DSA(Python)

1. Binary Search

In this code, the list l is sorted first. Then, a binary search is performed for the element s=2. If the element is found in the list, its index is printed, else "not found" is printed.

```
l=[1,2,3,4,2,3,2,4,5,4,4]
l.sort()
i=0
s=2
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
    else:
        i=mid
else:
    print("not found")
```

2. Linear Search

Linear search is a simple search algorithm that checks every element in a list or an array sequentially until the desired element is found or all elements have been checked. It does not require the list to be sorted.

```
l=[1,2,3,4,5,6,7,8]
for i in l:
    if i==8:
        print("found")
        break
```

```
else:
    print("not found")
```

3. Sorting

In Python, sorting refers to arranging data in a particular format. It can be in ascending (increasing) or descending (decreasing) order. Python provides several methods to sort data structures, such as lists. The simplest one is the `sort()` method, which modifies the list it is called on. Another method is `sorted()` which builds a new sorted list from an iterable. Sorting helps organize and structure data efficiently, which can be beneficial in search algorithms, data analysis, and more.

The types of sorting in Python covered in this document are:

1. Built-in `sort()` method
2. Bubble Sort
3. Selection Sort
4. Insertion Sort
5. Merge Sort
6. Quick Sort

Sorting

```
l=[2,3,4,5,6,7,9]
l.sort()
print(l)
```

4. Bubble Sort

Bubble Sort is a simple sorting algorithm in Python that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. The pass through the list is repeated until the list is sorted. The algorithm gets its name from the way smaller or larger elements "bubble" to the top of the list.

```
l=[1,2,3,4,2,3,2,4,5,4,4]
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]
print(l)
```

5. Selection Sort

Selection Sort is a simple sorting algorithm that maintains two subarrays in a given array. The first subarray is always sorted, while the second subarray remains unsorted. In every iteration, the smallest (or largest, depending on sorting order) element from the unsorted subarray is picked and moved to the sorted subarray. This process continues until the entire array is sorted.

```
b=[6,5,4,3,2]
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)
```

6. Insertion Sort

Insertion sort is a sorting algorithm in Python where the element to be sorted is considered as one and then compared with the other elements in the sequence. If the current element is smaller than the preceding one, it is swapped. This process continues moving towards the left until the correct position for the current element is found. The algorithm then moves to the next element in the sequence and repeats the process until the entire sequence is sorted.

```
b=[3,4,5,6,19,21,10]
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)
```

7. Merge Sort

Merge sort is a divide-and-conquer algorithm that splits a list into two halves, sorts them, and then merges them. First, the list is split into the smallest unit (1 element), then each element is continually merged back into larger and larger lists, maintaining the sorted order until the list is entirely reassembled.

```
def merge(arr,beg,mid,end):
    n1=mid-beg+1
    n2=end-mid#temp array size
    i=j=0
    left=arr[: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
def mergesort(arr,beg,end):
    if beg<end:
        mid=(beg+end)//2
```

```
        #recursion takes place
        mergesort(arr,beg,mid)
        mergesort(arr,mid+1,end)#left part
        merge(arr,beg,mid,end)
a=[8,7,6,4,3,2,1]
b=0
e=len(a)-1
mergesort(a,b,e)
print(a)
```

8. Quick Sort

Quick sort is a highly efficient sorting algorithm in Python. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. The process continues until the entire array is sorted.

```
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=[8,7,6,4,5,2,1,3]
b=0
e=len(a)-1
quicksort(a,b,e)
print(a)
```

9. Evaluation of postfix expression

Postfix expression evaluation in Python refers to the process of calculating the value of an expression written in postfix notation (also known as Reverse Polish Notation). In postfix notation, operators are placed after their operands. For example, the expression "2 3 +" in postfix notation equals to "2 + 3" in infix notation.

The algorithm for evaluating a postfix expression in Python works as follows:

1. Process each element in the expression from left to right.
2. If the element is a number, push it into a stack.
3. If the element is an operator, pop the top two elements from the stack, perform the operation, and push the result back into the stack.
4. Continue this process until all elements in the expression are processed.
5. The final value left in the stack is the result of the postfix expression.

Here is a Python code

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

```
            a.append(op2-op1)
        elif i=="*":
            a.append(op1*op2)
        elif i=="/":
            a.append(op1/op2)
print(a)
```

In this code, `l` is the postfix expression to be evaluated. The stack is implemented using a list `a`. The code loops over each character in the string `l`. If the character is a digit, it is converted to an integer and pushed into the stack. If it is an operator, two elements are popped from the stack and the operation is performed, then the result is pushed back into the stack. Finally, the code prints the result which is the final value left in the stack.

10. Implement Queue using Stack

Implementation of Queue using Stack: In this approach, a queue is implemented using two stacks. First, all elements are pushed into the first stack. When a pop operation is performed, all elements are moved to the second stack and the top element is popped. This way, the last inserted item is always at the top, mimicking queue behavior. Here's the Python code:

```
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)
```

In this code, `l` is the main stack and `s` is the auxiliary stack. The function `pop()` is responsible for implementing the dequeue operation. It pops all elements from `l` and pushes them into `s`, then pops the top element from `s`, effectively removing the first inserted element from `l`. After that, it pushes all elements back from `s` to `l`.

11. Implementation of Stack using Queue

Linked List

A linked list in Python is a linear data structure, where each element is a separate object (node) consisting of two parts: the data and a reference to the next node. Unlike arrays, linked lists do not have a fixed size and can grow and shrink during runtime. Here's a simple implementation:

1. Code for Front Insert

```
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 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(4)
l.printlist()
```

2. End Insert

```
class Node:
    def __init__(self,value):
        self.data=value
        self.next=None
class linkedlist:
    def __init__(self):
        self.head=None
    def insertatend(self,value):
        newnode=Node(value)
        if self.head==None:
            self.head=newnode
        else:
            curr=self.head
            while curr.next!= None:
                curr=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.insertatend(1)
l.insertatend(2)
l.insertatend(3)
l.insertatend(4)
l.printlist()
```

3. Searching

```
class Node:
    def __init__(self,value):
        self.data=value
        self.next=None
class linkedlist:
    def __init__(self):
        self.head=None
    def insertatend(self,value):
        newnode=Node(value)
        if self.head==None:
            self.head=newnode
        else:
            curr=self.head
            while curr.next!= None:
                curr=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.insertatend(1)
l.insertatend(2)
l.insertatend(3)
l.insertatend(4)
l.printlist()
```

4. Insert Anywhere

```
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 insertatend(self,value):
        newnode=Node(value)
        if self.head==None:
            self.head=newnode
        else:
            curr=self.head
            while curr.next!=None:
                curr=curr.next
            curr.next=newnode
    def search(self,value):
        curr=self.head
        while curr!=None:
             if curr.data==value:
                 print(value," element found")
                 break
             curr=curr.next
        else:
            print("element not found")
    def insertany(self,svalue,value):
        newnode=Node(value)
        curr=self.head
        while curr!=None:
             if curr.data==svalue:
                 newnode.next=curr.next
```

```
                curr.next=newnode
                break
             curr=curr.next
        else:
            print("element not found")

    def printlist(self):
        if self.head==None:
            print("no node present")
        else:
            curr=self.head
            while curr!=None:
                print(curr.data,"->",end=" ")
                curr=curr.next
l=linkedlist()
l.insertatbeg(1)
l.insertatbeg(2)
l.insertatbeg(3)
l.insertatbeg(4)
l.insertatend(5)
l.insertatend(6)
l.search(3)
l.insertany(3,8)
l.printlist()
```

5. Insert Middle

```
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 insertatend(self,value):
        newnode=Node(value)
        if self.head==None:
            self.head=newnode
        else:
            curr=self.head
            while curr.next!=None:
                curr=curr.next
            curr.next=newnode
    def searching(self,key):
        curr=self.head
        while curr!=None:
            if curr.data==key:
                print(key, "is found")
                break
            curr=curr.next
        else:
            print(key, "is not found")
    def count(self):
        count=0
        if self.head==None:
            print("no.of nodes=0")
        else:
            curr=self.head
            while curr!=None:
                count+=1
                curr=curr.next
        print(count)
    def insertinganywhere(self,value,key):
        newnode=Node(key)
```

```
        curr=self.head
        while curr!=None:
            if curr.data==value:
                newnode.next=curr.next
                curr.next=newnode
                break
            curr=curr.next
        else:
            print("element not found")
    def insertatmid(self,value):
        newnode=Node(value)
        count=0
        if self.head.next==None:
            self.head=newnode
        elif self.head.next==None:
            self.head.next=newnode
        else:
            curr=self.head
            while curr!=None:
                count+=1
                curr=curr.next
            curr=self.head
            for i in range(count//2):
                curr=curr.next
            newnode.next=curr.next
            curr.next=newnode
    def middle(self,value):
        newnode=Node(value)
        count=0
        if self.head.next==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
    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(4)
l.insertatend(5)
l.insertatend(6)
l.searching(7)
l.insertinganywhere(3,8)
l.insertatmid(9)
l.middle(7)
l.count()
l.printlist()
```

Trees:

Trees in Python are data structures that model a hierarchical structure. Each element in the tree, known as a node, connects to one or more nodes. There is a root node and each connection represents a relationship between nodes. Each node can have any number of child nodes, but only one parent node. Trees are used for various applications, such as file systems, HTML DOM, and AI algorithms. The two main types of trees are Binary Trees and Binary Search Trees.

1. Construct tree of inorder , preorder and postorder

```
class node:
    def __init__(self,data):
        self.left=None
        self.data=data
        self.right=None
def inorder(root):
    if root:
        inorder(root.left)
        print(root.data,end=" ")
        inorder(root.right)
def preorder(root):
    if root:
        print(root.data,end=" ")
        preorder(root.left)
        preorder(root.right)
def postorder(root):
    if root:
        postorder(root.left)
        preorder(root.right)
        print(root.data,end=" ")
r=node(1)
r.left=node(2)
r.right=node(3)
r.left.left=node(4)
r.left.right=node(5)
inorder(r)
print()
preorder(r)
print()
postorder(r)
```

Output

4 2 5 1 3
1 2 4 5 3
4 5 2 3 1

2. Binary Search Tree

```
class node:
    def _init_(self,data):
        self.left=None
        self.data=data
        self.right=None
class trees:
    def _init_(self):
        self.root=None
    def insert(self,value):
        newnode=node(value)
        if self.root is None:
            self.root=newnode
        else:
            curr=self.root
            while True:
                if value<=curr.data:
                    if curr.left is None:
                        curr.left=newnode
                        break
                    else:
                        curr=curr.left
                else:
                    if curr.right is None:
                        curr.right=newnode
                        break
                    else:
                        curr=curr.right
def inorder(root):
        if root:
            inorder(root.left)
            print(root.data,end=" ")
            inorder(root.right)
def preorder(root):
```

```
        if root:
            print(root.data,end=" ")
            preorder(root.left)
            preorder(root.right)
def postorder(root):
        if root:
            postorder(root.left)
            postorder(root.right)
            print(root.data,end=" ")

r=trees()
r.insert(5)
r.insert(2)
r.insert(3)
r.insert(9)
preorder(r.root)
print()
postorder(r.root)
print()
inorder(r.root)
print()
```

Graphs :

1. create a 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(1,3)
g.addvertex(1,4)
g.addvertex(2,3)
g.addvertex(2,4)
g.addvertex(3,2)
g.print()
```

Output

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

2. Create a Adjacency List

```
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(1,3)
g.addvertex(1,4)
g.addvertex(2,3)
g.addvertex(2,4)
```

```
g.addvertex(3,2)
g.print()
```

Output

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

3. Breadth First Search(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):
        visited=[]
        queue=[data]
        while queue:
            vertex=queue.pop(0)
            print(vertex)
            if vertex in self.matrix:
                for i in self.matrix[vertex]:
                    if i not in visited:
                        visited.append(i)
                        queue.append(i)

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

```
g.addvertex(3,2)
g.print()
g.bfs(1)
```

Output

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

4. Depth First Search(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):
        visited=[]
        stack=[data]
        while stack:
            vertex=stack.pop()
            print(vertex)
            if vertex in self.matrix:
                for i in self.matrix[vertex]:
                    if i not in visited:
                        visited.append(i)
                        stack.append(i)

g=Graph()
```

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

Output

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