DATA STRUCTRES!

Use:

It reduces time and space complexity.

LINEAR SEARCH:

searching the value one by one. it requires more time.

```
l=[1,2,3,4,5,6,7,8,9,10]
for i in l:
    if i==7:
        print("found")
        break
else:
    print("not found")
```

BINARY SEARCH:

→use 2 pointers to search

→one pointing to first number(left) - increment it

i=0   #left=0

→another to the last one(right) - decrement it

*for loop : when we know how many times to run it

*while loop : we do not know how many times but know the condition

*when we use elif : only one condition will run

*when we use multiple if : all conditions will run

```
l=[1,17,3,4,10,8,12,22]
l.sort()
print(l)
s=8
i=0
j=len(l)-1
while i<j:
    mid=i+j//2
    if l[mid]==s:
        print(mid,"found")
        break
    elif l[mid]>s:
        j=mid-1
    else:
        i=mid+1
else:
    print("not found")
```

TIME COMPLEXITY:

1. Best [O(1)]:

searching in the first and getting the first value.

2. average [O(n)]

it depends the value of n

3. Worst [O(n)]

only one loop is used

SPACE COMPLEXITY: O(1)

memory space will be fixed.

O(log(n)):

is used when the size is decreasing by half.

n is number of elements.

SORTING:

the 5 sorting techniques are:

1. bubble
2. merge
3. quick
4. selection
5. insert

1. BUBBLE SORT:

best case scenario for sorting is when the list of elements are already sorted.

- the greater number will move to the last

```
l=[9,7,78,10,5,1,0]
for i in range(0,len(l)-1):
    for j in range(0,len(l)-i-1):
        if l[j]>l[j+1]:
            l[j],l[j+1]=l[j+1],l[j] #swap {a,b=b,a}
print(l)
```

2. SELECTION SORT:

```
b=[2,6,8,4,19,5,44]
for i in range(0,len(b)-1):
    m=i
    for j in range(i+1,len(b)):
        if b[m]>=b[j]:
            m=j
    b[i],b[m]=b[m],b[i]
print(b)
```

3. INSERTION SORT:

→always starts with second number

while sorting:

→the sorted array will be on left(backward)

→unsorted array will be on right(forward)

⇛i will be all the forward elements

⇛j will be all the backward elements

```
b=[2,6,8,4,19,5,44]
for i in range(1,len(b)):
    j=i-1
    a=b[i]
    while j>=0 and b[j]>a:
        b[j+1]=b[j]
        j-=1
    b[j+1]=a
print(b)
```

4. MERGE SORT:

- works on divide and conquer rule.

→break the given array by dividing it until we get a singular element

this is done by using the condition beginning<ending (base condition)

```
#merge sort
def merge(arr,mid,beg,end):
    n1=mid-beg+1
    n2=end-mid
    i=j=0
    left=arr[beg:mid+1]
    right=arr[mid+1:end+1]
    k=beg
    while i<n1 and j<n2:
        if left[i]<right[j]:
            arr[k]=left[i]
            i+=1
        else:
            arr[k]=right[j]
            j+=1
        k+=1
    while i<n1:
        arr[k]=left[i]
        k+=1
        i+=1
    while j<n2:
        arr[k]=right[j]
        k+=1
        j+=1
```

5. QUICK SORT:

it is the most effective sorting technique.

follows 2 pointer approach (start and end pointers)

```
def partition(arr,low,high):
    pivot=arr[low]
    start=low+1
    end=high
    while True:
        while start<=end and arr[start]<=pivot:
            start+=1
        while start<=end and arr[end]>pivot:
            end-=1
        if start<end:
            arr[start],arr[end]=arr[end],arr[start]
        else:
            break
    arr[low],arr[end]=arr[end],arr[low]
    return end
def quicksort(arr,beg,end):
    if beg<end:
        p=partition(arr,beg,end)
        quicksort(arr,beg,p-1)
        quicksort(arr,p+1,end)
a=[8,7,6,1,4,5,2,3]
b=0
e=len(a)-1
quicksort(a,b,e)
print(a)
```

STACKS AND QUEUES:

STACKS:

stacks and queues can be implemented using list.

stacks and queues are implemented using oops concept.

for implementing stack we should create a class and inside a class constructor is used.

terminologies for implementing stack:

push, pop, and peek(topmost element of stack)

code for implementing stack:

```
class stack:
    def __init__(self):
        self.top=-1 #top value can also be given using negative
        self.size=5 #just like how we give MAX in C
        self.list=[]
    def push(self,a):
        if len(self.list)>=5:
            print("list is full")
            return 0
        self.top+=1
        self.list.append(a)
    def pop(self):
        if len(self.list)==0:
            print("list is empty")
            return 0
        self.top-=1
    def peek(self):
        print(self.list)
        if len(self.list)==0:
            print("list is empty")
            return 0
        elif self.top>5:
            print("out of index")
        else:
            print(self.list[self.top])
s=stack()
s.push(1)
s.push(2)
s.push(3)
s.push(4)
s.peek()
s.pop()
s.peek()
```

```
s.pop()
s.peek()
```

QUEUES:

```
class queue:
    def __init__(self):
        self.front=-1 #top value can also be given using negati
        self.rear=-1
        self.size=5 #just like how we give MAX in C.
        self.list=[]
    def enqueue(self,a):
        if len(self.list)>=5:
            print("list is full")
            return 0
        self.rear+=1
        self.list.append(a)
        if self.front==-1:
            self.front=0
    def dequeue(self):
        if len(self.list)==0:
            print("list is empty")
            return 0
        elif self.front>self.rear:
            print("list is empty")
        else:
            self.list.pop(0)
            self.front+=1
        print(self.list)
    def display(self):
        print(self.list)
        if len(self.list)==0:
            print("list is empty")
            return 0
```

```
s=queue()
s.enqueue(1)
s.enqueue(2)
s.enqueue(3)
s.enqueue(4)
s.enqueue(5)
s.display()
s.dequeue()
s.dequeue()
s.dequeue()
s.dequeue()
s.dequeue()
```

#evaluation of postfix expression:

```
s="5678+-*"
s1=[]
for i in s:
    if i.isdigit():
        s1.append(int(i))
    else:
        op2=s1.pop()
        op1=s1.pop()
        if i=="+":
            s1.append(op1+op2)
        elif i=="-":
            s1.append(op2-op1)
        elif i=="*":
            s1.append(op1*op2)
        elif i=="/":
            s1.append(op1/op2)
print(s1)
```

#Implement queue using stack:

in queue we delete elements from 0th index but in stack we delete from the top.

```
l=[1,2,3,4]
s=[]
def pop():
    for i in range(len(l)):
        s.append(l.pop())
    s.pop()
    for i in range(len(s)):
        l.append(s.pop())
pop()
pop()
print(l)
```

OR

```
class MyQueue:
    def __init__(self):
        self.s1=[]
        self.s2=[]
    def push(self, x: int) -> None:
        self.s1.append(x)
    def pop(self) -> int:
        for i in range(len(self.s1)):
            self.s2.append(self.s1.pop())
        a=self.s2.pop()
        for i in range(len(self.s2)):
            self.s1.append(self.s2.pop())
        return a
    def peek(self) -> int:
        return self.s1[0]
    def empty(self) -> bool:
        return len(self.s1)==0
```

LINKED LIST:

Linked lists:
why linked list when there is set, list etc.?

1. time complexity
2. space complexity

linked list is represented by nodes
node contains 2 sections:

1. data section
2. address section(address of next node)

#insert beginning and delete beginning

```
class Node:
    def _init_(self,v):
        self.data=v
        self.next=None #NULL
        #make sure u understand each and every line sowmyaa!!!!
class linkedlist:
    def _init_(self):
        self.head=None
    def insertbeg(self,v):
        nn=Node(v)
        if self.head==None:
            self.head=nn
        else:
            nn.next=self.head
            self.head=nn
    def delbeg(self):
        if self.head==None:
            print("list empty")
        else:
            temp=self.head
            self.head=temp.next
    def display(self):
        curr=self.head
        while curr!=None:
            print(curr.data)
            curr=curr.next
        print("null")
l=linkedlist()
l.insertbeg(10)
l.insertbeg(2)
l.insertbeg(8)
l.display()
l.delbeg()
l.display()
```

#insert end and delete end

```
class Node:
    def __init__(self,v):
        self.data=v
        self.next=None #NULL
        #make sure u understand each and every line sowmyaa!!!!
class linkedlist:
    def __init__(self):
        self.head=None
    def insertend(self,v):
        nn=Node(v)
        if self.head==None:
            self.head=nn
        else:
            curr=self.head
            while curr.next!=None:
                curr=curr.next
            curr.next=nn
    def delend(self):
        if self.head==None:
            print("list empty")
        else:
            curr=prev=self.head
            while curr.next!=None:
                prev=curr
                curr=curr.next
            prev.next=None
    def display(self):
        curr=self.head
        while curr!=None:
            print(curr.data,"->",end=" ")
            curr=curr.next
        print("null")
l=linkedlist()
l.insertend(10)
l.insertend(2)
l.insertend(8)
l.insertend(11)
l.display()
l.delend()
l.display()
```

#insert anywhere and delete anywhere

```
class Node:
    def _init_(self,v):
        self.data=v
        self.next=None #NULL
        #make sure u understand each and every line sowmyaa!!!!
class linkedlist:
    def _init_(self):
        self.head=None
    def insertend(self,v):
        nn=Node(v)
        if self.head==None:
            self.head=nn
        else:
            curr=self.head
            while curr.next!=None:
                curr=curr.next
            curr.next=nn
    def insertany(self,v,k):
        nn=Node(v)
        if self.head==None:
            print("list is empty")
        else:
            curr=self.head
            while curr!=None:
                if curr.data==k:
                    nn.next=curr.next
                    curr.next=nn
                    break
                curr=curr.next
    def search(self,k):
        curr=self.head
        while curr!=None:
            if curr.data==k:
                print("found")
                break
            curr=curr.next
        else:
            print("not found")
    def delany(self,k):
        if self.head==None:
            print("list empty")
        else:
            curr=prev=self.head
            while curr!=None:
                if curr.data==k:
                    prev.next=curr.next
                    break
                prev=curr
                curr=curr.next
            else:
                print("key not found")
    def display(self):
        curr=self.head
        while curr!=None:
            print(curr.data,"->",end=" ")
            curr=curr.next
        print("null")
l=linkedlist()
l.insertend(10)
l.insertend(2)
l.insertend(8)
l.insertend(11)
l.insertany(17,2)
l.display()
l.search(8)
l.delany(8)
l.display()
```

#insert at middle of the list using count

```
class Node:
    def __init__(self,value):
        self.data=value
        self.next=None
class linkedlist:
    def __init__(self):
        self.head=None
    def insertatbeg(self,value):
        newnode=Node(value)
        if self.head==None:
                self.head=newnode
        else:
```

```
                 newnode.next=self.head
                 self.head=newnode
     def insertmid(self,value):
         newnode=Node(value)
         c=0
         if self.head==None:
             self.head=newnode
         elif self.head.next==None:
              self.head.next=newnode
         else:
             curr=self.head
             while curr!=None:
                 c+=1
                 curr=curr.next
             curr=self.head
             for i in range(c//2):
                  curr=curr.next
             newnode.next=curr.next
             curr.next=newnode
     def printlist(self):
         curr=self.head
         while(curr!=None):
                 print(curr.data,"->",end="")
                 curr=curr.next
         print("null")
 l=linkedlist()
 l.insertatbeg(1)
 l.insertatbeg(2)
 l.insertatbeg(3)
 l.insertatbeg(8)
 l.insertmid(4)
 l.printlist()
```

#insert at middle of the list without using count

```
def insertmid(self,val):
        newnode=node(val)
        if self.head==None:
            self.head=newnode
        elif self.head.next==None:
            self.head.next=newnode
        else:
            fast=self.head
            slow=self.head
            while fast.next!=None and fast.next.next!=None:
                fast=fast.next.next
                slow=slow.next
            newnode.next=slow.next
            slow.next=newnode
```

#reverse of a linked list

Trees:

#creating a node in a normal tree

```
class node:
    def __init__(self,v):
        self.left=None
        self.right=None
        self.data=v
def inorder(root):
    if root:
        inorder(root.left)
        print(root.data)
        inorder(root.right)
```

```
r=node(1)
r.left=node(2)
r.right=node(3)
r.left.left=node(4)
r.left.right=node(5)
inorder(r)
```

we create a function to print the contents of the tree instead of method because:

since there is no root inside node class, for traversing we need to pass root as parameter hence we create a function for traversing not a method.

Binary search tree:

```
class node:
    def __init__(self,v):
        self.left=None
        self.right=None
        self.data=v
class trees: #in trees root is the only variable we can have
    def __init__(self):
        self.root=None
    def insert(self,value):
        nn=node(value)
        if self.root is None:
            self.root=nn
        else:
            curr=self.root
            while True:
                if value<=curr.data:
                    if curr.left==None:
                        curr.left=nn
                        break
                    else:
                        curr=curr.left
                else:
                    if curr.right==None:
                        curr.right=nn
                        break
                    else:
                        curr=curr.right
    def preorder(self,root):
        if root:
            print(root.data)
            self.preorder(root.left)
            self.preorder(root.right)

def inorder(root):
    if root:
        inorder(root.left)
        print(root.data)
        inorder(root.right)
def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.data)
r=trees()
r.insert(2)
r.insert(8)
r.insert(1)
r.preorder(r.root)
print(" ")
inorder(r.root)
print(" ")
postorder(r.root)
```

Graphs:

It is a non primitive no linear data structure which contains edges and vertices.

Applications of graphs:

1. maps
2. social media apps

#adjacency matrix

```
class Graph:
    def __init__(self):
        self.matrix=[[0]*5 for i in range(5)]
        print(self.matrix)
    def addvertex(self,a,b):
        self.matrix[a][b]=1
    def print(self):
        for i in self.matrix:
            print(i)
g=Graph()
g.addvertex(1,2)
g.addvertex(4,2)
g.addvertex(1,4)
g.addvertex(2,3)
g.addvertex(4,3)
g.print()
```

O/P:

```
[[0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0], [0, 0, 0, 0, 0]]
[0, 0, 0, 0, 0]
[0, 0, 1, 0, 1]
[0, 0, 0, 1, 0]
[0, 0, 0, 0, 0]
[0, 0, 1, 1, 0]
```

#adjacency list using dictionary

```
class Graph:
    def __init__(self):
        self.matrix={}
    def addvertex(self,a,b):
        if a not in self.matrix:
            self.matrix[a]=[b]
        else:
            self.matrix[a].append(b)
    def print(self):
        print(self.matrix)
g=Graph()
g.addvertex(1,2)
g.addvertex(4,2)
g.addvertex(1,4)
g.addvertex(2,3)
g.addvertex(4,3)
g.print()
```

O/P:

{1: [2, 4], 4: [2, 3], 2: [3]}

#bfs

```
class Graph:
    def __init__(self):
        self.matrix={}
    def addvertex(self,a,b):
        if a not in self.matrix:
            self.matrix[a]=[b]
        else:
            self.matrix[a].append(b)
    def print(self):
        print(self.matrix)
    def bfs(self,data):
        v=[]
        q=[data]
        while q:
            vertex=q.pop(0)
            print(vertex)
            if vertex in self.matrix:
                for i in self.matrix[vertex]:
                    if i not in v:
                        v.append(i)
                        q.append(i)
g=Graph()
g.addvertex(1,2)
g.addvertex(4,2)
g.addvertex(1,4)
g.addvertex(2,3)
g.addvertex(4,3)
g.print()
g.bfs(1)
```

#dfs

```
class Graph:
    def __init__(self):
        self.matrix={}
    def addvertex(self,a,b):
        if a not in self.matrix:
            self.matrix[a]=[b]
        else:
            self.matrix[a].append(b)
    def print(self):
        print(self.matrix)
    def dfs(self,data):
        v=[]
        q=[data]
        while q:
            vertex=q.pop()
            print(vertex)
            if vertex in self.matrix:
                for i in self.matrix[vertex]:
                    if i not in v:
                        v.append(i)
                        q.append(i)
```

```
g=Graph()
g.addvertex(1,2)
g.addvertex(4,2)
g.addvertex(1,4)
g.addvertex(2,3)
g.addvertex(4,3)
g.print()
g.dfs(1)
```