Scott Elliott
Design

Move-to-Front and Transpose Linked List Implementation

Definitions

Linked List: A data structure consisting of nodes where each node contains data and pointer(s) to other nodes. Unlike arrays, linked lists do not require contiguous memory allocation.

Circular Doubly Linked List: A linked list where each node has two pointers (previous and next), and the last node's next pointer points back to the first node, forming a circle. The first node's previous pointer points to the last node.

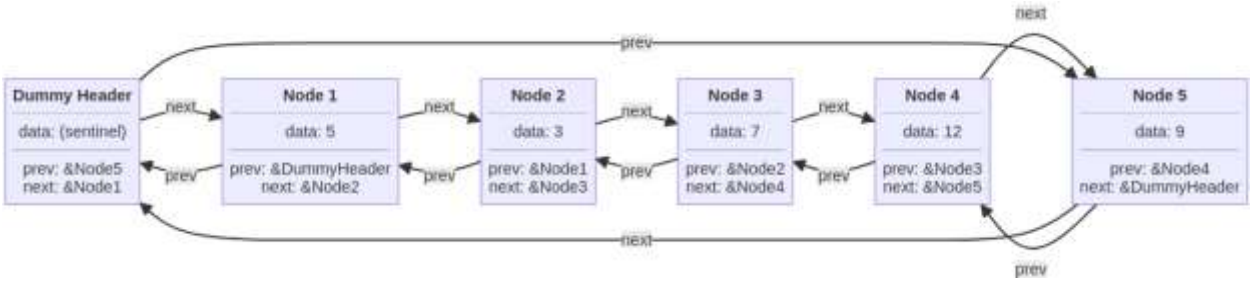
Dummy Header: A special node that does not contain meaningful data but serves as a fixed entry point to the list. It simplifies edge case handling by ensuring the list always has at least one node. The dummy header's next pointer points to the first real data node, and its previous pointer points to the last real data node.

Move-to-Front (MTF): Every time an element is accessed, it is moved to the front of the list (immediately after the dummy header). This strategy works well when recently accessed items are likely to be accessed again soon.

Transpose Strategy: A self-organizing list method where every time an element is accessed, it is swapped with its immediate predecessor in the list. This is a more conservative approach than MTF, gradually moving frequently accessed items toward the front.

Traverse Count: A metric tracking the number of node-to-node movements during list operations. This helps measure the efficiency of different list organizations.

Visual Representation of Circular Doubly Linked List with Dummy Header:



Specification

Purpose

This program implements three variants of a circular doubly linked list to demonstrate how self-organizing strategies (Move-to-Front and Transpose) can improve search performance when certain elements are accessed more frequently than others. The program will measure and compare the average number of nodes traversed during searches across different access patterns.

Requirements

Input: For testing purposes, the program accepts individual integer values to add, remove, or search. For statistics collection, it uses automatically generated random integers following a uniform or normal distribution from LLStats.cpp.

Output: During testing, the program displays boolean results of operations and the current state of the list. For statistics, it outputs the average traverse count for different list types and access patterns.

Performance: The memory complexity is O(n) where n is the number of elements in the list. For time complexity, the Add operation runs in O(n) time because it must check for duplicates. The Remove operation also runs in O(n) time since it must search for the element. The Contains operation in the base implementation runs in O(n) time due to sequential search. For the MTF variant, Contains has an O(n) worst case but an improved average case. Similarly, the Transpose variant has an O(n) worst case with an improved average case.

Constraints: All nodes must be dynamically allocated, and no memory leaks are allowed, which will be verified using valgrind. The list stores integers only, and no duplicate values are allowed in the list.

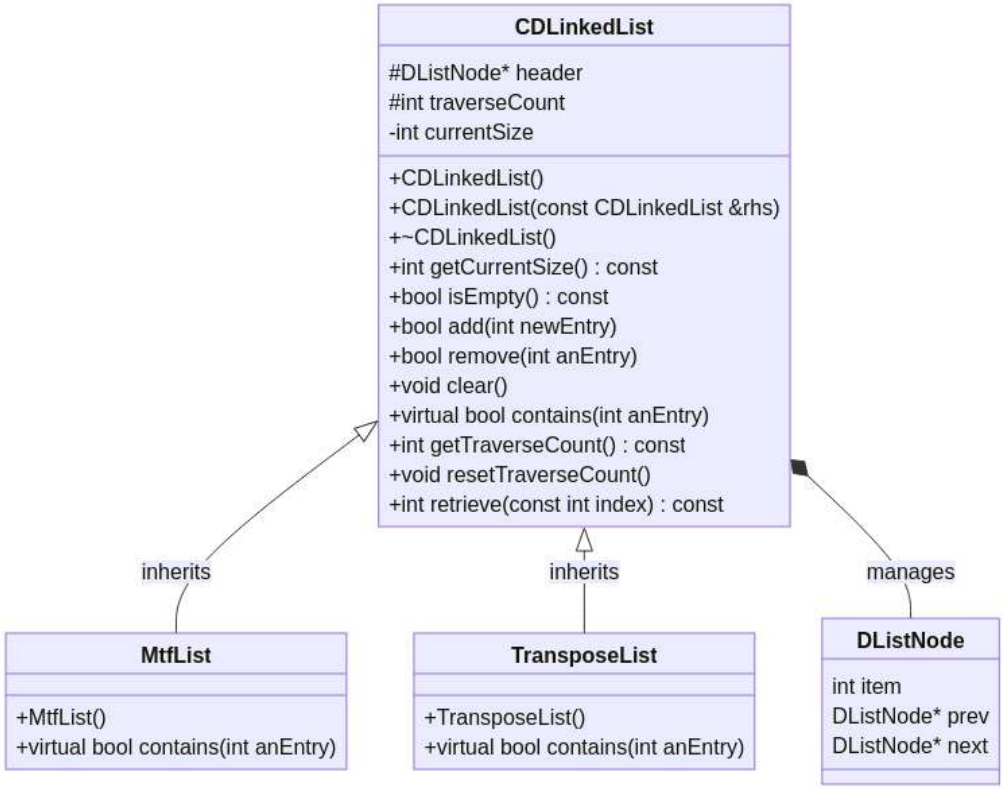
Assumptions: The program assumes that input values for testing are valid integers. For statistical analysis, it assumes that access patterns follow specified distributions, either uniform or normal.

Algorithm Overview

Base List (CDLinkedList): The base list maintains a circular doubly linked list with a dummy header. Each DListNode contains an integer item, a prev pointer, and a next pointer for bidirectional navigation. Elements are added to the front if not present, searched sequentially while counting traversals, and removed by unlinking.

MTF List: Inherits base functionality but overrides contains() to move accessed elements directly to the front, bringing frequently accessed items forward for faster searches.

Transpose List: Inherits base functionality but overrides contains() to swap accessed elements with their predecessor, gradually moving frequently accessed items forward through a more conservative approach.



Each list uses multiple DListNode instances
CDLinkedList has-a DListNode* (header)

- MtfList IS-A CDLinkedList (inheritance)
- TransposeList IS-A CDLinkedList (inheritance)
- CDLinkedList HAS-A DListNode (composition - header and all data nodes)
- Polymorphism: Can use base class pointer/reference to call overridden contains()

Edge Cases

- Empty list operations (remove, contains on empty list)
- Single element list
- Accessing first element (already at front for MTF, no previous for Transpose)
- Accessing dummy header (should never contain searchable data)
- Adding duplicate values (should be ignored)
- Removing non-existent values

Input

For Testing (using driver.cpp):

- Individual integer values for add operations

    - Ex.: `testlist.add(5);  testlist.add(10); testlist.add(15);`
- Individual integer values for remove operations
    - `Ex.: testlist.remove(10);`
- Individual integer values for search/contains operations
    - Ex.: `testlist.contains(5);  testlist.contains(15);`
- Each class (CDLinkedList, MtfList, TransposeList) tested separately by including appropriate header files
    - Ex. compilation: `g++ CDLinkedList.cpp driver.cpp`
    - Ex. compilation: `g++ CDLinkedList.cpp mtflist.cpp driver.cpp`

For Statistics Collection (using LLStats.cpp):

- Automatically generated random integers following a normal distribution
    - Ex.: Random integers between 0-99 generated internally
- LLStats.cpp modified to test each list variant separately (CDLinkedList, MtfList, or TransposeList)
    - Ex. compilation: `g++ CDLinkedList.cpp LLStats.cpp`
    - Ex. compilation: `g++ CDLinkedList.cpp mtflist.cpp LLStats.cpp`
- Access pattern specifications (uniform or normal distribution)

Outputs

For Testing (from driver.cpp):

- Boolean results indicating success/failure of operations (add, remove, contains)
    - Ex.: `add(5) returned: true` (element added successfully)
    - Ex.: `add(5) returned: false` (duplicate, not added)
    - Ex.: `contains(10) returned: true` (element found)
    - Ex.: `remove(20) returned: false` (element not in list)
- Current state of the list (visual representation showing node arrangement)
    - Ex.: `List contents: 15 -> 10 -> 5`
    - Ex.: `List is empty`

For Statistics (from LLStats.cpp):

- Average traverse count for the base CDLinkedList
    - Ex: `CDLinkedList average traversals: 25.67`
- Average traverse count for the MTF list
    - Ex: `MtfList average traversals: 12.34`
- Average traverse count for the Transpose list
    - Ex: `TransposeList average traversals: 18.92`
- Results recorded separately for each class (requires recompiling with different list types)

- Comparison of performance across different access patterns (uniform vs. normal distribution)
    - Ex.: `Normal distribution access pattern: MTF shows 52% improvement over base`
    - Ex.: `Uniform distribution access pattern: MTF shows 28% improvement over base`

Error Handling

- The program must verify memory allocation failures by checking the return value of `new` operator or using try-catch blocks
- The program must verify that pointers are not null before dereferencing
- The program must verify empty list operations and handle contains/remove on empty list gracefully (return false)
- The program must verify that the index in retrieve() is within range [0, size-1]
- The program must verify that the destructor deallocates all nodes to prevent memory leaks
- The program must verify that the copy constructor creates an independent deep copy of all nodes
- The program must verify that all pointer updates maintain the circular structure

---

Design

Overview

The implementation uses inheritance to create specialized list behaviors. CDLinkedList provides the base circular doubly linked list with standard operations. MtfList and TransposeList inherit from CDLinkedList and override only the `contains()` method to implement their respective reorganization strategies. All three classes share the same node structure and basic list manipulation logic.

Component 1: DListNode

Purpose: Represents a single node in the doubly linked list.

```
struct DListNode {
  int item;           // The data stored in this node
  DListNode *prev;    // Pointer to previous node
  DListNode *next;    // Pointer to next node
};
```

This is a simple struct with no methods. All manipulation is handled by the list classes.

Component 2: CDLinkedList

Purpose: Base class implementing a circular doubly linked list with dummy header. Provides fundamental list operations and tracks traversal counts.

Data Members: **Public (+), Private (-), and Protected (#)**

- `# DListNode *header` - Pointer to the dummy header node. Never deallocated until destructor. This node's item value is meaningless.
- `# int traverseCount`
  - Counter tracking total nodes traversed.
  - Initialized to 0 and reset to 0 by resetTraverseCount().
  - Incremented during add() and contains().
  - Allows derived classes to update count during overridden operations
- `- int currentSize` - Number of actual data nodes (excluding header). Used by getCurrentSize() and isEmpty().
- `+ CDLinkedList()`

  Constructor: Allocates dummy header node, sets header->next and header->prev to point to header itself (empty circular list), sets currentSize to 0, traverseCount to 0.
- `+ CDLinkedList(const CDLinkedList &rhs)`

  Copy constructor: Creates deep copy of rhs list. Allocates new header, then traverses rhs list and adds each element to this list using add().
- `+ ~CDLinkedList()`

  Destructor: Calls clear() to remove all data nodes, then deletes the header node.
- `+ int getCurrentSize() const`

  Returns currentSize.
- `+ bool isEmpty() const`

  Returns true if currentSize == 0, false otherwise.
- `+ bool add(int newEntry)`
  - Adds newEntry to front of list (after header) if not already present.
  - First calls contains(newEntry) to check for duplicates (this increments traverseCount).
  - If contains returns true, returns false (duplicate, don't add).
  - Otherwise, creates new DListNode with item = newEntry, inserts after header, increments currentSize, returns true.
  - Insertion logic:

    ```
    newNode->next = header->nextnewNode->prev = headerheader->next->prev = newNodeheader->next = newNode
    ```

- `+ bool remove(int anEntry)`
    - Searches for anEntry in list. If found, unlinks node and deallocates it, decrements currentSize, returns true. If not found, returns false.
    - Increments traverseCount during search.
    - Unlinking logic:

      ```
      nodeToRemove->prev->next = nodeToRemove->nextnodeToRemove->next->prev = nodeToRemove->prevdelete nodeToRemove
      ```

- `+ void clear()`
    - Removes all data nodes, leaving only the header.
    - Traverses list from header->next, deleting each node until reaching header again.
    - Sets header->next and header->prev to header, sets currentSize to 0.
    - Does not reset traverseCount.
- `+ virtual bool contains(int anEntry)`
    - Searches for anEntry in list. Returns true if found, false otherwise.
    - Starts at header->next, traverses until finding anEntry or reaching header again.
    - Increments traverseCount for each node examined (including the node being searched for if found).
    - Virtual to allow overriding in derived classes.
- `+ int getTraverseCount() const`
    - Returns current value of traverseCount.
- `+ void resetTraverseCount()`
    - Sets traverseCount to 0.
- `+ int retrieve(const int index) const`
    - Returns the item value at position index (0-based, where 0 is first real node after header).
    - Traverses to the index-th node and returns its item value.
    - If index is out of range [0, currentSize-1], behavior should return -1 or throw exception (design choice: return -1 for simplicity).
    - Does not increment traverseCount (this is a diagnostic method).

Component 3: MtfList

Purpose: Derived subclass of `CDLinkedList` that implements move-to-front strategy by overriding `CDLinkedList::contains()`.

Inheritance: `public CDLinkedList`

Data Members: None (inherits all from CDLinkedList)

Public Methods:

- `+ MtfList()`
    - Constructor: Calls base class constructor CDLinkedList().
- `+ virtual bool contains(int anEntry)` (override)
    - Searches for anEntry while incrementing traverseCount.
    - If found:
        1. Unlink the node from its current position
        2. Insert it immediately after header (front of list)
        3. Return true
    - If not found, return false.
    - Move-to-front logic (if node found and not already at front):

      ```
      // Unlink from current positionfoundNode->prev->next = foundNode->nextfoundNode->next->prev = foundNode->prev// Insert after headerfoundNode->next = header->nextfoundNode->prev = headerheader->next->prev = foundNodeheader->next = foundNode
      ```

    - Note: If found node is already at front (header->next), no movement needed but still return true.

Component 4: TransposeList

Purpose: Derived class that implements transpose strategy by overriding contains().

Inheritance: `public CDLinkedList`

Data Members: None (inherits all from CDLinkedList)

Public Methods:

- `+ TransposeList()`
    - Constructor: Calls base class constructor CDLinkedList().
- `+ virtual bool contains(int anEntry)` (override)
    - Searches for anEntry while incrementing traverseCount.
    - If found:
        1. If found node is not immediately after header (has a real predecessor):
            - Swap found node with its predecessor
        2. Return true
    - If not found, return false.
    - Swap logic (if prevNode is not header):

      ```
      // Let foundNode be the node containing anEntry// Let prevNode = foundNode->prev// Only swap if prevNode is not the headerif
      ```

```
(prevNode != header) {  // Unlink foundNode  prevNode->next =
foundNode->next  foundNode->next->prev = prevNode    // Insert
foundNode before prevNode  foundNode->prev = prevNode->prev
foundNode->next = prevNode  prevNode->prev->next = foundNode
prevNode->prev = foundNode}
```

Implementation and Test Plan

Strategy

I will use a bottom-up approach, implementing and testing the base CDLinkedList class first, then the derived classes which only override one method each. I will create three separate test driver files—one for each list type —to verify correctness and demonstrate proper multi-file project management.

1. driver_cdlist.cpp
2. driver_mtf.cpp
3. driver_transpose.cpp

Implementation Order

Phase 1: CDLinkedList - Complete Implementation

What to implement:

- DListNode struct
- Constructor, destructor, copy constructor
- `add()`, `remove()`, `clear()`, `contains()`
- `getCurrentSize()`, `isEmpty()`, `retrieve()`
- `getTraverseCount()`, `resetTraverseCount()`

How to test (driver_cdlist.cpp):

```
// Test empty list
CDLinkedList list1;
assert(list1.isEmpty() == true);
assert(list1.getCurrentSize() == 0);

// Test add and duplicates
assert(list1.add(5) == true);
assert(list1.add(10) == true);
assert(list1.add(5) == false); // Duplicate
assert(list1.getCurrentSize() == 2);
```

```
// Test contains and traverse count
list1.resetTraverseCount();
assert(list1.contains(5) == true);
assert(list1.getTraverseCount() > 0);

// Test remove
assert(list1.remove(10) == true);
assert(list1.getCurrentSize() == 1);
assert(list1.remove(999) == false); // Non-existent

// Test retrieve
assert(list1.retrieve(0) == 5);
assert(list1.retrieve(10) == -1); // Out of bounds

// Test copy constructor
CDLinkedList list2(list1);
assert(list2.getCurrentSize() == 1);
list1.add(20);
assert(list2.getCurrentSize() == 1); // Independent copy

// Test clear
list1.clear();
assert(list1.isEmpty() == true);
```

Compile and run:

```
g++ -g CDLinkedList.cpp driver_cdlist.cpp -o test_cdlist
./test_cdlist
valgrind --leak-check=full ./test_cdlist
```

Phase 2: MtfList Implementation

What to implement:

- mtflist.h and mtflist.cpp
- Inherit from CDLinkedList

- Override contains() to move accessed element to front

How to test (driver_mtf.cpp):

```
MtfList list;
for (int i = 1; i <= 5; i++) list.add(i);
// Order: 5 -> 4 -> 3 -> 2 -> 1

// Test move-to-front
assert(list.contains(1) == true);
assert(list.retrieve(0) == 1); // Moved to front

// Test repeated access improves performance
list.resetTraverseCount();
list.contains(2);
int count1 = list.getTraverseCount();

list.resetTraverseCount();
list.contains(2); // Should be at front now
int count2 = list.getTraverseCount();
assert(count2 < count1);

// Test inherited methods still work
assert(list.add(1) == false); // Duplicate
assert(list.remove(3) == true);
list.clear();
assert(list.isEmpty() == true);

// Test copy constructor
MtfList list2;
list2.add(10);
MtfList list3(list2);
assert(list3.getCurrentSize() == 1);
```

Compile and run:

```
g++ -g CDLinkedList.cpp mtflist.cpp driver_mtf.cpp -o test_mtf
```

```
./test_mtf
valgrind --leak-check=full ./test_mtf
```

Phase 3: TransposeList Implementation

What to implement:

- transposelist.h and transposelist.cpp
- Inherit from CDLinkedList
- Override contains() to swap accessed element with predecessor

How to test (driver_transpose.cpp):

```
TransposeList list;
for (int i = 1; i <= 5; i++) list.add(i);
// Order: 5 -> 4 -> 3 -> 2 -> 1

// Test swap with predecessor
assert(list.contains(1) == true);
assert(list.retrieve(3) == 1); // Swapped with 2

// Test front element (no predecessor)
assert(list.contains(5) == true);
assert(list.retrieve(0) == 5); // Stays at front

// Test gradual movement
TransposeList list2;
for (int i = 1; i <= 5; i++) list2.add(i);

int pos1 = 4; // Element 1 starts at back
list2.contains(1);
// Find new position - should be 3
int pos2 = -1;
for (int i = 0; i < 5; i++) {
    if (list2.retrieve(i) == 1) pos2 = i;
}
assert(pos2 < pos1);
```

```
// Test inherited methods
assert(list2.add(1) == false);
assert(list2.remove(3) == true);
list2.clear();
assert(list2.isEmpty() == true);

// Test copy constructor
TransposeList list3;
list3.add(10);
TransposeList list4(list3);
assert(list4.getCurrentSize() == 1);
```

Compile and run:

```
g++ -g CDLinkedList.cpp transposelist.cpp driver_transpose.cpp -o
test_transpose
./test_transpose
valgrind --leak-check=full ./test_transpose
```

Phase 4: Statistics Collection

What to do: Modify LLStats.cpp three times (once for each list type) and collect statistics:

```
# CDLinkedList statistics
g++ CDLinkedList.cpp LLStats.cpp -o stats_cdlist
./stats_cdlist > results_cdlist.txt

# MtfList statistics
g++ CDLinkedList.cpp mtflist.cpp LLStats.cpp -o stats_mtf
./stats_mtf > results_mtf.txt

# TransposeList statistics
g++ CDLinkedList.cpp transposelist.cpp LLStats.cpp -o stats_transpose
./stats_transpose > results_transpose.txt
```

Expected results:

- CDLinkedList: ~50 average traversals for both distributions
- MtfList: ~50 for uniform, much lower (~15-20) for normal
- TransposeList: ~50 for uniform, moderate improvement (~30-35) for normal

---

Files to Submit

1. CDLinkedList.h, CDLinkedList.cpp
2. mtflist.h, mtflist.cpp
3. transposelist.h, transposelist.cpp
4. driver_cdlist.cpp, driver_mtf.cpp, driver_transpose.cpp
5. Writeup (Word/PDF) with testing results and statistics analysis

Success Criteria

- All three drivers compile and pass all tests
- Valgrind reports no memory leaks for all three
- Statistics show expected performance patterns
- Code works with instructor's test files