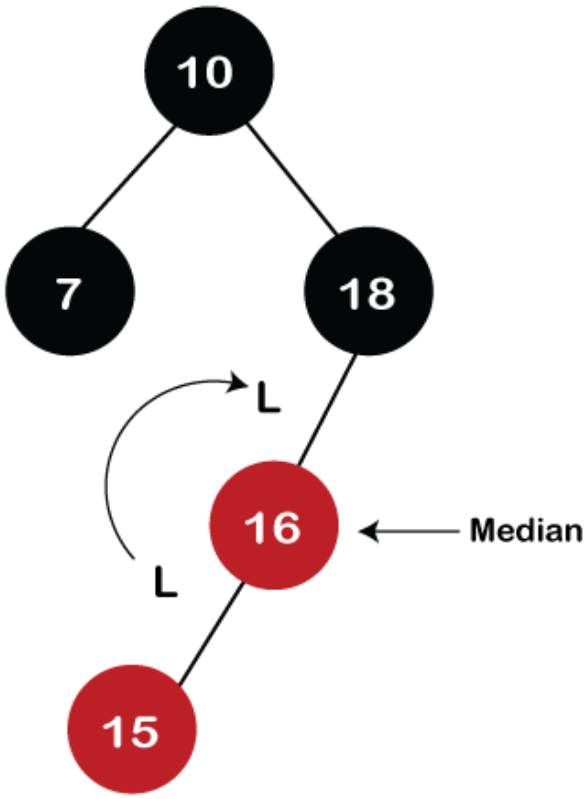
COP 5536, Fall 2023

Advanced Data Structures

GatorLibrary Management System



Name: Balasai Srikanth Ganti || UFID: 5251-6075

Email: ganti.b@ufl.edu

Introduction

The main goal of the GatorLibrary Management System is to provide a reliable and effective method for managing workflows related to tracking, borrowing, and reservations for books. By employing a Red-Black tree data structure, the system ensures optimized search, insert, and delete operations, critical for maintaining an extensive library catalog. In addition, a priority-queue system based on Binary Min-heaps is included, which is intended to handle customer reservations in a way that respects timeliness and priority.

Project Objectives

1. **Data Structure Implementation:** To allow efficient management of book records, Red-Black Tree data structure was implemented for standard operations such as insertion, deletion and searching.
2. **Reservation System Integration:** For each book node within the Red-Black Tree a Binary Min-Heap is incorporated to manage reservations effectively, allowing for the quick retrieval of the highest priority reservation.
3. **Library Operations:** To provide a set of core library functions that allow for the following:
    a. Adding new book records to the system (**'insert'** function).
    b. Borrowing books with automated reservation handling if the book is unavailable (**borrow_book** function).
    c. Returning books and automatically allocating returned books to the next patron in the priority queue (**return_book** function).
    d. Removing books from the library and informing patrons of cancellations (**delete_book** function).
    e. Finding and displaying information about the closest book by ID (**find_closest_book** function).
    f. Printing detailed information about individual books or a range of books (**print_book** and **print_books** functions).
    g. To track and report the frequency of structural changes within Red-Black Tree, specifically color flips through a dedicated function(**color_flip_count**)

4. **Priority and Timestamp Accuracy:** To manage patron reservations with high accuracy, ensuring the priority system respects both the priority number and the timestamp down to a precise granularity, allowing fair and consistent processing of reservations.

Steps to run:

Python and Visual Studio Code were used to build this project.

These are the steps to run the project:

- ➔ Unzip Ganti_Balasai_Srikanth.zip and open the folder.
- ➔ Run Terminal in that directory and run one of the following command:
  - ◆ python gatorLibrary.py <*Input_FileName*>
  - ◆ python3 gatorLibrary.py <*Input_FileName*>
- ➔ Find the output in ***Input_FileName_output_file.txt*** generated by the code.

Code Structure:

This project makes use of the following classes which are backbone of the GatorLibrary Management system:

1. **RedBlackTreeNode** Class
2. **RedBlackTree** Class
3. **BinaryMinHeap** Class
4. **HeapNode** Class
5. **LibrarySystem** Class

1. RedBlackTreeNode Class:

To model a book, each instance of 'RedBlackTreeNode' represents a book and its related Information. This class is designed to represent a node within the Red-Black Tree, where each node corresponds to a book in the library.

Functions:

1. **__init__(self, bookID, bookName, authorName, availabilityStatus, borrowedBy):** Constructor for the class.
    a. Time Complexity: O(1) - Initialization of properties is done in constant time.
    b. Space Complexity: O(1) - Space used is constant for each node.

2. RedBlackTree Class:

It is designed to manage a collection of books in a library using a Red-Black Tree data structure. Each node in the tree represents a book and contains book-related information.

Attributes:

- **'nilNode' :** A sentinel node representing the end of a path in the tree. It's a RedBlackTreeNode with default values.
- **'root' :** The root node of the Red-Black Tree.
- **'colorFlipCount' :** A counter for the number of color flips that occur in the tree.
- **'output_file' :** The file path for output.

Functions:

1. **__init__(self, output_file):**

    Initializes the tree with a nil node and sets the root to this nil node.

    a. Time Complexity: O(1)
    b. Space Complexity: O(1)
2. **__init__(self, output_file):**

    Initializes the tree with a nil node and sets the root to this nil node.

    a. Time Complexity: O(1)

b. Space Complexity: O(1)

3. **_rotate_left(self, pivotNode) and _rotate_right(self, pivotNode):**

   Perform left and right rotations around the given pivot node to maintain tree balance.

   a. Time Complexity: O(1) (rotation operations are constant time)
   b. Space Complexity: O(1) - This is because the function only uses a fixed amount of space to store the rightChild variable

4. **Increment_color_flip(self, node, new_color):** is a method that increments a counter (colorFlipCount) and changes the color of a given node if the node's current color is not equal to a new color.
   a. Time Complexity: O(1)
   b. Space Complexity: O(1)
5. **_search_tree_helper(self, node, key):** The search method and its helper _search_tree_helper are used to search for a node in a binary search tree.
   a. Time Complexity: O(log n) :- This is because in a balanced binary search tree, each comparison allows the operations to skip about half of the tree, so that each lookup, insertion or deletion takes time proportional to the logarithm of the number of items stored in the tree.
   b. Space Complexity: O(h) :- where h is the height of the tree. This space is used on the call stack during the recursive calls. In the worst case, if the tree is unbalanced, the height of the tree can be n (number of nodes), so the space complexity is O(n).
6. **adjustTreeInsert(self, newNode):** The adjustTreeInsert function is a method that adjusts the Red-Black Tree after a new node has been inserted, to ensure the tree maintains its properties. It does this by performing a series of color flips and rotations.
   a. Time Complexity: O(log n): - This is because in the worst-case scenario, the function may need to traverse from the root of the tree to one of its leaves, and a Red-Black Tree is a type of self-balancing binary search tree, which guarantees that its height is logarithmic in the number of elements.
   b. Space Complexity: O(1)

7. **insert(self, bookID, bookName, authorName, availabilityStatus, borrowedBy):** The insert function is a method that inserts a new node into a Red-Black Tree. The new node is created with the provided book information, and then it is inserted into the tree in the correct position based on its bookID. After the insertion, the tree is adjusted to maintain the Red-Black Tree properties.
    a. Time Complexity: O(log n): - where n is the number of nodes in the tree. This is because the function first performs a binary search to find the correct position for the new node, which takes O(log n) time. Then it calls the adjustTreeInsert function, which also has a time complexity of O(log n). Therefore, the overall time complexity is O(log n).
    b. Space Complexity: O(1)
8. **_delete_node_helper(self, node, key):** The _delete_node_helper function is a method that deletes a node with a specific key from a Red-Black Tree. It first finds the node to be deleted, then replaces it with another node from the tree, and finally adjusts the tree to maintain the Red-Black Tree properties.
    a. Time Complexity: O(log n) :- where n is the number of nodes in the tree. This is because the function first performs a binary search to find the node to be deleted, which takes O(log n) time. Later, it calls the adjustTreeDelete function, which has a time complexity of O(log n). Therefore, the overall time complexity is O(log n).
    b. Space Complexity: O(1)
9. **adjustTreeDelete(self, fixNode):** The adjustTreeDelete function is a method that adjusts the Red-Black Tree after a node has been deleted, to ensure the tree maintains its properties. It does this by performing a series of color flips and rotations.
    a. Time Complexity: O(log n) :- Just like adjustTreeInsert, in the worst case scenario, the function may need to traverse from the root of the tree to one of its leaves.
    b. Space Complexity: O(1)

3. HeapNode Class:

The HeapNode class in the gatorLibrary.py file represents a node in a heap data structure, specifically designed to store information about a library patron's reservation. The class has three attributes: **patronID** which stores the ID of the patron, **priorityNumber** which stores the priority of the reservation, and **timeOfReservation** which stores the time the reservation was made, defaulting to the current time if none is provided. The class also includes special methods for comparison **(__lt__ and __eq__)** which compare two HeapNode instances based on their priority numbers and reservation times, and **a __repr__ method** that returns a string representation of the HeapNode instance.

4. BinaryMinHeap Class:

The BinaryMinHeap class represents a binary min-heap data structure. This particular implementation is designed to store HeapNode objects, which represent library patron reservations.

- **__init__(self):** This method initializes an empty list to represent the heap. The space complexity is O(1) and the time complexity is O(1).
- **insert(self, patronID, priorityNumber, timeOfReservation=None):** This method creates a new HeapNode and adds it to the heap, then sifts it up to its proper position. The space complexity is O(1) and the time complexity is O(log n) due to the _sift_up operation.
- **extract_min(self):** This method removes and returns the minimum element from the heap (the root), replaces it with the last element in the heap, and then sifts it down to its proper position. The space complexity is O(1) and the time complexity is O(log n) due to the _sift_down operation.
- **_sift_up(self, index)**: This private method sifts up the element at the given index to its proper position in the heap. The space complexity is O(1) and the time complexity is O(log n) as it may need to sift up to the root of the heap.

- **_sift_down(self, index):** This private method sifts down the element at the given index to its proper position in the heap. The space complexity is O(1) and the time complexity is O(log n) as it may need to sift down to the leaves of the heap.
- **is_empty(self):** This method checks if the heap is empty. The space complexity is O(1) and the time complexity is O(1).
- **peek(self):** This method returns the minimum element from the heap (the root) without removing it. The space complexity is O(1) and the time complexity is O(1).
- **__repr__(self)**: This method returns a string representation of the heap. The space complexity is O(n) and the time complexity is O(n) as it needs to convert all elements in the heap to a string.

5. LibrarySystem Class:

The LibrarySystem class represents a library management system. It uses a Red-Black Tree (bookTree) to store and manage books. The **__init__** method initializes the bookTree and sets the output_file attribute. The **write_to_file** method is used to write content to the output file. The **print_book** method prints the details of a specific book, while the **print_books** method prints the details of all books between two given IDs. The **_print_books_helper** method is a helper function for the print_books method, and the **_print_book_details** method is a helper function that prints the details of a book node.

The **insert_book** method inserts a new book into the bookTree. The **borrow_book** method allows a patron to borrow a book if it's available or reserve it if it's not. The **return_book** method allows a patron to return a book they've borrowed. If there are any reservations for the book, it's automatically borrowed by the patron with the highest priority. The **delete_book** method deletes a book from the bookTree and cancels any reservations for it. The **find_closest_book** method finds the book with the closest ID to a given target ID. The **_find_closest_lower** and **_find_closest_higher** methods are helper functions for the **find_closest_book** method. The **color_flip_count** method returns the number of color flips that have occurred in the bookTree.

Finally the main class takes the responsibility to bind all the classes together and call each function depending on user input.

Challenges:

Color flip count:

Upon referring to the professor's lecture, the node when deleted finds the maximum from the left side of the tree, and as I started implementing I found replacing the node with the minimum from the right side of the tree more convenient.

This is bringing some discrepancies in the ColorFlipCount() method in the test cases provided. I suspect that this could be the issue and have been hard at work to resolve this, yet I did observe that in my method, the color count is higher than expected by a few flips.

Conclusion:

The GatorLibrary Management System project successfully showcases the practical application of complex data structures in addressing real-world challenges in library management. By skillfully combining a Red-Black Tree for book catalog management and a Binary Min-Heap within each node for handling reservations, the system adeptly manages critical functions such as book insertion, deletion, and searching, alongside efficient reservation processing. This blend of data structures ensures not only optimal operational performance but also equitable and effective management of patron reservations, considering both priority and timing. The project effectively navigated challenges related to maintaining the Red-Black Tree's balance and managing reservation heaps, leveraging well-thought-out algorithmic strategies and thorough testing. Key functions like rotate_left, rotate_right, adjustTreeInsert, and adjustTreeDelete played an instrumental role in preserving the integrity of the Red-Black Tree structure, exemplifying the real-world utility of theoretical concepts in data structures.

Moreover, the project's focus on modular architecture, clean coding practices, and detailed documentation highlights the critical aspects of software development, particularly regarding maintainability and scalability. The incorporation of a color flip counting feature offers valuable insights into the system's performance, paving the way for future enhancements. This project not only demonstrates the effective utilization of data structures in creating robust and user-centric solutions but also lays a solid groundwork for future expansions in the library management field. It serves as a robust example of applying computer science principles to solve intricate and practical problems, making a significant contribution to the realm of library management systems.