Práctica 2 - Listas

Listas simplemente enlazadas

1. Lea la implementación provista de listas simplemente enlazadas de enteros, y el ejemplo presentado en el archivo `main.c`. Asegúrese comprenderlo.

2. Extienda la implementación anterior con las siguientes funciones:

a) **slist_longitud** que devuelve la longitud de una lista.

b) **slist_concatenar** que devuelve la concatenación de dos listas, modificando la primera.

c) **slist_insertar** que inserta un dato en una lista en una posición arbitraria.

d) **slist_eliminar** que borra de una lista un dato apuntado en una posición arbitraria.

e) **slist_contiene** que determina si un elemento está en una lista dada.

f) **slist_indice** que devuelve la posición de la primera ocurrencia de un elemento si el mismo está en la lista dada, y -1 en caso que no esté.

g) **slist_intersecar** que devuelve una nueva lista con los elementos comunes (independientemente de la posición) de dos listas dadas por parámetro. Las listas originales no se modifican.

h) **slist_intersecar_custom** que trabaja como la anterior pero recibe un parámetro extra que es un puntero a una función de comparación que permite definir la noción de igualdad a ser usada al comparar elementos por igualdad.

i) **slist_ordenar** que ordena una lista de acuerdo al criterio dado por una función de comparación (que usa los mismos valores de retorno que *strcmp()*) pasada por parámetro.

j) **slist_reverso** que obtenga el reverso de una lista.

k) **slist_intercalar** que dadas dos listas, intercale sus elementos en la lista resultante. Por ejemplo, dadas las listas [1, 2, 3, 4] y [5, 6], debe obtener la lista [1, 5, 2, 6, 3, 4].

l) **slist_partir** que divide una lista a la mitad. En caso de longitud impar ($2n + 1$), la primer lista tendrá longitud $n + 1$ y la segunda n. Retorna un puntero al primer elemento de la segunda mitad, siempre que sea no vacía.

Indique cuáles son las operaciones que piensa que más tiempo consumen en ejecutarse. ¿Cuáles de ellas dependen del tamaño de sus argumentos?

3. Considere la siguiente definición de listas simplemente enlazadas de enteros, que mantiene adicionalmente un puntero al último elemento de la lista:

```
typedef struct _SNodo {
  int dato;
  struct _SNodo *sig;
} SNodo;
```

```
typedef struct SList {
  SNodo *primero;
  SNodo *ultimo;
} SList;
```

Reimplemente para este caso las funciones `slist_crear`, `slist_agregar_inicio` y `slist_agregar_final`. ¿Detecta alguna ventaja o desventaja con la nueva definición?

Listas doblemente enlazadas

4. Implemente listas doblemente enlazadas de enteros y sus funciones básicas usando la siguiente definición:

```
typedef struct _DNodo {
  int dato;
  struct _DNodo* sig;
  struct _DNodo* ant;
} DNodo;

typedef struct {
  DNodo* primero;
  DNodo* ultimo;
} DList;
```

Llame a los ficheros como `dlist.h` y `dlist.c`. En ambos casos implemente `dlist_recorrer` de manera que se pueda elegir si se avanza o retrocede en el recorrido, utilizando el tipo:

```
typedef enum {
  DLIST_RECORRIDO_HACIA_ADELANTE,
  DLIST_RECORRIDO_HACIA_ATRAS
} DListOrdenDeRecorrido;
```

5. De una implementación de listas doblemente enlazadas de enteros circulares. ¿Es necesario mantener un puntero al último elemento de la lista? Llame a los ficheros como `cdlist.h` y `cdlist.c`. Implemente `cdlist_recorrer` de manera que solo ejecute una pasada sobre cada elemento.

Listas generales

6. Las listas enlazadas presentadas hasta este punto guardan específicamente datos de tipo `int`. En consecuencia, para guardar otros tipos de datos es necesario modificar varias partes del código, haciéndolo poco reutilizable. Esto puede evitarse utilizando el tipo de dato general `void*` y abstrayendo del código aquellas partes que hacían uso del tipo de dato específico.
Lea la implementación provista de listas simplemente enlazadas generales y el ejemplo presentado en el archivo `main.c`. Compárelo con la implementación de listas simplemente enlazadas de enteros ¿qué diferencias encuentra?

7. Filtrando los elementos de una lista.

a) Agregue a la implementación anterior una función

`GList glist_filtrar(GList lista, FuncionCopia c, Predicado p);`

que dada una lista general, retorne una nueva lista con los elementos que cumplen con el predicado. El tipo de la función `p` está declarado como `typedef int (*Predicado) (void *dato)`, y retorna 1 cuando el dato cumple con el predicado y 0 en caso contrario.

b) Utilice esta función en `main.c` para filtrar todos los contactos mayores a 60 años.

8. Considere la siguiente definición de listas generales ordenadas

```
typedef GList SGList;
```

e implemente las siguientes funciones:

a) `SGList sglist_crear()` que retorna una lista ordenada vacía.

b) `void sglist_destruir(SGList, FuncionDestructora)` que destruye una lista ordenada.

c) `int sglist_vacia(SGList)` que determina si una lista ordenada es vacía.

d) `sglist_recorrer(GList, FuncionVisitante)` que aplica la función visitante a cada elemento de la lista ordenada.

e) `SGList sglist_insertar(SGList, void *, FuncionCopia, FuncionComparadora)` que inserta un nuevo dato en la lista ordenada. La función de comparación es la que determina el criterio de ordenación, su tipo está declarado como `typedef int (*FuncionComparadora)(void *, void *)`, y retorna un entero negativo si el primer argumento es menor que el segundo, 0 si son iguales, y un entero positivo en caso contrario.

f) `int sglist_buscar(GList, void *, FuncionComparadora)` que busca un dato en la lista ordenada, retornando 1 si lo encuentra y 0 en caso contrario (aprovechar que la lista está ordenada para hacer esta búsqueda más eficiente).

g) `SGList sglist_arr(void **, int, FuncionCopia, FuncionComparadora)` que construye una lista ordenada a partir de un arreglo de elementos y su longitud.