Segundo teste de Algoritmos e Estruturas de Dados

18 de Novembro de 2019 **14h10m – 15h00m**

Responda a todas as perguntas no enunciado do teste. Justifique todas as suas respostas.
O teste é composto por 5 grupos de perguntas.

Nome: _____

N. Mec.: _____

4.0 **1:** Pretende-se que a seguinte função implemente uma pesquisa binária. Complete-a (isto é, preencha as caixas).

```
int binary_search(int n,int a[n],int v)
{
  int low = [        ];
  int high = [         ];
  while(high [    ] low)
  {
    int middle = [                    ];
    if(a[middle] == v)
      return middle;
    if(a[middle] [    ] v)
      [         ] = middle - 1;
    else
      [         ] = middle + 1;
  }
  return [    ];
}
```

Indique (não é preciso justificar) qual é a complexidade computacional desta função.

Resposta:

4.0 **2:** Explique como está organizado um *max-heap*. Para o *max-heap* apresentado a seguir, insira o número 8. Não apresente apenas o resultado final; mostre, passo a passo, o que acontece ao *array* durante a inserção. Em cada linha, basta escrever as entradas do *array* que foram alteradas.

Respostas:

9	6	7	3	1	4	5	2	
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

3.0 **3:** Pretende-se implementar uma fila (*queue*) usando um *array* circular. O código seguinte define a estrutura de dados a usar (para simplificar, vamos usar variáveis globais).

```
#define array_size  1024

int array[array_size];
int read_pos = 1;  // incremented after reading
int write_pos = 0; // incremented before writing
int count = 0;     // equal to (read_pos - write_pos - 1) % array_size
```

Responda às seguintes perguntas:

1.5 **a)** Por que é que neste caso é vantajoso usar um *array* circular?

Resposta:

1.5 **b)** Use algumas das seguintes linhas de código para implementar a função `enqueue` (que coloca um item de informação na fila). Risque as linhas que estão a mais.

```
int enqueue(int v)
void enqueue(int v)
{
  if(count == 0) exit(1); // underflow
  if(count == array_size) exit(1); // overflow
  array[write_pos] = v;
  v = array[write_pos];
  write_pos = (write_pos + 1 == array_size) ? 0 : write_pos + 1;
  write_pos = (write_pos > 0) ? write_pos - 1 : array_size - 1;
  array[write_pos] = v;
  v = array[write_pos];
  count--;
  count++;
  return v;
}
```

5.0 **4:** Um programador pretende utilizar uma *hash table* (tabela de dispersão, dicionário) para contar o número de ocorrências de palavras num ficheiro de texto. O programador está à espera que o ficheiro tenha cerca de 6000 palavras distintas, pelo que usou uma *hash table* do tipo *separate chaining* com 10007 entradas, e usou a seguinte *hash function*:

```
unsigned int hash_function(unsigned char *s,unsigned int hash_table_size)
{
  unsigned int sum;

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

Infelizmente, as expetativas do programador estavam erradas, e o ficheiro de texto era muito maior que o esperado, tendo cerca de 1000000 palavras distintas. Responda às seguintes perguntas:

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

3.0 **b)** Uma implementação do tipo *separate chaining* usa habitualmente uma lista ligada para armazenar todas as chaves (neste caso, as palavras) para as quais a *hash function* tem o mesmo valor. Que vantajens/desvantagens teria uma implementação que usa uma árvore binária ordenada em vez da lista ligada? E se for uma árvore binária ordenada e balanceada?

Respostas:

4.0 **5:** Apresentam-se a seguir várias funções que visitam todos os nós de uma árvore binária, e mostram-se várias ordens pelas quais a função `visit` foi chamada para cada um dos nós (1 significa que o nó correspondente foi o primeiro a chamar a função `visit`, 2 que foi o segundo, e assim por diante). Para cada uma das ordens apresentadas, indique que função, ou funções, deram origem a essa ordem.

```
void f1(tree_node *link)
{
  queue *q = new_queue();
  enqueue(q,link);
  while(is_empty(q) == 0)
  {
    link = dequeue(q);
    if(link != NULL)
    {
      visit(link);
      enqueue(q,link->left);
      enqueue(q,link->right);
    }
  }
  free_queue(q);
}
```

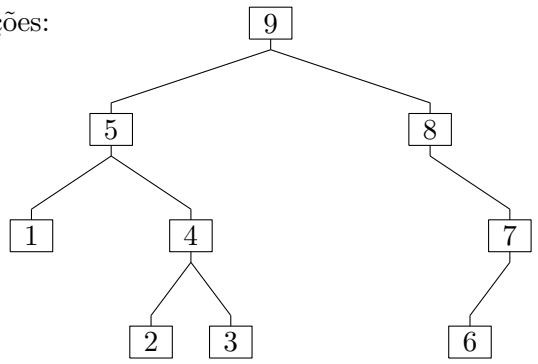
```
void f2(tree_node *link)
{
  stack *s = new_stack();
  push(s,link);
  while(is_empty(s) == 0)
  {
    link = pop(s);
    if(link != NULL)
    {
      visit(link);
      push(s,link->right);
      push(s,link->left);
    }
  }
  free_stack(s);
}
```

```
void f3(tree_node *link)
{
  if(link != NULL)
  {
    visit(link);
    f3(link->left);
    f3(link->right);
  }
}
```

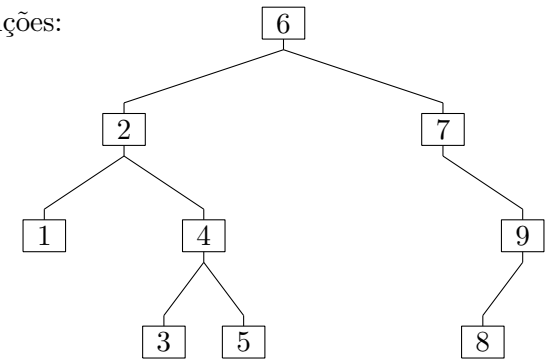
```
void f4(tree_node *link)
{
  if(link != NULL)
  {
    f4(link->left);
    visit(link);
    f4(link->right);
  }
}
```

```
void f5(tree_node *link)
{
  if(link != NULL)
  {
    f5(link->left);
    f5(link->right);
    visit(link);
  }
}
```

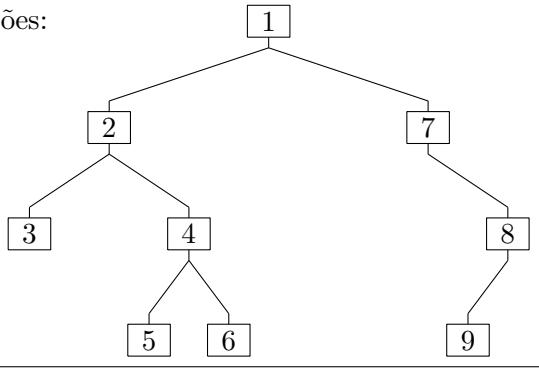
Funções:



Funções:



Funções:



Funções: