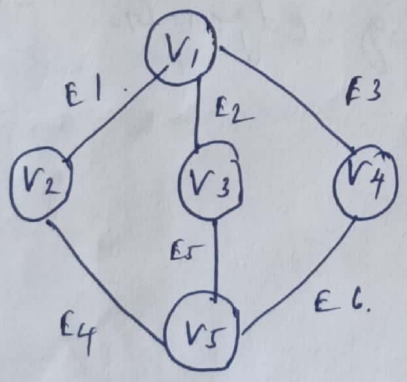
UNIT-V GRAPH STRUCTURES.

Graph ADT - Representation of graph - graph traversals - DAG - topological ~~sorting~~ Ordering - shortest paths - minimum spanning tree.

Graph - Definition.

- A Graph is a Collection of two sets V and E. Where V is a finite non empty set of vertices and E is a finite non empty set of edges.
- Vertices are nothing but the nodes in the graph.
- Two adjacent vertex are joined by edges.
- graph is denoted by $G = \{V, E\}$.

Ex:



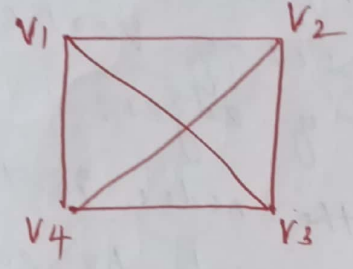
Comparision between Graph & tree.

Graph	Tree
1. Graph is non Linear DS.	Tree is non Linear DS.
2. It is Collection of vertices and edges.	It is a collection of vertices & edges.
3. Each node can have any number of edge.	In binary tree every node can have at the most two child nodes.
4. There is no unique node like tree.	There is unique node called root.
5. A cycle can be formed.	There will not be any cycle.
6. Applications: For finding shortest path in networking graph is used.	Application: Expression tree, game tree.

Basic Terminologies:

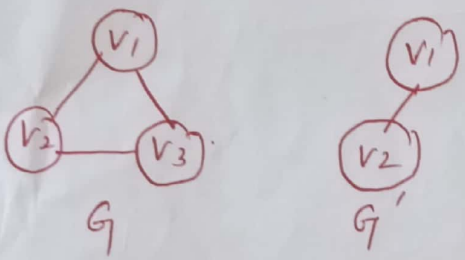
1. Complete Graph.

If an undirected graph of n vertices consist of $n(n-1)/2$ number of edges then it is called a Complete Graph.



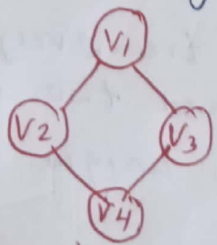
2. Sub graph.

A Subgraph G' of graph G is a graph such that the set of vertices and set of edges of G' are proper subset of the set of edges G.



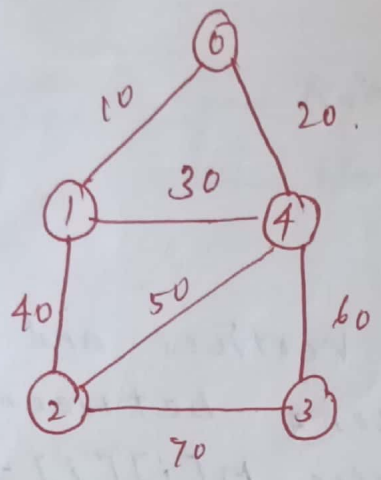
3. Connected Graph.

An undirected graph is said to be connected if for every pair of distinct vertices v_i and v_j in V(G). there is a graph from v_i to v_j in G.



4. Weighted Graph.

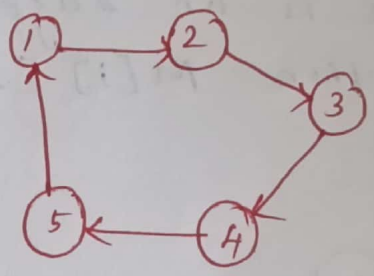
It is a graph which Consists of weights along with its edges.



5. Path:

A path is denoted using sequence of vertices and there exists an edge from one vertex to next vertex.

Ex:

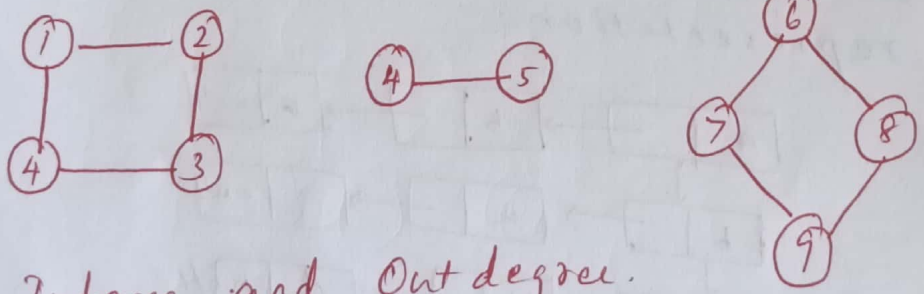


6. Cycle.

A closed walk through the graph with repeated vertices, mostly having the same starting & ending vertex is called cycle.

7. Component

The maximal connected subgraph of a graph is called Component of a graph.



8. Indegree and Outdegree.

- Indegree of a vertex is the number of edges that incident to that vertex.
- Outdegree of the vertex is total number of edges that are going away from the vertex.

9. Self Loop

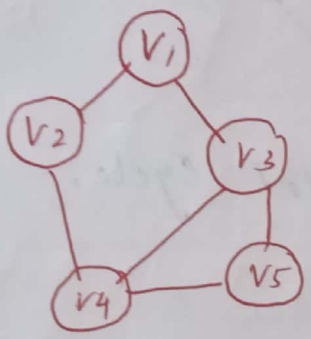
It is an edge that connects the same vertex itself.

Representations of Graph.

1. Adjacency Matrix Representation.
2. Adjacent List Representation.

1. Adajacency Matrix

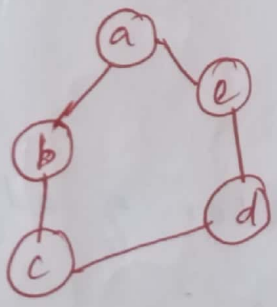
Consider a graph G of n vertices and the Matix M. If there is an edge present between vertices Vi and Vj then M[i][j]=1 else M[i][j]=0.

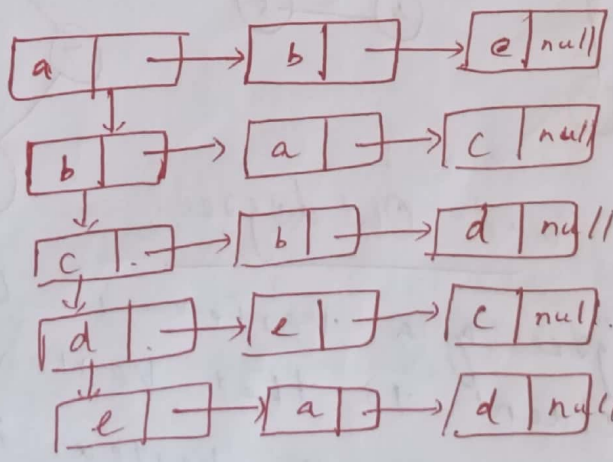


	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	1
4	0	1	1	0	1
5	0	0	1	1	0

2. Adjacency List

The type of representation of a graph. in which the linked list is used. is called. Adjacency list representation.

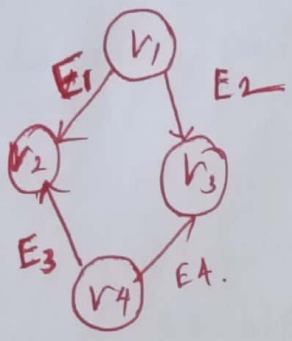




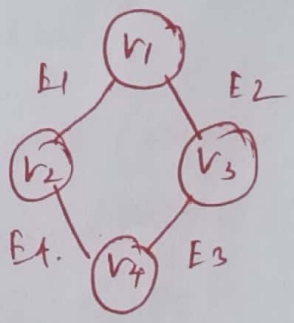
Types of Graph

2 Types
- Directed Graph.
- Undirected Graph.

Directed Graph.



Undirected Graph.



Graph : Traversals.

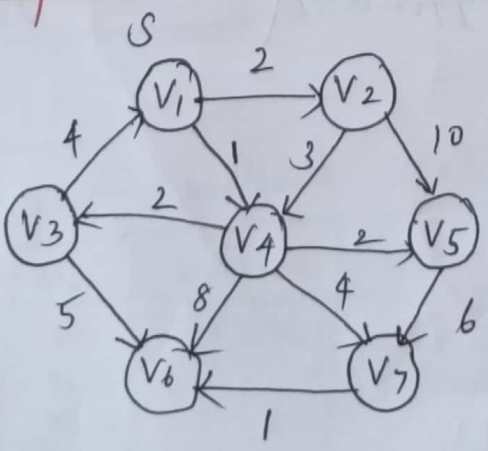
Breadth First Traversal.

Shortest path Algorithm.

- Shortest path problem is finding a path between two vertices in a graph. such that the total sum of the edge weight is minimum.
- Dijkstra's algorithm is used for weighted shortest path algorithm.

Algorithm Shortestpath (G, s):
Initialize d[s] = 0 and d[v] = ∞ for each vertex v ≠ s.
Let a priority Queue Q Contain all the vertices of G using the D labels as keys.
while Q is not empty do.
    u = value returned by Q.remove min().
    for each vertex v adjacent to u such that v is in Q do.
        if d[u] + w(u,v) < d[v] then
            d[v] = d[u] + w(u,v).
            Change to d[v] the key of vertex v in Q.
    return the label d[v] of each vertex v.

Example:



Step 1:

V_1 is taken as source node.

V	Known	d_v	P_v
V_1	0	0	0
V_2	0	∞	0
V_3	0	∞	0
V_4	0	∞	0
V_5	0	∞	0
V_6	0	∞	0
V_7	0	∞	0

Known - Status Va
if it is proce
set 1 otherwis
0.

d_v - distance from
source to destina

P_v is a book keepin
variable.
actual path.

Step 2:

Now V_1 is known. So vertex V_1 is marked as 1.

Its adjacent ~~matrix~~ Vertices V_2 and V_4.

V	Known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	0	0
V_4	0	1	V_1
V_5	0	0	0
V_6	0	0	0
V_7	0	0	0

Step 3:

Select the Vertex with minimum distance ie V_4.

So it is marked as known. Its adjacent V_5, V_7, V_3, V_6.

V	Known	d_v	P_v
V_1	1	0	0
V_2	0	2	V_1
V_3	0	3	V_4
V_4	1	1	V_1
V_5	0	3	V_4
V_6	0	9	V_4
V_7	0	5	V_4

Step 4:

Select the vertex with minimum distance is v_2. So it is masked as known. Its adjacent vertex is

v_5:

V	known	dv	Pv
v_1	1	0	0
v_2	1	2	v_1
v_3	0	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	9	v_4
v_7	0	5	v_4

Step 5:

Next minimum vertex v_3 & v_5. Adjacent vertex of v_3 is v_6.

V	known	dv	Pv
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	0	3	v_4
v_6	0	8	v_3
v_7	0	5	v_4

Step 6:

Next Smallest Vertex is v_5. Adjacent vertex is v_7.

V	known	dv	Pv
v_1	1	0	0
v_2	1	2	v_1
v_3	1	3	v_4
v_4	1	1	v_1
v_5	1	3	v_4
v_6	0	8	v_3
v_7	0		

Step 7:

Next selected node is V_7. Its adjacent vertex V_6.

V	Known	dv	Pv
V_1	1	0	0
V_2	1	2	V_1
V_3	1	3	V_4
V_4	1	1	V_1
V_5	1	3	V_4
V_6	0	6	V_7
V_7	1	3	V_4

Step 8:

Next node is V_6. No adjacent vertex.

Shortest path is,

$V_1 \rightarrow V_2 = 2$

$V_1 \rightarrow V_3 = 3$

$V_1 \rightarrow V_4 = 1$

$V_1 \rightarrow V_5 = 3$

$V_1 \rightarrow V_6 = 6$

$V_1 \rightarrow V_7 = 5$.

Time Complexity $= O|E| + |V|^2$

$= O(|V|^2)$

Dis Adv:

→ Does a blind search thereby wasting time & necessary resources.

→ It cannot handle graph with negative edges.

Minimum Spanning tree.

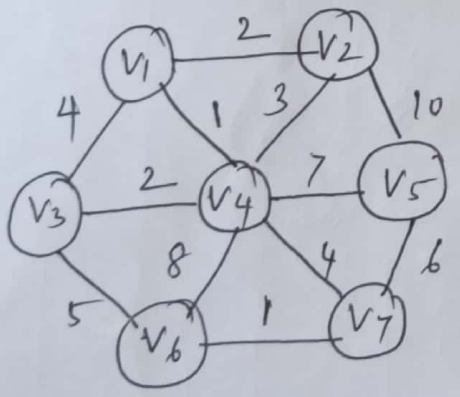
- A tree, that Contains every Vertex of a Connected Graph G is said to be a spanning tree and the problem of Computing a spanning Tree T with smallest total weight is known as the minimum spanning tree (MST).
- It Consists of 2 types.

1. Prim's Algorithm.
2. Kruskal's Algorithm.

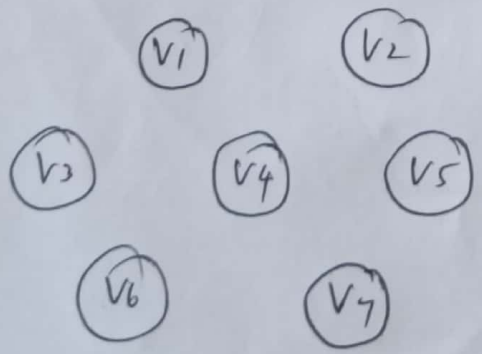
1. Prim's Algorithm.

It is used to find minimum spanning tree from a single root vertex u.

Example:

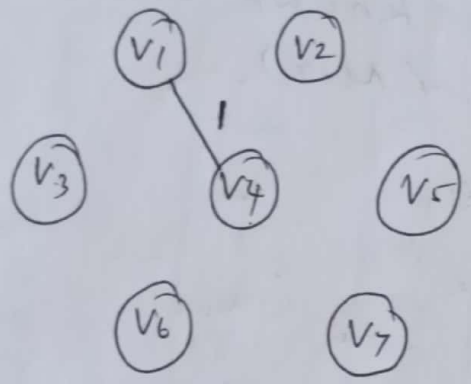


Step 1 Initial Configuration.



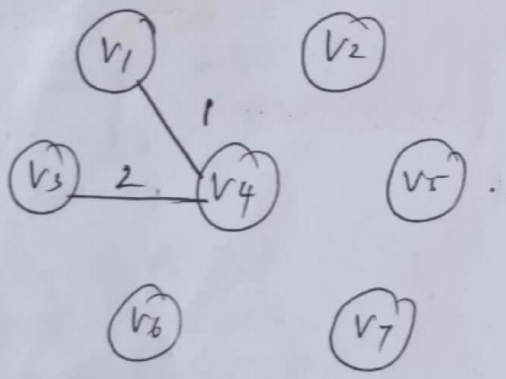
Step2:

Start with vertex V_1. V_1 adjacent node is V_3, V_4, V_2.



Step3

V_4 adjacent node is V_3, V_5, V_2. Select minimum value is V_3.



step 4: