EJERCICIO 1

Variante 1

1. Ordenar ascendente las siguientes complejidades:

   a. $T(n)= \log_2 2^n$

   b. $T(n)= n^2 * 2^{3 *(\log_2 2^n)}$

   c. $T(n)= 4T(n/2) + n^2$

2. Determinar la complejidad computacional del siguiente fragmento de código:

```
int k, j;
for (k = n * n; k >= 1; k--)
    for (j = k; j >= 1; j = j/2){
        fragmento de ejecución constante
     }
```

Variante 2

1. Ordenar ascendente las siguientes complejidades:

   a. $T(n)= \log_2 2^n$

   b. $T(n)= 2^{3*(\log_2 n^2)}$

   c. $T(n)= 4T(n/2) + n$

2. Determinar la complejidad computacional del siguiente fragmento de código:

```
int j, k;
for (k = 1; k <= n; ++k)
    for (j = 1; j <= k; j = 2 * j){
        fragmento de ejecución constante
     }
```

Variante 3

1. Ordenar ascendente las siguientes complejidades:

   a. $T(n)= \log_2 2^n$

   b. $T(n)= 2 * (\log_2 2^n)$

   c. $T(n)= 2T(n/2) + n^2$

2. Calcular las complejidades de la Búsqueda Binaria en su versión recursiva vs. su versión iterativa, debe justificarse formalmente.

EJERCICIO 2

Parte A

Explique brevemente por qué con **falsas** las siguientes afirmaciones:

Variante 1

1. El contrato nos permite abstaernos de la implementacion, de esta forma, no importa qué le mandemos a las funciones.
2. No es una buena opción implementar el TDA Cola con un vector estático.
3. La complejidad de destruir una lista es O(n).

Variante 2

1. El contrato nos permite saber como está implementado el TDA.
2. Un TDA debe contener como mínimo 1 estructura.
3. La complejidad de insertar en una cola es O(n).

Variante 3

1. Si a nuestro TDA Lista le agregamos un puntero al último elemento, la complejidad de borrar del final será O(1).
2. Un TDA es un conjunto de funciones que no necesariamente están relacionadas.
3. La complejidad de insertar en una pila es O(n).

Variante 4

1. Cualquier conjunto de funciones y estructuras forman un TDA.
2. La frase *Mil Ques, Un Como* denota que puede haber mil cosas para hacer, pero solo una forma de implementarlo.
3. Agregar punteros a las estructuras de mi TDA, disminuye la complejidad algorítmica.

Parte B

El *Mundo Pokemon*, inserta a los entrenadores pokemon en grandes aventuras, el sueño de cualquier entrenador es *Atraparlos a Todos!*
Para cumplirlo, cada uno recibe al iniciarse un pokemon que se convertirá en su gran amigo, un set de 6 pokebolas y una pokedex.
En la pokedex, se van registrando todas las nuevas especies vistas, algo así como una encicloedia. Éste no es el único uso que tiene la pokedex, ya que también tiene la funcionalidad **mapa**, que ayuda al entrenador a moverse entre ciudades.
Pucci, hace poco recibió a su pokemon inicial (un *Magikarp* hermoso), el set de pokebolas y la pokedex para iniciar su camino de *Entrenador Pokemon*, sin embargo, entre la ansiedad por empezar a recorrer el mundo y la obligación de lavarse las manos *#pandemia*, no se dió cuenta que su pokedex se mojó :(
La puso un rato en arroz y cuando la prendió, notó que la mayor parte funciona bien, solo no le anda una cosita del mapa, y necesita arreglarla porque no sabe salir de Lanus.
Analizando un poco el software, dedujo que el mapa es un **TDA** y que está implementado como una **lista doblemente enlazada**, es decir que **cada ciudad del mapa tiene un puntero a la próxima ciudad y a la anterior**.
Además, como el **TDA no estaba bien encapsulado**, obtuvo la estructura:

```
typedef struct ciudad {
    char nombre[MAX_NOMBRE];
    int habitantes;
    ...
    struct ciudad* proxima;
    struct ciudad* anterior;
} ciudad_t;

typedef struct mapa {
    ciudad_t* origen;
    ...
} mapa_t;
```

Variante 1

Se Pide

1. La funcionalidad que no le anda es la de **agregar una ciudad al mapa**, ayuda al joven *Pucci* a implementarla, cumpliendo con la siguiente firma:

```
/*
 * Agregará una ciudad al mapa, justo después de aquella cuyo nombre coincida con el recibido por parámetro.
 * Devolverá 0 si pudo agregarla o -1 si no.
 * Ejemplo:
 * Supongamos que queremos agregar la ciudad Azul luego de la ciudad Roja.
 * A su vez, la ciudad Roja tiene como siguiente a la ciudad Verde.
 * La ciudad Roja tendrá como siguiente a la ciudad Azul.
 * La ciudad Azul tendrá como siguiente a la ciudad Verde.
 * La ciudad Azul tendrá como anterior a la ciudad Roja.
 * La ciudad Verde tendrá como anterior a la ciudad Azul.
 */
int agregar_ciudad(mapa_t* mapa, ciudad_t* ciudad_nueva, char ciudad_anterior[MAX_NOMBRE]);
```

Variante 2

Se Pide

1. La funcionalidad que no le anda es la de **crear el mapa partiendo de un vector de ciudades**, ayuda al joven *Pucci* a implementarla, cumpliendo con la siguiente firma:

```
/*
 * Creará la estructura del mapa reservando la memoria necesaria.
 * Devolverá un puntero al mapa creado si pudo agregarla o NULL en caso contrario.
 * El mapa se cargará con las ciudades que se reciben por parámetro en el vector.
 * La relación entre las ciudades en el TDA mapa debe ser una analogía a la del vector.
 * Ejemplo:
 * En el vector, la ciudad Azul está en la posición 3, la ciudad Roja en la 4 y la Verde en la 5.
 * En la estructura mapa, la ciudad Azul tendrá como siguiente a la Roja.
 * La ciudad Roja tendrá como siguiente a la Verde.
 * La ciudad Roja tendrá como anterior a la Azul.
 * La ciudad Verde tendrá como anterior a la Roja.
 */
mapa_t* crear_mapa(ciudad_t ciudades[MAX_CIUDADES]);
```

Variante 3

Se Pide

1. La funcionalidad que no le anda es la de **borrar una ciudad del mapa**, ayuda al joven *Pucci* a implementarla, cumpliendo con la siguiente firma:

```
/*
 * Borrará del mapa a la ciudad cuyo nombre coincida con el recibido por parámetro.
 * Devolverá 0 si pudo o -1 si no.
 * Ejemplo:
 * La ciudad Azul tiene como siguiente a la Roja y la Roja a la Verde.
 * Se quiere borrar la ciudad Roja
 * La ciudad Azul tendrá como siguiente a la Verde
 * La ciudad Verde tendrá como anterior a la Azul.
 */
int borrar_ciudad(mapa_t* mapa, char ciudad_a_borrar[MAX_NOMBRE]);
```

Variante 4

Se Pide

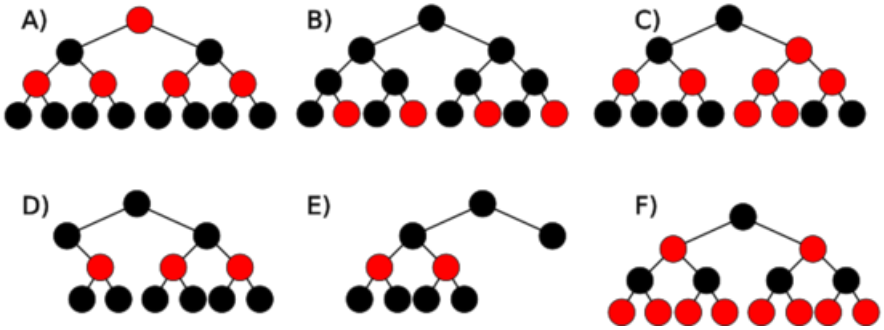
1. La funcionalidad que no le anda es la de **invertir los extremos**, ayuda al joven *Pucci* a implementarla, cumpliendo con la siguiente firma:

```
/*
 * Intercambiará las ciudades de los extremos del mapa.
 * Devolverá 0 si pudo agregarla o -1 si no.
 * La ciudad del comienzo de la lista debe quedar al final y la del final debe quedar adelante.
 */
int intercambiar_extremos(mapa_t* mapa);
```

EJERCICIO 3

Variante 1

¿Cuáles de los siguientes árboles rojo-negro son válidos para una determinada implementación? Justifique cada caso.

1. 

2. Insertar los siguientes elementos en (1,2,3,4,5,6) :

   a. AVL

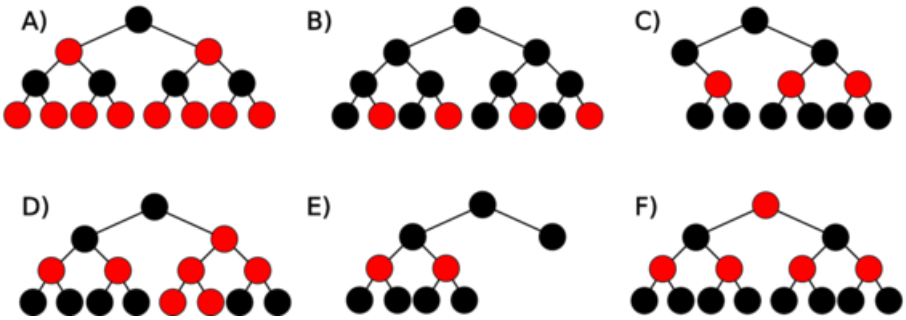
   b. Árbol Rojo Negro

   c. Árbol de Binario de Búsqueda

   Que conclusiones puede extraer

Variante 2

¿Cuáles de los siguientes árboles rojo-negro son válidos para una determinada implementación? Justifique cada caso.

1. 

2. Insertar los siguientes elementos en (7,8,9,10,11,12) :

   a. AVL

   b. Árbol Rojo Negro

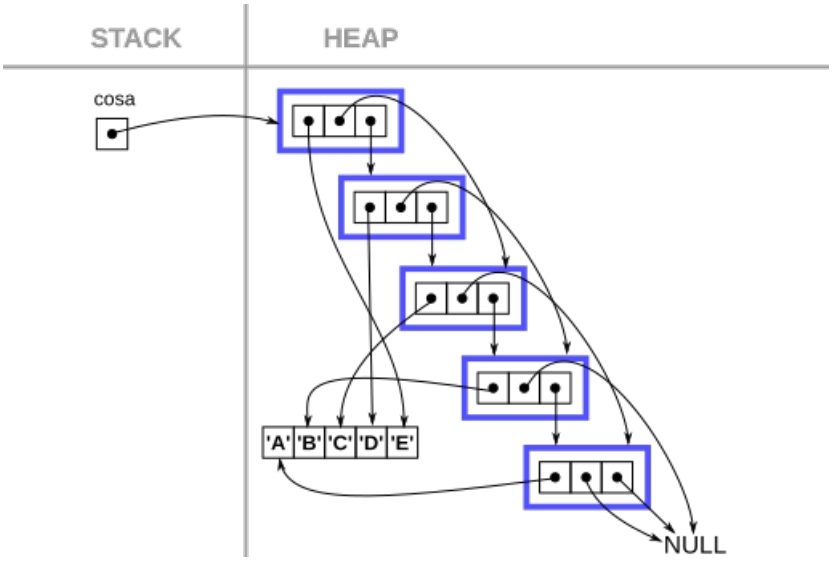
   c. Árbol de Binario de Búsqueda

   Que conclusiones puede extraer.

EJERCICIO 4

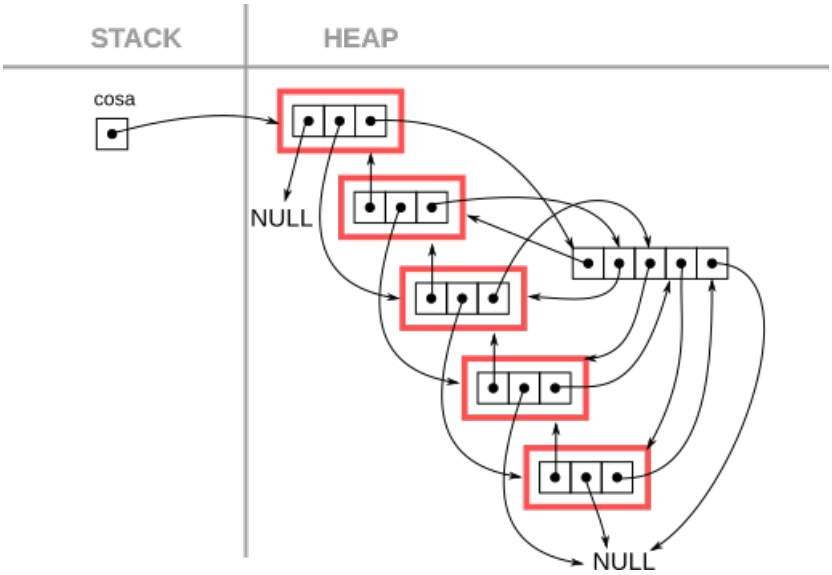
Variante 1

Escriba un programa (definiendo las estructuras y tipos que crea conveniente) de forma tal que el uso de memoria del mismo sea como el que se muestra a continuación. Muestre el código que hace que dicho programa libere correctamente toda la memoria reservada si es necesario.



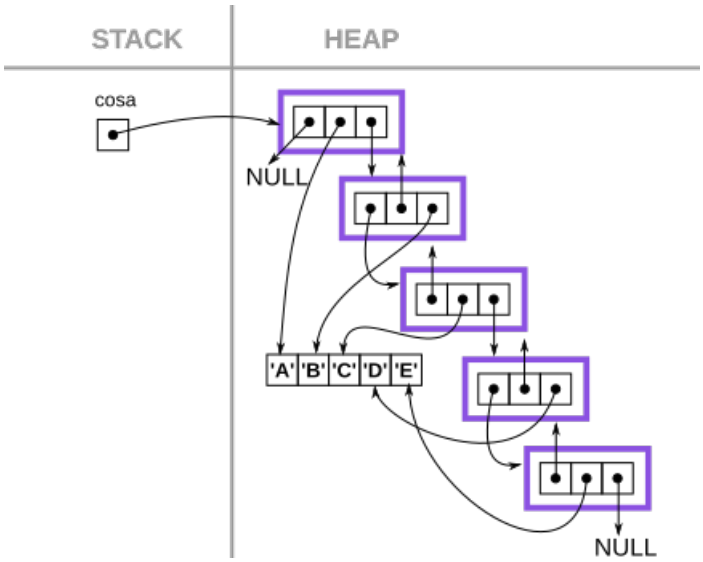
Variante 2

Escriba un programa (definiendo las estructuras y tipos que crea conveniente) de forma tal que el uso de memoria del mismo sea como el que se muestra a continuación. Muestre el código que hace que dicho programa libere correctamente toda la memoria reservada si es necesario.



Variante 3

Escriba un programa (definiendo las estructuras y tipos que crea conveniente) de forma tal que el uso de memoria del mismo sea como el que se muestra a continuación. Muestre el código que hace que dicho programa libere correctamente toda la memoria reservada si es necesario.


STACK
HEAP
cosa
NULL
'A'
'B'
'C'
'D'
'E'
NULL

EJERCICIO 5

Variante 1

Escriba (sin utilizar **for**, **while**, **do**) de manera recursiva, el algoritmo de ordenamiento por *burbujeo*. La funcion a implementar debe recibir un parámetro `int*`, con los elementos a ordenar y otro `size_t` con la cantidad de elementos del vector de enteros. Puede imlpementar funciones auxiliares si las necesita.

Variante 2

Escriba (sin utilizar **for**, **while**, **do**) de manera recursiva, una función `buscar_posiciones`. La funcion a implementar debe recibir un parámetro `const char*` (un string), y otro parámetro `char` y debe devolver un elemento del tipo `int*`.

La función debe encontrar en el string, la posición de todos los elementos con el valor del caracter pasado y devolver un vector de posiciones (terminado en -1) que indique las posiciones de dichos elementos dentro del string.

Se puede utilizar como máximo una función auxiliar (además de `buscar_posiciones`) y se puede utilizar un solo `malloc` / `calloc`. No se permite el uso de `realloc`.

Se puede asumir que `calloc` / `malloc` no fallan nunca y nunca devuelven `NULL`.

Ejemplo:

`buscar_posiciones("ABCDEFG", ',') => [-1]`

`buscar_posiciones("A,B,C,DEF,G", ',') => [1 , 3 , 5 , 9 , -1]`

`buscar_posiciones("155321197", '1') => [0 , 5 , 6 , -1]`

Variante 3

El algoritmo *Babilónico* para el cálculo de la raíz cuadrada de `x` consiste en seleccionar una estimación inicial de la raíz a calcular (por ejemplo `e=x/2`) y luego iterativamente verificar si la diferencia entre `e*e` y el numero original difieren a lo sumo un valor máximo `PRECISION`. A cada paso del algoritmo, se vuelve a calcular una nueva estimación de la forma `nueva_e=(e+x/e)/2`.

Sea `PRECISION=0.0001`, y sin utilizar **do**, **while** ni **for** (y obviamente sin utilizar la biblioteca matemática de C), escriba la función recursiva `raiz` que reciba un numero `double` **positivo** y devuelva la raíz cuadrada de ese número.