Explique de qué se trata el **teorema maestro** y cómo se utiliza.Ejemplifique cada caso que resuelve con una ecuación de concurrencia.

El teorema maestro proporciona una solución a las ecuaciones de recurrencia de la forma:

$$T(n) = aT(n/b) + f(n):$$

Cuando a>=1, b>1 y f(n) es una función polinómica asintóticamente positiva. La solución que propone el teorema es diferente según la relación existente entre f(n) y los coeficientes a y b. Los casos posibles ,con sus soluciones T(n), son:

$$\begin{cases} f(n)\ es\ menor\ que\ n^{\log_b a} entonces & T(n) = \Theta(n^{\log_b a}). \\ f(n)\ es\ igual\ que\ n^{\log_b a}\ entonces & T(n) = \Theta(n^{\log_b a}\ lg\ n) \\ f(n)\ es\ polinomicamente\ mayor\ que\ n^{\log_b a},\ entonces & T(n) = \Theta(f(n)). \end{cases}$$

Ejemplo 1:

Para T(n) = 2T(n/2)+1, calculamos n^(log_b(a)) = n^1 = n. Como f(n) es menor al coeficiente, la solución del problema es **T(n)=Θ(n)**

Ejemplo 2:

Para **T(n) = T(n/2)+1**, calculamos n^(log_b(a)) = n^0 = 1. Como f(n) es igual al coeficiente, la solución del problema es **T(n)=Θ(log(n))**

Ejemplo 3:

Para **T(n) = T(n/2)+n**, calculamos n^(log_b(a)) = n^0 = 1. Como f(n) es polinomicamente mayor, la solución del problema es **T(n)=Θ(n)**

Dada la siguiente definición de ABB recursivo:

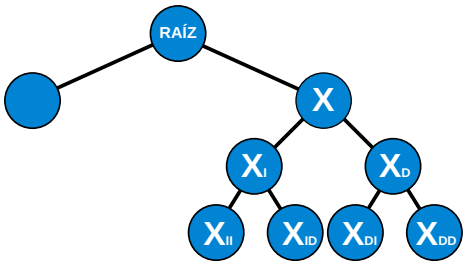
```c
typedef struct nodo{
  void* dato;
  struct nodo* izquierda;
  struct nodo* derecha;
}nodo_t;
```

A. Implemente las siguientes funciónes:

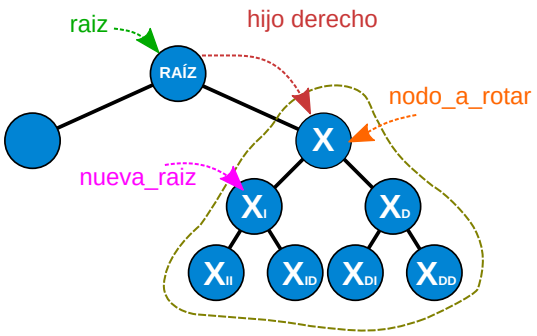
```c
nodo_t* rotar_derecha(nodo_t* nodo_a_rotar);
nodo_t* rotar_izquierda(nodo_t* nodo_a_rotar);
```

B. Complete el siguiente código con una llamada a la función implementadapara rotar el nodo X.

```c
int main(){
  nodo_t* arbol = NULL;
  arbol = arbol_crear(/*....*/);
  arbol = arbol_insertar(/*....*/);

  //... mas cosas ...//

  //Acá quiero rotar el nodo X antes de terminar

  <su código aquí>

  arbol_destruir(arbol);
  return 1;
}
```

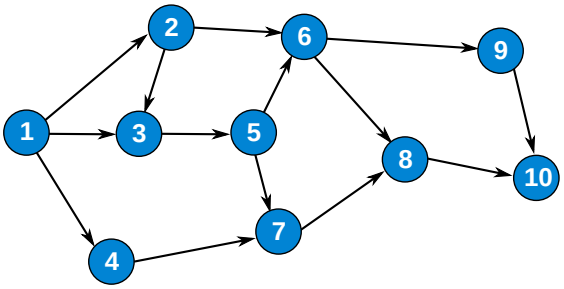



```c
int main(){
  nodo_t* arbol = NULL;
  arbol = arbol_crear(/*....*/);
  arbol = arbol_insertar(/*....*/);

  //... mas cosas ...//

  //Acá quiero rotar el nodo X antes de terminar

  arbol->derecha = rotar_derecha(arbol->derecha);

  arbol_destruir(arbol);
  return 1;
}


nodo_t* rotar_derecha(nodo_t* nodo_a_rotar){
  nodo_t* nueva_raiz = nodo_a_rotar->izquierda;

  nodo_a_rotar->izquierda = nueva_raiz->derecha;
  nueva_raiz->derecha = nodo_a_rotar;
  return nueva_raiz;
}
```

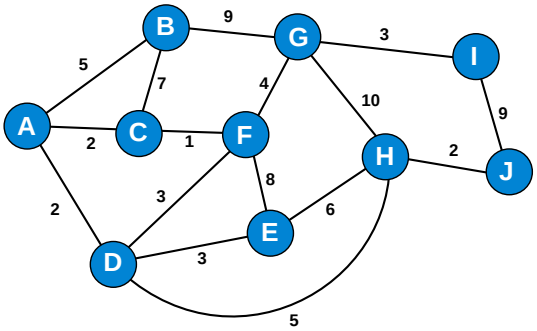


Exprese el siguiente grafo como **matriz de adyacencias** y como **matriz de incidencias**

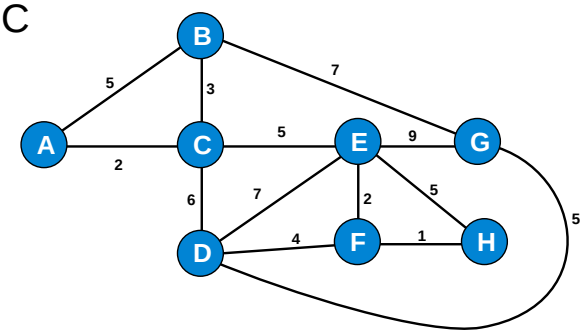
A



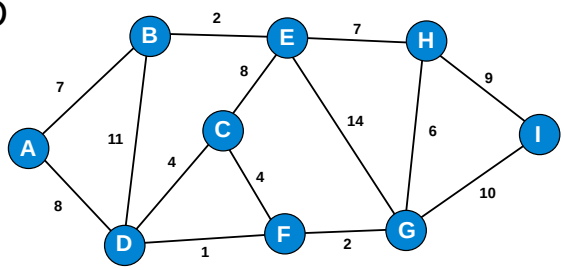
B



C



D



Simplemente se pide armar las matrices. Por cuestiones de espacio solo se muestran A y B.

A

	1	2	3	4	5	6	7	8	9	10
1	0	1	1	1						
2		0	1			1				
3			0		1					
4				0			1			
5					0	1	1			
6						0		1	1	
7							0	1		
8								0		1
9									0	1
10										0

	1	2	3	4	5	6	7	8	9	10
1-2	-1	1								
1-3	-1		1							
1-4	-1			1						
2-3		-1	1							
2-6		-1				1				
3-5			-1		1					
4-7				-1			1			
5-6					-1	1				
5-7					-1		1			
6-8						-1		1		
6-9						-1			1	
7-8							-1	1		
8-10								-1		1
9-10									-1	1

B

	A	B	C	D	E	F	G	H	I	J
A	0	5	2	2						
B	5	0	7				9			
C	2	7	0			1				
D	2			0	3	3		5		
E				3	0	8		6		
F			1	3	8	0	4			
G		9				4	0	10	3	
H				5	6		10	0		2
I							3		0	9
J								2	9	0

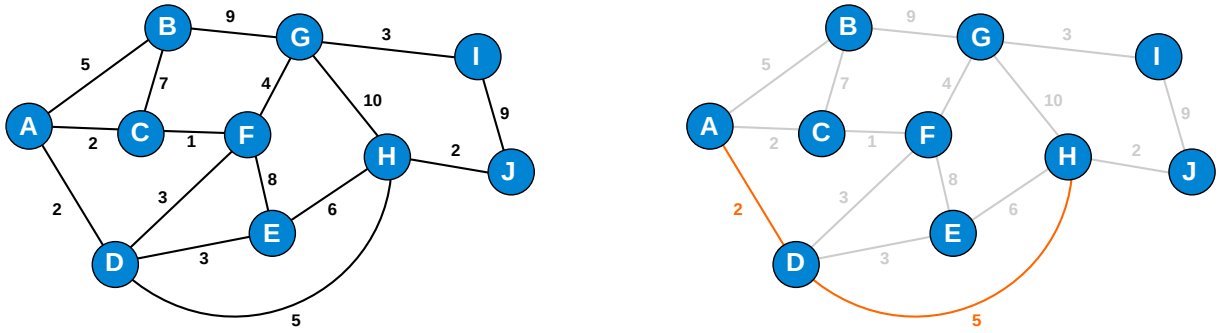
	A	B	C	D	E	F	G	H	I	J
A-B	5	5								
A-C	2		2							
A-D	2			2						
B-C		7	7							
B-G		9					9			
C-F			1			1				
D-E				3	3					
D-F				3		3				
D-H				5				5		
E-F					8	8				
E-H					6			6		
F-G						4	4			
G-H							10	10		
G-I							3		3	
H-J								2		2
I-J									9	9

Explique como funciona el algoritmo de **Floyd-Warshall** y muestre (en código) cómo podría implementarse.

El algoritmo de Floyd-Warshall encuentra el camino mínimo entre todos los pares de vértices de un grafo. Funciona tomando todos los pares de vértices A,B posibles del grafo y comparando la distancia conocida A-B con la distancia A-C-B (para todo vertice C del grafo). De esta forma se va actualizando y minimizando la distancia A-B a medida que se comparan mas recorridos.

```
for(int i=0;i<cantidad_vertices(grafo);i++)
  for(int j=0;j<cantidad_vertices(grafo);j++)
    for(int k=0;k<cantidad_vertices(grafo);k++)
      matriz_distancias[i][j] = minimo(distancia(i,j), distancia(i,k)+distancia(k,j));
```

Explique como funciona el algoritmo de **Dijkstra** y apliquelo desde el vértice **A** hasta el **H**



El algoritmo de Dijkstra arma una tabla con las distancias conocidas (directas) desde un vertice inicial hacie el resto de los vertices (inicialmente solo con sus vecinos directos). En cada paso el algoritmo toma el vértice con menor distancia de la lista (aun no visitado) y actualiza las distancias en base a la distancia recorrida para llegar hasta dicho vértice y la distancia a sus vecinos directos. El algoritmo finaliza cuando el menor vértice no visitado de la tabla es el vértice al que queremos llegar. Adicionalmente, si se quiere reconstruir el camino mínimo (además de conocer su valor) se debe guardar en la tabla la información del vértice desde el cual se llega a cada uno de los visitados.

Vértice	A	B	C	D	E	F	G	H	I	J
Distancias (1)	0+	5	2	2	-	-	-	-	-	-
Distancias (2)	0*	5	2+	2	-	3	-	-	-	-
Distancias (3)	0*	5	2*	2+	5	3	-	7	-	-
Distancias (4)	0*	5	2*	2*	5	3+	7	7	-	-
Distancias (5)	0*	5+	2*	2*	5	3*	7	7	-	-
Distancias (6)	0*	5*	2*	2*	5+	3*	7	7	-	-
Distancias (7)	0*	5*	2*	2*	5*	3*	7+	7	10	-
Distancias (8)	0*	5*	2*	2*	5*	3*	7*	7+	10	9

En este caso, se ve como en el paso 7 el menor camino a un vértice sin visitar es el de H (7), por lo tanto no puede existir mejor camino que el encontrado. Con esta versión de la tabla, no podríamos recuperar el camino que nos lleva de A hasta H ya que no tenemos el vértice anterior guardado, sin embargo, al tener todos los estados de la tabla, podemos recorrer hacia arriba hasta encontrar un cambio de valor en la columna y saltando al activo en esa fila. El valor de H cambia en la fila 3, cuando estaba activo D y D cambia su valor en la fila 1 (estaba desde el principio) cuando estaba activo A. Por lo tanto el camino es **A-D-H** con peso 7.

Explique si la siguiente estructura puede o no representar algún tipo de **tabla de hash**, y justifique.

A1. | 6 | | | | 4 | 5 |

A2. | | 7 | | 3 | | 5 |

A3. | 6 | | 2 | | | 5 |

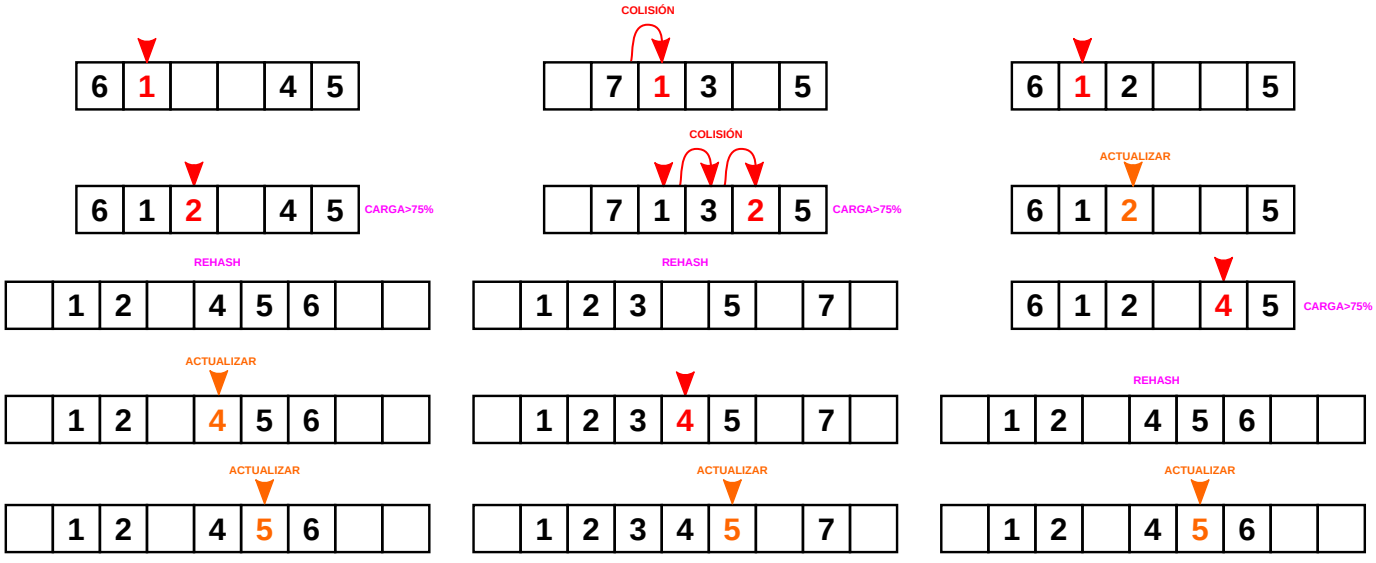
En caso negativo, primero explique (y defina) qué es necesario para que pueda ser una **tabla de hash**. Caracterize la tabla, explique sus propiedades y cómo funciona. Inserte en la tabla los valores **1**, **2**, **4**, y **5** mostrando cómo queda la tabla luego de cada operación.

Acá las respuestas pueden variar. Pero básicamente se podría tener un hash con esa estructura sólo en el caso de **hashing cerrado**. El hashing abierto requiere encadenar colisiones y la estructura es solamente un vector con valores.

No importa tanto si dicen que puede ser o que no puede ser, importa la justificación. Puede ser porque la estructura encaja con un **hashing cerrado**, puede no ser porque la estructura sola no alcanza (tiene que existir una función de hashing, y en este caso una estrategia de resolución de colisiones, también se puede aceptar si dicen que necesitan almacenar ademas de la clave un valor, marcas de borrado, información extra como ser capacidad del vector, etc). Como siempre les decimos, las respuestas posibles pueden ser muchas (e incluso contradictorias entre sí), lo importante es que la justificación sea razonable y esté bien planteada.

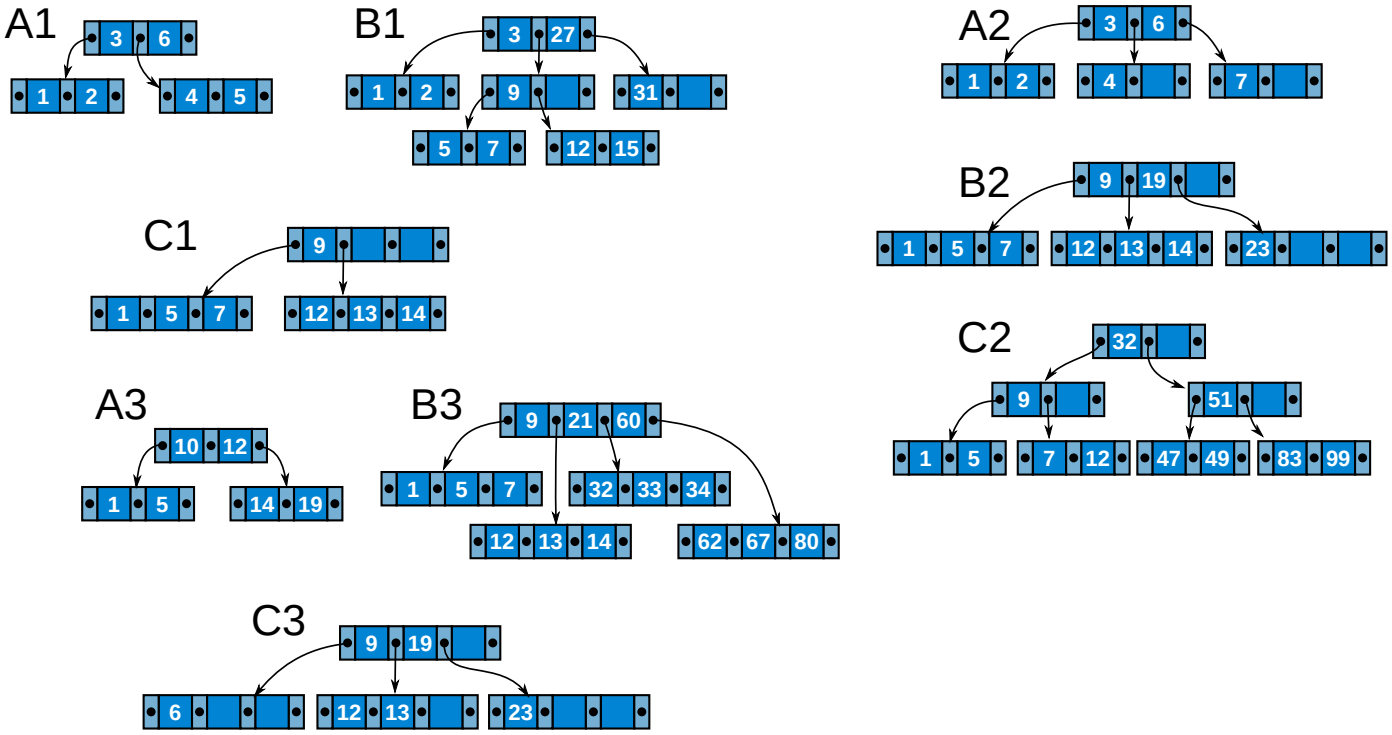
Según la justificación, la siguiente parte varía, pero en esencia consiste en terminar de definir las características del hash. En primer lugar ya sabemos que es un **hashing cerrado**. Podemos decir que la resolución de colisiones por ejemplo es lineal y que no presenta zona de desborde. Lo que nos queda fundamental para poder utilizar la tabla, es una función de hashing. En principio no importa qué función se decida utilizar, pero tiene que ser consistente con los valores ya insertados en la tabla. Las 3 tablas presentadas pueden acomodarse con la función **hash(n) = n**, ya que cada valor termina correspondiendose con su posición en el vector (mod 6, la capacidad del vector). También era válido decir que la función de hashing era **hash(n)=n%6** (y en realidad cualquier otra que se les ocurra que coincida), aunque en este caso la función va a ser bastante mala a medida que aumente el tamaño del hash (ya que siempre va a utilizar solamente las primeras 6 posiciones).

Una vez definido esto, se pueden insertar los valores:



OBS: El **rehash** debería haber agrandado mas el vector, pero no se dibuja por motivos de espacio.

Indique si los siguientes son **árboles B** válidos. Justifique.



Simplemente se pde verificar que los árboles cumplan las reglas de árbol B. Todos los nodos hoja deben estar al mismo nivel. Los punteros a cada lado de un valor insertado en un nodo no hoja deben apuntar a un nodo hijo, sin excepcion. Un nodo (no raíz) puede tener un mínimo de m/2-1 claves. Las claves se insertan ordenadamente, menores a la izquierda, mayores a la derecha.