Advanced Data Structures

Gator Library Management System

Abhinav Reddy Pannala

UFID:- 7031-4901

pannala.abhinav@ufl.edu

Problem Description:

Gator Library, a network of libraries, needs a specialized software tool that will aid in the management of their extensive book collection, services offered to library members, and the various aspects of book lending. The report that follows delves into the specifics of a Python-based software program, tailor-made for the Gator Library System's needs. This program is centered around facilitating the numerous tasks associated with managing books in a library setting. It employs sophisticated data structures to ensure that these tasks are executed efficiently and effectively. This in-depth examination will cover how the program handles these book-related operations, highlighting its functionality in the context of a library's operational requirements.

Overview of the Program Architecture:

1) Class **Minimum_Heap**:-

    - Implements a minimum Heap data structure
    - Methods:
        1. __init__(): Initializes an empty heap.
        2. insert(self, k): Inserts an element 'k' into the Heap.
        3. extractMinimum(self): This method removes and returns the smallest element from the heap, which is always at the root in a min-heap.
        4. Heapify(self, i): This a helper method used to maintain the min-heap property from a given index downwards.
        5. BubbleUp(self, i): Another helper method used to adjust the heap upwards.

2) Class **Book**:-

    - Presents a book in the library.
    - Methods:

i. __inti__(self, book_id, book_name, author_name, availability_status, borrowed_by, reservation_heap): This constructor initializes a new instance of a book with various parameters.
ii. __repr__(self): This method provides a string representation of a Book object. It's used for debugging and logging purposes to easily understasnd the state of a Book instance.

3) Class **Red_Black_Tree**:-

- Implements a red-black tree data structure.
- Methods:
  i. __init__(key, color): Initializes the red-black tree.
  ii. minimum_value(): Finds the node with the minimum value (smallest book ID) starting from a given node.
  iii. transplant_value(): Replaces one subtree as a child of its parent with another subtree.
  iv. delete(): Removes a node from the tree.
  v. delete_helper(): It fixes the tree after deletion to maintain its Red-Black properties.
  vi. flip_color(): Changes the color of the node from red to black or vice-versa.
  vii. insert(): Inserts a new node into the tree.
  viii. _insert_helper(): Helper method for insert() and fixes the tree after insertion to maintain Red-Black properties.
  ix. rotate_left() and rotate_right(): Performs a left and right rotation around a given node.
  x. search(): searches for a book by its ID in the tree.
  xi. Print_books_range(): Prints the details of books within a specified range of book ID's.
  xii. _print_books_range(): A helper recursive method that assists the above method respectively in traversing the tree and collecting book details.

4) Class **Gatorlibraryufl**:-

- Manages the library operations
- Methods:
  i. __init__(): Intializes the GatorLibraryUFL with a Red-Black Tree and a counter for color flips during tree operations.
  ii. read_commands_from_file(): reads commands from a input file returns them as a list of strings.
  iii. write_output_to_file(): writes given output lines to a output file.
  iv. Insert_book(): inserts a new book to the library's Red-Black tree and updates the color flip count.
  v. Print_book(): prints details of a book with the given book ID.

vi. Print_book(): Handles the borrowing of a book by a patron. It updates the book's status and reservation queue as necessary.
vii. Find_closet_book() and _find_closet_book(): First one finds the book(s) closet to a given target book ID and the other is the helper function traversing the Red-Black Tree to find the closest book(s).
viii. Delete_book(): deletes a book from the library, handling any active reservations for that book.
ix. Operation(): Parses and executes a given command string, facilitation various library operations like inserting books, borrowing books, etc.
x. Read_commands_from_file() and write_output_to_file(): The first one reads commands from a input file and the other method used to print the output onto a file.

6) Function **main(file_name)**:- This structure is the entry point for running the GatorLibraryUFL library management system. It reads commands from input files specified as command line arguments, executes each command using the GatorLibraryUFL class, and writes the results to an output file. This script checks if the command line count is correct and takes the output file name from the file name.

Every class and function in this project have been meticulously crafted to fulfill distinct roles within the framework of the library management system. This design effectively manages various aspects of library operations, including the management of books, the handling of patron interactions, and the execution of various library-specific tasks. The architecture of the script is grounded in the principles of object-oriented programming. This approach involves the encapsulation of data and associated behaviors within classes, a strategy that brings several benefits. For one, it enhances the clarity and organization of the code, making it easier to understand, maintain, and expand. Each class in the project is like a self-contained module with its own responsibilities. For instance, the Book class is solely concerned with attributes and operations related to individual books, such as managing reservations and keeping track of availability status. In contrast, the Gator_Library class oversees the broader library operations, integrating the functionalities of books, patrons, and the library's catalog system.

Furthermore, this structured approach fosters a clear separation of concerns. This principle ensures that each part of the script handles a specific, independent function. Such separation makes the code more modular and flexible, allowing for easier updates and modifications. For example, changes in the book reservation system would primarily involve modifications in the Book class, often without the need to alter other parts of the system like the patron management or the Red-Black Tree implementation.

In essence, the code in the project represents a well-organized, efficient system where each component plays a distinct role, aligned with the overarching goal of streamlined library management. The use of object-oriented programming not only aids in creating a robust and scalable system but also makes the codebase more intuitive and adaptable to future enhancements or changes in library management requirements.

Problem raised to me: *I would like to inform you that the entire code for this project is functioning correctly and efficiently, except for one specific feature: the color flip count. Despite numerous attempts and the implementation of various strategies to address this issue, I have been unable to resolve it. I sincerely apologize for this inconvenience. Please understand that this is the only aspect of the project that isn't working as intended. All other operations and functionalities within the project are performing as expected, delivering accurate and reliable results. I regret any inconvenience this may cause and appreciate your understanding regarding this singular limitation in an otherwise fully functional system. Rest assured, every effort has been made to rectify this issue, and I remain committed to finding a solution. Thank you for your patience and support.*