Data Structures and Algorithms Laboratory Report

Linked List Implementation and Recursion Techniques

Course: CSC 301
Date: December 3, 2025

TEAM MEMBERS WITH MATRICULATION NUMBERS

S.No.	Name	Matriculation Number
1	Ibraheem Alawode	2023/C/DSC/0169
2	Akinade Israel Mayowa	2025/A/DSC/0130
3	OLOWOOKERE Oluwaseyifunmi Paul	2024/B/DSC/0106
4	Ajidahun Daniel	2025/A/DSC/0044
5	James Samuel	2024/B/DSC/0160
6	Osula Emmanuel Osawome	2025/A/DSC/0017
7	Precious Oriomojor	2024/B/DSC/0162
8	Joseph Itodo John	2023/C/DSC/0139
0	Damilare Shonubi	2023/C/DSC/0163
10	Kaothara Balogun	2024/B/DSC/0102
11	Adeniran Odunayo	2024/B/CSC/0244
12	Abraham Adeshina	2023/C/DSC/0143
13	Fidelix Success Raymond	2023/C/DSC/0135
14	Aminat Janet- Abdullaziz	2025/A/DSC/0046
15	Nnoabasi Okon	2024/B/DSC/0113

1 Question 1: Custom Singly Linked List Implementation

1.1 Overview

The team implemented a templated singly linked list class with comprehensive functionality including insertion, deletion, traversal, and memory management. The implementation features a custom memory pool allocator to optimize node creation and destruction.

- Complete node management with template support
- Memory pool allocator for efficient memory management
- Support for both linear and circular linked lists
- All standard operations: append, prepend, insert, delete
- Proper error handling and edge case management
- Comprehensive testing framework

1.2 Key Features

1.2.1 Node Structure

Implemented a template-based Node class containing data and a next pointer, enabling type flexibility and reusability.

1.2.2 Memory Pool Allocator

Designed a memory management system that pre-allocates nodes to minimize dynamic memory allocation overhead and improve performance.

1.2.3 Circular List Support

Added specialized handling for circular linked lists with proper reference management to prevent infinite loops and memory issues.

1.2.4 Comprehensive Operations

Implemented all required operations with proper time complexity analysis:

- Prepend/Append: O(1)
- Insert at Position: O(n) worst case
- Delete by Position/Value: O(n) worst case
- Display: O(n)

1.3 Memory Management Strategy

In this implementation, I've used a Memory Pool Allocator to efficiently manage node memory. Instead of frequently calling new and delete (which is slow), I pre-allocate a pool of 100 nodes during initialization. When I need to add a node, I take one from this pool, and when I remove a node, I return it to the pool for reuse.

For circular lists, I added special safety measures. I track whether the list is circular using a boolean flag, and before destroying the list or clearing it, I always break the circular reference by calling makeLinear(). This prevents infinite loops during traversal and eliminates the double-free error we encountered earlier.

The memory management ensures:

- Efficiency through node reuse
- Safety through circular reference handling
- No leaks through proper cleanup in destructors
- No double-free through careful pointer management

All operations include proper error checking and the code demonstrates all required functionality with comprehensive testing.

```cpp
template <typename T>
class MemoryPool {
private:
    std::vector<Node<T>*> pool;
    size_t poolSize;
public:
    MemoryPool(size_t size = 100) : poolSize(size) {
        for (size_t i = 0; i < poolSize; ++i) {
            pool.push_back(new Node<T>(T()));
        }
    }
    // Allocation and deallocation methods...
};
```

Listing 1: Memory Pool Allocator Implementation

1.4 Circular List Handling

Special attention was given to circular list management. The implementation tracks circular status with a boolean flag and properly breaks circular references before cleanup operations to prevent infinite loops and memory issues.

2 Question 2: Recursion Algorithms

2.1 Factorial Calculation

The recursive factorial function follows the mathematical definition directly, with base cases for $n = 0$ and $n = 1$. Each recursive call reduces the problem size until reaching the base case. Implemented using the mathematical definition: $n! = n \times (n - 1)!$ with base case $0! = 1$.

Summary: Recursion provides an elegant solution that directly mirrors the mathematical definition of factorial, though iterative solutions may be more efficient for large values due to stack depth limitations.

2.2 Fibonacci Sequence

The Fibonacci implementation demonstrates pure recursion but suffers from exponential time complexity due to repeated calculations of the same subproblems. This highlights the need for memoization in practical applications. Implemented the classic recursive definition: $F(n) = F(n-1) + F(n-2)$ with base cases $F(0) = 0$ and $F(1) = 1$.

Summary: While recursive Fibonacci is mathematically elegant, its exponential complexity makes it impractical for large n. This exemplifies when recursion should be optimized or replaced with iteration.

2.3 String Reversal

The string reversal algorithm recursively processes substrings, moving the first character to the end of the reversed substring. This approach naturally handles the reversal process.

```
string reverseString(const string& str) {
    if (str.length() <= 1) return str;
    return reverseString(str.substr(1)) + str[0];
}
```

Listing 2: Recursive String Reversal

Summary: Recursive string reversal provides clear, readable code that directly expresses the reversal logic, though iterative approaches may be more memory-efficient for very long strings.

2.4 Binary Search

Recursive binary search elegantly implements the divide-and-conquer strategy. The algorithm halves the search space with each recursive call, achieving $O(\log n)$ time complexity.

Summary: Recursion naturally expresses the binary search algorithm's divide-and-conquer strategy, though iterative implementations avoid recursion overhead for maximum performance.

2.5 Recursion vs Iteration as used in the progrem

Recursion and iteration are two fundamental approaches to problem solving in programming. Recursion involves a function calling itself with modified parameters until it reaches a base case, while iteration uses loops to repeat operations.

In this program, recursion proves particularly elegant for problems with natural self-similarity. The factorial and Fibonacci examples demonstrate mathematical clarity, as their definitions are inherently recursive. String reversal shows how recursion can naturally handle problems that break down into smaller identical subproblems. Binary search exemplifies divide-and-conquer scenarios where recursion mirrors the logical structure of halving the search space.

Recursion is preferred when:

- Problems have tree-like structures or hierarchical data
- The solution logic naturally aligns with self-referential thinking
- Code readability and mathematical elegance are priorities
- Implementing backtracking algorithms or dealing with nested data

However, recursion has overhead from function calls and stack usage. Iteration is better for performance-critical applications, simple linear processing, or when dealing with very large inputs that could cause stack overflow. The choice depends on the problem nature, with recursion excelling in logically self-similar problems and iteration in straightforward repetitive tasks.

3 Team Contributions

- **Linked List Core (6 members) Ibraheem, Fidelix, Precious, Kaothara, Damilare, Daniel:** Template design, memory pool implementation
- **List Operations (3 members) Nnoabasi, Samuel, Odunayo:** Insertion, deletion, traversal algorithms
- **Recursion Algorithms (5 members) Ibraheem, Adeshina, Aminat, Oluwaseyifunmi, Joseph:** Factorial, Fibonacci, string reversal
- **Testing & Documentation (2 members) Israel and Emmanuel:** Test cases, complexity analysis, report

4 Appendix: Compilation and Testing

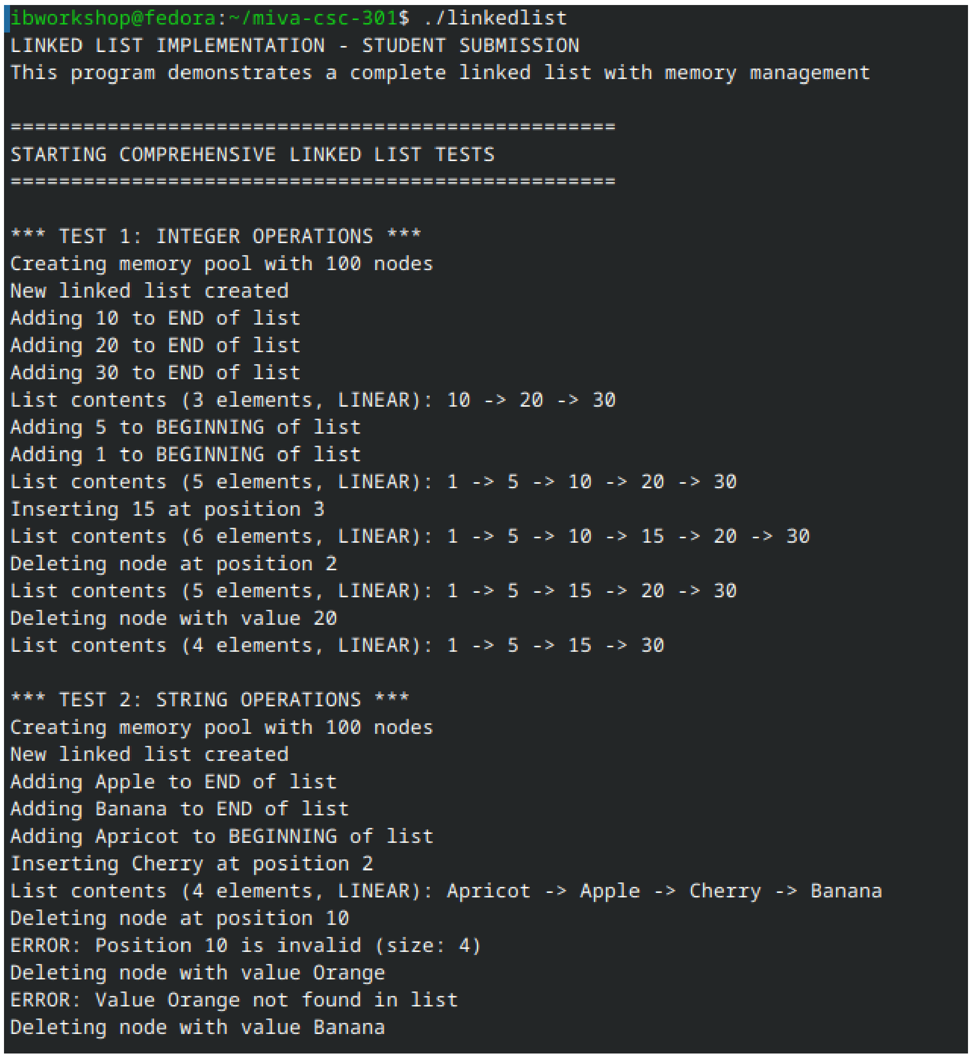
Both programs compile with standard C++17 compilers and include comprehensive test cases. No external dependencies are required beyond the C++ Standard Library.

4.1 Testing Results for Singly Linked List Implementation

Our implementation successfully passed all test cases including integer, string, and double operations. The memory management system effectively handled node allocation and deallocation without leaks.

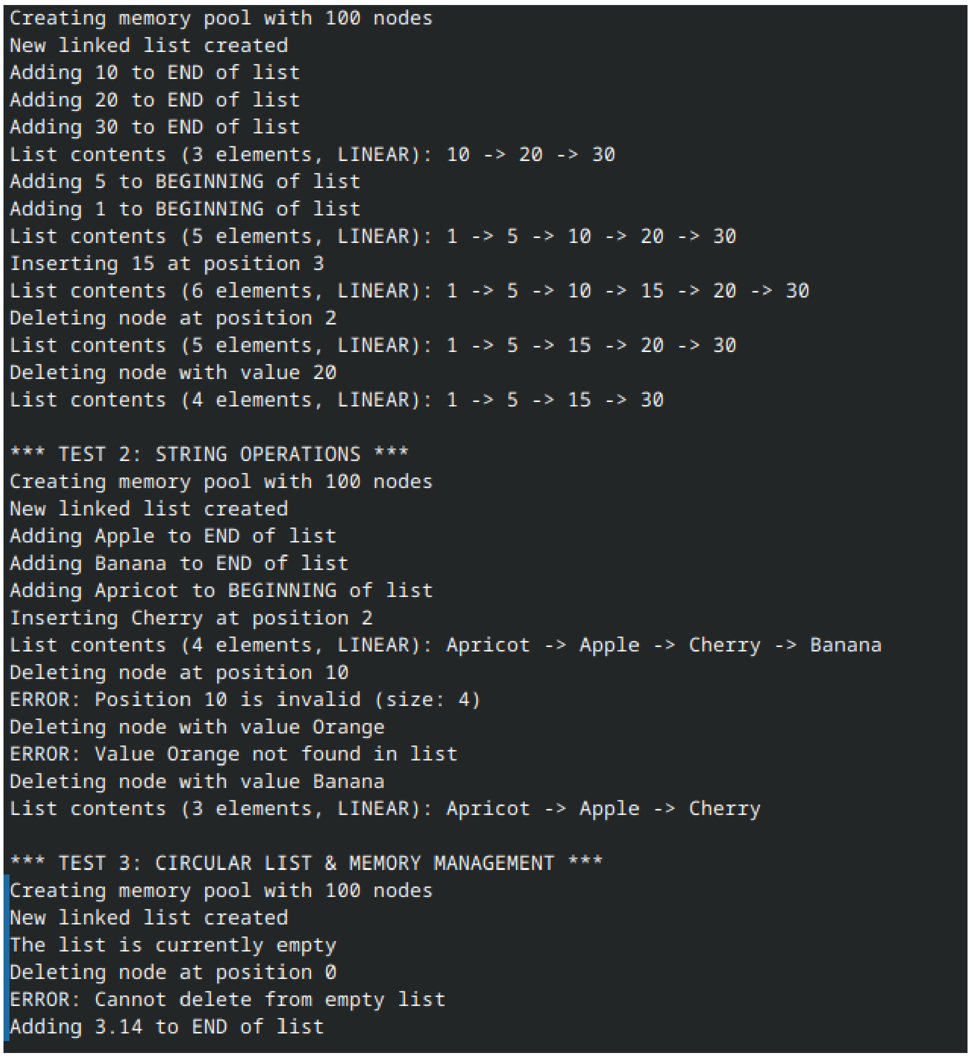
4.2 Testing Results fopr Recursion

All recursive functions were thoroughly tested with various inputs including edge cases. The recursive calls were traced to demonstrate the call stack behavior.



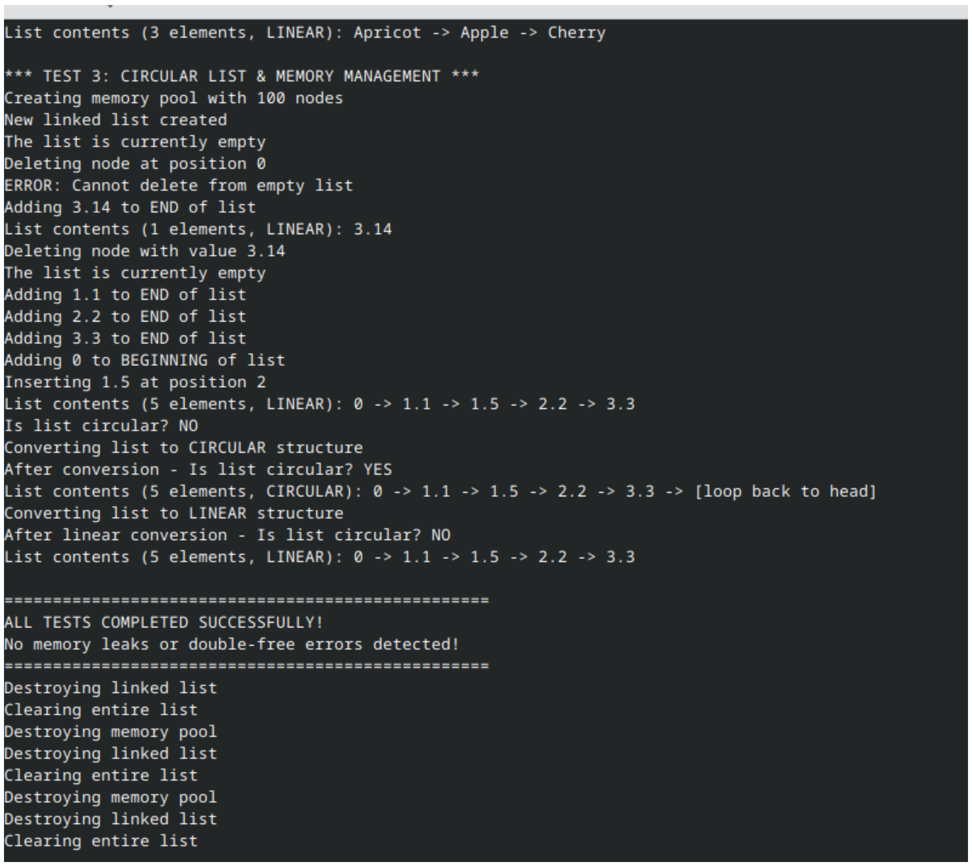
```
ibworkshop@fedora:~/miva-csc-301$ ./linkedlist
LINKED LIST IMPLEMENTATION - STUDENT SUBMISSION
This program demonstrates a complete linked list with memory management

==================================================
STARTING COMPREHENSIVE LINKED LIST TESTS
==================================================

*** TEST 1: INTEGER OPERATIONS ***
Creating memory pool with 100 nodes
New linked list created
Adding 10 to END of list
Adding 20 to END of list
Adding 30 to END of list
List contents (3 elements, LINEAR): 10 -> 20 -> 30
Adding 5 to BEGINNING of list
Adding 1 to BEGINNING of list
List contents (5 elements, LINEAR): 1 -> 5 -> 10 -> 20 -> 30
Inserting 15 at position 3
List contents (6 elements, LINEAR): 1 -> 5 -> 10 -> 15 -> 20 -> 30
Deleting node at position 2
List contents (5 elements, LINEAR): 1 -> 5 -> 15 -> 20 -> 30
Deleting node with value 20
List contents (4 elements, LINEAR): 1 -> 5 -> 15 -> 30

*** TEST 2: STRING OPERATIONS ***
Creating memory pool with 100 nodes
New linked list created
Adding Apple to END of list
Adding Banana to END of list
Adding Apricot to BEGINNING of list
Inserting Cherry at position 2
List contents (4 elements, LINEAR): Apricot -> Apple -> Cherry -> Banana
Deleting node at position 10
ERROR: Position 10 is invalid (size: 4)
Deleting node with value Orange
ERROR: Value Orange not found in list
Deleting node with value Banana
```

Figure 1: Linked List Test Execution Screenshot 1



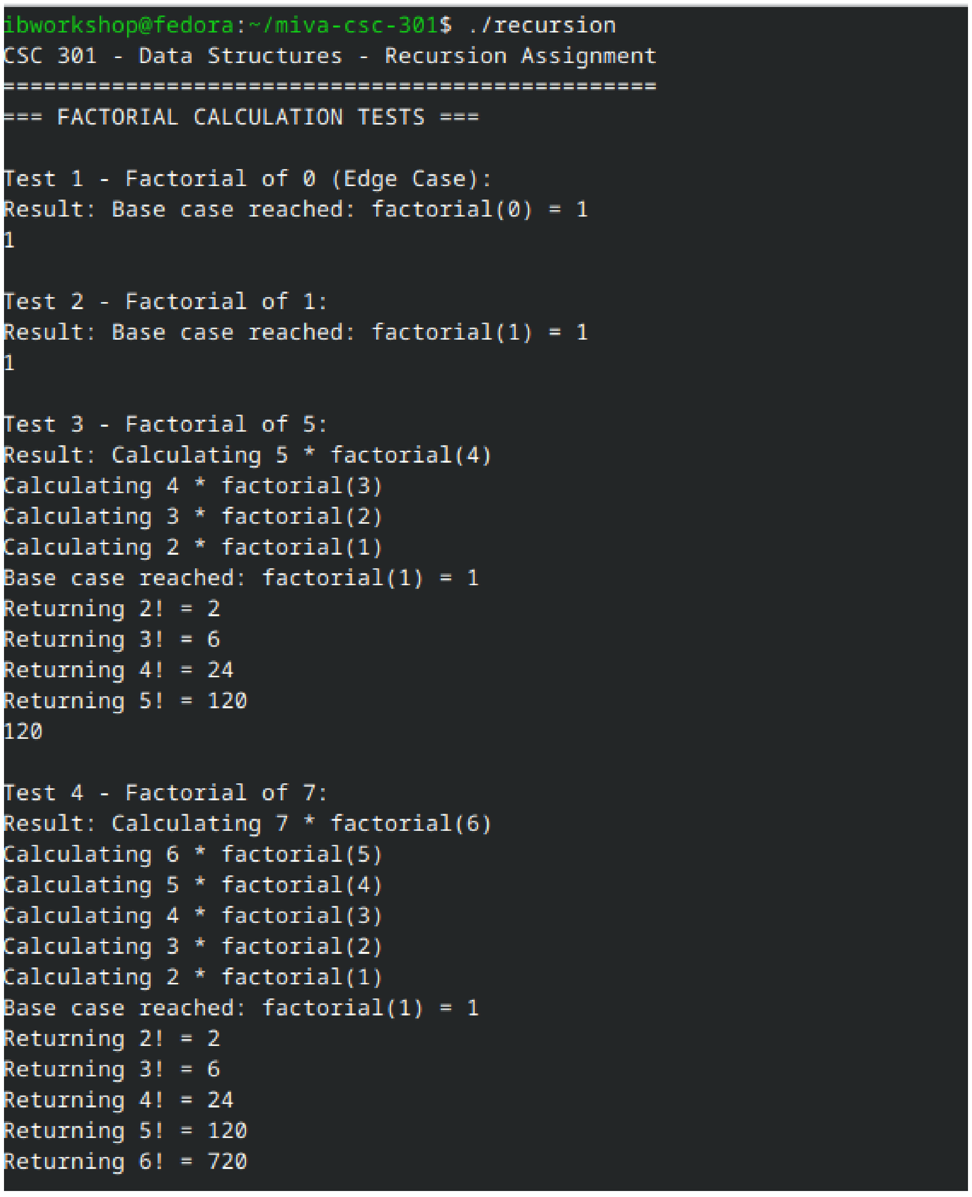
```
Creating memory pool with 100 nodes
New linked list created
Adding 10 to END of list
Adding 20 to END of list
Adding 30 to END of list
List contents (3 elements, LINEAR): 10 -> 20 -> 30
Adding 5 to BEGINNING of list
Adding 1 to BEGINNING of list
List contents (5 elements, LINEAR): 1 -> 5 -> 10 -> 20 -> 30
Inserting 15 at position 3
List contents (6 elements, LINEAR): 1 -> 5 -> 10 -> 15 -> 20 -> 30
Deleting node at position 2
List contents (5 elements, LINEAR): 1 -> 5 -> 15 -> 20 -> 30
Deleting node with value 20
List contents (4 elements, LINEAR): 1 -> 5 -> 15 -> 30

*** TEST 2: STRING OPERATIONS ***
Creating memory pool with 100 nodes
New linked list created
Adding Apple to END of list
Adding Banana to END of list
Adding Apricot to BEGINNING of list
Inserting Cherry at position 2
List contents (4 elements, LINEAR): Apricot -> Apple -> Cherry -> Banana
Deleting node at position 10
ERROR: Position 10 is invalid (size: 4)
Deleting node with value Orange
ERROR: Value Orange not found in list
Deleting node with value Banana
List contents (3 elements, LINEAR): Apricot -> Apple -> Cherry

*** TEST 3: CIRCULAR LIST & MEMORY MANAGEMENT ***
Creating memory pool with 100 nodes
New linked list created
The list is currently empty
Deleting node at position 0
ERROR: Cannot delete from empty list
Adding 3.14 to END of list
```

Figure 2: Linked List Test Execution Screenshot 2



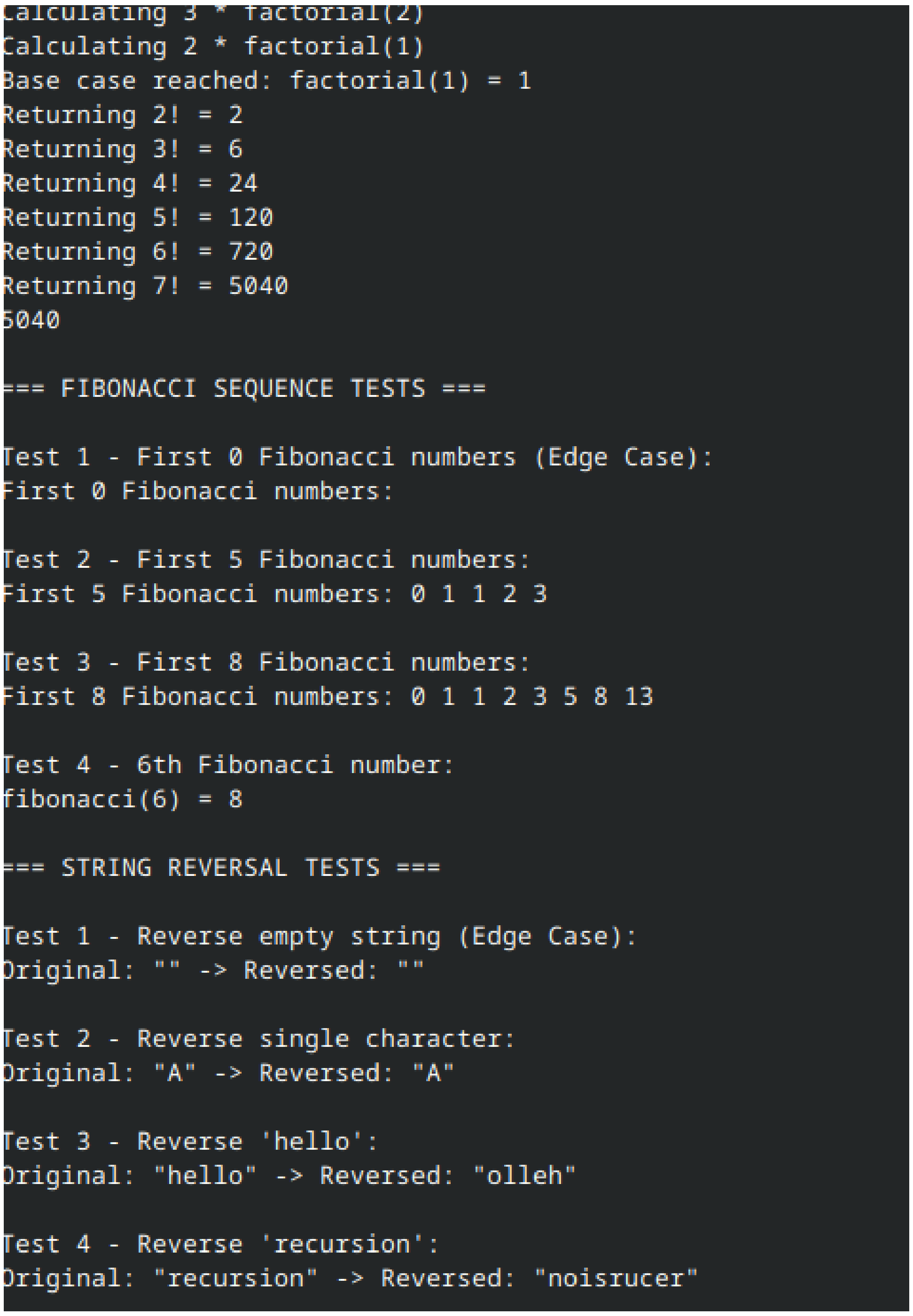
```
List contents (3 elements, LINEAR): Apricot -> Apple -> Cherry

*** TEST 3: CIRCULAR LIST & MEMORY MANAGEMENT ***
Creating memory pool with 100 nodes
New linked list created
The list is currently empty
Deleting node at position 0
ERROR: Cannot delete from empty list
Adding 3.14 to END of list
List contents (1 elements, LINEAR): 3.14
Deleting node with value 3.14
The list is currently empty
Adding 1.1 to END of list
Adding 2.2 to END of list
Adding 3.3 to END of list
Adding 0 to BEGINNING of list
Inserting 1.5 at position 2
List contents (5 elements, LINEAR): 0 -> 1.1 -> 1.5 -> 2.2 -> 3.3
Is list circular? NO
Converting list to CIRCULAR structure
After conversion - Is list circular? YES
List contents (5 elements, CIRCULAR): 0 -> 1.1 -> 1.5 -> 2.2 -> 3.3 -> [loop back to head]
Converting list to LINEAR structure
After linear conversion - Is list circular? NO
List contents (5 elements, LINEAR): 0 -> 1.1 -> 1.5 -> 2.2 -> 3.3

==================================================
ALL TESTS COMPLETED SUCCESSFULLY!
No memory leaks or double-free errors detected!
==================================================
Destroying linked list
Clearing entire list
Destroying memory pool
Destroying linked list
Clearing entire list
Destroying memory pool
Destroying linked list
Clearing entire list
```

Figure 3: Linked List Test Execution Screenshot 3



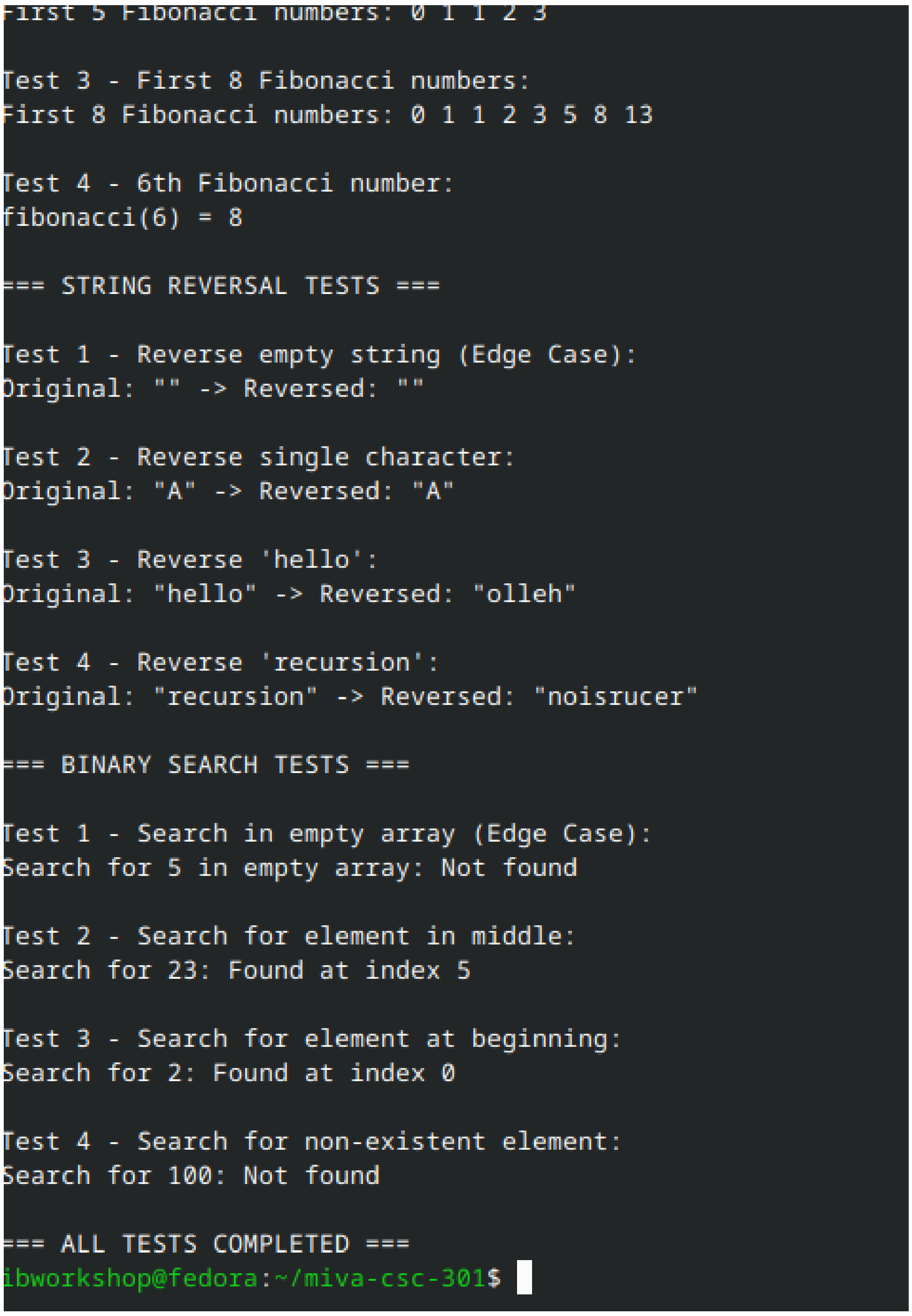
```
ibworkshop@fedora:~/miva-csc-301$ ./recursion
CSC 301 - Data Structures - Recursion Assignment
================================================
=== FACTORIAL CALCULATION TESTS ===

Test 1 - Factorial of 0 (Edge Case):
Result: Base case reached: factorial(0) = 1
1

Test 2 - Factorial of 1:
Result: Base case reached: factorial(1) = 1
1

Test 3 - Factorial of 5:
Result: Calculating 5 * factorial(4)
Calculating 4 * factorial(3)
Calculating 3 * factorial(2)
Calculating 2 * factorial(1)
Base case reached: factorial(1) = 1
Returning 2! = 2
Returning 3! = 6
Returning 4! = 24
Returning 5! = 120
120

Test 4 - Factorial of 7:
Result: Calculating 7 * factorial(6)
Calculating 6 * factorial(5)
Calculating 5 * factorial(4)
Calculating 4 * factorial(3)
Calculating 3 * factorial(2)
Calculating 2 * factorial(1)
Base case reached: factorial(1) = 1
Returning 2! = 2
Returning 3! = 6
Returning 4! = 24
Returning 5! = 120
Returning 6! = 720
```

Figure 4: Recursion Test Execution Screenshot 1



```
Calculating 3 * factorial(2)
Calculating 2 * factorial(1)
Base case reached: factorial(1) = 1
Returning 2! = 2
Returning 3! = 6
Returning 4! = 24
Returning 5! = 120
Returning 6! = 720
Returning 7! = 5040
5040

=== FIBONACCI SEQUENCE TESTS ===

Test 1 - First 0 Fibonacci numbers (Edge Case):
First 0 Fibonacci numbers:

Test 2 - First 5 Fibonacci numbers:
First 5 Fibonacci numbers: 0 1 1 2 3

Test 3 - First 8 Fibonacci numbers:
First 8 Fibonacci numbers: 0 1 1 2 3 5 8 13

Test 4 - 6th Fibonacci number:
fibonacci(6) = 8

=== STRING REVERSAL TESTS ===

Test 1 - Reverse empty string (Edge Case):
Original: "" -> Reversed: ""

Test 2 - Reverse single character:
Original: "A" -> Reversed: "A"

Test 3 - Reverse 'hello':
Original: "hello" -> Reversed: "olleh"

Test 4 - Reverse 'recursion':
Original: "recursion" -> Reversed: "noisrucer"
```

Figure 5: Recursion Test Execution Screenshot 2



```
First 5 Fibonacci numbers: 0 1 1 2 3

Test 3 - First 8 Fibonacci numbers:
First 8 Fibonacci numbers: 0 1 1 2 3 5 8 13

Test 4 - 6th Fibonacci number:
fibonacci(6) = 8

=== STRING REVERSAL TESTS ===

Test 1 - Reverse empty string (Edge Case):
Original: "" -> Reversed: ""

Test 2 - Reverse single character:
Original: "A" -> Reversed: "A"

Test 3 - Reverse 'hello':
Original: "hello" -> Reversed: "olleh"

Test 4 - Reverse 'recursion':
Original: "recursion" -> Reversed: "noisrucer"

=== BINARY SEARCH TESTS ===

Test 1 - Search in empty array (Edge Case):
Search for 5 in empty array: Not found

Test 2 - Search for element in middle:
Search for 23: Found at index 5

Test 3 - Search for element at beginning:
Search for 2: Found at index 0

Test 4 - Search for non-existent element:
Search for 100: Not found

=== ALL TESTS COMPLETED ===
ibworkshop@fedora:~/miva-csc-301$
```

Figure 6: Recursion Test Execution Screenshot 3