Segundo teste de **Algoritmos e Estruturas de Dados**

22 de Novembro de 2017 **Duração máxima: 1 hora 30 minutos**

Justifique todas as suas respostas.

Nome:

N. Mec.:

Nas perguntas sobre árvores binárias, cada nó da árvore usa a seguinte estrutura de dados:

```
typedef struct tree_node
{
  struct tree_node *left;   // pointer to the left branch (a sub-tree)
  struct tree_node *right;  // pointer to the right branch (a sub-tree)
  struct tree_node *parent; // pointer to the parent node (NULL for the root node)
  int data;                 // the data (only one node can have a given data value)
}
tree_node;
```

2.0 **1:** Escreva uma função recursiva que, dada a raíz de uma árvore binária **não ordenada**, e um valor `v`, conta o número de nós da árvore que armazenam valores menores ou iguais a `v`. Qual é a complexidade computacional da sua função?

3.0 **2:** Escreva uma função recursiva **eficiente** que, dada a raíz de uma árvore binária **ordenada**, e um valor `v`, conta o número de nós da árvore que armazenam valores menores ou iguais a `v`. Qual é a complexidade computacional da sua função?

2.0 **3:** Explique como está organizada a informação num *min-heap*. Ilustre a sua exposição inserindo os números, por esta ordem, num *min-heap* inicialmente vazio: **7**, **3**, **9**, **1**, **2**.

3.0 **4:** É possível implementar eficientemente uma fila (*queue*) usando uma lista simplesmente ligada. Como?

2.0 **5:** Dos vários algoritmos de ordenação que conhece, existem alguns que funcionam naturalmente de uma forma recursiva. Explique o funcionamento de um deles (recursivo!).

2.0 **6:** Compare dois algoritmos de ordenação à sua escolha no que diz respeito a i) complexidade computacional, ii) melhor caso, iii) pior caso.

3.0 **7:** Explique como pode procurar informação numa lista biligada. Explique também como pode tornar a procura mais eficiente quando a informação de que se está à procura não estiver uniformemente distribuida.

3.0 **8:** Explique como funciona uma *hash table*. Indique as vantagens e desvantagens das implementações de *hash tables* usando *open-addressing* e *chaining*.

(1)

```
int countLoE(tree_node *link, int v) {
        int counter = 0;
        if(link != NULL) {
                counter = counter + countLoE(link->left, v);
                counter = counter + countLoE(link->right, v);
                if (link->data <= v) {
                        counter++;
                }
        }
        return counter;
}
```

Complexidade computacional: O(n)

(2)

```
int countLoE(tree_node *link, int v) {
        int counter = 0;
        if(link != NULL) {
                if (link->data <= v) {
                        counter++;
                        counter = counter + countLoE(link->left, v);
                        counter = counter + countLoE(link->right, v);
                }
                else { counter = counter + countLoE(link->right, v); }
        }
        return counter;
}
```

Complexidade computacional: O(n), pior caso é igual ao de cima

(3)

Uma *min-heap* é uma árvore binária completa em que o valor de cada nó interno is menor ou igual ao valor de qualquer um dos seus nós descendentes. 7,3,9,1,2

7

7, 3

3, 7

3, 7, 9

3, 7, 9, 1

3, 1, 9, 7

1, 3, 9, 7

1, 3, 9, 7, 2

1, 2, 9, 7, 3

(4)

Uma fila (*queue*) é uma estrutura de dados na qual as operações efetuadas seguem uma ordem específica. Esta ordem é FIFO (*First In First Out*), o que significa que, se quisermos retirar um elemento da fila, temos sempre que retirar o que já lá foi colocado há mais tempo.

De modo a implementar uma fila através de uma lista ligada, temos sempre dois ponteiros: *front* e *rear*. O ponteiro *front* aponta para o primeiro elemento da fila (índice 0 da lista ligada) e o ponteiro *rear* para o último (último índice da lista ligada). Para adicionar um novo elemento à fila, temos a função *enQueue()*, que adiciona um novo nó após o nó para o qual rear aponta e move o *rear* para o novo nó criado. Para retirar um elemento da fila, temos a função *deQueue()* que remove o nó para o qual o ponteiro *front* aponta e move o ponteiro para o próximo nó.

(5)

O algoritmo *mergesort* funciona de forma recursiva e através do raciocínio "*divide and conquer*". Este algoritmo divide o *array* em duas metades, chama-se a si próprio para cada uma das metades e depois funde as soluções de cada uma para ordenar o *array* inteiro.

(6)

Quicksort:

- complexidade computacional: O(n*log(n)) (caso médio)
- melhor caso: O(n*log(n))
- pior caso: O(n^2)

Mergesort:

- complexidade computacional: Θ(n*log(n)) (caso médio)
- melhor caso: O(n*log(n))
- pior caso: O(n*log(n))

(7)

Para efetuar uma pesquisa numa lista biligada, é necessário percorrer a lista toda até (1) encontrarmos o elemento que queremos ou (2) chegarmos ao fim da lista. Para isto, utilizamos um ponteiro que começa para apontar para o primeiro elemento da lista e efetuamos a comparação entre o seu valor e o valor que queremos encontrar. Se for igual, retornamos o seu endereço (valor do ponteiro). Caso contrário, incrementamos o ponteiro de modo a que ele aponte para o próximo elemento da lista e repetimos o processo de comparação. O ponteiro torna-se *NULL* se terminar de iterar a lista sem encontrar o valor pretendido, situação na qual se retorna *NULL*.

Se a informação de que se está à procura não estiver uniformemente distribuída, podemos mover cada elemento um pouco mais para o início da lista sempre que ele é pesquisado. Ao fim de algumas pesquisas, isto resulta nos elementos que são mais frequentemente pesquisados se encontrarem no início da lista, reduzindo o seu tempo de procura e aumentando a eficiência do processo de pesquisa em geral.

(8)

Uma *hash table* é uma estrutura de dados que implementa um *array* associativo (estrutura de dados abstrata que mapeia chaves para valores). A *hash table* usa uma *hash function* para calcular um índice, também denominado *hash code*, correspondente a um espaço no *array*. Durante uma pesquisa, a chave é passada como argumento à *hash function* e o retorno da mesma indica onde, no *array* de valores, o valor pretendido está guardado.