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.

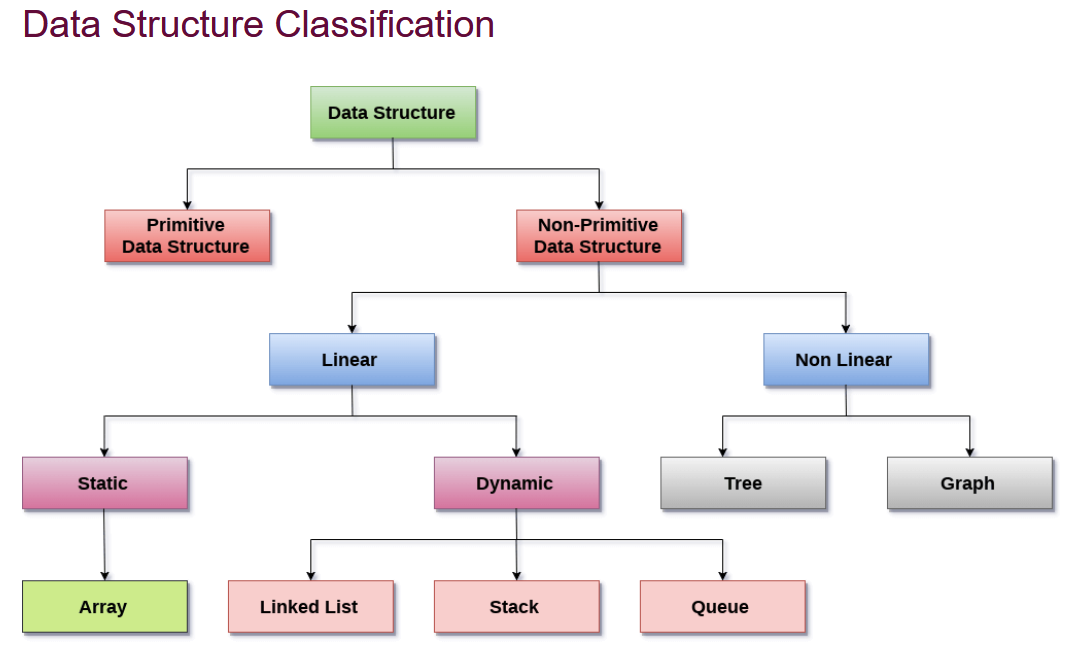
The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues.

**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:**

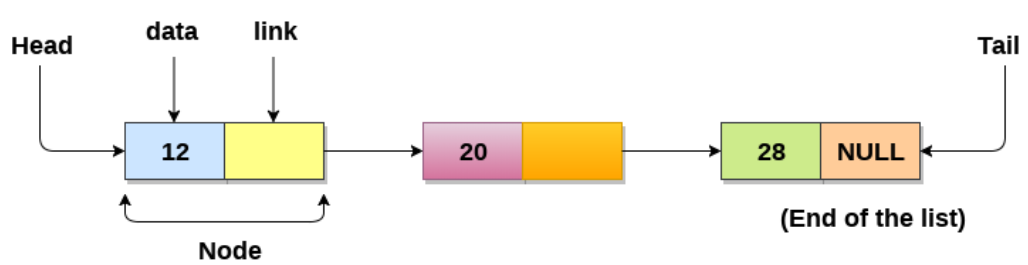
It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible. e.g - list

**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.



We can further break linked list in three parts—

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

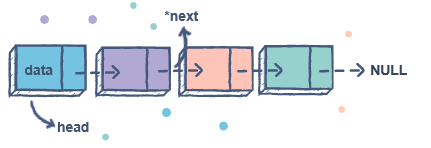
Linked list is same as array but array have below limitation which can be removed in linked list-

**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 of 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()

**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 deletedata(self,data):

        current\_node=self.headval

        previous\_node=None

        while current\_node:

            if current\_node.dataval!=data:

                previous\_node=current\_node #assing curret\_node to previous\_node variable

                current\_node=current\_node.nextval #change current\_node to next node

            else: #data found, come out of loop

                break

        if previous\_node is None: #required data is at first Node, just change the headval

            self.headval=current\_node.nextval

            return 1

        if current\_node is None: #required data is not present in Node

            return 1

            print('data is not present in Node')

        else:                    #requried data is presetn in Node **----- Line xyx**

            previous\_node.nextval=current\_node.nextval

**line xyz**

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

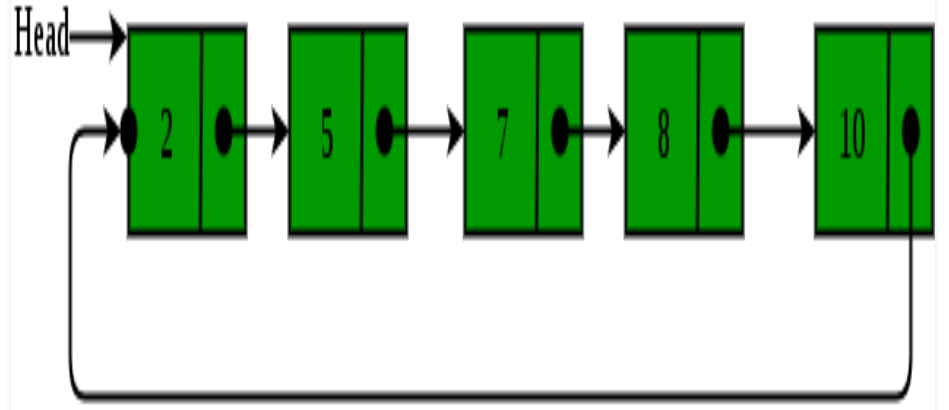
#############################################################################

# Circular linked list #

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

The circular linked list is the collection of nodes in which tail node also point back to head node.

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



**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

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

            self.head=self.head.next\_node

            self.end=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\_node=current\_node.next\_node #assign previous\_node=next\_node of curent node

                    break

                else:

                    previous\_node=current\_node

                    current\_node=current\_node.next\_node

            else:

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

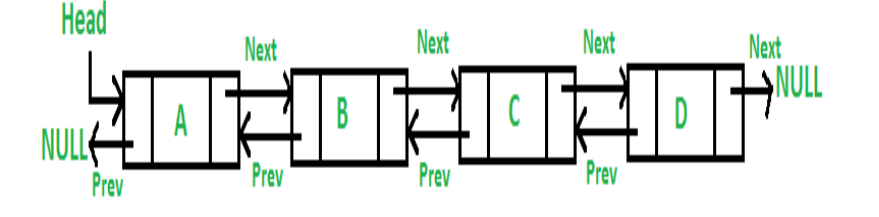
Steps:

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.

**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:

We can traverse if any direction, i.e. it’s bidirectional.

**Creating Doubly liked 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()

**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**

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

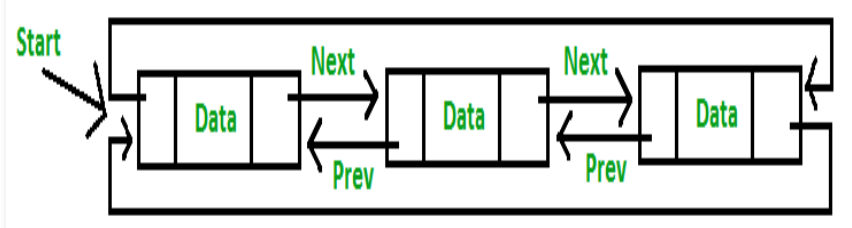
                print('data not present in node')

############################################################################## 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.



**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

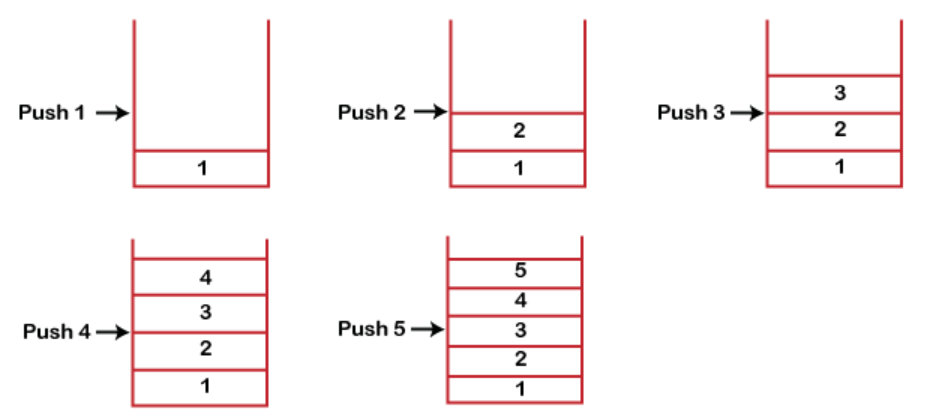
                    current\_node=current\_node.next\_node

############################################################################## 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.



**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.

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.

**Note:**

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

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])

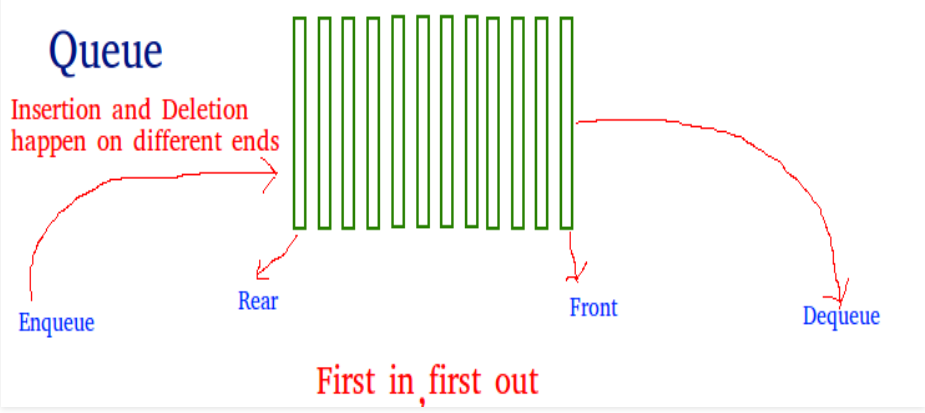
############################################################################

# Queue #

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

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

Size of queue is fixed



**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.

**Rear**: Get the last item from queue.

**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)

**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')

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

# 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 #

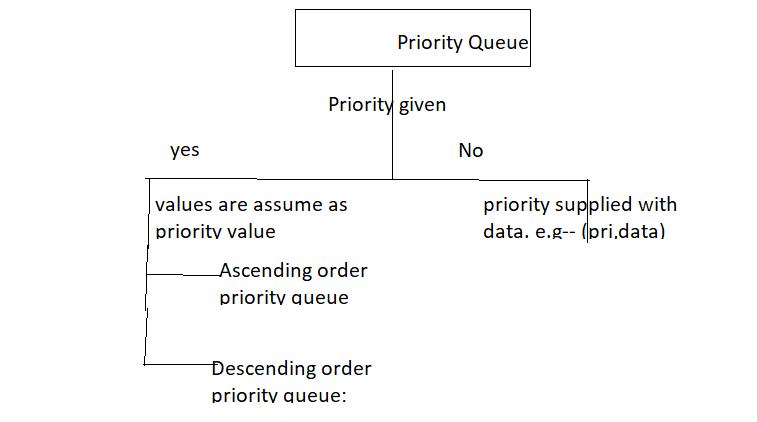
###########################

Priority Queue is an extension of queue with following properties.

**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. 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.

**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')

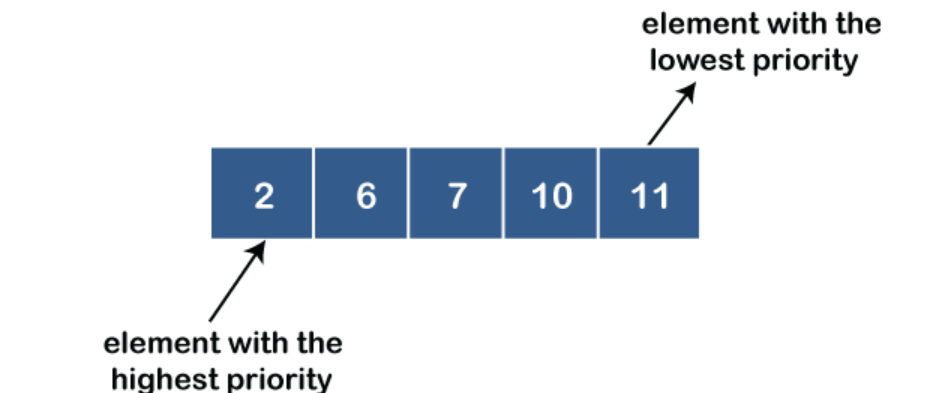
**#####################################**

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

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

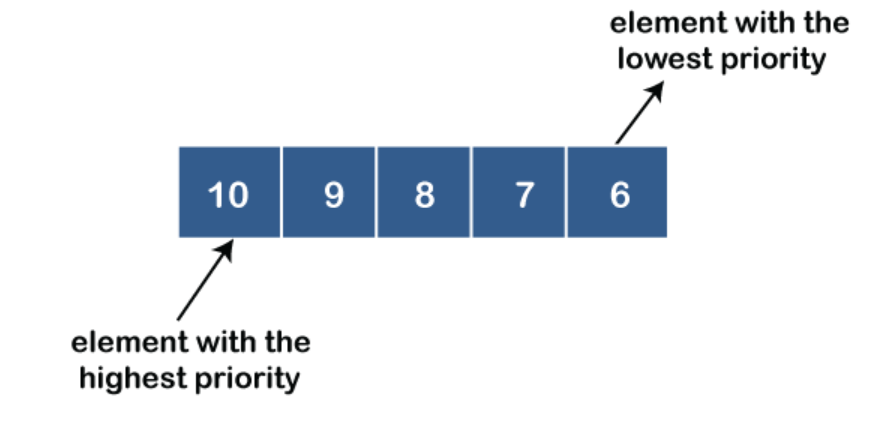
**Ascending order priority queue:**

Data are stored in ascending order and data with lowest value is supposed to have highest priority.



**Descending order priority queue:**

Data are stored in descending order and data with highest value is supposed to have highest priority.



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

# Tree Data Structure #

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

**What are Trees?**

A tree is a Hierarchical data structure that naturally hierarchically stores the information. The Tree data structure is one of the most efficient and mature. The nodes connected by the edges are represented.

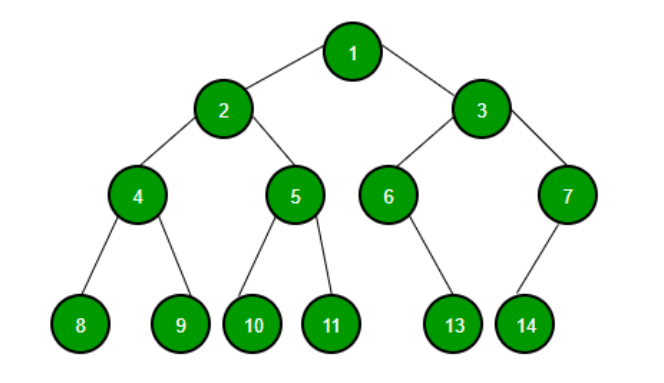
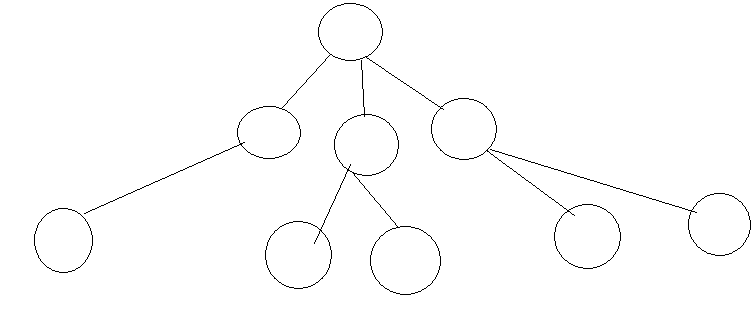
**Properties of Tree**

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/tree
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**

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.

**Internal 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.

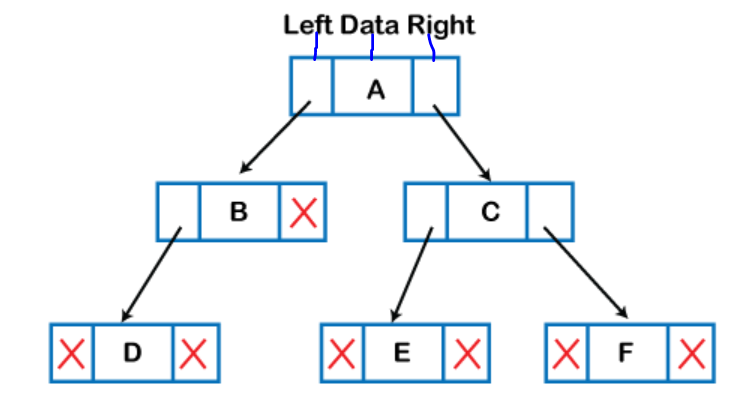
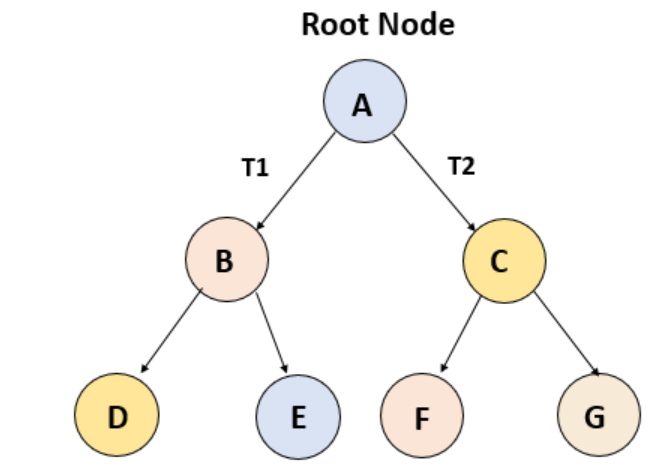
**Descendant**

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

**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.

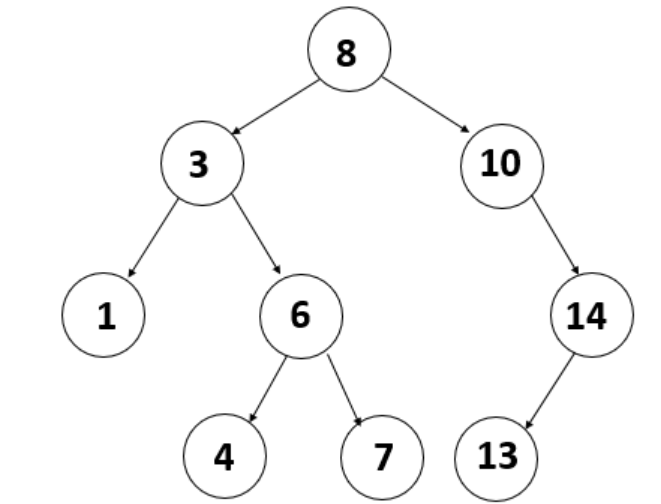
When certain constraints and characteristics are applied in a Binary tree, a number of others such as AVL tree, BST (Binary Search Tree), RBT tree, etc.



**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 should in BST be less than or equal to the parent value.
2. The right child value should always be greater than or equal to the parent’s value



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

# 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 ifNodeExists(self, key):

        if (self.data == key):

            return True

        p=self.children

        if p:

            # node found, no need to look further

            if (p[0]).ifNodeExists(key):#check first data/left data equal to searching data

                return True

            #res2 = (p[1]).ifNodeExists(key)

elif (p[1]).ifNodeExists(key): #checking for right/seconda data

                return True

        else:

            return False

Logic:

1. We have defined the ifNodeExists() , which will be called on any node instance and will search for data from that node to downwards (in each child).
2. If (self.data == key) --- this checks if current node data is same data what we want to search. If True then return True and stops execution.
3. if (p[0]).ifNodeExists(key):#--- calling again same method which will check for left/first data equal to searching data
4. elif (p[1]).ifNodeExists(key): # calling again same method which will check for right/second data equal to searching data

**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.

**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 in compare 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 compare 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 allows so will use left and right as instance variable instead of parent and children.

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

**Logic:**

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

**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**

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**

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

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

**Post Order traversing**

#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.

**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)**

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.

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

**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

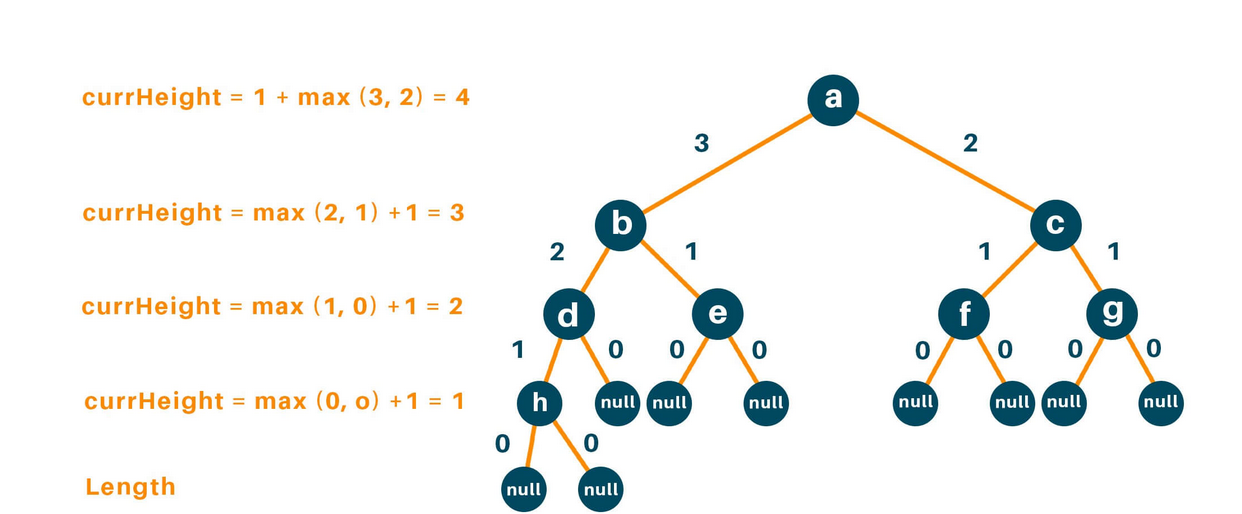
**Finding height of BST/BT:**

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

1. Depth first approach
2. Breach first approach

**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

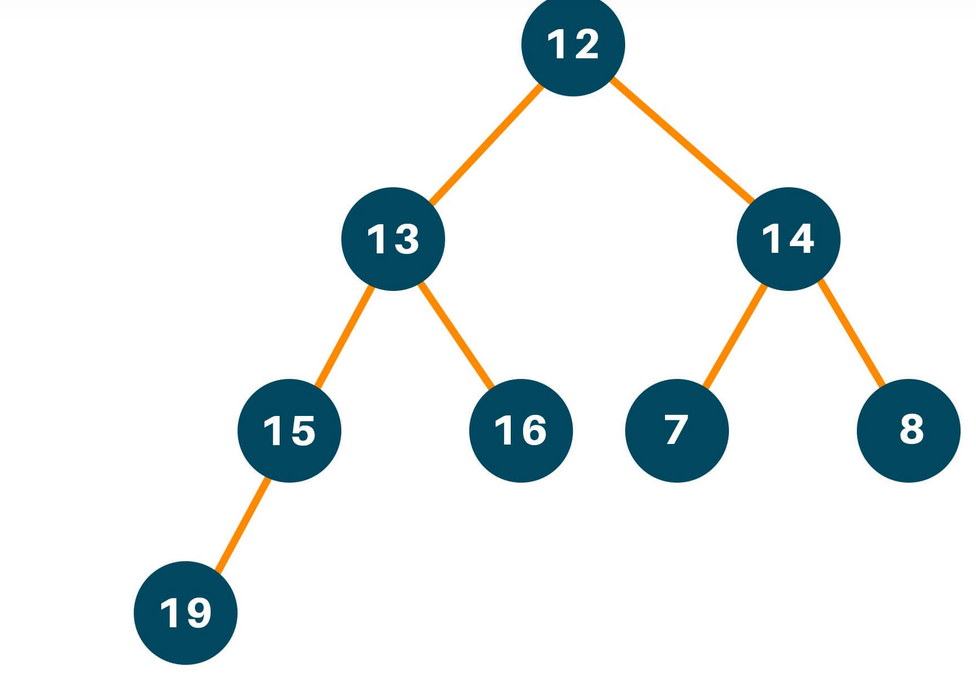
    leftAns = height(root.left)

    rightAns = height(root.right)

    # Return max(leftHeight, rightHeight) at each iteration

    return max(leftAns, rightAns) + 1

**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