COP 5536 – Advanced Data Structures

PROJECT ASSIGNMENT

Tarun Reddy Nimmala
UFID: 53126269
tarunred.nimmala@ufl.edu

Overview:

The gatorLibrary project is a sophisticated library management system developed in Python. It offers an array of functionalities for efficiently managing its books, patrons, and borrowing operations. The system is structured into multiple modules, each responsible for distinct aspects of the library's operations.

A Red-Black tree data structure is employed by the system to guarantee effective book management. Using Binary Min-heaps as a data structure, a priority-queue method can be implemented to manage book reservations if a book is not currently available for borrowing. Every book will have a min-heap to record the reservations that readers have made.

Execution:

1. **Python Installation:**
    - Ensure you have Python installed on your system.
2. **Download the Source Code:**
    - Obtain the ***gatorLibrary*** source code files.
3. **Navigate to the Directory:**
    - Open a terminal or command prompt.
    - Use the `cd` command to navigate to the directory where the `gatorLibrary` files are located.
4. **Input File Preparation:**
    - Prepare an input file (`input_file.txt`) that contains the commands and data to perform library operations.

5. **Execute the Script:**

    - Run the `gatorLibrary.py` script by typing the following command in the terminal:

```
tarunreddy@Taruns-MacBook-Pro ~ % python3 gatorLibrary.py test1.txt
```

    - Replace `test1.txt` with the actual name of your input file and python3 with your python version.

6. **Operation Execution:**

    - The script will interpret the commands from the input file and execute the corresponding library operations.
    - Operations include inserting books, deleting books, searching for books, borrowing/returning books, and more.

7. **Output File Generation:**

    - The script will generate an output text file containing the results of the executed operations in the same directory.

8. **Review Results:**

    - Open the generated output file to review the results of the performed operations. This file will contain information regarding the success or details of each operation.
    - If the directory contains the input files with the same name, output file is overwritten based on the input file you are running.

Sample Input:

test1.txt

```
InsertBook(1, "Book1", "Author1", "Yes")
PrintBook(1)
BorrowBook(101, 1, 1)
InsertBook(2, "Book2", "Author2", "Yes")
BorrowBook(102, 1, 2)
PrintBooks(1, 2)
ReturnBook(101, 1)
Quit()
```

Sample Output:

test1_output_file.txt

```
BookID = 1
Title = Book1
Author = Author1
Availability = Yes
BorrowedBy = None
Reservations = []

Book 1 Borrowed by Patron 101

Book 1 Reserved by Patron 102

BookID = 1
Title = Book1
Author = Author1
Availability = No
BorrowedBy = 101
Reservations = [102]

BookID = 2
Title = Book2
Author = Author2
Availability = Yes
BorrowedBy = None
Reservations = []

Book 1 Returned by Patron 101

Book 1 Alloted to Patron 102

Program Terminated!!

```

Operations Supported:

- **Insertion:** Adding books to the library.
- **Deletion:** Removing books and managing reservations.
- **Search:** Finding books by their ID.
- **Range Search:** Retrieving books within a specified range of IDs.
- **Closest Book:** Finding the closest book to a given bookID.
- **Borrowing & Returning:** Handling book borrowing and returning by patrons.
- **Color Flip Count:** Tracking color flips in the Red-Black Tree.

Function Prototypes:

a) gatorLibrary.py

gatorLibrary.py > ...

```python
from helper import helper_main
import sys

if __name__ == '__main__':
    input_file = sys.argv[1]
    input_file_name, extension = input_file.split('.')
    if extension == 'txt':
        output_file_name = input_file_name + '_output_file.'+ 'txt'
        helper_main(input_file, output_file_name)
    else:
        print("File not found or path is incorrect.")
```

- This is the main driver program. It takes the input file name from command line argument, processes it if it is valid .txt file, and calls the helper module to perform the library operations.

b) helper.py

It also initializes the required data structures and output file.

1. helper_main ()

```
def helper_main(input_file_name, output_file_name):
    # Helper Main function to Create the RedBlackTree and output_file object
    global library #declaring library object as global
    library = RedBlackTree()

    global output_file #declaring output_file as global
    output_file = open(output_file_name, 'w')

    try:
        with open(input_file_name, "r") as file: #opening the file in read only mode
            for line in file:
                operation = line.split('(')[0]  # Extract operation before '('
                operation=operation.lstrip() #removing white spaces at the start of the line
                if operation == 'Quit':
                    write_to_file('Program Terminated!!')
                    break
                elif operation == 'ColorFlipCount':
                    colorCount = library.get_colorFlipCount()
                    write_to_file('Color Flip Count: {}\n'.format(colorCount))
                else:
                    info = line.split('(')[1].split(')')[0]  # Extracting information within '(' and before ')'
                    info_list = []
                    for elem in info.split(','):
                        elem = elem.strip().strip('"') # Split info into list removing spaces and quotes
                        # Trying to convert to integer if not enclosed in quotes
                        try:
                            elem = int(elem)
                        except ValueError:
                            pass  # It remains as a string if it can't be converted to an integer
                        info_list.append(elem)
                    operations(operation, info_list)

    except FileNotFoundError:
        print("File not found or path is incorrect.")

    output_file.close()
```

- The helper module contains the main helper method that processes each line of input, extracts operation and info, calls appropriate operation functions, and handles output.

2. operations ()

```
def operations(operation, info_list):
    # Function to call various library operations
    if operation == 'InsertBook':
        library.insert(BookNode(info_list[0],info_list[1],info_list[2],info_list[3]))

    elif operation == 'DeleteBook':
        reserveList = library.delete(info_list[0])
        printDeleteBook(info_list[0],reserveList)

    elif operation == 'PrintBook':
        bookData = library.search(info_list[0])
        printBook(bookData,info_list[0])

    elif operation == 'PrintBooks':
        range_book_nodes = library.range_books(info_list[0],info_list[1])
        if (len(range_book_nodes)==0):
            write_to_file('No book is in the range ({}, {})\n'.format(info_list[0],info_list[1]))

        else:
            for bookNode in range_book_nodes:
                printBook(bookNode)

    elif operation == 'FindClosestBook':
        closest_smaller_node, closest_larger_node = library.closest_keys_with_min_difference(info_list[0])
        printClosestBooks(info_list[0],closest_smaller_node, closest_larger_node)

    elif operation == 'BorrowBook':
        book_id, reserve_flag, patron_id = library.BorrowBook(info_list[0],info_list[1],info_list[2])
        printBorrowBook(book_id, reserve_flag, patron_id)

    elif operation == 'ReturnBook':
        book_id, nextPatron = library.ReturnBook(info_list[0],info_list[1])
        currPatron = info_list[0]
        printReturnBook(book_id, currPatron, nextPatron)
```

- The operations () method in helper.py is the method that takes in the operation name and information and calls the appropriate library function to perform that operation.
- It takes in two parameters:

1. **operation**: the name of the operation to perform, extracted from the input line.
2. **info_list**: a list containing the relevant info extracted from input line.

3. write_to_file ()

```
def write_to_file(content):
    output_file.write(content+'\n') #Writing the content into output file
```

- The write_to_file () method in helper.py is a simple helper function to write output to the result file.

c) redBlackTree.py

This module contains the Red Black Tree class with methods for all the library operations.

1. BookNode Constructor

```
class BookNode:
    # Class BookNode for storing the info about the book.
    def __init__(self, bookID, title, author, availability='Yes', borrowedBy=None):
        self.bookID = bookID
        self.title = title
        self.author = author
        self.availability = availability
        self.borrowedBy = borrowedBy
        self.reservationHeap = MinHeap()  # Binary min-heap object for reservations
        self.reservations = [] # PatronID waitlist for the book ordered by the patron's priority
```

- It is used within the Node class of the RedBlackTree to store book information.
- It contains a MinHeap object to manage reservations for the book.
- The reservations list stores a waitlist of patrons wanting to borrow book.

2. Node Constructor

```
class Node:
    # Each Node represents a book in the Red Black Tree
    def __init__(self, bookNode, color='red'): #Assigning the Red color for a newly created Node
        self.book = bookNode #bookNode object
        self.key = bookNode.bookID
        self.left = None
        self.right = None
        self.parent = None
        self.color = color
```

- The Node class represents each node in the RedBlackTree and contains a BookNode object to hold book information.

3. RedBlackTree Constructor

```
class RedBlackTree:
    def __init__(self): #RedBlackTree Constructor
        self.root = None # Initialize the root of the Red-Black Tree
        self.colorDic={} #Dictionary to hold the Key, Color pairs
        self.colorFlipCount=0
```

- It maintains a root node pointer to root of tree, dictionary to hold the key-color pairs and color flip count.

4. insert

```
def insert(self, bookNode):
    new_node = Node(bookNode)
    new_key = new_node.key
    # If the tree is empty, new node is the root
    if not self.root:
        self.root = new_node
        self.root.color = 'black' # color of the root should be black
    else:
        current = self.root
        parent = None
        # Traversing the tree to find the appropriate position for the new node
        while current:
            parent = current
            if new_key < current.key:
                current = current.left
            else:
                current = current.right
        # Setting the parent (variable) key: Any djusting left/right child accordingly
        new_node.parent = pa
        if new_key < parent.key:
            parent.left = new_node
        else:
            parent.right = new_node
        self.insert_fix(new_node) # Fixing any violations of Red-Black Tree properties after insertion

    self.colorFlipCount_update() # Updating color flip count after insertion
```

- It first creates a new Node and finds appropriate spot by traversal.
- Links it with the parent and calls insert_fix() to balance properties.
- insert_fix() handles property violations by doing rotations via rotate_left() and rotate_right()
- Updates color flip count after insertion.

5. search

```
def search(self, bookID):
    # Searching for the key i.e bookID using binary search tree traversal technique
    current = self.root
    while current and current.key != bookID:
        if bookID < current.key:
            current = current.left
        else:
            current = current.right
    return current  # Returns the node if found, None otherwise
```

- It performs a standard binary search tree search operation.
- Starts from root and traverses down. The complexity is O(h) where ‘h’ is height of the tree.

6. delete

```
def delete(self, key):
    # Function to delete a node with the given key from the Red-Black Tree
    node_to_delete = self.search(key)
    if node_to_delete is None:
        return None # If the node to delete is not found, return None
    # Handling scenarios for deletion based on the type of node and its children
    if node_to_delete == self.root and not node_to_delete.left and not node_to_delete.right:
        # Case: Deleting the root when it has no children
        self.root = None
    else:
        self.delete_node(node_to_delete) # Delete the found node

    self.colorFlipCount_update() # Update the color flip count after deletion

    reserveList = node_to_delete.book.reservations
    return reserveList # Return the list of reservations (if any) for the deleted node
```

- Deletes a node with the given `key` from the Red-Black Tree.
- It begins by searching for the node with the specified `key` using the `search` method.
- After deletion, it updates the color flip count and returns the list of reservations for the deleted node.
- delete_node() determines the node to be removed and its potential child for reassignment.
- delete_fix() performs restructuring and recoloring to ensure the Red-Black Tree properties are maintained.
- predecessor() is used during deletion to find the appropriate node for replacement when a node has both left and right children.

7. inorder_traversal

```
def inorder_traversal(self):
    result = []
    self._inorder(self.root, result) # Starting the inorder traversal from the root node
    return result

def _inorder(self, node, result):
    if node:
        self._inorder(node.left, result)
        result.append(node) #Appending the Red Black Tree nodes into the result List
        self._inorder(node.right, result)
```

- Retrieves a list of nodes from the Red-Black Tree in ascending order based on their keys.
- _inorder() helper method takes a node as input and recursively traverses its left subtree, then the node itself, and finally its right subtree.

8. range_books

```
def range_books(self, start_key, end_key):
#Using the inorder traversal to find the books in the given range of BookID's
    range_book_nodes =[]
    inorder = self.inorder_traversal()
    for node in inorder:
        if(node.key>=start_key and node.key<=end_key):
            range_book_nodes.append(node)

    return range_book_nodes
```

- Retrieves a list of book nodes whose keys fall within the specified range.
- By performing an inorder traversal, the method ensures that nodes are retrieved in ascending order of their keys.
- During the traversal, it selectively collects nodes whose keys lie within the specified range by comparing each node's key to the start_key and end_key.

9. closest_keys_with_min_difference

```
def closest_keys_with_min_difference(self, target_key):
#Using the inorder traversal to find the closest bookID for a given bookID
    inorder = self.inorder_traversal()

    closest_smaller = float('-inf') #Assigning minus infinity to closest_smaller
    closest_larger = float('inf') #Assigning infinity to closest_larger

    closest_smaller_node = None
    closest_larger_node = None

    for node in inorder:
        if node.key == target_key: #If the bookID is present in the Red Black Tree, return the Node
            return node, None
        elif node.key < target_key:
            #Getting the bookID with maximum value from all the bookID's smaller than the given bookID
            if closest_smaller != max(closest_smaller, node.key):
                closest_smaller = max(closest_smaller, node.key)
                closest_smaller_node = node
        elif node.key > target_key:
            #Getting the bookID with minimum value from all the bookID's greater than the given bookID
            if closest_larger != min(closest_larger, node.key):
                closest_larger = min(closest_larger, node.key)
                closest_larger_node = node

    if closest_smaller == float('-inf'):
        closest_smaller_node = None # Assigning None value if their is no bookID smaller than the given bookID
    if closest_larger == float('inf'):
        closest_larger_node = None # Assigning None value if their is no bookID greater than the given bookID

    return closest_smaller_node, closest_larger_node
```

- The method uses inorder traversal to get a sorted list of nodes, ensuring that nodes are retrieved in ascending order of their keys.
- While traversing the sorted list, it compares each node's key to the target_key and updates the closest smaller and larger keys accordingly.

10. BorrowBook

```
def BorrowBook(self, patron_id, book_id,priority):
# Function to assign book to the Patron and updating the reservation Heap
    bookNode = self.search(book_id) #Searching whether the book exists in the Library
    reserveHeap = bookNode.book.reservationHeap
    if bookNode == None:
        return None, False, patron_id
    else:
        book_availability = bookNode.book.availability
        book_availability = book_availability.lower()
        if (book_availability == 'no'):
            # Inserting the Patron into Reservation Heap as the book is not available
            timestamp = time.time() # Using timestamp for reservation if the priority is same
            reservation = (priority, timestamp, patron_id)
            reserveHeap.insert(reservation)
            traverseList = reserveHeap.traverse_heap()
            # Assiging the List of only PatronID's to the reservationList on the basis of Priority
            bookNode.book.reservations = traverseList
            #bookNode.book.borrowedBy = patron_id
            return book_id, True, patron_id
        else:
            # Assigning the book to the Patron as the book is available
            bookNode.book.borrowedBy = patron_id
            bookNode.book.availability = 'No'
            traverseList = reserveHeap.traverse_heap()
            bookNode.book.reservations = traverseList
            return book_id, False, patron_id
```

- The BorrowBook method handles the process of borrowing a book by a patron, updating the book's availability and reservation heap.
- Searches for the book_id within the Red-Black Tree using the search method to locate the corresponding book node.
- If the book is available, it updates the book's status and assigns it to the borrowing patron.
- If the book is not available, it adds the patron to the reservation list with their priority and timestamp.
- Manages the reservation heap for the book, inserting new reservations based on priority and timestamp.

11. ReturnBook

```
def ReturnBook(self, patron_id, book_id):
# Function to return book by the Patron and assigning the book to next Patron in reservation Heap
    bookNode = self.search(book_id)
    reserveHeap = bookNode.book.reservationHeap
    nextReservation = reserveHeap.extract_min() # Calling the extract_min method in MinHeap to assign the book to next Patron
    bookNode.book.reservations = reserveHeap.traverse_heap()
    if nextReservation is None:
        bookNode.book.borrowedBy = None
        bookNode.book.availability = 'Yes' # If their are no reservations then make the book available
        return book_id, None
    else:
        bookNode.book.borrowedBy = nextReservation[2]

    return book_id, nextReservation[2]
```

- Handles the return of a book by a patron and assigns it to the next patron in the reservation list.
- If there's a next reservation, assigns the book to the next patron and updates the book's status accordingly.
- If there are no further reservations, updates the book's status to available for borrowing.

12. colorFlipCount_update

```
def colorFlip_inorder_traversal(self):
    # Inorder traversal of the red Black Tree to create a dictionary with the latest key, color pairs
    dic = {}
    self.colorFlip_inorder(self.root, dic)
    return dic

def colorFlip_inorder(self, node, dic):
    if node:
        self.colorFlip_inorder(node.left, dic)
        dic[node.key]=node.color # Updating the dictionary with key, color pairs after insert or delete for comparision
        self.colorFlip_inorder(node.right, dic)

def colorFlipCount_update(self):
# Function to update the color flip count
    comp_dic1 = self.colorDic # Dictionary before insert ot delete
    comp_dic2 = self.colorFlip_inorder_traversal() # Dictionary after insert ot delete
    for key in comp_dic2:
        if key in comp_dic1:
            if(comp_dic1[key] != comp_dic2[key]):
                self.colorFlipCount += 1 # If the color is different after the update or delete we increment the color flip co
    self.colorDic = comp_dic2

def get_colorFlipCount(self):
    # Getter for color flip count
    return self.colorFlipCount
```

- The colorFlipCount_update method in the Red-Black Tree class is responsible for updating the count of color flips that occur during insertions or deletions in the tree.
- Compares the color configurations before and after insertions or deletions to track changes in node colors.
- Utilizes the colorFlip_inorder_traversal method to obtain a dictionary with the latest key-color pairs after insertions or deletions.
- Compares the colors in the dictionaries before and after changes and increments the **colorFlipCount** for each color change.

d) minHeap.py

1. MinHeap Constructor

```
class MinHeap:
    def __init__(self):
        # Initialize the heap as an empty list
        self.heap = []
```

- Initializes the heap as an empty list.
- Sets up the initial state of the MinHeap.

2. parent

```
    def parent(self, i):
        # Calculate the index of the parent node
        return (i - 1) // 2
```

- Calculates the index of the parent node for a given index ‘i’.
- Helps in locating the parent node of a specified node in the heap.

3. left_child

```
    def left_child(self, i):
        # Calculate the index of the left child node
        return 2 * i + 1
```

- Calculates the index of the left child node for a given index ‘i’.
- Helps in locating the left child node of a specified node in the heap.

4. right_child

```
def right_child(self, i):
    # Calculate the index of the right child node
    return 2 * i + 2
```

- Calculates the index of the right child node for a given index ‘i’. Helps in locating the right child node of a specified node in the heap.

5. swap

```
def swap(self, i, j):
    # Swap elements at indices i and j in the heap
    self.heap[i], self.heap[j] = self.heap[j], self.heap[i]
```

- Swaps elements at indices i and j in the heap. Used to interchange elements during heap operations, maintaining the heap structure.

6. insert

```
def insert(self, reservation): # priority = reservation[0], timestamp = reservation[1]

    # Add a reservation to the heap
    self.heap.append(reservation)
    index = len(self.heap) - 1
    #Maintaining the heap property by swapping elements if necessary
    while index > 0 and self.heap[self.parent(index)] > self.heap[index]:
        self.swap(index, self.parent(index))
        index = self.parent(index)
```

- Adds a reservation to the heap while maintaining the heap property.
- Inserts a new element (reservation) into the heap and ensures that the heap remains a valid MinHeap by performing necessary swaps to satisfy the heap property.

7. heapify

```
def heapify(self, i):
    # Heapify the subtree rooted at index i
    left = self.left_child(i)
    right = self.right_child(i)
    smallest = i
    # Check if left child exists and compare with smallest
    if left < len(self.heap) and self.heap[left] < self.heap[smallest]:
        smallest = left
    # Check if right child exists and compare with smallest
    if right < len(self.heap) and self.heap[right] < self.heap[smallest]:
        smallest = right
    # If smallest is not the current root, swap and continue heapifying
    if smallest != i:
        self.swap(i, smallest)
        self.heapify(smallest)
```

- Adjusts the heap rooted at index i to maintain the heap property.
- Helps to restore the heap property by checking and potentially swapping elements to ensure the smallest element is at the root of the subtree rooted at index i.

8. extract_min

```
def extract_min(self):
    # Extracting the minimum element from the heap
    if len(self.heap) == 0:
        return None
    if len(self.heap) == 1:
        return self.heap.pop()

    root = self.heap[0]
    self.heap[0] = self.heap.pop()
    self.heapify(0)

    return root
```

- Extracts and returns the minimum element from the heap.
- Removes and returns the root element (minimum element) from the heap while maintaining the heap structure by reorganizing elements after removal.

9. get_min

```
def get_min(self):
    # Get the minimum element without removing it from the heap
    return self.heap[0] if self.heap else None
```

- Retrieves the minimum element without removing it from the heap.
- Allows the user to see the minimum element without altering the heap structure.

10. traverse_heap

```
def traverse_heap(self):
    #Return the patronID from reservations in the heap
    return [item[2] for item in self.heap] # patronID=item[2]
```

- Returns the patronID from reservations in the heap.
- Collects and returns the patronID information from the reservations stored in the heap, providing a list of patronIDs present in the heap.

Summary:

The gatorLibrary project offers a robust and versatile library management system, leveraging the efficiency of Red-Black Tree data structure and Binary Min Heap for managing the book reservations. The tree's balanced nature allows for quick searches, aiding in finding books by their unique identifiers (IDs). This is crucial in library environments where swift book retrieval is essential. The Min Heap structure allows for constant-time extraction of the patron with the highest priority, ensuring swift allocation of books to waiting patrons.

This combination of Red-Black Trees for library inventory management and Binary Min Heaps for reservations offer speed, efficiency, and adaptability, catering to the diverse needs of a library environment. With their balanced nature and efficient operations, they ensure optimal resource utilization, smooth patron interactions, and streamlined administrative tasks, making the gatorLibrary project an asset for efficient library management.