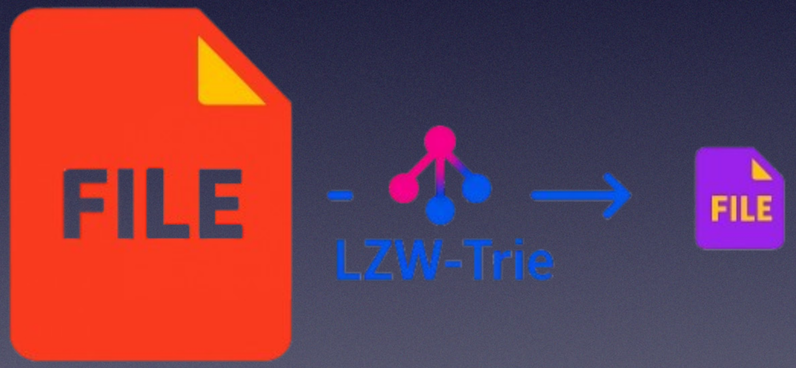
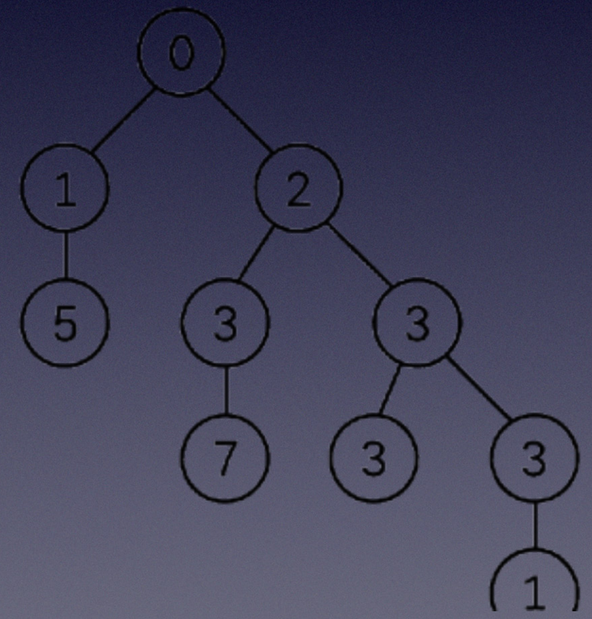
# LZW-Trie: High-Speed Data Compression Using Trie Dictionary

- A C++/Python Implementation for Efficient Lossless Compression




Team Leader : Sumit Kumar
DAA-IV-T251

Team Member : Aviral Maheshwari,
Anubhav Vashistha

Guided by : Dr. Jyoti Agarwal

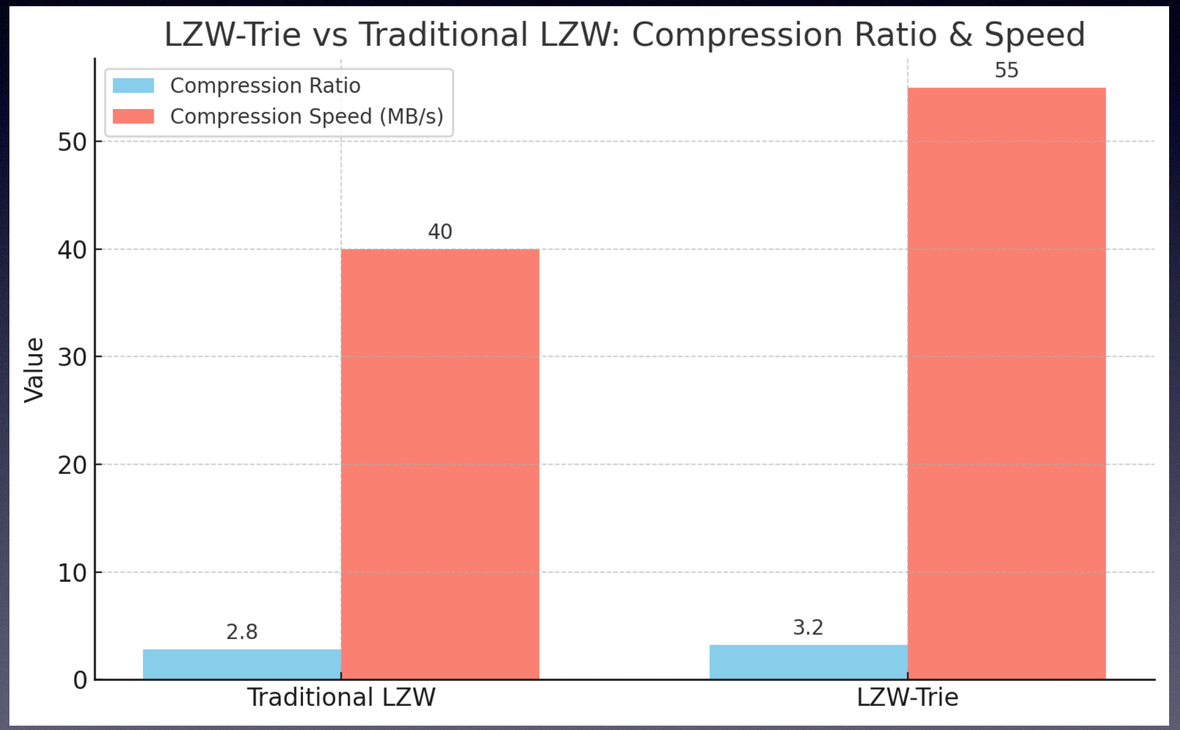
# What We Aim to Achieve

**Efficiency:**
Reduce text file sizes by **40–60%** using adaptive trie compression.

**Speed:**
Achieve **20% faster** compression than standard hash-table LZW.

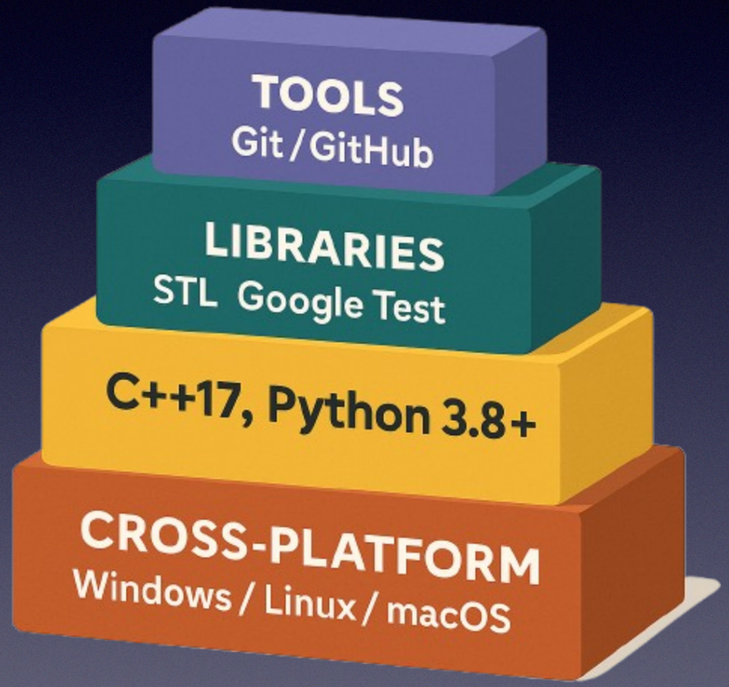
**Scalability**:
Handle files from **1KB to 1GB+** with dynamic memory management.

**Education:**
Demonstrate real-world trie applications in data compression.



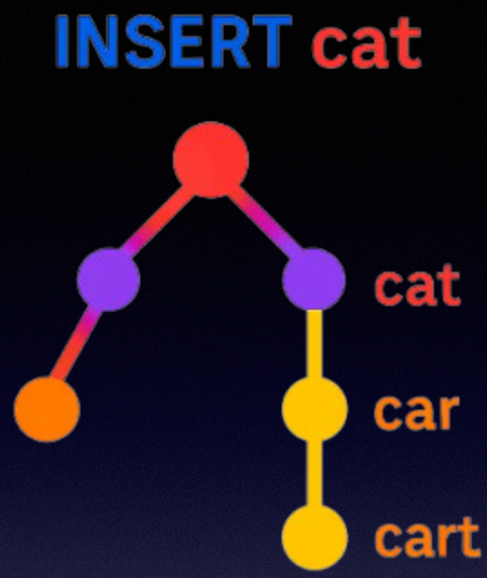
# Built with Modern Tools

- **Languages:** C++17 (Core), Python 3.8+ (Reference)
- **Libraries:** STL (<unordered_map>, <memory>), Google Test
- **Tools:** Git/GitHub
- **OS:** Cross-platform (Windows/Linux/macOS)




**Hybrid Trie-Hash Dictionary**: Optimizes both speed and memory.

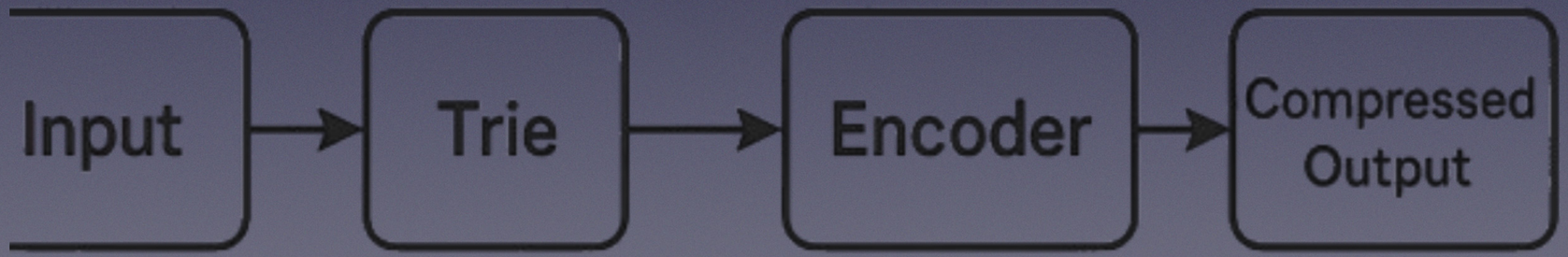
# Why LZW + Trie?




- **Problem**: Traditional LZW with hash tables is slow for prefix searches.
- **Solution**: Trie cuts search time from **O(n) → O(k)** (k = key length).
- **Use Case**: Ideal for compressing:
  - Log files
  - Source code
  - Repetitive datasets

# Compression Engine

- **Input**: Read raw data (e.g., "TOBEORNOTTOBEORTOBEORNOT").
- **Trie Build**: Dynamically grow dictionary (e.g., "TO" → 256, "TOB" → 257).
- **Output**: Emit codes (e.g., 84, 79, 66, 69 → 256, 79, 258).

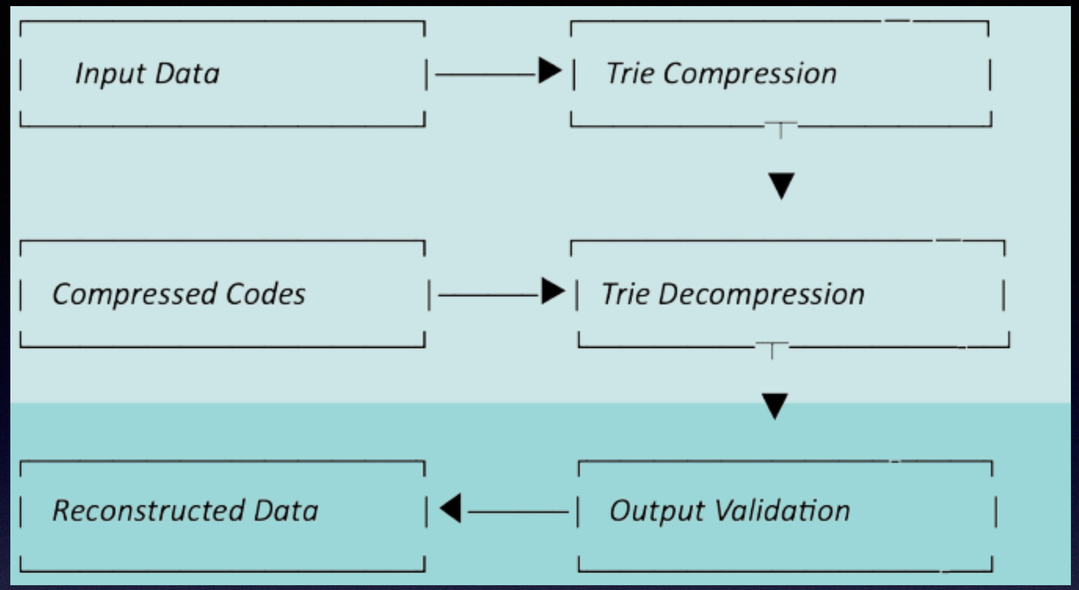


# Key Innovations

- **Adaptive Code Width**: Starts at 9 bits, grows to 16 bits as dictionary expands.
- **Memory Safety**: Smart pointers (**unique_ptr**) prevent leaks.
- **Threading**: Optional parallel compression for large files.

```
$ compressoor example.txt
File    Before  After Reduction
-------------------------
example. 1,0MB  500 kB  51,6%
```

# End-to-End Pipeline




- **Input Data**: Original uncompressed information
- **Compressed Codes**: Encoded representations of data patterns
- **Trie Compression**: Prefix-tree based encoding process
- **Trie Decompression**: Dictionary rebuilding from compressed trie paths
- **Reconstructed Data**: Decompressed output regenerated from codes
- **Output Validation**: Verification of data integrity post-compression

# Who Does What?

| Member | Role | Deliverables |
| --- | --- | --- |
| **Sumit** | Trie Architect | Trie design, compression core |
| **Aviral** | System Engineer | Decompression, I/O, memory optimization |
| **Anubhav** | QA Lead | Test cases, benchmarks, user docs |