Teste 11 de Janeiro 2022

Algoritmos e Complexidade

Universidade do Minho

Questão 1 [4 valores]
Considere uma tabela de *hash* implementada por *open addressing* e *linear probing* sobre um array dinâmico cujo comprimento é um número primo N, utilizando-se a habitual função de hash $h(x) = x\%N$. O array é realocado quando fica totalmente preenchido e se faz nova inserção, passando a ter como tamanho o número primo imediatamente superior ao dobro do tamanho actual. Ao realocar é necessário inserir todos os elementos na nova tabela, com a função de hash actualizada. Considere que se utiliza a chave especial `EMPTY` para sinalizar as posições inicialmente vazias.

Simule a evolução de uma tabela de comprimento inicial 3 ao longo da sequência de inserção dos seguintes números, mostrando claramente o seu tamanho e conteúdo em cada passo:

```
10   17   25   27   38
```

Resolução:

```
0 | EMPTY |
1 | EMPTY |
2 | EMPTY |

0 | EMPTY |
1 |  10   |
2 | EMPTY |

```

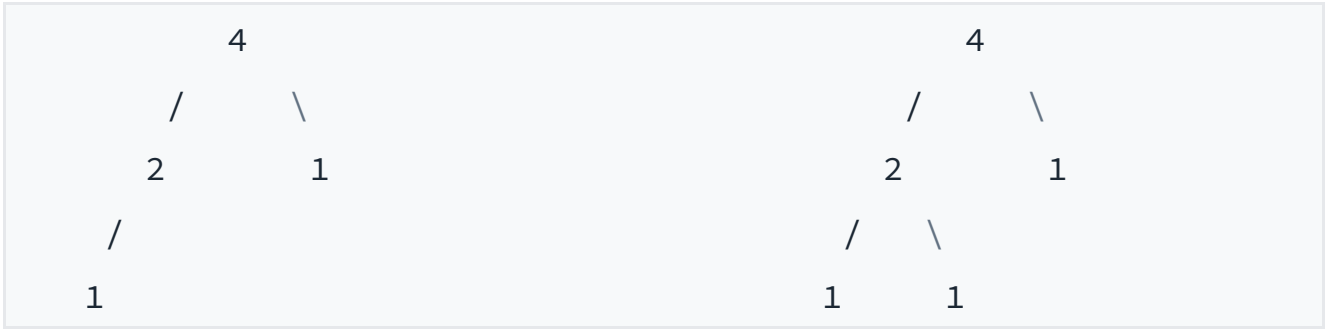
```
0 | EMPTY |
1 |  10   |
2 |  17   |

(colisão pos. 1)
0 |  25   |
1 |  10   |
2 |  17   |

(realocação com tamanho 7 e reinserção de 25; 10; 17 (colisão na pos. 3))
0 | EMPTY |
1 | EMPTY |
2 | EMPTY |
3 |  10   |
4 |  25   |
5 |  17   |
6 | EMPTY |

0 | EMPTY |
1 | EMPTY |
2 | EMPTY |
3 |  10   |
4 |  25   |
5 |  17   |
6 |  27   |

(colisão pos. 3)
0 |  38   |
1 | EMPTY |
2 | EMPTY |
3 |  10   |
```

```
4 |  25  |
5 |  17  |
6 |  27  |
```

Questão 2 [5 valores]
Considere uma noção de árvore binária em que os nós contêm um campo adicional `weight` em que se armazena o **peso** de cada nó. O peso de um nó N define-se como o número total de nós contidos na árvore que tem N como raíz.

Escreva uma função `int checkWeight(Btree t)` que verifica se os pesos de uma árvore se encontram correctamente preenchidos (devolvendo 1 caso estejam, e 0 caso não estejam).
Analise depois o tempo de execução da função.

Por exemplo, nas figura seguinte, os pesos estão bem preenchidos na árvore da esquerda, mas não na da direita (os número nos nós correspondem ao campo `weight`, omitindo-se outra informação.



Assuma o seguinte tipo de dados para as árvores:

```
struct btree {
  int info;
  int weight;
  struct btree *left, *right;
};
typedef struct btree *Btree;
```

(obs: para quem estiver autorizado a codificar em Python, sugere-se a seguinte representação simples para as árvores binárias: `None` para a árvore vazia, e um tuplo `(info, wegth, esq, dir)` para um nó intermédio — e.g. a árvore da esquerda seria representada como `(10, 4, (5, 2, (3, 1, None, None), None), (22, 1, None, None))`)

Resolução:

```
int checkWeight(Btree t) {
  int p = 1;
  if (t == NULL) return 1 ;
  if (t->left != NULL) p += t->left->weight ;
  if (t->right != NULL) p += t->right->weight ;
  return (p == t->weight && checkWeight (t->left) && checkWeight (t->right)) ;
}
```

$T(N) = \Omega(1)\ ,\ \mathcal{O}(N)$

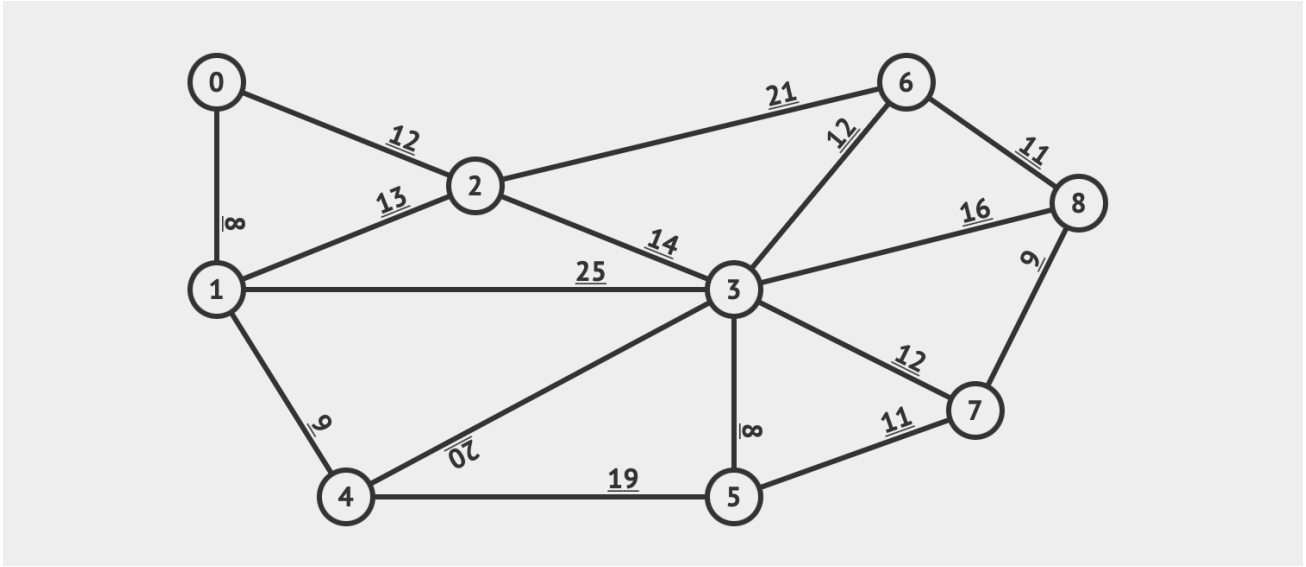
As operações executadas em cada chamada da função, excluindo-se as chamadas recursivas, executam em tempo constante.

O melhor caso ocorre quando falha a comparação `p == t->weight` envolvendo os pesos guardados na raíz e nos seus descendentes imediatos. Não são efectuadas quaisquer chamadas recursivas.

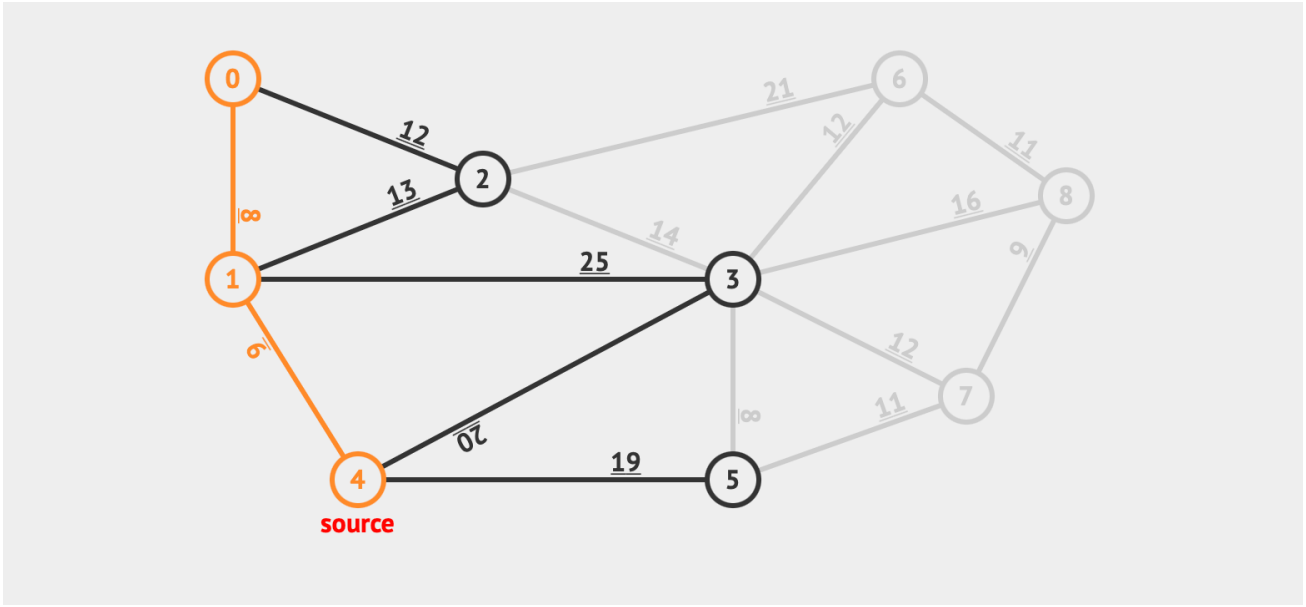
O pior caso ocorre quando nenhuma dessas comparações falha, sendo a função invocada para todos os nós da árvore.

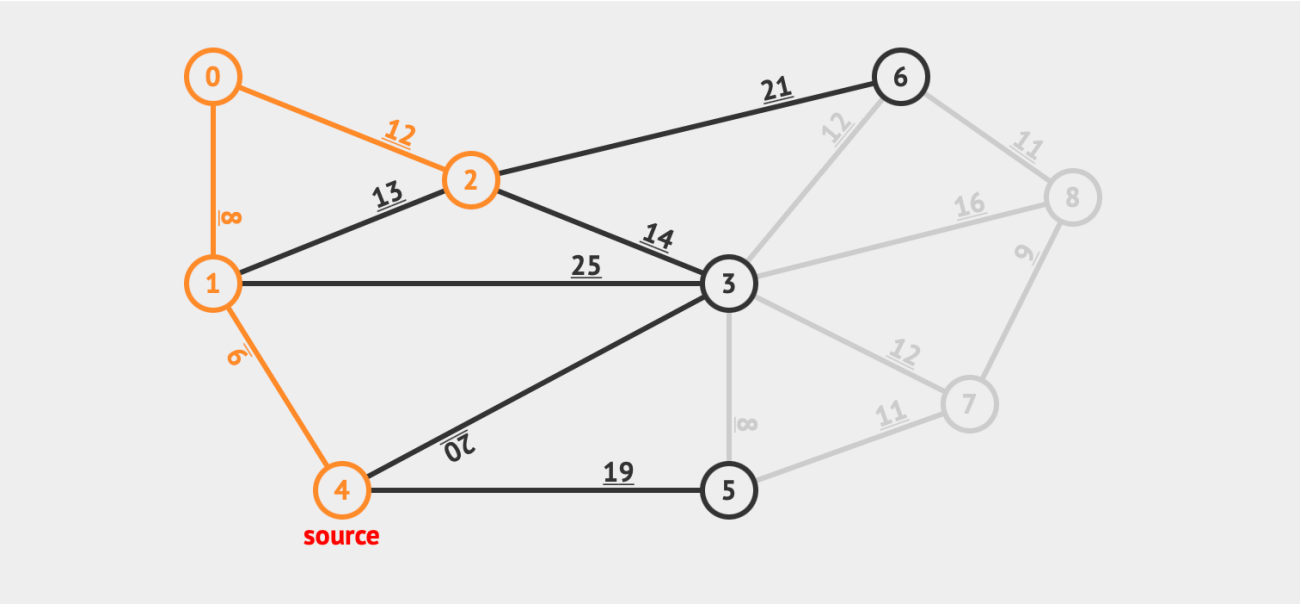
Questão 3 [4 valores]
Aplique o algoritmo de Prim para calcular uma árvore geradora mínima do seguinte grafo não-orientado, tomando o vértice 4 como inicial, e:

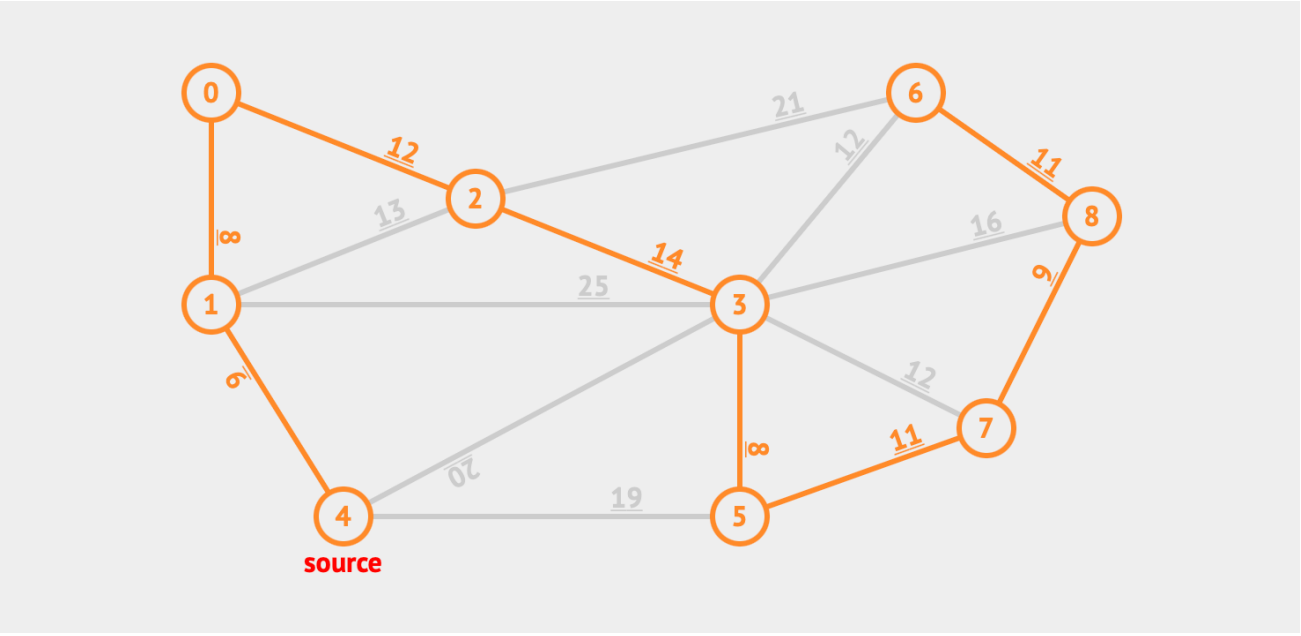
1. desenhe a árvore intermédia contendo apenas os 3 primeiros vértices, e explique o passo seguinte de expansão da árvore
2. desenhe a árvore geradora mínima (final).


0
1
2
3
4
5
6
7
8
12
21
12
11
8
13
16
14
9
25
12
9
20
8
11
19

Resolução:


0
1
2
3
4
5
6
7
8
12
21
12
11
8
13
16
14
9
25
12
9
20
8
11
19
source





Questão 4 [5 valores]
Considere os seguintes tipos de dados para a representação de grafos orientados, sem pesos, por listas de adjacências:

```
struct edge {
  int dest;
  struct edge *next;
};
typedef struct edge *GraphL[MAX];
```

Defina em C a função `int shortestBy (GraphL g, int s, int d, int i)` que recebe um grafo orientado sem pesos e três vértices `s`, `d`, `i`, e :

- devolve -1 caso `d` não seja alcançável a partir de `s`
- devolve 0 caso nenhum caminho mais curto de `s` até `d` contenha o vértice `i`
- devolve 1 caso um caminho mais curto de `s` até `d` passe pelo vértice `i`

Resolução:
Se os caminho mais curto entre um par de vértices fosse único, esta questão poderia ser resolvida calculando esse caminho (fazendo uma travessia em largura), e verificando depois, caso ele existisse, se continha ou não `i`. O problema é que não sendo único, o facto de o caminho calculado pelo algoritmo não conter `i` não significaria que não existisse um outro, com o mesmo peso, que contivesse `i`.

Existe no então uma solução simples para o problema, baseada um duas travessias: calculamos as distâncias entre `s` e `i` e entre `s` e `d` (fazendo uma travessia com origem em `s`), e a distância entre `i` e `d` (fazendo uma segunda travessia, com origem em `i`).

Existe um caminho mais curto entre `s` e `d` que contém `i` se e só se $D(s,d) = D(s,i) + D(i,d)$,

Por exemplo recorrendo às funções da Ficha 4:

```
int shortestBy (GraphL g, int s, int d, int i)  {
  int v[MAX], p[MAX], ls[MAX, li[MAX] ;
  BF (g, s, v, p, ls) ;
  BF (g, i, v, p, li) ;
  return (ls[d] == ls[i] + li[d]) ;
}
```

Note-se que esta não é a solução mais eficiente, uma vez que as travessias efectuadas serão exaustivas, e poderiam ser interrompidas quando o vértice `d` é alcançado. Uma solução mais eficiente (com menor tempo de melhor caso) passaria por implementar directamente as travessias, interrompendo-as quando fosse alcançado o vértice relevante.

Questão 5 [2 valores]
Seja α uma constante tal que $0.5 \leq \alpha < 1$ e relembre o conceito de peso de um nó da Questão 2.

Uma árvore binária de procura diz-se $\alpha-$balanceada se em toda a sub-árvore, os rácios dos números dos seus nós da contidos à sua esquerda e à sua direita são ambos $\leq \alpha$.
Por outras palavras, para todo o nó X, $\frac{\text{peso}(X.esq)}{\text{peso}(X)} \leq \alpha$ e $\frac{\text{peso}(X.dir)}{\text{peso}(X)} \leq \alpha$. O valor $\alpha = 0.5$ corresponde ao balanceamento ideal.

Demonstra-se que o tempo de execução de uma operação de procura numa árvore $\alpha-$balanceada é logarítmico no tamanho da árvore.

1. Quando, na sequência de uma operação de inserção ordenada seguindo o algoritmo habitual, uma sub-árvore de tamanho M deixa de ser $\alpha-$balanceada, é possível reajustá-la, por forma a torná-la $0.5-$balanceada, em tempo $\Theta(M)$ (desde que se possa utilizar espaço auxiliar também de tamanho $\Theta(M)$).

   Mostre (por palavras ou utilizando pseudo-código) como pode ser implementado este reajuste (baseie-se na estrutura definida na Questão 2).

   Resolução:
   Basta fazer uma travessia *inorder* da árvore que deixou de estar balanceada, escrevendo o resultado num array (daí o requisito de se dispor de espaço de tamanho linear). Este array está necessariamente ordenado. Pode agora criar-se um nó a partir da posição central do array, e recursivamente repetir este processo construindo a sub-árvore da esquerda (resp. direita) a partir do sub-array à esquerda (resp. direita) da posição central (a recorrência que descreve o tempo de execução deste algoritmo tem solução linear). O facto de metade dos elementos serem colocados à esquerda (e metade à direita) garante que se trata de uma árvore $0.5-$balanceada.

2. O tempo *amortizado* deste reajuste é no entanto $\mathcal{O}(1)$, o que significa que o

tempo amortizado da operação de inserção numa árvore $\alpha-$balanceada de tamanho N é de facto $\mathcal{O}(\lg N)$ (uma vez que os reajustes necessários são todos feitos em nós de um caminho desde a raíz até ao nó mais profundo em que o balanceamento foi infringido, e este caminho tem comprimento logarítmico).

Considere a seguinte função de potencial sobre árvores:

$$\Phi(t) = c\,.\,\sum_{X \in t} \Delta(X)$$

com
$\Delta(X) = \mid peso(X.esq) - peso(X.dir) \mid$
se $\mid peso(X.esq) - peso(X.dir) \mid \geq 2$
e
$\Delta(X) = 0$ se $\mid peso(X.esq) - peso(X.dir) \mid < 2$

Determine um valor apropriado para a constante c em função do valor de α. Mostre que a função Φ satisfaz os requisitos de uma função de potencial e mostre que o custo amortizado resultante para a operação de reajuste é efectivamente constante.

Pistas de Resolução:
Uma inserção que provoque um reajuste de uma sub-árvore de tamanho M será executada em tempo M.

A única parcela que se altera no somatório da função de potencial é a que diz respeito ao nó X que é "rebalanceado", e esta parcela:

- vale $c.\mid peso(X.esq) - peso(X.dir) \mid$ antes da inserção;
- passa a valer 0 depois da inserção, uma vez que $\Delta(X) = 0$ por definição para nós X que sejam $0.5-$balanceados.

Relembre-se que o custo amortizado será dado por $c_i = t_i - \Phi_{i-1} + \Phi_i$
Temos então:
$c_i = M - c.\mid peso(X.esq) - peso(X.dir) \mid$

Se a árvore da esquerda tiver maior peso, note-se que
$peso(X.dir) = peso(X) - 1 - peso(X.esq) = M - 1 - peso(X.esq)$,
logo

$$| \, peso(X.esq) - peso(X.dir) \, | = 2.peso(X.esq) - M + 1$$

sabe-se que $peso(X.esq) \leq \alpha.M$, logo

$$| \, peso(X.esq) - peso(X.dir) \, | \leq \ 2.\alpha.M - M + 1$$

(. . .)