1. Reimplemente el contenedor Stack_t de la práctica anterior utilizando una lista simplemente ligada en lugar de un vector para mantener los elementos. No modifique la interfaz externa de uso. Verifique que la función StackFilter de la práctica anterior sigue funcionando sin modificaciones.

2. Se desea implementar un servicio que permita filtrar listas. La función a implementar debe, **sin modificar la lista original**, crear una nueva lista que resulta de aplicar un filtro sobre la primera. Para poder utilizar distintos criterios de filtrado (por ejemplo: que filtre los nodos con contenido par, que filtre los nodos con contenido mayor a un valor dado, que filtre los nodos que no se encuentren en un determinado rango, etc.) la función debe recibir como argumentos, además de la lista a filtrar, el criterio de filtrado (como puntero a una función) y los parámetros del filtro. Para los filtros que necesitan parámetros (por ejemplo el valor umbral para los filtros por umbral, o el rango válido para los filtros por rango, etc.) los parámetros se pasan como del tipo void *, haciendo que cada función filtro en particular interprete esa dirección como un puntero a una estructura de parámetros propia del tipo de filtro.

   Utilizando la definición siguiente:

```
// Puntero a función que define un criterio de filtrado sobre el contenido
// de un nodo, si el 'valor' debe estar contenido en la lista filtrada la
// función debe retornar 1, sino 0
typedef int (*FilterFun) (int valor, void *params);
```

Se solicita:

   a. Implementar la función que filtra una lista, aplicando un criterio de filtrado pasado a la propia función:

```
List_t FilterList( List_t lst, FilterFun filtro, void *params);
```

   b. Implementar filtro **`pasaBajo`** (solo pasan los valores menores a un cierto umbral)
   c. Implementar filtro **`pasaPares`** (solo pasan los valores pares)
   d. Implementar filtro **`pasaEnRango`** (solo pasan los valores en un dado rango)
   e. Probar todo utilizando el **`main`** dado.

3. ***Diccionarios Binarios:*** Dado un diccionario binario de strings (El contenido de un nodo es del tipo **const char ***) se solicita realizar las siguientes operaciones:

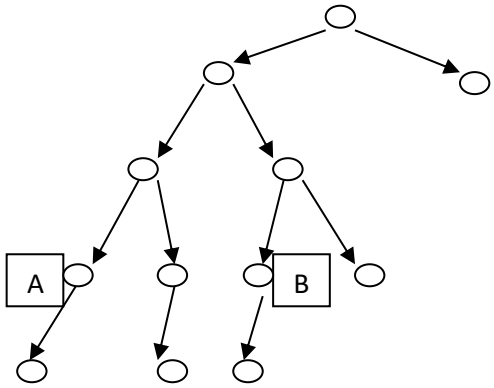
   a. **Recorrido por niveles**: El recorrido por nivel de los nodos de un árbol funciona de la siguiente manera: primero recorre la raíz, luego todos los nodos a profundidad uno, todos los nodos a profundidad dos, etc. Los nodos de la misma profundidad se recorren de izquierda a derecha. Se solicita implementar la función para listar los nodos de un árbol binario por nivel:

```
void PrintArbolPorNivel(Arbol_t *pArbol);
```

   b. **Distancia entre dos nodos**: Suponiendo que la distancia entre un nodo y un nodo hijo de este es 1, realizar una función y las auxiliares necesarias, que permitan calcular la distancia entre 2 nodos en un diccionario binario. Prototipo de la función requerida:

```
int DistanciaArbol(Arbol_t *pArbol, const char *A,
                   const char *B);
```

En el ejemplo, la distancia entre A y B es 4



4. Cuando se desea remover un elemento de un diccionario binario, el elemento a remover, si está presente, necesita ser reemplazado por otro nodo del árbol de tal forma que el árbol resultante siga siendo un diccionario. Para encontrar al elemento reemplazante se podría optar por dos nodos candidatos: a) el mayor de los nodos del subárbol izquierdo (mayor de los menores), o b) el menor de los nodos del subárbol derecho (menor de los mayores), en cualquiera de los casos se produciría un proceso recursivo, porque para usar ese reemplazo, hay que removerlo del subárbol que corresponda. Se solicita definir el algoritmo e implementarlo en la siguiente función:

```
Tree_t TreeRemove(Tree_t tree, int e);
```