Corrección Dall’acqua Denise - TDA Lista:

Facultad de Ingeniería de la Universidad de Buenos Aires.

Algoritmos y Programación II [7541/9515].

Primer cuatrimestre 2022.

Corrección por Gamberale Luciano Martín.

---

1. General:

1.1. Aspectos positivos

- Buen informe.
- Se chequea que se reserve correctamente la memoria.
- Se verifican los parámetros recibidos en todas las funciones.
- Se hace un buen manejo de punteros.
- Se reutilizan funciones correctamente.
- La implementación del TDA Lista pasa las pruebas propuestas por el alumno.
- La implementación del TDA Lista pasa las pruebas propuestas por la cátedra.

1.2. Aspectos negativos

- No se incluyó el enunciado del Trabajo Práctico.
- No se informa la manera en que debe ser compilado y ejecutado el programa.
- No se incluyó la biblioteca `pa2m.h`.
- Faltaron casos de prueba.
- Faltó información en la descripción de las pruebas.
- Se observaron ciertas líneas de código mal indentadas.
- No se respeta el estilo del código kernel de Linux para el espaciado.
- No realiza una división acerca de las distintas acciones que se realizan en una función mediante la utilización de saltos de línea.
- Las funciones son extensas por falta de modularización (no se crearon funciones auxiliares).

2. Código:

2.1. Comentarios generales sobre el código:

- Existen ciertas líneas mal indentadas. A continuación se citarán los fragmentos observados:
  - `lista_insertar()`

    ```
      if(!lista) //mal indentado
      return NULL;
    nodo_t *nuevo_nodo = malloc(sizeof(nodo_t));
    ```

  - `lista_destruir_todo()`

    ```
    while(lista->nodo_inicio){
        struct nodo *auxiliar=lista->nodo_inicio->siguiente; //mal indentado
      if(funcion)
        funcion(lista->nodo_inicio->elemento);
      free(lista->nodo_inicio);
      lista->nodo_inicio = auxiliar;
      lista->cantidad--;
    }
    ```

- Dentro de las funciones no se colocaron saltos de líneas para diferenciar las partes de la misma, como ser inicialización de variables, reservación de memoria, entre otras. En el siguiente fragmento de código se visualiza lo comentado anteriormente:

  ```
  lista_t *lista_insertar(lista_t *lista, void *elemento)
  {
      if(!lista)
      return NULL;
    nodo_t *nuevo_nodo = malloc(sizeof(nodo_t));
    if(!nuevo_nodo){
      return ERROR_1;
    }
  ```

```
  nuevo_nodo->elemento = elemento;
  nuevo_nodo->siguiente = NULL;
  if(lista_tamanio(lista)!=VACIO){
    lista->nodo_fin->siguiente = nuevo_nodo;
  }else{
    lista->nodo_inicio = nuevo_nodo;
  }
  lista->nodo_fin = nuevo_nodo;
  (lista->cantidad)++
  return lista;
}
```

Una solución posible podría haber sido la siguiente, en donde se diferencian las distintas acciones que se realizan dentro de la función para que la misma sea más clara y explícita a la hora de leerla:

```
lista_t *lista_insertar(lista_t *lista, void *elemento)
{
  if (!lista)
    return NULL;

  nodo_t *nuevo_nodo = malloc(sizeof(nodo_t));
  if (!nuevo_nodo)
    return NULL;
  nuevo_nodo->elemento = elemento;
  nuevo_nodo->siguiente = NULL;

  if (lista_tamanio(lista) != VACIO)
    lista->nodo_fin->siguiente = nuevo_nodo;
  else
    lista->nodo_inicio = nuevo_nodo;

  lista->nodo_fin = nuevo_nodo;
  (lista->cantidad)++
  return lista;
}
```

- Se observaron ciertos casos en donde el código no se encuentra correctamente espaciado según el estilo de código del kernel de Linux, donde indica lo siguiente: *"Utilice un espacio en ambos lados de la mayoría de los operadores binarios y ternarios, como todos estos operadores"* (3.1. Espaciado):
  - `lista_elemento_en_posicion()`

    ```
    if((!lista)||(posicion>=(lista_tamanio(lista))))
        return ERROR_1;
    ```

  - `lista_ultimo()`

    ```
    if((!lista)||(lista_tamanio(lista)==VACIO)){
        return ERROR_1;
    ```

- No se realizaron funciones auxiliares para modularizar ciertos aspectos de algunas funciones, y poder así generar una mayor claridad en el código y una menor cantidad de líneas. Algunas de las abstracciones más generales que se podrían haber realizado son:
  - `nodo_crear()`
  - `obtener_nodo_de_posicion()`
- A modo de comentario, no era necesario verificar si la pila o cola recibida era válida dentro de las funciones implementadas, ya que las funciones de lista manejaban los casos en donde la lista era nula. Por lo tanto hubiese alcanzado con retornar la función implementada para lista directamente. De todas maneras te felicito por la reutilización de las funciones de lista para cola y pila.
- La manera en que se plantea la realización de una pila a partir de una lista conlleva a una complejidad algorítmica de O(n) a la hora de desapilar un elemento. Esto es porque se optó por insertar los elementos al final de una lista en la acción de apilar (complejidad algorítmica O(1)) y eliminar el último elemento de una lista en la acción de desapilar (complejidad algorítmica O(n)). Para reducir la complejidad algorítmica se debería haber optado por insertar los elementos en la primera posición de una lista en la acción de apilar (complejidad algorítmica O(1)) y eliminar el primer elemento de una lista en la acción de desapilar (complejidad algorítmica O(1)).

3. Informe:

- Te felicito por el informe realizado.
- Se explican correctamente todos los ítems teóricos.
- Se detallan correctamente las funciones implementadas.
- Se realiza un análisis profundo acerca de las complejidades algorítmicas de cada una de las funciones dependiendo de los parámetros recibidos.
- Se realizan gráficos con grandes detalles sobre las funciones implementadas.
- En la sección *"2.1.5 quitar_lista_posicion()"* se puede observar que hay un error en cuanto a la complejidad algorítmica para el caso en que la posición dada es mayor a la cantidad de elementos que hay en la lista, ya que para estos casos en el código implementado se utiliza la función `lista_quitar()`, la cual posee una complejidad algorítmica de O(n).

4. Pruebas:

4.1. Creación

- [x] ~~Se prueba crear una lista válida y verificar que empieza vacía~~

4.2. Inserción

- [x] ~~Se prueba insertar al final en una lista nula~~
- [ ] Se prueba insertar en cualquier posición en una lista nula
- [x] ~~Se prueba insertar al final en una lista vacía y se verifica que está en la posición correcta~~
- [ ] Se prueba insertar en cualquier posición en una lista vacía y se verifica que está en la posición correcta
- [x] ~~Se prueba insertar en cualquier posición en una lista con elementos y se verifica que está en la posición correcta~~
- [x] ~~Se prueba insertar un elemento nulo en una lista válida~~
- [ ] Se prueba insertar en una posición inexistente en la lista, y se verifica que la posición de ese elemento es la última
- [x] ~~Se prueba insertar varios elementos y que todos estén en la posición que le corresponde~~

4.3. Eliminación

- [x] ~~Se prueba quitar de una lista nula~~
- [ ] Se prueba quitar de cualquier posición de una lista nula
- [ ] Se prueba quitar de una lista vacía
- [ ] Se prueba quitar de cualquier posición de una lista vacía
- [x] ~~Se prueba quitar un elemento que sí existía~~
- [ ] Se prueba quitar de una posición inexistente en la lista y se verifica que se eliminó el último elemento de la lista

4.4. Obtención

- [ ] Se prueba obtener un elemento de una lista nula
- [ ] Se prueba obtener un elemento de una lista vacía
- [ ] Se prueba obtener un elemento de una posición inexistente
- [ ] Se prueba obtener un elemento existente en la lista
- [ ] Se prueba obtener un elemento que sí se encuentra en la lista, quitarlo, y luego verificar que ya no se encuentra

4.5. Buscar

- [x] ~~Se prueba buscar un elemento de una lista nula~~
- [ ] Se prueba buscar un elemento de una lista vacía
- [ ] Se prueba buscar un elemento de una lista con un comparador nulo
- [ ] Se prueba buscar un elemento con un comparador que devuelve siempre distinto de cero
- [ ] Se prueba buscar un elemento con un comparador que devuelve siempre cero y se obtiene el primer elemento
- [x] ~~Se prueba buscar un elemento inexistente en la lista~~
- [x] ~~Se prueba buscar un elemento existente en la lista~~
- [ ] Se prueba buscar un elemento que sí se encuentra en la lista, quitarlo, y luego verificar que ya no se encuentra

4.6. Cantidad de elementos y contener

- [x] ~~Se prueba obtener la cantidad de elementos de una lista nula~~
- [x] ~~Se prueba obtener la cantidad de elementos de una lista vacía~~
- [x] ~~Se prueba obtener la cantidad de elementos de una lista con elementos~~
- [x] ~~Se prueba verificar si una lista nula está o no vacía~~
- [ ] Se prueba verificar si una lista vacía está o no vacía
- [ ] Se prueba verificar si una lista con elementos está o no vacía

4.7. Primero y último

- [x] ~~Se prueba obtener el primer elemento de una lista nula~~
- [x] ~~Se prueba obtener el último elemento de una lista nula~~
- [ ] Se prueba obtener el primer elemento de una lista vacía
- [ ] Se prueba obtener el último elemento de una lista vacía
- [ ] Se prueba obtener el primer elemento de una lista con elementos
- [ ] Se prueba obtener el último elemento de una lista con elementos

4.8. Iterador externo

- [x] ~~Se prueba crear un iterador externo con una lista nula~~
- [x] ~~Se prueba crear un iterador externo con una lista con elementos~~
- [ ] Se prueba avanzar un iterador nulo

- [x] ~~Se prueba avanzar un iterador que todavía tiene elementos para iterar~~
- [x] ~~Se prueba avanzar un iterador que ya no tiene elementos para iterar~~
- [ ] Se prueba iterar el elemento actual de un iterador nulo
- [x] ~~Se prueba iterar el elemento actual de un iterador que todavía tiene elementos para iterar~~
- [ ] Se prueba iterar el elemento actual de un iterador que ya no tiene elementos para iterar
- [ ] Se prueba verificar si un iterador nulo tiene más elementos para iterar
- [x] ~~Se prueba verificar si un iterador que tiene más elementos para iterar, los tenga~~
- [x] ~~Se prueba verificar si un iterador que no tiene más elementos para iterar, los tenga~~

4.9. Iterador interno

- [ ] Se prueba iterar una lista nula
- [ ] Se prueba iterar una lista vacía
- [ ] Se prueba iterar una lista con una función nula
- [ ] Se prueba iterar una lista con un auxiliar nulo
- [ ] Se prueba iterar una lista en su totalidad
- [ ] Se prueba iterar una lista deteniéndose antes de terminar de visitar todos los elementos

4.10. Estilo

- Las descripciones de las pruebas no informan lo que se está realizando en las mismas. A continuación se citaran ciertas pruebas para ilustrar qué es lo que se debería haber hecho:

```
lista_t *lista = lista_crear();
int elemento = 2;
int elemento1 = 3;
lista_insertar(lista,&elemento);
pa2m_afirmar(lista->cantidad == 1,"Se aniadio la cantidad correcta de elementos");
pa2m_afirmar(lista->nodo_inicio->elemento == &elemento,"Se aniadio el elemento 1 correctamente");
lista_insertar(lista,&elemento1);
pa2m_afirmar(lista->cantidad == 2, "Se aniadio el segundo elemento correctamente");
pa2m_afirmar(lista->nodo_inicio->siguiente->elemento == &elemento1,"elemento 3 a la segunda posicion");
pa2m_afirmar(lista->nodo_fin->elemento == &elemento1,"elemento 3 en el nodo final");
...
```

  Como podemos ver, las descripciones carecen de contexto, ya que ninguna indica de que se trata de una lista vacía en un principio, a la cual se le están insertando elementos. Además del problema de la falta de información en las descripciones, no se colocan saltos de líneas para diferenciar las distintas acciones que se realizan dentro de las pruebas. Esto se ve reflejado a lo largo de todas las pruebas y es por esto que se ejemplificará cómo se esperaba que se realicen las mismas:

```
lista_t *lista = lista_crear();
int elemento1 = 1;
int elemento2 = 2;

lista_insertar(lista, &elemento1);

pa2m_afirmar(lista->cantidad == 1,"Lista vacia: al insertarle un elemento la cantidad es igual a uno.");
pa2m_afirmar(lista->nodo_inicio->elemento == &elemento1,"Lista vacia: al insertarle un elemento, el mismo se inserta en la primera posicion de la lista.");

lista_insertar(lista, &elemento2);

pa2m_afirmar(lista->cantidad == 2, "Lista con un elemento: al insertarle un elemento la cantidad es igual a dos");
pa2m_afirmar(lista->nodo_inicio->siguiente->elemento == &elemento2,"Lista con un elemento: al insertar un elemento, el mismo se inserta en la segunda posicion de la lista.");
pa2m_afirmar(lista->nodo_fin->elemento == &elemento2,"Lista con un elemento: al insertar un elemento, el mismo se inserta en la ultima posicion de la lista.");
```

- Los nombres de las funciones realizadas para la modularización de los casos de prueba no poseen un formato correcto, en donde se informe el contexto, la acción realizada y la consecuencia esperada. El formato mencionado anteriormente se puede realizar de la siguiente manera: `dadaUnaSerieDeEventos_CuandoRealizoUnaAccion_SucedeLaConsecuenciaEsperada()`.

  Para que el ejemplo quede más claro se tomará de ejemplo las pruebas para la inserción de elementos dentro de una lista, en donde algunos casos de pruebas podrían haber sido:

  - `dadaUnaListaNula_cuandoQuieroInsertarUnElemento_devuelveNulo()`
  - `dadaUnaListaVacia_cuandoInsertoUnElemento_seInsertaEnLaPrimeraPosicion()`
  - `dadaUnaListaConElementos_cuandoInsertoUnElemento_seInsertaAlFinalDeLaLista()`

5. Sugerencias:

- Las variables locales de una función es recomendable crearlas al comienzo de la misma, luego de haber verificado que todos los parámetros recibido sean válidos, ya que genera mayor claridad y entendimiento en el código.

¡Ojalá te sirvan las correcciones y cualquier cosa no dudes en consultarme!

Nota: 0,75 de un máximo de 2 puntos.