COP 5536 - ADVANCED DATA STRUCTURES

Aadithya Kandeth
UFID:69802791

Library Management System Project Report

aadithya.kandeth@ufl.edu

Problem Description:

GatorLibrary is a fictional library that needs a software system to efficiently manage its books, patrons, and borrowing operations. The system should utilize a Red-Black tree data structure to ensure efficient management of the books. Implement a priority-queue mechanism using Binary Min-heaps as a data structure for managing book reservations in case a book is not currently available to be borrowed. Each book will have its own min-heap to keep track of book reservations made by the patrons.Function Prototypes:

Function Prototypes:

1. Class BookNode:

```
def __init__(self, bookId, bookName, authorName, isAvailable)
def get_reservations(self) -> List[int]
def add_reservation(self, patronId: int, priorityNumber: int) -> str
def remove_reservation(self) -> Optional[Tuple[int, int, float]]
```

2. Class RedBlackNode:

```
def __init__(self, val: BookNode)
```

3. Class RedBlackTree:

```
def __init__(self)
def insert(self, val: BookNode) -> None
def delete(self, bookId: int) -> None
def post_insert_rotations(self, new_node: RedBlackNode) -> None
def post_delete_rotations(self, x: RedBlackNode, dict_start: Dict[int, int]) -> None
def find(self, bookId: int) -> Optional[RedBlackNode]
def rotateRight(self, x):
def rotateLeft(self, x):
def node_transplant(self, x, y):
def minimum(self, x):
```

4. Class ReservationNode:

```
def __init__(self, patronId, priorityNumber, reservationTime):
```

5. Class BinaryMinHeap:

```
def __init__(self)
def insert(self, element: Tuple[int, int, float]) -> None
def remove_min(self) -> Optional[Tuple[int, int, float]]
def pop(self):
def get_elements(self):
def heapify_up(self, curr_idx):
def heapify_down(self):
def swap(self, i, j):
```

6. Class LibrarySystem:

```
def __init__(self)
def quit(self):
def add_book(self, bookId: int, bookName: str, authorName: str, isAvailable: str) -> None
def print_book(self, bookId: int) -> str
def print_books(self, node: RedBlackNode, book_id1: int, book_id2: int) -> List[str]
def insert_book(self, bookId, bookName, authorName, isAvailable, borrowing_patron=None,
                reservation_heap=None):
def borrow_book(self, patronId: int, bookId: int, patron_priority: int) -> str
def return_book(self, patronId: int, bookId: int) -> str
def delete_book(self, bookId: int) -> str
def search_book(self, node, bookId):
def find_closest_book(self, node: RedBlackNode, target: int) -> List[str]
def findClosestBookHelper(self, node, target, lowerBook=None, higherBook=None):
def get_book_details(self, node):
def cancel_reservations(self, bookId, patrons):
def color_flip_count(self):
```

Functions and their descriptions:

a.) For the Class: BookNode

1. **__init__(self, bookId, bookName, authorName, isAvailable):**

Members:
- `bookId`: Integer representing the unique identifier of the book.
- `bookName`: String representing the name of the book.
- `authorName`: String representing the name of the author.
- `isAvailable`: Boolean indicating whether the book is available for borrowing.

2. **<u>def get_reservations(self):</u>**

<u>Working:</u>
Retrieves all reservations for the book node and returns a list of patron IDs who have reserved the book. It iteratively removes the minimum element (reservation) from the binary min heap and appends the corresponding patron ID to the reservations list.

<u>Time Complexity:</u>
O(n * log(n)), where n is the number of reservations. In the worst case, all reservations need to be retrieved. This is because each removal from the binary min heap takes O(log(n)) time, and it needs to be done for each reservation.

3. **<u>def add_reservation(self, patronId, priorityNumber):</u>**

<u>Working:</u>
Adds a new reservation to the book node's binary min heap. The reservation is a tuple containing priority, patron ID, and timestamp. If the number of reservations exceeds 20, a message "Waitlist full" is returned.

<u>Time Complexity:</u>
O(log(n)), where n is the number of reservations. This is the complexity of inserting an element into the binary min heap.

4. **<u>def remove_reservation(self):</u>**

<u>Working:</u>
Removes and returns the reservation with the highest priority (minimum priority number) from the binary min heap. If the heap is empty, returns None.

<u>Time Complexity:</u>
O(log(n)), where n is the number of reservations. This is the complexity of removing the minimum element from the binary min heap.

<u>For the Class: RedBlackNode</u>

1. **<u>__init__(self, val: BookNode)</u>** :

<u>Working:</u>
Initializes a Red-Black Tree node with a book node value. The node is initially set to black.

<u>Time Complexity:</u>

O(1), constant time operation for initializing the members.

<u>For the Class: RedBlackTree</u>

1. **<u>__init__(self):</u>**

<u>Working:</u>
Initializes an empty Red-Black Tree with a sentinel node.

<u>Time Complexity:</u>
O(1), constant time operation.

2. **<u>def insert(self, val):</u>**

<u>Working:</u>
The insert function facilitates the addition of a new node to a red-black tree, starting by creating a red node with the specified value. It then traverses the tree using a binary search based on the bookId attribute, determining the appropriate position for insertion. Once the correct spot is found, the new node is linked to its parent and assigned as the root if necessary. To maintain the red-black tree properties, the *post_insert_rotations method* is called to handle any necessary rotations and color adjustments. This ensures that the tree remains balanced and satisfies the constraints of a red-black tree, such as color-coding and structural properties. Overall, the function seamlessly combines binary search principles with red-black tree maintenance for an effective and balanced insertion operation.

<u>Time Complexity:</u>
O(log(n)), where n is the number of nodes in the tree. This is the maximum height of the tree.

3. **<u>def delete(self, val):</u>**

<u>Working:</u>
The delete method in a red-black tree removes a node with the given value while maintaining the red-black tree properties. It first finds the node to delete (z), handles the replacement logic using its successor (y), and then adjusts colors and performs rotations as needed. The method tracks color changes before and after deletion, calculating the number of color flips and updating the color_flip_count accordingly.

<u>Time Complexity:</u>
O(log(n)), where n is the number of nodes in the tree. This is the maximum height of the tree. This is because the tree is balanced, and the maximum height is log(n).

4. **<u>def post_insert_rotations(self, new_node):</u>**

Working:
The post_insert_rotations method is responsible for maintaining the red-black tree properties after a node insertion. It utilizes a while loop to traverse the tree upwards from the newly inserted node (new_node) to the root, checking and adjusting the colors of nodes as needed. The algorithm considers cases based on the relationship between the new node, its parent, and its uncle (sibling of the parent). If the uncle is red, a series of color flips occur, and the algorithm moves up the tree. Otherwise, rotations are applied to restore balance. The method keeps track of color flips with the color_flip_count variable. After the loop, it ensures that the root of the tree is black. The rotations (rotateLeft and rotateRight) are crucial for maintaining the red-black tree structure, and the color adjustments guarantee adherence to the red-black tree properties, resulting in a balanced and efficient structure.

Time Complexity:
O(log(n)), where n is the number of nodes in the tree. This is the maximum height of the tree.

5. **def post_delete_rotations:**

Working:
This method manages the rebalancing of the Red-Black Tree after a node deletion. It employs rotations and color adjustments to maintain the tree's properties. The while loop traverses from the deleted node to the root, making it logarithmic in time complexity. The function also calculates color flips by comparing node colors before and after deletion.

Time Complexity:
O(log N), where N is the number of nodes in the tree. This is because rotations and color adjustments are proportional to the height of the tree. The while loop iterates from the deleted node to the root, performing constant-time operations in each iteration. Since the tree is balanced, the height is logarithmic in the number of nodes.

6.**def find:**

Working:
The find method searches for a given value in the Red-Black Tree. It starts from the root and traverses the tree by comparing values. In the worst case, it has a logarithmic time complexity, reaching the height of the tree. If the value is found, it returns the corresponding node; otherwise, it returns None.

Time Complexity:
O(log N), where N is the number of nodes in the tree. In the worst case, it traverses the height of the tree. The method iterates through the tree, going left or right based on the comparison of the target value with the current node's value. The tree's balanced nature ensures logarithmic time complexity.

7. **def rotateRight:**

Working:
The rotateRight function performs a right rotation at a given node, adjusting pointers to maintain the Binary Search Tree (BST) property. It has a constant time complexity, involving a fixed number of pointer assignments. This rotation is crucial for tree balancing during insertions and deletions.

Time Complexity:
O(1), as the rotation involves reassigning pointers and does not depend on the size of the tree. The operation involves a constant number of pointer assignments, making it independent of the tree size.

8. **def rotateLeft:**

Working:
The rotateLeft method executes a left rotation at a specified node, preserving the BST structure. Similar to rotateRight, it has a constant time complexity, making it efficient for tree rebalancing. Left rotations are vital for maintaining the Red-Black Tree properties during various operations.

Time Complexity: O(1), similar to rotateRight, it involves a constant number of pointer assignments. The operation has a fixed number of pointer assignments, making it constant time.

9. **def node_transplant:**

Working:
The node_transplant function facilitates the swapping of two subtrees rooted at nodes x and y. It determines whether x is a left or right child and updates the parent's pointer accordingly. With a constant time complexity, this function aids in tree restructuring during deletions and transplant operations.

Time Complexity:
O(1), involves reassigning pointers, and the time complexity is constant. The operation consists of a fixed number of pointer assignments, making it constant time.

For the Class: ReservationNode:

1. **__init__(self, patronId, priorityNumber, reservationTime):**

Working:
Initializes a reservation node with patron ID, priority, and reservation timestamp.

Time Complexity:
O(1), constant time operation, as it performs a constant number of operations.

For the Class: BinaryMinHeap:

1. **__init__(self):**

Working:
Initializes an empty binary min heap.

Time Complexity:
O(1), constant time operation, as it performs a constant number of operations.

2. **def insert(self, element):**

Working:
Inserts a new element into the binary min heap and performs heapify-up operation.

Time Complexity:
O(log(n)), where n is the number of elements in the heap. This is the maximum height of the heap. This is because inserting into a binary min heap takes O(log(k)) time.

3. **def remove_min(self):**

Working:
Removes and returns the minimum element (reservation) from the binary min heap and performs heapify-down operation. If the heap is empty, returns None.

Time Complexity:
O(log(n)), where n is the number of elements in the heap. This is the maximum height of the heap.

def heapifyUp(self, current_index):

Working:
The heapifyUp method is used to maintain the heap property by moving a node up the heap until its parent has a smaller value. The method takes one argument, which is the index of the node to be moved up the heap. The method works by repeatedly comparing the node at index x with its parent node at index x/2. If the node has a smaller value than its parent, the nodes are swapped. This process continues until the node is moved to its correct position, i.e., its parent has a smaller value or it reaches the root of the heap.

Time Complexity:
The time complexity of this function is O(log n), where n is the number of nodes in the heap, since the function traverses the height of the heap.

def heapifyDown(self):

Working:
The function takes one argument, the index x of the node to be shifted down. The function starts by entering a loop that continues until the index of the current node x is greater than the index of the last child of the heap. Within the loop, the function calls the getMinChildIndex method to find the index of the minimum child node of the current node x. If the minimum child node violates the heap property (i.e., if the ride object associated with the minimum child node is less than the ride object associated with the current node x according to the comparator method), the function swaps the two nodes using the swapNodes method. Otherwise, the function breaks out of the loop because the heap property has been restored. Finally, the function updates the index of the current node to the index of its minimum child node and continues the loop.

Time Complexity:
The time complexity of this function is O(log n) where n is the size of the heap. The function starts at the given node x and recursively swaps it with its child nodes until the heap property is satisfied. At each level of the heap, the function checks at most two nodes and the height of a binary tree is log n, so the overall time complexity is O(log n).

For the Class: LibrarySystem

1. **def __init__(self):**

Working:
Initializes a LibrarySystem with an empty Red-Black Tree and an empty dictionary for patrons.

Time Complexity:
O(1), constant time operation, as it involves a constant number of operations.

2. **def color_flip_count(self):**

Working:
Returns the count of color flips during Red-Black Tree operations. The detailed working is provided in a separate section below along with images.

Time Complexity:

O(1), constant time operation.

3. **<u>def quit(self):</u>**

<u>Working:</u>
Exits the program.

<u>Time Complexity:</u>
O(1), constant time operation.

4. **<u>def add_book(self, bookId, bookName, authorName, isAvailable):</u>**

<u>Working:</u>
Adds a new book to the LibrarySystem by inserting it into the Red-Black Tree.

<u>Time Complexity:</u>
O(log(n)), where n is the number of books in the Red-Black Tree.

5. **<u>def print_book(self, bookId):</u>**

<u>Working:</u>
Prints the details of a specific book using its book ID.

<u>Time Complexity:</u>
O(log(n)), where n is the number of books in the Red-Black Tree.

6. **<u>def search_book(self, node, bookId):</u>**

<u>Working:</u>
Searches for a book in the Red-Black Tree with the given book ID and returns the corresponding RedBlackNode. Returns None if the book is not found.

<u>Time Complexity:</u>
O(log(n)), where n is the number of books in the Red-Black Tree.

7. **<u>def delete_book(self, bookId) -> None</u>**

Members:
- `bookId`: Integer representing the unique identifier of the book.

<u>Working:</u> Deletes a book from the LibrarySystem by removing it from the Red-Black Tree.

Time Complexity: O(log(n)), where n is the number of books in the Red-Black Tree.

10. **def borrow_book(self, bookId, patronId) -> Union[str, None]**

Working:
Allows a patron to borrow a book. Updates the book's availability status and adds the book to the patron's borrowed books list.

Time Complexity:
O(log(n)), where n is the number of books in the Red-Black Tree.

11. **def return_book(self, bookId, patronId):**

Working: Allows a patron to return a borrowed book. Updates the book's availability status and removes the book from the patron's borrowed books list.

Time Complexity:
O(log(n)), where n is the number of books in the Red-Black Tree.

12. **def cancel_reservation(self, bookId, patronId):**

Working: Allows a patron to cancel their reservation. Removes the reservation from the book's reservation list.
Time Complexity: O(log(n)), where n is the number of books in the Red-Black Tree.

13. **def find_closest_book(self, node, target):**

Working: Finds the two closest books to the target book that is provided. It does this by recursively iterating through each node and finding the closest lower and closest higher. If the target node exists, it returns the node itself.

Time Complexity: O(log(n)), where n is the number of books in the Red-Black Tree.

Main Program Executor:

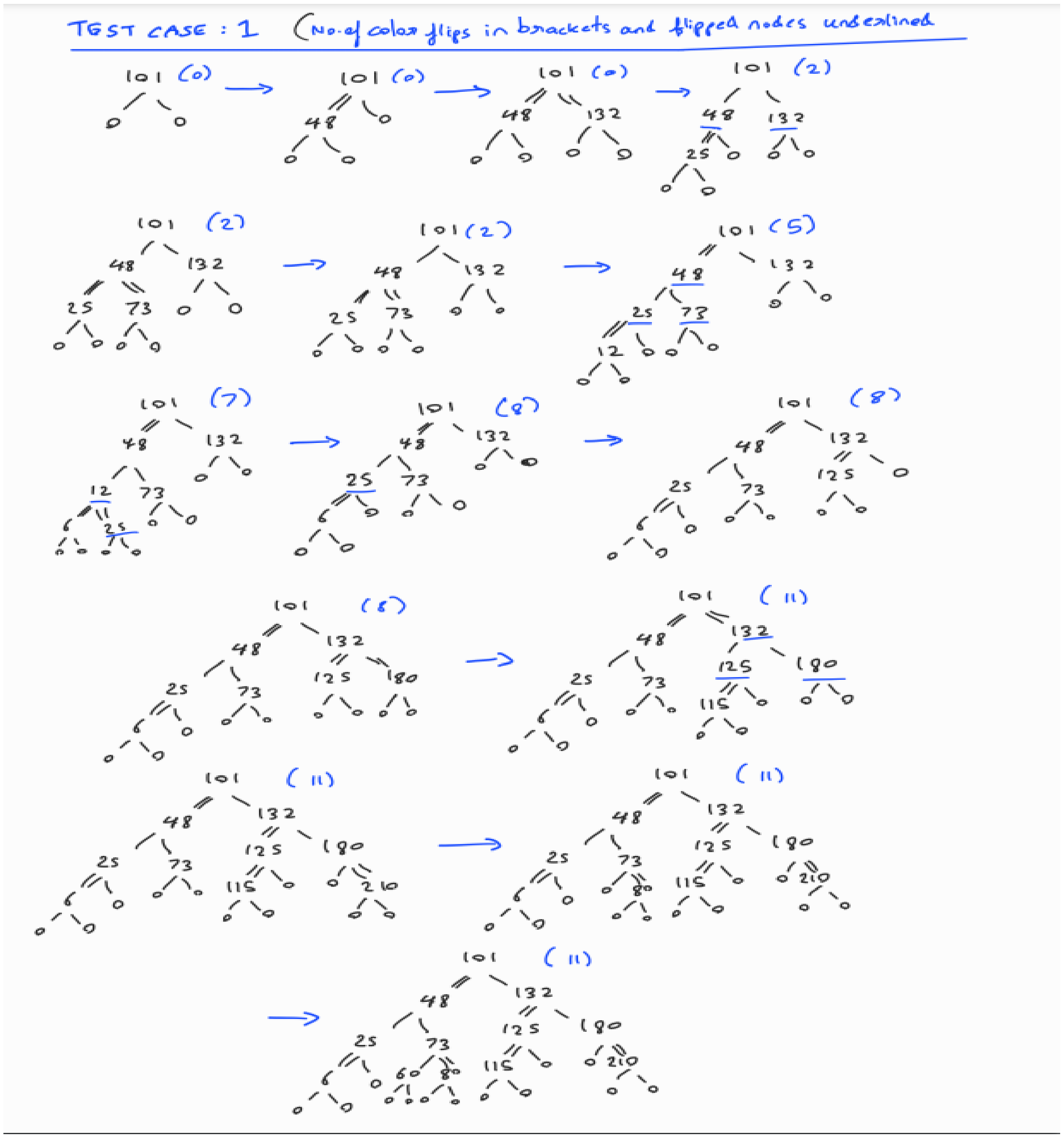
Working:

1. This file takes in the input from the system arguments passed in main, splits the input and reads the lines.
2. It then parses each command and calls the appropriate method associated with the command.
3. The final step is to write into the output file.

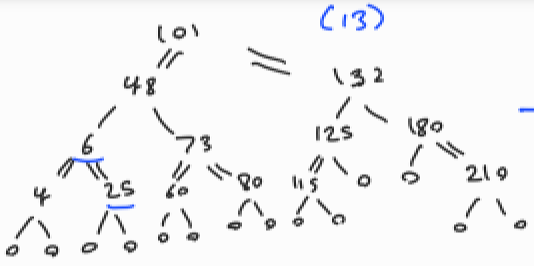
ColorFlip Working:

For the color flips during insert, each increment is counted when a node flips its color during the execution. For the color flips during deletion, a dictionary is used to store the color of the nodes at the beginning and end of the operations. The manual working and proof of correctness for the same is presented in the below diagrams.

Testcase 1: (The number of color flips in the end turns out to be 27). While this differs from the test case results mentioned, I have provided the manual red black tree formation in the images below to highlight how the nodes that are color flipped are counted in each iteration.

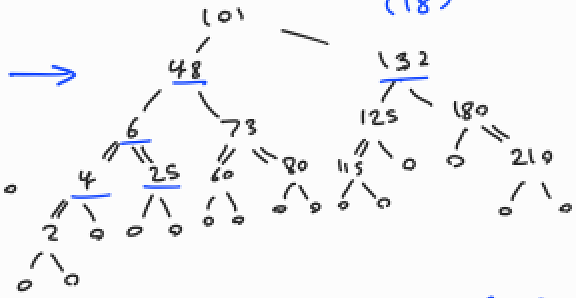


(13)

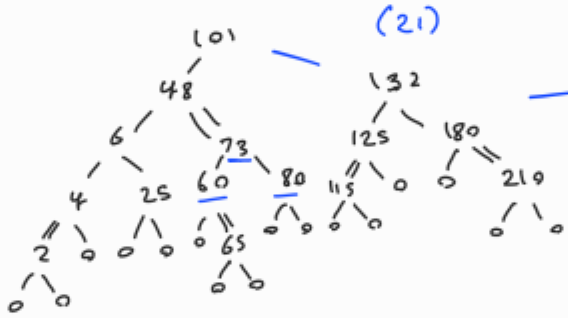


$\rightarrow$

(18)

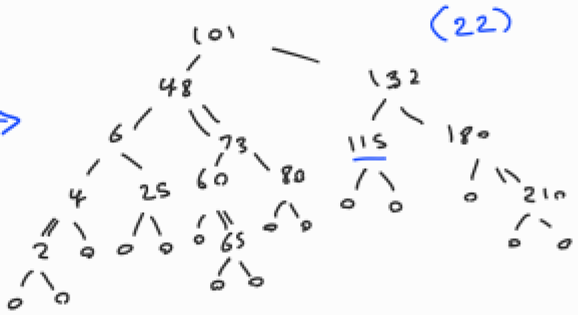


(21)

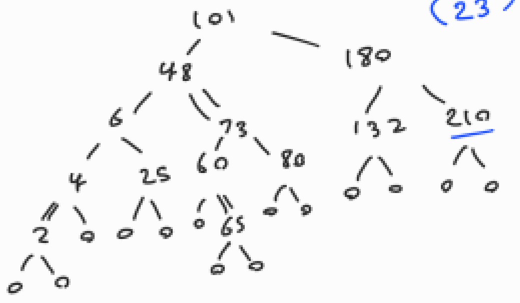


$\rightarrow$

(22)

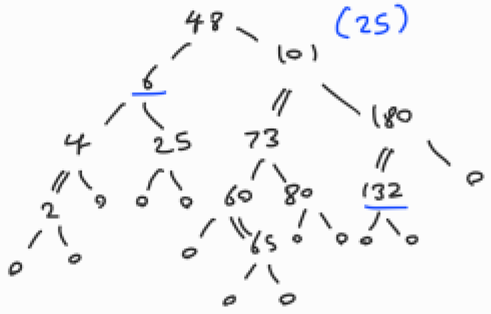


(23)

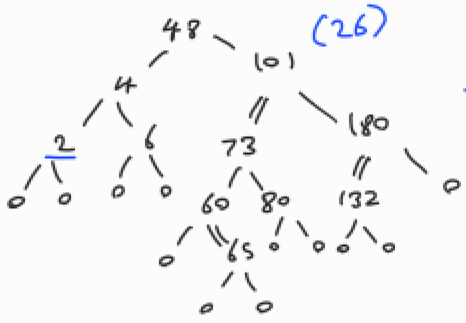


$\rightarrow$

(25)

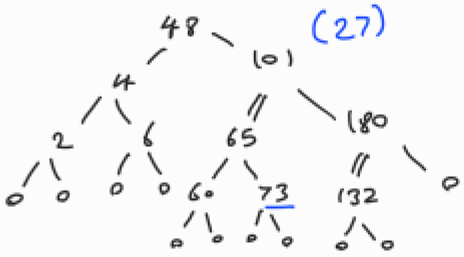


(26)

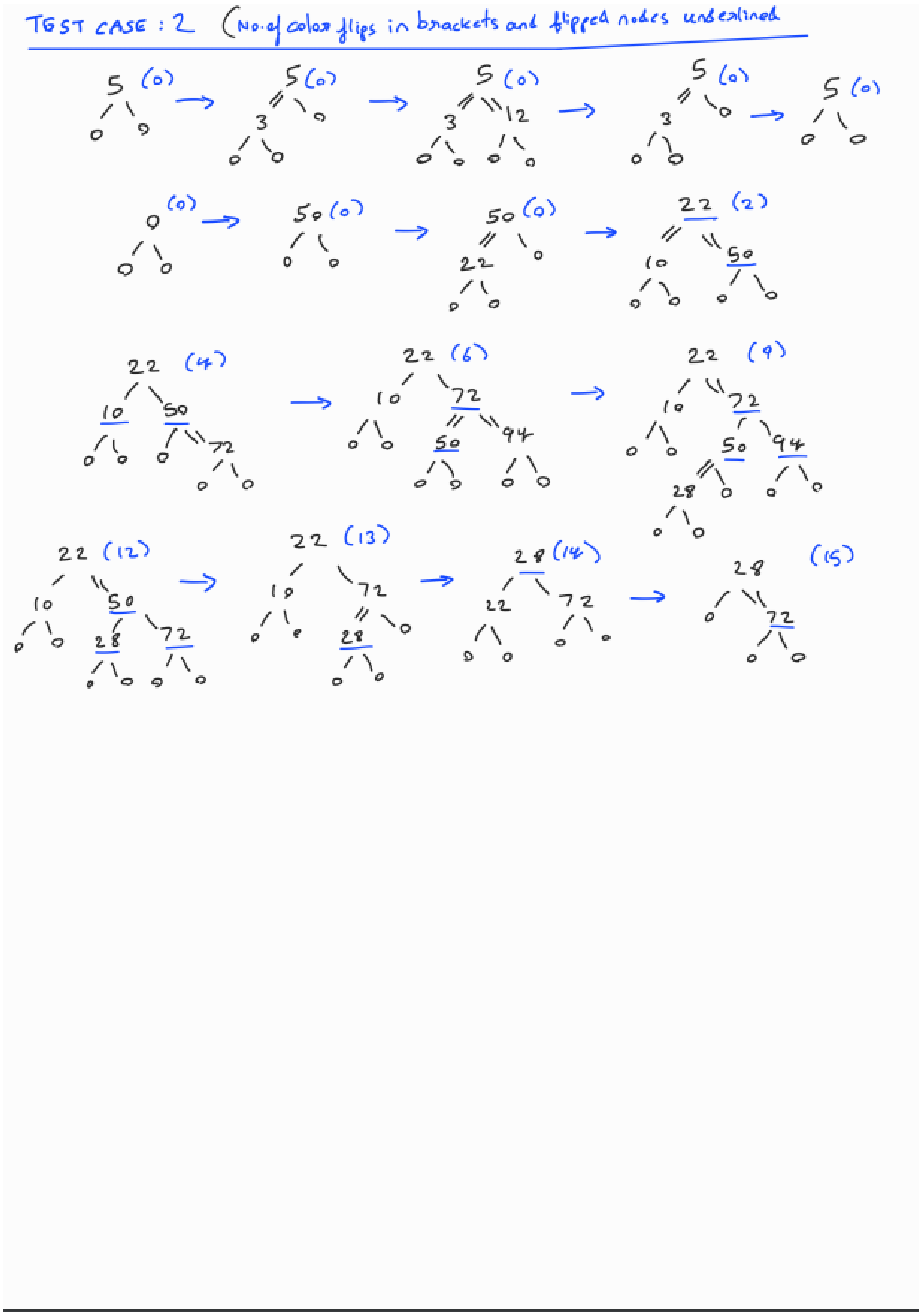


$\rightarrow$

(27)



Testcase 2: The number of color flips in the end turns out to be 15.



EXECUTION STEPS:

1. Download the necessary files
2. Open a cmd window in the project folder
3. Use the ‘make run’ command to run the program
4. Output is generated in input_output_file.txt
5. NOTE: the current input given in the MakeFile is input.txt. This has to be replaced with a file name of your choice that exists in the same folder.

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 binary min heap and red black tree operations to implement the gator library project and in accordance with the specifications given in the problem statement.