Appello del 19 febbraio 2021
Soluzioni parte in C

Esercizio 1
Si richiede di implementare un programma in C che permetta di inserire, all'interno di una lista concatenata, dei nodi costituiti da una chiave intera e da un contatore intero:

```
struct node {
    struct node *next;
    int key;
    int counter;
}
```

Il programma deve supportare l'esecuzione delle seguenti due funzioni:

- `void insert_node(struct list **head, float value)`: accetta un valore float espresso in notazione IEEE 754 a 32 bit. La funzione estrae dal valore in virgola mobile una stringa binaria costituita dalla mantissa ed interpreta tale stringa binaria come un intero. Effettua quindi una scansione della lista puntata da head. Se viene trovato un nodo la cui chiave corrisponde al valore della mantissa, incrementa il contatore del nodo corrispondente. Viceversa, se non viene trovato alcun nodo contrassegnato da questa chiave, inserisce un nuovo nodo in testa alla lista, impostando il contatore del nodo a 1.

- `void remove_node(struct list **head, float value)`: questa funzione accetta un valore float espresso in notazione IEEE 754 a 32 bit. La funzione estrae dal valore in virgola una stringa binaria costituita dalla mantissa ed interpreta tale stringa binaria come un intero. Effettua quindi una scansione della lista puntata da head. Se viene trovato un nodo la cui chiave corrisponde al valore della mantissa, rimuove dalla lista tale nodo.

Per semplicità, si assuma di implementare tale programma su un'architettura che utilizza un ordine dei byte per la rappresentazione dei dati in memoria di tipo little endian.

Soluzione.
La soluzione dell'esercizio richiede di risolvere due problemi di natura differente. Il primo problema riguarda la manipolazione dei numeri in virgola mobile in formato IEEE 754 a 32 bit, ovverosia un float nei sistemi convenzionali.
L'esercizio richiede di estrarre la mantissa dal numero, considerando un sistema che memorizza i valori con un ordine dei byte di tipo little endian. Questo vuol dire che, considerando ad esempio il valore n = $(1{,}234)_{10}$, corrispondente a (00111111100111011111001110110110) in formato IEEE 754 (o 0x3f9df3b6 in notazione esadecimale), questo verrà mantenuto in memoria partendo dal byte meno significativo, a partire dall'indirizzo associato alla variabile di tipo float. Pertanto si avrà una rappresentazione in memoria del tipo:

B6	F3	9D	3F
indirizzo di n	indirizzo di n + 1 byte	indirizzo di n + 2 byte	indirizzo di n + 3byte

La mantissa si troverà nei byte meno significativi. Per estrarre correttamente il suo valore, si può ricorrere ad una *union cast*, sfruttando la seguente definizione:

```
union flt {
   struct ieee754 {
     uint32_t mantissa: 23;
     uint32_t exponent: 8;
     uint32_t sign: 1;
   } raw;
   float f;
};
```

Si noti che il primo membro della struct è la mantissa, a causa della rappresentazione little endian discussa.
Utilizzando questa unione, è possibile definire una macro per estrarre la mantissa da un valore float come segue:

```
#define mantissa(val) ({      \
      union flt _f;           \
      _f.f = val;             \
      (int)_f.raw.mantissa;   \
})
```

Un'alternativa è quella di sfruttare cast tra puntatori per permettere di interpretare il contenuto di un'area di memoria in maniera differente, senza soffrire di problemi legati alla endianness:

```
#define mantissa_ptr(val)  ({           \
      int _converted = *(int *)&val;    \
      _converted & 0x7fffff;            \
})
```

dove `0x7fffff` è una maschera di bit con i 23 bit meno significativi impostati a 1.

È invece un errore grave pensare di poter estrarre la mantissa con un cast a intero come segue:

```
int mantissa = (int)value;
```

poiché il cast converte *tutto* il numero ad un valore intero (effettuando di fatto il troncamento della parte frazionaria).

Un'alternativa, utilizzata da alcuni, è quella di utilizzare la funzione di libreria `float frexpf(float x, int *exp)`, che permette di convertire un numero in virgola mobile in una parte frazionaria e nell'esponente. Tuttavia, la parte frazionaria viene restituita ancora una volta come float, pertanto per convertirla ad intero sarà necessario utilizzare nuovamente una delle tecniche discusse precedentemente. Peraltro, l'implementazione di `frexpf()` in buona parte delle librerie standard utilizza una union cast, oppure delle maschere di bit, esattamente come discusso in precedenza.

La seconda parte dell'esercizio richiede di implementare una lista singolarmente concatenata, realizzando due funzioni di inserimento/rimozione. Nella soluzione proposta, si utilizza una lista con nodo sentinella, al fine di semplificare l'implementazione degli algoritmi.

Per quanto riguarda la funzione di inserimento, il "caso particolare" da gestire è quello dell'inserimento di un nodo associato ad una chiave non presente in lista. Per cercare di "regolarizzare" il più possibile il caso particolare, si può optare per effettuare prima una scansione della lista, utilizzando un cursore `curr`: se non si individua un nodo associato alla chiave cercata, si procede ad effettuare l'inserimento in testa, andando ad impostare il contatore del nuovo nodo a 0. In questo modo, si può "far finta" che la scansione lineare abbia avuto successo e che `curr` stia puntando al nodo trovato. In questo modo, si effettuerà sempre l'incremento del contatore a fine funzione.

```
void insert_node(struct node **head, float value)
{
      struct node *new, *curr = *head;
      int mantissa = mantissa(value);

      while (curr && curr->key != mantissa)
            curr = curr->next;

      if(!curr) {
            new = malloc(sizeof(struct node));
            new->key = mantissa;
            new->counter = 0;
            new->next = (*head)->next;
            curr = (*head)->next = new;
      }

      curr->counter++;
}
```

L'eliminazione segue uno schema simile: si effettua una ricerca lineare e, se si individua un nodo che deve essere eliminato, lo si rimuove. Tuttavia, in fase di eliminazione, è necessario evitare memory leak, andando a liberare la memoria associata al nodo che si sta rimuovendo. Poiché l'eliminazione logica del nodo (ovverosia il momento in cui si fa puntare il membro next del nodo al successivo) è un'operazione distruttiva (ovverosia, ci fa perdere il riferimento al nodo da eliminare), è necessario creare una copia temporanea del puntatore per effettuare correttamente la `free()`.

```
void remove_node(struct node **head, float value)
{
	struct node *ptr;
	int mantissa = mantissa(value);

	while ((*head) && (*head)->key != mantissa)
		head = &(*head)->next;

	if(*head) {
		ptr = *head;
		*head = (*head)->next;
		free(ptr);
	}
}
```

Esercizio 2
Una matrice sparsa è una matrice in cui la maggior parte degli elementi sono zero, ovverosia una matrice è sparsa se almeno (m * n) / 2 elementi sono zero.

Scrivere una funzione C che accetti in input una matrice di dimensione arbitraria e che restituisca true se questa matrice è sparsa.

Soluzione.
Il cuore di questo esercizio è un doppio ciclo che consente di contare quanti elementi della matrice valgono zero.
La richiesta di gestire matrici di dimensioni arbitrarie ci impedisce di poter conoscere a tempo di compilazione la taglia delle matrici che si vogliono controllare. Pertanto, il prototipo della nostra funzione potrà essere uno dei due seguenti:

```
bool check(size_t r, size_t c, int M[r][c]);
bool check(size_t r, size_t c, int **M);
```

Se nel primo caso l'ordine dei parametri è obbligato (per la natura degli array dinamici), nel secondo caso è possibile riordinare i parametri. Si noti l'utilizzo di `size_t` anziché `int` per le dimensioni della matrice, poiché stiamo parlando di indici di array che non dovrebbero essere negativi.

Il primo punto cui prestare attenzione è la condizione di sparsità: calcolare (m * n) / 2, infatti, richiede di effettuare una moltiplicazione che potrebbe portare ad un overflow, per quanto estremamente improbabile. Un modo per verificare di non ricadere in questo caso è quello di "simulare" la moltiplicazione controllando che il risultato sia minore del massimo intero rappresentabile mediante un `size_t`.

Come anticipato, il resto della funzione si riduce a un doppio ciclo annidato, in cui si incrementa un contatore ogni volta che si incontra un intero di valore zero. Si possono tuttavia effettuare delle piccole ottimizzazioni.

La prima ottimizzazione riguarda il confrontro con il valore soglia. Infatti, se all'interno del ciclo si scrivesse:

```
if(zeros >= (r * c) / 2)
```

ci ritroveremmo a ricalcolare ad ogni iterazione la moltiplicazione e la divisione (assumendo che il compilatore non effettui per noi alcuna ottimizzazione, il che è possibile). Possiamo quindi precalcolare il valore soglia e confrontare ogni volta il numero di zeri con questo.

La seconda ottimizzazione può essere definita *early abort*. In particolare, se durante la scansione della matrice individuiamo un numero di zeri che supera il valore soglia, questa condizione sarà vera da quel momento in poi. Possiamo quindi decidere di effettuare ad ogni iterazione il confronto, per terminare la funzione prima di aver completato la scansione della matrice, il che può risultare positivo soprattutto qualora si incontrino matrici sparse molto grandi.
Si noti però che questa ottimizzazione potrebbe portare ad un peggioramento delle prestazioni qualora la probabilità di incontrare una matrice sparsa sia bassa. Non sapendo qual è il caso operativo dell'esercizio, entrambe le soluzioni sono considerate valide.

Si noti invece che un controllo (finale) di questo tipo:

```
if(zeros >= threshold)
	return true;
else
	return false;
```

è ridondante, poiché è ovvio, per il flusso di controllo dell'`if` e del `return`, che se non si esegue `return true` si eseguirà `return false`.

Mettendo insieme tutti i pezzi, una possibile soluzione è:

```
bool check(size_t r, size_t c, int m[r][c])
{
	unsigned int zeros = 0;
	int i, j, threshold;

	// Check for overflow
	if (r && c > (size_t)-1/r) {
		errno = E2BIG;
		return false;
	}
	threshold = (r * c) / 2;

	for(i = 0; i < r; i++) {
		for(j = 0; j < c; j++) {
			if(m[i][j] == 0) {
				if(++zeros >= threshold)
					return true;
			}
		}
	}

	return false;
}
```