Homework 5: Linked List & Recursive

CS 219 - Advanced Data Structures

November 21, 2024

Due Date: [12/03/2024 11:59 PM]

Abstract

This assignment aims to deepen your understanding of linked lists in C++. You will work with two programs that implement singly and doubly linked lists. By completing this assignment, you will:

- Gain hands-on experience with dynamic memory allocation and pointers.
- Implement and manipulate linked list structures.
- Enhance problem-solving skills by handling various edge cases.

Contents

Instructions

1. Read each question carefully and adhere to the specified requirements.
2. Write clean, well-documented code with appropriate comments.
3. Ensure proper memory management to prevent leaks.
4. Submit your source code files (`.cpp` files) along with any necessary documentation.
5. Include output screenshots or logs demonstrating your program's functionality.

1 Part 1: Movie List Management (Doubly Linked List)

Overview

You are provided with a C++ program that manages a list of movies using a **doubly linked list**. The program includes a `MovieList` class with methods to insert movies in various ways and display the list.

Objectives

- Implement additional functionalities in a doubly linked list.
- Handle edge cases in linked list operations.
- Practice dynamic memory management and pointer manipulation.

UML Diagram

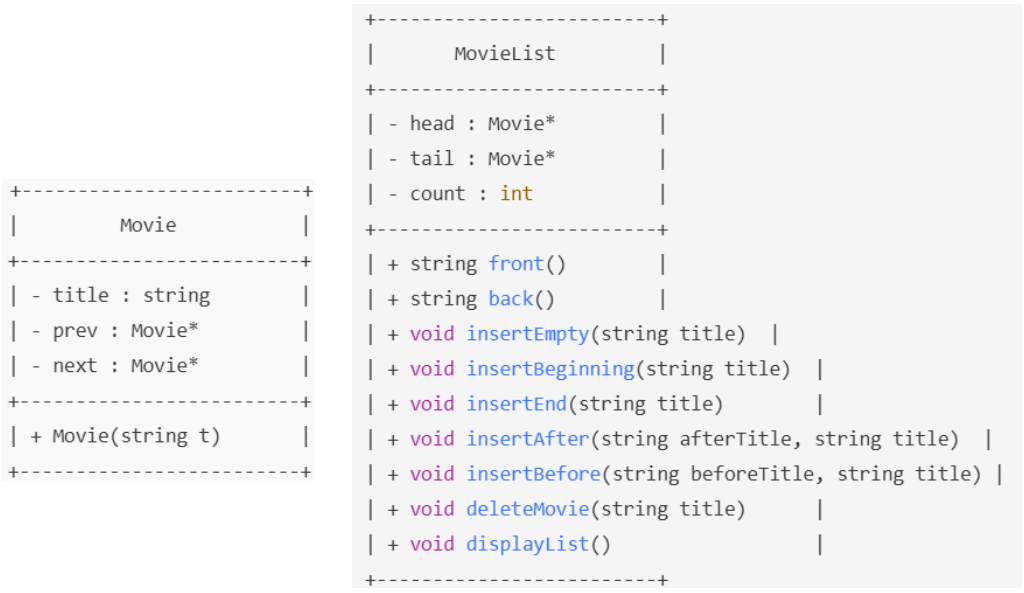

Figure 1: UML Diagram for Question 1

Requirements

1. **Review the Provided Code**

   Examine the provided code carefully, paying attention to the structure of the `Movie` struct and the `MovieList` class.

2. **Implement Additional Features**

   a. **Delete a Movie by Title**

   - **Method Signature**:

   ```
   void deleteMovie(string title);
   ```

   - **Functionality**:
     - Search for the movie with the given `title`.
     - Remove it from the list by updating the `prev` and `next` pointers of neighboring nodes.
     - Handle edge cases where the movie is at the head or tail.
     - If the movie is not found, display an appropriate message.
   - **Output Messages**:
     - If deleted: `Deleted movie: "<title>".`
     - If not found: `Movie "<title>" not found!`

   b. **Insert Before a Specific Movie**

   - **Method Signature**:

   ```
   void insertBefore(string beforeTitle, string title);
   ```

   - **Functionality**:
     - Search for the movie with `beforeTitle`.
     - Insert a new movie with `title` before it.
     - Update the `prev` and `next` pointers accordingly.
     - Handle edge cases where the movie is not found or is at the head.
   - **Output Messages**:
     - If inserted: `Inserted "<title>" before "<beforeTitle>".`
     - If not found: `Movie "<beforeTitle>" not found!`

3. **Test Your Methods**

   Modify the `main()` function to include test cases for your new methods. Use the sample test cases provided or create your own.

4. **Handle Edge Cases**

   Ensure your methods handle the following scenarios:

   - Empty list operations.
   - Single-node list operations.
   - Deletion or insertion at the head or tail.

5. **Document Your Code**

   Add comments to explain your logic, especially in complex sections. Ensure your code is readable and follows consistent styling conventions.

Implementation Guidelines

- **Understand the Structure**

  The `Movie` struct and `MovieList` class are defined as:

```cpp
struct Movie {
    string title;
    Movie* prev;
    Movie* next;

    Movie(string t) : title(t), prev(nullptr), next(nullptr) {}
};

class MovieList {
private:
    Movie* head;
    Movie* tail;
    int count;
    // ...
};
```

- **Method Details**

  deleteMovie(string title):

  - Traverse the list to find the movie.
  - Update pointers to exclude the node.
  - Delete the node to free memory.
  - Update `head` or `tail` if necessary.

  insertBefore(string beforeTitle, string title):

  - Find the node with `beforeTitle`.
  - Create a new node with `title`.
  - Insert the new node before the found node.
  - Update `head` if inserting at the beginning.

Sample Test Cases and Expected Output

1. **Delete a Movie in the Middle**

```cpp
// Delete a movie in the middle
movieList.deleteMovie("Inception");
movieList.displayList();
```

Expected Output:

```
Deleted movie: "Inception".
Movie List: Interstellar -> The Dark Knight -> Dunkirk -> END
```

2. **Delete a Movie That Doesn't Exist**

```
// Delete a movie that doesn't exist
movieList.deleteMovie("Avatar");
movieList.displayList();
```

Expected Output:

```
Movie "Avatar" not found!
Movie List: Interstellar -> The Dark Knight -> Dunkirk -> END
```

3. **Insert Before the First Movie**

```
// Insert before the first movie
movieList.insertBefore("Interstellar", "Tenet");
movieList.displayList();
```

Expected Output:

```
Inserted "Tenet" at the beginning.
Movie List: Tenet -> Interstellar -> The Dark Knight -> Dunkirk ->
    END
```

4. **Insert Before a Movie That Doesn't Exist**

```
// Insert before a movie that doesn't exist
movieList.insertBefore("Avatar", "Memento");
movieList.displayList();
```

Expected Output:

```
Movie "Avatar" not found!
Movie List: Tenet -> Interstellar -> The Dark Knight -> Dunkirk ->
    END
```

Deliverables

- Updated `MovieList` class with `deleteMovie` and `insertBefore` methods.
- Modified `main()` function demonstrating the new methods.
- Output screenshots or console logs showing the results of your tests.

2 Part 2: Task List Management (Singly Linked List)

Overview

The second program manages a list of tasks using a **singly linked list**. It includes a `TaskList` class with methods to add, delete, remove, and swap tasks.

Objectives

- Enhance a singly linked list with additional functionalities.
- Practice pointer manipulation and list traversal.
- Handle edge cases and ensure robustness.

UML Diagram

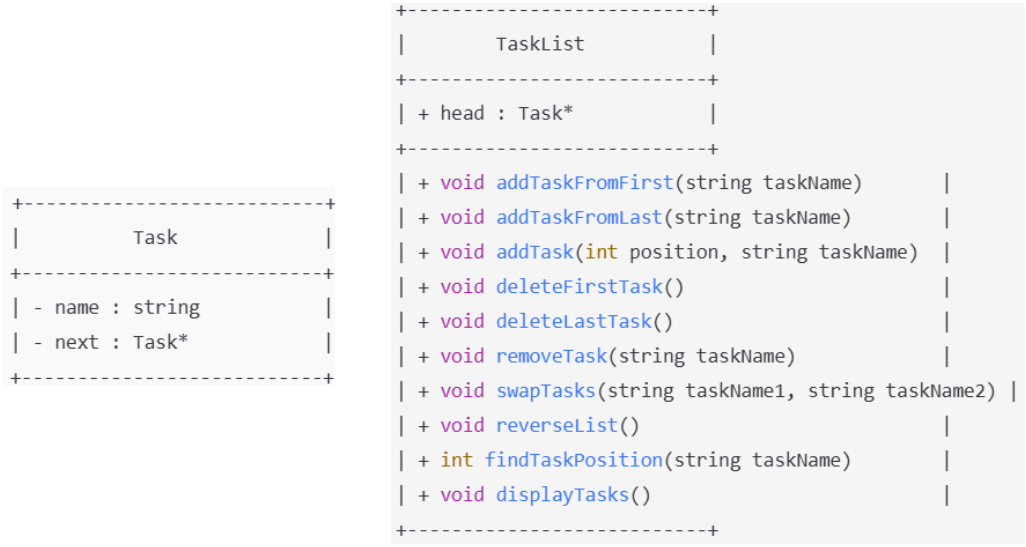

Figure 2: UML Diagram for Question 2

Requirements

1. **Examine the Provided Code**

   Review the `Task` struct and `TaskList` class to understand the current implementation.

2. **Enhance the Task List**

   a. **Reverse the Task List**

   - **Method Signature**:

   ```
   1 void reverseList();
   ```

   - **Functionality**:
     - Reverse the order of the tasks in the list.

  - Update the `next` pointers of each node accordingly.
  - Handle cases where the list is empty or has only one node.
- **Output Message**:
  - `Reversed the task list.`

b. **Find Task Position by Name**

- **Method Signature**:

```
int findTaskPosition(string taskName);
```

- **Functionality**:
  - Traverse the list to find the task with `taskName`.
  - Return its position (0-based index).
  - If the task is not found, return `-1`.
- **Output Messages**:
  - If found: `Task "<taskName>" found at position <position>.`
  - If not found: `Task "<taskName>" not found.`

3. **Test Your Enhancements**

   Modify the `main()` function to include test cases for the new methods. Use the sample test cases provided or create your own.

4. **Edge Case Handling**

   Ensure your methods handle the following scenarios:

   - Reversing an empty list.
   - Reversing a single-node list.
   - Finding the position of tasks at different positions.
   - Attempting to find a task that doesn't exist.

5. **Document Your Code**

   Add comments to explain your logic. Ensure your code is readable and follows consistent styling conventions.

Implementation Guidelines

- **Understand the Structure**

  The `Task` struct and `TaskList` class are defined as:

```
struct Task {
    string name;
    Task* next;
};

class TaskList {
public:
    Task* head;
    // ...
};
```

- **Method Details**

  reverseList():

  - Use three pointers: `prev`, `current`, and `next`.
  - Iterate through the list, reversing the `next` pointers.
  - Update `head` to the new front of the list.

  findTaskPosition(string taskName):

  - Initialize a position counter.
  - Traverse the list comparing each node's `name` with `taskName`.
  - Return the position if found; otherwise, return `-1`.

Sample Test Cases and Expected Output

1. **Reverse the Task List**

```
// Add tasks
tasks.addTaskFromFirst("Task A");
tasks.addTaskFromLast("Task B");
tasks.addTaskFromLast("Task C");
tasks.addTaskFromLast("Task D");
tasks.displayTasks();

// Reverse the list
tasks.reverseList();
tasks.displayTasks();
```

   Expected Output:

```
Added task at the start: Task A
Added task at the end: Task B
Added task at the end: Task C
Added task at the end: Task D
Task A -> Task B -> Task C -> Task D -> NULL

Reversed the task list.
Task D -> Task C -> Task B -> Task A -> NULL
```

2. **Find Task Positions**

```
// Find positions
int pos = tasks.findTaskPosition("Task A");
if (pos != -1)
    cout << "Position of 'Task A': " << pos << endl;
else
    cout << "'Task A' not found." << endl;

pos = tasks.findTaskPosition("Task X");
if (pos != -1)
    cout << "Position of 'Task X': " << pos << endl;
else
    cout << "'Task X' not found." << endl;
```

Expected Output:

```
Task "Task A" found at position 3.
Position of 'Task A': 3
Task "Task X" not found.
'Task X' not found.
```

3. **Edge Cases**

   Reversing an Empty List:

```
TaskList emptyTasks;
emptyTasks.reverseList();
emptyTasks.displayTasks();
```

   Expected Output:

```
Reversed the task list.
NULL
```

   Reversing a Single-Node List:

```
TaskList singleTask;
singleTask.addTaskFromFirst("Only Task");
singleTask.reverseList();
singleTask.displayTasks();
```

   Expected Output:

```
Added task at the start: Only Task
Reversed the task list.
Only Task -> NULL
```

Deliverables

- Updated `TaskList` class with `reverseList` and `findTaskPosition` methods.
- Modified `main()` function demonstrating the new methods.
- Output screenshots or console logs showing the results of your tests.

3 Question 3: Recursive Fibonacci Execution Time Analysis

Description

In this question, you will analyze the execution time of a recursive Fibonacci function. By measuring the execution time for different input values, you will observe how the computational complexity of recursive algorithms affects performance. You will then plot the execution times and explain your observations based on the nature of the recursive Fibonacci function.

Objectives

- Implement a recursive Fibonacci function in C++.
- Measure and record the execution time for varying input sizes.
- Plot the relationship between input size and execution time.
- Analyze and explain the observed execution times based on algorithm complexity.

Requirements

1. **Implement the Recursive Fibonacci Function**
   - Use the provided code structure to implement the recursive function `rFibNum(int a, int b, int n)`.
   - `a` and `b` represent the first two numbers in the sequence (e.g., `a = 1`, `b = 2`).
   - `n` is the position in the sequence for which you want to compute the Fibonacci number.
2. **Measure Execution Time**
   - Modify the `main()` function to execute `rFibNum()` with `n = 10`, `n = 20`, `n = 30`, and `n = 40`.
   - Use the `chrono` library to measure the execution time in milliseconds.
   - Record the execution time for each value of `n`.
3. **Plot the Execution Time**
   - Create a graph (e.g., line chart) that shows the relationship between the input size (`n`) and the execution time.
   - Label the axes appropriately and include units.
4. **Explain Observations**
   - Based on the graph, explain why the execution time increases with larger values of `n`.
   - Discuss the time complexity of the recursive Fibonacci function and how it impacts performance.

Implementation Guidelines

- **Recursive Fibonacci Function**

  Implement the recursive Fibonacci function as per the following code:

```cpp
long long int rFibNum(int a, int b, int n) {
    if (n == 1) return a;
    if (n == 2) return b;
    return rFibNum(a, b, n - 1) + rFibNum(a, b, n - 2);
}
```

- **Measuring Execution Time**

  Use the `chrono` library to measure the execution time:

```cpp
#include <chrono>

// Inside your main function
auto start = std::chrono::high_resolution_clock::now();

// Call the rFibNum function
long long int result = rFibNum(a, b, n);

auto end = std::chrono::high_resolution_clock::now();
auto duration = std::chrono::duration_cast<std::chrono::
    milliseconds>(end - start).count();
std::cout << "Result: " << result << std::endl;
std::cout << "Execution time: " << duration << " ms" << std::endl;
```

- **Handling Large Numbers**

  - Since Fibonacci numbers grow exponentially, use `long long int` to handle larger values.

- **Sample Main Function**

  Here is a sample `main()` function structure:

```cpp
int main() {
    int a = 1, b = 2;
    int n_values[] = {10, 20, 30, 40};

    for (int n : n_values) {
        auto start = std::chrono::high_resolution_clock::now();
        long long int result = rFibNum(a, b, n);
        auto end = std::chrono::high_resolution_clock::now();

        auto duration = std::chrono::duration_cast<std::chrono::
            milliseconds>(end - start).count();
        std::cout << "n = " << n << ", Result: " << result << ",
            Execution time: " << duration << " ms" << std::endl;
    }

    return 0;
}
```

- **Graph Plotting**

  - You can use software like Microsoft Excel, Google Sheets, or programming libraries like Matplotlib (Python) or MATLAB to plot the graph.
  - Ensure the graph is clear, with labeled axes and a descriptive title.

Example Output

```
n = 10, Result: 143, Execution time: 0 ms
n = 20, Result: 17711, Execution time: 1 ms
n = 30, Result: 2178309, Execution time: 16 ms
n = 40, Result: 267914296, Execution time: 180 ms
```

Explanation of Observations

- The execution time increases exponentially with the input size `n`.
- The recursive Fibonacci function has exponential time complexity $O(2^n)$ because it makes two recursive calls for each non-base case.
- As `n` increases, the number of function calls grows exponentially, leading to longer execution times.
- This inefficiency is due to the repeated calculations of the same subproblems (overlapping subproblems).

Deliverables

- Source code of your program (`.cpp` file) with the implemented recursive Fibonacci function and execution time measurement.
- A document or image containing the graph plotting execution time against input size `n`.
- A written explanation of your observations based on the graph.

Tips and Hints

- **Understanding Pointers**:

  Be cautious with pointer assignments and updates, especially in deletion and reversal operations. Draw diagrams to visualize pointer changes during insertions and deletions.

- **Edge Cases**:

  Always consider what happens when your list is empty, has one node, or when operations involve the head or tail nodes. Test your methods thoroughly with different scenarios.

- **Testing**:

  Use the provided expected outputs to verify your program's correctness. Add additional test cases to cover more scenarios.

- **Debugging**:

  Use debugging tools or insert `cout` statements to trace the execution flow if you encounter issues. Check for memory leaks using tools like Valgrind (if available).

- **Resources**:

  Refer to your textbook or online resources for additional explanations on linked list operations. Standard C++ documentation can help with understanding language features.

Academic Integrity

Ensure that all work submitted is your own. Plagiarism or any form of academic dishonesty will result in disciplinary action as per university policies.

By completing this assignment, you'll gain valuable experience with linked lists, which are fundamental data structures in computer science. Good luck, and don't hesitate to reach out if you have any questions!

Happy coding!