Before we get into data structure let’s learn about time and space complexity.

**Time Complexity:**

The time complexity of an algorithm quantifies the **amount of time taken by an algorithm to run as a function of the length of the input**.

Here time is, time to run is a function of the length of the input and **not the actual execution time** of the machine on which the algorithm is running on.

**Note:**

1. For calculating time complexity, we assume that it always takes constant time to perform a operation.

e.g—for operation “a+b” it will take same time for any of input pair of a and b (1,11),(-99,234),(89328,99283)

1. For calculating time complexity we always check for worst case scenario ( case where will take more time)
2. We use below rule for calculating time complexity-
3. **Keep fastest growing term**
4. **Drop constant**

**Question:**

For given array of length n find time complexity to calculate whether a pair (x,y) exists whose sum will be z (given)

Method/code is:-

def findPair(a, n, z) :

    # Iterate through all the pairs

    for i in range(n) :

        for j in range(n) :

            # Check if the sum of the pair

            # (a[i], a[j]) is equal to z

            if (i != j and a[i] + a[j] == z) :

                return True

    return False

Solution:

We assume that each operation take a constant time C.

Number of line code to be executed depends on Z.

During analyses of the algorithm, mostly the worst-case scenario is considered, i.e., when there is no pair of elements with sum equals Z. ***In the worst case,***

N\*c operations are required for input.

The outer loop i loop runs N times.

For each i, the inner loop j loop runs N times.

*So total execution time is* ***N\*c + N\*N\*c + c***

**Now applying rule from point number 3:-**

*Execution time –* ***N\*N\*c ------*** Consider the fastest growing term-

*Execution time* **– N\*N** ------- Dropping constant

**Time complexity = O(N\*N) ------- Answer**

**###################**

**# Space complexity #**

**###################**

Space Complexity of an algorithm is the total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

**What is Data Structure?**

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as- Array, Linked list etc.

**Types of Data Structures**

There are two types of data structures:

1. Primitive data structure
2. Non-primitive data structure

**Primitive Data structure**

The primitive data structures are primitive data types. e.g- int, char, float, double etc.

**Non-Primitive Data structure**

The non-primitive data structure is divided into two types:

1. Linear data structure
2. Non-linear data structure

**Linear Data Structure**

The arrangement of data in a sequential manner is known as a linear data structure.

In these data structures, one element is connected to only one element in a linear form.

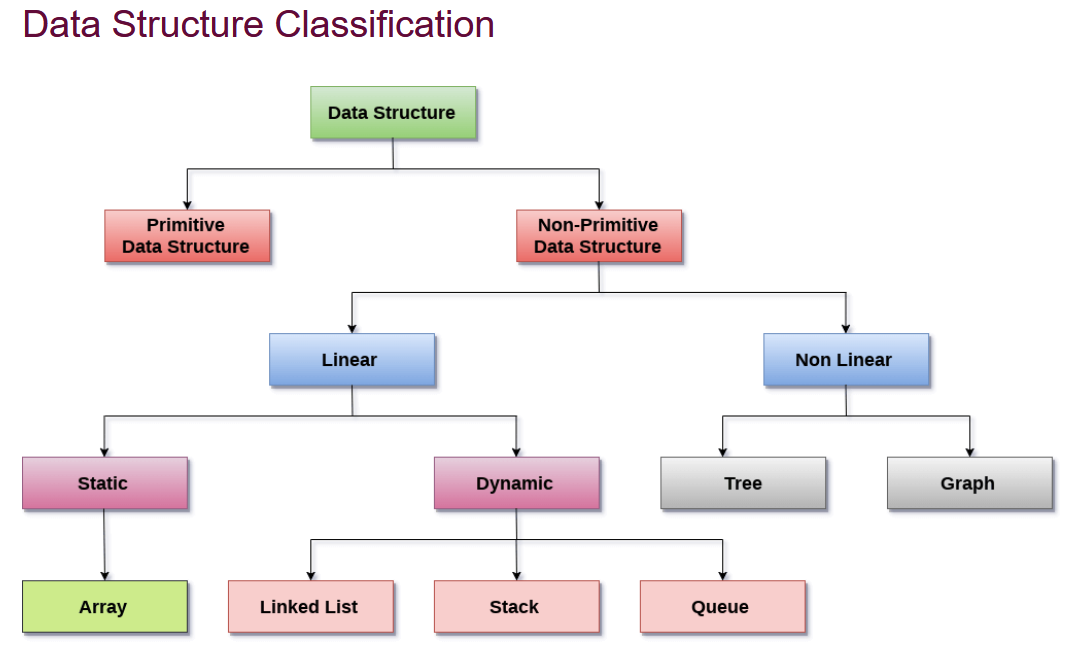
The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues.

**Non-linear data structure**

When one element is connected to the 'n' number of elements known as a non-linear data structure.

In this case, the elements are arranged in a random manner.

The best example is trees and graphs.



**Static data structure:**

It is a type of data structure where the size is allocated at the compile/creation time. Therefore, the maximum size is fixed. e.g.-- tuple

**Dynamic data structure:**

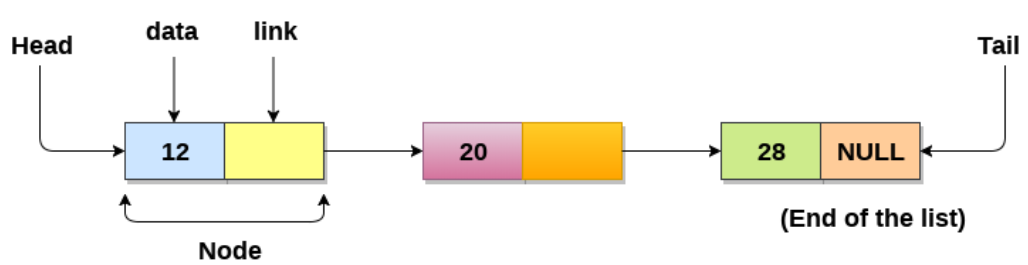
It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible. e.g - list

**Linked List**

Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.

A node contains two fields i.e. data and the pointer/link which contains the address of the next node in the memory.

The last node of the list must contain pointer to the null.



**Array**

Here we are discussing on linked list, will spent 2 minutes on array also-

Array is used for homogeneous data type of fixed length/size and immutable.

In array data is stored in index basis

**Array vs Linked list**

|  |  |
| --- | --- |
| Array | Linked list |
| * Data to be stored is of fixed length * Stores homogenous data * Data is stored sequency in memory | * Data to stored is not fixed length * Stores heterogeneous * Data is stored in nodes and it’s not sequentially. |

We can further break linked list in three parts—

1. Singly Linked list
2. Doubly linked list
3. Circular linked list

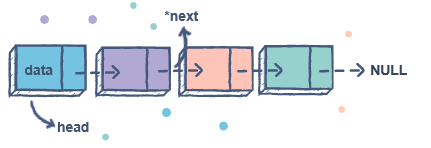
Linked list is same as array, but array have below limitation which can be removed in linked list-

**Limitation of array**

1. The size of array must be known in advance before using it in the program.
2. All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.
3. Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.

**Singly Linked list**

A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).



**Creating singly linked list**

class Node:

   def \_\_init\_\_(self, dataval=None):

      self.dataval = dataval

      self.nextval = None

class SLinkedList:

    def \_\_init\_\_(self):

      self.headval = None

list1 = SLinkedList()

list1.headval = Node("Mon")

e2 = Node("Tue")

e3 = Node("Wed")

# Link first Node to second node

list1.headval.nextval = e2

# Link second Node to third node

e2.nextval = e3

print(list1.getvalue())

**Note:**

'headval' is instance variable which point to first value/node of the list. Don’t change of assign any other value to this variable b/c first node value may get changed.

**Accessing all values from singly linked list:**

class Node:

   def \_\_init\_\_(self, dataval=None):

      self.dataval = dataval

      self.nextval = None

class SLinkedList:

    def \_\_init\_\_(self):

      self.headval = None

    def getvalue(self):

        print('head value is: ',self.headval)

        while self.headval:

            print(self.headval.dataval)

            self.headval=self.headval.nextval # assign next node

list1 = SLinkedList()

list1.headval = Node("Mon")

e2 = Node("Tue")

e3 = Node("Wed")

# Link first Node to second node

list1.headval.nextval = e2

# Link second Node to third node

e2.nextval = e3

list1.getvalue()

**Inserting values at end of linked list**

Here we have defined a method insertval. When nextval points to None that means we are at end of list and will assign it’s address pointer to new value.

class Node:

   def \_\_init\_\_(self, dataval=None):

      self.dataval = dataval

      self.nextval = None

class SLinkedList:

    def \_\_init\_\_(self):

      self.headval = None

    def insertval(self, data):

        newNode = Node(data)

        if(self.headval):

            current = self.headval

            while(current.nextval):

                current = current.nextval

            current.nextval = newNode # we are at last element, add new data in list

        else:

            self.headval = newNode

    def getvalue(self):

        # print('head value is: ',self.headval.dataval)

        while self.headval:

            print(self.headval.dataval)

            self.headval=self.headval.nextval

list1 = SLinkedList()

list1.headval = Node("Mon")

e2 = Node("Tue")

e3 = Node("Wed")

# Link first Node to second node

list1.headval.nextval = e2

# Link second Node to third node

e2.nextval = e3

# list1.getvalue()

list1.insertval('Thu')

list1.getvalue()

Getting the size of linked list

class SLinkedList:

    def \_\_init\_\_(self):

      self.headval = None

    def getsize(self):

        size=0

        if (self.headval) != None:

            while self.headval:

                print(self.headval.dataval)

                self.headval=self.headval.nextval

                size+=1

            print('size is: ',size)

        else:

            print('size is: ',size)

**Note:**

Here we have created getsize method which count number of data till the last node.

Last data means nextval is None.

**Questions**

Do insert operation in the linked list and then get the size of linked list

class Node:

   def \_\_init\_\_(self, dataval=None):

      self.dataval = dataval

      self.nextval = None

class SLinkedList:

    def \_\_init\_\_(self):

      self.headval = None

    def getsize(self):

        size=0

        if (self.headval) != None:

            while self.headval:

                print(self.headval.dataval)

                self.headval=self.headval.nextval

                size+=1

            print('size is: ',size)

        else:

            print('size is: ',size)

    #this methos is for inserting new value at end of linkedlist

    def insertval(self, data):

        newNode = Node(data)

        if(self.headval) != None:

            current = self.headval

            while(current.nextval):

                current = current.nextval

            current.nextval = newNode

        else:

            self.headval = newNode

list1 = SLinkedList()

list1.headval = Node("Mon")

e2 = Node("Tue")

e3 = Node("Wed")

# Link first Node to second node

list1.headval.nextval = e2

# Link second Node to third node

e2.nextval = e3

list1.insertval('Thu')

print()

list1.getsize()

**Method 2:** ------ More suggestable

def add\_node(self,data):

        if self.head==None:

            print('SLL was empty')

            self.head=Node(data)

        else:

            current=self.head

            while current:

                if current.next==None:

                    break

                else:

                    current=current.next

            current.next=Node(data)

**Searching data into linked list**

class Node:

   def \_\_init\_\_(self, dataval=None):

      self.dataval = dataval

      self.nextval = None

class SLinkedList:

    def \_\_init\_\_(self):

      self.headval = None

    def search(self,data):

        firstnode=self.headval

        while firstnode:

            if firstnode.dataval==data:

                print(data,' found in node')

                return 1

            firstnode=firstnode.nextval

        else:

            print(data,' not found in node')

Note:

We have below logic in search method:

1. Get the head(first value) value of node if comparing\_data is present the return success.
2. If comparing\_data not present the go to next node(till next node is not None, means last node) compare with next node data.
3. If comparing \_data found then return success else repeat step 2.

**Delete data from linked list:**

This is same as search but here wi will have to remember the previous node also.

When data is found then then we will assign/move pointer/address of previous\_node next of found node.

class Node:

   def \_\_init\_\_(self, dataval=None):

      self.dataval = dataval

      self.nextval = None

class SLinkedList:

    def \_\_init\_\_(self):

      self.headval = None

    def del\_data(self,data):

        if self.headval==None:

            return

        elif self.headval.dataval==data:

            current=self.headval

            self.headval=self.headval.nextval

            current.nextval=None

            return

        else:

            previous=None

            current=self.headval

            while current:

                if current.dataval==data:

                    previous.nextval=current.nextval

                    current.nextval=None

                    break

                else:

                    previous=current

                    current=current.nextval

**line xyz**

here we are assigning pointer of previous\_node next of current\_node.

#############################################################################

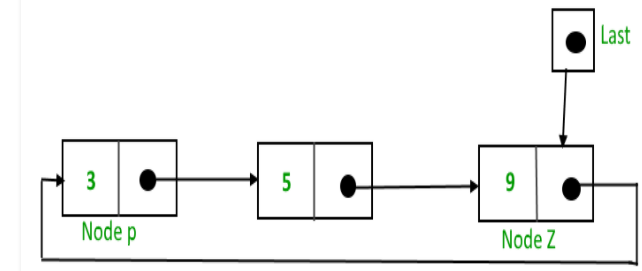
# Circular linked list #

#############################################################################

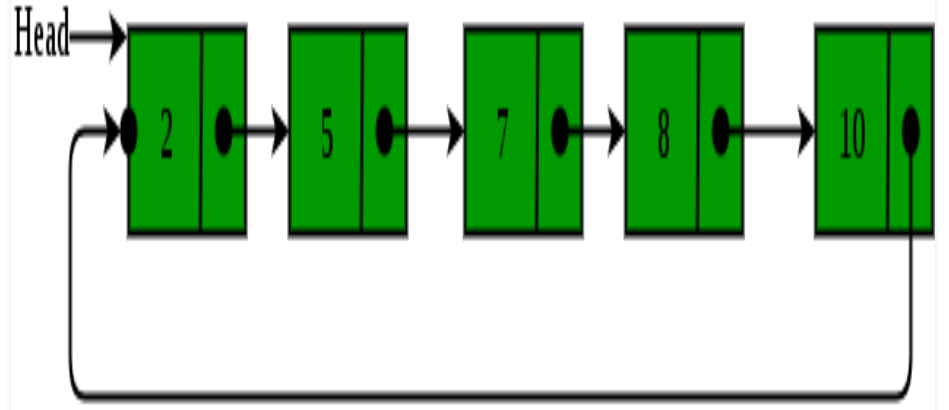
The circular linked list is the collection of nodes in which tail node also point back to head node. We can traverse and identify all node using a single pointer

Head will point to the first element of the list, and tail will point to the last element in the list.

If maintaining last node pointer



If maintaining first node pointer



**Advantage of Circular linked list:**

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. We don’t need to maintain two pointer (if required) we can maintain last pointer and first can be obtained as next of last.
3. Circular lists are useful in applications to repeatedly go around the list. E.g- On any operating system we can put all programs/process in a circular linked list and then allocate time to execute a bit of task for each program/process.

**Operation on Singly Circular linked list:**

We will be doing below operation on circular linked list-

1. Add at beginning
2. Add at end
3. Add in between
4. Deletion
5. Search

**Implementation of Circular linked list**

To implement a circular singly linked list, we take one pointer that points to the last node of the list, the next node of last node will point to the first node.

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class CircularLinkedList:

    def \_\_init\_\_(self):

        self.last = None

**Adding elements in linked list**

A node can be added in below 3 ways-

* Insertion at the beginning of the list
* Insertion at the end of the list
* Insertion in between the nodes

**Adding node at beginning**

If we are adding any element at beginning, then we will check if list is empty and add accordingly.

class Node:

    def \_\_init\_\_(self, data):

        self.data = data

        self.next = None

class CircularLinkedList:

    def \_\_init\_\_(self):

        self.last = None

    def addBegin(self,data):

        node=Node(data)

        #If ndoe list is empty

        if self.last==None:

            self.last=node

            self.last.next=self.last

        #If list is not empty

        else:

            node.next=self.last.next

            self.last.next = node

**Getting data from circular linked list:**

For getting data first we will check if list is empty, if empty then exit from method else getting data from list from each node of list.

If list is not empty then go to first node and keep iterating till you come back again to first node.

    def traverse(self):

        #if list is empty

        if (self.last == None):

            print("List is empty")

            return

        #control will code here if list is not emtpy

        temp = self.last.next #temp points here to first node

        while temp:

            print(temp.data, end = " ")

            temp = temp.next #Go to next node

            if temp == self.last.next: #if reached back to first node then terminate loop

                break

**Adding node at end**

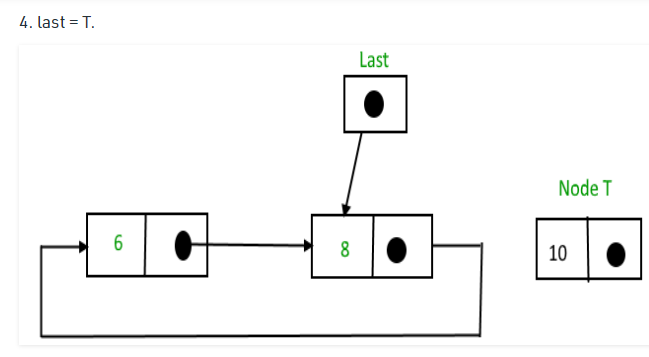
Let’s us suppose we have to add new Node T in list. To add data in list as ned we will follow below procedure-

1. Create a node, say T.

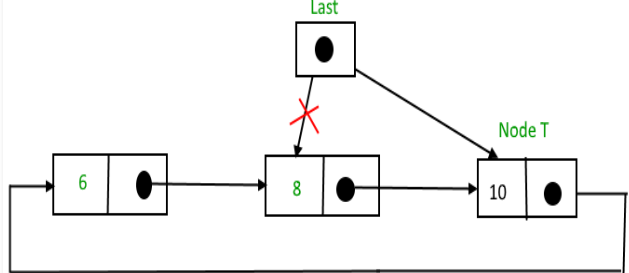
2. Make T -> next = last -> next;

3. last -> next = T.

4. last = T.



After adding it will look like



    def addLast(self,data):

        node=Node(data)

        #check if list is emtpy

        if self.last==None:

            self.last=node

            self.last.next=self.last

        #node is not empty

        else:

            node.next=self.last.next #next of new node=first node

            self.last.next=node #nest of last = new node

            self.last=node      #last node=new node

**Adding data after given value:**

Here we will add a node in list after a given value. We will follow procedure-

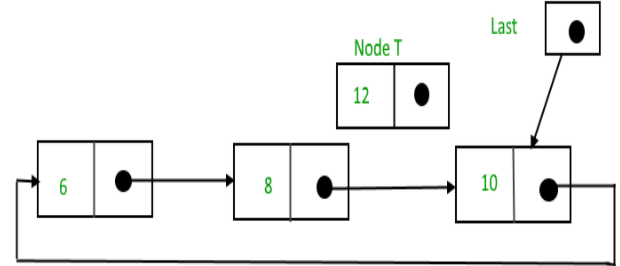
1. Create a node, say T.

2. Search for the node after which T needs to be inserted, say that node is P.

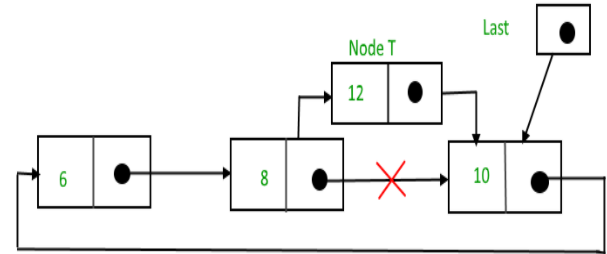
3. Make T -> next = P -> next;

4. P -> next = T.

Let say we want to insert after 8.



After insertion:



    def addAfter(self,data,item): #item is the value after which node will be inserted

        node=Node(data)

#In case list was emtpy

        if self.last==None:

            print('list is empty')

            return

        p=self.last.next

        while p:

            if p.data==item: #True means we have to add after current node

                print('\*')

                node.next=p.next

                p.next=node

                #check we item we found was last node

                #if yes then we need to update last variable

                if self.last==p:#item was at last node

                    self.last=node#changed the last node value

                    return

                else:

                    return

            p=p.next

            if p==self.last.next:

                print('value ',item,' not found in singly circular list')

                return

**Question:**

WAP to add a new node before given node.

Hint: Probably we will have to use a variable to node information of previous node

**Deleting node from Circular Singly linked list**

Here we are having only one pointer (last) so we need to have one variable which will have information of last node in doing delete operation.

    def delete\_data(self,item):

        if self.last==None:

            print('operation can not be performed on empty scll')

            return

        else:

            #if list have only one node

            if self.last.next==self.last:

                if self.last.data==item:

                    self.last=None

                    return

                else:

                    print(item, 'not found in in list' )

                    return

            #if item is at first node

            if self.last.next.data==item:

                self.last.next=self.last.next.next

                return

            privous=None

            p=self.last.next

            while p:

                if p.data==item:

                    privous.next=p.next

#p.next=None #---- In case want t delink node from list

                    if p ==self.last:

                        self.last=privous

                        return

                    else:

                        return

                privous=p

                p=p.next

                if p==self.last.next:

                    print(item,' not found in list')

                    return

**#############################################################################**

**# Circular linked list using two pointers (head and tail) #**

**#############################################################################**

**----** not suggestable

**Note:**

In circular linked list, we need to have two instance variable – head (points to first node) and end (last node).

Creating circular linked list

class Node:

    def \_\_init\_\_(self,data=None,next\_node=None):

        self.dataval=data

        self.next\_node=next\_node

class CLL():

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=end

    def getvalue(self):

        current\_node=self.head

if current\_node==None:

            print('list is empty')

            return

        while current\_node.next\_node:

            print(current\_node.dataval)

            current\_node=current\_node.next\_node

            if current\_node==self.head:

                break

    def insert\_end(self,data):

        new\_node=Node(data)

        #if CLL is empty

        if self.head==None:

            self.head=new\_node

            self.head.next\_node=new\_node

            self.end=new\_node

        #if CLL is not empty

        else:

            self.end.next\_node=new\_node

            self.end= new\_node

            new\_node.next\_node=self.head

    def insert\_start(self,data):

        new\_node=Node(data)

        first\_node=self.head

        self.end.next\_node=new\_node

        self.head=new\_node

        self.head.next\_node=first\_node

**Size of circular linked list**

class Node:

    def \_\_init\_\_(self,data=None,next\_node=None):

        self.dataval=data

        self.next\_node=next\_node

class CLL():

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=end

    def size(self):

        current\_node=self.head

        last\_node=self.end

        size=0

        while current\_node:

            if current\_node==last\_node: #either one node in list or at end node of list

                size+=1                 #increase size by 1

                break

            if current\_node!=last\_node: #not at end node, so go to next node and increase size

                size+=1

                current\_node=current\_node.next\_node

        print('size is: ',size)

Searching for data in circular linked list:

class Node:

    def \_\_init\_\_(self,data=None,next\_node=None):

        self.dataval=data

        self.next\_node=next\_node

class CLL():

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=end

    def search(self,data):

        current\_node=self.head

        last\_node=self.end

        pos\_index=0

        while current\_node:

            if current\_node.dataval==data:

                print('data found at position(starting position=1) : ',pos\_index)

                break

            elif current\_node.dataval!=data and current\_node.next\_node!=self.head:

                pos\_index+=1

                current\_node=current\_node.next\_node

            else:

                print('print data not found')

                break

**Deleting data from circular linked list**

This is same as singly linked list, we need to remember the previous node (previous\_node).

class Node:

    def \_\_init\_\_(self,data=None,next\_node=None):

        self.dataval=data

        self.next\_node=next\_node

class CLL():

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=end

    def delet\_data(self,data):

        current\_node=self.head

        last\_node=self.end

        previous\_node=None

#if circular linked list have only one node

        if self.head==self.tail:

            self.head=self.tail=None

            return

        if current\_node.dataval==data: #if data is at first node

            self.head=self.head.next\_node

            self.end.next=self.head

            return

        else:                         #data not at first node

            previous\_node=current\_node

            current\_node=self.head.next\_node

            while current\_node!=self.head:#till we come back to first node

                if current\_node.dataval==data: #current node data is required data

                    previous\_node.next=current\_node.next

                    #if we have to deleted/delinked last node

                    if current\_node==self.tail:

                        previous\_node.next=current\_node.next

                        self.tail=previous\_node

                        return

                    else:

                        return

                else:

                    previous\_node=current\_node

                    current\_node=current\_node.next\_node

            else:

                print('data not present, delete operation failed')

Steps:

prevous\_node ---- this remember the last node

1. Check if required data is at first node. If yes then just change the head node and last node’s next node. If no then continue steps 2
2. If current node is not the last node then check value’s if value not same then go to next node. If values are same then assign next\_node of previous node to be next\_node of current node.

**Method 2:**

    def del\_data(self,data):

        current\_node=self.headval

        previous\_node=None

        if current\_node.dataval==data:#if data is first node

            current\_node=current\_node.next

            self.tail.next=current\_node

            self.headval=current\_node

            return

        if self.tail.dataval==data: #data is at tail end

            while current\_node!=self.tail:

                previous\_node=current\_node

                current\_node=current\_node.next

            previous\_node.next=self.headval

            self.tail=previous\_node

            return

        if current\_node.dataval!=data and current\_node!=self.tail:#data is not at first and last node

            previous\_node=current\_node

            current\_node=current\_node.next

            while current\_node!=self.tail:

                if current\_node.dataval==data:

                    print('data found')

                    previous\_node.next=current\_node.next

                    return

                else:

                    previous\_node=current\_node

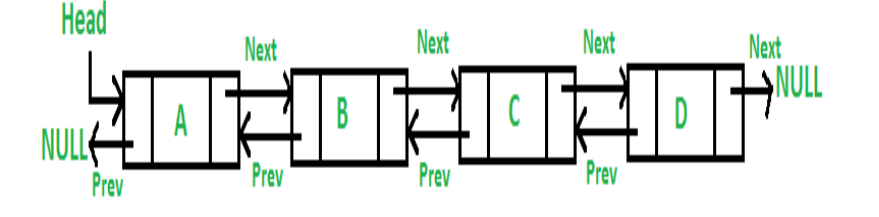
                    current\_node=current\_node.next

            else:

                print('data not found')

**Doubly linked list**

Doubly linked list is data structure in which each node contains three fields: data, next node pointer and previous\_node pointer.



**data**: represents the data value stored in the node

**previous\_node**: represents a pointer that points to the previous node. Null for first Node

**next\_node**: represents a pointer that points to the next node in the list. Null for last node

**Advantage of Doubly linked list:**

1. We can traverse if any direction, i.e. it’s bidirectional.
2. Delete operation is more efficient b/c we have previous node and next node point in each node
3. We can quickly insert a data before/after given node or data

**Drawback on doubly linked list:**

1. Every node of DLL requires extra space for a previous pointer
2. All operations require an extra pointer previous to be maintained

**Application of doubly linked list:**

* It is used in the navigation systems where front and back navigation is required.
* It is used by the browser to implement backward and forward navigation of visited web pages that is a back and forward button.
* It is also used to represent a classic game deck of cards.

**Creating Doubly liked list**

Here each node contains data, next node address, previous address so Node class will have three instance variables.

In double linked list class, we need to have to instance variable Head and Tail. In case of circular linked list we can manage from one variable b/c or circular behaviors

class Node:

    def \_\_init\_\_(self,dataval=None,next\_node=None,previous\_node=None):

        self.dataval=dataval

        self.next\_node=next\_node

        self.previous\_node=previous\_node

class DLL:

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=None

**Inserting data in Doubly Linked List**

We can do insertion in doubly linked list in below 4 ways-

1) At the front of the DLL

2) After a given node.

3) At the end of the DLL

4) Before a given node.

**Ingesting at front and end of doubly linked list**

class Node:

    def \_\_init\_\_(self,dataval=None,next\_node=None,previous\_node=None):

        self.dataval=dataval

        self.next\_node=next\_node

        self.previous\_node=previous\_node

class DLL:

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=None

    def insert\_end(self,data):

        new\_node=Node(data)

        #if list was empty

        if self.head==None:

            self.head=self.end=new\_node

        else:

            self.end.next\_node=new\_node #set next node of last node as newly created node

            new\_node.previous\_node=self.end #previous node newly node=last node

            self.end=new\_node #now last node will be set to newly node

    def insert\_start(self,data):

        new\_node=Node(data)

        #if list was empty

        if self.head==None:

            self.head=self.end=new\_node

        else:

            self.head.previous\_node=new\_node #previous node of head node will be newly node

            new\_node.next\_node=self.head #next node of new node will be head/first node

            self.head=new\_node #change head node as newly node

    def getvalue\_in\_pos\_x(self): #getting data in +x direction, define for -x direction

        print('getting data',self.head.dataval)

        current\_node=self.head

        while current\_node!=None:

            print(current\_node.dataval)

            current\_node=current\_node.next\_node

d=DLL()

d.insert\_end(10)

d.insert\_end(33)

d.insert\_start('first')

d.insert\_end(21)

d.getvalue\_in\_pos\_x()

**Inserting data after given data/node:**

We will start from head node and check each value till tail node.

When we found the value after which data must insert then will create new node and insert. If item is at tail node, then we will have to tail value and add new node.

        def add\_after(self,value,item): #item is value after which value will be added

        node=Node(value)

        current=self.head

        while current:

            if current.dataval==item:

#if item is at tail node

                if current==self.tail:

                    node.previous\_node=self.tail

                    self.tail.next\_node=node

                    self.tail=node

                    break

                else:

                    node.previous\_node=current

                    node.next\_node=current.next\_node

                    current.next\_node.previous\_node=node

                    current.next\_node=node

                    break

            current=current.next\_node

        else:

            print(item,' not found in list, operation can not be performed')

**Adding data before a node**

Here procedure is same as above (after a value) but we will have to check if item is present at head node or not. In case present on head node then will have to update the head value.

    def add\_before(self,value,item):

        node=Node(value)

        current=self.head

        while current:

            if current.dataval==item:

                #check if item is at head then update the head value

                if current==self.head:

                    node.next\_node=self.head

                    self.head.previous\_node=node

                    self.head=node

                    return

                else:

                    node.previous\_node=current.previous\_node

                    node.next\_node=current

                    current.previous\_node.next\_node=node

                    current.previous\_node=node

                    return

            current=current.next\_node

        else:

            print(item,'  not found in DLL operation can not be performed')

**Searching data into doubly linked list:**

Start checking data from head node till last node.

class Node:

    def \_\_init\_\_(self,dataval=None,next\_node=None,previous\_node=None):

        self.dataval=dataval

        self.next\_node=next\_node

        self.previous\_node=previous\_node

class DLL:

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=None

    def search(self,data):

        current\_node=self.head

        while current\_node:

            if current\_node.dataval==data:

                print('data was found')

                break

            else:

                current\_node=current\_node.next\_node

        else:

            print('data was not found')

**Deleting given data from Node**

If we look the DLL structure then we don’t need to remember the previous node b/c we can access previous node using previous\_node instance variable.

**Method 1:** Don remember the previous\_node ----- suggestable and easy

    def delete\_data(self,data):

        current=self.head

        while current:

            if current.dataval==data:

                #if data was is at head then head need to be updated

                if current==self.head:

                    self.head.next\_node.previous\_node=None

                    self.head=self.head.next\_node

                    current.next\_node=None

                    return

                #if data was at tail then tail need to be updated

                if current==self.tail:

                    self.tail.previous\_node.next\_node=None

                    self.tail=self.tail.previous\_node

                    current.previous\_node=None

                    return

                else:

                    current.previous\_node.next\_node=current.next\_node

                    current.next\_node.previous\_node=current.previous\_node

                    return

            current=current.next\_node

        else:

            print(data,' not found in DLL')

**Method 2:** Remember the previous\_node ---- not suggestable and difficult

It’s same as deleting from DLL or SLL. Here we need to remember data of one node in backward direction ( back\_node ).

Here we will have to separately compare for last node apart from first node for delete operation b/c next node of last node is not present it will create problem in changing refence if previous node ---Check explanation in if block of while loop.

class Node:

    def \_\_init\_\_(self,dataval=None,next\_node=None,previous\_node=None):

        self.dataval=dataval

        self.next\_node=next\_node

        self.previous\_node=previous\_node

class DLL:

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=None

    def delete\_data(self,data):

        current\_node=self.head

        back\_node=None #remember data of one node back

        #if data was at head node

        if current\_node.dataval==data:

            current\_node=current\_node.next\_node

            current\_node.previous\_node=None

            self.head=current\_node

        #if data is at end of node

        elif self.end.dataval==data:

            back\_node=self.end.previous\_node

            self.end=back\_node

            back\_node.next\_node=None

        #data not at head node, may or may not be in mid

        else:

            back\_node=current\_node

            current\_node=current\_node.next\_node

            while current\_node!=None:

                if current\_node.dataval==data: #data found in mid

                    back\_node.next\_node=current\_node.next\_node #for last node it will be None

                    current\_node=current\_node.next\_node #if we were at last Node then current\_node.next\_node will be None. That will create problem at next line

                    current\_node.previous\_node=back\_node

                    break

                else: #requried data and node data not equal, then continue with next node

                    back\_node=current\_node

                    current\_node=current\_node.next\_node

            else: #finished for checking each node but not found, hence data not present

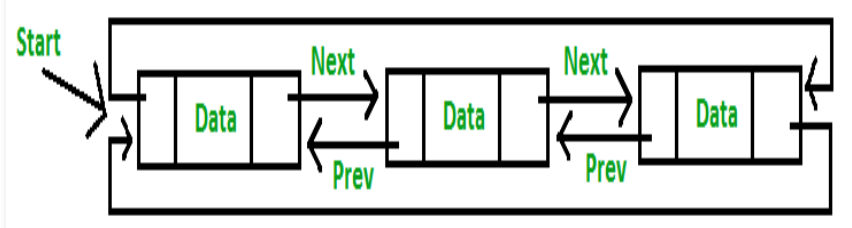
                print('data not present in node')

############################################################################## Circular Doubly linked list #

#############################################################################

Circular doubly linked list is a more complexed type of data structure in which a node contains pointers to its previous node as well as the next node.

It has characteristic of both circular and doubly linked list.



**Note:**

If we see above diagram, we can find that if DCLL have only one variable (head or tail) then we can refer any variable.

Any node can work as a head or tail end, all nodes are equally good.

If we know head then we can get tail node(head.previous\_node) easily and all node one by one.

###########################################################################

# CDLL using one variable #

###########################################################################

**Creating CDLL** --- Using one variable, more suggestable.

class Node:

    def \_\_init\_\_(self,data=None):

        self.dataval=data

        self.next\_node=None

        self.previous\_node=None

class DCLL:

    def \_\_init\_\_(self):

        self.head=None

**Adding data at tail end of CDLL ---** Using one variable, more suggestable.----Imp.

Here we will use two temporary variable (first and last) to store value of head and tail end.

class Node:

    def \_\_init\_\_(self,data=None):

        self.dataval=data

        self.next\_node=None

        self.previous\_node=None

class DCLL:

    def \_\_init\_\_(self):

        self.head=None

    def add\_end(self,data):

        node=Node(data)

        #if node is empty

        if self.head==None:

            print('cdll si empty')

            node.next\_node=node.previous\_node=node

            self.head=node

            return

        else:

            first=self.head

            last=self.head.previous\_node

            node.previous\_node=last

            node.next\_node=first

            self.head.previous\_node=node

            last.next\_node=node

**Adding data at beginning of CDLL**

This is exactly same as above just we need to update the head value also.

class Node:

    def \_\_init\_\_(self,data=None):

        self.dataval=data

        self.next\_node=None

        self.previous\_node=None

class DCLL:

    def \_\_init\_\_(self):

        self.head=None

    def add\_begin(self,data):

        node=Node(data)

        #if DCLL is emtpy

        if self.head==None:

            node.next\_node=node.previous\_node=node

            self.head=node

            return

        #if node is not emtpy

        else:

            first=self.head

            last=self.head.previous\_node

            node.previous\_node=last

            node.next\_node=first

            self.head.previous\_node=node

            last.next\_node=node

            self.head=node #update the head node value

            return

**Deleting node**

Here this is CDLL so we don’t need previous node pointer variable b/c each node have next and previous node pointer.

    def del\_data(self,data):

        current=self.head

        while current:

            if current.dataval==data:

                #if data is at head

                if current==self.head:

                    first=self.head

                    last=self.head.previous\_node

                    self.head.previous\_node.next\_node=self.head.next\_node

                    self.head.next\_node.previous\_node=self.head.previous\_node

                    self.head=self.head.next\_node #updating the head variable

#current.next\_node=current.previous\_node=None

                    return

                else:

                    current.previous\_node.next\_node=current.next\_node

                    current.next\_node.previous\_node=current.previous\_node

                    return

            current=current.next\_node

            if current==self.head:

                break

        else:

            print('data is not present in node')

**Adding data after given value:**

We will start from head node and will check each value. If item (after which new node to be added) is found, then we will add new node.

#############################################################################

# CDLL using two variables #

#############################################################################

**Creating Circular Doubly linked list**

class Node:

    def \_\_init\_\_(self,dataval=None,next\_node=None,prev\_node=None):

        self.data=dataval

        self.next\_node=next\_node

        self.prev\_node=prev\_node

class DCLL:

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=end

#insering data at start/beginning of list

    def add\_start(self,data):

        new\_node=Node(data)

        #if DCLL is empty

        if self.head==None:

            print('new was empty')

            self.head=new\_node

            self.end=new\_node

            self.head.prev\_node=new\_node

            self.head.next\_node=new\_node

            self.end.prev\_node=new\_node

            self.end.next\_node=new\_node

        else:

            first=self.head

            last=self.end

            self.head.prev\_node=new\_node

            self.head=new\_node

            self.head.next\_node=first

            self.end.next\_node=self.head

            self.head.prev\_node=self.end

#Intserting data at end of list

    def add\_end(self,data):

        new\_node=Node(data)

        #if DCLL is empty

        if self.head==None:

            print('new was empty')

            self.head=new\_node

            self.end=new\_node

            self.head.prev\_node=new\_node

            self.head.next\_node=new\_node

            self.end.prev\_node=new\_node

            self.end.next\_node=new\_node

        else:  #DCLL had come data

            print('node was not empty')

            first=self.head

            last=self.end

            last.next\_node=new\_node

            self.end=new\_node

            self.end.prev\_node=last

            self.end.next\_node=first

            self.head.prev\_node=new\_node

    def getvalue(self):

        current\_node=self.head

        while current\_node:

            print(current\_node.data)

            current\_node=current\_node.next\_node

            if current\_node==self.end:

                print(current\_node.data)

                break

        # print()

        # current\_node=self.head

        # while current\_node:

        #     print(current\_node.data)

        #     current\_node=current\_node.next\_node

        #     if current\_node==self.head:

        #         break

Getting size of linked list:

class Node:

    def \_\_init\_\_(self,dataval=None,next\_node=None,prev\_node=None):

        self.data=dataval

        self.next\_node=next\_node

        self.prev\_node=prev\_node

class DCLL:

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=end

    def getsize(self):

        size=0

        current\_node=self.head

        #if list has some data

        while current\_node:

            if current\_node==self.end:

                size+=1

                print('size is: ',size)

                return

            else:

                current\_node=current\_node.next\_node

                size=size+1

        #if list is empty

        print('size is: ',size)

**Inserting given data at given position:**

Here the starting position of any data is 1.

class Node:

    def \_\_init\_\_(self,dataval=None,next\_node=None,prev\_node=None):

        self.data=dataval

        self.next\_node=next\_node

        self.prev\_node=prev\_node

class DCLL:

    def \_\_init\_\_(self,head=None,end=None):

        self.head=head

        self.end=end

    def getsize(self):

        size=0

        current\_node=self.head

        #if list has some data

        while current\_node:

            if current\_node==self.end:

                size+=1

                print('size is: ',size)

                return size

            else:

                current\_node=current\_node.next\_node

                size=size+1

        #if list is empty

        print('size is: ',size)

        return size

    def insert\_at\_pos(self,data,position):

        new\_node=Node(data)

        initial=1

        back\_node=None

        #if inserting at first index

        if position==1:

            start=self.head

            start.prev\_node=new\_node

            self.head=new\_node

            self.head.next\_node=start

        #inseting at end

        elif position>=self.getsize():

            last=self.end

            back\_node=self.end.prev\_node

            back\_node.next\_node=new\_node

            self.end.prev\_node=new\_node

            new\_node.next\_node=self.end

            new\_node.prev\_node=back\_node

# if insert operation in not at ned or starting

        else:

            current\_node=self.head

            back\_node=current\_node

            current\_node=current\_node.next\_node

            for i in range(**2**,self.getsize()):

                if i==position:

                    new\_node.prev\_node=back\_node

                    new\_node.next\_node=current\_node

                    current\_node.prev\_node=new\_node

                    back\_node.next\_node=new\_node

                else:

                    back\_node=current\_node

                    current\_node=current\_node.next\_node

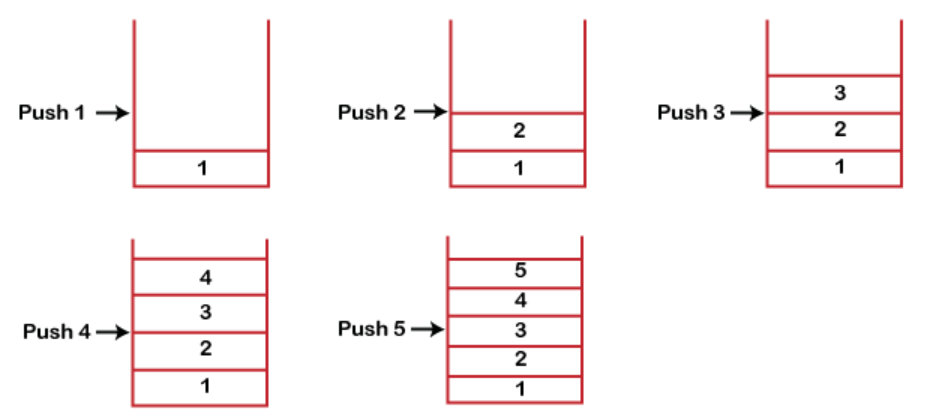
############################################################################## Stack #

#############################################################################

In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

It follows the LIFO (Last-In-First-Out) principle.

We can’t use tuple to implement stack b/c tuple are immutable.



**Some key points related to stack**

1. It is called as stack because it behaves like a real-world stack, piles of books, etc.
2. It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

**Application of Stacks**

1. Saving the web page history
2. Undoing the operation (ctrl+z) in editing application ( notepad, notepad++)
3. Evaluation of arithmetic operation

**Standard Stack Operations:**

1. **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
2. **pop():** When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
3. **isEmpty():** It determines whether the stack is empty or not.
4. **isFull():** It determines whether the stack is full or not.'
5. **peek():** It returns the element at the given position.
6. **count():** It returns the total number of elements available in a stack.
7. **change():** It changes the element at the given position.
8. **display():** It prints all the elements available in the stack.

**Stack implementation using Abstract data types-**

1. Using Linkedlist
2. Using Array

**Implementing stack using non Abstract Data types:**

There are many data types in python using which we can perform stack implementation as-

1. Using list
2. Using collections.deque

**Note:**

We can’t use tuple to implement stack b/c tuple are immutable.

For creating stack we will take advantage of list data types.

For our stack class we need only on instance variable of list data types to implement stack.

class Stack:

    def \_\_init\_\_(self):

        self.stack=[]

    def isEmpty(self):

        if len(self.stack)==0:

            return True

    def create(self,data):

        self.stack.append(data)

    def pop\_data(self):

        return self.stack.pop()

    def stack\_len(self):

        return len(self.stack)

    def push\_data(self,data):

        self.stack.append(data)

    def all\_values(self):

        for i in range(self.stack\_len()):

            print(self.stack[-1-i])

**Method 2: Implementing Stack**

We will declare list type variable as private b/c list have many other feature (inserting at end, deleting/getting data from any position) and making stack of fixed size.

class Stack:

    def \_\_init\_\_(self,max\_size):

        self.\_\_max\_size=max\_size

        self.\_\_elements=[None]\*self.\_\_max\_size

        self.\_\_top=-1

    def is\_full(self):

        if(self.\_\_top==self.\_\_max\_size-1):

            return True

        return False

    def push(self,data):

        if(self.is\_full()):

            print("The stack is full!!")

        else:

            self.\_\_top+=1

            self.\_\_elements[self.\_\_top]=data

**max\_size:** Indicates maximum number of elements in stack

**top:** Indicates index position of topmost element in stack, top=-1 means stack is empty.

|  |  |  |
| --- | --- | --- |
| When empty | When some value | When full |
| top=-1 | top=n and top<max\_size-1 | top=max\_size-1 |

**Push operation:**

Already done in above code.

**Pop operation**

|  |  |  |
| --- | --- | --- |
| When full | When have some value | When empty |
| top=max\_size-1 | top<max\_size-1 and top=n and top!=-1 | top=-1 |

class Stack:

    def \_\_init\_\_(self,max\_size):

        self.\_\_max\_size=max\_size

        self.\_\_elements=[None]\*self.\_\_max\_size

        self.\_\_top=-1

    def pop(self):

        data=self.\_\_elements[self.\_\_top]

        if self.\_\_top>=0:

            self.\_\_top=self.\_\_top-1

            print(data)

        else:

            print('stack is empty')

############################################################################

# Stack using Singly linked list #

############################################################################

We will define two methods to for push and pop operations-

1. removeHead()
2. addHead()

**Operation on Stack using Singly linked list:**

We can do all above operation if using linked list

class Node:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.next\_node=None

class Stack:

    def \_\_init\_\_(self):

        self.head=None

        pass

    # def traverse(self):

    #     current=self.head

    #     if current==None:

    #         print('Stack is emtpy')

    #     else:

    #         while current:

    #             print(current.data,end=' ')

    #             current=current.next\_node

    def push(self,data):

        #is empty

        if self.head==None:

            node=Node(data)

            self.head=node

        #add new node and make head as latest added node

        else:

            node=Node(data)

            node.next\_node=self.head

            self.head=node

    def peek(self):

        current=self.head

        if self.head==None:

            print('Stack is empty')

        else:

            return self.head.data

    def pop(self):

        if self.head==None:

            print('pop operation can not be performed on empty stack')

        else:

            current=self.head

            #there is only one node in stack

            if self.head.next\_node==None:

                self.head=None

                current.next\_node=None

                return current.data

            else:

                self.head=self.head.next\_node

                current.next\_node=None

                return current.data

############################################################################

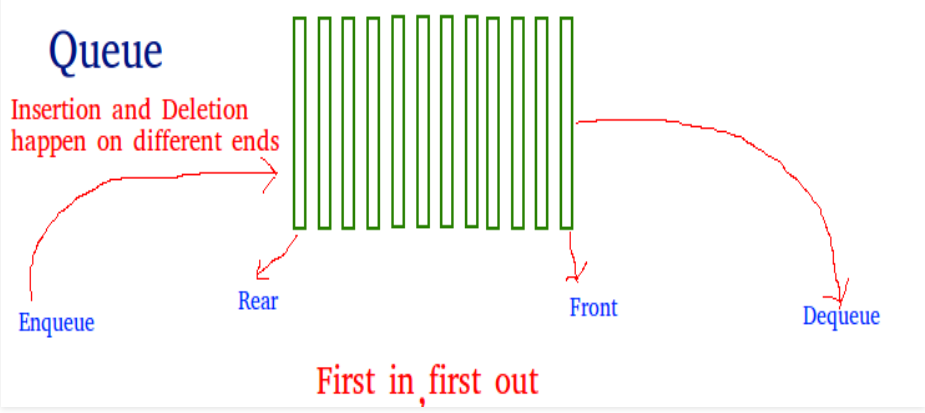
# Queue #

############################################################################

Queue is data structure in which data is stored in sequential manner i.e it implements FIFO.

Size of queue is fixed.

The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



**Application of Queue**

1. Used in printer
2. To access shared memory or resource
3. Web server responding to queue
4. Computer CPU sharing

**Types of queues**

There are mainly 4 types of queues-

1. Simple queue
2. Circular Queue
3. Priority Queue
4. Double ended queue

**Queue Operations**

**Enqueue**: Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.

**Dequeue**: Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition

**Front**: Get the front item from queue. We always get item from front end.

**Rear**: Get the last item from queue. We always insert from rear end.

**Implementing que**

We can implement queue using –

1. python list
2. using Queue class from queue module (queue.Queue)
3. Using dequeue class from collections (collections.dequeue)
4. Using abstract data types (linked list, array)

**Implementing queue using list**

class Queue:

    def \_\_init\_\_(self,size=0):

        self.item=[]

        self.size=size

    def enqueue(self,data):

        if len(self.item)==self.size:

            print('queue is full, no item can be added')

        else:

            self.item.insert(0,data)

    def getvalues(self):

        data=self.item

        print(data)

        for each in data:

            print(each)

    def dequeue(self):

        if len(self.item)>0:

            print('removed itme is: ',self.item.pop())

            print('now queue data is: ',self.item)

        else:

            print('queue is empty')

**Method 2: Implementing queue**

class Stack:

    def \_\_init\_\_(self,max\_size):

        self.\_\_max\_size=max\_size

        self.\_\_data=[]

    def push(self,data):

        if len(self.\_\_data)==self.\_\_max\_size:

            print('stack is full')

        else:

            self.\_\_data.append(data)

    def pop(self):

        if len(self.\_\_data)==0:

            print('stack is empty')

        else:

            print(self.\_\_data[-1])

            del self.\_\_data[-1]

**Implementing queue:**

Before implementing let’s take below two picture to understand 'front' and 'rear' variable.

|  |  |
| --- | --- |
| **When empty** | **When filled with value** |
|  |  |

**Note:**

* If queue is empty, then value of 'rear' will be -1 else it point to index of value.
* element is list type which contains data, initial values are null with given size.
* front is variable (integer) which always point to first value of queue
* rear is the variable (int type) which points to last value of queue in que.

class Queue:

    def \_\_init\_\_(self,max\_size):

        self.\_\_max\_size=max\_size

        self.\_\_elements=[None]\*self.\_\_max\_size #initializing element as list with None

        self.\_\_rear=-1

        self.\_\_front=0

    def is\_full(self):

        if(self.\_\_rear==self.\_\_max\_size-1):

            return True

        return False

    def enqueue(self,data):

        if(self.is\_full()):

            print("Queue is full!!!")

        else:

            self.\_\_rear+=1

            self.\_\_elements[self.\_\_rear]=data

    def get\_data(self):

        curent=self.\_\_front

        while curent<=self.\_\_rear:

            print(self.\_\_elements[curent])

            curent=curent+1

|  |  |
| --- | --- |
| **When empty** | **When filled with value** |
|  |  |

**Dequeue operation:**

Deleting data from queue in FIFO order is called queue operation.

Let’s take below screenshot to understand the dequeue operation-

|  |  |  |
| --- | --- | --- |
| When queue is full | When not empty and not full | When empty |
| front=0 and front<=rear | front=n and front<=rear | front>rear |

Note:

For dequeue operation we will change the value to front variable.

class Queue:

    def \_\_init\_\_(self,max\_size):

        self.\_\_max\_size=max\_size

        self.\_\_elements=[None]\*self.\_\_max\_size

        self.\_\_rear=-1

        self.\_\_front=0

    def dequeue(self):

        start\_index=self.\_\_front

        if start\_index<self.\_\_rear:

            self.\_\_front=self.\_\_front+1

        else:

            print('queue is empty')

**#######################################**

**# Implementing queue using singly linked list #**

**#######################################**

We are using singly linkedlist to implement queue so Node class will have two instance variable --- a) data b) next node pointer

In Queue class we will have two instance variable – a) front b) rear to identify front and rear end.

In singly linked we have same variable naming convention.

class Node:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.next\_node=None

class Que:

    def \_\_init\_\_(self):

        self.front=None

        self.rear=None

    def enque(self,data):

        node=Node(data)

        #if que is empty

        if self.front==None and self.rear==None:

            self.front=self.rear=node

        else:

            self.rear.next\_node=node

            self.rear=node

    def deque(self):

        #check is que is empty

        if self.rear==None and self.front==None:

            print('que is emtpy')

        else:

            current=self.front

            self.front=current.next\_node

            current.next\_node=None

            #que is empty so update the rear variable else not required

            if self.front==None:

                self.rear=None

                return current.data

            return current.data

q = Que()

**Time complexity:**

Time complexity of enqueue or dequeue operation is O(1) b/c as we only change few pointers in both operations. There is no loop in any of the operations.

#########################

# queue module in python #

#########################

The queue module implements multi-producer, multi-consumer queues.

The Queue class in this module implements all the required locking semantics.

It has below classes which can be used to implement different version of quque data structure.

1. class queue.Queue(maxsize=0)

Constructor for a FIFO queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached.

1. class queue.LifoQueue(maxsize=0)

Constructor for a LIFO queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached. (like stack)

1. class queue.PriorityQueue(maxsize=0)

Constructor for a priority queue. maxsize is an integer that sets the upperbound limit on the number of items that can be placed in the queue. Insertion will block once this size has been reached.

1. class queue.SimpleQueue

Constructor for an unbounded FIFO queue.

Simple queues lack advanced functionality such as task tracking.

**Queue objects/queue instance methods**

It has below instance methods

1. **que\_instance.put(item, block=True, timeout=None)**

Put item into the queue.

1. **que\_instance.get(block=True, timeout=None)**

Remove and return an item from the queue.

1. **que\_instance.qsize()** ------It return the size of que
2. **que\_instance.empty()** ------It returns True if queue is True else False

**Implementing queue using queue.Queue**

Just use Queue class and it’s methods from queue.Queue.

q=Queue(maxsize=4)

print("Initial Size Before Insertion:",q.qsize())

q.put('A')

q.put('AA')

q.put('AAA')

q.put('AAAA')

###########################

# Priority Queue #

###########################

Priority Queue is an extension of queue with following properties.

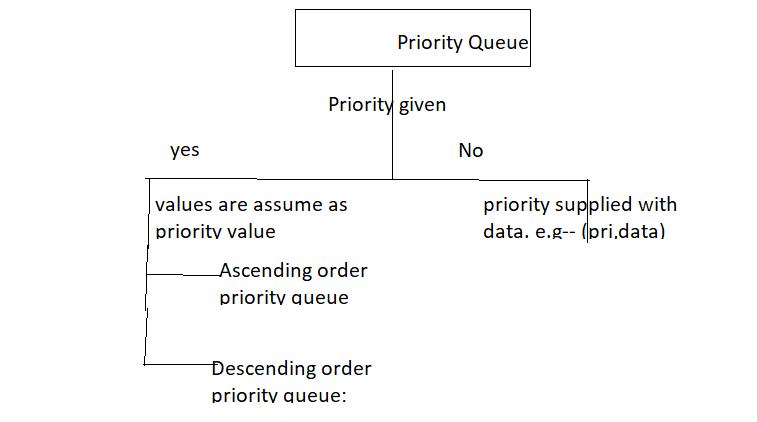
The priority of the elements in a priority queue determines the order in which elements are removed from the priority queue

Using priority queue data can be arranged in ascending or descending order.

**Properties of Priority Queue**

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority
3. Eelement whose priority is supposed to be high will be added at front.
4. If two elements have the same priority, they are served according to their order in the queue.

Priority queue can be further classified as below-



**Implementing priority Que**

We can implement priority key using many data types/structure –

1. Using List
2. Using queue.Priority Queue
3. Using heap

**#############################################**

**# Implementing Queue if data and priority are given #**

**#############################################**

**Implementing priority queue using List when data and priority given**

Here we will assume the data with priority is given in below format –

(priority\_value, value\_data)

class PriQue:

    def \_\_init\_\_(self):

        self.item=[]

        pass

    def enquiue(self,priority=0,data=0):

        self.item.append((priority,data))

        print(self.item)

        data=self.item

        data.sort(reverse=True) #sorts data in descending order of priority

        #print(self.item)  #here data will be in priority order

        #Below three lines can also be used instead of osrt function

        """

        sorted\_data=self.item

        sorted\_data=sorted(sorted\_data,key=lambda x:x[0],reverse=True)

        print(sorted\_data)

        """

**Dequeue operation of priority queue implemented on list (priority and data were given as i/p)**

We have defined here dequeue function which will give data.

**Note:**

1. Here all data will come at a time but the data will come in descending order or priority.
2. We can customize our logic to get one data at time and ascending/descending order.

class PriQue:

    def \_\_init\_\_(self):

        self.item=[]

        pass

    def enquiue(self,priority=0,data=0):

        self.item.append((priority,data))

        data=self.item

        data.sort(reverse=True)

        #print(self.item)  #here data will be in priority order

        #Below three lines can also be used instead of osrt function

        """

        sorted\_data=self.item

        sorted\_data=sorted(sorted\_data,key=lambda x:x[0],reverse=True)

        print(sorted\_data)

        """

    def dequeue(self):

        data=sorted(self.item,key=lambda x:x[0],reverse=True)

        for each in data:

            print(each)

**Get size of priority queue**

Do it by yourself.

**Note:**

1. If priorityQueue is implementing as descending priority que then front element will be with highest priority.
2. If priorityQueue is implementing as ascending priority que then front element will be with lowest priority.

**PriorityQueue using queue.PriorityQueue**

q = PriorityQueue() #Creating instance of PriorityQueue

# insert into queue

q.put((2, 'g')) #adding data, first argument is priority value

q.put((3, 'e'))

q.put((4, 'k'))

q.put((5, 's'))

q.put((1, 'e'))

print(q.get()) #(1, 'e')

print(q.get()) #(2, 'g')

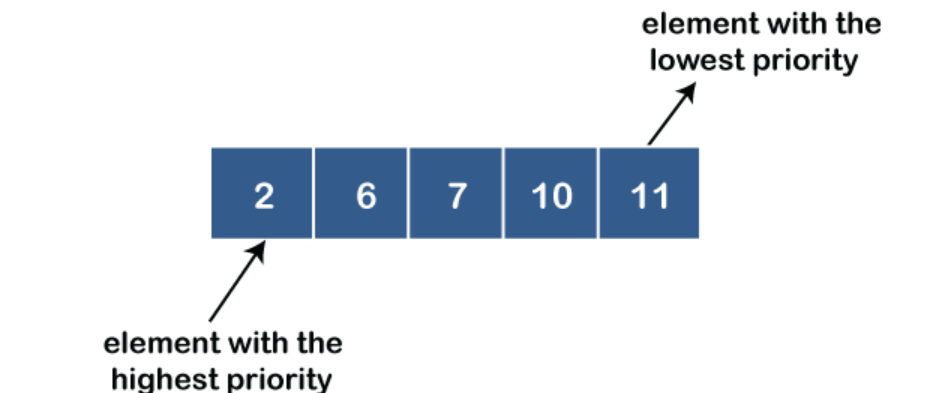
**#####################################**

**# Implementing Queue if only data is given #**

**#####################################**

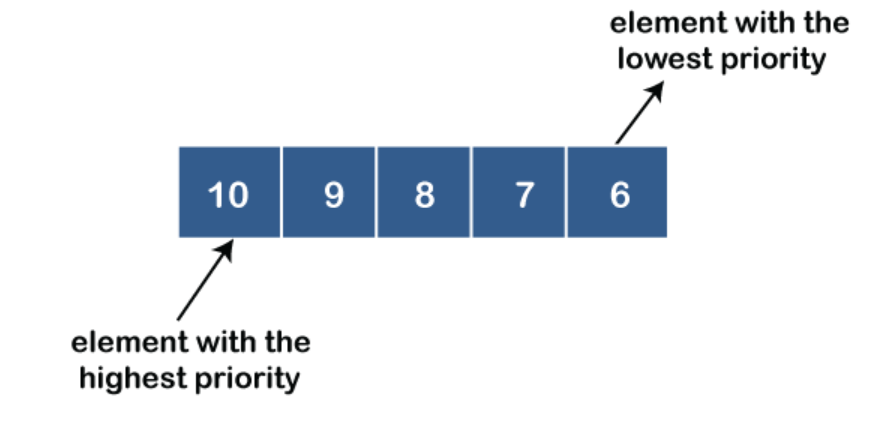
**Ascending order priority queue:**

Data are stored in ascending order and data with lowest value is supposed to have highest priority or specify the priority for each element and arrange accordingly.



**Descending order priority queue:**

Data are stored in descending order and data with highest value is supposed to have highest priority specify the priority for each element and arrange accordingly.



**#######################**

**# Double ended queue #**

**#######################**

Queue in which data can be pushed or popped from both end is called double ended queue.

############################################################################

# Tree Data Structure #

############################################################################

**What are Trees?**

A tree is a Hierarchical data structure that stores the information/data/nodes in hierarchy form. The Tree data structure is one of the most efficient and mature.

The nodes connected by the edges are represented.

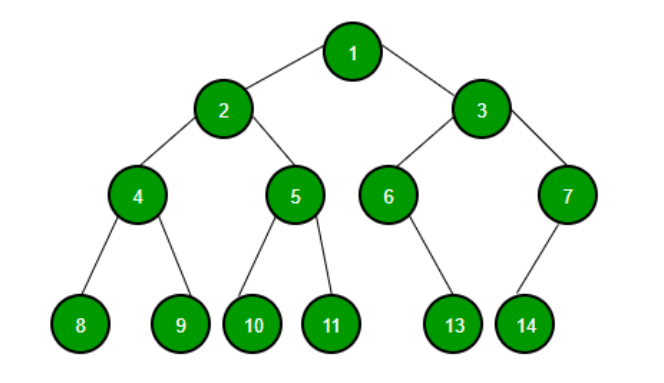
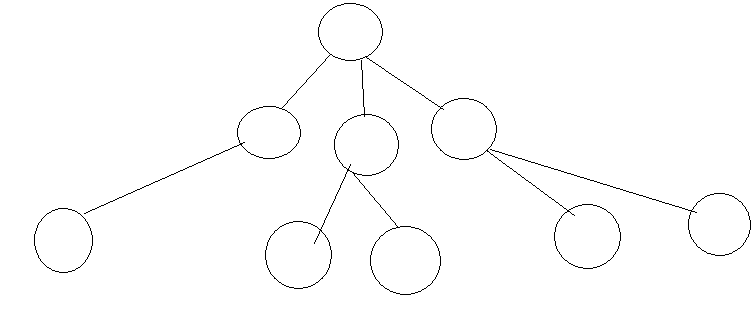
**Properties of Tree**

1. Every tree has a specific root node.
2. A root node can cross each tree node. It is called root, as the tree was the only root.
3. Every child has only one parent, but the parent can have many children.

**Types of Tree**

There are many types of trees available, few of them are-

1. Binary Tree – can have max two child node/branches/tree
2. General Tree ---- can have any number of parent/node/trees
3. Binary Search Tree --- derived from binary tree
4. RBT tree
5. AVL tree

Binary tree Normal Tree

**Commonly used terms in Tree:**

**Root**

The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent.

**Child node**

If the node is a descendant of any node, then the node is known as a child node.

**Parent**:

If the node contains any sub-node, then that node is said to be the parent of that sub-node.

**Sibling**:

The nodes that have the same parent are known as siblings.

**Leaf Node/External nodes**

The node of the tree, which doesn't have any child node, is called a leaf node.

A leaf node is the bottom-most node of the tree.

There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

**Edges:**

Links through which each node is connected. Generally, we use line/arrow.

Number of edges=n-1 , n=Number of nodes

**Internal nodes/Non leaf nodes**

A node has at least one child node known as an internal

**Ancestor node:**

An ancestor of a node is any predecessor node on a path from the root to that node.

The root node doesn't have any ancestors.

**Descendant**

The immediate successor of the given node is known as a descendant of a node.

**Level and Height**

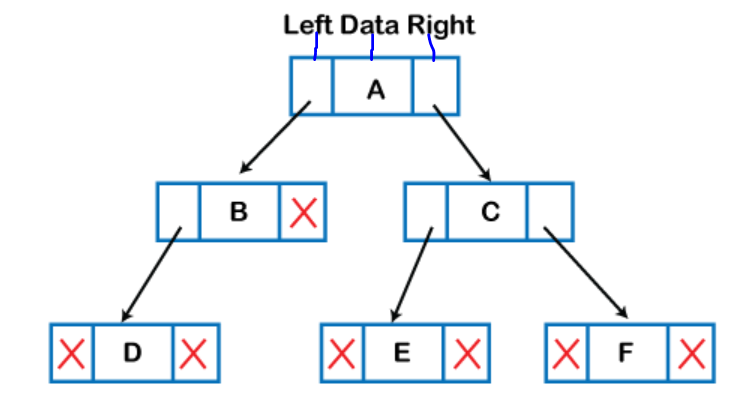
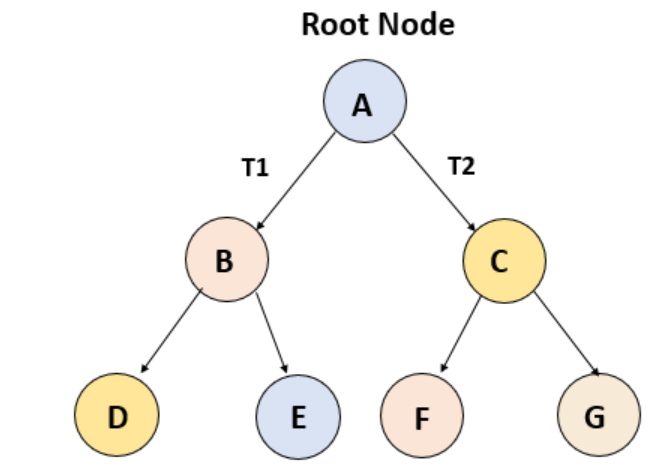
Level/Depth --- Level of a node is defined as number of edges from root node to that node.

Height --- Height of a node is defined as number of nodes above that node.

**Binary Tree**

The binary tree is the kind of tree in which most two children can be found for each parent. The kids are known as the left kid and right kid.

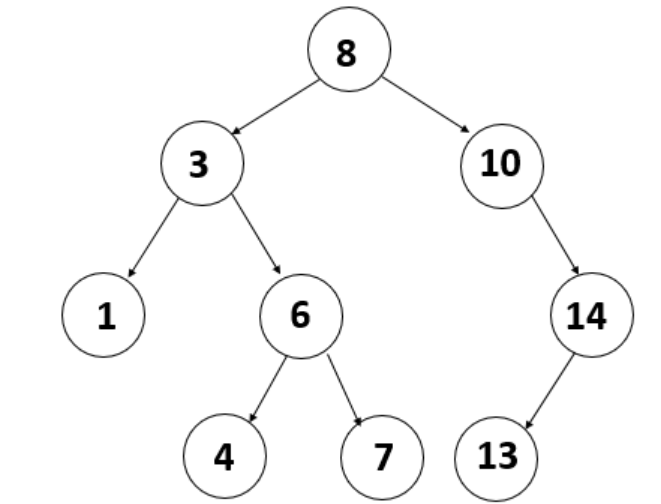
When certain constraints and characteristics are applied in a Binary tree, number of others such as AVL tree, BST (Binary Search Tree), RBT tree, etc.



**Binary Search Tree (BST)**

Binary Search Tree (BST) is a binary tree extension with several optional restrictions---

1. The left child value of a node in BST should be less than the parent value.
2. The right child value should always be greater than the parent’s value.
3. No duplicate data are allowed in BST



**Advantages/uses of Tree**

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy.
2. If we organize keys in form of a tree (with some ordering e.g., BST), we can search for a given key in moderate time (quicker than Linked List and slower than arrays).
3. We can insert/delete keys in moderate time (quicker than Arrays and slower than Unordered Linked Lists).

**Application of Tree**

Some of the applications of trees are:

* **Filesystems** —files inside folders that are in turn inside other folders.
* **Comments on social media** — comments, replies to comments, replies to replies etc form a tree representation.
* **Family trees** — parents, grandparents, children, and grandchildren etc that represents the family hierarchy.

################################

# Implementing Normal Tree #

################################

<https://www.youtube.com/watch?v=4r_XR9fUPhQ>

Here we are going to implement a normal tree. We can notice below points for normal tree-

1. Any node of tree contain/have following info – a.) data b.) parent node c.) children node
2. A normal tree can have any number of child node.
3. For root node there is no parent node i.e. parent of root node = None

We will use above two points while implementing the tree.

**Logic for node creation**

1. We will create root node first
2. If any node is called on any instance of TreeNode then that TreeNode instance will work as parent node for that node.

**Implementation code**

class TreeNode:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.children=[]

        self.parent=None #for root node, it will be None

    def add\_child(self,child):

        child.parent=self #create parent fir current node

        self.children.append(child)

    def getdata(self): #this is for getting all child tree from called instance tree

        space=' '\*self.get\_level()\*3

        print(space+self.data)

        for child in self.children:

            child.getdata()

def get\_level(self):#find level of any node,root node=0,child of root=1,grandchild of root=2.

        level=0

        p=self.parent

        while p:

            level+=1

            p=p.parent

        return level

get\_data() --- this is instance methods and gives all data from node on which this method is called to bottom node or last node

get\_level() --- this is instance methods, it gives level value of instance on which it is called. Level for root node=0, child of root=1, grandchild of root=2, great grandchild of root=2 ..

**Binary Tree Implementation:**

Binary tree is tree in which any node can have max 2 child nodes in it.

Code-

import sys

class TreeNode:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.children=[]

        self.parent=None #for root node, it will be None

    def add\_child(self,child):

        child.parent=self #create parent fir current node

        if len(self.children)>=2: #Terminate the program with message

            try:

                raise Exception('there can not be more than 2 children for any node')

            except Exception as e:

                print(e)

                sys.exit(1)

        else:

            self.children.append(child)

    def getdata(self):

        space=' '\*self.get\_level()\*3

        print(space+self.data)

        for child in self.children:

            child.getdata()

    def get\_level(self):

        level=0

        p=self.parent

        while p:

            level+=1

            p=p.parent

        return level

**Note:**

1. If we try to create more than 2 child node for any node then it terminates and no further execution happens.
2. We can customize our code to delete any child node or skip to adding any node in case if we want to add/create more than 2 child node.

**Searching data in Tree:**

Here we have to define this for normal tree i.e there can be any number of child node (say- left and right) for each node.

Our below code is applicable for search operation for binary/normal tree.

class TreeNode:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.children=[]

        self.parent=None #for root node, it will be None

    def add\_child(self,child):

        child.parent=self #create parent fir current node

        self.children.append(child)

    def getdata(self): #this is for getting all child tree from called instance tree

        space=' '\*self.get\_level()\*3

        print(space+self.data)

        for child in self.children:

            child.getdata()

    def get\_level(self):#find level of any node,root node=0,child of root=1,grandchild of root=2.

            level=0

            p=self.parent

            while p:

                level+=1

                p=p.parent

            return level

    def search\_data(self,search\_data):

        if self.data==search\_data:

            print('data found')

            return True

        else:

            for each in self.children:

                res=each.search\_data(search\_data)

                if res==True: #True means data found no more recursion required

                    return True

**Get elements of normal tree in list form**

    def get\_items(self,elements=[]):

        if self.data:

            elements.append(self.data)

        for each in self.children:

            each.get\_items(elements)

        #elements is list type

        return elements

**Calculating size of normal Tree**

Get the elements in list form and return length that list.

    def get\_items(self,elements=[]):

        if self.data:

            elements.append(self.data)

        for each in self.children:

            each.get\_items(elements)

        #elements is list type

        return len(elements)

**Calculating size of Binary Tree**

Here this method is applicable only for binary tree.

    def get\_using\_lr(self,size=1):

        if len(self.children)==0: #no child node, means last node, return the size value

            return size

        else:                  # child node are there

            if len(self.children)==1:

                return self.children[0].get\_using\_lr(size)+1 #caculate size of child+1(for current node)

            if len(self.children)==2:

                return self.children[0].get\_using\_lr(size)+1+self.children[1].get\_using\_lr(size)#caculate size of child+1(for current node)

**Logic:**

1. First check if child node is present, if present then call same method recursively for each child.
2. If no child nodes are not present, then return the size value.

**Deleting a node from binary tree**

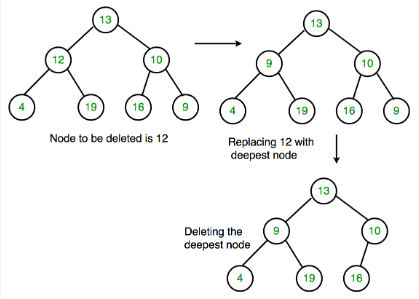
Given a binary tree, delete a node from it by making sure that tree shrinks from the bottom (i.e. the deleted node is replaced by the bottom-most and rightmost node). This is different from BST deletion. Here we do not have any order among elements, so we replace with the last element.

**Algorithm**

1. Starting at the root, find the deepest and rightmost node in binary tree and node which we want to delete.

2. Replace the deepest rightmost node’s data with the node to be deleted.

3. Then delete the deepest rightmost node.



# function to delete the given deepest node (d\_node) in binary tree

def deleteDeepest(root,d\_node):

    q = []

    q.append(root)

    while(len(q)):

        temp = q.pop(0)

        if temp is d\_node:

            temp = None

            return

        if temp.right:

            if temp.right is d\_node:

                temp.right = None

                return

            else:

                q.append(temp.right)

        if temp.left:

            if temp.left is d\_node:

                temp.left = None

                return

            else:

                q.append(temp.left)

# function to delete element in binary tree

def deletion(root, key):

    if root == None :

        return None

    if root.left == None and root.right == None:

        if root.key == key :

            return None

        else :

            return root

    key\_node = None

    q = []

    q.append(root)

    temp = None

    while(len(q)):

        temp = q.pop(0)

        if temp.data == key:

            key\_node = temp

        if temp.left:

            q.append(temp.left)

        if temp.right:

            q.append(temp.right)

    if key\_node :

        x = temp.data

        deleteDeepest(root,temp)

        key\_node.data = x

    return root

**Calculating size of normal tree**

This approach is applicable to normal tree as well as binary tree for calculating size.

We have used global variable (g\_lbl\_size) for calculating size. In this variable we will add 1 for each node/leaf.

We have used size instance variable which will receive value of global variable (g\_lbl\_size) for each iteration and will add 1 to increase the value of global variable.

g\_lbl\_size=0 #this is global variable used for size calculation

import sys

class TreeNode:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.children=[]

        self.parent=None #for root node, it will be None

    def add\_child(self,child):

        child.parent=self

        self.children.append(child)

    def get\_using\_lr(self,size=0): #size receives value of g\_lbl\_size for each iteration

        if len(self.children)==0: #no child node, means last node, return

            globals()['g\_lbl\_size']=size+1 #add 1 in g\_lbl\_size

            return

        else:                  # child node are there

            globals()['g\_lbl\_size']=size+1 #add 1 in g\_lbl\_size

            for each in range(len(self.children)):

                self.children[each].get\_using\_lr(g\_lbl\_size) #caculate size of child+1(for current node)

        return g\_lbl\_size

**Note/Logic:**

For each node add +1 in g\_lbl\_size.

For each child node call the same method recursively.

**###################################**

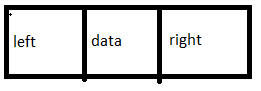
**# Binary Search Tree #**

**###################################**

Binary search tree is kind of binary tree with below properties-

1. The left subtree of a node contains only nodes with keys lesser than the node’s key.
2. The right subtree of a node contains only nodes with keys greater than the node’s key.
3. The left and right subtree must also be a binary search tree.
4. There must be no duplicate nodes.

Diagram

Description automatically generated 

**Advantage of binary search tree**

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure compared to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes O(log2n) time. In worst case, the time it takes to search an element is 0(n).
3. It also speeds up the insertion and deletion operations as compared to that in array and linked list.

**Creating binary Search tree:**

In binary search tree or binary tree there are only two child nodes or leaf nodes allowed so we will use left and right as instance variable instead of parent and children.

**Note:**

By carefully observing above figure we can conclude below point:-

1. Each Node will have three instance variable –

a) Left node pointer

b) right node pointer

c) data of that node

1. Left node pointer points to left node of current node and same way right also.

*class BST*:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.left=None

        self.right=None

    def insert(self,value):

        if self.data==None:

            BST(value)

            return

""""

        #if duplicate

        if value==self.data:

            print('duplicates not allowed in BST')

            return

        """"

        if value<self.data:

            if self.left:

                return self.left.insert(value)

            #inster to left

            else:

                self.left=BST(value)

return

        if value>self.data:

            if self.right:

                return self.right.insert(value)

            #insert to right

            else:

                self.right=BST(value)

                return

s=BST(45)

l=[15, 79, 90, 10, 55, 12, 20, 50]

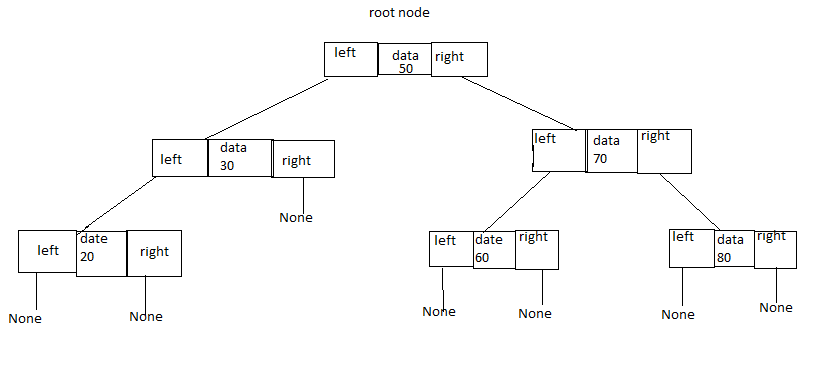
for each in l:

    s.insert(each)

print(s.inorder())

**Logic:**

Check the value that we want to inset on left or right recursively and then decide it needs to be added on left or right.



**###########**

**# Method 2# -------- Without recursion**

**###########**

    def add\_child(self,data):

        node=Node(data)

        if self.root==None:

            self.root=node

            return

        current=self

        while True:

            if data<current.data:

                if current.left:

                    current=current.left

                else:

                    current.left=node

                    return

            elif data>current.data:

                if current.right:

                    current=current.right

                else:

                    current.right=node

                    return

            else:

                print('duplicate data not allowed in BST')

s=BST(45)

l=[15, 79, 90, 10, 55, 12, 20, 50]

for each in l:

    s.add\_child(each)

print(s.inorder())

**Traversing/getting value from binary search tree:**

There are three types of traversing in BST/BT-

1. In Order traversing ------------ #left --> root ----> right
2. Pre Order traversing ---------- # root ----> left ----> right
3. Post order traversing---------- # right -----> root ------->left

**In order Traversing for DFS**

In this traversing technique we start from left most node then come back to root node and goes to right node also **called as (LNR)**

def in\_order\_traversal(self):

        elements = []

        if self.left:

            elements += self.left.in\_order\_traversal()

        elements.append(self.data)

        if self.right:

            elements += self.right.in\_order\_traversal()

        return elements

If traversing is in order traversing, then resultant values are in ascending order.

**Pre Order Traversing of DFS**

#root node---->left---->right

Method 1: ---- Using recursion

def pre\_order(self):

        elements=[]

        elements.append(self.data)

        if self.left:

            elements += self.left.pre\_order()

        if self.right:

            elements += self.right.pre\_order()

        return elements

**Method 2: ----- Using stack**

Algo-

Following is a simple stack based iterative process to print Preorder traversal.

* Create an empty stack nodeStack and push root node to stack.
* Do the following while nodeStack is not empty.

1. Pop an item from the stack and print it.
2. Push right child of a popped item to stack
3. Push left child of a popped item to stack

def iterativePreorder(self,root):

    if root is None:

        return

    nodeStack = []

    result=[]

    nodeStack.append(root)

    while(len(nodeStack) > 0):

        # Pop the top item from stack and print it

        node = nodeStack.pop()

        result.append(node.data)

        # Push right and left children of the popped node

        # to stack

        if node.right:

            nodeStack.append(node.right)

        if node.left:

            nodeStack.append(node.left)

**Post Order traversing of DFS**

#right node ---> root node -----> left node

def post\_order(self):

        elements=[]

        count=0

        if self.right:

            elements += self.right.pre\_order()

        elements.append(self.data)

        if self.left:

            elements += self.left.pre\_order()

In post order data doesn’t comes in descending order but in in\_order traversing it comes in ascending order.

**Method 2:- Using stack**

def iterativePreorder(root):

    stack=[]

    result=[]

    if root==None:

        return root

    else:

        stack.append(root)

        while stack:

            #pop item from stack and append to result

            node=stack.pop()

            #if left node exits of popped item then add in stack

            result.append(node.val)

            if node.left:

                stack.append(node.left)

            #if left node exits of popped item then add in stack

            if node.right:

                stack.append(node.right)

        #reverse the result and return it

        return result[::-1]

**In order transversal without recursion -----** Applicable for normal binary tree

Using **Stack** (list of LIFO behavior) is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack.

We will us below loop for inorder transversal without using recursion-

1) Create an empty stack S.

2) Initialize current node as root

3) Push the current node to S and set current = current->left until current is NULL

4) If current is NULL and stack is not empty then

a) Pop the top item from stack.

b) Print the popped item, set current = popped\_item->right

c) Go to step 3.

5) If current is NULL and stack is empty then we are done.

    def in\_order\_no\_loop(self):

        stack=[]

        current=self

        result=[] #Use it if want to return output

        while True:

            #if current node is not None

            if current!=None:

                stack.append(current)

                current=current.left

            #if current node is None and stack is not empty

            elif(stack):

                current=stack.pop()

                print(current.data,end=',')

                #result.append(current.data) #In case want to return result

                current=current.right

            #if current node is None and stack is empty

            else:

                break

        #Uncomment below two lines if want to retunr

        # print()

        # return result

**#Level Order Traversal #**

In level order traversal we get the value from tree from each level.

Algo:

Start from root node and create variable for current level node in list form and child level node down that level.

Get the all value from current level node in result list and append the child node of all current level into int child level node and swap the value of child level node and current level node.

Keep doing this operation till current level node list have some values.

Method 1:

    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:

        if root is None:

            return []

        queue=[root]

        next\_queue=[]

        result=[]

        while queue!=[]:

            temp=[]

            for each in queue:

                if each!=None:

                    temp.append(each.val)

                    #if left node exitst

                    if root.left is not None:

                        next\_queue.append(each.left)

                    #if right node exists

                    if root.right is not None:

                        next\_queue.append(each.right)

            #append temp to result if temp is not empty

            if temp!=[]:

                result.append(temp)

            #swap the queue and next\_queue

            queue=next\_queue

            #make next\_queue empty

            next\_queue=[]

        return result

Method 2: ---- Using helper function

def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:

        result=[]

        def helper(nodes):

            data=[]

            child\_nodes=[]

            #if node have no data then return

            if nodes==None or nodes==[]:

                return

            for each in nodes:

                #append the value of each node in data

                data.append(each.val)

                #append the left node of current node in child\_node

                if each.left:

                    child\_nodes.append(each.left)

                #append the right node of current node in child\_node

                if each.right:

                    child\_nodes.append(each.right)

            #append the data in result

            result.append(data)

            #Call the helper function with child\_nodes

            helper(child\_nodes)

        if root==None:

            return []

        #callign helpder funtion bu supplying current node in list form

        helper([root])

        return result

**Method 3: ---- Very easy**

class Solution:

    #Function to find the height of a binary tree.

    def level\_order\_traversal(self, root):

        # code here

        stack=[root]

        temp\_stack=[]

        #result to store the value at for each level

        result=[]

        while stack:

            for each in stack:

                #append the current node data in result

                result.append(each.data)

                #is left node exitst of current node then append in temp\_stack

                if each.left:

                    temp\_stack.append(each.left)

                #is right node exitst of current node then append in temp\_stack

                if each.right:

                    temp\_stack.append(each.right)

            #assign the value of temp\_stack into stack

            stack=temp\_stack

            #empty the tmep\_stack

            temp\_stack=[]

        return result

**Checking if given Tree is BST**

* Each node can have maximum of two child nodes.
* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree must also be a binary search tree.
* There must be no duplicate nodes.

If inorder traversal of a binary tree is sorted, then the binary tree is BST.

**Algorithm**:

* We will select the range for each node data, for root node we will assume that data can be anything from -ve infinity to +ve infinity.
* For child node we will update the value range.

**let say left child**— it value can be anything, less than parent node to -ve infinity or we can say as: -ve\_infinity<child data< parent\_data

**let say for right child**, we can conclude it to: parent\_data<right child<+ve\_infinity

Method1: --- By checking data with left anf right range

    def isBST(self):

        def valid(node,left, right):

            if not node:

                return True

            if not (node.data<right and node.data>left):

                return False

            return (valid(node.left, left,node.data) and valid(node.right, node.data,right))

        return valid(self,float("-inf"),float("inf"))

Method 2: ---- Using BST property (in order traversal is in ascending order and no duplicate)

def helper(root):

            element=[]

            if root.left:

                element+=helper(root.left)

            element.append(root.val)

            if root.right:

                element+=helper(root.right)

            return element

        data=helper(root)

        sorted\_data=sorted(data)

        if data==sorted\_data and len(sorted\_data)==len(set(data)):

            return True

        return False

**Searching data in BST/BT:**

We can search data in BST/BT in two ways:

1. **Get all value from tree(as list) then check searching values exist in that list**----do yourself
2. **Checking value directly in tree** ------ check below

**Searching data in BST (directly in tree) -----** Only for BST

1. Here we will check if current node data is equal to value searching, if yes then return True.
2. If above steps fail then check if left or right node exists and then check in left or right node.
3. Keep following above steps recursively, checking in left or right node will be decided based on current node value and searching value.

**###########**

**## Method # ------- Using recursion**

**##########**

def search(self,value):

        if self.data==value:

            return True

        #check in left node

        if value<self.data:

            if self.left:

                return self.left.search(value)

            else:

                return False

        #check in right node

        if value>self.data:

            if self.right:

                return self.right.search(value)

            else:

                return False

**############**

**# Method 1 # -------- Without recursion**

**###########**

1. For searching data, we will start from root node/given node.
2. Based on value of current node and searching value, update the current node data.
3. Keep on searching till we find current==None.

    def seach\_no\_loop(self,value):

        current=self

        #check we reached till end

        while current!=None:

            #If value < current node data then update current by left of current node

            if value<self.data:

                current=current.left

            #If value > current node data then update current by right of current node

            if value>self.data:

                current=current.right

            #We foud the seaching item

            else:

                print('found in tree')

                return True

        else:

            return False

**Size of BST without recursion** -------- Application for only BST

**Method 1: --- Only for BST**

    def check\_len(self):

        #this method will get the data from tree

        def get\_data(self,elements=[]):

            if self.left:

                get\_data(self.left)

            elements.append(self.data)

            if self.right:

                if self.right:

                    get\_data(self.right)

            return elements

        #call the get\_data() and return the lenght

        return len(get\_data(self))

**Size of BST/BT**

Below method is applicable for binary tree and binary search tree both.

We will use recursive approach to fins the size of tree.

    def getsize(self):

        size=0

        #tree is empty, just eixt from method

        if self.data==None:

            return

        #node have some data, increase size

        if self.data:

            size+=1

        if self.left:

            size+=self.left.getsize() #calling recursively

        if self.right:

            size+=self.right.getsize() #calling recursively

        return size

**Finding height of BST/BT:**

Level/Depth --- Level of a node is defined as number of edges from root node to that node.

Height --- Height of a node is defined as number of nodes above that node.

This is applicable for binary tree or binary search tree. There are two approaches for finding depth of tree-

1. Depth first approach
2. Breach first approach
3. Using level order traversal

**Method 1: ---- Using level order traversal approach ---- Very easy**

class Solution:

    #Function to find the height of a binary tree.

    def height(self, root):

        # code here

        stack=[root]

        temp\_stack=[]

        height=0

        while stack:

            height+=1

            for each in stack:

                if each.left:

                    temp\_stack.append(each.left)

                if each.right:

                    temp\_stack.append(each.right)

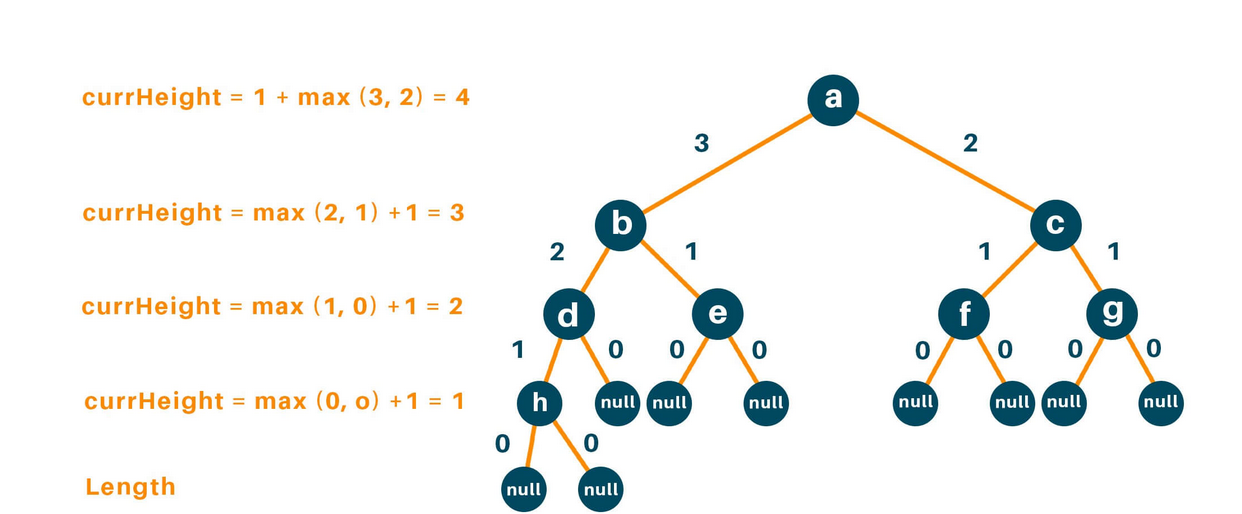
            stack=temp\_stack

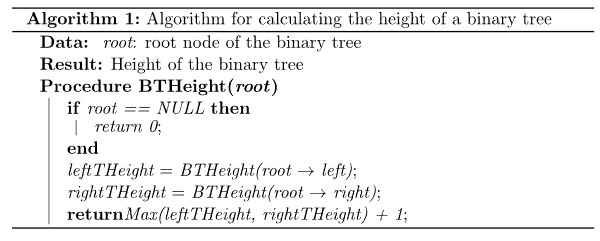
            temp\_stack=[]

        return height

**Depth using depth first approach**

In this approach we first visit/initialize to lowest node a then then keep on going upside





Note:

We have used/defined static method not instance method.

import collections

class BST:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.left=None

        self.right=None

    def insert(self,value):

        if self.data==None:

            BST(value)

            return

        if value<self.data:

            if self.left:

                return self.left.insert(value)

            #inster to left

            else:

                self.left=BST(value)

        if value>self.data:

            if self.right:

                return self.right.insert(value)

            #insert to right

            else:

                self.right=BST(value)

                return

def height(root):

    # Check if the binary tree is empty, for bottom most node(left or right side) it will run

    if root is None:

        # If TRUE return 0

        return 0

    # Recursively call height of each node

    leftAns = height(root.left)

    rightAns = height(root.right)

    # Return max(leftHeight, rightHeight) at each iteration

    return max(leftAns, rightAns) + 1

**###########**

**# Method 2 #** ----- Using instance method

**###########**

    def maxDepth(self):

        self.lDepth=0

        self.rDepth=0

        if self.data is None:

            return 0

        else :

            # Compute the depth of each subtree

            if self.left:

                self.lDepth = self.left.maxDepth()

            if self.right:

                self.rDepth = self.right.maxDepth()

            # Use the larger one

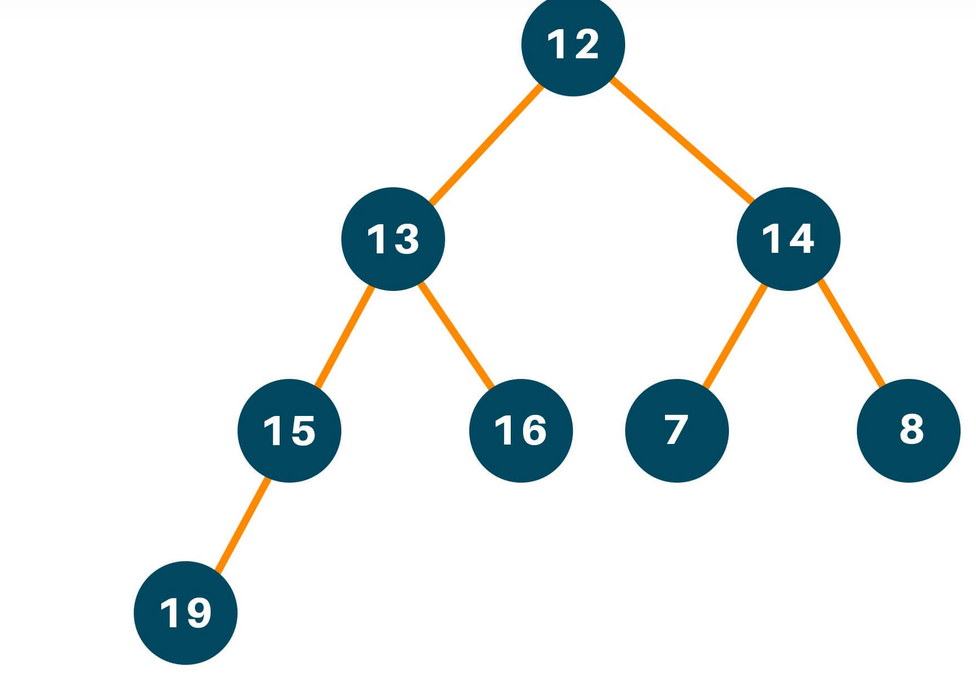
            if (self.lDepth > self.rDepth):

                return self.lDepth+1

            else:

                return self.rDepth+1

**Breadth first approach**



For the leaf node 19, the number of nodes along the edges is 4.

For the leaf node 16, the number of nodes along the edges is 3.

For the leaf node 7, the number of nodes along the edges is 3.

For the leaf node 8, the number of nodes along the edges is 3.

The maximum number of node from root to farthest leaf is: max(4, 3, 3, 3) is 4. **Hence, height of the binary tree is 4.**

Step1:

Use a queue data structure to approach this problem statement, hence, initialize an empty queue data structure.

Step2:

Enqueue root node to the queue and process it in a while loop until there are no elements left and perform the same process for the other nodes, ie.

Step3:

Run a while loop until currSize = 0, and till then keep dequeuing elements and after processing the elements when the value of currSize = 0, increment the value of ans

Therefore, dequeue 12, and check for its left child which is 13 and the right child which is 14, and enqueue them.

Now:

currSize = 0

currNode = 12

Since, currSize = 0

ans = 1

At next iteration currSize = 2

Dequeue 13 and repeat **steps 2 and 3**

**Now:**

currSize = 1

currNode = 13

Again, dequeue 14 and repeat **steps 2 and 3**

**Now:**

currSize = 0

currNode = 14

Since, currSize = 0

ans = 2

Note:

These is static method, not instance methods.

import collections

class BST:

    def \_\_init\_\_(self,data=None):

        self.data=data

        self.left=None

        self.right=None

    def insert(self,value):

        if self.data==None:

            BST(value)

            return

        if value<self.data:

            if self.left:

                return self.left.insert(value)

            #inster to left

            else:

                self.left=BST(value)

        if value>self.data:

            if self.right:

                return self.right.insert(value)

            #insert to right

            else:

                self.right=BST(value)

                return

def finddepth(root):

    # Set result variable to 0

    ans = 0

    # Initialise the queue

    queue = collections.deque()

    # Check if the tree has no nodes

    if root is None:

        return ans

    # Append the nodes to queue and process it in while loop until its empty

    queue.append(root)

    # Process in while loop until there are elements in queue

    while queue:

        currSize = len(queue)

        # Unless the queue is empty

        while currSize > 0:

            # Pop elements one-by-one

            currNode = queue.popleft()

            currSize -= 1

            # Check if the node has left/right child

            if currNode.left is not None:

                queue.append(currNode.left)

            if currNode.right is not None:

                queue.append(currNode.right)

        # Increment ans when currSize = 0

        ans += 1

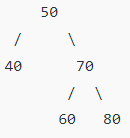
    return ans

**In order successor -- (smallest in the right subtree)**

In Binary Tree, Inorder successor of a node is the next node if we do in Inorder traversal of the Binary Tree. It is smallest in right of that node (if right node exists)

Example:

For below Tree find the inorder successor of 50



Answer:

If we do inorder traversal then output will be –

[40, 50, 60, 70, 80]

Node next of node 50 is 60 so inorder transversal of 50 is - 60

**In order predecessor of node ( largest in left subtree)**

In Binary Tree, Inorder predecessor of a node is node which comes before the asking node if we do inorder transversal.

Example:

In order predecessor of above tree for node 50 will be 40.

**Preorder successor**

Node which come after the asking node if we do preorder transversal.

**Preorder predecessor**

Node which come after the asking node if we do preorder transversal.

**Deleting a node in BST**

If we are deleting a node from BST then we will consider three different cases-

1. Node to be delete is leaf node:

In this case simply remove from tree

50 50

/ \ delete(20) / \

30 70 ---------> 30 70

/ \ / \ \ / \

20 40 60 80 40 60 80

1. Node to be delete has only one child:

Copy the child to the node and delete the child

50 50

/ \ delete(30) / \

30 70 ---------> 40 70

\ / \ / \

40 60 80 60 80

1. Node to be deleted has two child ( left and right ):

Find inorder (smallest in the right subtree) successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used

50 60

/ \ delete(50) / \

40 70 ---------> 40 70

/ \ \

60 80 80

def minValueNode(node):

        current = node

        # loop down to find the leftmost leaf

        while(current.left is not None):

            current = current.left

        return current

    def deleteNode(root, key):

        # Base Case

        if root is None:

            return root

        # If the key to be deleted

        # is smaller than the root's

        # key then it lies in left subtree

        if key < root.data:

            root.left = root.left.deleteNode(key)

        # If the kye to be delete

        # is greater than the root's key

        # then it lies in right subtree

        elif(key > root.data):

            root.right = root.right.deleteNode(key)

        # If key is same as root's key, then this is the node

        # to be deleted

        else:

            # Node with only one child or no child

            if root.left is None:

                temp = root.right

                root = None

                return temp

            elif root.right is None:

                temp = root.left

                root = None

                return temp

            # Node with two children:

            # Get the inorder successor

            # (smallest in the right subtree)

            temp = root.right.minValueNode()

            # Copy the inorder successor's

            # content to this node

            root.key = temp.key

            # Delete the inorder successor

            root.right = root.right.deleteNode(temp.key)

        return root

Example output:

r=BST(50)

r.insert(30)

r.insert(20)

r.insert(40)

r.insert(70)

r.insert(60)

r.insert(80)

print(r.in\_order\_traversal())

print(r.post\_order\_traversal())

print()

r.deleteNode(20)

print(r.in\_order\_traversal())

**###########**

**#Method 2:# --------- Without recursion**

**###########**

def delete(self,e):

        p = self.\_root

        pp = None

        while p and p.\_element != e:

            pp = p

            if e < p.\_element:

                p = p.\_left

            else:

                p = p.\_right

        if not p:

            return False

        #If node which need to deleted have both left and right child or subtree

        if p.\_left and p.\_right:

            s = p.\_left

            ps = p

            while s.\_right:

                ps = s

                s = s.\_right

            p.\_element = s.\_element

            print('value of s is: ',s.\_element)

            p = s #Now p is ponting to node which is greatest in left sub tree and need to be deleted

            pp = ps #Now pp points to parent of node s(s need to be deleted)

        c = None

        if p.\_left:

            c = p.\_left

        else:

            c = p.\_right

        if p == self.\_root:

            self.\_root = None

        else:

            if p == pp.\_left:

                pp.\_left = c

            else:

                pp.\_right = c

**Space and time complexity of binary search tree**

Let suppose:-

h=height of binary search tree

n=Number of nodes in tree

**Time Complexity**

|  |  |  |
| --- | --- | --- |
|  | Average case | Worst case scenario |
| Search | O(log(n)) | O(h) |
| Insert | O(log(n)) | O(h) |
| Delete | O(log(n)) | O(h) |

In worst case: Height is proportional to number of nodes---------> O(n)

**Space complexity**

Space complexity = O(n), n=Number of nodes/elements

**############################################################################**

**# Balanced Search Tree #**

**############################################################################**

We have many balanced search tree – AVL tree, Red-Black Tree, Splay Tree

**Main aim of Balanced Search Tree:**

Reduces the Height of binary search tree

Rotation or Restructuring

Modifies the relationship between parent and child

**##########**

**# AVL tree #**

**##########**

AVL tree is binary search tree with some modification. AVL tree have better performance.

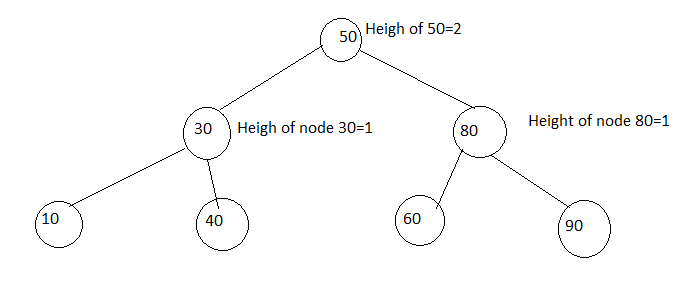
Any binary Search Tree which satisfies- height-balance factory is called AVL tree.

Balance – factor= left subtree height – right subtree height

In AVL tree balance factor could be either +1,0 or -1.

AVL tree is also called is height balanced tree.

Height--- Number of edges from that node to leaf node.



Let us calculate the balance factor for each node in above tree.

**Node 50:**

Balance factor= left subtree height – right subtree height

= 2-2=0

Balance factor of node 50=0

Node 80:

Balance factor= 1-2=0

Node 30:

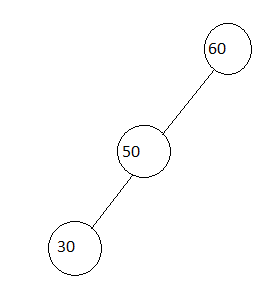
Balance factor=1-1=0

Node 10:

Balance factor = 0-1=0

**Question:**

Check if below binary Search Tree is AVL tree.



|  |  |
| --- | --- |
| Node name | Balance factor |
| 60 | 2-0=2 |
| 50 | 1-0=1 |
| 30 | 0-0=0 |

From above calculation it’s clear the above tree is not AVL tree b/c balance factor of on node is greater than 1.

AVL tree Rotation:

There are 4 types of rotation to balance AVL tree:

1. LL rotation
2. RR rotation
3. LR rotation
4. RL rotation

**Note:**

When we add any new data/node in tree that time we have to check it’s balanced or not and kind of rotation required.

1. LL rotation

It occurs due to adding data in left of left node. To balance it to rotate the parent of newly added node in clockwise.

1. RR Rotation

It occurs due to adding data in right of right node. To balance it to rotate the parent of newly added node in anticlockwise.

1. LR Rotation:

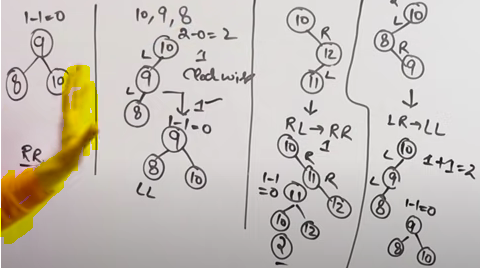
It occurs when BST gets unbalance after adding a node in left then right node. To balance it we need to do two rotations on parent of new added node. First make it RR then do anticlockwise.

To make RR , we add new added data right of it’s parent node.

1. RL rotation:

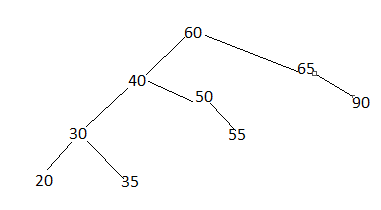
It occurs when BST gets unbalance after adding a node in right then left node. To balance it we need to do two rotations on parent of new added node. First make it LL then do anticlockwise.

To make LL , we add new added data LEFT of it’s parent node.

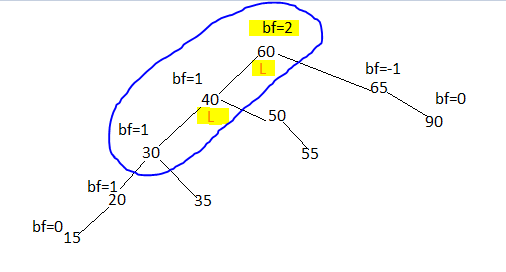


**Question:**

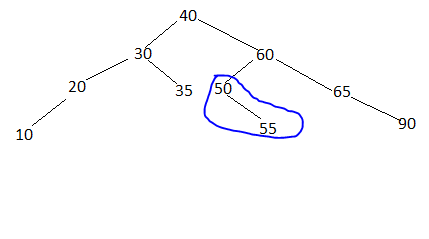
We have below AVL tree add 15 in it and make it AVL.



Let say we added 15 into tree then it will be like below-



Tree have got unbalanced at node 60, we will have to LL rotation on node 60 (b/c it got unbalanced at 60 by adding new value left of 60 then again next of 60).

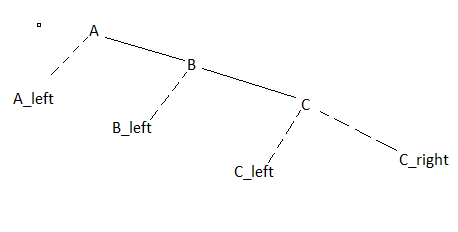


Note:

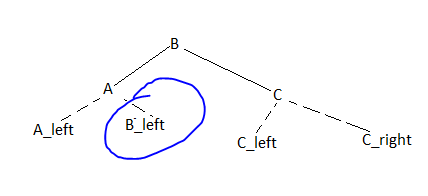
We can see that right child of node 40 are transferred to 60.

**RR rotation**

Let say we have below tree which got unbalanced at A while adding C\_right, to balance it we will have to do RR rotation

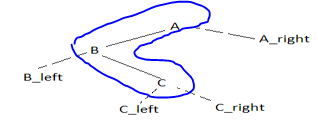


Balanced structure of Tree.



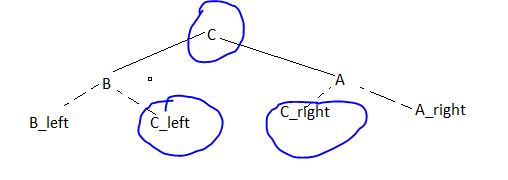
**LR rotation**

Let say we have below tree which is unbalanced we must balance it.



Here node A,B,C will be involved in rotation, b/c Node A is unbalanced b/c left of A addition (node B) then right of Node B(node C ) addition.

We need to do LR rotation b/c node A is unbalanced b/c it’s unbalanced by addition of left side then right-side addition of node A (unbalanced node)



Note:

1. Parent of leaf node (from where unbalancing occurred, in this case C) will be changed to root node, left of A will go to left of C, right of A will go to right of C.
2. Then arrange the data in down the stream according to their parent child and value.

**############################################################################**

**# Heap DataSturcture #**

**############################################################################**

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. That’s why heaps are also called as binary Heap.

Key/value of each node is greater than it’s children value.

Tree must be complete binary tree.

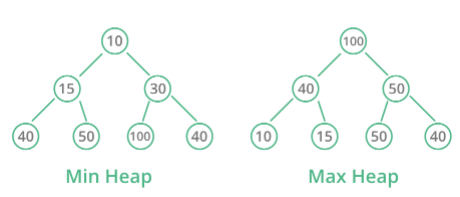
Generally, Heaps can be of two types:

**Max-Heap:** In a Max-Heap the key present at the root node must be greatest among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

**Min-Heap:** In a Min-Heap the key present at the root node must be minimum among the keys present at all of it’s children. The same property must be recursively true for all sub-trees in that Binary Tree.

**Complete Binary tree**

Type of tree in which all the level except last level is completely filled with nodes and last level can be completely filled or nodes are filled from left to right.



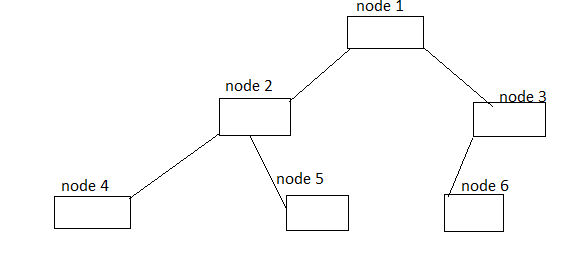
|  |  |
| --- | --- |
| Min Heap | Max Heap |
| The key at the root node is smaller than or equal to the key of their children node | The key at the root node is larger than or equal to the key of their children node |
| The minimum key element is the root node | The maximum key element is the root node |
| Uses the ascending priority | Uses the descending priority |
| The smallest element has priority while construction of min-heap | The largest element has priority while construction of max-heap |
| The smallest elements are popped out of the heap | The largest element is popped out of the heap |

**Inserting data into heap/ creating Maxheap:**

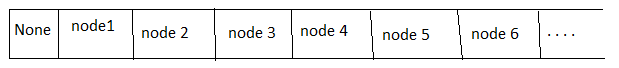
We will be using array data structure to implement heap in data at index 0 will be null and from index 1 onwards data will be of heap data structure.

In heap data structure each node will be represented by number starting from root node.

Naming convention will be – root/parent -- > left child ---- > right child, at each level.



Array representation of above tree/heap



class Heap:

    def \_\_init\_\_(self):

        self.\_maxsize = 10

        self.\_data = [-1] \* self.\_maxsize

        self.\_csize = 0

    def \_\_len\_\_(self):

        return len(self.\_data)

    def isempty(self):

        return len(self.\_data) == 0

    def insert(self, e):

        if self.\_csize == self.\_maxsize:

            print('No Space in Heap')

            return

        self.\_csize = self.\_csize + 1

        hi = self.\_csize

        while hi > 1 and e > self.\_data[hi // 2]:

            self.\_data[hi] = self.\_data[hi // 2]

            hi = hi // 2

        self.\_data[hi] = e

**Creating minheap using python:**

This is exactly same as above just we have to check for less than case in while loop.

If new element is less than last element, then we need to push it up till correct position.

def insert(self,e):

        if self.csize==self.size:

            print('heap is full')

            return

        self.csize=self.csize+1

        hi=self.csize

        while hi>1 and e<self.elements[hi//2]:

            self.elements[hi]=self.elements[hi//2]

            hi=hi//2

        self.elements[hi]=e

        return self.elements[1:self.csize+1]

    def get\_data(self):

        return self.elements[1:self.csize+1]

**Deleting data from heap:**

For deleting data from heap, we will swap value of root with last value of heap element.

After swapping check if node value are greater than it’s child node values.

    def deleteMax(self):

        if self.csize==0:

            print('heap is empty')

            return

        e=self.elements[1]

        self.elements[1]=self.elements[self.csize]

        self.elements[self.csize]=None

        self.csize-=1

        i=1

        j=i\*2

        while j<=self.csize:

            #checkign which child left or right is greater then moving j to that child

            if self.elements[j]<self.elements[j+1]:#check if left child is leass than right,

                j=j+1                              #if yes then move the j pointer to right child

                                                   #if this comparion is false that means left child is greater

            #checking the root value to it's child value (left or right)

            #then interchanging with it's root node (child's root node)

            if self.elements[i]<self.elements[j]:

                temp=self.elements[i]

                self.elements[i]=self.elements[j]

                self.elements[j]=temp

                i=j

                j=i\*2

            #if neither left of right child are greater then just break while loop

            else:

                break

            return e

**###########################################################################**

**# Heapq module in python #**

**###########################################################################**

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm. This module implements min-heap i.e root element will be smallest into heap.

The following functions are provided:

**heapq.heappush(heap, item)**

Push the value item onto the heap(it’s list type variable), maintaining the heap invariant.

**heapq.heappop(heap)**

Pop and return the smallest item from the heap (it’s list type variable), maintaining the heap invariant. If the heap is empty, IndexError is raised. To access the smallest item without popping it, use heap[0].

**heapq.heappushpop(heap, item)**

Push item on the heap(it’s list type variable), then pop and return the smallest item from the heap(it’s list type variable). The combined action runs more efficiently than heappush() followed by a separate call to heappop().

**heapq.heapify(x)**

Transform list x into a heap, in-place, in linear time.

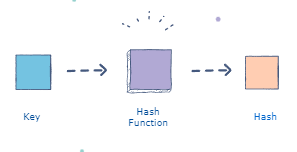
**#########################################################################**

**# Hashing #**

**#########################################################################**

**Hashing:**

Hashing is the process of *converting a given key into another value*. A hash function is used to generate the new value according to a mathematical algorithm. The result of a hash function is known as a hash value or simply, a hash. *Hash values are stored in hash table.*



**Drawback of hashing:**

Sometimes hashing function generates same key for two different values that causes collision, to avoid it we use chaining.

Example:

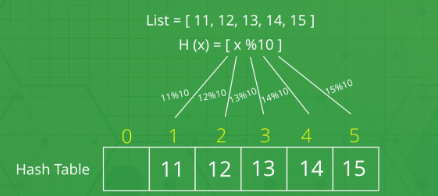
If we are using h(x)=x%10 hash function then it will generate same key for x=22,32,42,..

Hashing is used in searching, inserting, deleting elements from collection.

Time complexity of hashing --- O(1) , for searching, deleting

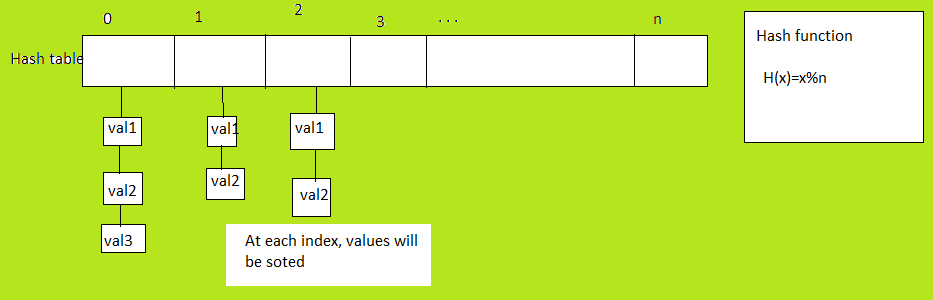
**Collision in Hashing:**

If more than two key/value result to same hash from hashing function such scenario is called as collision. There are many ways to handle it – chaining, liner probing, quadratic probing etc.



**Chaining, Collision handling scheme:**

To handle collision in has table we store values in form of iterable data and value at each index of hash table will be sorted.



**Creating/Implementing Hash table:**

We will be using h(x)=x%n hash function and at each index value will be stored in form of linked list.

#####

# Linked list part of code #

#####

class \_Node:

    \_\_slots\_\_ = '\_element', '\_next'

    def \_\_init\_\_(self, element, next):

        self.\_element = element

        self.\_next = next

class LinkedList:

    def \_\_init\_\_(self):

        self.\_head = None

        self.\_tail = None

        self.\_size = 0

    def isempty(self):

        return self.\_size == 0

    def insertsorted(self,e):

        newest = \_Node(e, None)

        if self.isempty():

            self.\_head = newest

        else:

            p = self.\_head

            q = self.\_head

            while p and p.\_element < e:

                q = p

                p = p.\_next

            if p == self.\_head:

                newest.\_next = self.\_head

                self.\_head = newest

            else:

                newest.\_next = q.\_next

                q.\_next = newest

        self.\_size += 1

    def display(self):

        p = self.\_head

        while p:

            print(p.\_element,end='-->')

            p = p.\_next

        print()

##

# Hashing implemention#

##

class HashChain:

    def \_\_init\_\_(self):

        self.hashtable\_size = 10

        self.hashtable = [None] \* self.hashtable\_size #creating hash table of requried size

        for i in range(self.hashtable\_size):

            self.hashtable[i] = LinkedList() #assigning each node data as linked list data

    def hashcode(self, key):

        return key % self.hashtable\_size

    def insert(self, element):

        i = self.hashcode(element)

        self.hashtable[i].insertsorted(element)#get hash table data from index requried index

                                                #that data will be linked list type and then

                                                # call insert method of linked list

    def display(self):

        for i in range(self.hashtable\_size):

            print('[',i,']',end=' ')

            self.hashtable[i].display()

        print()

H = HashChain()

H.insert(54)

H.insert(78)

H.insert(64)

H.insert(92)

H.insert(34)

H.insert(86)

H.insert(28)

H.display()

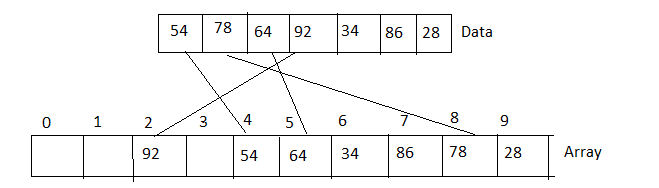
**Load factor = (total number of data points)/ (size of array)**

**Linear probing:**

It is one of the open addressing scheme to handle collision.

In this we find the index of element to be added using hashing function, if it is occupied then we choose next index. We can generalize the hashing function as below-

H(x)= ( h(x)+i)%10 ; i=0,1,2,3,..



**Note:**

For linear probing load factor should be less than .5, that mean half of array should be empty.

**Implementing Linear probing**

class HashLinearProbe:

    def \_\_init\_\_(self):

        self.hashtable\_size = 10

        self.hashtable = [None] \* self.hashtable\_size

    def hashcode(self, key):

        return key % self.hashtable\_size #get the index of array

    def lprobe(self, element):

        i = self.hashcode(element)

        j = 0

        #implementing liner probing

        while self.hashtable[(i+j) % self.hashtable\_size] != None:

            j = j + 1

        return (i + j) % self.hashtable\_size

    def insert(self, element):

        i = self.hashcode(element)

        if self.hashtable[i] == None:

            self.hashtable[i] = element

        else:

            i = self.lprobe(element)

            self.hashtable[i] = element

    def search(self, key):

        i = self.hashcode(key)

        j = 0

        while self.hashtable[(i+j) % self.hashtable\_size] != key:

            if self.hashtable[(i+j) % self.hashtable\_size] == None:

                return False

            j = j + 1

        return True

    def display(self):

        print(self.hashtable)

**Quadratic probing:**

This is another open addressing scheme to handle collision.

It’s logically same as liner probing just difference is of hash function.

H’(x)=(h(x)+i\*i)%10 ; i=0,1,2,3. . . and h(x)=x%10

**Note:**

Load factor should be always less than 0.5.

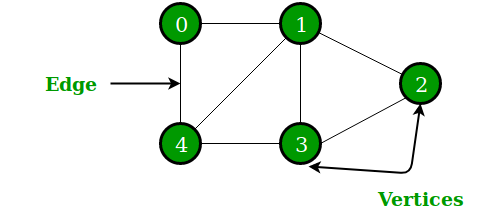
**Drawback of Quadratic probing:**

* Load fact should be less than 0.5 then mean half of array will be empty.
* There could be secondary collision.

**############################################################################ # Graph Data Structure #**

**############################################################################**

A Graph is a *non-linear* data structure consisting of nodes/vertices and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.

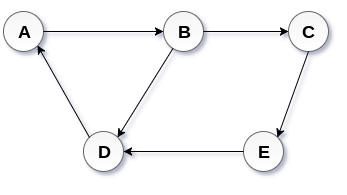


**Application of graph:**

* Graphs are used to *represent networks*. The networks may include paths in a city or telephone network or circuit network.
* Graphs are also used in *social networks like LinkedIn, Facebook.* For example, in Facebook, each person is represented with a vertex (or node). *Each node/vertex is a structure and contains information like person id, name, gender, locale etc*.

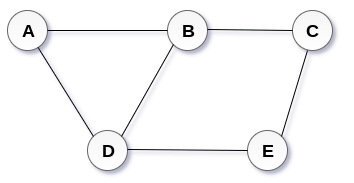
**Directed graph / Digraph**

Graph in which edges have direction. They represent specific path from one node to another node.



**Undirected graph**

In an undirected graph, *edges are not associated with the directions with them*. An undirected graph is shown in the below figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.



**End Vertices:** Two vertices joined by and edge

**Adjacent Vertices:** Two vertices are adjacent if there is an edge between them

**Incident edge:** If vertex is one of the end point

**Degree of vertex deg(v): --- for undirected graph**

Deg(v)= number of edges for that vertex.

For above undirected graph:

deg(A)=2, deg (B)=3

**Degree of directed graph:**

For directed graph degree of any node is defined in two way-

* **In-degree indeg(v):**

Number of incoming edges to that node.

* **Out-degree outdeg(v):**

Number of outgoing edges from that node.

**Weighted Graph:**

Graph in which weight is assigned to each edge. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

*We can assign weight for each node for directed and undirected graph.*

**Path:**

Path is sequence of edges starting from one vertex and ending at another vertex.

**Cycle:**

Path that starts and ends at same vertex.

**Directed Acyclic Graph:**

A graph when there is no cycle in graph is called acyclic graph.

**Sub graph:**

It is also a *graph whose edges and vertices are subset of vertices and edges of another graph.*

**Articulation Point:**

Vertex whose *removal still results in connected graph is called articulation point.*

**Bi-connected component:**

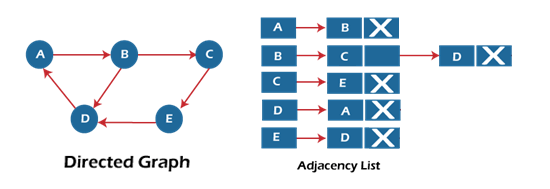
Components connected by two edges.

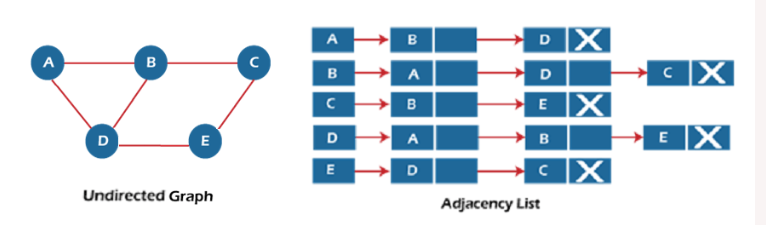
**Strongly connected graph:**

All vertices are reachable from any vertex.

**Adjacency list:**

For each vertex, a separate list of edges is maintained adjacent to that vertex. A linked list is used for storing the adjacency list (outgoing edges) . Let’s assume we have below graph and want to create adjacency list-



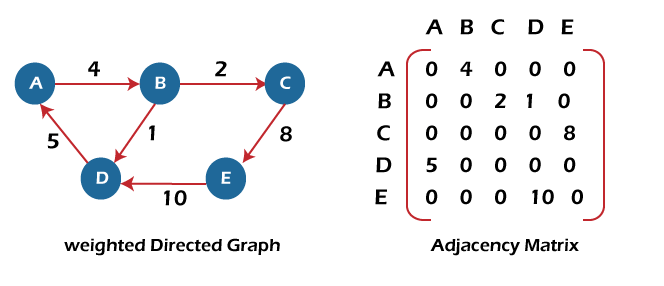


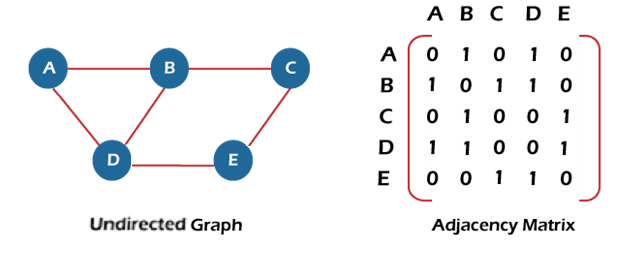
For undirected graph there is no direction so we will assume all are outgoing edges at any node.

**Adjacency Matrix:**

Maintains a matrix of edges, where each cell of matrix stores the reference/weight of edges.

*If there is no edge between two vertex for that we use Null or 0 Values.*





**Creating graph**

For creating graph, we will use adjacent array/matrix to implement it.

import numpy as np

class Graph:

    def \_\_init\_\_(self, vertices):

        self.\_vertices = vertices

        self.\_adjMat = np.zeros((vertices, vertices)) #Create adjacent matrix

                                                      # initialize 0 for each edges/cell

    def insert\_edge(self, u, v, x=1): #u and v are nodes, by default weight for any edge=1

                                    #v is weight for edges between u and v

        self.\_adjMat[u][v] = x #assign the weight for edge between node u and v

    def display\_adjMat(self):

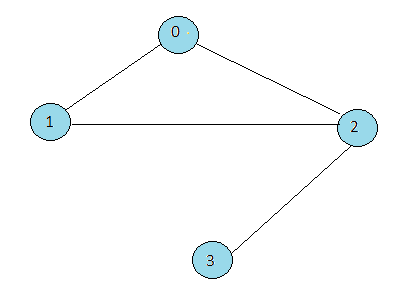
        print(self.\_adjMat)

**Note:**

Above code can implement any type of graph- directed weighted graph, undirected weighted graph, directed unweighted graph or undirected unweighted graph.

Implementing undirected unweighted graph:

Let’s suppose we have to implement the below graph. *Basic code for creating graph will be same as above.*

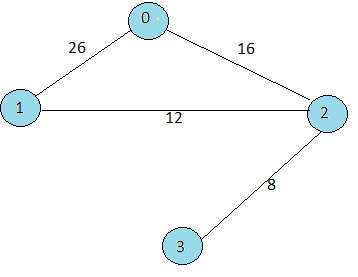


|  |  |
| --- | --- |
| g=Graph()  g.insert\_edge(0,1)  g.insert\_edge(1,0)  g.insert\_edge(0,2)  g.insert\_edge(2,0)  g.insert\_edge(1,2)  g.insert\_edge(2,1)  g.insert\_edge(2,3)  g.insert\_edge(3,2)  g.display\_adjMat() | Adjancet matrix    Note:  For undirected graph, same edge will come twice in adjacent matrix b/c no direction |

**Implementing weighted undirected graph:**

It’s same way as above we have to just provide the weight for each edge.

*This is undirected case so each edge will come twice in adjacent matrix with its weight.*

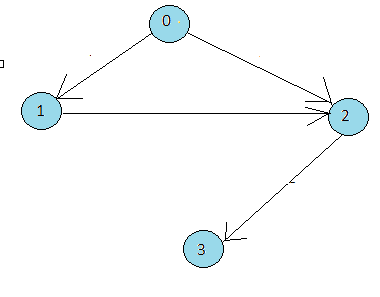


|  |  |
| --- | --- |
| g=Graph(4) #Graph class is same as above  g.insert\_edge(0,1,26)  g.insert\_edge(1,0,26)  g.insert\_edge(0,2,16)  g.insert\_edge(2,0,16)  g.insert\_edge(1,2,12)  g.insert\_edge(2,1,12)  g.insert\_edge(2,3,8)  g.insert\_edge(3,2,8)  g.display\_adjMat() |  |

**Implementing directed unweighted graph:**

Let say we have below directed graph that we want to implement.

It’s exactly same as above but we will create edges based on directed.



|  |  |
| --- | --- |
| g=Graph(4) #Graph class is same as above  g.insert\_edge(0,1)  g.insert\_edge(0,2)  g.insert\_edge(1,2)  g.insert\_edge(2,3)  g.display\_adjMat() |  |

**Implementing directed, weighted graph:**

Do it yourself.

**Checking if edge exist between two nodes:**

Check the value at index [u][v] in adjacent matrix.

If it contains default (0 or None value) then edges doesn’t exists between those two nodes.

import numpy as np

class Graph:

    def \_\_init\_\_(self, vertices):

        self.\_vertices = vertices

        self.\_adjMat = np.zeros((vertices, vertices)) #Create adjacent matrix

                                                      # initialize 0 for each edges/cell

    def exist\_edge(self, u, v):

        return self.\_adjMat[u][v] != 0

**Removing edges between two nodes/vertex:**

If want to remove the edge between node/vertex u,v then remove or set default value at index [u][v].

import numpy as np

class Graph:

    def \_\_init\_\_(self, vertices):

        self.\_vertices = vertices

        self.\_adjMat = np.zeros((vertices, vertices)) #Create adjacent matrix

                                                      # initialize 0 for each edges/cell

    def remove\_edge(self, u, v):

        self.\_adjMat[u][v] = 0

**Getting edges count:**

For undirected graph , this method will return double of actual number of edges because there is no direction in edges that causes duplication of edge adding in adjacent matrix.

def edge\_count(self):

        count = 0

        for i in range(self.\_vertices):

            for j in range(self.\_vertices):

                if self.\_adjMat[i][j] != 0:

                    count = count + 1

        return count

**Indegree and Outdegree count:**

def outdegree(self, v):

        count = 0

        for j in range(self.\_vertices):

            if self.\_adjMat[v][j] != 0:

                count = count + 1

        return count

def indegree(self, v):

        count = 0

        for i in range(self.\_vertices):

            if self.\_adjMat[i][v] != 0:

                count = count + 1

        return count

**Getting edges:**

def edges(self):

        for i in range(self.\_vertices):

            for j in range(self.\_vertices):

                if self.\_adjMat[i][j] != 0:

                    print(i,'--',j)

**#############################################################################**

**# DFS and BFS in graph #**

**#############################################################################**

BFS is kind of level order traversal

|  |  |  |
| --- | --- | --- |
|  | **BFS** | **DFS** |
| **Full form** | BFS stands for Breadth First Search. | DFS stands for Depth First Search. |
| **Technique** | It a vertex-based technique to find the shortest path in a graph. | It is an edge-based technique because the vertices along the edge are explored first from the starting to the end node. |
| **Definition** | BFS is a traversal technique in which all the nodes of the same level are explored first, and then we move to the next level. | DFS is also a traversal technique in which traversal is started from the root node and explore the nodes as far as possible until we reach the node that has no unvisited adjacent nodes. |
| **Data Structure** | Queue data structure is used for the BFS traversal.  It do not uses recursion for visiting nodes | Stack data structure is used for the BFS traversal.  It uses recursion for visiting nodes. |
| **Backtracking** | BFS does not use the backtracking concept. | DFS uses backtracking to traverse all the unvisited nodes. |
| **Number of edges** | BFS finds the shortest path having a minimum number of edges to traverse from the source to the destination vertex. | In DFS, a greater number of edges are required to traverse from the source vertex to the destination vertex. |
| **Optimality** | BFS traversal is optimal for those vertices which are to be searched closer to the source vertex. | DFS traversal is optimal for those graphs in which solutions are away from the source vertex. |
| **Speed** | BFS is slower than DFS. | DFS is faster than BFS. |
| **Suitability for decision tree** | It is not suitable for the decision tree because it requires exploring all the neighboring nodes first. | It is suitable for the decision tree. Based on the decision, it explores all the paths. When the goal is found, it stops its traversal. |
| **Memory efficient** | It is not memory efficient as it requires more memory than DFS. | It is memory efficient as it requires less memory than BFS. |

**Breadth First Search (BFS):**

* It subdivides the vertices into levels and proceeds in rounds.
* Start at a vertex, which any be any vertex.
* identifies all vertices reachable from source to level 1 and mark them visited.
* In next round, identifies new vertices reachable from level1 vertices whcih are not yet visited, mark them visited.
* This process continues until no vertices are found.

class Graph:

    def \_\_init\_\_(self, vertices):

        self.\_adjMat = np.zeros((vertices, vertices))

        self.\_vertices = vertices

    def BFS(self, s):

        i = s #s is start vertex for BFS

        #q is queue data structure to do enque or dequeue operation

        #I have used my own implemented queue class but it can be any queue

        #which can do enque and dequeue operation

        q = QueuesLinked()

        #visited is list type initialize to all 0

        #if any node is visited then value of that index will be >0

        visited = [0] \* self.\_vertices

        print(i,end=' ')

        visited[i] = 1

        q.enqueue(i)

        while not q.isempty():

            i = q.dequeue()

            for j in range(self.\_vertices):

                #check if path of adjacent vertices

                #if yes then check also if it was not visited

                if self.\_adjMat[i][j] == 1 and visited[j] == 0:

                    print(j,end=' ')

                    visited[j] = 1

                    q.enqueue(j)

**Depth First Search (DFS):**

* Depth-First Search starts at a vertex.
* Selects the adjacent vertex from the start vertex.
* Visit the adjacent vertex, mark as visited.
* Now visites vertex will be current vertex and source.
* Now recursively call the method for DFS.

import numpy as np

class Graph:

    def \_\_init\_\_(self, vertices):

        self.\_adjMat = np.zeros((vertices, vertices))

        self.\_vertices = vertices

        self.\_visited = [0] \* vertices

    def DFS(self,s):

        if self.\_visited[s]==0:

            print(s,end=' ')

            self.\_visited[s]=1

            for j in range(self.\_vertices):

                if self.\_adjMat[s][j]==1 and self.\_visited[j]==0:

                    self.DFS(j)

**############################################################################ # Interview Questions #**

**############################################################################**

1. What is data structure?

A data structure is a *way of organizing data* so that the data can be used efficiently. Different kinds of data structures are suited to different kinds of applications, and some are highly specialized for specific tasks.

1. What is a Linear Data Structure?

A data structure is linear if all its *elements or data items are arranged in a sequence or a linear order*.

The elements are stored in a non-hierarchical way so that each item has successors and predecessors except the first and last element in the list.

Example – linked list, array, stack, queue

1. What is Non-Linear data structure?

In non-linear data structure datas are stored in hierarchical order or non-linear order.

Example – tree, Graph

1. What are applications of data structure?

There are many usage in industry, some very basic usages are:-

* Arrays are used in *image processing and speech processing*
* *Music players and image sliders* use Linked Lists to move to next or previous items.
* A Queue is used for *job scheduling, the arrangement* of data packets for communication.
* Technologies like *Blockchain, cryptography* are based on Hashing algorithms.
* Matrices are widely used to represent data and *plotting graphs, performing statistical analysis.*
* *Tree data structure is used for comment and replies in social media.*

1. What is difference between linked list and array?

|  |  |
| --- | --- |
| Array | Linked list |
| It stores homogeneous data  Size of array is fixed  Random access of element is allowed  No extra space is required  Insertion and deletion of element is costly | It stores heterogeneous data  Size of array is flexible  Random access of element is not allowed  Extra space is required for storing pointer  Insertion, deletion of element is easy |

1. What is Stack and where it can be used?

Stack is a linear data structure in which the order *LIFO(Last In First Out) or FILO(First In Last Out)* for accessing elements. Basic operations of the stack are: Push, Pop, Peek

Application of stack -

* Saving the web page history.
* Undoing the operation (ctrl+z) in editing application ( notepad, notepad++)
* Evaluation of arithmetic operation.
* Reversing string

1. What is a Queue, how it is different from the stack and how is it implemented?

The queue is a linear structure that follows the order is First In First Out (FIFO) to access elements but stack follows FILO or LIFO data structure.

1. What is a Linked List and What are its types?

A linked list is a linear data structure (like arrays) where data are stored in nodes. Each node contains data pointer and next or previous node pointer (based on types of linked list)

Types of linked list-

1. Singly linked list
2. Doubly linked list
3. Circular linked list
4. When array is best choice?
5. When we know number of elements to be stored.
6. When want random access of elements.
7. When we need speed for iteration through elements.
8. When we need to store homogeneous data.
9. When to use linked list?
10. When want to store heterogeneous data.
11. When there is large number of add to remove operation required to do.
12. When there is no large number of random-access operation
13. When speed is not important when iterating through the elements.
14. Implement queue data structure using stack.

**Method 1:**

This method makes sure that oldest entered element is always at the top of stack 1, so that deQueue operation just pops from stack1. To put the element at top of stack1, stack2 is used

enQueue(q, x):

* While stack1 is not empty, push everything from stack1 to stack2.
* Push x to stack1 (assuming size of stacks is unlimited).
* Push everything back to stack1.

class Queue:

    def \_\_init\_\_(self):

        self.s1 = [] #s1 working as stack1

        self.s2 = [] #s2 working as stack2

    def enQueue(self, x):

        # Move all elements from s1 to s2

        while len(self.s1) != 0:

            self.s2.append(self.s1[-1])

            self.s1.pop()

        # Push item into self.s1

        self.s1.append(x)

        # Push everything back to s1

        while len(self.s2) != 0:

            self.s1.append(self.s2[-1])

            self.s2.pop()

    # Dequeue an item from the queue

    def deQueue(self):

            # if first stack is empty

        if len(self.s1) == 0:

            print("Q is Empty")

        # Return top of self.s1

        x = self.s1[-1]

        self.s1.pop()

        return x

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

    q = Queue()

    q.enQueue(1)

    q.enQueue(2)

    q.enQueue(3)

    print(q.deQueue())

    print(q.deQueue())

    print(q.deQueue())

**Method 2:**

In this method, in en-queue operation, the new element is entered at the top of stack1. In de-queue operation, if stack2 is empty then all the elements are moved to stack2 and finally top of stack2 is returned.

enQueue(q, x)

* Push x to stack1 (assuming size of stacks is unlimited).
* Here time complexity will be O(1)

deQueue(q)

* If both stacks are empty then error.
* If stack2 is empty
* While stack1 is not empty, push everything from stack1 to stack2.
* Pop the element from stack2 and return it.

class Queue:

    def \_\_init\_\_(self):

        self.s1 = []

        self.s2 = []

    # EnQueue item to the queue

    def enQueue(self, x):

        self.s1.append(x)

    # DeQueue item from the queue

    def deQueue(self):

        # if both the stacks are empty

        if len(self.s1) == 0 and len(self.s2) == 0:

            print("Q is Empty")

            return

        # if s2 is empty and s1 has elements

        elif len(self.s2) == 0 and len(self.s1) > 0:

            while len(self.s1):

                temp = self.s1.pop()

                self.s2.append(temp)

            return self.s2.pop()

        else:

            return self.s2.pop()

    # Driver code

if \_\_name\_\_ == '\_\_main\_\_':

    q = Queue()

    q.enQueue(1)

    q.enQueue(2)

    q.enQueue(3)

    print(q.deQueue())

    print(q.deQueue())

    print(q.deQueue())

1. How to check if given Tree is BST?

* Each node can have maximum of two child nodes.
* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree must also be a binary search tree.
* There must be no duplicate nodes.

If inorder traversal of a binary tree is sorted, then the binary tree is BST.

**Algorithm**:

* We will select the range for each node data, for root node we will assume that data can be anything from -ve infinity to +ve infinity.
* For child node we will update the value range.

**let say left child**— it value can be anything, less than parent node to -ve infinity or we can say as: -ve\_infinity<child data< parent\_data

**let say for right child**, we can conclude it to: parent\_data<right child<+ve\_infinity

    def isBST(self):

        def valid(node,left, right):

            if not node:

                return True

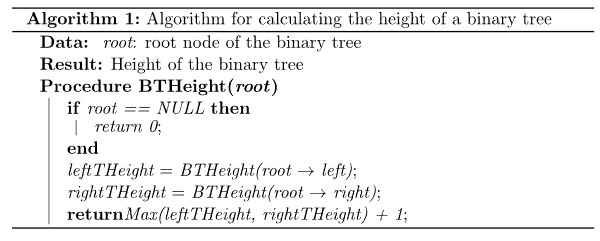
            if not (node.data<right and node.data>left):

                return False

            return (valid(node.left, left,node.data) and valid(node.right, node.data,right))

        return valid(self,float("-inf"),float("inf"))

1. WAP to find out the height of binary tree or binary search tree.



    def maxDepth(self):

        self.lDepth=0

        self.rDepth=0

        if self.data is None:

            return 0

        else :

            # Compute the depth of each subtree

            if self.left:

                self.lDepth = self.left.maxDepth()

            if self.right:

                self.rDepth = self.right.maxDepth()

            # Use the larger one

            if (self.lDepth > self.rDepth):

                return self.lDepth+1

            else:

                return self.rDepth+1

1. How are linked lists more efficient than arrays?
2. **Insertion and Deletion**

* Insertion and deletion process is expensive in an array as the room has to be created for the new elements and existing elements must be shifted.
* But in a linked list, the same operation is an easier process, as we only update the address present in the next pointer of a node.

1. **Dynamic Data Structure**

* Linked list is a dynamic data structure that means there is no need to give an initial size at the time of creation as it can grow and shrink at runtime by allocating and deallocating memory.
* Whereas, the size of an array is limited as the number of items is statically stored in the main memory.

1. **No wastage of memory**

* As the size of a linked list can grow or shrink based on the needs of the program, there is no memory wasted because it is allocated in runtime.
* In arrays, if we declare an array of size 10 and store only 3 elements in it, then the space for 3 elements is wasted. Hence, chances of memory wastage is more in arrays.

1. Which is more efficient linked list or array?

Linked list and array both have it’s own efficiency based on its application.

* Array :- Array is efficient for operation where there is more requirement of random access or accessing the elements and want to store homogeneous data.
* Linked list:- it’s more efficient when there is more requirement of deleting, updating the data.

1. Implement stack using queue.

We can implement stack using queue in two different ways-

1. **By doing enque operation costly.**

In this method we need to que instance variable.

**push(s, data):**

Enqueue data to q2

Dequeue elements one by one from q1 and enqueue to q2.

Swap the names of q1 and q2

**pop(s):**

dequeue from q1 and return it.

class Stack:

    def \_\_init\_\_(self):

        self.q1=[]

        self.q2=[]

        self.stack=[]

    def enque(self,data):

        self.q2.append(data) #appending to q2

        while len(self.q1)!=0:

            self.q2.append(self.deque()) #deque from q1 and enque to q2

        self.q1,self.q2=self.q2,self.q1 #swaping names

    def deque(self):

        if len(self.q1)==0:

            print('no item in que')

            return

        item=self.q1[0]

        del self.q1[0]

        #print(item)

        return item

    def all\_data(self):

        print(self.q1)

1. **By making pop operation costly**

* In push operation, the element is enqueued to q1.
* In pop operation, *all the elements from q1 except the last remaining element, are pushed to q2 if it is empty. That last element remaining of q1 is dequeued and returned.*
* class Stack:
* def \_\_init\_\_(self):
* self.q1=[]
* self.q2=[]
* self.stack=[]
* def enque(self,data):
* self.q1.append(data) #appending to q1

* def deque(self):
* if len(self.q1)==0:
* print('no item in que')
* return
* while len(self.q1)!=1:#deque from q1 and enque to q2 exxcept last element
* self.q2.append(self.q1[0])
* del self.q1[0]
* temp=self.q1.pop() #store last element value to temp
* self.q1,self.q2=self.q2,self.q1 #swap names
* return temp        #return temp
* def all\_data(self):
* print(self.q1)

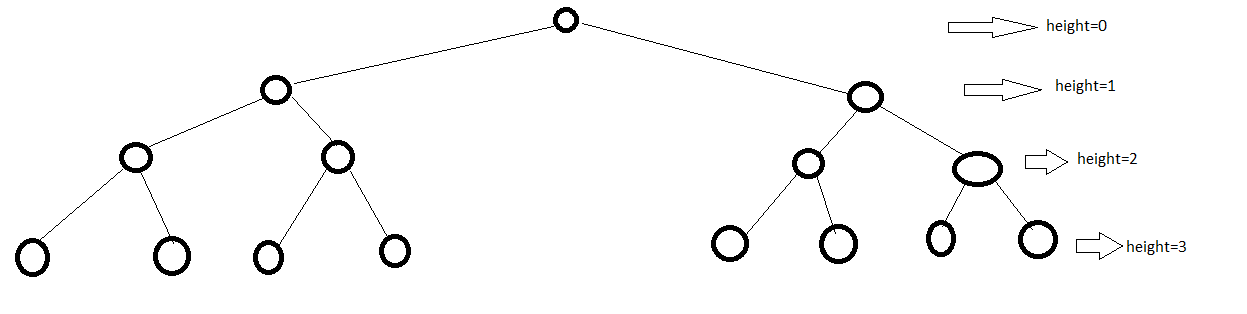
1. What are applications of Tree data structure?

Some common applications of tree are-

* Filesystems —files inside folders that are inturn inside other folders.
* Comments on social media — comments, replies to comments, replies to replies etc form a tree representation.
* Family trees — parents, grandparents, children, and grandchildren etc that represents the family hierarchy.
* If numbers are stored in tree then searching, sorting operation are quicker (for BST tree case)

1. What is the maximum number of nodes in a binary tree of height k?

The maximum nodes are : 2k+1-1 where k >= 1



**Height** = Height of any node is number of edges from root node to that node.

1. WAP to get the count of number of nodes.

We can get count in two ways –

**Method 1**: Length of in order traversal. Do if your self.

**Method 2:**

Initialize the count variable to 0 then recursively call the method on left and right node.

    def nodes\_count(self):

        count=0

        if self.left:

            count+=self.left.nodes\_count()

        count+=1

        if self.right:

            count+=self.right.nodes\_count()

OR

    def nodes\_count(self):

        count=0

        if self.left:

            count+=self.left.nodes\_count()

        if self.right:

            count+=self.right.nodes\_count()

        count+=1

        return count

**Note:**

Here we are not doing inorder or post order traversal so recurvely calling on left or right node can be interchanged.

**#############################################################################**

**# SORTING #**

**#############################################################################**

Sorting algorithms are a set of instructions that take an iterable as an input and arrange the items into a particular order.

Sorts are most commonly in numerical or a form of alphabetical (or lexicographical) order, and can be in ascending (A-Z, 0-9) or descending (Z-A, 9-0) order.

We can classify sorting based on various parameter, e.g- comparison sorting, index based sorting etc.

|  |  |
| --- | --- |
| **Comparison** | **Index Based** |
| Selection Sort  Insertion Sort  Bubble Sort  Merge Sort  Quick Sort  Shell Sort  Heap Sort | Count Sort  Bucket Sort  Radix Sort |

**Stable sorting and Unstable Sorting**

**Stable Sort:**

If elements are sorted and duplicate elements order are preserved then it’s stable sort.

**Unstable Sorting:**

If elements are sorted and duplicate elements order are not preserved then it’s stable sort.

**Selection Sort**

The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.

The algorithm maintains two subarrays in a given array.

**Example:**

Suppose we have to sort below array.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **64** | 25 | 12 | 22 | 11 |

Iteration 1: Find the minimum and place at first position and exchange the data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **11** | 25 | 12 | 22 | 64 |

sorted to be sorted



Note:



Element at index 0 (64)is swapped with minimum element (11). i.e -

Iteration 2: Find the minimum from unsorted data and place at second position and exchange the data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | **25** | 22 | 64 |

Sorted Unsorted



Here minimum (12) from remaining unsorted data is replaced.



Iteration 3: Find the minimum from unsorted data and place at third position and exchange the data.



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | **22** | 25 | 64 |



Sorted data Unsorted data

Iteration 4: Find the minimum from unsorted data and place at fourth position and exchange the data.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | 22 | **25** | 64 |



Sorted data unsorted data

Iteration 5: There is only one data left so no sorting is needed.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **11** | **12** | **22** | **25** | **64** |

Note:

1. Selection sorting is unstable sorting, i.e. duplicate value change their occurrence order in sorted result.
2. Number of swapping done = O(n)
3. Number of comparisons done O(n2)

Implementing Selection Sorting:

def selection\_sort(arr):

        for i in range(len(arr)-1):

                #print('iteration number: ',i,end='->')

                position=i

                for j in range(i+1,len(arr)):

                        #get the value less than value at  index i in remaining

                        if arr[position]>arr[j]:

                                position=j

                temp=arr[i]

                arr[i]=arr[position]

                arr[position]=temp

                #print(arr,end='\n')

        return arr

**Insertion Sorting:**

It is type of sorting technique in which first element of unsorted arrary(iterable) is compared with all elements of sorted elements and swapped to place at correct position.

This is stable sorting technique.

Let us try to understand it by below example:

Consider an example: arr[]: {12, 11, 13, 5, 6}

| **12** | **11** | **13** | **5** | **6** |
| --- | --- | --- | --- | --- |

**First Pass:**

Initially, the first two elements of the array are compared in insertion sort.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **12** | **11** | 13 | 5 | 6 |

Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.

So, for now 11 is stored in a sorted sub-array.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **11** | **12** | 13 | 5 | 6 |

**Second Pass:**

 Now, move to the next two elements and compare them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | **12** | **13** | 5 | 6 |

Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

**Third Pass:**

Now, two elements are present in the sorted sub-array which are **11** and **12**

Moving forward to the next two elements which are 13 and 5

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | **13** | **5** | 6 |

Both 5 and 13 are not present at their correct place so swap them

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | 12 | **5** | **13** | 6 |

After swapping, elements 12 and 5 are not sorted, thus swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 11 | **5** | **12** | 13 | 6 |

Here, again 11 and 5 are not sorted, hence swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **5** | **11** | 12 | 13 | 6 |

here, it is at its correct position

**Fourth Pass:**

Now, the elements which are present in the sorted sub-array are **5, 11** and **12**

Moving to the next two elements 13 and 6

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | 12 | **13** | **6** |

Clearly, they are not sorted, thus perform swap between both

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | 12 | **6** | **13** |

Now, 6 is smaller than 12, hence, swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 11 | **6** | **12** | 13 |

Here, also swapping makes 11 and 6 unsorted hence, swap again

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | **6** | **11** | 12 | 13 |

Finally, the array is completely sorted.

Observation from example:

First element of unsorted array is compare with each element of unsorted element(start from right most to left most) and elements are exchanged if needed.

Example:

Sort the data [3,5,8,9,6,2]

Solution.

Assume that data at index 0 is sorted remaining at unsorted.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 5 | 8 |  | 9 | 6 | 2 |

Sorted unsorted



Step 1:

Assume that data at index 0 (3) is sorted and compare with first data of unsorted data (5). Since 3 is less than 5 so nothing need to do.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 8 | 9 | 6 | 2 |



Sorted Unsorted

Step 2:

Now take the first data from unsorted array/iterable and compare with sorted element at place at correct position.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 8 | 9 | 6 | 2 |

Sorted Unsorted



Steps 3:

Now take the first data from unsorted array/iterable and compare with sorted element at place at correct position.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 8 | 9 | 6 | 2 |

Sorted unsorted



Step 4:

Now take the first data from unsorted array/iterable and compare with sorted element at place at correct position.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 8 | 6 | 9 | 2 |
|  |  |  |  |  |  |
| 3 | 5 | 6 | 8 | 9 | 2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 3 | 5 | 6 | 8 | 9 | 2 |



Sorted Unsorted

Step 5:

Now take the first data from unsorted array/iterable and compare with sorted element at place at correct position.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2 | 3 | 5 | 6 | 8 | 9 |

Sorted



Complexity of Insertion Sort:

Number of comparisons = Number of swapping = O(n2)

**Implementing insertion sort**

def insertion\_sort(arr):

        for i in range(1,len(arr)):

                #print('iteration number: ',i,end='->')

                cvalue=arr[i]

                position=i

                #check if first element of unsorted is greated that last element of sorted

                #if yes, then do right shifting of sorted data

                while position>0 and arr[position-1]>cvalue:

                        arr[position]=arr[position-1]

                        position=position-1

                arr[position]=cvalue

                #print(arr,end='\n')

        return arr

**Bubble Sort:**

It is the sorting algorithm which compare the two consecutive element in each step.

It is called bubble sort b/c when we complete the first pass on all elements then higher element is bubbled at end.

This is not suitable if data set have large data b/c it has high complexity.

Bubble sorting is stable sorting.

Example:

Sort the below data [5,1,4,2,8] using bubble sort

**First Pass:**

* Bubble sort starts with very first two elements, comparing them to check which one is greater.
  + ( **5** **1** 4 2 8 ) –> ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.
  + ( 1 **5** **4** 2 8 ) –>  ( 1 **4** **5** 2 8 ), Swap since 5 > 4
  + ( 1 4 **5** **2** 8 ) –>  ( 1 4 **2** **5** 8 ), Swap since 5 > 2
  + ( 1 4 2 **5** **8** ) –> ( 1 4 2 **5** **8** ), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

**Second pass:**

Now, during second iteration it should look like this:

* ( **1** **4** 2 5 8 ) –> ( **1** **4** 2 5 8 )
* ( 1 **4** **2** 5 8 ) –> ( 1 **2** **4** 5 8 ), Swap since 4 > 2
* ( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )
* ( 1 2 4 **5** **8** ) –>  ( 1 2 4 **5** **8** )

**Third Pass:**

* Now, the array is already sorted, but our algorithm does not know if it is completed.
* The algorithm needs one **whole** pass without **any** swap to know it is sorted.
  + ( **1** **2** 4 5 8 ) –> ( **1** **2** 4 5 8 )
  + ( 1 **2** **4** 5 8 ) –> ( 1 **2** **4** 5 8 )
  + ( 1 2 **4** **5** 8 ) –> ( 1 2 **4** **5** 8 )
  + ( 1 2 4 **5** **8** ) –> ( 1 2 4 **5** **8** )

**Note:**

Number of comparisons=O(n2)

Number of swapping =O(n2)

Implementing Bubble Sort:

**Method 1:**

def bubble\_sort(arr):

        for passes in range(len(arr)-1,0,-1):

                for i in range(passes):

                        #if preceder is greater than successor

                        #do the swaping of element

                        if arr[i]>arr[i+1]:

                                temp=arr[j]

                                arr[i]=arr[i+1]

                                arr[i+1]=temp

        return arr

**Method 2:**

def bubble\_sort(arr):

    for i in range(len(arr)-1,0,-1):

        for passes in range(i):

            if arr[passes]>arr[passes+1]:

                arr[passes],arr[passes+1]=arr[passes+1],arr[passes]

Shell Sort: