University of Liege

Programming Techniques

Bytestream mapper

Boustani Mehdi : S221594
Albashityalshaier Abdelkader : S211757

1. Introduction

In this project, we implemented the MAGIC ADT to efficiently track byte positions in dynamic data streams. The challenge was to handle modifications (additions and removals) and position mapping queries quickly. To achieve this, we designed our solution using an Interval Tree based on a Red-Black Tree, providing balanced performance for all operations. This report explains our approach, analyzes its time and space complexity, presents our testing methodology, and evaluates the implementation's performance.

2. Algorithms and data structures

2.1. Motivation

Our implementation of the MAGIC ADT uses an Interval Tree based on a Red-Black Tree structure to efficiently track and query byte position mappings. This approach was chosen for several reasons :

- Operations on byte streams (add, remove) can be represented as intervals [low, high], with position and length.
- Red-Black Trees properties provide guaranteed O(log n) operations for insertions and lookups, as they keep the tree balanced with rotations and recoloring.
- Augmenting the tree with interval metadata (which will be the minimum low of all intervals in the subtree) allows for efficient range queries and pruning.

2.2. Implementation

Each node (Interval Node) in our tree represents a single operation (add or remove) and stores :

- The interval boundaries (low, high), where low represents the starting position (pos), and high represents the ending position (pos + length) of the operation.
- Operation type (add or remove)
- Sequence number to track chronological ordering
- Minimum subtree value for efficient pruning during queries
- Color (red or black) and pointers for tree balancing

```
1  /* Opaque Structure for Interval Node */
2  typedef struct INode_t INode;
3  typedef enum {RED=0, BLACK=1} Color;
4  typedef enum {REMOVE=0, ADD=1} OperationType;
5  struct INode_t {
6      unsigned int low;
7      unsigned int high;
8      unsigned int seqNumber;
9      OperationType opType;
10     unsigned int minSubtree;
11
12     Color color;
13     INode *left, *right, *parent;
14 };
15 /* MAGIC ADT */
16 struct magic {
17     INode *root;
18     size_t size;
19 };
```

Our program organizes operation nodes by sequence number to maintain their chronological order, as operations are applied sequentially to affect positions in the bytestream. [1]

How ?

Thanks to our implementation of Red Black Insertion function **rbInsert**, we can insert nodes in order of their sequence number, while maintaining and updating the metadata in order to allow efficient mapping. The insert function also uses a fixup **rbInsertFixup**, which allows to keep the balance of the tree by performing rotations and/or recolorings of nodes. The choice of Red Black tree rather than AVL tree is because Red Black tree properties are less strict so it is better for insertion-heavy operations, requiring fewer rotations while still maintaining O(log n) height guarantees. This makes our implementation more efficient in scenarios where many operations (additions and removals) are performed in succession. **MAGICadd** and **MAGICremove** use rbInsert to add new operation nodes to the tree while incrementing the sequence counter, ensuring operations are stored in the order they were performed.

For the mapping algorithms (**MAGICmap**), we implemented two specialized tree traversal functions :

- **mapInOut** - Maps positions from input to output stream :
  - Traverses the tree in-order (chronological sequence)
  - For ADD operations : shifts positions forward if they're after the insertion point
  - For REMOVE operations : returns -1 if position was removed, or shifts positions backward if after the removal
- **mapOutIn** - Maps positions from output to input stream :
  - Traverses the tree in reverse chronological order
  - Reverses the effects of each operation encountered
  - Handles added positions (which do not exist in input stream)

Both algorithms use the **minSubtree** metadata for pruning, which allows us to skip entire subtrees when the position being mapped couldn't possibly be affected by operations in that subtree. This optimization reduces the average time complexity from O(n) to O(log n) for mapping operations.

3. Complexity analysis

3.1. Time Complexity

- **MAGICinit :** O(1) - only allocates the magic structure and initializes its fields.
- **MAGICadd :** O(log n) - Creates a new node and inserts it into the Red-Black tree based on sequence number. Thanks to the self-balancing property of Red-Black trees, insertion takes logarithmic time.
- **MAGICremove :** O(log n) - Identical complexity to MAGICadd as it also creates a node and inserts it into the tree.
- **MAGICmap :** O(log n) average case - With pruning using the minSubtree metadata, we can skip entire subtrees that don't affect the position being mapped. Without pruning, this would be O(n) in the worst case, but our implementation approaches O(log n) for most practical queries.
- **MAGICdestroy :** O(n) - Must traverse and free every node in the tree.

1. We had initially started by organizing our nodes by low boundary of intervals, but we had problems of ordering that required much more complex code even while tracking sequence numbers and using pruning, so we moved to sequence number to ensure correctness before all

3.2. Space Complexity

The space complexity of our implementation is O(n), where n is the number of operations performed (adds and removes). Each operation requires a single node in our tree. Since we store exactly one node per operation, the space grows linearly with the number of operations. The auxiliary space used during recursion for mapping is O(log n) due to the balanced height of the Red-Black tree.

4. Testing plan and Performance study

Our testing strategy for the MAGIC ADT implementation follows a systematic approach to ensure correctness, robustness, and performance efficiency. We designed several tests to evaluate different aspects of the implementation :

- **Correctness Tests :** These tests includes :
  - **Basic functionality tests** to verify that our implementation is convenient with the example (Figure 1 of project statement)
  - **Edge case tests** to evaluate boundary conditions (Empty MAGIC instance, only add operations, only remove operations, overlapping add/remove operations)
  - **Sequential operations tests** to verify correct behavior when performing multiple sequential add/remove operations.
  - **Error handling tests** to evaluate Evaluate robustness with invalid inputs like negative positions, or negative lengths.
- **Performance Tests :** We created several performance tests to evaluate efficiency under various workloads. These tests include :
  - **Scalability tests :** Measure performance with increasing workloads (increasing number of operations and mappings).
  - **Stress testing :** Evaluate performance under high load with 30,000 clustered operations concentrated in specific position ranges, followed by targeted mapping operations in each cluster.
  - **Spike testing :** Verify handling of sudden large increases in load, comparing mapping performance before and after a rapid spike of 20,000 operations.
  - **Volume testing :** Ensure the implementation can handle large bytestream sizes.
- **Endurance Test :** Evaluate performance stability over time. Consists of running the system continuously for 30 seconds with batches of 1,000 random operations and 10,000 mappings per iteration, then monitor and measure processing times throughout the test duration to detect any performance degradation.

5. Results and Conclusion

Our implementation successfully passed all correctness tests while demonstrating logarithmic scaling with increasing operation counts. Performance testing confirmed efficient mapping times even under high loads, and endurance testing showed stable performance over time. The minSubtree pruning optimization proved highly effective, allowing the implementation to skip entire subtrees during mapping operations and maintaining on average O(log n) performance across wide position ranges. Our Interval Tree approach based on a Red-Black Tree structure provides an optimal balance of time and space efficiency, making it suitable for handling high-throughput bytestream modifications in real-world environments.