COP5536: Advanced Data Structures

Project - Gator Ticket Master

Name: Sai Pande

UFID: 37696687

Email Id: saipande@ufl.edu

University of Florida

Computer Science and Information Science Engineering

Introduction:

The Gator Ticket Master is a seating booking system designed to maximize seat allocation and reservation for Gator Events. This system uses advanced data structures such as Red_Black Tree and Min Heap for seat reservation, cancelation, and waiting list operations.

Problem Statements:

1. The challenge is to develop a software system that can handle following functions:
a. Seat Reservations
b. Cancellations
c. Waitlist Management
d. Dynamic Seat Additions

2. All the functions should also maintain efficient performance.

3. The system must prioritize seat assignments based on:
a. Availability
b. User Priority
c. Timestamp.

4. The system should handle various operations:
a. Initializing seats
b. Reserving seats
c. Canceling reservations
d. Updating user priorities
e. Releasing seats

Data Structures Used:

The system utilizes two primary data structures:

1. **Red-Black Tree:** A self-balancing binary search tree is used to handle reserved seat information. Each node has a user ID (key) and a seat ID. The Red-Black Tree guarantees O(log n) time complexity for operations like insertion, deletion, and searching.

2. **Min Heap:** To control the wait list, a priority queue was implemented. It prioritizes users based on their priority and date, ensuring that the highest-priority users are served first when seats become available.

Project Structure:

The system is implemented in Python and includes numerous classes:

1. Node refers to a node in the Red-Black Tree.
2. RedBlackTree implements the Red-Black Tree data structure.
3. MinHeap class uses Min Heap for queue management.
4. GatorTicketMaster is the main class that includes all features.

Working of the Program:

The program receives commands from an input file, processes them with the GatorTicketMaster class, and then outputs the results to an output file. It supports a variety of operations, including seat initialization, reservation, cancellation, addition, priority update, and seat release.

Programming Environment:

The Gator Ticket Master system is implemented in Python.

How to Interact with the Program:

The program is executed from the command line with the input file name as an argument:
$python3 gatorTicketMaster.py <input_filename>

Example: $python3 gatorTicketMaster.py input.txt

The program reads commands from the input file and writes results to an output file named "<input_filename>_output_file.txt".

Example: input_output_file.txt

There is a Makefile in the same directory which can be used to execute the program.
To run the program, you need to put the following command in the command prompt -> $make run INPUT_FILE={input_file_name.txt}

Function Prototypes and Explanations:

Function Prototype and Program Structure:

```
class Node:
    def __init__(self, user_id: int, seat_id: int)

class RedBlackTree:
    def __init__(self)
    def search(self, user_id: int)
    def insert(self, user_id: int, seat_id: int)
    def delete(self, user_id: int)
    def _correct_insert_node(self, new_node)
    def _correct_delete_node(self, fix_node)
    def inorder_traversal(self)
   def _right_rotation(self, gp_node)
   def _left_rotation(self, gp_node)
   def _inorder_helper(self, node, result)
   def _transplantation(self, old_node, replace_node)
   def _find_minimum_node(self, node)

class MinHeap:
    def __init__(self)
    def insert(self, priority, user_id)
    def _compare(self, node1, node2)
    def extract_min(self)
    def remove(self, user_id)
    def update_priority(self, user_id, new_priority)
    def _heapify_up(self, index)
    def _heapify_down(self, index)
```

```
class GatorTicketMaster:
    def __init__(self)
    def initialize(self, seat_count: int) -> str
    def available(self) -> str
    def reserve(self, user_id: int, user_priority: int) -> str
    def cancel(self, seat_id: int, user_id: int) -> str
    def add_seats(self, count: int) -> str
    def print_reservations(self) -> str
    def exit_waitlist(self, user_id: int) -> str
    def update_priority(self, user_id: int, user_priority: int) -> str
    def release_seats(self, user_id1: int, user_id2: int) -> str

def process_commands(input_file: str, output_file: str)
```

GatorTicketMaster Class Functions

1. **__init__(self)**
   Initializes the GatorTicketMaster with a Red-Black Tree for reserved seats, a MinHeap for the waitlist, and lists for available seats and seat count.

2. **initialize(self, seat_count: int)**
   Initializes the ticket system with the given number of seats.
   It sets up the available seats list and returns a confirmation message.

3. **available(self)**
   Returns a string indicating the number of available seats and the size of the waitlist.

4. **reserve(self, user_id: int, user_priority: int)**
   Reserves a seat for the given user_id if available, otherwise adds the user to the waitlist with the given priority.

5. **cancel(self, seat_id: int, user_id: int)**
   Cancels the reservation for the given user_id and seat_id. If successful, it assigns the seat to the next user in the waitlist or adds it back to available seats.

6. **add_seats(self, count: int)**
   Adds the specified number of new seats to the system. It processes the waitlist to assign new seats to waiting users if possible.

7. **print_reservations(self)**
   Returns a string representation of all current reservations, sorted by seat_id.

8. **exit_waitlist(self, user_id: int)**
   Removes the given user_id from the waitlist if present.

9. **update_priority(self, user_id: int, user_priority: int)**
   Updates the priority of the given user_id in the waitlist. If the user is not in the waitlist, it returns an appropriate message.

10. **release_seats(self, user_id1: int, user_id2: int)**
    Releases all reservations for users with IDs in the range [user_id1, user_id2]. It also removes these users from the waitlist if present and reassigns seats to waiting users.

Main Function

1. **process_commands(input_file: str, output_file: str)**
   Reads commands from the input file, processes them using the GatorTicketMaster instance, and writes the results to the output file. It handles various commands like Initialize, Reserve, Cancel, etc., and terminates when it encounters the Quit command.This implementation provides a comprehensive ticket reservation system with efficient data structures for managing reservations, waitlists, and seat availability.

RedBlackTree Class Functions:

1. **__init__(self)**
   Initializes the Red-Black Tree with a NIL (sentinel) node. The NIL node is black and serves as the leaf nodes of the tree. The root is initially set to NIL.

2. **search(self, user_id: int)**
   Searches for a node with the given user_id in the tree. It uses the helper function _search_helper to perform a recursive binary search.

3. **insert(self, user_id: int, seat_id: int)**
   Inserts a new node with the given user_id and seat_id into the Red-Black Tree. It performs a standard binary search tree insertion and then calls _fix_insert to maintain Red-Black Tree properties.

4. **delete(self, user_id: int)**
   Deletes the node with the given user_id from the tree. It uses the standard binary search tree deletion algorithm and then calls _fix_delete to maintain Red-Black Tree properties.

5. **_correct_insert_node(self, new_node)**
   A helper function called after insertion to restore Red-Black Tree properties. It performs color changes and rotations to ensure the tree remains balanced and adheres to Red-Black Tree rules.

6. **_correct_delete_node(self, fix_node)**
   A helper function called after deletion to restore Red-Black Tree properties. It performs color changes and rotations to ensure the tree remains balanced.

7. **inorder_traversal(self)**
   Returns an inorder traversal of the tree as a list of (seat_id, user_id) tuples. It uses the helper function _inorder_helper to perform the traversal recursively.

8. **_right_rotation(self, gp_node)**
   Performs a right rotation on the given node gp_node. This is a helper function used in balancing the Red-Black Tree during insertion and deletion operations.

9. **_left_rotation(self, gp_node)**
   Performs a left rotation on the given node gp_node. This is a helper function used in balancing the

Red-Black Tree during insertion and deletion operations.

10. **_inorder_helper(self, node, result)**
    A recursive helper function for inorder_traversal. It traverses the left subtree, appends the current node's data, then traverses the right subtree.

11. **_transplantion(self, old_node, replace_node)**
    A helper function used in the delete operation. It replaces the subtree rooted at old_node with the subtree rooted at replace_node.

12. **_find_minimum_node(self, node)**
    Finds the node with the minimum key in the subtree rooted at the given node. Used in the delete operation to find the successor of a node with two children.

MinHeap Class Functions

1. **__init__(self)**
   Initializes the MinHeap with an empty list for the heap, a counter for entries, and a dictionary to keep track of user indices.

2. **insert(self, priority, user_id)**
   Inserts a new entry with the given priority and user_id into the heap. It uses _heapify_up to maintain the heap property after insertion.

3. **_compare(self, node1, node2)**
   A helper function that compares two heap entries based on priority and entry time. It's used to determine the order of elements in the heap.

4. **extract_min(self)**
   Removes and returns the minimum element from the heap. It uses _heapify_down to maintain the heap property after extraction.

5. **remove(self, user_id)**
   Removes the entry with the given user_id from the heap. It uses _heapify_up or _heapify_down to maintain the heap property after removal.

6. **update_priority(self, user_id, new_priority)**
   Updates the priority of the entry with the given user_id. It uses _heapify_up or _heapify_down to maintain the heap property after the update.

7. **_heapify_up(self, index)**
   A helper function that moves an element up the heap to its correct position after insertion. It ensures that the heap property is maintained.

8. **_heapify_down(self, index)**
   A helper function that moves an element down the heap to its correct position after extraction or update. It ensures that the heap property is maintained.

Class Node Functions

1. **def __init__(self, user_id: int, seat_id: int)**
   initializes a new node with a user ID and seat ID, setting its color to red (1) and its left, right, and parent pointers to None

Conclusion:

The Gator Ticket Master system is an effective way to manage event seat reservations and waitlists. By combining the characteristics of Red-Black Trees and Min Heaps, it offers excellent performance for a variety of activities, making it ideal for large-scale events with dynamic seat allocations and user priorities.To demonstrate the system's efficiency, consider the time complexity of the key operations in Red-Black Trees and Min Heaps:

Red-Black Tree Time Complexities:

- Search: O(log n)
- Insertion: O(log n)
- Deletion: O(log n)

These time complexities ensure that the seat reservation system operates efficiently even when the number of seats or events increases.

Min Heap Time Complexities

- Get Minimum (root) element: O(1)
- Insert: O(log n)
- Delete Minimum: O(log n)
- Heapify: O(log n)
- Delete arbitrary element: O(log n)

These time complexities enable efficient waitlist management, notably in swiftly accessing the highest-priority user and maintaining the priority order as users are added or withdrawn.

1. The Gator Ticket Master system combines data structures to accomplish fast seat lookups and updates utilizing the Red-Black Tree.
2. Efficient waitlist management with easy access to the top priority user via the Min Heap.

This combination makes a system capable of handling large numbers of seats and users while preserving logarithmic time complexity for most tasks. This scalability makes it suitable for managing events of all sorts, from small gatherings to large-scale concerts or sporting events, ensuring that seat allocations and queue management remain timely and efficient regardless of event size.