Algoritmos e Complexidade

2º Teste

17 de Janeiro de 2013 – Duração: 90 min

Parte I

1. Considere uma tabela de *hash* (para implementar um conjunto de inteiros) com tratamento de colisões por *open addressing* (com *linear probing*) e em que a remoção de chaves é feita usando uma marca (de apagado).

   Suponha que existem definidas as funções `add (int k)`, `remove (int k)`, e `exists (int k)`, sendo que esta última devolve verdadeiro/falso.

   Suponha ainda que o tamanho da tabela é 7 e que a função de *hash* é `hash(x) = x % 7`. Apresente a evolução da tabela quando, a partir de uma tabela inicialmente vazia, se executa a seguinte sequência de operações:

   ```
   add 15; add 25; add 9; add 1; rem 9; add 38; rem 15; add 6; add 10;
   ```

2. Considere os seguintes tipos de dados para a representação de grafos (sem pesos) por listas de adjacências.

   ```
   struct edge {
     int dest;
     struct edge *next;
   }
   typedef struct edge *Grafo[MaxV];
   ```

   Usando uma travessia, defina uma função `int succN (Grafo g, int v, int N)` que, dado um grafo `g`, um vértice `v` e um inteiro `N`, determina quantos vértices em `g` estão a uma distância de `v` menor ou igual a `N`, i.e., para os quais existe um caminho com `N` ou menos arestas.

3. Considere as definições ao lado para implementar árvores AVL de inteiros.
   Defina uma função `int altura (AVL a)` que calcula a altura de uma AVL em tempo logarítmico no tamanho (número de nodos) da árvore.

   ```
   #define Bal 0  // Balanceada
   #define Esq -1 // Esq mais pesada
   #define Dir 1  // Dir mais pesada

   typedef struct avlNode *AVL;

   struct avlNode {
       int bal; // Bal/Esq/Dir
       int valor;
       struct avlNode *esq,*dir;
   }
   ```

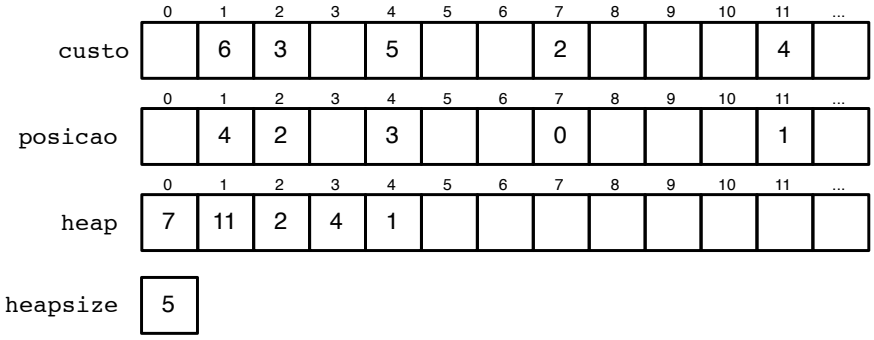
Parte II

1. Em alguns algoritmos sobre grafos (Prim, Dijkstra) uma optimização possível consiste em implementar a orla com uma fila de prioridades. Uma forma de se implementar essa estrututa consiste em organizar os vértices numa *min-heap* ordenada pelo custo de cada vértice. Contudo, como o custo de cada vértice pode mudar, precisamos ainda de ter acesso à posição de cada vértice na heap, e implementar operações de actualização do custo de um vértice, com a consequente mudança da posição desse vértice na *heap*.

   Considere então as seguintes definições para implementar uma fila de prioridades dos vértices de um grafo:

```
int custo[MaxV];    // custo de cada vértice
int posicao[MaxV];  // posição na heap
int heap[MaxV];
int heapsize;       // número de elementos na heap;
```

   Por exemplo, se a orla for composta pelos vértices 1, 2, 4, 7 e 11, com custos (respectivamente) 6, 3, 5, 2 e 4, uma possível configuração destas variáveis seria:



   Defina a função `void updateCusto (int v, int c)` que altera o custo do vértice `v` para um custo menor `c`.

2. *PageRank é uma família de algoritmos de análise de rede que dá pesos numéricos a cada elemento de uma colecção de documentos hiperligados, como as páginas da Internet, com o propósito de medir a sua importância nesse grupo por meio de um motor de busca.* (in Wikipedia).

   A rede poder ser vista simplesmente como um grafo não-orientado, sem pesos. O cálculo do *page rank* de cada vértice é feito por iterações progressivamente mais exactas. Para cada vértice u num grafo com N vértices tem-se então:

$$\begin{aligned} pr(u,0) &= 1/N \\ pr(u,t+1) &= \sum_{v \in Adj(u)} \frac{pr(v,t)}{grau(v)} \end{aligned}$$

   (a) Escreva em C uma função `updatePRank(Grafo g, float pK[], float pK1[])` que, dado um grafo `g` representado por uma matriz de adjacências e uma vector `pK` com o pagerank dos vértices na iteração k, preenche o vevtor `pK1` com o pagerank na iteração $k+1$.

   (b) Escreva em C uma função `void pageranks(Grafo g, int k, float p[])` que dado um grafo representado por uma matriz de adjacências e um inteiro k, calcula o o pagerank dos vértices na iteração k.