ADVANCED DATA STRUCTURE

COP5536, FALL 2023

PROJECT REPORT

NAME : PRATIKSHA DEODHAR

UFID : 7069-2093

EMAIL : pdeodhar@ufl.edu

UNDER THE GUIDANCE OF

Dr. SARTAJ SAHNI

PROJECT OBJECTIVE:

The objective for the GatorLibrary management system is to develop a software system that efficiently manages the books, patrons, and borrowing operations of a fictional library named GatorLibrary. The system is designed to employ a Red-Black tree data structure for the effective management of books.
To handle reservation processes when books are not immediately available for loan, the system adopts a priority-queue strategy utilizing Binary Min-heaps. In this setup, each book is allocated an individual min-heap to effectively track the reservations made by patrons

CONCEPTS INVOLVED:

The implementation of the library management system incorporates two fundamental data structures: the Red-Black Tree and the Binary Min-Heap. These structures are crucial for efficient data management and retrieval.

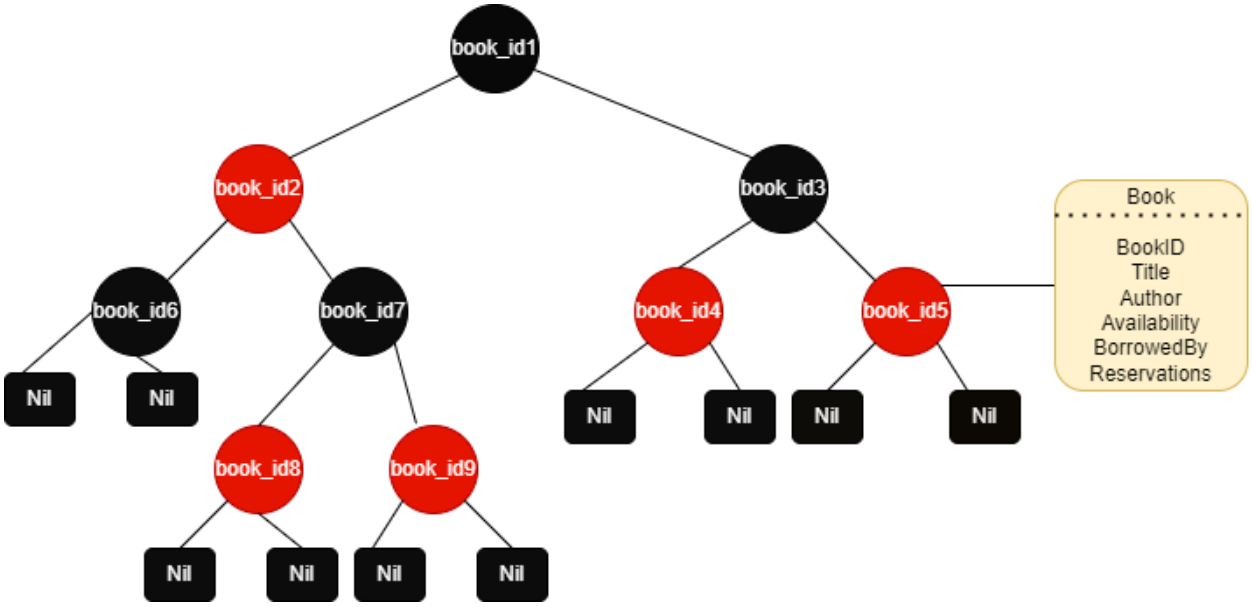
RED-BLACK TREE

A Red-Black Tree is a specialized binary search tree with self-balancing capabilities. Each node in the tree is assigned a color, either red or black. This color-coding mechanism enables the tree to maintain a specific set of properties that ensure efficient search, insertion, and deletion operations.

Color Property: All nodes are either red or black.
Root Property: The root node is always black.
Nil Property: Every leaf (null child) is black.
Red-Property: Red nodes cannot have red children. Therefore, a red node must have two black children.
Black-Height Property: Every path leading from the root node to any leaf node in the Red-Black Tree traverses the same number of black nodes, ensuring a consistent depth across all paths.

These properties guarantee that the tree maintains a nearly balanced state, leading to efficient insertion, deletion, and search operations with a time complexity of O(log n), where n denotes the number of nodes in the tree.

In the Code: The Red-Black Tree is used to manage the collection of books in the library. Each book is represented as a node in the tree (Node class), with attributes like book ID, title, author, availability status, borrowed by, and reservations. The tree structure allows for efficient searching (to find a book), insertion (adding a new book), and deletion (removing a book).



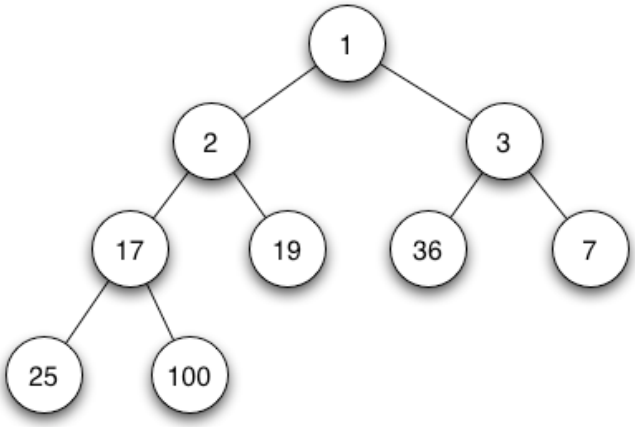
BINARY-MIN HEAP

A Binary Min-Heap is a specialized tree structure where each node's value is greater than or equal to the value of its parent. This unique property ensures that the root node always holds the minimum value within the entire heap. Typically represented as an array, this data structure enables efficient retrieval of the smallest element.
Key operations on a Min-Heap include:

Insertion: Introduce a new element while preserving the heap's structural integrity. This involves placing the element at the end of the heap and subsequently moving it upwards to its rightful position (heapify-up).

Extract Min: Remove and return the smallest element, which resides at the root. The last element in the heap is then relocated to the root, and the heap property is restored by moving this element downwards to its correct position (heapify-down).

In the Code: The Binary Min-Heap is used within each book node to manage reservations for the book. When a patron wants to borrow a book that is not currently available, their reservation is added to this min-heap, prioritized by their reservation priority and time of reservation. This structure ensures that the book reservations are handled efficiently and fairly.



INTEGRATION IN THE CODE

The integration of these two data structures provides a powerful combination for the library management system:

- The Red-Black Tree offers a highly efficient mechanism for managing the overall set of books, ensuring fast access and modification operations.
- The Binary Min-Heap within each book node efficiently manages the queue of reservations, ensuring that patrons are served in the correct order based on their priority.

This combination allows the system to handle various library operations such as adding, borrowing, returning, and managing book reservations effectively, ensuring both speed and fairness in the library management process.

OPERATIONS SUPPORTED

1. **printBook(bookId):** This method displays details of a particular book, uniquely identified by its bookId, including its bookName, author, current availability status, borrowed by, and reservations. In instances where the specified book cannot be located within the library's collection, the method outputs a message stating, "Book not found in the Library."

2. **printBooks(bookId1, bookId2):** This function retrieves and displays details of all books within the specified range of bookIds, from bookId1 to bookId2, inclusive.

3. **insertBook(bookID, bookName, authorName, availabilityStatus):** This method is responsible for adding a new book entry to the library's catalog. The book_id serves as a unique identifier, and the availability status indicates whether the book is currently available for lending.

4. **borrowBook(patronId, bookId, patronPriority)**: This function enables patrons to borrow an available book and updates the book's status accordingly. For books that are not available, it generates a reservation entry in the heap, prioritized by the patron's assigned priority level (patronPriority).

5. **returnBook(patronId, bookId)** Through this method, patrons can return books they have borrowed. It updates the book's availability and, if applicable, allocates the book to the next patron in line with the highest priority on the reservation heap.

6. **deleteBook(bookId)**: This function removes a book from the library's system and informs patrons on the reservation list that the book is no longer available for borrowing.

7. **findClosestBook(targetId):** This method searches for and presents information on the book whose ID is nearest to the specified targetId. If there are two equally close books, it displays information for both, sorted by their book_ids.

8. **colorFlipCount():** In the Red-Black tree structure, this function is utilized to assess and report the frequency of color flips. It monitors changes in node colors occurring during various tree operations, such as insertions, deletions, and rotations.

PROGRAM EXECUTION FLOW

1. Initialization: The program starts in the main function, where it initializes the Red-Black Tree.
2. Input Processing:
    - Reads the input file and iterates through each command.
    - Each line/command is parsed to determine the operation and its parameters.
3. Command Execution:
    - Based on the parsed information, the corresponding method in the Red-Black Tree or Binary Min-Heap is executed.
    - Operations include inserting a new book, borrowing or returning a book, deleting a book, and other book management functionalities.
4. Output Handling:
    - The results of operations are written to an output file
5. Error Handling:
    - Throughout the process, the program includes error handling to manage exceptions, especially in file operations and input parsing.

FUNCTION PROTOTYPES/CODE STRUCTURE:

The program is structured around two main data structures: the Red-Black Tree and the Binary Min-Heap, embodied in the RedBlackTree, BinaryMinHeap, and Node classes

1. **BinaryMinHeap Class**: Each book node contains a min-heap for managing reservations. This heap prioritizes reservations based on a patron's priority number and the time of reservation, allowing efficient management of waitlists for borrowed books.
   Implementing a min-heap for managing reservations includes following key functions:
   - **insertReservation(res):** Inserts a reservation into the min-heap while maintaining the heap property. It appends the reservation at the end of the heap array and then applies a heapify-up procedure to maintain the min-heap property.

     Time complexity: The worst-case time complexity of the operation is O(log k), where k represents the number of reservations in the heap. This is because in the worst-case scenario, the element may need to be swapped repeatedly until it reaches the root position.

   - **heapify(idx):** Maintains the min-heap property by arranging elements starting from the specified index. It compares the node with its children and swaps with the smallest child if necessary, continuing the process down the tree.

     Time complexity: O(log k), as it may need to heapify down to the leaf nodes.

   - **extractMin()**: Removes and returns the minimum reservation (based on priority and timestamp) from the heap. Swaps the root with the last element, removes the last element (now the minimum), and then applies a heapify-down procedure from the root.

     Time complexity: O(log k), since it involves removing the root and re-heapifying the tree.

2. **Node Class:** Represents a book in the Red-Black Tree, with attributes like book ID, title, author, availability status, borrowed by and a reservation heap.
   Implementing node class involves defining a constructor with following attributes-

   - **init(bookId, bookName, authorName, availabilityStatus, borrowedBy=None) :** Initializes a new Node object representing a book with its attributes. Sets the book's properties like ID, title, author, availability status, and the borrower's ID, along with initializing an empty reservation heap.'

     Time complexity: O(1), as it's just the initialization.

3. **RedBlackTree Class**: This class manages the library's collection of books. Each book is represented as a node in the tree (Node class) with unique identifiers and attributes like title, author, and availability status. The tree structure ensures efficient operations for adding, deleting, and searching books, maintaining balance through rotations and color flips. Implementing the Red-Black Tree includes following Key functions:

    - **printBooks(bookId1, bookId2):** This function retrieves and displays information about all books within the specified range of book Ids (inclusive of bookId1 and bookId2). It uses in-order traversal of the Red-Black Tree to find and print the books.

      Time Complexity: The operation takes O(log n + k) time, where n is the total number of books and k is the number of books in the range. Log n comes from searching the tree, and k comes from accessing books in the range.

    - **printBook(bookID):** This method is responsible for displaying the information of a specific book identified by its unique bookId. It presents various details such as the book's title, author, and availability status. If the book corresponding to the given bookId is not found in the library's collection, the method outputs a message indicating that the book is not found.

      Time complexity: The time complexity of this operation is O(log n), where n is the total number of books in the tree. This complexity arises from the requirement to search through the tree in order to locate the specified book_id.

    - **insertBook(bookID, bookName, authorName, availabilityStatus)**: It adds a new book entry to the library with its details. The book's ID is unique, and its availability status indicates if it's available for borrowing. Following the Red-Black Tree property that all new nodes are colored red, it initializes the new node with a red color. Beginning from the root of the tree, it searches for the appropriate location to insert the new node by comparing its book ID to the current node's book ID. If the new node's book ID is less than the current node's book ID, it moves to the left child; otherwise, it moves to the right child. Once the correct position is found, the new node is inserted as a leaf. If the tree was initially empty, this new node becomes the root. If the new node's parent is the root or if the tree was previously empty, the new node is recolored to black. After insertion, it checks for violations of Red-Black Tree properties to ensure the tree remains balanced and efficient.

      Time Complexity: The complexity is O(log n), where n is the number of books in the tree. This is due to the self-balancing properties of Red-Black Trees, which maintain relatively even tree heights.

- **borrowBook(patronID, bookID, patronPriority):** It searches the Red-Black Tree for the book and modifies the Binary Min-Heap for reservations. It allows a patron to borrow a book if available, updating the book's status. For unavailable books, it adds a reservation in the min-heap based on the patron's priority. The book specified by bookId is first located using the searchTreeHelper. If the book is not found, a message is displayed. When found, the method checks if the book is available. For available books, it updates the status to 'borrowed' and assigns the patronId. If the book is already borrowed, it adds the patron's reservation to the book's reservation heap, prioritizing by patronPriority and timestamp. This process involves updating node information without altering the tree's structure, ensuring efficient library management without the need for Red-Black Tree balancing operations.

  Time Complexity: The complexity is O(log n) for searching the book, and O(log k) for inserting into the heap, where n is the number of books and k is the number of reservations.

- **returnBook(patronId, bookId)**: It enables a patron to return a borrowed book, updating its availability status and allocating it to the next highest-priority patron in the reservation heap if applicable. If the returning patron matches the current borrower (patronId), the book's availability is updated to 'available'. The method then checks the book's reservation heap; if there are reservations, the book is assigned to the highest priority patron. Otherwise, it remains available for others. This streamlined process efficiently manages book returns and the reservation system, ensuring fair access to library resources.

  Time Complexity: The complexity is O(log n) for tree manipulation, and O(log k) for heap operations, where n is the number of books and k is the number of reservations.

- **deleteBook(bookId)**: Removes a book from the library and notifies patrons in the reservation list that it is no longer available. After deleting the node, it handles any necessary rebalancing.
  It starts by locating the book using searchTreeHelper with the given bookId. If found, the method notifies all patrons with reservations about the book's removal, maintaining transparent communication. Subsequently, it employs deleteNodeHelper to remove the book node from the tree, ensuring the tree's balance and properties are preserved. This process is essential for updating the library's inventory while maintaining the efficiency of the tree structure.

  Time Complexity: O(log n), where n is the number of books in the tree.

- **findClosestBook(targetId)**: It performs a targeted traversal in the Red-Black Tree to find the closest bookId. If there are ties, it prints information for both books, ordered by their IDs. It traverses the tree, tracking the closest lower and higher book IDs relative to the target. If an exact match is found, its details are displayed. Otherwise, it determines which of the closest books, if any, is nearer to the target ID and presents its details. This method effectively assists in finding alternative options when a specific book is not available.

  Time Complexity: O(log n), where n is the number of books in the tree.

- **colorFlipCount():** It monitors and reports the frequency of color flips in the Red-Black Tree, tracking changes in node colors during tree operations.

  Time Complexity: O(1) as it only accesses and reports a counter variable.

The program structure also includes some utility functions such as -

1. **readInputFile(inputFile)**: This function is designed to read and retrieve the contents of an input file specified by input_file. It opens the file, reads each line, and stores these lines in a list, which is then returned. This function is essential for processing the input file containing various library operation commands.

   Time Complexity: O(m), where m is the number of lines in the file. The complexity is linear with the size of the file, as each line is read sequentially.

2. **parseLine(line):** This function parses a single line from the input file to extract the method name and its arguments. It uses regular expressions to identify the method name and the list of arguments, separating them for further processing. This is crucial for interpreting and executing the commands in the input file.

   Time Complexity: O(n), where n is the length of the input line. The complexity depends on the length of the line being parsed.

3. **main(inputFile):** It acts as the entry point of the program, orchestrating the overall flow. It reads the input file using read_input_file, processes each line with parse_line, and executes the corresponding functions in the library management system based on the operations specified in the input file.

EXECUTION STEPS:

1. Unzip Deodhar_Pratiksha.zip.

2. Open Terminal in the folder that is created after unzipping the file, i.e., “Deodhar_Pratiksha” and run the command in the following format:

```
python app.py <input_filename>.txt
```

Example:

```
python app.py test1.txt
```

3. Find the output in the text file generated by the code. It will be in the following format:

```
input_filename_output_file.txt
```

Example:

```
test1_output_file.txt
```