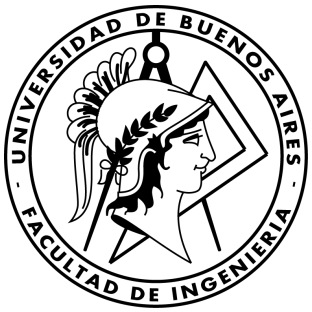

TDA Lista

Algoritmos y Programación II
Segundo cuatrimestre de 2022

Alumno:	Fernández, Facundo Nahuel
Número de padrón:	109097
Email:	ffernandez@fi.uba.ar

Índice

1. Introducción **2**

2. Teoría **2**
2.1. Lista 2
2.2. Pila 2
2.3. Cola 3

3. Detalles de implementación **3**
3.1. lista_insertar 3
3.2. lista_insertar_en_posición 3
3.3. lista_quitar 3
3.4. lista_quitar_de_posición 4
3.5. Iterador interno 4

4. Diagramas **4**

5. Compilación y Ejecución **5**

1. Introducción

El objetivo de este trabajo es implementar una lista utilizando nodos simplemente enlazados y crear un iterador interno y un iterador externo para dicha lista. Además, se reutiliza la implementación de la lista para implementar los tipos de dato abstracto pila y cola.

En este informe se desarrolla los conceptos de lista, pila y cola, además de una explicación de cómo funcionan las partes más importantes del código.

2. Teoría

2.1. Lista

Es un tipo de dato abstacto que agrupa elementos, donde cada uno tiene un sucesor y un predecesor. Se puede acceder a cualquier elemento de la lista, además de poder insertar o eliminar elementos de cualquier posición de la misma. Existen tres tipo de implementaciones: vector estático, vector dinámico y con nodos enlazados.

En el caso de la implementación con vector estático, se tiene un tope, que será la cantidad de elementos que almacena la lista (se guarda para saber donde termina la lista), y una capacidad, que será la cantidad de elementos que se puede almacenar. El problema de esta implementación es que un vector estático puede guardar solamente una cantidad limitada de elementos.

La implementación con vector dinámico es similar, pero en este caso no tenemos una limitación para guardar elementos, pero corremos con el riesgo de quedarnos sin memoria contigua para reservar. El problema es que cada vez que se quiera insertar un elemento en una posición que no sea la última, se debe mover a todos los elementos que vienen después hacia la derecha, lo que causa que las operaciones sean más complejas (esto vale también para la implementación con vector estático).

Por último, tenemos la implementación de lista con nodos enlazados. ¿Qué es un nodo enlazado? Es otro tipo de dato abstacto que no sólo guarda un elemento (que puede ser de cualquier tipo), sino que conoce al nodo que lo sucede (o que lo antecede, o a ambos).

En el caso de una lista simplemente enlazada (como la implementada en este trabajo), cada nodo conoce al que le sigue, de modo tal que se puede correr desde el primer nodo hasta el último y no en otro sentido (ver Figura 1). En nuestro trabajo, una lista guarda el nodo inicial, el nodo final y la cantidad de nodos (elementos) que hay en la lista, aunque podría implementarse sólo guardando el primer nodo (obteniendo el elemento final y la cantidad recorriendo la lista).

Una lista también puede ser doblemente enlazada. Esta es similar a la anterior mencionada, pero cada nodo guarda una referencia a su predecesor y a su antecesor. Se tiene un inicio y un fin, pero de este modo se puede recorrer también la lista hacia atrás y no unidireccionalmente.

Por último, una lista puede ser circular (Ver Figura 2). La implementación de esta puede ser con nodos simple o doblemente enlazados, la diferencia reside en que, en este caso, el último nodo apunta al primero, haciendo que se pueda recorrer çircularmente"la lista. En estos casos, se debe contar la cantidad de elementos o la lista debe guardar una referencia al último nodo, pues el último ya no apunta a NULL y es fácil "perderlo".

2.2. Pila

La pila es una colección ordenada de elementos en la que pueden insertarse (apilarse) y eliminarse (desapilarse) elementos por un extremo, al que se denomina tope.

Las implementaciones de pila son iguales a la de lista, sólo que ahora se tiene acceso al tope de la pila y no a los elementos en cualquier posición. El problema con la implementación con vectores son los mismos (excepto el de insertar en una posición intermedia, pues en este tda se apila únicamente).

2.3. Cola

La cola es una colección ordenada de elementos al igual que la lista y la pila. En este caso, se inserta por el final (encola) y se desencola por el frente (desencola).

Las implementaciones son las mismas que las de lista, y los problemas con las implementacione con vectores son los mismos también: para desencolar en un vector, se debe desplazar todos los demás elementos hacia la izquierda, por lo que la complejidad es O(n) y no O(1) como se desearía, además de los problemas ya mencionados de tener capacidad limitada o memoria limitada.

3. Detalles de implementación

3.1. lista_insertar

La implementación de esta función es simple: se reserva memoria para crear un nuevo nodo, el elemento del nodo será el que se recibe y, como se debe insertar al final, el siguiente será NULL. Se contemplan dos casos: si la lista está vacía, el nuevo nodo será tanto el inicio como el fin de la lista; si no, se establece que el nodo siguiente al que era el fin de la lista será el nuevo nodo creado, y se actualiza el fin a este nuevo nodo.

Esta implementación es claramente O(1) pues no se deben recorrer los elementos de la lista, no depende del tamaño de esta. Se utiliza esta función para encolar.

3.2. lista_insertar_en_posición

Se modifica el nodo inicial de la lista mediante una función recursiva, que recibe a este mismo nodo, el elemento a insertar y la posición en la que se debe insertar. Si la posición no es 0 y el nodo inicial existe, significa que todavía no debo insertar, por lo que se pasa a modificar el siguiente del nodo inicial con la misma función recursiva (ahora recibe a este nodo, el mismo elemento que se debe insertar y la posición disminuida en uno, pues avancé sobre la lista). Así se seguirá recorriendo la lista hasta llegar a dos posibles casos: el priemro, si el nodo sobre el que se está parado no existe (es NULL), lo que significa que llegué al final de la lista, o si el nodo sobre el que se está parado existe pero la posición es 0.

En ambos casos, se crea un nuevo nodo con el elemento que se debe insertar y se establece que el siguiente a este sea el nodo sobre el que estoy parado (NULL en el primer caso). Se devuelve este nodo, por lo que se modifica al siguiente del nodo sobre el que estuve parado antes (en el primer caso, el último de la lista), y al modificar este, significa que se modifico el siguiente de su antecesor, así hasta llegar al nodo inicial, que era el que se quería cambiar en primera instancia. Quizás el nodo inicial sea igual, pero en algún lugar de la lista fue insertado el elemento que se quería.

Si se dió el caso en que se llegó al final de la lista, se debe actualizar el final de esta ya que se agregó un elemento allí, por lo que se recorre nuevamente la lista y se establece como fin el nuevo nodo agregado.

Cuando se recibe directamente 0 como posición, sucede lo relatado anteriormente, pero no se recorre la lista ya que se ingresa directamente al caso en que la posición es 0 y se modifica el inicio directamente. Si la lista no estaba vacía, el inicio ya fue modificado y no se debe modificar el fin. Si estaba vacía, simplemente se modifica el fin al nodo insertado sin recorrer la lista. Finalmente, en este caso la complejidad es O(1), pues las acciones realizadas no dependen del tamaño de la lista. Esto es lo que se utiliza para apilar (en la implementación utilizada, se considedra el tope de la pila como el inicio de la fila).

3.3. lista_quitar

Se establece un nodo auxiliar (llamado nodo_actual) para recorrer la lista y se contemplan dos casos: si el primer nodo de la lista no tiene siguiente, significa que la lista tiene tamaño 1, por lo que el inicio y el fin de la lista pasan a ser NULL y se elimina (libera la memoria de) el nodo

del final de la lista, previamente guardado en otro nodo auxiliar; si el caso no es el de la lista de tamaño 1, se recorre hasta el anteúltimo nodo, se declara como el fin de la lista y su siguiente como NULL (antes el siguiente era el nodo final de la lista) y se elimina el que antes era el nodo final.

No pude utilizar esta función para la implementación de pila pues es O(n), ya que para actualizar el final de una lista de más de un elemento debo sí o sí recorrerla para poder actualizar el fin de la lista.

3.4. lista_quitar_de_posición

Se utiliza una función recursiva para recorrer y modificar la lista. Esta recibe a la misma lista, al nodo anterior al que estaba parado, al nodo sobre el que estoy parado (con .[es]tar parado"me refiero al nodo sobre el que estoy recorriendo) y la posición en la que quiero eliminar un elemento. En un principio, el nodo anterior es NULL y el actual es el nodo inicial, ya que empiezo a recorrer desde el inicio de la lista.

Se pueden dar distintos casos (en todos ellos, se devuelve el elemento eliminado, que es previamente guardado en una variable auxiliar):

El primero, si estoy parado sobre el nodo inicial (es decir, el anterior es NULL) y la posición es 0, significa que quiero eliminar el primer elemento de la lista, por lo que se modifica el inicio de la lista y se establece que ahora el inicio es el que era el segundo elemento para posteriormente eliminar (liberar la memoria de) el nodo que era el inicial.

Otros dos casos se dan si no tengo un nodo después del nodo sobre el que estoy parado (es decir, estoy en el final de la lista). Si se da que tampoco hay nodo anterior, significa que la lista era de tamaño 1 (el inicio es el mismo que el fin), por lo que ambos pasan a ser NULL y se elimina el nodo. Si lo anterior no es el caso, establezco que el nodo siguiente al anterior debe ser NULL y que además pasará a ser el final de la lista (ver figura 3).

Por último, si la posición es 0, se hace lo que se ve en la figura 4. Se elimina el elemento sobre el que se está parado, pero antes estableciendo que el siguiente al anterior es el siguiente al nodo sobre el que estoy parado.

En el primer caso, las acciones no dependen del tamaño de la lista pues no se recorre a la misma, por lo que la complejidad es O(1). Esto es lo que se utiliza para denencolar y desapilar.

3.5. Iterador interno

Se establece un contador y un nodo auxiliar para recorrer la lista, además de un booleano que indica si se debe seguir iterando o no.

Mientras el nodo actual exista (es decir, no terminé de recorrer la lista) o el booleano sea true, seguiré iterando sobre la lista. Siempre que itere el contador va a aumentar en uno y voy a avanzar al nodo siguiente, además de aplicarsele la función al elemento del nodo sobre el que estoy iterando y al contexto. Si esta devuelve false, significa que no debo iterar más, por lo que se modifica el booleano a false. Se devuelve el contador para conocer sobre cuántos elementos se iteró.

4. Diagramas

Figura 1: Lista simplemente enlazada.

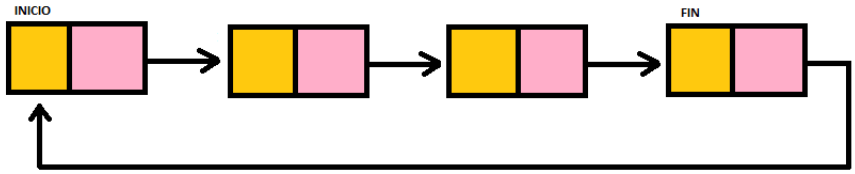

Figura 2: Lista circular simplemente enlazada.

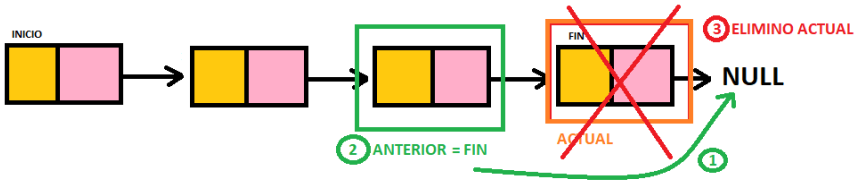

Figura 3: Eliminación de un nodo al final.

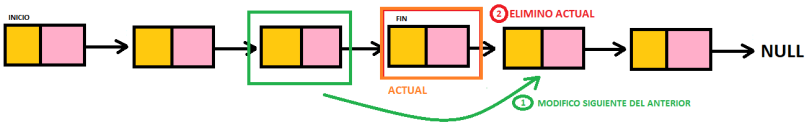

Figura 4: Eliminación de un nodo en posición arbitraria.

5. Compilación y Ejecución

Para compilar y ejecutar los programas de prueba, se utiliza el Makefile incluido.

- Compilar el programa de ejemplo:
  make ejemplo
- Ejecutar el programa de ejemplo:
  ./ejemplo
- Compilar y ejecutar las pruebas con Valgrind:
  make valgrind-pruebas
- Limpiar archivos compilados:
  make clean
  (ejecutando 'make' ejecuta el clean y corre las pruebas con valgrind).