Algoritmos e Complexidade – 26 de Janeiro de 2017 – Duração: 2h

Parte A

1. Considere a função ao lado que coloca em `r` o índice do menor elemento do vector `v`.
   Anote convenientemente o programa e calcule as condições de verificação resultantes.

```
// N > 0
r = 0; i = 1;
while (i < N) {
  if (v[i] < v[r]) r = i;
  i = i+1;
}
// forall (0<=k<N) v[k] >= v[r]
```

2. Considere a seguinte definição de uma função que calcula o número de bits a `1` na representação de um número inteiro (*hamming weight*).
   Identifique o melhor e pior casos do custo da execução (número de iterações do ciclo `while`) desta função em termos do tamanho (número de bits usados) da representação dos números inteiros.
   Note que para um tamanho N fixo do input, a gama de valores possíveis para o input vai de 0 (todos os bits a `0`) até $2^N - 1$ (todos os bits a `1`).

```
int hamming (unsigned int x){
  int r=0;
  while (x!=0) {
    if (x%2 == 1) r++;
    x=x/2;
  }
  return r;
}
```

3. Considere a definição de tipos à direita para representar tabelas de Hash de inteiros usando open addressing com vectores dinâmicos e tratamento de colisões por linear probing. Considere ainda que

   - a função de hash usada é `hash(x) = x%tamanho`
   - sempre que o número de células livres é menor do que 25% do tamanho da tabela esta é realocada para uma tabela de tamanho `2*tamanho+1`.
   - sempre que o número de células apagadas é maior ou igual ao número de células ocupadas é feita uma *garbage collection*.

```
struct celula {
  int k;
  char estado; //L/O/A
}
typedef shash {
  int tamanho,
      ocupados,
      apagados;
  struct celula *Tabela;
} *THash;
```

   Apresente a evolução de uma tabela, inicialmente vazia e de tamanho (inicial) `7`, quando são efectuadas as seguintes operações de inserção (`ins`) e remoção (`del`) (por esta ordem).

   `ins 10`, `ins 4`, `ins 16`, `del 10`, `del 4`, `ins 9`, `ins 13`, `ins 0` `ins 7`

4. Considere as definições (apresentadas à direita) de listas ligadas e de árvores AVL em que cada nodo possui ainda uma marca para assinalar nodos apagados.
   Defina uma função `AVL fromList (LInt l, int n)` que constrói uma árvore AVL a partir de uma lista **ordenada** com `n` elementos.
   Não se esqueça de garantir que (1) a árvore produzida está balanceada, (2) os factores de balanço estão correctamente calculados e (3) a função executa em tempo linear no número de elementos da lista.

```
typedef struct llint {
  int value;
  struct llint *next;
} *LInt;

typedef struct avl {
  int value;
  int bal, deleted;
  struct avl *left, *right;
} *AVL;
```

5. Considere as definições para representar *grafos não-orientados*.
   Defina uma função `maxcomp` que calcula o número de vértices do *maior* componente ligado do grafo `g` (i.e., o componente que contém mais vértices).

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

typedef AdjList Graph [N];
```

6. Suponha que um dado grafo `g` tem 15 elementos (`#define NV 15`) e que se executa o seguinte extracto de código:

```
int i; int pesos [NV], int pais [NV];
for (i=0;i<NV;pais [i++] = -2);

dijkstraSP (g,3,pais,pesos);
```

   Depois disso os arrays `pais` e `pesos` têm o seguinte conteúdo:

   - `pais =`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
5	3	3	-1	-2	2	14	6	-2	1	1	-2	-2	-2	3

   - `pesos =`

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
20	4	3	0	2	9	8	13	20	5	8	13	5	42	2

   Indique, justificando, quais das seguintes afirmações são verdadeiras e quais são falsas. Considere que a distância entre dois vértices é o peso (soma dos pesos das arestas) do caminho mais curto que liga o primeiro ao segundo. *Sugestão: comece por desenhar a árvore calculada pela função* ***`dijkstraSP`***.

   (a) os vértices `0` e `8` estão à mesma distância do vértice `3`.

   (b) o vértice mais distante do vértice `3` é o vértice `0`.

   (c) a distância do vértice `3` a `1` é igual à distância do vértice `1` ao vértice `10`.

   (d) existe no grafo uma aresta que liga o vértice `9` ao vértice `5` com peso `3`.

Parte B

1. Uma forma alternativa de permitir remoções em árvores AVL é marcar os nodos apagados com uma flag (tal como exemplificado no exercício 4). Esta solução tem complexidade logarítmica no número de nodos da árvore, e é por isso aceitável desde que o número de nodos apagados não seja muito grande (se o número de apagados corresponder a metade dos elementos, a altura da árvore é sensivelmente igual à altura da árvore sem os apagados).

   Considere que existe definida uma função `int inorder (AVL a, LInt *l)` que coloca em `*l` o resultado de uma travessia inorder da árvore `a`, retornando o comprimento da lista produzida. Essa função não inclui na lista os nodos marcados como apagados.

   Assuma que esta função executa em tempo linear no número de elementos da árvore.

   Considere agora que sempre que o número de apagados ultrapassa o número de não apagados, se faz uma limpeza da árvore e que consiste em gerar uma lista dos elementos da árvore para depois gerar uma árvore a partir desta lista (funções `inorder` e `fromList` da alínea 4). Assuma que essa operação executa em tempo linear no tamanho da árvore.

   Mostre, usando um dos métodos de análise amortizada estudados, que esta operação de remoção (que inclui eventualmente a operação de limpeza) tem um custo amortizado logarítmico.

2. Relembre a função `hamming` apresentada na questão 2.

   (a) Assumindo uma amostra aleatória (em cada posição da representação do número pode estar, com igual probabilidade, `0` ou `1`), diga qual a probabilidade de acontecer cada um dos casos identificados na questão 2.

   (b) Calcule o custo médio da execução (número de iterações do ciclo `while`) dessa função.