Linear search/ sequential search

```
def seq_search(alist,item):
    pos=0
    found=False
    while pos<len(alist) and not found:
        if alist[pos]==item:
            found=True
        else:
            pos=pos+1
    return pos
array=[10,48,65,23,54,89,90,42,11]
print(array)
num=int(input("enter number to search="))
pos=seq_search(array,num)
if(pos!=len(array)):
    print(f"{num} is present of {pos+1} position")
else:
    print(f"{num} is not present in array")
```

BINARY SEARCH

```
def binary_search(alist,num):
    first=0
    last=len(alist)-1
    found=False
    while first <= last and not found:
        midpoint=(first+last)//2
        if alist[midpoint]==num:
            found=True
        else:
```

```
        if num< alist[midpoint]:
            last=midpoint-1
        else:
            first=midpoint+1
    return found
array=[10,45,65,23,54,34,32,31,89,9,45]
array.sort()
print(array)
num=int(input("enter number to search="))
if(binary_search(array,num)):
    print(f"{num} is present")
else:
    print(f"{num} is not present")
```

BUBBLE SORT

```
def bubblesort(alist):
    for passnum in range (len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp=alist[i]
                alist[i]=alist[i+1]
                alist[i+1]=temp
        passnum=passnum-1
        print(alist)
alist=[56,89,65,32,12,56,43,89,76,56]
bubblesort(alist)
print(alist)
```

STACK

```
import sys
class Stack:
```

```
def __init__(self,max_size):
    """Initialize the stack with a fixed maximum size"""
    self.max_size=max_size
    self.stack=[]


def push(self,item):
    """Add an item to the stack if it's not full"""
    if self.is_full():
        print("Stack is full,cannot push.")
    else:
        self.stack.append(item)
        print(f"Item {item} pushed to stack.")
def pop(self):
    """Remove and return the top item from the stack if it's not empty"""
    if self.is_empty():
        print("Stack is empty,cannot pop.")
        return None
    else:
        return self.stack.pop()
def peek(self):
    """Return Topmost element"""
    return(self.stack[len(self.stack)-1])
def is_empty(self):
    """Check if the stck is empty"""
    return len(self.stack)==0
def is_full(self):
    """Check if the stack is full"""
    return len(self.stack)==self.max_size
def display(self):
```

```
        """Display the current stack"""
        print("Current Stack:",self.stack)
obj=Stack(5)
while(1):
    print("Menu\n 1.Push\n2.Pop\n3.Peek\n4.IsEmpty\n5.IsFull\n6.Display\n7.Exit")
    ch=int(input("Enter your choice: "))
    if ch==1:
        d=int(input("Enter element to be pushed:"))
        obj.push(d)
    elif ch==2:
        print(f"Popped element is {obj.pop()}")
    elif ch==3:
        print(f"Top element is {obj.peek()}")
    elif ch==4:
        if(obj.is_empty()):
            print("Stack is Empty")
        else:
            print("Stack is Not Empty.")
    elif ch==5:
        if(obj.is_full()):
            print("Stack is Full.")
        else:
            print("Stack is Not Full.")
    elif ch==6:
        obj.display()
    else:
        sys.exit()
```

MERGE SORT

```
def mergeSort(alist):
    print("Splitting",alist)
    if len(alist)>1:
        mid =len(alist)//2
        lefthalf=alist[:mid]
        righthalf=alist[mid:]
        mergeSort(lefthalf)
        mergeSort(righthalf)
        i=0
        j=0
        k=0
        while i<len(lefthalf) and j<len(righthalf):
            if lefthalf[i]<righthalf[j]:
                alist[k]=lefthalf[i]
                i=i+1
            else:
                alist[k]=righthalf[j]
                j=j+1
            k=k+1
        while i<len(lefthalf):
            alist[k]=lefthalf[i]
            i=i+1
            k=k+1
        while j<len(righthalf):
            alist[k]=righthalf[j]
            j=j+1
            k=k+1
        print("Merging",alist)
alist=[54,26,93,17,77]
```

```
mergeSort(alist)
print(alist)
```

QUICK SORT

```
def quickSort(alist):
    quickSortHelper(alist,0,len(alist)-1)
    print(alist)
def quickSortHelper(alist,first,last):
    if first<last:
        splitpoint = partition(alist,first,last)
        quickSortHelper(alist,first,splitpoint-1)
        quickSortHelper(alist,splitpoint+1,last)
def partition(alist,first,last):
    pivotvalue=alist[first]
    leftmark=first + 1
    rightmark=last
    done=False
    while not done:
        while leftmark <= rightmark and alist[leftmark] <= pivotvalue:
            leftmark = leftmark + 1
        while alist[rightmark] >= pivotvalue and rightmark >= leftmark:
            rightmark = rightmark - 1
        if rightmark < leftmark:
            done=True
        else:
            temp=alist[leftmark]
            alist[leftmark]=alist[rightmark]
            alist[rightmark]=temp
    temp=alist[first]
    alist[first]=alist[rightmark]
```

```
    alist[rightmark]=temp
  return rightmark
alist=[54,26,93,17,77,31,44,55,20]
quickSort(alist)
print(alist)
```

DYNAMIC SINGLY LINKED LIST

```
import sys
class Stack:
    def __init__(self,max_size):
        """Initialize the stack with a fixed maximum size"""
        self.max_size=max_size
        self.stack=[]

    def push(self,item):
        """Add an item to the stack if it's not full""'
        if self.is_full():
            print("Stack is full,cannot push.")
        else:
            self.stack.append(item)
            print(f"Item {item} pushed to stack.")
    def pop(self):
        """Remove and return the top item from the stack if it's not empty"""
        if self.is_empty():
            print("Stack is empty,cannot pop.")
            return None
        else:
            return self.stack.pop()
    def peek(self):
        """Return Topmost element"""
```

```
        return(self.stack[len(self.stack)-1])
    def is_empty(self):
        """Check if the stck is empty"""
        return len(self.stack)==0
    def is_full(self):
        """Check if the stack is full"""
        return len(self.stack)==self.max_size
    def display(self):
        """Display the current stack"""
        print("Current Stack:",self.stack)

obj=Stack(5)
while(1):
    print("Menu\n 1.Push\n2.Pop\n3.Peek\n4.IsEmpty\n5.IsFull\n6.Display\n7.Exit")
    ch=int(input("Enter your choice: "))
    if ch==1:
        d=int(input("Enter element to be pushed:"))
        obj.push(d)
    elif ch==2:
        print(f"Popped element is {obj.pop()}")
    elif ch==3:
        print(f"Top element is {obj.peek()}")
    elif ch==4:
        if(obj.is_empty()):
            print("Stack is Empty")
        else:
            print("Stack is Not Empty.")
    elif ch==5:
        if(obj.is_full()):
```

```
            print("Stack is Full.")
        else:
            print("Stack is Not Full.")
    elif ch==6:
        obj.display()
    else:
        sys.exit()
```

POSTFIX EVALUATION

```
class stack:
    def __init__(self):
        self.__item=[]

    def push(self,item):
            self.__item.append(item)

    def pop(self):
        if self.is_empty():
            return None
        else:
            return self.__item.pop()
    def peek(self):
        if len(self.__item) <=1:
            return (self.__item[len(self.__item)-1])
        else:
            return (self.__item[len(self.__item)-1])
    def is_empty(self):
        return len(self.__item)==0

    def display(self):
```

```
        print("current stack",self.__item)

    def size(self):
        return len(self.__item)

    def __str__(self):
        return str(self.__item)

    def __len__(self):
        return len(self.__item)
def postfixEval(postfixExpr):
    operandStack = stack()
    tokenList = postfixExpr.split()
    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()
def doMath(op, op1, op2):
    if op == "*":
        return op1 * op2
    elif op == "/":
        return op1 // op2
    elif op == "+":
        return op1 + op2
```

```
    else:
        return op1 - op2

print(postfixEval('7 8 + 3 2 + /'))
print(postfixEval('5 1 2 + 4 * + 3 -'))
```

INFIX TO POSTFIX

```
class stack:
    def __init__(self):
        self.__item=[]

    def push(self,item):
            self.__item.append(item)

    def pop(self):
        if self.is_empty():
            return None
        else:
            return self.__item.pop()
    def peek(self):
        if len(self.__item) <=1:
            return (self.__item[len(self.__item)-1])
        else:
            return (self.__item[len(self.__item)-1])
    def is_empty(self):
        return len(self.__item)==0

    def display(self):
        print("current stack",self.__item)
```

```
    def size(self):
        return len(self.__item)

    def __str__(self):
        return str(self.__item)

    def __len__(self):
        return len(self.__item)

def Infixtopostfix(infixexpr):
    prec={}
    prec["*"]=3
    prec["/"]=3
    prec["+"]=2
    prec["-"]=2
    prec["("]=1

    opstack=stack()
    postfixlist=[]
    tokenlist=infixexpr.split()

    for token in tokenlist:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixlist.append(token)
        elif token == '(':
            opstack.push(token)
        elif token == ')':
            toptoken=opstack.pop()
            while toptoken != '(':
```

```
            postfixlist.append(toptoken)
            toptoken =opstack.pop()
        else:
            while(not opstack.is_empty() and prec[opstack.peek()]>= prec[token]):
                postfixlist.append(opstack.pop())
            opstack.push(token)

        while not opstack.is_empty():
            postfixlist.append(opstack.pop())
    return " ".join(postfixlist)
print(Infixtopostfix("A+B*C-D"))
print(Infixtopostfix("A*B+C*D"))
print(Infixtopostfix("(A+B)*C-(D-E)*(F+G)"))
print(Infixtopostfix("(A+B)*C"))
```

STACK USING LINKED LIST

```
class Node:
    def __init__(self,data):
        self.data = data
        self.next = None

class stack:
    def __init__(self):
        self.top=None

    def is_empty(self):
        return self.top is None
    def push(self,data):
        new_node = Node(data)
        new_node.next = self.top
```

```
    self.top = new_node
    print(f"pushed{data} onto the stack")

def pop(self):
    if self.is_empty():
        print("stack underflow! cannot pop from the empty stack")
        return None
    popped_node = self.top
    self.top = self.top.next
    print(f"popped {popped_node.data}from the stack")
    return popped_node.data
def peek(self):
    if self.is_empty():
        print("stack is empty")
        return None
    return self.top.data
def size(self):
    count = 0
    current = self.top
    while current:
        count +=1
        current = current.next
        return count
def display(self):
    if self.is_empty():
        print("stack is empty")
        return
    print("stack:")
    current = self.top
```

```
        while current:
            print(current.data,"->",end='')
            current = current.next
            print("None")
obj=stack()
obj.push(20)
obj.push(90)
obj.push(40)
obj.push(10)
obj.display()
obj.peek()
print("\ntotal elements:",obj.size())
print("\npoped Elements:",obj.pop())
obj.display()
```

SINGLY LINKED LIST

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class SinglyLinkedList:
    def __init__(self):
        self.head = None

    def is_empty(self):
        return self.head is None

    def addFirst(self, data):
        new_node = Node(data)
```

```
    new_node.next = self.head
    self.head = new_node

def addLast(self, data):
    new_node = Node(data)
    if self.is_empty():
        self.head = new_node
    else:
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node

def addAfter(self, prev_data, data):
    current = self.head
    while current and current.data != prev_data:
        current = current.next

    if not current:
        print(f"No node with data {prev_data}")
        return
    new_node = Node(data)
    new_node.next = current.next
    current.next = new_node

def deleteFirst(self):
    if self.is_empty():
        print("List is empty")
        return
```

```
    temp = self.head
    self.head = self.head.next
    del temp

def deleteLast(self):
    if self.is_empty():
        print("List is empty")
        return
    if self.head.next is None:  # Only one node
        del self.head
        self.head = None
        return

    current = self.head
    while current.next.next:
        current = current.next
    del current.next
    current.next = None

def deleteAfter(self, prev_data):
    current = self.head
    while current and current.data != prev_data:
        current = current.next
    if not current or not current.next:
        print(f"No node after {prev_data}")
        return
    temp = current.next
    current.next = current.next.next
    del temp
```

```
def search(self, key):
    current = self.head
    pos = 0
    while current:
        if current.data == key:
            print(f"{key} found at position {pos}")
            return True
        current = current.next
        pos += 1
    print(f"{key} not found in the list")
    return False
def create(self, L1):
    for d in L1:
        self.addLast(d)
    print("List created")
def sort(self):
    if self.is_empty():
        return
    current = self.head
    while current:
        next_node = current.next
        while next_node:
            if current.data > next_node.data:
                current.data, next_node.data = next_node.data, current.data
            next_node = next_node.next
        current = current.next

def reverse(self):
```

```
        if self.is_empty():
            print("List is empty")
            return
        prev = None
        current = self.head
        while current:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        self.head = prev

    def display(self):
        current = self.head
        if self.is_empty():
            print("List is empty")
            return
        print("Singly linked list:")
        while current:
            print(current.data, "->", end=" ")
            current = current.next
        print("None")
# Driver code to test the linked list
s = SinglyLinkedList()
print("Options: 1-> is_empty, 2-> addFirst, 3-> addLast, 4-> addAfter, 5-> deleteFirst, 6-> deleteLast, 7-> deleteAfter, 8-> search, 9-> create, 10-> sort, 11-> reverse, 12-> display")
while True:
    ch = int(input("Enter your choice (1-12, 0 to exit): "))
    if ch == 0:
```

```
        break
    elif ch == 1:
        print("List is empty" if s.is_empty() else "List is not empty")
    elif ch == 2:
        num = int(input("Enter data: "))
        s.addFirst(num)
    elif ch == 3:
        num = int(input("Enter data: "))
        s.addLast(num)
    elif ch == 4:
        pnum = int(input("Enter previous data: "))
        num = int(input("Enter data: "))
        s.addAfter(pnum, num)
    elif ch == 5:
        s.deleteFirst()
    elif ch == 6:
        s.deleteLast()
    elif ch == 7:
        pnum = int(input("Enter previous data: "))
        s.deleteAfter(pnum)
    elif ch == 8:
        num = int(input("Enter data to search: "))
        s.search(num)
    elif ch == 9:
        num = [20, 43, 24, 67, 54, 34, 98]
        s.create(num)
    elif ch == 10:
        s.sort()
    elif ch == 11:
```

```
        s.reverse()
    elif ch == 12:
        s.display()
    else:
        print("Invalid choice, please try again.")
dynamic queue
class Queue:
    def __init__(self):
        self._items = []
    def isEmpty(self):
        return self._items == []
    def enqueue(self, item):
        self._items.insert(0,item)
    def dequeue(self):
        return self._items.pop()
    def size(self):
        return len(self._items)
    def peek(self):
        return self._items[0]
    def __str__(self):
        return str(self._items)
    def __len__(self):
        return len(self._items)
obj=Queue()
print(obj.isEmpty())
obj.enqueue(4)
obj.enqueue('ashwini')
print(obj.peek())
obj.enqueue(True)
```

```
print(obj.size())
print(obj.isEmpty())
obj.enqueue(8.4)
print(obj.dequeue())
print(obj.dequeue())
print(obj.size())
print(len(obj))
print(obj)
```

CIRCULAR QUEUE

```
class CircularQueue:
    def __init__(self, max_size):
        self.queue = [None] * max_size
        self.max_size = max_size
        self.front = -1
        self.rear = -1

    def is_full(self):
        return (self.rear + 1) % self.max_size == self.front

    def is_empty(self):
        return self.front == -1

    def enqueue(self, item):
        if self.is_full():
            print("Circular Queue is full, cannot enqueue.")
            return
        if self.is_empty():
            self.front = 0
        self.rear = (self.rear + 1) % self.max_size
```

```
    self.queue[self.rear] = item
    print(f"Enqueued: {item}")

def dequeue(self):
    if self.is_empty():
        print("Circular Queue is empty, cannot dequeue.")
        return None
    item = self.queue[self.front]
    if self.front == self.rear:
        # Queue has become empty
        self.front = -1
        self.rear = -1
    else:
        self.front = (self.front + 1) % self.max_size
    print(f"Dequeued: {item}")
    return item

def peek(self):
    if self.is_empty():
        print("Circular Queue is empty.")
        return None
    return self.queue[self.front]

def display(self):
    if self.is_empty():
        print("Circular Queue is empty.")
        return
    print("Circular Queue:", end=" ")
    i = self.front
```

```
        while True:
            print(self.queue[i], end=" ")
            if i == self.rear:
                break
            i = (i + 1) % self.max_size
        print()
# Main menu-driven program
def main():
    size = int(input("Enter the size of the Circular Queue: "))
    cq = CircularQueue(size)
    while True:
        print("\nMenu:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Peek")
        print("4. Display")
        print("0. Exit")
        choice = int(input("Enter your choice: "))
        if choice == 0:
            break
        elif choice == 1:
            item = int(input("Enter the item to enqueue: "))
            cq.enqueue(item)
        elif choice == 2:
            cq.dequeue()
        elif choice == 3:
            front_item = cq.peek()
            if front_item is not None:
                print(f"Front item: {front_item}")
```

```
        elif choice == 4:
            cq.display()
        else:
            print("Invalid choice, please try again.")

if __name__ == "__main__":
    main()
```

Implementation of an algorithm that reverses string of characters using stack and checks whether a string is a palindrome. 1

```
def reverse_string(input_str):
    stack = []
    for char in input_str:
        stack.append(char)
        reversed_str = ''
    while stack:
        reversed_str += stack.pop()
    return reversed_str
def is_palindrome(input_str):
    # Remove any spaces and convert to lowercase for case-insensitive comparison
    input_str = input_str.replace(" ", "").lower()
    reversed_str = reverse_string(input_str)
        return input_str == reversed_str
input_str = "madam"
print(f"Reversed String: {reverse_string(input_str)}")
print(f"Is the string a palindrome? {is_palindrome(input_str)}")
```

BINARY SEARCH TREE

```
class Node:
    def __init__(self, key):
        self.key = key
```

```
        self.left = None
        self.right = None


class BinarySearchTree:
    def __init__(self):
        self.root = None
    # Insert a node in the BST
    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
        else:
            self._insert(self.root, key)
    def _insert(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = Node(key)
            else:
                self._insert(node.left, key)
        else:
            if node.right is None:
                node.right = Node(key)
            else:
                self._insert(node.right, key)
    # In-order traversal (Left, Root, Right)
    def inorder(self):
        return self._inorder(self.root)
    def _inorder(self, node):
        res = []
        if node:
```

```
        res = self._inorder(node.left)
        res.append(node.key)
        res = res + self._inorder(node.right)
    return res
# Pre-order traversal (Root, Left, Right)
def preorder(self):
    return self._preorder(self.root)
def _preorder(self, node):
    res = []
    if node:
        res.append(node.key)
        res = res + self._preorder(node.left)
        res = res + self._preorder(node.right)
    return res
# Post-order traversal (Left, Right, Root)
def postorder(self):
    return self._postorder(self.root)

def _postorder(self, node):
    res = []
    if node:
        res = self._postorder(node.left)
        res = res + self._postorder(node.right)
        res.append(node.key)
    return res

# Count total nodes
def count_total_nodes(self):
    return self._count_total_nodes(self.root)
```

```
    def _count_total_nodes(self, node):
        if node is None:
            return 0
        return 1 + self._count_total_nodes(node.left) + self._count_total_nodes(node.right)

    # Count leaf nodes (nodes with no children)
    def count_leaf_nodes(self):
        return self._count_leaf_nodes(self.root)

    def _count_leaf_nodes(self, node):
        if node is None:
            return 0
        if node.left is None and node.right is None:
            return 1
        return self._count_leaf_nodes(node.left) + self._count_leaf_nodes(node.right)

    # Count non-leaf nodes (nodes with at least one child)
    def count_non_leaf_nodes(self):
        return self._count_non_leaf_nodes(self.root)

    def _count_non_leaf_nodes(self, node):
        if node is None or (node.left is None and node.right is None):
            return 0
        return 1 + self._count_non_leaf_nodes(node.left) + self._count_non_leaf_nodes(node.right)

# Main menu-driven program
def main():
    bst = BinarySearchTree()
```

```
while True:
    print("\nMenu:")
    print("1. Insert a node")
    print("2. In-order traversal")
    print("3. Pre-order traversal")
    print("4. Post-order traversal")
    print("5. Count total nodes")
    print("6. Count leaf nodes")
    print("7. Count non-leaf nodes")
    print("0. Exit")

    choice = int(input("Enter your choice: "))
    if choice == 0:
        break
    elif choice == 1:
        key = int(input("Enter the key to insert: "))
        bst.insert(key)
        print(f"Inserted {key} into the BST.")
    elif choice == 2:
        print("In-order traversal:", bst.inorder())
    elif choice == 3:
        print("Pre-order traversal:", bst.preorder())
    elif choice == 4:
        print("Post-order traversal:", bst.postorder())
    elif choice == 5:
        print("Total nodes:", bst.count_total_nodes())
    elif choice == 6:
        print("Leaf nodes:", bst.count_leaf_nodes())
    elif choice == 7:
```

```
            print("Non-leaf nodes:", bst.count_non_leaf_nodes())
        else:
            print("Invalid choice, please try again.")

if __name__ == "__main__":
    main()
```

GRAPH ADJACENCY MATRIX

```
class DirectedGraphMatrix:
    def __init__(self, num_nodes):
        self.num_nodes = num_nodes
        # Initialize a matrix with 0s
        self.adj_matrix = [[0 for i in range(num_nodes)] for j in range(num_nodes)]

    def add_edge(self, from_node, to_node):
        # Assuming nodes are labeled from 0 to num_nodes-1
        self.adj_matrix[from_node][to_node] = 1

    def remove_edge(self, from_node, to_node):
        self.adj_matrix[from_node][to_node] = 0

    def display(self):
        for row in self.adj_matrix:
            print(row)
    def get_out_degree(self, node):
        if 0 <= node < self.num_nodes:
            return sum(self.adj_matrix[node])

    def get_in_degree(self, node):
        if 0 <= node < self.num_nodes:
```

```
        return sum(self.adj_matrix[i][node] for i in range(self.num_nodes))

    def get_all_degrees(self):
        in_degrees = {}
        out_degrees = {}
        for node in range(self.num_nodes):
            out_degrees[node] = self.get_out_degree(node)
            in_degrees[node] = self.get_in_degree(node)
        return in_degrees, out_degrees
    def dfs(self, start_node):
        if not (0 <= start_node < self.num_nodes):
            print("Start node must be within the range of the graph.")
            return
        visited = [False] * self.num_nodes
        dfs_order = []
        self.dfs_traversal(start_node, visited, dfs_order)
        return dfs_order
    def dfs_traversal(self, node, visited, dfs_order):
        visited[node] = True
        dfs_order.append(node)
        for neighbor in range(self.num_nodes):
            if self.adj_matrix[node][neighbor] == 1 and not visited[neighbor]:
                self.dfs_traversal(neighbor, visited, dfs_order)

g=DirectedGraphMatrix(3)
g.add_edge(1,2)
g.add_edge(0,2)
g.add_edge(1,2)
g.add_edge(0,2)
```

```
g.add_edge(2,1)
g.add_edge(0,1)
g.display()
print( " Indegree of 0 is ", g.get_in_degree(0))
print( " Outdegree of 0 is ", g.get_out_degree(0))
ind, outd = g.get_all_degrees()

print( f" Indegree of 0 is{ind} and outdegree {outd}")
print("\nIn-degree of each node:")
for node in sorted(ind):
    print(f"Node {node}: {ind[node]}")

print("\nOut-degree of each node:")
for node in sorted(outd):
    print(f"Node {node}: {outd}")

print("DFS ",g.dfs(0))
```