Program 4/5 Specification

Name: Evelyn Nguyen

Course: CS302 - Programs #4 and #5

Date: 12/6/2025

Project: Social Media Analytics Tool (Instagram, Facebook, TikTok)

1. Introduction

The purpose of Programs #4 and #5 is to design a complete social media analytics system using Python classes, inheritance, and data structures. This projects does not connect to real social media platforms. Instead, users provide data through a menu interface, and the program stores and analyzespatterns across three platforms: Instagram, Facebook, and TikTok.

Program #4 focuses on building the core class hierarchy with object-oriented design, inheritance, proper use of super(), protected attributes, operator overloading, exception handling, and NumPy arrays. Program #4 is tested entirely through PyTest with no user interface.

Program #5 is to implement a Red-Black Tree data structure (recursively) and a complete menu application for managing social media accounts.

This assignment emphasizes:

- Object-oriented design with single inheritance

- Red-Black Tree implementation using recursion exclusively

- Operator overloading in Python

- Comprehensive exception handling

- Test-driven development with PyTest

- Use of NumPy arrays and Python lists

2. Program Overview

Core Hierarchy (Program #4)

The class hierarchy includes:

Base class: SocialMediaAccount (abstract base class)

Derived classes: InsAcc (Instagram), FbAcc (Facebook), TiktokAcc (TikTok)

The base class stores information common to all social media platforms (username, followers, daily usage time), while derived classes add platform-specific data and behaviors.

All classes follow these requirements:

- All instance variables are protected (use underscore prefix)

- Constructors must call the parent constructor using super()

- No duplicated logic - derived classes only implement unique behavior

- Strict validation of all inputs with appropriate exceptions

- All behavior must be testable with PyTest

- NumPy arrays used for engagement statistics

Data Structure (Program #5)

- Red-Black Tree for storing and managing all social media accounts

- All tree operations implemented recursively

- Accounts organized by username (case-insensitive)

- Supports insert, retrieve, display all, count by type, and statistics operations

Application (Program #5)

- Menuinterface with 8 options

- Add accounts for Instagram, Facebook, or TikTok

- View individual accounts or all accounts sorted by username

- Record usage session time

- Display analytics statistics (counts by platform, total followers)

- Full exception handling with user-friendly error messages

3. Class Descriptions

Base Class: SocialMediaAccount (Abstract)

Protected Variables:

_username: str - unique identifier for account

_followers: int - number of followers (must be >= 0)

_avg_daily_minutes: float - average daily usage time (must be >= 0)

Constructor:

__init__(self, username: str, followers: int, avg_daily_minutes: float)

Exception Handling:

- username must be non-empty string, if not raise ValueError("username cannot be empty")

- followers must be int >= 0, if not raise ValueError("followers cannot be negative")

- avg_daily_minutes must be numeric >= 0, if not raise ValueError("avg_daily_minutes cannot be negative")

- Any incorrect type raise TypeError with appropriate message

- Whitespace-only username raise ValueError("username cannot be empty")

Abstract Methods (must be implemented by derived classes):

1. display() -> str

   Returns formatted string summary of account information

Raises NotImplementedError if called on base class

2. get_engagement_stats() -> numpy.ndarray

   Returns NumPy array of 3 platform-specific metrics

   Format: np.array([metric1, metric2, metric3], dtype=float)

   Example: [total_posts, average_likes, total_views]

   Raises NotImplementedError if called on base class

Concrete Methods (shared by all derived classes):

1. record_session(minutes: float) -> None

   Updates _avg_daily_minutes by adding usage session time

   Parameters: minutes (int or float) must be >= 0

   Exception handling:

   - minutes must be numeric, if not raise TypeError("minutes must be a number")

   - minutes >= 0, if not raise ValueError("minutes cannot be negative")

2. engagement_rate() -> float

   Calculates engagement rate as followers per minute

   Returns followers / avg_daily_minutes

   If _avg_daily_minutes == 0, return 0.0 to avoid division by zero

3. update_followers(new_followers: int) -> None

   Updates follower count to new value

   Parameters: new_followers (int) must be >= 0

   Exception handling:

- new_followers must be int, if not raise TypeError("new_followers must be an int")

- new_followers >= 0, if not raise ValueError("new_followers cannot be negative")

Operator Overloading:

1. __lt__(self, other: SocialMediaAccount) -> bool

Compares accounts by username (case-insensitive alphabetical order)

Returns True if self._username < other._username (both lowercase)

Wrong type raise TypeError("Can only compare with SocialMediaAccount objects")

Example: InsAcc("alice", ...) < FbAcc("bob", ...) returns True

2. __eq__(self, other: object) -> bool

Compares accounts by username (case-insensitive)

Returns True if usernames match (ignoring case)

Returns False (not TypeError) if other is not SocialMediaAccount

Example: InsAcc("Alice", ...) == FbAcc("alice", ...) returns True

3. __add__(self, minutes: float) -> SocialMediaAccount

Adds session minutes to account

Usage: account + 15.5

Internally calls record_session(minutes)

Returns self for method chaining

Example: acc + 10.0 adds 10 minutes to acc._avg_daily_minutes

Getter Methods:

- get_username() -> str: returns _username

- get_followers() -> int: returns _followers

- get_avg_daily_minutes() -> float: returns _avg_daily_minutes

Derived Classes

Instagram Class: InsAcc

Instagram focuses on photo posts with likes and short video reels with views.

Additional Protected Variables:

_post_likes: list[int] - stores like count for each post (initialized as empty list)

_reel_views: list[int] - stores view count for each reel (initialized as empty list)

Constructor:

__init__(self, username: str, followers: int, avg_daily_minutes: float)

Calls super().__init__(username, followers, avg_daily_minutes)

Initializes _post_likes and _reel_views as empty lists

Platform-Specific Methods (3 required):

1. add_post(likes: int) -> None

   Adds post's like count to _post_likes list

   Parameters: likes (int) must be >= 0

   Exception handling:

   - likes must be int, if not raise TypeError("likes must be an int")

   - likes >= 0, if not raise ValueError("likes cannot be negative")

Example: ig.add_post(150) adds 150 to _post_likes

2. add_reel(views: int) -> None

Adds reel's view count to _reel_views list

Parameters: views (int) must be >= 0

Exception handling:

- views must be int, if not raise TypeError("views must be an int")

- views >= 0, if not raise ValueError("views cannot be negative")

Example: ig.add_reel(5000) adds 5000 to _reel_views

Override Methods:

3. get_engagement_stats() -> numpy.ndarray

Returns np.array([total_posts, avg_post_likes, total_reel_views], dtype=float)

- total_posts = len(_post_likes)

- avg_post_likes = mean of _post_likes (0.0 if empty list)

- total_reel_views = sum of _reel_views

Example: [3.0, 125.5, 8000.0] means 3 posts, avg 125.5 likes, 8000 total reel views

4. display() -> str

Returns formatted string:

"Instagram Acount: {username}

Followers: {followers}

Avg Daily Minutes: {minutes}

Total Posts: {posts}, Avg Likes/Post: {avg:.2f}, Total Reel Views: {views}"

Facebook Class: FbAcc

Facebook emphasizes posts with reactions (likes, love, wow, etc.) and group memberships.

Additional Protected Variables:

_post_reactions: list[int] - stores reaction count for each post (initialized as empty list)

_groups: list[str] - stores names of joined groups (initialized as empty list)

Constructor:

__init__(self, username: str, followers: int, avg_daily_minutes: float)

Calls super().__init__(username, followers, avg_daily_minutes)

Initializes _post_reactions and _groups as empty lists

Platform-Specific Methods (3 required):

1. add_post(reactions: int) -> None

   Adds post's reaction count to _post_reactions list

   Parameters: reactions (int) must be >= 0

   Exception handling:

   - reactions must be int, if not raise TypeError("reactions must be an int")

   - reactions >= 0, if not raise ValueError("reactions cannot be negative")

   Example: fb.add_post(45) adds 45 to _post_reactions

2. join_group(group_name: str) -> None

Adds group name to _groups list

Parameters: group_name (str) must be non-empty

Exception handling:

- group_name must be string, if not raise TypeError("group must be a string")

- group_name cannot be empty or whitespace, if empty raise ValueError("group cannot be empty")

Example: fb.join_group("CS302 Study Group")

Override Methods:

3. get_engagement_stats() -> numpy.ndarray

Returns np.array([num_posts, avg_reactions, num_groups], dtype=float)

- num_posts = len(_post_reactions)

- avg_reactions = mean of _post_reactions (0.0 if empty list)

- num_groups = len(_groups)

Example: [5.0, 32.4, 3.0] means 5 posts, avg 32.4 reactions, 3 groups joined

4. display() -> str

Returns formatted string:

"Facebook Account: {username}

Followers: {followers}

Avg Daily Minutes: {minutes}

Total Posts: {posts}, Avg Reactions/Post: {avg:.2f}, Groups: {groups}"

TikTok Class: TiktokAcc

TikTok focuses on short-form video content with likes and shares.

Additional Protected Variables:

_video_likes: list[int] - stores like count for each video (initialized as empty list)

_shares: list[int] - stores share count for each video (initialized as empty list)

Constructor:

__init__(self, username: str, followers: int, avg_daily_minutes: float)

Calls super().__init__(username, followers, avg_daily_minutes)

Initializes _video_likes and _shares as empty lists

Platform-Specific Methods (3 required):

1. add_video(likes: int) -> None

   Adds video's like count to _video_likes list

   Parameters: likes (int) must be >= 0

   Exception handling:

   - likes must be int, if not raise TypeError("likes must be an int")

   - likes >= 0, if not raise ValueError("likes cannot be negative")

   Example: tk.add_video(2000) adds 2000 to _video_likes

2. add_share(shares: int) -> None

   Adds share count to _shares list

   Parameters: shares (int) must be >= 0

   Exception handling:

- shares must be int, if not raise TypeError("shares must be an int")

- shares >= 0, if not raise ValueError("shares cannot be negative")

Example: tk.add_share(150) adds 150 to _shares

Override Methods:

3. get_engagement_stats() -> numpy.ndarray

Returns np.array([total_videos, avg_likes, total_shares], dtype=float)

- total_videos = len(_video_likes)

- avg_likes = mean of _video_likes (0.0 if empty list)

- total_shares = sum of _shares

Example: [10.0, 1500.5, 450.0] means 10 videos, avg 1500.5 likes, 450 total shares

4. display() -> str

Returns formatted string:

"TikTok Account: {username}

Followers: {followers}

Avg Daily Minutes: {minutes}

Total Videos: {videos}, Avg Likes/Video: {avg:.2f}, Total Shares: {shares}"

4. Data Structure: Red-Black Tree (Program #5)

Node Class

Protected Variables:

_account: SocialMediaAccount - the stored account object

_color: Color - enum value (Color.RED or Color.BLACK) _left: Optional[Node] - left child (None if no left child)

_right: Optional[Node] - right child (None if no right child)

_parent: Optional[Node] - parent node (None if root)

Constructor:

__init__(self, account: SocialMediaAccount)

Exception handling:

- account must be SocialMediaAccount, if not raise TypeError("account must be a SocialMediaAccount")

Initializes _color to Color.RED (new nodes always start red)

Initializes all pointers (_left, _right, _parent) to None

Methods:

Getters:

- get_account() -> SocialMediaAccount: returns _account
- get_color() -> Color: returns _color
- get_left() -> Optional[Node]: returns _left
- get_right() -> Optional[Node]: returns _right
- get_parent() -> Optional[Node]: returns _parent

Setters:

- set_color(color: Color) -> None: sets _color

  Exception: color must be Color enum, if not raise TypeError("Color must be a Color enum")
- set_left(node: Optional[Node]) -> None: sets _left
- set_right(node: Optional[Node]) -> None: sets _right

- set_parent(node: Optional[Node]) -> None: sets _parent

RBTree Class

Protected Variables:

_root: Optional[Node] - root of tree (None if empty)

_size: int - number of accounts in tree (initialized to 0)

Constructor:

__init__(self)

Initializes _root to None and _size to 0

Public Methods:

1. insert(account: SocialMediaAccount) -> None

   Inserts account into tree maintaining Red-Black properties

   Process:

   - Creates new Node with account

   - Calls recursive helper _insert_recursive() to insert as in BST

   - Calls _fix_insert() to restore Red-Black properties

   - Ensures root is always BLACK

   - Increments _size

   Exception handling:

   - account must be SocialMediaAccount, if not raise TypeError("Can only insert SocialMediaAccount obj")

   - duplicate username (case-insensitive) raise ValueError("Account already exist")

2. retrieve(username: str) -> Optional[SocialMediaAccount]

   Searches tree for account by username (case-insensitive)

Returns account if found, None if not found

Calls recursive helper _retrieve_recursive()

Exception handling:

- username must be string, if not raise TypeError("username must be a string")

- username cannot be empty, if empty raise ValueError("username cannot be empty")

3. display_all() -> List[str]

Returns list of display strings for all accounts

Uses in-order traversal (recursive) to get accounts in sorted order by username

Returns empty list if tree is empty

Calls recursive helper _display_recursive()

4. count_by_type(account_type: str) -> int

Counts how many accounts are of specified type

Parameters: account_type is "Instagram", "Facebook", or "TikTok"

Returns count (0 if none found or tree empty)

Calls recursive helper _count_by_type_recursive()

Implementation: checks if account_type appears in account.display() string

5. get_total_followers() -> int

Calculates sum of followers across all accounts

Returns 0 if tree is empty

Calls recursive helper _get_total_followers_recursive()

6. get_size() -> int

   Returns number of accounts in tree (_size)

7. get_root() -> Optional[Node]

   Returns root node (for testing purposes)

   Returns None if tree is empty

Operator Overloading:

1. __len__() -> int

   Returns number of accounts in tree

   Usage: len(tree)

   Returns _size

2. __contains__(username: str) -> bool

   Checks if username exists in tree (case-insensitive)

   Usage: "alice" in tree

   Returns True if found, False otherwise

   Returns False (not exception) for invalid types

   Internally calls retrieve(username) and checks if result is not None

3. __iadd__(account: SocialMediaAccount) -> RBTree

   Adds account to tree using += operator

   Usage: tree += account

   Calls insert(account)

   Returns self for method chaining

Private Recursive Helper Methods:

_insert_recursive(current: Optional[Node], new_node: Node, parent: Optional[Node]) -> Node

- Base case: if current is None, set new_node's parent and return new_node

- Recursive case: compare usernames (case-insensitive) and go left or right

- Raise ValueError if duplicate username found

- Returns the node that should be at this position

_fix_insert(node: Node) -> None

- Fixes Red-Black violations after insertion

- Handles 3 cases for left and right sides (6 total cases)

- Uses _rotate_left() and _rotate_right() for restructuring

- Recolors nodes as needed

_rotate_left(node: Node) -> None

- Perform left rotation around node

- Updates all affected pointers (parent, children)

- Updates root if necessary

_rotate_right(node: Node) -> None

- Performs right rotation around node

- Updates all affected pointers (parent, children)

- Updates root if necessary

_retrieve_recursive(current: Optional[Node], username: str) -> Optional[SocialMediaAccount]

- Base case: if current is None, return None

- Compare usernames (case-insensitive)

- Recursively search left or right subtree

- Return account if found

_display_recursive(current: Optional[Node], result: List[str]) -> None

- Base case: if current is None, return

- In-order traversal: left, current, right

- Appends current.get_account().display() to result list

_count_by_type_recursive(current: Optional[Node], account_type: str) -> int

- Base case: if current is None, return 0

- Check if account_type appears in current account's display string

- Recursively count in left and right subtrees

- Return sum of counts

_get_total_followers_recursive(current: Optional[Node]) -> int

- Base case: if current is None, return 0

- Get followers from current account

- Recursively sum followers from left and right subtrees

- Return total sum

<u>5. Application Interface (Program #5)</u>

1. Add Instagram Account
2. Add Facebook Account
3. Add TikTok Account
4. View Specific Account
5. View All Accounts (Sorted)
6. Record Session Time
7. Display Analytics Statistics
8. Exit

Menu Options Detailed Description:

Option 1: Add Instagram Account

Prompts user for:

Process:

- Creates InsAcc object with provided data

- Add all posts using add_post() method

- Adds all reels using add_reel() method

- Uses += operator to add account to tree: tree += account

- Displays success message: "SUCCESS: Instagram account '{username}' added to the system!"

Exception handling:

- Catches ValueError and displays "VALUE ERROR: {error message}"

- Catches TypeError and displays "TYPE ERROR: {error message}"

- Catches general Exception and displays "ERROR: {error message}"

- If error occurs, account is not added, user informed to try again

Option 2: Add Facebook Account

same as Option 1

Option 3: Add TikTok Account

same as Option 1

Option 4: View specific Account

Process:

- Uses 'in' operator to check if username exists: if username in tree
- If found:
  * Retrieves account using tree.retrieve(username)
  * Displays account.display() output
  * Displays engagement rate: "Engagement Rate: {rate:.4f} followers/minute"
- If not found:
  * Displays "WARNING: Account '{username}' not found in the system."
  * Displays "Please check the username and try again."

Exception handling:

- Catches ValueError, TypeError, and general Exception
- Displays appropriate error messages

Option 5: View All Accounts (Sorted)

Process:

- Uses len() operator to check if tree is empty: if len(tree) == 0
- If empty:
  * Displays "WARNING: No accounts in the system."

  * Displays "Please add accounts using options 1-3 from the main menu."

- If not empty:

  * Displays header with total count: "ALL ACCOUNTS (Total: {len(tree)})"

  * Displays "(Sorted by Username)"

  * Calls tree.display_all() to get list of display strings

  * Displays each account with number: "[Account #{i}]" followed by display string

  * Separates accounts with line of dashes

Exception handling:

- Catches general Exception and displays error message

Option 6: Record Session Time

Prompts user for:

1. Username (non-empty string)

2. Session minutes to add (non-negative float)

Process:

- Retrieves account using tree.retrieve(username)

- If account not found:

  * Displays "WARNING: Account '{username}' not found in the system."

  * Returns without recording

- If found:

  * Uses + operator to add minutes: account + minutes

  * Displays success message: "SUCCESS: Added {minutes:.2f} minutes to '{username}'s session time."

Exception handling:

- Catches ValueError, TypeError, and general Exception
- Displays appropriate error messages

Option 7: Display Analytics Statistics

Process:

- Uses len() operator to check if tree empty
- If empty: displays warning message
- If not empty:
  * Displays "ANALYTICS STATISTICS" header
  * Shows total accounts: "Total Accounts in System: {len(tree)}"
  * Calls tree.count_by_type() for each platform:
    - Instagram count: tree.count_by_type("Instagram")
    - Facebook count: tree.count_by_type("Facebook")
    - TikTok count: tree.count_by_type("TikTok")
  * Calls tree.get_total_followers() for total
  * Calculates and displays average: total / len(tree)
  * Formats follower numbers with commas: "{total:,}"

Exception handling:

- Catches general Exception and displays error message

Option 8: Exit

No user input required

Application Helper Methods for Input Validation:

_get_non_empty_input(prompt: str) -> str

- Displays prompt and gets user input

- Strips whitespace

- If empty, displays "Input cannot be empty. Please try again." and re-prompts

- Returns valid non-empty string

- Raises ValueError if EOFError (input stream closed)

_get_positive_int(prompt: str, allow_zero: bool = False) -> int

- Displays prompt and gets user input

- Validates input is integer

- If allow_zero is False, value must be > 0

- If allow_zero is True, value must be >= 0

- Error messages:

  * Empty input: "Input cannot be empty. Please try again."

  * Negative: "Value cannot be negative. Please try again."

  * Zero when not allowed: "Value must be greater than 0. Please try again."

  * Invalid format: "Invalid integer format. Please enter a whole number."

- Returns valid integer

- Raises ValueError if EOFError

_get_positive_float(prompt: str) -> float

- Displays prompt and gets user input

- Validates input is numeric

- Value must be >= 0

- Error messages:

  * Empty input: "Input cannot be empty. Please try again."

  * Negative: "Value cannot be negative. Please try again."

  * Invalid format: "Invalid number format. Please enter a valid number."

- Returns valid float

- Raises ValueError if EOFError

6. Input Rules

All input must be validated before use:

String Input Rules:

- Must be non-empty after stripping whitespace

- Empty or whitespace-only strings raise ValueError

- Type must be str, otherwise raise TypeError

Integer Input Rules:

- Must be int type (not float), otherwise raise TypeError

- Must be >= 0, negative values raise ValueError

- Some methods allow 0 (like number of posts to add)- Some methods require > 0 (like followers)

Float Input Rules:

- Can be int or float type (converted to float)

- Must be >= 0, negative values raise ValueError

- Used for avg_daily_minutes and session time

Username Rules:

- Non-empty string

- Case insensitive for comparison and storage

- Whitespace stripped before validation

Validation Occurs:

- At constructor level (all classes)
- At method level (add_post, join_group, etc.)
- At application input level (helper methods)
- No invalid data should ever reach data structure

7. Output Rules

Display Format:

All display() methods return formatted multi-line strings

Format is consistent across platforms:

- Line 1: "{Platform} Account: {username}"
- Line 2: "Followers: {followers}"
- Line 3: "Avg Daily Minutes: {minutes}"
- Line 4: Platform-specific metrics

Numeric Output:

- Floating point numbers displayed with 2 decimal places: {value:.2f}
- Large numbers formatted with commas: {value:,}
- Engagement rate shown with 4 decimal places: {rate:.4f}

NumPy Arrays:

- Always dtype=float for consistency
- Always 3 elements representing platform metrics
- Elements can be accessed by index: stats[0], stats[1], stats[2]

Menu Output:

- Clear section headers with lines of = or -

- Consistent spacing and alignment

- Success messages clearly marked

- Error messages clearly marked with WARNING or ERROR prefix

No Direct Printing:

- All methods return values

- Application layer handles printing to console

- Separation of logic and presentation

8. Exception Handling Summary

ValueError Raised When:

- Empty or whitespace-only strings provided

- Negative numeric values provided

- Duplicate username inserted into tree

- Username not found during retrieval (for some operations)

TypError Raised When:

- Wrong type provided for parameter (e.g., string instead of int)

- Comparing SocialMediaAccount with non-SocialMediaAccount using < operator

- Non-Color value assigned to node color

- Non-SocialMediaAccount passed to Node constructor

Handled Gracefully (No Exception Raised):

- __eq__ returns False instead of TypeError for non-SocialMediaAccount

- __contains__ returns False instead of exception for invalid types

- Division by zero prevented by checking before division

Application Level Exception Handling:

- All menu options wrapped in try-except blocks
- Specific exception types caught and handled appropriately
- General Exceptioncaught as fallback
- User always informed of what went wrong
- Application continues running after errors

No Silent Failures:

- All errors reported with clear messages
- No errors ignored or hidden
- Exceptions logged/displayed before recovery
- User guided on how to correct errors

9. Testing Strategy

Glass Box Testing (Program #5):

Based on implementation details and code structure

Tests include:

- All code paths executed
- Tree structure verification after operations
- Rotation case in Red-Black Tree
- Recursive base cases and recursive calls
- Internal helper methods
- Parent pointer correctness
- Color property maintenance

Glass Box Test Categories:

1. Tree structure tests - verify node relationships

2. Rotation tests - left and right rotations
3. Color tests - verify red-black properties maintained
4. Recursion tests - verify correct recursion depth
5. Edge cases - empty tree, single node, complex structures

Integration Testing:

Complete workflows tested:

- Add multiple accounts of different types
- Retrieve and modify accounts
- Display all account
- Calculate statistics
- Use all operator overloads together