Introduction to Data Structures

You can follow me on -
LinkedIn - @Alok Choudhary
X - @AlokChoudh78331
GitHub - @alokchoudhary05

Why?

- Raise leval of Programming
- Efficient Programming
- Able to Solve Complex Problem.
- Campus placement
- A. Grade Company Placements.

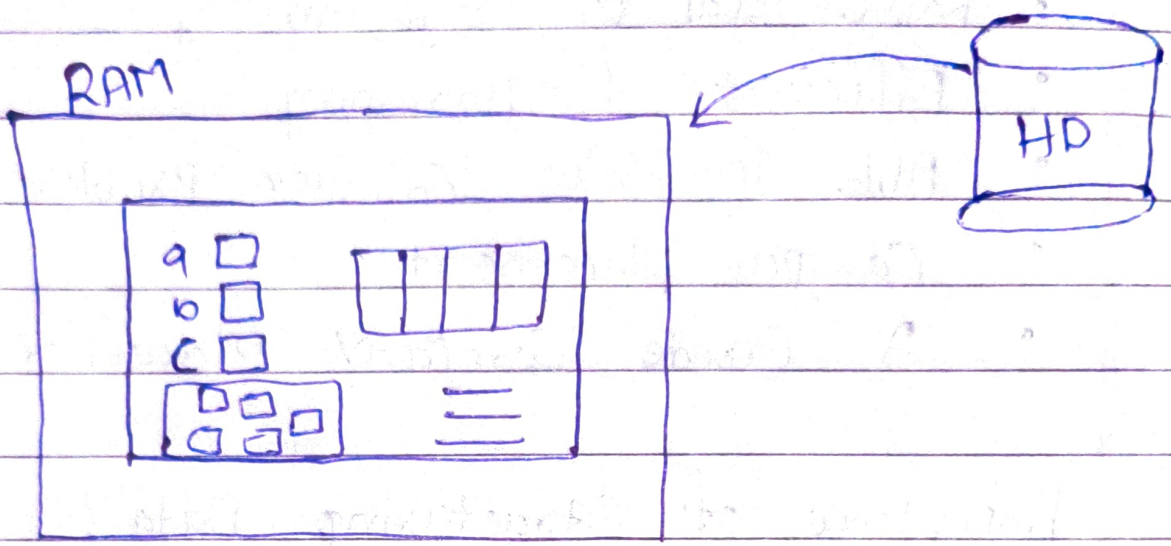
Importance of Structuring Data?

eg:- 1. Dictionary - Fastly, Not take large time.
(Arrangement of Data)

2. Map
3. Ledger
4. Almira

→ Data Structure is a Particular way of storing and organizing data in a Computer so that it can be used efficiently.

Where are data Structures resides?



Classification of Data Structure.

1. Linear data Structures

   Array, dynamic array, linked list, stack, queue deque, etc.

2. Non-Linear data Structures

   BST, AVL, B-Tree, B+Tree, graph etc.

Algorithm

An algorithm is the step by step, linguistic representation of logic to Solve a given Problem.

Array and list

~~Difference.~~

~~Array~~	~~List~~
built-in module	— builtins
These are several classes defined in builtins module	— int, float, str, list, tuple, dict, set, range, bytes, object, Exception, ----
Another module	— Array

→ Array is not a built-in data structure and therefore need to be Imported.

Array

This module defines an object type which can efficiently represent an array of basic values. characters, Integers, floating point numbers.

Arrays are Sequence types and behave very much like lists, but arrays can have elements of limited types.

Creating Array

(less use)

```
from array import *
a1 = array (type code , [elements])
```

type code	C TYPE
'b'	Signed Integer
'B'	Unsigned Integer
'u'	Unicode Characters
'h'	Signed Integer
'H'	Unsigned Integer
'i'	Signed Integer
'I'	Unsigned Integer
'l'	Signed Integer
'L'	Unsigned Integer
'q'	Signed Integer
'Q'	Unsigned Integer
'f'	Floating Point
'd'	Floating Point

Example

```
from array import *
a1 = array ('i' , [5, 10, 15, 40])

for i in range (4):
    print (a1[i], end = ' ')
```

o/p = 5, 10, 15, 40

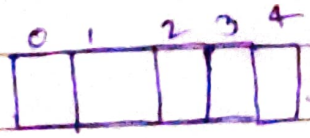
array methods

append ()
count ()
extend ()
fromlist ()
index ()
insert ()
POP ()
remove ()
reverse ()
to list ()

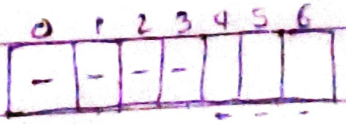
List

list is a class
list is a mutable
list element are indexed
list is an iterable
list can grow (dynamic array)
list can contain different type elements.

array
- collection of same type element.
- Fixed size
- Indexed



Dynamic array.
- Collection of same type element
- Resizable
- Indexed



Create list object.

```
l1 = [10, 20, 30]
l2 = [ ]
l3 = [10, 5.7, 'abc']
```

Method of list.		builtin methods
append()	insert()	len()
clear()	pop()	sum()
count()	remove()	max()
extend()	sort()	min()
index()	reverse()	sorted()

→ list and array both are growable.

→ list can contain heterogeneous data (different)

→ array can contain homogeneous data. (one type)

→ If you want to perform mathmatical calculations then you should use Numpy arrays by importing Numpy package.

otherwise use lists as it work in a similar way and more flexiable to work with.

Numpy

```
import numpy as np
a = np.array([1,2,3])
Print(a)

b = np.array([[1,2,3],[10,20,30]])
Print(b)
```

← 2d array

```
c = np.array([[1,2,3],[10,20])
Print(c)
```

← 1d array with two list type elements.

Class and Objects

~~class~~

~~class~~ is Noun

1. Common Noun (class)
2. Proper Noun (object)

Excapsulation

An act of Combining Properties and methods related Same object is known a Encapsulation.

Clas is a way to implement excapsulation.

Class

→ clas is an description of an object.

```
class Test:
    # attributes
```

It defines vasious attributes of an object.

Defining a class is creating a data type.

Attributes

Attributes are member variables and member functions.

```
class Test:
    x = 5
    def f1():
        # some code.
```

• x and f1 are attributes.

Object

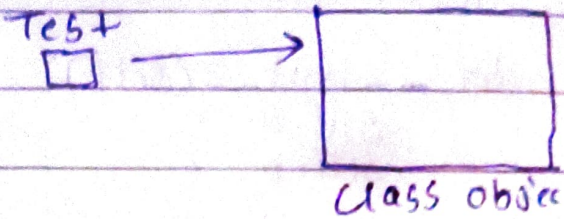
→ Object is an instance of a class.

→ objects are of two types
- clas object
- Instance object.

→ t_1 = Test() t_1 and t_2 are instance objects of
t_2 = Test() Test class.

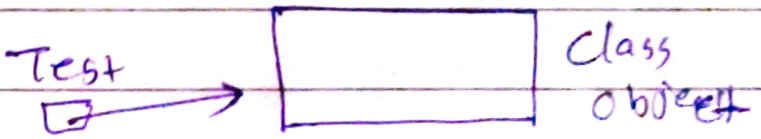
Class object

Test vs Test()

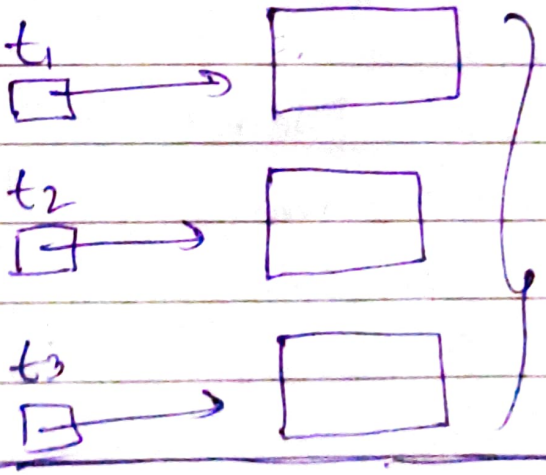


One class has exactly one class object but can have any number of Instance objects.

class Test:
=



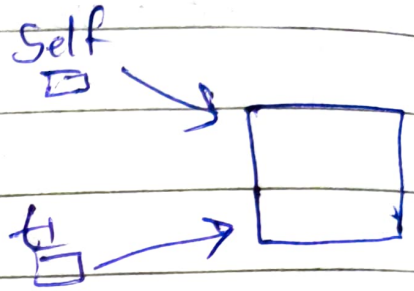
t_1 = Test()
t_2 = Test()
t_3 = Test()

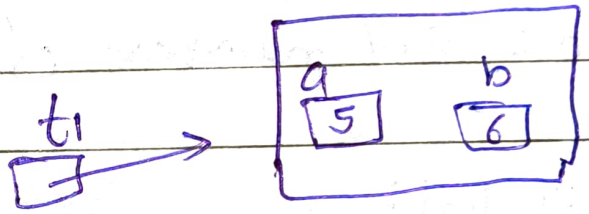


① class object variables ⟶ Static variable.
② Instance object variables.

__init__() Method.

```
class Test:
    def __init__(self):
        self.a = 5
        self.b = 6
t1 = Test()  # __init__(t1)
```





→ a and b are Instance object variables.

Methods.

1. Instance Methods.

```
class Test:
    def f1(self):
        pass
```

2. ~~Static~~ Class Method.

```
class Test:
    @classmethod
    def f2(cls):
        pass
```

3. Static Method.

```
Class Test:
    @staticmethod
    def f3():
        pass
```

Assign Create a class Employee with attributes empid, name, Salary and also define methods to access properties of Employee.

Singly Linked List

Agenda

1. What is a list?
2. What is a node?
3. Defining a node
4. Singly Linked list
5. Elementary operations.

What is a list?

~~Example~~: List is a linear Collection of data items also known as List Item.

Example 1:- list of marks. (ints)

30, 32, 20, 35, 41, 38

Example 2:- list of city names. (Str)

"Bhopal", "Itarsi", "Indore", "Delhi", "Jaipur", "Pune"

Example 3:- list of employes. (Employee)

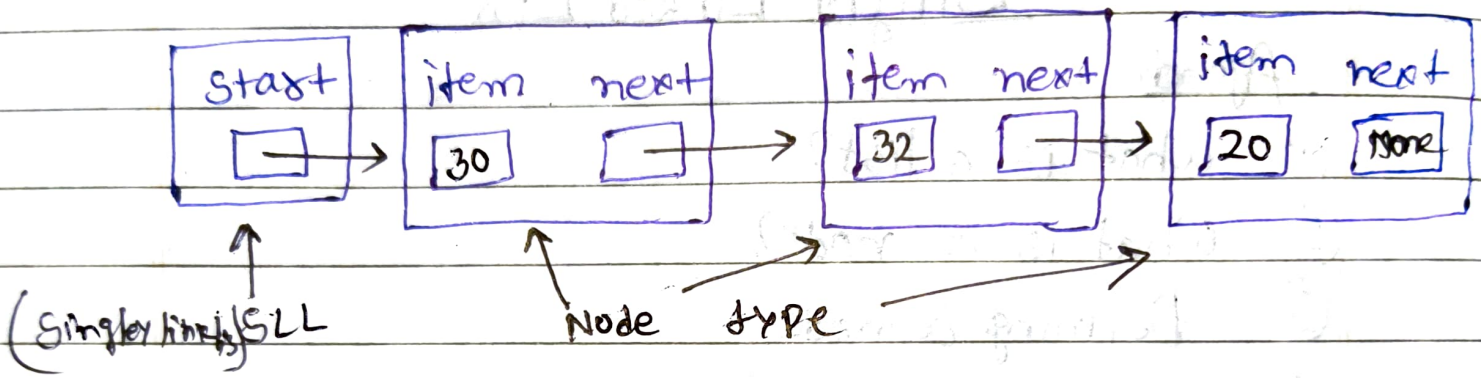
①	②	③	④
100	101	102	103
"Atal"	"Alok"	"Avush"	"Jeni"
25000	3L	10L	50K

What is node?

Example 1: list of marks (int)

30, 32, 20, 35, 41, 38

x → 30 y → 32 z → 20 X

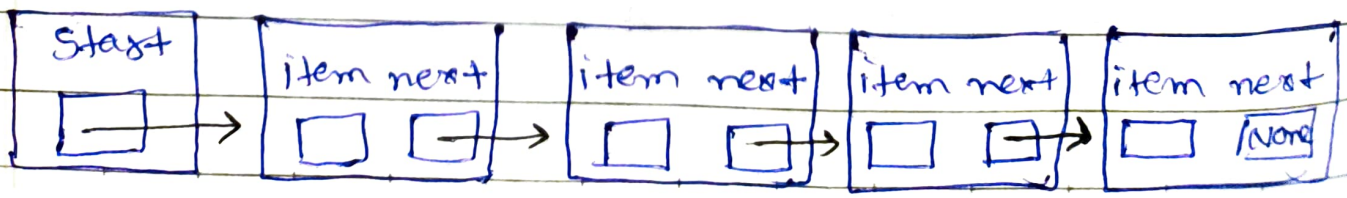


Defining a node

```
Class Node:
    def __init__(self, item = None, next = None):
        Self.item = item
        Self.next = next
```

Singly Linked list

- SLL is a linear data structure.
- It can grow and shrink.
- SLL → object



Operations on Singley Linked List.

Inserting
deletion
is_empty
traverse.

```
obj = SLL()
obj.insert_at_start(10)
obj.insert_at_last(20)
obj.insert_at_start(50)
obj.delete_first()
```

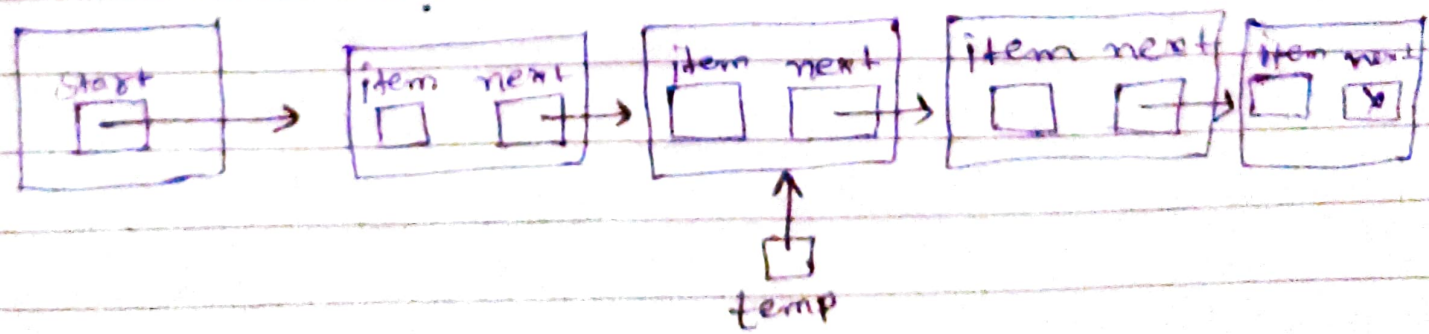
```
for x in obj:
    print(x)
```

Doubly Linked List

Agenda

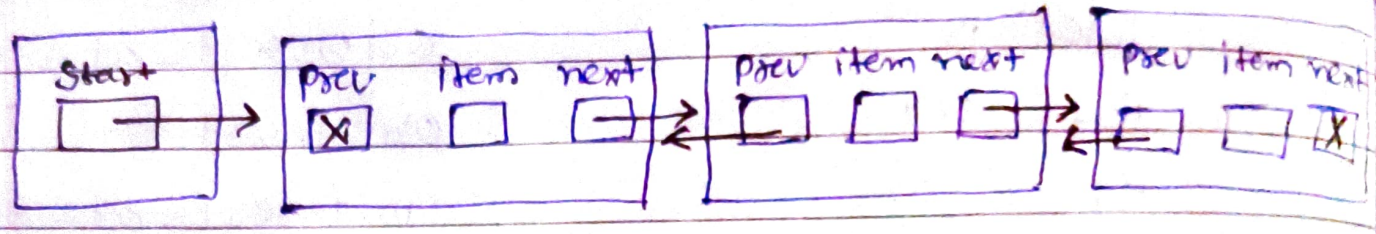
1. Limitations of Singly Linked List
2. Doubly Linked List
3. Elementary operations on DLL.

Limitations of SLL



Doubly Linked List

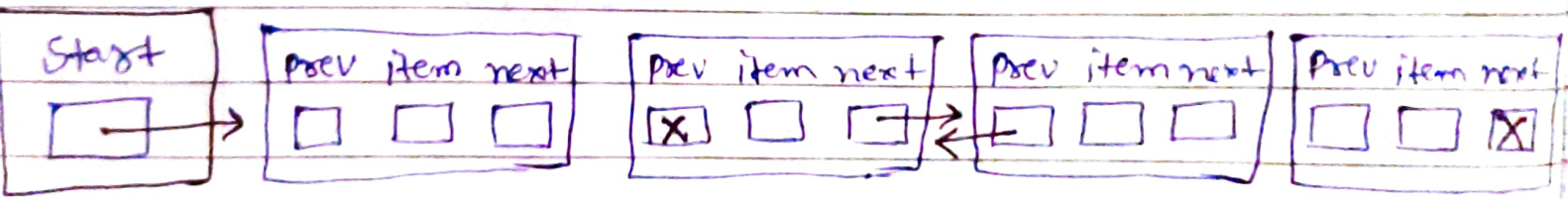
→ Doubly Linked List is a linear data Structure.



Elementary Operations

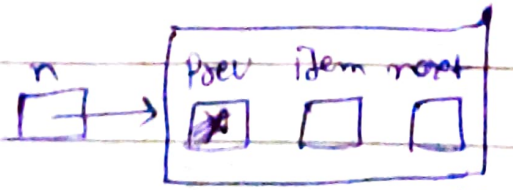
1. Insertion
2. Deletion
3. Traversing
4. Searching
5. Checking for empty list.

Insertion



Start. prev = n

Start = n



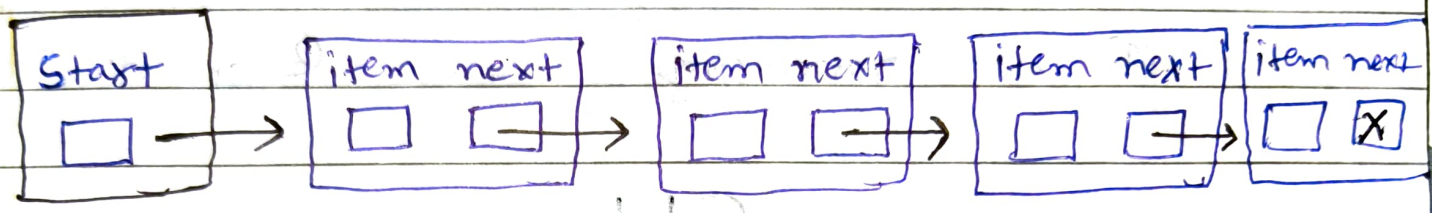
n. next = start.

Circular Linked List

Agenda

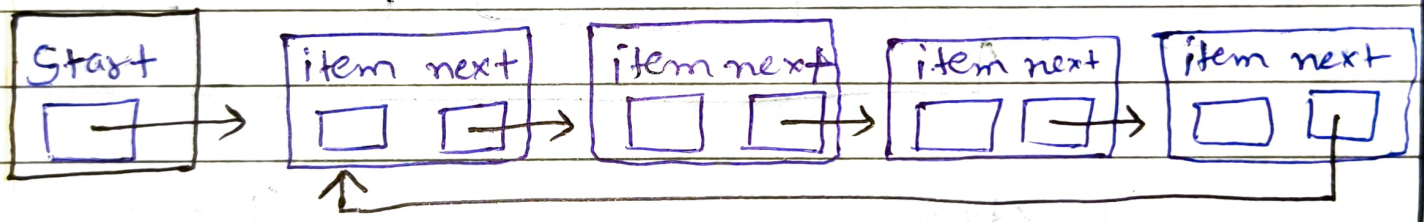
1. Limitations of Singley Linked List
2. Circular Linked List
3. Elementary Operations on CLL

Limitations of SLL



Circular Linked List

Circular Linked List is a linear data structure.



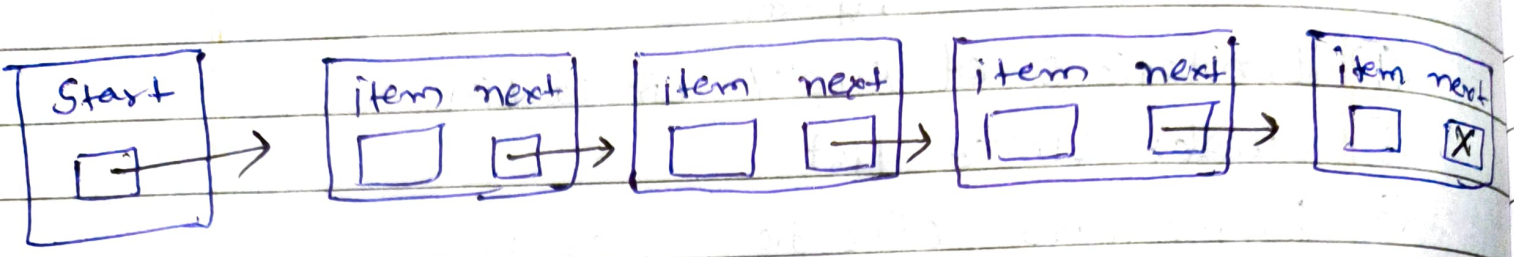
Check for last node:

```
if temp.next == Start:
```

Elementary Operations

1. ~~Deletion~~ Insertion
2. ~~Traversing~~ Deletion
3. ~~Searching~~ Traversing
4. Serching
5. Checking for empty list.

SLL



Deletion

① First NO traversing

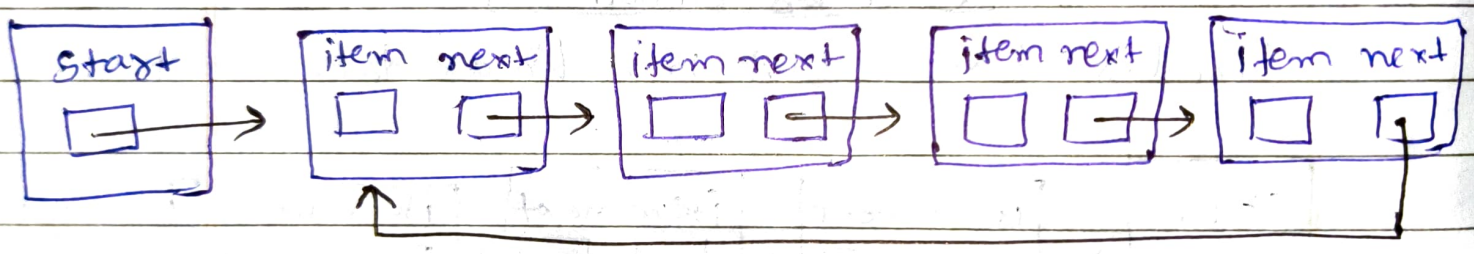
② Last Traversing

Insertion

① at start NO Traversing

② at last Traversing

CLL



Deletion

① First Traversing

② last Traversing

Insertion

① at start Traversing

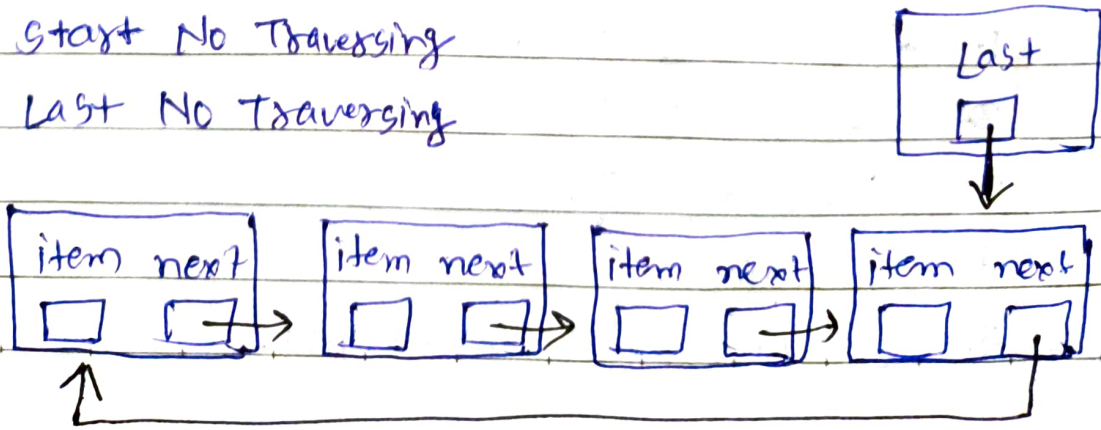
② at last Traversing

Repeat

Insertion

① start No Traversing

② Last No Traversing



Deletion

① Start No traversing

② Last Traversing.

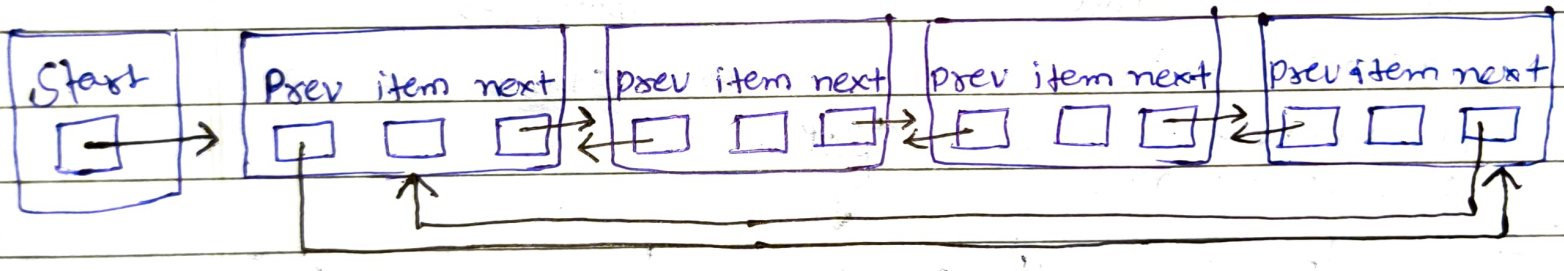
Circular Doubly Linked List

Agenda

① Circular Doubly Linked List

② Elementary operations on CDLL

Blend of Circular and Doubly Linked List.



Elementary operations on CDLL

① Insertion

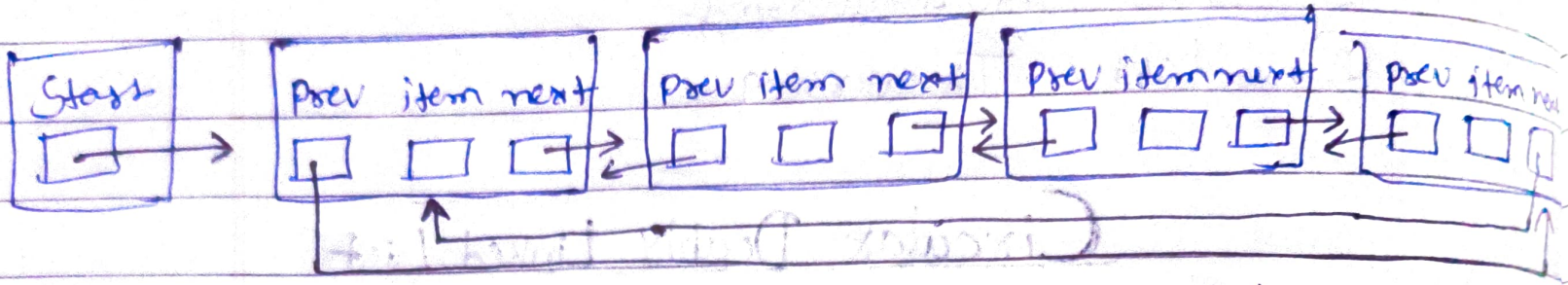
② Delation

③ traversing

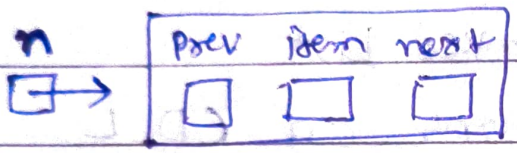
④ Search

⑤ Chak for empty

Insertion



Start add



n.item = data

n.next = start

n.prev = start.prev

start.prev.next = n

start.prev = n

start = n

Last add

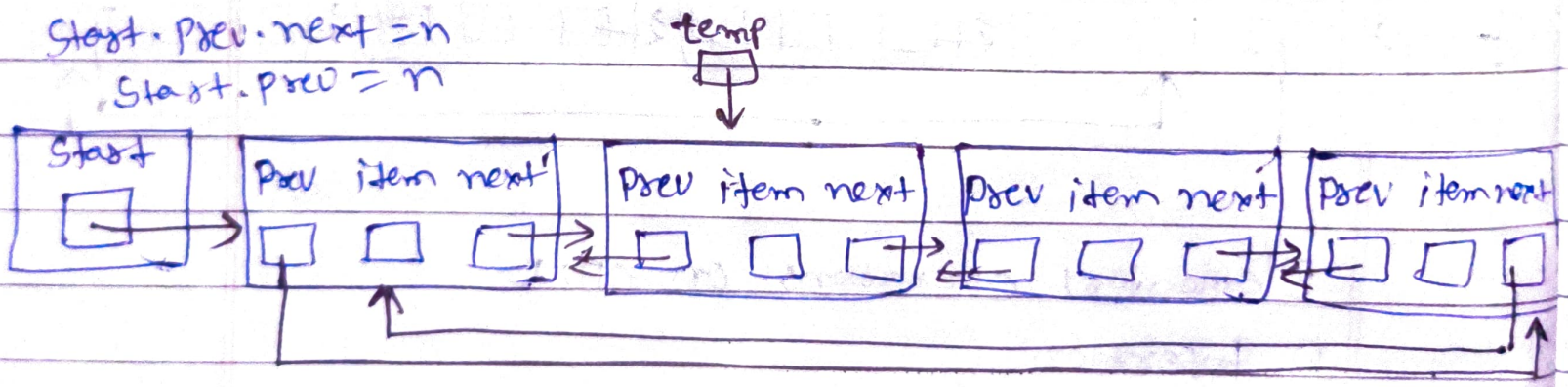
n.item = data

n.prev = start.prev

n.next = start

start.prev.next = n

start.prev = n



After add

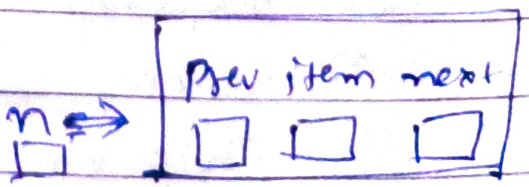
n.item = data

n.prev = temp

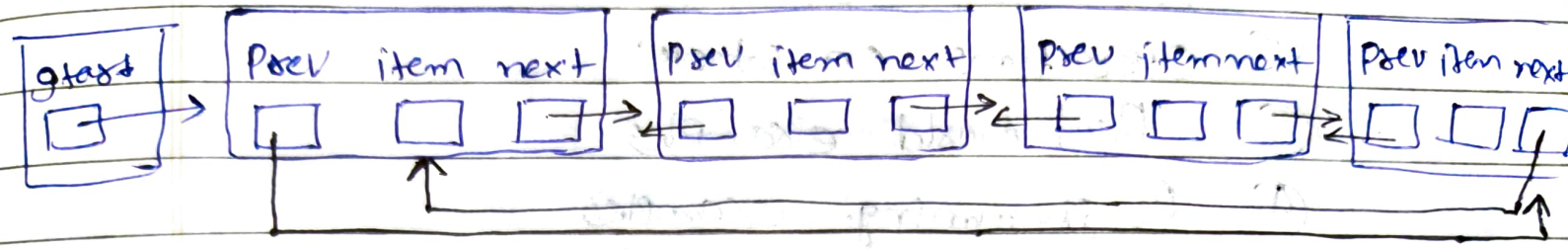
n.next = temp.next

temp.next.prev = n

temp.next = n



Deletion

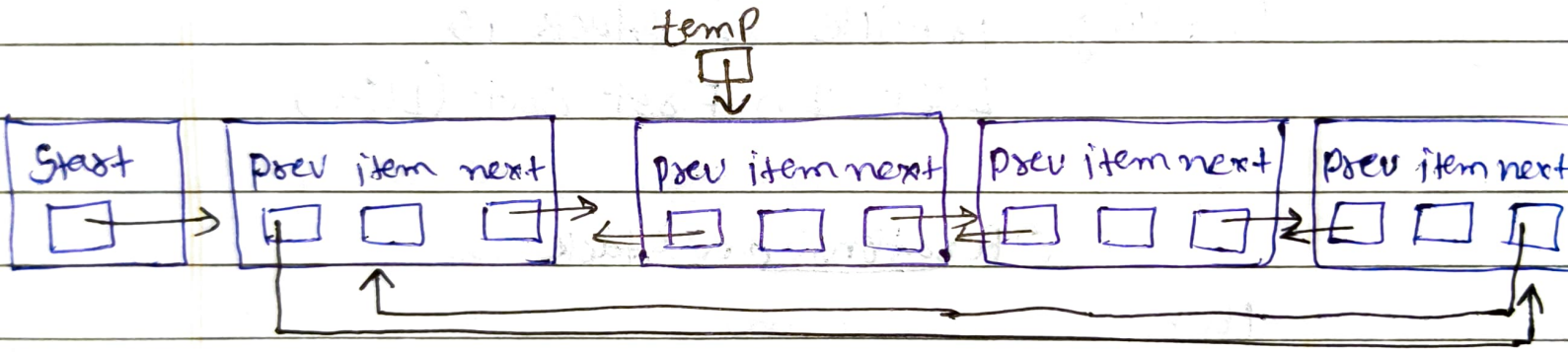


First Node delete

Start.Prev.next = Start.next

Start.next.Prev = Start.Prev

Start = Start.next



Last Node delete

Start.Prev.Prev.next = Start

Start.Prev = Start.Prev.Prev

After delete

temp.Prev.next = temp.next

temp.next.Prev = temp.Prev

Stack

Agenda

1. What is a Stack?
2. Operations on stack.
3. Real world examples
4. Programming examples
5. Implementation of stack.

What is Stack?

→ Stack is a linear data Structure.
Working principle of Stack is
Last In First Out (LIFO)

Operations on Stack

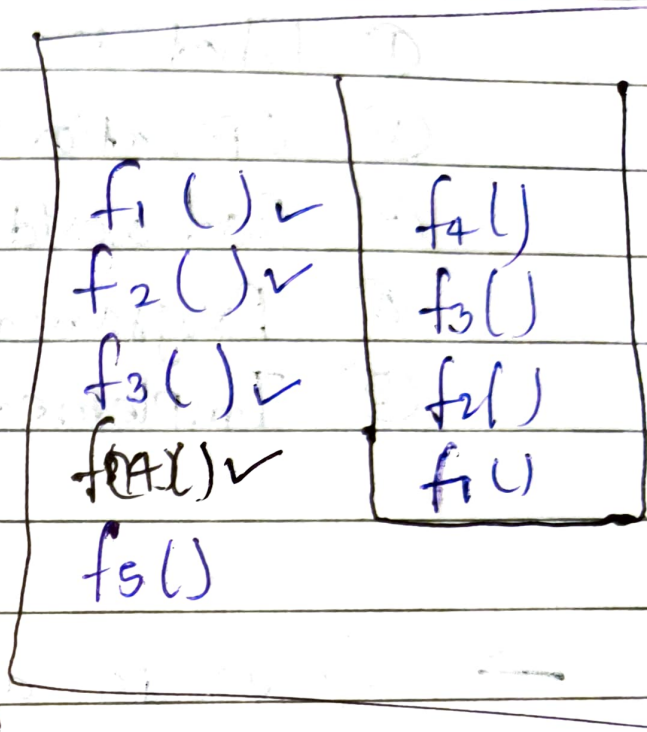
Push
POP
is-empty
Peek
Size

Real world Examples of Stack.

1. Travelling Bag
2. Compiled plates in buffet
3. Stack of books
4. Bread packet
5. Call History

Programming Examples.

1. Function call stack
2. Evalating expressions
3. Parenthesis matching
4. Iterative Solution of binary tree traversal
5. Depth first Search
6. Undo - redo operations



Implementation of Stack

1. Using list
2. by extending list class
3. Using Singly linked list class
4. by Extending Singly linked list class
5. Using Linked list Concept.

Quene

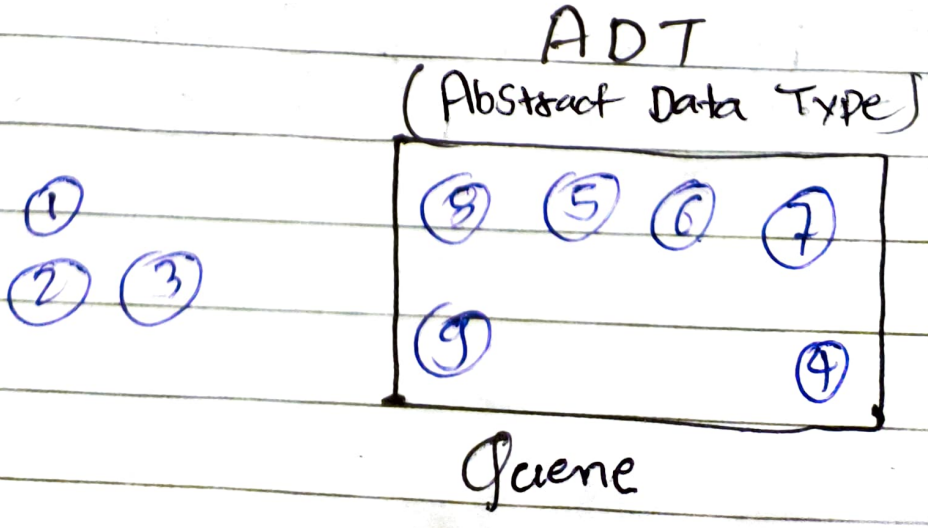
Agenda

1. What is a quene?
2. Operations on quene
3. Real world examples
4. Programming examples
5. Implementation of Quene.

What is a Quene?

→ Queue is a linear data structure.
Working principle of Quene is
First In First out (FIFO)

rear (Insertion) = 8 9
Front (deletion) = 2 3 4

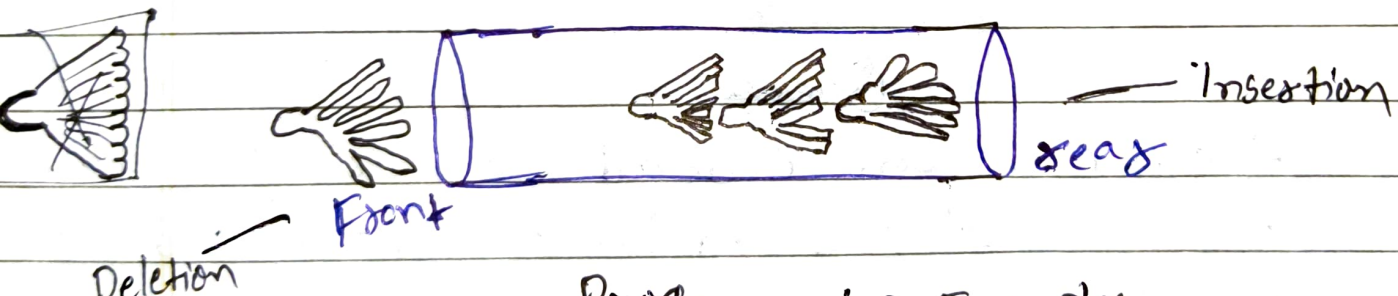


Operations on Queue

Enter
enqueue (Insertion)
dequeue (Deletion)
deleted
is_empty
get_front
get_rear
Size

Real world Examples of Queue

1. Queue for roller Coaster ride in an amusement Park.
2. Shuttecock (Badminton) in a cylindrical box



Programming Examples

1. Print tasks in a Printer
2. Admission module to implement first come first serve.
3. Breadth First Search

Implementation of Queue

1. Using list
2. by extending list class
3. Using Singly linked list class
4. by extending Singly linked list class
5. Using linked list concept.

Deque

Agenda

1. Variations of queue
2. What is a deque?
3. Operations on deque
4. Implementation of deque.

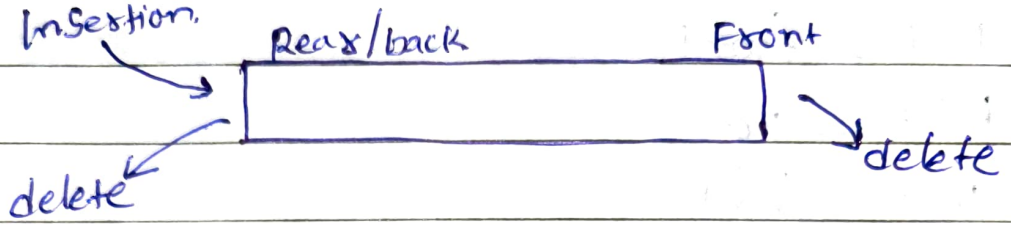
Queue

A queue is an ordered list in which insertions are done at one end (rear or back) and deletions are done at other end (front)

Working principle is First In First Out.

Variations of Queue.

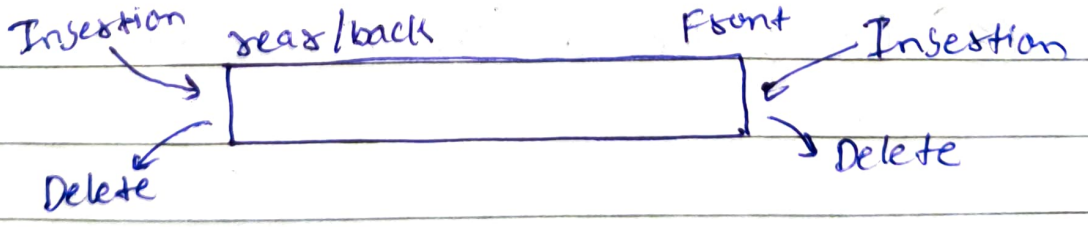
- Insertion restricted.



- Deletion restricted

Deque

- Deque is another variation of queue.
- Deque is Double-Ended-Queue.
- Deque is a linear data structure.
- Both ends can be used for insertion and deletion



- Some deques may support random access, depending on implementation.

Operations on Deque.

insert_front

insert_rear

delete_front

delete_rear

get_front

get_rear

is_empty

Size

Implementation of Deque.

1. Using list ✓
2. by extending list class
3. Using doubly linked list class
4. by extending doubly linked list class
5. Using linked list Concept. ✓

Priority Queue

Agenda

1. What is a priority Queue?
2. Operations on priority Queue?
3. Implementation of priority Queue

Priority Queue

A priority queue is a collection of elements such that each element has been assigned a priority. The order of elements are deleted and processed comes from the following rules:-

→ An element of higher priority is processed before any element of lower priority.

→ Two element with the same priority are processed according to the order in which they were added in the priority queue.

Operations on Priority Queue.

Push()
Pop()
is_empty()
Size()

Implementation of Priority Queue

1. Using linked list concept
2. Using list
3. Using heap (we will see later)

Recursion

Function calling itself it called recursion.

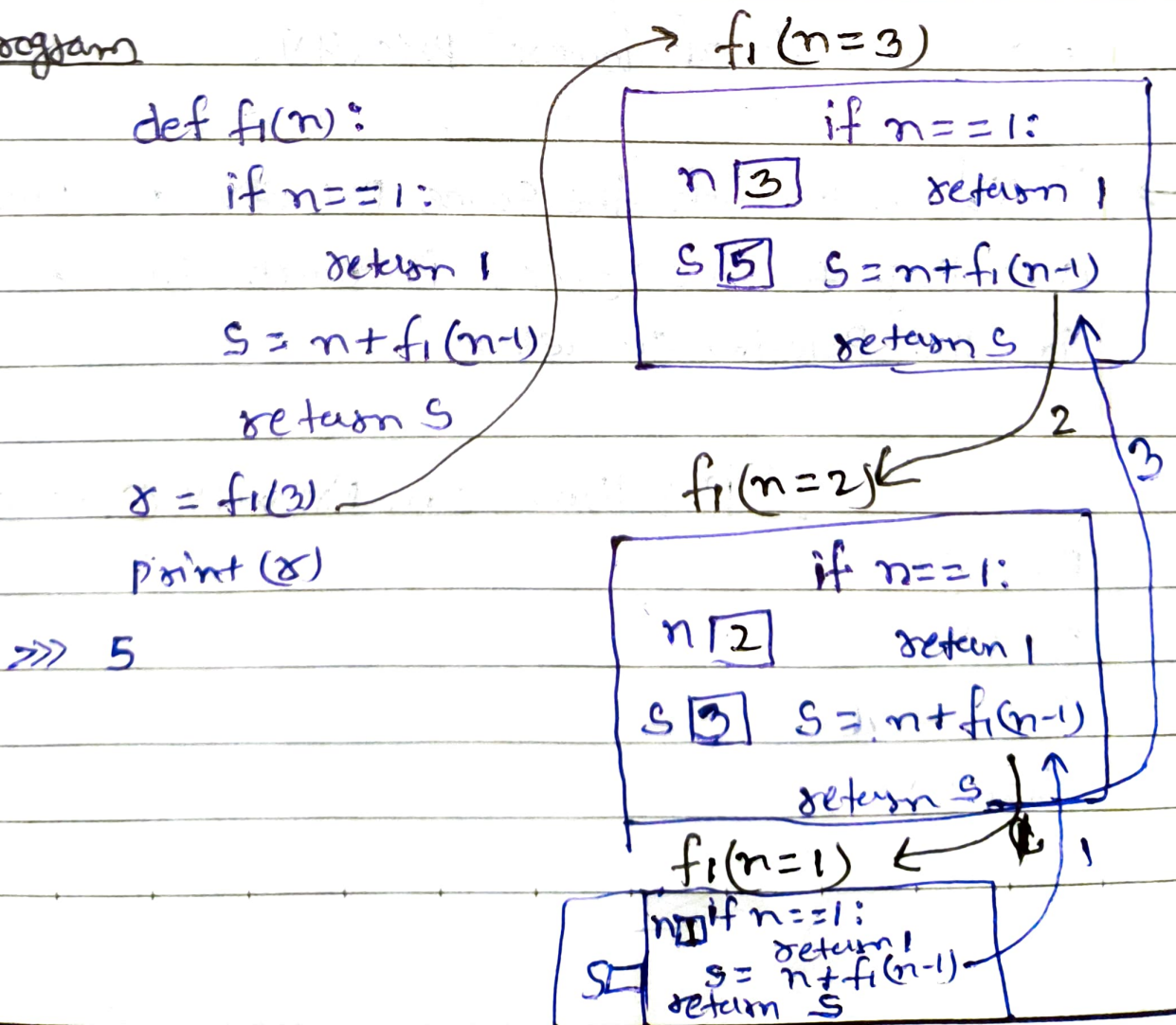
```
def f1():

f1()
```

Program

```
def f1(n):
    if n==1:
        return 1
    s = n+f1(n-1)
    return s
r = f1(3)
print(r)
```

>>> 5



$f_1(1)$ 1

$f_1(2)$ 1+2

$f_1(3)$ 1+2+3

$f_1(4)$ 1+2+3+4

$f_1(n)$ 1+2+3+4+...+n

Recursive case — Fcall

Base case — FNotcall.

$f_1(100)$

$100 + f_1(99)$

$99 + f_1(98)$

$98 + f_1(97)$

⋮

$4 + f_1(3)$

$3 + f_1(2)$

$2 + f_1(1)$

1

→ In recursion, Problem is Solved in terms of Problem itself.

→ Each time recursive function call to itself for little simpler version of the original Problem.

How to approach recursive function?

Write a recursive function to calculate sum of first n natural numbers.

→ suppose

① $f_1(n)$ $1+2+3+4+...+n$

R.C ② $f_1(n-1)+n$ $1+2+3+4+...+(n-1)+n$

B.C ③ $n==1$ 1

```
def f1(n):
    if n==1:
        return 1
    return n+f1(n-1)
```

Tree

Agenda

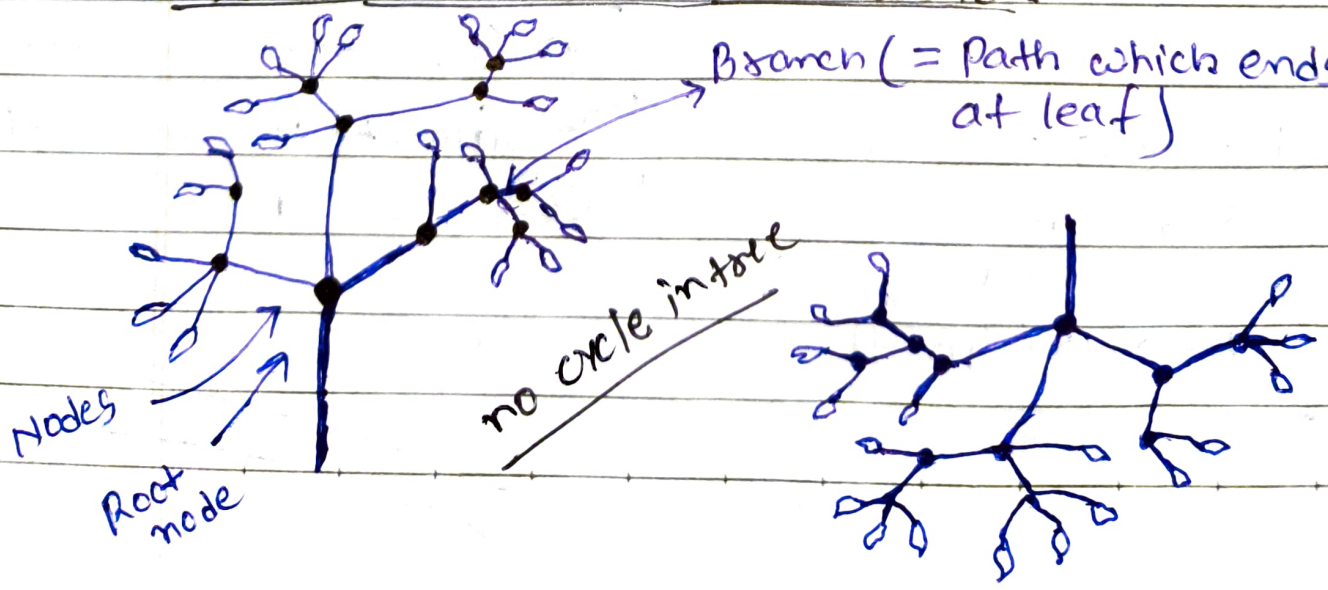
1. Tree
2. Real world Analogy
3. Degree, leaf, Parent-child
4. Siblings, Ancestors and descendents
5. Level number, height, Generation.

Tree

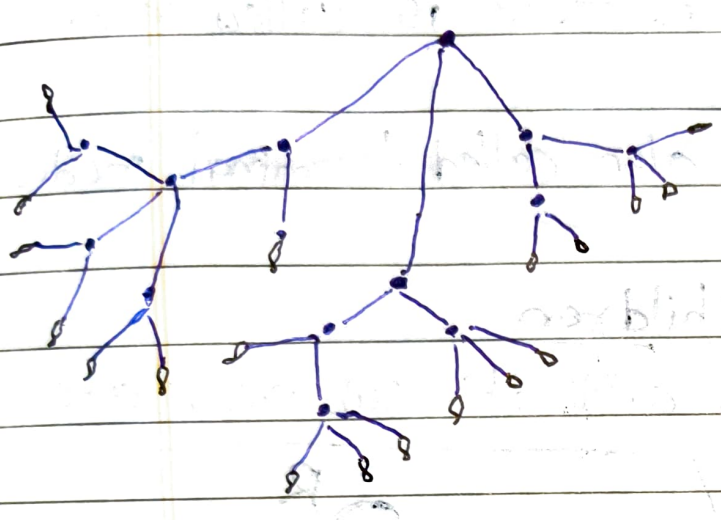
A tree is defined as a finite set of one or more data items (nodes) such that :-

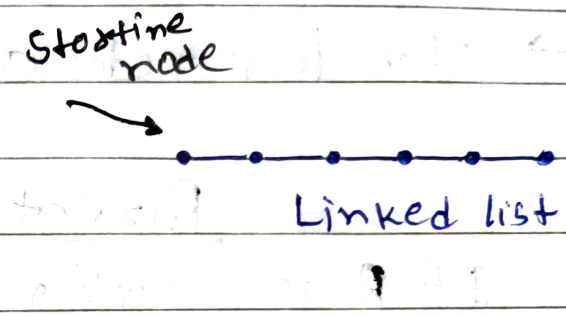
- There is a special node called the root node of the tree.
- The remaining nodes are partitioned into $n \geq 0$ disjoint subsets, each of which is itself a tree, and they are called Subtrees.

What is a Tree in Real world?



→ Tree is a hierarchical data structure.

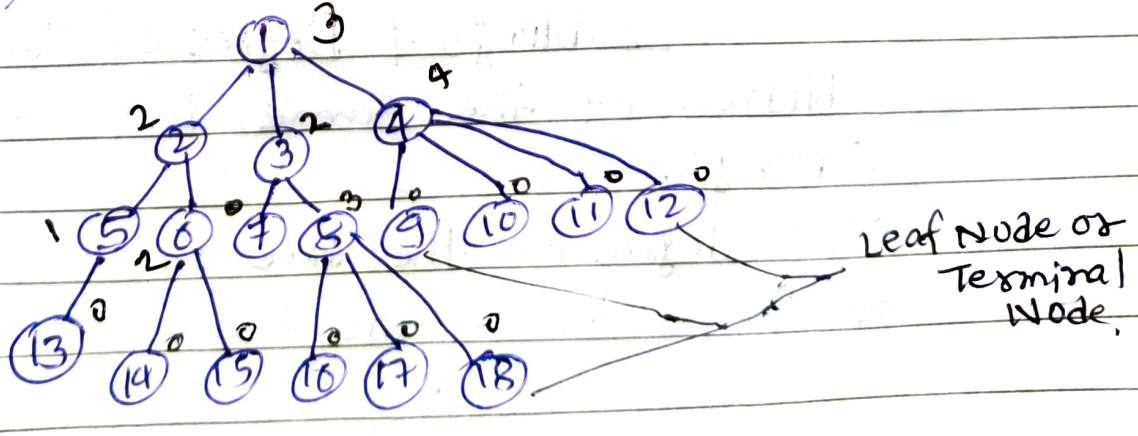




→ A tree is a non-linear data structure, which is used to represent hierarchical relationship existing among several data items.

Degree

The number of subtrees of a node is called its degree. A tree with degree 2 is called a Binary Tree.



Degree of the tree = 4.

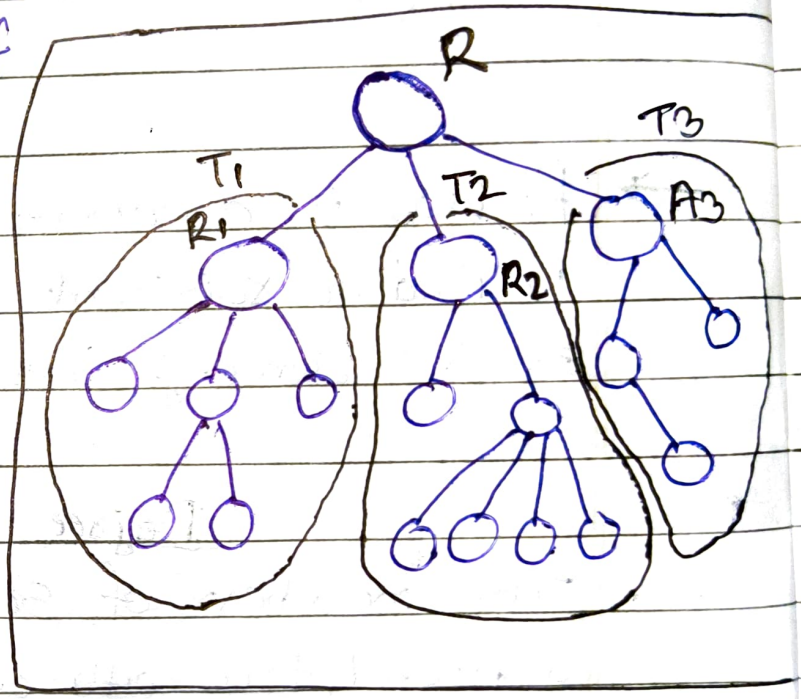
Leaf Node

→ A Node with degree zero is called leaf.

→ The leaf nodes are also called terminal nodes.

Parent - Children

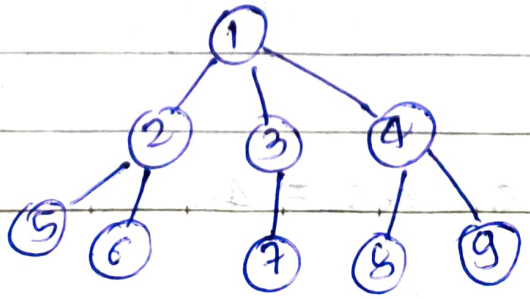
If R is a root node and its subtrees are T_1, T_2, T_3 and root of the subtrees are R_1, R_2, R_3, then R_1, R_2, R_3 are called children of R and R is called Parent of R_1, R_2, R_3.



Siblings & Degree of Tree

Children of the same Parent are called Siblings.

The degree of the tree is maximum degree of the nodes in the tree.



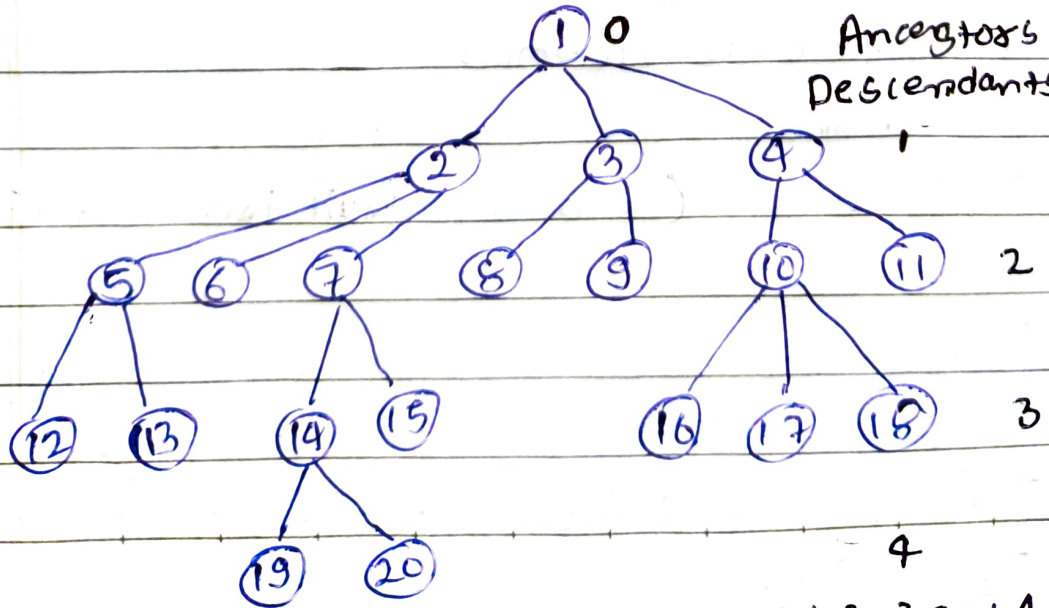
Ancestors and Descendants.

The ancestors of a node are all the nodes along the path from the root to that node.

The descendents of a node are all the nodes along the path from node to terminal node.

Level Numbers.

- Each node is assigned a level numbers.
- The root node of the tree is assigned a level number 0.
- Every other node assign a level number which is one more than the level number of its parent.



Generation

Nodes with the same level number are said to belong to the same generation.

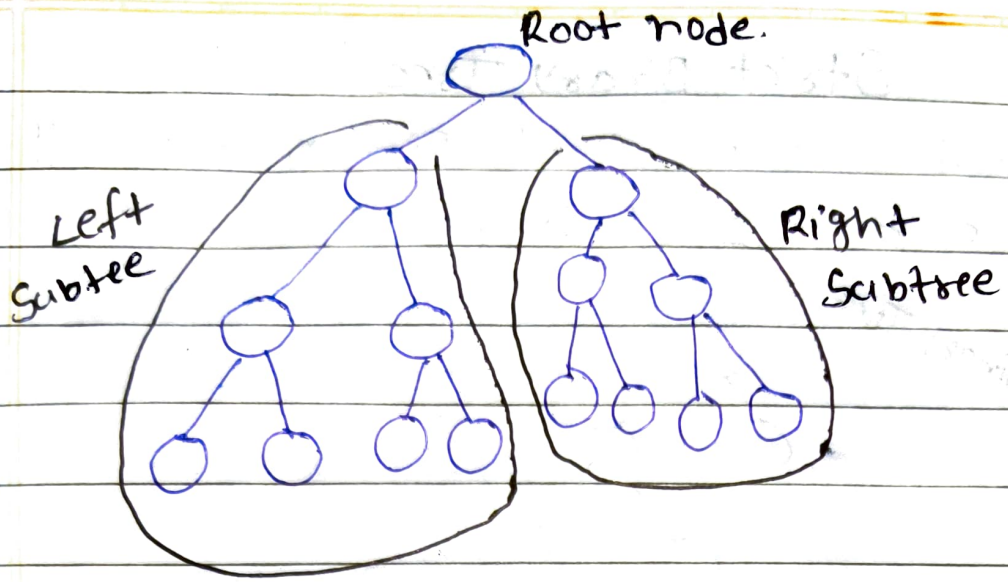
Height or Depth

- The height or depth of a tree is the maximum number of nodes in a branch
- A line drawn from a node to its children is called an edge.
- Sequence of consecutive edges is called path.
- Path ending in a leaf is called a branch.

Binary Tree

A binary tree is defined as a finite set of elements, called nodes, such that

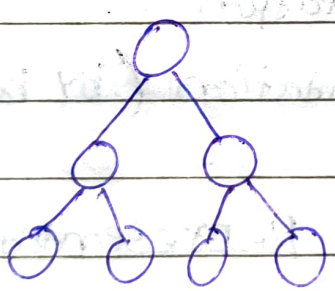
- T is empty (called the Null tree or empty tree) or
- T contains a distinguished node R, called the root of T, and the remaining nodes of T form an ordered pair of disjoint binary trees T_1 and T_2.



Any node in the binary tree has either 0, 1 or 2 child nodes.

Complete Binary Tree

All levels are Completetely filled.

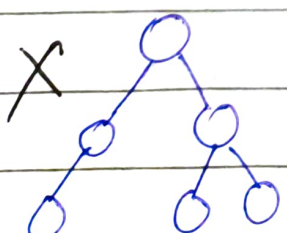


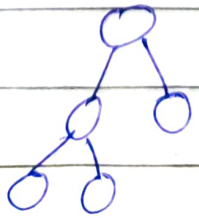
$L_0 - 1$
$L_1 - 2$
$L_2 - 4$
$L_3 - 8$
$L_4 - 16$
$L_5 - 32$
$L_6 - 64$
$L_7 - 128$
$L_n - 2^n$

Almost Complete Binary Tree.

All levels are completely filled, except possibility the last level and nodes in the level are all left aligned.

X

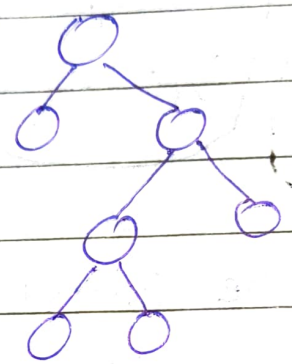




Strict Binary Tree

Each node of a Strict Binary Tree will have either 0 or 2 children

Full Binary Tree.

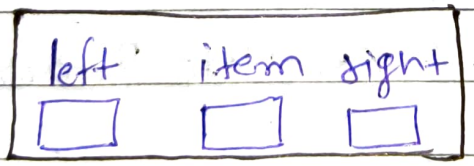


Representation of Binary Tree.

There are two possible representation of binary Tree.

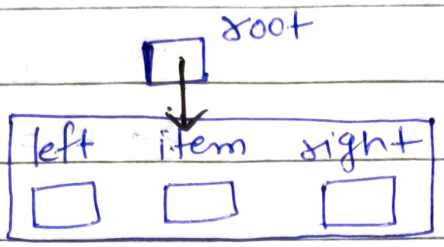
① Array Representation
② Linked Representation (by default)

Linked Representation

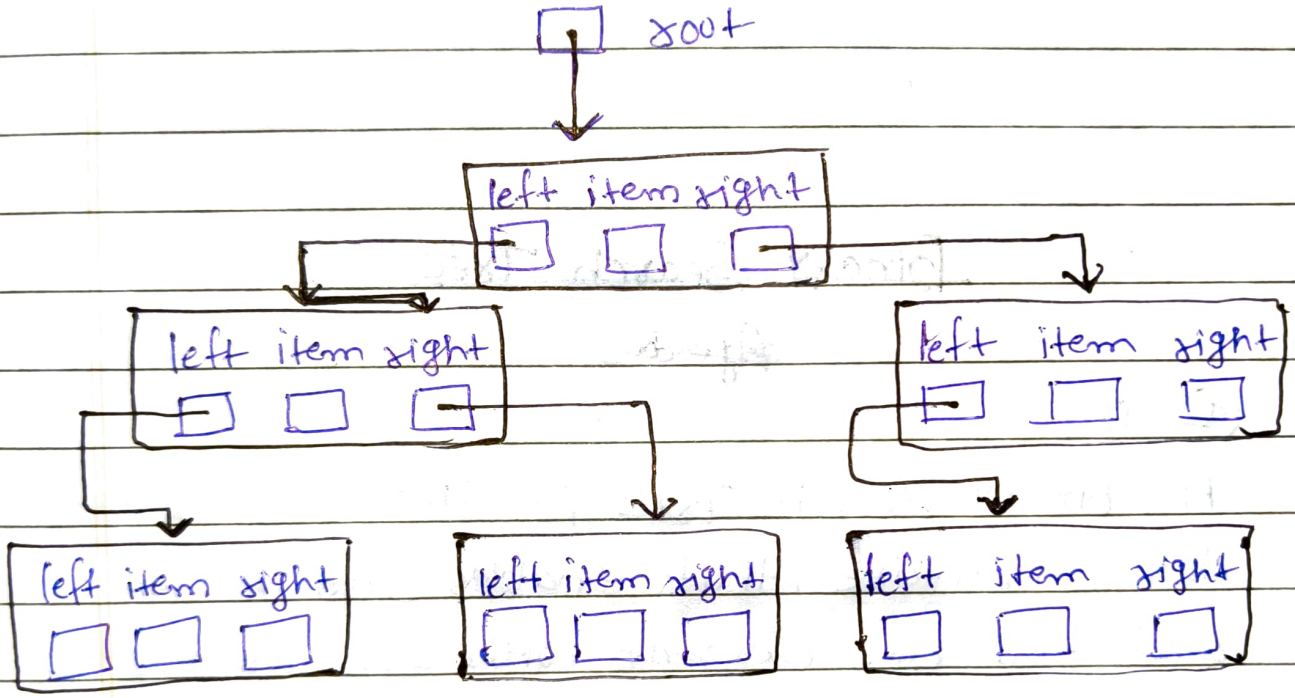



```
Class Node:
    def __init__(self, item=None, left=None, right=None):
        self.item = item
        self.left = left
        self.right = right
```

Root

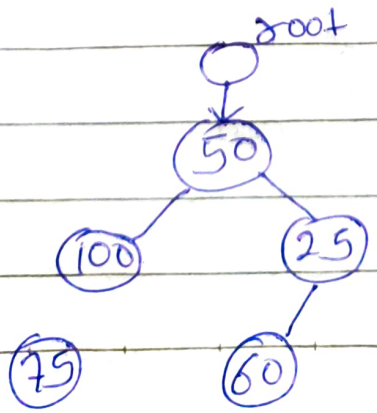


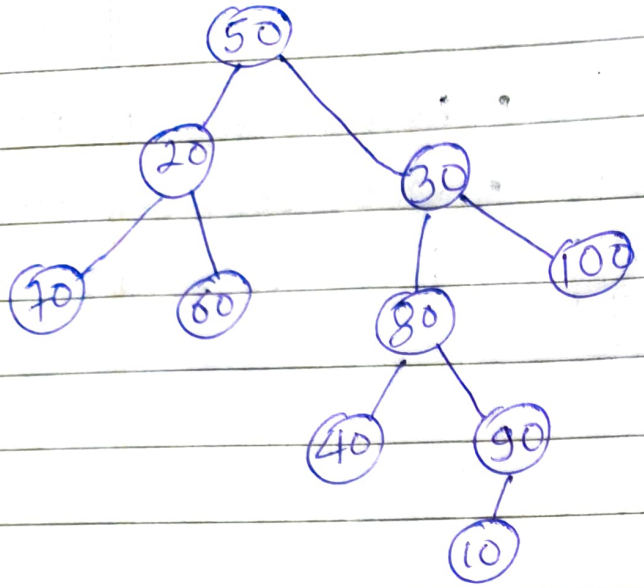
- root is a node Pointer
- When root certains NULL, tree is empty.



Discuss

- How to insert an item in a BT?
- How to traverse a BT?





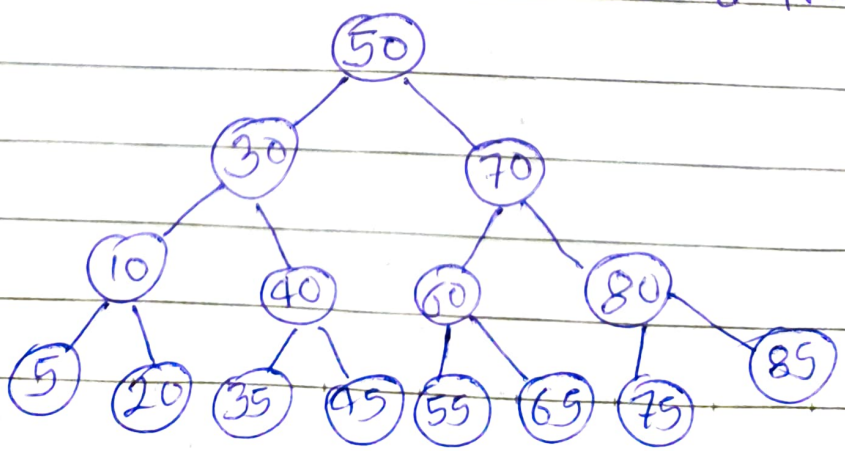
print (temp.item) 50
temp = temp.left
print (temp.item) 20
temp = temp.left

Binary Search Tree

→ A binary Search Tree is the most important data structure, that enables one to search for and find an element with an average running time.

$$f(n) = O(\log_2 n)$$

Duplicate values are not allowed in BST (by default)



Binary Search Tree is a binary tree with the value of node N is greater than every value in the left subtree of N and is less than every value in the right subtree of N.

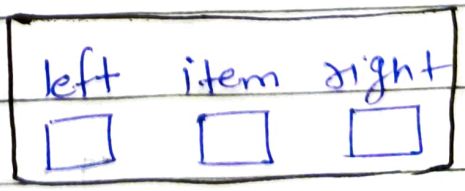
Unless, explicitly said, BST dosen't allow duplicate values.

Implementation

1. node
2. Insertion
3. Traversing
4. Search
5. Deletion

Node

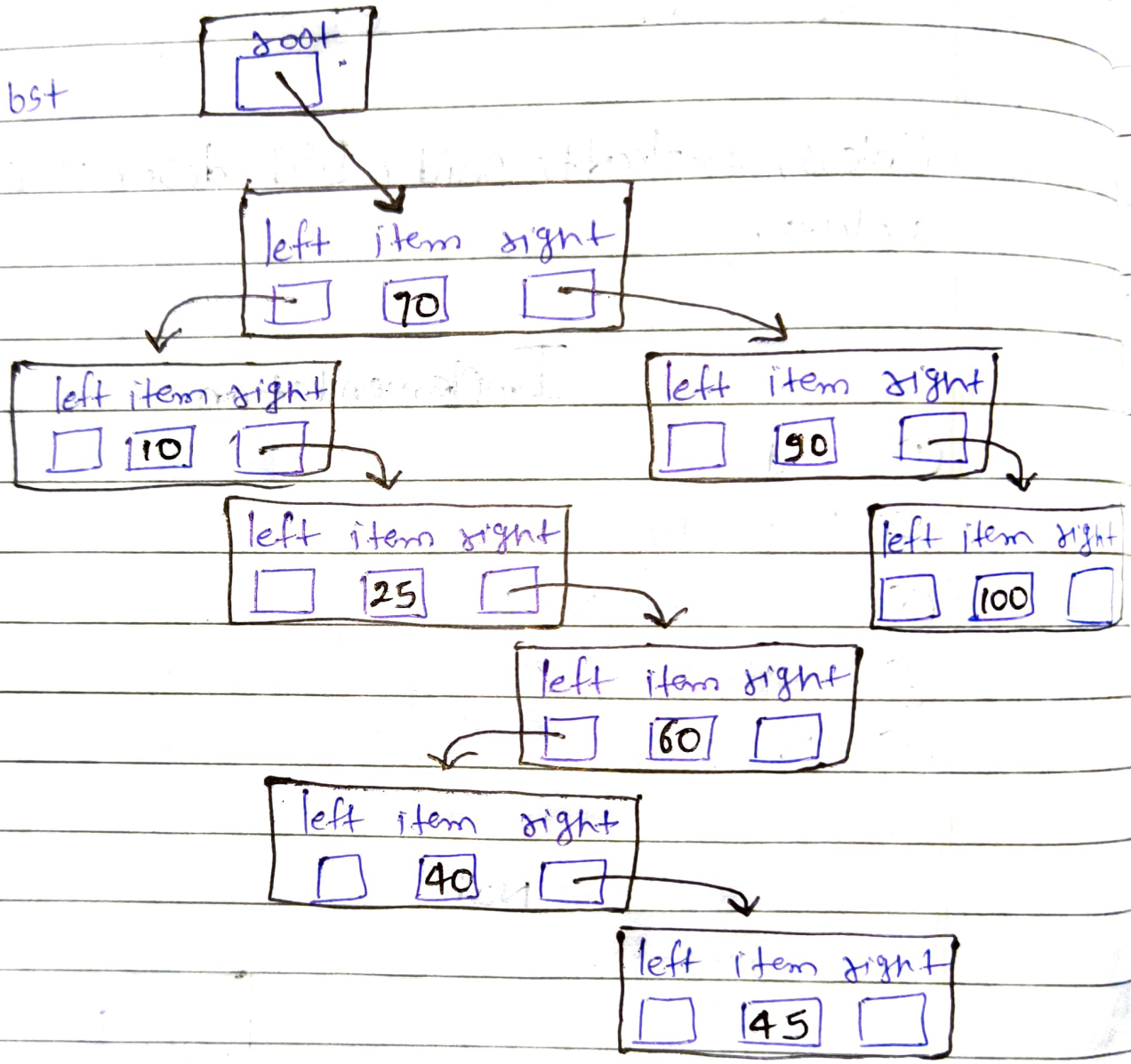
class Node:



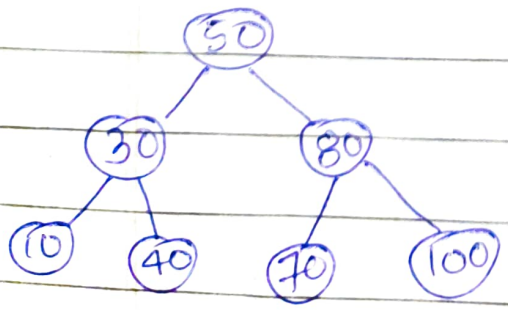
Insertion

bst = BST()

70, 10, 25, 90, 60, 40, 100, 45

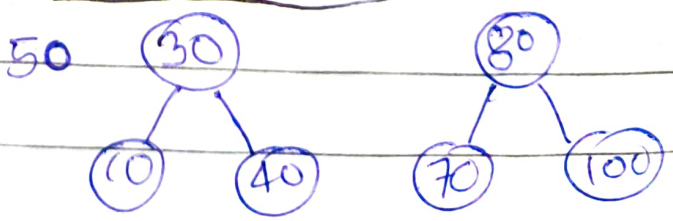


traversing



Preorder

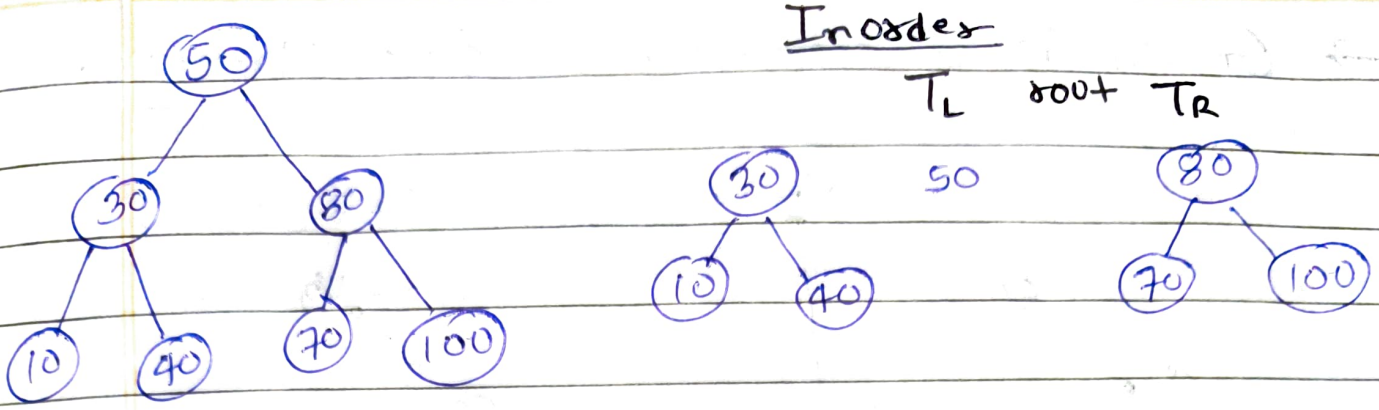
root TR



50 30 10 40 80 70 100

Inorder

T_L root T_R

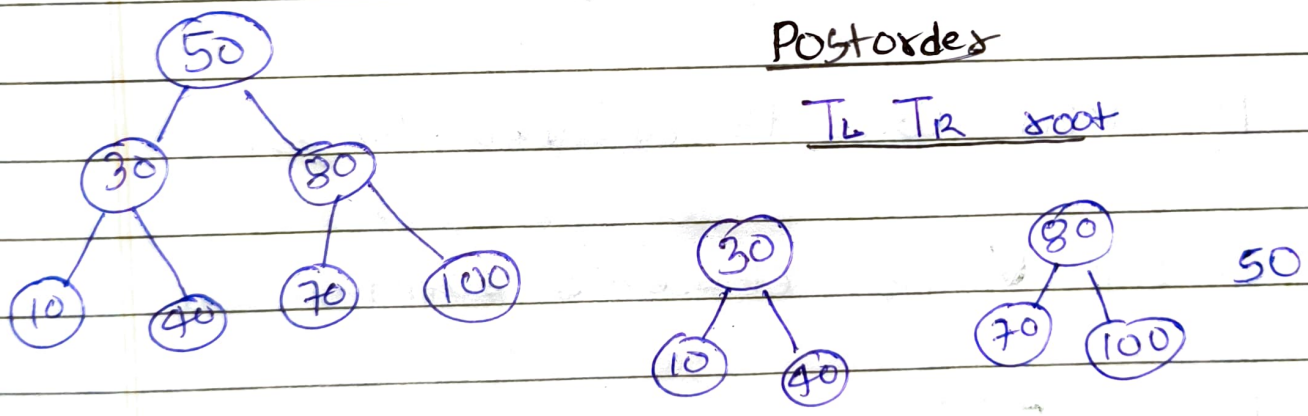


10 30 40 50 70 80 100

→ Inorder traversing of BST gives values in sorted order.

Postorder

T_L T_R root



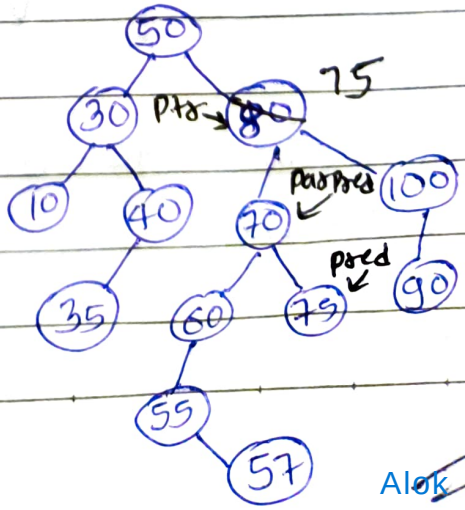
10 40 30 70 100 80 50

Deletion

① No child (10)

② Single child (40)

③ Two children (80)



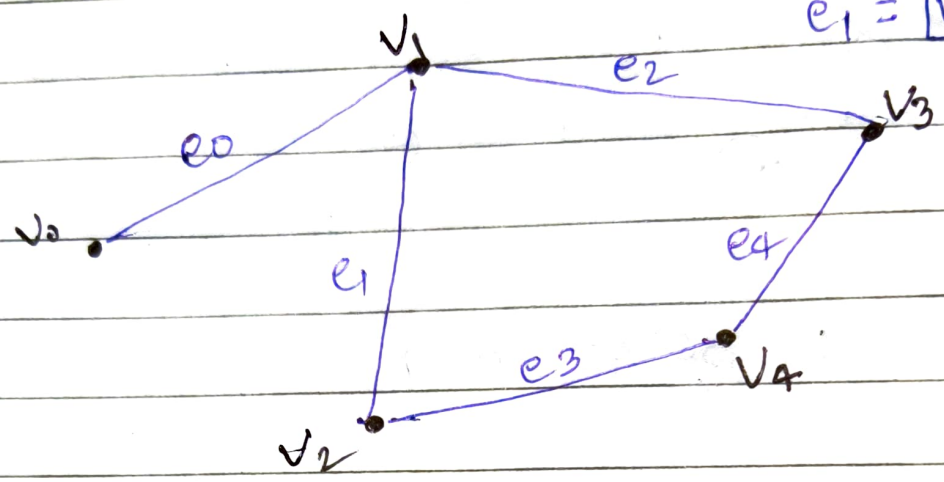
Alok

Graph

→ Graph is a non-linear data structure.

$e_0 = [v_0, v_1]$

$e_1 = [v_1, v_2]$



$V = \{v_0, v_1, v_2, v_3, v_4\}$

$E = \{e_0, e_1, e_2, e_3, e_4\}$

$(G = V, E)$

→ A Graph consists of two things.

→ A set **V** of elements called nodes.

→ A set **E** of edges such that each edge e in E is identified with a unique (unordered) pair $[u, v]$ of nodes in **V**, denoted by $e = [u, v]$

→ We indicate the parts of the graph by writing $G = (V, E)$

Adjacent nodes

If $e = [u, v]$, than u and v are called adjacent nodes or neighbors.

Degree of node.

The degree of node u, written $\deg(u)$, is the number of edges containing u.

→ If $\deg(u) = 0$, then u is called isolated node.

Path

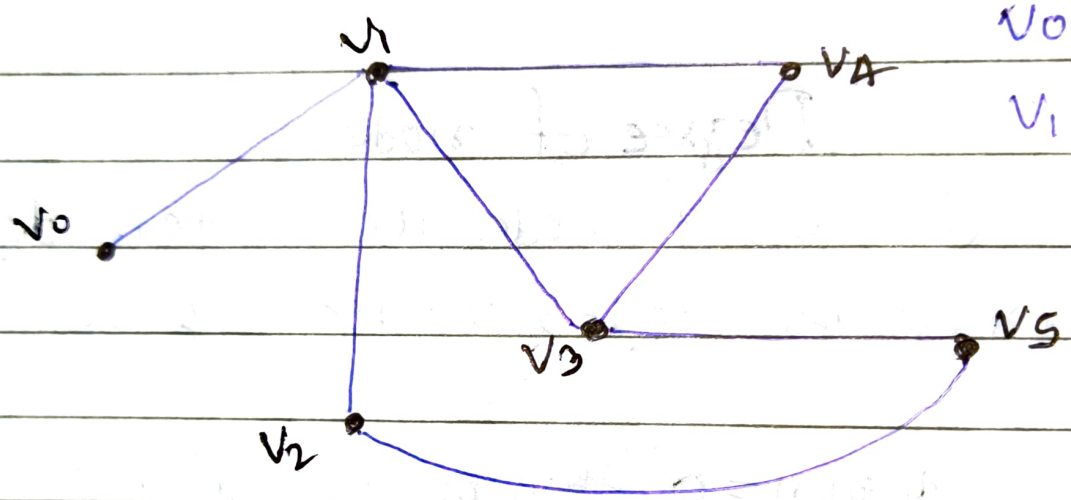
A Path of length n from a node u to a node v is defined as a sequence of $n+1$ nodes.

$$P = (v_0, v_1, v_2, \ldots v_n)$$

The Path is said to be closed if $v_0 = v_n$

Simple & Complex Path.

The Path is Said to be Simple if all the nodes are distinct, with the exception that V_0 may equal to V_n otherwise it is Complex Path.

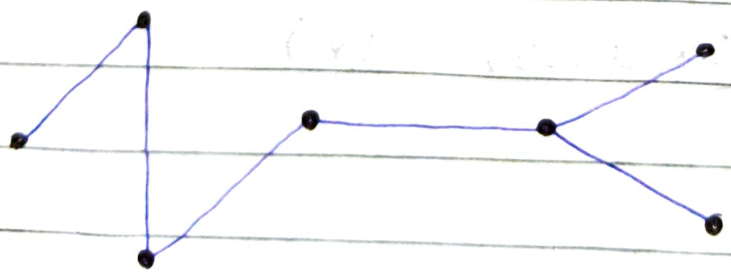


V_0, V_1, V_4, V_3

V_1, V_2, V_5, V_3

Connected Graph.

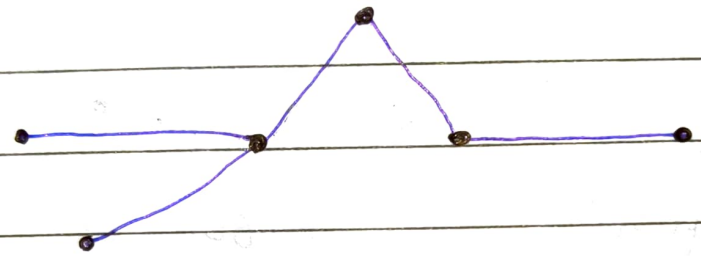
A graph is said to be connected if these is a path between any two of its nodes.



Tree Graph.

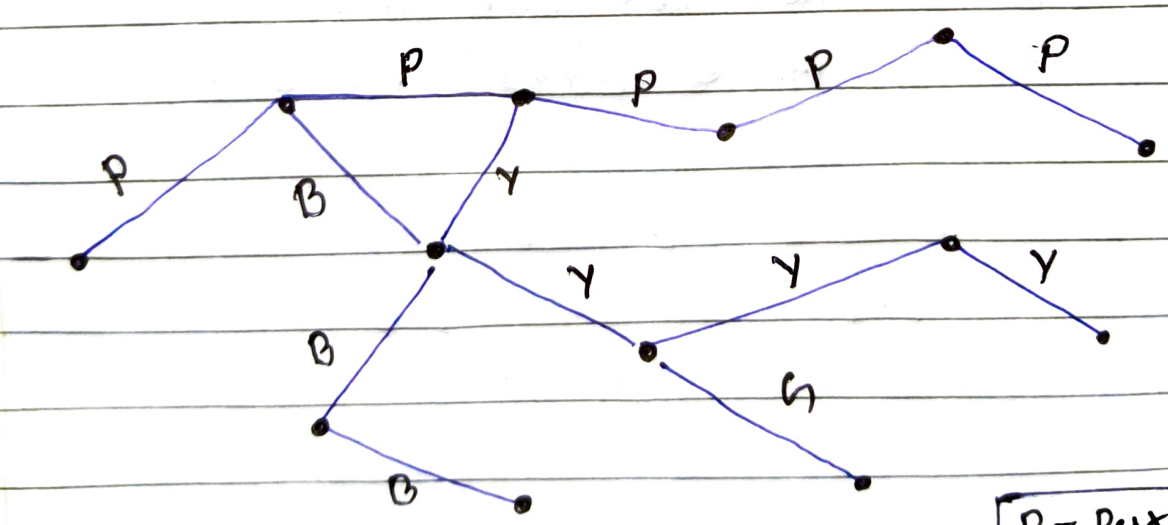
A Connected graph T without any cycles is called a tree graph or free tree, or Simpley a tree.

This means is particular, that there is a unique simple path P between any two nodes u and V.



Labelled Graph.

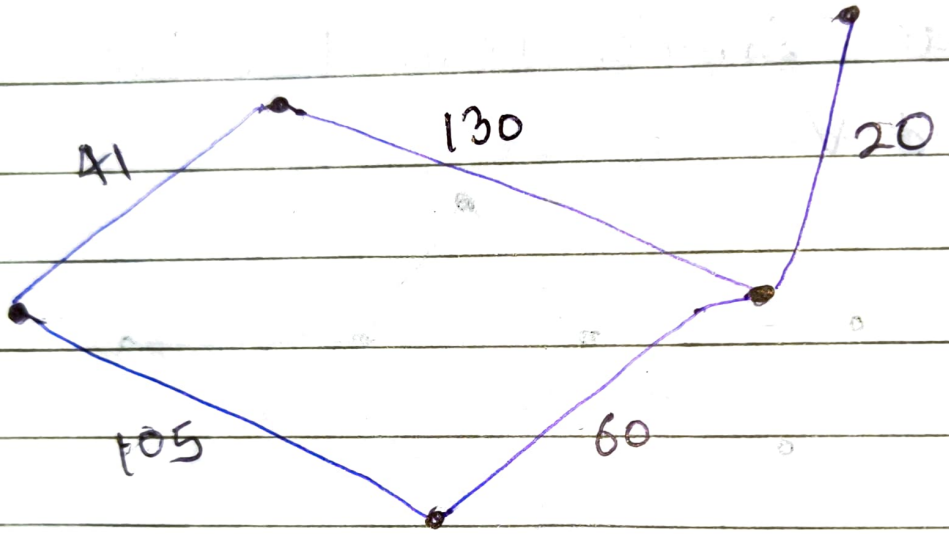
A graph is to be labelled if its edges are assigned data.



P - Purple line
B - Blue line
Y - Yellow line
G - Green line.

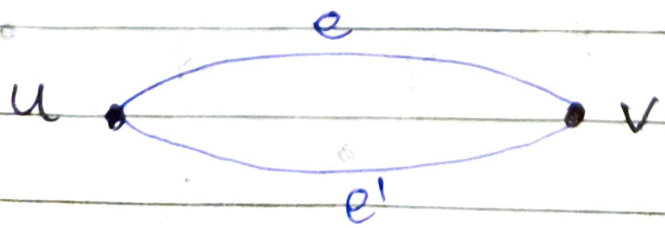
Weighted Graph

A graph G is said to be weighted if each edge e in G is assigned a non-negative numerical value W(e) called the weight or length of e.



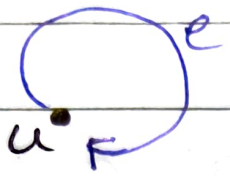
Multiple edges

Distinct edges e and e' are called multiple edges if they connect the same end points, that is, if e = [u,v] and e' = [u,v]



Loop

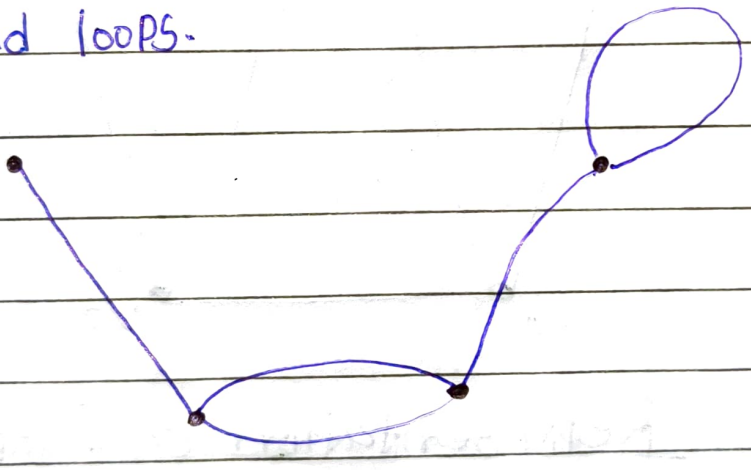
An edge is called loop if it has identical end points.



$e = [u, u]$

Multi Graph

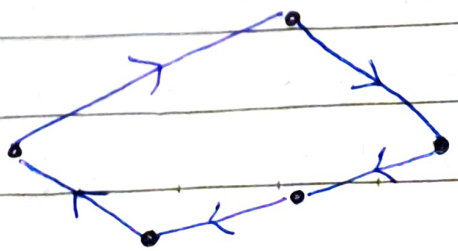
Multi Graph is a graph consisting of multiple edges and loops.



Directed Graph

A directed graph G also called digraph in same as multigraph except that each edge e is assigned a direction.

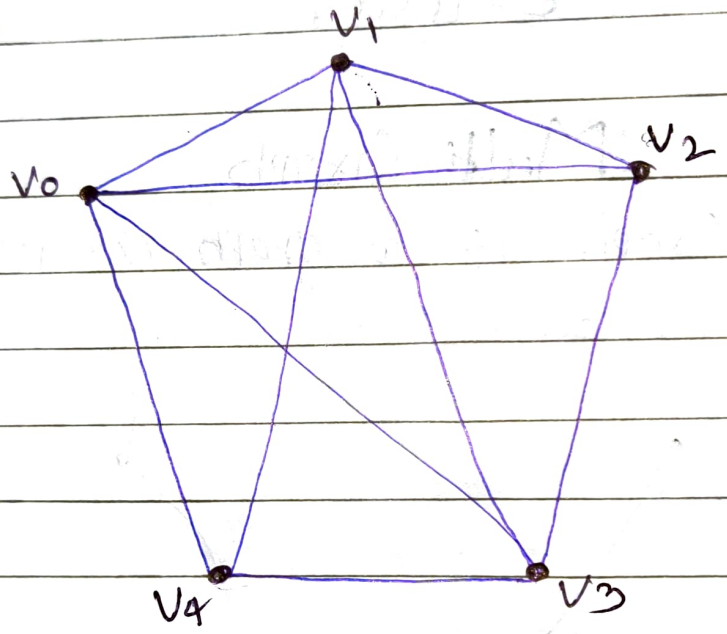
$e = (u, v)$ ← ordered pair



Complete Graph

A Simple graph in which there exists an edge between every pair of vertices is called a Complete graph.

It is also known as a Universal graph or Clique.



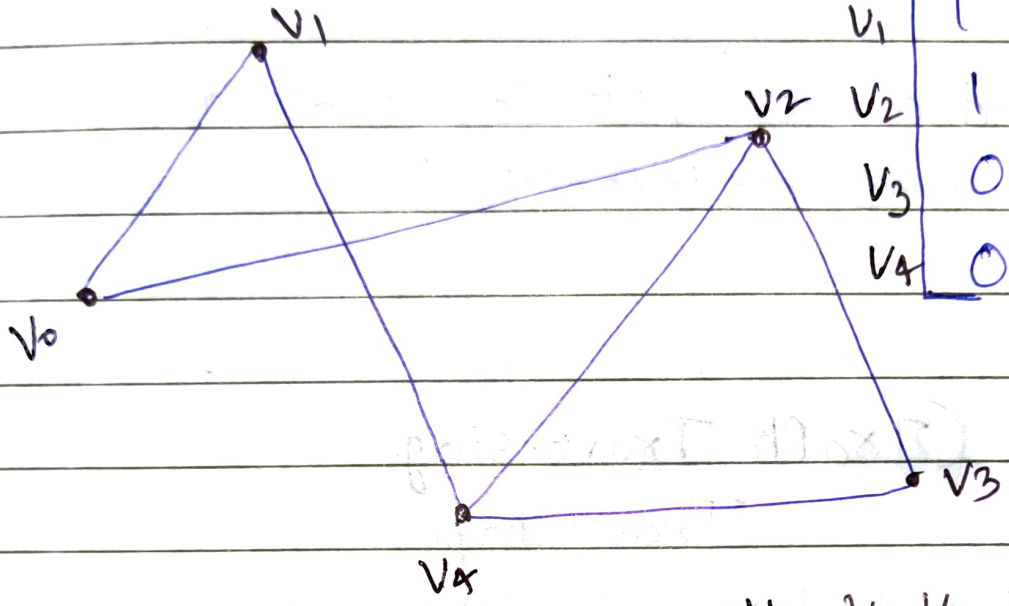
Representation of Graph

① Adjacency Matrix Representation

Suppose G is a Simple graph with m nodes, and suppose the nodes of G have been ordered and are called $V_1, V_2, V_3 \dots V_m$. Then the adjacency matrix $A = (a_{ij})$ of the graph G is the $m \times m$ matrix defined as follows:-

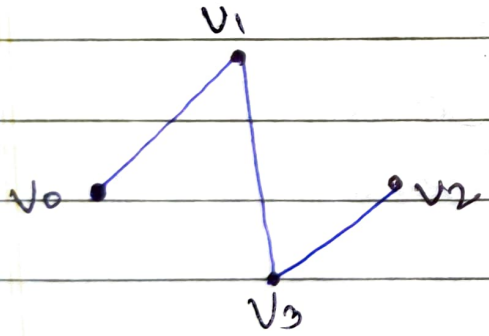
$$a_{ij} = \begin{cases} 1 & \text{if } v_i \text{ is adjacent to } v_j \\ 0 & \text{otherwise.} \end{cases}$$

eg.1.



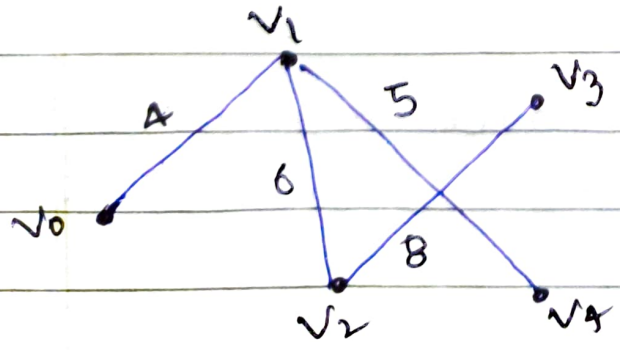
	v_0	v_1	v_2	v_3	v_4
v_0	0	1	1	0	0
v_1	1	0	0	0	1
v_2	1	0	0	1	1
v_3	0	0	1	0	1
v_4	0	1	1	1	0

eg.2.



	v_0	v_1	v_2	v_3
v_0	0	1	0	0
v_1	1	0	0	1
v_2	0	0	0	1
v_3	0	1	1	0

Weighted Graph



	v_0	v_1	v_2	v_3	v_4
v_0	0	4	0	0	0
v_1	4	0	6	0	5
v_2	0	6	0	8	0
v_3	0	0	8	0	0
v_4	0	5	0	0	0

(2) Adjacency List Representation

→ The adjacency list stores information about only those edges that exists.

→ The adjacency list contains a directory (dict) containing adjacency list for each vertex.

Graph Traversing

Traversing

There are two standard way of traversing a graph.

① BFS Breadth First Search

② DFS Depth First Search

BFS

Traversing graph has only issue that graph may have cycle. You may revisit a node.

To avoid processing a node more than once, we divide the vertices into two categories:

① visted

② Not visited

A boolean visited array is used to mark the visited vertics

BFS (Breadth First Search) uses a queue data structure for traversal.

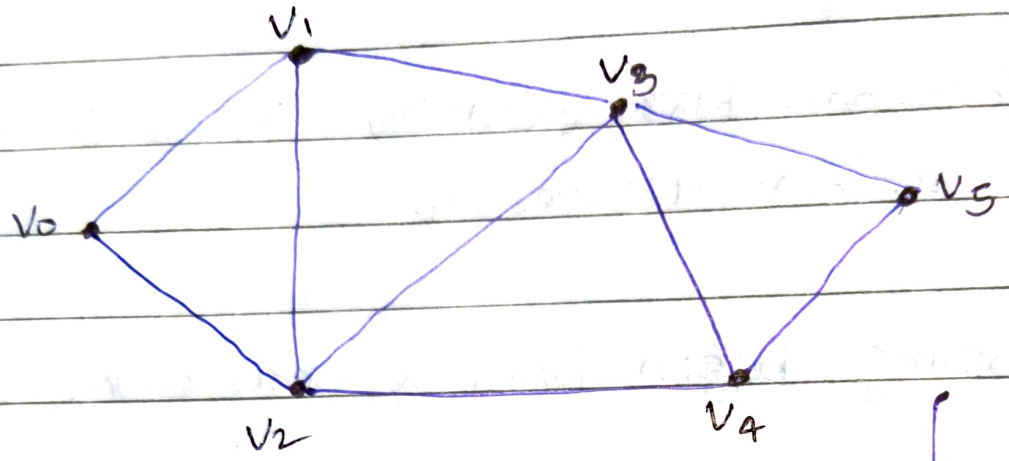
Traversing begin from a node called Source node

Logic for BFS

```
BFS (G, S)
    Let Q be the queue
    Q.insert(S)
    V[S] = True
    while (!Q.isEmpty())
    {
        n = Q.getFront()
        Q.del()
        for all the neighbors u of n
            if V[u] == False
                Q.insert(u)
                V[u] = True
    }
```

Example

Source = V_0



$L_0 = V_0$

$L_1 = V_1$

L_1 V_2

L_2 V_3

L_2 V_4

L_3 V_5

n = V_5

Q

	0	1	2	3	4	5
V	T	T	T	T	T	T

DFS

DFS is Depth First Search. The main defference between BFS and DFS is that the DFS uses Stack in Place of Queue.

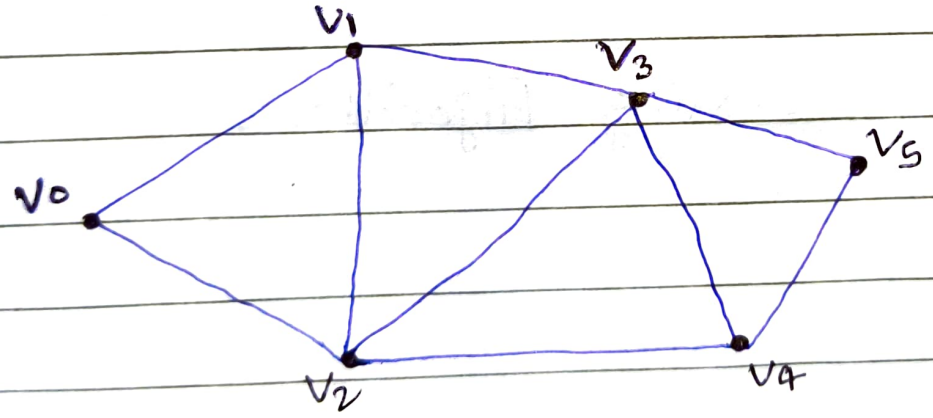
Logic for DFS

```
DFS(G, S)
    Let Stack be the stack
    Stack.Push(S)
    V[S] = True
```

```
While ( ! Stack. isEmpty())
{
    n = Stack. Peek()
    Stack. Pop()
    for all the neighbors u of n
        if V[u] == False
            Stack. Push(u)
            V[u] = True
}
```

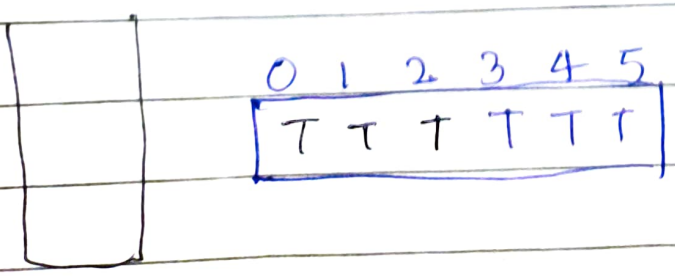
Example.

Source = V_0



Lo	Vo
L_1	V_2
L_2	V_4
L_3	V_5
L_1	V_3
L_1	V_1

n = V_1



Sorting

→ Arranging data elements in some logical order is known as sorting.

→ Sorting we are going to cover is also known as internal sorting.

→ When elements are numbers, sorting means arranging numbers in ascending order (by default)

→ When elements are strings, sorting means arranging strings in dictionary order (alphabetical order. (by default)

Various Sorting Algorithms

1. Bubble Sort
2. Modified Bubble Sort
3. Selection Sort
4. Insertion Sort
5. Quick Sort
6. Merge Sort
7. Heap Sort.

Bubble Sort

0	1	2	3	4	5
24	58	11	67	92	43

① (0,1) (1,2) (2,3) (3,4), (4,5)

24 11 58 67 43 92

② (0,1) (1,2) (2,3) (3,4)

11 24 58 43 67 92

③ (0,1) (1,2) (2,3)

11 24 43 58 67 92

④ (0,1) (1,2)

11 24 43 58 67 92

⑤ (0,1)

11 24 43 58 67 92

Modified Bubble Sort

n elements

Round - K $0 < K < n$

No swapping in k^{th} round means elements are sorted now. no more rounds to be executed.

Selection Sort

0	1	2	3	4	5	6	7	8
38	90	47	69	52	88	71	18	20

⇒

18	90	47	69	52	88	71	38	20
18	20	47	69	52	88	71	38	90
18	20	38	69	52	88	71	47	90
18	20	38	47	52	88	71	69	90
18	20	38	47	52	88	71	69	90
18	20	38	47	52	69	71	88	90

Insertion Sort

0	1	2	3	4	5	6	7	8	9
50	20	37	91	64	18	43	59	72	81

⇒

20	50	37	91	64	18	43	59	72	81
20	37	50	91	64	18	43	59	72	81
20	37	50	91	64	18	43	59	72	81
20	37	50	64	91	18	43	59	72	81
18	20	37	50	64	91	43	59	72	81
18	20	37	43	50	64	91	59	72	81
18	20	37	43	50	59	64	91	72	81
18	20	37	43	50	59	64	72	91	81
18	20	37	43	50	59	64	72	81	91

Quick Sort

0	1	2	3	4	5	6	7	8	9
58	62	91	43	29	37	88	72	16	30

Quick (0,9) → (0,4) ⇄
Quick (0,9) → (6,9) ⇄

left = 0/1/2...
right = 9/8...
loc = 0/9-
8
...

while l[loc] < l[right]
right -= 1
Swap
while l[left] < l[loc
left += 1

⇒

30	62	91	43	29	37	88	72	16	58
30	58	91	43	29	37	88	72	16	62
30	16	91	43	29	37	88	72	58	62
30	16	58	43	29	37	88	72	91	62
~~0, 1, 2~~ 30	16	37	43	29	58	88	72	91	62

Stop

Now 2 sublist
again apply loop.

Merge Sort

0	1	2	3	4	5	6	7	8	9	10	11	12
75	29	83	42	16	90	56	34	20	71	88	92	7

⇒ 75 29 83 42 16 90 56 34 20 71 88 92 7

Quick Sort

0	1	2	3	4	5	6	7	8	9
53	11	72	68	41	25	18	37	44	80

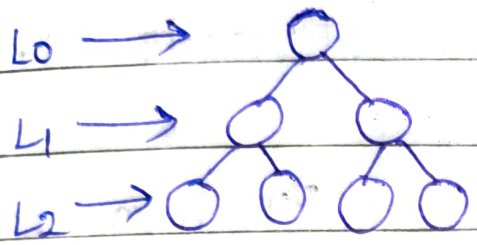
lesser = [11, 41, 25, 18, 37, 44]

greater = [72, 68, 80]

QS(lesser) + [53] + QS(greater)

Heap

Complete Binary Tree



All levels must be completed filled.

$L_0 \rightarrow 1$

$L_1 \rightarrow 2$

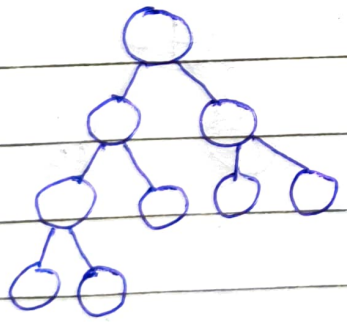
$L_2 \rightarrow 4$

$L_3 \rightarrow 8$

$L_4 \rightarrow 16$

$L_n \rightarrow 2^n$

Almost Complete Binary Tree



All the levels must be completely filled except possibly the last level and all the nodes in the last level must be left aligned.

Heap

→ Heap is a data Structure.

→ Heap is used in a Sorting algorithem known as heapsort.

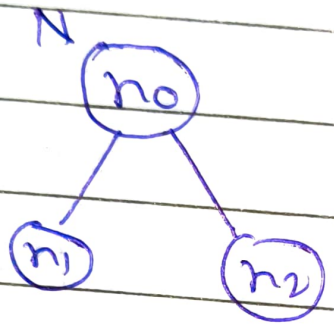
→ Heap are of two types

- Max Heap (default)
- Min Heap

Heap Properties

The value of node N is greater than or equal to value of each children of node N.

Heap must be an almost complete binary tree.

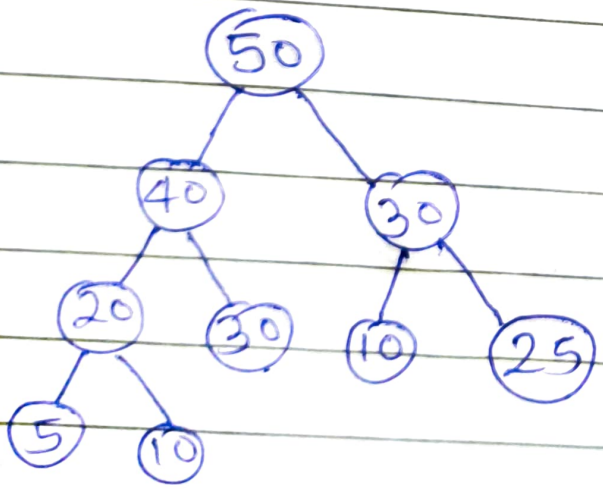


$n_0 \geq n_1$ and $n_0 \geq n_2$

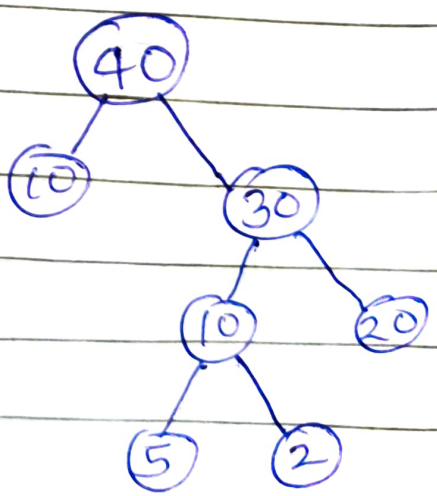
This heap is max heap

For min heap $n_0 \leq n_1$ and $n_0 \leq n_2$

Example



Max-heap

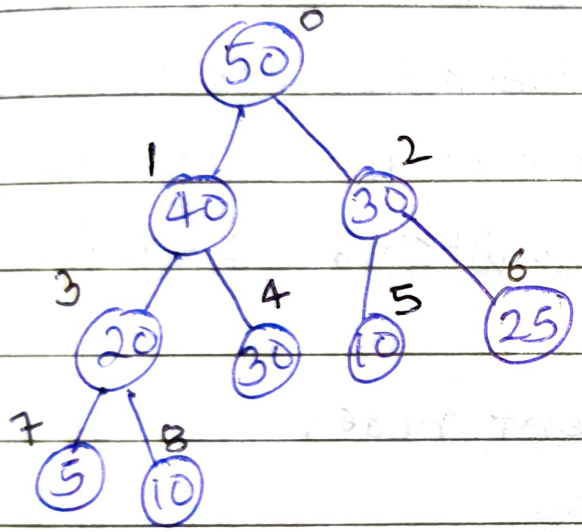


Not a heap

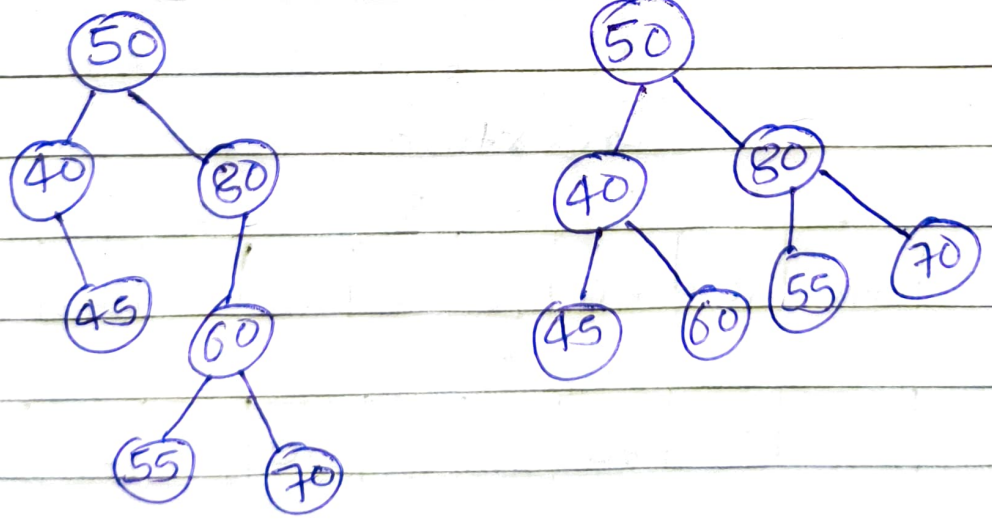
Representation of Heap.

Unless other wise stated, heap is maintained in memory by a linear array (list)

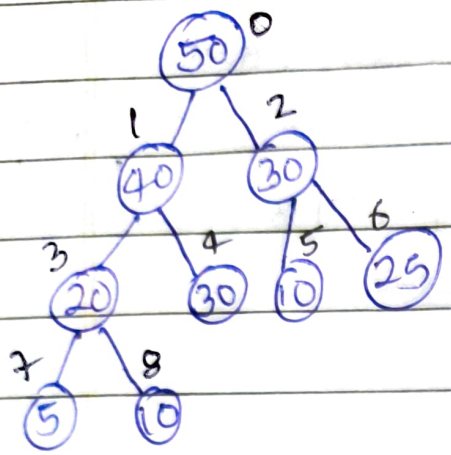
	0	1	2	3	4	5	6	7	8
H	50	40	30	20	30	10	25	5	10



	0	1	2	3	4	5	6
	50	40	80	45	60	55	70



How to find Parent or child node?



	0	1	2	3	4	5	6	7	8
H	50	40	30	20	30	10	25	5	10

Q. How to Find index of child nodes?

⇒ index is index of node N

index of left child = 2 * index + 1

index of right child = 2 * index + 2

Q. How to find index of parent node?

index of node N = index

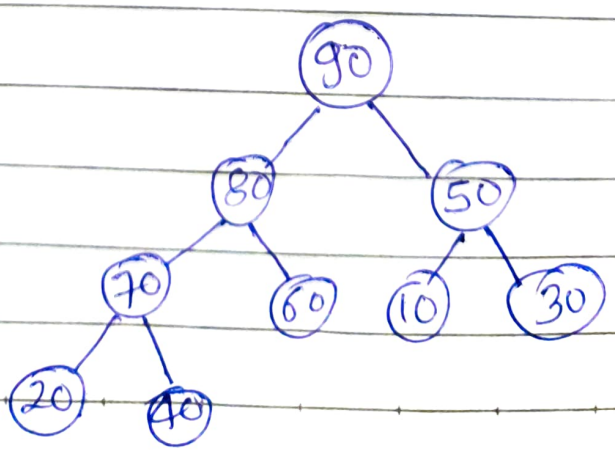
index of parent node of N = $\frac{(index-1)}{2}$

Insertion

	0	1	2	3	4	5	6	7	8
	40	70	10	90	60	30	50	20	80

	0	1	2	3	4	5	6	7	8
H	90	80	50	70	60	10	30	20	40

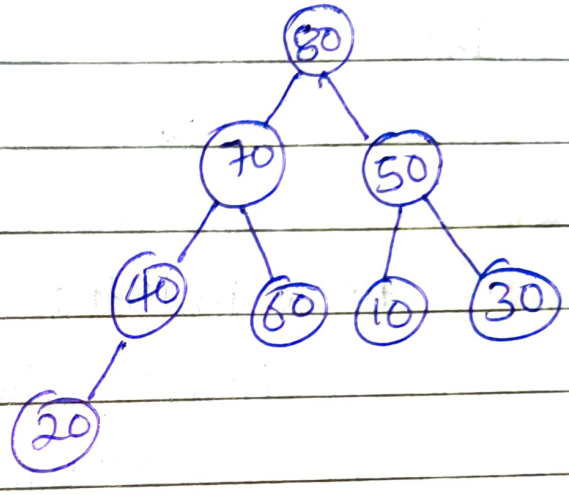
[ ]



Deletion.

	0	1	2	3	4	5	6	7	8
H	80	70	50	40	60	10	30	20	

[90]
temp
[40]



Heap Sort

→ Delete values from the heap (max-heap) and store them in an array from right to left. As a result, at the end of deleting all the elements of heap, array becomes sorted.

→ Heap can be used to implement Priority Queue.

Searching

① Linear Search

	0	1	2	3	4	5	6	7	8	9
list →	34	41	29	62	87	91	43	18	27	5

Item → 25

```
for e in list1:
    if e == item:
        return list1.index(e)
return None
```

Time Complexity = O(n)

② Binary Search

	0	1	2	3	4	5	6	7	8	9
	35	40	29	18	90	87	62	53	61	59

Sort →

	0	1	2	3	4	5	6	7	8	9
	18	29	35	40	53	59	61	62	87	90

(medil index) $m = \frac{l+u}{2}$

Time Complexity :→

$O(\log_2 n)$

```
if item == list1[m]
    return m
if item < list1[m]
    binary Search(list1, 0, 3)
else:
    binary Search(list1, 5, 9)
```

Hashing

Why Hashing

Hashing is designed to solve the problem of needing to efficiently find or store an item in a collection.

→ If we have a list of 1,00,000 words of English and we want to search a given word in the list, it should be inefficient to sucessively compare the word with all 100000 words until we find a match.

Hashing

It is a technique of mapping (hashing) keys, and values into the hash table by using function.

→ If implemented correctly, than one can achieve O(1) time complexity in searching an item in the collection of elements.

→ Efficiency of mapping depends on the efficiency of the hash function used.

Key Terms

Hash Table:- It is the data structure used to store elements.

Hash Function:- It is a function to map (Hash) key-values to the memory address.

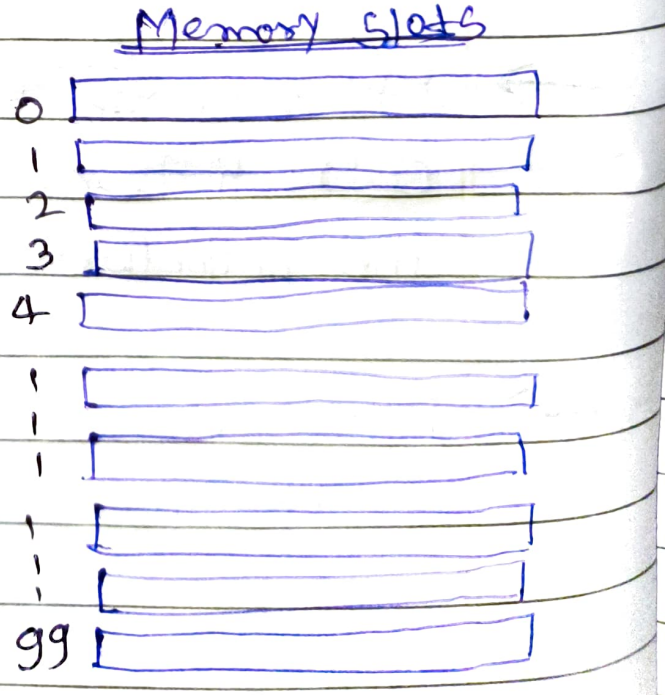
Hashing :- It is a method for storing and retrieving records from a database.

Suppose 100 Student records need to be stored in an array of 100 memory slots. Use hashing to perform efficient access of student data.

Roll NO → 1 to 100

key

```
i = HF (key)
def HF(int key)
{
    return key-1;
}
```



→ Hashing is a method of storing and retrieving records from data structure.

→ It lets you insert, delete and search records based on search key value.

→ When properly implemented these operations can be performed in constant time.

→ This is fast better then O(log2n) average cost required to do a binary search on a sorted array.

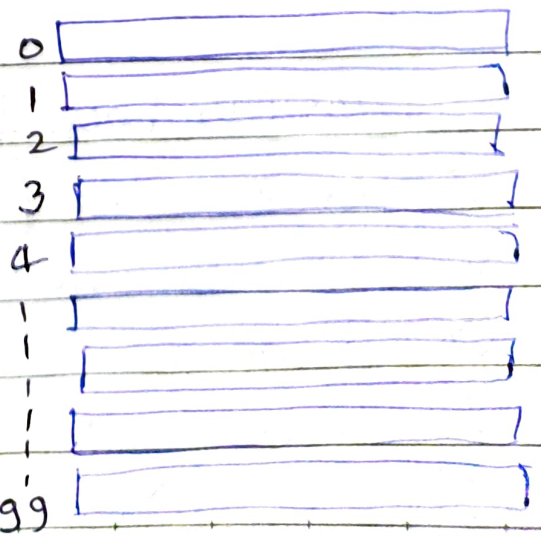
→ Hashing generally takes records whose key values come from a large range and stores those records in a table with a relatively small number of slots.

Roll No → 5 digit Number

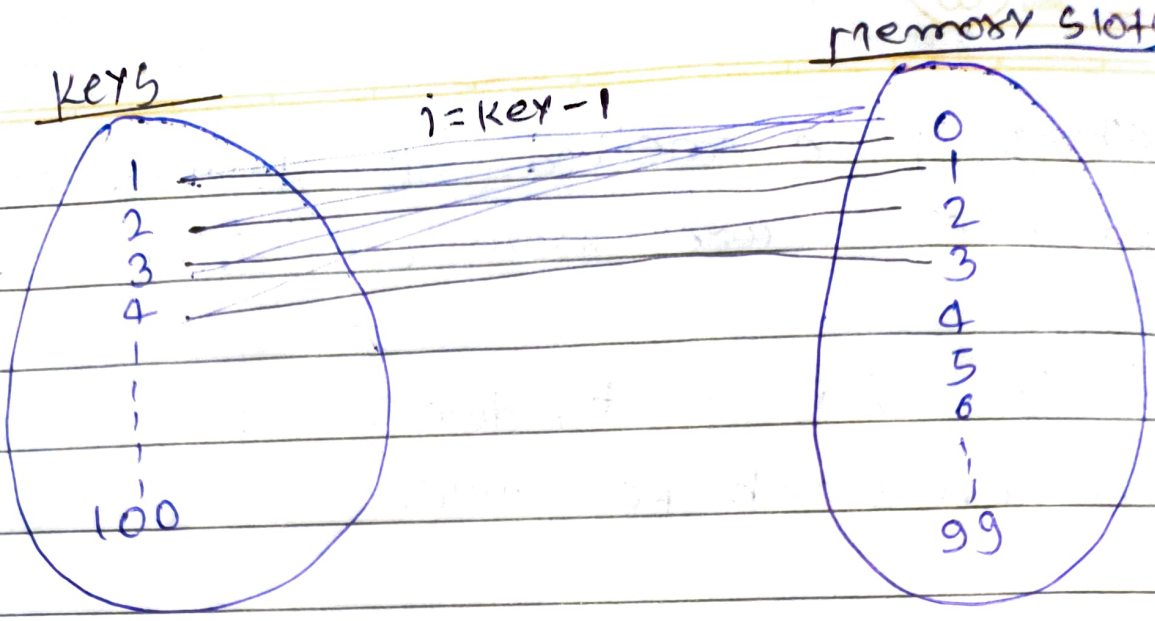
10000 to 99999

10025 → 25
25287 → 87
51128 → 28
48225 → 25

collision




```
def HF(key):
    return key % 100;
```



Collisions

Collision occurs when two records hash to the same slot in the table.

Unfortunately, even number under the best of circumstances, collisions are nearly unavoidable

Hash Function

The hash function creates a mapping between key and value, this is done through the use of mathematical formula.

Various hash functions.

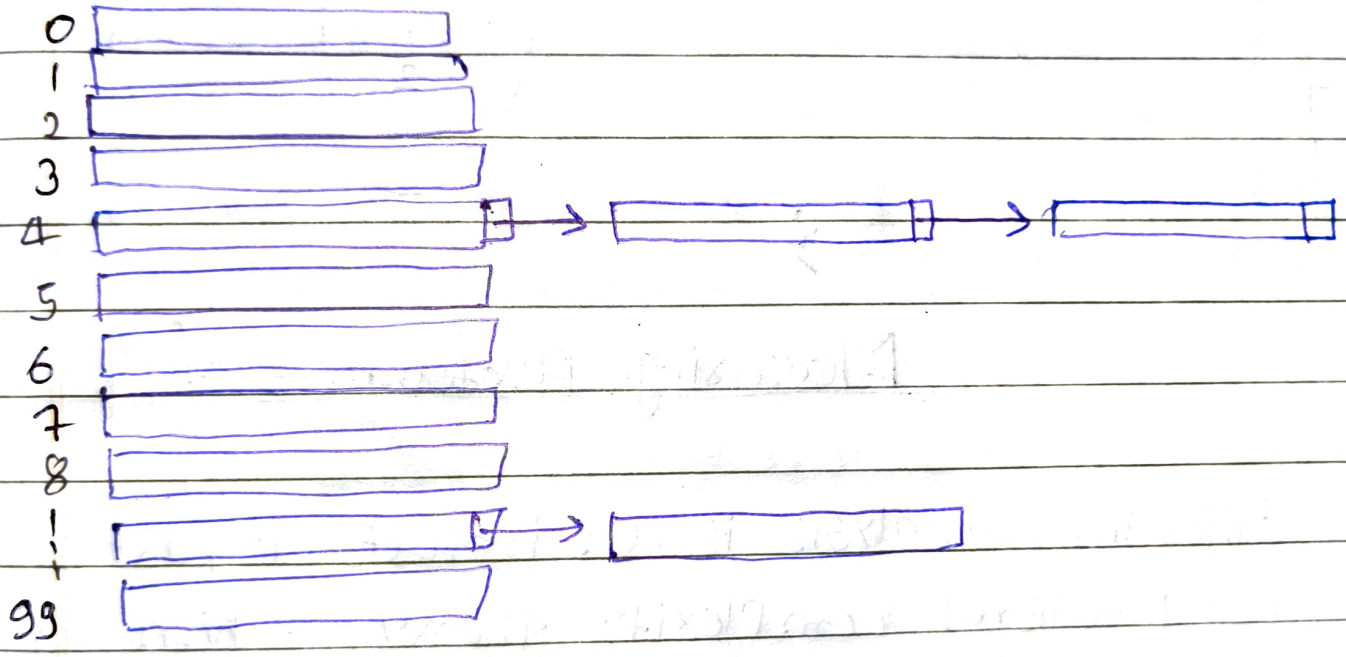
1. Simple mod function
2. Division method
3. Binning
4. mid Square Method.
5. ---- You can create your own functions.

Collision Resolution

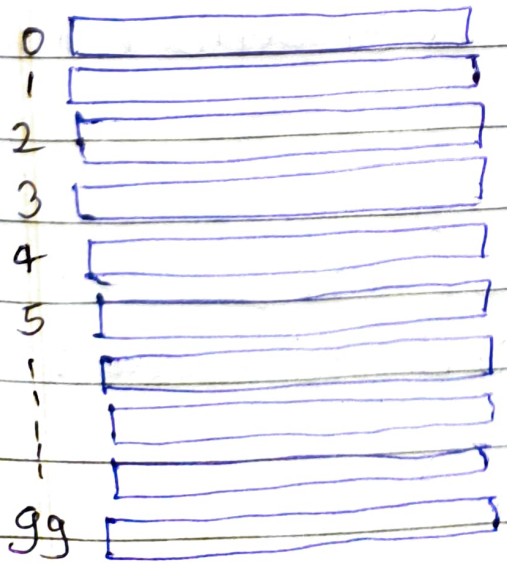
Two types of resolutions:-

1. Open Hashing (chaining)
2. Closed Hashing (Open addressing)

Open Hashing



Closed Hashing



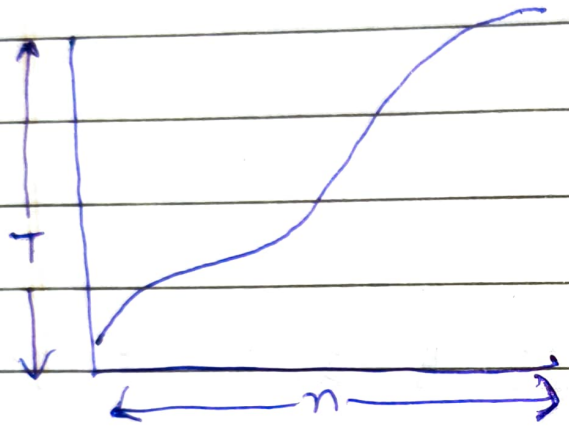
Programming

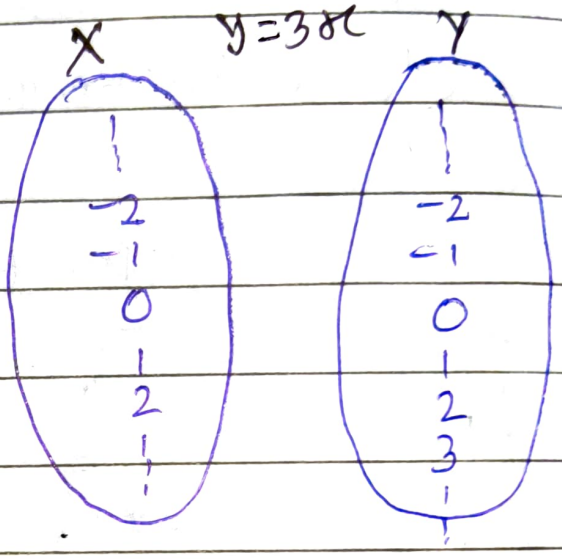
- → Linear Probing
- → Quadratic Probing
- → Double hashing

Time Complexity

Basics of Functions

$T = f(n)$ $[y = f(x)]$





Measuring Performance of Algorithms

Algorithm analysis is an important part of Computational Complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific Computational Problem.

It is the determination of the amount of time and space resources required to execute it.

Why Analysis is required?

→ Predicting behavior of an algorithm without implementing.

→ Analysis is only approximation
These are many influencing factors.

→ It helps us in determining the best algorithem to solve a Programming Problem.

Type of Algorithm Analysis

① Best case ② Worst case ③ Average case

→ Certain input takes less time for an algorithm to accomplish its task. This is **best case**.

→ Certain input takes max time for analgorithm to accomplish its task. This is **Worst case**.

→ All other cases / total no. of cases = Average case.

Time Complexity

It is a measure of rate of change in time with respect to change in input size.

→ Various asymptotic notations are these to represent time complexity of an algorithm.

Asymptotic Analysis

"A situation that can never be achieved while being close."

→ It is used to analyze performance of algorithms in terms of input size.

→ We donot calculate the actual time taken by the algorithm to solve the problem, rather we are interested in how the time taken by an algorithm increases with the input size.

Asymptotic Notation

Big O notation

Provides an upper bound on the growth rate of time. It represents the worst case scenario.

Omega notation

Provides a lower bound on the growth rate of time
It represents the best case scenario.

Theta notation

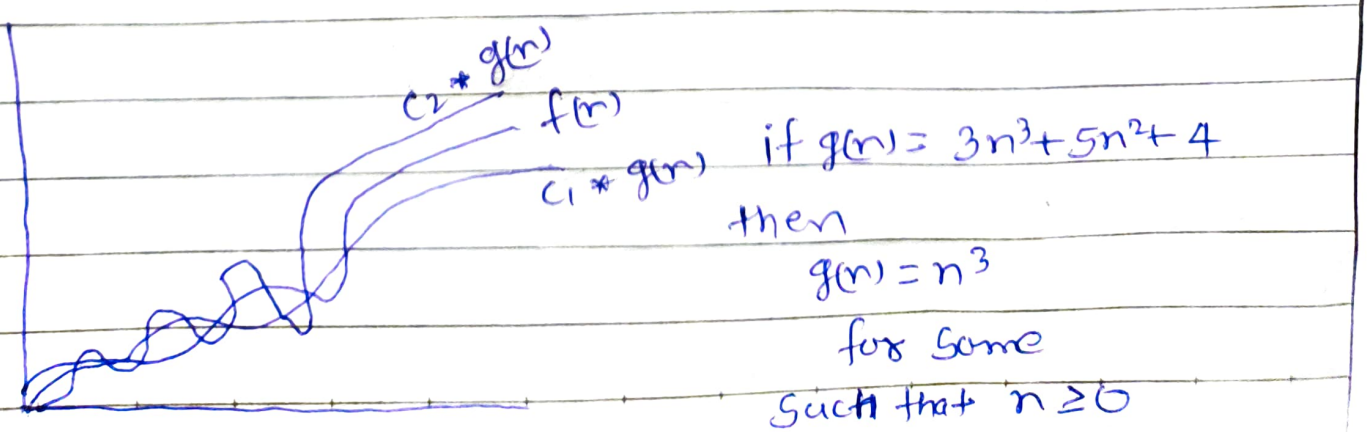
Provides both an upper bound and lower bound on the growth rate of time. It represents the average case scenario.

Theta Notation

$\Theta(g(n)) = f(n)$

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

c_1, c_2 are +ve contants for all $n \geq n_0$



if $g(n) = 3n^3 + 5n^2 + 4$

then

$g(n) = n^3$

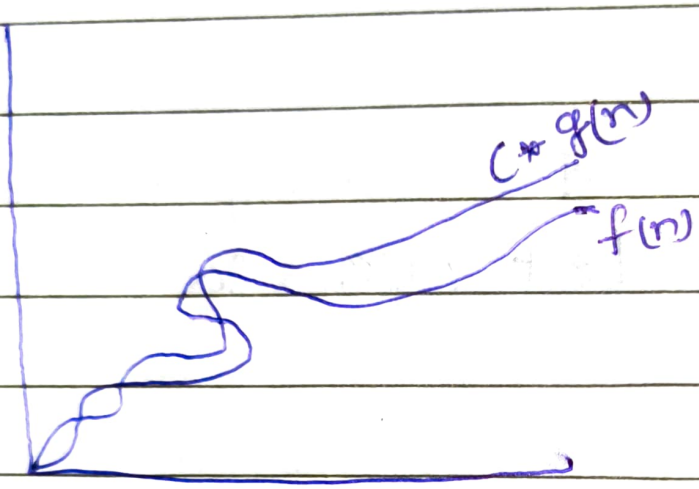
for some

such that $n \geq 0$

Big O notation

$O(g(n)) = f(n)$

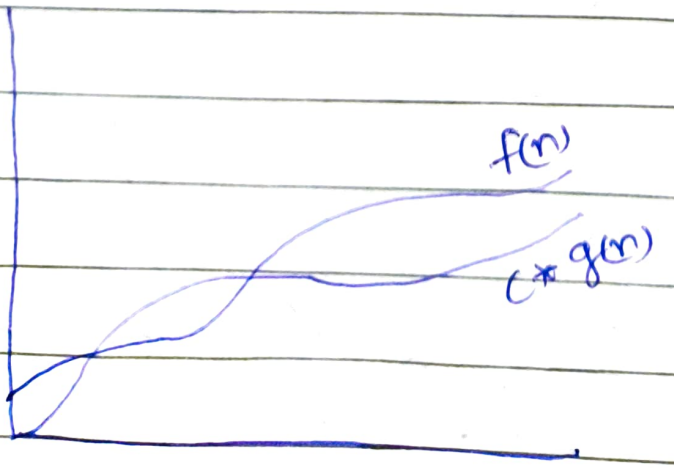
$0 \leq f(n) \leq c\, g(n)$
for all $n \geq n_0$



Omega notations

$\Omega(g(n)) = f(n)$

$c * g(n) \leq f(n)$
for all $n \geq n_0$



Calculate O(n)

① Factorial

② Insertion sort.

① Factorial

```
f = 1          ← c1
if n == 0      ← c2
    return 1
for i = 1 to n
    f = i * f  ← c3        } n · c3
return f       ← c4
```

$f(n) = c_3 \cdot n + c$

$g(n) = n$

$O(n)$

② Insertion sort

```
for (i = 1 to n)                          → c1
    temp = A[i]
    for (j = i-1 to 0)
        if (A[j] > temp):                 → c2   } i c2
            A[j+1] = A[j]
        else:
            break
    A[j+1] = temp                         → c3
```

$n(c_1 + n c_2 + c_3)$

$n(n c_2 + c)$

$f(n) = c_2 n^2 + c_n$

$g(n) = n^2$

$O(n^2)$

General perseption

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$$

$$< O(n^2 \log n) < O(n^3) < O(a^n)$$