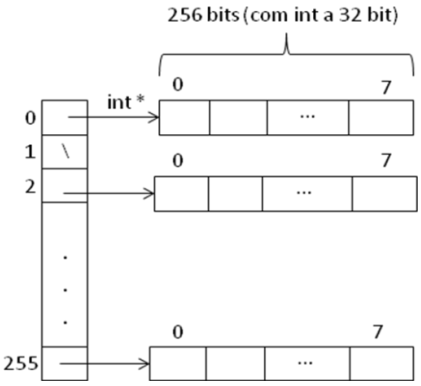
1. [9] Na implementação de um sistema de *queries* sobre uma base de dados, com um máximo de 64k itens, surgiu a necessidade de criar *bitmaps* para identificar o conjunto de elementos que satisfazem determinada pesquisa. Para além disso, os *bitmaps* facilitam operações lógicas (`and`, `or`) sobre conjuntos. Tendo-se verificado que o conjunto de itens encontrados é tipicamente muito pequeno em comparação com os 64k possíveis, optou-se por uma implementação de *bitmaps* esparsos, como mostra a fig. ao lado:



bitmap.h

```
#define NPARTS 256
#define PARTBITS 256
#define INTBITS (sizeof(unsigned int)*8)
#define PARTSIZE (PARTBITS/INTBITS)

typedef unsigned int *part_ptr_t;
typedef part_ptr_t *bitmap_ptr_t;

bitmap_ptr_t bm_create();
void bm_destroy(bitmap_ptr_t  bm);

void bm_set(bitmap_ptr_t bm, int bit);

bitmap_ptr_t bm_or(bitmap_ptr_t bm1,
                   bitmap_ptr_t bm2);

void bm_show(bitmap_ptr_t bm);
```

test.c

```
void test(bitmap_ptr_t bm) {
    bm_set(bm, 2);
    bm_set(bm, 257);
    bm_show(bm);
}

int main() {
    bitmap_ptr_t bm = bm_create();
    test(bm);
    bm_destroy(bm);
    return 0;
}
```

bitmap.c

```
/* Cria um bitmap vazio */
bitmap_ptr_t bm_create() { /* To do */ }

/* Destrói um bitmap previamente criado com bm_create */
void bm_destroy(bitmap_ptr_t  bm) { /* To do */ }

/* aloca uma (das 256) parte de bitmap */
static part_ptr_t bmpart_create() {
    int i;
    part_ptr_t bp = (part_ptr_t) malloc(sizeof(unsigned int)*PARTSIZE);
    for(i=0; i < PARTSIZE; ++i) bp[i] = 0;
    return bp;
}

/* faz set a um bit (0 a 64k-1) do bitmap */
void bm_set(bitmap_ptr_t bm, int bit) {
    int bm_index = bit >> 8, bit_mod = bit & (NPARTS-1),
        part_index = bit_mod / sizeof(unsigned int),
        int_index = bit_mod & (INTBITS-1);
    part_ptr_t bp = bm[bm_index];
    if (bp == NULL) bp=bmpart_create();
    bp[part_index] |= (1 << int_index);
    bm[bm_index]=bp;
}
/* criar um novo bitmap resultado do "or" entre bm1 e bm2 */
bitmap_ptr_t bm_or(bitmap_ptr_t bm1, bitmap_ptr_t bm2)  { /* To do */ }

/* mostra o índice dos bits a 1 no bitmap */
void bm_show(bitmap_ptr_t bm) { /* To do */  }
```

a) [1] Implemente em C as funções `bm_create` e `bm_destroy`.

b) [1] Implemente, em *assembly* IA-32, a função `test`, presente no ficheiro `test.c`.

c) [2] Implemente em C a função `bm_or`, presente no ficheiro `bitmap.c`, tendo o cuidado de optimizar a eficiência da solução.

d) [3] Implemente, em *assembly* IA-32, a função `bm_set`, presente no ficheiro `bitmap.c`.

e) [2] Implemente em C a função `bm_show`, presente no ficheiro `bitmap.c`, que mostra no *standard output* os índices de todos os bits a 1 no *bitmap* passado como argumento.

2. [2] Um sistema utiliza uma *cache 4-way set-associative*, com capacidade total de armazenamento de dados de 128KiB. O compilador utilizado define as mesmas dimensões para os tipos primitivos que o ambiente de referência da unidade curricular. Um programador definiu o tipo `struct DataItem` de modo a ter a dimensão exacta de uma linha de *cache*, cuja dimensão conhecia, sendo esse o único propósito do campo `padding`. Neste âmbito, responda às seguintes questões justificando devidamente as suas respostas.

```
struct DataItem {
  int   id;
  char  key;
  char  alt;
  short num_codes;
  int * codes;
  int   data[3];
  char  padding[8];
};
```

a) [1] Quantos *sets* tem a *cache* do sistema?

b) [1] Considere um *array* de `struct DataItem`, cujo endereço inicial é múltiplo de 1KiB. Que vantagem pode resultar da presença do campo `padding`? E que vantagem pode resultar de o retirar? Considere na sua resposta cenários de acesso sequencial e aleatório.

(continua)

3. [2] Considere um programa duvidoso, cujos únicos ficheiros fonte, `f1.c` e `f2.c`, são apresentados ao lado.

a) [0,5] Indique que símbolos existirão no ficheiro objecto resultante da compilação de `f1.c`. Para os símbolos definidos, indique também as respectivas secções.

b) [0,5] Indique que símbolos existirão no ficheiro objecto resultante da compilação de `f2.c`. Para os símbolos definidos, indique também as respectivas secções.

f1.c

```
#define VAL 0x215C215C

char * sum(char * p1,
           int   p2);

int value = VAL;

int main() {
  unsigned char res =
    proc(&value);
  printf("%X\n", res);
  puts(sum("135", 2));
  return 0;
}
```

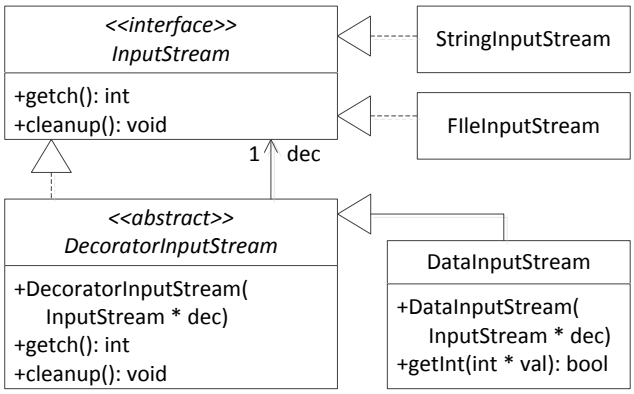
f2.c

```
typedef unsigned char uchar;

struct info {
  uchar isValid;
  uchar masked;
  short data;
};

static const uchar INV = -1;

uchar proc(struct info * pi) {
  return pi->isValid ?
    (pi->data | pi->masked) : INV;
}
```

c) [1] É possível ligar `f1.o` com `f2.o`? Se sim, o que acontece ao executar o programa resultante? Se não, que erros são reportados pelo *linker*?

4. [1,5] Considere um cenário em que se realiza o carregamento dinâmico de múltiplas bibliotecas (*shared objects*) utilizando *dlopen*. Considere ainda que uma das primeiras bibliotecas a ser carregada tem alguns símbolos coincidentes com os de outra carregada mais tarde. É garantido que o carregamento desta última biblioteca falha devido à repetição de símbolos? Se sim, que cuidados deve tomar o programador para evitar a situação? Se não, como lida o programador com os símbolos duplicados, se vier a precisar de ambos?

5. [5,5] Considere o diagrama de classes que define uma hierarquia de `InputStreams`. As classes `FileInputStream` e `StringInputStream` são concretizações de `InputStream` que retornam sucessivamente os *bytes* presentes no *stream* por cada chamada ao método `getch`. `FileInputStream` tem como fonte de dados um ficheiro enquanto `StringInputStream` tem como fonte uma *string*. O `cleanup` de um `FileInputstream` fecha o descritor para o ficheiro aberto na construção, enquanto `StringInputStream` liberta a memória alocada, também na construção, para alojar a sequência de caracteres. A classe `DecoratorInputStream` guarda uma referência para o `InputStream` que irá decorar, recebida no seu processo de construção (`dec`); a concretização de `getch` deverá retornar o *byte* retornado pelo `InputStream` decorado; no processo de construção não aloca recursos. A classe `DataInputStream` representa um exemplo concreto de um decorador que lê inteiros (`getInt`) de um `InputStream`. O primeiro *byte* lido do *stream* corresponde ao LSB do inteiro. O método `getInt` retorna `true` caso tenha lido um inteiro com sucesso; se for atingido o fim do *stream* durante a sua execução é retornado `false`. O código em C apresentado a seguir define a interface `InputStream`.




```
#include <stdio.h>

struct InputStreamMethods;
typedef struct InputStream {
  InputStreamMethods * vptr;
} InputStream;

typedef int (*IS_getc_t)(InputStream *);
typedef int (*IS_cleanup_t)(InputStream *);
```

```
typedef struct InputStreamMethods {
  /* leitura de um char da stream
   *   retorna -1 se não há mais caracteres
   */
  IS_getc_t getc;
  /* liberta os recursos internos da stream */
  IS_cleanup_t cleanup;
} InputStreamMethods;
```

a) [4,0] Implemente os construtores e defina os tipos, variáveis e funções de `DecoratorInputStream` e `DataInputStream` que permitam uma utilização de `DataInputStream`.

b) [1,5] Utilizando o tipo `DataInputStream`, escreva o programa que apresenta na consola os primeiros 4 inteiros presentes no ficheiro de nome `teste.dat`.

Duração: 2 horas e 30 minutos

Bom teste!