1)

Let’s break down the provided Python code step by step to understand its working.

Objective

The program calculates specific groups of students based on their participation in sports (cricket, badminton, and football) using basic list operations without relying on Python’s `set` built-in functions.

Step-by-Step Explanation

1. Functions for List Operations

- **`union(a, b)`**
  - Combines two lists (`a` and `b`) without duplicates.
  - Starts with a copy of `a` and adds elements from `b` that are not already in `a`.
  - Example:
    ```python
    A = [1, 2, 3]
    B = [3, 4, 5]
    Union(a, b) -> [1, 2, 3, 4, 5]
    ```

- **`minus(a, b)`**
  - Finds elements in list `a` that are **not** in list `b`.
  - Example:
    ```python
    ```

  A = [1, 2, 3]

  B = [3, 4]

  Minus(a, b) -> [1, 2]

  ```

- **`intersection(a, b)`**

 - Finds common elements between lists `a` and `b`.

 - Example:

  ```python

  A = [1, 2, 3]

  B = [3, 4]

  Intersection(a, b) -> [3]

  ```

---

2. Input Lists

```python

U = input(“ENTER THE UNIVERSAL LIST (all students): “).split(“,”)

B = input(“ENTER THE BADMINTON LIST: “).split(“,”)

C = input(“ENTER THE CRICKET LIST: “).split(“,”)

F = input(“ENTER THE FOOTBALL LIST: “).split(“,”)

```

- The program takes input as comma-separated strings for:

 - `u`: Universal list of all students.

 - `b`: Students who play badminton.

 - `c`: Students who play cricket.

 - `f`: Students who play football.
```

- **Input Cleaning**

```python
U = list(dict.fromkeys([x.strip() for x in u]))
B = list(dict.fromkeys([x.strip() for x in b]))
C = list(dict.fromkeys([x.strip() for x in c]))
F = list(dict.fromkeys([x.strip() for x in f]))
```

- `x.strip()`: Removes extra spaces from each student's name.

- `list(dict.fromkeys(...))`: Removes duplicate entries.

---

3. A) Students Who Play Both Cricket and Badminton

```python
Intersection(c, b)
```

- Calls `intersection` to find common students in cricket (`c`) and badminton (`b`).

---

4. B) Students Who Play Either Cricket or Badminton but Not Both

```python
Union(minus(c, intersection(c, b)), minus(b, intersection(c, b)))
```

- Finds students who play **either** cricket or badminton but **not both**:

  - `minus(c, intersection(c, b))`: Students in cricket but not badminton.

  - `minus(b, intersection(c, b))`: Students in badminton but not cricket.

  - Combines both groups using `union`.

---

5. C) Number of Students Who Play Neither Cricket nor Badminton

```python
Neither = minus(minus(u, c), b)
Len(neither)
```

- Steps:
  - `minus(u, c)`: Students in the universal list (`u`) but not in cricket.
  - `minus(..., b)`: Students from the above result who also don't play badminton.
  - `len(neither)`: Counts these students.

---

6. D) Students Who Play Cricket and Football but Not Badminton

```python
Cricket_football_not_badminton = minus(intersection(c, f), b)
Len(cricket_football_not_badminton)
```

- Steps:
  - `intersection(c, f)`: Students who play both cricket and football.
  - `minus(..., b)`: Excludes students who also play badminton.
  - `len(...)`: Counts these students.

Output

The program prints the following:

1. List of students who play both cricket and badminton.

2. List of students who play either cricket or badminton but not both.

3. Count of students who play neither cricket nor badminton.

4. Count of students who play cricket and football but not badminton.

Example Input and Output

Input:

```
ENTER THE UNIVERSAL LIST (all students): A, B, C, D, E, F, G

ENTER THE BADMINTON LIST: A, B, C

ENTER THE CRICKET LIST: B, C, D

ENTER THE FOOTBALL LIST: C, D, E
```

Processing:

- (a) Intersection of cricket and badminton: `[‘B’, ‘C’]`

- (b) Either cricket or badminton but not both: `[‘A’, ‘D’]`

- (c) Neither cricket nor badminton: `[‘E’, ‘F’, ‘G’]`

- (d) Cricket and football but not badminton: `[‘D’]`

Output:

```
List of students who play CRICKET and BADMINTON: [‘B’, ‘C’]

List of students who play either CRICKET or BADMINTON but not BOTH: [‘A’, ‘D’]

Number of students who play neither CRICKET nor BADMINTON: 3

Number of students who play CRICKET and FOOTBALL but not BADMINTON: 1
```

This modular and well-structured code ensures clarity and flexibility in handling similar group-based problems.

2): Prem Chuniyan

Roll Number:75

ASSIGNMENT NO.:02

TITLE: Write a Python program to store marks scored in subject“Fundamental of Data

Structure” by N students in the class. Write functions to compute following:

a) The average score of class

b) Highest score and lowest score of class

c) Count of students who were absent for the test

d) Display mark with highest frequency

CODE:

N = int(input(“Enter the number of students: “)) present, Total, min_score, max_score, absent = 0, 0, 51, -1, 0 marks

= [] for i in range(n):

Temp = input(f"Enter the marks of student {i + 1} or AB if absent: ")

Marks.append(temp)

If temp != "AB":

Present += 1 score

= int(temp) total

+= score

If score > max_score:

Max_score = score if

Score < min_score:

Min_score = score

Else:

Absent += 1

Calculate average score avg = total /

Present if present > 0 else 0

Frequency calculation

Frequency = {} for mark

In marks:

If mark != “AB”:

3)### **Objective**

The code processes a list of books with the following operations:

1. Removes duplicate entries (based on book ID).

2. Sorts the books in ascending order of cost.

3. Counts the books with a cost greater than 500.

4. Copies books with a cost less than or equal to 500 into a new list.

Code Explanation

**1. Function: `delet(a)` **

```python
Def delet(a):
   Ans = []
   Seen = set()
   For item in a:
      If item[0] not in seen:
         Ans.append(item)
         Seen.add(item[0])
   Return ans
```

- **Purpose**: Removes duplicate books based on their ID (first element of each sublist).

- **Steps**:

  1. Initialize an empty list `ans` to store unique books and a set `seen` to track Ids.

  2. Iterate through the input list `a`.

  3. If the book ID (`item[0]`) is not in `seen`:

     - Add the book to `ans`.

     - Mark the ID as "seen" by adding it to the set.

  4. Return the list `ans` without duplicates.

- **Time Complexity**:

  - Iterating through \(a \): \(O(n) \).

  - Checking and adding Ids in `seen` (set operations): \(O(1) \) per element.

  - Overall: \(O(n) \).

- **Space Complexity**:

  - List `ans` stores unique books: \(O(n) \) in the worst case (no duplicates).

  - Set `seen` stores unique Ids: \(O(n) \).

  - Total: \(O(n) \).

---

2. Function: `count(a)`

```python
Def count(a):
    F = 0
    Less = []
    For i in a:
        If int(i[1]) > 500:
            F += 1
        Else:
            Less.append(i)
    Return f, less
```

- **Purpose**:
  1. Count books with a cost greater than 500.
  2. Create a new list `less` containing books with a cost less than or equal to 500.
- **Steps**:
  1. Initialize `f` to count books with a cost greater than 500.
  2. Iterate through the list `a`.
  3. Convert the cost (`i[1]`) to an integer and check:
     - If greater than 500, increment `f`.
     - Otherwise, append the book to the `less` list.
  4. Return the count `f` and the list `less`.

- **Time Complexity**:
  - Iterating through a: $O(n)$.
  - Cost conversion (string to integer): $O(1)$ per element.

  - Overall: \(O(n) \).

- **Space Complexity**:

  - List `less` stores books with a cost ≤ 500: \(O(n) \) in the worst case.

  - Total: \(O(n) \).

---

3. Function: `sort(a)`

```python
Def sort(a):
  Ans = a.copy()
  For i in range(len(ans)):
    For j in range(0, len(ans) – i – 1):
      If int(ans[j][1]) > int(ans[j + 1][1]):
        Ans[j], ans[j + 1] = ans[j + 1], ans[j]
  Return ans
```

- **Purpose**: Sorts the books in ascending order of cost using the **Bubble Sort** algorithm.

- **Steps**:

  1. Create a copy of the input list `a` to avoid modifying the original.

  2. Perform nested iterations to compare adjacent elements.

  3. If the cost of the current book (`ans[j][1]`) is greater than the next book:

     - Swap their positions.

  4. Return the sorted list.

- **Time Complexity**:

  - Bubble Sort requires \(O(n^2) \) comparisons in the worst case.

  - For \(n \) elements: \(O(n^2) \).

- **Space Complexity**:
  - List `ans` (copy of `a`): \(O(n) \).
  - Total: \(O(n) \).

---

4. Main Program
```python
N = int(input("ENTER THE NO OF BOOKS: "))
Books = []

For i in range(n):
   A = input("ENTER THE ID OF BOOK " + str(i + 1) + ": ")
   B = input("ENTER THE COST OF BOOK " + str(i + 1) + ": ")
   Books.append([a, b])

Print("ORIGINAL LIST:", books)
```
- **Steps**:
  1. Input \(n \): Number of books.
  2. Collect book data (ID and cost) in a list `books`.
     - Each book is stored as a sublist: `[ID, cost]`.

- **Time Complexity**:
  - Input loop: \(O(n) \).
  - Input operations per book: \(O(1) \).
  - Total: \(O(n) \).
- **Space Complexity**:
  - List `books`: \(O(n) \).

---

5. Removing Duplicates

```python
Books_no_duplicates = delet(books)

Print("DELETE THE DUPLICATE ENTRIES:", books_no_duplicates)
```

- **Uses `delet` function** to remove duplicate books based on Ids.

- **Time Complexity**: \(O(n) \).

- **Space Complexity**: \(O(n) \).

---

6. Sorting Books by Cost

```python
Sorted_books = sort(books_no_duplicates)

Print("DISPLAY BOOK IN ASCENDING ORDER BASED ON COST OF BOOK:", sorted_books)
```

- **Uses `sort` function** to sort the books by cost.

- **Time Complexity**: \(O(n^2) \).

- **Space Complexity**: \(O(n) \).

---

7. Counting and Copying Books

```python
Tup = count(books_no_duplicates)

Print("COUNT NO OF BOOKS WITH COST MORE THAN 500:", tup[0])

Print("COPY BOOKS IN A NEW LIST WHICH HAS COST LESS THAN 500:", tup[1])
```

```

- **Uses `count` function** to:

  1. Count books with cost > 500.

  2. Create a list of books with cost ≤ 500.

- **Time Complexity**: \( O(n) \).

- **Space Complexity**: \( O(n) \).

---

### **Overall Complexity**

#### **Time Complexity**

- Input collection: \( O(n) \)

- Removing duplicates: \( O(n) \)

- Sorting books: \( O(n^2) \) (dominates overall complexity).

- Counting books: \( O(n) \)

- **Total Time Complexity**: \( O(n^2) \) (due to Bubble Sort).

#### **Space Complexity**

- Storage of books: \( O(n) \)

- Intermediate lists (`ans`, `less`): \( O(n) \)

- Set for duplicate removal: \( O(n) \)

- **Total Space Complexity**: \( O(n) \).

4)

### **Explanation of the Code**
```

This Python program performs the following tasks:

1. Sorts a list of percentages using two algorithms: **Selection Sort** and **Bubble Sort**.

2. Displays the top five scores from the sorted lists.

---

1. `selection_sort` Function

- **Purpose**: Implements the **Selection Sort** algorithm to sort a list in ascending order.

- **Process**:

  1. Iterate through the array using an outer loop (`i` is the current index being processed).

  2. For each index `i`, find the smallest element in the unsorted part of the array.

  3. Swap this smallest element with the element at index `i`.

  4. Repeat until the array is fully sorted.

- **Time Complexity**: \(O(n^2) \), as it involves two nested loops.

- **Space Complexity**: \(O(1) \), as it sorts in place.

---

2. `bubble_sort` Function

- **Purpose**: Implements the **Bubble Sort** algorithm to sort a list in ascending order.

- **Process**:

  1. Iterate through the array multiple times.

  2. During each pass, compare adjacent elements and swap them if they are out of order.

  3. The largest element “bubbles up” to its correct position after each pass.

  4. Repeat until the array is fully sorted.

- **Time Complexity**: \(O(n^2) \), as it involves two nested loops.

- **Space Complexity**: \(O(1) \), as it sorts in place.

---

3. `display_top_five` Function

- **Purpose**: Displays the top five scores in descending order from a sorted list.

- **Process**:

  1. Use the `sorted()` function to sort the array in descending order (`reverse=True`).

  2. Slice the first five elements from the sorted array.

  3. Print each of these scores with two decimal places using `f”{score:.2f}”`.

---

4. Main Program

- **Data**: A list of percentages: `[75, 82.3, 91, 89.88, 53, 63, 85, 95, 99]`.

- **Steps**:

  1. **Selection Sort**:

     - Call `selection_sort` with a copy of the percentages list.

     - Display the top five scores using `display_top_five`.

  2. **Bubble Sort**:

     - Call `bubble_sort` with another copy of the percentages list.

     - Display the top five scores using `display_top_five`.

Execution Flow

1. **Input**: The percentages list: `[75, 82.3, 91, 89.88, 53, 63, 85, 95, 99]`.

2. **Selection Sort**:

   - Sort the list in ascending order: `[53, 63, 75, 82.3, 85, 89.88, 91, 95, 99]`.

   - Extract the top five scores in descending order: `[99, 95, 91, 89.88, 85]`.

3. **Bubble Sort**:

- Similarly sorts the list in ascending order.

- Displays the same top five scores in descending order: `[99, 95, 91, 89.88, 85]`.

Output

The program outputs:

```
Selection Sort:

Top five scores:

99.00

95.00

91.00

89.88

85.00


Bubble Sort:

Top five scores:

99.00

95.00

91.00

89.88

85.00
```

Both sorting algorithms produce the same result as the input data is consistent. The code demonstrates the working of both algorithms and displays the top performers.### **Time Complexity Analysis**

1. Selection Sort

- **Outer Loop**: Runs \(n \) times, where \(n \) is the size of the array.

- **Inner Loop**: For each iteration of the outer loop, the inner loop runs \(n – i – 1 \) times.

- **Overall Time Complexity**:

\[

O(n^2)

\]

  - Because of the nested loops, the time complexity is quadratic.

---

2. Bubble Sort

- **Outer Loop**: Runs \(n \) times.

- **Inner Loop**: For each iteration of the outer loop, the inner loop runs \(n – i – 1 \) times.

- **Overall Time Complexity**:

\[

O(n^2)

\]

  - Similar to Selection Sort, it also has quadratic time complexity due to nested loops.

---

3. Display Top Five Scores

- **Sorting with `sorted()`**: The `sorted()` function sorts the array in \(O(n \log n) \) time using Timsort.

- **Slicing Top Five Elements**: Slicing a list takes \(O(k) \), where \(k = 5 \). This is constant time, \(O(1) \), since \(k \) is fixed.

- **Overall Time Complexity**:

\[

O(n \log n)

\]

Main Program’s Total Time Complexity

- **Selection Sort**: \(O(n^2) \)

- **Bubble Sort**: \(O(n^2) \)

- **Display Top Five (each call)**: \(O(n \log n) \)

Since Selection Sort and Bubble Sort are the dominant contributors to the program’s runtime, the **overall time complexity** is:

\[

O(n^2)

\]

Space Complexity Analysis

1. Selection Sort

- The algorithm sorts the array in place without using any additional data structures.

- **Space Complexity**:

\[

O(1)

\]

---

2. Bubble Sort

- Like Selection Sort, this algorithm also sorts the array in place.

- **Space Complexity**:

 \[

 O(1)

 \]

---

3. Display Top Five

- The `sorted()` function creates a new sorted copy of the array.

- **Space Complexity**:

 \[

 O(n)

 \]

Main Program’s Total Space Complexity

- Sorting using Selection Sort and Bubble Sort: \(O(1) + O(1) \)

- Displaying Top Five: \(O(n) \)

- **Overall Space Complexity**:

\[

O(n)

\]

5)### **Explanation of the Code**

This Python program demonstrates sorting an array of percentages using **Insertion Sort** and **Shell Sort**, and displays the top 5 scores in descending order.

---

1. `insertion_sort` Function

- **Purpose**: Implements the **Insertion Sort** algorithm to sort a list in ascending order.

- **Process**:

  1. Start from the second element (index 1) and iterate through the list.

  2. Compare the current element (`key`) with its preceding elements.

  3. Shift larger elements one position to the right to make space for the `key`.

  4. Insert the `key` into its correct position.

- **Time Complexity**:

  - Best case: \(O(n) \) (if the array is already sorted).

  - Worst case: \(O(n^2) \) (if the array is sorted in reverse order).

- **Space Complexity**: \(O(1) \), as sorting is performed in place.

---

2. `shell_sort` Function

- **Purpose**: Implements the **Shell Sort** algorithm, a generalized version of Insertion Sort that improves efficiency by comparing elements separated by a "gap".

- **Process**:

  1. Start with a gap equal to half the array size (\(n//2 \)).

  2. Perform insertion sort on elements separated by the gap.

  3. Reduce the gap until it becomes 0, at which point the array is fully sorted.

- **Time Complexity**:

  - Best case: \(O(n \log n) \) (depends on gap sequence used).

  - Worst case: \(O(n^2) \).

- **Space Complexity**: \(O(1) \), as sorting is performed in place.

---

3. `display_top_five` Function

- **Purpose**: Displays the top 5 scores in descending order.

- **Process**:

  1. Use the `sorted()` function to sort the array in descending order.

  2. Slice the first 5 elements (`[:5]`) to extract the top scores.

  3. Print each score.

- **Time Complexity**:

  - Sorting: \(O(n \log n) \).

  - Slicing: \(O(1) \) (constant time for a fixed slice).

- **Space Complexity**:

  - Sorting creates a new sorted array: \(O(n) \).

---

4. Main Program

- **Steps**:

  1. Prompt the user to input the total number of scores and the scores themselves.

  2. Store the scores in a list (`percentage`).

  3. **Insertion Sort**:

     - Sort the scores using `insertion_sort`.

     - Display the sorted list and the top 5 scores using `display_top_five`.

  4. **Shell Sort**:

     - Sort the scores using `shell_sort`.

     - Display the sorted list and the top 5 scores using `display_top_five`.

Execution Flow

1. **Input**:

  - Total number of scores: \(m \).

  - Individual scores: e.g., `85.5, 92.3, 76.4, 88.9, 54.2`.

2. **Insertion Sort**:

  - Sort the list: e.g., `[54.2, 76.4, 85.5, 88.9, 92.3]`.

  - Display the top 5: `[92.3, 88.9, 85.5, 76.4, 54.2]`.

3. **Shell Sort**:

  - Sort the list (same result as Insertion Sort if implemented correctly).

  - Display the top 5: `[92.3, 88.9, 85.5, 76.4, 54.2]`.

Output

For input percentages `[85.5, 92.3, 76.4, 88.9, 54.2]`, the output will be:

```

Original percentages: [85.5, 92.3, 76.4, 88.9, 54.2]

Sorted percentages using insertion sort: [54.2, 76.4, 85.5, 88.9, 92.3]

Top 5 scores:

92.3

88.9

85.5

76.4

54.2

Sorted percentages using shell sort: [54.2, 76.4, 85.5, 88.9, 92.3]

Top 5 scores:

92.3

88.9
```

85.5

76.4

54.2

```

---

### **Overall Complexity**

- **Time Complexity**: \( O(n^2) \) (dominant due to sorting in both algorithms).

- **Space Complexity**: \( O(n) \) (temporary storage for sorted results).

6)### **Explanation of the Code**

This Python program sorts a list of percentages in ascending order using the **Quicksort algorithm**, and displays the top five percentages in both ascending and descending order.

---

### **1. `quicksort` Function**

- **Purpose**: Sorts an array using the **Quicksort algorithm**, a divide-and-conquer approach.

- **Process**:

 1. **Base Case**: If the array has one or zero elements, it is already sorted; return it.

 2. **Divide**:

   - Choose the first element (`pivot`) as the dividing element.

   - Partition the array into:

     - `left`: All elements smaller than the pivot.

     - `middle`: All elements equal to the pivot.

     - `right`: All elements greater than or equal to the pivot.
```

3. **Conquer**:

   - Recursively apply Quicksort to the `left` and `right` partitions.

4. **Combine**:

   - Concatenate the sorted `left`, `middle`, and `right` partitions.

- **Time Complexity**:

  - Best/Average Case: $O(n \log n)$ (when partitions are balanced).

  - Worst Case: $O(n^2)$ (when partitions are highly unbalanced, e.g., sorted input).

- **Space Complexity**: $O(n)$, as it uses additional memory to create subarrays for `left`, `middle`, and `right`.

---

2. Input Handling

- **Steps**:

  1. Prompt the user for the number of percentages.

  2. Collect the percentages into the list `percentage`.

  - Example Input:

    ```
    85, 92, 76, 88, 54
    ```

---

3. Sorting and Top 5 Percentages

- **Sorting**:

  - Call `quicksort(percentage)` to sort the percentages in ascending order.

- **Extract Top 5**:

  - Use slicing `[-5:]` to extract the last 5 elements (highest scores) from the sorted array.

- **Display**:

1. Print the top 5 scores in ascending order.

2. Reverse the order of the top 5 scores using `reverse()` to display them in descending order.

Execution Flow

1. **Input**:

```
ENTER THE number of PERCENTAGE: 5

ENTER THE PERCENTAGE: 85

ENTER THE PERCENTAGE: 92

ENTER THE PERCENTAGE: 76

ENTER THE PERCENTAGE: 88

ENTER THE PERCENTAGE: 54
```

2. **Quicksort**:

- Sorted List: `[54, 76, 85, 88, 92]`.

3. **Top 5 Percentages**:

- Ascending Order: `[54, 76, 85, 88, 92]`.

- Descending Order: `[92, 88, 85, 76, 54]`.

Output

For the input percentages `85, 92, 76, 88, 54`, the output will be:

```
PERCENTAGE IN ASCENDING ORDER:

[54, 76, 85, 88, 92]
```

TOP FIVE PERCENTAGE IS:

[54, 76, 85, 88, 92]

PERCENTAGE IN DESCENDING ORDER:

[92, 88, 85, 76, 54]

` ` `

Overall Complexity

- **Time Complexity**:

  - Sorting (Quicksort): \(O(n \log n) \) (on average).

  - Extracting and reversing the top 5: \(O(5) = O(1) \).

  - Total: \(O(n \log n) \).

- **Space Complexity**:

  - Quicksort creates new lists for partitions: \(O(n) \).

  - Total: \(O(n) \).

7)### Explanation of the Code

The given code implements a **linked list** structure to manage a club committee with the roles of **President**, **Secretary**, and members. The operations include creating the list, displaying it, inserting a member, deleting a member, and counting the total members.

Code Components

1. Structure `node`

- Represents a node in the linked list.
- Contains:
  - `prn`: Unique PRN of a member.
  - `name`: Name of the member.
  - `next`: Pointer to the next node.

---

2. Class `pinnacle`

- Manages the linked list operations.

a. `create()`

- **Purpose**: Create the initial linked list.
- **Steps**:
  1. Accept `PRN` and `name` for a new node.
  2. If the list is empty:
     - Set the new node as `head` and `tail`.
  3. If the list already exists:
     - Append the new node at the end of the list.

---

b. `member()`

- **Purpose**: Insert a new member after a specific `PRN`.
- **Steps**:
  1. Accept `PRN` and `name` for the new member.
  2. Accept the `PRN` after which the new member is to be inserted.
  3. Traverse the list to find the node with the given `PRN`.
  4. Insert the new node after the found node.

---

c. `display()`

- **Purpose**: Display all members in the linked list.

- **Steps**:

 1. Traverse the list from `head` to `tail`.

 2. Print `PRN` and `name` of each node.

---

d. `Delete_pre()`

- **Purpose**: Delete the **President** (first node).

- **Steps**:

 1. If the list is not empty:

   - Set `head` to the second node.

   - Delete the old `head`.

 2. If the list is empty:

   - Print an error message.

---

e. `Delete_sec()`

- **Purpose**: Delete the **Secretary** (last node).

- **Steps**:

 1. Traverse the list to find the second last node.

 2. Set the `next` pointer of the second last node to `NULL`.

 3. Update `tail` and delete the last node.

---

f. `Delete_mem()`

- **Purpose**: Delete a member after a specific `PRN`.

- **Steps**:

  1. Traverse the list to find the node with the given `PRN`.

  2. If a next node exists:

    - Delete it and update the `next` pointer.

  3. If not found or no member exists after the given `PRN`:

    - Print an error message.

---

g. `total()`

- **Purpose**: Count the total number of members.

- **Steps**:

  1. Traverse the list from `head` to `tail`.

  2. Increment a counter for each node.

---

3. Main Function

- Menu-driven program for user interaction.

- **Options**:

  - Create the linked list.

  - Display the list.

  - Insert a member.

  - Delete a member (President, Secretary, or after a specific PRN).

  - Count the total members.

- Loops until the user decides to exit.

Corrections and Improvements

1. **Logical Errors**:

  - Multiple redundant sections in code (e.g., repeated variable declarations like `char name1[20]`).

  - Some `case` labels have missing logic.

2. **Indentation**:

  - Improper indentation makes the code difficult to read.

3. **Error Messages**:

  - Improve user feedback when operations fail (e.g., deletion in an empty list).

Complexity Analysis

Time Complexity

1. **Create**: \(O(1)\) for appending a node.

2. **Insert Member**: \(O(n)\) to traverse the list.

3. **Display**: \(O(n)\) to traverse the list.

4. **Delete President**: \(O(1)\) for removing the first node.

5. **Delete Secretary**: \(O(n)\) to find the second last node.

6. **Delete Member**: \(O(n)\) to find the node with the given `PRN`.

7. **Count Total Members**: \(O(n)\) to traverse the list.

Space Complexity

- **Linked List**: \(O(n)\), where \(n\) is the number of nodes.

- **Auxiliary Space**: \(O(1)\), as no additional data structures are used.

Sample Output

Input:

```

1. CREATE

2. DISPLAY

3. INSERT MEMBER

4. TOTAL MEMBERS

5. DELETE

```

Operations:

1. Add President: PRN=1, Name=”John”.

2. Add Secretary: PRN=2, Name=”Doe”.

3. Insert Member: PRN=3, Name=”Alice”, after PRN=1.

4. Display List: `1 → John, 3 → Alice, 2 → Doe`.

5. Delete Secretary: `2 → Doe`.

6. Total Members: 2.

Output:

```

MENU

1. CREATE

2. DISPLAY

3. INSERT MEMBER

4. TOTAL MEMBERS

5. DELETE

ENTER YOUR CHOICE: 1

Enter the PRN: 1

Enter the Name: John
```

...

Total Members: 2

```
```