FACULTAD DE INGENIERIA

Universidad de Buenos Aires

TDA – Lista, Pila y Cola

[7541/9515] Algoritmos y Programación II

Segundo cuatrimestre de 2021

| Alumno: | Hurtado, Daniel | | ---------- | ---------- | | Número de padrón: | 107404 | | Email: | dhurtado@fi.uba.ar |

1. Introducción

Este trabajo práctico tiene como objetivo introducir el conocimiento o teoría de lo que es un TDA (Tipo de dato abstracto), cuya definición y/o propósito consisten en la creación de estructuras y funcionalidades definidas que no hagan referencia a datos específicos o contenido irrelevante para su propósito (abstracción), brindando la posibilidad de ser reutilizados en infinitud de situaciones que requieran dichas funcionalidades, sin importar la índole de la información, datos o procesos manejados en donde se requiera implementarla.

En este TDA, se implementará las definiciones de lista (con nodos simplemente enlazados), pila y cola, siendo estas últimas casos específicos de la lista, cuyos requerimientos pueden ser satisfechos en su totalidad con la reutilización de funcionalidades de la lista.

Adicionalmente, en este TDA, se introducen también la noción o enfoque de TDD (Test Development Drive, o en español, desarrollo guiado por pruebas), cuya finalidad es realizar la implementación conociendo de antemano cual sería su correcto y esperado funcionando. Debido a esto, las pruebas encontradas en pruebas.c, son pruebas realizadas por el estudiante, a diferencia de tps anteriores.

2. Teoría

Previo a profundizar en cada uno de los TDA a implementar, seguramente querramos saber el por qué de los mismos. En un lenguaje C, cuyas tipos de datos y funcionalidades son limitadas con respecto a otros lenguajes más completos en esos aspectos como Python o Javascript; los programadores carecemos de maneras nativas de utilizar, o en este caso, organizar de mejor manera gran cantidad de información de

manera accesible y ordenada, sin depender a su vez de las limitaciones de memoria de estos lenguajes fuertemente tipados. Debido a esta razón, se realizan implementaciones de nuevos tipos de datos que nos permitan organizar, agrupar y manejar datos de diferentes tipos.

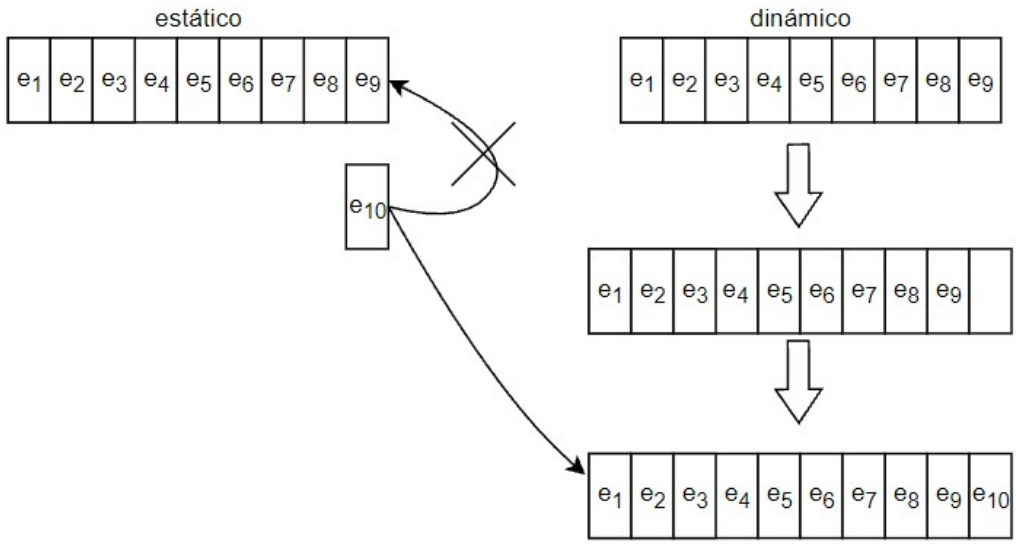
1. Lista

Una lista consiste en una sucesión de datos (que se encuentran en posiciones determinadas), que según la implementación, pueden ser relacionados entre si (su posición con respecto a otro), por contener la información de cada uno de sus elementos en un mismo bloque de memoria (tanto estática o dinámica), o mediante nodos.

- Estático: Consiste en un vector (o array) de datos, el cual contaría con un valor límite preestablecido en su compilación y podría expandirse en momento de ejecución, por lo que su uso es estrictamente en situaciones donde conoceríamos los límites de datos a contener. En este caso no sería necesario hacer uso de memoria dinámica para su implementación.

  Su forma puede verse como (diagrama comparativo, izquierda estático):

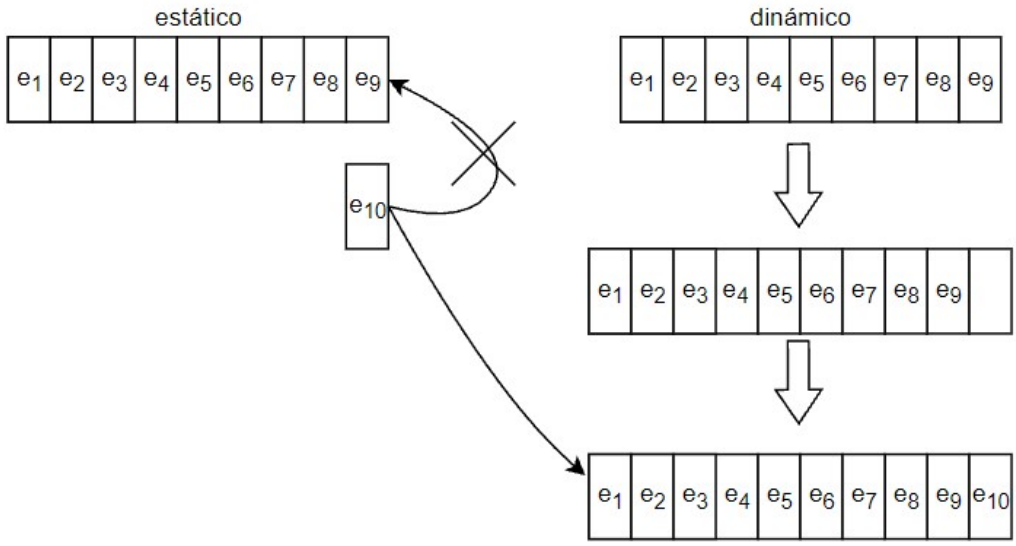
  ```
  void* lista[MAXIMO];
  ...
  void* primero = lista[0];
  ```



- Dinámica (vector): Al igual que la anterior, consistiría en un vector (o array) de datos, pero el mismo en este caso será creado en memoria dinámica en tiempo de ejecución, agrandando el mismo según sea necesario, y recorriendo sus datos con aritmética de punteros.

  Puede verse como (diagrama compartivo, dinámico a la derecha):

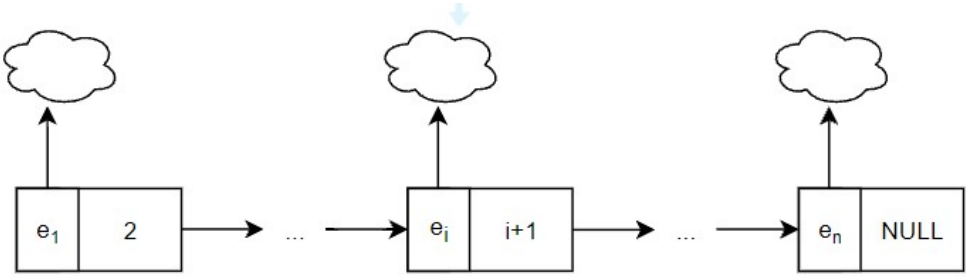
  ```
  void* lista = realloc(sizeof(void*) * CANTIDAD_ELEMENTOS);
  ...
  void* primero = lista[0];
  ```



- Dinámica (nodos): Tiene como finalidad ser capaces de albergar datos de manera ilimitada (con la limitación del hardware) que se puedan encontran en diferentes ubicaciones de la memoria, permitiendo así un mejor aprovechamiento de la misma y mejor rendimiento a la hora de agregar elementos por no tener la necesidad redimensionar nuestra lista de manera constante.

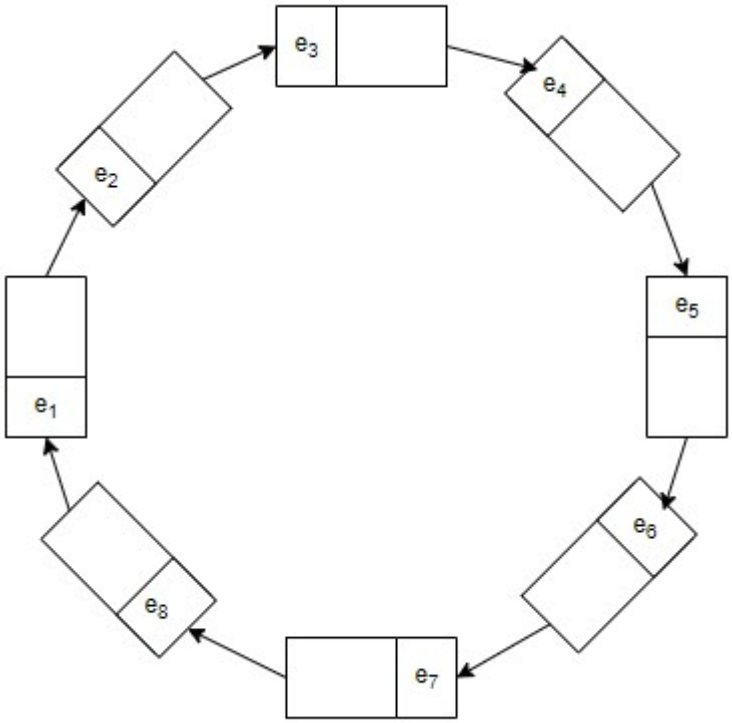
  En este caso debemos conocer primero lo que es un nodo. Son tipos datos estructurados que contienen además del dato o elemento guardado, una referencia o relación entre otros nodos, los cuales en conjunto formarían la lista. Pueden verse como structs de la siguiente manera, variando según la implementación requerida:

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



  En donde elemento, es un puntero sin un tipo de dato específico, que recordando la definición de TDA, nos permite realizar una implementación abstracta y libre de especificidad que no tiene que ver con nuestro problema. Ese ejemplo, se le conoce como nodos simplemente enlazados, pero también existen otras implementaciones tales como:

- Nodos simplemente enlazados circulares: Similar al caso anterior, en cada uno de nuestros nodos contenemos la información al siguiente, con la diferencia de que el último elemento de la lista tendrá la referencia al primero, creandose un ciclo circular entre ellos:



- Nodos doblemente enlazados:
  Por el contrario podríamos implementar una lista doblemente enlazada, que consistiría en el mismo struct antes mencionado, agregando también la referencia al elemento anterior:

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


De esta manera podemos definir una lista de nodos como un tipo de dato que contiene una referencia al primer elemento de la misma, pudiendo contener (según su implementación) a una referencia a su último elemento o la cantidad de elementos que contiene, un ejemplo de esta podría ser la siguiente:

```
typedef struct lista {
  int cantidad_elementos;
  nodo_t* primero;
} lista_t;
```

Conociendo los tipos de lista, podemos definir lo que podemos hacer con sus datos, de manera abstracta:

- Agregar elementos a la lista (en cualquier posición).
- Obtener un elemento de la lista (cualquiera sea su posición).
- Sacar un elemento de la lista (cualquiera sea su posición).

Además claro, de la creación y destrucción de la misma.

2. Pila

Una pila consiste en un tipo de dato abstracto limitado de una lista, la cual puede ser implementada de las mismas opciones antes mencionadas, teniendo como diferencia los siguientes aspectos:

- Solo se pueden agregar elementos al final de la lista
- Solo se pueden sacar elementos del final de la lista

Un ejemplo de este tipo de dato que todos podríamos conocer sería un envase de papas pringles, que consiste en un tubo cerrado a excepción de arriba, en donde la última papa que ingresó al envase, será la última en salir, ya que tendrá papas encimas que deberán salir primero pese haber entrado después.

3. Cola

Una cola al igual que una fila consiste en un tipo de dato abstracto limitado de una lista, la cual además poder ser implementada de las mismas maneras que una lista, tiene las siguientes limitantes:

- Solo se puede agregar elementos al final de la lista
- Solo se puede sacar elementos del inicio de la lista

Esto es lo que se denomina FIFO, First In First Out (Primero en entrar es el primero en salir).

Un ejemplo de la vida cotidiana de este tipo de implementación puede verse, como una fila o cola en el supermercado: la primera persona en formarse es la primera persona en poder pagar e irse, y el resto debe esperar a que salgan los que se encuentre adelante de ellos.

3. Detalles de implementación

La implementación de dicho TDA, gira en torno a implementar una el TDA lista de manera completa, pudiendo reutilizar sus funcionalidades en los TDA de pila y cola. Según requería la consigna, se realizó una implementación de lista con nodos simplemente enlazados entre sí.

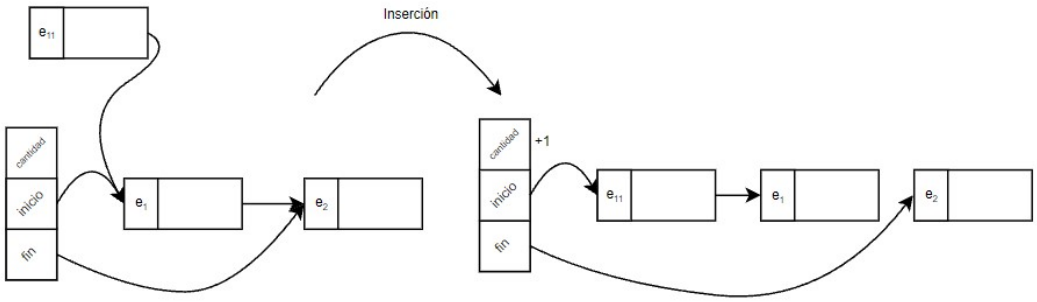
Entre las funciones requeridas de implementar, de mayor complejidad tenemos:

1. Función: lista_insertar_en_posicion Consiste en insertar un elemento dado en la posición específica de una lista. Al ser diversas las posibles resultados de

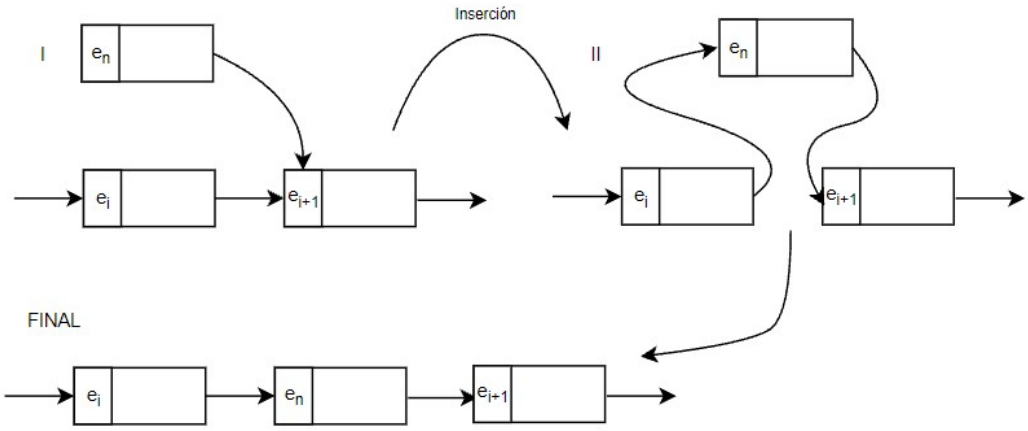
dicha función, se optó por implementarlo de manera simplificada diviendo en casos según sea necesario:

1- La lista está vacía: El caso más fácil, se guardará el elemento en un nuevo nodo que no tendrá un nodo siguiente, y este mismo será el nuevo nodo inicial y a su vez nodo final de nuestra lista.

2- La lista no está vacía, la posición a insertar es 0: En este caso, guardamos el elemento recibido en nuevo nodo y la referencia de siguiente al actual nodo inicial de la lista, para posteriormente establecer este nuevo nodo como el nodo inicial.



3- La lista no está vacía y la posición no es la inicial: En este caso, guardamos el elemento recibido en un nuevo nodo y buscaremos el nodo anterior a la posición solicitada (y en caso esta posición no exista, a la última posición de la lista), una vez encontrado, reemplazaremos la referencia de siguiente de dicho nodo al nuevo, guardando también la referencia al que anteriormente ese nodo anterior tenía como siguiente, y asignandola al nuevo nodo.



En caso la posición no exista, quiere decir que sumaremos el nodo al final de la lista como se mencionó antes, por lo que además de insertarlo, actualizaremos el nodo_final de nuestra lista al nuevo nodo.

2. Función: lista_quitar_de_posicion Al igual que el caso anterior, tenemos diversas situaciones:

1- Quitar en la posición inicial: Asignamos como nuevo nodo inicial al nodo siguiente al actual inicial, devolviendo el elemento quitado.

2- Quitar en una posición diferente a la inicial: Recorremos la lista hasta llegar a la posición anterior a la posición a quitar (o si la posición solicitada es mayor al la cantidad de elementos, el último) y reemplazaremos la referencia al siguiente por el siguiente del siguiente (de la posicion a quitar, en caso tenga siguiente). Y adicionalmente, si esta posición a reemplazar es la última, reasignaremos el nodo final al anterior de nuestra lista.

3. Función: lista_con_cada_elemento

   Esta funcion nos permite iterar cada uno de los elementos contenidos en nuestra lista, aplicando una función recibida como parámetro, iterando hasta que dicha función devuelve false.