Advanced Data Structures Project Report

Name – Srishti Jaiswal

UFID – 80385159

OBJECTIVE

The goal is to build a library management system for GatorLibrary that allows adding, deleting, borrowing, and returning books, along with other useful operations like printing book info. To ensure efficient catalog management, a Red-Black tree will organize unique book data. Additionally, each book entry will include a priority-ordered min heap to manage waitlisted reservations by patrons. Input commands from a text file will drive these operations, and the results will be outputted to another file. The system also tracks the count of color flips in the Red-Black tree for analytics. Its core objective is to implement these functionalities from scratch, emphasizing the use of Red-Black trees and heaps for effective library catalog management.

INTRODUCTION

GatorLibrary is a fictional library that needs an efficient system to manage its collection of books and provide services to patrons. The library requires a software solution to catalog its inventory of books and facilitate essential operations like adding new books, deleting books, allowing patrons to borrow and return books, and querying information about books. To enable fast searches, inserts and deletes in the book catalog, a Red-Black tree data structure will be utilized, with each node containing information about a unique book. Since books may be borrowed but are still in demand, a reservation system is needed to handle waitlisting of patrons requesting a popular booked title. This will be implemented using a min heap for each book, ordered by patron priority. The system needs to process commands from an input file to perform actions like borrowing, returning, printing book details etc. and output appropriate results to another file. Key functionality also includes tracking the number of color flips in the Red-Black tree for analytics. Overall, this program will showcase efficient techniques to catalog and manage a library's inventory of books and provide optimized services to patrons using data structures like Red-Black trees and heaps.

PROGRAM STRUCTURE

To execute this project, you need the file called gatorLibrary.py.

You can run it by using either of the following commands:

python3 gatorLibrary.py <inputfile.txt>

Class Implementation

The code provided utilizes several classes:

Book_Node: Represents a node in the book structure, containing information about a book, such as its ID, name, author, availability status, borrower, and a reservation heap for patrons.

Red_Black_Node: Represents a node in the Red-Black Tree structure, where each node contains a BookNode object.

Red_Black_Tree: Represents the Red-Black Tree data structure, handling operations like insertion, deletion, and balancing nodes. It utilizes RedBlackNode instances to form the tree.

Reservation_Node: Represents a node in the reservation heap, storing details like patron ID, priority, and timestamp of reservation.

Binary_Min_Heap: Implements a binary min-heap structure used for managing reservations in the BookNode class.

Library_System: Represents the main system for managing the library. It utilizes the Red-Black Tree and manages operations such as adding books, printing book details, borrowing, returning, deleting books, finding closest books, and managing reservations.

Main function: Orchestrates the execution by reading commands from an input file, processing those commands to perform library operations using the Library System, and generating output based on the executed commands.

FUNCTION PROTOTYPE

Class Book_Node:
def __init__(self, bookID, bookName, authorName, availabilityStatus):
def make_a_reservation(self, patronID, priorityNum):
def cancel_reservation(self):
def get_reservationHeap(self):

Class Red_Black_Node:
def __init__(self, val: Book_Node):

Class Red_Black_Tree:
def __init__(self):
def insert(self, val):
def search(self, val):
def delete(self, val):
def left_rotation(self, p):
def right_rotation(self, p):
def balance_after_insert(self, inserted_node)
def balance_after_delete(self, p, dict_a)
def reposition(self, u, v)
def get_minimum(self, p)

Class Reservation_Node:
def __init__(self, patronID, priorityNumber, timeOfReservation):

Class Binary_Min_Heap:
def __init__(self):
def insert(self, element):
def pop(self):
def return_elements(self):
def upward_heapify(self, current_index):
def swap(self, i, j):
def remove_min(self):
def downward_heapify(self):

Class Library_System:
def __init__(self):
def color_flip_count(self):
def quit(self):
def add_book(self, bookID, bookName, authorName, availabilityStatus):
def print_book(self, bookID):
def search_book(self, node, bookID):
def print_books(self, node, bookID1, bookID2):
def insert_book(self, bookID, bookName, authorName, availabilityStatus, borrowedBy=None, reservation_heap=None):
def borrow_book(self, patronID, bookID, patron_reservation_priority):

def return_book(self, patronID, bookID):
def delete_book(self, bookID):
def cancel_reservationHeap(self, bookID, patrons):
def find_closest_book(self, node, target_id):
def get_book_details(self, node):
def find_closest_book_helper(self, node, target_id, closest_lower=None, closest_higher=None):

DESCRIPTION AND WORKING

Book_Node Class:

Description:

The BookNode class defines the structure for a node in the library's Red-Black Tree. Each node holds essential details about a specific book, including its unique ID, title, author, availability status, borrower information, and a reservation heap to manage patrons waiting for the book.

Working:

Upon initialization, the class stores book-related information in its attributes. It facilitates adding reservations by creating a tuple with the patron's priority, ID, and timestamp, inserting this reservation into the reservation heap. This class also allows removing reservations and fetching the reservation heap for the book node.

__init__(self, bookID, bookName, authorName, availabilityStatus)

- Description: Initializes a BookNode object with essential book details.
- Working: Stores book-related information like ID, title, author, availability status, borrower details, and initializes an empty reservation heap.
- Time Complexity: This function initializes a BookNode object and operates in constant time, O(1), as it involves simple attribute assignments.

make_a_reservation(self, patronID, priorityNum)

- Description: Adds a reservation for a book based on patron priority.
- Working: Creates a reservation tuple with priority, patron ID, and timestamp. Inserts this reservation into the reservation heap, limiting the heap to 20 reservations.
- Time Complexity: The time complexity for adding a reservation involves inserting an element into the reservation heap, which is a binary min-heap. The complexity of insertion in a binary min-heap is O(log n), where n is the number of elements in the heap. In this case, the limit is set to 20, so it's bounded by log(20), which is constant.

cancel_reservation(self)

- Description: Removes a reservation from the book's reservation heap.
- Working: Removes the minimum priority reservation (highest priority) from the reservation heap if it exists.
- Time Complexity: Removing a reservation from the heap has a time complexity of O(log n), as it involves removing the minimum element from the heap, ensuring the heap property is maintained.

get_reservationHeap(self)

- Description: Retrieves the reservation heap for the book.
- Working: Extracts and returns all patron IDs from the reservation heap.
- Time Complexity: This method extracts all reservations from the reservation heap, which requires removing elements one by one. Since it involves removing all elements, the time complexity is O(n * log n), where n is the number of elements in the heap.

Red_Black_Tree Class:

Description:

The RedBlackTree class implements the Red-Black Tree data structure to organize and manage the book catalog efficiently. It ensures that the tree remains balanced, enabling quick search, insertion, and deletion of book nodes.

Working:

During insertion, this class maintains the Red-Black Tree properties by appropriately balancing the tree through rotations and color flips. It includes methods for inserting book nodes, finding nodes based on book IDs, deleting nodes, and balancing the tree after insertion and deletion to ensure it adheres to the Red-Black Tree rules.

__init__(self)

- Description: Initializes a Red-Black Tree instance.
- Working: Sets up the tree with a nil sentinel node, sets the root to nil, and initializes the color flip counter.
- Time Complexity: The __init__ method in the Red-Black Tree class involves initializing instance variables, setting up the root, and creating the nil node. These operations are not dependent on the size of the tree. Hence, the time complexity is constant, O(1).

insert(self, val)

- Description: Inserts a book node into the Red-Black Tree.
- Working: Inserts a node into the tree based on the book's ID and ensures the tree remains balanced by performing rotations and color flips as needed.
- Handles rotations: Left and right rotations restructure the tree to maintain the binary search tree properties and tracks the number of color flips occurring during insertion to monitor tree balance.
- Time Complexity: In the worst case, inserting a node involves traversing the tree from the root to the leaf node. As the tree is balanced, the height of a Red-Black Tree with n nodes is log n. After insertion, it performs a rebalancing operation that may take O(log n) time in the worst case due to rotations and color flips.

search(self, val)

- Description: Searches for a book node with a given book ID.
- Working: Conducts a search in the tree to find the node corresponding to the provided book ID. Utilizes binary search tree properties to efficiently locate the specific node based on the book's ID.
- Time Complexity: Similar to insertion, searching in a Red-Black Tree involves traversing the tree from the root to the leaf node, making it logarithmic in the number of nodes, therefore O(log n).

delete(self, val)

- Description: Deletes a book node from the Red-Black Tree.
- Working: Removes a node from the tree while maintaining its properties by balancing the tree and counts color flips post-deletion to ensure it remains a valid Red-Black Tree.
- Time Complexity: Deletion involves finding the node to delete and possibly replacing it with its successor. The deletion and rebalancing operations have a maximum time complexity of O(log n) due to tree traversal and possible rebalancing operations.

balance_after_insert(self, inserted_node)

- Description: Balances the tree after inserting a node.
- Working: Ensures the Red-Black Tree properties are preserved after inserting a node by considering different cases of tree structure and colors. Also, performs necessary rotations and color flips to maintain balance in the tree.
- Time Complexity: The rebalancing process after insertion involves a maximum of O(log n) rotations and color flips to restore the Red-Black Tree properties, maintaining a balanced structure.

balance_after_delete(self, p, dict_a)

- Description: Balances the tree after deleting a node.

- Working: Restores the Red-Black Tree properties after deleting a node by adjusting the tree structure and colors. Handles different cases of tree structure and node colors post-deletion and performs rotations and color adjustments as required to maintain the Red-Black Tree properties.
- Time Complexity: Similar to insertion, after deletion, rebalancing operations might be needed. The maximum time complexity for rebalancing is O(log n) due to rotations and color flips.

left_rotation(self, p)

- Description: Performs a left rotation around a given node.
- Working: Restructures the tree by performing a left rotation around the specified node. Adjusts pointers to ensure the binary search tree property is maintained post-rotation.
- Time Complexity: Rotations perform constant time operations by rearranging pointers without iterating through tree nodes, hence O(1).

right_rotation(self, p)

- Description: Performs a right rotation around a given node.
- Working: Restructures the tree by performing a right rotation around the specified node. Adjusts pointers to ensure the binary search tree property is maintained post-rotation.
- Time Complexity: Rotations perform constant time operations by rearranging pointers without iterating through tree nodes, hence O(1).

reposition(self, u, v)

- Description: Replaces one subtree as a child of its parent with another subtree.
- Working: Replaces the subtree rooted at node u with the subtree rooted at node v. Updates parent-child relationships accordingly.
- Time Complexity: The reposition operation performs pointer rearrangements, which are constant time operations, resulting in O(1).

get_minimum(self, p)

- Description: Finds the minimum node in a subtree rooted at a given node.
- Working: Traverses the left child of the given node recursively until finding the node with the minimum value and returns the node with the smallest value found within the subtree.
- Time Complexity: The minimum method is used to find the minimum value in the Red-Black Tree, starting from a given node x. In a binary search tree (BST), the minimum value resides in the leftmost node. As it traverses only the left child pointers until it reaches the leftmost node, the time complexity is O(log n) in a balanced tree. However, if it's not a balanced tree, the time complexity can be O(n) in the worst-case scenario if the tree resembles a linked list.

Binary_Min_Heap Class:

Description:

The BinaryMinHeap class represents a min heap structure used for managing reservations of books. It prioritizes reservations based on patron priority numbers and timestamps.

Working:

The class manages the heap by allowing the insertion of reservations, popping the minimum priority reservation (highest priority), and ensuring the heap properties are maintained through heapify-up and heapify-down operations.

__init__(self)

- Description: Initializes an instance of a Binary Min Heap.
- Working: Initializes an empty list (heap) to store elements.
- Time Complexity: Initializing an empty list to store heap elements is a constant-time operation so, O(1).

__iter__(self)

- Description: Enables iterating through the elements of the heap.
- Working: Allows iteration through the elements present in the heap.
- Time Complexity: The __iter__ method returns an iterator object for the heap, which takes constant time to set up and return, so it will be O(1).

insert(self, element)

- Description: Inserts a reservation into the Binary Min Heap.
- Working: Adds a reservation tuple into the heap and maintains the heap properties by heapify-ing up.
- Time Complexity: Insertion into a binary min-heap involves adding the element to the end of the list and then performing a "heapify-up" operation to maintain the heap property. This operation has a time complexity of O(log n), where n is the number of elements in the heap.

pop(self)

- Description: Removes and returns the minimum priority reservation from the heap.
- Working: Removes the root element (minimum priority) and rearranges the heap to maintain its structure.
- Time Complexity: Removing the minimum element (root) from the heap involves swapping it with the last element, removing the last element, and then performing a "heapify-down" operation to maintain the heap property. The "heapify-down" operation has a time complexity of O(log n).

remove_min(self)

- Description: Removes the minimum priority reservation from the heap.
- Working: Removes the minimum priority reservation element while maintaining the heap structure.
- Time Complexity: Similar to the pop method, removing the minimum element performs a "heapify-down" operation to maintain the heap property, resulting in a time complexity of O(log n).

upward_heapify(self, current_index)

- Description: Ensures the heap property is maintained by moving an element up.
- Working: Adjusts the heap by moving an element up the heap structure based on its priority and timestamp.
- Time Complexity: This method adjusts the heap from a specific index upward to maintain the heap property. The time complexity is O(log n) because it traverses up the heap's height, which is log n in a binary heap.

downward_heapify(self)

- Description: Maintains the min heap property by moving an element down to its correct position.
- Working: Compares the element with its children iteratively, swapping positions until the heap property is satisfied.
- Time Complexity: This method adjusts the heap from a specific index downward to maintain the heap property. The time complexity for this operation is O(log n) because it traverses down the heap's height, which is log n in a binary heap.

return_elements(self)

- Description: Returns all elements in the heap.
- Working: Returns the entire list of elements representing the heap.
- Time Complexity: Retrieving all elements from the heap involves returning the entire list, which takes constant time.

swap(self, i, j)

- Description: Swaps the positions of elements at two specified indices.
- Working: Exchanges the elements at the given indices in the heap list.
- Time Complexity: The swap method swaps the elements at indices i and j in the heap list. Since it directly operates on list elements and performs constant-time swapping, its time complexity is O(1)

Library_System Class:

Description:

The LibrarySystem class serves as the core system orchestrating library operations. It manages the Red-Black Tree structure, handles patrons, and executes book-related operations.

Working:

This class integrates the Red-Black Tree and Binary Min Heap to execute book operations such as borrowing, returning, printing book details, finding closest books, and deleting books. It interacts with the other classes to manage the library catalog efficiently, ensuring book availability, maintaining reservations, and handling patron interactions.

__init__(self)

- Description: Initializes an instance of the LibrarySystem class.
- Working: Creates a Red-Black Tree instance to manage the book catalog and initializes a dictionary to store patron information.
- Time Complexity: O(1) Initializes the LibrarySystem object. It involves simple variable assignments and doesn't depend on any iteration or other operations.

color_flip_count(self)

- Description: Retrieves the count of color flips in the Red-Black Tree.
- Working: Returns the count of color flips tracked in the Red-Black Tree.
- Time Complexity: Retrieves the color flip count from the Red-Black Tree, which is stored as a variable in the RedBlackTree object. This operation is constant time, therefore, O(1).

quit(self)

- Description: Exits the program.
- Working: Calls the exit() function to terminate the program.
- Time Complexity: Executes the exit() function, which terminates the program. This is a constant time operation.

add_book(self, bookID, bookName, authorName, availabilityStatus)

- Description: Adds a book to the library catalog.
- Working: Creates a new BookNode instance with the provided book details and inserts it into the Red-Black Tree.
- Time Complexity: Inserts a new book node into the Red-Black Tree. The time complexity is logarithmic O(logN) because it involves searching and inserting into the Red-Black Tree, which has a logarithmic complexity for insertion.

print_book(self, bookID)

- Description: Prints details of a specific book.
- Working: Finds the book using its ID in the Red-Black Tree and retrieves its details. Returns the book details if found; otherwise, returns a message indicating that the book was not found.
- Time Complexity: Searches for a specific book in the Red-Black Tree by its ID. The time complexity is logarithmic i.e. O(logN) due to searching in the Red-Black Tree.

search_book(self, node, bookID)

- Description: Searches for a specific book in the Red-Black Tree.
- Working: Traverses the Red-Black Tree to find the book by its ID.

print_books(self, node, bookID1, bookID2)

- Description: Prints details of multiple books within a specified range of IDs.
- Working: Traverses the Red-Black Tree to find books within the given range of IDs. Collects the details of the books found and returns them as a list.

- Time Complexity: Prints details of books within the specified ID range. It traverses a portion of the tree using an inorder traversal, visiting N nodes where N is the number of books within the specified range, therefore, O(N).

insert_book(self, bookID, bookName, authorName, availabilityStatus, borrowedBy=None, reservation_heap=None)

- Description: Inserts a book into the catalog with additional details.
- Working: Creates a BookNode instance with the provided details, including availability status, borrower information, and reservation heap (if available). Inserts the new book into the Red-Black Tree.
- Time Complexity: Inserts a book into the Red-Black Tree. The time complexity is logarithmic O(logN) due to the insertion operation in the Red-Black Tree.

borrow_book(self, patronID, bookID, patron_reservation_priority)

- Description: Handles the borrowing of a book by a patron.
- Working: Checks the availability of the book. If available, allows the patron to borrow it. If the book is unavailable, adds the patron to the reservation list. Returns appropriate messages indicating the status of the borrowing process.
- Time Complexity: The method borrows a book or adds a reservation if the book is unavailable. It operates on the reservationHeap within the book node and has a time complexity of O(log N) for Red-Black Tree operations.

return_book(self, patronID, bookID)

- Description: Handles the return of a book by a patron.
- Working: Checks if the book is borrowed by the specified patron. If yes, returns the book and assigns it to the next patron in the reservation list (if any). Updates the book status and borrower information accordingly.
- Time Complexity: If the book is borrowed, it checks reservations and updates the book's availability status or assigns it to the next patron. The time complexity is O(log N) for Red-Black Tree operations.

delete_book(self, bookID)

- Description: Deletes a book from the catalog.
- Working: Deletes the specified book from the Red-Black Tree if found. Cancels reservations for the deleted book. Returns appropriate messages indicating the status of the deletion process.
- Time Complexity: Deletes a book node from the Red-Black Tree using the delete method, which has a time complexity of O(log N).

cancel_reservations(self, bookID, patrons)

- Description: Cancels reservations for a deleted book.
- Working: Iterates through the reservation list of the deleted book and cancels reservations for the respective patrons.
- Time Complexity: O(K log N), where K is the number of canceled reservations. Cancels reservations associated with a book. It depends on the number of canceled reservations and Red-Black Tree operations.

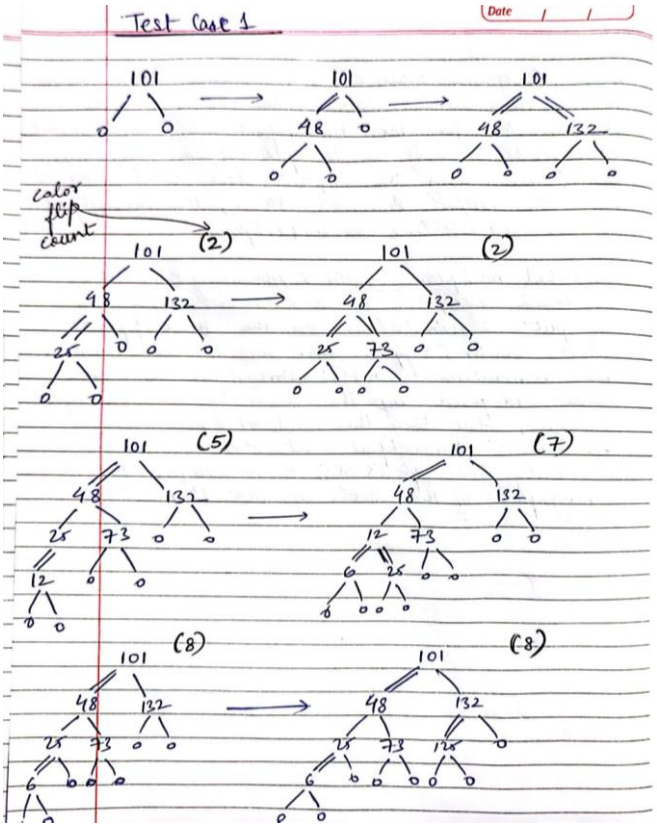
find_closest_book(self, node, target_id)

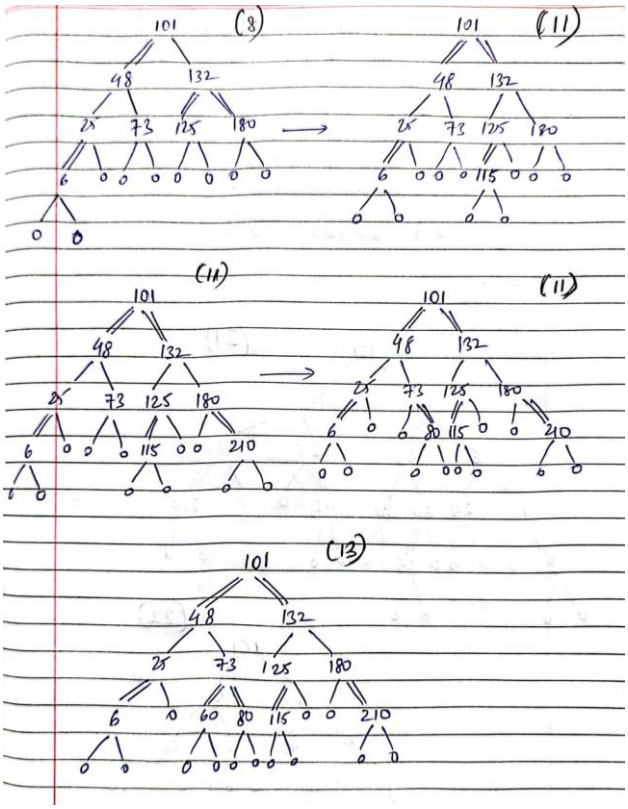
- Description: Finds the closest book(s) to a specified book ID.
- Working: Traverses the Red-Black Tree to find the book(s) closest to the provided ID. Returns the details of the closest book(s) found.
- Time Complexity: Finds the closest book ID in the Red-Black Tree, with a time complexity of O(log N) for tree traversal.

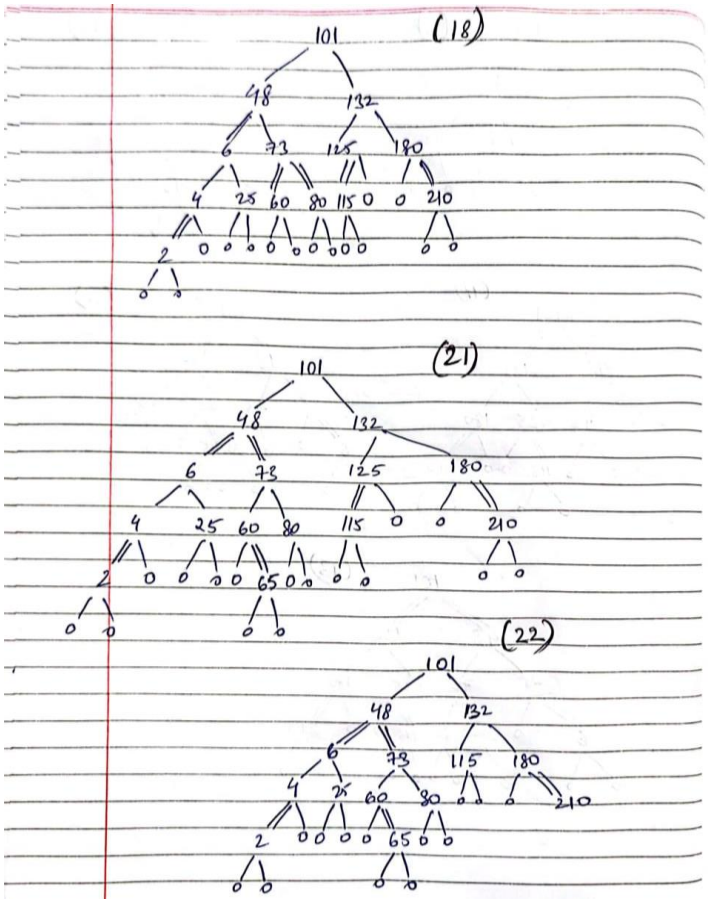
get_book_details(self, node)

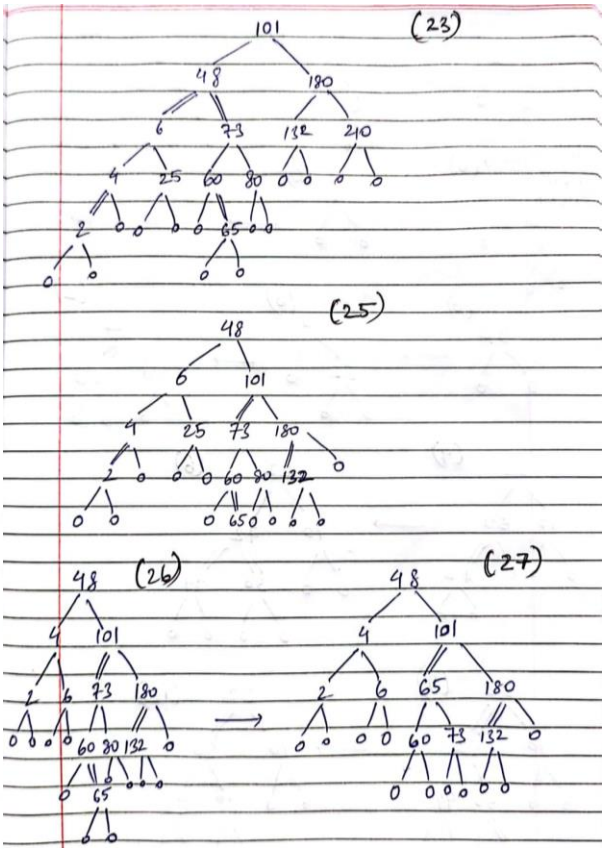
- Description: Retrieves details of a book node
- Working: Accesses the attributes of the provided node to gather book details like ID, title, author, availability, borrower, and reservation heap.
- Time Complexity: O(1)- Retrieves details of a book node, performing a constant number of attribute accesses and list operations.

ADS TestCases corrected (Test Case-1 indicating color flips)









ADS TestCases corrected (Test Case-2 indicating color flips)