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

**12 de Novembro de 2018** **Duração máxima: 1 hora**

**Justifique todas as suas respostas.**

---

**3.0** **1:** Explique como pode usar uma lista simplesmente ligada para implementar uma pilha. Que vantagens e desvantagens tem uma implementação deste tipo quando comparada com uma outra que usa um *array*?

**3.0** **2:** Um *array* circular pode ser utilizado para implementar uma fila. Explique como. Que vantagens e desvantagens tem uma implementação deste tipo quando comparada com uma outra que usa uma lista ligada?

**3.0** **3:** Explique como está organizado um *max-heap* e como se insere informação nesta estrutura de dados. (Para explicar a inserção, pode usar como exemplo inserir os números 1, 2, 3, 4 e 5, num *max-heap* inicialmente vazio.)

**2.0** **4:** Explique como pode procurar informação numa lista biligada não ordenada, e indique qual a complexidade computacional do algoritmo que descreveu. O que é que pode fazer para tornar a procura mais eficiente quando alguns itens de informação são mais procurados que outros?

**4.0** **5:** Numa aplicação que pretende contar o número de ocurrências de palavras num texto extenso usando uma *hash table* (tabela de dispersão, dicionário) um aluno, com pouca experiência nestas coisas, usou a seguinte *hash function*:

```
#define hash_table_size 1000003u

unsigned int hash_function(unsigned char *s)
{
  unsigned int sum;
  for(sum = 0;*s != '\0';s++)
    sum += (unsigned int)(*s);
  return sum % hash_table_size;
}
```

Sabe-se que o número de palavras distintas que existem no texto é inferior mas próximo de um milhão, e que o número de letras médio de uma palavra é inferior a 10.

2.0 **a)** Explique porque é que neste caso deve usar uma implementação da *hash table* que usa *separate chaining*, em vez de usar uma que usa *open addressing*.

2.0 **b)** A *hash function* apresentada acima é muito má. Porquê? Sugira uma outra que seja bem melhor.

---

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

```
typedef struct tree_node
{
  struct tree_node *left;
  struct tree_node *right;
  long data;
}
tree_node;
```

**2.0** **6:** A seguinte função é uma implementação **errada** de uma função recursiva que procura informação numa árvore binária **não ordenada**.

```
tree_node *tree_search(tree_node *n,long data)
{
  if(n == NULL || n->data == data)
    return NULL;
  return tree_search((data < n->data) ? n->left : n->right,data);
}
```

Explique o que está errado na função e corrija-a.

**3.0** **7:** Escreva uma função que indique se uma árvore binária está balanceada ou não. Recorda-se que numa árvore binária balanceada as alturas das sub-árvores dos lados direito e esquerdo não podem diferir em mais de 1.

```
int is_balanced(tree_node *n)
{
  // put your code here
}
```

Considere que se o valor devolvido pela função for -1 então a árvore não está balanceada.

**(1)**

Uma pilha (stack) é uma estrutura de dados que segue uma ordem particular para executar instruções. A ordem é LIFO (Last In First Out), o que significa que, se quisermos retirar um elemento da pilha, temos sempre que retirar o último que lá foi colocado.
De modo a implementar uma pilha usando uma lista ligada, vamos ter um ponteiro a apontar para o "topo" da pilha (ou seja, para o último elemento que lá foi colocado). Como é óbvio, é no topo da pilha que acontecem as operações de inserção e remoção de elementos. Quando queremos colocar um novo elemento na pilha, criamos um novo elemento da lista ligada que contém o valor pretendido e que aponta para o topo da pilha. Este elemento torna-se o novo topo da pilha. Quando queremos retirar um elemento da pilha, apagamos o elemento da lista ligada que representa o topo da pilha, ou seja, o primeiro (lista ligada é sequencial).

Vantagens da utilização da lista ligada:
- operações de inserção e remoção de elementos numa lista ligada são mais rápidas que num array
- listas ligadas são dinâmicas e flexíveis, podendo expandir e contrair o seu tamanho consoante seja necessário; arrays têm tamanho fixo

Desvantagens da utilização da lista ligada:
- num array, os elementos pertencem a índices; numa lista ligada, temos sempre que começar pelo primeiro nó e iterar ao longo dos nós para chegar a um dado elemento; assim sendo, acesso a um elemento específico (que não seja o primeiro) é mais lento em listas ligadas do que em arrays, mas nesta situação específica esse aspeto não é relevante porque, numa pilha, só estamos interessados em aceder ao elemento no topo, que corresponde sempre ao primeiro nó da lista ligada
- memória de um array é alocada durante a compilação; para uma lista ligada, temos que constantemente alocar e desalocar espaço de memória durante a execução do programa, o que torna o uso de listas ligadas mais demorado

**(2)**

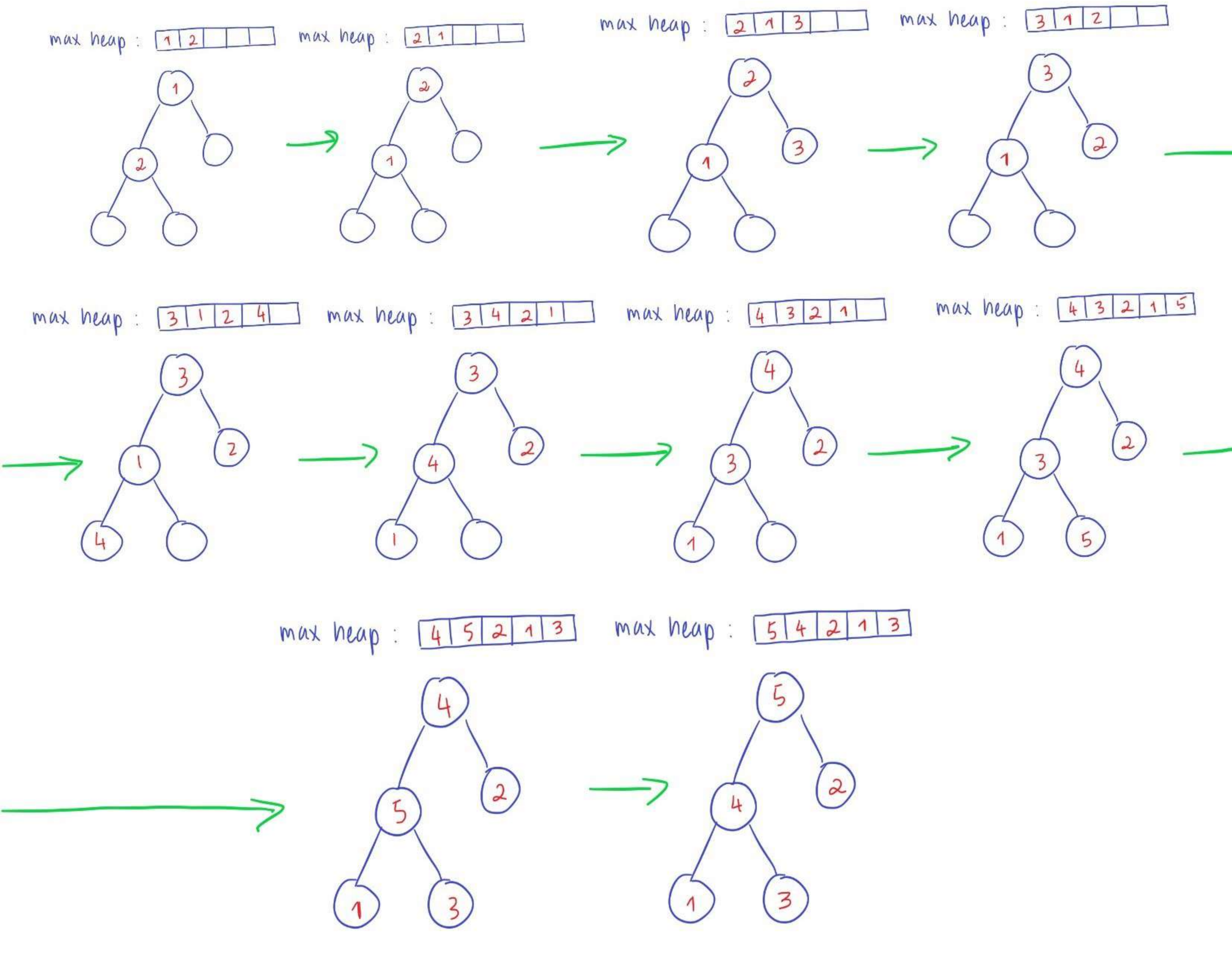
Uma fila (queue) é uma estrutura de dados que segue uma ordem particular para executar instruções. A ordem é FIFO (First In First Out), o que significa que, se quisermos retirar um elemento da fila, temos sempre que retirar o primeiro que lá foi colocado (ou seja, o elemento que já foi colocado há mais tempo, de entre os disponíveis).
De modo a implementar uma pilha usando um array circular (estrutura de dados que usa um array como se o seu fim e o seu ínicio estivessem conectados), vamos ter dois ponteiros: um ponteiro de leitura e um ponteiro de escrita. O ponteiro de leitura vai apontar sempre para o fundo da fila (ou seja, avançando pelo array circular no sentido dos ponteiros do relógio, o ponteiro de leitura aponta para a primeira casa ocupada do array circular, a seguir aos espaços vazios; isto corresponde ao primeiro elemento colocado). O ponteiro de escrita aponta sempre para o topo da fila, ou seja, para o último elemento que foi colocado. Para colocar um novo elemento na fila, avança-se o ponteiro de escrita e faz-se a inserção. O ponteiro de leitura não muda de sítio. Para remover um elemento da fila, avançamos o ponteiro de leitura. O ponteiro de escrita não muda de sítio. Tecnicamente, o valor que se pretende apagar continua no array circular mas, como o ponteiro de leitura não está a apontar para ele, se o ponteiro de escrita lá chegar vai assumir que o espaço está vazio e escrever um novo valor por cima.

Vantagens da utilização de um array circular:
- numa fila, quando queremos retirar um elemento, temos que o retirar do fundo da fila; com um array circular, podemos simplesmente aceder ao índice desse elemento com o ponteiro de leitura e retirá-lo; numa lista ligada, teríamos que percorrer a lista inteira de cada vez que quiséssemos efetuar uma remoção só para conseguir aceder ao elemento a retirar
- memória de um array é alocada durante a compilação. logo poupa tempo durante a execução do programa; para uma lista ligada, temos que constantemente alocar e desalocar espaço de memória durante a execução do programa

Desvantagens da utilização de um array circular:
- operações de inserção e remoção de elementos numa lista ligada são mais lentas que numa lista ligada
- arrays têm tamanho fixo; listas ligadas são dinâmicas e flexíveis, podendo expandir e contrair o seu tamanho consoante seja necessário

**(3)**

Um max heap é uma árvore binária completa em que o valor de cada nó interno é maior ou igual aos valores de todos os nós que descendem de si próprio. Exemplo de inserção:




**(4)**

Como a lista não é ordenada, temos que percorrer todos os elementos e, para cada um, avaliar se é aquele que se pretende. Sendo a variável head representativa do primeiro elemento da lista, a variável current representativa do elemento que estamos a avaliar de momento e a variável data representativa da informação que pretendemos encontrar na lista, começamos por avaliar se head==NULL. Se sim, a lista não foi inicializada e a pesquisa termina. Caso contrário, começamos pelo primeiro elemento (current=head;). Utilizando um ciclo while (enquanto current for diferente de NULL), se current-> data for igual a data, encontrámos a posição onde estava guardada a informação pretendida e a pesquisa termina. Finalmente, se a posição seguinte (ou seja, current->next) for diferente de NULL, passamos à posição seguinte (current = current->next;); caso contrário, interrompemos o ciclo while, já que a informação não existe na lista.

Se alguns items forem mais procurados que outros, podemos incorporar código adicional no algoritmo de pesquisa que, a cada pesquisa de um dado item, o move um pouco mais para perto do primeiro elemento. Isto faz com que, após algumas pesquisas, os items mais frequentemente pesquisados estejam concentrados no início da lista (onde é mais rápido que o algoritmo de pesquisa chegue a eles) e os menos pesquisados no fim.

**(5)**

**a)** Neste caso, temos quase 1.000.000 palavras mas uma hash table com tamanho de 1.000.003. Os números são tão próximos que é basicamente garantido que haja várias colisões. O método de open addressing só há mais eficiente que separate chaining quase temos basicamente a certeza de que não vão haver colisões (nunca podemos ter 100% de certeza).

**(b)** A *hash function* apresentada é muito má porque utiliza uma simples cifra de substituição, ou seja, não tem em conta a posição de cada caracter na *string* - apenas o seu valor. Isto significa que palavras com a mesma combinação de letras em posições diferentes (anagramas) vão ter o mesmo *hash code*. Para além disso, é possível adicionar um valor aleatório a uma letra (por exemplo, A+1=B) e subtrair o mesmo valor noutra letra (por exemplo, N-1=M) e, mudando a ordem dos caracteres, criar novas *strings*, todas com o mesmo *hash code*. Fazendo isto para todas as letras de uma palavra, podemos criar uma palavra com um conjunto de letras inteiramente diferente que, mesmo assim, tem o mesmo *hash code* da palavra inicial. Com este mesmo truque, podemos mesmo variar o comprimento das novas palavras geradas e obter os mesmos *hash codes*. Uma maneira simples de resolver este problema seria modificar a linha dentro do ciclo *for* para:

sum += (157u * sum) + (unsigned int)(*s) ,

sendo o fator multiplicativo 157 escolhido quase arbitrariamente (é um número primo).

**(6)**

No if statement, se um dado nó for não-existente (NULL) ou a sua informação coincidir com a informação que procuramos, então retornamos esse mesmo nó (e não NULL, como o código sugere).
Para além disso, a árvore binária não está ordenada, por isso na chamada recursiva à função tree_search não faz sentido analisar o nó esquerdo do nó atual se a informação que se procura é menor que a informação do nó atual, e analisar o nó direito caso contrário. Basicamente, vamos ter que analisar todos os nós até encontrarmos a informação que queremos (não podemos implementar uma espécie de binary search como apresentado). Assim sendo, se a chamada recursiva à esquerda retornar não-NULL, é porque o nó certo foi encontrado e retornamo-lo. Caso contrário, analisamos à direita.

**Função corrigida:**

```
tree_node *tree_search(tree_node *n, long data)
{
	if (n == NULL || n->data == data)
		return n;
	if ((res = tree_search(n->left, data)) != NULL)
		return res;
	return tree_search(n->right, data);
}
```

**(7)**

```
int is_balanced(tree_node *n)
{
	int lh; /* height of left subtree */
	int rh; /* height of right subtree */

	/* if tree is empty then return true */
	if (n == NULL)
	return 1;

	/* get the height of left and right subtrees */
	lh = height(n->left);
	rh = height(n->right);

	if (abs(lh - rh) <= 1 && isBalanced(n->left) && isBalanced(n->right))
	return 1;

	/* if we reach here then tree is not height-balanced */
	return 0;
}

int height(struct node* node)
{
	/* base case tree is empty */
	if (node == NULL)
	return 0;

	/* if tree is not empty then height = 1 + max of left height and right heights */
	return 1 + max(height(node->left), height(node->right));
```