Corrección Lista

Alumno: Zacarias Rojas, Victor Manuel. 107080

Comentarios generales

Es un muy buen tp. Cumple con las buenas practicas de la programación. El informe explica detalladamente lo que es una lista/cola/pila y sus diferentes maneras de implementación. Las pruebas están muy bien aunque le faltaron casos por testear.

Cosas buenas :D

- Código limpio
- Buenos nombres para las pruebas
- Buenos nombres para las variables
- Las pruebas corren correctamente

Cosas malas :c

1. Código repetido a la hora de buscar nodos y/o crear nodos
2. Desapilar en O(n)
3. No se prueban los casos de NULL en las pruebas

Vamos a repasar cada ítem.

---

1 - Código repetido a la hora de buscar nodos y/o crear nodos

Este es un error que se encuentra en varias partes del código. Por ejemplo en la función de lista_insertar tenemos el siguiente código

```
       nodo_auxiliar = calloc(1, sizeof(nodo_t));
        if(!nodo_auxiliar){
            return NULL;
        }
        nodo_auxiliar->elemento = elemento;
```

Y a su vez, en lista_insertar_en_posicion:

```
    nodo_t* nodo_a_insertar = calloc(1, sizeof(nodo_t));
    nodo_t* nodo_auxiliar;

    if(!nodo_a_insertar){
        return NULL;
    }

    nodo_a_insertar->elemento = elemento;
```

Entonces la funcionalidad de crear un nodo y asignarle un elemento, es algo que se repite, lo mejor sería que este encapsulado en una función. La solución posible sería la siguiente:

```
nodo_t* crear_nodo(void* elemento) {
    nodo_t* nodo = calloc(1, sizeof(nodo_t));
    if(!nodo){
        return NULL;
    }
    nodo->elemento = elemento;
    return nodo;
}
```

A la hora de buscar nodos en una posición especifica ocurre lo mismo. En la función lista_insertar_en_posicion, lista_quitar, lista_quitar_en_posicion y lista_elemento_en_posicion, se repite el siguiente algoritmo:

```
    while(i < posicion-1){
        nodo_aux = nodo_aux->siguiente;
        i++;
    }
```

El algoritmo implementado en las funciones, solamente varía la posicion a la cual se quiere acceder. Entonces si tenemos un bloque de código en el cual solamente varia un elemento, es una señal de que se podría refactorizar el bloque de código. Una posible solución podría ser la siguiente:

```
RECURSIVA
nodo_t* buscar_nodo_en_posicion(nodo_t* nodo_actual, size_t posicion){
    if(!nodo_actual)
        return NULL;

    if (posicion == 0)
        return nodo_actual;

    return buscar_nodo_en_posicion(nodo_actual->siguiente, posicion -1);
}
```

Entonces si nosotros aplicamos esta refactorización, el código sera mucho más prolijo y legible.

---

2 - Desapilar en O(n)

En la implementación de la pila podemos ver el siguiente código:

```
pila_t* pila_apilar(pila_t* pila, void* elemento){

    if(!pila){
        return NULL;
    }

    pila_t* pila_apilada = pila;
    lista_t* lista_auxiliar;
    lista_auxiliar = lista_insertar(pila->lista, elemento);
    if(!lista_auxiliar){
        return NULL;
```

```
    }
    pila_apilada->lista = lista_auxiliar;
    return pila_apilada;
}

void* pila_desapilar(pila_t* pila){

    if(!pila){
        return NULL;
    }
    return lista_quitar(pila->lista);
}
```

A la hora de insertar, el código tiene una complejidad de O(1), lo cual esta perfecto. Pero al momento de desapilar, lo hace en O(n), lo cual es un error de implementación. Porque, usando la analogía de una pila de platos (utilizado en el informe), para sacar el ultimo plato de la pila ¿Vamos recorriendo desde el piso de la pila hasta el tope? No, simplemente se desapila el que se encuentra en el tope.

---

3 - No se prueban los casos de NULL en las pruebas

Las pruebas me aseguran que las acciones de la lista/pila/cola funcionan con elementos y lista/pila/cola no nulas, pero ninguna de las acciones de una lista/cola/pila se testea con elementos NULL o lista/pila/cola nula.

¿Que pasa si a lista_insertar le envío una lista nula? ¿Lo inserta o rompe?

¿Si envío un elemento NULL lo agrega? ¿O no?

¿Pasa lo mismo con las pilas y colas? ¿O funcionan diferente?

Es difícil poder abarcar todos los posibles escenarios, por eso esta bueno partir siempre de dos preguntas

> *¿Como se comporta el código con NULLs? ¿Y con elementos validos?*

Y respondiendo esas preguntas con las pruebas, ya estas contemplando muchas situaciones.

Sugerencias

Simplificar el código

Hay secciones del código en el cual hay código repetido que no se repiten en otros lugares (como el error 1). Por ejemplo veamos el siguiente código

```
lista_quitar_de_posicion:

if(lista->cantidad == 1 || posicion >= lista->cantidad){
        elemento_eliminado = lista_quitar(lista);
    }
    else{
        ....
```

```
        if(posicion != 0){

            while(i < posicion-1){

                nodo_aux = nodo_aux->siguiente;
                i++;
            }

            nodo_anterior_al_eliminar = nodo_aux;
            nodo_a_eliminar = nodo_anterior_al_eliminar->siguiente;
            nodo_anterior_al_eliminar->siguiente = nodo_a_eliminar->siguiente;
            lista->cantidad--;
            elemento_eliminado = nodo_a_eliminar->elemento;
            free(nodo_a_eliminar);
        }
        else{

            nodo_a_eliminar = lista->nodo_inicio;
            elemento_eliminado = nodo_a_eliminar->elemento;
            lista->nodo_inicio = nodo_a_eliminar->siguiente;
            lista->cantidad--;
            free(nodo_a_eliminar);
        }
    }

    return elemento_eliminado;
```

Dentro del primer else, se repiten las siguientes líneas:

```
    lista->cantidad--;
    free(nodo_a_eliminar);
```

Entonces una posible manera de poder simplificar esas dos líneas repetidas, podría ser la siguiente:

```
lista_quitar_de_posicion:

if(lista->cantidad == 1 || posicion >= lista->cantidad){
        elemento_eliminado = lista_quitar(lista);
    }
    else{
        ...
        if(posicion != 0){
            while(i < posicion-1){

                nodo_aux = nodo_aux->siguiente;
                i++;
            }
            nodo_anterior_al_eliminar = nodo_aux;
            nodo_a_eliminar = nodo_anterior_al_eliminar->siguiente;
            nodo_anterior_al_eliminar->siguiente = nodo_a_eliminar->siguiente;
            elemento_eliminado = nodo_a_eliminar->elemento;
        }
```

```
        else{
            nodo_a_eliminar = lista->nodo_inicio;
            elemento_eliminado = nodo_a_eliminar->elemento;
            lista->nodo_inicio = nodo_a_eliminar->siguiente;
        }
        lista->cantidad--;
        free(nodo_a_eliminar);
    }
    return elemento_eliminado;
```

Haciendo ese cambio, se arregla el código repetido y queda el código un poco más organizado.

Eliminar los archivos ejecutables

Dentro del zip que se se envío al chanutron, se dejaron archivos ejecutables de "pruebas" "ejemplo". Cuando enviemos el trabajo terminado, eliminemos estos archivos ejecutables.

Nota

1/2