Distinct Palindromic Subsequences

Use Case

Design a **PalindromicSubsequenceCounter** class that analyzes a given string and identifies all **distinct palindromic subsequences** present in it.

Objective

To implement a solution using **Object-Oriented Programming (OOP)** concepts that:

- Generates all possible non-empty subsequences of a string

- Checks which subsequences are palindromes

- Counts and displays **distinct palindromic subsequences**

- Validates input constraints safely

Class Description

The **PalindromicSubsequenceCounter** class acts as a blueprint for processing a string and performing palindrome-related operations.
 It encapsulates input validation, subsequence generation, palindrome checking, counting, and displaying results.

Properties (Data Members)

1. **s**
   Stores the input string provided by the user.

2. **n**
   Stores the length of the input string.

Behaviors (Member Functions)

1. `validate_input()`

- Checks whether the length of the input string is between **2 and 9**.

- Raises an error if the condition is not satisfied.

- Ensures the algorithm runs within safe limits.

---

2. `is_palindrome(text)`

- Accepts a string as input.

- Compares the string with its reverse.

- Returns `True` if it is a palindrome, otherwise `False`.

---

3. `generate_subsequences()`

- Generates all **non-empty subsequences** of the input string.

- Uses **bit masking** to explore every possible character combination.

- Returns a list containing all subsequences.

---

4. `count_distinct_palindromic_subsequences()`

- Validates the input string.

- Iterates through all generated subsequences.

- Checks each subsequence for the palindrome property.

- Stores palindromic subsequences in a **set** to ensure uniqueness.

- Returns the total count of distinct palindromic subsequences.

---

5. `display_palindromes()`

- Identifies all distinct palindromic subsequences.

- Sorts them alphabetically.

- Displays each palindrome clearly to the user.

Working Process

1. The user enters a string.

2. A **PalindromicSubsequenceCounter** object is created using the input string.

3. The input string is validated for length constraints.

4. All possible non-empty subsequences are generated.

5. Each subsequence is checked to determine if it is a palindrome.

6. Distinct palindromic subsequences are stored using a set.

7. The total count of distinct palindromic subsequences is calculated.

8. All distinct palindromic subsequences are displayed in sorted order.

9. Errors are handled gracefully using exception handling.

Conclusion

The **PalindromicSubsequenceCounter** class provides a **structured and reliable approach** to identifying palindromic patterns within a string.
 By combining input validation, bitwise logic, and set-based uniqueness, it demonstrates effective use of **Object-Oriented Programming principles**, including **encapsulation, modularity, and data safety**.

DFD

0 level DATA FLOW DIAGRAM

Level 0 (Context): The user provides a raw string, and the system transforms it into statistical data (the count) and structured data (the list).
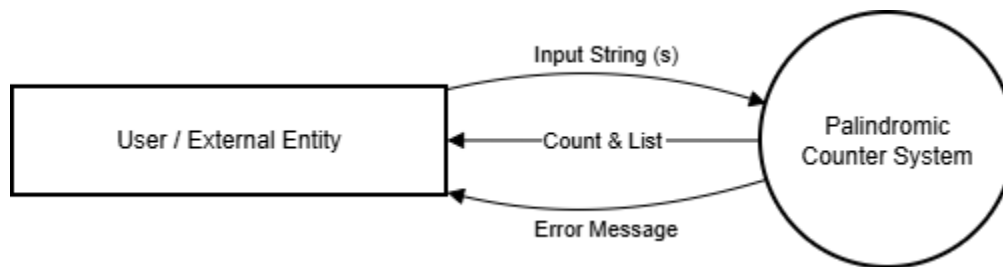
1 LEVEL DATA FLOW DIAGRAM

Level 1 :The logic is split into linear stages:

1. **Sanitization: Ensuring the data is safe to process (Validation).**
2. **Transformation: Converting the single string into a multitude of potential substrings (Generation).**
3. **Selection: Keeping only the data that satisfies the condition (Palindrome Check).**
4. **Reduction: Removing redundancy (Set operation)**

DIAGRAM–

```mermaid
flowchart TD
    Start([Start])
    Start --> EnterString[/Enter string s/]
    EnterString --> CreateObject[Create object<br/>PalindromicSubsequenceCounter<br/>store s and n]
    CreateObject --> Call[Call countDistinctPalindromicSubsequences]
    Call --> IsLengthValid{Is length valid}
    IsLengthValid -->|Yes| CreateSet[Create empty set palindromes]
    IsLengthValid -->|No| DisplayError[Display input error]
    CreateSet --> GenerateAll[Generate all subsequences]
    GenerateAll --> Loop[Loop through each subsequence]
    Loop --> IsPalindrome1{Is palindrome}
    IsPalindrome1 -->|No| Loop
    IsPalindrome1 -->|Yes| AddSubseq[Add subsequence to set]
    AddSubseq --> Loop
    Loop --> ReturnCount[Return count of set]
    ReturnCount --> PrintTotal[Print total count]
    PrintTotal --> CallDisplay[Call displayPalindromes]
    CallDisplay --> GenerateAgain[Generate subsequences again]
    GenerateAgain --> IsPalindrome2{Is palindrome}
    IsPalindrome2 -->|No| GenerateAgain
    IsPalindrome2 -->|Yes| AddToSet[Add to set]
    GenerateAgain --> PrintAll[Print all palindromic subsequences]
    PrintAll --> End([End])
    DisplayError --> End
```

- Start
- Enter string s
- Create object PalindromicSubsequenceCounter store s and n
- Call countDistinctPalindromicSubsequences
- Is length valid
  - Yes → Create empty set palindromes
  - No → Display input error
- Create empty set palindromes
- Generate all subsequences
- Loop through each subsequence
  - Is palindrome
    - No
    - Yes → Add subsequence to set
- Return count of set
- Print total count
- Call displayPalindromes
- Generate subsequences again
  - Is palindrome
    - No
    - Yes → Add to set
- Print all palindromic subsequences
- Display input error
- End