INFO0027: Bytestream-Mapper

Merve SICIM s184346, Sam EL MASRI s190377

April 2025

Table des matières

Introduction

This project presents the design and implementation of the MAGIC ADT (Abstract Data Type), a data structure capable of dynamically managing insertions and deletions in a bytestream. The system is built to support high-performance mapping between input and output byte positions in a streaming context, where data can span from a few bytes to several gigabytes. Operations must be efficient both in terms of time and space, and robust under frequent modifications.

1 Data Structures and Algorithms

The MAGIC ADT is implemented using a red-black tree [1], where each node represents a modification (insertion or deletion) as a delta at a specific position in the input stream. This approach allows us to maintain a cumulative transformation of the stream with logarithmic insertion and deletion complexity.

Each node holds the following information :

— **pos** : Position in the input stream where the modification occurs.
— **delta** : Positive value for additions, negative for deletions.
— **totalDelta** : Sum of all deltas in the subtree, used for cumulative mapping.

In addition to the red-black tree, MAGIC maintains two caches :

— **inMapping[]** : Maps input positions to output positions.
— **cacheMapping[]** : Maps output positions to input positions.

These arrays are updated on-demand and allow efficient O(1) access when valid. The tree supports O(log n) updates via insertion of new deltas.

2 Functional Suitability

The following public API is implemented :

— `MAGICinit()` : Initializes a new MAGIC instance.
— `MAGICadd(m, pos, len)` : Adds a sequence of bytes at the specified output position.
— `MAGICremove(m, pos, len)` : Removes a sequence of bytes from the specified output position.
— `MAGICmap(m, direction, pos)` : Returns the corresponding position in the stream (input/output).
— `MAGICdestroy(m)` : Frees all allocated memory.

3 Performance Efficiency

3.1 Time Performance

MAGIC's performance is based on the red-black tree structure :

— **Insertion/Removal** : O(log n) in the worst case, due to balanced tree operations.
— **Mapping** : O(1) when cache is valid ; O(log n) otherwise.
— **Cache updates** : Amortized efficient thanks to partial re-evaluation starting from `cacheDirtyFrom`.

1. from the course

A special optimization ensures that cache updates only affect positions that have changed. Positions that are removed are marked in a boolean array during update. This ensures high performance even when handling gigabyte-scale streams.

3.2 Space Performance

In terms of space, the MAGIC ADT maintains :

— A red-black tree with up to nodes, where is the number of insertions and deletions applied. Each node uses constant space, leading to **O(n)** space in the worst case.

— Two dynamic arrays (`inMapping[]` and `cacheMapping[]`) that grow based on the maximum positions seen in the input and output streams, respectively. In the worst case, these arrays may reach sizes proportional to the length of the final stream, i.e., **O(m)** with the maximum stream length.

Thus, the total space complexity is **O(n + m)** in the worst case.

In the average case (under regular insert/delete workloads and when the stream length remains moderate), the tree remains balanced and the arrays are not significantly oversized, which results in a practical average space usage of **O(n)**.

4 Test Plan

We designed a suite of tests to validate the correctness and performance of MAGIC :

4.1 Unit Tests

— Additions and removals at various positions (start, middle, end).

— Mapping consistency : ensure `MAGICmap(IN` $\rightarrow$ `OUT)` and `(OUT` $\rightarrow$ `IN)` are inverses.

— Removal of already removed bytes.

All unit tests were passed successfully [2] :

— IN $\rightarrow$ OUT : Test 1 to Test 5 passed

— OUT $\rightarrow$ IN : Test A and Test B passed

4.2 Stress Tests

— Sequences of over 10,000 mixed operations.

— Alternating additions/removals to force tree rotations and cache invalidations.

4.3 Performance Benchmarks

Performance tests [3] were conducted with 1,000,000 operations. Results :

— `MAGICadd` #1000000 : 0.409 sec

— `MAGICremove` #1000000 : 0.077 sec

— `MAGICmap IN -> OUT` #1000000 : 0.003 sec

— `MAGICmap OUT -> IN` #1000000 : 0.002 sec

These results confirm that the data structure scales very well and supports real-time operations even at high volume.

2. Example of the assignment included
3. CPU : Ryzen 7 5800x3d

Conclusion

The MAGIC ADT provides an efficient and robust solution to the problem of tracking dynamic modifications in a bytestream. Its combination of balanced tree structures and intelligent caching ensures excellent performance even in large-scale environments.