Algoritmos e Complexidade – 6 de Janeiro de 2017 – Duração: 90 min

Parte A

1. Na implementação de tabelas de hashing com *open addressing* muitas vezes guarda-se para cada chave inserida o número de colisões que essa inserção teve que resolver (`probe == 0` significa que não houve qualquer colisão).
   Considere a definição ao lado utilizada para implementar essas tabelas e apresente o resultado de inserir as chaves `40, 80, 60, 260, 54, 65, 140` por esta ordem, numa tabela inicialmente vazia usando linear probing.
   Assuma que a tabela tem tamanho `Hsize=13` e que a função de hash é apenas `hash (t) = t % Hsize`.

```
#define Hsize ...
#define FREE    0
#define USED    1
#define DELETED 2
typedef struct entry {
  int key;
  int probe;
  int status;
} Entry;
typedef Entry THash [Hsize];
```

2. Relembre a definição de árvores AVL (de inteiros) apresentado ao lado e o algoritmo de inserção balanceada estudado.

   (a) Defina uma função `AVL maisProfundo (AVL a)` que, dada uma árvore AVL, determina um nodo da árvore que se encontra à profundidade máxima. A função retorna `NULL` no caso da árvore ser vazia e deverá executar em tempo logoraítmico no número de nodos da árvore.

   (b) Qual o resultado da função que definiu quando for aplicada à árvore que resulta de se inserirem, numa árvore inicialmente vazia, os valores `20, 40, 10, 50, 30, 15, 29` (por esta ordem)? Justifique.

```
#define LEFT   1
#define BAL    0
#define RIGHT -1
typedef struct avl {
  int value;
  int bal;
  struct avl *left, *right;
} *AVL;
```

3. Considere as definições ao lado para representar *grafos não-orientados e ligados*.

   (a) Complete a definição da função `bicolor` que tenta colorir os vértices de um grafo com apenas duas cores (`RED` e `GREEN`) por forma a que *vértices adjacentes tenham cores diferentes* (assume-se que **o grafo é ligado**). A função deve devolver 1 ou 0 consoante isso seja ou não possível.

   Deve implementar uma *travessia em largura*, tentando pintar os vértices por níveis alternando cores (vértice inicial vermelho, adjacentes do inicial verdes, adjacentes dos adjacentes vermelhos, e assim sucessivamente).

   (b) Analise o tempo de execução da função no melhor e no pior caso e descreva as situações em que esses casos ocorrem.

```
#define N ...
typedef struct edge {
  int dest;
  struct edge *next;
} *Adjlist;
typedef AdjList Graph [N];

#define WHITE 0
#define RED   1
#define GREEN 2

int bicolor(Graph g, int color[]) {
  Queue q; init_queue(q);
  for (v=0; v<N; v++)
    { color[v] = WHITE; }
  enqueue(q, 0); color[0] = RED;
  while (!is_empty(q) && ...) {
    v = dequeue(q);
    ...
  }
  return ...
}
```

Parte B

Recorde o problema de coloração de grafos, que consiste em decidir se é ou não possível colorir um grafo não-orientado com k (ou menos) cores. A função do exercício 3 resolve o problema de forma eficiente para $k = 2$, mas para k arbitrário o problema é NP-completo.

1. Considerando de novo as declarações do exercício 3, apresente um algoritmo não-determinístico para este problema. Para isso:

   - Diga como codificaria as soluções propostas.
   - Apresente uma implementação em C da parte determinística do algoritmo.

2. Apresente uma implementação em C de uma função determinística `int k-color (Graph g, int k)` para resolver este problema (por procura exaustiva) baseada no algoritmo não determinístico definido atrás. Para isso deverá gerar todas as soluções possíveis até que seja encontrada uma solução válida ou que sejam esgotadas todas as possibilidades.

3. Assumindo que a função de teste apresentada é linear no tamanho do grafo, qual a complexidade da função apresentada na alínea anterior no melhor e pior casos?