**DSA**

**Binary search:**

Time complexity: O(log(N)

arr = [int(x) for x in input("Enter the sorted list of numbers seperated by the spaces:").split()]

target = int(input("Enter the target value to serach:"))

start, end = 0, len(arr) - 1

found = False

while start <= end:

mid = (start + end) // 2

if arr[mid] == target:

print(f"Target {target} found at index {mid}.")

found = True

break

elif arr[mid] < target:

start = mid + 1

else:

end = mid - 1

if not found:

print(f"Target {target} not found in the array.")

**output:**

Enter the sorted list of numbers separated by the spaces:1 2 5 7 9 12 15

Enter the target value to serach:7

Target 7 found at index 3.

**Linear search:**

Time complexity: O(N)

**Bubble Sort:**

Compare the each and every element and then swap the elements.

def bubble\_sort(arr):  
 n=len(arr)  
  
 for i in range(n):  
 for j in range(0,n-i-1):  
 if arr[j]>arr[j+1]:  
 arr[j],arr[j+1]=arr[j+1],arr[j]  
my\_list=[64,34,25,12,32,11,90]  
print("Original\_List:",my\_list)  
bubble\_sort(my\_list)  
print("Sorted List:",my\_list)

**output:**

Original\_List: [64, 34, 25, 12, 32, 11, 90]

Sorted List: [11, 12, 25, 32, 34, 64, 90]

**Selection Sort:**

We can compare the smallest elements in the list and sort or swap the elements in this sort.

def selection\_sort(arr):  
 n=len(arr)  
  
 for i in range(n):  
 min\_index = i  
 for j in range(i + 1, n):  
 if arr[j] < arr[min\_index]:  
 min\_index = j  
  
 arr[i], arr[min\_index] = arr[min\_index], arr[i]  
  
my\_list=[64,34,25,12,32,11,90]  
print("Original\_List:",my\_list)  
selection\_sort(my\_list)  
print("Sorted List:",my\_list)

**output:**

Original\_List: [64, 34, 25, 12, 32, 11, 90]

Sorted List: [11, 12, 25, 32, 34, 64, 90]

**Insertion Sort:**

def insertion\_sort(arr):  
 n = len(arr)  
  
 for i in range(1, n):  
 key = arr[i]  
  
 j = i - 1  
 while j >= 0 and key < arr[j]:  
 arr[j + 1] = arr[j]  
 j -= 1  
  
 arr[j + 1] = key  
  
my\_list = [5, 2, 4, 6, 1, 3]  
  
print("Original List:", my\_list)  
insertion\_sort(my\_list)  
print("Sorted List:", my\_list)

**output:**

Original List: [5, 2, 4, 6, 1, 3]

Sorted List: [1, 2, 3, 4, 5, 6]

**Merge sort:**

def merge\_sort(arr):  
 if len(arr) > 1:  
 mid = len(arr) // 2  
 left\_half = arr[:mid]  
 right\_half = arr[mid:]  
  
 merge\_sort(left\_half)  
 merge\_sort(right\_half)  
  
 merge(arr, left\_half, right\_half)  
  
  
def merge(arr, left, right):  
 i = j = k = 0  
  
 while i < len(left) and j < len(right):  
 if left[i] < right[j]:  
 arr[k] = left [i]  
 i += 1  
 else:  
 arr[k] = right[j]  
 j += 1  
 k += 1  
  
 while i < len(left):  
 arr[k] = left[i]  
 i += 1  
 k +=1  
  
 while j < len(right):  
 arr[k] = right[j]  
 j += 1  
 k += 1  
  
my\_list = [6, 4, 11, 14, 22, 33, 44, 0]  
print("Original\_List:",my\_list)  
merge\_sort(my\_list)  
print("Sorted List:",my\_list)

**output:**

Original\_List: [6, 4, 11, 14, 22, 33, 44, 0]

Sorted List: [0, 4, 6, 11, 14, 22, 33, 44]

**Linked List:**

class Node:  
 def \_\_init\_\_(self, data): # Corrected the init method  
 self.data = data  
 self.next = None  
  
class LinkedList:  
 def \_\_init\_\_(self): # Corrected the init method  
 self.head = None  
  
 def insertAtBegin(self, data):  
 new\_node = Node(data)  
 if self.head is None:  
 self.head = new\_node  
 else:  
 new\_node.next = self.head  
 self.head = new\_node  
  
 def insertAtIndex(self, data, index):  
 new\_node = Node(data)  
 current\_node = self.head  
 position = 0  
 if position == index:  
 self.insertAtBegin(data)  
 else:  
 while current\_node is not None and position + 1 != index:  
 position += 1  
 current\_node = current\_node.next  
  
 if current\_node is not None:  
 new\_node.next = current\_node.next  
 current\_node.next = new\_node  
 else:  
 print("Index not present")  
  
 def insertAtEnd(self, data):  
 new\_node = Node(data)  
 if self.head is None:  
 self.head = new\_node  
 return  
  
 current\_node = self.head  
 while current\_node.next:  
 current\_node = current\_node.next  
  
 current\_node.next = new\_node  
  
 def remove\_first\_node(self):  
 if self.head is None:  
 return  
  
 self.head = self.head.next  
  
 def remove\_last\_node(self):  
 if self.head is None:  
 return  
  
 if self.head.next is None: # Special case if there is only one element  
 self.head = None  
 return  
  
 current\_node = self.head  
 while current\_node.next.next:  
 current\_node = current\_node.next  
  
 current\_node.next = None  
  
 def remove\_at\_index(self, index):  
 if self.head is None:  
 return  
  
 current\_node = self.head  
 position = 0  
 if position == index:  
 self.remove\_first\_node()  
 else:  
 while current\_node is not None and position + 1 != index:  
 position += 1  
 current\_node = current\_node.next  
  
 if current\_node is not None and current\_node.next is not None:  
 current\_node.next = current\_node.next.next  
 else:  
 print("Index not present")  
  
 def remove\_node(self, data):  
 current\_node = self.head  
  
 if current\_node is not None and current\_node.data == data:  
 self.remove\_first\_node()  
 return  
  
 prev\_node = None  
 while current\_node is not None and current\_node.data != data:  
 prev\_node = current\_node  
 current\_node = current\_node.next  
  
 if current\_node is not None:  
 prev\_node.next = current\_node.next  
 else:  
 print("Node with data not found")  
  
 def sizeofLL(self):  
 size = 0  
 current\_node = self.head  
 while current\_node:  
 size += 1  
 current\_node = current\_node.next  
 return size  
  
 def printLL(self):  
 current\_node = self.head  
 while current\_node:  
 print(current\_node.data)  
 current\_node = current\_node.next  
  
llist = LinkedList()  
  
llist.insertAtEnd('a')  
llist.insertAtEnd('b')  
llist.insertAtBegin('c')  
llist.insertAtEnd('d')  
llist.insertAtIndex(data='g', index=2)  
  
print("Node Data")  
llist.printLL()  
  
print("\nRemove First Node")  
llist.remove\_first\_node()  
llist.printLL()  
  
print("\nRemove Last Node")  
llist.remove\_last\_node()  
llist.printLL()

**output:**

Node Data

c

a

g

b

d

Remove First Node

a

g

b

d

Remove Last Node

a

g

b

Process finished with exit code 0

**Double linked list:**

class Node:  
 def \_\_init\_\_(self, data=None):  
 self.data = data  
 self.next = None  
 self.prev = None  
  
class DoublyLinkedList:  
 def \_\_init\_\_(self):  
 self.head = None  
 self.tail = None  
  
 def is\_empty(self):  
 return self.head is None  
  
 def append(self, data):  
 new\_node = Node(data)  
 if self.is\_empty():  
 self.head = new\_node  
 self.tail = new\_node  
 else:  
 new\_node.prev = self.tail  
 self.tail.next = new\_node  
 self.tail = new\_node  
  
 def prepend(self, data):  
 new\_node = Node(data)  
 if self.is\_empty():  
 self.head = new\_node  
 self.tail = new\_node  
 else:  
 new\_node.next = self.head  
 self.head.prev = new\_node  
 self.head = new\_node  
  
 def delete(self, data):  
 if self.is\_empty():  
 return  
  
 current = self.head  
 while current is not None and current.data != data:  
 current = current.next  
  
 if current is not None:  
 if current.prev is not None:  
 current.prev.next = current.next  
 else:  
 self.head = current.next  
  
 if current.next is not None:  
 current.next.prev = current.prev  
 else:  
 self.tail = current.prev  
  
 def display\_forward(self):  
 elements = []  
 current = self.head  
 while current:  
 elements.append(current.data)  
 current = current.next  
 print(" -> ".join(map(str, elements)))  
  
 def display\_backward(self):  
 elements = []  
 current = self.tail  
 while current:  
 elements.append(current.data)  
 current = current.prev  
 print(" -> ".join(map(str, elements)))  
  
# Creating an instance of the DoublyLinkedList  
my\_doubly\_linked\_list = DoublyLinkedList()  
  
# Append elements to the list  
my\_doubly\_linked\_list.append(1)  
my\_doubly\_linked\_list.append(2)  
my\_doubly\_linked\_list.append(3)  
my\_doubly\_linked\_list.append(0)  
  
# Display the list forward and backward  
my\_doubly\_linked\_list.display\_forward()  
my\_doubly\_linked\_list.display\_backward()  
  
# Delete an element from the list  
my\_doubly\_linked\_list.delete(2)  
  
# Display the list forward and backward again  
my\_doubly\_linked\_list.display\_forward()  
my\_doubly\_linked\_list.display\_backward()

**output:**

1 -> 2 -> 3 -> 0

0 -> 3 -> 2 -> 1

1 -> 3 -> 0

0 -> 3 -> 1

Circular list:

class Node:  
 def \_\_init\_\_(self, data=None):  
 self.data = data  
 self.next = None  
  
class CircularLinkedList:  
 def \_\_init\_\_(self):  
 self.head = None  
  
 def is\_empty(self):  
 return self.head is None  
  
 def append(self, data):  
 new\_node = Node(data)  
 if self.is\_empty():  
 self.head = new\_node  
 new\_node.next = self.head  
 else:  
 current = self.head  
 while current.next != self.head:  
 current = current.next  
 current.next = new\_node  
 new\_node.next = self.head  
  
 def prepend(self, data):  
 new\_node = Node(data)  
 if self.is\_empty():  
 self.head = new\_node  
 new\_node.next = self.head  
 else:  
 current = self.head  
 while current.next != self.head:  
 current = current.next  
 current.next = new\_node  
 new\_node.next = self.head  
 self.head = new\_node  
  
 def delete(self, data):  
 if self.is\_empty():  
 return  
  
 if self.head.data == data:  
 current = self.head  
 while current.next != self.head:  
 current = current.next  
 if self.head.next == self.head: # Only one node in the list  
 self.head = None  
 else:  
 current.next = self.head.next  
 self.head = self.head.next  
 else:  
 current = self.head  
 while current.next != self.head and current.next.data != data:  
 current = current.next  
 if current.next.data == data:  
 current.next = current.next.next  
  
 def display(self):  
 elements = []  
 current = self.head  
 if current:  
 while True:  
 elements.append(current.data)  
 current = current.next  
 if current == self.head:  
 break  
 print(" -> ".join(map(str, elements)))  
  
 def search(self, target):  
 if self.is\_empty():  
 return False  
 current = self.head  
 while True:  
 if current.data == target:  
 return True  
 current = current.next  
 if current == self.head:  
 break  
 return False  
  
# Example usage:  
my\_circular\_linked\_list = CircularLinkedList()  
my\_circular\_linked\_list.append(1)  
my\_circular\_linked\_list.append(2)  
my\_circular\_linked\_list.append(3)  
my\_circular\_linked\_list.prepend(0)  
  
my\_circular\_linked\_list.display() # Output: 0 -> 1 -> 2 -> 3  
  
my\_circular\_linked\_list.delete(2)  
my\_circular\_linked\_list.display() # Output: 0 -> 1 -> 3

output:

0 -> 1 -> 2 -> 3

0 -> 1 -> 3

**Stack:**

class Stack:  
 def \_\_init\_\_(self):  
 self.items = []  
  
 def is\_empty(self):  
 return len(self.items) == 0  
  
 def push(self, item):  
 self.items.append(item)  
  
 def pop(self):  
 if not self.is\_empty():  
 return self.items.pop()  
 else:  
 raise IndexError("pop from an empty stack")  
  
 def peek(self):  
 if not self.is\_empty():  
 return self.items[-1]  
 else:  
 raise IndexError("peek from an empty stack")  
  
 def size(self):  
 return len(self.items)  
  
  
stack = Stack()  
print("Is the stack is empty?", stack.is\_empty())  
stack.push(1)  
stack.push(2)  
stack.push(3)  
print("Stack:", stack.items)  
print("Top of the stack:", stack.peek())  
print("pop:", stack.pop())  
print("Stack after pop:", stack.items)  
print("Is the stack empty?", stack.is\_empty())  
print("Size of the stack:", stack.size())

**output:**

Is the stack is empty? True

Stack: [1, 2, 3]

Top of the stack: 3

pop: 3

Stack after pop: [1, 2]

Is the stack empty? False

Size of the stack: 2

def valid\_paren(input\_str):  
 # Declaraing a stack.  
 stack = []  
 # Iterating over the entire string  
 for paren in input\_str:  
 # If the input string contains an opening parenthesis,  
 # push in on to the stack.  
 if paren == '(' or paren == '[' or paren == '{':  
 stack.append(paren)  
 else:  
 # In the case of valid parentheses, the stack cannot be  
 # be empty if a closing parenthesis is encountered.  
 if not stack:  
 print(input\_str, "contains invalid parentheses.")  
 return  
 else:  
 # If the input string contains a closing bracket,  
 # then pop the corresponding opening parenthesis if  
 # present.  
 top = stack[-1]  
 if paren == ')' and top == '(' or \  
 paren == ']' and top == '[' or \  
 paren == '}' and top == '{':  
 stack.pop()  
 else:  
 print(input\_str, "contains invalid parentheses.")  
 return  
 # Checking the status of the stack to determine the  
 # validity of the string.  
 if not stack:  
 print(input\_str, "contains valid parentheses.")  
 else:  
 print(input\_str, "contains invalid parentheses.")  
  
input1 = "{{}}()[()]"  
input2 = "{][}"  
input3 = "(])"  
valid\_paren(input1)  
valid\_paren(input2)  
valid\_paren(input3)

**stack using linked list:**

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
class Stack:  
 def \_\_init\_\_(self):  
 self.head = None  
  
 def is\_empty(self):  
 return self.head is None  
  
 def push(self, data):  
 new\_node = Node(data)  
 new\_node.next = self.head  
 self.head = new\_node  
  
 def pop(self):  
 if self.is\_empty():  
 return None  
 popped = self.head.data  
 self.head = self.head.next  
 return popped  
  
 def peek(self):  
 if self.is\_empty():  
 return None  
 return self.head.data  
  
 def display(self):  
 current = self.head  
 while current:  
 print(current.data, end=" -> ")  
 current = current.next  
 print("None")  
  
# Example usage:  
stack = Stack()  
stack.push(1)  
stack.push(2)  
stack.push(3)  
  
stack.display()  
  
print("Popped:", stack.pop())  
print("Peek:", stack.peek())  
  
stack.display()

**output:**

3 -> 2 -> 1 -> None

Popped: 3

Peek: 2

2 -> 1 -> None

**Queue:**

Pushing an element.

Enqueue:

Dequeue:

class QueueList:  
 def \_\_init\_\_(self):  
 self.items = []  
  
 def is\_empty(self):  
 return len(self.items) == 0  
  
 def enqueue(self, item):  
 self.items.append(item)  
  
 def dequeue(self):  
 if not self.is\_empty():  
 return self.items.pop(0)  
 else:  
 raise IndexError("dequeue from an empty queue")  
  
 def peek(self):  
 if not self.is\_empty():  
 return self.items[0]  
 else:  
 raise IndexError("peek from an empty queue")  
  
 def size(self):  
 return len(self.items)  
  
queue = QueueList()  
  
queue.enqueue(1)  
queue.enqueue(2)  
queue.enqueue(3)  
  
print("Front of the queue:", queue.peek())  
  
print("Dequeue:", queue.dequeue())  
  
print("Size of the queue:", queue.size())

**output:**

Front of the queue: 1

Dequeue: 1

Size of the queue: 2

**Queue LinkedList:**

class Node:  
 def \_\_init\_\_(self, data):  
 self.data = data  
 self.next = None  
  
  
class QueueLinkedList:  
 def \_\_init\_\_(self):  
 self.front = None  
 self.rear = None  
  
 def is\_empty(self):  
 return self.front is None  
  
 def enqueue(self, item):  
 new\_node = Node(item)  
 if self.is\_empty():  
 self.front = new\_node  
 self.rear = new\_node  
 else:  
 self.rear.next = new\_node  
 self.rear = new\_node  
  
 def dequeue(self):  
 if not self.is\_empty():  
 dequeued\_item = self.front.data  
 self.front = self.front.next  
 if self.front is None:  
 self.rear = None  
 return dequeued\_item  
 else:  
 raise IndexError("dequeue from an empty queue")  
  
 def peek(self):  
 if not self.is\_empty():  
 return self.front.data  
 else:  
 raise IndexError("peek from an empty queue")  
  
 def size(self):  
 count = 0  
 current = self.front  
 while current:  
 count += 1  
 current = current.next  
 return count  
  
  
# Example usage:  
queue = QueueLinkedList()  
queue.enqueue(1)  
queue.enqueue(2)  
queue.enqueue(3)  
  
print("Front of the queue:", queue.peek())  
print("Dequeue:", queue.dequeue())  
print("Size of the queue:", queue.size()

**output:**

Front of the queue: 1

Dequeue: 1

Size of the queue: 2

**Infix:** A+B

**Prefix:** +AB

**Postfix:** AB+

**Infix to postfix:**

from collections import deque  
  
def infix\_to\_postfix(expression):  
 def get\_precedence(op):  
 precedence = {'+': 1, '-': 1, '\*': 2, '/': 2}  
 return precedence.get(op, 0)  
  
 output = []  
 stack = deque()  
  
 for token in expression:  
 if token.isnumeric(): # if the token is an operand (number ie ABC OR 123)  
 output.append(token)  
 elif token == '(': # left parenthesis  
 stack.append(token)  
 elif token == ')': # right parenthesis  
 while stack and stack[-1] != '(':  
 output.append(stack.pop())  
 stack.pop() # remove '(' from stack  
 else: # operator  
 while stack and stack[-1] != '(' and get\_precedence(stack[-1]) >= get\_precedence(token):  
 output.append(stack.pop())  
 stack.append(token)  
  
 # pop all the operators left in the stack  
 while stack:  
 output.append(stack.pop())  
  
 return ''.join(output)  
  
# Example usage  
expression = "3+5\*2/(7-2)"  
print(f"Infix: {expression}")  
print(f"Postfix: {infix\_to\_postfix(expression)}")

**output:**

Infix: 3+5\*2/(7-2)

Postfix: 352\*72-/+

**Error Handling:**

try:  
 result = int(input("Enter a number:"))  
 print(10 / result)  
  
except Exception as e:  
 print("An error Occurred:", e)

**output:**

Enter a number:h

An error Occurred: invalid literal for int() with base 10: 'h'

**Error2: Program**

try:  
 print("Open File")  
 result = 10 / 0  
except ZeroDivisionError:  
 print("Cannot divide by zero")  
else:  
 print("Divivsion Successful, result:", result)  
finally:  
 print("CLose")  
 print("Execution completed")

**output:**

Open File

Cannot divide by zero

Close

Execution completed

**TREE DATA STRUCTURE:**

Tree data structure is a non-linear data structure in which a collection of elements known as nodes are connected to each other via edges such that there exists exactly one path between any two nodes.

Types of tree data structure:

Binary tree: It has maximum of two children linked to it.

Ternary tree: Each node has at most three child nodes.

N-array or generic tree: Consists of large no of child nodes.

Types of Binary trees:

1)Complete binary tree: All the levels are completely filled.

2)Perfect Binary tree: In this tree it should become 0 or either two.

In this the no.of leaf nodes is the no.of internal nodes +1.

**Binary Search Tree.**

All the great number comes on the right side and the lesser number comes on the left side.

**Tree Traversal:**

In-order: Left root right

Pre-order: root left right

Post-order: left right root

**Binary search Tree:**

class TreeNode:  
 def \_\_init\_\_(self, key):  
 self.val = key  
 self.left = None  
 self.right = None  
  
class BST:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, root, key):  
 if root is None:  
 return TreeNode(key)  
 else:  
 if root.val < key:  
 root.right = self.insert(root.right, key)  
 else:  
 root.left = self.insert(root.left, key)  
 return root  
  
 def inorder\_traversal(self, root):  
 if root:  
 self.inorder\_traversal(root.left)  
 print(root.val, end=" ")  
 self.inorder\_traversal(root.right)  
  
 def search(self, root, key):  
 if root is None or root.val == key:  
 return root  
 if root.val < key:  
 return self.search(root.right, key)  
 return self.search(root.left, key)  
  
  
bst = BST()  
bst.root = bst.insert(bst.root, key=50)  
bst.insert(bst.root, key=30)  
bst.insert(bst.root, key=20)  
bst.insert(bst.root, key=40)  
bst.insert(bst.root, key=70)  
bst.insert(bst.root, key=60)  
bst.insert(bst.root, key=80)  
  
print("Inorder traversal of the BST:")  
bst.inorder\_traversal(bst.root)  
print()  
  
  
key = 60  
result = bst.search(bst.root, key)  
if result:  
 print(f" found in the BST")  
else:  
 print(f" not found in the BST")

**output:**

Inorder traversal of the BST:

20 30 40 50 60 70 80

found in the BST

**LeafNode:**

class TreeNode:  
 def \_\_init\_\_(self, val=0, left=None, right=None):  
 self.val = val  
 self.left = left  
 self.right = right  
  
  
def printLeafNodes(node):  
  
 if node is None:  
 return  
  
  
 if node.left is None and node.right is None:  
 print(node.val)  
 return  
  
  
 if node.left:  
 printLeafNodes(node.left)  
 if node.right:  
 printLeafNodes(node.right)  
  
  
root = TreeNode(1)  
root.left = TreeNode(2)  
root.right = TreeNode(3)  
root.left.left = TreeNode(4)  
root.left.right = TreeNode(5)  
root.right.right = TreeNode(6)  
root.right.right.left = TreeNode(7)  
  
  
printLeafNodes(root)

output:

4

5

7

Geeks for Geeks: #User function Template for python3

'''

# Node Class:

class Node:

def \_init\_(self,val):

self.data = val

self.left = None

self.right = None

'''

class Solution:

#Function to find the height of a binary tree.

def height(self, root):

if root is None:

return 0

return (1 + max(self.height(root.left), self.height(root.right)))1)Height of the binary tree:

**output:**

For Input:

1 2 3

Your Output:

2

Expected Output:

2

**GRAPHS:**

All tress are graphs but not all graphs or not trees.

Adjacent matrix:

def print\_adj\_list(graph):  
 print("Adjacent List:")  
 for node,neighbors in enumerate(graph):  
 print(f"{node}:{'.'.join(map(str,neighbors))}")  
def print\_adj\_matrix(matrix):  
 print("Adjacency matrix:")  
 for row in matrix:  
 print(".".join(map(str,row)))  
  
adj\_list=[  
 [1,2],  
 [2],  
 [0,3],  
 [3]  
  
]  
num\_nodes=4  
adj\_matrix=[[0]\*num\_nodes for \_ in range(num\_nodes)]  
edges=[  
 (0,1),(0,2),  
 (1,2),  
 (2,0),(2,3),  
 (3,3)  
]  
  
for src,dest in edges:  
 adj\_matrix[src][dest]=1  
  
print\_adj\_list(adj\_list)  
  
print\_adj\_matrix(adj\_matrix)

o/p:

Adjacent List:

0:1.2

1:2

2:0.3

3:3

Adjacency matrix:

0.1.1.0

0.0.1.0

1.0.0.1

0.0.0.1

**Problems:**

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def speak(self, language):

print(f"Hi, my name is {self.name} and my age is {self.age} and my language is {language}")

p1 = Person(name="Akshaya", age=20)

p2 = Person(name="Pandu", age=21)

p1.speak("Malayalam.")

**output:**

Hi, my name is Akshaya and my age is 20 and my language is Malayalam.

**2)**class Animal:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def description(self):

print(f"Hi, my name is {self.name} and I am {self.age} years old.")

class Dog(Animal):

def \_\_init\_\_(self, name, age, breed):

super().\_\_init\_\_(name, age)

self.breed = breed

def description(self):

print(f"This is {self.name}, a cute dog who is {self.age} years old. His breed is {self.breed}.")

dog = Dog(name="Yuvraj", age=5, breed="Pomeranian")

animal = Animal(name="Leo", age=3)

dog.description()

animal.description()

**output:**

This is Yuvraj, a cute dog who is 5 years old. His breed is Pomeranian.

Hi, my name is Leo and I am 3 years old.

**ii)** class Animal:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

def description(self):

print(f"Hi, my name is {self.name} and I am {self.age} years old.")

class Dog(Animal):

def \_\_init\_\_(self, name, age, breed):

super().\_\_init\_\_(name, age)

self.breed = breed

def description(self):

print(f"This is {self.name}, a cute dog who is {self.age} years old. His breed is {self.breed}.")

class Parrot(Animal):

def \_\_init\_\_(self,name, age):

super(),\_\_init\_\_(name, age)

def description(self):

print(f" {self.name}" )

dog = Dog(name="Yuvraj", age=5, breed="Pomeranian")

animal = Animal(name="Leo", age=3)

parrot = Animal(name="Bunny", age=2)

dog.description()

animal.description()

parrot.description()

**output:**

This is Yuvraj, a cute dog who is 5 years old. His breed is Pomeranian.

Hi, my name is Leo and I am 3 years old.

Hi, my name is Bunny and I am 2 years old.

**To search the number in a list:**

arr = input("Enter the last numbers seperated by spaces:").split()

arr = [int(x) for x in arr]

target = int(input("enter the target value to serach:"))

found = False

for i in range(len(arr)):

if arr[i] == target:

print(f"Target {target} found at index {i}.")

found = True

break

if not found:

print(f"Target {target} not found in the array.")

**output:**

Enter the last numbers seperated by spaces:1 2 3 4 5

enter the target value to serach:5

Target 5 found at index