TDA Lista

[7541/9515] Algoritmos y Programación II

Primer cuatrimestre de 2022

Alumno:	SALLUZZI, Luca
Número de padrón:	108088
Email:	lsalluzzi@fi.uba.ar

1. Introducción

En este trabajo se buscaba afianzar el manejo y el desarrollo de tres Tipos de Datos Abstractos: lista, pila y cola. Se debía tener una clara visión del funcionamiento de cada uno, sus restricciones y posibles implementaciones, así como de sus primitivas y el grado de complejidad algorítmica de las mismas.

2. Teoría

1. TDA Pila con nodos enlazados
    1. Una pila con nodos (simplemente) enlazados, tenemos una pila que apunta al nodo final, al nodo inicial, una cantidad y un tope. A medida que vamos agregando elementos, enlazamos el nodo

anterior con el nuevo agregado, vamos incrementando la cantidad y modificando, debidamente, cuál es el nodo inicio, ya que cada vez que agregamos uno el primer nodo (el que deberá desapilarse primero) se modifica. Si en algún momento la cantidad se vuelve igual al tope, en vez de quedarnos con una pila "llena" y no poder agregar más, aumentamos la memoria asignada (mediante un realloc) y continuamos agregando (esto tomando como supuesto que no hacemos un realloc en cada adición).
Luego, para desapilar elementos, buscamos el elemento que esté en primera posición y se lo damos al usuario, reorganizando la pila del tal manera que el nodo que antes estaba segundo pase a ser el nodo inicial.
Otras primitivas son: crear, la cual crea la lista, inicializando nodo_inicio y nodo_fin como nulos, cantidad y tope como 0. Cantidad, que devuelve la cantidad. Y está vacía, que se fija si la cantidad es mayor a 0.

2. La complejidad algorítmica de sus primitivas son O(1) (menos destruir, que depende de la cantidad de elementos). "Crear" varía según la cantidad, y push y pop, tomando "la parte superior" de la pila como el nodo_inicio, son siempre O(1) porque no hay que recorrer el TDA entero.

crear	Destruir	Encolar	Desencolar
O(1)	O(n)	O(1)	O(1)

2. TDA Cola Circular con vectores estáticos
    1. En una cola circular, la cantidad siempre va a estar siguiendo al tope. Si el tope (cantidad máxima del vector) es 9, y hay 4 elementos cargados (cantidad = 4), siempre va a haber 5 espacios libres, no importa que estos sean "al inicio" de la cola, porque el inicio y el final se combinan, por la circularidad. Al tener un vector estático, si llegamos al punto en el que cantidad == tope, no hay nada más que hacer, no podés encolar más elementos, a no ser que desencolemos otros antes.

En una cola, por otro lado, se encola por atrás y se desencola siempre por adelante, por lo que al agregar un elemento quitarlo tendremos siempre O(1), ya que podemos acceder a las posiciones adyacentes al elemento actual sin problema al estar trabajando con un vector. Otras primitivas son: tope, cantidad, destruir o crear.

2. En cuanto a complejidad algorítmica tenemos que

crear	Destruir	Encolar	Desencolar
O(1)	O(n)	O(1)	O(1)

3. TDA Lista con nodos enlazados
    1. Una lista con nodos enlazados es similar a la pila con nodos enlazados, pero con mas primitivas. Tal y como vimos arriba, Agregar y quitar por nodo_inicio es siempre O(1), pero el problema llega cuando queremos realizar otras adiciones o disminuciones en posiciones que no son el principio de la lista. Ahi es cuando debemos iterar por el resto de la lista hasta llegar al nodo anterior al que queremos quitar o agregar. De esta forma, podemos realizar correctamente las asignaciones tanto si borramos como si agregamos.
    Al borrar en posición, por ejemplo, debemos conocer el nodo anterior, el cual apunta al que queremos borrar. Liberaríamos el nodo a borrar para luego conectar el nodo anterior al nodo siguiente. Al agregar en posición, debemos también conocer ese nodo anterior, lo haríamos apuntar al nodo que estamos agregando y este nodo que estamos agregando apuntaría al nodo siguiente del anterior.
    2. Complejidad Algorítmica

crear	Destruir	Agregar	Quitar	Agregar en posición	Quitar en posicion
O(1)	O(n)	O(1)	O(1)	O(n)	O(n)

---

3. Detalles de implementación

Compilacion y corrida

El archivo de pruebas se compilan utilizando las lineas `gcc -g -std=c99 -Wall -Wconversion -Wtype-limits -pedantic -Werror -O2 src/*.c pruebas.c -o pruebas 2>&1` y luego se corre mediante el comando `./pruebas 2>&1`

Lista

En mi implementación prioricé la legibilidad al ahorro de líneas, hay funciones (sobre todo las más simples), que se podrían hacer en una sola línea con un retiró, pero preferí crear variables que permitieran entender mejor el paso a paso del código.
Empecé haciendo las diferentes primitivas de la lista. Implementando desde lo más simple a lo más complejo. Primero creando y destruyendo la lista (con sus respectivas pruebas), para pasar luego a la adición y la sutracción de elementos (tomando en cuenta casos especiales como cantidad igual a 0). En el caso de la eliminación, implementé tanto la eliminación de nodos con elementos no cargados en memoria como de elementos que si necesitaran de su liberación para no presentar ningún error de perdida de bytes.
De ahí pasé a la adición y la eliminación de elementos en posiciones, iterando adecuadamente y tomando también en cuenta casos particulares como cantidad mayor a cero, posición igual a cero o posición mayor a cantidad. Para esto modularicé el llenado de nodos, ya que era algo repetitivo y que, si bien modularizarlo no hace la gran diferencia, en mi opinión mejora la legibilidad del código.

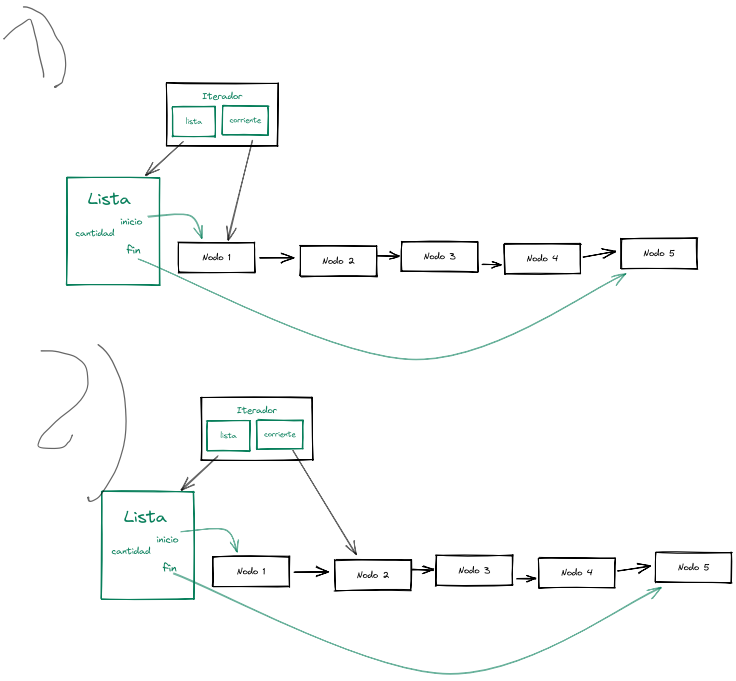
Una vez terminada la parte de carga y descarga de elementos, pasé a la búsqueda de elementos en lista. Estas son similares a las de insertar y quitar en posición, ya que se encargan de iterar nodo a nodo no que la posición pedida sea uno de los casos especiales nombrados arriba (cantidad igual a 0, posición igual a 0 o posición mayor a cantidad). En el caso de `lista_buscar_elemento`, se itera hasta que el comparador devuelva false.

Las funciones lista_ primero, último, vacío y tamanio son por diferencias las más simples, ya que requieren solamente buscar un campo del struct lista y compararlo o devolverlo. No hay mucha complejidad.

Iteradores

Los iteradores son una forma de acceder a los elementos de una lista, sin necesidad de recorrerla completa. Esto se logra con la creación de un puntero que apunta al primer elemento de la lista, y que se mueve a través de los nodos de la misma.

El primero y más importante es el iterador externo, que vive dentro de un struct cargado en memoria y tiene un nodo corriente por el que va avanzando según se le pida. Sus funciones son crear (que se encarga de asignarlo en memoria y apuntar su nodo corriente a nodo inicio), tiene_siguiente (que se fija si el nodo actual es null, de serlo, no tendrá siguiente), avanzar (que avanza una posición en el nodo_corriente), elemento_actual (que devuelve el elemento del nodo corriente) y destruir (que destruye el nodo, no asi los nodos por los que itero porque esos forman parte de la lista, el iterador solo apunta).

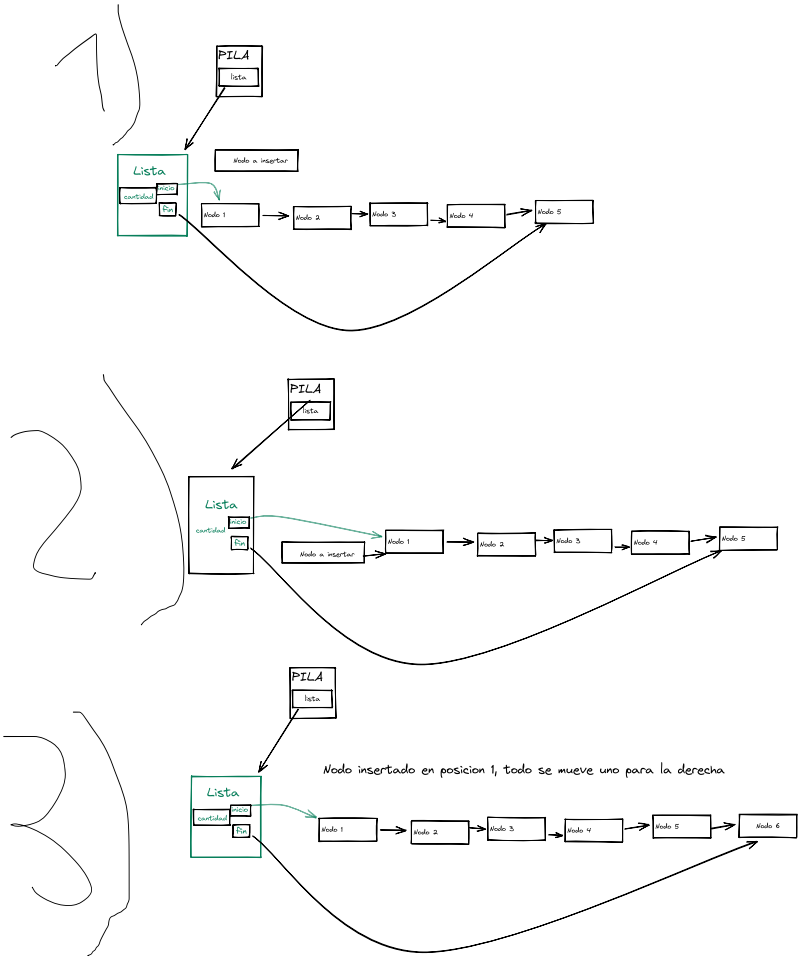


El segundo es el iterador externo, que "pertenece" al usuario. Él decide bajo que condiciones itera y corta, nosotros solo le brindamos la posibilidad de recorrer la lista hasta que la funcion devuelva false, momento en el cual cortamos el ciclo y devolvemos la cantidad de elementos iterados.

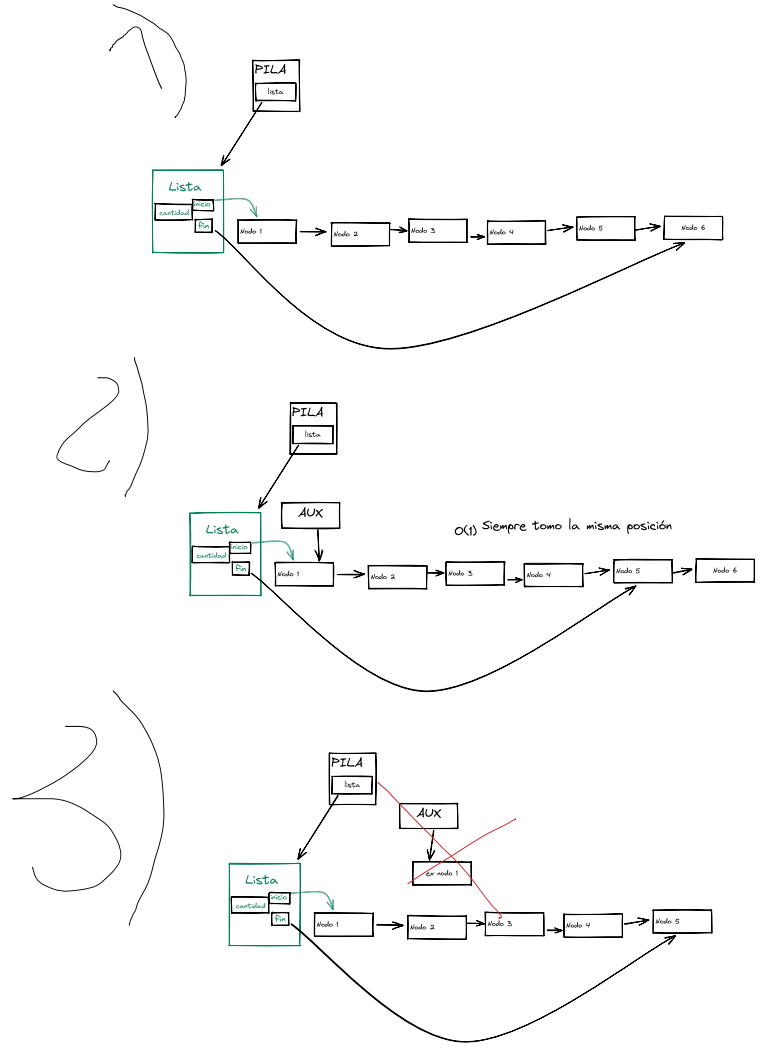
Pila

En la pila, los elementos se almacenan siempre en la primera posición y se quitan tambien por ahí. El ultimo en entrar es el primero en salir. Reutilicé las primitivas de lista con ciertas modificaciones y restricciones, tomando el la posicion de tope, la posición a agregar y la posicion a quitar como 0. De esta forma, todas esas funciones son O(1).

Apilar



Desapilar



Cola

En la cola, los elementos se almacenan en la ultima posición pero se quitan de la primera, el primero en entrar es el primero en salir. Para lograr esto, encolé utilizando lista_insertar (ya que siempre inserta al final y es O(1)) y desecolé quitando el elemento en la posicion 0, al igual que hice con pila.

Detalles de Funciones

1. lista_con_cada_elemento

   Está funcion recibe recibe otra funcion por parametro, esta es creada por el usuario, el cual (si cumple con el contrato) debe desarollarlar para devolver true o false segun si los elementos actuales cumplen con el contexto deseado. Esto hace que lista_con_cada_elemento itere

elemento por elemento hasta que el iterador interno le indique que tiene que parar. En ese momento, devuelve la cantidad de elementos recorridos.
En un paso a paso: Se crea una variable elementos_recorridos inicializada en cero un bool que va a ser el encargado de comunicar si el iterados autoriza que hay que continuar. Incializa el nodo actual como el nodo inicial y empieza a iterar, cortando cuando nos hayamos quedado sin nodos por recorrer o cuando el iterador interno nos lo diga.

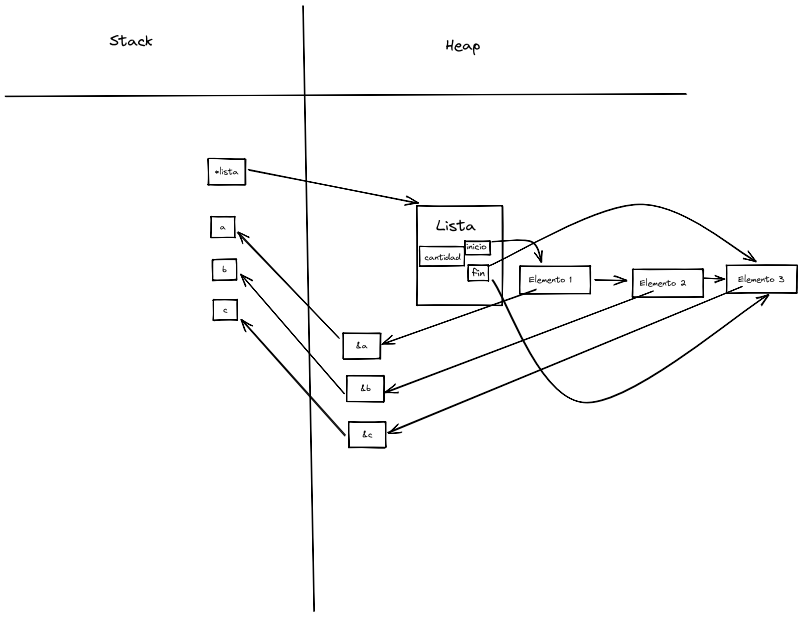
2. lista_insertar_en_posicion()

   Esta función se encarga de insertar el elemento en la posición que corresponda. Para esto evalua diferentes situaciones:

   - Si la posicion es mayor a la cantidad de nodos actuales, lo cual no permitiria insertar nada, la modifica a la ultima posicion posible de la lista.
   - Luego, si no hay cantidad, o sea, si la lista está vacia, inserta el elemento y asigna nodo_inicio y nodo_fin a ambos.
   - Si la posicion en la que se pide ingresar es cero, se carga este nuevo nodo, como nodo_inicio y como siguiente el nodo_inicio anterior
   - Si la posicion no es ni cero ni la maxima (cantidad de nodos actuales) se itera hasta el nodo de posicion anterior al que queremos agregar. Asignamos el siguiente del nodo anterior como el siguiente del nodo actual y el nodo actual como el siguiente del anterior.
   - Como ultima opcion, si la posicion es la ultima de toda la lista. Insertamos normalmente con `lista_insertar` , la cual lo carga al final

4. Diagramas

1. Diagrama de memoria



En este diagrama se puede ver el funcionamiento del stack y del heap en la creacion y el llenado de lista. Siendo que cada uno de los valores en el stack tiene su correspondiente memoria en el heap a la cual apunta cada uno de los nodos respectivamente.