TDA — Lista

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

Alumno:	SANCHEZ FERNANDEZ DE LA VEGA, Manuel
Número de padrón:	107951
Email:	msanchezf@fi.uba.ar

1. Introducción

El objetivo del TDA es construir una lista y con ella implementar los TDA pila y cola. Esto es posible ya que los ultimos 2 mencionados son listas "especiales". En otras palabras, tienen la misma estructura pero sus funcionalidades varian un poco.

Desde el punto de vista de caja blanca, el TDA lista es implementado usando nodos enlazados. La ventaja de esto es que cada vez que se quiere insertar un nodo nuevo, no hay que hacer un `realloc()`, sino que un `malloc()` y enlazar los nodos. Otro punto a favor es que al no usar `realloc()`, no es necesario tener memoria contigua para guardar la lista. Además, esta implementacion facilita la posterior implementacion de pila y cola usando el TDA lista.

2. Teoría

1. Explicacion del TDA lista El TDA lista es un TDA que consiste en una especie de arreglo de elementos sobre el cual se definen ciertas operaciones, `lista_crear()`, `lista_insertar()`, `lista_quitar()`, entre otros. La lista esta completamente encapsulada por lo que es imposible acceder a la estructura, para poder hacer uso del TDA es necesario usar las funciones mencionadas.

   En una lista, los elementos tienen un orden establecido. Es decir, siempre hay un elemento atras y uno adelante (exceptuando al primer y ultimo elemento de la lista). Hay distintas implementaciones que nos permiten hacer esto:

   - **Vector estatico**: La implementacion de vector estatico consiste en tener un registro con un campo que sea un vector estatico de `void*`, es decir, declarado como `vector[MAX_VECTOR]` y otro campo que sea la cantidad de elementos. Esta implementacion no es escalable ya que el tamaño del

vector es fijo. Si en algun momento se necesita alterar la cantidad maxima de elementos, seria un gran problema ya que es imposible. Asimismo, al implementar algunas funciones que requieran cambiar la posicion de elementos como `insertar_en_posicion()` o `quitar_de_posicion()`, se vuelve tedioso ya que hay que mover cada elemento para hacer espacio para nuevos elementos o reemplazar el espacio vacio segun corresponda.

- **Vector dinamico**: Esta implementacion es muy similar a la de vector estatico. Sin embargo, la principal diferencia es que no es un vector de tamaño fijo, sino que cada vez que quiera cambiar el tamaño maximo del vector simplemente hago un `realloc()`. No obstante, sufre el mismo problema del vector estatico a la hora de tener que reposicionar elementos al usar ciertas funciones. Sin embargo, una ventaja posible es que tambien se puede achicar el vector si quitamos elementos.
- **Lista de nodos**: Esta implementacion es distinta a las 2 anteriores. Consiste en una serie de nodos enlazados. Estos nodos son registros que contienen un valor y un puntero al siguiente nodo. De esta forma, todos los nodos quedan conectados mediante este puntero. Una gran ventaja de esta implementacion es que no es necesario reordenar el vector al insertar en cierta posicion o quitar algun elemento. Simplemente alcanza con cambiar la direccion a la que apuntan los nodos y listo. Ademas, no es necesario tener memoria contigua en esta implementacion ya que los nodos estan distribuidos en la memoria sin orden alguno (el orden esta dado por los punteros).

2. Explicacion del TDA Pila El TDA Pila es muy similar en esencia a lista. Consiste en una serie de elementos sobre los cuales, nuevamente se definen operaciones. Estas operaciones son muy similares a las de lista. La principal diferencia es a la hora de insertar y quitar elementos. No se pueden insertar elementos en cualquier parte de la pila. La pila es LIFO (last in first out) lo que significa que el ultimo elemento insertado es el primero desapilado. Una analogia posible es una pila de libros, el ultimo libro que pongo en la pila es el primero que puedo sacar. Al ser tan similar a lista, es posible usar las funciones de lista para definir pila ya que basicamente, es una lista con un funcionamiento excepcional. Las pilas, al igual que las listas tienen 3 implementaciones, vector estatico, dinamico y nodos enlazados.

3. Explicacion del TDA Cola El TDA Cola, nuevamente, es muy similar a lista. Tenemos una serie de elementos en un orden especifico sobre la cual podemos realizar ciertas operaciones como encolar y desencolar, entre otras. La diferencia entre cola y el resto es que las colas son FIFO (first in first out), es decir, que el primer elemento ingresado, es el primero en ser removido. En otras palabras, funciona como una cola de un supermercado, el primero que llega es el primero "atendido". Las colas, al igual que las pilas, son listas con un funcionamiento excepcional, por lo que es posible hacer uso de estas para definir cola.

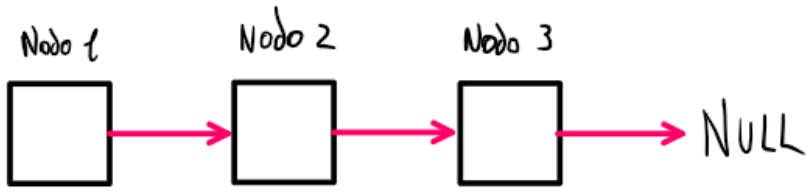
3. Detalles de implementación

Para mi implementacion, defini a la lista como un registro con una serie de nodos simplemente enlazados. El registro cuenta con un puntero al principio de esta "cadena" y otro al final, ademas de la cantidad de nodos. Estos nodos, al ser simplemente

enlazados cuentan con un unico puntero al siguiente nodo, y un puntero al elemento de cada nodo.

Cuando llego la hora de definir pila y cola, opte por no definir sus registros. Por el otro lado, al ser casos especiales de listas, opte por reutilizar completamente las funciones de lista, casteando punteros para que todos los tipos sean compatibles.

1. Detalles del iterador externo Encontre dificultades a la hora de implementar la funcion `lista_iterador_tiene_siguiente()` y `lista_iterador_avanzar()`. Esto se debe a que me centre mucho en el "como" y no en el "que". Lo que quiero decir es que el objetivo del iterador es poder iterar sobre todos los elementos de una lista. Para poder hacer esto, hice que la funcion `lista_iterador_tiene_siguiente()` devuelva **true** si aun hay elementos iterables. Es decir, si el elemento sobre el que esta parado el iterador es distinto de `NULL`. Por el otro lado, `lista_iterador_avanzar()` devuelve **true** solo si se pudo mover el puntero y el nuevo elemento apuntado es distinto de `NULL`. Usando el iterador, soy capaz de mover el puntero hasta llegar al final de la lista que esta representado por un `NULL`. A partir de ese momento, no puedo mover el puntero y ambas funciones me retornan `false`.



Tiene siguiente	true		true		true		false
Avanzar	true		true		false		false

2. Detalles de `nodo_destruir()` Esta es una funcion privada que defini para poder liberar de forma recursiva a todos los nodos de la lista. Lo que hace es llamarse a si misma hasta llegar al ultimo nodo, para luego ir liberando desde el final hasta el principio (para no perder la referencia de ningun nodo). Decidí no realizar ninguna prueba especifica para esta funcion ya que es una funcion relativamente simple y su funcionamiento queda incluido en el de `destruir_lista()`, por lo que directamente decidi realizar pruebas sobre esta ultima.

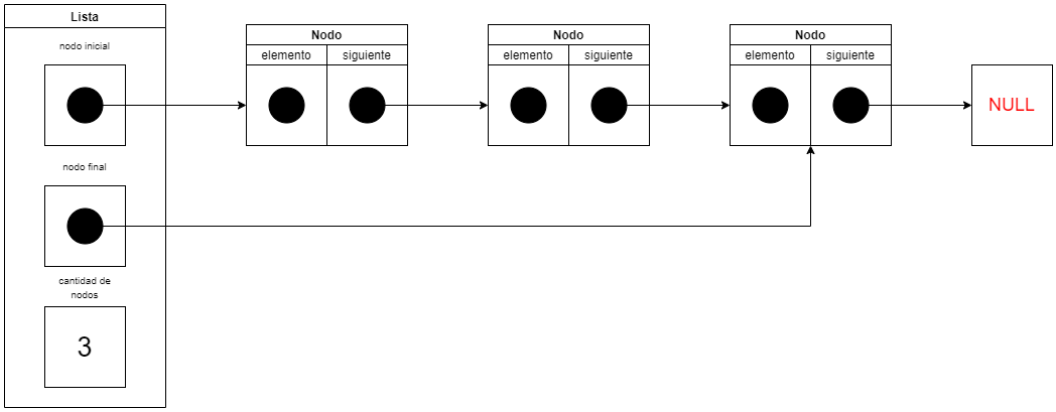
3. Detalles de `lista_insertar_en_posicion()` Esta funcion me resulto dificil de implementar ya que encontre varios casos borde. Uno de ellos es que la lista este vacia, por lo que tengo que actualizar tanto el nodo inicio como el nodo fin. Elegi reutilizar codigo y para este caso borde llame a la funcion `lista_insertar()` ya que para este caso no hay ninguna diferencia. Por el otro lado, si la insercion ocurre al principio, debo actualizar el nodo_inicio. Si ocurre al final de la lista, hay que actualizar el nodo fin. Asimismo, si la posicion a insertar es mayor o igual a la cantidad de elementos, se debe insertar en la ultima posicion, por lo que si se da esta condicion, reuso nuevamente la funcion `lista_insertar()`.

4. Detalles de `lista_quitar_de_posicion()` Al igual que la funcion explicada en el punto 3, esta funcion tiene numerosos casos borde. Para empezar, si la lista esta vacia, devuelve NULL ya que no hay elemento a quitar. Si la lista tiene un elemento o si la posicion a quitar es mayor o igual a la cantidad de elementos, reuso `lista_quitar()`. Si la posicion a eliminar es la posicion inicial, es decir 0, me aseguro de cambiar nodo_inicio al elemento siguiente al inicial para que aun se pueda usar la lista.

5. Detllaes de `lista_con_cada_elemento()` En mi implementacion de esta funcion, decidi permitir que contexto pueda valer NULL. Esto se debe a que quizas hay algunas funciones que pasa el usuario que no necesiten un parametro extra, por lo que no le interesa tener el parametro de contexto.

4. Diagramas

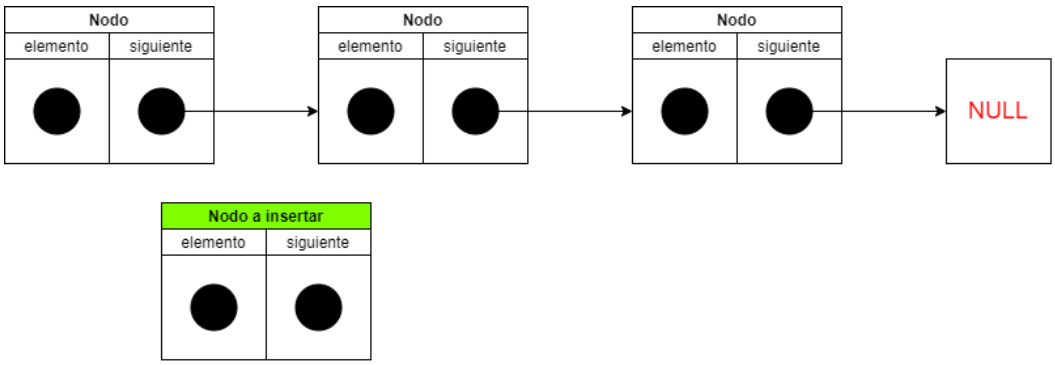
1. Diagrama de la estructura de lista

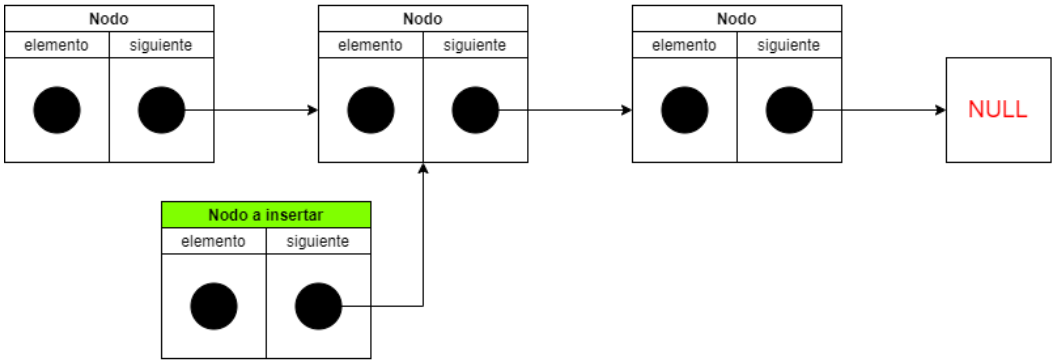


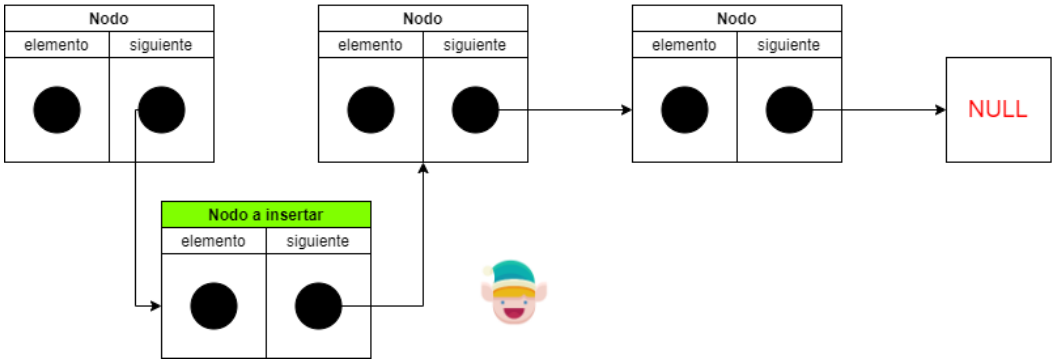
   Un detalle importante de la implementacion es que el ultimo nodo siempre apunta a NULL. Otra aclaracion es que los punteros a elementos pueden apuntar a cualquier tipo de elemento. En el diagrama no dibuje ningun elemento para no contaminar el dibujo y que sea mas entendible, pero la idea es que es un puntero a cualquier tipo de dato.

   Cada vez que se realiza una operacion, es muy importante mantener actualizados los punteros al nodo_inicio y nodo_fin. Por ejemplo, si en algun momento deseo remover el primer elemento, debo cambiar el nodo_inicio. Algo similar sucede con el nodo_fin al insertar un elemento en la ultima posicion.

2. Diagrama a detalle de insercion de elementos en lista

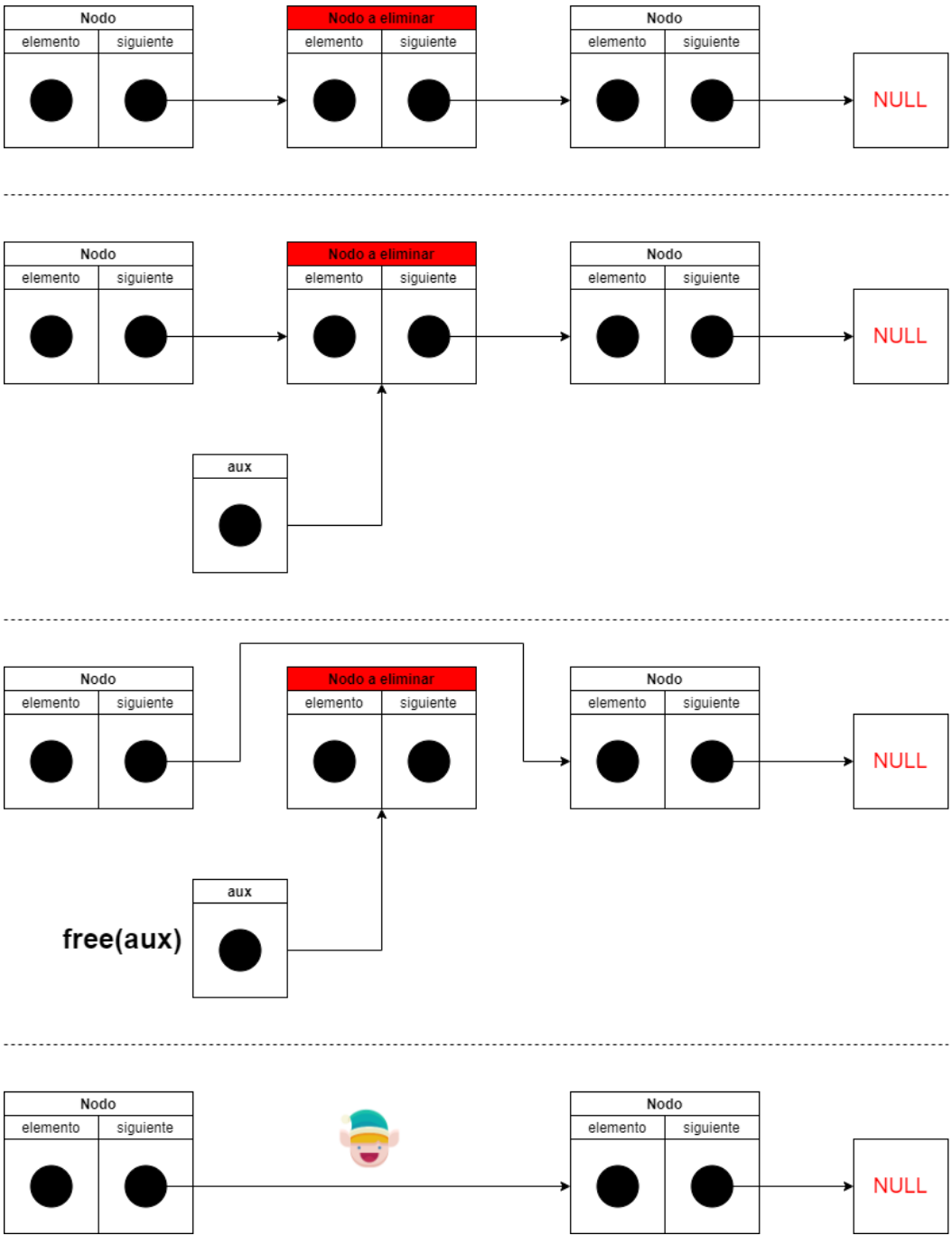






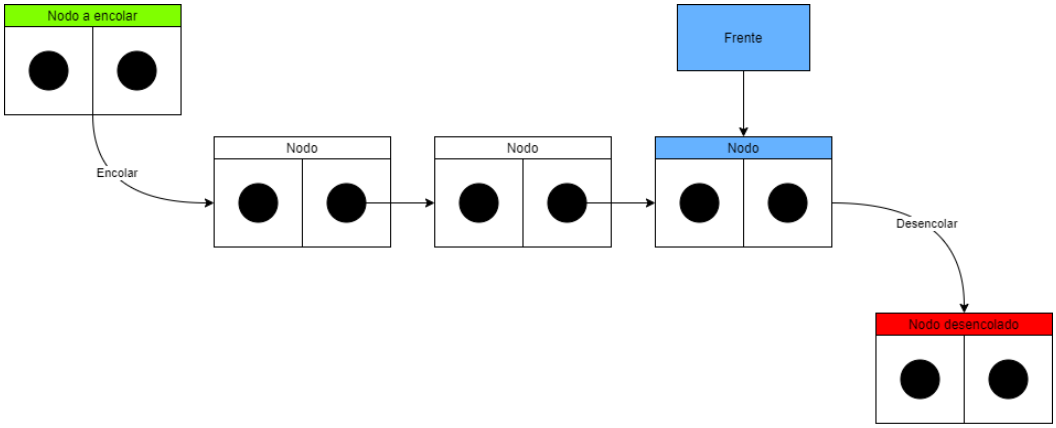
Este diagrama muestra un ejemplo de insercion de elemento en la posicion 1 de la lista. Primero se asigna el siguiente del nodo que quiero insertar al nodo en la posicion siguiente a donde lo quiero insertar. Luego se asigna el siguiente del nodo de la posicion previa a donde quiero insertarlo al nodo que quiero insertar. En criollo seria que primero apunto el nodo nuevo a el nodo que esta en la posicion 1, y luego apunto el siguiente del nodo en la posicion 0 a mi nuevo nodo. Una vez hecho eso, quedaria todo conectado y no habria problema alguno. Sin embargo, esta funcion tiene casos bordes que serian a la hora de insertar en la ultima o primera posicion ya que habria que ocuparse de mantener actualizados los campos de los registros (actualizar nodo_fin o nodo_inicio respectivamente).

3. Diagrama a detalle de eliminacion de elementos en lista



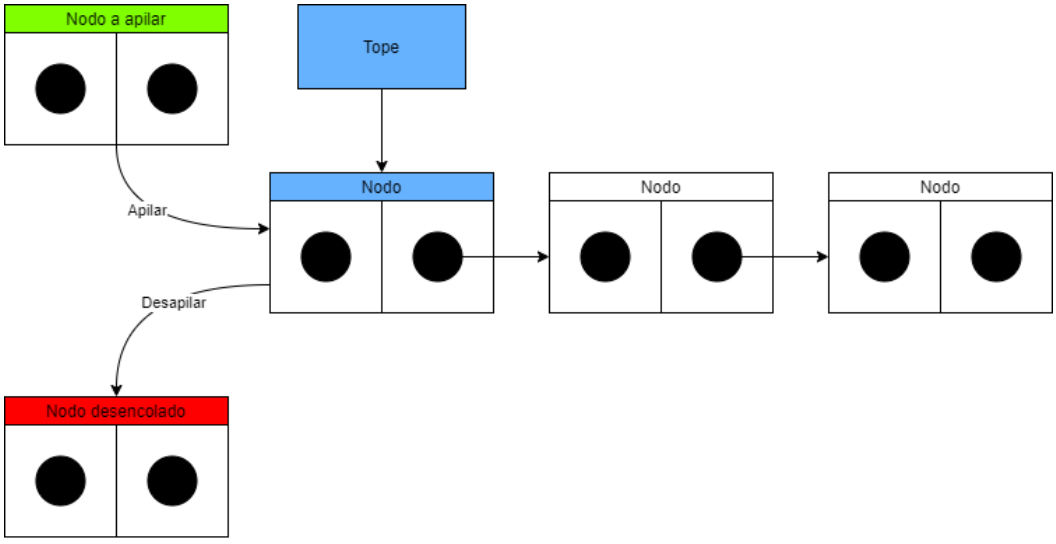
En este caso, se puede apreciar el funcionamiento de la eliminacion de elementos. Para eliminar un elemento es necesario tener un puntero auxiliar para poder liberar la memoria del nodo a eliminar. Si no se hiciera de esta manera, perderiamos la referencia al nodo y no lo podriamos liberar.

4. Diagrama de funcionamiento de cola



En el diagrama se puede ver un elemento que esta por ser encolado en la ultima posicion, y uno que recien fue desencolado (previamente era el frente de la cola pero ese puesto paso al nodo que ahora esta marcado en azul). En este TDA, las operacion desencolar tiene un efecto directo en el frente, mientras que encolar agrega un elemento en la ultima posicion de la pila sin provocar ningun cambio al frente.

5. Diagrama de funcionamiento de pila



En este caso, se puede observar un nodo apunto de ser apilado para pasar a ser el nuevo tope sustituyendo al tope actual (marcado en azul). Tambien se ve como hay un nodo que ya fue desapilado (previamente tuvo que haber sido el tope). Es decir, en este TDA las operaciones apilar y desapilar tienen un efecto directo en el tope.