FACULTAD DE INGENIERIA
Universidad de Buenos Aires

TDA Lista-Pila-Cola

[7541/9515] Algoritmos y Programación II
Segundo cuatrimestre de 2021

Alumno:	Aldazabal, Lucas Rafael
Número de padrón:	107705
Email:	laldazabal@fi.uba.ar

Índice

1. Introducción

En este proyecto se va a desarrollar la implementación de un TDA lista, cola y pila mediante nodos simplemente enlazados y una serie de pruebas diseñadas para verificar el correcto funcionamiento de todo la interfaz.

2. Teoría

En la Introducción se plantean varios conceptos a tener en cuenta y que serán explicados a continuación como TDA, nodo simplemente enlazados o interfaz.

2.1. ¿Que es un TDA?

Un TDA (Tipo de Dato Abstracto) es, en fines prácticos, la creación de un nuevo objeto con sus características y operaciones (por ejemplo el string es un TDA con características como los caracteres o el largo y operaciones como comparar o sustraer). Otra característica primordial de los TDA es la abstracción, a un usuario que va a guardar un string con el nombre de un cliente no le interesa saber como es que luego se comparara con una búsqueda para saber si es el mismo string, le interesa que esa funcionalidad exista y funcione, por eso en un TDA uno entrega una interfaz al usuario, un contrato donde se muestran las posibles operaciones con el. A este concepto donde el usuario solo sabe que puede hacer y no como se hace se lo conoce como caja negra y en el caso del lenguaje c consiste en entregar el .h donde se encuentran declaradas las operaciones.

2.2. TDA Lista

El TDA lista es un, como dice el nombre, un objeto donde se pueden almacenar una cantidad definida o no de elementos con una serie de operaciones como crear, destruir, agregar en cualquier posición, eliminar cualquier elemento, obtener el elemento en cada posición y recorrerlos. A priori es una implementación similar a un vector mas moldeado. Para poder implementar este TDA existen varias maneras con sus virtudes y defectos.

2.2.1. Vector Estático

El vector estático es la implementación mas sencilla de una lista, donde se tiene una estructura que guarda el vector, la cantidad de elementos cargados y el tamaño del vector. Uno puede insertando elementos en cualquier posición desplazando a los que irían después de la posición deseada generando el hueco donde insertar y al quitar un elemento se hace lo contrario, se trae los elementos de vuelta para no dejar espacios libres. Esta implementación cuenta con un defecto importantísimo, tienes predefinida la cantidad máxima de elementos y esto puede ser un problema en la mayoría de implementaciones en el mundo real, ya sea porque se reserva menos espacio del necesario y no podemos seguir agregando o porque tenemos mucho mas de lo necesario, para solucionar esto podemos utilizar un vector dinámico.

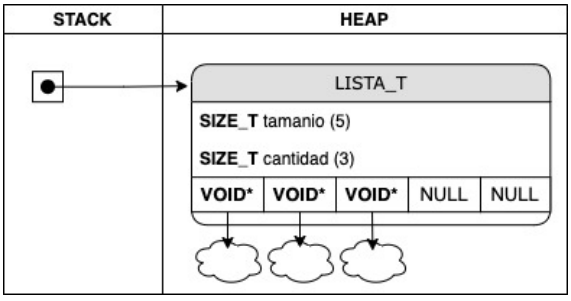

Figura 1: Ejemplo implementación vector estático.

2.2.2. Vector Dinámico

Esta implementación es similar en la anterior pero utilizando la memoria dinámica, cuando la cantidad sea igual al tamaño y quieran agregar elementos, se reserva mas memoria posibilitando agregar mas elementos, dependiendo de la finalidad se puede ir reservando de a un elemento mas o mas agresivamente como por ejemplo duplicar el tamaño. Aunque soluciona el principal problema del vector estático, aun hay cosas por mejorar, ya que a medida que la lista crece y empieza a ocupar mas memoria es mas difícil poder tener un espacio para reservar tanta memoria continua haciendo que muchas ampliaciones sean costosas por tener que mover el vector entero hasta otro lugar, pero lo mas grave es que quizás no tengas un bloque de memoria tan grande para reservar siendo imposible agregar mas elementos pero quizás si tienes muchos bloques mas pequeños y de ahí surge la siguiente implementación: nodos enlazados.

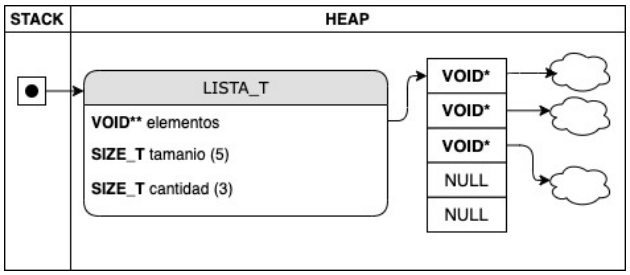

Figura 2: Ejemplo implementación vector dinámico.

2.2.3. Nodo Enlazados

Esta implementación propone un cambio de paradigma para poder solucionar el problema de tener que utilizar memoria contigua, en este caso, se reserva memoria individualmente para cada elemento, almacenándolo en un nodo que lo contiene junto a un puntero al nodo siguiente formando una cadena de nodos desde el primero al ultimo de manera recursiva. De esta manera si se quiere agregar un elemento es cuestión de buscar al anterior, hacer que el nuevo nodo tenga como siguiente el siguiente del nodo anterior y este ultimo reemplazarlo por el nuevo, y para quitar lo contrario, se hace que el siguiente del nodo anterior apunte al nodo siguiente del nodo a eliminar para no romper la cadena. Otra gran diferencia es cuando queremos obtener un elemento, al no estar continuos con tener el inicio no se sabe la posición de todas las demás, solo de la siguiente, por lo que para buscar un elemento tienes que ir avanzando nodo por nodo hasta llegar. Cuando el nodo almacena el puntero al nodo siguiente y al anterior se llama doblemente enlazado, si tiene únicamente el siguiente es simplemente enlazado.

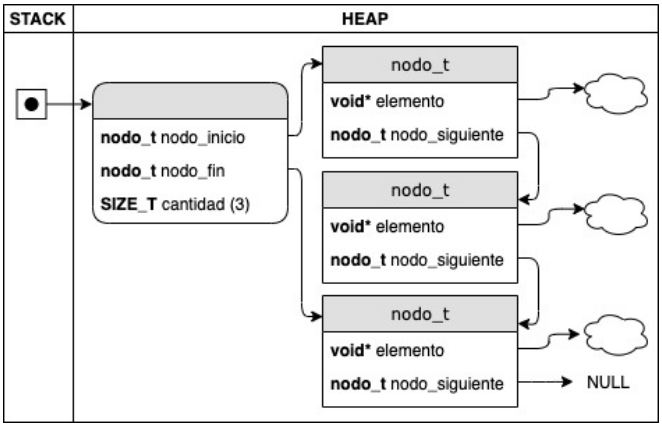

Figura 3: Ejemplo implementación nodos simplemente enlazados.

2.3. TDA Pila y Cola

El TDA Pila y Cola son muy similares al Lista, pero con menos libertades, en la pila tienes únicamente la posibilidad de agregar y quitar el ultimo elemento (LIFO) y en la cola agregar a lo ultimo y quitar lo primero (FIFO). Las implementaciones pueden hacerse con los mismos métodos que la lista e incluso utilizarla como base para la implementación. Igualmente para la cola existe una implementación mas que seria un vector "circular" donde utilizas un vector estático pero al quitar en vez de correr los otros elementos dejas el espacio vació y al llegar al tope del vector, pero tener lugares vacíos al comienzo continuar la cola al principio como una víbora en el juego snake.

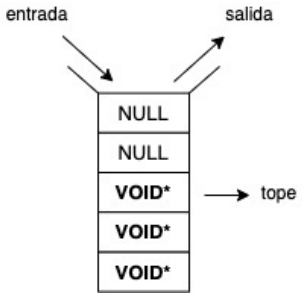

Figura 4: Ejemplo de pila.

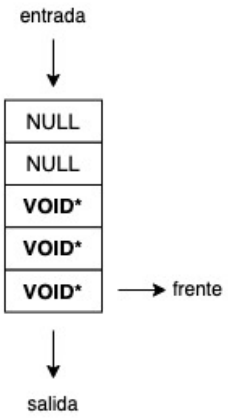

Figura 5: Ejemplo de cola.

3. Detalles de implementación

Como se mencionó anteriormente, en este proyecto se utilizaran nodos simplemente enlazados para crear la lista. Para la pila y cola se utilizara la lista de base para crear las operaciones más limitadas.

3.1. Operaciones de la Lista

La lista tiene unas operaciones básicas para su utilización: crear, destruir, insertar (por default al final), insertar en posición, quitar (por default el ultimo), quitar de posición, ver el primer o el ultimo elemento, elementos en una posición, el tamaño y ejecutar una función personalizada a cada elemento.

La complejidad de las funciones es la siguiente:

- lista_crear O(1)
- lista_insertar O(1)
- lista_insertar_en_posicion O(n)
- lista_quitar O(n)
- lista_quitar_de_posicion O(n)
- lista_elemento_en_posicion O(n)
- lista_primero O(1)
- lista_ultimo O(1)

- lista_vacia O(1)
- lista_tamanio O(1)
- lista_destruir O(n)
- lista_con_cada_elemento O(n)

3.1.1. Recorrer los elementos de la lista

Viendo las operaciones que se le entregan al usuario, se puede ver que hay un problema de eficiencia si el usuario quiere recorrer los elementos porque por cada elemento que hay, hay que utilizar la función lista_elemento_en_posicion que es O(n) llevando la complejidad a O(n*n) cuando lo lógico viendo lo que uno quiere hacer es poder recorrer uno por uno como un vector normal. Para solucionar esto se crea un iterador externo, que básicamente es una instancia donde se guarda la lista y el nodo actual y con unas operaciones puedes ir recorriendo al siguiente elemento con 0(1) y puede utilizarse en un for. Las funciones que agrega son:

- lista_iterador_crear O(1)
- lista_iterador_tiene_siguiente O(1)
- lista_iterador_avanzar O(1)
- lista_iterador_elemento_actual O(1)
- lista_iterador_destruir O(1)

Como se puede ver, las primeras tres funciones se corresponden con los tres parámetros del for y el cuarto se utilizaría para obtener el elemento en cada vuelta pudiendo recorrer todo en O(n).

3.1.2. Agregar y eliminar elementos

En esta implementación, la pila es exactamente igual al diagrama de la figura 3. Con este esquema a la hora de agregar y quitar elementos hay que tener en cuenta tres cosas, vincular el nodo_siguiente del nodo anterior, el nodo_siguiente de este nodo al antiguo nodo_siguiente del anterior y los punteros nodo_inicio y nodo_fin del lista_t. La primera se tiene que hacer en todas las situaciones donde insertes o quites habiendo elementos previamente, en caso de que estemos insertando en primera posicion habría que utilizar el nodo_inicio para guardar el nuevo nodo y no un nodo anterior, y cuando se inserta en la ultima posicion hay que actualizar el valor de nodo_fin. Cuando quitamos el comportamiento es igual en el final y en el inicio habría que hacer que el nodo_inicio apunte al siguiente del elemento a borrar.

3.2. Pila

En la implementación de pila, como se menciono en la teoría, se utiliza como base el TDA Lista para crear nuevas operaciones con menor libertad para que le funcionamiento sea el esperado y acorde al LIFO. Para poder utilizar apilar y desapilar con una menor complejidad se hace la pila "invertida", cuando menor posicion, mas arriba en la pila (explicación del beneficio mas adelante). Las funciones con su complejidad son las siguientes:

- pila_crear O(1) =>utiliza lista_crear
- pila_apilar O(1) =>utiliza lista_insertar_en_posicion*
- pila_desapilar O(1) =>utiliza lista_quitar_de_posicion*
- pila_tope O(1) =>utiliza lista_primero
- pila_tamanio O(1) =>utiliza lista_tamanio

- pila_vacia O(1) =>utiliza lista_vacia
- pila_destruir O(n) =>utiliza lista_destruir

* Las funciones lista_insertar_en_posicion y lista_quitar_de_posicion tienen una complejidad de O(n), pero en esta implementación se utilizan únicamente con la posicion 0, por lo que la peor situación posible pasa de ser el ultimo elemento (lo que proporcionaba el n).a el primero siendo sin importar la cantidad de elementos la misma cantidad de pasos.

3.3. Cola

En la implementación de pila, como se menciono en la teoría, se utiliza como base el TDA Lista para crear nuevas operaciones con menor libertad para que le funcionamiento sea el esperado y acorde al LIFO. Las funciones con su complejidad son las siguientes:

- cola_crear O(1) =>utiliza lista_crear
- cola_encolar O(1) =>utiliza lista_insertar
- cola_desencolar O(1) =>utiliza lista_quitar_de_posicion*****
- cola_frente O(1) =>utiliza lista_primero
- cola_tamanio O(1) =>utiliza lista_tamanio
- cola_vacia O(1) =>utiliza lista_vacia
- cola_destruir O(n) =>utiliza lista_destruir

* La función lista_quitar_de_posicion tiene una complejidad de O(n), pero en esta implementación se utiliza únicamente la posicion 0, por lo que la peor situación posible pasa de ser el ultimo elemento (lo que proporcionaba el n).a el primero siendo sin importar la cantidad de elementos la misma cantidad de pasos.