COP 5536 - ADVANCED DATA STRUCTURES

Anol Kurian Vadakkeparampil
UFID: 5626 8544
anolkuri.vadakke@ufl.edu

Overview:

The code implements a library management system using data structures like binary min heap and red black tree etc. It allows operations like adding a book, printing book details, borrowing, returning, deleting a book, find closest book etc.

Function Prototypes:

- **Class BookNode:**
  def __init__(self, bookId, bookName, authorName, availabilityStatus)
  def addReservation(self, patronId, priorityNumber)
  def getReservations(self)

- **Class RedBlackNode:**
  def __init__(self, val: BookNode)

- **Class RedBlackTree:**
  def __init__(self)
  def insert(self, val)
  def find(self, val)
  def rotateLeft(self, x)
  def rotateRight(self, x)
  def delete(self, val)
  def fixInsert(self, newNode)
  def fixDelete(self, x)
  def transplant(self, u, v)
  def minimum(self, x)

- **Class ReservationNode:**
  def __init__(self, patronId, priorityNumber, timeOfReservation)

- **Class BinaryMinHeap:**
  def __init__(self)
  def __iter__(self)
  def insert(self, element)
  def pop(self)
  def heapifyUp(self, currentIndex)
  def swap(self, i, j)
  def removeMin(self)

def heapifyDown(self)

- **Class LibrarySystem:**
  def __init__(self)
  def colorFlipCount(self)
  def printBook(self, bookId)
  def printBooks(self, node, bookId1, bookId2)
    def processBook(book_node)
    def inorderTraversal(node)
  def insertBook(self, bookId, bookName, authorName, availabilityStatus, borrowedBy=None, reservation_heap=None)
  def borrowBook(self, patronId, bookId, patronPriority)
  def returnBook(self, patronId, bookId)
  def deleteBook(self, bookId)
  def cancelReservations(self, bookId, patrons)
  def findClosestBook(self, node, targetId)
  def getBookDetails(self, node)
  def findClosestBookHelper(self, node, targetId, closestLower=None, closestHigher=None)

- **Main Driver Function**

Functions Definitions and Time Complexities:

- **Class BookNode:**
  - *def __init__(self, bookId, bookName, authorName, availabilityStatus)*
    This is the constructor method that creates a new BookNode object to represent a book entity. It takes book id, name, author name and availability status as parameters and assigns these values to attributes or instance variables.
    Time Complexity: This constructor initializes a BookNode by assigning passed parameters to attributes. Only constant time operations like assignment are done. Hence time complexity is O(1).
  - *def addReservation(self, patronId, priorityNumber)*
    This method is used to add a new reservation to the book's waitlist which is maintained as a min heap. It accepts patron id and priority number, creates a ReservationNode object with these values and timestamp, and inserts this reservation object into the min heap. It also checks if heap size exceeds 20, in which case it returns a waitlist full message.
    Time Complexity: This adds a reservation by inserting into a min heap. Insertion into a min heap takes O(logN) time as the element has to float up the heap to maintain heap order property. Hence the complexity is O(logN)
  - *def getReservations(self)*
    This method retrieves all the reservations made for this book. It iterates through the min heap of reservations attached to the BookNode and collects the patron ids into a list. This list containing

patron ids of all reservations is returned by the method.
Time Complexity: This retrieves reservations by iterating through the min heap of size N and collecting ids. Traversing a linear structure gives O(N) complexity.

- **Class RedBlackNode:**
  - *def __init__(self, val: BookNode)*
    This is the constructor method which creates a new node object for the RedBlackTree data structure. It takes the BookNode object as a parameter, initializes the color to red, side links to null/nil, sets the BookNode as its value, and leaves the parent link as null.
    Time Complexity: This constructor only does constant time operations like assignment. So the time complexity is O(1).

- **Class RedBlackTree:**
  - *def __init__(self)*
    This constructor method initializes an empty RedBlackTree. It creates a sentinel nil node, sets its color to black and pointers to null. It also sets nil as the tree root initially.
    Time Complexity: Initializes empty tree with a single sentinel node. Only constant time ops, so O(1) complexity.
  - *def insert(self, val)*
    This method inserts a new node into the RedBlackTree. The normal binary search insert is first done to add the node. Then it calls fixInsert method to fix any red-black properties violated and balance the tree. It handles case logic and performs rotations as needed.
    Time Complexity: Inserts by doing normal binary search insert taking O(logN) time, followed by O(logN) fixup. Total complexity is O(logN).
  - *def find(self, val)*
    This method searches the RedBlackTree recursively for a node containing the given search value passed as parameter. At each step, it compares the node value to the search key and decides whether to traverse left or right subtree. It returns the node if found, else returns None.
    Time Complexity: Searches recursively like binary search, dividing search space in half at each step. Hence O(logN) complexity.
  - *def rotateLeft(self, x)*
    This method rotates the subtree rooted at the node passed as parameter to the left. It makes the right child of x as the new parent, makes x the left child of its right child, and handles parent pointer updates.
    Time Complexity: Rotates a subtree rooted at node by changing constant number of pointers. Hence, O(1) complexity.
  - *def rotateRight(self, x)*
    This method rotates the subtree rooted at the node passed as parameter to the right. It makes the left child of x as new parent, makes x the right child of its left child, and handles parent pointer updates.
    Time Complexity: Rotates subtree right by changing constant number of pointers. Hence, O(1) complexity.

- *def delete(self, val)*
  This method deletes the node with given value if it exists in the tree by first locating it and then handling successor replacement and rebalancing. It calls fixDelete to ensure red-black properties.
  Time Complexity: Deletes by searching for element in O(logN) time, then doing successor replacement and fixup in O(logN) time. Hence overall O(logN).
- *def fixInsert(self, newNode)*
  This method is called after insertion to fix any red-black tree properties violated due to the insertion. It iterates from the newly inserted node up to the root, performs color flips and rotations as needed to balance the tree.
  Time Complexity: Walks up tree from inserted node to root, doing constant color flips/rotations at each step. Tree has height O(logN), hence O(logN).
- *def fixDelete(self, x)*
  This method is called after deletion to fix any red-black tree properties violated due to the removal. It iterates from the child of deleted node to root, performs rotations and color flips to rebalance the tree.
  Time Complexity: Walks up tree from node to root, doing constant color flips/rotations. Height is O(logN), so O(logN).
- *def transplant(self, u, v)*
  This method replaces the subtree rooted at node u with the subtree at node v by transferring the parent pointer of u to v. It is used during deletion and rotation operations.
  Time Complexity: Changes constant number of parent pointers. Hence, O(1).
- *def minimum(self, x)*
  This method finds and returns the node with minimum value under the subtree at the given node. It traverses left links starting from given node until a leaf is reached.
  Time Complexity: Traverses left links downwards till finding minimum. Height is O(logN), so O(logN).

- **Class ReservationNode:**
  - *def __init__(self, patronId, priorityNumber, timeOfReservation)*
    This is the constructor method which initializes a new ReservationNode object representing a booking for a book. It takes patron id, priority number and timestamp as parameters and stores them into attributes of the node.
    Time Complexity: Assigns passed params to attributes in constant time. Hence, O(1) complexity.
- **Class BinaryMinHeap:**
  - *def __init__(self)*
    This constructor initializes an empty min heap data structure by creating an empty list to hold the elements.
    Time Complexity: Creates a constant size list for heap in constant time. Hence, O(1) complexity.
  - *def __iter__(self)*
    This method exposes an iterator object for the underlying list structure of the heap to enable

iterating through the elements.
Time Complexity: Returns iterator object over list in constant time. Hence, O(1) complexity.

- *def insert(self, element)*
  This method inserts a new element into the min heap by appending to the list and calling heapifyUp method to maintain the heap order property. The element floats up to the correct spot.
  Time Complexity: Inserts at end then bubbles up which takes O(logN) in a balanced heap. Hence, O(logN) complexity.
- *def pop(self)*
  This method removes and returns the top minimum element from the heap. The last element is swapped with first, then heapifyDown is called to readjust the heap by pushing the new top element down.
  Time Complexity: Swaps elements in constant time then bubbles down in O(logN) time. Hence, O(logN) complexity.
- *def heapifyUp(self, currentIndex)*
  This method is called on insert to move an element up the heap by swapping it with its parent until heap property is satisfied. It compares based on priority and time attributes.
  Time Complexity: Compares element with parents up the path to root. Height is O(logN), so O(logN).
- *def swap(self, i, j)*
  This is a helper method to swap two elements in the heap's list given their indexes. Used by other heap operations. Runs in constant time.
  Time Complexity: Swaps two elements in constant time. Therefore, O(1) complexity.
- *def removeMin(self)*
  This method removes and returns the minimum element from the heap. It calls pop followed by heapifyDown to readjust the heap.
  Time Complexity: Combines pop and heapifyDown, each O(logN). So total is O(logN).
- *def heapifyDown(self)*
  This method is called after removal to move an element down the heap by swapping with child nodes until heap order property is restored. Compares priority and timestamp.
  Time Complexity: Compares element with children down the path to leaf. Height is O(logN), so O(logN).

- **Class LibrarySystem:**
  - *def __init__(self)*
    This constructor initializes an empty library system by creating a RedBlackTree instance to hold books and an empty dictionary to store patrons mapped to ids.
    Time Complexity: Creates empty tree and dictionary in constant time. Hence, O(1) complexity.
  - *def colorFlipCount(self)*
    This simple getter method returns the colorFlipCount variable storing number of rotations done on the RedBlackTree. Useful for analysis.
    Time Complexity: Accessing a variable takes constant time. Hence, O(1) complexity.

- *def printBook(self, bookId)*
  This method searches for the BookNode with given id in the library's RedBlackTree and prints details including borrowed status and reservations if found. Else prints not found.
  Time Complexity: Searches RedBlackTree for book in O(logN) time. Hence, O(logN) complexity.
- *def printBooks(self, node, bookId1, bookId2)*
  Performs an inorder traversal on the RedBlackTree to visit all nodes, collects details of only those books with ids between given range into a list. This list is returned.
  Time Complexity: Traverses RedBlackTree inorder in O(N) time to print N books. Hence, O(N) complexity.
    - def processBook(book_node)
    - def inorderTraversal(node)
- *def insertBook(self, bookId, bookName, authorName, availabilityStatus, borrowedBy=None, reservation_heap=None)*
  This method allows inserting a new book into the library system. It creates a BookNode and inserts it into the RedBlackTree. Can optionally specify borrowed status and reservations heap.
  Time Complexity: Inserts into RedBlackTree in O(logN) time. Hence, O(logN) complexity.
- *def borrowBook(self, patronId, bookId, patronPriority)*
  This method attempts to borrow the book with given id for specified patron. If book available, it updates details. Else adds a reservation based on priority number given. Handles waitlist logic.
  Time Complexity: Searches for book in O(logN) time, then possibly inserts reservation in O(logN) time. Hence, O(logN) complexity.
- *def returnBook(self, patronId, bookId)*
  This method tries to return the given book for specified patron id. If reservations exist, it allots book to next patron. Handles invalid return case. Prints message.
  Time Complexity: Searches for book in O(logN) time and performs constant time operations. So O(logN) complexity.
- *def deleteBook(self, bookId)*
  This method deletes the book with given id from library tree after cancelling any associated reservations. Prints message about deletion and cancelled reservations if any.
  Time Complexity: Deletes book from RedBlackTree in O(logN) time. Hence, O(logN) complexity.
- *def cancelReservations(self, bookId, patrons)*
  This method is used to cancel any existing reservations for a book when it is deleted or no longer available. It takes the bookId and list of patrons with reservations as parameters. It iterates through each patron in the patrons list and calls the cancel_reservation method on that patron object, passing the bookId.
  Time Complexity: Since it traverses the list of patrons, the time complexity is O(N) where N is the number of patrons with reservations for that book.
- *def findClosestBook(self, node, targetId)*
  This method finds the book nodes closest to the given targetId in the library's RedBlackTree. It makes use of a recursive helper method, passing the root node, targetId and initial closestLower and closestHigher as None. The helper method traverses the tree recursively like binary search, at each step updating closestLower and closestHigher if a closer match is found. The closest

nodes with minimum absolute difference in ids are returned.
Time Complexity: Since it divides the search space in half at each recursion, the time complexity is O(logN) where N is total nodes in the tree.

- *def getBookDetails(self, node)*
  This is a helper method that formats and returns the details of a book from the given BookNode. It accesses the attributes like id, name, author etc. from the node and formats them into a string.
  Time Complexity: Since it does constant work O(1) per node, the overall time complexity is O(1).
  The string is returned.
- *def findClosestBookHelper(self, node, targetId, closestLower=None, closestHigher=None)*
  This method finds and returns the book node(s) closest to the given target book id by calling recursive helper method to traverse left/right based on target id comparison. Returns closest lower and higher nodes with minimum id difference.
  Time Complexity: Searches recursively like binary search dividing search space in half at each step. Therefore, O(logN) complexity.

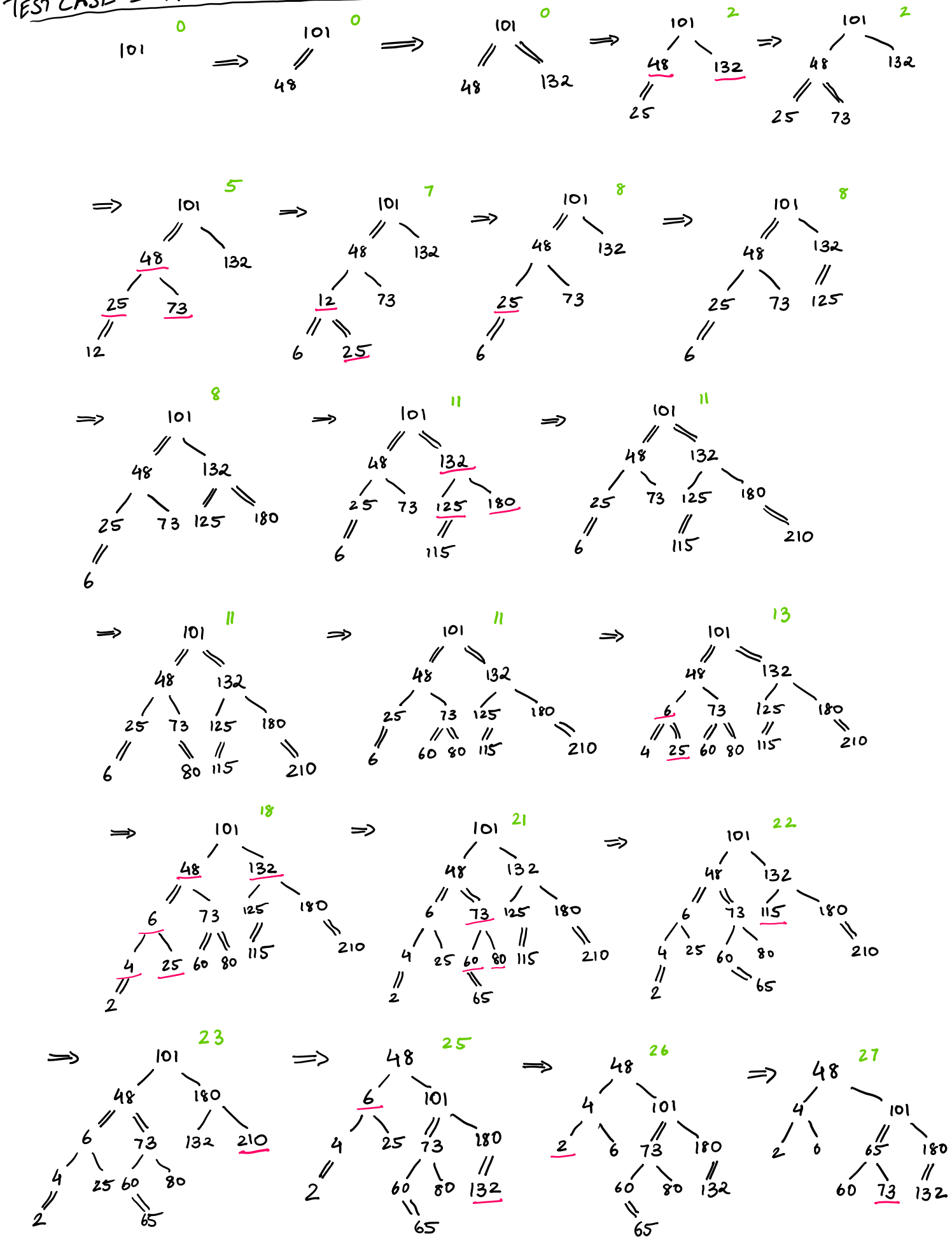
- **Main Driver Function**
  The main driver code opens input command file, parses each line into function name and arguments, calls appropriate library system functions, collects return values and prints outputs. Finally writes output to another text file.
  Time Complexity: Parses input line by line and calls functions. Total lines is O(N). Hence, O(N) complexity.

COLOUR FLIP COUNT LOGIC SHOWCASED IN NEXT SLIDE ONWARDS:
The logic used for red-black tree implementation and the corresponding calculation for colour flip count calculation has been drawn in next pages and the corresponding values have been verified for the testcases.

CONCLUSION:

I was able to implement the above functions mentioned along with the required time complexity and with the correct output obtained. I used all the heap and red black tree operations to implement the gator library functions and in accordance with the specifications given in the problem statement.

TESTCASE 1 FROM ADDITIONAL FILES:-



TESTCASE 2 FROM ADDITIONAL FILES: