75.41 Algoritmos y Programación II
Tda Iterador

Dr. Mariano Méndez[1]

[1]Facultad De Ingeniería. Universidad de Buenos Aires

10 de marzo de 2020

1. Introducción

A partir del tda lista, se puede extraer un comportamiento común a muchos otros tdas. Este comportamiento consiste **en poder recorrer secuencialmente los datos almacenados en el tda**, por ejemplo, sin tener que ser conscientes de la estructura del mismo. para ello se utiliza una estructura o tda llamada **Iterador**. Muchas son las operaciones que se pueden realizar a través de un iterador. Algunas de estas son :

- Primero()
- Siguiente()
- HayMas()
- ElementoActual()

Suponer que se quiere determinar cuantos elementos de una lista son pares o impares:

```
lista_t *lista = ...; /// la lista vino de algún lado

int impar = 0;
int par = 0;

for (int i = 0; i < list_size(list); ++i){
        elemento_t elem = lista_obtener(lista, i);
        if (elemento.i % 2 == 0){
                ++par;
        }
        else{
                ++impar;
        }
}
```

En esta pequeña porción de código fuente existe un hecho que no es tan obvio. Recordando Análisis de Algoritmos, ¿Cuan eficiente es? ¿$O(n)$? Bueno, justamente no!, el algoritmo es $O(n^2)$, algo inaceptable para una estructura tan sencilla como está.

Existen dos tipos de Iteradores:

- **Iterador Intero**: maneja la iteración dentro de la estructura de datos.
- **Iterador Externo**: es un tda que permite manejar la iteración mediante sus propias primitivas.

1.1. Iterador Interno

Un iterador interno permite recorrer todos los elementos de un tda sin tener que controlar el ciclo en el cual se recorre el mismo. Para ello normalmente un iterador interno en una lista es una función que recibe tres parámetros:

- la lista

- un puntero a una función, que recibe un dato como el contenido en el nodo, otro puntero extra por si la función lo requiere
- un puntero a void extra que funciona como memoria común a todos los nodos visitados.

```
void lista_iterar(lista_t *lista, bool visitar(void *dato, void *extra), void *extra);
```

Por ejemplo, suponer que se quiere contar alguna propiedad de todos los nodos de una lista utilizando un iterador interno. Para ello hay que crear la función visitar que siempre tiene que ser del estilo:

```
bool  visitar (void *dato, void *extra)
```

supongamos que se quieren contar los números pares, esta podría ser el prototipo de función visitar:

```
bool  contar_pares (void *numero, void *contador){
        int * un_numero=numero;
        int * un_contador=contador;

        if (*un_numero%2==0) {
                (*un_contador)++;
        }
        return true;
}
```

Por otro lado se necesita una variable que sea un acumulador, la llamada al iterador interno quedaría:

```
lista_iterar(lista, contar_pares, &contador_de_pares);
```

2. Iterador Externo

un iterador externo es un **tda** que provee un set de primitivas especiales para recorrer una estructura (**otro tda**).

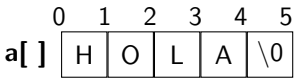
2.1. El Qué

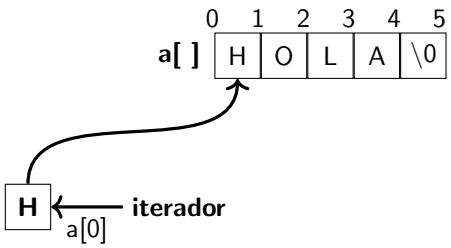
Si se supone que se tiene un arreglo de 10 elementos enteros en C, poder recorrer dicho elemento usando un iterador externo con las siguientes operaciones:

- Crear()
- Primero()
- Siguiente()
- HaySiguiente()
- ElementoActual()

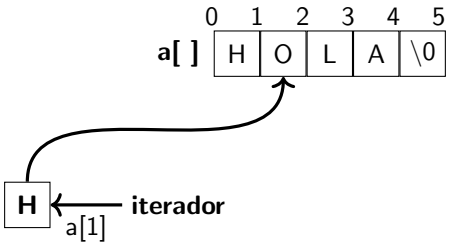
En primer lugar se crea el iterador pasando el vector:

```
iterador_crear(a[]);
```

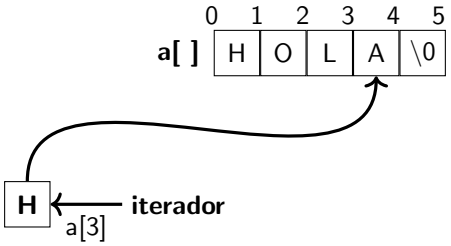




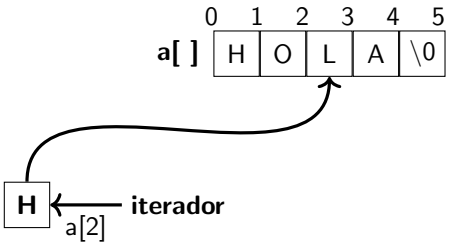
```
iterador_hay_siguiente()==true;
iterador_siguiente();
```



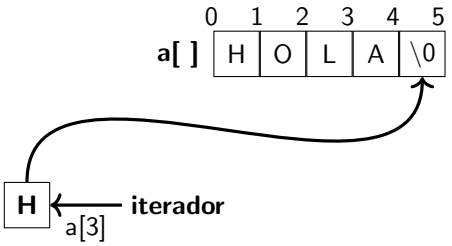

```
iterador_hay_siguiente()==true;
iterador_siguiente();
```




```
iterador_hay_siguiente()==true;
iterador_siguiente();
```




```
iterador_hay_siguiente()==true;
iterador_siguiente();
```




```
iterador_hay_siguiente()==false;
```

La iteración clásica quedaría de la siguiente forma:

```
iterador=iterador_crear(lista);
while ( iterador_hay_siguiente(iterador) ){
        void * elemento = iterador_elemento(iterador);
        proceso(elemento);
        iterador_siguiente(iterador);
}
iterador_destruir(iterador);
```

2.2. El Cómo

Un iterador externo de una lista puede ser implementado como un objeto opaco que mantiene un puntero a la lista. Cada vez que queremos que el iterador avance hacia el próximo elemento de la lista, solo hay que pedirle que de un paso hacia adelante. El factor clave que hace esto posible es que el iterador mantiene el estado de la iteración.

Con una lista adecuadamente encapsulada, es posible construir un iterador eficiente, pero cuidado que el iterador debe conocer cómo está representada la lista, por ende el iterador es implementados conjuntamente con la lista.

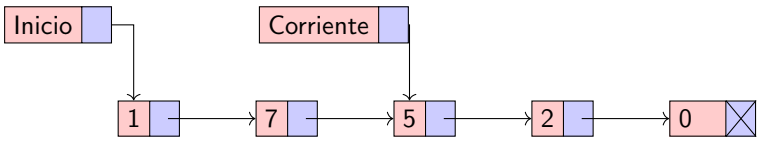
2.3. La Interfaz del Iterador

¿Qué es una interface, cuando se habla de un tds? Un **tda** consiste no solo de operaciones, sino también de valores de los datos subyacentes y de restricciones en las operaciones. Una "**interfaz**" generalmente se refiere **solo a las operaciones**, y quizás a algunas de las restricciones en las operaciones, en particular las pre y post condiciones, pero no otras restricciones, como las relaciones entre las operaciones.

El iterador soportará las siguientes operaciones:

- Chequear si el iterador esta posicionado al final de la lista
- Mover el iterador hacia adelante una posición
- Obtener el elemento corriente
- Eliminar el elemento corriente
- Resetear el iterador (posicionarlo en el primer elemento otra vez)

para ello el iterador se define como un puntero al primer elemento de la lista (o a la lista) y un puntero a un nodo , llamado nodo actual.



2.4. Implementación

En primer lugar hay que definir la estructura del iterador, como se explicó anteriormente es mejor que el usuario, sepa la menor cantidad posible de detalles de la implementación, por ello se ofuscaran algunas cosas:

```
/// Esto va en lista.h
typedef struct iterterador iterador_t;

/// Esto va en lista.c
struct iterador {
        nodo_t*                 corriente;
        lista_t*        lista;
};
```

la interfaz del iterador es :

```
iterador_t *iterador_crear(lista_t *lista)

bool iterator_hay_siguiente (iterador_t *iterrador)

elem_t iterador_elemento(iterador_t *iterador)

void iterador_siguiente(iterador_t *iterador)

void iterador_primero(iterador_t *iterador)

void iterador_destruir(iterador);
```

2.5. Implementación de la Interfaz

Una de las cosas importantes en el diseño del iterador, está puesto en que la referencia al primer elemento de la lista siempre tiene que se conocido. El elemento corriente es el elemento al cual apunta el iterador en este preciso instante.

Crear

```
struct iterador {
        nodo_t *corriente;
        lista_t *lista;
};

iterador_t *iterador_crear(lista_t *lista){

        if (!lista)
                return NULL;

        iterador_t *resultado = malloc(sizeof(struct iterador));

        result->corriente = list->inicio;
        result->lista = lista; /// El iterador sabe de donde vino

        return resultado;
}
```

Una vez hecho esto se puede resetear el iterador de la siguiente forma:

```
void iterador_primero(iterador_t *iterador){
        iterador->corriente = iterador->lista->inicio;
}
```

Chequear que el iterador haya alcanzado el final de la lista solo requiere chequear que si el siguiente del nodo corriente es NULL entonces la lista no tiene más elementos.

```
bool iterador_hay_siguiente(iterador_t *iterador)
{
        return iterador->corriente->siguiente != NULL;
}
```

El Código para obtener el elemento corriente

```
elemento_t iterador_elemento(iterador_t *iterador)
{
        if (iterator->corriente)
                return iterator->corriente->elemento;
}
```

Mover el elemento corriente al próximo en la lista

```
void iterador_siguiente(iterador_t *iterador){
        if (iterador->corriente)
                return iterador->corriente = iterador->corriente->siguiente;
}
```

Finalmente el destructor libera el espacio en el heap.

```
void iterator_destruir(iterador_t *iterador){
        free(iterador);
}
```

Referencias