Practical No 1

Aim: Write a program to implement Abstract Data Type (ADT)

```
# Book Class
class Book:
    def __init__(self, isbn, title, author):
        self.isbn = isbn.strip()  # Removes extra whitespaces from ISBN
        self.title = title.strip()  # Removes extra whitespaces from title
        self.author = author.strip()  # Removes extra whitespaces from author
        self.available = True  # Book is available by default
        self.borrowed_by = None  # No borrower initially

    def borrow(self, name):
        # Borrow the book if it is available
        if self.available:
            self.available = False  # Mark the book as borrowed
            self.borrowed_by = name.title()  # Save borrower's name in title-case
            return True  # Return True if borrowed successfully
        return False  # Return False if already borrowed

    def return_book(self):
        # Return the book if it is currently borrowed
        if not self.available:
            self.available = True  # Mark the book as available
            self.borrowed_by = None  # Clear borrower's name
            return True  # Return True if returned successfully
        return False  # Return False if not borrowed

    def get_info(self):
        # Return formatted string with the book details
        return f"{self.title.upper()} by {self.author.upper()} - " + \
               ("Available" if self.available else f"Borrowed by {self.borrowed_by}")

# Library Class
class Library:
    def __init__(self):
        # Initialize the library with an empty dictionary of books
        self.books = {}  # Key = ISBN, Value = Book Object

    def add_book(self, book):
        # Add a book to the library if ISBN not already present
        if book.isbn not in self.books:
            self.books[book.isbn] = book  # Add book to the library dictionary
            return True  # Return True if added successfully
        return False  # Return False if ISBN already present
```

```
    def remove_book(self, isbn):
        # Remove a book by ISBN using dictionary pop()
        removed = self.books.pop(isbn, None)
        return removed is not None  # Return True if removed successfully, False otherwise

    def borrow_book(self, isbn, name):
        # Borrow a book using its ISBN and borrower's name
        book = self.books.get(isbn)  # Get the book if exists
        return book.borrow(name) if book else False  # Borrow the book if exists, False otherwise

    def return_book(self, isbn):
        # Return a book using its ISBN
        book = self.books.get(isbn)
        return book.return_book() if book else False

    def show_books(self):
        # Display all books in the library with their status
        print("Library Books List:")
        for key, book in self.books.items():  # Loop through dictionary items
            print(f"[{key}] {book.get_info()}")  # Show book details

    def search_books(self, keyword):
        # Search books by keyword in title or author (case-insensitive)
        keyword = keyword.lower()
        found = []  # List to store matching books
        for book in self.books.values():
            # Check if keyword is in title or author
            if keyword in book.title.lower() or keyword in book.author.lower():
                found.append(book)  # Add matching book to the list
        if found:
            # Sort found books by title
            found.sort(key=lambda b: b.title)
            print("Search Results:")
            for b in found:
                print(f" - {b.get_info()}")
        else:
            print("No book found with that keyword")

# Example Usage

# Create a library instance
library = Library()

# Create book instances
b1 = Book("101", "Python Programming", "John Zelle")
b2 = Book("102", "Artificial Intelligence", "Stuart Russell")
b3 = Book("103", "Data Structures", "Mark Allen Weiss")

# Add books to the library
```

```
library.add_book(b1)
library.add_book(b2)
library.add_book(b3)

# Display all books
library.show_books()

# Borrow a book with ISBN "102" by user "Alice"
library.borrow_book("102", "Alice")
print("\nAfter Borrowing:\n")
library.show_books()  # Display updated list

# Return the borrowed book with ISBN "102"
library.return_book("102")
print("\nAfter Returning:\n")
library.show_books()  # Display updated list

# Search for books by keyword "Data"
print("\nSearch by 'Data':")
library.search_books("Data")
```

```
Library Books List:
[101] PYTHON PROGRAMMING by JOHN ZELLE - Available
[102] ARTIFICIAL INTELLIGENCE by STUART RUSSELL - Available
[103] DATA STRUCTURES by MARK ALLEN WEISS - Available

After Borrowing:

Library Books List:
[101] PYTHON PROGRAMMING by JOHN ZELLE - Available
[102] ARTIFICIAL INTELLIGENCE by STUART RUSSELL - Borrowed
by Alice
[103] DATA STRUCTURES by MARK ALLEN WEISS - Available

After Returning:

Library Books List:
[101] PYTHON PROGRAMMING by JOHN ZELLE - Available
[102] ARTIFICIAL INTELLIGENCE by STUART RUSSELL - Available
[103] DATA STRUCTURES by MARK ALLEN WEISS - Available

Search by 'Data':
Search Results:
- DATA STRUCTURES by MARK ALLEN WEISS - Available
```

Practical No 2

Aim: Write a program for building and using Singly Linked List

```
# Node class for singly linked-list
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Linked list class to manage tasks
class TaskList:
    def __init__(self):
        self.head = None

    def add_task(self, task):
        new_node = Node(task)
        if not self.head:
            self.head = new_node
            return
        curr = self.head
        while curr.next:
            curr = curr.next
        curr.next = new_node

    def remove_task(self, task):
        if not self.head:  # List is empty
            return False
        if self.head.data == task:
            self.head = self.head.next  # Remove head node
            return True
        curr = self.head
        while curr.next:
            if curr.next.data == task:
                curr.next = curr.next.next  # Bypass the node
                return True
            curr = curr.next
        return False  # Task not found

    def display_tasks(self):
        if not self.head:
            print("Task list is empty")
            return
        print("Task list:")
        curr = self.head
        while curr:
            print(" -", curr.data)
            curr = curr.next

    def search_task(self, keyword):
        curr = self.head
        found = False
```

```
        keyword = keyword.lower()
        while curr:
            if keyword in curr.data.lower():
                print(f"Found: {curr.data}")
                found = True
            curr = curr.next
        if not found:
            print("No matching task found")

# Example Usage
# Create an instance of TaskList
todo = TaskList()

# Add tasks to the list
todo.add_task("Prepare monthly financial report")
todo.add_task("Email project updates to team")
todo.add_task("Schedule client meeting for next week")
todo.add_task("Organize files and documents")
todo.add_task("Update website with new content")

# Display all tasks
todo.display_tasks()

# Remove a task
print("\nRemoving 'Organize files and documents'")
todo.remove_task("Organize files and documents")

# Display updated list of tasks
todo.display_tasks()

# Search for tasks containing 'project'
print("\nSearching for 'project':")
todo.search_task("project")

# Display tasks again to confirm state
todo.display_tasks()
```

```
Task list:
 - Prepare monthly financial report
 - Email project updates to team
 - Schedule client meeting for next week
 - Organize files and documents
 - Update website with new content

Removing 'Organize files and documents'
Task list:
 - Prepare monthly financial report
 - Email project updates to team
 - Schedule client meeting for next week
 - Update website with new content

Searching for 'project':
Found: Email project updates to team
Task list:
 - Prepare monthly financial report
 - Email project updates to team
 - Schedule client meeting for next week
 - Update website with new content
```

Practical No 3

Aim: Write a program for polynomial operations using Linked List

```
# Node to represent a term in the polynomial
class Node:
    def __init__(self, coeff, exp):
        self.coeff = coeff
        self.exp = exp
        self.next = None

# Polynomial class
class Polynomial:
    def __init__(self):
        self.head = None

    def add_term(self, coeff, exp):
        if coeff == 0:
            return
        new = Node(coeff, exp)
        # Insert at the beginning or before head if exponent is largest
        if not self.head or exp > self.head.exp:
            new.next = self.head
            self.head = new
            return
        curr = self.head
        prev = None
        # Traverse to find the correct position or combine like terms
        while curr and curr.exp >= exp:
            if curr.exp == exp:
                curr.coeff += coeff
                # Remove term if coefficient becomes zero
                if curr.coeff == 0:
                    if prev:
                        prev.next = curr.next
                    else:
                        self.head = curr.next
                return
            prev = curr
            curr = curr.next
        # Insert in the middle or end
        new.next = curr
        if prev:
            prev.next = new
        else:
            self.head = new

    def display(self):
```

```
        curr = self.head
        result = []
        while curr:
            coeff = curr.coeff
            exp = curr.exp
            if coeff > 0 and result:
                result.append(f"+ {coeff}x^{exp}")
            else:
                result.append(f"{coeff}x^{exp}")
            curr = curr.next
        print(" ".join(result) if result else "0")

    def add(self, other):
        return self._merge(other, add=True)

    def subtract(self, other):
        return self._merge(other, add=False)

    def _merge(self, other, add=True):
        p1 = self.head
        p2 = other.head
        result = Polynomial()
        while p1 and p2:
            if p1.exp == p2.exp:
                c = p1.coeff + p2.coeff if add else p1.coeff - p2.coeff
                result.add_term(c, p1.exp)
                p1, p2 = p1.next, p2.next
            elif p1.exp > p2.exp:
                result.add_term(p1.coeff, p1.exp)
                p1 = p1.next
            else:
                c = p2.coeff if add else -p2.coeff
                result.add_term(c, p2.exp)
                p2 = p2.next
        while p1:
            result.add_term(p1.coeff, p1.exp)
            p1 = p1.next
        while p2:
            c = p2.coeff if add else -p2.coeff
            result.add_term(c, p2.exp)
            p2 = p2.next
        return result

# Example Usage

# P1 = 4x^4 + 2x^2 + 6
p1 = Polynomial()
p1.add_term(4, 4)
p1.add_term(2, 2)
```

```
p1.add_term(6, 0)

# P2 = 3x^5 - x^2 + 4x + 1
p2 = Polynomial()
p2.add_term(3, 5)
p2.add_term(-1, 2)
p2.add_term(4, 1)
p2.add_term(1, 0)

print("P1: ", end=""); p1.display()
print("P2: ", end=""); p2.display()
print("P1 + P2: ", end=""); p1.add(p2).display()
print("P1 - P2: ", end=""); p1.subtract(p2).display()

# P3 = 5x^3 - 3x + 2
p3 = Polynomial()
p3.add_term(5, 3)
p3.add_term(-3, 1)
p3.add_term(2, 0)

# P4 = -2x^3 + 4x^2 - x + 5
p4 = Polynomial()
p4.add_term(-2, 3)
p4.add_term(4, 2)
p4.add_term(-1, 1)
p4.add_term(5, 0)

print("\nP3: ", end=""); p3.display()
print("P4: ", end=""); p4.display()
print("P3 + P4: ", end=""); p3.add(p4).display()
print("P3 - P4: ", end=""); p3.subtract(p4).display()

# P5 = x^6 + 3x^4 + 2x^2
p5 = Polynomial()
p5.add_term(1, 6)
p5.add_term(3, 4)
p5.add_term(2, 2)

# P6 = -x^5 + 2x^4 - x + 7
p6 = Polynomial()
p6.add_term(-1, 5)
p6.add_term(2, 4)
p6.add_term(-1, 1)
p6.add_term(7, 0)

print("\nP5: ", end=""); p5.display()
print("P6: ", end=""); p6.display()
print("P5 + P6: ", end=""); p5.add(p6).display()
print("P5 - P6: ", end=""); p5.subtract(p6).display()
```

```
P1: 4x^4 + 2x^2 + 6x^0
P2: 3x^5 -1x^2 + 4x^1 + 1x^0
P1 + P2: 3x^5 + 4x^4 + 1x^2 + 4x^1 + 7x^0
P1 - P2: -3x^5 + 4x^4 + 3x^2 -4x^1 + 5x^0

P3: 5x^3 -3x^1 + 2x^0
P4: -2x^3 + 4x^2 -1x^1 + 5x^0
P3 + P4: 3x^3 + 4x^2 -4x^1 + 7x^0
P3 - P4: 7x^3 -4x^2 -2x^1 -3x^0

P5: 1x^6 + 3x^4 + 2x^2
P6: -1x^5 + 2x^4 -1x^1 + 7x^0
P5 + P6: 1x^6 -1x^5 + 5x^4 + 2x^2 -1x^1 + 7x^0
P5 - P6: 1x^6 + 1x^5 + 1x^4 + 2x^2 + 1x^1 -7x^0
```

Practical No 4

Aim: Write a program for working with Doubly Linked list

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

# Doubly Linked List class
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
        self.size = 0

    # Insert at the beginning
    def insert_from_head(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
        self.size += 1

    # Insert at the end
    def insert_from_tail(self, data):
        new_node = Node(data)
        if not self.tail:
            self.tail = new_node
            self.head = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
        self.size += 1

    # Insert at a specific position (middle)
    def insert_at_position(self, data, position):
        if position <= 0:
            self.insert_from_head(data)
        elif position >= self.size:
            self.insert_from_tail(data)
```

```
        else:
            new_node = Node(data)
            temp = self.head
            for _ in range(position - 1):
                temp = temp.next
            new_node.next = temp.next
            new_node.prev = temp
            temp.next.prev = new_node
            temp.next = new_node
            self.size += 1

    # Delete from beginning
    def delete_from_head(self):
        if not self.head:
            print("List is empty.")
            return
        self.head = self.head.next
        if self.head:
            self.head.prev = None
        else:
            self.tail = None
        self.size -= 1

    # Delete from end
    def delete_from_tail(self):
        if not self.tail:
            print("List is empty.")
            return
        self.tail = self.tail.prev
        if self.tail:
            self.tail.next = None
        else:
            self.head = None
        self.size -= 1

    # Delete from a specific position (middle)
    def delete_at_position(self, position):
        if self.size == 0:
            print("List is empty.")
            return
        if position <= 0:
            self.delete_from_head()
        elif position >= self.size - 1:
            self.delete_from_tail()
        else:
            temp = self.head
            for _ in range(position):
                temp = temp.next
            temp.prev.next = temp.next
```

```
        temp.next.prev = temp.prev
        self.size -= 1

    # Print list from head to tail
    def print_list(self):
        cur = self.head
        while cur:
            print(cur.data, end=" ")
            cur = cur.next
        print()

    # Print list from tail to head
    def print_list_reverse(self):
        cur = self.tail
        while cur:
            print(cur.data, end=" ")
            cur = cur.prev
        print()

    # Traverse the list (same as print_list)
    def traverse(self):
        self.print_list()

# Main function for testing with new examples
def main():
    dll = DoublyLinkedList()

    # Insert at head
    dll.insert_from_head(100)
    dll.insert_from_head(200)

    # Insert at tail
    dll.insert_from_tail(300)
    dll.insert_from_tail(400)

    print("List after head & tail insertions:")
    dll.print_list()

    # Insert at middle (position 2)
    dll.insert_at_position(250, 2)
    print("List after inserting 250 at position 2:")
    dll.print_list()

    # Delete from middle (position 2)
    dll.delete_at_position(2)
    print("List after deleting from position 2:")
    dll.print_list()

    # Delete from head
```

```
    dll.delete_from_head()
    print("List after deleting from head:")
    dll.print_list()

    # Delete from tail
    dll.delete_from_tail()
    print("List after deleting from tail:")
    dll.print_list()

    # Reverse print
    print("Reverse list:")
    dll.print_list_reverse()

# Run the main function
if __name__ == "__main__":
    main()
```

```
List after head & tail insertions:
200 100 300 400
List after inserting 250 at position 2:
200 100 250 300 400
List after deleting from position 2:
200 100 300 400
List after deleting from head:
100 300 400
List after deleting from tail:
100 300
Reverse list:
300 100
```

Practical No 5

Aim: Write a program for implementing and using Stack ADT

```
# Stack ADT implementation
class Stack:
    def __init__(self):
        self.items = []

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

    def pop(self):
        return self.items.pop() if not self.is_empty() else None

    def peek(self):
        return self.items[-1] if not self.is_empty() else None

    def is_empty(self):
        return len(self.items) == 0

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

# Balanced Delimiter Checker with detailed comments
class DelimiterMatcher:
    def __init__(self):
        # Mapping of closing brackets to their corresponding opening brackets
        self.pairs = {')': '(', '}': '{', ']': '['}

    def is_matching(self, code_string):
        # Create an empty stack to hold opening brackets
        stack = Stack()

        # Traverse each character in the input string
        for char in code_string:
            if char in "([{":
                stack.push(char)
            elif char in ")]}":
                # Step 1: If the stack is empty, there's no opening bracket
                if stack.is_empty():
                    return False
                # Step 2: Pop the last opening bracket from the stack
                top = stack.pop()
                # Step 3: Get the expected matching opening bracket
                expected_opening = self.pairs[char]
                # Step 4: Compare the popped bracket with the expected one
                if top != expected_opening:
```

```
                return False  # Mismatch found
        # Step 5: After processing all characters, if the stack is empty, all brackets matched correctly
        return stack.is_empty()

# Prefix to Postfix Converter
class PrefixToPostfixConverter:
    def __init__(self):
        pass

    def is_operand(self, ch):
        return ch.isalpha() or ch.isdigit()

    def convert(self, prefix_expr):
        stack = Stack()
        for char in reversed(prefix_expr):
            if self.is_operand(char):
                stack.push(char)
            else:
                op1 = stack.pop()
                op2 = stack.pop()
                new_expr = op1 + op2 + char
                stack.push(new_expr)
        return stack.pop()

# Example Usage

def main():
    # Test DelimiterMatcher
    match = DelimiterMatcher()
    c_code = """
    #include <stdio.h>
    int main(){
        int a=10;
        if (a>5){
            printf("hello");
        } else {
            printf("Bye");
        }
        return 0;
    }
    """
    print("Delimiter check (C code):", "Balanced" if match.is_matching(c_code) else "Not Balanced")

    expr1 = "{[()()]}"
    print("Delimiter check ({[()()]}):", "Balanced" if match.is_matching(expr1) else "Not Balanced")

    expr2 = "(())()"
    print("Delimiter check ((())()):", "Balanced" if match.is_matching(expr2) else "Not Balanced")
```

```python
    expr3 = "{[(])}"
    print("Delimiter check ({[(])}):", "Balanced" if match.is_matching(expr3) else "Not Balanced")

    # Test PrefixToPostfixConverter
    converter = PrefixToPostfixConverter()
    prefix_expr = "*+AB-CD"
    postfix_expr = converter.convert(prefix_expr)
    print("\nPrefix expression:", prefix_expr)
    print("Converted Postfix expression:", postfix_expr)

# Run the main function
if __name__ == "__main__":
    main()
```

```
Delimiter check (C code): Balanced
Delimiter check ({[()()]}): Balanced
Delimiter check ((())()): Balanced
Delimiter check ({[(])}): Not Balanced

Prefix expression: *+AB-CD
Converted Postfix expression: AB+CD-*
```