Data structure in Python

Data structure is a way to **store and organise the data**, so that it can be accessed or effectively also used and maintained easily. Eg. In dictionary words organised alphabetically, in a library categorisation and order of books maintained. Choosing appropriate data structure is the most important task for programmer to understand when to use which data structure.

Data structure types:
a) Built-in: 1) list 2) tuple 3) set 4) dictionary
b) User defined: 1) stack 2) queue 3) linked list 4) tree 5) graph
Data structure Types Based on linearity:
Linear: Data items arranged linearly or sequentially & possible to traverse serially in single run. eg stack, queue, linked list
Non-linear: Data items not arranged linearly or sequentially. eg tree, graph.

a) Built-in
1) List: L1 = list() or L2 = [1,2,3]
Properties: List can store data of different data types & maintain order i.e. index. If any element removed, index will change but order maintained, element accessed using zero based indexing, negative indexing applicable, nested list possible, list are mutable i.e. can append delete modified element list are dynamic as grow by adding and shrink by deleting element.

2) Tuple: t1 = tuple() or t2 = (1,2,3) or 1,2,3 or t3 = (2,)
Properties: It is immutable i.e. append, delete, modify element not possible similar to list, elements can accessed by zero based indexing, negative indexing possible, can be nested.
Need: tuples faster than list, tuple guards against accidental modification.

3) Dictionary: d1 = {key1:value1, key2:value2,...}
Properties: Key must be unique & immutable ie can't have list, dictionary as a key but can be string, tuple as key. Dictionary values are mutable.
Eg. d1 = dict(), {}, d2 = {'A':'b','A':2,3:'a'} -> {'A':2,3:'a'}.
Dictionaries are ordered and values can be accessed or modified using keys.

Eg. del d2['A']->d2={3:'a'}, d1['1']=1->d1={'1':1}

4) Sets: s1=set(), set('Hello'), set([1,2,3]), s2={1,2,3,4}, s3={1,2,3,1,2,3}->s3={1,2,3}
Properties: Set contains unique (no duplicate), it is unordered, it is mutable ie can add, modify elements but it contains only immutable elements like tuples ie list can't be element of set & so can't be nested. Set element can't access using index instead can use for loop or in operator.

b) User defined data structure

1) Stack

Stack: ordered collection of items with addition & removal of elements always takes place from top end. Stack store item in last in first out (LIFO) or first in last out(FILO) manner. It's like plates placed in kitchen.

Operations in stack: most used push and pop
1) Push: suppose, from statement 'hello, I am ram and studying Python', if we remove Python, I, ram sequentially then it will push in stack shown in figure.

3	Ram
2	I
1	Python

2) Pop: in above example if we do undo operation by ctrl+z, ram remove and appeared first, next I, next python. This is done using operation call LIFO or FILO.
Application : can used to reverse a string, expression evaluation and expression conversion ie infix to prefix, postfix, forward and backward feature in web browsers, in algorithm like tower of Hanoi.
3) Peek or top: Give element present at top
4) isEmpty: tell stack is empty as not.

Implementation of stack in Python

A) **List:** for push operation in stack in adding element from top we can used append method & for pop ie removing element we used pop method from list

Eg.
```
s= []
s.append(1)
s.append(2)
s.append(3)
s # [1,2,3]
s.pop() # 3
s.pop() # 2
s.pop() # 1
s.pop() # error
```

To check list or stack empty or not len(s) ==0 -> True or not s -> True
To get last or top position element use s[-1] -> 3

Python Program for stock operations.

```
stack = []
def push():
      if len(stack) == n:
           print('list is full!')
      else:
           element = input('Enter the element:')
           stack.append(element)
           print(stack)

def pop_element():
      if not stack:
           print('stack is empty!")
      else:
            e = stack.pop()
            print('removed element:', e)
            print(stack)

n = int(input ( limit of stack: '))
while True:
     print('select operation 1.push 2.pop 3.quit!)
     choice = int(input())
     if choice == 1:
            push()
      elif choice == 2:
            pop_element()
      elif choice == 3:
            break
      else:
             print("Enter the correct operation !")
```

B) Modules:

a) deque class from collections module is double ended que so can add element on both sides but as stack work in LIFO order, so we use only one end to add & remove element. For push & pop use append & pop same as list.

Eg.

```
> import collections
stack = collections.deque()
```

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

check stack is empty or not by
not stack # true
b)LifoQueue class from queue module can use to create stack & for push & pop operation use put&get method. eg

```
> import queue
stack = queue.LifoQueue(2)
stack.put(10)
stack.put(20)
stack.put(30, timeout=1) # queue.Full
stack.get() # 20
stack.get() # 10
stack.get(timeout=1) # _queue.Empty
```

2) Queue

Queue: Queue is linear data structure where element inserted from one side of removed from other side. It is open at both end so insertion & removal done at different ends, end where element added called as back/rear/tail of queue & end where element removed called as head or front of queue. Queue follows FIFO or LILO methodology. It is similar to queue in real world ie line for buying ticket or grocery or something.

Queue operations -
1) Enqueue - Adding element
2) Dequeue - Removing element.
3) isFull - check queue is full or not or can set limit
4) isEmpty - check queue is empty or not
Also we can check element present at front & rear
Application: Queue used in uploading bunch of photo, printing multiple documents, call center calls.

Implement Queue in Python

A) **List:** In queue enqueue ie adding element done using append method & dequeue ie removing done by pop method with index (list.pop(0)).

Eg
```
queue = []
queue.append(1)
queue.append(2)
queue.append(3)
queue # [1, 2, 3]
queue.pop(0) # 10
queue.pop(0) # 20
queue.pop(0) # 30
queue.pop(0) # error
Reverse dir :
q = []
q.insert(0,10)
q.insert(0,20)
q # [20, 10]
q.pop() # 10
q.pop() # 20
Check empty :
not q
last element :
```

```
q[-1]
1st element :
q[0]
```

Program for queue operation is similar to stack program

B) Modules:

a) deque class from collections module is double ended que, which used to implement queue with FIFO methodology. Use appendleft to insert value & pop to remove value or append to insert value & popleft to remove value.

```
import collections
q1 = collections.deque()
q1.appendleft(10)
q1.appendleft(20)
q1 # deque([20,10])
q1.pop() # 10
q1.pop() # 20
q1.pop() # error
import collections
q2=collections.deque()
q2.append(30)
q2.append(40)
q2 # deque ([30,40])
q2.popleft() # 30
q2.popleft() # 40
q2.popleft() # error
```

To check queue is empty:
not q
Left element of queue:
q[0]
Right element of queue:
q[-1]

(b) Queue() class from queue module create queue data structure contains methods to check size, empty, full & Put (item, block=True, timeout) will put item to queue.

2	3	4	5	6

If block is true & timeout is none it will block until free slot is available & once queue is full it will block item until free slot is available. It won't give queue full message instead will wait until free slot available. If in Put() if block = false & timeout = 1 then after waiting 1min it will print massage. When we take maximum size of 3 & when enter 4th element instead of printing queue is full massage it will block, until get free slot & if we don't want that we can set block as false or can fix timeout or use method put_nowait which will not wait for free slot & print queue is full massage when q full.
To get element we can use get or get nowait method.

```
> import queue
q= queue.Queue ()
q.put(10)
q.put(30)
q #queue object
q.get() # 10
q.get() # 30
q.get() # waiting (as time out not set).
```

Priority Queue: It is modified version of queues in which each element is associated with priority & serves according to its priority & if priority is same served as per order. Eg In hospital critical condition patient treated first. Priority of element set by element value or tuple of element & priority. In case of element value first sort all elements of set king Lowest or highest values as priority.
1) Lowest element > high priority
Eg. for 1,5,3,7,2 -> 1,2,3,5,7 & In pop removed in sequence of 1,2,3,5,7.
2) Highest elements high priority eg removed as 7,5,3,2,1

Implementation in Python

A) List:

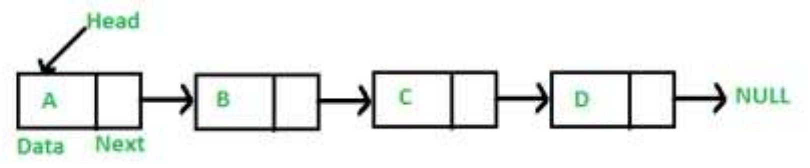
```
q = []
q.append(10)
q.append(40)
q.sort()
q.append(20)
q.sort()
q # [10, 20, 40]
q.pop(0) # 10
q.pop(0) # 20
q.pop(0) # 40
```

B) **Module:** PriorityQueue: Element sorted using binary heap queue module

```
>import queue
q=queue.PriorityQueue()
q.put(10)
q.put(60)
q.put(40)
q.put(40)
q.get() # 10
q.get() # 40
q.get() # 40
q.get() # 60
OR
q=[]
q.append((1, 'A'))
q.append((4, 'B'))
q.append((2, 'C'))
q.sort(reverse = True)
q # [(4, 'B'), (2, 'C'), (1, 'A') ]
q.pop(0) # (4, 'B')
q.pop(0) # (2, 'C')
q.pop(0) # (1, A')
```

3. Linked list

Linked list: Linear data structure with chain of nodes. Each node contains data field & link to next node or data field & previous node link & next node link, ie it contains Data field or info or value or object & link or reference or pointer.

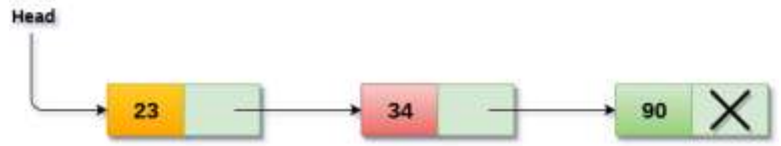


The node is elements of linked list & First link called head link or head pointer & last node called tail or end node with pointer to null. Also linked list is dynamic ie size is not fixed same as list & it is stored randomly in memory so don't require contiguous memory location ie consecutive memory block allocation Eg. In treshur hunt game clues are linked. 2) Relay running baton passed throw linked runners.

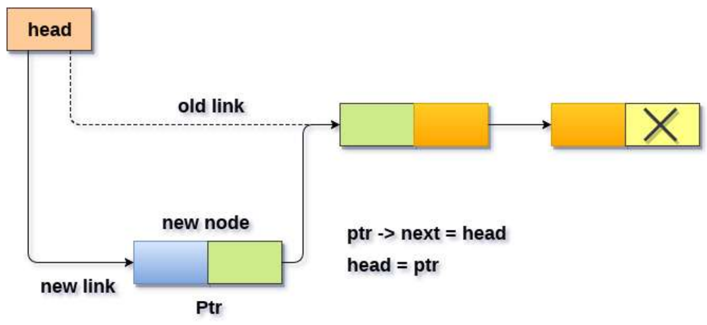
Advantages:
1) Dynamic data structure
2) Element insertion & deletion is easy ie no need to shift element
3) Using linked last we can implement stack, queue, graph
Applications: used to represent & manipulate polynomials, in web browser to in next or previous page, songs linked in music player, images linked in image viewer, etc.
Disadvantages:
1) Need extra memory to store link
2) Random element access not possible as linked.

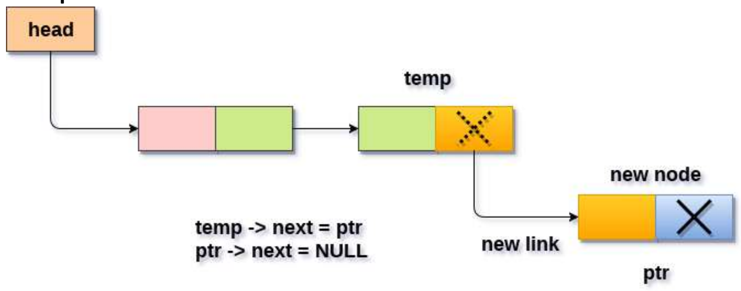
Types of linked list:
A) Singly linked list B) Doubly linked list C) Circular linked list

A) **Singly or simple linked list:** If each node contain only one link ie link of next node called singly linked list, most commonly used & allowed to go in forward direction only. Moving in back direction is not easier.

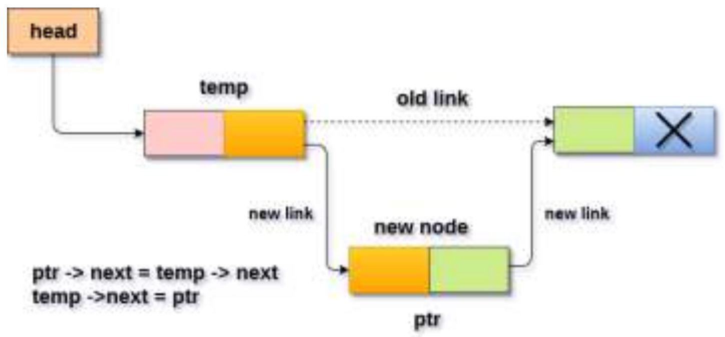


[Images Source : javatpoint.com]

Operations in Linked list:
1) add 2) delete 3) traversal
1) add / Insertion: Add element of node at
a) Begin: Update head pointer with address of new node & link or point new node to 1st node.
Note: Always update link or pointer of new node first to have at least one link.
Eg.


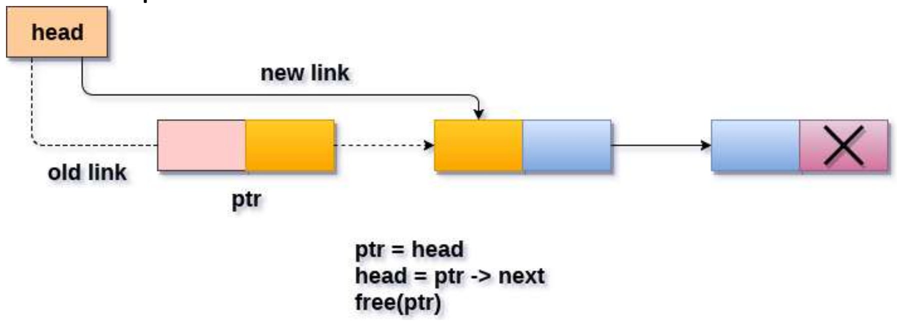
So, steps 1) Create new node 2) Link new node to first node 3) Update head pointer with new node address.
b) End: Steps
1) Create node with null pointer
2) Change last node pointer to new node


c) In Between:
Types: After node & Before node
Steps
1) Create node
2) 1st update link or pointer of new node with address or location of node just after required position
3) Update node before required position to address of new node.

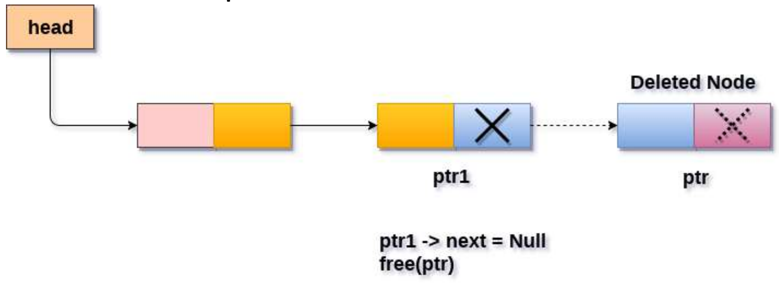


2) Deletion: Delete node or element at

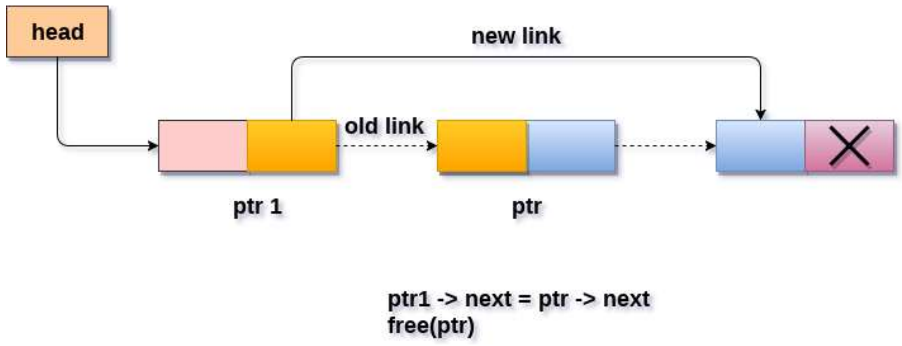
a) Begin: Point head pointer from 1st to 2nd node.



b) End: Change 2nd last node pointer to null.



c) Between: Change link of node before node to be deleted & connect it to next node.



3) Traversal: Going through every single node.

Python Programs :

Singly Linked list

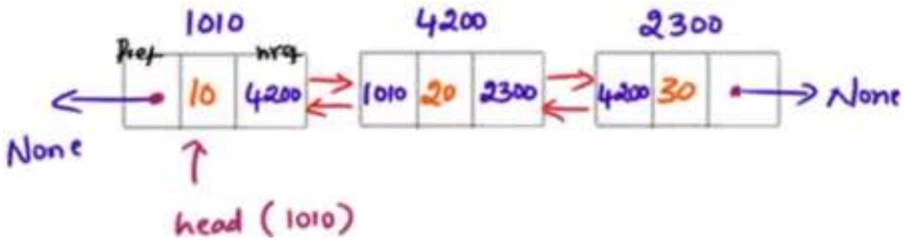
```
class Node:
   def __init__(self, data):
      self.data=data
      self.ref= None
# Create connections between nodes
class LinkedList:
   def __init__(self):
      self.head = None
# traversal of linked list
   def print_LL(self):
      if self. head is None:
         print("Linked list is empty!")
      else:
         n = self.head
         while n is not None:
            print (n.data,"--->",end=" ")
            n = n.ref

#1.Insertion/Adding Elements
   def add_begin(self,data):
      new_node = Node(data) # create node object from node class with data passed
      new_node.ref = self.head
      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.ref is not None:
            n = n.ref
         n.ref = new_node

   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 presesnt in LL")
        else:
            new_node = Node(data)
            new_node.ref = n.ref
            n.ref = new_node

    def add_before(self,data,x):
        if self.head is None:
            print("Linked List is empty!")
            return
        if self.head.data==x:
            new_node = Node(data)
            new_node.ref = self.head
            self.head = new_node
            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

    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!")
```

#2) Deletion / Removing an Element

```
    def delete_begin(self):
        if self.head is None:
            print("Linked List is empty can't delete!")
        else:
            self.head=self.head.ref

    def delete_end(self):
        if self.head is None:
            print("Linked List is empty can't delete!")
        else:
            n=self.head
            while n.ref.ref is not None:
                n = n.ref
            n.ref = None

    def delete_by_value(self,x):
        if self.head is None:
            print("can't delete LL 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
```

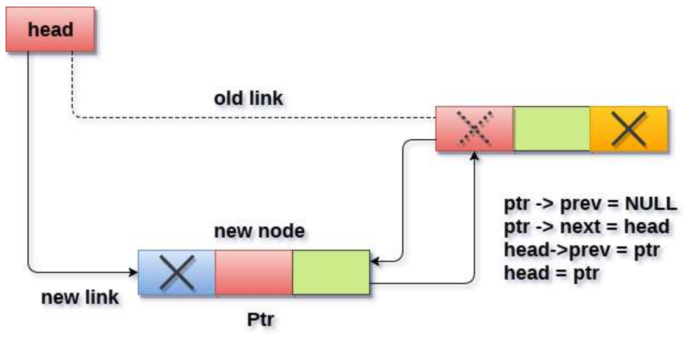
B) **Doubly linked list:** Each node contains data field & 2 links ie one is link of next node & another is link of previous node.



Operations in doubly linked list:
1) Insertion - a) begin b) end c) in between
2) Deletion - a) begin b) end c) by value
3) Traversal - Moving forward & backward is easier as node contain reference to both next & previous node.
Disadvantage -Need extra memory to store 2 links as compared to singly linked list.

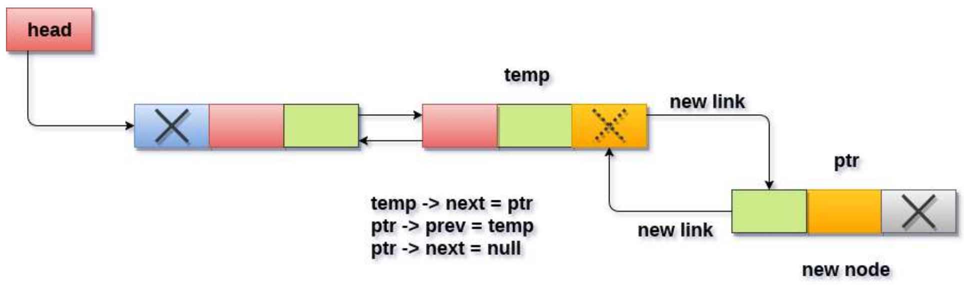
1) Insertion:

a) Begin:



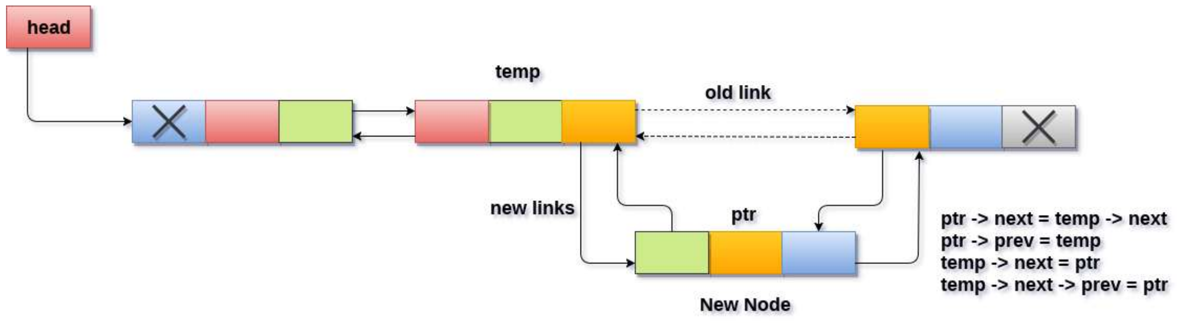
1) Create new node with 2 None link.
2) Store address of 1st node to n.ref link from head
3) store address new_node to first node p.ref 4) Point head to new node

b) End:



1) Create new node with 2 None links.
2) Change last node reference from None to new node
3) Change new node p.ref to last node.

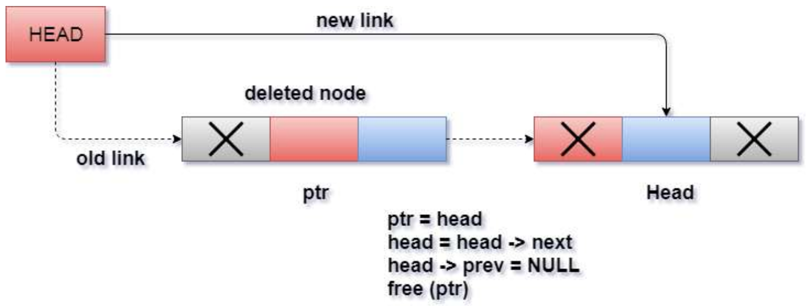
c) In between:



1) Create new node -
2) Store previous node address to pref link of new node & in previous node change nref to new node address.
3) Store next node address to nref link of new node & in next node change pref to new node address.

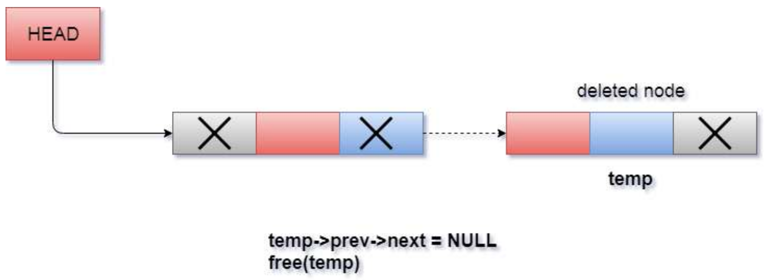
2) Deletion:

a) Begin:



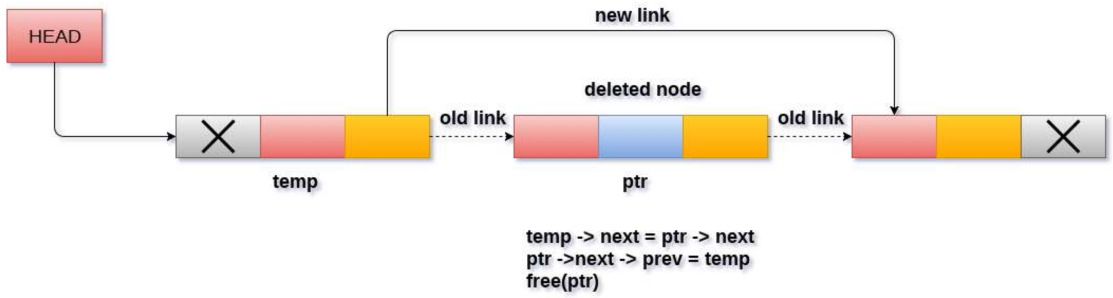
1) Point head to 2nd node.
2) Change previous link of node to None.

b) End:



1) Change nref link of 2nd last node to None.

c) By value:



1) Change previous node ref, nref to pref of next to next node.

2) Change next node ref pref to previous node reference nref.

Python Programs:

Doubly Linked list

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

# Create connections between nodes
class doublyLL:
        def __init__(self):
                self.head = None
        # traversal of Doubly LL
Forward Traversing:
   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

Backward Traversing:
   def print_LL_reverse(self):
      print()
      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
```

1. Insertion/Adding Elements

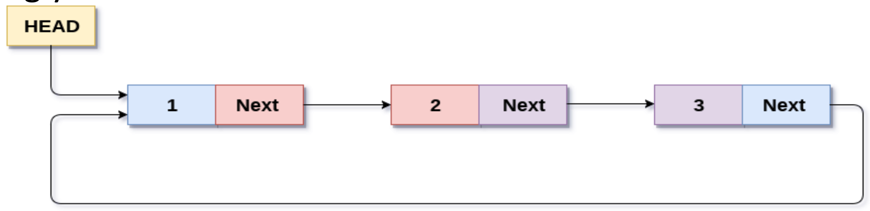
```
def insert_empty(self,data):
    if self.head is None:
        new_node = Node(data)
        self.head = new_node
    else:
        print("Linked List is not empty!")

 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):
    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 Linked List!")
    elif n.nref is None:
        new_node = Node(data)
        n.nref = new_node
        new_node.pref = n
    else:
        new_node = Node(data)
        n.nref.pref = new_node
        new_node.nref = n.nref
        n.nref = new_node
        new_node.pref = n

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

2) Deletion / Removing an Element

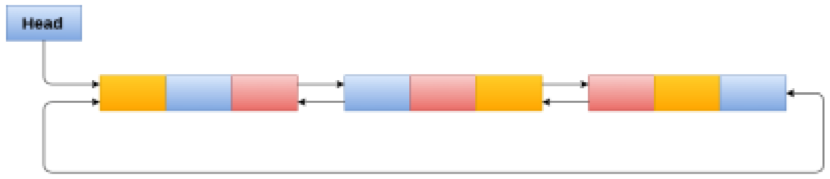
```
def delete_begin(self):
        if self.head is None:
            print("DLL is empty can't delte !")
            return
        if self.head.nref is None:
            self.head = None
            print("DLL is empty after deleting the node!")
        else:
```

```
        self.head = self.head.nref
        self.head.pref = None

    def delete_end(self):
        if self.head is None:
            print("DLL is empty can't delte !")
            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 can't delte !")
            return
        if self.head.nref is None:
            if x==self.head.data:
                self.head = None
            else:
                print("x is not present in DLL")
            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 in dll!")
```

C) Circular Linked list (CLL): linked list connected in such way that it forms a circle.
Types:
a) Circular singly linked list - Last node contains link of 1st node

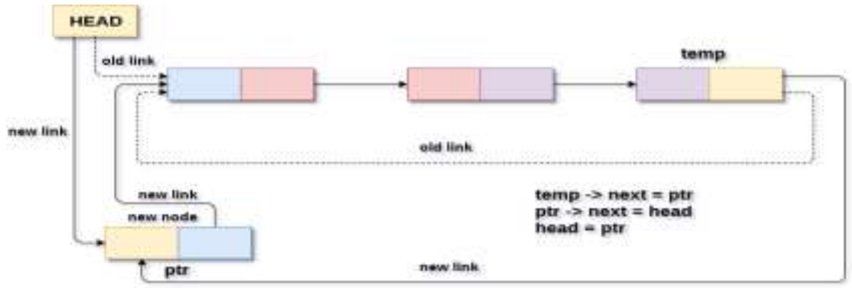


b) Circular doubly linked list - Last node nref connected to 1st node also in pref in 1st node contain reference of last node.



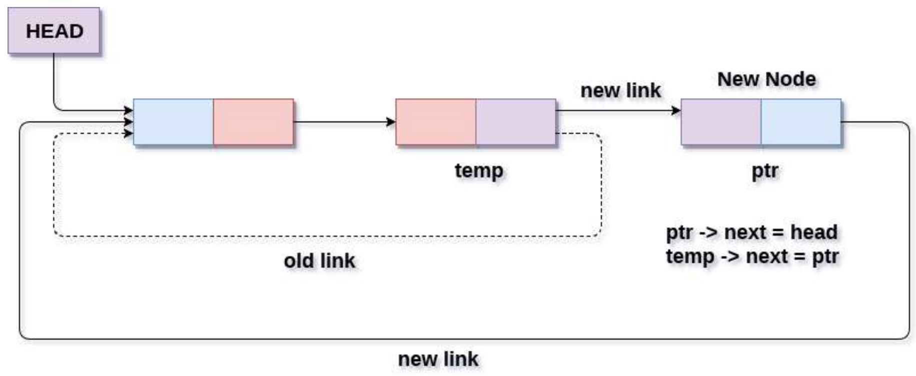
Here, we can start from any node & traverse through entire list but if proper caution not taken then it may leads to an infinite loop because we may not identify stopping condition as it doesn't contain None value

Circular Linked list Operations
1) Insertion:
a) Begin:



1) Create new node
2) Change link of new node to previous first node
3) Point head to new node
4) Change last node link to new node.

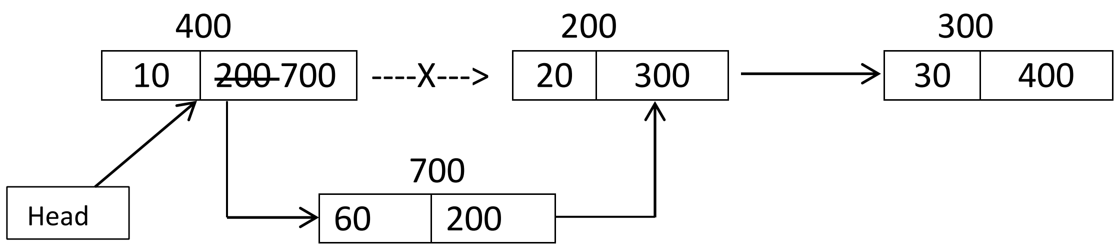
b) End:



```
ptr -> next = head
temp -> next = ptr
```

1) Create new node
2) Store new node ref in last mode of linked list
3) Store 1st node ref in new node.

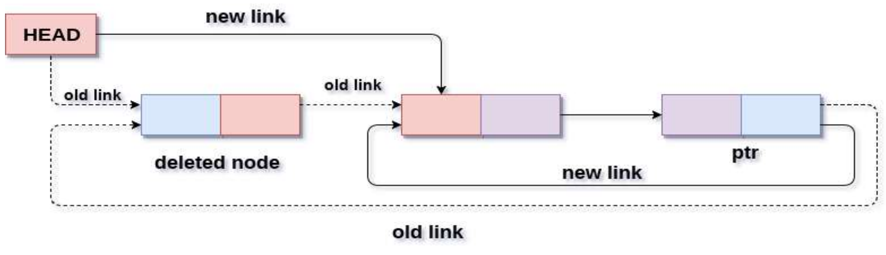
c) In between:



1) Create new node
2) Store reference of next node to new node
3) Change previous node reference to new node.

2) Delete:

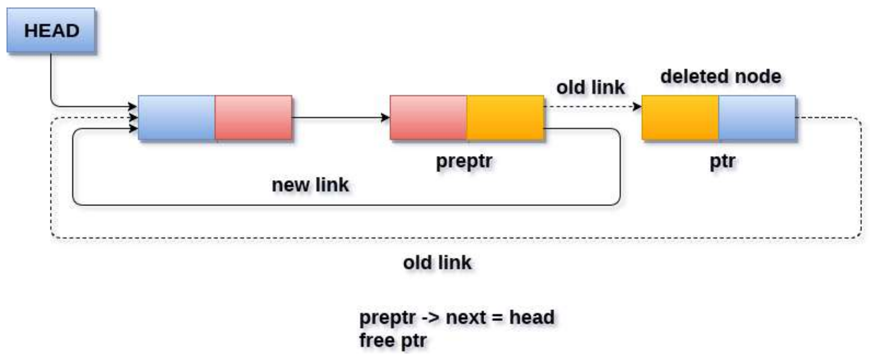
a) Begin:



```
ptr -> next = head -> next
free head
head = ptr -> next
```

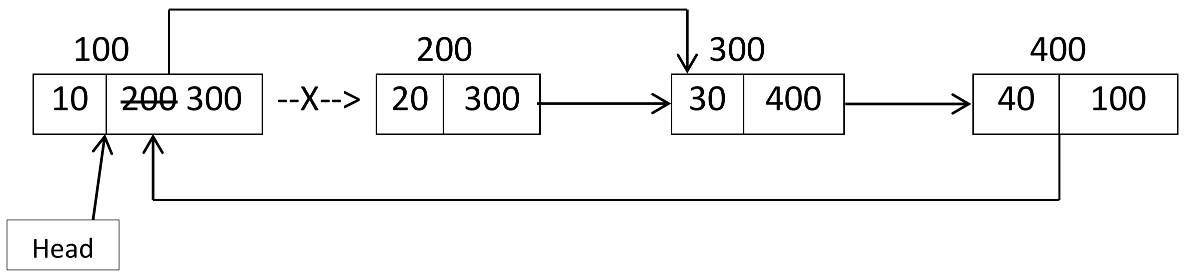
1) Point head to 2nd node
2) Change last node reference to 2nd node.

b) End:



1) Change last but one node reference to 1st node.

c) In between:

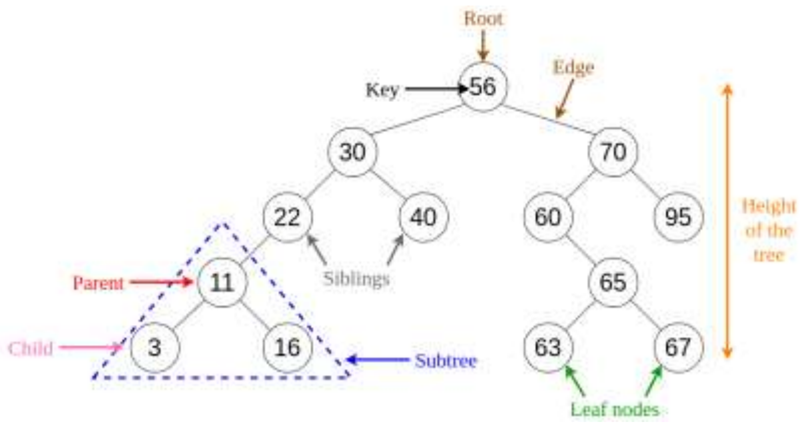


1) Change previous node reference to next node.

3) Traversal: In CLL for stopping condition, we check for head reference ie 1st node.
In Singly or doubly linked list last node point to None or null, but in CLL start from next node of last node ie 1st node then print until reach node which contains reference to 1st node ie last node.

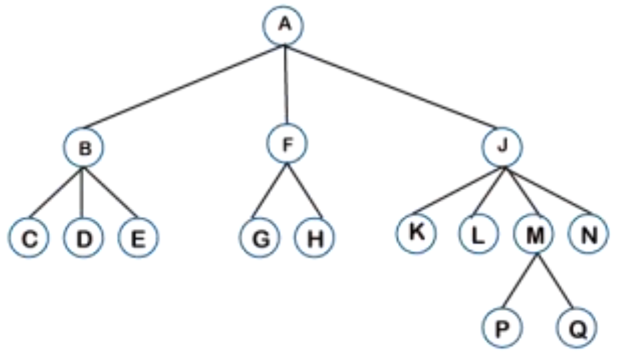
4) Tree

Tree:
A tree data structure is defined as a collection of objects or entities known as nodes that are linked together to represent or simulate hierarchy. Relationship between different nodes is non-linear with hierarchy. eg family, company positions.



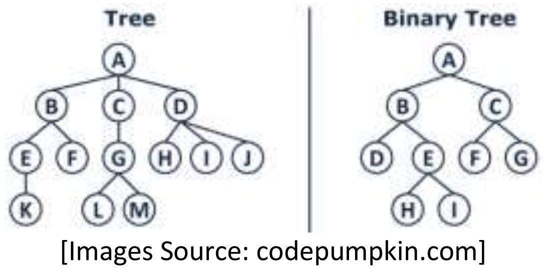
Terminologies in Tree:
Edge: Trees are collection of nodes connected by edges or legs. Every node contains data or value also contains link to other node.
Root: Topmost node is origin of tree called root or root node. Tree must contain only one root node.
Parent /Parent node: Node having branch to other node or node having child or children node.
Child / Child node: Node with link from parent. Except root node every node is child node.
Siblings: nodes belong to same parent.
Terminal / External / Leaf node: node not having child.
Non-terminal / Internal node: node having at least one child.
Path: Sequence of nodes & edges from one to another node.

Characteristics of tree:
1) Root node: Every tree has only one top most node called root node where tree originates.
2) If tree contains N nodes then number of edges / links always N-1 because root node don't have link.
3) Every child will have only one parent but parent can have multiple child.
4) Tree is a recursive data structure ie contains subtrees in them.



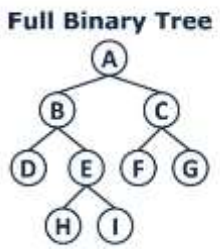
Degree of node: It is total number of children's of that node. D(B) = 3, D(F) = 2
Degree of tree: It is highest degree of node among all the nodes in tree. Eg. A - 3, B - 3, F – 2, J – 4, M - 2
So, Degree of tree = 4
Level of tree: Each step from top to bottom called level of tree, it start from 0 & increment by 1.
Eg Level 0 – A ; level 1- B,F,J ; level 2- C,D,E,G,H,K,L,M,N ; level 4 – P,Q
Height of node: Total no of edges that lies on longest path from any leaf node to particular node. A - 3, B -1, F – 1, J - 2
Height of tree: Height of root node A-3
Depth of node: Total number edges from root note to particular node, eg C-2, B - 1, P - 3
So, height of root node always zero & depth of root node always zero.
Depth of tree: No of edges from root node to leaf node in longest path. D = 3 (edges=3)

Types of Tree data structure
1) General trees: Each node can have any number of child nodes ie no limitation imposed on how many child mode each node can have.
2) Binary trees: Every node can have at most two children ie 0, 1, 2 child nodes.

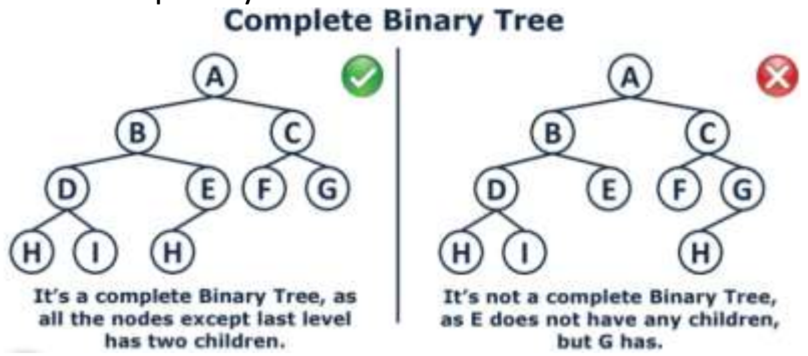


[Images Source: codepumpkin.com]

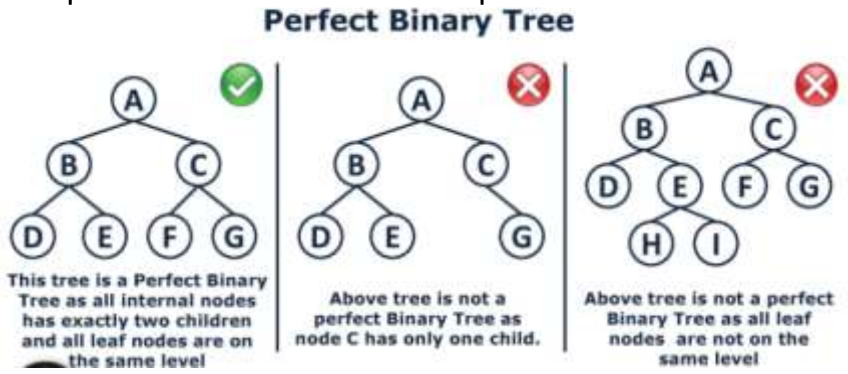
Types of binary tree:
1) Full binary tree: every node can have 0 or 2 child nodes other than leaf node ie 0, 2 child nodes.



2) Complete Binary tree: Except last level all levels completely filled & for last level either it should completely filled or nodes need to be filled from left to right.


Complete Binary Tree
It's a complete Binary Tree, as all the nodes except last level has two children.
It's not a complete Binary Tree, as E does not have any children, but G has.

3) Perfect binary tree: Binary tree in which all internal nodes contain 2 children & all leaf nodes are present at same level or depth.


Perfect Binary Tree
This tree is a Perfect Binary Tree as all internal nodes has exactly two children and all leaf nodes are on the same level
Above tree is not a perfect Binary Tree as node C has only one child.
Above tree is not a perfect Binary Tree as all leaf nodes are not on the same level

If in tree, k – Number of levels,

Number of nodes (n) = $2^k - 1$

Eg for 3 levels tree k=3, Number of nodes, n= $2^3 - 1 = 8 - 1 = 7$

If n is number of nodes in tree, levels(k) = $\log_2(n+1)$

Eg for 15 number of nodes n = 63,

Number of levels, k = $\log_2(63+1) = \log_2(64) = \log_2(2^6) = 6*\log_2 2 = 46$

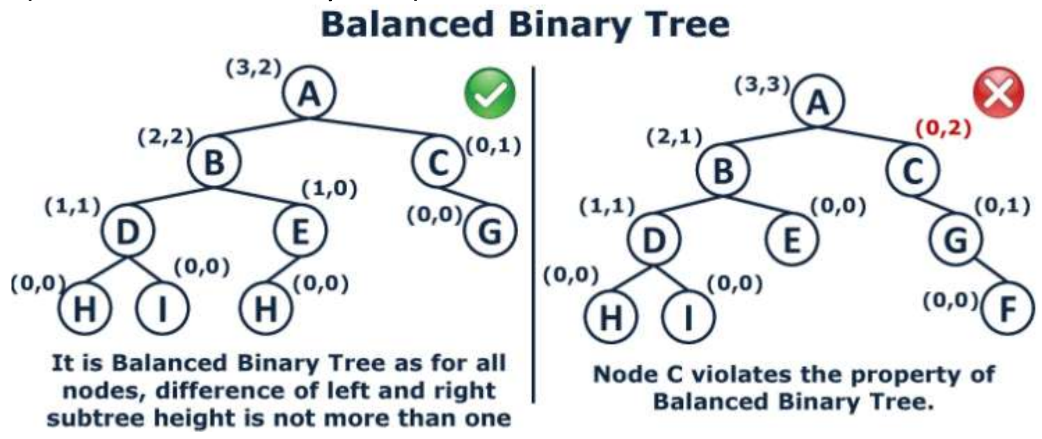
4) Balanced binary tree: Binary tree in which height of left & right subtrees of every node may differ at most 1.

For Every node,

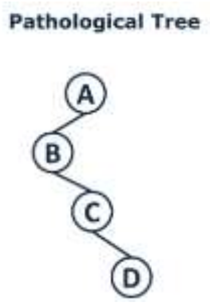
diff=| height of left subtree - height of right subtree|

diff <=1 (Balanced binary tree)

diff >1   (Not Balanced binary tree)


Balanced Binary Tree
(3,2) A
(2,2) B
(0,1) C
(1,1) D
(1,0) E
(0,0) G
(0,0) H
(0,0) I
(0,0) H
It is Balanced Binary Tree as for all nodes, difference of left and right subtree height is not more than one
(3,3) A
(2,1) B
(0,2) C
(1,1) D
(0,0) E
(0,1) G
(0,0) H
(0,0) I
(0,0) F
Node C violates the property of Balanced Binary Tree.

5) Pathological / Degenerate Binary tree: Every parent node has only one child node.



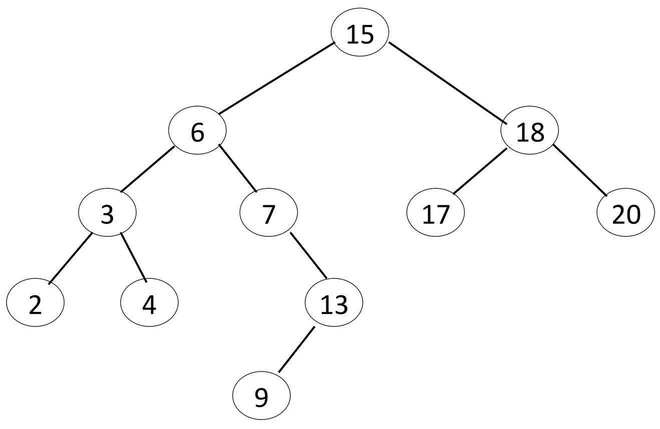
Binary Search Tree (BST)/ Ordered / Sorted binary tree:

Binary tree with following properties:

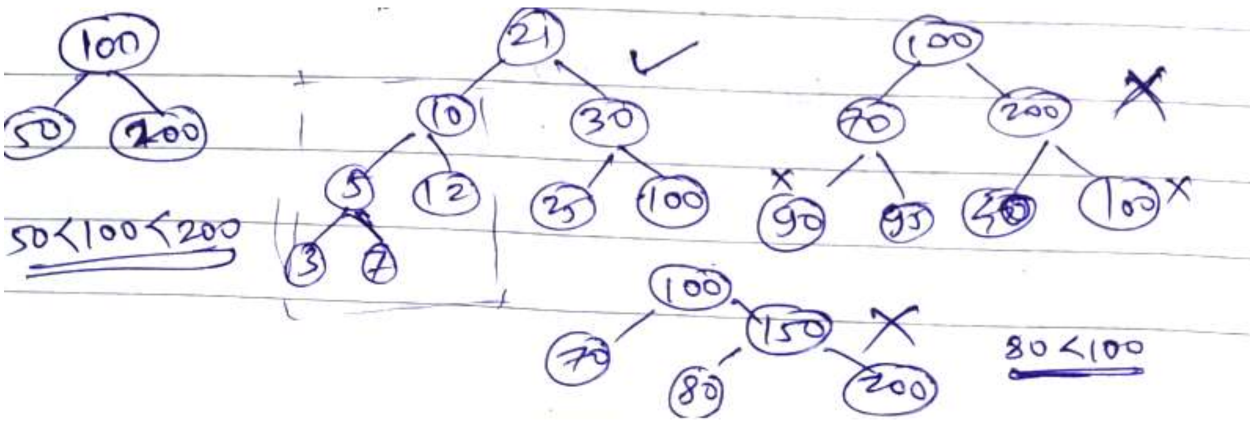
1) The left subtree of a node contains only nodes with keys lesser than node's key.

2) The right subtree of a node contains only nodes with keys greater than node's key.

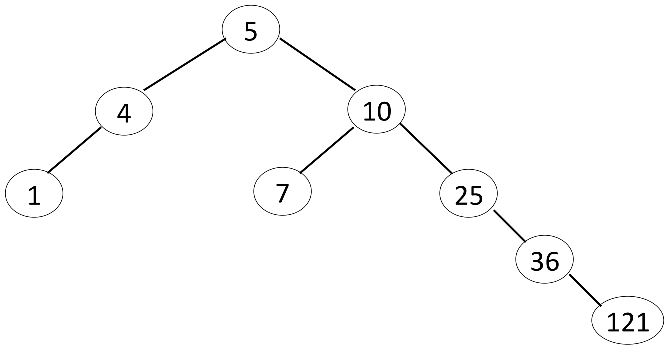
3) The left subtree & right subtree each must also be a binary search tree.



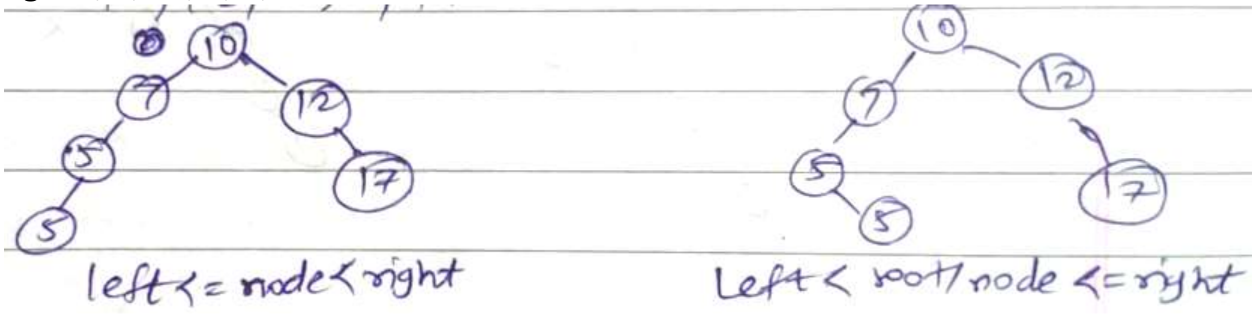
[Ref : Book- Into to Data Structure by cormen]



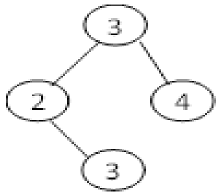
Building Binary search tree:
5, 4, 10, 7, 25, 36, 1, 121



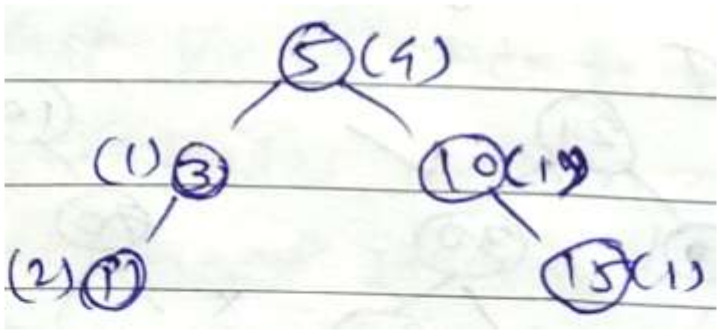
Binary search tree with Duplicate values:
Whether to permit duplicate keys depends on application
1) left < root/node < right: Duplicate values not allowed
2) left <= root/node < right
3) left < root/node <= right
Eg. 10, 7, 12, 5, 5, 17



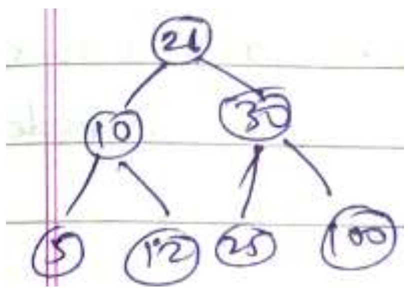
Duplicate value can add complexity eg



So, checking for duplicate element existence is not easy.
To avoid this we can follow another approach. That is instead of representing duplicate as separate node we can place counter that will keep number of occurrences of that key.
eg 5, 10, 15, 3, 1, 5, 5, 1



Operations of Binary search tree
1) Search: Used to find whether given node present in BST or not.
Steps 1) check BST is empty: if
Yes : print massage
No : compare root key with given value.
2) root == given value:
Yes: then print key found
No: checks where to search for key
3) given value < root key:
Yes : serach left subtree
No : search in right subtree

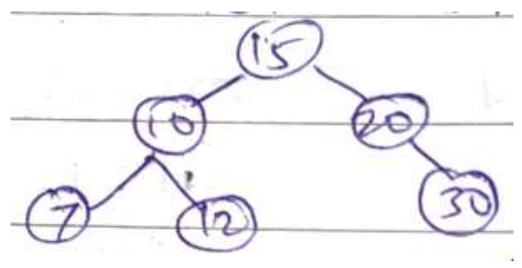
	Given : 12 12==21 : False 12<21 : True – Go Left 12<10 : False - Go Right 12 == 12 : True - Found	Given : 11 11==21 : False 11<21 : True – Go Left 11<10 : False – Go Right 11 == 12 : False 11 < 12 : True – Go Left None – Value Not Present

2) Insertion: Need to add new node with given value at correct position in BST following BST Rules.
Steps 1) Check BST is empty : if
Yes : insert new node
No : compare root key with given value.
2) root key < given value:
Yes : then go to right subtree, find correct position of new node & insert new node
No : then go to left subtree find & insert new node to correct position

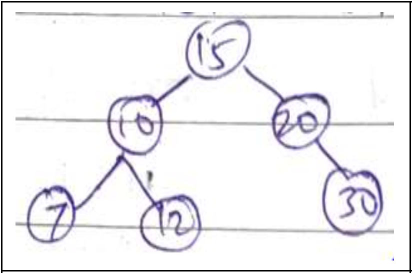
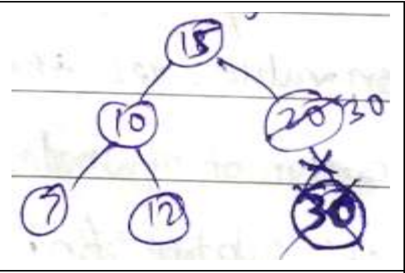
3) Deletion: Delete given node if it is present in tree. While performing deletion operation at most care should be taken that the properties of binary search tree are not violated & nodes not lost in process.
Steps
1) Check BST is empty :
Yes : Print Can't delete massage
No : Search mentioned key
2) Is node available:
Yes : Delete node
No : Print node not present in tree

Situation or scenarios while deleting node:
1) node with 0 child node ie leaf node
2) node with 1 child node
3) node with 2 child node

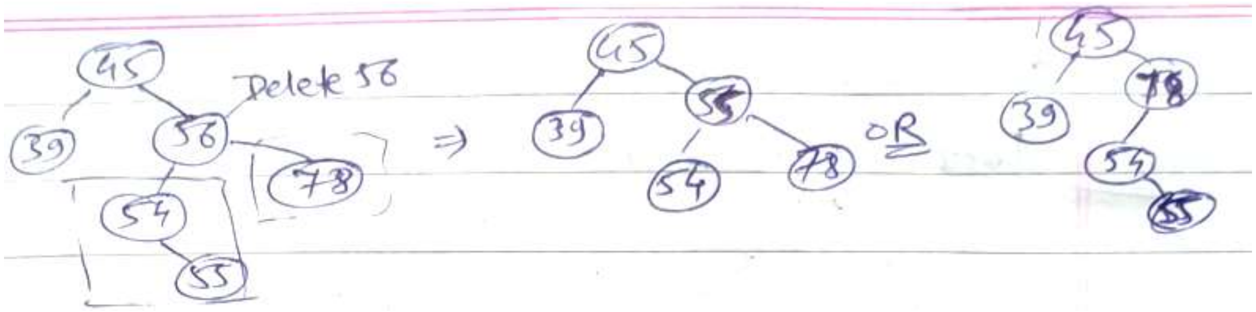
Case 1) Node with 0 or no child: Compare given node with tree node sequentially to go left or right side of tree & subtree & if found given node in tree delete that node.

	Given : 30 30==15 : False 30<15 : False - Go Right 30==20 : False 30<20 : False - Go Right 30 == 30 : True – Delete Node

Case 2) Node with 1 child node: Replace parent node with its child and if node is left child of it's parent node's child become left child of the node's parent and similar for right side also.

		Given : 20 20==15 : False 20<15 : False - Go Right 20==20 : True – Delete Node & replace by child node 30
Before	**After**	

Case 3) Node with 2 child node: Replace nodes value with its largest value in left subtree or with smallest value in right subtree.



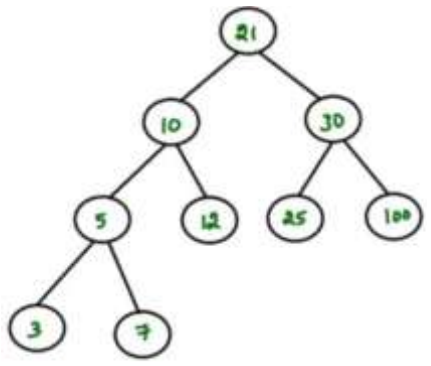
4) Traversal Operation: Different way or algorithms for tree traversal & they differ in order of node visited.
Algorithms:
1) Pre-order Traversal: To traverse a non-empty BST in preorder following operations performed recursively at each node.
1) First visit root node.
2) Traverse left subtree & finally
3) Traverse right subtree
So, Pattern is
Root -> Left subtree -> Right subtree

2) In-order Traversal:
1) First traverse left subtree
2) Visit root node & finally
3) Traverse right subtree
So, Pattern is
Left subtree -> Root -> Right subtree

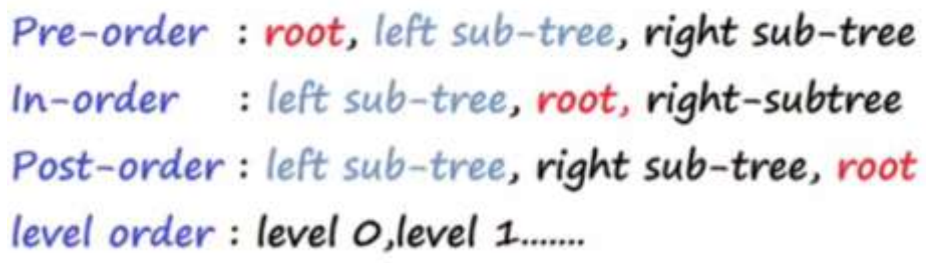
3) Post-order traversal:

1) First traversing left subtree
2) Traverse right subtree & finally
3) Visit root node
So, Pattern is
Left subtree -> Right subtree -> Root

(4) Level order Traversal: Visit node levelwise or in level order
So, Pattern is
Level0, level 1, level 2, ...

	**Pre-order** 21, 10, 5, 3, 7, 12, 30, 25, 100 **In-order** - (Values in descending order) 3, 5, 7, 10, 12, 24, 25, 30, 100 **Post-order** 3, 7, 5, 12, 10, 25, 100, 30, 21 **Level order** 21, 10, 50, 5, 12, 25, 100, 3, 7

Generally, as per BST Rule
Left subtree <= Root < Right subtree
Pre-order - Take Root BEFORE
In-order - SAME Order
Post-order - Take Root AFTER


Pre-order : root, left sub-tree, right sub-tree
In-order : left sub-tree, root, right-subtree
Post-order : left sub-tree, right sub-tree, root
level order : level 0,level 1.......

Minimum & maximum value in BST:
1) Smallest node: BST rule is left subtree <= root/node < right subtree
So, smallest value lies in left side or if left subtree of root node is null or none then value of root node will be smallest as compared to value in right subtree.
Smallest node is leftmost node in left subtree.
(2) Largest or greatest value in BST: It is rightmost node in right subtree.

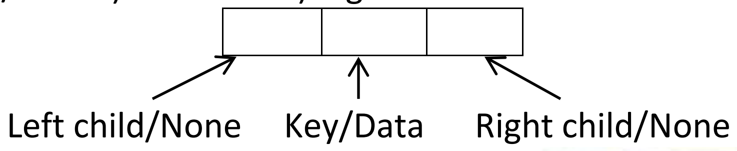
Total number of nodes in binary tree BST = number of nodes in LST + number of nodes in RST + 1

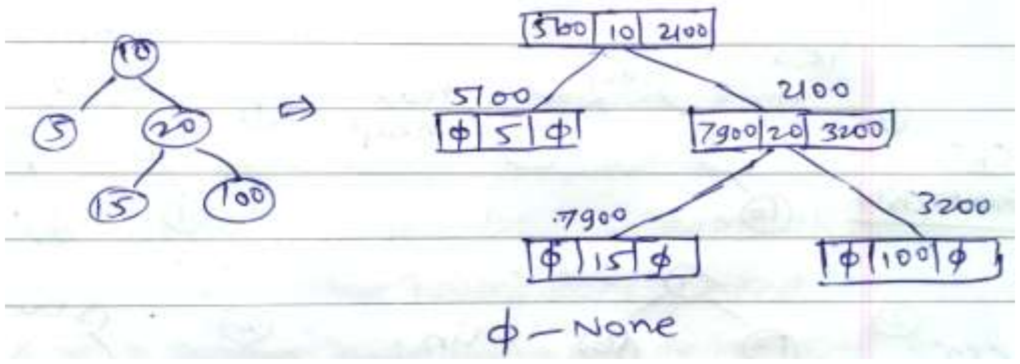
n(BST) = n(LST) + n(RST) + 1

Implementation of BST in Python

Every mode of BST contains

1) key/data 2) Left child 3) Right child.





Binary heap data structure:

Heap is specialized tree based data structure.

Types - binary heap, binomial heap, Fibonacci heap.

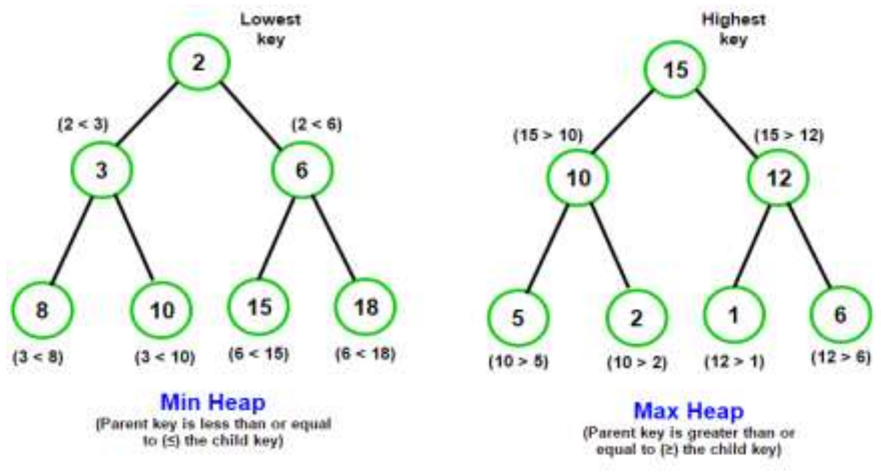
Binary heap: It is complete binary tree which satisfies the heap property.

Complete or almost complete binary tree: All levels except last level is completely filled with nodes & last level can completely filled or nodes are filled from left to right.

Heap property: property of node in which key of every parent node need to be lesser than or equal to OR greater than or equal to child node's key.

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.



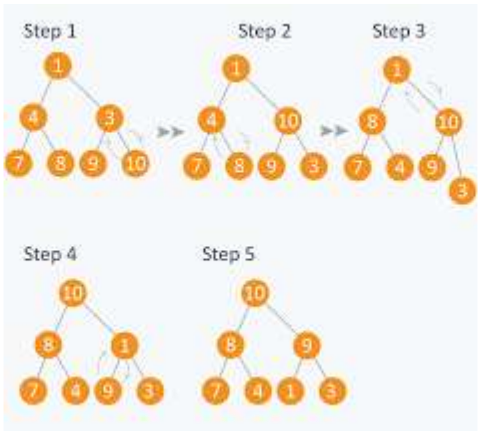
Applications:
1) To implement priority queue.
2) In heap sort algorithm.
3) To find kth largest or smallest element in list of numbers

Binary heap operations:
1) Heapify: The process of rearranging elements of the heap in order to maintain heap property which make sure that every node of tree follows heap property. It is used to create binary heap from complete binary tree and also used in insertion operation, deletion operation, while creating a binary heap from given array.

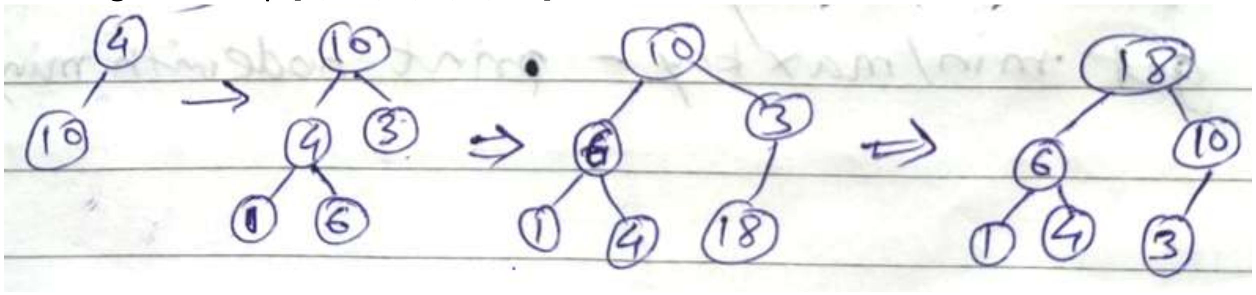
Types of heapify operation:
A) Heapify_Up: It follow bottom up approach & here we compare child node with its parent & check whether we following heap property. If not then we arrange the nodes then move upward & compare node with its parent of repeat process. It is used in insertion operation.
Operation also called with other names, up_heap, bubble_up, percolate_up, sift_up, trickle-up, swim_up, cascade_up.

B) heapify down: It follow top down approach & nodes not following heap property need to rearrange nodes. Compare node from top to its child then downward, hence called heapify down. Used in deletion operation. Other names are down_heap, bubble_down, percolate_down, sift_down, extract_min/extract_max, sink_down

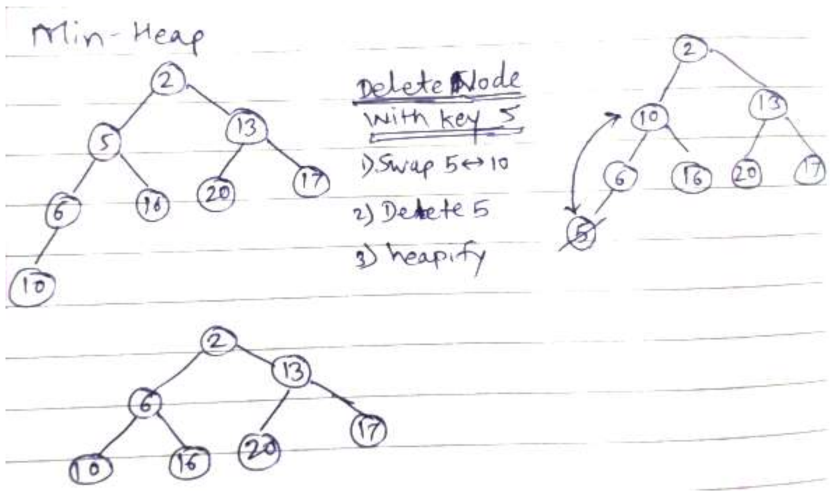
2) Insertion: inserting new node by maintaining properties ie binary tree structure & heap order property
Steps:
1) add new node to first open spot available in lower level
2) heapify new node
Eg adding 10 in maxheap



Here node with key 10 added to open spot at last level which follow complete binary tree but not following minimum heap property, as in maxheap parent node should be maximum, so rearrange or swap 10 with 3 & continue process.
Building max heap [4, 10, 3, 1, 6, 18]

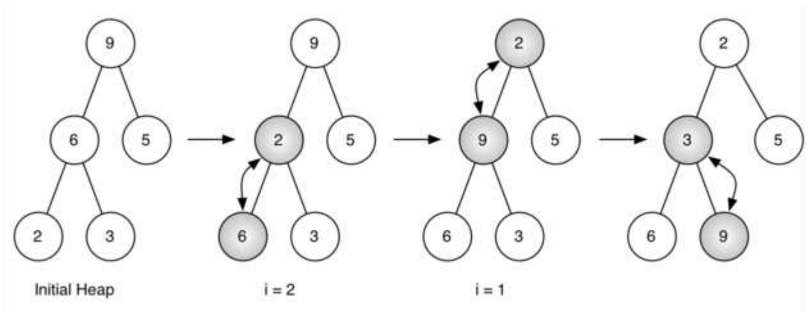


3) Deletion: Removing node from binary heap by maintaining its properties.
Steps
1) Swap node you want to delete with last node
2) delete last node.
3) heapify last node which now placed in deleted node position.

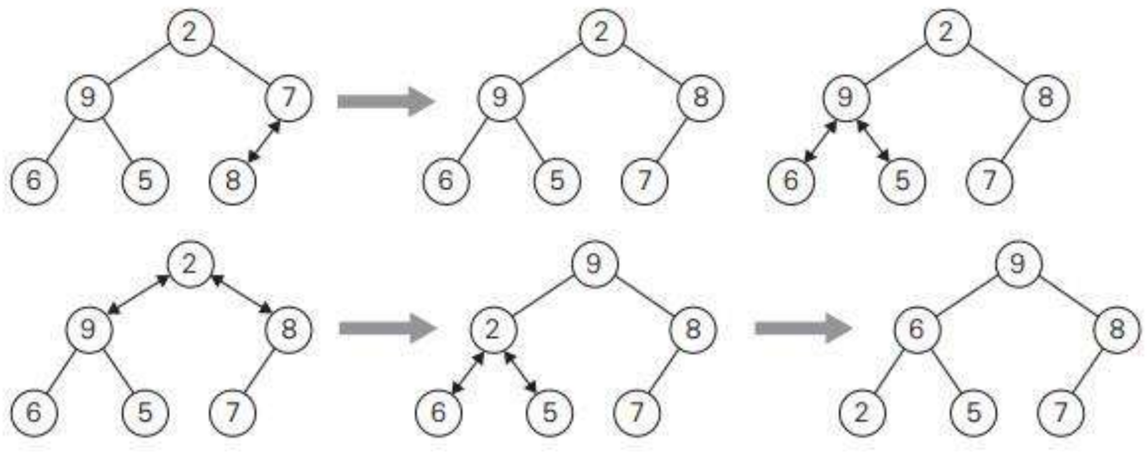


4) Extract min/max key: It is printing & deleting root node key
5) Get min/max key: print node with min/max key

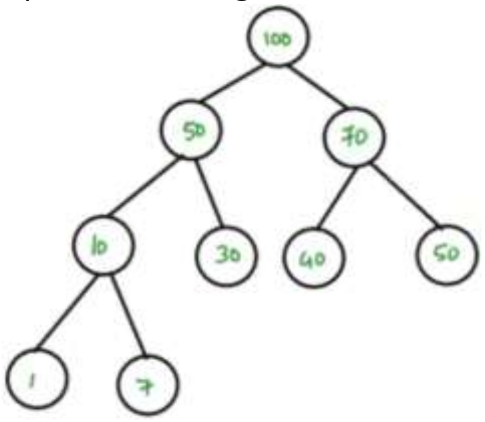
Building Binary heap min/max from list of numbers:
1st method: Willam's method / insertion methods: Insert element in empty heap one by one & check whether following heap properties or not. It is discussed in insertion operation.
2nd method: first construct complete binary tree using given list of numbers. Then start heapifying tree, start from last leaf node
Eg.
Min heap
Construct for 9, 6, 5, 2, 3



Max heap
Construct for 2, 9, 7, 6, 5, 8



List representation of binary heap: In complete binary tree we calculate number of nodes in each level except last level. eg Lo-1, L1-2, L3-4.



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

Root – [0]

Element at ith index – list[i]

Parent node – (i-1)//2

Eg parent of 30 ie i=4,

Parent node location = (i-1)//2 = (4-1)//2 = 1, Parent Node is list[1] = 50

Left child = (2*i)+1

Right child = (2*i)+2

Eg. For 70, i = 2, Left child location = 2i+1 = 2*2+1 = 5, Left child = list[5] = 40

Right child location = 2i+2 = 2*2+2 = 6, Right child = list[6] = 50

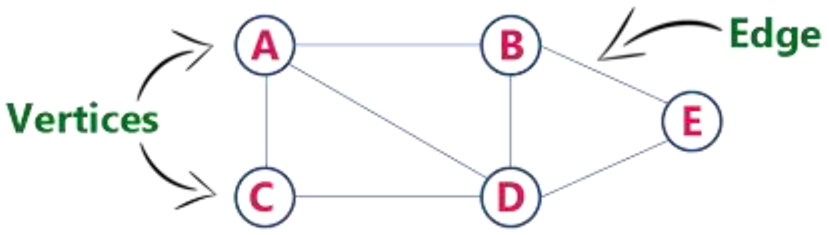
Python Program:

```
#Tree Data Structure
# Create new node
class BST:
    def __init__(self,key):
        self.key = key
        self.lchild= None
        self.rchild= None

# 1.Insertion/Adding Elements

    def insert(self,data):
        if self.key is None:
            self.key = data
            return
        if 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)

#2) Search Element
    def search(self,data):
        if self.key == data:
            print("Node is found!")
            return
        if self.key > data:
            if self.lchild:
                self.lchild.search(data)
            else:
                print("Node is not present in tree!")
        else:
            if self.rchild:
                self.rchild.search(data)
            else:
                print("Node is not present in tree!")

# 3) Traversal
    def preorder(self):
        print(self.key, end=" ")
        if self.lchild:
            self.lchild.preorder()
        if self.rchild:
            self.rchild.preorder()

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

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

#4) Deletion @ a) Other than Root Node
    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 in the tree")
        elif data > self.key:
            if self.rchild:
                self.rchild = self.rchild.delete(data)
            else:
                print("Given Node is Not present in the tree")
        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
            self.key = node.key
            self.rchild = self.rchild.delete(node.key)
        return self

# Deletion @ b) Root Node

#Minimum & Maximum Node
    def min_node(self):
        current = self
        while current.lchild:
            current = current.lchild
        print("Node with minimum key is :",current.key)

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

5) Graph

Graph:
In tree there is only one root node of every node have only one parent except root node. So for 'n' nodes there are 'n-1' edges & there is only one path from one node to another node.
In Graph there is no root node & every node is equal & also between two nodes we can have multiple paths. Also graph can contain cycles. Every tree is graph but every graph is not tree.



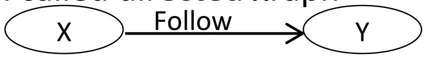
Eg
Multiple paths eg between C&E :
CABE, CDE, CDBE, CDABE
Cycle: ABDCA, ABDA.

A---B : A&B called vertices & AB is called edge.
Graph is math concept of study of graph called graph theory
ie G = (V , E)
G - Graph
V - Set of vertices,
E - Set of Edges.

Applications of graph:
1) Google maps & GPS to find shorted path from one destination to another.
2) Social networks like facebook & linkedin to show connection between users.
3) Ecommerce websites to show recommendations.
4) Google search algorithm uses graph.

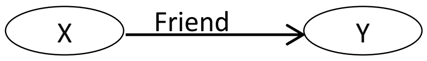
Types of Graph
1) Based on direction

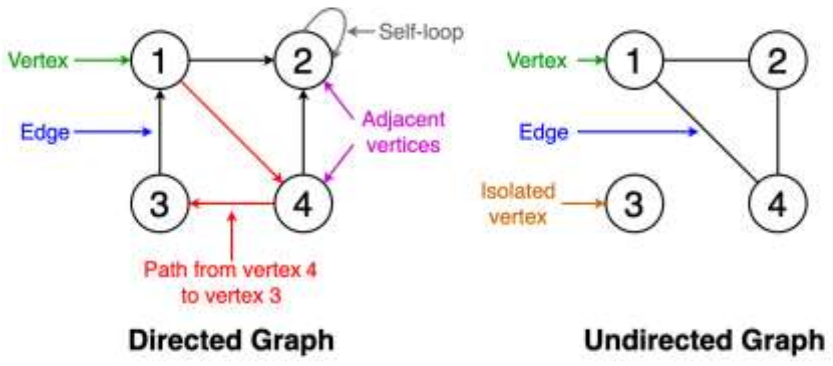
a) Directed Graph: If every edge of graph contains direction or edges are unidirectional called directed graph

Eg twitter


b) Undirected or Bidirectional Graph:

A Graph in which all the edges are bidirectional ie edges not point in any specific direction

Eg facebook




2) Based on weight

a) Weighted Graph: A graph in which each edge is assigned with some weight/cost/value eg length, cost, distance

b) Unweighted Graph: here no value or weight associated with edge.

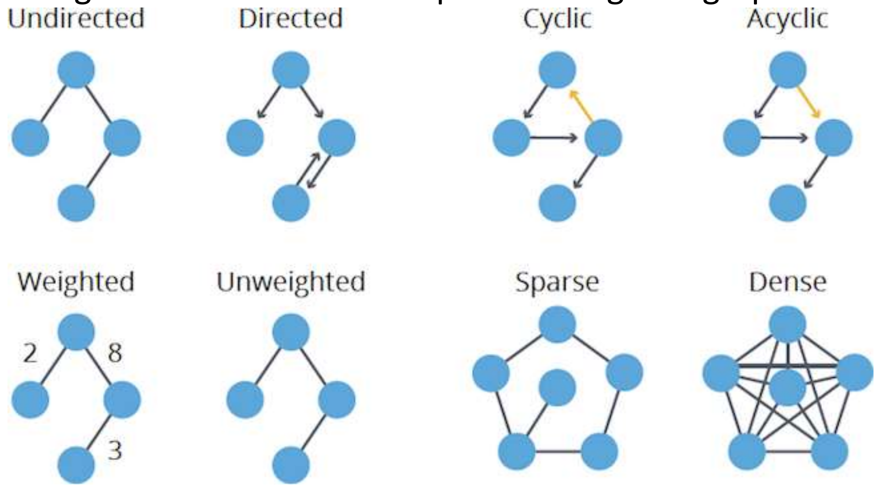
3) Based on cycle:

a) Cyclic graph: contain cycle

b) Acyclic graph: don't contain cycle eg tree.

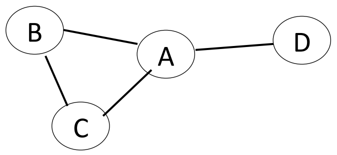
Dense graph: Edges equal to maximum possible edges in graph.

Sparse graph: Edges less than maximum possible edges in graph.

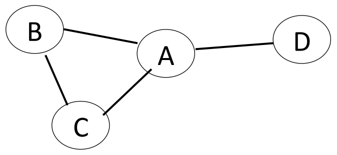


Terminologies in Graph:
Graph is non-linear data structure consisting of nodes & edges. In graph nodes are also called as vertices. Single node is called vertex. Link or connection between any two nodes called as edge.
Adjacent / Neighbor nodes: if there is edge from one node to another then that two nodes are adjacent nodes.
A ---- B , A&B are adjacent nodes..
A ---> B , A is adjacent to B, but B is not adjacent to B

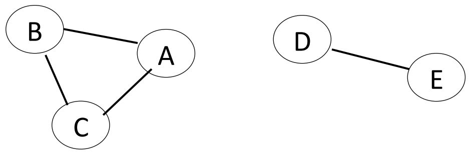
Path: Sequence of vertices in which each pair of successive nodes is connected by an edge. Multiple paths from one node to another node are possible.
Length of path is no of edges in path & it should be equal or greater than 1.
Simple Path: all of its vertices are distinct.



Here DABC is simple path but DABCAD is not simple path.
Closed path: 1st & last node is same
Eg DABCAD
Cycle: first & last node is same & all other nodes are distinct.
eg DABCAD not cycle, but ABCA is cycle.
Connected graph: all nodes connected.



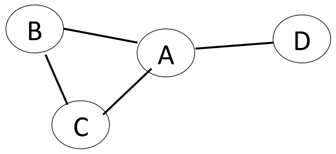
Disconnected Graph: all nodes not connected



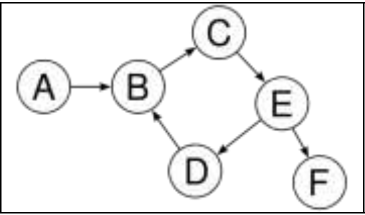
Strongly connected graph: In directed graph if there is no path between some node but if it converted to undirected graph nodes are connected by path then graph called strongly connected graph

Weakly connected graph: there doesn't exist any path between any two pair of vertices.

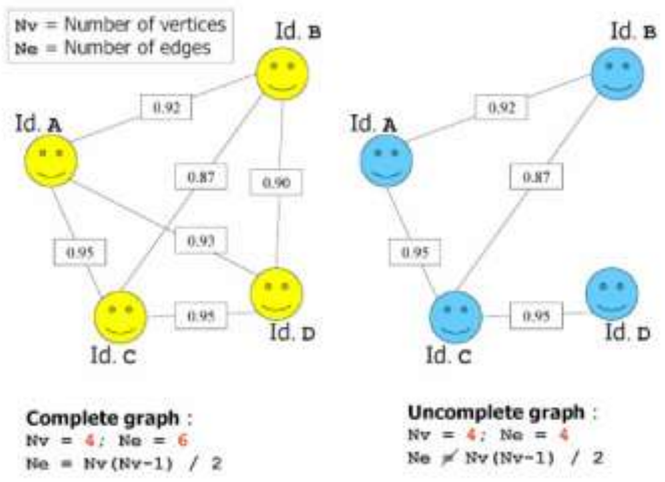
Degree of node: number of edges connected to node.eg deg of B-2, A-3



In directed graph indegree of node is number of edges coming to that node & outdegree of node is number of edges going outside from node.

	For B indgree - 2, outdegree - 1 For E indgree - 1, outdegree - 2

Complete graph: Any node is adjacent to all nodes ie every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

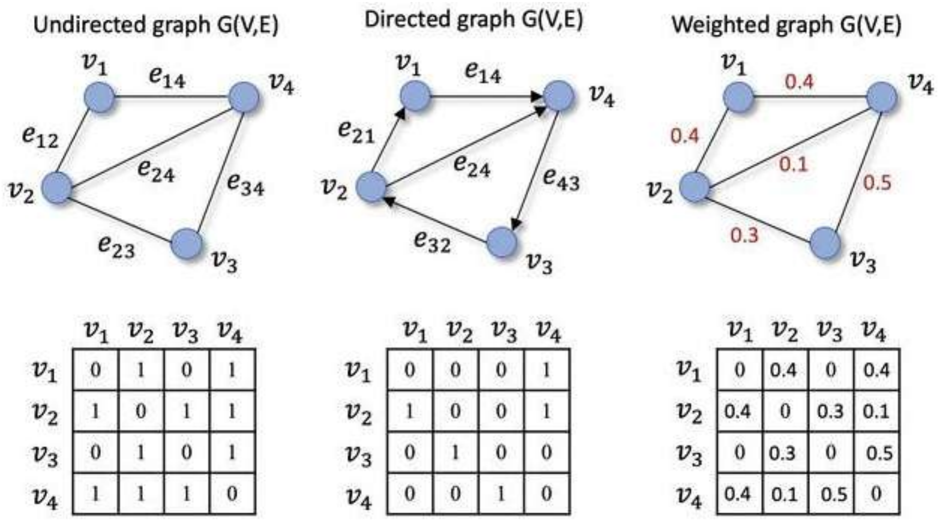


Graph representation:

Adjacency Matrix: It represents connection between nodes in matrix form

Steps:

1) Create k x k matrix where k is number of nodes.
2) Row & column represents nodes of the graph
3) If edge is present then store 1 otherwise store 0.



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

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

	v_1	v_2	v_3	v_4
v_1	0	0.4	0	0.4
v_2	0.4	0	0.3	0.1
v_3	0	0.3	0	0.5
v_4	0.4	0.1	0.5	0

Symmetric matrix represent undirected graph & weight in matrix represent weighted graph. Here storing data for connected & not connected so using more memory to store zeros.

2) Adjacency list: we store list of adjacent node of each node.
eg
Undirected graph:
v1: [v2,v4], v2: [v1,v3, v4], v3: [v2,v4], v4: [v1,v2,v3]
Directed graph:
v1: [v4], v2: [v1, v4], v3: [v2], v4: [v3]
Weighted graph:
v1: [(v2, 0.4), (v4, 0.4)]
v2: [(v1, 0.4), (v3, 0.3), (v4, 0.1)]
v3: [(v2, 0.3), (v4, 0.5)]
v4: [(v1, 0.4), (v2, 0.1), (v3, 0.5)]

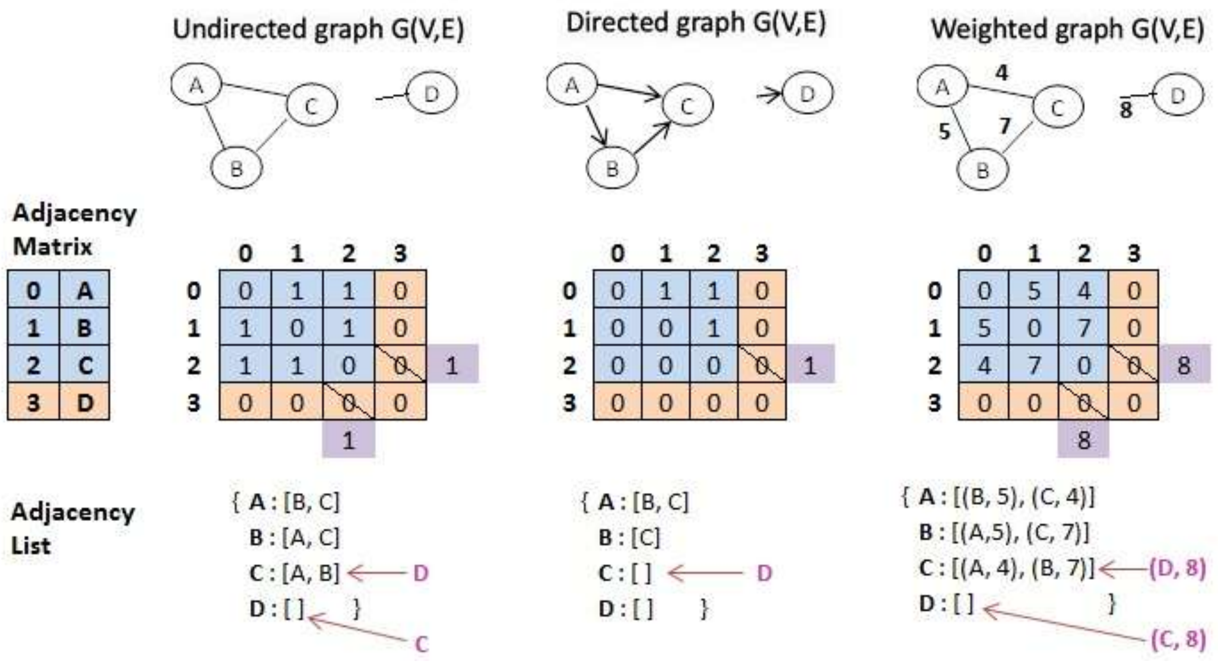
For this we can use nested list or dictionary in Python.

Graph Operation:

1) Insertion: Add node or edge to graph.

a) add node : For adding new node in undirected or directed or weighted graph add 1 row & 1 column to adjacency matrix filled with zero & in adjacency list add empty list as value to dictionary which representing graph.

b) add edge : Add new edge between mentioned vertices. Change value of matrix element or list in graph dictionary as per connection given



0	A
1	B
2	C
3	D

$$\begin{array}{c|cccc|c} & 0 & 1 & 2 & 3 & \\ \hline 0 & 0 & 1 & 1 & 0 & \\ 1 & 1 & 0 & 1 & 0 & \\ 2 & 1 & 1 & 0 & \cancel{0} & 1 \\ 3 & 0 & 0 & \cancel{0} & 0 & \\ \hline & & & 1 & & \end{array} \qquad \begin{array}{c|cccc|c} & 0 & 1 & 2 & 3 & \\ \hline 0 & 0 & 1 & 1 & 0 & \\ 1 & 0 & 0 & 1 & 0 & \\ 2 & 0 & 0 & 0 & \cancel{0} & 1 \\ 3 & 0 & 0 & 0 & 0 & \end{array} \qquad \begin{array}{c|cccc|c} & 0 & 1 & 2 & 3 & \\ \hline 0 & 0 & 5 & 4 & 0 & \\ 1 & 5 & 0 & 7 & 0 & \\ 2 & 4 & 7 & 0 & \cancel{0} & 8 \\ 3 & 0 & 0 & \cancel{0} & 0 & \\ \hline & & & 8 & & \end{array}$$

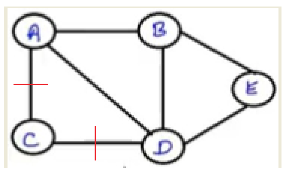
```
{ A : [B, C]            { A : [B, C]           { A : [(B, 5), (C, 4)]
  B : [A, C]              B : [C]                B : [(A,5), (C, 7)]
  C : [A, B] <-- D        C : [ ] <-- D          C : [(A, 4), (B, 7)] <-- (D, 8)
  D : [ ] <-- C }         D : [ ]    }           D : [ ] <-- (C, 8)     }
```

2) Deletion:

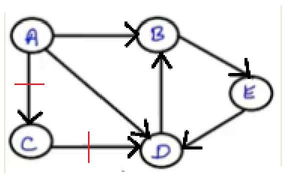
a) delete node/ vertex : delete node & all edges or connections of that node from graph. In adjacency matrix delete row & column associated with deleted node. In adjacency list dictionary, delete key-value pair of node & also delete mode from other key's values.

Deletion of Node C

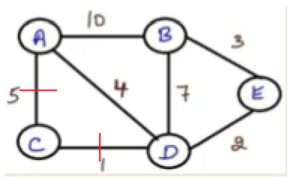
Undirected graph G(V,E)



Directed graph G(V,E)



Weighted graph G(V,E)



Adjacency Matrix

Before Deletion

0	A
1	B
2	C
3	D
4	E

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

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

	0	1	2	3	4
0	0	10	5	4	0
1	10	0	0	7	3
2	5	0	0	1	0
3	4	7	1	0	2
4	0	3	0	2	0

After Deletion

0	A
1	B
2	C
3	D
4	E

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

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

	0	1	2	3	4
0	0	10		4	0
1	10	0		7	3
2					
3	4	7		0	2
4	0	3		2	0

Final Matrix

0	A
1	B
2	D
3	E

	0	1	2	3
0	0	1	1	0
1	1	0	0	1
2	1	1	0	1
3	0	1	1	0

	0	1	2	3
0	0	1	1	0
1	0	0	0	1
2	0	1	0	0
3	0	0	1	0

	0	1	2	3
0	0	10	4	0
1	10	0	7	3
2	4	7	0	2
3	0	3	2	0

Adjacency List

Before Deletion

{ **A** : [B, C, D]
B : [A, D, E]
C : [A, D]
D : [A, B, C, E]
E : [B, D] }

{ **A** : [B, C, D]
B : [E]
C : [D]
D : [B]
E : [D] }

{ **A** : [(B,10), (C,5), (D,4)]
B : [(A,10), (D,7), (E,3)]
C : [(A,5), (D,1)]
D : [(A,4), (B,7), (C,1), (E,2)]
E : [(B,3), (D,2)] }

After Deletion

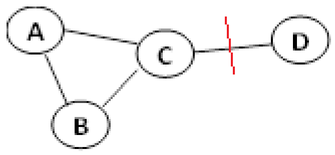
{ **A** : [B, D]
B : [A, D, E]
D : [A, B, E]
E : [B, D] }

{ **A** : [B, D]
B : [E]
D : [B]
E : [D] }

{ **A** : [(B,10), (D,4)]
B : [(A,10), (D,7), (E,3)]
D : [(A,4), (B,7), (E,2)]
E : [(B,3), (D,2)] }

b) delete edge : delete edge if present in graph, in case of adjacency matrix change value & in adjacency list delete nodes from list for required edge.

Delete Edge CD



Adjacency Matrix

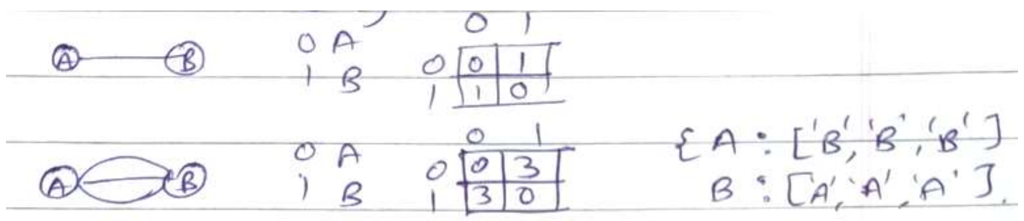
0	A
1	B
2	C
3	D

	0	1	2	3	
0	0	1	1	0	
1	1	0	1	0	
2	1	1	0	~~1~~	0
3	0	0	~~1~~	0	
			0		

Adjacency List

{ **A** : [B, C]
B : [A, C]
C : [A, B, D]
D : [C] }

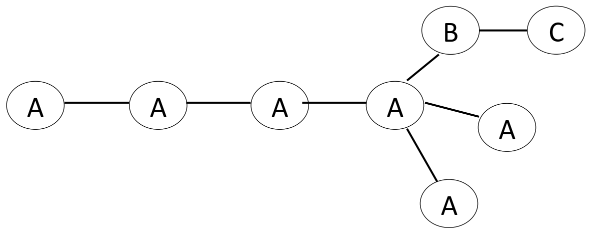
Multi-edge graph: For undirected & directed multi-edge graph in adjacency matrix instead of storing 1, store number of edges.



In program, add counter in add_edge() function to count edge & in deletion add decrement counter in delete_edge() function.

Graph traversal Algorithms:

1) DFS - Depth first search: First visit any node as starting node & print node, then visit any unvisited adjacent node & print that node & repeat process until all nodes covered. If traversal operation not completed & not have option to move forward ie don't have unvisited adjacent node then we can take one step back



Result: A, B, C, D, K, M, E, F

So, after reaching dead-end, we use back tracking & move back to search remaining part.

Implementation of DFS using:

1) Stack
2) Recursion
3) Iterative approach

Python Program:

```
# Adjacency Matrix
# 1) Adding Node
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(format(graph[i][j],"<3"),end=" ")
      print()

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

# directed Graph
def add_edge_dir(v1,v2):
   if v1 not in nodes:
      print(v1,"is not present in the graph")
   elif v1 not in nodes:
      print(v1,"is not present in the graph")
   else:
      index1 = nodes.index(v1)
      index2 = nodes.index(v2)
```

```
        graph[index1][index2] = 1
        #graph[index2][index1] = 1

# weighted Graph
def add_edge_wted(v1,v2,cost):
    if v1 not in nodes:
        print(v1,"is not present in the graph")
    elif v1 not in nodes:
        print(v1,"is not present in the graph")
    else:
        index1 = nodes.index(v1)
        index2 = nodes.index(v2)
        graph[index1][index2] = cost
        graph[index2][index1] = cost

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)

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

nodes = []
graph = []
node_count = 0

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

# Add Edge
# undirected graph
def add_edge_undir_l(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)

# directed graph
def add_edge_dir_l(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)

# weighted graph
def add_edge_wted_l(v1,v2,cost):
    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:
        list1 = [v2,cost]
        list2 = [v1,cost]
        graph[v1].append(list1)
        graph[v2].append(list2)

# Delete node
# undirected & directed graph
def delete_node_l(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)

# weighted graph
def delete_node_wted_l(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]
            for j in list1:
                if v == j[0]:
                    list1.remove(j)
                    break

# Delete Edge
# undirected graph
def delete_edge_undir_l(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)

# directed graph
def delete_edge_dir_l(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)

# weighted graph
def delete_edge_wted_l(v1,v2,cost):
    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:
        temp = [v1,cost]
        temp1 = [v2,cost]
        if temp1 in graph[v1]:
            graph[v1].remove(temp1)
            graph[v2].remove(temp)

graph = {}

# Graph traversal
# Using Recursion function
def DFS(node,visited,graph):
    if node not in graph:
        print("Node is not present in graph")
        return
    if node not in visited:
        print(node)
        visited.add(node)
        for i in graph[node]:
            DFS(i,visited,graph)

visited = set()
graph = {}

# Using Iterative Approach
def DFS_iterative(node,graph):
    visited = set()
    if node not in graph:
        print("Node not present in the graph")
        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 = {}
```