Teste 21 Dezembro 2019

Algoritmos e Complexidade

Universidade do Minho

Questão 1 [3 valores]
Considere uma estrutura de dados *min-heap* implementada sobre um *array* dinâmico, com comprimento inicial igual a 1. Quando completamente preenchido, o array é realocado com o dobro do tamanho. Apresente todos os estados do array, incluindo o comprimento alocado em cada estado, para a sequência de operações seguinte:

```
Insert 30; Insert 20; Insert 10; Insert 100; Insert 90; Insert 80; ExtractMin; ExtractMin; Insert 40;Insert 50; Insert 60; Insert 20; Insert 10; ExtractMin; ExtractMin
```

Resolução:
[**30**]
[30, **20**] →(BU) [20, 30]
[20, 30, **10**, -] → (BU) [10, 30, 20, -]
[10, 30, 20, **100**]
[10, 30, 20, 100, **90**, -, -, -]
[10, 30, 20, 100, 90, **80**, -, -]
[**80**, 30, 20, 100, 90, -, -, -] → (BD) [20, 30, 80, 100, 90, -, -, -]
[**90**, 30, 80, 100, -, -, -, -] → (BD) [30, 90, 80, 100, -, -, -, -]
[30, 90, 80, 100, **40**, -, -, -] → (BU) [30, 40, 80, 100, 90, -, -, -]
[30, 40, 80, 100, 90, **50**, -, -] → (BU) [30, 40, 50, 100, 90, 80, -, -]
[30, 40, 50, 100, 90, 80, **60**, -]
[30, 40, 50, 100, 90, 80, 60, **20**] → (BU) [20, 30, 50, 40, 90, 80, 60, 100]
[20, 30, 50, 40, 90, 80, 60, 100, **10**, -, -, -, -, -, -, -] → (BU) [10, 20, 50, 30, 90, 80, 60, 100, 40, -, -, -, -, -, -, -]
[**40**, 20, 50, 30, 90, 80, 60, 100, -, -, -, -, -, -, -, -] → (BD) [20, 30, 50, 40, 90, 80, 60,

100, -, -, -, -, -, -, -, -]
[**100**, 30, 50, 40, 90, 80, 60, -, -, -, -, -, -, -, -, -] → (BD) [30, 40, 50, 100, 90, 80, 60, -, -, -, -, -, -, -, -, -]

Questão 2 [3 valores]
Simule a evolução de uma árvore AVL, inicialmente vazia, ao longo da seguinte sequência de inserções. Identifique claramente todos os pontos em que o invariante é violado e a forma como é reposto.

```
Insert 10; Insert 20; Insert 30; Insert 100; Insert 90; Insert 80; Insert 40; Insert 50;
```

Resolução:

```
  10

  10
    \
    20

  10
    \
    20
      \
      30
invariante violado no nó 10; rotação simples à esquerda
      20
     /  \
    10  30

      20
     /  \
    10  30
          \
```

```
         100

      20
     /  \
    10  30
          \
          100
         /
        90
invariante violado no nó 30; rotação dupla
      20
     /  \
    10  30
          \
          90
            \
           100
      20
     /  \
    10  90
       /  \
      30  100

      20
     /  \
    10  90
       /  \
      30  100
        \
        80
invariante violado no nó 20; rotação dupla
      20
```

```
     /  \
    10  30
          \
          90
         /  \
        80  100
      30
     /  \
   20    90
  /     /  \
 10    80  100

      30
     /  \
   20    90
  /     /  \
 10    80  100
      /
     40

      30
     /  \
   20    90
  /     /  \
 10    80  100
      /
     40
       \
       50
invariante violado no nó 80; rotação dupla
      30
     /  \
```

```
  20    90
 /     /  \
10    80  100
     /
    50
   /
  40
     30
    /  \
  20    90
 /     /  \
10    50  100
     /  \
    40  80
```

Questão 3 [4 valores]
Pretende-se desenhar uma estrutura de dados para a implementação de *conjuntos de números naturais*, suportando as seguintes operações:

- Inserção
- Teste (dado um inteiro, testar se pertence / não pertence ao conjunto)
- *Rank* (dado um inteiro, contar o número de elementos do conjunto de valor inferior ou igual a ele).

Proponha uma implementação eficiente desta estrutura de dados e (sem escrever código) analise o tempo de execução de cada uma das operações, justificando e referindo assunções adicionais da sua análise.

Resolução:
Uma árvore AVL poderia ser usada, o que resultaria em tempos $\Theta(\log N)$ para a inserção, $\Omega(1), \mathcal{O}(\log N)$ para o teste, e $\Omega(\log N), \mathcal{O}(N)$ para o *rank*. Esta última operação obriga a uma contagem do número de elementos inferiores ao dado como parâmetro, ocorrendo o pior caso quando este é o maior elemento do conjunto. O melhor caso ocorrer para qualquer elemento guardado em nós do caminho descendente mais à esquerda da árvore.

Outra possibilidade seria a utilização de uma tabela de *hash.* Assumindo-se uma implementação que permita manter o tempo de inserção e consulta tendencialmente constante, as duas primeiras operações executariam em tempo $\Theta(1)$. O custo a pagar está na operação de *rank*, que obriga a percorrer todas as chaves da tabela, em tempo $\Theta(N)$.

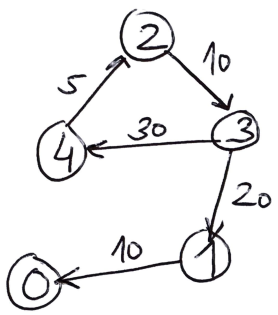
Finalmente, uma solução simples seria a utilização de um *array ordenado*, com inserção em tempo $\Omega(1), \mathcal{O}(N)$, teste (por pesquisa binária) em tempo $\Omega(1), \mathcal{O}(\log N)$, e *rank* também em tempo $\Omega(1), \mathcal{O}(\log N)$, uma vez que depois de encontrado o elemento em questão, a posição em que ele se encontra no *array* corresponde ao seu *rank*.

Dois comentários: (i) é possível melhorar estas implementações incluindo-se informação adicional na estrutura de dados; (ii) a escolha da estrutura a utilizar deveria ser feita em função do padrão esperado para as operações mais frequentes a efectuar sobre a estrutura. Por exemplo, se for expectável que o cálculo de *rank* seja feito raramente, então a tabela de *hash* será a melhor escolha. Mas se for a operação mais frequente, será preferível a utilização de um *array* ordenado.

Nas questões que se seguem considere os seguintes tipos de dados para a representação de grafos por matrizes e por listas de adjacências:

```
typedef int WEIGHT;

#define NE -1
  // utilizado na repr. por matrizes para identificar arestas inexistentes

typedef WEIGHT GraphM[MAX][MAX];

struct edge {
  int dest;
  WEIGHT weight;
```

```c
  struct edge *next;
};


typedef struct edge *GraphL[MAX];
```



Questão 4 [4 valores]

Defina em C a função `int pesoC (GraphM g, int V[], int k)` que calcula o custo do caminho do grafo g constituído por k vértices armazenados no *array* V. A função deverá devolver -1 caso a sequência V não corresponda a um caminho no grafo.

Por exemplo no grafo ao lado, o array V = {2, 3, 1} com k=3, corresponde ao caminho constituído pelas arestas (2, 3) e (3, 1), e o peso calculado deverá ser 10+20 = 30.

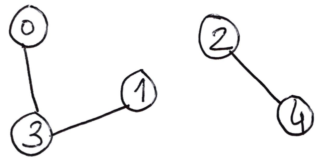
Note que o grafo é representado por uma **matriz de adjacências!**

Resolução:

```c
int pesoC (GraphM g, int V[], int k)
{
  int i, r=0;
  for (i=0; i<k-1; i++) {
    a = v[i]; b = v[i+1];
    if (g[a][b] != NE) r += g[a][b];
    else return -1;
  }
  return r;
}
```

Questão 5 [6 valores]

(i) Pretende-se etiquetar os vértices de um **grafo não-orientado** com um número que identifique o **componente ligado** a que pertence. Escreva a função `void componentes(GraphL g, int n, int comp[])` que coloca esta informação no *array* comp. Por exemplo se num grafo com 5 vértices tivermos os vértices 0, 1, 3 num componente e 2, 4 num outro, no final da execução teremos comp[0] = comp[1] = comp[3] = 0, e comp[2] = comp[4] = 1.



(ii) Analise o tempo de execução da função.

```
void componentes(GraphL g, int n, int comp[])
{
  int i, c=0;
    // c = índice dos componentes, começando em 0
  for (i=0; i<n; i++) comp[i] = -1
    // este array servirá também para controlar a travessia
    // dispensando um array de cores ou 'visitados'
  for (i=0; i<n; i++)
    if (comp[i] == -1) {
      df(g, i, comp, c);
      c++;
    }
}

void df(GraphL g, int o, int comp[], int c)   // travessia em profundidade; poderia ser em largura
{
  struct edge *p;
  comp[o] = c;
  for (p=g[o]; p; p=p->next)
    if (comp[p->dest] == -1)
      df(g, p->dest, comp, c)
}
```

Tempo de execução: $T(V, E) = \Theta(V + E)$, como em qualquer travessia completa de um grafo.