75.41 Algoritmos y Programación II Curso 4
TDA Lista

Simplemente enlazada

30 de octubre de 2020

1. Enunciado

Se pide implementar una Lista simplemente enlazada. Para ello se brindan las firmas de las funciones públicas a implementar y se deja a criterio del alumno la creación de las funciones privadas del TDA para el correcto funcionamiento de la Lista cumpliendo con las buenas prácticas de programación.

Dentro de este TDA, se pide incluir también la implementación de las funciones básicas de los TDAs Pila y Cola, cumpliendo así que este satisfaga el comportamiento de las tres estructuras.

Adicionalmente se pide la creación de un iterador interno y uno externo para la lista.

2. lista.h

```c
#ifndef __LISTA_H__
#define __LISTA_H__

#include <stdbool.h>
#include <stddef.h>

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

typedef struct lista{
    nodo_t* nodo_inicio;
    nodo_t* nodo_fin;
    size_t cantidad;
}lista_t;

typedef struct lista_iterador{
    nodo_t* corriente;
    lista_t* lista;
}lista_iterador_t;

/*
 * Crea la lista reservando la memoria necesaria.
 * Devuelve un puntero a la lista creada o NULL en caso de error.
 */
lista_t* lista_crear();

/*
 * Inserta un elemento al final de la lista.
 * Devuelve 0 si pudo insertar o -1 si no pudo.
 */
int lista_insertar(lista_t* lista, void* elemento);

/*
 * Inserta un elemento en la posicion indicada, donde 0 es insertar
 * como primer elemento y 1 es insertar luego del primer elemento.
 * En caso de no existir la posicion indicada, lo inserta al final.
 * Devuelve 0 si pudo insertar o -1 si no pudo.
 */
```

```c
int lista_insertar_en_posicion(lista_t* lista, void* elemento, size_t posicion);

/*
 * Quita de la lista el elemento que se encuentra en la ultima posición.
 * Devuelve 0 si pudo eliminar o -1 si no pudo.
 */
int lista_borrar(lista_t* lista);

/*
 * Quita de la lista el elemento que se encuentra en la posición
 * indicada, donde 0 es el primer elemento.
 * En caso de no existir esa posición se intentará borrar el último
 * elemento.
 * Devuelve 0 si pudo eliminar o -1 si no pudo.
 */
int lista_borrar_de_posicion(lista_t* lista, size_t posicion);

/*
 * Devuelve el elemento en la posicion indicada, donde 0 es el primer
 * elemento.
 *
 * Si no existe dicha posicion devuelve NULL.
 */
void* lista_elemento_en_posicion(lista_t* lista, size_t posicion);

/*
 * Devuelve el último elemento de la lista o NULL si la lista se
 * encuentra vacía.
 */
void* lista_ultimo(lista_t* lista);

/*
 * Devuelve true si la lista está vacía o false en caso contrario.
 */
bool lista_vacia(lista_t* lista);

/*
 * Devuelve la cantidad de elementos almacenados en la lista.
 */
size_t lista_elementos(lista_t* lista);

/*
 * Apila un elemento.
 * Devuelve 0 si pudo o -1 en caso contrario.
 */
int lista_apilar(lista_t* lista, void* elemento);

/*
 * Desapila un elemento.
 * Devuelve 0 si pudo desapilar o -1 si no pudo.
 */
int lista_desapilar(lista_t* lista);

/*
 * Devuelve el elemento en el tope de la pila o NULL
 * en caso de estar vacía.
 */
void* lista_tope(lista_t* lista);

/*
 * Encola un elemento.
 * Devuelve 0 si pudo encolar o -1 si no pudo.
 */
int lista_encolar(lista_t* lista, void* elemento);

/*
 * Desencola un elemento.
 * Devuelve 0 si pudo desencolar o -1 si no pudo.
 */
int lista_desencolar(lista_t* lista);

/*
 * Devuelve el primer elemento de la cola o NULL en caso de estar
 * vacía.
 */
void* lista_primero(lista_t* lista);
```

```c

/*
 * Libera la memoria reservada por la lista.
 */
void lista_destruir(lista_t* lista);

/*
 * Crea un iterador para una lista. El iterador creado es válido desde
 * el momento de su creación hasta que no haya mas elementos por
 * recorrer o se modifique la lista iterada (agregando o quitando
 * elementos de la lista).
 *
 * Al momento de la creación, el iterador queda listo para devolver el
 * primer elemento utilizando lista_iterador_elemento_actual.
 *
 * Devuelve el puntero al iterador creado o NULL en caso de error.
 */
lista_iterador_t* lista_iterador_crear(lista_t* lista);

/*
 * Devuelve true si hay mas elementos sobre los cuales iterar o false
 * si no hay mas.
 */
bool lista_iterador_tiene_siguiente(lista_iterador_t* iterador);

/*
 * Avanza el iterador al siguiente elemento.
 * Devuelve true si pudo avanzar el iterador o false en caso de
 * que no queden elementos o en caso de error.
 *
 * Una vez llegado al último elemento, si se invoca a
 * lista_iterador_elemento_actual, el resultado siempre será NULL.
 */
bool lista_iterador_avanzar(lista_iterador_t* iterador);

/*
 * Devuelve el elemento actual del iterador o NULL en caso de que no
 * exista dicho elemento o en caso de error.
 */
void* lista_iterador_elemento_actual(lista_iterador_t* iterador);

/*
 * Libera la memoria reservada por el iterador.
 */
void lista_iterador_destruir(lista_iterador_t* iterador);

/*
 * Iterador interno. Recorre la lista e invoca la funcion con cada elemento de
 * la misma. Dicha función puede devolver true si se deben seguir recorriendo
 * elementos o false si se debe dejar de iterar elementos.
 *
 * La función retorna la cantidad de elementos iterados o 0 en caso de error.
 */
size_t lista_con_cada_elemento(lista_t* lista, bool (*funcion)(void*, void*), void *contexto);

#endif /* __LISTA_H__ */
```

3. Compilación y Ejecución

El TDA entregado deberá compilar y pasar las pruebas dispuestas por la cátedra sin errores, adicionalmente estas pruebas deberán ser ejecutadas sin pérdida de memoria.

Compilación:

```
gcc *.c -o lista_se -g -std=c99 -Wall -Wconversion -Wtype-limits -pedantic -Werror -O0
```

Ejecución:

```
valgrind --leak-check=full --track-origins=yes --show-reachable=yes ./lista_se
```

4. Minipruebas

Se les brindará un lote de minipruebas, las cuales recomendamos fuertemente sean ampliadas (primero porque va a ser la única forma de llegar a una buena implementación sin errores, **pero principalmente porque son parte de la nota**) ya que las minipruebas no son exhaustivas y no prueban los casos borde, solo son un ejemplo de como agregar, eliminar, obtener elementos de la lista y qué debería verse en la terminal en el **caso feliz**.

Minipruebas:

```c
#include "lista.h"
#include <stdio.h>

bool mostrar_elemento(void* elemento, void* contador){
    if(elemento && contador)
        printf("Elemento %i: %c \n", (*(int*)contador)++, *(char*)elemento);
    return true;
}

void probar_operaciones_lista(){
    lista_t* lista = lista_crear();
    char a='a', b='b', c='c', d='d', w='w';

    lista_insertar(lista, &a);
    lista_insertar(lista, &c);
    lista_insertar_en_posicion(lista, &d, 100);
    lista_insertar_en_posicion(lista, &b, 1);
    lista_insertar_en_posicion(lista, &w, 3);

    lista_borrar_de_posicion(lista, 3);

    printf("Elementos en la lista: ");
    for(size_t i=0; i<lista_elementos(lista); i++)
        printf("%c ", *(char*)lista_elemento_en_posicion(lista, i));

    printf("\n\n");

    printf("Imprimo la lista usando el iterador externo: \n");
    lista_iterador_t* it = NULL;

    for(it = lista_iterador_crear(lista);
        lista_iterador_tiene_siguiente(it);
        lista_iterador_avanzar(it))
        printf("%c ", *(char*)lista_iterador_elemento_actual(it));
    printf("\n\n");

    lista_iterador_destruir(it);

    int contador=0;
    size_t elementos_recorridos = 0;
    printf("Imprimo la lista usando el iterador interno: \n");
    elementos_recorridos = lista_con_cada_elemento(lista, mostrar_elemento, (void*)&contador);

    printf("Recorri %lu elementos con el iterador interno y sume %i elementos\n",
    elementos_recorridos, contador);

    printf("\n");
    lista_destruir(lista);
}

void probar_operaciones_cola(){
    lista_t* cola = lista_crear();

    int numeros[]={1,2,3,4,5,6};

    for(size_t i=0; i<sizeof(numeros)/sizeof(int); i++){
        printf("Encolo %i\n", numeros[i]);
        lista_encolar(cola, &numeros[i]);
    }

    printf("\nDesencolo los numeros y los muestro: ");
    while(!lista_vacia(cola)){
        printf("%i ", *(int*)lista_primero(cola));
        lista_desencolar(cola);
    }
    printf("\n");
```

```c
    lista_destruir(cola);
}

void probar_operaciones_pila(){
    lista_t* pila = lista_crear();
    char* algo="somtirogla";

    for(int i=0; algo[i]!= 0; i++){
        printf("Apilo %c\n", algo[i]);
        lista_apilar(pila, &algo[i]);
    }

    printf("\nDesapilo y muestro los elementos apilados: ");
    while(!lista_vacia(pila)){
        printf("%c", *(char*)lista_tope(pila));
        lista_desapilar(pila);
    }
    printf("\n");
    lista_destruir(pila);
}

int main(){

    printf("Pruebo que la lista se comporte como lista\n");
    probar_operaciones_lista();

    printf("\nPruebo el comportamiento de cola\n");
    probar_operaciones_cola();

    printf("\nPruebo el comportamiento de pila\n");
    probar_operaciones_pila();

    return 0;
}
```

La salida por pantalla luego de correrlas con valgrind debería ser:

```
# valgrind ./lista_simple_minipruebas
==19449== Memcheck, a memory error detector
==19449== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==19449== Using Valgrind-3.16.1 and LibVEX; rerun with -h for copyright info
==19449== Command: ./lista_simple_minipruebas
==19449==
Pruebo que la lista se comporte como lista
Elementos en la lista: a b c d

Imprimo la lista usando el iterador externo:
a b c d

Imprimo la lista usando el iterador interno:
Elemento 0: a
Elemento 1: b
Elemento 2: c
Elemento 3: d
Recorri 4 elementos y sume 4 elementos


Pruebo el comportamiento de cola
Encolo 1
Encolo 2
Encolo 3
Encolo 4
Encolo 5
Encolo 6

Desencolo los numeros y los muestro: 1 2 3 4 5 6

Pruebo el comportamiento de pila
Apilo s
Apilo o
Apilo m
Apilo t
Apilo i
Apilo r
Apilo o
Apilo g
Apilo l
```

```
Apilo a

Desapilo y muestro los elementos apilados: algoritmos
==19449==
==19449== HEAP SUMMARY:
==19449==     in use at exit: 0 bytes in 0 blocks
==19449==   total heap usage: 26 allocs, 26 frees, 1,448 bytes allocated
==19449==
==19449== All heap blocks were freed -- no leaks are possible
==19449==
==19449== For lists of detected and suppressed errors, rerun with: -s
==19449== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5. Entrega

La entrega deberá contar con todos los archivos necesarios para compilar y ejecutar correctamente el TDA.

Dichos archivos deberán formar parte de un único archivo **.zip** el cual será entregado a través de la plataforma de corrección automática **Chanutron2021** (patente pendiente).

El archivo comprimido deberá contar, además del TDA con:

- El archivo con las pruebas agregadas para comprobar el correcto funcionamiento del TDA. Es importante notar que, **en este TDA es imprescindible realizar pruebas de caja negra y/o blanca y que son parte de la nota**.
- Un **Readme.txt** que contenga una primera sección, en donde se deberá explicar qué es lo entregado, como compilarlo (línea de compilación), como ejecutarlo (línea de ejecución) y el funcionamiento particular de la implementación elegida(no es necesario detallar función por función, solamente explicar como funciona el código) y por qué se eligió dicha implementación. En una segunda sección, se deberán desarrollar los siguientes conceptos teóricos:
  - ¿Qué es lo que entendés por una lista? ¿Cuáles son las diferencias entre ser simple o doblemente enlazada?
  - ¿Cuáles son las características fundamentales de las Pilas? ¿Y de las Colas?
  - ¿Qué es un iterador? ¿Cuál es su función?
  - ¿En qué se diferencia un iterador interno de uno externo?
- El enunciado.