Algoritmos e Complexidade

Exame da Época Especial

10 de Setembro de 2013

Parte I

1. Apresente as condições de verificação necessárias à prova da correcção parcial do seguinte programa anotado para calcular a raíz quadrada inteira de um número.

```
// n == n0 > 0
r = 1;
// n == n0 >= 0 && r == 1
while (r*r < n)
    // n == n0 >= 0 && (r-1)*(r-1) <= n
    r = r+1;
// n == n0 >= 0 && (r-1)*(r-1) <= n && r * r >= n
if (r * r > n) r = r-1;
// r*r <= n0 && (r+1)*(r+1) > n0
```

2. Assumindo que as operações elementares sobre inteiros (multiplicação, adição e subtracção) executam em tempo constante, faça a análise da complexidade do código acima para o pior caso (não se esqueça de caracterizar esse pior caso), em função do número de bits usados na representação do número `n`.

3. Considere uma tabela de *Hash*, com *open addressing* e tratamento de colisões por *linear probing*. Considere ainda que, para implementar remoções, cada célula da tabela tem uma *flag* que pode tomar três valores possíveis: `L`, `O` ou `A` (Livre, Ocupada ou Apagada).

   Assumindo que as chaves são inteiros e que a função de *hash* usada é `hash(n) = n%N`, considere o seguinte estado da tabela (para `N=11`).

0	1	2	3	4	5	6	7	8	9	10
12 L	78 O	34 L	45 L	15 O	37 A	28 O	73 O	95 O	49 O	98 L

   Partindo do princípio que a função `procura`, recebe o valor de uma chave e retorna a posição da tabela onde essa chave se encontra (e `-1` caso a chave não se encontre na tabela), qual o resultado de fazer, no estado apresentado, `procura(12)` e `procura(37)`. Para cada um dos casos, indique quais as posições da tabela que são consultadas.

4. Considere a seguinte definição para implementar uma árvore AVL de inteiros.

```
#define Bal 0
#define Esq 1
#define Dir (-1)
```

```
typedef struct avlnode {
int valor;
int balanco;
struct avlnode *esq, *dir;
} *AVL;
```

Defina uma função que calcula a altura de uma árvore em tempo logarítmico em relação ao número de nodos da árvore. Justifique brevemente que a sua função tem de facto complexidade logarítmica.

5. Considere um grafo com `101` vértices, numerados de 0 até 100, e onde existe uma aresta de `x` para `y` se e só se `x` é um divisor de `y`. Assuma que os sucessores de cada nodo se encontram armazenados **por ordem crescente**. Por exemplo, os sucessores do vértice 9 serão

   `0, 9, 18, 27, 36, 45, 54, 63, 72, 81, 90, 99`

   por esta ordem.

   Indique quais, e por que ordem, os vértices visitados ao efectuar uma travessia em profundidade a partir do vértice `10`.

Parte II

1. Considere a seguinte alternativa para o problema apresentado na alínea 1.

```
// n == n0 > 0
r = 0 ; s = n;
// ......
while (r < s-1) {
    // .....
    m = (r+s)/2;
    // ....
    if (m*m > n) s = m;
    else r = m;
}
// r*r <= n0 && (r+1)*(r+1) > n0
```

   (a) Complete as anotações do código acima de forma a provar a sua correcção **parcial**.

   (b) Faça a análise da complexidade deste código para o pior caso em função do número de bits usados na representação do número `n`.

2. Considere que, numa tabela de hash com open addressing, se decide efectuar uma *garbage collection* sempre que o número de células apagadas é igual ao número de células activas. Assuma, como simplificação, que esta operação é linear no número de células activas da tabela.

   Assumindo que as operações de inserção e remoção executam em tempo constante, mostre que ao incluir esta operação de remoção com *garbage collection*, as operações continuam a ter custo amortizado constante. Aplique um método à sua escolha de análise amortizada.