Name : Rahul Talari

UFID : 46063916

Mail : rtalari@ufl.edu

ADS project report

ABSTRACT:-

A complex software system has to be created for the fictional library GatorLibrary in order to manage its books, users, and borrowing procedures more efficiently. Using a Red-Black tree data structure for effective book management and putting in place a Binary Min-heap priority queue mechanism for seamless book reservations are the main goals. A unique numerical ID is assigned to each book, and the system guarantees that each book is distinct, upholding the idea of one copy per book.

The proposed system includes a number of features, including printing comprehensive descriptions of individual books and collections of books, adding new books to the library, letting users check out and return books with priority-based reservations, removing books from the library, locating the book that is closest to a given ID, and keeping track of how often the Red-Black tree structure's colour changes occur.

The implementation is thoroughly described in this project report, which also explains the underlying data structures—the Red-Black tree and Binary Min-heap—and the nuances of each function. Python is the programming environments of my choice, and testing is done on the thunder.cise.ufl.edu server.

Problems were solved during the development process, and the system underwent extensive testing to guarantee accuracy and effectiveness. The report ends with an analysis of the goals attained, a discussion of possible improvements for the future, and an acknowledgement of the people and materials that helped make the project a success.

The project's results provide the groundwork for a sophisticated library management system that incorporates reliable data structures and prioritization techniques, guaranteeing a smooth user experience for administrators and customers alike.

Introduction:-

GatorLibrary emerges as a vibrant imaginary library, where the timeless allure of books converges with the dynamic demands of modern library management. This endeavor is fueled by the recognition that traditional libraries are evolving, necessitating the infusion of cutting-edge technology to meet the growing expectations of patrons.

In this digital age, the concept of GatorLibrary materializes to address the imperative need for a sophisticated software system. This system aims not only to preserve the essence of a conventional library but also to revolutionize the way books are managed, patrons are served, and borrowing operations unfold.

This research explores the fields of data structures and algorithmic efficiency in its quest to design a complete solution. The main goal is to improve the entire experience for users by giving them a simple and easy-to-use platform for book exploration, borrowing, and return.

Objectives:-

The primary objective of this project is to design, implement, and analyze the efficiency of a software system tailored for GatorLibrary, a fictional library. The system aims to enhance the management of books, patrons, and borrowing operations through the incorporation of advanced data structures such as red-black trees, Binary heap.

PrintBook(bookID):-

Objective 1:- Display detailed information about a specific book identified by its unique bookID, including title, author, and availability status.

PrintBooks(bookID1, bookID2):

Objective: Provide information about all books within the specified range of bookIDs [bookID1, bookID2].

InsertBook(bookID, bookName, authorName, availabilityStatus, borrowedBy, reservationHeap):-

Objective: Add a new book to the library with a unique bookID and specified details, including availability and reservation information.

BorrowBook(patronID, bookID, patronPriority):-

Objective: Allow a patron to borrow an available book, updating its status. If unavailable, create a reservation based on patronPriority.

ReturnBook(patronID, bookID):-

Objective: Enable a patron to return a borrowed book. Update the book's status and assign it to the highest-priority patron in the Reservation Heap if reservations exist.

DeleteBook(bookID):-

Objective: Remove a book from the library and inform patrons in the reservation list about its unavailability.

FindClosestBook(targetID):-

Objective: Locate the book with an ID closest to the given targetID, considering both sides. Print details, and in case of ties, display both books ordered by bookIDs.

ColorFlipCount():-

Objective: Implement an analytics tool to monitor color flips (changes from red to black and vice versa) in the Red-Black tree during operations.

The above mentioned are the most important functions for Gator library and the implementation is done giving these objectives the prime importance.

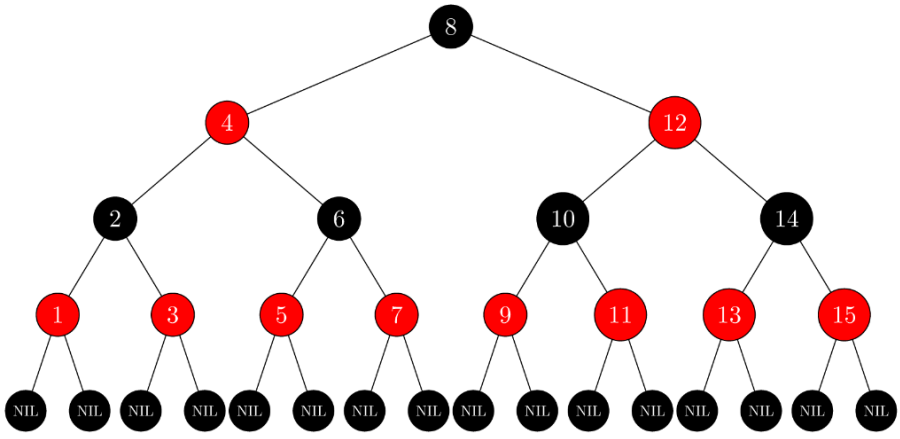
Project overview:-

The GatorLibrary system has been painstakingly developed to bring in a new level of sophistication and efficiency in library management. Fundamentally, the system utilizes a strong architecture that gracefully incorporates Red-Black trees and Binary Min-heaps—two essential data structures that are essential to the smooth operation of the library.

Red-Black Trees:-

The use of Red-Black trees is the basis of GatorLibrary's book management system. This self-balancing binary search tree guarantees optimal search, insertion, and deletion operations in addition to organizing the large book collection. The innate balancing capabilities of the Red-Black tree reduces performance bottlenecks and ensures a logarithmic time complexity for critical operations.

Important details about a book are contained in each node of the Red-Black tree, such as the book's distinct BookID, title, author, availability status, borrower information, and a Binary Min-heap for reservations. The efficient tracking and retrieval of information is made possible by the color-coded nodes, which enhance the overall efficacy of the book management system in the library.



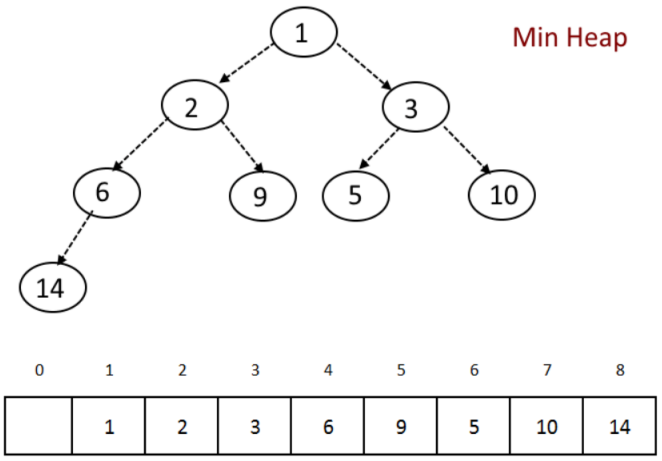
Binary Min-heaps:-

GatorLibrary uses Binary Min-heaps in parallel to expedite the reservation procedure. First-come, first-served bookings are avoided because to Min-heaps' priority-based booking system, which allows users to reserve books according to their level of priority. When a book becomes available, the heap

structure expeditiously maintains waitlists, enabling prompt retrieval of the highest-priority reservations.

The PatronID, priority number, and timestamp of each reservation are stored in the Binary Min-heap node, which represents a reservation. This system improves the entire customer experience by ensuring that book reservations are handled fairly and openly.

The interplay of Red-Black trees and Binary Min-heaps in the GatorLibrary system is a prime example of a well-balanced combination of data structures, carefully selected to maximize efficiency and provide an unmatched library administration experience.



Implementation of Red Black tree and Binary Min-heaps:-

The red black tree and binary min heap are implemented from the scratch for this project. Now lets see the implementation of Red Black Tree.

RedBlackTree:

Pseudo code:

```
def __init__(self):
    self.NIL = RBTreeNode(None, None, "BLACK", None, None, None)
    self.root = self.NIL
```

Represents the Red-Black tree itself.

NIL: Sentinel node representing a leaf (external) node with properties of BLACK color.

root: Root node of the Red-Black tree.

insert(key, value):

Pseudo code:

```
def insert(self, key, value):
```

```
    # Perform standard BST insert
    new_node = RBTreeNode(key, value, "RED", self.NIL, self.NIL, None)
    self._insert(new_node)
    # Fix any violations of Red-Black tree properties
    self._fix_insert(new_node)
```

Adds a new node with the given key and value to the tree.

Performs a standard Binary Search Tree (BST) insertion.

_insert(node):

Pseudo code:

```
def _insert(self, node):
    # Similar to standard BST insert, inserting a new node into the tree
```

Helper function for the actual insertion.

Inserts a new node into the tree, similar to a standard BST insert.

_fix_insert(node):

Pseudo code:

```
def _fix_insert(self, node):
    # Fix any violations of Red-Black tree properties after insertion.
```

Fixes any violations of Red-Black tree properties that might occur after insertion.

Performs rotations and color adjustments.

_left_rotate(x) and _right_rotate(y):

Pseudo code:

```
def _left_rotate(self, x):
    # Perform a left rotation on the tree
def _right_rotate(self, y):
    # Perform a right rotation on the tree
```

Helper functions to perform left and right rotations, respectively.

->Now lets see for the Bin Min heap.

MinHeap:

Pseudo code:

```
class MinHeap:
    def __init__(self):
        self.heap = []
```

Represents a Binary Min-Heap.

heap: List to store elements.

push(element):

```
def push(self, element):
        # Add an element to the heap and maintain the heap property
        self.heap.append(element)
        self._heapify_up(len(self.heap) - 1)
```

Adds a new element to the heap while maintaining the Min-Heap property.

_heapify_up(index) is called to adjust the heap.

pop():

Pseudo code:

```
def pop(self):
        # Remove and return the smallest element from the heap and maintain the heap property
        if not self.heap:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()
        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
    return root
```

Removes and returns the smallest element from the heap.

_heapify_down(index) is called to adjust the heap.

_heapify_up(index):

Pseudo code:

```
def _heapify_up(self, index):
    # Maintain heap property while moving an element up the heap
```

Moves an element up the heap to maintain the Min-Heap property.

_heapify_down(index):

Pseudo code:

```
def _heapify_down(self, index):
    # Maintain heap property while moving an element down the heap
```

Moves an element down the heap to maintain the Min-Heap property.

__iter__(self):

Pseudo code:

```
def __iter__(self):
    # method that returns the iterator object itself .
```

Programming Envieronment:-

Vscode

python

Code implementation:-

PrintBook(bookID):

Locate the book with the given bookID.

Print information about the book if found.

If not found, print a message indicating that the book is not in the library.

PrintBooks(bookID1, bookID2):

Iterate over the range of book IDs [bookID1, bookID2].

For each book ID, print information about the corresponding book if it exists.

InsertBook(bookID, bookName, authorName, availabilityStatus, borrowedBy, reservationHeap):

Create a new book node with the provided information.

Insert the new book into the Red-Black tree.

Ensure the uniqueness of the bookID.

Update the availability status, borrower, and reservation heap accordingly.

BorrowBook(patronID, bookID, patronPriority):

Locate the book with the given bookID.

If the book is available, update its status and borrower.

If the book is unavailable, create a reservation node in the priority queue (Binary Min-heap) based on patronPriority.

Ensure a book is not borrowed by the same patron twice.

ReturnBook(patronID, bookID):

Locate the book with the given bookID that is borrowed by the specified patron.

Update the book's status and borrower.

If there are reservations, assign the book to the patron with the highest priority.

DeleteBook(bookID):

Locate the book with the given bookID.

Delete the book from the Red-Black tree.

Notify patrons in the reservation list that the book is no longer available.

FindClosestBook(targetID):

Find the book with an ID closest to the given targetID by checking on both sides.

Print details about the closest book(s). In case of ties, print both books ordered by book IDs.

ColorFlipCount():

Track the frequency of color flips in the Red-Black tree during tree operations.Count color changes during insertions, deletions, and rotations.

STEPS TO RUN:-

Step 1: Unzip the folder Rahul_Talari.zip

Step 2: Enter into ADS Project

Step 3: To run the file, type in the command => python3 gatorlibrary.py < test "no".txt>. Ex: python3 gatorLibrary.py test_1.txt