

Roll No:- _____

**Sonopant Dandekar Shikshan Mandali's
Sonopant Dandekar Arts,V.S.Apte Commerce,
M.H.Mehta Science College**



DEPARTMENT OF COMPUTER SCIENCE

CERTIFICATE

Certified That Mr./Miss. _____
of _____ has satisfactorily completed a course of
necessary experiment in _____ under
my supervision in the SY.BSC Computer Science in the Year 2025 – 2026

Head of Department

Subject Teacher

Date: / / 2025

INDEX

SR NO.	Aim of Practical	Practical Date	Submission Date	Remarks
1.	Exploring Abstract Data Types (ADT) & Custom Structures <ul style="list-style-type: none">• Create and manipulate structures to model ADTs like Student, Book, or Employee.• Implement basic operations (create, update, delete) using structures.• Reflect on differences between primitive & abstract data types.	30/07/2025	06/08/2025	
2.	Building and Using Singly Linked Lists <ul style="list-style-type: none">• Construct a dynamic singly linked list with basic operations.• Apply linked lists to simulate scenarios such as managing a playlist or to-do list.• Compare static (array) vs dynamic (linked) approaches.	06/08/2025	13/08/2025	
3.	Polynomial Operations Using Linked Lists <ul style="list-style-type: none">• Represent polynomials using linked lists.• Perform polynomial addition and subtraction by merging lists.• Use structured representation to reinforce node manipulation	13/08/2025	23/08/2025	
4	Working with Doubly Linked Lists <ul style="list-style-type: none">• Create a doubly linked list with forward and backward traversal.• Implement insertion/deletion at head, tail, and specific positions.• Use in scenarios like browser history or undo-redo features.	23/08/2025	03/09/2025	
5	Implementing and Using Stack ADT <ul style="list-style-type: none">• Implement push, pop, peek using arrays or linked listsSolve problems like delimiter matching or undo mechanism.• Convert expressions from prefix to postfix and evaluate them.	03/09/2025	10/09/2025	
6	Understanding Queues and Circular Queues <ul style="list-style-type: none">• Develop linear and circular queues to simulate task scheduling.• Perform enqueue and dequeue with wrap-around logic.• Discuss memory utilization in linear vs circular queues.	10/09/2025	17/09/2025	
7	Tree Traversals and Binary Search Trees <ul style="list-style-type: none">• Create a binary search tree (BST) from a dataset.• Perform and visualize in-order, pre-order, and post-order traversals.• Use traversal results to derive sorted sequences.	17/09/2025	24/09/2025	
8.	Graph Representations and Traversals <ul style="list-style-type: none">• Represent graphs using adjacency matrices and lists. •Implement BFS and DFS to explore graph components.• Use graphs for mapping routes or exploring social networks.	24/09/2025	25/09/2025	

Practical No 1

Aim: Exploring Abstract data types (ADT) & Custom services.

- Create and manipulate structures to model ADT's like student, Book, or Employee.
- Implement basic operations(Create, update, delete using structures.)
- Reflect on Difference between primitive and non-primitive

1] Create and manipulate structures to model ADT's like student, Book, or Employee.

A] class student

Input:

```
class Student:
    def __init__(self,Student_id,Name,Grade):
        self.Student_id=Student_id
        self.Name=Name
        self.Grade=Grade
    def __str__(self):
        return "["+"StudentID:" +str(self.Student_id)+ " " "name:" +self.Name+ " " "Grade:" +str(self.Grade)+"]"
Student1=Student(1, "Khanak", 20)
Student2=Student(2, "Aarya", 16)
print(Student1)
print(Student2)
```

Output:

```
[StudentID:1 name:Khanak Grade:20]
[StudentID:2 name:Aarya Grade:16]
```

B] class Book.

Input:

```
class Book:
    def __init__(self,Book_title,Author,Year):
        self.Book_title=Book_title
        self.Author=Author
        self.Year=Year
    def __str__(self):
        return "["+"Book_Title:" +self.Book_title+" " "author:" +self.Author+" " "year:" +str(self.Year)+"]"
Book1=Book("Harry potter", "J.s.Howiling",1998)
```

```
Book2=Book("Atom Theory","M.jackson",1999)
print(Book1)
print(Book2)
```

Output:

```
=====
[Book_Title:Harry potter  author:J.s.Howiling year:1998]
[Book_Title:Atom Theory  author:M.jackson year:1999]
```

C] Class Employee

Input:

```
class Employee:
```

```
    def __init__(self,emp_id,name,salary):
```

```
        self.emp_id=emp_id
```

```
        self.name=name
```

```
        self.salary=salary
```

```
    def __str__(self):
```

```
        return "["+str(self.emp_id)+" " "Name:" +self.name+ " " "Salary:" +str(self.salary)+"]"
```

```
Employee1=Employee(101,"Khanak",20000)
```

```
Employee2=Employee(102,"Mansi",30000)
```

```
Employee3=Employee(103,"Pratiksha",40000)
```

```
Employee4=Employee(104,"Aarya",50000)
```

```
print(Employee1)
```

```
print(Employee2)
```

```
print(Employee3)
```

```
print(Employee4)
```

Output:

```
[Employee_id:101 Name:Khanak Salary:20000]
```

```
[Employee_id:102 Name:Mansi Salary:30000]
```

```
[Employee_id:103 Name:Pratiksha Salary:40000]
```

```
[Employee_id:104 Name:Aarya Salary:50000]
```

2] Implement basic operations(Create, update, delete using structures.)

A] Update and Delete

1. Student

Input:

class Student:

```
def __init__(self,Student_id,Name,Grade):
```

```
    self.Student_id=Student_id
```

```
    self.Name=Name
```

```
    self.Grade=Grade
```

```
def update_grade(self,new_grade):
```

```
    self.Grade=new_grade
```

```
def delete(self):
```

```
    self.Student_id=None
```

```
    self.Name=""
```

```
    self.Grade=0.00
```

```
def __str__(self):
```

```
    return "["+"StudentID:" +str(self.Student_id)+ " " "name:" +self.Name+ " " "Grade:" +str(self.Grade)+"]"
```

```
Student1=Student(1, "Khanak", 20)
```

```
Student2=Student(2, "Aarya", 16)
```

```
print(Student1)
```

```
print(Student2)
```

```
Student1.update_grade(40)
```

```
print(":After Updating Grade")
```

```
print(Student1)
```

```
Student2.delete()
```

```
print(":After Deleting 2nd entry data")
```

```
print(Student2)
```

Output:

```
[StudentID:1 name:Khanak Grade:20]
[StudentID:2 name:Aarya Grade:16]
:After Updating Grade
[StudentID:1 name:Khanak Grade:40]
:After Deleting 2nd entry data
[StudentID:None name: Grade:0.0]
```

2. Book

Input:

```

class Book:
    def __init__(self,Book_title,Author,Year):
        self.Book_title=Book_title
        self.Author=Author
        self.Year=Year
    def update_year(self,new_year):
        self.Year=new_year
    def delete(self):
        self.Author=""
        self.Year=""
    def __str__(self):
        return "["+self.Book_title+" " "author:" +self.Author+" " "year:" +str(self.Year)+"]"
Book1=Book("Harry potter", "J.s.Howiling",1998)
Book2=Book("Atom Theory","M.jackson",1999)
print(Book1)
print(Book2)
Book1.update_year(1996)
print(":After Updating Year")
print(Book1)
Book2.delete()
print(":After Deleting 2nd entry data")
print(Book2)

```

Output:

```

[Book_Title:Harry potter author:J.s.Howiling year:1998]
[Book_Title:Atom Theory author:M.jackson year:1999]
:After Updating Year
[Book_Title:Harry potter author:J.s.Howiling year:1996]
:After Deleting 2nd entry data
[Book Title:Atom Theory author: year:]

```

3. Employee

Input:

```

class Employee:
    def __init__(self,emp_id,name,salary):
        self.emp_id=emp_id
        self.name=name
        self.salary=salary
    def update_salary(self,new_salary):
        self.salary=new_salary
    def delete(self):
        self.salary=0.00
        self.name=""
    def __str__(self):
        return "["+str(self.emp_id)+" " "Name:" +self.name+" " "Salary:" +str(self.salary)+"]"

```

```
Employee1=Employee(101,"Khanak",20000)
Employee2=Employee(102,"Mansi",30000)
Employee3=Employee(103,"Pratiksha",40000)
Employee4=Employee(104,"Aarya",50000)
print(Employee1)
print(Employee2)
print(Employee3)
print(Employee4)
Employee1.update_salary(70000)
print(":After Updating Salary")
print(Employee1)
Employee2.delete()
print(":After Deleting 2nd entry data")
print(Employee2)
```

Output:

```
[Employee_id:101 Name:Khanak Salary:20000]
[Employee_id:102 Name:Mansi Salary:30000]
[Employee_id:103 Name:Pratiksha Salary:40000]
[Employee_id:104 Name:Aarya Salary:50000]
:After Updating Salary
[Employee_id:101 Name:Khanak Salary:70000]
:After Deleting 2nd entry data
[Employee_id:102 Name: Salary:0.0]
```

3] Reflect on differences between primitive & abstract data types.

Primitive data types:

Primitive data types are the most basic types provided by a programming language (like int, float, char, boolean). They store simple values directly and are generally faster and memory efficient.

Non-primitive data :

Non-primitive data types (like arrays, strings, classes, objects) are more complex, built using primitive types. They can store multiple values, have methods, and are more flexible but require more memory and processing.

Practical No 2

Aim: Building and Using Singly Linked Lists

- Construct a dynamic singly linked list with basic operations.
 - Apply linked lists to simulate scenarios such as managing a to-do list.
 - Compare static (array) vs dynamic (linked) approaches.
-

A] Construct a dynamic singly linked list with basic operations.**Input:**

single linked list

class Node:

```
def __init__(self,data):
```

```
    self.data=data
```

```
    self.next=None
```

class singlyLinkedList:

```
def __init__(self):
```

```
    self.head=None
```

```
def insert_beginning(self,data):
```

```
    new_node=Node(data)
```

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

```
    self.head=new_node
```

```
def insert_end(self,data):
```

```
    new_node=Node(data)
```

```
    if not self.head:
```

```
        self.head=new_node
```

```
    return
```

```
    curr=self.head
```

```
    while curr.next:
```

```
        curr=curr.next
```

```
    curr.next=new_node
```

```
def delete(self,key):
```

```
    curr=self.head
```

```
    if curr and curr.data == key:
```

```
        self.head=curr.next
```



```

        print(f"deleted key")
        return
    prev = None
    while curr and curr.data != key:
        prev=curr
        curr=curr.next
    if not curr:
        print(f"{key} not found")
        return
    prev.next=curr.next
    print(f"Deleted {key}")
def display(self):
    if not self.head:
        print("List is empty")
        return
    curr=self.head
    while curr:
        print(curr.data, end=" -> ")
        curr=curr.next
    print("None")
def main():
    ll=singlyLinkedList()
    print("Enter initial values (space_seperated)")
    for val in input().split():
        ll.insert_end(int(val))
    while True:
        print("\nMenu: 1) Insert start 2) Insert end 3) Delete 4) Display 5)Exit ")
        choice = input("Choice:")
        if choice == '1':
            val=int(input("Value to insert at start:"))
            ll.insert_beginning(val)
        elif choice == '2':

```

```

        val=int(input("value to insert at end:"))
        ll.insert_end(val)
    elif choice == '3':
        val=int(input("value to delete:"))
        ll.delete(val)
    elif choice == '4':
        ll.display()
    elif choice == '5':
        print("Bye!!!")
        break
    else:
        print("Invalid choice")
if __name__=="__main__":
    main()

```

Output:

```

Enter initial values (space_seperated)
5 7 9 4 6

```

```

Menu: 1) Insert start 2) Insert end 3) Delete 4) Display 5)Exit
Choice:1
Value to insert at start:2

```

```

Menu: 1) Insert start 2) Insert end 3) Delete 4) Display 5)Exit
Choice:3
value to delete:9
Deleted 9

```

```

Menu: 1) Insert start 2) Insert end 3) Delete 4) Display 5)Exit
Choice:4
2 -> 5 -> 7 -> 4 -> 6 -> None

```

B] Apply linked lists to simulate scenarios such as managing a playlist or to-do list.

Input:

```

class Task:
    def __init__(self, description):
        self.description=description
        self.next=None
class ToDoList:
    def __init__(self):
        self.head=None

```

```

def add_task(self, description):
    new_task=Task(description)
    if not self.head:
        self.head=new_task
        return
    curr=self.head
    while curr.next:
        curr=curr.next
    curr.next=new_task

def remove_task(self, description):
    curr=self.head
    prev=None
    while curr and curr.description != description:
        prev=curr
        curr=curr.next
    if not curr:
        print(f"Task '{description}' not found")
        return
    if prev:
        prev.next=curr.next
    else:
        self.head=curr.next
    print(f"Task '{description}' removed.")

def show_task(self):
    if not self.head:
        print("To-do list is empty")
        return
    curr=self.head
    print("To-Do List:")
    while curr:
        print(f"- {curr.description}")
        curr=curr.next

def main():
    todo=ToDoList()

```

```

print("Welcome to your To-Do List")
while True:
    print("\nOptions: 1)Add Task 2)Remove Task 3)Show Task 4)Exit")
    choice = input("Choose an option: ")
    if choice == '1':
        desc=input("Enter task description: ")
        todo.add_task(desc)
    elif choice == '2':
        desc=input("Enter task description to remove: ")
        todo.remove_task(desc)
    elif choice == '3':
        todo.show_task()
    elif choice == '4':
        print("Bye!")
        break
    else:
        print("Invalid choice")
if __name__ == "__main__":
    main()

```

Output:

```

Welcome to your To-Do List
Options: 1)Add Task 2)Remove Task 3)Show Task 4)Exit
Choose an option: 1
Enter task description: Dance
Options: 1)Add Task 2)Remove Task 3)Show Task 4)Exit
Choose an option: 1
Enter task description: Sing
Options: 1)Add Task 2)Remove Task 3)Show Task 4)Exit
Choose an option: 3
To-Do List:
- Dance

```

3] Compare static (array) vs dynamic (linked) approaches.

1. Memory Allocation

- **Static (Array):**
 - Fixed size. Memory is allocated at the time of creation and cannot be changed without creating a new array.
 - Inefficient if the journal needs to grow dynamically.

- Can waste space if the allocated size is too large or run out of space if too small.
- **Dynamic (Linked List):**
 - Memory is allocated as needed. Each new entry is linked to the previous one.
 - More efficient for growing the journal because it doesn't require resizing like arrays.
 - Memory usage is more flexible, but each element requires extra memory for the pointer to the next element.

2. Flexibility

- **Static (Array):**
 - Less flexible, as size is fixed and may require reallocation.
- **Dynamic (Linked List):**
 - Highly flexible. No need to know the number of entries in advance, and you can add as many elements as needed.

Practical No 3

Aim: Polynomial Operations Using Linked Lists

- Represent polynomials using linked lists.
 - Perform polynomial addition and subtraction by merging lists.
 - Use structured representation to reinforce node manipulation.
-

Input:

class Node:

def __init__(self,c,p):

self.coeff = c

self.pow = p

self.next = None

def add_polynomial(head1, head2):

if head1 is None:

return head2

if head2 is None:

return head1

if head1.pow > head2.pow:

next_ptr = add_polynomial(head1.next , head2)

head1.next = next_ptr

return head1

elif head1.pow < head2.pow:

next_ptr = add_polynomial(head1 , head2.next)

head2.next = next_ptr

return head2

next_ptr = add_polynomial(head1.next , head2.next)

head1.coeff += head2.coeff

head1.next = next_ptr

return head1

def subtract_polynomial(head1, head2):

if head1 is None:

return head2

if head2 is None:

return head1

if head1.pow > head2.pow:

```

    next_ptr = subtract_polynomial(head1.next, head2)
    head1.next = next_ptr
    return head1
elif head1.pow < head2.pow:
    next_ptr = subtract_polynomial(head1, head2.next)
    head2.next = next_ptr
    return head2
next_ptr = subtract_polynomial(head1.next, head2.next)
head1.coeff -= head2.coeff
head1.next = next_ptr
return head1
def print_list(head):
    curr = head
    while curr is not None:
        print(f"({curr.coeff},{curr.pow})", end=" ")
        curr = curr.next
    print()
if __name__ == "__main__":
    head1 = Node(5, 2)
    head1.next = Node(4, 1)
    head1.next.next = Node(2, 0)
    head2 = Node(-5, 1)
    head2.next = Node(-5, 0)
    print("Add:")
    head_add = add_polynomial(head1, head2)
    print_list(head_add)
    head1 = Node(5, 2)
    head1.next = Node(4, 1)
    head1.next.next = Node(2, 0)
    head2 = Node(-5, 1)
    head2.next = Node(-5, 0)
    print("Sub:")
    head_sub = subtract_polynomial(head1, head2)
    print_list(head_sub)

```

Output:

Add:

(5,2) (-1,1) (-3,0)

Sub:

(5,2) (9,1) (7,0)

3] Use structured representation to reinforce node manipulation

In this program, the polynomial is represented using a linked list structure, where each node stores:

- Coefficient (coeff) of the term
- Exponent (power) of the term
- Pointer (next) to the next node in the list

This structured representation allows:

- Easy insertion and deletion of polynomial terms
- Traversal of nodes to perform operations like addition and subtraction
- Clear understanding of node manipulation (linking and merging lists)

Structure used in the program:

```
struct Node {  
    int coeff;    // coefficient of the term  
    int power;    // exponent of the term  
    struct Node* next; // link to the next term  
};
```

Thus, the polynomial:

$$5x^3 + 2x^2 + 7$$

is represented in the linked list as:

[5,3] -> [2,2] -> [7,0] -> NULL

This structured representation reinforces the concept of dynamic node manipulation in polynomial operations.

Practical No 4**Aim: Working with Doubly Linked Lists**

- Create a doubly linked list with forward and backward traversal.
 - Implement insertion/deletion at head, tail, and specific positions.
 - Use in scenarios like browser history or undo-redo features.
-

Input:

```
class Node:
```

```
    def __init__(self,data):
```

```
        self.data=data
```

```
        self.prev=None
```

```
        self.next=None
```

```
class BrowserHistory:
```

```
    def __init__(self):
```

```
        self.current=None
```

```
    def visit(self,page):
```

```
        new=Node(page)
```

```
        if self.current:
```

```
            self.current.next=None
```

```
            new.prev=self.current
```

```
            self.current.next=new
```

```
        self.current=new
```

```
        print("visited:",page)
```

```
    def back(self):
```

```
        if self.current and self.current.prev:
```

```
            self.current=self.current.prev
```

```
            print("Back to:",self.current.data)
```

```
        else:
```

```
            print("No previous page")
```

```
    def forward(self):
```

```
        if self.current and self.current.next:
```

```
            self.current=self.current.next
```

```
            print("Forward to:",self.current.data)
```

```
        else:
```

```
            print("No Forward page")
```

```

def current_page(self):

    print("current page:",self.current.data if self.current else "None")

# Live testing Menu

bh=BrowserHistory()

while True:

    print("\n----Browser Menu---")

    print("1. Visit a new page")

    print("2. Go Back")

    print("3. Go Forward")

    print("4. Show current page")

    print("5. Exit")
    choice=input("Enter your choice in (1-5):")
    if choice == '1':
        page = input("Enter page URL or name: ")
        bh.visit(page)
    elif choice=='2':
        bh.back()
    elif choice=='3':
        bh.forward()
    elif choice=='4':
        bh.current_page()
    elif choice=='5':
        print("Exit ")
        break
    else:
        print("invalid choicr try again")

```

Output:

```

----Browser Menu---
1. Visit a new page
2. Go Back
3. Go Forward
4. Show current page
5. Exit
Enter your choice in (1-5):1
Enter page URL or name: www.google.com
visited: www.google.com

----Browser Menu---
1. Visit a new page
2. Go Back
3. Go Forward
4. Show current page
5. Exit
Enter your choice in (1-5):4
current page: www.google.com

```

3] Use in scenarios like browser history or undo-redo features.

1. Undo/Redo in Editors

- Each node represents one state of the document.
- Using prev, we can undo changes step by step.
- Using next, we can redo changes.

Thus, the doubly linked list provides efficient navigation in both directions and is well-suited for such scenarios.

Practical No 5**Aim: Implementing and Using Stack ADT**

- Implement push, pop, peek using arrays or linked lists.
 - Solve problems like delimiter matching or undo mechanism.
 - Convert expressions from prefix to postfix and evaluate them.
-

A] Implement push, pop, peek using arrays or linked lists.**Input:**

```
class stack:
```

```
    def __init__(self):
```

```
        self.stack=[]
```

```
    def push(self,value):
```

```
        self.stack.append(value)
```

```
        print(f"Pushed {value} onto the stack.")
```

```
    def pop(self):
```

```
        if self.is_empty():
```

```
            print(f"stack is empty.cannot pop!")
```

```
        else:
```

```
            value=self.stack.pop()
```

```
            print(f"Popped {value} from the stack")
```

```
            return value
```

```
    def peek(self):
```

```
        if self.is_empty():
```

```
            print("stack is empty. cannot peek!")
```

```
            return None
```

```
        else:
```

```
            top_value=self.stack[-1]
```

```
            print(f"Top element is{top_value}")
```

```
            return 0
```

```
    def traverse(self):
```

```
        if self.is_empty():
```

```
            print("stack is empty")
```

```
        else:
```

```
            print("Stack elements (top --> bottom):")
```

```
            for item in reversed(self.stack):
```

```
                print(item)
```

```
    def is_empty(self):
```

```
        return len(self.stack)==0
```

```
#Driver code for user
```

```
if __name__ == "__main__":
```

```
    stack=stack()
```

```
    while True:
```

```
        print("\nChoose an Operation:")
```

```
        print("1.push")
```

```
        print("2.pop")
```

```
        print("3.peek")
```

```
        print("4.traverse(Top-->Bottom)")
```

```
        print("5.Exit")
```

```

choice=input("Enter Your choice(1-6):")
if choice=="1":
    value=int(input("Enter element to push:"))
    stack.push(value)
elif choice=="2":
    stack.pop()
elif choice=="3":
    stack.peek()
elif choice=="4":
    stack.traverse()
elif choice=="5":
    print("Exit from the stack" )
else:
    print("invalid choice!")

```

Output:

```

Choose an Operation:
1.push
2.pop
3.peek
4.traverse(Top-->Bottom)
5.Exit
Enter Your choice(1-6):1
Enter element to push:10
Pushed 10 onto the stack.

```

B) Solve problems like delimiter matching or undo mechanism.

Input:

```
class TextEditor:
```

```

    def __init__(self):
        self.text = " "
        self.history = [ ]
    def write(self, new_text):
        self.history.append(self.text)
        self.text += new_text
        print(f'Written:"{new_text}"')
    def undo(self):
        if self.history:
            self.text=self.history.pop()
            print("Undo performed.")
        else:
            print("Nothing to undo.")
    def read(self):
        print(f'Current Text: "{self.text}"')

```

```
if __name__ == "__main__":
```

```

editor=TextEditor()
while True:
    print("\nChoose an operation:")
    print("1. Write")
    print("2. Undo")
    print("3. Read")
    print("4. Exit")
    choice = input("Enter your choice (1-4):")
    if choice == "1":
        new_text = input("Enter text to write:")
        editor.write(new_text)
    elif choice == "2":
        editor.undo()
    elif choice == "3":
        editor.read()
    elif choice == "4":
        print("Goodbye!")
        break;
    else:
        print("invalid")

```

Output:

```

Choose an operation:
1. Write
2. Undo
3. Read
4. Exit
Enter your choice (1-4):1
Enter text to write:welcome
Written:"welcome"

```

```

Choose an operation:
1. Write
2. Undo
3. Read
4. Exit
Enter your choice (1-4):1
Enter text to write:Hello
Written:"Hello"

```

```

Choose an operation:
1. Write
2. Undo
3. Read
4. Exit
Enter your choice (1-4):2
Undo performed.

```

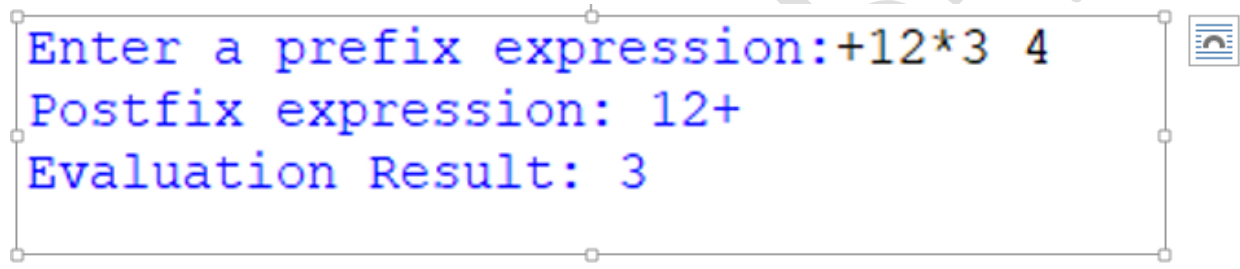
c)Prefix and Postfix addition.

INPUT:

```
class ExpressionConverter:
    def __init__(self):
        self.operators=set(['+', '-', '*', '/', '^'])
    def prefix_to_postfix(self,expression):
        stack=[]
        #scan right to left
        for char in expression[::-1]:
            if char not in self.operators: #operation
                stack.append(char)
            else:#operator
                op1=stack.pop()#operand 1
                op2=stack.pop()#op 2
                new_expr=op1+op2+char
                stack.append(new_expr)
        return stack[-1]
    def evaluate_prefix(self,expression):
        stack=[]
        #scan right to left
        for char in expression[::-1]:
            if char.isdigit():#operand
                stack.append(int(char))
            elif char in self.operators:
                op1=stack.pop()
                op2=stack.pop()
                if char=='+':
                    stack.append(op1+op2)
                elif char=='-':
                    stack.append(op1-op2)
                elif char=='*':
                    stack.append(op1*op2)
                elif char=='/':
                    stack.append(op1//op2)
```

```
elif char=='^':  
    stack.append(op1**op2)  
    return stack[-1]  
#Drive code for user input  
if __name__=="__main__":  
    converter=ExpressionConverter()  
    prefix_expr=input("Enter a prefix expression:")  
    postfix_expr=converter.prefix_to_postfix(prefix_expr)  
    print("Postfix expression:",postfix_expr)  
    result=converter.evaluate_prefix(prefix_expr)  
    print("Evaluation Result:",result)
```

Output:



```
Enter a prefix expression:+12*3 4  
Postfix expression: 12+ 3  
Evaluation Result: 3
```



```
Enter a prefix expression:+9*8 7  
Postfix expression: 98 *+  
Evaluation Result: 65
```


Practical No 6

Aim: Understanding Queues and Circular Queues

- Develop linear and circular queues to simulate task scheduling.
 - Perform enqueue and dequeue with wrap-around logic.
 - Discuss memory utilization in linear vs circular queues..
-

1] Linear Queue

Input:

```

class LinearQueue:
    def __init__(self,capacity):
        self.capacity=capacity
        self.queue= []
    def enqueue(self,task):
        if len(self.queue)>=self.capacity:
            print("Queue is full! cannot add task:",task)
        else:
            self.queue.append(task)
            print(f"task '{task}' scheduled")
    def dequeue(self):
        if not self.queue:
            print("Queue is empty, No task to exexute")
            return None
        task=self.queue.pop(0)
        print(f"task '{task}' executed")
        return task
    def display(self):
        if not self.queue:
            print("Queue is empty")
        else:
            print("current queue:",self.queue)

#Interactive simulation
print("-----Linear Queue Simulation-----")
capacity=int(input("Enter the maximum capacity of queue"))

```

```

linear_q=LinearQueue(capacity)
while True:
    print("\n Choose an operation:")
    print("1. Enqueue task")
    print("2. Dequeue task")
    print("3. Display task")
    print("4. Exit")
    choice=input("Enter your choice in (1-4):")
    if choice == '1':
        task=int(input("Enter the number"))
        linear_q.enqueue(task)
    elif choice=="2":
        linear_q.dequeue()
    elif choice=="3":
        linear_q.display()
    elif choice=="4":
        print("Exiting....")
        break
    else:
        print("Invalid choice, Try again")

```

Output:

```

-----Linear Queue Simulation-----
Enter the maximum capacity of queue7

Choose an operation:
1. Enqueue task
2. Dequeue task
3. Display task
4. Exit
Enter your choice in (1-4):1
Enter the number56
task '56' scheduled

Choose an operation:
1. Enqueue task
2. Dequeue task
3. Display task
4. Exit
Enter your choice in (1-4):2
task '56' executed

```

2] Circular Queue:

Input:

```
class CircularQueue:
    def __init__(self,capacity):
        self.capacity=capacity
        self.queue=[None] * capacity
        self.front=self.rear=-1
    def enqueue(self,task):
        #Queue full condition
        if (self.rear+1) % self.capacity == self.front:
            print("Queue is full! cannot add task:", task)
            return
        if self.front==-1:#First element
            self.front=0
        self.rear=(self.rear+1) % self.capacity
        self.queue[self.rear]=task
        print(f"task '{task}' scheduled")
    def dequeue(self):
        if self.front ==-1:
            print("Queue is empty no task to execute")
            return None
        task=self.queue[self.front]
        if self.front==self.rear:
            self.front=self.rear=-1
        else:
            self.front=(self.front+1)%self.capacity
            print(f"task '{task}' executed")
    def display(self):
        if self.front==-1:
            print("Queue is empty")
            return
        print("current queue:",end=" ")
        i=self.front
        while True:
            print(self.queue[i],end=" ")
            if i==self.rear:
                break
            i=(i+1)%self.capacity
        print()
#Interactive simulation
print("-----Circular Queue Simulation-----")
capacity=int(input("Enter the maximum capacity of queue"))
circular_q=CircularQueue(capacity)
while True:
    print("\n Choose an operation:")
    print("1. Enqueue task")
```

```

print("2. Dequeue task")
print("3. Display task")
print("4. Exit")
choice=input("Enter your choice in (1-4):")
if choice == '1':
    task=input("Enter the value")
    circular_q.enqueue(task)
elif choice=="2":
    circular_q.dequeue()
elif choice=="3":
    circular_q.display()
elif choice=="4":
    print("Exiting....")
    break
else:
    print("Invalid choice, Try again")

```

Output:

```

-----Circular Queue Simulation-----
Enter the maximum capacity of queue7

Choose an operation:
1. Enqueue task
2. Dequeue task
3. Display task
4. Exit
Enter your choice in (1-4):1
Enter the value89
task '89' scheduled

Choose an operation:
1. Enqueue task
2. Dequeue task
3. Display task
4. Exit
Enter your choice in (1-4):2

Choose an operation:
1. Enqueue task
2. Dequeue task
3. Display task
4. Exit
Enter your choice in (1-4):3
Queue is empty

```

3] Discuss memory utilization in linear vs circular queues

• Linear Queue

- Implemented using arrays.
- Once the rear reaches the end of the array, no more elements can be added even if there is free space at the beginning (after dequeues).
- Leads to memory wastage because front spaces cannot be reused without shifting elements.

• Circular Queue

- Uses wrap-around logic with modulo (%) operator.
- After rear reaches the end, it goes back to the start (if space is available).
- Utilizes memory efficiently since all slots of the array can be reused.
- No shifting is required.

Practical No 7

Aim: Tree Traversals and Binary Search Trees

- Create a binary search tree (BST) from a dataset.
 - Perform and visualize in-order, pre-order, and post-order traversals.
 - Use traversal results to derive sorted sequences.
-

1] To traversal of binary search tree**Input:**

class Node:

```
def __init__(self,key):
```

```
    self.key=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 Node(key)
```

```
    if key<root.key:
```

```
        root.left=self.insert(root.left,key)
```

```
    else:
```

```
        root.right=self.insert(root.right,key)
```

```
    return root
```

```
def inorder(self,root,result):
```

```
    if root:
```

```
        self.inorder(root.left,result)
```

```
        result.append(root.key)
```

```
        self.inorder(root.right,result)
```

```
def preorder(self,root,result):
```

```
    if root:
```

```
        result.append(root.key)
```

```
        self.preorder(root.left,result)
```

```
        self.preorder(root.right,result)
```

```

def postorder(self,root,result):
    if root:
        self.postorder(root.left,result)
        self.postorder(root.right,result)
        result.append(root.key)

#sample dataset
dataset=[50,30,70,20,40,60,80]

bst=BST()

root=None

for value in dataset:
    root=bst.insert(root,value)

#Traversals
inorder_result= []
bst.inorder(root,inorder_result)
preorder_result= []
bst.preorder(root,preorder_result)
postorder_result= []
bst.postorder(root,postorder_result)

#results
print("In-order Traversal: ",inorder_result)
print("pre-order Traversal: ",preorder_result)
print("post-order Traversal: ",postorder_result)

```

Output:

```

----- RESTART: D:/64010/pla
In-order Traversal:  [20, 30, 40, 50, 60, 70, 80]
pre-order Traversal:  [50, 30, 20, 40, 70, 60, 80]
post-order Traversal:  [20, 40, 30, 60, 80, 70, 50]

```

Practical No 8

Aim: Graph Representations and Traversals

- Represent graphs using adjacency matrices and lists.
- Implement BFS and DFS to explore graph components.
- Use graphs for mapping routes or exploring social networks.

Input:

```

from collections import deque
# Graph as adjacency list (dictionary)
graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A", "E"],
    "D": ["B"],
    "E": ["C"]
}
def bfs(start):
    visited, queue, order = set([start]), deque([start]), []
    while queue:
        node = queue.popleft()
        order.append(node)
        for nb in graph[node]:
            if nb not in visited:
                visited.add(nb)
                queue.append(nb)
    return order
def dfs(start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    order = [start]
    for nb in graph[start]:
        if nb not in visited:
            order.extend(dfs(nb, visited))
    return order
print("BFS from A:", bfs("A"))
print("DFS from A:", dfs("A"))

```

Output:

TEST

```

BFS from A: ['A', 'B', 'C', 'D', 'E']
DFS from A: ['A', 'B', 'D', 'C', 'E']

```