Instituto Tecnológico de Costa Rica

Área Académica de Ingeniería en Computadores

Algoritmos y Estructuras de Datos II (CE 2103)

Documento de diseño

Primer proyecto: MPointers

Profesor:
Jose Isaac Ramírez Herrera

Estudiantes:
Jestim Felix Espinoza Morales –

Tabla de contenidos

Introducción

El proyecto MPointer es una biblioteca desarrollada en C++ que tiene como objetivo gestionar eficientemente los punteros mediante la encapsulación en una clase template, permitiendo un mejor manejo de memoria a través de la automatización de la gestión de punteros. Esta solución está orientada a mejorar la confiabilidad y eficiencia en el uso de punteros en aplicaciones C++ mediante el uso de un recolector de basura personalizado que monitorea y libera la memoria de los punteros cuando ya no se utilizan. Esta documentación describe el diseño, las decisiones tomadas y los desafíos encontrados en el desarrollo de esta biblioteca, incluyendo la implementación de algoritmos de ordenamiento para listas doblemente enlazadas que utilizan `MPointer`.

Breve descripción del problema

La gestión manual de memoria en C++ es un proceso propenso a errores, como fugas de memoria y dobles liberaciones. Estos problemas se vuelven más difíciles de manejar a medida que los proyectos crecen en complejidad. Para solucionar estos problemas, se hace necesario un sistema que automatice la gestión de punteros, proporcionando seguridad y eficiencia sin requerir la intervención directa del usuario. Además, al trabajar con estructuras de datos como listas enlazadas, es crucial contar con algoritmos de ordenamiento eficientes que puedan operar sobre punteros gestionados de manera segura.

Descripción de la solución propuesta

El proyecto MPointer propone una biblioteca en C++ que encapsula los punteros en la clase `MPointer<T>`, proporcionando un recolector de basura (`MPointerGC`) para gestionar automáticamente la memoria asignada. Además, la biblioteca ofrece sobrecarga de operadores para facilitar el uso intuitivo de los punteros y soporta manejo de referencias y conteo de referencias. El diseño modular y extensible permite que la biblioteca se integre en aplicaciones más grandes sin complicaciones.

La biblioteca también incluye la implementación de una lista doblemente enlazada (`DoublyLinkedList`) que utiliza `MPointer` para almacenar sus nodos. Esta lista ofrece algoritmos de ordenamiento como Bubble Sort, Insertion Sort y Quick Sort, adaptados para trabajar con la gestión de memoria proporcionada por `MPointer`.

Decisiones de diseño

Alternativas consideradas:

1. Gestión de memoria manual o automática:

Se evaluó la posibilidad de dejar la gestión de punteros completamente en manos del programador, lo que hubiera requerido un control manual exhaustivo de la memoria. Sin embargo, esto aumentaría la probabilidad de errores.

2. Estructuras para la gestión de punteros:

Se consideraron estructuras como smart pointers ya existentes en C++ (como `std::shared_ptr` y `std::unique_ptr`), pero se optó por implementar una solución personalizada que ofreciera más control sobre la administración de la memoria y una mayor flexibilidad.

3. Recolector de basura:

Para la gestión automática de la memoria, se evaluaron diferentes tipos de recolectores de basura. Se consideraron enfoques de conteo de referencias y algoritmos de recolección basados en seguimiento de uso de memoria.

4. Algoritmos de ordenamiento:

Se evaluaron diferentes algoritmos de ordenamiento para la lista doblemente enlazada, considerando su eficiencia y adaptabilidad a la gestión de memoria de `MPointer`. Se analizaron opciones como Bubble Sort, Insertion Sort, Quick Sort y Merge Sort.

Alternativa seleccionada y razones de la selección

1. Gestión de memoria automática mediante un recolector de basura:

Se optó por implementar un sistema de conteo de referencias combinado con un recolector de basura. Esto permite un manejo automático de la memoria sin necesidad de intervención manual, reduciendo la posibilidad de errores como fugas de memoria o liberaciones dobles.

2. Estructura de punteros personalizada (`MPointer<T>`):

Se decidió desarrollar una clase `MPointer<T>` para encapsular la gestión de punteros y proporcionar sobrecarga de operadores para facilitar su uso, garantizando al mismo tiempo una administración de memoria eficiente. Esta clase se integra con el recolector de basura `MPointerGC` para hacer un seguimiento de los punteros y liberar memoria cuando sea necesario.

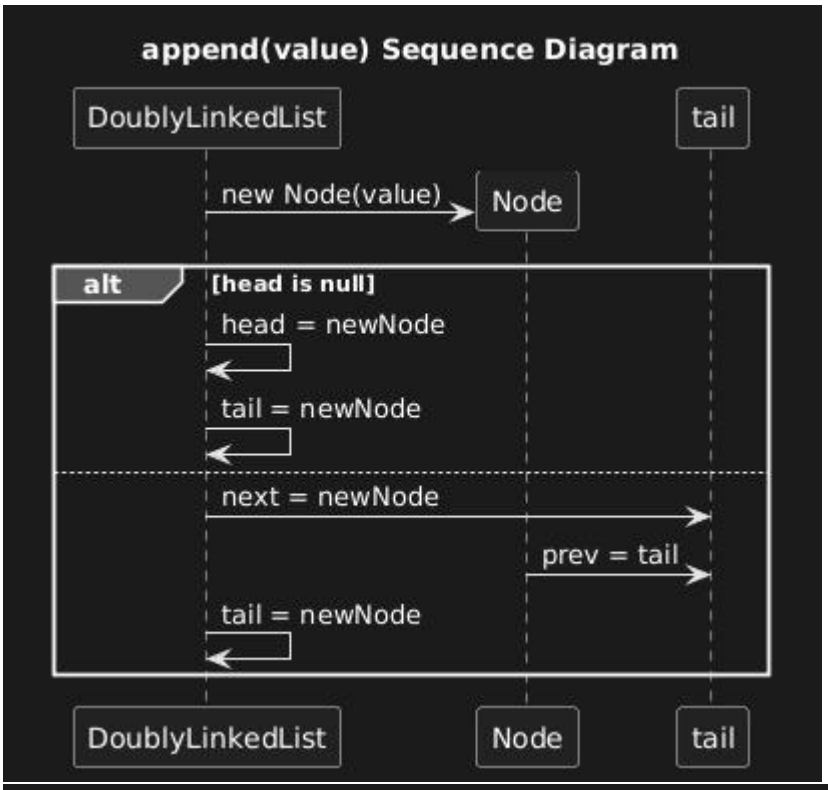
3. Uso de conteo de referencias:

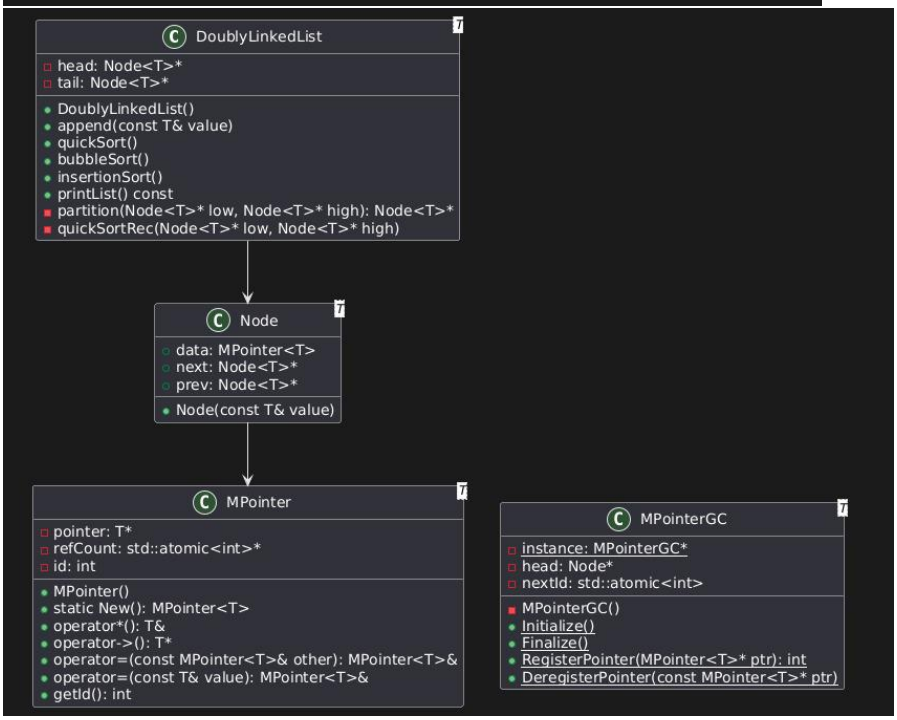
Se seleccionó un sistema de conteo de referencias para determinar cuándo un puntero ya no está en uso, lo que permite al recolector de basura liberar la memoria automáticamente cuando no hay más referencias a un objeto.

4. Implementación de Bubble Sort, Insertion Sort y Quick Sort:

Se eligieron estos algoritmos por su relativa facilidad de implementación y su adecuación para diferentes escenarios. Bubble Sort es sencillo pero menos eficiente para listas grandes, Insertion Sort funciona bien con listas casi ordenadas, y Quick Sort ofrece un buen rendimiento promedio en la mayoría de los casos.

Diagrama de clases UML de la solución propuesta





Algoritmos de ordenamiento implementados

La clase `DoublyLinkedList` incluye la implementación de los siguientes algoritmos de ordenamiento:

Bubble Sort: Ordena la lista comparando pares de elementos adyacentes e intercambiándolos si están en el orden incorrecto. Repite este proceso hasta que no se necesiten más intercambios.

Insertion Sort: Construye la lista ordenada un elemento a la vez, insertando cada nuevo elemento en su posición correcta dentro de la parte ya ordenada de la lista.

Quick Sort: Divide la lista en dos sub-listas alrededor de un elemento pivote, ordena las sub-listas recursivamente y combina las sub-listas ordenadas.

Estos algoritmos han sido adaptados para trabajar con `MPointer`, asegurando que la gestión de memoria se realice de manera segura y eficiente durante el proceso de ordenamiento.

Problemas encontrados

Gestión de múltiples punteros:

Al implementar el sistema de conteo de referencias, surgieron problemas al sincronizar el conteo cuando los punteros se movían o copiaban entre diferentes variables.

Eliminación segura de punteros:

Asegurarse de que los punteros se eliminen de manera segura sin provocar fugas de memoria o referencias colgantes presentó desafíos que requirieron pruebas exhaustivas.

Integración del recolector de basura:

Implementar un recolector de basura eficiente que no afectara negativamente el rendimiento del sistema fue uno de los mayores desafíos, especialmente al manejar grandes cantidades de punteros dinámicamente.

Adaptación de los algoritmos de ordenamiento:

Integrar los algoritmos de ordenamiento con la gestión de memoria de `MPointer` requirió consideraciones especiales para garantizar la correcta manipulación de los punteros y evitar problemas de memoria durante el ordenamiento.

Preguntas abiertas

¿Cómo puede optimizarse aún más el rendimiento del recolector de basura en entornos de alta concurrencia?

¿Existen otros mecanismos que podrían complementar el conteo de referencias para hacer más eficiente la liberación de memoria?

¿Qué optimizaciones podrían implementarse para mejorar la eficiencia en proyectos que gestionan grandes volúmenes de punteros dinámicamente?

-¿Cómo se podría extender esta solución para soportar entornos multithreading de manera más eficiente?

- ¿Qué otros algoritmos de ordenamiento podrían ser útiles para la lista doblemente enlazada, y cómo se adaptarían a `MPointer`?