TDA LISTA – README Faccini Trinidad

PRIMERA SECCION

COMPILACION

Para **compilar** el trabajo se utiliza el siguiente comando: **"gcc *c -o lista_se -g –std=c99 -Wall -Wconversion -Wtype-limits -pedantic -Werror -O0".**

- *c: el compilador tomará todos los archivos .c de la carpeta en la que se encuentra y los compilará.
- -o: para asignarle el nombre al fichero de salida.
- lista_se: el nombre del archivo ejecutable.
- -g: nos da la opción de guardar información de depuración del archivo. Nos permite luego poder seguir el código a través de un depurador para analizar los errores.
- -std=c99: especifica el estándar del lenguaje (en este caso c99)
- -Wall: habilitará todos los avisos.
- -Wconversion: avisará si hubo algún tipo de conversión que puede alterar un valor/resultado.
- -Wtype-limits: generará un aviso en el caso de que exista una comparación que de siempre verdadera o falsa, debido al rango limitado del tipo de datos.
- -pedantic: va en conjunto con –std=c99, le dice al compilador que rechace cualquier código que no sea compatible con el estándar del lenguaje.
- -Werror: esta opción convertirá a los avisos anteriores en errores.
- -O0: hace que el compilador compile el código fuente de la forma más sencilla posible.

EJECUCION

Para **ejecutar** el trabajo se utiliza la siguiente línea: **"valgrind --leak-check=full --track-origins=yes --show-reachable=yes /lista_se".**

- --leak-check=full: nos sirve para detectar fugas de memoria de manera individual, viendo cada una especificada.
- --track-origins=yes: nos deja ver los errores que pueden depender de variables no inicializadas.
- --show-reachable=yes: nos permite ver, además de los bloques de memoria que fueron definitiva o posiblemente perdidos, aquellos a los que si podemos acceder y aquellos que pueden haber sido perdidos indirectamente.

ESPECIFICACIONES DE LA IMPLEMENTACION

LISTA.C

En este trabajo decidí implementar lo más que pude la **recursividad**. En trabajos anteriores no quise tomar el riesgo, pero para este TDA me sentí mucho más segura al hacerlo y me sirvió para entender mucho mejor su manejo.

Al ser derivados, para pila y cola reutilicé las funciones que había hecho anteriormente para lista y de esta forma evité tener bloques de código repetido.

PRUEBAS.C

Para las **pruebas** decidí en primer lugar probar todas las funciones sobre una lista **vacía** y sobre una **nula**, de esta manera pude ver que la comprobación que hacía en la primera línea de cada función para que devuelvan un determinado valor cuando recibían a estos tipos de listas, funcionaba correctamente.

Aclaración: dentro de las de lista vacía, verifique si el valor que devuelve la función lista_con_cada_elemento al mandarle una función nula era el esperado.

A partir de allí, hice **pruebas unitarias** sobre las **funciones principales del TDA Lista** (insertar al final, insertar en posición, borrar al final, borrar de posición). Dentro de estas pruebas, comprobaba el correcto funcionamiento de las restantes, como *lista_ultimo, lista_elementos, lista_elemento_en_posicion, etc,* ya que no podía probar estas ultimas de manera aislada.

Para las pruebas de **pila y cola**, verifique que respetaban sus **políticas de entrada y salida** de datos utilizando solamente las funciones especificas de estos dos TDAs.

Finalmente, realicé las pruebas para los iteradores en forma conjunta, probando como funcionaban tanto para una lista que contiene solo un elemento como para una que contiene varios.

SEGUNDA SECCION

¿Qué es lo que entendés por una lista?

Una lista es un tipo de dato abstracto que sirve básicamente para agrupar elementos. Para este tipo de dato no existe ninguna política que determine la manera en la que se van agrupando los elementos y tampoco importa el orden de entrada/salida.

Operaciones básicas que realizamos con el TDA lista:

- Crear: inicializamos los valores necesarios para la utilización de esta.
- Destruir la lista.
- Insertar elementos (al final o en una posición especifica).
- Borrar elementos (al final o en una posición especifica).
- Ver elementos (en cualquier posición de la lista).
- Chequear si la lista se encuentra vacía.
- Ver la cantidad de elementos que tiene la lista.

Existen varios tipos de implementaciones para representar una lista, una de ellas es la lista de nodos/lista enlazada. Esta implementación nos permite agrupar a los elementos como una secuencia.

¿Cuáles son las diferencias entre ser simple y doblemente enlazada?

Ambas formas, simple y doblemente enlazada, se basan en nodos y en este tipo de implementación, la lista siempre mantiene referencia, **al menos**, al nodo inicial.

En una lista simple/simplemente enlazada cada nodo tiene referencia a su sucesor **o** predecesor (solo puede ser en un sentido), a diferencia de la doblemente enlazada en la que cada uno de estos tiene referencia a su predecesor **y** a su sucesor.

*¿Cuáles son las características fundamentales de las **pilas**?*

Una pila es un tipo de lista a la que se accede solamente desde su extremo final, con dos operaciones principales: **apilar** y **desapilar**.

Se accede desde su extremo final porque la política que rige para este TDA es la de **LAST IN - FIRST OUT**. Esto significa que el último elemento en ser apilado va a ser el primero en ser desapilado, y solo estamos trabajando con su elemento final.

Ejemplo: pila de platos.

El conjunto de **operaciones básicas** que se pueden realizar con este TDA y que he usado en el trabajo, además de las dos nombradas anteriormente, son:

- Crear
- Destruir
- Vacía
- Tope

*¿Cuáles son las características fundamentales de las **colas**?*

Una cola es otro tipo de lista a la que además de poder accederse desde su extremo final, se puede hacerlo desde su inicio. Las dos operaciones principales con las que se realiza son: **encolar** y **desencolar**.

En este caso, la política que rige es la de **FIRST IN – FIRS OUT**. Esto significa que el primer elemento en ser encolado es también el primero en ser desencolado.

Ejemplo: cola de supermercado, la primera persona en llegar a la fila va a ser la primera en pagar y en irse.

Para este TDA las **operaciones básicas**, además de las dos nombradas anteriormente, son:

- Crear
- Destruir
- Vacía
- Primero

¿Qué es un iterador? ¿Cuál es su función?

Un iterador es un tipo de dato abstracto cuya función es navegar secuencialmente y de distintas maneras sobre los elementos almacenados en una lista especifica. Los iteradores nos permiten abstraernos de la estructura de la lista que se recorrerá con el este.

- Operaciones básicas: primero() siguiente() hayMas() elementoActual().

¿En qué se diferencian un iterador externo y uno interno?

La diferencia principal está en el **control de la iteración**.

En el **interno** no es necesario controlar el ciclo en el cual recorre a la lista y en el **externo** debemos crear un set de funciones primitivas que usamos para recorrerla.

Normalmente un iterador **interno** es una función que recibe tres parámetros:

- La lista
- Un puntero a una función, que recibe un dato como el contenido en el nodo, otro puntero extra por si la función lo requiere.
- Un puntero a void extra que funciona como memoria común a todos los nodos visitados.

Las funciones primitivas del iterador externo son:

- crear()
- primero()
- siguiente()
- haySiguiente()
- elementoActual()