Nombre y apellido: _____

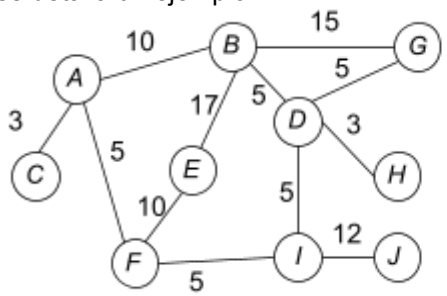
Ejercicios (puntaje)			Nota
1 (4 pts)	**2 (3 pts)**	**3 (3 pts)**	

CONSIDERACIONES: RESOLVER CADA EJERCICIO EN UNA HOJA DIFERENTE. LOS EJERCICIOS QUE NO ESTÉN CORRECTOS EN UN 50% NO SUMARÁN PUNTOS PARA LA NOTA FINAL. PARA CADA EJERCICIO DEBE **DEFINIR EL TIPO DE DATO** DE CADA TDA UTILIZADO. **EN TODOS LOS CASOS QUE UTILICE ESTRUCTURAS** QUE NO SEAN TDAs "ESTÁNDAR" **DEBE DEFINIR LOS STRUCTS**.

Ejercicio 1: Dada una red social modelada por un grafo ponderado, **desarrollar una función que devuelva una lista dinámica simplemente enlazada con los contactos sugeridos para un determinado perfil (contact)**. El vínculo entre dos perfiles es ponderado en función de las interacciones por mes que existen entre ellos y se van a considerar como contactos sugeridos aquellos que sean amigos de un contacto amigo con valor mayor a **Na** interacciones por mes y a su vez la interacción con el amigo en común tiene que ser mayor a **Naa** interacciones por mes.

Encabezado función principal: `fs_list* friendship_suggestion (graph* social_networking, t_graph_elem contact, int Na, int Naa );`

A continuación se detalla un ejemplo:



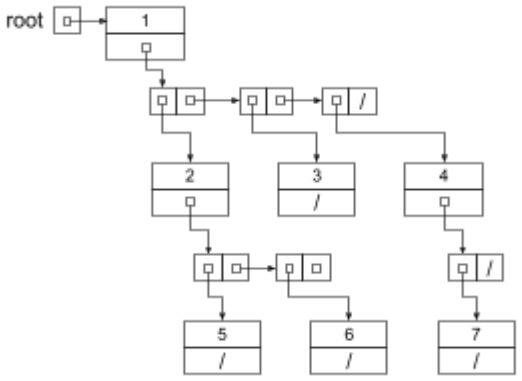
Para D con Na = 5 y Naa = 10 el vector de sugerencias de amistad devuelto es:

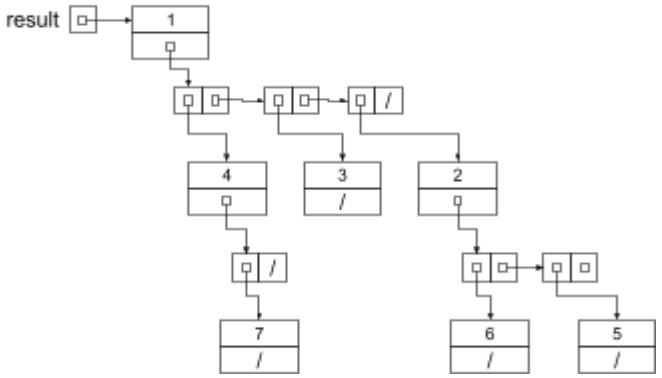
V = { A, E, J }

Ejercicio 2: Desarrollar una función y todas las necesarias para, dado un vector (TDA Vector) de punteros a personas (con DNI, NOMBRE, APELLIDO) **ordenado por apellido, crear un árbol binario de búsqueda de personas ordenado por apellido**. Resolver lo más eficientemente posible, de modo que el árbol quede balanceado.

Encabezado función principal: `btn* sbt_crete_from_vector (vector* v);`

Ejercicio 3: Desarrollar una función y todas las necesarias para que, dado un árbol n-ario implementado con listas dinámicas enlazadas, se **cree un árbol nuevo clonando el árbol en espejo;** es decir, el primer hijo de cada nodo debe ser el último y viceversa. Ejemplo:





Encabezado función principal: `ntn* ntn_create_mirror (ntn* root);`

TDAs

```c
typedef struct {...} vector; //define t_vector_elem
vector* vector_new(); // Crea el vector
void vector_free(vector* v); // Eliminar el vector
int vector_size(vector* v); // Permite obtener el tamaño actual del vector
int vector_isempty(vector* v); // 0 si no está vacío y 1 si está vacío.
t_vector_elem vector_get(vector* v, int index); // obtener el valor de una posición
t_vector_elem vector_set(vector* v, int index, t_vector_elem value); // reemplazar el valor
int vector_add(vector* v, t_vector_elem value); // agregar un elemento al final
int vector_insert(vector* v, int index, t_vector_elem value); // agregar en posición
void* vector_remove(vector* v, int index); // eliminar un elemento
```

```c
typedef struct {...} matrix; //define t_matrix_elem
matrix* matrix_new(); //Crea la matriz
void matrix_free(matrix* m); //Elimina la matriz
int matrix_rows(matrix* m); //Permite obtener la cantidad de filas
int matrix_columns(matrix* m); //Permite obtener la cantidad de columnas
t_matrix_elem matrix_get(matrix* m, int row, int col); //obtener el valor de posición
void matrix_set(matrix* m, int row, int col, t_matrix_elem value); //reemplazar
```

```c
typedef struct {...} list; //define t_list_elem
list *list_new();
void list_free(list *L);
bool list_isempty(list *L);
int list_length(list *L);
t_list_elem list_get(list *L, int index);
int list_search(list *L, t_list_elem elem, int cmp(t_list_elem a, t_list_elem b));
void list_insert(list *L, int index, t_list_elem elem);
void list_delete(list *L, int index);
t_list_elem list_remove(list *L, int index);
void list_traverse(list *L, bool look(t_elem elem, int index, void *ctx), void *ctx);
```

```c
typedef struct {...} graph; //define t_graph_elem
graph* graph_new(); //Crea el grafo
void graph_destroy(graph* g); //Destruye el grafo
int graph_add_vertex(graph* g, t_graph_elem vertex); //Agrega un vertice al grafo
t_graph_elem graph_vertex_get(graph* g, int index); //devuelve el elemento de un vértice
int graph_vertex_index(graph* g, t_graph_elem vertex, int cmp (t_graph_elem, t_graph_elem)); //devuelve el índice de un
vértice
int graph_add_edge(graph* g, int v1, int v2, int weight); //Agrega una arista al grafo
int graph_remove_edge(graph* g, int v1, int v2, int weight); //Elimina una arista del grafo
t_graph_elem graph_remove_vertex(graph* g, int v); //Elimina un vértice
int graph_get_edge_weight(graph* g, int v1, int v2); //Devuelve el peso de una arista
list* graph_vertex_adjacent_list(graph* g, int v); //Devuelve la lista de adyacencia de un vértice
```

```c
typedef struct {...} queue;
//define t_queue_elem
queue* queue_new();
void queue_free (queue* q);
int queue_getsize(queue* q);
int queue_isempty (queue* q);
void enqueue (queue* q, t_queue_elem elem);
t_queue_elem dequeue (queue* q);
t_queue_elem peek (queue* q);
```

```c
typedef struct {...} stack;
//define t_stack_elem
stack* stack_new();
void stack_free(stack* s);
int stack_getsize(stack* s);
int stack_isempty(stack* s);
void push(stack* s, t_stack_elem elem);
t_stack_elem pop(stack* s);
t_stack_elem top(stack* s);
```