Scott Elliott

Move-to-Front and Transpose Linked List Implementation Documentation

---

Files Submitted

Core Implementation Files

- CDLinkedList.h / CDLinkedList.cpp - Base class implementing circular doubly-linked list with dummy header
- mtflist.h / mtflist.cpp - Move-to-Front derived class that moves accessed elements to front
- transposelist.h / transposelist.cpp - Transpose derived class that swaps elements with predecessors

Comprehensive Testing Files

- *driver_se.cpp* - Comprehensive test driver that validates all three list implementations in a single run
- *LLStats_se.cpp* - Enhanced statistics program that measures all three list types simultaneously

Implementation Overview

Core Data Structure

The program implements a circular doubly-linked list using a dummy header node.
Each DListNode contains an integer value, a pointer to the previous node, and a pointer to the next node. The list maintains a circular structure where the last node points back to the header, and the header points to the last node.

Key Design Features

- **No Duplicates**: The *add*() method checks for existing values using *contains*() before insertion
- **Traversal Counting**: All operations increment a traverseCount metric to measure performance
- **Memory Management**: All nodes are dynamically allocated and properly deallocated in destructors
- **Deep Copying**: Copy constructors create independent list instances with their own nodes

List Variants

- **CDLinkedList**: The base implementation provides stable list behavior with no reorganization
- **MtfList**: Implements move-to-front strategy by relocating accessed elements to the front

- **TransposeList**: Implements transpose strategy by swapping accessed elements with their predecessors

Enhanced Testing Capabilities

***driver_se.cpp* - Comprehensive Test Driver**
This file provides a complete validation suite that tests all three list implementations in sequence. It includes:

- Unit tests for CDLinkedList covering empty lists, element addition, duplicate prevention, removal, and traversal counting
- Unit tests for MtfList validating move-to-front behavior and performance improvements
- Unit tests for TransposeList verifying swap operations and gradual element movement
- Automated execution of all test suites with detailed pass/fail reporting

***LLStats_se.cpp* - Comprehensive Statistics Program**
This enhanced version provides simultaneous performance analysis of all three list types:

- Tests CDLinkedList, MtfList, and TransposeList under identical conditions
- Measures performance under both uniform and normal distributions
- Calculates comparative improvements between list types
- Provides side-by-side results for easy performance comparison

Performance Analysis Results

Experimental Setup

- List Size: 1000 elements
- Search Operations: 100,000 accesses
- Distributions: Uniform (all elements equally likely) and Normal (skewed access pattern)

Measured Performance

List Type	Uniform Distribution	Normal Distribution	Improvement
CDLinkedList	499.23 nodes	500.06 nodes	Baseline
MtfList	501.24 nodes	388.20 nodes	**22.3% Improvement**
TransposeList	499.25 nodes	499.88 nodes	0.2% Improvement

Key Performance Insights

- Under uniform distribution, all three lists show nearly identical performance (~500 nodes per search), confirming that without access patterns, reorganization provides no benefit
- Under normal distribution, MtfList demonstrates significant improvement, reducing average traversals by 111.86 nodes compared to baseline
- TransposeList shows minimal improvement, suggesting the gradual swap strategy requires more extreme access patterns or longer adaptation periods
- The comprehensive testing in LLStats_se.cpp provides direct comparison between all three implementations under identical conditions

Compilation and Execution Guide

Comprehensive Performance Analysis (Recommended):

g++ CDLinkedList.cpp mtflist.cpp transposelist.cpp LLStats_SE.cpp -o stats_se

./stats_se

Comprehensive Unit Testing: (Comment in main() in driver_se.cpp)

g++ CDLinkedList.cpp mtflist.cpp transposelist.cpp driver_se.cpp -o tests_se

./tests_se

Individual List Testing:

g++ CDLinkedList.cpp mtflist.cpp transposelist.cpp LLStats.cpp -o stats

./stats

Memory Leak Validation:

valgrind --leak-check=full ./stats_se

Technical Implementation Details

Algorithm Specifications

- **MtfList::contains()**: When an element is found, it is unlinked from its current position and reinserted immediately after the dummy header
- **TransposeList::contains()**: When an element is found (and not already at front), it is swapped with its immediate predecessor
- **CDLinkedList::contains()**: Performs standard sequential search without any reorganization

Enhanced Testing Architecture

- **driver_se.cpp**: Provides systematic validation of all list operations including edge cases, memory management, and reorganization behavior
- **LLStats_se.cpp**: Implements comparative analysis framework that eliminates testing variables by running all list types under identical conditions
- **Modular Design**: Both enhanced files maintain compatibility with the original testing interfaces

Conclusions and Observations

Performance Findings
The comprehensive analysis provided by LLStats_se.cpp clearly demonstrates the relative performance of all three list implementations. MtfList's 22.3% reduction in average traversals shows the effectiveness of aggressive reorganization for frequently accessed items. The minimal improvement shown by TransposeList suggests that its conservative approach may require different parameters or access patterns to demonstrate significant benefits.

Testing Effectiveness
The enhanced driver_se.cpp provides thorough validation of all list operations, ensuring correctness across all three implementations. The comprehensive approach eliminates the need to run multiple separate test programs and provides consistent testing methodology across all list types.

Code Quality Assessment
The implementation maintains high code quality with comprehensive memory management (zero leaks confirmed via valgrind), robust error handling, and clear separation of concerns through object-oriented design. The enhanced testing suite provides complete validation coverage while maintaining clean, maintainable code structure.

This project provides a solid foundation for understanding and evaluating self-organizing data structures with empirical performance metrics and comprehensive testing validation.

References

DeepSeek AI. (2024). *DeepSeek* (Version 2) [Large language model]. https://www.deepseek.com/

- Used for research and debugging assistance during code development, particularly for identifying and resolving compilation errors related to constructor ambiguity and pointer manipulation in the linked list implementation. Also utilized for improving grammatical accuracy and structural cohesiveness in this documentation.

Microsoft Corporation. (2024). *Visual Studio Code Auto-complete and IntelliSense Features* [Integrated development environment].

- Employed Visual Studio's auto-fill and code completion features to assist with code documentation, including generating consistent comment patterns and maintaining coding standards throughout the implementation.

Note: All code implementation and architectural decisions were made by the author. AI tools were used solely for debugging assistance, documentation improvement, and productivity enhancement purposes. The core algorithms, design patterns, and final implementation represent original work.