TDA Lista.

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

Alumno: Martín Abramovich
Padrón: 108762
Email: mabramovich@fi.uba.ar

Contenido

1. Introducción

Se solicita implementar un tipo de dato abstracto lista (en adelante TDA lista) utilizando nodos simplemente enlazados. Para ello se cuenta con las firmas de las funciones públicas a implementar. Adicionalmente se pide la creación de un iterador interno y uno externo para la lista, y también reutilizar la implementación de lista simplemente enlazada para implementar los TDAS pila y cola.
Se implementa además una metodología orientada a pruebas, en que se solicita la creación de pruebas pertinentes para cada una de las diferentes primitivas del TDA.

2. Implementación

Para la elaboración del trabajo, comencé implementando el TDA lista teniendo en cuenta que una pila y una cola son listas restringidas, por lo que me iba a ser posible implementarlas reutilizando la implementación de la lista.

Para el desarrollo de las distintas funciones me fue muy útil graficar previamente, para poder tener una idea de qué sucede en cada caso (cuando una lista se crea, cuando se inserta un elemento, etc.).

La estructura de lista provista por el .h es la siguiente:

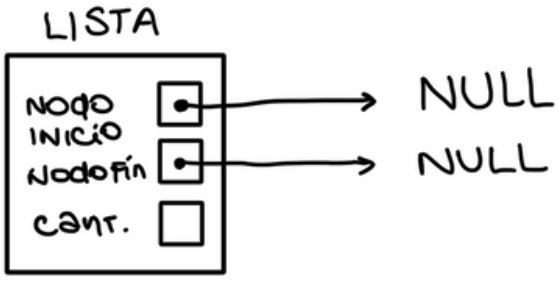
```
typedef struct lista {
  nodo_t *nodo_inicio;
  nodo_t *nodo_fin;
  size_t cantidad;
} lista_t;
```

Siendo la estructura de un nodo:

```
typedef struct nodo {
  void *elemento;
  struct nodo *siguiente;
} nodo_t;
```

2.1 Lista crear

Gráficamente la estructura de la lista al crearla quedaría de la siguiente manera:

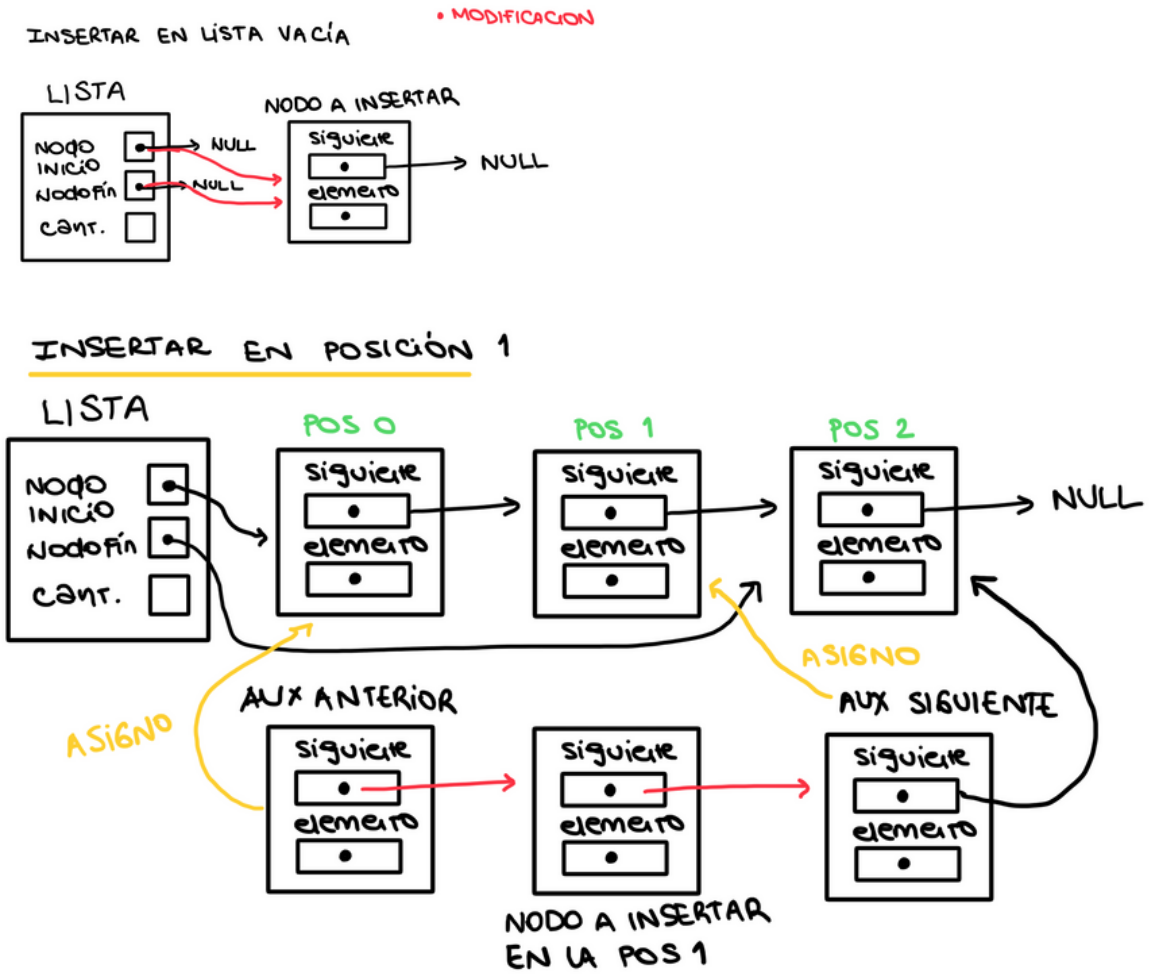


Por esta razón es que la asignación de memoria para la creación la realicé con *calloc,* que inicializa todo en 0.

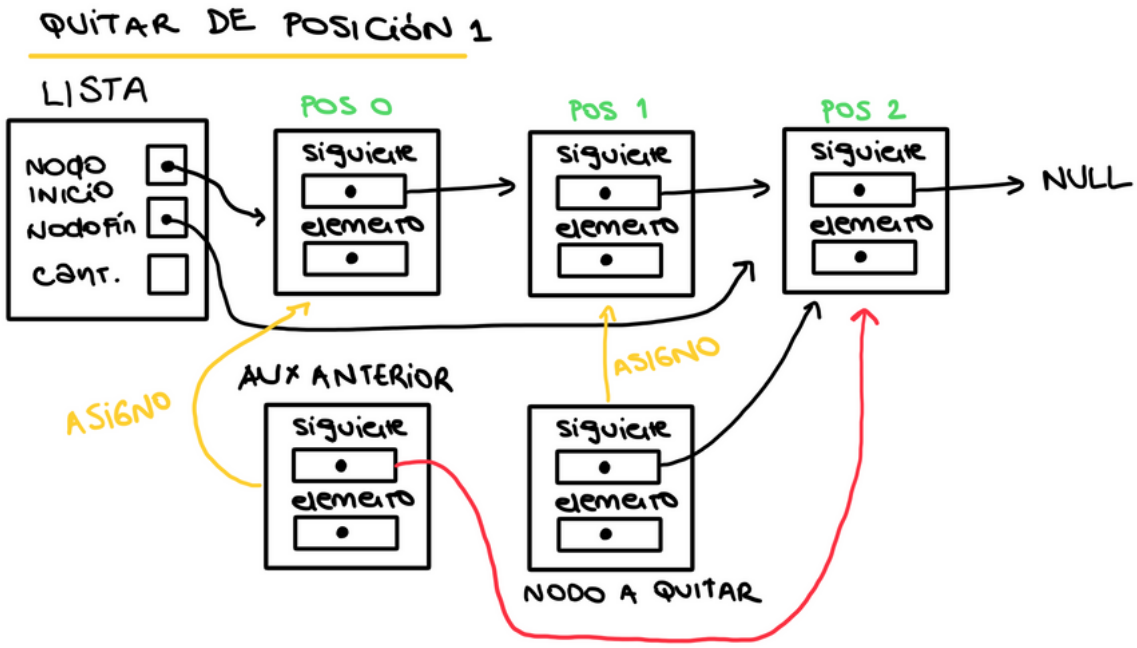
2.2 Lista insertar y quitar

Para insertar en una lista vacía, me aseguré de que tanto el nodo inicio como el nodo fin (que apuntaban a NULL) apunten al nodo insertado.

Para insertar en una posición determinada sin perder las referencias a los nodos, reservé memoria con *malloc* para el nodo a insertar y creé dos nodos auxiliares. Al primero, *anterior*, le asigno el nodo anterior al de la posición a insertar. Al segundo, *próximo*, le asigno el nodo siguiente al de la posición a insertar.
Luego, anterior->siguiente es el nodo a insertar y el siguiente de nodo a insertar es el nodo auxiliar *próximo.*
Contemplé los casos bordes, insertar en la primera y en la última posición. Para insertar en la última posición, se llama a la función lista_insertar que inserta un elemento al final de la lista. Para insertar en la primera posición, me aseguré que el nodo inicio apunte al nodo insertado.



A la hora de quitar, una vez más tuve que evitar perder las referencias a los nodos.
Lo que pensé fue crear un nodo auxiliar *anterior* al que le asigné el nodo de la posición anterior a quitar y luego a *anterior->siguiente* le asigné el nodo siguiente del elemento a quitar. Recién en ese momento puedo liberar con *free* el nodo a quitar.
Si se recibe una posición inexistente o se quiere liberar la última posición, entonces se llama a la función *lista_quitar* que quita el último elemento.
Contemplé el caso en que se quite el único elemento de la lista, para que nodo inicio y nodo fin apunten a NULL y el caso en que se quiera quitar en la primera posición, para que nodo inicio apunte al siguiente del quitado.



2.3. Elemento en posición y buscar elemento

Lista_elemento_en_posicion devuelve el elemento que se encuentra en la posición indicada. Si la lista no existe o la posición no existe en esa lista, retorna NULL. Creé un nodo auxiliar que comienza en nodo inicio y lo fui iterando hasta llegar al nodo de la posición indicada. Luego devuelvo el elemento de dicho nodo.

Para *lista_buscar_elemento* me aseguré que la lista y el comparador fueran válidos y que la lista no sea vacía. Luego, creé un nodo auxiliar que comienza en nodo inicio y lo fui iterando. En cada iteración fui llamando a la función comparador, si la misma encuentra coincidencia entre el contexto y el elemento, devuelve el elemento, de lo contrario sigue iterando. Si al recorrer todos los elementos no encontró el elemento buscado, devuelve NULL.

2.4. Lista destruir y lista destruir todo

En *lista_destruir* me creé un nodo auxiliar que fui iterando mientras destruía cada nodo con *free*.
Al terminar de destruir los nodos, destruí la lista con *free.*

Para *lista_destruir_todo* me aseguré de que la lista y la función no sean nulas, de la contrario llama a *lista_destruir*. Me creé un nodo auxiliar que fue iterando y en cada iteración llamé a la función destructora de elementos, para destruir el elemento del nodo al que apunta el auxiliar. Luego libero el nodo con *free* y finalmente la lista.

2.5. Iterador externo: crear

Para crear el iterador me aseguré de que la lista fuese válida. Luego reservé memoria para un iterador con *malloc* y, de ser correctamente creado, le asigno nodo inicio al corriente del iterador creado y la lista a la lista del iterador.

2.6. Iterador interno: lista con cada elemento

Para esta función me creé un nodo auxiliar al que le asigné el nodo inicio. Luego, comencé a iterar. En cada iteración llamo a la función que devuelve *true* si hay que seguir iterando o *false* en caso contrario. Si debe dejar de iterar, devuelvo la cantidad de iteraciones. Si recorre toda la lista, devuelvo la cantidad de elementos de la lista.

Para esta función decidí que el *for* que usé para la iteración comience con la variable *i* en 1 y no en 0, ya que si no la cantidad de iteraciones devueltas no sería la correcta.

2.7. Pila y cola

Para implementar la pila y la cola, decidí reutilizar la lista implementada a través de casteo de punteros ya que tanto pila como cola son listas restringidas. Es por esto que no tuve que tomar mayores decisiones de implementación en los respectivos .c ya que todas las consideraciones habían sido previas al desarrollar *lista.c*.

3. Conceptos teóricos

3.1. TDA Pila e implementación con nodos enlazados

Una pila es una colección ordenada de elementos que pueden insertarse y eliminarse por un extremo, denominado tope. El modo de acceso a sus elementos es de tipo LIFO (del inglés Last In, First Out, «último en entrar, primero en salir»). Sus primitivas son crear, apilar, desapilar, tope, destruir, esta_vacía.

Los nodos enlazados son un TDA en el que se trata de abstraer la idea de un contenedor que no solo va a contener un tipo de dato determinado, sino que también apunta a otro elemento.

En este tipo de implementación en que los elementos son nodos, cada uno tiene una referencia al nodo anterior. Se reserva memoria para cada nodo cuando se quiere apilar y se libera memoria para cada nodo cuando se quiere desapilar. La ventaja de esta implementación es que la memoria no debe ser contigua.

3.2. TDA Cola e implementación con vector estático (cola circular)

Una cola es una estructura que posee dos extremos por los que se realizan operaciones. Un extremo es el inicio o frente de la cola y el otro extremo es el final de la cola. La forma en que opera una estructura como una cola se denomina First In First Out o FIFO. El primer elemento que entra en la cola es el primer elemento que sale de la misma, es decir, se respeta el orden de llegada de los mismos.

Las primitivas de la cola son crear, encolar, desencolar, frente, esta_vacia y destruir.

La cola simple o con vector estático es una estructura de datos de tamaño fijo y cuyas operaciones se realizan por ambos extremos: permite insertar (encolar) elementos al final de la estructura y eliminar (desencolar) por el inicio de la misma. La cola circular es una mejora de la cola simple, debido a que es una estructura de datos lineal en la cual el siguiente elemento del último es el primero. La cola circular utiliza de manera más eficiente la memoria que una cola simple.

3.3. TDA Lista e implementación con nodos enlazados

Este tipo de dato está basado en los nodos enlazados. La idea del mismo reside en que al poder acceder al siguiente de un nodo se puede crear una lista de nodos, esta lista termina cuando el último nodo apunta a NULL. Existen muchas variantes, según como se estructure su nodo (anterior y siguiente) y según su comportamiento (lineal o cíclica)

El tipo de dato lista agrupa elementos, cada uno de estos -excepto el único- tiene sucesor y predecesor -menos el primero-. Este TDA es flexible ya que no tiene restricciones en cuanto a qué elemento se puede acceder. Hay distintos tipos de lista: simplemente enlazada, doblemente enlazada y circular.

Al implementar una lista con nodos simplemente enlazados, cada uno de estos tiene una referencia al nodo siguiente. Este tipo de lista fue la implementada en el trabajo.

Las primitivas de una lista simplemente enlazada son: crear, insertar, insertar en posición, quitar, quitar en posición, elemento en posición, ultimo, vacía, elementos, destruir.

Para este trabajo implementé un TDA Lista con nodos simplemente enlazados, por lo que su implementación puede verse reflejada tanto en el código implementado como en el desarrollo del presente informe.

4. Complejidad de las funciones implementadas

La complejidad de las funciones implementadas en el trabajo son O(1) u O(n).

Las funciones que implican iteración, como *lista_quitar* que recorre hasta el último elemento, son O(n) ya que se hacen n pasos que dependen del tamaño de la lista.

En algunas funciones, la complejidad depende del parámetro recibido. Un ejemplo es en lista quitar de posición, donde si la posición recibida es 0, la complejidad de la función es O(1) porque se hacen 3 pasos contados que no dependen de la cantidad de elementos que tiene la lista.

En otras palabras, si la función depende de la cantidad de elementos que tiene la lista o de un parámetro que se recibe, la función es O(n) -salvo excepciones como la mencionada en el párrafo anterior-. Caso contrario, decir, si no depende de la cantidad de elementos y no implica iteración es O(1).

FUNCIÓN	COMPLEJIDAD	FUNCIÓN	COMPLEJIDAD
Crear	O(1)	Tamaño	O(1)
Insertar	O(1)	Destruir	O(n)
Insertar en posición	O(1) / O(n)	Destruir todo	O(n)
Quitar	O(n)	Iterador crear	O(1)
Quitar de posición	O(1) / O(n)	Iterador tiene siguiente	O(1)
Elemento en posición	O(1) / O(n)	Iterador puede avanzar	O(1)
Buscar elemento	O(1) / O(n)	Iterador elemento actual	O(1)
Primero	O(1)	Iterador destruir	O(1)
Ultimo	O(1)	Lista con cada elemento	O(1) / O(n)
Vacía	O(1)	-	-