Magic
DATE
PAGE

✯ Data Structure :→

Way to store and organize the data so that it can be accessed effectively.

Built-in	User defined
1. List	1. Stack
2. Tuple	2. Queue
3. Set	3. Linked List
4. Dictionary	4. Tree
	5. Graph

✯ List Data Structure :→

```
1. list = [1, 2, 3, 4]
2. list1 = list()
```

→ We can store data of different data type.

```
3. list2 = [1, 2, 3, "Sahil", 10.1]
```

→ Lists are ordered

```
list1 = [1, 2, 3, 4, 5]
list2 = [2, 1, 3, 4, 5]
list1 == list2   # False
```

→ We can access list items using positive indexing or Negative indexing.

```
list[4]   or   list[-1]
```

CamScanner

→ list can be nested

```
list = [[1,2],1,2,3]
```

→ Lists are mutable (changeable)

```
list = [1,2]
list.append(3) # list is [1,2,3]
```

→ Lists are Dynamic.

* Tuple Data Structure :→

```
1. tpl = (1,2,3,4)
2. tpl = tuple()
```

→ Tuples are ~~un-m~~ immutable (unchangeable)
→ ~~Can~~ Stores the data off different type
→ We can access items using positive or negative indexing.
→ Tuples can be nested.

Why we need tuple?
→ Tuple is faster than the list.
→ Data can not be modified.

✱ Set Data Structure →

→ Collection of unique elements.

→ Empty { } denotes dictionary data structure. not to set data structure.

→ To create empty set we need to use set() constructor.

1. S = set()

2. S = set("hello")
   # {'h', 'e', 'o', 'l'}
   ← Repeated elements are ignored

3. S = set([1, 2, 3, 4])
   # {1, 2, 3, 4}

4. S = {1, 2, 3, 4}
   # {1, 2, 3, 4}

→ Set is unordered
   S = {1, 2}
   S1 = {2, 1}
   S == S1 # True

→ Set is mutable
   S.add(3)

→ Set can have different types of immutable objects.

Magic
DATE
PAGE

★ Dictionary Data Structure →

Syntax:-

Dictionary-name = {Key1: value 1, Key2 : value 2}

1. d = { } # Empty Dictionary.
2. d = {"Sahil" : "sahil8619@gmail.com"}
3. d = dict() # Empty Dictionary Using Constructor

#. → Keys are unique.

→ We can not use mutable type as a key.

→ Dictionary are Unorder.

→ Element can't access using indexing.

→ We can access elements using key.

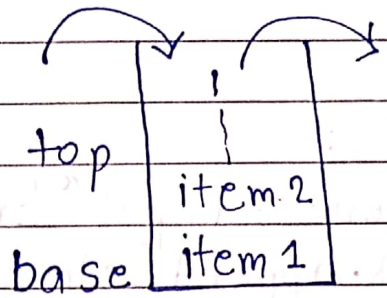
→ Dictionary are mutable.

→ Dictionary can be nested.

CamScanner

Stack

Last In First Out(LIFO)
OR
First In Last Out(FILO)



* Operations :-

1. push :- Add element
2. Pop :- Remove element
3. Peek or top :- Fetch top element
4. isEmpty :- Empty or Not

* Implement Stack :-

1. List
2. Modules.

1. List :-

Push ---> append
Pop ---> pop

Eg. Program :-

```
stack = []
stack.append(10)      # [10]
stack.append(20)      # [10,20]
stack.pop()
20
stack.pop()
10
```

Program:-

```
stack = []
def push():
    element = input("Enter the element:")
    stack.append(element)
    print(stack)
def pop_ele():
    if not stack:
        print("Stack is empty!")
    else:
        e = stack.pop()
        print("removed element:", e)
        print(stack)

while True:
    print("Select the operation 1. push 2. pop
                                3. quit")
    choice = int(input())
    if choice == 1:
        push()
    elif choice == 2:
        pop_ele()
    elif choice == 3:
        break
    else:
        print("Enter the correct operation!")
```

2. Modules :-

(i) Collections Modules :-

↳ deque

Program :-

```
import collections
stack = collections.deque()
stack  # deque([])
stack.append(10)
stack.append(20)
stack.  # deque([10, 20])
stack.pop()  # 20
```

(ii) Queue Module :-

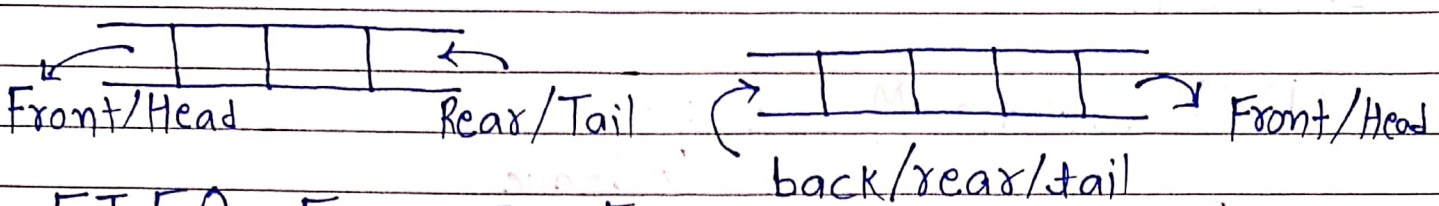
↳ LifoQueue()

Program :-

```
import queue
stack = queue.LifoQueue()
stack.put(10)
stack.put(20)
stack                    # (10, 20)
stack.get()  # 20
```

CamScanner

Queue



FIFO : First In First Out
LILO : Last In Last Out

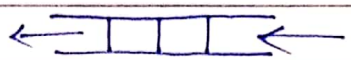
* Operations:-
  - enqueue- Adding element
  - dequeue :- Removing element.
  - isFull :- Full or Not
  - isEmpty :- Empty or Not

* Implementation Using List :-

1. enqueue :-
   Process of adding elements to queue.
2. dequeue :-
   Process of removing the elements from the queue.

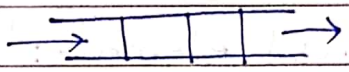
enqueue : append method
dequeue : pop method : list.pop(0)

Eg. Program:-



```
queue = []
queue.append(10)
queue.append(20)
queue.append(30)
queue  # [10, 20, 30]
queue.pop(0)   # 10
queue.pop(0)   # 20
```

OR



```
queue = []
queue.insert(0, 10)
queue.insert(0, 20)
queue.insert(0, 30)
queue  # [30, 20, 10]
queue.pop()   # 10
queue.pop()   # 20
```

Empty:-

```
queue = []
not queue   # True. (Empty)
```

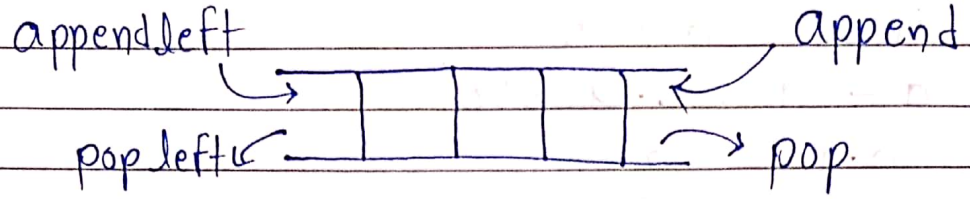
Program:-

```
queue = []
def enqueue():
    element = input("Enter the element:")
    queue.append(element)
    print(element, "is added to queue!")

def dequeue():
    if not queue:
        print("queue is empty!")
    else:
        e = queue.pop(0)
        print("Removed element:", e)

def display():
    print(queue)

while True:
    print("Select the operation 1. add 2. remove
          3. show 4. quit")
    choice = int(input()).
    if choice == 1:
        enqueue()
    elif choice == 2:
        dequeue()
    elif choice == 3:
        display()
    elif choice == 4: break
    else: printf("Enter the correct operation!")
```

CamScanner

☆ Implementation Using Collections :-

1. dequeue → Collections :-



Program :-

```
import collections
q = collections.deque()
q  # deque([])
q.appendleft(10)
q.appendleft(20)
q  # deque([20,10])
q.pop()    # 10
q.pop()    # 20
q.append(10)
q.append(20)
q  # deque([10,20])
q.popleft()  # 10
q.popleft()  # 20

not q  # True (Empty)
```

CamScanner

2. queue → Queue :-

Program :-

```
import queue
q = queue.Queue()
q.put(10)
q.put(20)
q.put(30)
q.get()  # 10
q.get()  # 20
q.get()  # 30
```

CamScanner

★ Priority Queue :-

1. Low → high Priority.
2. high → high Priority.

1. Implementation Using List :-

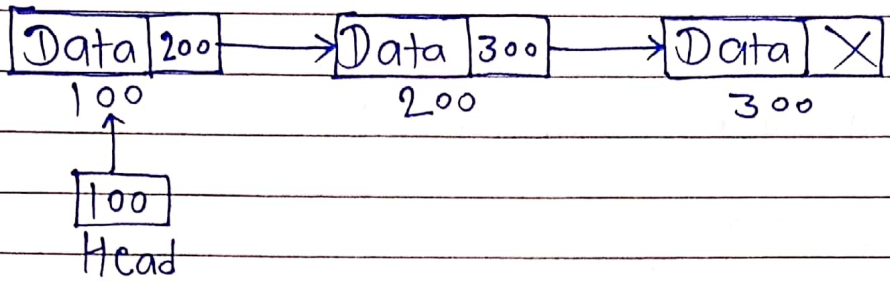
```
q = []
q.append(10)
q.append(20)
q.append(40)
q.sort()
q   # [10,20,40]
q.pop()   #10
q.pop()   # 20
```

2. Implementation Using Priority Queue Class :-

```
import queue
q = queue.PriorityQueue()
q.put(10)
q.put(60)
q.put(20)
q.put(40)
q.get()   # 10
q.get()   # 20
q.get()   # 40
q.get()   # 60
```

Linked List

Data	Link

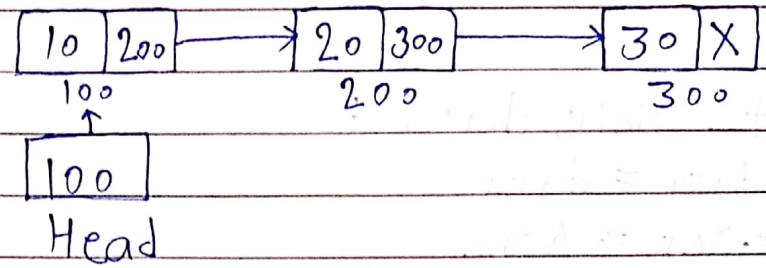


→ Node: Elements of linked list.
→ Dynamic data structure.
→ Insertion and deletion is easy.
→ Implementation :- Stack, Queue and Graph.
→ Represent and Manipulate polynomials.
→ Need Extra Memory.
→ Random access not possible.

★ Types :-

1. Single Linked List.
2. Doubly Linked List.
3. Circular Linked List.

☆ Single Linked List :-



Operations :-

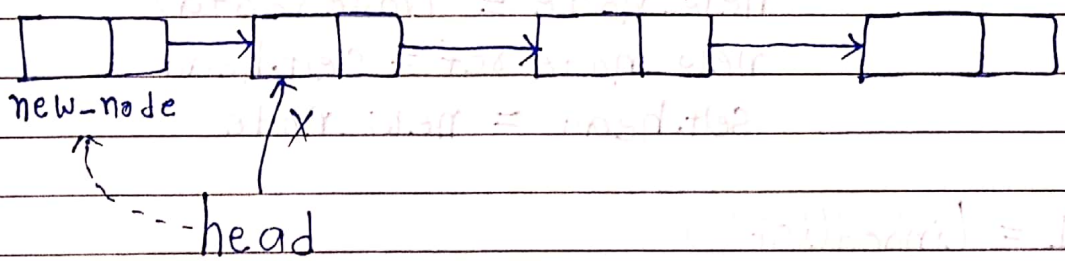
1. Add (begin, end, inbetween)
2. Delete (begin, end, inbetween)
3. Traversal

1. Add / Insertion node :-

(i) Add at begin :-

Steps :-

1. Create Node
2. new_node.ref = head
3. head = new_node



CamScanner

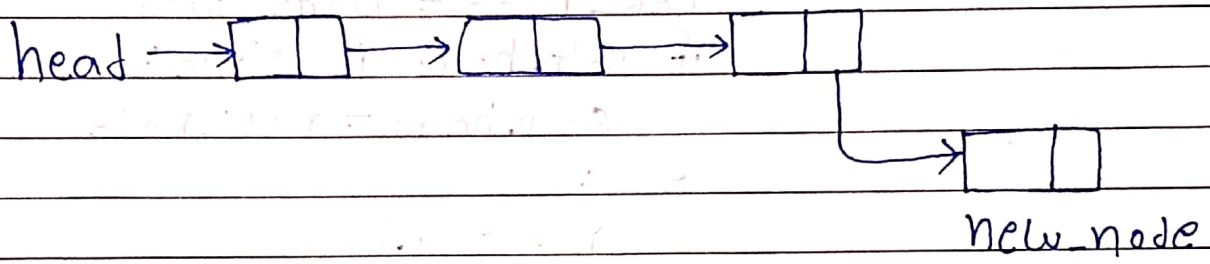
Program:-

```
class Node:
    def __init__(self, data):
        self.data = data
        self.ref = None

class LinkedList:
    def __init__(self):
        self.head = None

    def print_LL(self):
        if self.head is None:
            print("Linked list is empty!")
        else:
            n = self.head
            while n is not None:
                print(n.data)
                n = n.ref

    def add_begin(self, data):
        new_node = Node(data)
        new_node.ref = self.head
        self.head = new_node

LL1 = LinkedList()
LL1.add_begin(10)
LL1.print_LL()
```

(ii) Adding node at the end:-

Steps:-

1. Create Node
2. Go to last node (n)
3. n.ref = new_node



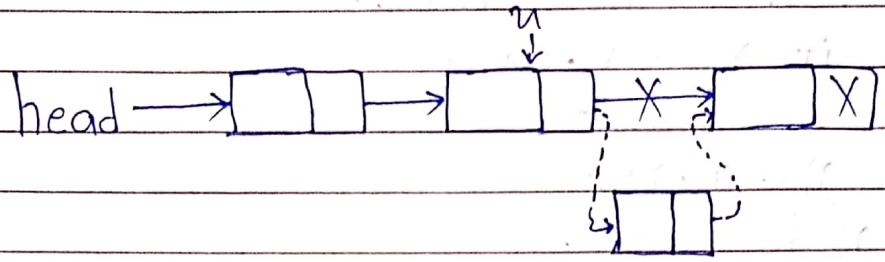
Program:-

```
class Node:
    def __init__(self, data):
        self.data = data
        self.ref = None

class LinkedList:
    def __init__(self):
        self.head = None

    def print_ll(self):
        if self.head is None:
            print("Linked List is empty!")
```

```
        else:
            n = self.head
            while n is not None:
                print(n.data)
                n = n.ref

    def add_end(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            n = self.head
            while n.ref is not None:
                n = n.ref
            n.ref = new_node

LL1 = LinkedList()
LL1.add_end(10)
LL1.add_end(20)
LL1.print_LL()
```

(iii) Add element Inbetween node:-

(A). Insert after given node:-

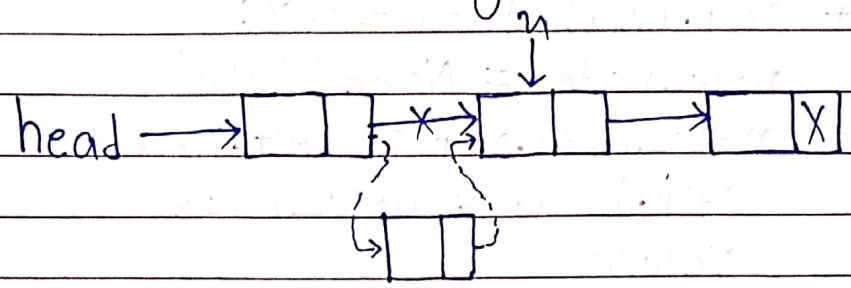


Program:-

```
class Node:
    def __init__(self,data):
        self.data = data
        self.ref = None

class LinkedList:
    def __init__(self):
        self.head = None
    def print_LL(self):
        if self.head is None:
            print("Linked list is empty!")
        else:
            n= self.head
            while n is not None:
                print(n.data,"-->", end = " ")
                n = n.ref
```

```
def add_after(self, data, x):
    n = self.head
    while n is not None:
        if x == n.data
            break
        n = n.ref
    if n is None:
        print("Node is not present")
    else:
        new_node = Node(data)
        new_node.ref = n.ref
        n.ref = new_node

ll1 = LinkedList()
ll1.add_after(200, 100)
ll1.print_ll()
```

(B) Insert before given node →



Program:-

```
class Node:
    def __init__(self, data):
        self.data = data
        self.ref = None

class LinkedList:
    def __init__(self):
        self.head = None
    def print_LL(self):
        if self.head is None:
            print("Linked List is empty!")
        else:
            n = self.head
            while n is not None:
                print(n.data, "--->", end=" ")
                n = n.ref
```

```
def add_before(self, data, x):
    if self.head.data == x:
        new_node = Node(data)
        new_node.ref = self.head
        self.head = new_node
        return
    if self.head is None:
        print("Empty")
        return

    n = self.head
    while n.ref is not None:
        if n.ref.data == x:
            break
        n = n.ref
    if n.ref is None:
        print("Node is not found!")
    else:
        new_node = Node(data)
        new_node.ref = n.ref
        n.ref = new_node

LL1 = LinkedList()
LL1.add_before(20,10)
LL1.print_LL()
```

Swap

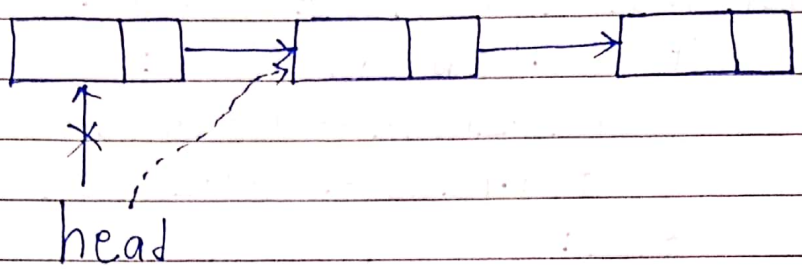
(c) Adding element into empty linked list :-

```
def insert_empty (self, data):
    if self.head is None:
        new_node = Node (data)
        self.head = new_node
    else:
        print("Linked list is not empty!")

LL1 = LinkedList()
LL1.insert_empty(10)
LL1.insert_empty(20)
LL1.print_LL()
```

CamScanner

2. Deletion Node :-

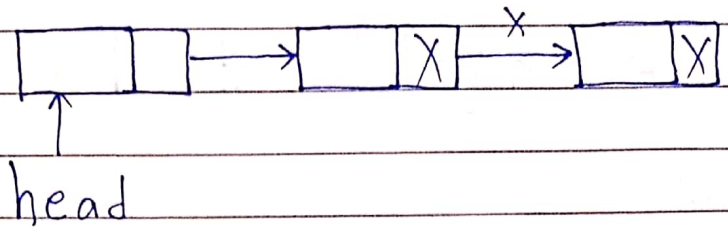
(i) Delete Node at begining :-



Program :-

```
def delete_begin(self):
    if self.head is None:
        print("LinkedList is empty!")
    else:
        self.head = self.head.ref
```

(ii) Delete node at the end:-

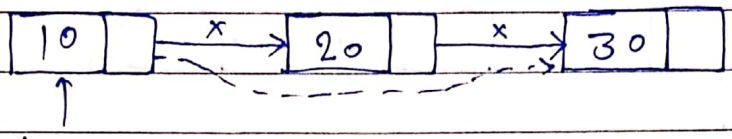


Program:-

```
def delete_end(self):
    if self.head is None:
        print("Linked List is empty!")
    else:
        n = self.head
        while n.ref.ref is not None:
            n = n.ref
        n.ref = None
    elif self.head.ref is None:
        self.head = None
```

(iii) Delete node in middle or by value :-

x = 20



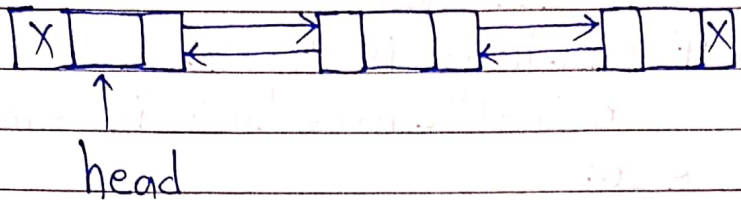
Program :-

```
def delete_by_value(self, x):
    if self.head is None:
        print("Linked List is empty!")
        return
    if x == self.head.data:
        self.head = self.head.ref
        return
    n = self.head
    while n.ref is not None:
        if x == n.ref.data:
            break
        n = n.ref
    if n.ref is None:
        print("Node is not present!")
    else:
        n.ref = n.ref.ref
```

3. Traversal:-

```
def print_LL(self):
    if self.head is None:
        print("Linked List is empty!")
    else:
        n = self.head
        while n is not None:
            print(n.data, "--->", end=" ")
            n = n.ref
```

☆ Doubly Linked List:-



Operations:-

→ Insertion (Begin, End, Middle)
→ Deletion (Begin, End, Middle)
→ Traversal (Forward, Backward)

1. Traversal:-

Program:-

```
class Node:
    def __init__(self, data):
        self.data = data
        self.nref = None
        self.pref = None

class doublyLL:
    def __init__(self):
        self.head = None
```

```
def print_ll(self):  #Forward traversal
    if self.head is None:
        print("Linked list is empty!")
    else:
        n = self.head
        while n is not None:
            print(n.data, "-->", end=" ")
            n = n.nref

def print_ll_reverse(self):
    if self.head is None:
        print("Linked list is empty!")
    else:
        n = self.head
        while n.nref is not None:
            n = n.nref
        while n is not None:
            print(n.data, "-->", end=" ")
            n = n.pref

dll = DoublyLL()
dll.print_ll()
dll.print_ll_reverse()
```

2. Insertion :-

- → Insertion when empty
- → At begin
- → At end
- → After a node
- → Before a node

Program :-

```
class Node:
    def __init__(self, data):
        self.data = data
        self.nref = None
        self.pref = None

class doublyLL:
    def __init__(self):
        self.head = None

    def print_LL(self):
        if self.head is None:
            print("Linked List is empty!..")
        else:
            n = self.head
            while n is not None:
                print(n.data, "--->", end = " ")
                n = n.nref
```

```
def insert_empty(self, data):
    if self.head is None:
        new_node = Node(data)
        self.head = new_node
    else:
        print("Linked list is not empty!")

def add_begin(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
    else:
        new_node.nref = self.head
        self.head.pref = new_node
        self.head = new_node

def add_end(self, data):
    new_node = Node(data)
    if self.head is None:
        self.head = new_node
    else:
        n = self.head
        while n.nref is not None:
            n = n.nref
        n.nref = new_node
        new_node.pref = n
```

```
def add_after(self, data, x):
    if self.head is None:
        print("Linked List is empty!")
    else:
        n = self.head
        while n is not None:
            if x == n.data:
                break
            n = n.nref
        if n is None:
            print("Given node is not
                  present in LL")
        else:
            new_node = Node(data)
            new_node.nref = n.nref
            new_node.pref = n
            if n.nref is not None:
                n.nref.pref = new_node
            n.nref = new_node


def add_before(self, data, x):
    if self.head is None:
        print("LL is empty!")
    else:
        n = self.head
```

```
while n is not None:
    if x == n.data:
        break
    n = n.nref
if n is None:
    print("Node not present in LL")
else:
    new_node = Node(data)
    new_node.nref = n
    new_node.pref = n.pref
    if n.pref is not None:
        n.pref.nref = new_node
    else:
        self.head = new_node
    n.pref = new_node
```

```
dl1 = doublyLL()
dl1.add_begin(4)
dl1.add_after(10,4)
dl1.add_before(20,10)
dl1.add_end(30)
dl1.print_LL()
```

Magic
DATE
PAGE

3. Deletion :-

→ Begin
→ End
→ By value.

Program :-

```
Class Node:
    def __init__(self, data):
        self.data = data
        self.nref = None
        self.pref = None

class doublyLL:
    def __init__(self):
        self.head = None

    def print_LL(self):
        if self.head is None:
            print("Linked list is empty!")
        else:
            n = self.head
            while n is not None:
                print(n.data, "--->", end = " ")
                n = n.nref
```

CamScanner

```
def delete_begin(self):
    if self.head is None:
        print("DLL is empty!")
        return
    if self.head.nref is None:
        self.head = None
        print("DLL is empty after deleting node!")
    else:
        self.head = self.head.nref
        self.head.pref = None

def delete_end(self):
    if self.head is None:
        print("DLL is empty!")
        return
    if self.head.nref is None:
        self.head = None
        print("DLL is empty after deleting the node")
    else:
        n = self.head
        while n.nref is not None:
            n = n.nref
        n.pref.nref = None
```

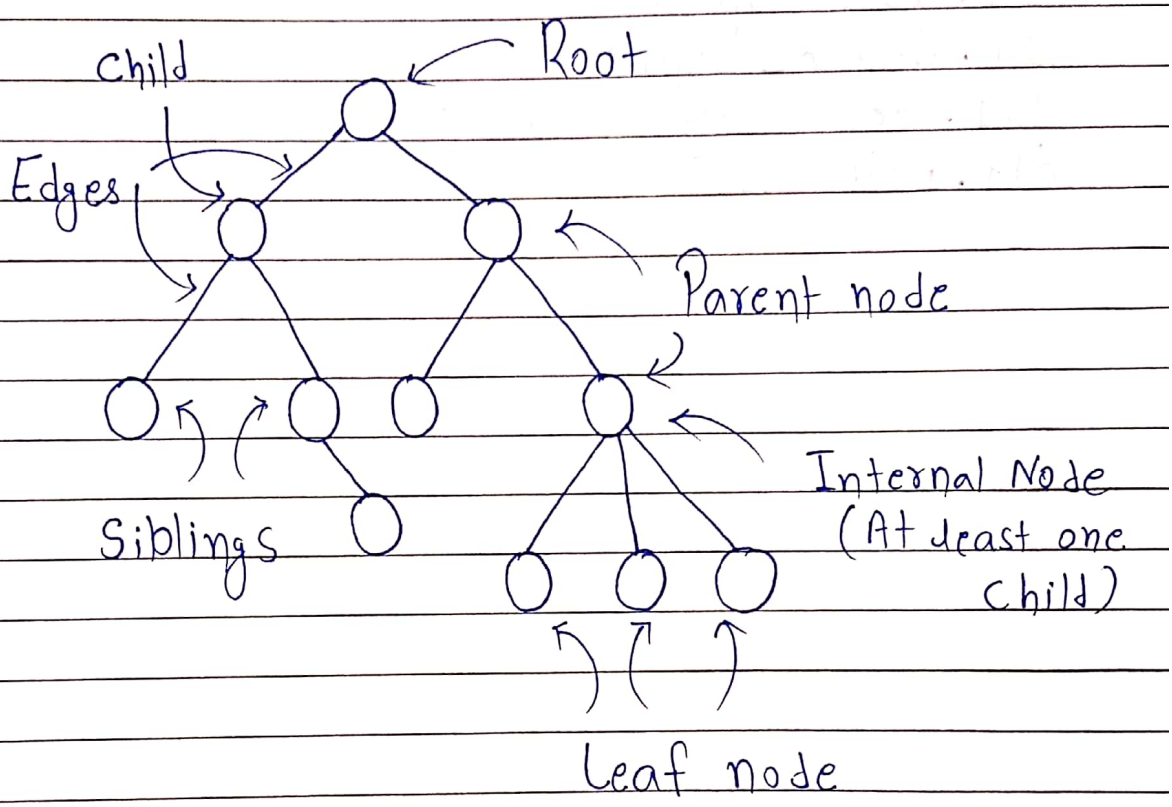
```
def delete_by_value(self, x):
    if self.head is None:
        print("DLL is empty!")
        return
    if self.head.nref is None:
        if x == self.head.data:
            self.head = None
        else:
            print("x is not present")
        return
    if self.head.data == x:
        self.head = self.head.nref
        self.head.pref = None
        return
    n = self.head
    while n.nref is not None:
        if x == n.data:
            break
        n = n.nref
    if n.nref is not None:
        n.nref.pref = n.pref
        n.pref.nref = n.nref
    else:
        if n.data == x:
            n.pref.nref = None
        else:
            print("x is not present")
```

```
dll = doublyLL()
dll.add_begin(4)
dll.add_before(10,4)
dll.delete_end()
dll.add_begin(10)
dll.add_begin(20)
dll.delete_begin()
dll.delete_by_value(10)
dll.print_LL()
```

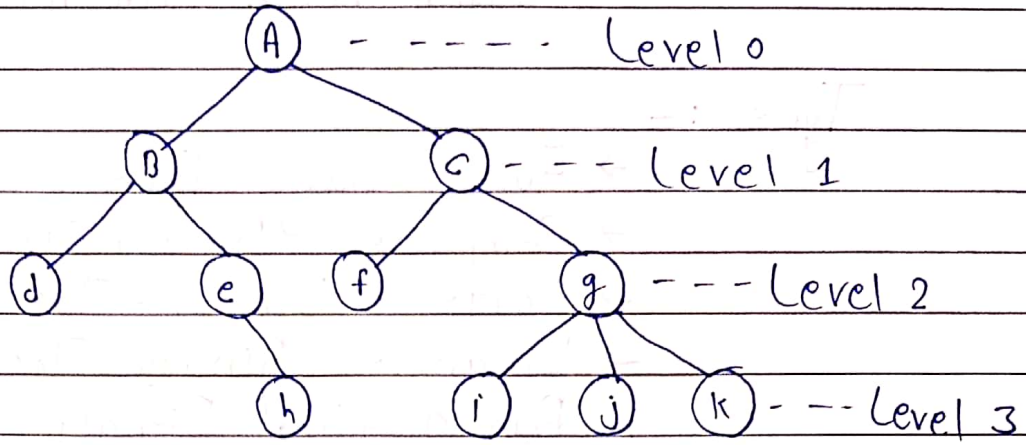
CamScanner

Tree

- → Non-linear Data structure
- → Represents relationship between nodes
- → Collection of entities called Nodes
- → Nodes are connected by edges



Characteristics :-



1. Root :- Top most node.
2. In a tree if we have N nodes then we will have N-1 edges/link.
3. Every child will have only one parent.
4. Tree is a recursive data structure.
5. Degree of Node :- Total no. of child of that node
6. Degree of a Tree :- Highest children of node
7. Level :- Each step from top to bottom is called level.
8. Hight of a node :- Total no. of edges that lies into longest path from any leaf node to particular leaf node.

   C → 2
   B → 2
   A → 3

9. Hight of a tree :- A → 3
10. Depth of node :- Root to node. (C → 1)
11. Depth of tree :- Root to leaf (longest path).

CamScanner

✱ Binary Tree :-

Each node can have 0, 1, 2 childs.

Types :-

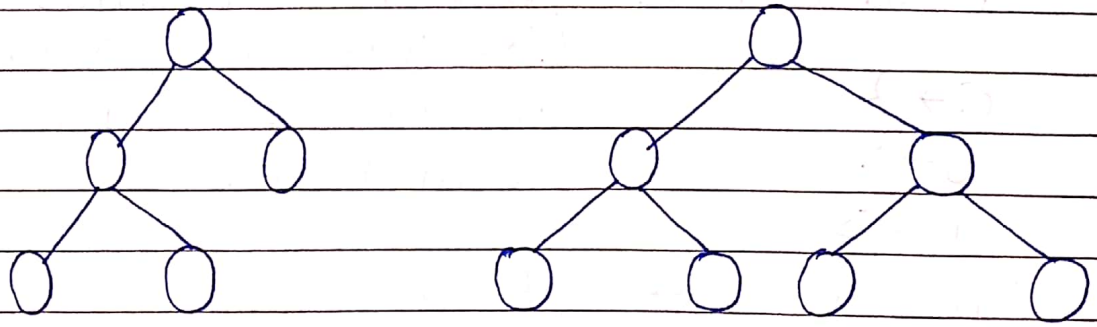
- → Full Binary Tree
- → Complete Binary Tree
- → Perfect Binary Tree
- → Balanced Binary Tree
- → Pathological Binary Tree

1. Full Binary Tree :-

Every node can have 0, 2 child nodes

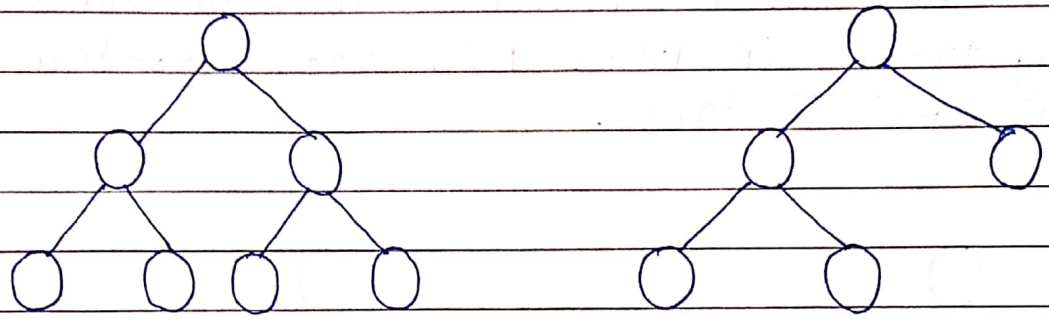
Full binary tree is a type of binary tree in which every node other than leaf nodes has 2 children.

Ex:-



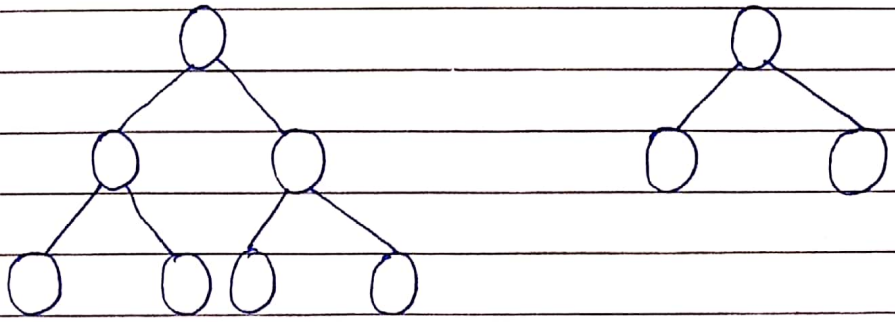
2. Complete Binary Tree :-

A complete binary tree is a binary tree in which all levels of tree are completely filled, except possibly for the last level, which is filled from left to right.



3. Perfect Binary Tree :-

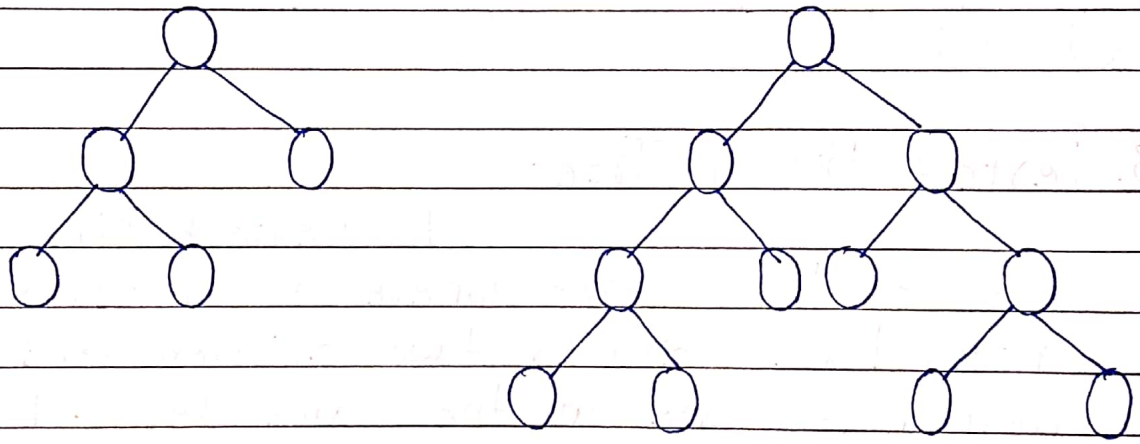
A perfect binary tree of binary tree where all internal nodes have exactly two children, and all leaf nodes are at the same level. Additionally all the nodes in the last level are as far left as possible.



4. Balanced Binary Tree:-

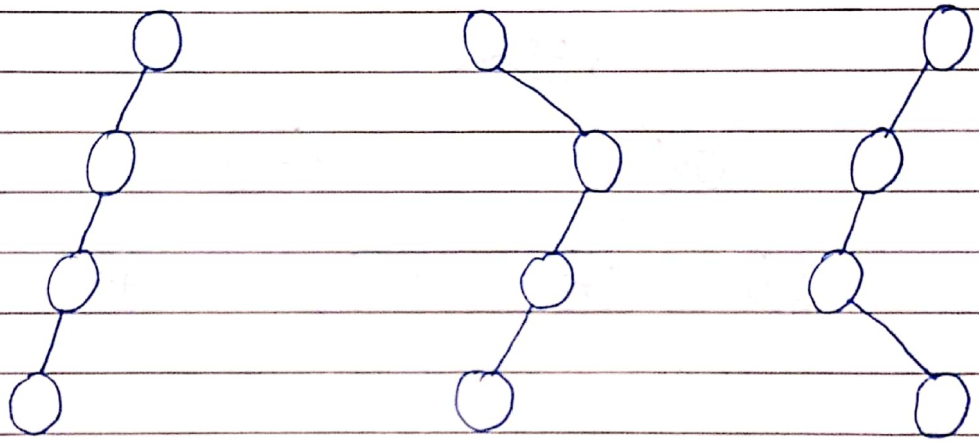
A balanced binary tree is a type of binary tree in which the difference in height between the left and right subtree of any node is at most one.

There are several different types of balanced binary trees, including AVL trees, red-black trees, and B-trees.



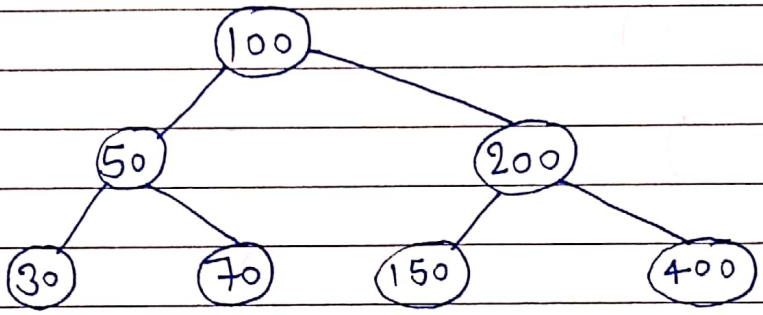
5. Pathological (Degenerated) Binary Tree :-

A degenerate binary tree is a tree where each parent node has only one child or no child at all.
This means that the tree is essentially a linear data structure, resembling a linked list rather than a tree.



☆ Binary Search Tree :-

↳ The left subtree of a node contains only nodes with keys lesser than node's key.
↳ The right subtree of a node contains only nodes with keys greater than node's key.
↳ The left subtree and right subtree each must also be a BST.



↳ Duplicate values not allowed
↳ Left <= root/node < right
↳ Left < root/node <= right

Operations :-

1. Search :-
   (i) Check BST is empty :
      if yes : print a message
      No : Compare root key with given value
   (ii) Root == Given value :
      if yes : then print key found
      No : Check where to search for key.
   (iii) Given value < Root key :
      if yes : Search left subtree, start from step 1.
      No : Search right subtree, start from step 1.

2. Insertion :-
   (i) Check BST is empty :
      if yes : Insert the new node
      No : Compare root key with given value
   (ii) Root key < Given value :
      if yes : then goto right subtree, find the correct position of newnode, insert.
      No : then goto left subtree, find ~~newnode.~~ the correct position of newnode insert.

3. Deletion :-

(i) If the node to be deleted is a leaf node, simply remove it from the tree.

(ii) If the node to be deleted has only one child, replace the node with its child.

(iii) If the node to be deleted has two children, find the node with the minimum key in its right subtree and replace the node to be deleted with this node. Then, deleted the node with the minimum or maximum key from its original position.

4. Traversal :-

(i) Pre-order Traversal :-

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

① Visit the root node.

② Traversing the left sub-tree.

③ Traversing the right sub-tree.

CamScanner

(ii) In-order traversal:-

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

① Traversing the left sub-tree
② Visit the root node
③ Traversing the right sub-tree

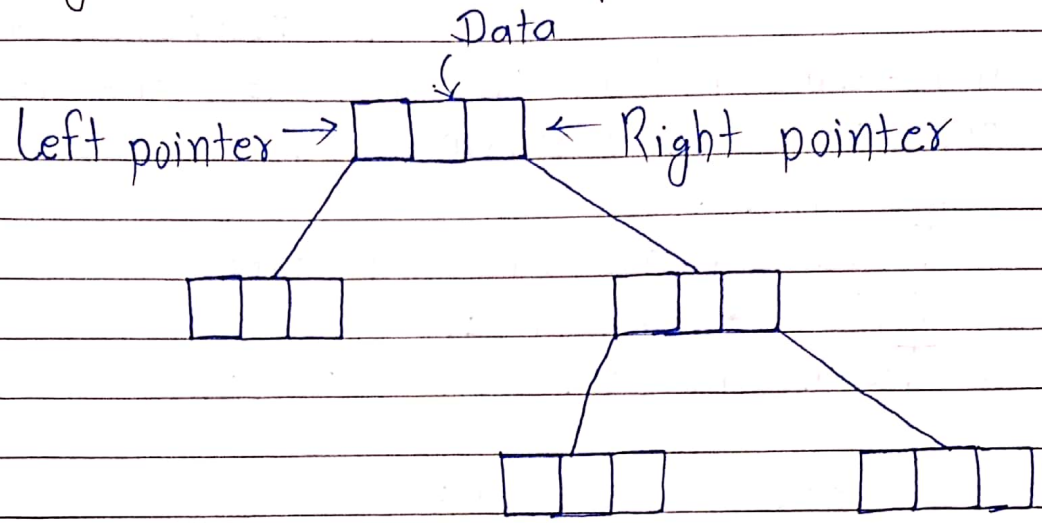
(iii) Post-order Traversal:-

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

① Traversing the left sub-tree
② Traversing the right sub-tree
③ Visit the root node.

CamScanner

✯ Binary Search Tree Implementation:-



Program:-

```
class BST:
    def __init__(self, key):
        self.key = key
        self.lchild = None
        self.rchild = None

root = BST(10)
print(root.key)
print(root.lchild)
print(root.rchild)
```

1. Insertion :-

Program :-

```
class BST:
    def __init__(self, key):
        self.key = key
        self.lchild = None
        self.rchild = None
    def insert(self, data):
        if self.key is None:
            self.key = data
            return
        if self.key > data:
            if self.lchild:
                self.lchild.insert(data)
            else:
                self.lchild = BST(data)
        else:
            if self.rchild:
                self.rchild.insert(data)
            else:
                self.rchild = BST(data)
        if self.key == data:
            return
root = BST(10)
list = [20, 4, 30, 4, 1, 5, 6]
for i in list:
    root.insert(i)
```

CamScanner

2. Search:-

Program:-

```
def search(self, data):
    if self.key == data:
        print("Node is found!")
        return
    if data < self.key:
        if self.lchild:
            self.lchild.search(data)
        else:
            print("Node is not present!")
    else:
        if self.rchild:
            self.rchild.search(data)
        else:
            print("Node is not present!")

root = BST(10)
list = [6, 3, 1, 98, 3, 7]
for i in list:
    root.insert(i)
root.search(7)
```

3. Traversal:-

(i) Pre-order:-

Program:-

```
def preorder(self):
    print(self.key, end = " ")
    if self.lchild:
        self.lchild.preoder()
    if self.rchild:
        self.rchild.preorder()
```

(ii) In-order:-

Program:-

```
def inorder(self):
    if self.lchild:
        self.lchild.inorder()
    print(self.key, end = " ")
    if self.rchild:
        self.rchild.inorder()
```

(iii) Post-order:-

Program:-

```
def postorder(self):
    if self.lchild:
        self.lchild.postorder()
    if self.rchild:
        self.rchild.postorder()
    print(self.key, end = " ")
```

CamScanner

4. Deletion:-

Program:-

```
def delete(self, data):
    if self.key is None:
        print("Tree is empty!")
        return
    if data < self.key:
        if self.lchild:
            self.lchild = self.lchild.delete(data)
        else:
            print("Given node is not present")
    elif data > self.key:
        if self.rchild:
            self.rchild = self.rchild.delete(data)
        else:
            print("Given node is not present")
    else:
        if self.lchild is None:
            temp = self.rchild
            self = None
            return temp
        if self.rchild is None:
            temp = self.lchild
            self = None
            return temp
        node = self.rchild
        while node.lchild:
            node = node.lchild
```

```
# Outside the while loop
self.key = node.key
self.rchild = self.rchild.delete(node.key)
return self
```

5. Min and Max node:-

Program:-

```
def min_node(self):
    current = self
    while current.lchild:
        current = current.lchild
    print("Smallest key is:", current.key)

def max_node(self):
    current = self
    while current.rchild:
        current = current.rchild
    print("Maximum key is:", current.key)
```

Binary Heap

Magic
DATE
PAGE

★ Min heap / Min binary heap :-

Complete binary tree where the key of every parent node is less than or equal to child node's key.

★ Max heap / Max binary heap :-

Complete binary tree, where the key of every parent node is greater than or equal to child node's key.

★ Operations :-

1. Heapify :-

It is a process to rearrange the elements of the heap in order to maintain the heap property.

- Make sure that every node of tree follows heap property.
- Used to create binary heap from a complete binary tree.

→ Where use heapify operation :-

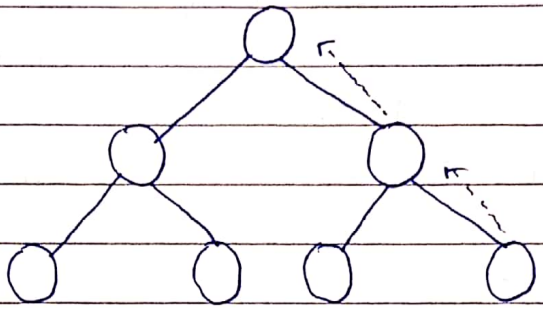
↳ Insertion operation
↳ Deletion operation
↳ While creating a binary heap from given array.

CamScanner

* There are two types of Heapify operation :-

(1.) heapify_up :-

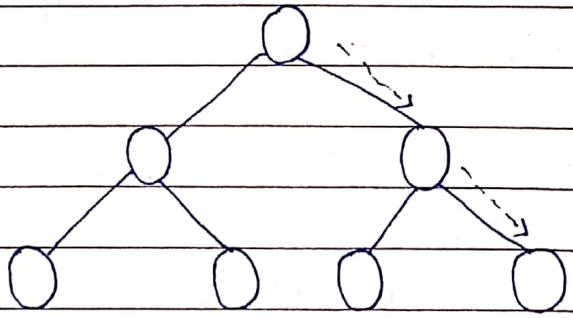
Perform this operation when we insert a new node to the binary heap.



Also known as:- up_heap, bubble_up, percolate_up, shift_up, trickle_up, swim_up, cascade_up

(2.) heapify_down :-

Perform this operation when we delete a node to the binary heap.



Also known as:- down_heap, bubble_down, percolate_down, sift_down, extract min/max, sink_down.

CamScanner

2. Insertion :-

Inserting a new node to binary heap by maintaining its properties.

(i) Add the new node to first open spot available in the lower level.
(ii) Heapify the new node.

3. Deletion :-

Removing the node from binary heap by maintaining its properties.

(i) Swap the node you want to delete with the last node.
(ii) Delete the last node.
(iii) Heapify the last node which is now placed in the deleted node position.

CamScanner

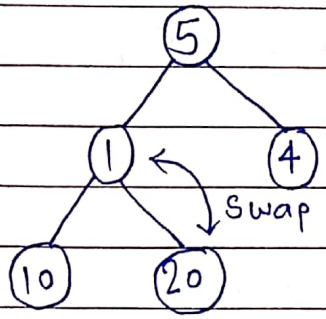
✯ Construction of Binary Heap:-

Steps:-

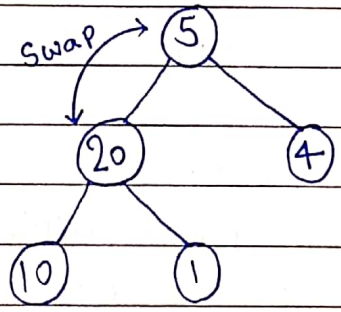
1. First create complete binary tree using given list of numbers.
2. Then start heapifying tree: Start from last internal node.

Numbers:- [5 1 4 10 20]

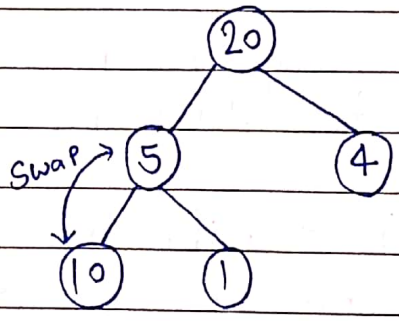
Step 1:-



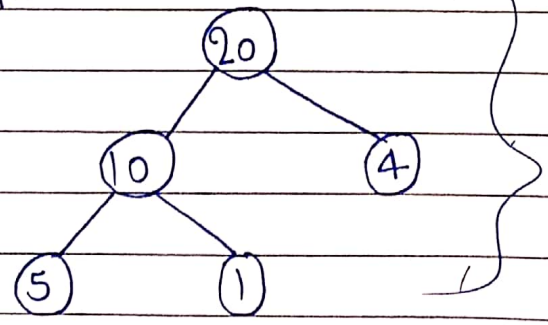
Step 2:-



Step 3:-



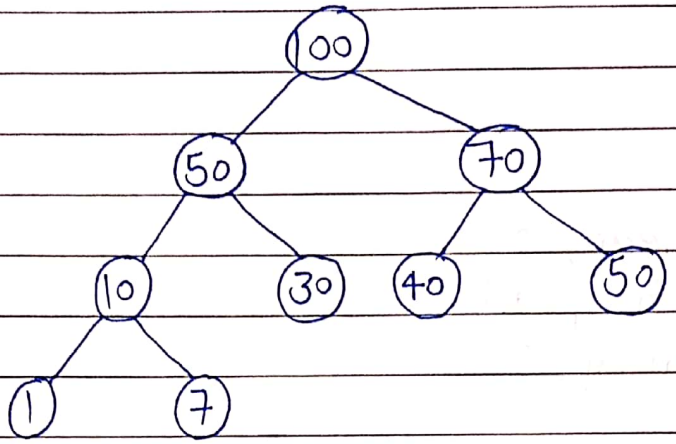
Step 4:-



Max-Heap

CamScanner

☆ Representation of Binary Heap :-



0	1	2	3	4	5	6	7	8
100	50	70	10	30	40	50	1	7

Root → [0]

i^{th} → list[i]

Parent → (i-1)//2

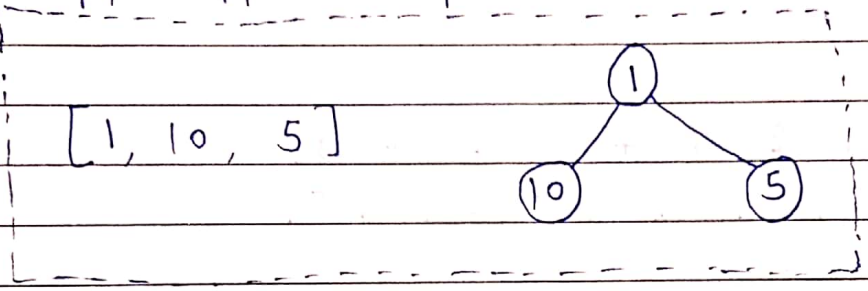
Left child → (2*i)+1

Right child → (2*i)+2

☆ Implementation :-

Program :-

```
import heapq
heap = []
heapq.heappush(heap, 10)
heapq.heappush(heap, 1)
heapq.heappush(heap, 5)
```




```
heapq.heappop(heap) # Remove smallest value = 1
heapq.heappop(heap) # 5 will be deleted.

list1 = [1, 3, 5, 2, 4, 6]
heapq.heapify(list1) # Perform heapify operation.
    # List1 will be = [1, 2, 5, 3, 4, 6]
heapq.heappushpop(list1, 89) # Perform push & pop operation
    # List1 will be = [2, 3, 5, 89, 4, 6]
heapq.heapreplace(list1, 100) # Perform pop & push operation
    # List1 will be = [3, 4, 5, 89, 100, 6]
heap = [1, 20, 5, 4, 3, 6, 2]
heapq.nsmallest(2, heap) # [1, 2]
heapq.nlargest(3, heap) # [20, 6, 5]
```

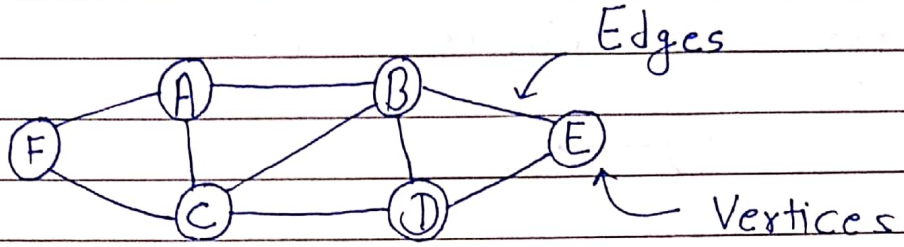
CamScanner

Graph

Magic
DATE
PAGE

☆ Graph :-

Graph is a non-linear data structure consisting of nodes and edges.



$G = (V, E)$

where,

G = Graph
V = Set of vertices
E = Set of Edges
( ) : Ordered pair
{ } : Unordered pair

Uses :-

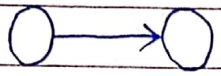
Google Maps And GPS
Facebook and LinkedIn
E-Commerce websites

CamScanner

✱ Types of Graph:-

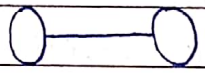
1. Directed Graph:-

A graph in which all the edges are unidirectional.



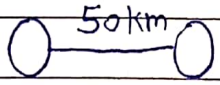
2. Undirected Graph:-

A graph in which all the edges are bi-directional.



3. Weighted Graph:-

A graph in which each edge is assigned with some weight/cost/value.



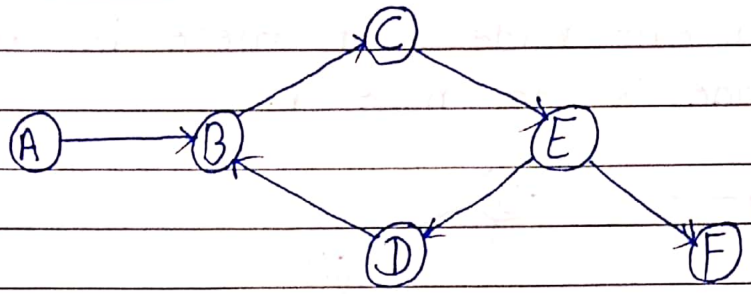
4. Unweighted Graph:-

A graph where there is no value or weight associated with the edge.

CamScanner

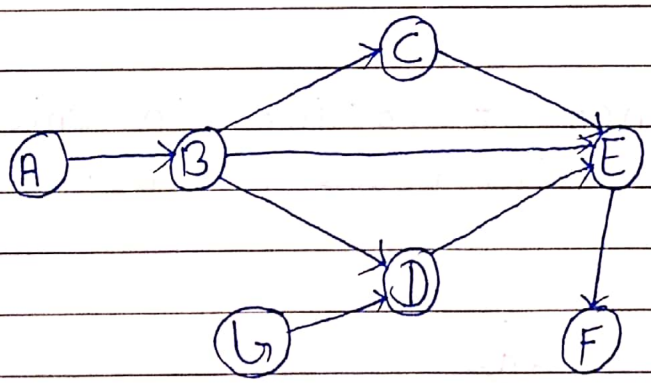
5. Cyclic Graph:-

A graph that contains at least one cycle is called a cyclic graph.



6. Acyclic Graph:-

A graph that does not contain any cycles is called an acyclic graph.

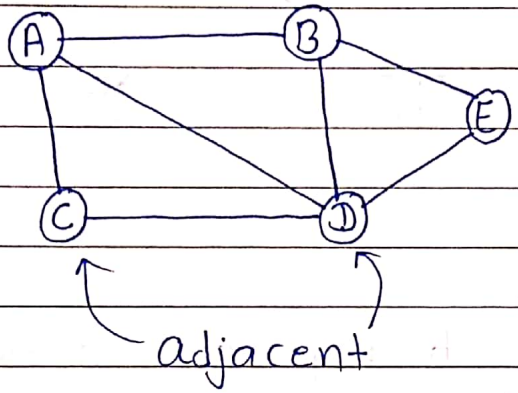


Magic
DATE
PAGE

★ Terminologies of Graph :-

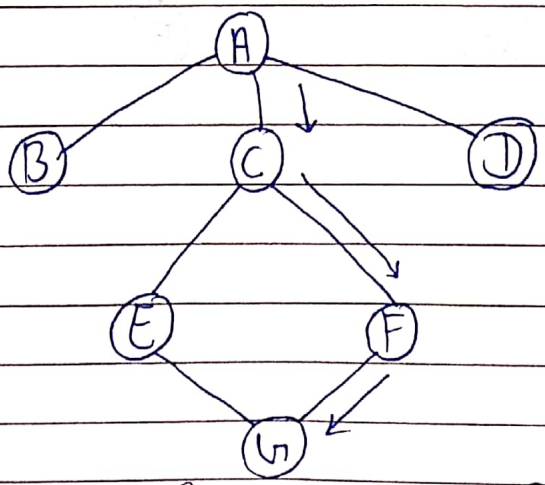
1. Adjacent Nodes / Neighbour Nodes :-

Node X is adjacent node Y if there is an edge from node X to node Y.



2. Path :-

Sequence of vertices in which each pair of successive nodes is connected by an edge.



path :- A → G { A → C, → F → G }

3. Cycle :-

Cycle is path in which first and last node need to be same and also all the other nodes need to be distinct.

4. Connected graph:-

A graph is said to be connected if there is a path from any node to any other node.

5. Degree :-

Degree of a node = number of edges Connected to it.

Indegree of a node = number of edges coming to that node.

Outdegree of a node = number of edges going outside from that node.

6. Complete Graph:-

A complete graph is a simple undirected graph where every pair of vertices is adjacent.

CamScanner

★ Graph Representation :-

1. Adjacency Matrix :-

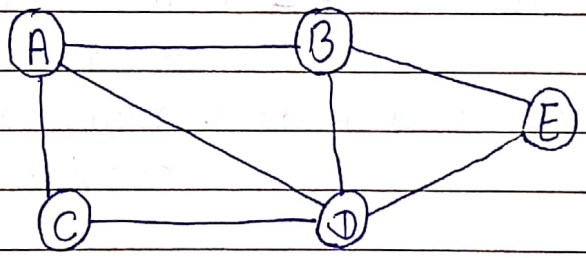
Represents the connection between the nodes in matrix form.

Steps :-

(i) Create K * K matrix.

(ii) Row and Column represents the nodes of the graph.

(iii) if edge is present then store 1 otherwise store 0.

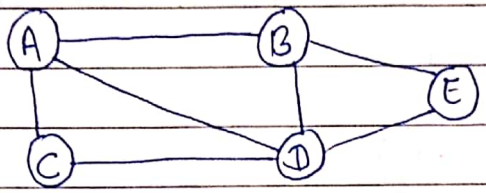


	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	0	1
E	0	1	0	1	0

CamScanner

2. Adjacency List :-

It consists of an array of linked lists, where each element in the array represents a vertex in the graph, and the linked list for each vertex contains the vertices adjacent to it.



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	0	1
E	0	1	0	1	0

A : [B, C, D]

B : [A, D, E]

C : [A, D]

D : [A, B, C, E]

E : [B, D]

CamScanner

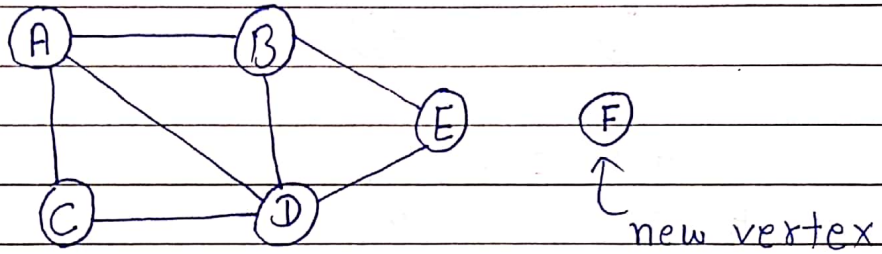
Magic
DATE
PAGE

✯ Graph Operations :-

→ Insertion
→ Deletion
→ Traversal

1. Insertion :-

(i) Add new node or vertex :-
add_node(F)



	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	1	1	0
C	1	0	0	1	0	0
D	1	1	1	0	1	0
E	0	1	0	1	0	0
F	0	0	0	0	0	0

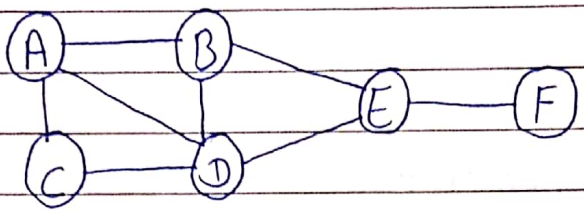
↑ new vertex

A : [B, C, D]
B : [A, D, E]
C : [A, D]
D : [A, B, C, E]
E : [B, D]
F : [ ]
↑ new vertex

CamScanner

(ii) Add edge :-

add_edge(E, F)



	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	1	1	0
C	1	0	0	1	0	0
D	1	1	1	0	1	0
E	0	1	0	1	0	1
F	0	0	0	0	1	0

add_edge(E, F)
graph[E][F] = 1
graph[F][E] = 1

A: [B, C, D]
B: [A, D, E]
C: [A, D]
D: [A, B, C, E]
E: [B, D, F]
F: [E]

* Insertion Functions :-

(1.) Function to add a node using adjacency matrix representation →

Program :-

```
def add_node(v):
    global node_count
    if v in nodes:
        print(v, "is already present in the graph")
    else:
        node_count = node_count + 1
        nodes.append(v)
        for n in graph:
            n.append(0)
        temp = []
        for i in range(node_count):
            temp.append(0)
        graph.append(temp)

def print_graph():
    for i in range(node_count):
        for j in range(node_count):
            print(graph[i][j], end = " ")
        print()
```

```
nodes = []
graph = []
node_count = 0
print("Before adding nodes")
print(nodes)
print(graph)
add_node("A")
add_node("B")
print("After adding nodes")
print("nodes:", nodes)
print(graph)
print_graph()
```

CamScanner

(2.) Function to add an edge using adjacency matrix representation →

Program:-

```
def add_edge(v1, v2):
    if v1 not in nodes:
        print(v1, "is not present in the graph")
    elif v2 not in nodes:
        print(v2, "is not present in the graph")
    else:
        index1 = nodes.index(v1)
        index2 = nodes.index(v2)
        graph[index1][index2] = 1
        graph[index2][index1] = 1
```

(3.) Function to add a node using adjacency List representation →

Program:-

```
def add_node(v):
    if v in graph:
        print(v, "is already present in graph")
    else:
        graph[v] = []

graph = { }
add_node("A")
add_node("B")
print(graph)
```

CamScanner

(4.) Function to add an edge using adjacency list representation →

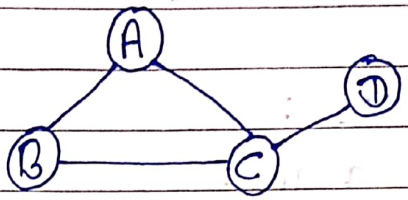
Program:-

```
def add_edge(v1, v2):
    if v1 not in graph:
        print(v1, "is not present in the graph")
    elif v2 not in graph:
        print(v2, "is not present in the graph")
    else:
        graph[v1].append(v2)
        graph[v2].append(v1)
```

DATE
PAGE

2. Deletion :-

(i) Delete node :- delete_node(c)



← Want to delete

	A	B	C	D
A	0	1	1	0
B	1	0	1	0
C	1	1	0	1
D	0	0	1	0

A : [B, C]
B : [A, C]
C : [A, B, D]
D : [C]

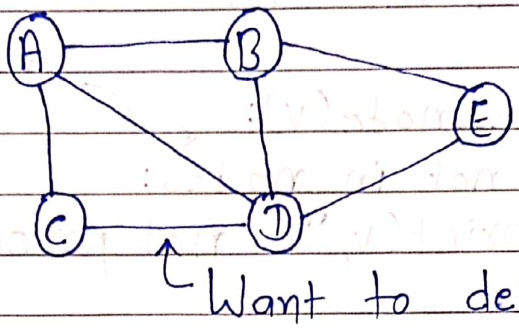
	A	B	D
A	0	1	0
B	1	0	0
D	0	0	0

A : [B]
B : [A]
D : [ ]

CamScanner

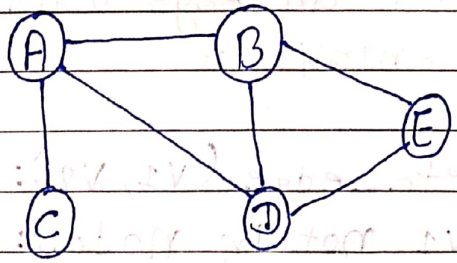
(ii) Delete edge :- delete_edge (V1, V2)

delete_edge (C, D)



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	0	1
E	0	1	0	1	0

A: [B, C, D]
B: [A, D, E]
C: [A, D]
D: [A, B, C, E]
E: [B, D]



	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	0	0
D	1	1	0	0	1
E	0	1	0	1	0

A: [B, C, D]
B: [A, D, E]
C: [A]
D: [A, B, E]
E: [B, D]

* Deletion Functions:-

(1.) Function to delete a node using adjacency matrix representation:-

Program:-

```
def delete_node(v):                    # global node_count
    if v not in nodes:
        print(v, "is not present in the graph")
    else:
        index1 = nodes.index(v)
        node_count = node_count-1
        nodes.remove(v)
        graph.pop(index1)
        for i in graph:
            i.pop(index1)
```

(2.) Function to delete an edge using adjacency matrix representation:-

Program:-

```
def delete_edge(v1, v2):
    if v1 not in nodes:
        print(v1, "is not present in graph")
    elif v2 not in nodes:
        print(v2, "is not present in graph")
    else:
        index1 = nodes.index(v1)
        index2 = nodes.index(v2)
        graph[index1][index2] = 0
        graph[index2][index1] = 0
```

CamScanner

(3.) Function to delete a node using adjacency list representation:-

Program:-

```
def delete_node(v):
    if v not in graph:
        print(v, "is not present in the graph")
    else:
        graph.pop(v)
        for i in graph:
            list1 = graph[i]
            if v in list1:
                list1.remove(v)
```

(4.) Function to delete an edge using adjacency list representation:-

Program:-

```
def delete_edge(v1, v2):
    if v1 not in graph:
        print(v1, "is not present in the graph")
    elif v2 not in graph:
        print(v2, "is not present in the graph")
    else:
        if v2 in graph[v1]:
            graph[v1].remove(v2)
            graph[v2].remove(v1)
```

CamScanner

3. Traversal :-

→ DFS
→ BFS

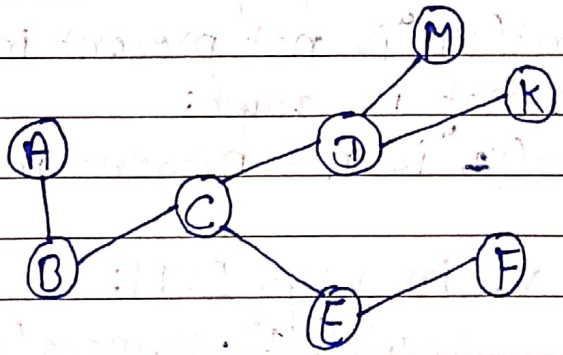
(1.) DFS :-

Depth First Search.

First visit the starting node.

Steps :-

1. Consider starting node as current node and visit that node.
2. Visit the unvisited adjacent node of current node, make that node as current node.
3. Follow step 2 until we reach dead end.
4. if unvisited nodes are present in the graph then back track, take recent visited node as current node repeat step 2.



A B C D K M F

Magic
DATE
PAGE

* Implementation Using Recursion:-

Program:-

```
def DFS(node, visited, graph):
    if node not in graph:
        print("Node is not present")
    if node not in visited:
        print(node)
        visited.add(node)
        for i in graph[node]:
            DFS(i, visited, graph)

visited = set()
graph = {}
add_node("A")
add_node("B")
add_node("C")
add_node("D")
add_node("E")
add_edge("A", "B")
add_edge("B", "E")
add_edge("A", "C")
add_edge("A", "D")
add_edge("B", "D")
add_edge("C", "D")
add_edge("E", "D")
print(graph)
DFS("A", visited, graph)
```

CamScanner

* Implementation DFS Using iterative approach:-

Program:-

```
def DFSiterative(node, graph):
    visited = set()
    if node not in graph:
        print("Node is not present")
        return
    stack=[]
    stack.append(node)
    while stack:
        current = stack.pop()
        if current not in visited:
            print(current)
            visited.add(current)
            for i in graph[current]:
                stack.append(i)

graph = {}

add_node("A")
add_node("B")
add_node("C")
add_node("D")
add_node("E")
```

```
add_edge("A", "B")
add_edge("B", "E")
add_edge("A", "C")
add_edge("A", "D")
add_edge("B", "D")
add_edge("C", "D")
add_edge("E", "D")
print(graph)
DFSiterative("A", graph)
```

CamScanner