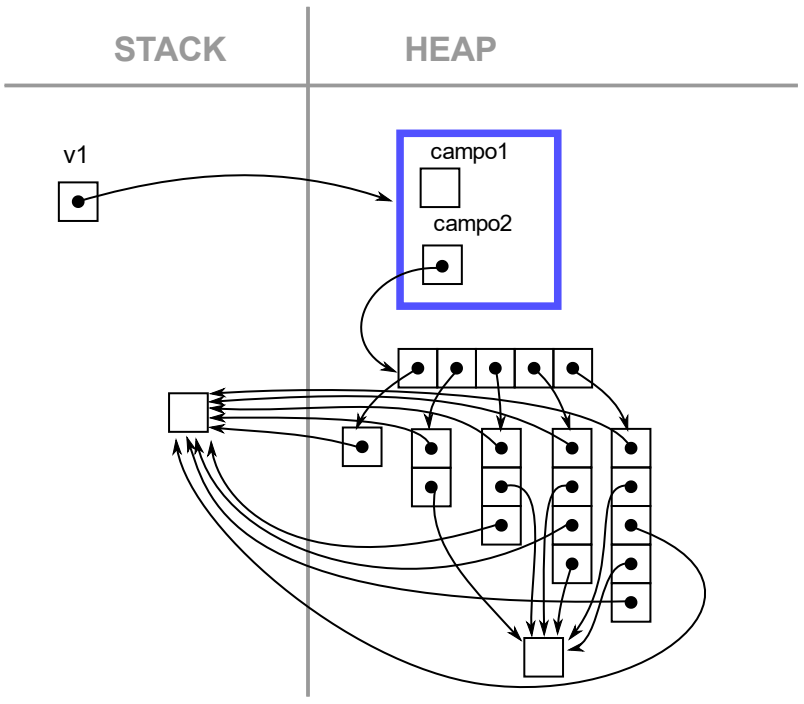
7541 - 20201C - 1er Parcial

Version 1

a. Escriba un programa (definiendo las variables, estructuras y tipos que crea conveniente) de forma tal que el uso de memoria del mismo sea como el que se muestra a continuación (puede agregar variables o punteros auxiliares si es necesario).

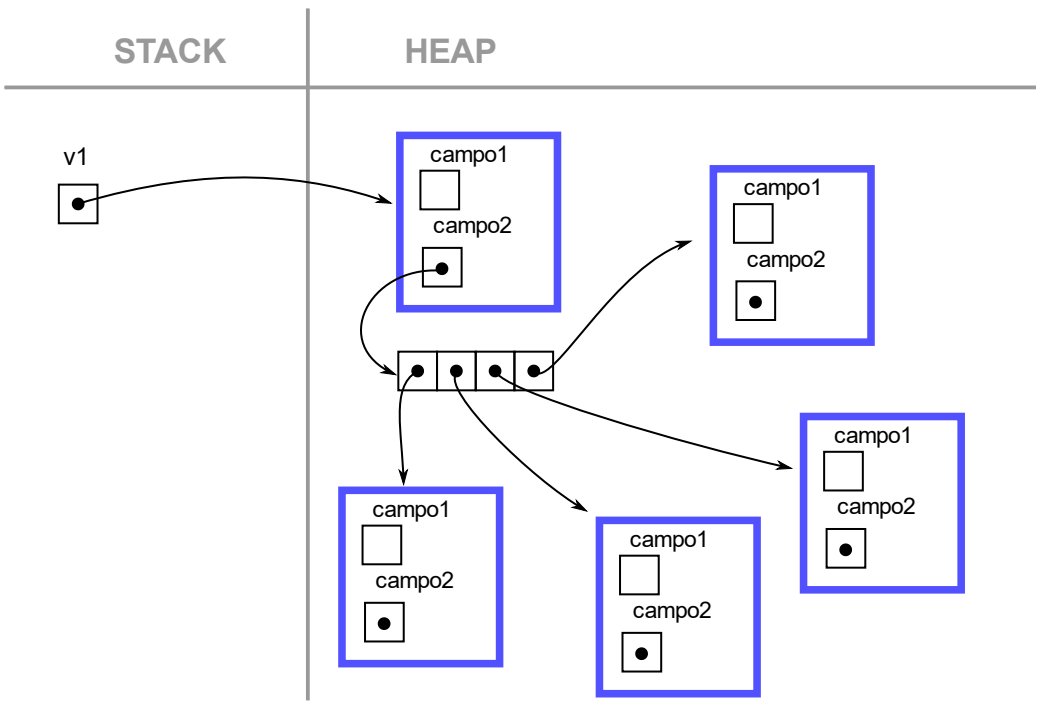
b. Muestre el código que hace que dicho programa libere correctamente toda la memoria reservada.



Version 2

a. Escriba un programa (definiendo las variables, estructuras y tipos que crea conveniente) de forma tal que el uso de memoria del mismo sea como el que se muestra a continuación (puede agregar variables o punteros auxiliares si es necesario).

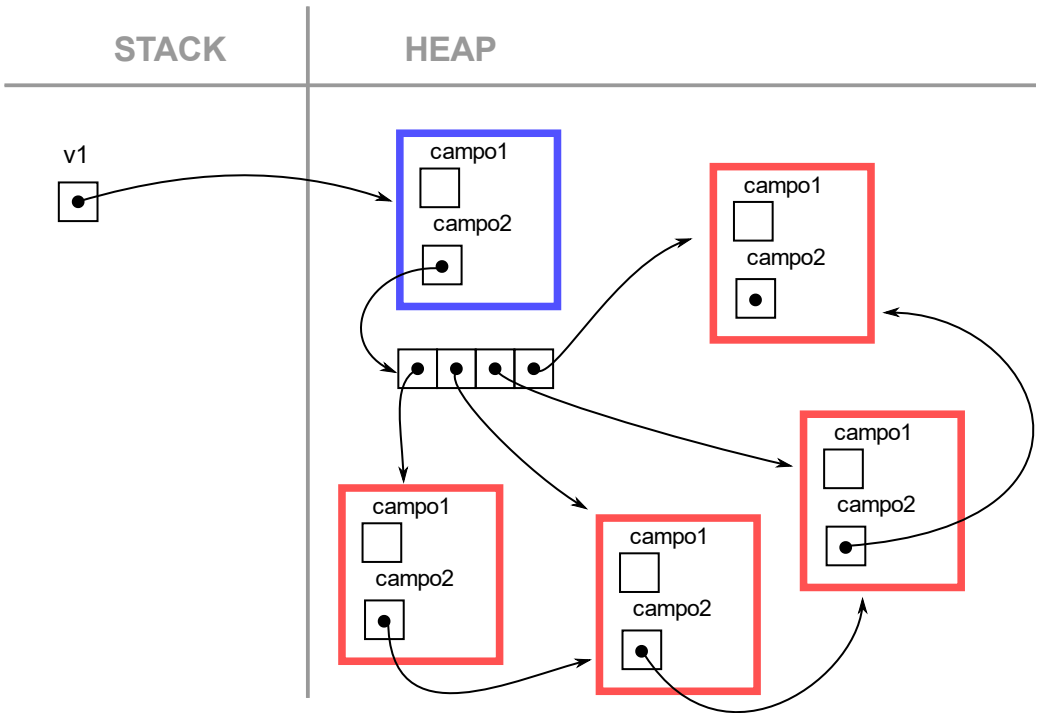
b. Muestre el código que hace que dicho programa libere correctamente toda la memoria reservada.



Version 3

a. Escriba un programa (definiendo las variables, estructuras y tipos que crea conveniente) de forma tal que el uso de memoria del mismo sea como el que se muestra a continuación (puede agregar variables o punteros auxiliares si es necesario).

b. Muestre el código que hace que dicho programa libere correctamente toda la memoria reservada.


STACK
HEAP
v1
campo1
campo2
campo1
campo2
campo1
campo2
campo1
campo2
campo1
campo2

7541 - 20201C - 1er Parcial

Version 1

a. Hallar la ecuación de recurrencia del **Merge Sort T(n)= A*T(n/B) + f(n)**, explicando detalladamente cómo se obtuvo. Calcular su Orden de complejidad en el peor, mejor y caso promedio.

b. Calcular la complejidad del siguiente código en C:

```c
for (int x = 0; x < n; x++)
    for (int y = 0; y < n; y++)
        for (int z = 0; z < n; z++)
            if (3*x + 9*y + 8*z == 79)
                printf("%i, %i, %i son solución de la ecuación", x, y, z);
```

Version 2

a. Hallar la ecuación de recurrencia del **Quick Sort T(n)= A*T(n/B) + f(n)**, explicando detalladamente cómo se obtuvo. Calcular su Orden de complejidad en el peor, mejor y caso promedio.

b. Calcular la complejidad del siguiente código en C:

```c
bool es_par (int numero) {
    return ((numero % 2) == 0);
}
```

Version 3

a. Explique detalladamente para qué sirve el **Teorema Maestro**, cuándo puede aplicarse, qué tres casos analiza, ponga un ejemplo de cada uno de ellos. Demuestre su aplicación en un caso especial.

b. Calcular la complejidad del siguiente código en C:

```c
int producto = 1;
for (int i = 0; i <= n; i += 1)
    printf ("%i", i);
for (int j = 1; j <= n; j += 1)
    for (int k = 243; k >= 1; k /= 3)
        producto = (k * j);
```

Version 4

Se sabe que la suma de los **n** primeros numeros de una secuencia es **O(n)**.

```c
int suma_seq (int n){
    int suma = 0;
    for (int i = 0; i <= n; i += 1) suma += i;
    return suma;
}
```

a. ¿Es posible mejorar el orden de complejidad sabiendo que esa suma es el resultado de **N(N+1)/2**? Implementar dicha Función.

b. Calcular la complejidad del siguiente código en C:

```c
int producto = 1;
for (int i = 0; i <= n; i += 1)
    printf ("%i", i);
for (int k = 1; k <= n; k *= 3)
    for (int i = 1; i <= k; i += 1 )
```

```
    for (int j = 1; j <= k; j += 1)
        producto = (k * j);
```

Version 5

a. Demostrar matematicamente que la busqueda binaria tiene la misma complejidad que **buscar** un elemento en un **AVL**.

b. Calcular la complejidad del siguiente código en C:

```
int suma = 0;
for (int i=0; i <= n; i += 1)
    for (int j = 0; j <= n ; j += 1)
        for (int k = 1; k <= n; k *= 3)
            suma = (i + j + k);
```

7541 - 20201C - 1er Parcial

Version 1

Los **Entrenadores pokemon** son personas que se encargan de capturar, criar, entrenar y cuidar de dichas criaturas.

Un entrenador en los juegos puede identificarse de diversas formas. No se dividen según su estrategia, sino por el tipo de pokemon que utilizan en batalla. De esta forma un nadador llevará pokemon de tipo agua, o un joven preferirá los de tipo normal o aquellos que sean muy frecuentes en su zona.

Existen pocas distinciones especiales entre los entrenadores pokemon: Líder de gimnasio, Alto Mando, Campeón de la Liga pokemon y As del Frente Batalla, cada una de ellas es muy díficil de alcanzar. Cada entrenador puede renombrar a sus pokemones, sin embargo, aunque los pokemones se intercambien, los nombres se mantienen.

Recomendamos, en base al texto anterior, usar su imaginación, si no le sale nada, piense que un **entrenador pokemon**:

- Puede tener muchos pokemones, pero no puede llevar a todos consigo, puede tener pokebolas vacías, etc.
- Puede especializarse en un tipo de pokemones o puede tener de todo tipo.
- Seguro recorrió ciudades, lucho contra otros entrenadores, intentó capturar pokemones y no pudo, etc.

Es necesario que haya manejo de memoria dinámica en las estructuras y en las primitivas a implementar. Las estructuras que almacenan muchos datos del mismo tipo, deben ser dinámicas.

a. Definir el TDA Entrenador pokemon (struct y primitivas).

b. Implementar una de las primitivas propuestas que utilice memoria dinámica.

c. Implementar las siguientes primitivas (puede agregar parámetros si lo considera necesario):

```
/*
 * Creará un entrenador pokemon.
 * Reservará toda la memoria necesaria e inicializará las variables que lo requieran.
 * Devolverá la referencia al pokemon creado o NULL si no pudo crearse.
 * Puede agregar parámetros si lo desea.
 */
entrenador_t* crear_entrenador(char nombre[MAX_NOMBRE], char ciudad_natal[MAX_CIUDAD]);

/*
 * Intentará capturar un pokemon.
 * En caso de poder, lo almacenará como pokemon actual.
 * Tener en cuenta que no el entrenador no puede llevar consigo muchos pokemones, si el entrenador es de
 * un tipo no suele capturar pokemones de otro tipo, los pokemones no se dejan capturar tan facilmente, etc.
 * Devolverá 0 si pudo capturar el pokemon o -1 si no.
 */
int capturar_pokemon(entrenador_t* entrenador, pokemon_t pokemon);
```

Version 2

Un **Gimnasio pokemon** es un edificio donde se disputan combates pokemon. Están presididos por un líder de gimnasio, que es el máximo responsable del mismo.

Los gimnasios oficiales de la Liga pokemon otorgan una medalla al entrenador que logre derrotar al líder del gimnasio.

Cada gimnasio se especializa en un tipo de pokemon y movimientos. Esto puede verse como ventaja para los jugadores, que siempre podrán encontrar un punto débil que explotar. Sin embargo, su cometido es ayudarles a entrenar y preparar estrategias para cada tipo, a fin de que mejoren como entrenadores pokemon.

Recomendamos, en base al texto anterior, usar su imaginación, si no le sale nada, piense que un **gimnasio pokemon**:

- Siempre tiene un líder y éste puede cambiar, a su vez, el líder tiene pokemones para utilizar.
- Muchos entrenadores luchan en el gimnasio, algunos ganan (se les entrega una medalla), otros no.
- Dentro del gimnasio se llevan a cabo peleas entre los pokemones del líder y los de un entrenador.

Es necesario que haya manejo de memoria dinámica en las estructuras y en las primitivas a implementar. Las estructuras

Es necesario que haya manejo de memoria dinámica en las estructuras y en las primitivas a implementar. Las estructuras que almacenan muchos datos del mismo tipo, deben ser dinámicas.

a. Definir el TDA Gimnasio pokemon (struct y primitivas).

b. Implementar una de las primitivas propuestas que utilice memoria dinámica.

c. Implementar las siguientes primitivas (puede agregar parámetros si lo considera necesario):

```
/*
 * Creará un gimnasio pokemon.
 * Reservará toda la memoria necesaria e inicializará las variables que lo requieran.
 * Devolverá la referencia al pokemon creado o NULL si no pudo crearse.
 * Puede agregar parámetros si lo desea.
 */
gimnasio_t* crear_gimnasio(entrenador_t* lider);

/*
 * Resgistrará una batalla llevada a cabo en el gimnasio.
 * Tener en cuenta que en caso de haber perdido el líder, se debe otorgar una medalla
 * la batalla tiene que tener un ganador, sería un error que haya un empate.
 * El líder, si perdió las últimas 10 batallas y el entrenador le ganó más del 50%
 * de las veces que se enfrentaron, será reemplazado por el entrenador.
 * Devolverá 0 si pudo registrarse la batalla o -1 si no.
 */
int registrar_batalla(gimnasio_t* gimnasio, batalla_t batalla);
```

Version 3

Los **Pokemones** son criaturas que, dependiendo de la especie o el tipo, pueden tener rasgos físicos parecidos a animales, plantas, rocas, fantasmas, o incluso humanos (aunque de tamaño en muchos casos muy superior a lo que mediría la especie a la que se parece).

Poseen poderes que pueden utilizar llevando a cabo movimientos o habilidades para atacar, defenderse o cumplir sus necesidades. Generalmente pueden vivir en todo tipo de entornos, subsistiendo en la libertad de la naturaleza o conviviendo con humanos y ayudándolos.

La evolución es una fase por la que pasan la mayoría de los pokemones durante su crecimiento y entrenamiento, cambiando algunas de sus características.

Recomendamos, en base al texto anterior, usar su imaginación, si no le sale nada, piense que un **pokemon**:

- Puede ser salvaje o pertenecer a un entrenador.
- Tiene un tipo, habilidades, ataques, es fuerte contra ciertos tipos y débil contra otros, etc.
- Puede tener la capacidad de evolucionar o no. Dicha evolución aumenta generalmente todas sus cualidades.

Es necesario que haya manejo de memoria dinámica en las estructuras y en las primitivas a implementar. Las estructuras que almacenan muchos datos del mismo tipo, deben ser dinámicas.

a. Definir el TDA Pokemon (struct y primitivas).

b. Implementar una de las primitivas propuestas que utilice memoria dinámica.

c. Implementar las siguientes primitivas (puede agregar parámetros si lo considera necesario):

```
/*
 * Creará un pokemon.
 * Reservará toda la memoria necesaria e inicializará las variables que lo requieran.
 * Devolverá la referencia al pokemon creado o NULL si no pudo crearse.
 * Puede agregar parámetros si lo desea.
 */
pokemon_t* crear_pokemon(char nombre[MAX_NOMBRE], char tipo, habilidad_t habiliadades[MAX_HABILIDADES], int
tope);

/*
 * Intentará evolucionar un pokemon.
 * Tener en cuenta que no todos las pokemones pueden evolucionar,
 * algunos porque no llegaron al nivel, otros porque directamente no pueden.
```

```
 * Al evolucionar, pueden aprender nuevas habilidades y potenciar las que ya tienen.

 * Devolverá 0 si pudo evolucionar el pokemon o -1 si no.
 */
int evolucionar(pokemon_t* pokemon, habilidad_t habiliadades[MAX_HABILIDADES], int tope);
```

Version 4

Una **Pokebola** es un objeto diseñado para servir dos funciones básicas en el mundo pokemon, almacenar y transportar pokemones, haciendo posible la captura de ellos.

Es probable que la primera pokebola fuese la de color rojo y blanco, que es la más común. A su vez, es la más utilizada por entrenadores pokemon y cuyas propiedades son más generales.

No obstante, existen más tipos. A lo largo de los años han surgido multitud de variedades con distintos colores y con efectos específicos que hacen unos tipos más idóneos con determinados pokemones o en determinados momentos.

Recomendamos, en base al texto anterior, usar su imaginación, si no le sale nada, piense que una **pokebola**:

- Puede o no contener un pokemon ahora mismo, y pudo haber almacenado otros pokemones en el pasado.
- Le pertenece a un entrenador, o le pertenecio a otros entrenadores, quizás no fue vendida aún.
- Tiene un tipo, que servirá a la hora de capturar un pokemon (será más efectiva o no).

Es necesario que haya manejo de memoria dinámica en las estructuras y en las primitivas a implementar. Las estructuras que almacenan muchos datos del mismo tipo, deben ser dinámicas.

a. Definir el TDA Pokebola (struct y primitivas).

b. Implementar una de las primitivas propuestas que utilice memoria dinámica.

c. Implementar las siguientes primitivas (puede agregar parámetros si lo considera necesario):

```
/*
 * Creará una pokebola.
 * Reservará toda la memoria necesaria e inicializará las variables que lo requieran.
 * Devolverá la referencia a la pokebola creada o NULL si no pudo crearse.
 */
pokebola_t* crear_pokebola(char tipo);

/*
 * Intentará capturar un pokemon.
 * En caso de poder, lo almacenará como pokemon actual.
 * Tener en cuenta que no todas las pokebolas son para todos pokemones, que la pokebola puede tener ya un
pokemon, etc.
 * Devolverá 0 si pudo capturarse el pokemon o -1 si no.
 */
int capturar_pokemon(pokebola_t* pokebola, pokemon_t pokemon);
```

Version 5

Un **Centro pokemon** es una edificación o lugar donde se cura o sana a los pokemon heridos, y donde los entrenadores pueden descansar y reponerse de su viaje pokemon e intercambiar los pokemones que uno lleva consigo.

En la planta baja se encuentra una enfermera. Casi siempre también hay gente, algunos llevan pokemon fuera de sus pokebolas, otros no.

Algunos días de la semana en algunos centros puedes encontrar un entrenador contra el que luchar al lado del mostrador. También se encuentra la tienda pokemon.

Recomendamos, en base al texto anterior, usar su imaginación, si no le sale nada, piense que un **centro pokemon**

- Tiene una o más personas dentro, que asisten por diferentes motivos.
- Un centro pokemon es como un hospital para ellos, se pueden hacer diferentes cosas.
- Los servicios que se brindan en el centro pueden estar habilitados o no y limitarse a una cantidad de pokemones por vez.

Es necesario que haya manejo de memoria dinámica en las estructuras y en las primitivas a implementar. Las estructuras que almacenan muchos datos del mismo tipo, deben ser dinámicas.

a. Definir el TDA Centro pokemon (struct y primitivas).

b. Implementar una de las primitivas propuestas que utilice memoria dinámica.

c. Implementar las siguientes primitivas (puede agregar parámetros si lo considera necesario):

```
/*
 * Creará un centro pokemon.
 * Reservará toda la memoria necesaria e inicializará las variables que lo requieran.
 * Devolverá la referencia al centro creado o NULL si no pudo crearse.
 */
centro_t* crear_centro_pokemon(char enfermera[MAX_NOMBRE]);

/*
 * Intentará sanar los pokemones recibidos.
 * Los centros que pueden sanar pokemones, solo pueden sanar una cantidad acotada por vez.
 * Devolverá 0 si pudo sanar a todos, o -1 si no.
 */
int sanar_pokemones(centro_t* centro, pokemon_t* pokemones[MAX_POKEMONES], int tope);
```

7541 - 20201C - 1er Parcial

Version 1

a. Dado un vector **v** y el método de ordenamiento **Quick Sort**, ponga un ejemplo en el cual el tiempo de ejecución sea **O(n^2)**. Explíquelo paso a paso.

Version 2

a. Dado un vector **v** ponga un ejemplo en el cual conviene usar **Merge Sort** en vez de **Quick Sort**. Explíquelo paso a paso.

Version 3

a. Dado el siguiente vector **v = {11, 2, 21, 4, 98, 67, 43, 5, 17}** ordenar el mismo utilizando **Quick Sort** eligiendo el pivote al elemento que corresponde al valor del medio del vector si la cantidad de elementos es impar, por el contrario si es par se toma el último elemento.

Version 4

a. Dado el siguiente vector **v = {11, 2, 21, 4, 98, 67, 43, 5, 17, 99, 22, 46, 55, 63, 51}** ordenar el mismo utilizando **Merge Sort** de 3 particiones en lugar de 2.

Version 5

a. Dado el siguiente vector **v = {11, 2, 21, 4, 98, 67, 43, 5, 17}** ordenar el mismo utilizando **Quick Sort** eligiendo el pivote al elemento que corresponde al valor del medio del vector si la cantidad de elementos es impar, por el contrario si es par se toma el primer elemento.

7541 - 20201C - 1er Parcial

Version 1

Dado el tipo recursivo **nodo_t**, y sin usar **while** / **for** / **do**, complete:

a. La función **calcular_altura** que cuente la cantidad de nodos existentes desde la raíz hasta la hoja mas alejada.

b. La función **completar_factor_de_equilibrio**, que complete el campo **fe** de todos los nodos del árbol.

```
typedef struct nodo {
    int fe;
    struct nodo* i;
    struct nodo* d;
} nodo_t;

int calcular_altura (nodo_t* raiz);

int completar_factor_de_equilibrio (nodo_t* raiz);
```

Version 2

Dado el tipo recursivo **nodo_t**, y sin usar **while** / **for** / **do**, complete:

a. La función **contar_hojas** que cuente la cantidad de nodos hoja de la estructura.

b. La función **contar_nodos_con_dos_hijos** que cuente la cantidad de nodos que poseen tanto hijo izquierdo como derecho.

```
typedef struct nodo {
    struct nodo* i;
    struct nodo* d;
} nodo_t;

int contar_hojas (nodo_t* raiz);

int contar_nodos_con_dos_hojas (nodo_t* raiz);
```

Version 3

a. Escriba una función recursiva **es_primo**, sin usar **while** / **for** / **do** (puede usar una función auxiliar), que indique si un número es primo o no.

b. Escriba de la misma forma la función **primeros_n_primos**, que devuelva un vector reservado con **malloc** / **calloc** / **realloc** con los primeros **n** numeros primos.

Ayuda: Un número es primo si es un entero mayor a 1 (1 no es primo) y sólo es divisible por si mismo (y por 1).
Dos enteros i, j son divisibles entre sí si **(i % j) == 0**.

```
bool es_primo (int n);

double* primeros_n_primos (int n);
```

Version 4

a. Escriba una función recursiva **invertir_de_a_pares**, sin usar **while** / **for** / **do** (puede usar una función auxiliar), que reciba un string e invierta la posición de los caracteres de a pares en secuencia.

Por ejemplo:
"1,2,3," -> ",1,2,3"
"ABCDEF" -> "BADCFE"
"ABCDE" -> "BADCE"

```
void invertir_de_a_pares (char* s);
```

b. Implemente (con las mismas condiciones) la función **proxima_potencia_de_dos**, que dado un entero **n > 0**, calcule la potencia de 2 mayor o igual a **n** mas cercana.

Por ejemplo:
1 -> 1
2 -> 2
3 -> 4
5 -> 8
7 -> 8
15 -> 16
1000 -> 1024

```
int proxima_potencia_de_dos (int n);
```
