Segunda parte do teste final de Algoritmos e Estruturas de Dados

4 de fevereiro de 2022 **17h00m – 17h55m**

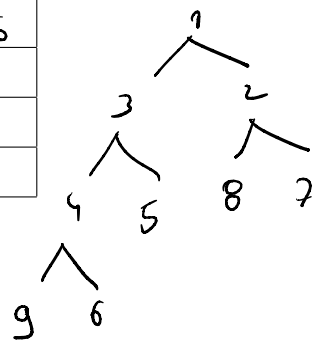
Responda a todas as perguntas no enunciado do teste. Justifique todas as suas respostas.

Nome: [redacted]

N. Mec.: _____

4.0 **1:** Explique como está organizado um *min-heap*. Para o *min-heap* apresentado a seguir, insira o número 3. 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.

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



4.0 **2:** Explique como funciona a rotina de ordenação *insertion sort*. Indique a sua complexidade computacional e quais os seus melhores e piores casos.

Insertion sort é um algoritmo de ordenação que divide um conjunto de dados em dois subconjuntos: um ordenado e um por ordenar. A cada iteração, o algoritmo pega no próximo elemento do subarray desordenado e insere-o na sua posição correta do array ordenado. Isto é feito comparando o elemento com os já ordenados, deslocando-os para a direita (ou esquerda dependendo do caso) se necessário

Pior caso: $O(n^2)$ → array por ordem inversa

Melhor caso: $O(n)$ → já está ordenado

4.0 **3:** 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? Tende não usar mais de 100 palavras.

Sendo a lista não onderada teremos de percorrer todos os elementos da lista e, para cada um, avaliar se é igual ao pretendido ou não, caso não seja passamos para o próximo. Isto é feito através de um ciclo while que aponta para a "casa" seguinte caso esta seja ≠ NULL. Tem complexidade de O(n) pois pode ter de percorrer o array todo

Uma maneira de tornar a procura mais eficientes é mover os endereços mais procurados para o início. Usando por exemplo a Via-Verde, se esta tiver um array com todos os IDs dos utilizadores é mais eficiente se tiver os IDs dos condutores que viajam mais nos primeiros endereços pois, assim, sempre que passarem no pórtigo é mais fácil de encontrar a sua informação

4.0 **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 = 0u;

  for(int i = 0;s[i] != '\0';s++)
    sum += (unsigned int)(i + 1) * (unsigned int)s[i];
  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:

1.0 **a)** A *hash function* apresentada não é das piores. Porquê?

3.0 **b)** Com *separate chaining* a *hash table* pode acomodar o milhão de palavras distintas mesmo tendo a *array* apenas 10007 entradas. Explique porquê, e explique o que é que acontece ao desempenho desta estrutura de dados.

a) Não é das piores pois, embora seja simples, consegue distinguir anagramas. Por exemplo "ola" e "alo" vão ter codigos diferentes

b) Será possível o armazenamento das 10^6 palavras devido ao uso de linked lists, onde podemos armazenar informação "infinitamente"

Um problema que este caso possui é que cada head corresponderá em média a 100 palavras, o que será muito time consuming ao percorrer a mesma

4.0 **5:** Apresentam-se a seguir várias funções (`f1` a `f5`) 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 *n)
{
  queue *q = new_queue();
  enqueue(q,n);
  while(is_empty(q) == 0)
  {
    n = dequeue(q);
    if(n != NULL)
    {
      enqueue(q,n->right);
      enqueue(q,n->left);
      visit(n);
    }
  }
  free_queue(q);
}
```

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

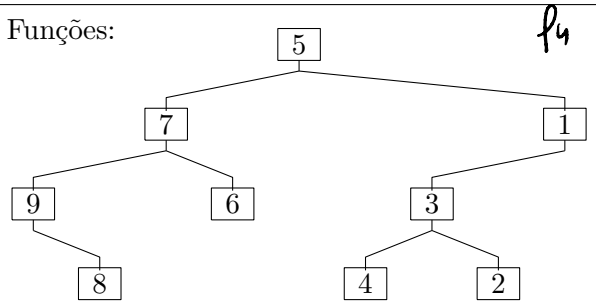
```
int cnt = 0;

void visit(tree_node *n)
{
  printf("%d\n",++cnt);
}
```

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

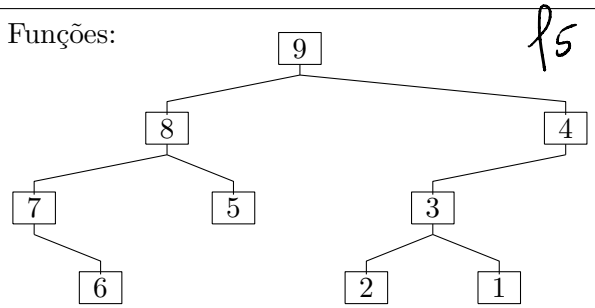
```
void f4(tree_node *n)
{
  if(n != NULL)
  {
    f4(n->right);
    visit(n);
    f4(n->left);
  }
}
```

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

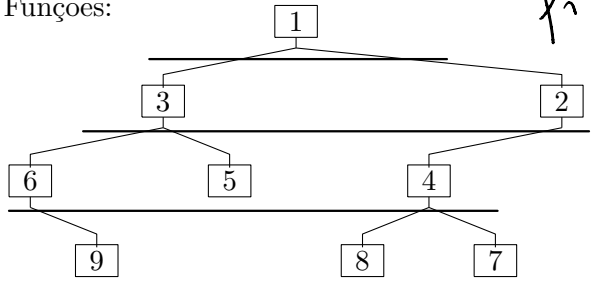
Funções: f4



Funções: f5



Funções: f2



Funções: f3