TP N° 11: Punteros a función y T.A.D.

La siguiente guía cubre los contenidos vistos en las clases teóricas:
18. Punteros a función
19. TAD

Ejercicio 1

Escribir una función recursiva **`removeIf`**, que reciba una lista de enteros (de tipo TList) y un puntero a función que reciba un entero y retorne un entero; dicha función se debe aplicar a cada elemento de la lista y eliminar aquellos que cumplan con la condición, esto es, que teniendo como parámetro el valor del nodo, lo elimine si retorna verdadero, y lo conserve si retorna cero. Por ejemplo, si la lista fuera los enteros del 1 al 10, y la función recibida retorna 1 (uno) si el número es par, en la lista sólo deben quedar los impares.

Ejercicio 2

Escribir un programa para hallar las raíces de una función matemática en un intervalo cerrado, recorriéndolo de forma tal que el intervalo quede dividido en 100000 (cien mil) particiones o subintervalos. Ejemplo: si el intervalo es [1, 50000] deberá evaluar la función en los puntos 1, 1.5, 2, etc. (también se tomarán como válidos los puntos 1, 1.49999, etc.).

El programa ofrecerá un menú de funciones matemáticas y deberá solicitar los extremos del intervalo, imprimiendo los resultados en la salida estándar. Todas las funciones reciben y devuelven un valor real.

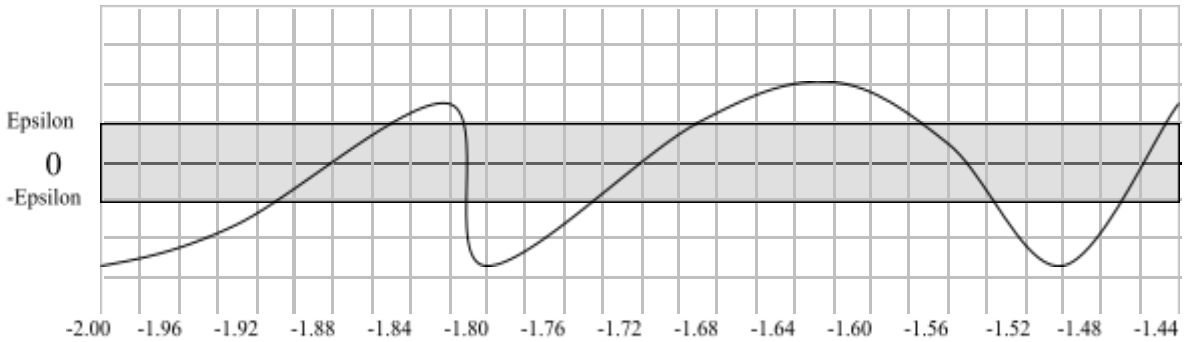
La función que realice la búsqueda de las raíces debe recibir como parámetros una estructura que represente al intervalo y una función a la cual se le quieren hallar las raíces, regresando en su nombre una estructura que empaquete un arreglo con aquellas particiones donde haya raíces y la dimensión de dicho arreglo.

Para detectar una raíz se deben considerar dos casos:

- Que la función cambie de signo entre dos puntos: En ese caso se agrega al arreglo una partición con ambos puntos.
- Que la función se haga cero en un punto (considerando un error de EPSILON): En ese caso la partición que se agrega al arreglo de resultados está formada por el punto anterior al que se detectó como raíz y el próximo que no lo sea.

Ejemplo:

Considerando un intervalo cuyo paso es de 0.02, se grafica un fragmento del mismo indicando qué raíces se encuentran:



Se detectarán raíces en :

- **-1.90**, porque el valor absoluto de la imagen en ese punto es menor que EPSILON. Pero los dos puntos siguientes (-1.88 y -1.86) también se consideran raíces por el mismo motivo. Por lo tanto en este caso la partición que se agrega con resultado toma los valores **-1.92** (por ser el anterior a la primera raíz) y **-1.84** (por ser el posterior a la última raíz).
- luego se detecta un cambio de signo de la función entre los puntos **-1.82** y **-1.80**, por lo que se agrega dicha partición.
- **-1.74** es raíz por comparación con EPSILON, al igual que los dos puntos siguientes (-1.72, -1.70) por lo que se agrega la partición **-1.76**, **-1.68**.
- **-1.56** es raíz por comparación con EPSILON, al igual que el punto siguiente -1.54, por lo que se agrega la partición **-1.58, -1.52**.
- **-1.46** es raíz por comparación con EPSILON, y el punto siguiente ya no lo es , por lo que se agrega la partición **-1.48, -1.44**.

Por lo tanto nuestra función debería retornar una estructura conteniendo los siguientes valores:

{ 5, { { -1.92, -1.84}, { -1.82, -1.80}, { -1.76, -1.68}, { -1.58, -1.52}, { -1.48, -1.44} } }

Ejercicio 3

¿Qué indican las siguientes expresiones?

```
a) int *(*p) (int (*a)[ ]);
b) int *p[10];
c) int (*p)[10];
d) int *p(void);
e) p (char *a);
f) int *p(char *a);
g) int (*p) (char *a);
h) int (*p(char *a))[10];
i) int p(char (*a)[ ]);
j) int p(char *a[ ]);
k) int *p(char a[ ]);
l) int *p(char (*a)[ ]);
m) int *p(char *a[ ]);
n) int (*p)(char(*a)[ ]);
o) int *(*p)(char *a[ ]);
p) int (*p[10])(void);
q) int (*p[10])(char a);
r) int *(*p[10])(char a);
s) int *(*p[10])(char *a);
t) int (*p) (char *a[ ]);
```

Ejercicio 4

Escribir las siguientes declaraciones complejas:

a) **p** es un arreglo de punteros a funciones que no reciben parámetros y devuelven punteros a double.

b) **p** es una función que recibe un puntero a char y devuelve un puntero a un arreglo de 3x5 enteros.

c) **p** es un puntero a un arreglo de N punteros a función que reciben un entero y retornan un puntero a entero.

Ejercicio 5

Crear un T.A.D. que maneje números complejos. Debe ofrecer como mínimo las siguientes funcionalidades:

- obtener la parte real
- obtener la parte imaginaria
- sumar dos números complejos (retorna un nuevo número complejo)

Ejercicio 6

Agregar al TAD de listas genéricas provisto por la Cátedra una función que reciba un número entero i y devuelve el i-ésimo elemento, donde el primer elemento tiene el índice cero.

Ejercicio 7

Agregar al TAD de listas una función map, que reciba un puntero a función que permita modificar cada elemento de la misma. Ejemplo:

```
   int doble(int n) {
      return 2*n;
   )

   int main(void) {
       listADT myList = newList();  // Depende de la
implementación
       // habrá que pasarle la función
       add(myList, 3);
       add(myList, 4);
       add(myList, 2);
       map(myList, doble);

       ...

       return 0;
   }
```

Luego de invocar a map, la lista queda con los elementos 6, 8 y 4

Ejercicio 8

Hacer un T.A.D. para conjuntos: elementos no repetidos, sin orden. Debe tener al menos las funciones de agregar o remover un elemento, unión, intersección y resta de conjuntos.

En la implementación usar listADT: dentro de setCDT tener un campo que sea un listADT.

Ejercicio 9

a) Si bien definir la estructura en el archivo fuente .c oculta la implementación del TAD, igualmente el usuario (si conoce cómo es realmente la estructura) puede modificar los datos de la misma desde el front-end. Explicar cómo.

b) Explicar cómo puede hacer el desarrollador de un TAD para evitar que el usuario pueda modificar la estructura. Debe ser un método 100% efectivo. Pista: la función new… no debe retornar un puntero.

Ejercicio 10

El siguiente es un TAD de "*bag*". Un bag o bolsa es un conjunto de elementos sin orden pero donde cada elemento puede aparecer más de una vez.

bagADT.h (en los test y soluciones lo llamamos tp11_ej10.h)

```
typedef struct bagCDT * bagADT;

typedef ... elemType;    // Tipo de elemento a insertar

/**
** Retorna 0 si los elementos son iguales
*/
static int compare (elemType e1, elemType e2) {
   ...
}

/* Retorna un nuevo bag de elementos genéricos. Al inicio está vacío */
bagADT newBag();

/* Inserta un elemento. Retorna cuántas veces está
** elem en el conjunto luego de haberlo insertado (p.e. si es la
** primera inserción retorna 1).
*/
unsigned int add(bagADT bag, elemType elem);

/* Retorna cuántas veces aparece el elemento en el bag */
unsigned int count(const bagADT bag, elemType elem);

/* Retorna la cantidad de elementos distintos que hay en el bag */
unsigned int size(const bagADT bag);

/* Remueve una aparición de un elemento. Retorna cuántas veces está
** elem en el conjunto luego de haberlo borrado
*/
unsigned int delete(bagADT bag, elemType elem);

/* Retorna el elemento que aparece más veces. Si hay más de uno
** con esa condición, retorna cualquiera de los dos.
** Precondición: el bag no debe estar vacío. En caso de estar vacío
** aborta la ejecución
*/
elemType mostPopular(bagADT bag);
```

a) <u>Implementar el TAD completo</u>

b) Si quiere poder recorrer los elementos, pero de forma de obtener únicamente los que tienen menos de N repeticiones, ¿cuál de las siguientes opciones le parece más adecuada agregar en el .h?

i) Opción 1: una función que reciba N y retorne un vector con todos los elementos, con un parámetro de salida indicando la dimensión.

```
elemType * filterN(bagADT bag, unsigned int n, unsigned int * dim );
```

ii) Opción 2: agregar una función "Preparar elementos" que reciba el N, luego un iterador para recorrer (toBegin, hasNext, next)

```
// Esta función se encarga de ordenar internamente y dejar
// listos para recorrer los elementos que tienen menos de n
// repeticiones.
void filterN(bagADT bag, unsigned int n);
void toBeginN(bagADT bag);
int hasNextN(const bagADT bag);
elemType nextN(bagADT bag);
```

iii) Opción 3: una función toBegin que reciba N y prepare los elementos, luego el usuario debe invocar a hasNext y next

```
void toBeginN(bagADT bag, unsigned int n);
int hasNextN(const bagADT bag);
elemType nextN(bagADT bag);
```

iv) Opción 4: una función que retorne un vector con todos los elementos, indicando para cada uno la cantidad de veces que aparece. El que invoca a la función podrá ordenarlos, descartar lo que aparezcan menos o más veces de las deseadas. De esta forma podrá hacer todos los filtros que desee

```
struct elemCount {
   elemType elem;
   size_t count;
};

struct elemCount * elems(const bagADT bag);
```

Ejercicio 11

Dado el siguiente contrato para un TAD de vectores de elementos genéricos, donde un elemento puede ser de cualquier tipo (entero, double, estructura, string, etc.)

```
typedef struct vectorCDT * vectorADT;

typedef .. elemType;    // Tipo de element a insertar

/* Crea un nuevo vector dinámico de elementos genéricos
** Inicialmente el vector está vacío
** Cada elemento a insertar será de tipo elemType
*/
vectorADT newVector( ¿? );

/* Libera todos los recursos reservados por el TAD */
void freeVector(vectorADT v);

/* Almacena los elementos de elems a partir de la posición index,
** donde elems es un vector de dim elementos.
** En caso de ser necesario agranda el vector.
** El resto de los elementos del vector no se modifican y permanecen
** en la misma posición.
** Si se recibe NULL o no se pudo insertar retorna cero.
** Si alguna posición está ocupada, la deja como estaba.
** Retorna cuántos elementos pudo almacenar.
** Ejemplo:
**   Si v tiene ocupadas las posiciones 1,3 y 6
**   Se invoca con index=2, dim=5
**   El vector actual quedará con los mismos elementos en las
**   posiciones 1, 3 y 6
**   Pero además v[2]=elems[0], v[4]=elems[2], v[5]=elems[3]
**   y la función retorna 3
*/
int put(vectorADT v, elemType * elems, size_t dim, size_t index);


/* Retorna el índice en el cual está insertado el elemento, o -1 si no lo
encuentra */
int getIdx(vectorADT v, elemType elem);


/* Elimina el elemento en la posición index. Si index está fuera del
vector no hace nada */
void deleteElement(vectorADT v, size_t index);


/* Retorna cuántos elementos hay insertados en el vector */
int elementCount(vectorADT v);
```

Donde ¿? en una lista de parámetros indica que usted (programador) debe definir cuáles son los parámetros necesarios para esa función, en base a las características del TAD.
Se pide

a) Completar la definición de `struct vectorCDT`
b) Escribir la función newVector
c) Escribir la función put

d) Escribir la función getIdx

DEFINIR TODOS LOS TIPOS DE DATOS QUE SE VAYAN A USAR EN LAS FUNCIONES PEDIDAS

SE ASUME QUE EL VECTOR TENDRÁ UN BAJO PORCENTAJE DE POSICIONES LIBRES

Ejercicio 12 (tomado en parcial, ejercicio optativo)

Se desea almacenar nombres de personas, y para cada persona los nombres de sus "allegados".

Se asegura para todos los nombres (de las personas y sus allegados) que se cumple:

1. se enviarán en minúsculas (no es necesario validarlo)
2. estarán compuestos por valores ASCII menores a 128 (US-ASCII)
3. no serán extensos (a lo sumo 20 caracteres)

El objetivo es poder mantener la base actualizada de personas y sus allegados, para lo cual se definió el siguiente contrato para manejarlo con un TAD.

Tener en cuenta que al almacenar nombres se almacena una COPIA de los mismos (ver ejemplos de uso antes de comenzar a programar el TAD)

```
typedef struct socialCDT * socialADT;

/* Crea un nuevo TAD vacío */
socialADT newSocial();

/* Libera todos los recursos reservados por el TAD */
void freeSocial(socialADT soc);

/* Almacena una nueva persona. Si la persona existe, no hace nada
** Guarda una copia del nombre, no simplemente el puntero */
void addPerson(socialADT soc, const char * name);

/* Si existe una persona con ese nombre, agrega la nueva relación
** Si la persona no existe, no hace nada
** Si related ya estaba relacionado, lo agrega repetido
** Almacena una copia de related, no simplemente el puntero
**
*/
void addRelated(socialADT soc, const char * name, const char * related);

/* Retorna una copia de los nombres relacionados con una persona
** en orden alfabético.
** Para marcar el final, después del último nombre se coloca NULL
** Si la persona no existe, retorna un vector que sólo tiene a NULL como
** elemento
*/
char ** related(const socialADT soc, const char * person);

/* Retorna una copia de los nombres de las personas en orden alfabético.
** Para marcar el final, después del último nombre se coloca NULL
** Si no hay personas, retorna un vector que sólo tiene a NULL como
** elemento
*/
char ** persons(const socialADT soc);
```

a) Implementar el TAD completo.

Ejemplo correcto de uso.

```
int main(void) {
   socialADT soc = newSocial();
   char ** rel;
   rel = persons(soc);  // rel = {NULL}
   free(rel);
   rel = related(soc, "carlitos");  // rel = {NULL};
   free(rel);

   char aux[30] = "juan";
   addPerson(soc, aux); // soc contiene a "juan"
   strcpy(aux,"luisa");
   addPerson(soc, aux); // soc contiene a "juan" y "luisa"

   strcpy(aux,"ana");
   addRelated(soc, "juan", "pedro");
   addRelated(soc, "juan", aux);
   addRelated(soc, "juan", "juana");

   char ** juanFriends = related(soc, "juan");
   // juanFriends es {"ana", "juana", "pedro", NULL};
   for(int i=0; juanFriends[i] != NULL; i++)
       free(juanFriends[i]);
   free(juanFriends);

   addPerson(soc, "andres");
   addPerson(soc, "analia");
   char **p = persons(soc);
   // p={"analia","andres","juan","luisa",NULL}
   for(int i=0; p[i] != NULL; i++)
       free(p[i]);
   free(p);

   addRelated(soc, "juan", "john");
   // Ahora los amigos de juan son "ana", "john", "juana" y "pedro"
   freeSocial(soc);

   return 0;
}
```

b) Para automatizar la carga de datos y que no sea manual, los mismos serán levantados de un archivo de texto, donde cada línea tenga al principio el nombre de una persona y a continuación, separados por punto y coma, los nombres de sus allegados. Indicar cuál de los siguientes cambios propuestos al contrato es más adecuado:

i) Agregar una función que reciba el nombre del archivo, lo abra y lo recorra línea a línea levantando los datos. La función devuelve cuántas personas agregó o si hubo un error

ii) Similar a la anterior pero que en vez del nombre del archivo reciba el archivo ya abierto

iii) Agregar una función que reciba un string con la línea. El front debería abrir el archivo e invocar a esta función con cada línea que levanta del mismo

iv) No haría ningún cambio en el contrato

Ejercicio 13 (ejercicio tomado en parcial)

Se desea implementar una colección que permita guardar elementos genéricos sin repeticiones. La colección se llamará **`rankingADT`**, ya que tiene la particularidad que tiene que servir para acceder fácilmente a los elementos que están al tope del ranking. Los elementos van "escalando posiciones" en el ranking a medida que son consultados.
Si la colección tiene N elementos, se dice que el que está en el tope del ranking está en el puesto 1. El que está último en el ranking está en el puesto N del ranking.
Implementar el .c completo

Ver archivo de encabezado en
rankingADT.h

Ver ejemplo de uso en ranking DT_test.c , asumiendo

```
typedef char * elemType;
```

Ejercicio 14 (ejercicio tomado en parcial)

Se desea implementar un TAD para **listas de elementos no repetidos**, **que permita recorrerla con dos criterios: en forma ascendente o por el orden de inserción de los elementos**.

tp11_ej14.h

```
typedef struct listCDT * listADT;

typedef ___ elemType;    // Tipo de elemento a insertar, por defecto int


/* Retorna una lista vacía.
** compare retorna 0 si los elementos son iguales, negativo si e1 es "menor" que e2 y
** positivo si e1 es "mayor" que e2
*/
listADT newList(int (*compare) (elemType e1, elemType e2));

/* Agrega un elemento. Si ya estaba no lo agrega */
void add(listADT list, elemType elem);

/* Elimina un elemento. */
void remove(listADT list, elemType elem);

/* Resetea el iterador que recorre la lista en el orden de inserción */
void toBegin(listADT list);

/* Retorna 1 si hay un elemento siguiente en el iterador que
** recorre la lista en el orden de inserción. Sino retorna 0
*/
int hasNext(listADT list);

/* Retorna el elemento siguiente del iterador que recorre la lista
** en el orden de inserción.
** Si no hay un elemento siguiente o no se invocó a toBegin aborta la ejecución.
```

```
*/
elemType next(listADT list);

/* Resetea el iterador que recorre la lista en forma ascendente */
void toBeginAsc(listADT list);

/* Retorna 1 si hay un elemento siguiente en el iterador que
** recorre la lista en forma ascendente. Sino retorna 0
*/
int hasNextAsc(listADT list);

/* Retorna el elemento siguiente del iterador que recorre la lista en forma ascendente.
** Si no hay un elemento siguiente o no se invocó a toBeginAsc aborta la ejecución.
*/
elemType nextAsc(listADT list);

/* Libera la memoria reservada por la lista */
void freeList(listADT list);
```

Implementar todas las funciones del TAD, excepto la función `remove`, teniendo en cuenta que las funciones `toBeginAsc`, `hasNextAsc`, `nextAsc`, `toBegin`, `hasNext` y `next` deben ser O(1), es decir, ninguna de ellas debe recorrer la lista.

Ejemplo de uso, definiendo elemType como int

```
#include "tp11_ej14.h"

int compInt(int e1, int e2) {
   return e1 - e2;
}

int
main(void) {
   listADT c = newList(compInt); // una lista, en este caso de int
   add(c, 3);   add(c, 1);   add(c, 5);   add(c, 2);
   toBegin(c);          // iterador por orden de inserción
   int n = next(c);     // n = 3
   n = next(c);         // n = 1
   toBeginAsc(c);       // iterador por orden ascendente
   n = nextAsc(c);      // n = 1
   n = next(c);         // n = 5
   n = next(c);         // n = 2
   hasNext(c);          // retorna 0 ( falso )
   n = nextAsc(c);      // n = 2
   hasNextAsc(c);       // retorna 1 ( true )
   n = nextAsc(c);      // n = 3
   return 0;
}
```

Ejercicio 15 (ejercicio tomado en parcial)

Se desea guardar una colección de elementos **no repetidos**, en la cual los elementos más "populares" (los que más se consultan) estén al principio de la colección. De esta

forma, será más rápido acceder a los elementos que más veces se consulten. Para ello se definió que el conjunto de datos opere de la siguiente forma:

- Cuando se inserta un elemento (no repetido) se lo inserta **al final**
- Cuando se consulta un elemento (con la función `get`) el mismo es enviado **al principio** de la colección
  El contrato con el TAD es el siguiente:

moveToFrontADT.h

```
typedef struct moveToFrontCDT * moveToFrontADT;

typedef ... elemType;   // Tipo de elemento a insertar

/* Retorna un nuevo conjunto de elementos genéricos. Al inicio está vacío
** La función compare retorna 0 si los elementos son iguales, negativo si e1 es
** "menor" que e2 y positivo si e1 es "mayor" que e2
*/
moveToFrontADT newMoveToFront(int (*compare) (elemType e1, elemType e2));

/* Libera todos los recursos del TAD */
void freeMoveToFront(moveToFrontADT m);

/* Inserta un elemento si no está. Lo inserta al final.
** Retorna 1 si lo agregó, 0 si no.
*/
unsigned int add(moveToFrontADT moveToFront, elemType elem);

/* Retorna la cantidad de elementos que hay en la colección */
unsigned int size(moveToFrontADT moveToFront);

/* Se ubica al principio del conjunto, para poder iterar sobre el mismo */
void toBegin(moveToFrontADT moveToFront);

/* Retorna 1 si hay un elemento siguiente en el iterador, 0 si no */
int hasNext(moveToFrontADT moveToFront);

/* Retorna el siguiente elemento. Si no hay siguiente elemento, aborta */
elemType next(moveToFrontADT moveToFront);

/* Retorna una copia del elemento. Si no existe retorna NULL.
** Para saber si el elemento está, usa la función compare.
** Si el elemento estaba lo ubica al principio.
*/
elemType * get(moveToFrontADT moveToFront, elemType elem);
```

Implementar el TAD completo (el archivo `moveToFrontCDT.c`).

Ejemplo de uso, considerando los siguientes cambios en `moveToFrontADT.h`

```
typedef struct {
    int code;
    char name[20];
```

```
} elemType;

static int compare (elemType e1, elemType e2) {
      return e1.code - e2.code;
}
```

```
#include "moveToFrontADT.h"

int
main(void) {
      moveToFrontADT p = newMoveToFront();
      elemType aux = {1, "uno"};
      add(p, aux);   // retorna 1
      strcpy(aux.name, "dos");
      add(p, aux);   // retorna 0
      aux.code = 2;
      add(p, aux);   // retorna 1
      aux.code = 3;
      strcpy(aux.name, "tres");
      add(p, aux);   // retorna 1
      aux.code = 4;
      strcpy(aux.name, "cuatro");
      add(p, aux);   // retorna 1
      toBegin(p);
      while (hasNext(p)) {
         aux = next(p);
         printf("%d %s ", aux.code, aux.name);
      }
      putchar('\n');

      aux.code = 5;
      elemType * q = get(p, aux);   // retorna NULL

      aux.code = 3;
      q = get(p, aux);
      printf("%d %s\n", q->code, q->name);
      free(q);

      toBegin(p);
      while (hasNext(p)) {
         aux = next(p);
         printf("%d %s ", aux.code, aux.name);
      }
      putchar('\n');
      freeMoveToFront(p);
      return 0;
}
```

Al ejecutar el programa la salida será:

```
1 uno 2 dos 3 tres 4 cuatro
3 tres
3 tres 1 uno 2 dos 4 cuatro
```

Ejercicio 16 (ejercicio tomado en parcial)

Se desea crear un TAD que dé soporte para **almacenar y recuperar frases**, donde **cada frase tiene asociada una clave numérica** (un valor entero positivo). **Las claves son únicas** (no puede haber dos frases con la misma clave, aunque sí podría pasar que dos claves tengan la misma frase).
Para ello se crea el siguiente contrato, y se cuenta además con un programa de prueba (leer completamente ambos antes de implementar el TAD).
No hay un límite previsto para la longitud de cada frase, pueden ser unos pocos o miles de caracteres.
Se espera que casi todas las claves estén usadas.
Ver archivo de encabezado en
phrasesADT.h

El objetivo es que todas las funciones sean lo más eficientes posible

Ejemplo de uso. En el ejemplo se usan frases cortas, pero en un uso normal se espera que sean mucho más extensas. Ver tp11_ej16_test.c

Ejercicio 17 (recuperatorio 2do cuatrimestre 2023)

Algunos parques (por ejemplo Central Park) **contabilizan la cantidad de ardillas** que hay en cada área del parque. El parque se divide en **áreas de N x N metros**, asumiendo que cada parque tiene una forma rectangular. Cada área se identifica por la coordenada de su esquina superior izquierda, siendo (0,0) el bloque que comprende un área de [0, N) metros en vertical y de [0, N) metros en horizontal (ver descripción en programa de prueba).

Para ello se definió la siguiente interfaz:

```
typedef struct squirrelCensusCDT * squirrelCensusADT;

/**
* Reserva los recursos para el conteo de ardillas en un parque agrupando las
* cantidades por bloques de tamaño blockSizeMeters metros x blockSizeMeters
* metros desde el extremo superior izquierdo del parque
* Si blockSizeMeters es igual a 0 retorna NULL
*/
squirrelCensusADT newSquirrelCensus(size_t blockSizeMeters);

/**
* Registra una ardilla en el bloque correspondiente al punto (yDistance,
* xDistance) donde
* - yDistance es la distancia vertical en metros del extremo superior
*   izquierdo del parque
* - xDistance es la distancia horizontal en metros del extremo superior
*   izquierdo del parque
* Retorna cuántas ardillas fueron registradas en ese mismo bloque
*/
size_t countSquirrel(squirrelCensusADT squirrelAdt, size_t yDistance, size_t
xDistance);

/**
* Retorna cuántas ardillas fueron registradas en el bloque correspondiente
* al punto (yDist, xDist)
*/
size_t squirrelsInBlock(const squirrelCensusADT squirrelAdt, size_t yDist, size_t
xDist);


/**
* Libera los recursos utilizados para el conteo de ardillas en un parque
*/
void freeSquirrelCensus(squirrelCensusADT squirrelAdt);
```

Se pide:

- **Implementar <u>todas las estructuras</u> necesarias**, de forma tal que la función **squirrelsInBlock** <u>pueda ser implementada de la forma más eficiente posible</u>
- **Implementar <u>todas las funciones</u>**

Ejercicios adicionales

Estos ejercicios son de alto nivel, están dirigidos a los que quieren superarse e ir más allá de los alcances de la materia. El nivel de estos ejercicios supera a los ejercicios de parcial.

Ejercicio 18

Un árbol binario de búsqueda (BST por sus siglas en inglés) es un árbol binario en donde para cada nodo se cumple que todos los nodos de su subárbol izquierdo son menores que él, y todos los nodos de su subárbol derecho son mayores que él. No hay elementos repetidos.

Implementar el ADT para BST, que se encuentra definido en forma parcial (faltan algunas funcionalidades) en el archivo bstADT.h

Ejercicio 19

Una Skip-list es una lista donde cada nodo, en vez de tener una referencia al nodo siguiente, tiene un vector de referencias, donde el elemento 0 contiene una referencia al siguiente nodo, pero los elementos siguientes tienen referencias a nodos que están más "lejanos". Para determinar el tamaño de ese vector de referencias se basa en la probabilidad. El resultado es una lista que empíricamente tiene un orden logarítmico para todas las funciones.

La altura de una skip-list está dada por la mayor dimensión del vector de referencias.

Crear un TAD que -dada una altura máxima- implemente una skip-list que permita mantener elementos ordenados, aceptando repetidos, y que también permita saber si un elemento está o no en la lista, y recorrerla con un iterador.

Para mayor detalles sobre cómo se debe implementar ver https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/skiplists.pdf