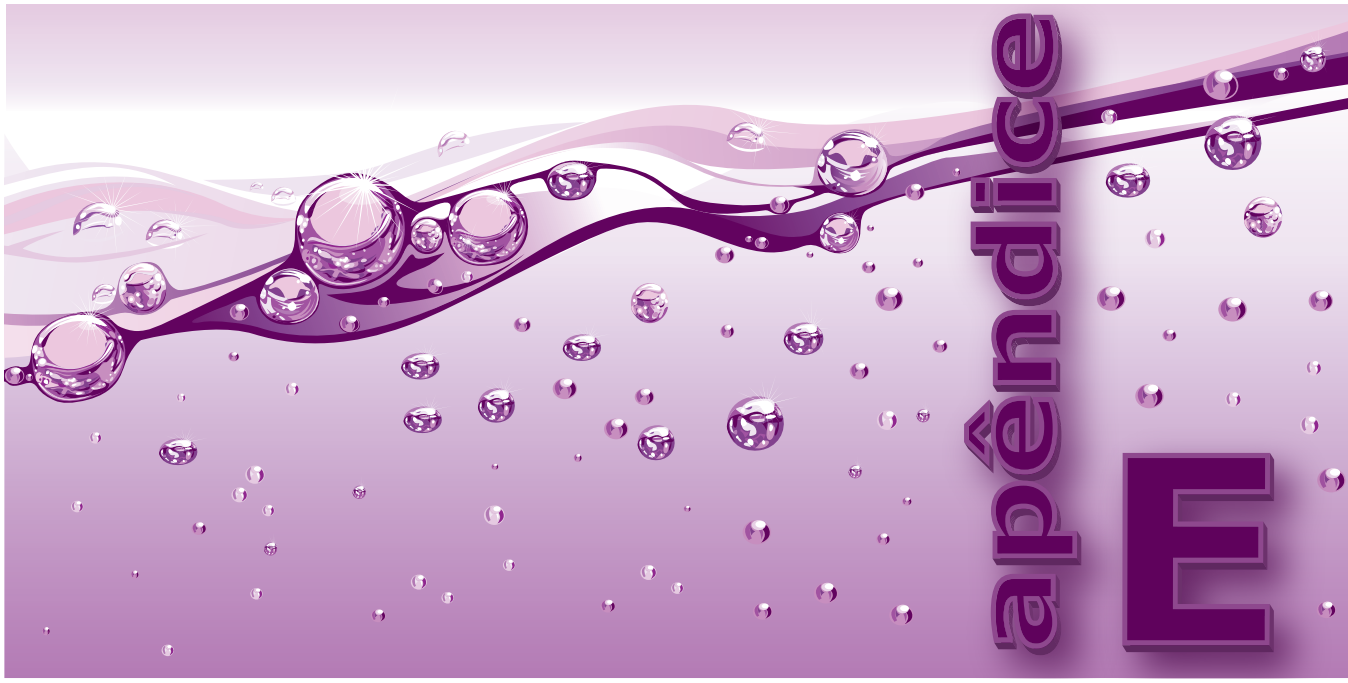

Respostas e Sugestões para os Exercícios de Revisão

Capítulo 1 — Organização de Memória

1. Consulte a **Seção 1.1.1**.
2. Consulte a **Seção 1.1.1**.
3. Consulte a **Seção 1.1.1**.
4. Consulte a **Seção 1.1.1**.
5. Consulte a **Seção 1.1.1**.
6. (a) Consulte a **Seção 1.1.1**. (b) Consulte a **Seção 1.1.2**. (c) Memória RAM, registradores, memória cache. (d) Memórias ROM, discos, fitas magnéticas, memórias flash.
7. Leia a **Nota de rodapé [1]** na **página 56**.
8. Consulte a **Seção 1.1.2**.
9. Leia a **Nota de rodapé [2]** na **página 57**.
10. Consulte a **Seção 1.1.3**.
11. Consulte a **Seção 1.1.3**.
12. Consulte a **Seção 1.1.3**.
13. Consulte a **Seção 1.1.3**.
14. Consulte a **Seção 1.1.3**.
15. Um sistema de arquivos dá suporte a tarefas, tais como organização, armazenamento e recuperação de dados, e gerenciamento de informações num meio de armazenamento não volátil. Um sistema de arquivos é parte integrante de qualquer sistema operacional.

16. Consulte a **Seção 1.1.3**.
17. Consulte a **Seção 1.1.3**.
18. Aproximadamente 18,4 GB.
19. Consulte a **Seção 1.1.3**.
20. Consulte a **Seção 1.1.3**.
21. Consulte a **Seção 1.1.3**.
22. Se todos os blocos que compõem um arquivo fazem parte de um mesmo cilindro, o acesso a esses blocos será mais rápido porque as cabeças de leitura/escrita só precisam ser movidas uma única vez.
23. Consulte a **Seção 1.1.4**.
24. Consulte a **Seção 1.1.4**.
25. Consulte a **Seção 1.1.4**.
26. Consulte a **Seção 1.1.4**.
27. Consulte a **Seção 1.1.4**.
28. Consulte a **Seção 1.1.5**.
29. A justificativa é a enorme quantidade de dados que uma fita magnética pode armazenar.
30. Aproximadamente 8,2 GB.
31. *10,51 ms*.
32. Consulte a **Seção 1.1.3**.
33. Consulte a **Seção 1.1.3**.
34. Consulte a **Seção 1.1.3**.
35. Consulte a **Seção 1.1.3**.
36. Consulte a **Seção 1.1.3**.
37. Bloco.
38. Consulte a **Seção 1.1.3**.
39. É o mesmo que bloco.
40. Consulte a **Seção 1.1.3**.
41. Veja a **Nota de rodapé** [7] na **página 65**.
42. Consulte a **Seção 1.1.3**.
43. Consulte a **Seção 1.1.3**.
44. Porque se um registro ocupar apenas um bloco, será necessária apenas uma operação de acesso para lê-lo ou escrevê-lo.
45. Consulte a **Seção 1.2.2**.
46. Consulte a **Seção 1.2.2**.
47. Consulte a **Seção 1.2.2**.
48. Consulte a **Seção 1.2.2**.
49. Consulte a **Seção 1.2.3**.
50. Consulte a **Seção 1.2.3**.
51. Consulte a **Seção 1.2.3**.
52. Consulte a **Seção 1.2.3**.
53. Consulte a **Seção 1.2.3**.
54. (a) Em programação, *buffer* tem vários significados dependendo do contexto no qual esse termo é empregado. Neste livro, buffer é sempre um espaço em memória principal no qual são temporariamente armazenados dados que foram lidos ou que serão escritos num arquivo. (b) Os objetivos de um buffer e de uma memória

cache são os mesmos: acelerar o processamento de dados que residem num meio de armazenamento mais lento, mas existem muitas diferenças entre buffer e cache, dentre as quais podem ser mencionadas: memória cache armazena temporariamente dados que se encontrem numa memória mais lenta de uma hierarquia de memória e não apenas dados armazenados em arquivo, como faz um buffer; tipicamente, o programador tem controle direto sobre buffers, o que não ocorre com memórias cache; as políticas de desalojamento de cache normalmente não se aplicam a gerenciamento de buffers.

55. Consulte a **Seção 1.3**.

56. Existem inúmeras variações. Pesquise na internet.

57. Para ter conhecimento para decidir, por exemplo, qual é a estrutura de dados mais adequada para ser usada num certo nível da hierarquia. Por exemplo, árvores binárias são adequadas para memória principal, mas esse não é o caso para memória secundária. Além disso, o conhecimento de localidade de referência, que é baseado em hierarquias de memória, permite ao programador escrever programas mais eficientes.

58. Precisamente, memória externa recebe essa denominação porque ela está associada a um periférico de armazenamento, que, por definição, se encontra fisicamente fora do computador. Memória secundária está num nível logo abaixo da memória principal e também está associada a um periférico. Portanto toda memória secundária é uma memória externa, mas a recíproca não é necessariamente verdadeira. Quer dizer, memória terciária, por exemplo, também é memória externa. Além disso, nem memória externa nem memória secundária precisam estar associados a um disco. Neste livro, porém, por razões práticas, na maioria das vezes, memória externa, memória secundária e disco significam a mesma coisa. Disco magnético é um meio de armazenamento usado para compor uma memória externa e HD é um tipo de disco magnético.

59. Consulte a **Seção 1.4**.

60. O conceito de caching é baseado em localidade de referência que, por sua vez, é baseado em hierarquias de memória.

61. Consulte a **Seção 1.4**.

62. Porque o tamanho de uma memória cache é tipicamente bem menor do que o tamanho da memória da qual provêm os dados.

63. Consulte a **Seção 1.4**.

64. Consulte a **Seção 1.4**.

65. Consulte a **Seção 1.4**.

66. Consulte a **Seção 1.5**.

67. Consulte a **Seção 1.5.1**.

68. Consulte a **Seção 1.5.1**.

69. Consulte a **Seção 1.5.1**.

70. Consulte a **Seção 1.5.1**.

71. Consulte a **Seção 1.5.1**.

72. Consulte a **Seção 1.5.4**.

73. Consulte a **Seção 1.5.2**.

74. Consulte a **Seção 1.5.3**.

75. Consulte a **Seção 1.5.3**.

76. Depende do sistema de arquivos utilizado. Consulte a **Seção 1.5.3**.

77. (a) 2. (b) Trocando os índices `i` e `j` dos laços, como na função `Transposta2()` a seguir:

```
typedef int tMatriz[200][200];

void Transposta2(tMatriz destino, tMatriz origem)
{
   int i, j;

   for (j = 0; j < 200; j++)
      for (i = 0; i < 200; i++)
         destino[j][i] = origem[i][j];
}
```

78. Examine a V. **Figura E–1**, que mostra como um elemento de um array de elementos do tipo `tPonto` é armazenado em memória. A função `F1()` acessa os elemento do array com padrão de referência 1 e, portanto, possui a melhor localidade espacial. A função `F2()` acessa os elemento do array sequencialmente, mas não faz o mesmo com os elementos dos arrays que constituem os campos de cada elemento do array recebido como parâmetro. Portanto `F2()` tem padrão de referência pior do que aquele de `F1()`. A função `F1()` tem o pior padrão de referência das três funções. Agora faça sua parte e explique por quê.

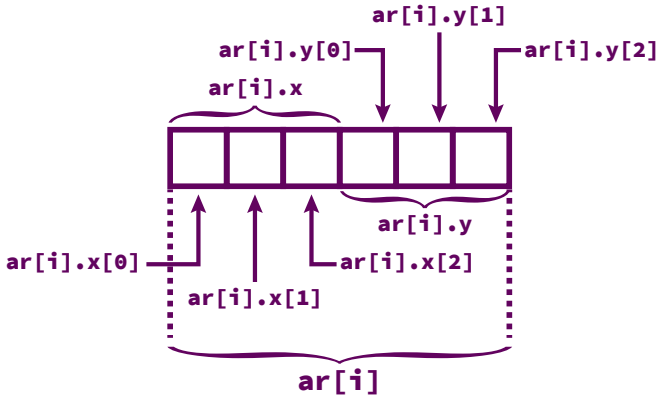

Figura E–1: Questão 78 — Capítulo 1

79. Compiladores modernos conhecem ótimas estratégias de alocação de registradores.

80. Consulte a **Seção 1.5.3**.

81. Ambos os casos exibem boa localidade de referência.

82. (a) Sim. (b) Não.

83. Sim. Os laços devem ser trocados, de tal modo que os índices mais à direita sejam alterados mais rapidamente, como mostra a função `SomaArrayTri2()` abaixo.

```
#define N 10

int SomaArrayTri2(int ar[][N][N], int n1, int n2, int n3)
{
   int  i, j, k, soma = 0;

   for (k = 0; k < n3; k++)
      for (i = 0; i < n1; i++)
         for (j = 0; j < n2; j++)
            soma += ar[k][i][j];

   return soma;
}
```

84. O padrão de referência da função `MultiplicaMatrizes()` pode ser melhorado trocando-se a ordem dos laços internos, assim:

```
for (i = 0; i < N; i++)
   for (k = 0; k < N; k++)
      for (j = 0; j < N; j++)
         C[i][j] = C[i][j] + A[i][k] * B[k][j];
```

85. Aquele cujos código e dados cabem inteiramente em memória cache.
86. Consulte a **Seção 1.6**.
87. Em memória interna, o custo temporal de um algoritmo é medido em termos do número de operações executadas. Quando se lida com memória externa, esse custo é medido em termos do número de acessos à memória.
88. Consulte a **Seção 1.6**.
89. Consulte a **Seção 1.6**.
90. Leia os comentários que acompanham a essa função.
91. Consulte a **Seção 1.7**.

Capítulo 2 — Processamento de Arquivos em C

1. Consulte a introdução do **Capítulo 2**.
2. Consulte a introdução do **Capítulo 2**.
3. Consulte a introdução do **Capítulo 2**.
4. Esse cabeçalho contém componentes que lidam com entrada e saída em geral.
5. Consulte a **Seção 2.1**.
6. Consulte a **Seção 2.1**.
7. Consulte a **Seção 2.1**.
8. Consulte a **Seção 2.2**.
9. Consulte a **Seção 2.2**.
10. Por meio de estruturas do tipo **FILE**.
11. Na maioria das vezes, não faz diferença.
12. (a) Consulte a **Seção 2.2**. (b) No cabeçalho `<stdio.h>`.
13. Consulte a **Seção 2.2**.
14. Consulte a **Seção 2.3**.
15. Consulte a **Seção 2.3**.
16. Essa constante simbólica representa o tamanho mínimo que deve ter um array que armazena um nome de arquivo numa dada implementação de C.
17. (1) O arquivo não existe, (2) falha de dispositivo de entrada ou saída, (3) o programa não tem permissão do sistema operacional para acessar o arquivo.
18. (a) Consulte a **Seção 2.3**. (b) Aborto de programa.
19. Consulte a **Seção 2.3**.
20. (a) Não há interpretação de caracteres que representam quebras de linha. (b) Ocorre a referida interpretação. (**NB**: Em sistemas da família Unix, não há diferença.)
21. Pode. Mas, normalmente, não deve.
22. Consulte a **Seção 2.3**.
23. É um modo de abertura de arquivo que permite leitura e escrita.
24. Consulte a **Seção 2.3**.
25. Formato de texto.

26. Modos de abertura para streams binários incluem a letra *b*. (Modos de abertura para streams de texto podem incluir a letra *t*, mas, na prática, raramente ela é usada.)

27. (a) `"r"` é usado com streams de texto; `"rb"` é usado com streams binários. (b) Não há diferença; ambos são usados com streams de texto. (c) A diferença é que `"rt"` é usado com streams de texto, enquanto `"rb"` é usado com streams binários.

28. Consulte a **Tabela 2–3** na **Seção 2.3**.

29. Modos de abertura que contêm a letra *w* detonam arquivos que tenham o mesmo nome usado como primeiro parâmetro de **fopen()**.

30. (a) Sim. (b) Sim, mas se ele for aberto em modo binário será mais eficiente, pois não haverá interpretação de quebra de linha. Em sistemas da família Unix não faz a menor diferença.

31. Testar se a abertura do arquivo foi bem-sucedida.

32. Consulte a **Seção 2.3**.

33. Usando a constante simbólica **FILENAME_MAX**, definida em `<stdio.h>`.

34. A constante simbólica **FILENAME_MAX** deve ser usada para dimensionar arrays que armazenam nomes de arquivos.

35. A constante simbólica **FOPEN_MAX** representa o número máximo de arquivos que a implementação de C ora utilizada garante que podem estar abertos simultaneamente.

36. O problema é que, muito provavelmente, o nome do arquivo introduzido pelo usuário inclui o caractere `'\n'` e esse caractere não faz parte de nenhum nome de arquivo. Esse problema só não ocorre se o usuário digitar um nome de arquivo contendo um número de caracteres igual `FILENAME_MAX - 1`, o que é pouco provável porque esse valor é muito grande. A **Seção 2.11.1** mostra como remover o caractere `'\n'` de um string lido com **fgets()**.

37. Consulte a **Seção 2.3**.

38. Um valor que indica se operação que ela tenta realizar foi bem-sucedida ou não.

39. Consulte a **Seção 2.3**.

40. Sim, porque libera o espaço ocupado pela estrutura **FILE** associada ao arquivo.

41. A função **fclose()** deveria receber o ponteiro `p` como parâmetro, e não o string `"teste.bin"`.

42. Consulte a **Seção 2.3**.

43. Consulte a **Seção 2.4**.

44. Pode ser, mas, hoje em dia, é muito pouco provável.

45. Consulte a **Seção 2.4**.

46. Porque o valor dessa constante pode indicar que o final do arquivo ora processado foi atingido ou a ocorrência de erro de processamento do mesmo arquivo.

47. Consulte a **Seção 2.4**.

48. Porque o uso de **EOF** pode ser ambíguo. O valor retornado por **feof()** nunca é ambíguo.

49. A função **ferror()** permite verificar se ocorreu erro numa operação de entrada ou saída.

50. O valor retornado por **ferror()** não é ambíguo.

51. Ele continuará indicando ocorrência de erro em operações subsequentes de entrada ou saída, mesmo que esse não seja o caso.

52. (a) Por meio de uma chamada de **rewind()**, **fseek()** ou **ungetc()**. (b) Por meio de uma chamada de **clearerr()**.

53. (a) Por meio de uma chamada de **fseek()**, **rewind()** ou **ungetc()**. (b) Por meio de uma chamada de **clearerr()**.

54. (a) A função **clearerr()** serve para remover indicativo de erro ou final de arquivo. (b) Porque existem funções que efetuam essa tarefa implicitamente.

55. Consulte a **Seção 2.5**.

56. Consulte a **Seção 2.5**.
57. Consulte a **Seção 2.5**.
58. Consulte a **Seção 2.5**.
59. Para descarregar explicitamente streams de saída (apenas).
60. A função **fflush()** não deve ser usada com parâmetros que representam streams de entrada, como é o caso de **stdin**.
61. Todos os streams de saída correntemente abertos no programa que contém essa instrução serão descarregados.
62. Consulte a **Seção 2.6**.
63. Porque existem funções [p. ex., **scanf()** e **printf()**] que realizam operações de entrada e saída nesses streams sem que eles precisem ser especificados explicitamente.
64. Consulte a **Seção 2.7**.
65. A função **scanf()** efetua leitura apenas no stream padrão **stdin**. A função **fscanf()** permite que se especifique o stream no qual a leitura será feita.
66. (a) Sim. (b) O primeiro parâmetro deve ser **stdin**. (c) Apenas se o primeiro parâmetro de **fscanf()** for **stdin**.
67. A função **printf()** escreve apenas em **stdout**; a função **fprintf()** permite que se especifique um stream no qual a escrita será efetuada.
68. (a) Sim. (b) O primeiro parâmetro deve ser **stdout**. (c) Apenas se o primeiro parâmetro de **fprintf()** for **stdout**.
69. (a) Consulte a **Seção 2.7**. (b) Quando se deseja converter números em strings, por exemplo.
70. Ela pode escrever além do limite do array recebido como parâmetro, causando corrupção de memória.
71. Arquivos temporários são usados para armazenar dados temporariamente enquanto outro arquivo, que irá armazenar esses dados definitivamente, está sendo processado.
72. Consulte a **Seção 2.8**.
73. Não.
74. Sim. Nesse caso, o segundo parâmetro da função `FechaArquivo()` deve ser **NULL**, já que não se conhece o nome do arquivo temporário.
75. Consulte a **Seção 2.8**.
76. Consulte a **Seção 2.8**.
77. Consulte a **Seção 2.8**.
78. Consulte a **Seção 2.8**.
79. (a) Sim. (b) Não.
80. A função **tmpfile()** cria e fecha arquivos temporários automaticamente. A função **tmpnam()** não faz isso.
81. Nesse caso, deve-se usar **tmpnam()**, pois o arquivo criado por **tmpfile()** é sempre aberto no modo `"w+b"`.
82. Consulte a **Seção 2.9**.
83. Consulte a **Seção 2.9**.
84. O resultado da operação não é especificado pelo padrão de C e, assim, depende de implementação.
85. Consulte a **Seção 2.9**.
86. Consulte a **Seção 2.10**.
87. A função **ungetc()** *escreve* em streams de *entrada*.
88. A função **ungetc()** é normalmente usada quando se deseja devolver um caractere, que não deve ser processado, ao stream no qual esse caractere foi lido.
89. Quando há duas ou mais chamadas dessa função sem instruções de leitura intervenientes entre elas.
90. Porque **ungetc()** decrementa o indicador de posição de arquivo.
91. Consulte a **Seção 2.11**.

92. Consulte a **Seção 2.11**.

93. (a) Streams de texto ou binários. (b) Streams de texto. (c) Streams binários. (d) Streams de texto.

94. (a) **fgetc()** e **fputc()**. (b) **fgets()** e **fputs()**. (c) **fread()** e **fwrite()**. (d) Funções das famílias scanf e printf.

95. (a) São partições conceituais (ou lógicas) de um arquivo. (b) É uma partição de um registro.

96. (a) Consulte a **Seção 2.11**. (b) Porque, quando um arquivo é aberto em modo de texto, pode ocorrer interpretação de caractere que representa quebra de linha. Quando um arquivo é aberto em modo binário, tal interpretação nunca ocorre.

97. Não.

98. Devido ao fato de o operador diferente (representado por `!=` em C) ter precedência maior do o operador de atribuição, à variável `c` será sempre atribuído `1` ou `0`.

99. Se a função **fgetc()** retornar **EOF**, esse valor será escrito no stream `streamSaida`. Para piorar, a expressão condicional do laço **while** não testa a ocorrência de erro de leitura ou escrita.

100.

```
rewind(streamA);

while (1) {
   c = fgetc(streamA);

   if (feof(streamA) || ferror(streamA))
      break;

   fputc(c, streamB);

   if (ferror(streamB))
      break;
}
```

101. Consulte a **Seção 2.11**.

102. Consulte a **Seção 2.11**.

103. Consulte a **Seção 2.11**.

104. **getchar()** só efetua leitura em **stdin**; **fgetc()** permite a especificação de um stream de entrada.

105. A função **fscanf()** deve ser usada em leitura formatada (em streams de texto), enquanto **fread()** deve ser usada em processamento de blocos (em streams binários).

106. O tipo da variável `c` deveria ser **int**, em vez de **char**.

107. Esse programa escreve a última linha do arquivo duas vezes na tela. Uma maneira de corrigir esse programa é substituindo o laço **while** desse programa por:

```
while(1) {
   fgets(linha, sizeof(linha), stream);

   if (feof(stream))
      break;

   fputs(linha, stdout);
}
```

108. Esse programa é abortado porque, quando o final do arquivo é atingido, a função **fgets()** retorna **NULL**, de modo que **fputs()** é chamada tendo esse valor como primeiro parâmetro. Como, para um string ser escrito, seus caracteres precisam ser acessados, essa última função aplica o operador de indireção sobre um ponteiro nulo, o que causa o aborto do programa.

109. (1) O tipo da variável `ch` deveria ser **int**, e não **char**. (2) A chamada de **feof()** deveria ocorrer antes do processamento do caractere, e não depois.

110. Consulte a **Seção 2.11**.

111. Consulte a **Seção 2.11.2**.

112. **stdin**.

113. Se os dois nomes de arquivo recebidos como argumentos pelo programa forem os mesmos e existir um arquivo com esse nome, seu conteúdo será destruído.

114. Consulte a **Seção 2.11**.

115. Consulte a **Seção 2.11**.

116. Zero indica que a chamada dessa função foi bem-sucedida; um valor diferente de zero indica o contrário.

117. Para streams binários, o valor retornado por **ftell()** representa o número de bytes calculado a partir do início do arquivo. Para streams de texto, o valor retornado por **ftell()** é dependente de implementação.

118. Consulte a **Seção 2.11**.

119. (a) Chamar uma função de posicionamento. (b) Chamar uma função de posicionamento ou **fflush()**.

120. Consulte a **Seção 2.11.3**.

121. (a) Antes da inserção de um registro, consulta-se a estrutura de dados `removidos` (v. **Seção 2.11.3**) para verificar se existe algum registro logicamente removido. Se esse for o caso, substitui-se o primeiro registro removido encontrado pelo novo registro. (b) Se o número de registros inseridos for maior do que ou igual ao número de registros removidos, ao final do programa o arquivo de dados não precisará ser reconstruído.

122. Consulte a **Seção 2.11.2**.

123. Consulte a **Seção 2.11.2**.

124. Consulte a **Seção 2.12**.

125. Consulte a **Seção 2.12**.

126. A função **rewind()** é recomendada quando se deseja garantir que o processamento de um arquivo começa em seu primeiro byte. (Mas, o uso de **fseek()** tem preferência.)

127. (a) Para garantir que a leitura começa no início do stream. (b) Porque, nesse caso, com certeza, o apontador de posição do arquivo aponta para o primeiro byte desse arquivo.

128. Suponha que `stream` é um stream que permite acesso direto. Então a chamada de **fseek()**: `fseek(stream, 0, SEEK_SET)` pode ser usada em substituição à chamada de **rewind()**: `rewind(stream)`.

129. Porque a função **fseek()** permite verificar quando ela é bem-sucedida, o que não é caso de **rewind()**.

130. Consulte a **Seção 2.13**.

131. Consulte a **Seção 2.13**.

132. Consulte a **Seção 2.13**.

133. Consulte a **Seção 2.13**.

134. Consulte a **Seção 2.14**.

135. Consulte a **Seção 2.14**.

136. Consulte a **Seção 2.14**.

137. Consulte a **Seção 2.14**.

138. Consulte a **Seção 2.14**.

139. Consulte a **Seção 2.14**.

140. Os registros devem ter tamanho fixo (i.e., todos eles devem ter o mesmo tamanho).

141. Consulte a **Seção 2.15.1**.

142. Sim. Consulte a **Seção 2.15.2**.

143. Consulte a **Seção 2.15.2**.

144. (a) Consulte a **Seção 2.15.3**. (b) Idem.

Capítulo 3 — Busca Linear em Memória Principal

1. Consulte a **Seção 3.1**.
2. Consulte a **Seção 3.1**.
3. Consulte a **Seção 3.1**.
4. Quando se procura um livro usando um catálogo de biblioteca, se a busca for bem-sucedida, obtém-se uma indicação (seção da biblioteca, prateleira, etc.) de onde o livro se encontra. De posse dessa informação, o usuário obtém finalmente o livro procurado. Uma busca com chave externa funciona de modo análogo.
5. (a) Sim. (b) Sim.
6. Consulte a **Seção 3.1**.
7. Consulte a **Seção 3.1**.
8. Quando a chave é secundária e a busca retorna um registro de cada vez.
9. (a) Consulte a **Seção 3.1**. (b) Lista indexada ordenada e sem ordenação, lista encadeada e lista com saltos.
10. Consulte a **Seção 3.1**.
11. Consulte a **Seção 3.2**.
12. Para que o programador não precise digitar os valores dos registros que constituem as tabelas de busca.
13. Consulte a **Seção 3.3**.
14. Porque talvez todas as chaves precisem ser comparadas.
15. (a) $\theta(n)$. (b) Não, porque, em qualquer situação, todas as chaves precisam ser comparadas.
16. Como a tabela é alocada dinamicamente, é de bom alvitre ter uma função que libere o espaço ocupado pela tabela, que é o que faz a função `DestroiTabelaIdx()`.
17. (a) Por causa do uso de **realloc()** que, no pior caso, tem custo $\theta(n)$. (b) $\theta(n)$.
18. Porque, nessa abordagem, a remoção física de um registro requer reconstrução do arquivo (mas existem outras abordagens).
19. (a) 100. (b) 50. (c) 50.
20. Faça você mesmo.
21. Idem.
22. Se a chave não se encontra na tabela, $2 \cdot n + 1$ comparações são necessárias: duas em cada execução do corpo do laço e uma que determina o final do laço. Suponha agora que a chave se encontra na posição i ($1 \leq i < n$) da tabela. Como, por hipótese, qualquer valor de i é equiprovável, o número médio de comparações quando a chave se encontra na tabela é dado por:

$$\frac{\sum_{i=1}^{n} 2i}{n} = \frac{2n(n+1)}{2n} = n+1$$

Portanto o número esperado de comparações efetuadas para encontrar uma chave é dado por:
$[(2 \cdot n + 1) + (n + 1)]/2 = (3 \cdot n + 2)/2$
A primeira expressão entre parênteses é decorrente da situação quando a chave não se encontra na tabela enquanto a segunda expressão entre parênteses é derivada do caso em que a chave é encontrada.
23. (a) 100. (b) 50. (c) 50.
24. (a) 50 (v. Prova do **Teorema 3.3**). (b) 50. (c) 50.
25. $\theta(1)$.
26. Porque, se não houver redimensionamento da tabela indexada, em ambos os casos o custo de inserção é $\theta(1)$.
27. Consulte a **Seção 3.4**.

28. Consulte a **Seção 3.4**.
29. Porque elas criam expectativas que nem sempre são satisfeitas.
30. Porque essa operação pode requerer movimentação de quase todos os elementos da lista, de modo que o custo temporal da movimentação é $\theta(n)$.
31. (a) $\theta(1)$. (b) Não.
32. (a) $\theta(1)$ se a lista for encadeada; $\theta(n)$ se a lista for indexada. (b) Sim.
33. No pior caso, o custo temporal de uma busca sem movimentação é $\theta(n)$. O custo da operação de movimentação também é $\theta(n)$. Portanto o custo de uma busca com movimentação é $\theta(n)$, que é o mesmo custo de uma busca sem movimentação.
34. Faça você mesmo.
35. Consulte a **Seção 3.5**.
36. Suponha que a chave de busca seja menor do que a chave que se encontra no meio da tabela. Então, de acordo com o algoritmo de busca binária, a busca deve prosseguir na metade inferior da tabela. Mas, se a tabela não estiver ordenada, a chave procurada poderá se estar na metade superior e nunca será encontrada. Um raciocínio semelhante pode ser aplicado se a chave de busca for maior do que a chave que se encontra no meio da tabela.
37. O algoritmo de busca binária sempre compara a chave de busca com a chave que se encontra no meio de uma tabela. Portanto, se houver chaves duplicadas, essa chave do meio pode não ser a primeira chave da tabela que casa com a chave de busca. Um exemplo trivial é uma lista na qual todas as chaves são iguais.
38. Porque busca binária requer acesso direto aos elementos da lista e lista encadeada só permite acesso sequencial.
39. Consulte a **Seção 3.5**.
40. Consulte a **Seção 3.5**.
41. (a) Leia os comentários incluídos na função. (b) Essa função compara dois elementos da tabela de busca.
42. (a) O índice é `5` e é feita apenas uma comparação. (b) Três comparações.
43. (a) Na prática, uma pessoa não começa procurando um nome que começa, por exemplo, com Z na metade de uma lista telefônica. Ou seja, uma pessoa normal sabe que, nesse caso, o nome está mais próximo do final da lista. (b) Porque, se o elemento que se está procurando começa com uma letra próxima ao início do alfabeto, abre-se o dicionário próximo ao seu início, se a palavra começa com uma letra próxima ao final do alfabeto, abre-se o dicionário próximo ao seu final e assim por diante.
44. A primeira expressão é mais eficiente porque contém uma operação a menos, mas pode causar overflow. A segunda expressão não oferece esse perigo.
45. (a) Consulte a **Seção 3.5.2**. (b) Consulte a **Seção 3.5.2**. (c) Consulte a **Seção 3.5.3**.
46. Consulte a **Seção 3.5.3**.
47. Consulte a **Seção 3.5.3**.
48. (a) O índice é `3` e são feitas quatro comparações. (b) Quatro comparações.
49. Consulte a **Seção 3.5.3**.
50. Siga a prova do **Caso 1** como modelo.
51. Consulte a **Seção 3.6**.
52. Consulte a **Seção 3.6.1**.
53. Consulte a **Seção 3.6**.
54. Consulte a **Seção 3.6**.
55. Consulte a **Seção 3.6**.
56. Consulte a **Seção 3.6**.
57. Porque criar e manter uma lista dessa natureza tem custo computacional muito elevado.

58. (a) Consulte a **Seção 3.6**. (b) Porque qualquer busca termina nesse nível.
59. Consulte a **Seção 3.6.1**.
60. Consulte a **Seção 3.6**.
61. Consulte a **Seção 3.6**.
62. Consulte a **Seção 3.6**.
63. Consulte a **Seção 3.6**.
64. A cabeça da lista desempenha o papel de sentinela.
65. Consulte a **Seção 3.6.2**.
66. `S` → `25` → `S` → `37` → `29` (`S` = sentinela)
67. `S` → `25` → `S` → `37` → `29` → `37` (`S` = sentinela)
68. V. **Figura E–2**.

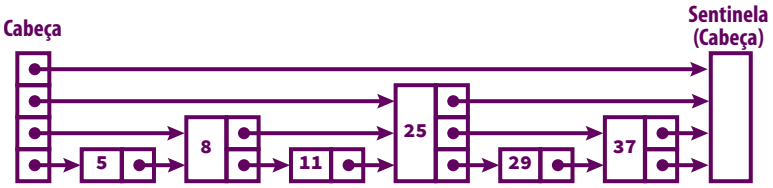

Figura E–2: Questão 68 — Capítulo 3

69. V. **Figura E–3**.

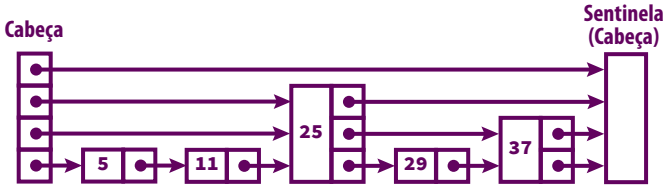

Figura E–3: Questão 69 — Capítulo 3

70. Acessando cada nó no primeiro nível da lista e contando quantos nós são acessados.
71. O fato de duas estruturas de dados apresentarem o mesmo custo espacial não significa que eles usam a mesma quantidade de espaço. Consulte o **Capítulo 6** do **Volume 1**.
72. Consulte a **Seção 3.7.1**.
73. Consulte a **Seção 3.7.1**.
74. (a) Consulte a **Seção 3.7.1** (**Figura 3–24**). (b) Consulte a **Seção 3.7.1** (**Figura 3–25**).
75. (1) Encontre a posição (índice) de uma ocorrência da chave usando busca binária. (2) A partir dessa posição, acesse as chaves que antecedem a referida ocorrência até encontrar uma chave diferente. (3) Faça o mesmo com as chaves que sucedem a referida ocorrência. O custo temporal da etapa (1) é $\theta(\log n)$ e o custo temporal das etapas (2) e (3) é $\theta(s)$. Logo o custo temporal de toda a operação é $\theta(\log n + s)$.
76. Encontre a chave usando busca binária. Como a chave de busca é primária e a tabela está ordenada, o piso será a chave do elemento antecessor daquele que contém a chave de busca e o teto será a chave do elemento sucessor daquele que contém a chave de busca.
77. (a) Encontre a chave usando busca binária. Para encontrar o piso, efetue uma busca sequencial descendente na metade inferior da tabela a partir da posição da chave de busca até encontrar a primeira chave diferente da chave de busca. Para encontrar o teto, use um raciocínio semelhante usando a metade superior da tabela. (b) $\theta(n)$. (c) $\theta(\log n)$.
78. **1.** Encontre a chave inicial do intervalo usando busca binária. **2.** Encontre as $s - 1$ chaves subsequentes usando busca sequencial. O custo do **Passo 1** é $\theta(\log n)$ e o custo do **Passo 2.** é $\theta(s)$, de modo que os dois passos combinados apresentam custo temporal $\theta(\log n + s)$.

Capítulo 4 — Busca Hierárquica em Memória Principal

1. Consulte a **Seção 4.1**.
2. Consulte a **Seção 4.1**.
3. Basta apresentar exemplos que contrariem as afirmações, como as árvores da **Figura E–4**.

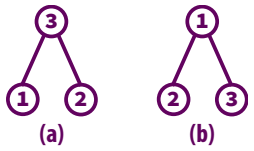

Figura E–4: Questão 3 — Capítulo 4

4. (a) Encontrando a menor chave da subárvore que tem como raiz o filho direto do nó. (b) Encontrando a maior chave da subárvore que tem como raiz o filho esquerdo do nó.
5. (a) A folha mais à esquerda na árvore. (b) A folha mais à direita na árvore.
6. (a) Se o sucessor imediato tivesse filho esquerdo, esse filho seria o sucessor imediato. (b) Use um raciocínio similar.
7. (a) É necessário usar recursão ou uma pilha, pois o sucessor é um dos ancestrais do nó. (b) Idem.
8. O resultado é mostrado na **Figura E–5**.

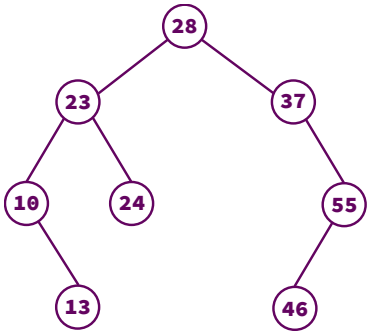

Figura E–5: Questão 8— Capítulo 4

9. Consulte a **Seção 4.1**.
10. Sim. Substituir o nó a ser removido por seu antecessor imediato tem o mesmo efeito e é igualmente eficiente.
11. Sim, a não ser que ele contenha a maior chave da árvore. Nesse caso, o sucessor encontra-se num dos ancestrais do nó.
12. Consulte a **Seção 4.1**.
13. Consulte a **Seção 4.1**.
14. A árvore resultante é mostrada na **Figura E–6**.

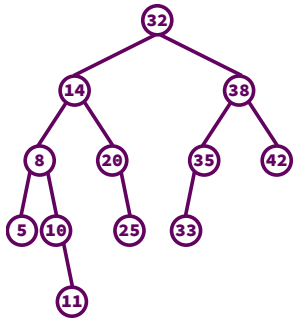

Figura E–6: Questão 14 — Capítulo 4

15. A árvore resultante é mostrada na **Figura E–7**.

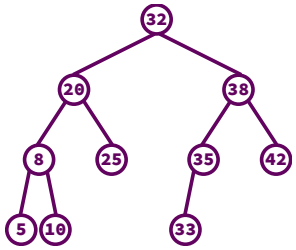

Figura E–7: Questão 15 — Capítulo 4

16. A árvore resultante é mostrada na **Figura E–8**.

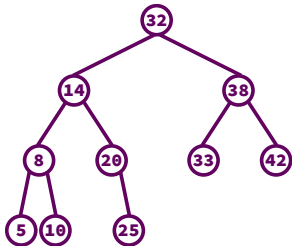

Figura E–8: Questão 16 — Capítulo 4

17. A árvore resultante é mostrada na **Figura E–9**.

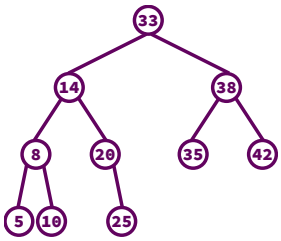

Figura E–9: Questão 17 — Capítulo 4

18. Considere todas as permutações de ordem das chaves *1*, *2* e *3* (que são *3! = 6* permutações) e você obterá todas as árvores binárias de busca possíveis (v. **Figura E–10**).

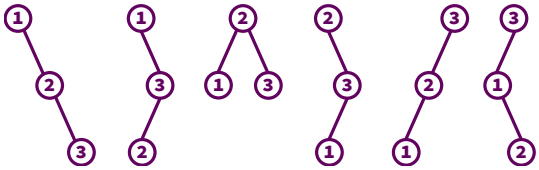

Figura E–10: Questão 18 — Capítulo 4

19. Esse programa cria uma árvore de busca tendo como chaves os elementos do array `alfabeto[]`. Como essas chaves estão em ordem crescente, a árvore obtida será inclinada à direita.

20. Usando essa abordagem, primeiro insere-se a chave que se encontra no meio do array. Depois repete-se o procedimento para as metades inferior e superior do array. Desse modo, a árvore obtida será bem balanceada.

21. (a) Por meio de um caminhamento. (b) $\theta(n)$. (c) Se o nó que armazena a chave a ser removida tiver, no máximo, um filho, procede-se como numa remoção em árvore binária. Se esse nó tiver dois filhos, substitui-se sua chave pela chave do nó mais à esquerda de sua subárvore esquerda e, em seguida, remove-se esse último nó como no caso trivial (pois ele tem no máximo um filho).(d) Não faz sentido balancear uma *CrazyTree* porque o custo de busca em tal árvore continuará sendo $\theta(n)$ mesmo se ela for balanceada.

22. A árvore resultante é mostrada na **Figura E–11**.

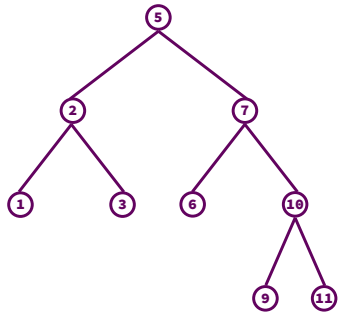

Figura E–11: Questão 22 — Capítulo 4

23. (a) A árvore resultante é mostrada na **Figura E–12 (a)**. A árvore resultante é mostrada na **Figura E–12 (b)**.

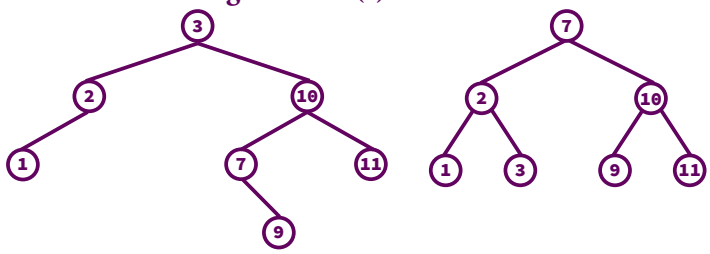

Figura E–12: Questão 23 — Capítulo 4

24. (a) Como se supõe que se trata de uma árvore binária de busca, um caminhamento em ordem infixa acessa as chaves armazenadas dessa árvore em ordem crescente, de acordo com o **Teorema 4.1**. Substituindo-se um nó pelo nó que contém a chave sucessora (ou antecessora) resulta na mesma sequência crescente de chaves, mas sem a chave removida. Portanto a árvore continua sendo uma árvore binária de busca, de acordo com o mesmo teorema.

25. Cinco dessas permutações são as seguintes:
- *5*, *2*, *10*, *1*, *3*, *7*, *11*, *9*
- *5*, *2*, *10*, *3*, *1*, *11*, *7*, *9*
- *5*, *10*, *2*, *1*, *3*, *7*, *11*, *9*
- *5*, *10*, *2*, *3*, *1*, *7*, *11*, *9*
- *5*, *10*, *2*, *3*, *1*, *7*, *9*, *11*

26. (a) Essa função testa se a chave do filho esquerdo de cada nó é menor do que a chave de seu pai e se a chave do filho direito de cada nó é maior do que a chave do seu pai. (b) De acordo com essa função, a árvore da **Figura E–13** é uma árvore binária de busca (verifique isso).

27. Vantagem: as remoções serão razoavelmente mais rápidas. Desvantagens: além de haver desperdício de memória, as operações de busca e inserção serão mais lentas.

28. Em ordem decrescente, pois, desse modo, as chaves com maiores frequências estarão mais próximas da raiz da árvore.

29. Utilize o mesmo raciocínio empregado na resolução da questão **18**.

30. Para uso didático.

31. Melhor caso: $\theta(\log n)$. Pior caso: $\theta(n)$. Caso médio: $\theta(n)$.

32. Basta apresentar uma árvore binária cuja menor chave esteja na folha mais à esquerda que não seja árvore de busca, como a árvore da **Figura E–13**.

33. Caminhando-se numa árvore binária de busca em ordem infixa e inserindo-se as chaves sequencialmente no array à medida que os nós são visitados.

34. O antecessor imediato desse nó é seu primeiro ancestral esquerdo, como mostra a **Figura E–14**.

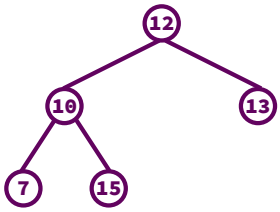

Figura E–13: Questão 32 — Capítulo 4

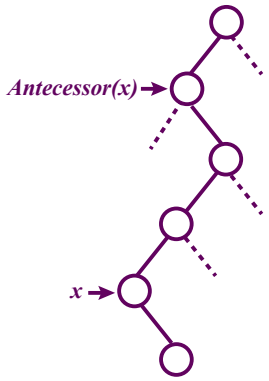

Figura E–14: Questão 34 — Capítulo 4

35. Consulte a **Seção 4.2**.

36. Consulte a **Seção 4.2**.

37. Consulte a **Seção 4.2**.

38. A seguinte sequência de rotações produz o resultado desejado [*RD(n)* significa rotação direita do nó com conteúdo *n*]:
RD(23) → *RD(15)* → *RD(10)* → *RD(10)*

39. Subárvores.

40. A resposta pode ser deduzida examinando-se a **Figura 4–25** na **Seção 4.2**.

41. (a) V. **Figura E–15 (a)**. (b) V. **Figura E–15 (b)**.

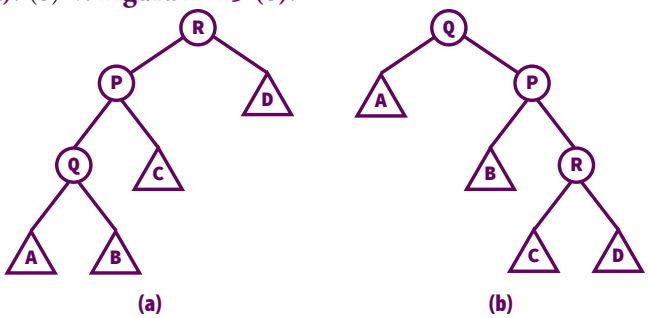

Figura E–15: Questão 41 — Capítulo 4

42. *θ(1)*.

43. A seguinte sequência de rotações produz o resultado desejado [*RD(n)* e *RE(n)* significam, respectivamente, rotação direita e rotação esquerda do nó com conteúdo *n*]:
RE(21) → *RD(21)* → *RD(21)*

44. Consulte a **Seção 4.3**.

45. Uma árvore perfeitamente balanceada é uma árvore binária repleta e, como é mostrado no **Capítulo 12** do **Volume 1**, seu número de nós é exatamente $2^p - 1$.

46. No **Capítulo 12** do **Volume 1**, mostra-se que o número de folhas numa árvore binária repleta é $(n + 1)/2$, em que n é número total de nós. Como o número total de nós é dado por $2^p - 1$, tem-se que o número de folhas numa árvore binária repleta é 2^{p-1}.

47. Porque essas árvores têm altura $\theta(\log n)$, de modo que as operações de busca, inserção e remoção têm custo temporal $\theta(\log n)$.

48. Considerando-se a árvore da **Figura E–16 (a)** e removendo-se as chaves *16*, *23*, *19* e *26* obtém-se a árvore da **Figura E–16 (b)**, que é degenerada.

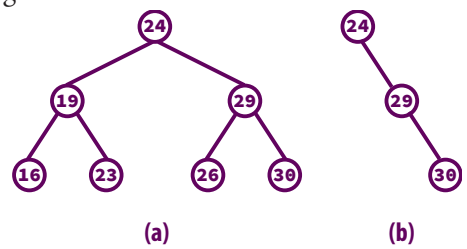

Figura E–16: Questão 48 — Capítulo 4

49. (a) Essas árvores têm altura $\theta(\log n)$. (b) São difíceis (ou mesmo impossíveis) de construir e manter.

50. A profundidade de a decresce de *1*, a profundidade de c é acrescida de *1* e a profundidade de b permanece a mesma.

51. Consulte a **Seção 4.4**.

52. Consulte a **Seção 4.4** e faça sua parte.

53. Árvores AVL são relativamente fáceis de balancear. Além disso, árvores binárias perfeitamente balanceadas são, frequentemente, impossíveis de obter na prática devido à quantidade de chaves disponíveis.

54. Consulte a **Seção 4.4**.

55. Por definição, uma árvore AVL é uma árvore binária de busca na qual as alturas de quaisquer duas subárvores nunca diferem em mais de *1*. O balanceamento de um nó é definido como a altura de sua subárvore esquerda menos a altura de sua subárvore direita. Portanto existem três hipóteses a considerar: (1) a altura da subárvore esquerda de um nó é a e a altura de sua subárvore direita é $a + 1$, (2) a altura da subárvore direita de um nó é a e a altura de sua subárvore esquerda é $a + 1$ ou (3) as duas subárvores de um nó possuem a mesma altura a. No caso (1), o balanceamento é -1; no caso (2), o balanceamento é *1* e, no caso (3), o balanceamento é *0*.

56. Consulte a **Seção 4.4**.

57. Para rebalancear a árvore.

58. Consulte a **Seção 4.4**.

59. V. **Figura E–17**.

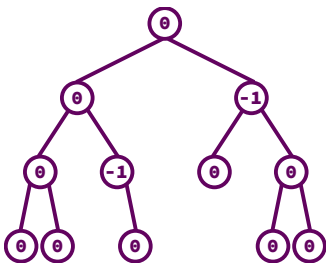

Figura E–17: Questão 59 — Capítulo 4

60. Na **Figura E–18**, esses nós são rotulados com **B**.

61. Consulte a **Seção 4.4**.

62. Consulte a **Seção 4.4**.

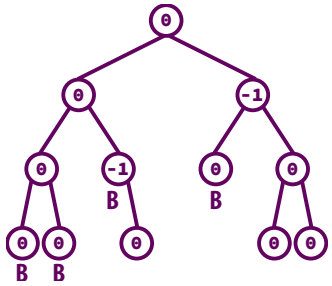

Figura E–18: Questão 60 — Capítulo 4

63. *D1*, *D2*: esquerda-esquerda; *D3*, *D4*: esquerda-direita; *D5*, *D6*: esquerda-esquerda; *D7*, *D8*: esquerda-direita; *D9*, *D10*: direita-esquerda; *D11*, *D12*: direita-direita.

64. (a) Esquerda-esquerda. (b) Esquerda-direita. (c) Direita-esquerda. (d) Direita-direita.

65. V. **Figura E–19**.

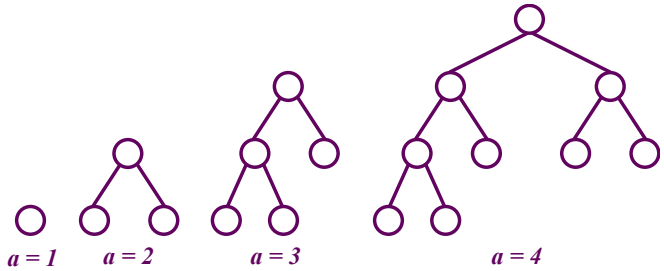

Figura E–19: Questão 65 — Capítulo 4

66. $\theta(\log n)$ para as três operações.

67. (a) V. **Figura E–20**. (b) Sim.

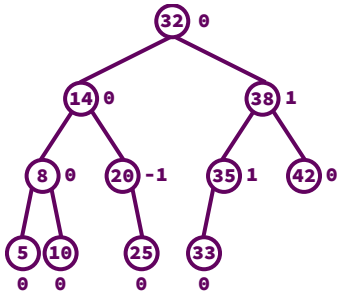

Figura E–20: Questão 67 — Capítulo 4

68. V. **Figura E–21**.

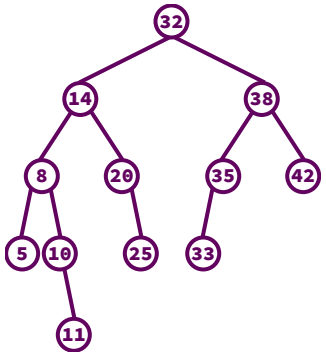

Figura E–21: Questão 68 — Capítulo 4

69. V. **Figura E–22**.

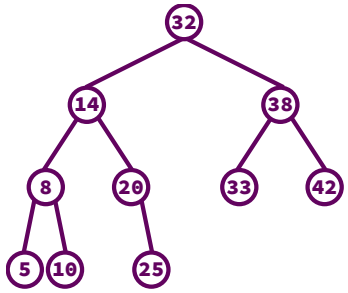

Figura E–22: Questão 69 — Capítulo 4

70. V. **Figura E–23**.

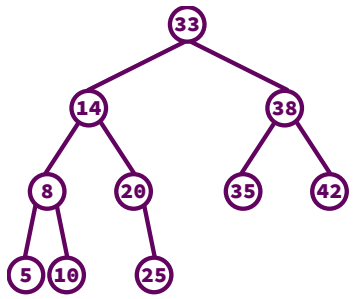

Figura E–23: Questão 70 — Capítulo 4

71. Não. O custo temporal das operações básicas será sempre $\theta(\log n)$.

72. V. **Figura E–24**.

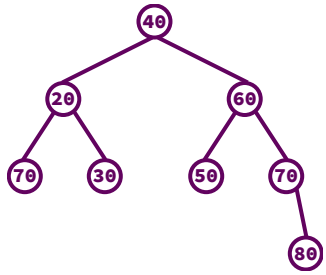

Figura E–24: Questão 72 — Capítulo 4

73. (a) Não. Se a árvore for inclinada, o balanceamento de sua raiz será *2* ou *–2* e, portanto, ela não será AVL. (b) Não (usando um raciocínio similar).

74. Essa remoção torna desbalanceado o nó com chave *3*. O rebalanceamento desse último nó é feito por meio de uma rotação direita como se vê na **Figura E–25**.

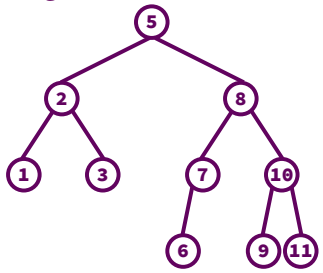

Figura E–25: Questão 74 — Capítulo 4

75. V. **Figura E–26**.

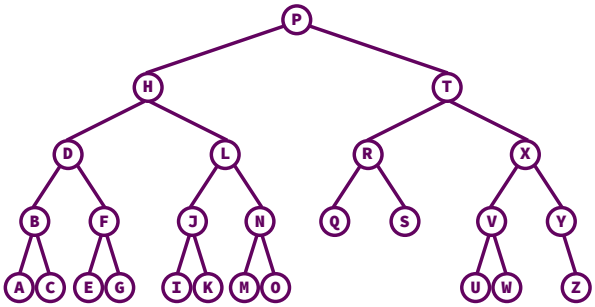

Figura E–26: Questão 75 — Capítulo 4

76. (a) O número mínimo de nós de uma árvore AVL é dado por: $n_a = n_{a-1} + n_{a-2} + 1$, em que a é a altura da árvore, $a > 0$, $n_1 = 1$ e $n_2 = 2$. Portanto $n_3 = n_2 + n_1 + 1 = 4$, $n_4 = n_3 + n_2 + 1 = 7$. Agora que você já conhece os números de nós das árvores AVL mínimas, construa-as você mesmo. (b) $n_5 = n_4 + n_3 + 1 = 12$.

77. A remoção do nó com chave *6* requer rebalanceamento da subárvore cuja raiz contém a chave *7*. Esse rebalanceamento é efetuado por meio de uma rotação esquerda, resultando na árvore da **Figura E–27**.

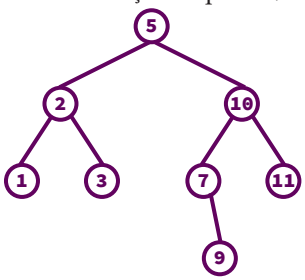

Figura E–27: Questão 77 — Capítulo 4

78. O rebalanceamento envolve rotações direita e esquerda (nessa ordem). A árvore resultante é mostrada na **Figura E–28**.

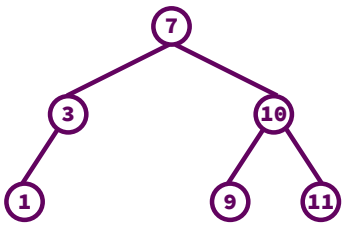

Figura E–28: Questão 78 — Capítulo 4

79. O rebalanceamento envolve uma simples rotação direita. A árvore resultante é mostrada na **Figura E–29**.

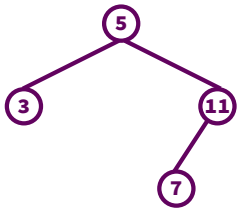

Figura E–29: Questão 79 — Capítulo 4

80. Consulte a **Seção 4.5**.

81. Consulte a **Seção 4.5**.

82. V. **Figura E–30**.

83. V. **Figura E–31**.

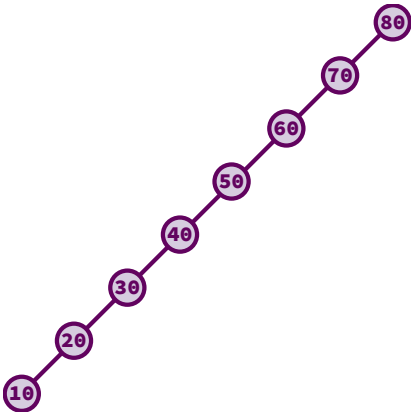

Figura E–30: Questão 82 — Capítulo 4

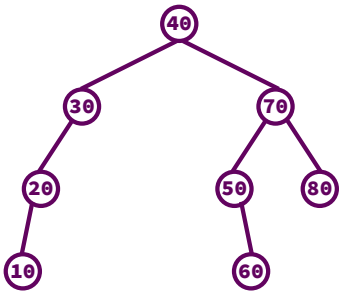

Figura E–31: Questão 83 — Capítulo 4

84. V. **Figura E–32**.

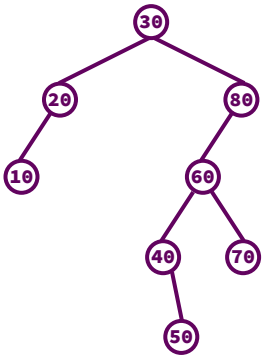

Figura E–32: Questão 84 — Capítulo 4

85. V. **Figura E–33**.

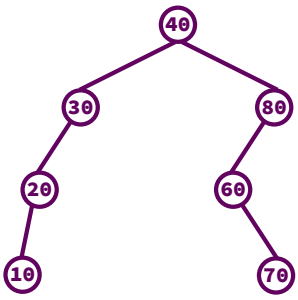

Figura E–33: Questão 85 — Capítulo 4

86. Resposta curta: árvore afunilada não tem compromisso com balanceamento, como ocorre com árvores AVL. Resposta longa: releia a **Seção 4.5**.

87. Consulte a **Seção 4.5**.

88. Consulte a **Seção 4.5**.

89. Consulte a **Seção 4.5**.

90. Consulte a **Seção 4.5**.

91. Consulte a **Seção 4.5**.

92. V. **Figura E–34**.

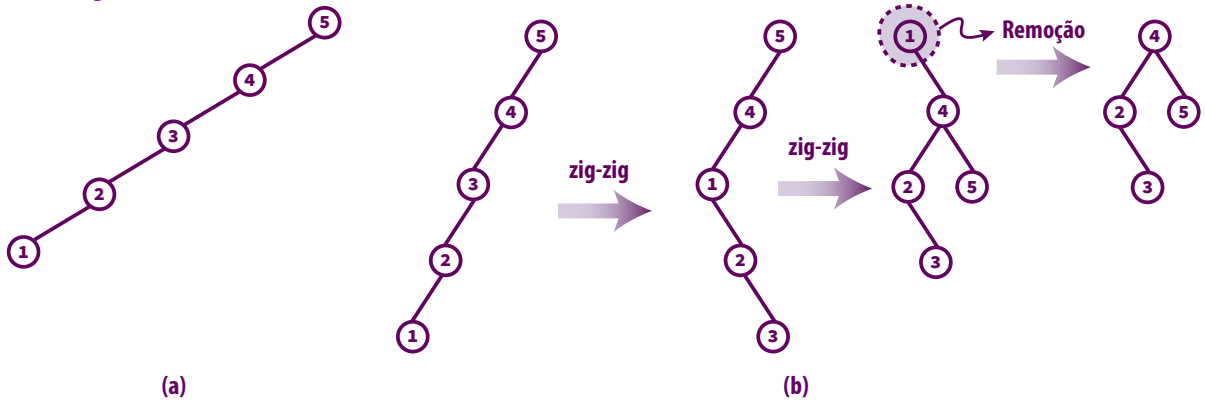

Figura E–34: Questão 92 — Capítulo 4

93. V. **Figura E–35**.

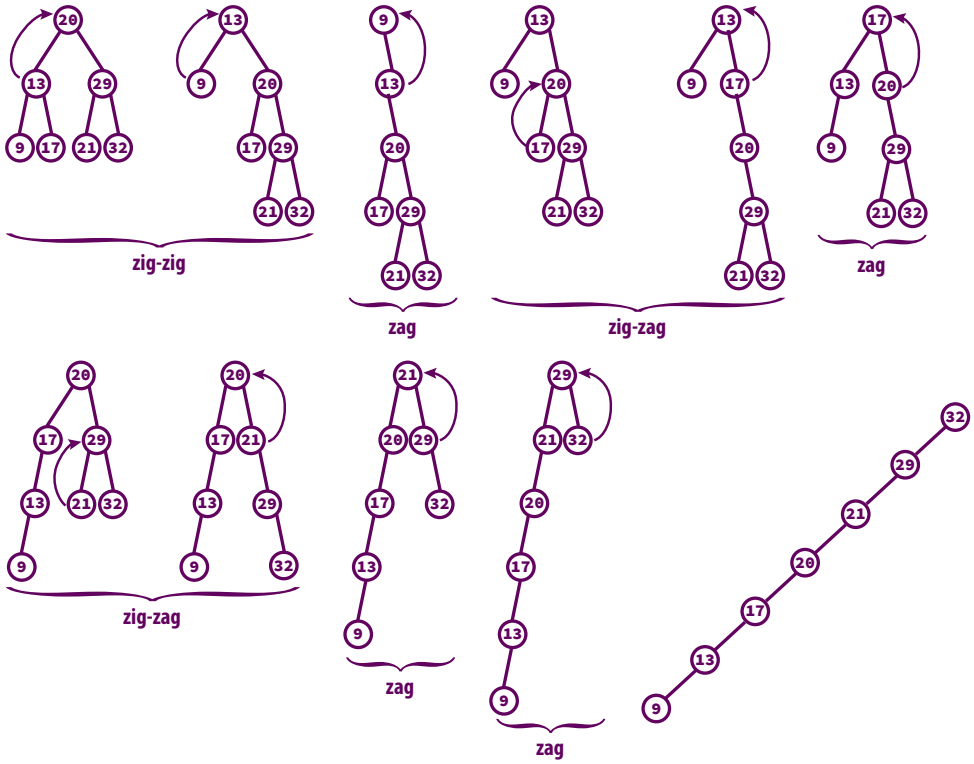

Figura E–35: Questão 93 — Capítulo 4

94. V. **Figura E–36**.

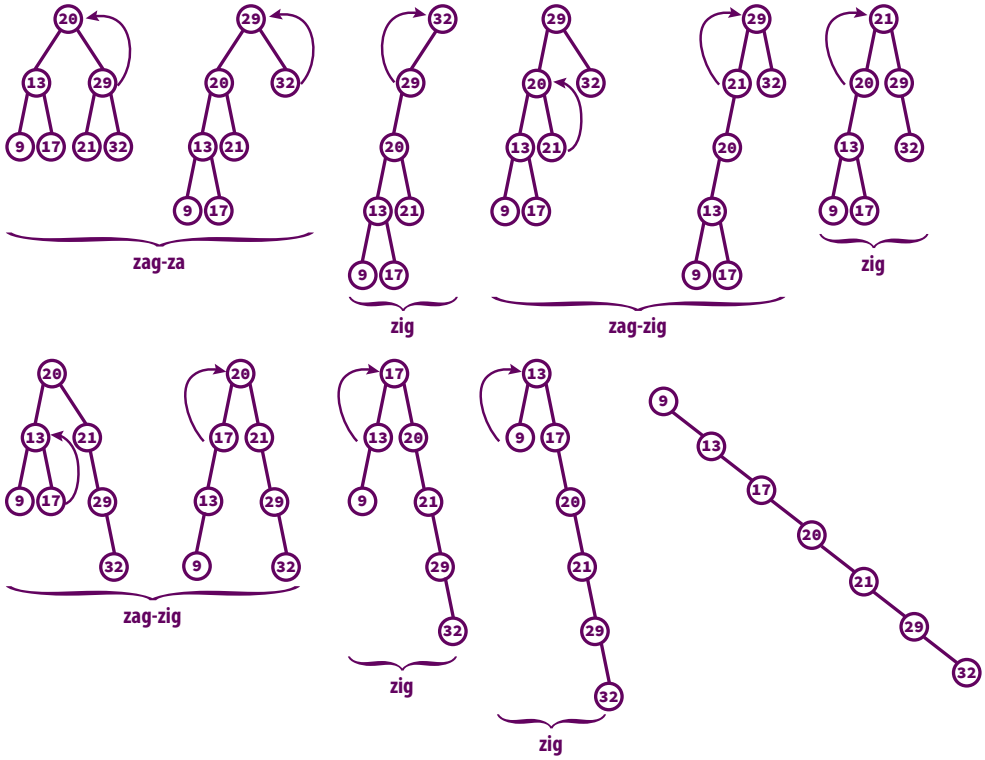

Figura E–36: Questão 94 — Capítulo 4

95. Consulte a **Seção 4.5.1**.

96. Estude os exemplos da **Seção 4.5.1** e apresente seus próprios exemplos.

97. (a) zag-zig e zig-zig (nessa sequência, sendo c_1 o alvo). (b) A sequência de rotações é a seguinte [*RD(n)* e *RE(n)* significam, respectivamente, rotação direita e rotação esquerda do nó com conteúdo *n*]:

$RD(c_1) \rightarrow RE(c_1) \rightarrow RE(c_1) \rightarrow RE(c_1)$

98. (a) Consulte a **Seção 4.5.2**. (a) Consulte a **Seção 4.5.2**. (c) Tente implementar afunilamento ascendente.

99. $\theta(n)$.

100. (a) V. **Figura E–37**. (b) Não. A árvore resultante é aquela mostrada na **Figura E–38**.

101. Porque todas elas são árvores binárias de busca.

102. Consulte a **Seção 4.6**.

103. Quando é admissível que, eventualmente, ocorram operações com custo $\theta(n)$.

104. Quando o custo de qualquer operação deve sempre ser $\theta(\log n)$.

105. Consulte a **Seção 4.7.2**.

106. Consulte a **Seção 4.7.3**.

107. (a) Essa variável armazena o endereço do último nó visitado no caminhamento em ordem infixa efetuado pela função. (b) Porque, se ela tivesse duração automática, ela seria reiniciada a cada chamada recursiva da função.

108. Consulte a **Seção 4.7.4**.

109. $\theta(n)$.

110. $\theta(n)$ (se a árvore for ordinária ou afunilada) ou $\theta(\log n)$ (se a árvore for AVL).

111. $\theta(n)$.

112. $\theta(n)$.

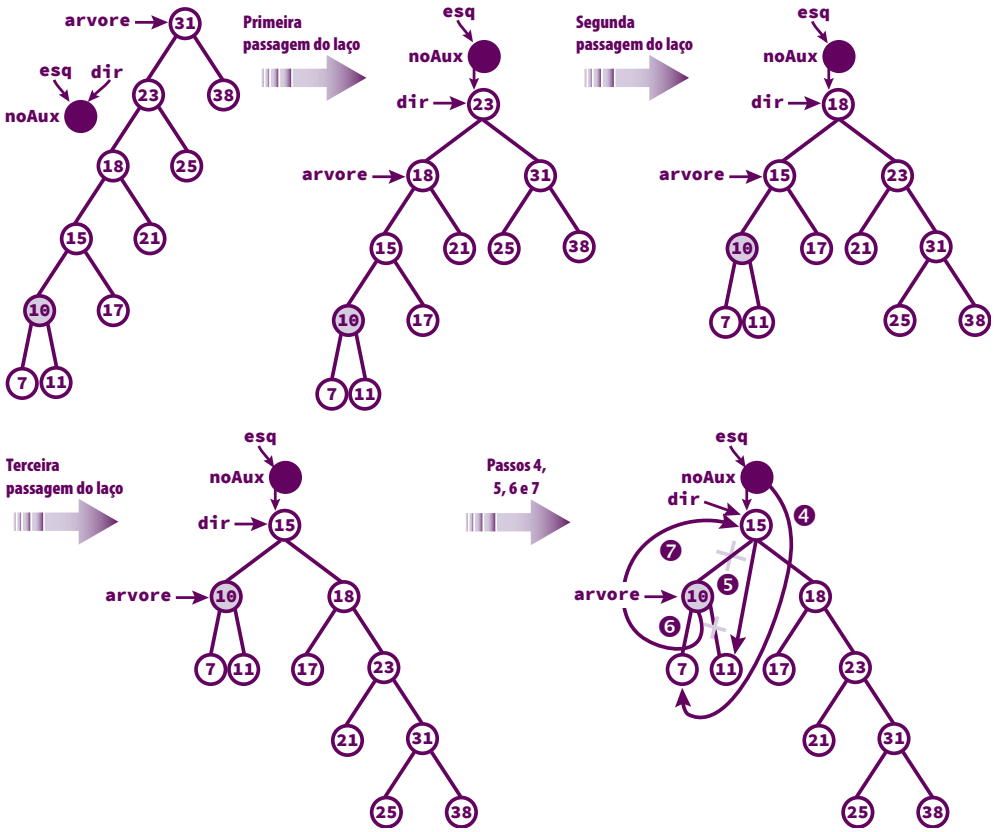

FIGURA E–37: QUESTÃO 100 (A) — CAPÍTULO 4

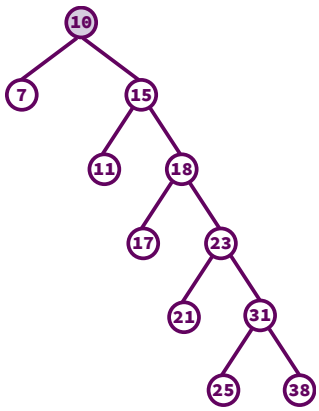

FIGURA E–38: QUESTÃO 100 (B) — CAPÍTULO 4

Capítulo 5 — Análise Amortizada

1. Consulte a **Seção 5.1**.
2. Consulte a **Seção 5.1**.
3. Consulte a **Seção 5.1**.
4. Consulte a **Seção 5.1**.
5. Consulte a **Tabela 5–1**.
6. Não faz sentido considerar uma sequência de operações de ordenação.

7. (a) Tabela de dispersão e pilha dinâmica. (b) Lista indexada estática e lista com saltos.

8. Consulte a **Seção 5.2**.

9. Consulte a **Seção 5.2.2**.

10. Não. Consulte a **Seção 5.2.2**.

11. É um somatório no qual todos os termos, com exceção do primeiro e do último, são mutuamente cancelados. Consulte o **Apêndice B** do **Volume 1**.

12. Consulte a **Seção 5.2.3**.

13. Informalmente, em conformidade com a Física, um objeto armazena energia potencial (mecânica) de acordo com sua posição espacial. Analogamente, segundo a análise amortizada, uma estrutura de dados armazena *energia potencial* de acordo com sua configuração.

14. (a) Consulte a **Seção 5.2.5**. (b) Essa estrutura de dados não tem nenhuma utilidade prática. Ela é usada apenas como uma ferramenta didática.

15. Consulte a **Seção 5.3.4**.

16. Consulte a **Seção 5.3.4**.

17. Consulte a **Seção 5.3.1**.

18. Consulte a **Seção 5.4**.

19. (a) V. **Figura E–39**. Nessa figura, números em círculos escuros representam tamanhos e números em círculos claros representam postos. (b) O potencial é *4*.

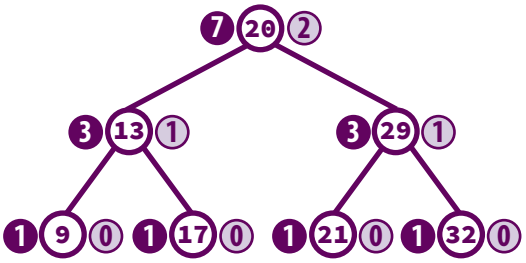

Figura E–39: Questão 19 — Capítulo 5

20. *O(n)*.

21. *O(log n)*.

22. (a) Potencial máximo: $O(n \cdot \log n)$; potencial mínimo: $O(1)$. (b) No máximo, $O(n \cdot \log n)$. (c) No máximo, $O(n \cdot \log n)$.

23. Siga os seguintes passos:

1. Suponha que o crédito atribuído a cada nó da árvore seja seu posto (conforme foi definido na **Seção 5.4**). Note que, se árvore estiver vazia ou contém apenas um nó, seu crédito será 0. À medida que a árvore cresce, seu saldo vai aumentando.
2. Prove, como um lema, que o número de moedas virtuais necessárias para efetuar o afunilamento de um nó x de uma árvore afunilada A é, no máximo, $3 \cdot [c(A) - c(x) + O(1)]$, em que $c(A)$ é o crédito da árvore (i.e., a soma dos créditos de todos os nós). Por razões de simetria, esse lema precisa considerar separadamente apenas as operações zig, zig-zig e zig-zag, como foi feito na **Seção 5.4**. Esse passo é o mais complicado e sugere-se que o leitor imite a sequência seguida na prova do **Teorema 5.3**.
3. Como teorema, prove que qualquer operação de afunilamento tem custo amortizado $O(\log n)$.
4. Prove, como corolário, que os custos amortizados de busca, inserção e remoção numa árvore afunilada são todos $O(\log n)$.

24. Não, pois essa é uma afirmação vaga. O que diferencia os custos temporais dessas estruturas são os adjetivos empregados para qualificá-los. O custo temporal de uma lista com saltos é *esperado*, o custo temporal de

uma árvore AVL é *garantido* e o custo temporal de uma árvore afunilada é *amortizado*. Portanto, dentre essas estruturas, aquela que realmente apresenta o melhor custo temporal é a árvore AVL.

25. Estude a prova do **Teorema 5.3**.

26. Estude a prova do **Teorema 5.3**.

27. Porque o potencial poderia aumentar até n durante o acesso a um nó.

Capítulo 6 — Busca Hierárquica em Memória Secundária

1. (a) Consulte a **Seção 6.1**. (b) Bancos de dados, sistemas de arquivos, etc.

2. (a) Consulte a **Seção 6.1**. (b) Consulte a **Seção 6.2**.

3. Consulte a **Seção 6.1**.

4. Porque, tipicamente, um nó de uma árvore multidirecional não possui apenas dois filhos, como ocorre com árvores binárias.

5. (a) Consulte a **Seção 6.1**. (b) Por definição, qualquer folha é uma semifolha. Resta mostrar que um nó incompleto que não seja folha não pode ser semifolha. Agora, numa árvore multidirecional de busca descendente, um nó só terá seu primeiro filho após possuir o número máximo de chaves (i.e., se ele for completo). Portanto um nó incompleto só será semifolha se ele for folha e um nó completo poderá ser semifolha mesmo sem ser folha. ■

6. Consulte a **Seção 6.1**.

7. Porque, normalmente, armazenando apenas chaves e as posições dos respectivos registros em arquivo, cada nó da árvore é capaz de armazenar um número bem maior de chaves do que seria o caso se registros inteiros fossem armazenados em cada nó. Para entender melhor esse argumento, consulte a **Seção 6.3**.

8. Consulte a **Seção 6.1.3**.

9. Consulte a **Seção 6.1.3**.

10. Uma desvantagem do método de inserção em árvores multidirecionais descendentes é que são criadas folhas contendo apenas uma chave e algumas folhas podem ser criadas antes que outras folhas estejam completas. Por isso, esse método pode causar grande desperdício de memória e fazer com que essa árvore torne-se profunda.

11. (a) Consulte a **Seção 6.1.4**. (b) Porque após essa operação, pode-se obter uma árvore com um nó incompleto que não é folha.

12. (a) Pela mesma razão pela qual árvores binárias ordinárias de busca: elas podem se tornar degeneradas. (b) Para facilitar a aprendizagem de árvores B, por exemplo.

13. Consulte a **Seção 6.2**.

14. Consulte a **Seção 6.2**.

15. Consulte a **Seção 6.2**.

16. Consulte a **Seção 6.2**.

17. Consulte a **Seção 6.2**.

18. Índice de registro num arquivo contendo partições (registros) de mesmo tamanho. Para entender melhor, consulte a **Seção 6.2**.

19. Usando um valor inteiro negativo.

20. Consulte a **Seção 6.3**.

21. Consulte a **Seção 6.3**.

22. (a) Consulte a **Seção 6.3.5**. (b) Consulte a **Seção 6.3.4**.

23. Consulte a **Seção 6.3.1**.

24. Essa constante indica uma posição inválida em arquivo, assim como **NULL** indica um endereço inválido em memória principal.

25. Consulte a **Seção 6.3.3**.

26. Consulte a **Seção 6.3.6**.

27. Consulte a **Seção 6.3.1**.

28. (a) Todos os elementos de um array têm o mesmo tamanho, de maneira que basta alinhar o primeiro elemento que os demais estarão alinhados. (b) Não.

29. Escrevendo cada campo da estrutura separadamente.

30. Consulte a **Seção 6.3.1**.

31. Consulte a **Seção 6.4**.

32. Consulte a **Seção 6.4**.

33. Numa árvore B, uma operação de inserção começa sempre numa folha e essa folha nunca deixa de ser folha. Quando ocorre uma divisão de folhas, uma chave e um ponteiro para a nova folha sobem para serem inseridos num nó interno. Nessa última inserção, o nó interno não passa a ser semifolha, pois ele recebe uma chave e um filho dessa chave que não é nulo. Portanto, numa árvore B, uma semifolha só pode ser folha. ■

34. Porque, certamente, essa raiz não caberá num único bloco. Assim, apesar de a raiz poder ser lida com uma única chamada de função [**fread**(), por exemplo] serão necessários múltiplos acessos ao meio de armazenamento para lê-la ou escrevê-la.

35. Consulte a **Seção 6.4**.

36. (a) Porque pode ser que ocorram divisões de nós que se propaguem até a raiz. (b) Porque quando um nó que se encontra no caminho é alterado, é preciso saber em que posição ele se encontra no arquivo para que se possa atualizá-lo. (c) V. implementação da função `InsereB()` na **Seção 6.4.5**.

37. Consulte a **Seção 6.4**.

38. Consulte a **Seção 6.4**.

39. As chaves `C`, `N`, `G` e `A` são inseridas normalmente na raiz. Quando se tenta inserir `H` na raiz, não se encontra mais espaço. Assim esse nó é dividido em dois nós, movendo-se `G` para cima e criando-se uma nova raiz. Note que, na prática, deixam-se `A` e `C` no corrente nó e coloca-se `H` e `N` num novo nó à direita do nó antigo. O resultado é mostrado na **Figura E–40**.

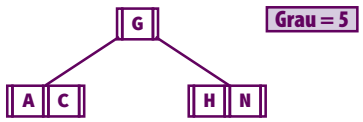

FIGURA E–40: QUESTÃO 39 — CAPÍTULO 6

40. V. **Figura E–41**.

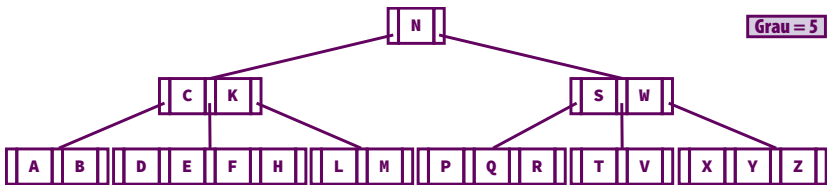

FIGURA E–41: QUESTÃO 40 — CAPÍTULO 6

41. (a) Descendo-se até a folha mais à esquerda da árvore, a menor chave é a primeira chave dessa folha. (b) Descendo-se até a folha mais à direita da árvore, a maior chave é a última chave dessa folha.

42. Antecessora.

43. Uma árvore B de altura *a* contendo o número máximo de nós possui a seguinte distribuição de nós por nível:

Número máximo de nós no nível *0*: 1 +
Número máximo de nós no nível *1*: G +
Número máximo de nós no nível *2*: G^2 +
⋮ ⋮ +
Número máximo de nós no nível $a-1$: G^{a-1}

Assim o número de nós dessa árvore B é dado por:

$$N_{\max} = \sum_{i=0}^{a-1} G^i = \frac{G^a - 1}{G - 1}$$

É fácil mostrar por indução (faça isso) que uma árvore B de altura *a* com todos os seus nós preenchidos armazena $n = G^a - 1$ chaves. Logo o número máximo de nós é dado por $N_{max} = n/(G - 1)$.

44. O que custa mais numa operação de busca é a transferência de dados entre memória principal e memória secundária. Além disso, na prática, o número de chaves num nó não é tão grande, de modo que o ganho obtido usando busca binária em detrimento a busca sequencial é irrelevante.

45. Só há duas árvores B de grau *6* possíveis contendo as chaves *1*, *2*, *3*, *4* e *5*, como mostra a **Figura E–42**.

Figura E–42: Questão 45 — Capítulo 6

46. A inserção de `M` requer uma divisão. Note que `M` é a chave média e por isso é movida para o nó-pai. A **Figura E–43** mostra isso.

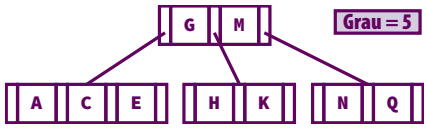

Figura E–43: Questão 46 — Capítulo 6

47. Quando a chave `Z` é acrescentada, a folha mais à esquerda deve ser dividida e a chave média `T` é movida para o nó-pai (v. **Figura E–44**).

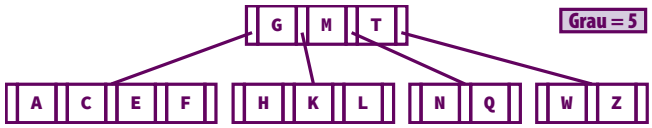

Figura E–44: Questão 47 — Capítulo 6

48. A inserção de `D` faz com que a folha mais à esquerda seja dividida. A chave `D` é a chave média e é aquela que deve ser movida para o nó-pai. As chaves `P`, `R`, `X` e `Y` são inseridas sem necessidade de divisão de nó, como mostra a **Figura E–45**.

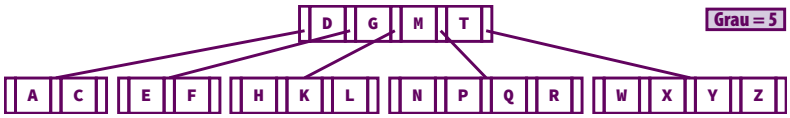

Figura E–45: Questão 48 — Capítulo 6

49. Quando a chave `S` é inserida, o nó contendo `N`, `P`, `Q` e `R` é dividido, com `Q` subindo para o nó-pai. Como esse nó-pai está completo, ele é dividido e `M` sobe para formar uma nova raiz. O resultado é mostrado na **Figura E–46**.

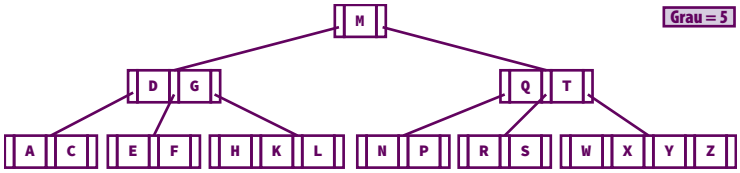

Figura E–46: Questão 49 — Capítulo 6

50. Consulte a **Seção 6.4**.

51. Consulte a **Seção 6.4**.

52. Consulte a **Seção 6.4**.

53. Consulte a **Seção 6.4**.

54. A função `EncontraCaminhoB()` mantém numa pilha o caminho de nós visitados desde a raiz até o nó no qual a chave se encontra ou deve ser inserida. A função `EncontraNoMultiMS()` guarda apenas a posição do nó no qual a chave está ou deveria estar.

55. Consulte a **Seção 6.4**.

56. Consulte a **Seção 6.4**.

57. Porque podem ocorrer junções de nós que se propaguem até a raiz.

58. Como a chave `H` está numa folha e essa folha tem um número maior do que o mínimo permitido de chaves, a remoção é imediata: move-se `K` para a posição onde `H` estava e `L` para a posição antes ocupada por `K`, como mostra a **Figura E–47**.

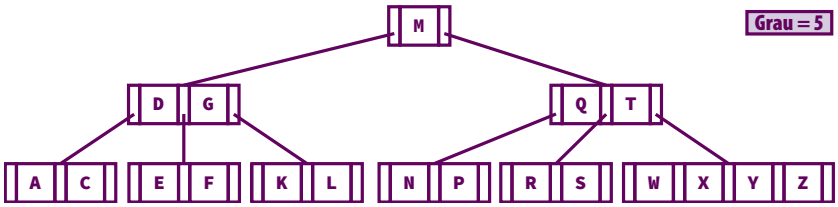

Figura E–47: Questão 58 — Capítulo 6

59. Como `T` não está numa folha, primeiro encontra-se sua chave sucessora, que é a chave `W` e move-se `W` para o lugar de `T`. Já que essa folha tem chaves extras, a remoção está concluída. A **Figura E–48** mostra o resultado.

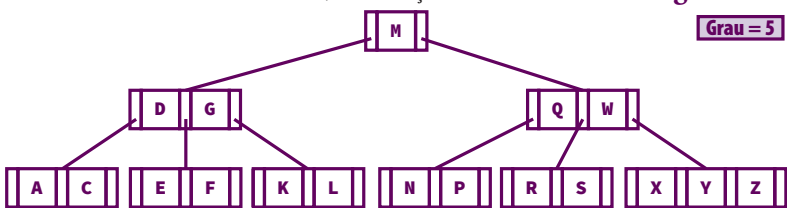

Figura E–48: Questão 59 — Capítulo 6

60. A remoção de `R` resultará num nó com apenas uma chave, o que não é aceitável. Como o irmão direito do nó contendo `R` e `S` tem chave sobressalente, o sucessor de `S`, que é `W`, é movido do nó-pai para o nó contendo `R` e `S` e a chave `X` é movida para o nó-pai. Antes de mover `W` do nó-pai para o filho, deve-se afastar `S` para ceder espaço para `W`, como mostra a **Figura E–49**.

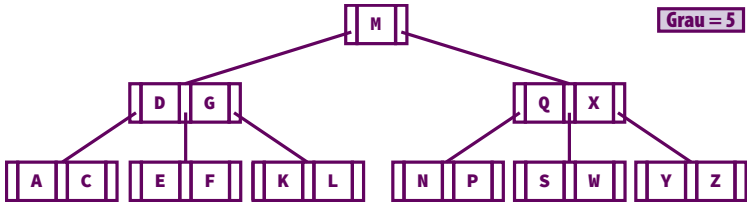

Figura E–49: Questão 60 — Capítulo 6

61. Embora `E` esteja numa folha, essa folha não tem chave sobressalente e o mesmo ocorre com os irmãos vizinhos do nó contendo `E`. Nesse caso, a folha contendo `E` deve ser combinada com um desses irmãos. Isto inclui mover para baixo a chave do nó-pai que está entre os filhos que serão combinados. Aqui, serão combinadas a folha contendo `F` e aquela contendo `A` e `C`, com a chave `D` sendo movida para o novo nó abaixo, como mostra a **Figura E–50 (a)**. Após essas alterações, o nó contendo `G` ficou com apenas uma chave, o que não é aceitável, e esse nó não possui nenhum irmão vizinho que lhe possa ceder uma chave. Assim deve-se novamente combinar os vizinhos e mover `M` do nó-pai para baixo. Nesse caso, a profundidade da árvore diminui um nível, conforme mostra a **Figura E–50 (b)**.

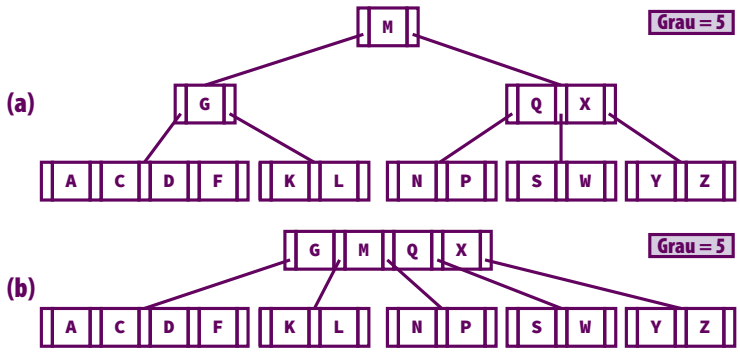

Figura E–50: Questão 61 — Capítulo 6

62. O primeiro passo na remoção de `C` é encontrar sua chave sucessora, que é a chave `D`. Essa chave sucessora deve substituir `C`, mas essa substituição deixa o nó contendo `E` com um número de chaves menor do que o mínimo permitido [v. **Figura E–51 (a)**]. Como nenhum dos irmãos vizinhos do nó contendo `E` tem chaves sobressalentes, deve-se combinar esse nó com um dos seus irmãos. Aqui combinam-se o nó contendo `A` e `B` com aquele contendo `E` [v. **Figura E–51 (b)**]. Agora o nó contendo `F` fica com um número insuficiente de chaves, mas o irmão desse nó tem uma chave extra que pode ser cedida. Assim pega-se a chave `M` do irmão desse nó, move-se para o nó-pai e traz-se `J` para juntar-se a `F`. O nó contendo `K` e `L` torna-se ligado à direita de `J`. O resultado final é mostrado na **Figura E–51 (c)**.

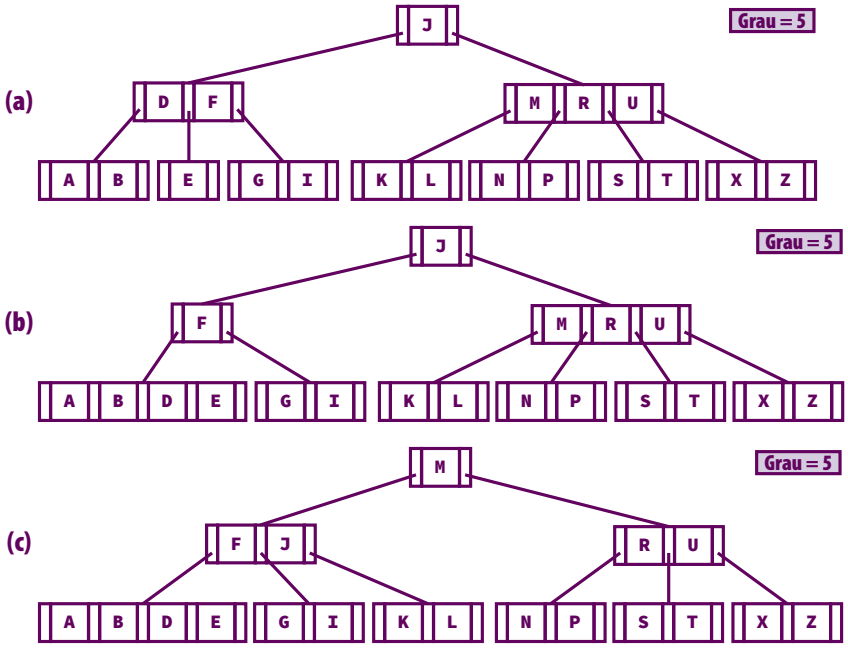

Figura E–51: Questão 62 — Capítulo 6

63. (a) Descendo a árvore até encontrar uma folha e contando os nós encontrados nesse caminho. (É mais fácil descer até o nó mais à esquerda ou mais à direita da árvore.) (b) Porque, numa árvore multidirecional descendente, as folhas podem estar em níveis diferentes e é necessário encontrar a folha mais profunda.
64. Não.
65. Consulte a **Seção 6.4.6**.
66. Consulte a **Seção 6.4.6**.
67. Isso não é estritamente necessário, mas facilita a implementação, pois a raiz de uma árvore B pode passar a ser outro nó, o que não ocorre com árvores descendentes.
68. É número mínimo de filhos que um nó pode ter (i.e., $\lceil G/2 \rceil$, em que G é grau da árvore).
69. V. **Teorema 6.2**.
70. Consulte a **Seção 6.5**.
71. Consulte a **Seção 6.5**.
72. Todos os filhos de uma folha de árvore B são vazios. Uma folha de árvore B+ simplesmente não possui filho algum.
73. Consulte a **Seção 6.5**.
74. Consulte a **Seção 6.5**.
75. Consulte a **Seção 6.5**.
76. Estude a **Seção 6.8.6**.
77. No dimensionamento de árvores B+ devem ser levados em consideração dois tipos de nós.
78. As constantes desse tipo são usadas para informar se um nó é interno, folha ou está vazio.
79. Porque, mesmo removidas, essas chaves ainda podem guiar as operações de busca, inserção e remoção.
80. Consulte o **Teorema 6.3**.
81. (a) O número mínimo de nós no último nível do índice é dado por $2 \cdot d^{a-2}$ (mostre isso). Multiplicando-se por d, obtém-se o número de folhas dessa árvore, que é $2 \cdot d^{a-1}$. Multiplicando-se pelo número mínimo de chaves numa folha, obtém-se o resultado desejado. (b) O número máximo de filhos no último nível da árvore é G^a (mostre isso). O número máximo de chaves é obtido quando cada folha possui G chaves. (c) V. prova do **Teorema 6.1**. (d) V. resposta da questão **43**.
82. No pior caso, cada nó é preenchido pela metade, de maneira que o espaço alocado é $\theta(2 \cdot n)$. Mas $\theta(2 \cdot n)$ é o mesmo que $\theta(n)$.
83. A prova é semelhante àquela do **Teorema 6.2** para árvores B.
84. Idem.
85. Idem.
86. A maneira mais fácil é descendo até a folha mais à esquerda (ou à direita) e contando o número de nós encontrados nesse caminho.
87. A resposta é idêntica àquela da questão similar sobre árvores B.
88. Quando uma folha é dividida, a folha da esquerda não recebe mais nenhuma chave, já que as demais chaves que serão inseridas são maiores do que as chaves dessa folha. Assim a única folha que poderá ter mais da metade de sua capacidade preenchida é a última folha. Um raciocínio similar pode ser usado para mostrar que a maioria dos nós internos é preenchida apenas pela metade.
89. Consulte a **Seção 6.6**.
90. Consulte a **Seção 6.6**.
91. Não.
92. (a) Uma vantagem é que, com exceção da raiz, todos os demais nós são preenchidos com 2/3 da capacidade de cada nó. Outra vantagem é que, quando uma inserção deve ocorrer num nó que se encontra completo,

em vez de ser dividido imediatamente, como ocorre com nós de árvores B, esse nó tem suas chaves divididas entre seus vizinhos. Desse modo, divisões de nós que oneram as operações de inserção podem ser adiadas. (b) A principal desvantagem é que a implementação da operação de remoção é bem mais complicada do que ocorre com árvores B.

93. Pesquise sobre árvores B# na internet.

94. Consulte a **Seção 6.7**.

95. Consulte a **Seção 6.7**.

96. Descendo sempre pela esquerda, encontra-se a menor chave. Depois segue-se em frente na lista encadeada de folhas.

97. (a) Sim. (b) Não.

98. (a) Sim. (b) Não, mas os mesmos raciocínios que norteiam essas funções podem ser usados com árvores B+.

99. (a) Como todas as folhas estão no último nível, desce-se até a folha mais à esquerda (ou à direita) da árvore contando-se os nós encontrados no caminho. (b) Sim.

100. Numa árvore B, todas as folhas se encontram no mesmo nível, o que não ocorre com árvores multidirecionais descendentes de busca.

101. $\theta(n)$.

102. $\theta(\log_d n)$, em que d é o grau mínimo da árvore e n é seu número de chaves.

103. $\theta(n)$.

104. $\theta(n)$.

105. $\theta(\log_d n)$, em que d é o grau mínimo da árvore e n é seu número de chaves.

Capítulo 7 — Dispersão em Memória Principal

1. Consulte a **Seção 7.1.1**.

2. São dois: (1) encontrar funções de dispersão e (2) resolver colisões.

3. Consulte a **Seção 7.1.4**.

4. Consulte a **Seção 7.1.1**.

5. Consulte a **Seção 7.1.2**.

6. Porque as chaves numa tabela de dispersão não obedecem a nenhum critério de ordenação.

7. Pela mesma razão apresentada na questão anterior.

8. Nesse caso, consulta de intervalo só pode ser implementada com busca sequencial.

9. Consulte a **Seção 7.2.1**.

10. (a) Consulte a **Seção 7.2.3**. (b) Para manter os valores de dispersão dentro do intervalo de índices da tabela.

11. Consulte a **Seção 7.2.1**.

12. (a) Consulte a **Seção 7.2.3**. (b) Tipicamente, esse método é usado com strings.

13. Consulte a **Seção 7.2.3**.

14. É um método de dispersão que aceita chaves de qualquer tipo.

15. Consulte a **Seção 7.2.3**.

16. Consulte a **Seção 7.2.3**.

17. Consulte a **Seção 7.2.3**.

18. Consulte a **Seção B.9.2** no **Apêndice B**.

19. (a) O resultado (q) da divisão inteira é obtido como se o quociente da divisão real dos respectivos operandos tivesse sua parte fracionária descartada. Tendo calculado o resultado da divisão inteira, o resto (r)

dessa divisão é obtido por meio da fórmula: $r = x - q \times y$. (b) O resultado não é portável, pois existem dois resultados possíveis.

20. (a) Existem testes estatísticos mais sofisticados, mas o teste apresentado na **Seção 7.8.1** é simples e pode ser suficiente. (b) Consulte a **Seção 7.2.5**.

21. Consulte a **Seção 7.2.5**.

22. O método de Horner permite calcular valores de dispersão usando o método polinomial de modo bem eficiente sem precisar calcular explicitamente as potências dos termos do polinômio. Consulte a **Seção 9.6** para entender os detalhes desse método.

23. (a) Essa função de dispersão mistura todos os bytes de uma estrutura, inclusive aqueles que podem fazer parte de um eventual preenchimento efetuado pelo compilador. E, para azar dessa função, as estruturas desse programa são preenchidas e preenchimento não é iniciado com nenhum valor. (b) Esse problema pode ser resolvido, de modo paliativo, evitando que a estrutura seja preenchida, mas isso nem sempre é possível ou portável. Assim a solução ideal consiste em reescrever a função de dispersão de modo que ela leve em consideração cada campo da estrutura individualmente. Essa solução deve ser considerada um princípio fundamental na escrita de funções que calculam valores de dispersão de estruturas.

24. Todas as chaves colidirão no mesmo valor: *10*.

25. Consulte o **Apêndice B**.

26. Consulte o **Apêndice B**.

27. Consulte o **Apêndice B**.

28. Essa expressão atribui a `z` o menor valor dentre `x` e `y`.

29. (a) `x` recebe $2^{n+1} - 1$. (b) `y` recebe sempre *1*. (c) `z` recebe o mesmo que `x`.

30. Consulte o **Apêndice B**.

31. Consulte o **Apêndice B**.

32. `FFFFFFFFFFFFDB29`.

33. Consulte a **Seção B.4** no **Apêndice B**.

34. Consulte a **Seção B.4** no **Apêndice B**.

35. Consulte a **Seção B.4** no **Apêndice B**.

36. Consulte a **Seção B.4** no **Apêndice B**.

37. Quando a máscara é do tipo **int** ou **short** ou **long** e seu bit mais significativo é `1`, ela é dependente das larguras desses tipos, que dependem de implementação. Para torná-la independente de implementação usa-se o complemento da máscara (p. ex., `~m`) e a operação de mascaramento é escrita novamente usando esse operador (p. ex., `x & ~~m`).

38. Consulte a **Seção B.7** no **Apêndice B**.

39. Consulte a **Seção B.3**.

40. Consulte a **Seção B.3**.

41. (a) Essa função conta o número de bits de seu parâmetro que estão ligados. Portanto um bom nome seria *BitsLigados*. (b) `nBitsLigados`.

42. Consulte um bom texto sobre Lógica Matemática.

43. `Misterio1()` retorna a soma de seus parâmetros. `Misterio2()` retorna o produto de seus parâmetros.

44. Devido à ocorrência de overflow no segundo laço.

45. Consulte a **Seção 7.3.1**.

46. Consulte a **Seção 7.3.1**.

47. Não, mas influenciará seus posicionamentos nos coletores, a não ser que eles sejam ordenados, o que não faz muito sentido.

48. Consulte a **Seção 7.3.1**.

49. Consulte a **Seção 7.3.1**.

50. Para garantir que o valor de dispersão é um índice válido da tabela.

51. $\theta(M)$.

52. Uma busca bem-sucedida é normalmente mais rápida. Se uma chave não está presente numa tabela, todos os nós da respectiva lista (coletor) precisam ser acessados.

53. (a) Sim. (b) Sim.

54. V. **Figura E–52**.

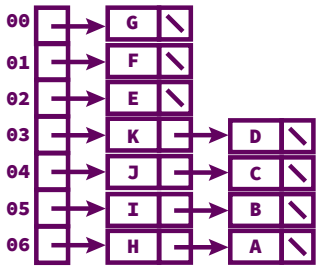

Figura E–52: Questão 54 — Capítulo 7

55. V. **Figura E–53**.

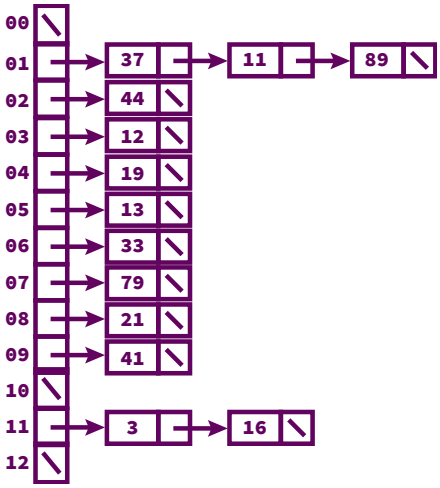

Figura E–53: Questão 55 — Capítulo 7

56. V. **Figura E–54**.

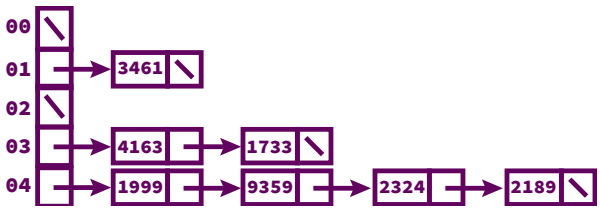

Figura E–54: Questão 56 — Capítulo 7

57. (a) O pior caso de uma tabela de dispersão com encadeamento acontece quando todas as chaves dispersam num único coletor. Nesse caso, o custo de uma operação de busca, inserção ou remoção é $\theta(n)$, em que n é o número de chaves na tabela. Uma função de dispersão (extremamente ruim, aliás) que resultaria em tal desempenho seria $f(c) = k$, em que k é alguma constante dentro do intervalo de índices da tabela. (b) Usando uma função de dispersão que distribua as chaves uniformemente.

58. Consulte a **Seção 7.4.1**.

59. Consulte a **Seção 7.4.1**.

60. Consulte a **Seção 7.4.1**.

61. Quando o valor de dispersão de uma chave resulta no índice de um coletor (lista encadeada), não há nenhum outro coletor no qual essa chave possa estar.

62. Sim, a não ser que não ocorra nenhuma colisão. Consulte a **Seção 7.4.2**.

63. Consulte a **Seção 7.4.2**.

64. (a) O custo espacial de dispersão com endereçamento aberto é $\theta(1)$, enquanto o custo espacial de dispersão com encadeamento é $\theta(n)$. Além disso, dispersão com endereçamento aberto pode apresentar boa localidade de referência, o que não ocorre com dispersão com encadeamento. (b) A dependência do desempenho com relação ao fator de carga é menor no caso de dispersão com encadeamento do que no caso de dispersão com endereçamento aberto. Além disso, uma tabela de dispersão com encadeamento nunca deixa de funcionar devido à necessidade de redimensionamento, como ocorre com dispersão com endereçamento aberto.

65. Consulte a **Seção 7.4.2**.

66. Consulte a **Seção 7.4.2**.

67. (a) O número de passos já realizados. (b) A chave de busca.

68. Sondagem quadrática causa agrupamento secundário.

69. Se o tamanho não for primo, o tamanho do passo pode dividir a tabela em partes iguais, o que pode fazer com uma sondagem não alcance certas posições da tabela.

70. Consulte a **Seção 7.4.2**.

71. Consulte a **Seção 7.4.2**.

72. (a) Consulte a **Seção 7.4.5**. (b) Sondagem quadrática é mais fácil de programar do que sondagem com dispersão dupla e um pouco mais rápida quando o fator de carga é pequeno.

73. Consulte a **Seção 7.4.2**.

74. Consulte a **Seção 7.4.2**.

75. V. resposta da questão **64**.

76. O resultado aparece na **Figura E–55**.

00	01	02	03	04	05	06	07	08	09
140	660	777		244	444	166	386		

10	11	12	13	14	15	16	17	18	19
	151	52	92	34	135	76	371	118	555

Figura E–55: Questão 76 — Capítulo 7

77. O resultado aparece na **Figura E–56**.

00	01	02	03	04	05	06	07	08	09
140			660	244	371	166	92		555

10	11	12	13	14	15	16	17	18	19
	151	52	386	34	135	76	777	118	444

Figura E–56: Questão 77 — Capítulo 7

78. V. **Figura E–57**.

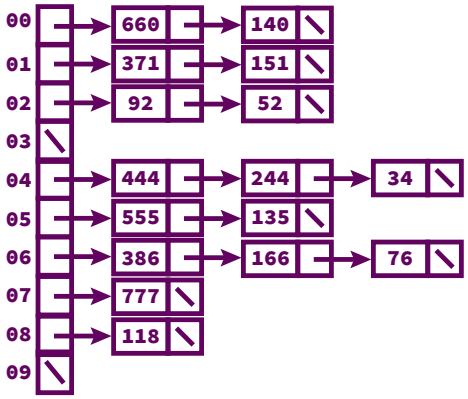

Figura E–57: Questão 78 — Capítulo 7

79. V. **Figura E–58**.

Chave	Exercício 76	Exercício 77	Exercícios 78
34	*1*	*1*	*3*
444	*2*	*6*	*1*
555	*5*	*4*	*1*
777	*6*	*1*	*1*
244	*1*	*1*	*2*
76	*1*	*1*	*3*
16	*8*	*7*	*4*

Figura E–58: Questão 79 — Capítulo 7

80. V. **Figura E–59**.

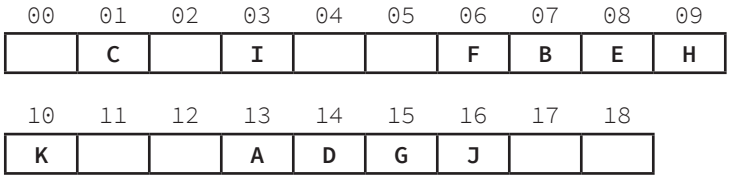

Figura E–59: Questão 80 — Capítulo 7

81.

(a) V. **Figura E–60**.

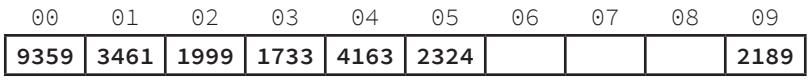

Figura E–60: Questão 81 (a) — Capítulo 7

(b) V. **Figura E–61**.

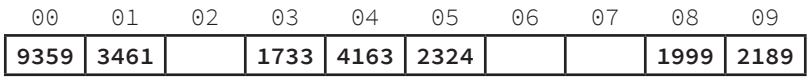

Figura E–61: Questão 81 (b) — Capítulo 7

(c) V. **Figura E–62**.

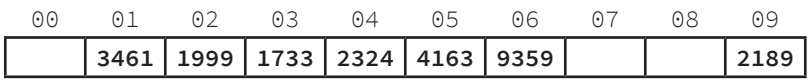

Figura E–62: Questão 81 (c) — Capítulo 7

82. (a) **F**. (b) **F**. (c) **F**. (d) **V**. (e) **F**.

83. Não, pois o número de chaves da tabela não pode ser maior do que o número de elementos capazes de armazená-las.

84. V. **Figura E–63**.

00	01	02	03	04	05	06	07	08	09	10	11	12
3	89	11	12	37	13	33	79	21	41	19	16	44

Figura E–63: Questão 84 — Capítulo 7

85. V. **Figura E–64**. Note que a chave *3* não pode ser inserida, mesmo havendo espaço disponível.

00	01	02	03	04	05	06	07	08	09	10	11	12
	89	11	12	19	13	33	79	21	41	37	16	44

Figura E–64: Questão 85 — Capítulo 7

86. V. **Figura E–65**.

00	01	02	03	04	05	06	07	08	09	10	11	12
13	79	41	37	11	16	19	33	21	3	44	89	12

Figura E–65: Questão 86 — Capítulo 7

87. Se **realloc()** fosse usada, as chaves armazenadas na antiga tabela seriam copiadas com o mesmo valor de dispersão para a tabela nova.

88. Consulte a **Seção 7.4.4**.

89. (a) Uma constante do tipo `tStatusDEA` indica o status de um elemento de uma tabela de dispersão com endereçamento aberto. (b) Faça sua parte e explique por que isso não faz sentido.

90. (a) O desempenho de pior caso de busca numa tabela de dispersão com endereçamento aberto ocorre quando a tabela está repleta e a chave de busca não é encontrada. Nesse caso, o custo temporal da operação é $\theta(m)$, em que m é o número de posições da tabela. Essa situação pode ocorrer com qualquer função de dispersão. (b) Mantendo o fator de carga baixo (v. **Seção 7.4.5**).

91. (a) Como o valor inicial da sequência determina toda uma sequência de sondagem, o número de sequências possíveis é m. (b) Idem. (c) Nesse caso, uma sequência de sondagem depende de dois valores de dispersão [i.e., de $f_1(c)$ e $f_2(c)$, sendo c uma chave], de modo que o número de sequências de sondagem pode ser, no máximo, m^2.

92. (a) Sondagem linear. (b) Sondagem com dispersão dupla.

93. Consulte a **Seção 7.5**.

94. Para obter o custo amortizado sugerido pela questão, é preciso que se suponha que o redimensionamento da tabela é efetuado geometricamente. Ou seja, se a tabela precisar aumentar de tamanho, ele deverá ser dobrado e quando o tamanho tiver que ser reduzido, ele será reduzido à metade (v. **Capítulo 5**). Além disso, para simplificar, o fator de carga (α) máximo será considerado igual a *1*.

A função potencial é definida de modo que ela armazene energia suficiente para redimensionar a tabela como:

$\Phi(h) = 2|n - m/2|$

Como α é $\theta(1)$, pode-se assumir que uma operação de busca tem custo temporal $\theta(1)$.

Inserir um elemento incrementa o valor de n e três casos que devem ser levados em consideração:

- $1/2 \leq \alpha < 1$. Nesse caso, o potencial aumenta *2*, de modo que o custo temporal amortizado é $1 + 2 = 3$.
- $\alpha < 1/2$. Nesse caso, o potencial diminui 2, de sorte que o custo temporal amortizado é $1 - 2 = -1$.
- $\alpha = 1$. Aqui, a tabela de dispersão é redimensionada, de modo que o custo temporal real é $1 + m$. Mas o potencial varia de m a *0*, de maneira que o custo temporal amortizado é $1 + m - m = 1$.

Numa operação de busca, o potencial não é alterado, de modo que essa operação tem custo temporal amortizado igual ao seu custo real, que é *1*.

Uma operação de remoção reduz *1* de *n* e, novamente, há três casos a ser considerados:

- $1/2 \leq \alpha < 1$. O potencial diminui *2*, de maneira que o custo temporal amortizado é $1 - 2 = -1$.
- $\alpha < 1/2$. O potencial aumenta *2* e, assim, o custo temporal amortizado é $1 + 2 = 3$.
- $\alpha = 1$. Nesse caso, a tabela de dispersão precisa ser redimensionada, de maneira que o custo temporal real é $1 + m/4$. O potencial varia de $m/2$ até *0*, de maneira que o custo temporal amortizado é $1 + m/4 - m/2 = 1 - m/4$.

Em cada caso apresentado acima, o custo temporal amortizado é $\theta(1)$. Se uma tabela de dispersão for iniciada com $\alpha = 1/2$, seu potencial inicial será *0*. Assim, sabe-se que ele nunca diminuirá, de modo que o custo temporal amortizado será um limite superior do custo temporal real. Conclui-se, portanto, que uma sequência de *n* operações sobre a tabela tem custo $\theta(n)$.

95. Consulte a **Seção 7.6**.

96. Consulte a **Seção 7.6.1**.

97. Consulte a **Seção 7.6.1**.

98. Consulte a **Seção 7.6.1**.

99. Consulte a **Seção 7.6.1**.

100. Consulte a **Seção 7.6.4**.

101. Consulte a **Seção 7.6.4** e a **Seção 7.7**.

102. Consulte a **Seção 7.7**.

103. Porque cada chave é mapeada na posição da tabela em que ela se encontra ou deveria se encontrar (ou, pelo menos, a poucos passos dessa posição quando ocorre uma colisão). Fazer uma busca sequencial numa tabela de dispersão não é diferente de fazer uma busca sequencial numa tabela indexada, como aquelas vistas no **Capítulo 3**.

104. Consulte a **Seção 7.3**.

105. Consulte a **Seção 7.7**.

106. Consulte a **Seção 7.7**.

107. Dispersão com encadeamento é a melhor escolha, pois as listas serão relativamente pequenas e terão aproximadamente o mesmo tamanho.

108. Dispersão com encadeamento é a melhor escolha, pois a tabela não precisará ser redimensionada, a não ser que o desempenho seja severamente afetado.

109. Uma busca bem-sucedida é mais rápida porque, em média, apenas metade dos elementos de uma lista precisa ser examinada.

110. Consulte a **Seção 7.7**.

111. Consulte a **Seção 7.8.1**.

112. Consulte a **Seção 7.8.2**.

113. Consulte a **Seção 7.8.2**.

114. Consulte a **Seção 7.8.2**.

115. Consulte a **Seção 7.8.2** (ou um médico...).

116. Erro falso-negativo.

117. Erro falso-negativo.

118. Erro falso-positivo.

119. Consulte o **Apêndice B**.

Capítulo 8 — Dispersão em Memória Secundária

1. Consulte a **Seção 8.1.1**.
2. Consulte a **Seção 8.1.1**.
3. Consulte a **Seção 8.1**.
4. Consulte a **Seção 8.1.4**.
5. (a) Consulte a **Seção 8.1.2**. (b) Consulte a **Seção 8.1.4**.
6. (a) Em ambos os casos são usados coletores que podem ser acessados diretamente por meio de valores de dispersão; nos dois casos, as operações de busca num coletor são sequenciais. (b) Dispersão estática é usada em memória secundária, enquanto dispersão com encadeamento é usada em memória principal; dispersão estática usa coletores excedentes, o que não faz sentido em dispersão com encadeamento; remoção de um registro em dispersão estática pode requerer movimentação de vários outros registros que fazem parte do mesmo coletor, enquanto remoção de um registros em dispersão com encadeamento requer apenas alteração de ponteiros; quando um coletor fica vazio em dispersão com encadeamento, ele deixa de ocupar espaço em memória, o que é complicado obter em dispersão em dispersão estática (pois requer reconstruir o arquivo).
7. (a) Consulte a **Seção 8.1.2**. (b) Idem.
8. Redimensionamento de uma tabela armazenada em memória secundária pode não ser tão suportável quanto o redimensionamento de uma tabela armazenada em memória principal, pois ele requer muitos acessos ao meio de armazenamento externo. Se você não entendeu, volte ao **Capítulo 1**.
9. Consulte a **Seção 8.2**.
10. Bancos de dados e sistemas de arquivos.
11. Consulte a **Seção 8.2**.
12. Consulte a **Seção 8.2.1**.
13. 2^{g-l}.
14. Consulte a **Seção 8.2.1**.
15. V. **Figura E–66**.

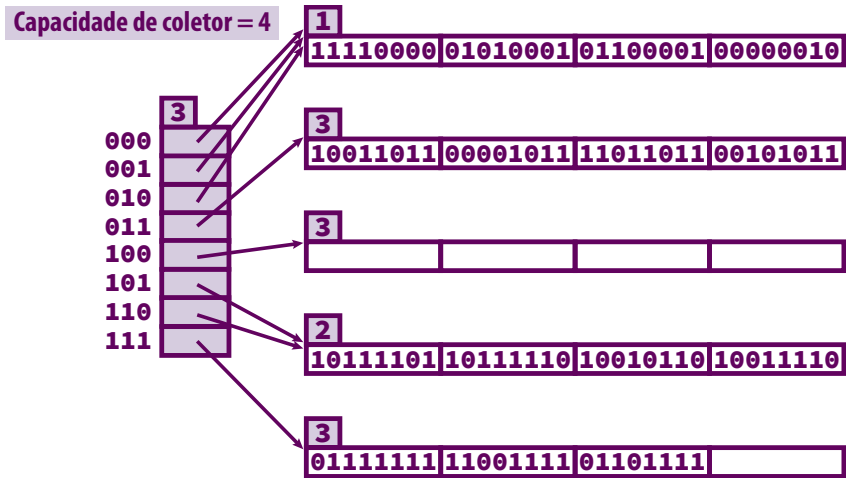

Figura E–66: Questão 15 — Capítulo 8

16. V. **Figura E–67**.

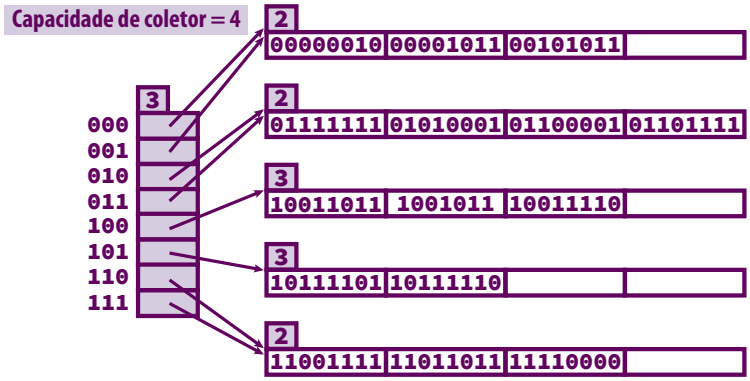

Figura E–67: Questão 16 — Capítulo 8

17. (a) A **Figura E–68 (a)** mostra por que muitos texto sobre o assunto usam bits mais significativos nas ilustrações de dispersão extensível: usando esses bits, em vez de bits menos significativos, podem-se desenhar ilustrações sem cruzamento das linhas que representam as referências. (b) V. **Figura E–68 (b)**. (c) A remoção do registro cuja chave tem valor de dispersão igual a *10*, embora seja o último de seu coletor, não provoca compressão do diretório [v. **Figura E–68 (c)**].

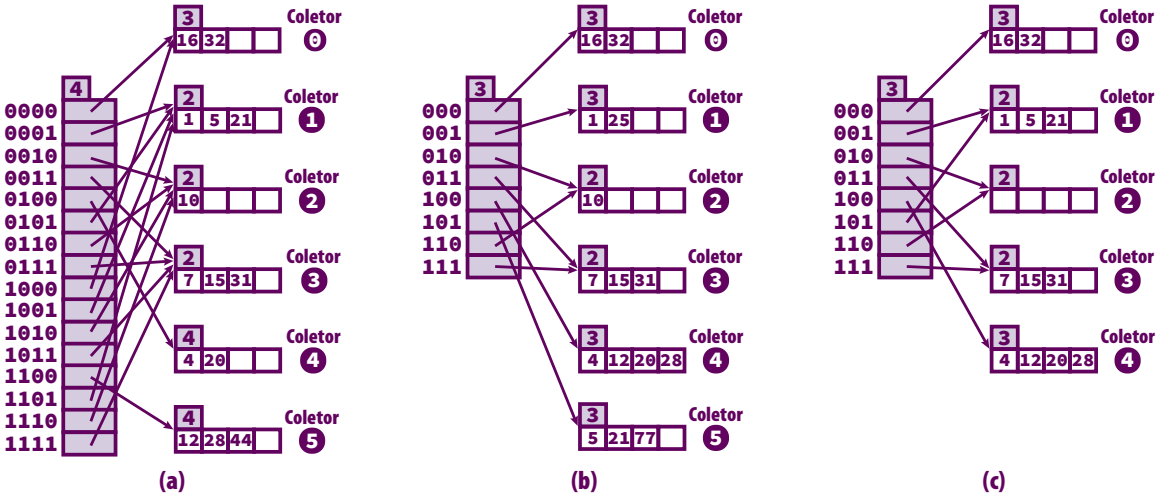

Figura E–68: Questão 17 — Capítulo 8

18. Consulte o **Apêndice B**.

19. Consulte o **Apêndice B**.

20. Consulte a **Seção 8.2.5**.

21. Consulte a **Seção 8.2.5**.

22. A diferença é adverbial: dispersão estática *frequentemente* usa coletores excedentes, enquanto dispersão extensível *raramente* o faz.

23. Consulte a **Seção 8.2.5**.

24. Consulte a **Seção 8.2.5**.

25. Consulte a **Seção 8.2.6**.

26. Consulte a **Seção 8.2.2**.

27. Consulte a **Seção 8.2.4**.

28. Consulte a **Seção 8.2.4**.

29. Consulte a **Seção 8.2.4**.

30. (a) Exatamente dois coletores terão apenas uma referência para si após a duplicação de um diretório. Isso ocorre porque, quando o diretório é duplicado, um dos coletores deve ter sido dividido fazendo com que um elemento do diretório faça referência para cada um deles. (b) Sim, pois esses coletores são camaradas.

31. Os pré-requisitos são dois: (1) o diretório deve caber em memória principal e (2) não deve haver coletores excedentes (3) a chave deve ser armazenada junto com o registro (i.e., a tabela de busca usa chaves internas).

32. Apenas aqueles associados aos coletores que foram duplicados e que causaram a duplicação do diretório.

33. Não. Nenhum desses esquemas de tabelas de dispersão é conveniente para memória externa, pois eles podem requerer uma grande quantidade de acessos ao meio de armazenamento, o que é inaceitável para memória externa.

34. Consulte a **Seção 8.3**.

35. Consulte a **Seção 8.3**.

36. $\theta(n/M)$, em que n é o número de registros e M é o número máximo de registros num coletor.

37. Consulte a **Seção 8.4.2**.

Capítulo 9 — Strings e Texto

1. Consulte a **Seção 9.1**.

2. Consulte a **Seção 9.1**.

3. Consulte a **Seção 9.1**.

4. Consulte a **Seção 9.1**.

5. (a) Não há borda. (b) `"A"` e `"AA"`.

6. (a) 0. (b) 1. (c) 2. (d) 1.

7. Consulte a **Seção 9.1**.

8. Consulte a **Seção 9.1**.

9. Quando o texto é lido num stream e não é armazenado.

10. Leia a **Nota de rodapé [1]** na **página 457**.

11. Consulte a **Seção 9.2**.

12. Consulte a **Seção 9.2**.

13. Consulte a **Seção 9.2**.

14. Imite a **Figura 9–5.**

15. Consulte a **Seção 9.3**.

16. V. **Figura E–69**.

j	0	1	2	3	4	5	6
p[j]	A	AG	AGC	AGCT	AGCTA	AGCTAG	AGCTAGT
tmb[j]	0	0	0	0	1	2	0

Figura E–69: Questão 16 — Capítulo 9

17. V. **Figura E–70**.

j	0	1	2	3	4	5
p[j]	a	ab	aba	abac	abaca	abacab
tmb[j]	0	0	1	0	0	2

Figura E–70: Questão 17 — Capítulo 9

18. Ela é usada para determinar o tamanho de cada salto quando ele é necessário.

19. (a) `"C"`, `"CG"`, `"CGT"`, `"CGTA"`, `"CGTAC"`, `"CGTACG"` e `"CGTACGT"`. (b) `"T"`, `"TT"`, `"GTT"`, `"CGTT"`, `"ACGTT"`, `"TACGTT"`, `"GTACGTT"`. (c) Compare os itens (a) e (b) e conclua.
20. É sua vez.
21. Idem.
22. Idem.
23. Idem.
24. (a) Sua vez. (b) Imite a **Figura 9–8.**
25. Consulte a **Seção 9.3**.
26. Consulte a **Seção 9.3**.
27. Consulte a **Seção 9.4**.
28. Consulte a **Seção 9.4**.
29. Consulte a **Seção 9.4**.
30. Consulte a **Seção 9.4**.
31. Porque cada uma delas é baseada em expectativa e não há garantia de funcionamento do algoritmo **BM** quando uma delas é usada isoladamente.
32. Consulte a **Seção 9.4**.
33. Porque o salto obtido com a regra do mau caractere pode ser negativo.
34. Imite a **Figura 9–24.**
35. Consulte a **Seção 9.5**.
36. Consulte a **Seção 9.5**.
37. Consulte a **Seção 9.5**.
38. Imite a **Figura 9–29.**
39. Sua vez.
40. É a sua vez.
41. O uso de dispersão (*hashing*).
42. Consulte o **Capítulo 7**.
43. Porque qualquer permutação dos caracteres de um string resulta no mesmo valor de dispersão.
44. Consulte a **Seção 9.6**.
45. Consulte o **Apêndice B** do **Volume 1** ou qualquer livro-texto decente de Matemática do ensino médio.
46. Consulte a **Seção 9.6**.
47. Consulte a **Seção 9.6**.
48. Consulte a **Seção 9.6**.
49. Para evitar a ocorrência de overflow.
50. Consulte a **Seção 9.6**.
51. Consulte a **Seção 9.6**.
52. (a) Consulte a **Seção 9.6**. (b) Algoritmo de força bruta.
53. Os valores de dispersão do padrão e da primeira janela de texto e o valor da maior potência do polinômio são calculados.
54. 9.
55. Apenas um casamento falso.
56. Consulte a referida seção.
57. Consulte a **Seção 9.6**.
58. Consulte a **Seção 9.6**.
59. Consulte a **Seção 9.6**.

60. Sim, pois, apesar de b e q serem aprovados no teste $b.(b-1).(q-1) \leq max$, os valores dos caracteres do alfabeto não são mapeados no intervalo $[0..b-1]$.

61. É a sua vez.

62. Idem.

63. (a) $\theta(1)$. (b) $\theta(m)$.(c) $\theta(|\sum|)$. (d) $\theta(|\sum|)$.(e) $\theta(1)$.

64. Consulte a **Seção 9.7**.

65. Consulte a **Seção 9.8**.

66. Sim, basta associar o valor de uma chave ao seu nó final na trie.

67. V. **Figura E–71**.

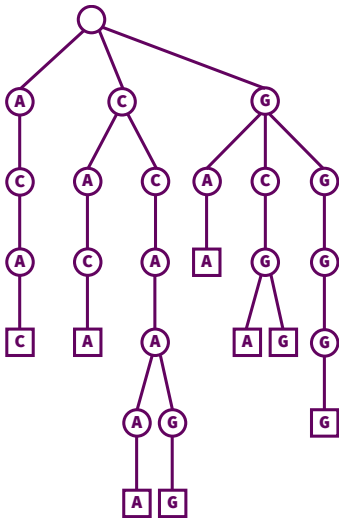

Figura E–71: Questão 67 — Capítulo 9

68. V. **Figura E–72**.

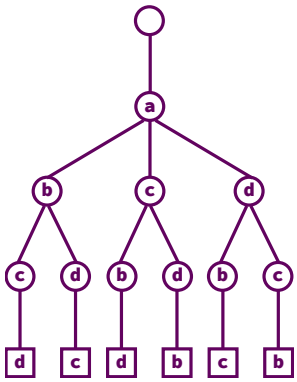

Figura E–72: Questão 68 — Capítulo 9

69. Consulte a **Seção 9.8**.

70. Porque uma trie pode ser excessivamente profunda e, consequentemente, muito ineficiente para operações de busca, inserção e remoção em memória secundária.

71. Consulte a **Seção 9.8**.

72. Consulte a **Seção 9.8**.

73. Consulte a **Seção 9.8.2**.

74. Consulte a **Seção 9.8.3**.

75. Consulte a **Seção 9.8.3**.
76. Consulte a **Seção 9.8.3**.
77. Consulte a **Seção 9.8.4**.
78. Consulte a **Seção 9.9**.
79. Consulte a **Seção 9.9**.
80. Consulte a **Seção 9.10.2**.
81. Consulte a **Seção 9.10.1**.
82. (a) Porque a função **strtok()** pode alterar seu primeiro parâmetro. (b) Violação de memória com o consequente aborto de programa.
83. Consulte a **Seção 9.10.1**.
84. Consulte a **Seção 9.10.2**.
85. Consulte a **Seção 9.10.3**.
86. `"010101"`.
87. Sistemas de controle de versão (p. ex., Git).
88. Consulte a **Seção 9.10.5**.
89. Porque seu custo temporal é muito elevado.
90. Siga a recomendação.
91. (a) Consulte a **Seção 9.10.6.** (b) Os algoritmos KMP, BM e BMH são baseados em saltos; i.e., a eficiência deles é assegurada por meio de saltos bem maiores do os saltos unitários dos algoritmos FB e KR. Mas adaptar os algoritmos KMP, BM e BMH para casamentos múltiplos seria bem mais complicado do que ocorre com o algoritmo KR (que, aliás, é bem fácil de adaptar como foi visto na **Seção 9.10.6**). Além disso, o custo espacial desses algoritmos para k padrões com tamanhos m seria $\theta(k \cdot m)$, enquanto o custo espacial do algoritmo KR seria apenas $\theta(k)$. O concorrente restante é o algoritmo FB, que não tem chance de competir com o algoritmo KR.
92. Consulte a **Seção 9.10.7**.
93. Não, porque ele apresenta retrocesso.
94. Idem.
95. Porque ele faz comparações de trás para frente (i.e., do último para o primeiro caractere do padrão).
96. (a) Essa função requer que o usuário saiba simular final de arquivo via teclado (i.e., `[CTRL] + [Z]` no Windows ou `[CTRL] + [D]` em sistemas da família Unix) quando o texto digitado não casa. (b) Essa alteração não resolveria o problema porque isso impediria o funcionamento do programa com textos que contivessem quebra de linha.

Capítulo 10 — Filas de Prioridade e Heaps

1. Consulte a **Seção 10.1.1**.
2. Consulte a **Seção 10.1.3**.
3. Lista indexada sem ordenação, pois o custo temporal da inserção é $\theta(1)$. Lista encadeada sem ordenação também apresenta esse custo, mas seu custo espacial é $\theta(n)$ (devido ao armazenamento de ponteiros).
4. Consulte a **Seção 10.2.1**.
5. Consulte a **Seção 10.2.1**.
6. Consulte a **Seção 10.2.1**.
7. Consulte a **Seção 10.2.1**.
8. Consulte a **Seção 10.2.1**.

9. Os cálculos utilizados na determinação do pai e o filho esquerdo de um nó requerem uma operação aritmética a menos.

10. (a) $Pai(i) = i/2$, se $i \neq 1$. Se $i = 1$, o nó i é a raiz da árvore. (b) $FilhoEsquerdo(i) = 2i$, se $2i \leq n$. Se $2i > n$, o nó i não possui filho esquerdo. (c) $FilhoDireito(i) = 2i + 1$, se $2i + 1 \leq n$. Se $2i + 1 > n$, o nó i não possui filho direito.

11. Consulte o **Capítulo 12** do **Volume 1**.

12. V. **Figura E–73**.

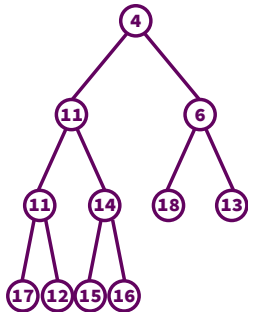

Figura E–73: Questão 12 — Capítulo 10

13. V. **Figura E–74**.

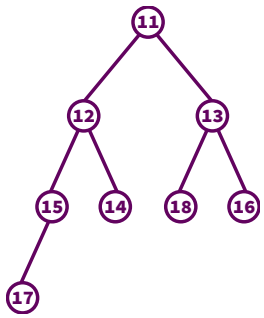

Figura E–74: Questão 13 — Capítulo 10

14. V. **Figura E–75**.

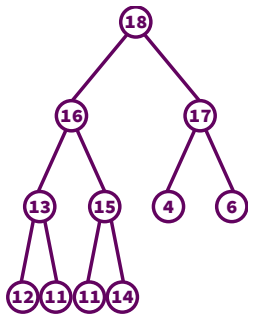

Figura E–75: Questão 14 — Capítulo 10

15. (a) Não. (b) Sim.

16. V. **Figura E–76**.

17. (a) Não se sabe ao certo, mas, seguramente, ele está no segundo nível da árvore (i.e., ele é um dos filhos da raiz). (b) No segundo ou no terceiro nível.

18. V. **Figura E–77**.

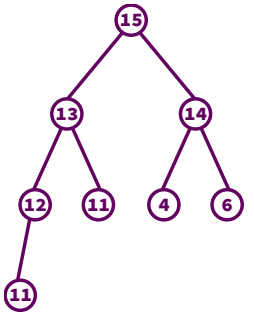

Figura E–76: Questão 16 — Capítulo 10

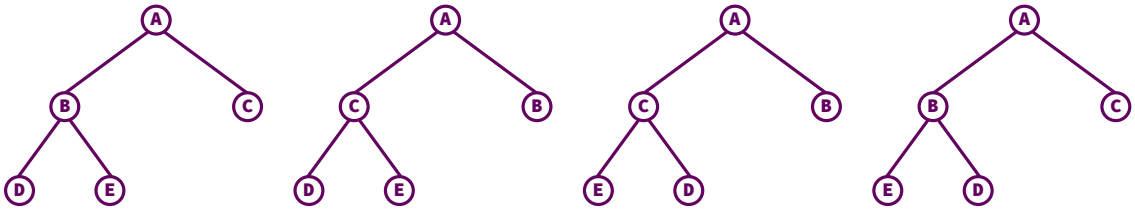

Figura E–77: Questão 18 — Capítulo 10

19. Consulte a **Seção 10.2.2**.

20. Consulte a **Seção 10.2.2**.

21. (a) Percolação ascendente. (b) Idem.

22. Trocar *maior* por *menor* e vice-versa.

23. V. **Figura E–78**.

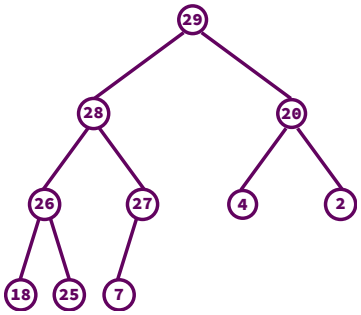

Figura E–78: Questão 23 — Capítulo 10

24. Qualquer heap é uma árvore binária completa e, como é provado no **Apêndice B** do **Volume 1**, sua profundidade p é dada por $p = \lfloor log_2\, n + 1 \rfloor$.

25. Seja p a profundidade do heap. Até a profundidade $p - 1$, o heap é uma árvore binária repleta cujo número de nós é dado por $2^{p-1} - 1$. O índice i da última folha é $n - 1$ e, assim, o índice de seu pai é dado por: $\lfloor (i - 1)/2 \rfloor = \lfloor (n - 2)/2 \rfloor$. Portanto, até o penúltimo nível, o número de nós é: $\lfloor (n - 2)/2 \rfloor = \lfloor n/2 \rfloor$. Logo o número de folhas é dado por $n - \lfloor n/2 \rfloor = \lceil n/2 \rceil$. ■

26. Numa folha, embora, em princípio, não se possa afirmar qual é essa folha.

27. Índices `7` e `8`: `A`, `B`, `C` ou `D`. Índice `9`: apenas `A`.

28. Não.

29. V. **Figura E–79**.

0	1	2	3	4	5	6	7	8	9	10	11	12
A	C	B	G	D	E	J	I	H	K	F	P	L

Figura E–79: Questão 29 — Capítulo 10

30. (1) `L` é removido e ocupa o lugar de `A`; (2) `L` troca de posição com `B`; (3) `L` troca de posição com `E`.

31. (a) `M` é inserido como filho esquerdo de `J` e só.

32. V. **Figura E–80**.

0	1	2	3	4	5	6	7	8
J	H	I	D	G	F	A	B	C

FIGURA E–80: QUESTÃO 32 — CAPÍTULO 10

33. Sua vez.

34. Sua vez.

35. Consulte a **Seção 10.2.3**.

36. Consulte a **Seção 10.2.3**.

37. Consulte a **Seção 10.2.3**.

38. (a) Sim. (b) Ascendente.

39. Sim.

40. Use indução finita (v. **Apêndice B** do **Volume 1**).

41. (a) Como um heap é uma árvore binária completa, o número mínimo de nós é $2^a - 1 + 1 = 2^a$. (b) O número máximo de elementos é $2^{a+1} - 1$. (V. **Capítulo 12** do **Volume 1**.)

42. Suponha que um heap contendo n elementos tenha altura a. Como foi visto no **Capítulo 12** do **Volume 1**, tem-se que $2^a \leq n \leq 2^{a+1} - 1 < 2^{a+1}$, de modo que $a \leq log\ n < a + 1$. Agora, como a deve ser um valor inteiro tem-se que, usando a definição de piso, $a = \lfloor log\ n \rfloor$.

43. Suponha que a afirmação não é verdadeira e complete a prova por contradição.

44. V. tabela abaixo.

Implementação via...	Custo Temporal de...	
	Inserção	Remoção
Lista encadeada sem ordenação	$\theta(1)$	$\theta(n)$
Lista encadeada ordenada	$\theta(n)$	$\theta(1)$
Lista indexada sem ordenação	$\theta(1)$	$\theta(n)$
Lista indexada ordenada	$\theta(n)$	$\theta(1)$
Árvore binária de busca balanceada	$\theta(log\ n))$	$\theta(log\ n)$
Árvore binária de busca sem balanceamento	$\theta(n)$	$\theta(n)$

45. Não.

46. (1) O custo espacial é menor usando heap, pois, nesse caso, não são usados ponteiros ou informações sobre balanceamento. (2) Implementação com heap é muito mais fácil.

47. Consulte a **Seção 10.4**.

48. Consulte a **Figura 10–13**.

49. Consulte a **Seção 10.4**.

50. Consulte a **Seção 10.4**.

51. Distribuição exponencial é usada na geração de intervalos de tempo aleatórios entre eventos.

52. Consulte a **Seção 10.5.1**.

53. Consulte a **Seção 10.5.1**.

54. De acordo com o algoritmo de Huffman, o número de folhas da árvore de codificação é igual ao número n de caracteres codificados, de modo que não há o que provar. Agora, de acordo com o **Teorema 12.2** apresentado no **Capítulo 12** do **Volume 1**, o número de folhas de uma árvore binária é $n = n_2 + 1$, em que

n_2 é o número de nós de grau 2. Como essa árvore é estritamente binária, todos os nós internos possuem grau 2. Portanto, o número de nós internos é $n - 1$. ■

55. A codificação canônica não requer que a árvore de codificação seja escrita no cabeçalho da codificação; i.e., apenas os tamanhos dos códigos precisam ser escritos. Isso, além de tornar a implementação bem mais simples, também economiza espaço.

56. 4.

57. A **Figura E–81** mostra a árvore resultante da codificação. Utilizando-se essa figura e o algoritmo da **Figura 10–16**, obtêm-se os seguintes códigos: `'N': 0`, `'C': 10`, `'T': 110`, `'A': 111`.

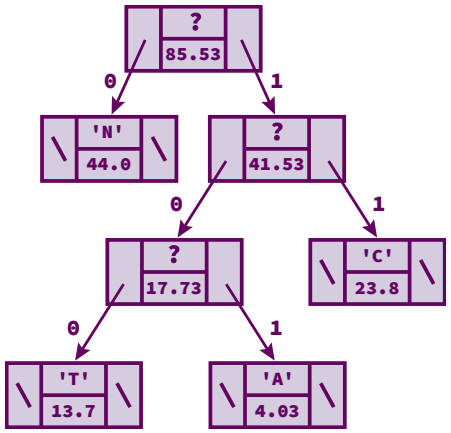

Figura E–81: Questão 57 — Capítulo 10

58. Consulte a **Seção 10.5.1**.

59. A árvore de codificação mais profunda que pode existir é aquela que possui dois nós em cada nível abaixo da raiz, sendo que um deles é folha e o outro possui dois filhos. Assim a profundidade dessa árvore é igual ao seu número de folhas. Como, por definição, o número de folha de uma árvore de codificação é igual ao número de bytes distintos, a profundidade máxima de uma árvore de codificação é igual a esse número.

60. Consulte a **Seção 10.5.1**.

61. Consulte a **Seção 10.5.1**.

62. (a) Um cabeçalho contém informações que permitem que arquivo codificado possa ser decodificado. (b) Apenas os tamanhos dos códigos e o número de bytes do arquivo. (c) Está implícito que o mesmo algoritmo de atribuição de códigos será usado quando o arquivo for decodificado.

63. Quando todos os valores possíveis de bytes ocorrem no arquivo e eles são equiprováveis.

64. Consulte a **Seção 10.5.1**.

65. Muito raramente um byte será codificado com uma sequência de bits desse tamanho, mas, mesmo que isso aconteça, as pequenas sequências de bits associadas à maioria dos demais bytes compensam as eventuais sequências longas.

Capítulo 11 — Ordenação em Memória Principal

1. Consulte a **Seção 11.1.1**.

2. Há duas razões principais: uma de natureza didática e a outra de natureza pragmática. Do ponto de vista didático, ordenação oferece uma excelente oportunidade para a aprendizagem e a prática de análise de algoritmos. Do ponto de vista prático, cada algoritmo de ordenação tem seus méritos, de modo que cada um deles tem uma situação na qual ele é o mais apropriado.

3. Consulte a **Seção 11.1.1**.

4. (a) Consulte a **Seção 11.1.1**. (b) **CountingSort**, **BucketSort** e **RadixSort**.

5. (a) Normalmente, quando se tem uma lista de nomes para colocar em ordem alfabética, usa-se a seguinte abordagem: **1.** Encontra-se o nome que aparece em primeiro lugar considerando a ordem alfabética usual e escreve-se esse nome numa segunda lista. **2.** Risca-se esse nome na lista original. **3.** Repetem-se os **Passos 1** e **2** até que todos nomes na lista original tenham sido riscados e escritos na segunda lista. (b) Seleção direta (mas não é assim que **SelectionSort** é descrito, pois ele não precisa de uma segunda lista).

6. Consulte a **Seção 11.1.1**.

7. Consulte a **Tabela 11–12**.

8. Consulte a **Seção 11.1.1**.

9. Consulte a **Seção 11.1.1**.

10. Consulte a **Seção 11.1.1**.

11. (a) Não. (b) Não. (c) Sim.

12. Consulte a **Seção 11.1.1**.

13. Consulte a **Seção 11.1.1**.

14. (a) É um tipo de ordenação na qual ocorrem trocas *explícitas* de posições de elementos da tabela que está sendo ordenada. (b) **BubbleSort**, **QuickSort**, **HeapSort** e **SelectionSort**.

15. Consulte a **Seção 11.2.1**.

16. V. **Figura E–82**.

0	1	2	3	4	5	6	7	8	9
8	13	9	23	11	25	19	28	55	70

Figura E–82: Questão 16 — Capítulo 11

17. O número de vezes que o laço externo é executado é igual ao número de vezes que a variável *emOrdem* assume o valor *0*, que, no pior caso é $n - 1$ (por que?). Agora faça sua parte e complete a prova.

18. V. **Figura E–83**.

Melhor Caso		Caso Médio		Pior Caso	
Tabela ordenada	$\theta(n)$	**Tabela aleatória**	$\theta(n^2)$	**Tabela inversamente ordenada**	$\theta(n^2)$

Figura E–83: Questão 18 — Capítulo 11

19. (a) Quando a tabela é pequena ou já está quase ordenada. (b) É a negação de (a).

20. $\theta(1)$.

21. Consulte a **Seção 11.2.3**.

22. V. **Figura E–84**.

0	1	2	3	4	5	6	7	8	9
8	13	28	55	23	9	25	11	70	19

Figura E–84: Questão 22 — Capítulo 11

23. Em termos de análise assintótica, os resultados são os mesmos apresentados pelo método **BubbleSort** (v. **Figura E–83**).

24. Consulte a **Seção 11.2.3**.

25. $\theta(1)$.

26. (a) É um algoritmo que seleciona o próximo elemento a ser colocado em sua posição ordenada. (b) **SelectionSort** e **HeapSort**.

27. (a) Consulte a **Seção 11.2.2**. (b) Porque ele seleciona *diretamente* o próximo elemento a ser colocado em sua posição correta. O método **HeapSort** também é um algoritmo de seleção, mas o próximo elemento a ser colocado em ordem não é escolhido diretamente, pois ele sempre se encontra na raiz do heap.

28. V. **Figura E–85**.

0	1	2	3	4	5	6	7	8	9
8	9	11	28	23	55	25	13	70	19

FIGURA E–85: QUESTÃO 28 — CAPÍTULO 11

29. Os resultados são quase os mesmos apresentados pelo método **BUBBLESORT** (v. **Figura E–83**). A diferença é que o custo de melhor caso de **SELECTIONSORT** é $\theta(n^2)$.

30. $\theta(1)$.

31. Consulte a **Seção 11.2.2**.

32. $\theta(n)$.

33. Siga o modelo apresentado na **Figura 11–9**. A primeira iteração do laço externo do algoritmo é mostrado na **Figura E–86**. Agora faça sua parte completando o exercício.

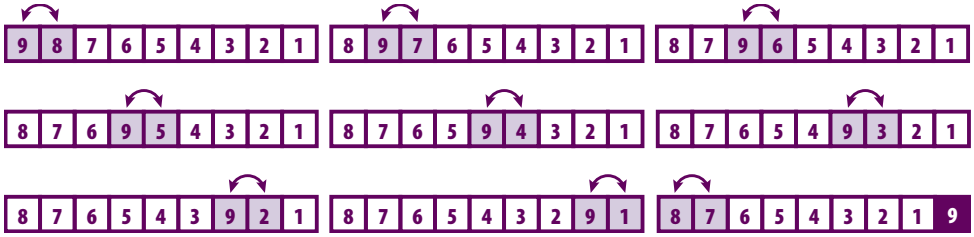

FIGURA E–86: QUESTÃO 33 — CAPÍTULO 11

34. Nesse caso, tanto **INSERTIONSORT** quanto **BUBBLESORT** serão executados com custo temporal $\theta(n)$. Ambos os algoritmos também usam o mesmo número de comparações nesse caso. O campeão nesse caso é o algoritmo **BUBBLESORT**, pois ele não efetua nenhuma outra operação além de comparações, enquanto **INSERTIONSORT** efetua duas atribuições em cada execução do corpo do laço externo.

35. **INSERTIONSORT** (porque esse algoritmo não efetua trocas).

36. (a) $n - 1$. (b) Uma.

37. Nada. Ele é inerentemente estável.

38. **SELECTIONSORT** é o algoritmo que efetua o menor número de trocas dentre todos os algoritmos baseados em trocas.

39. Porque, em qualquer caso de avaliação, seu custo é $\theta(n^2)$.

40. Não. Se você não souber explicar a razão para essa resposta, estude o **Capítulo 6** do **Volume 1** ou um bom texto sobre análise assintótica.

41. Zero, pois **INSERTIONSORT** não efetua trocas.

42. $4 \cdot (n - 1)$.

43. No pior caso, a atribuição da variável *emOrdem* no corpo do laço interno indica sempre que a lista está desordenada, e isso ocorre $n - 2$ vezes. Portanto, nesse caso, o corpo do laço externo será executado $n - 2$ vezes.

44. Consulte a **Seção 11.3.1**. (b) Pois, na maioria das situações práticas, ele é o método mais rápido (em inglês, *quick* significa rápido).

45. Consulte a **Seção 11.3.1**.

46. Depende de como o pivô é escolhido. Se a escolha for usando a abordagem de mediana de três, por exemplo, ele será sensível à ordenação inicial da tabela.

47. V. **Figura E–87**.

Melhor Caso		Caso Médio		Pior Caso	
Depende do pivô	$\theta(n \cdot \log n)$	**Tabela aleatória**	$\theta(n \cdot \log n)$	**Depende do pivô**	$\theta(n^2)$

FIGURA E–87: QUESTÃO 47 — CAPÍTULO 11

48. (a) Quando o estado de ordenação dos registros é desconhecido. (b) Quando a tabela pode estar ordenada ou inversamente ordenada.

49. $\theta(\log n)$.

50. (a) Cria-se uma árvore binária de busca usando a chave de ordenação como chave de busca. Efetuando um caminhamento em ordem infixa nessa árvore, obtêm-se os registros em ordem crescente. (b) Quando a árvore já existe. (c) Quando a árvore ainda não existe ou quando os registros estão direta ou inversamente ordenados. (d) Quando os dados estão ordenados ou inversamente ordenados. (e) $\theta(n^2)$. (f) $\theta(n \log n)$. (g) $\theta(n)$.

51. Consulte a **Seção 11.3.2**.

52. O conteúdo é o mesmo.

53. Não existe pior, melhor ou médio caso. O custo é sempre $\theta(n \cdot \log n)$.

54. $\theta(n)$.

55. (a) Quando há espaço suficiente em memória. (b) Em caso contrário.

56. Consulte a **Seção 11.3.3**.

57. (a) Não. (b) Usando uma segunda chave de ordenação.

58. (a) Quando a tabela já está ordenada ou inversamente ordenada e o pivô é escolhido como primeiro ou último elemento. (b) Usando uma técnica tal como mediana de três.

59. Consulte a **Seção 11.3.1**.

60. V. **Figura E–88**.

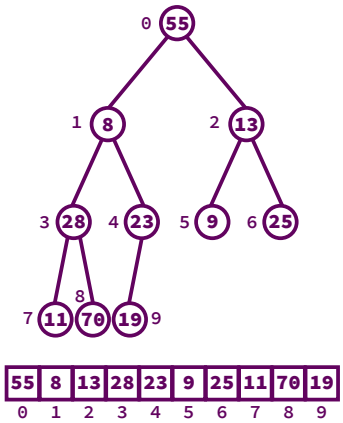

Figura E–88: Questão 60 — Capítulo 11

61. V. **Figura E–89**.

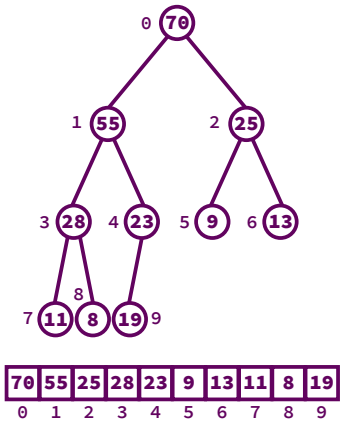

Figura E–89: Questão 61 — Capítulo 11

62. V. **Figura E–90**.

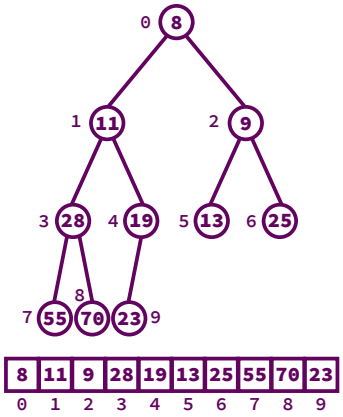

FIGURA E–90: QUESTÃO 62 — CAPÍTULO 11

63. O custo temporal é $\theta(n \cdot \log n)$ nos três casos.

64. Por duas razões: (1) o processo de criação do heap é baseado apenas na chave de ordenação (i.e., ele não respeita a ordenação prévia dos itens da lista) e (2) o processo de reordenação do heap após uma remoção também é baseado apenas na chave de ordenação.

65. Depende de como o pivô é escolhido. Mas o uso do algoritmo básico não é indicado quando a lista está ordenada, inversamente ordenada ou as chaves são iguais.

66. Porque, quando **QUICKSORT** é bem implementado, seu pior caso é muito difícil de acontecer.

67. Consulte a **Seção 11.3.1**.

68. Consulte a **Seção 11.3.1**.

69. Consulte a **Seção 11.3.1**.

70. V. **Figura E–91**.

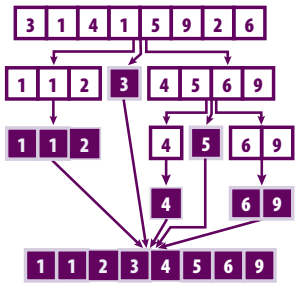

FIGURA E–91: QUESTÃO 70 — CAPÍTULO 11

71. (a) $\theta(n^2)$. (b) $\theta(n^2)$. (c) $\theta(n \log n)$.

72. **MERGESORT** não é afetado por ordenação prévia. Portanto seu custo temporal é $\theta(n \log n)$ nos três casos.

73. É sua vez. Consulte a **Seção 11.3.3** em caso de dúvida.

74. Heap de mínimo.

75. $\theta(n \log n)$.

76. Nos dois casos, antes de serem ordenadas, as tabelas precisam ser transformadas em heaps. Portanto o custo é o mesmo.

77. V. **Figura E–92**.

78. (a) $\theta(n^2)$. (b) Não tem jeito, é o mesmo.

79. Sim porque e tabela estiver ordenada ou inversamente ordenada, o elemento do meio será a mediana.

80. (a) Quando as chaves são todas iguais, o custo temporal de uma operação de remoção é $\theta(1)$ (faça sua parte e explique por quê). Como o algoritmo **HEAPSORT** efetua n operações de remoção seu custo temporal, nesse

caso, é $\theta(n)$. (b) O fato de uma tabela ter todas as chaves iguais não é considerado um *caso*, mas, mesmo que o fosse, seria um melhor caso e o teorema não seria aplicável.

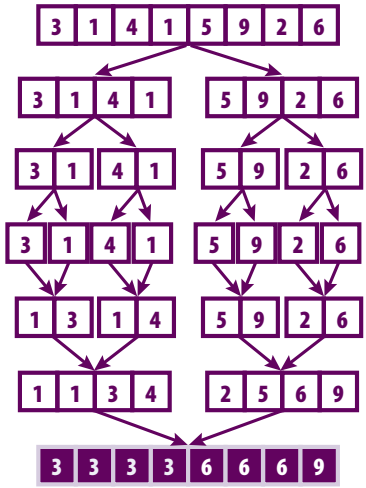

Figura E–92: Questão 77 — Capítulo 11

81. (a) Sim. (b) Porque, nesse caso, não se aplica a regra do produto de análise assintótica (v. **Capítulo 6** do **Volume 1**). Aqui aplica-se a regra da soma: $\theta(\log n) + \theta(n)$ resulta em $\theta(n)$.

82. De fato, a segunda chamada recursiva de **QuickSort** pode ser facilmente transformada em iteração, mas esse não é o caso da primeira chamada recursiva.

83. (a) Usando uma pilha (explícita) em vez da pilha de execução (v. **Capítulo 8** do **Volume 1**). (b) Não.

84. Custo temporal $\theta(n \log n)$ sempre, o que **QuickSort** não garante.

85. É sua vez.

86. Alterando o algoritmo de partição de modo que as chaves maiores fiquem à esquerda do pivô e as chaves menores fiquem à direita.

87. As provas serão feitas por indução forte (v. **Apêndice B** do **Volume 1**).

(a) Deve-se provar que $T(n) \geq \frac{1}{2} \cdot n \cdot \log n$.

Base da indução. Para $n = 1$, tem-se: $\frac{1}{2} \cdot 1 \cdot \log 1 = 0 \leq T(1) = 0$. Portanto aquilo que se deseja provar vale para $n = 1$.

Hipótese indutiva. Suponha que $T(n) \geq \frac{1}{2} \cdot n \cdot \log n$, $\forall n \mid 1 \leq n < k$.

Passo indutivo. Deve-se mostrar que $T(k) \geq \frac{1}{2} \cdot k \cdot \log k$.

Caso 1: *k* é par. Neste caso, tem-se o seguinte:

		Justificativa
	$T(k) = T(\lfloor k/2 \rfloor) + T(\lceil k/2 \rceil) + k - 1$	Por definição de $T(k)$
$\Rightarrow$	$T(k) = T(k/2) + T(k/2) + k - 1$	Por hipótese, k é par
$\Rightarrow$	$T(k) = 2 \cdot T(k/2) + k - 1$	
$\Rightarrow$	$T(k) \geq 2 \cdot \frac{1}{2} \cdot k/2 \cdot \log k/2 + k - 1$	Hipótese indutiva e o fato: $k/2 < k$
$\Rightarrow$	$T(k) \geq k/2 \cdot (\log k - \log 2) + k - 1$	Propriedade de logaritmos
$\Rightarrow$	$T(k) \geq k/2 \cdot (\log k - 1) + k - 1$	Idem
$\Rightarrow$	$T(k) \geq k/2 \cdot \log k - k/2 + k - 1$	Fatoração
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot k \cdot \log k + (k - 2)/2$	Idem
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot k \cdot \log k$	Se $k > 1$, $(k - 2)/2 \geq 0$

Caso 2: *k* é ímpar. Neste caso, como $k > 1$, $k = 2 \cdot q + 1$, para algum inteiro q, tal que $1 \leq q < k$. Logo $\lfloor k/2 \rfloor = q$ e $\lceil k/2 \rceil = q + 1$. Assim tem-se:

		Justificativa
	$T(k) = T(\lfloor k/2 \rfloor) + T(\lceil k/2 \rceil) + k - 1$	Por definição de $T(k)$
$\Rightarrow$	$T(k) = T(q) + T(q + 1) + 2q$	Por substituição
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot q \cdot \log q + \frac{1}{2} \cdot (q + 1) \cdot \log (q + 1) + 2q$	Hipótese indutiva e os fatos: $q < k$ e $q + 1 < k$
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot [q \cdot \log q + (q + 1) \cdot \log (q + 1) + 4 \cdot q]$	Fatoração
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot [q \cdot \log q + 2 \cdot q + (q + 1) \cdot \log (q + 1) + 2 \cdot q]$	$4 \cdot q = 2 \cdot q + 2 \cdot q$
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot [q \cdot (\log q + 2) + (q + 1) \cdot \log (q + 1) + q + 1]$	Fatoração e o fato: $q \geq 1 \Rightarrow 2 \cdot q \geq q + 1$
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot [q \cdot (\log q + 2) + (q + 1) \cdot (\log (q + 1) + 1)]$	Fatoração
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot [q \cdot (\log q + \log 4) + (q + 1) \cdot (\log (q + 1) + \log 2)]$	Propriedades de logaritmos
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot [q \cdot \log 4 \cdot q + (q + 1) \cdot \log (2 \cdot q + 2)]$	Idem
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot [q \cdot \log (2 \cdot q + 1) + (q + 1) \cdot \log (2 \cdot q + 1)]$	$q \geq 1 \Rightarrow 2 \cdot q + 1 \leq 2 \cdot q + 2 \leq 4 \cdot q$
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot (2 \cdot q + 1) \cdot \log (2 \cdot q + 1)$	Fatoração
$\Rightarrow$	$T(k) \geq \frac{1}{2} \cdot k \cdot \log k$	$k = 2 \cdot q + 1$, por hipótese

Os casos 1 e 2 mostram que $T(n) \geq \frac{1}{2} \cdot n \cdot \log n$. ■

(b) Deve-se provar que $T(n) \leq 2 \cdot n \cdot \log n$.

Base da indução. Para $n = 1$, tem-se: $2 \cdot 1 \cdot \log 1 = 0 \geq T(1) = 0$. Portanto a proposição vale para $n = 1$.

Hipótese indutiva. Suponha que $T(n) \leq 2 \cdot n \cdot \log n$, $\forall n \mid 1 \leq n < k$.

Passo indutivo. Deve-se mostrar que $T(k) \leq 2 \cdot k \cdot \log k$.

Caso 1: *k* é par. Neste caso, obtém-se o seguinte:

		Justificativa
	$T(k) = T(\lfloor k/2 \rfloor) + T(\lceil k/2 \rceil) + k - 1$	Por definição de $T(k)$
$\Rightarrow$	$T(k) = T(k/2) + T(k/2) + k - 1$	Por hipótese, k é par
$\Rightarrow$	$T(k) = 2 \cdot T(k/2) + k - 1$	
$\Rightarrow$	$T(k) \leq 2 \cdot 2 \cdot k/2 \cdot \log (k/2) + k - 1$	Hipótese indutiva e o fato: $k/2 < k$
$\Rightarrow$	$T(k) \leq 2 \cdot k \cdot (\log k - \log 2) + k - 1$	Propriedade de logaritmos
$\Rightarrow$	$T(k) \leq 2 \cdot k \cdot (\log k - 1) + k - 1$	Idem
$\Rightarrow$	$T(k) \leq 2 \cdot k \cdot \log k - 1$	Fatoração
$\Rightarrow$	$T(k) \leq 2 \cdot k \cdot \log k$	

Caso 2: *k* é ímpar. Neste caso, como $k > 1$, $k = 2 \cdot q + 1$, para algum inteiro q, tal que $1 \leq q < k$. Logo $\lfloor k/2 \rfloor = q$ e $\lceil k/2 \rceil = q + 1$. Portanto, neste caso, tem-se:

		Justificativa
	$T(k) = T(\lfloor k/2 \rfloor) + T(\lceil k/2 \rceil) + k - 1$	Por definição de $T(k)$
$\Rightarrow$	$T(k) = T(q) + T(q + 1) + 2 \cdot q$	Por substituição de k
$\Rightarrow$	$T(k) \leq 2 \cdot q \cdot \log q + 2 \cdot (q + 1) \cdot \log (q + 1) + 2 \cdot q$	Hipótese indutiva e os fatos: $q < k$ e $q + 1 < k$
$\Rightarrow$	$T(k) \leq 2 \cdot [q \cdot \log q + (q + 1) \cdot \log (q + 1) + q]$	Fatoração

		Justificativa
⇒	$T(k) \leq 2 \cdot [q \cdot (\log q + 1) + (q + 1) \cdot \log (q + 1)]$	Fatoração
⇒	$T(k) \leq 2 \cdot [q \cdot (\log q + \log 2) + (q + 1) \cdot \log (q + 1)]$	Propriedade de logaritmos
⇒	$T(k) \leq 2 \cdot [q \cdot \log (2 \cdot q) + (q + 1) \cdot \log (q + 1)]$	Idem
⇒	$T(k) \leq 2 \cdot [q \cdot \log k + (q + 1) \cdot \log k]$	$\log_2 x$ é crescente e os fatos: $q + 1 \leq q + q = 2 \cdot q < k$
⇒	$T(k) \leq 2 \cdot [(q + (q + 1)) \cdot \log k]$	Fatoração
⇒	$T(k) \leq 2 \cdot (2 \cdot q + 1) \cdot \log k$	
⇒	$T(k) \leq 2 \cdot k \cdot \log k$	$k = 2 \cdot q + 1$, por suposição

Os casos 1 e 2 mostram que $T(n) \leq 2 \cdot n \cdot \log n$. ■

88. Cada chamada recursiva do algoritmo **MergeSort** usa o mesmo array auxiliar de tamanho igual a n (v. implementação na **Seção 11.3.2**). Isso significa que o resultado armazenado nesse array numa dessas chamadas não é mais usado na próxima chamada recursiva desse algoritmo. Ou seja, é como se existissem cerca de $\log n$ arrays auxiliares, mas eles não fossem usados ao mesmo tempo. Como há $\log n$ chamadas recursivas aproximadamente, o custo espacial devido à criação de registros de ativação é $\theta(\log n)$. Assim o custo espacial do algoritmo **MergeSort** é obtido usando-se a regra da soma: $\theta(n) + \theta(\log n)$.

89. Consulte a **Seção 11.4.1**.

90. Se puder haver chaves negativas na lista a ser ordenada, deve-se encontrar a menor chave (*min*) da lista. Se o valor de *min* for negativo, cada chave da lista deve ser substituída por seu valor mais $|min|$. Depois disso, todas as chaves serão não negativas e o algoritmo descrito na **Seção 11.4.1** pode ser seguido normalmente. Antes de retornar, deve-se subtrair $|min|$ de cada chave da lista para que ela volte a ter seu valor original.

91. Consulte a **Seção 11.4.1** (especialmente a **Figura 11–33**).

92. Tente implementar esse algoritmo sem o array auxiliar e veja o que acontece.

93. Os três últimos passos do algoritmo podem ser substituídos por: armazene na lista original, a partir de seu primeiro índice, o índice do array de contagem, repetindo esse armazenamento pelo número de vezes determinado pelo valor do respectivo elemento do array de contagem. Por exemplo, suponha que `lista[]` seja o array contendo a lista original, `cont[]` seja o array de contagem e `tam` seja o tamanho desse último array. Então o trecho de programa a seguir poderia implementar esse novo passo do algoritmo.

```
i = 0;

for (j = 0; j < tam; j++)
   for (k = 0; k < cont[k]; ++k)
      lista[i++] = j;
```

94. Consulte a **Seção 11.4.1**.

95. Consulte a **Seção 11.4.1**.

96. A fase mais complicada de **BucketSort** é a fase de distribuição porque pode ser difícil encontrar um meio de distribuir as chaves uniformemente.

97. Tipicamente, a ordenação dos elementos em cada coletor é feita por um algoritmo que usa comparações. (Mas, mesmo assim, ele ainda é considerado um algoritmo que não é *baseado* em comparação.)

98. Consulte a **Seção 11.4.2**.

99. Consulte a **Seção 11.4.2**.

100. Ordenação com coletores requer o uso de um coletor para cada elemento, enquanto ordenação por contagem armazena um único número (a quantidade de elementos) por coletor.

101. Consulte a **Seção 11.4.2**.

102. Consulte a **Seção 11.4.3**.

103. Quando as chaves possuem os mesmos tamanhos (i.e., o número de componentes de cada chave é fixo).

104. Porque ele não compara as chaves que serão ordenadas.

105. Consulte a **Seção 11.4.3**.

106. Alterando-se o último passo do algoritmo de tal modo que as filas sejam esvaziadas começando-se com a fila associada ao maior dígito e terminando-se com a fila associada ao menor dígito.

107. Não. Estude a **Seção 11.4.3** e explique você mesmo.

108. Como se estão ordenando inteiros de tamanhos fixos, **RadixSort** é muito mais eficiente do que qualquer algoritmo de ordenação baseada em comparação. Os demais algoritmos com custos temporais $\theta(n)$ também não são tão eficientes quanto **RadixSort**.

109. Não, porque uma função genérica de ordenação requer um ponteiro para função como parâmetro (v. **Capítulo 11** do **Volume 1**). Esse ponteiro é usado pela função de ordenação para chamar outra função que compara elementos. Como o algoritmo **BucketSort** não usa comparação esse ponteiro não faz sentido. Além disso, esse algoritmo faz suposições sobre a natureza das chaves que serão ordenadas, de modo que ele não pode ser aplicado a chaves genéricas. (Consulte o **Capítulo 11** do **Volume 1** para entender como uma função genérica de ordenação é implementada.)

110. Todos usam coletores. Quando se usa ordenação por base e os números são decimais, por exemplo, 10 coletores, são necessários, um para cada dígito de 0 a 9. Quando se usa ordenação por contagem, é preciso um coletor para cada valor único na lista de entrada. Quando se usa **BucketSort**, em princípio, não se sabe quantos coletores se estará usando.

111. **BucketSort** tem custo temporal $\theta(n + k)$ e custo espacial $\theta(n + k)$. Quando as chaves são densamente distribuídas, o valor de k é relativamente pequeno. Os custos temporal e espacial de **RadixSort** independem do tamanho do intervalo de chaves.

112. Não. Sem entrar no mérito da afirmação de que as palavras da língua portuguesa são formadas com as 26 letras do alfabeto romano, do ponto de vista de programação, há muito mais caracteres. Por exemplo, `'A'`, `'a'`, `'ã'` e `'à'` são caracteres diferentes.

113. (a) Se a ordenação for feita manualmente, primeiro se efetua uma ordenação com coletores, depois ordenam-se os valores que compartilham o primeiro dígito. Isso funciona, mas divide o problema em muitos subproblemas. Por outro lado, **RadixSort** nunca divide a lista; i.e., ele aplica ordenação com coletores várias vezes à mesma lista. (b) Em **RadixSort**, a última passagem de ordenação com coletores é aquela que tem mais efeito na ordenação geral, de modo que ela deve usar o dígito mais significativo. As ordenações anteriores são usadas apenas para lidar com o caso em que dois elementos têm a mesma chave (*mod 10*) na última passagem.

114. Porque, uma vez que um elemento tenha sido atribuído a um lugar de acordo com o valor do dígito numa posição menos significativa, seu lugar não deve mudar a não ser que a ordenação em um dos dígitos mais significativos requeira.

115. As chaves devem ser uniformemente distribuídas.

116. $\theta(n \log n)$.

117. Consulte a **Seção 11.5**.

118. Como foi argumentado na **Seção 11.5**, a árvore de decisão associada a um algoritmo de ordenação baseado em comparação possui, no mínimo, $n!$ folhas. Como se está em busca de um limite mínimo, pode-se considerar que a árvore possui exatamente esse número de folhas. O nível de uma folha corresponde ao número de comparações que o algoritmo efetua sobre a lista de entrada para obter permutação associada a essa folha. O objetivo aqui é mostrar que a profundidade média dessa folha é, pelo menos, $\lfloor \log_2 (n!) \rfloor$. Se a árvore em tela for repleta, a prova estará completa, porque todas as folhas estarão no mesmo nível e a profundidade delas é $\lceil \log_2 (n!) \rceil$ ou $\lfloor \log_2 (n!) \rfloor$ (v. **Capítulo 12** do **Volume 1**). Resta mostrar que,

dentre todas as árvores com um certo número de nós, aquela que minimiza sua profundidade média é uma árvore binária completa.

Suponha que se tenha uma árvore binária que não seja completa. Suponha ainda que duas folhas irmãs que se encontram no nível mais baixo da árvore sejam removidas e acrescentadas como filhas de uma folha que se encontre na menor profundidade. Como a diferença entre a maior e a menor profundidade dessas folhas é, pelo menos, dois (caso contrário, a árvore seria completa), essa operação reduz a profundidade média das folhas (e da própria árvore). Como qualquer árvore que não é completa pode ser modificada dessa maneira para que ela tenha uma profundidade média menor, tal árvore não pode ser aquela que minimiza sua profundidade média. Logo a árvore com menor profundidade média deve ser completa. ■

119. (a) Um algoritmo de divisão e conquista divide o problema a ser resolvido em subproblemas cada vez menores, resolve-os separadamente e, finalmente, combina suas soluções. (b) Porque eles seguem essa descrição.

120. Pesquise na internet.

121. Resposta curta: busca binária não segue a descrição apresentada na resposta da questão **119**. Resposta longa: busca binária divide sucessivamente a tabela na qual será efetuada a busca em duas partes (assim como faz **QuickSort**, por exemplo), mas, em seguida, o algoritmo não efetua busca nessas duas partições (i.e., ele escolhe uma delas para efetuar a busca). Concluindo, o algoritmo de busca binária é mais precisamente classificado como um **algoritmo de redução e conquista**, visto que ele reduz sucessivamente o tamanho do problema para, então, resolvê-lo (i.e., conquistá-lo).

122. (a) 99. (b) 9900. (c) 9900. (d) 99.

123. (a) 9801. (b) 9900. (c) 9900. (d) 9900.

124. O custo de implementação.

125. Durante a fase de testes de um programa, precisa-se ordenar uma tabela relativamente pequena e o computador usado no desenvolvimento do programa é possante.

126. Devido ao ônus associado às chamadas de funções (v. **Capítulo 4** do **Volume 1**).

127. Consulte o **Capítulo 4** do **Volume 1**.

128. Porque não faz sentido considerar uma sequência de operações de ordenação.

129. **InsertionSort**.

130. Porque apesar de ambos terem custo temporal assintótico $\theta(n^2)$, **InsertionSort** é mais rápido (e também não é tão difícil de implementar).

131. **RadixSort**.

132. Consulte a **Seção 11.7**.

133. $\theta(n \cdot \log n)$ ou $\theta(n)$, dependendo do tipo de chave e da ordenação usada para ordenar a lista.

134. Todos os algoritmos com custo temporal $\theta(n^2)$ examinados no **Capítulo 11**, pois todos eles acessam os elementos da tabela sequencialmente e, portanto, apresentam padrão de referência sequencial (v. **Seção 11.5**). Dentre os algoritmos com custo temporal $\theta(n \cdot \log n)$ examinados na **Seção 11.3**, o único que apresenta boa localidade de referência é **QuickSort**. Nenhum algoritmo com custo temporal $\theta(n)$ apresentado na **Seção 11.4** possui boa localidade de referência.

135. Consulte a **Seção 11.7.2**.

136. Consulte a **Seção 11.3.2**.

137. (a) Consulte a **Seção 11.8.2**. (b) Consulte a **Seção 11.8.2**.

138. Consulte a **Seção 11.8.3**.

139. (a) Consulte a **Seção 11.8.5**. (b) Consulte a **Seção 11.8.5**.

Capítulo 12 — Ordenação em Memória Secundária e Bulkloading

1. Consulte a introdução do **Capítulo 12**.
2. Porque, se a ordenação é externa, supõe-se que não seja conveniente carregar o arquivo inteiro em memória principal e o custo temporal [$\theta(n \log n)$] desse algoritmo é considerado alto para memória externa.
3. Pelo número de transferências de blocos entre memória principal e memória secundária.
4. Consulte a **Seção 12.1**.
5. (a) Ordenação e intercalação. (b) Consulte a **Seção 12.1**.
6. Consulte a **Seção 12.2.1**.
7. V. tabela a seguir.

Intercalação Binária	MergeSort
É iterativo	**É recursivo**
Atua em memórias secundária e principal	**Atua apenas em memória e principal**
Usa um algoritmo auxiliar de ordenação	**Não usa algoritmo auxiliar de ordenação**
Tem custo espacial $\theta(1)$ em memória principal	**Tem custo espacial $\theta(n)$**

8. Em memória principal, o que conta é a taxa de crescimento do número de operações em função do tamanho da tabela a ser ordenada. Em memória secundária, o que deve ser levado em consideração é o número de operações de entrada e saída requeridas pelo algoritmo de ordenação.
9. Consulte a **Seção 12.2.3**.
10. (a) 1250. (b) 13. (c) O número de acessos ao disco é dado por $2 \cdot N \cdot (\lceil \log_2 N \rceil + 1)$, em que N é o número de blocos do arquivo, o que resulta em 960.000.
11. Consulte a **Seção 12.3**.
12. O heap é usado para escolher o menor elemento a ser incluído no array que armazena o resultado da intercalação.
13. Nos dois casos, a técnica de intercalação é essencialmente a mesma.
14. $\theta(n \cdot k \cdot \log k)$.
15. (a) $\theta(n \cdot k \cdot \log k)$ ou $\theta(n \cdot k)$, dependendo do tipo de chave de ordenação (consulte o **Capítulo 10**). (b) Não. Nesse caso é melhor copiar todos os dados para o array que conterá o resultado e ordená-lo.
16. Consulte a **Seção 12.4**.
17. É o mesmo que intercalação multíplice.
18. Consulte a **Seção 12.4.1**.
19. É mais rápida porque requer um número menor de operações de entrada/saída.
20. Consulte a **Seção 12.4.1**.
21. O tempo de processamento em memória principal é desprezível em face ao tempo de operações de entrada e saída.
22. Consulte a **Seção 12.4**.
23. (a) É um elemento que ainda não foi intercalado. (b) É um elemento que ainda não foi alterado.
24. O buffer de entrada é recarregado com dados do arquivo (série) associado a ele.
25. O buffer é descarregado para o (i.e., escrito no) arquivo associado a ele.
26. Consulte a **Seção 12.4.2**.
27. Consulte a **Seção 12.4.2** e faça sua parte.
28. Consulte a **Seção 12.4.4**.

29. Usando-se a **Fórmula 12–2** apresentada na **Seção 12.4.4**, tem-se que o tamanho máximo do arquivo em bytes deverá ser:

$$\frac{M^2}{B} = \frac{(512 \cdot 2^{20})^2}{8 \cdot 2^{10}} = 2^{45}$$

Dividindo-se esse valor pelo tamanho de um registro, obtém-se que o número máximo de registros é 2^{27}.

30. Usando-se a **Fórmula 12–2** apresentada na **Seção 12.4.4**, tem-se:

$$\frac{M^2}{B} \approx 2^{48} = 256\,TiB$$

31. **Análise da Fase 1**. O número de blocos que cabe em *100* MB de memória é $100 \times 10^6/(20 \times 10^3)$ (tamanho de memória principal/tamanho de bloco) ou aproximadamente *5.000* blocos. Assim preenche-se a memória *20* vezes (= ⌈*100.000/5.000*⌉), ordenam-se os registros em memória principal e escrevem-se *20* séries ordenadas no disco. Nessa fase, são lidos e escritos *100.000*, de forma que ocorrem *200.000* transferências de disco e, assim, o tempo gasto será:

$$200.000 \times 11{,}5\ ms = 2.300\ seg = 38\ min$$

Análise da Fase 2. Cada bloco contendo registros das listas ordenadas é lido em arquivo exatamente uma vez. Assim o número total de leituras de bloco é *100.000* nessa fase (que é o mesmo resultado obtido para a primeira fase). De modo similar, o número de escritas de bloco na segunda fase também é *100.000*. Logo a segunda fase consumirá mais *38* minutos.

Total. As duas fases consumirão *76* minutos.

32. O número máximo de buffers de entrada que se pode ter na **Fase 2** é $M/B - 1$, sendo, pelo menos, um bloco por buffer de entrada e um bloco para o buffer de saída. O número máximo de sublistas ordenadas na **Fase 1** é $M/B - 1$ e o número máximo de registros que pode ser ordenado numa sublista é M/R. Consequentemente é possível ordenar: $(M/R) \cdot [(M/B) - 1] \cong M^2/R \cdot B$ registros.

33. O número máximo de registros é dado por M^2/RB, enquanto o número de bytes é dado por M^2/B. Assim $M^2/RB = (10^8)^2 / (100 \times 20.000) = 6 \times 10^9$ registros de *100* bytes cada. Ou seja, um arquivo de *600* GB (com apenas *100* MB de memória!).

34. Deseja-se ter um buffer maior para cada série no qual sejam armazenados vários blocos ao mesmo tempo, levando vantagem de acesso sequencial e, assim, deseja-se ter um número relativamente pequeno de séries grandes *s*:

- **Fase 1**: $s > N/M$, porque se pode ordenar no máximo *M* bytes de entrada em memória principal
- **Fase 2**: $s < M/B - 1$, porque não se pode ter mais buffers de entrada de tamanho pelo menos um bloco.

Portanto o número de séries deve satisfazer a relação:

$$M/B - 1 > s > N/M$$

35. O tamanho máximo que se pode ordenar em três passos é: M^3/B^2. Portanto, em três passos, pode-se ordenar um arquivo com $(10^8)^3 / 4 \times 10^8 = 2{,}5 \times 10^{15}$ bytes $= 2{,}5\ PB$.

36. Consulte a **Seção 12.5**.

37. Consulte a **Seção 12.6**.

38. (a) Árvores da família de árvores B. (b) Porque são estruturas de dados apropriadas para memória secundária contendo nós com chaves ordenadas e que crescem de baixo para cima.

39. Consulte a **Seção 12.6**.

40. 100%, exceto, talvez, para a última folha (i.e., aquela mais à direita).

41. (a) 50%, exceto, talvez, para o último nó interno de cada nível. (b) Alterando o algoritmo de divisão de nós, de modo que o nó da esquerda continue repleto e o nó da direita fique temporariamente vazio. Isso poderá violar uma das regras de árvores B+, pois o último nó de cada nível poderá ficar com um número de nós abaixo do permitido (mas é por uma boa causa).

42. Supondo que já se tenha o arquivo ordenado e que o grau da árvore seja *G*, um algoritmo de *bulkloading* para árvores B é apresentado na **Figura E–93**.

ALGORITMO BULKLOADINGDEÁRVORESB

ENTRADA: Arquivo contendo pares chave/índice ordenados
SAÍDA: Arquivo contendo a árvore B construída

1. Leia *G* pares no arquivo ordenado e preencha a primeira folha da árvore mantida em memória principal (*folha esquerda*)
2. Escreva essa folha no arquivo que contém a árvore
3. Se o final do arquivo foi atingido, torne essa folha a raiz da árvore e encerre
4. Crie um nó em memória principal e faça com que seu primeiro filho aponte para a primeira folha
5. Crie uma lista indexada de nós ativos para armazenar os nós internos mantidos em memória principal
6. Enquanto o final do arquivo de pares não for atingido, faça o seguinte:
   - 6.1 Leia *G* + *1* pares no arquivo ordenado
   - 6.2 Insira os *G* últimos pares lidos na próxima folha em memória principal (*folha direita*)
   - 6.3 Acrescente a folha direita ao arquivo que contém a árvore e obtenha sua posição nesse arquivo
   - 6.4 Insira o primeiro par lido e a posição dessa última folha em arquivo no nó que conterá o pai dela usando um algoritmo similar a **INSEREACIMAEMÁRVOREB+** descrito na **Figura 12–19**
   - 6.5 Faça com que a folha esquerda passe a ser a folha direita
7. Atualize no arquivo da árvore todos os nós ativos

FIGURA E–93: QUESTÃO 42 — CAPÍTULO 12

Agora faça sua parte e escreva o algoritmo equivalente a **INSEREACIMAEMÁRVOREB+** para árvores B.

43. (a) É a mesma resposta da questão **40**. (b) É a mesma resposta da questão **41 (a)**.

44. V. **Figura E–94**.

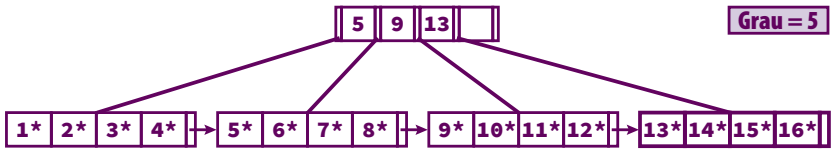

FIGURA E–94: QUESTÃO 44 — CAPÍTULO 12

45. V. **Figura E–95**.

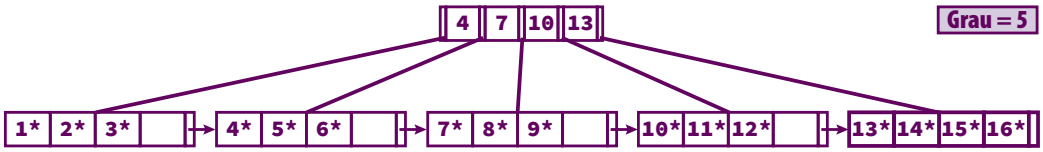

FIGURA E–95: QUESTÃO 45 — CAPÍTULO 12

46. O número de nós mantidos em memória principal é, no máximo, igual ao número corrente de níveis da árvore, incluindo o nível que contém as folhas.

47. (a) Como as chaves dos registros estão ordenadas em ordem crescente, cada nova folha pertencerá ao nó da direita mais recentemente criado (i.e., resultante de uma divisão de nós) no primeiro nível interno logo acima das chaves. Um raciocínio similar se aplica aos nós que estão em níveis superiores. (b) A altura da árvore pode ser maior do que poderia ser se esse não fosse o caso.

48. V. **Figura E–96**.

49. V. **Figura E–97** na **página 782**.

50. Consulte a **Seção 12.7.1**.

51. O custo de transferência (i.e., acesso a disco) é $\theta(n)$ e o custo temporal de ordenação em memória principal é $\theta(n \cdot log\ n)$.

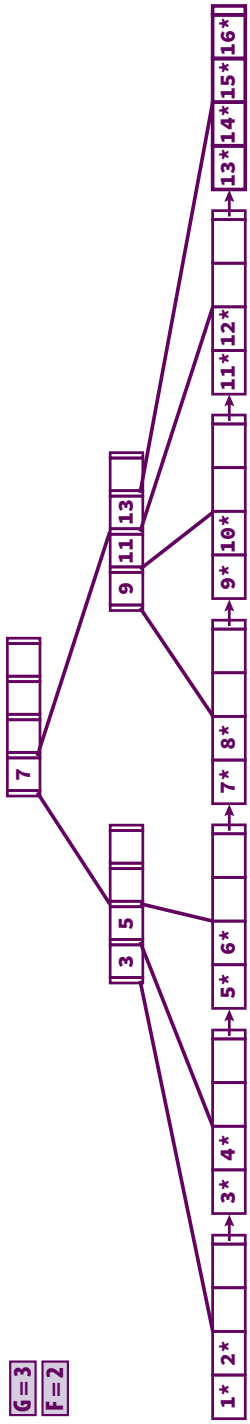

Figura E–96: Questão 48 — Capítulo 12

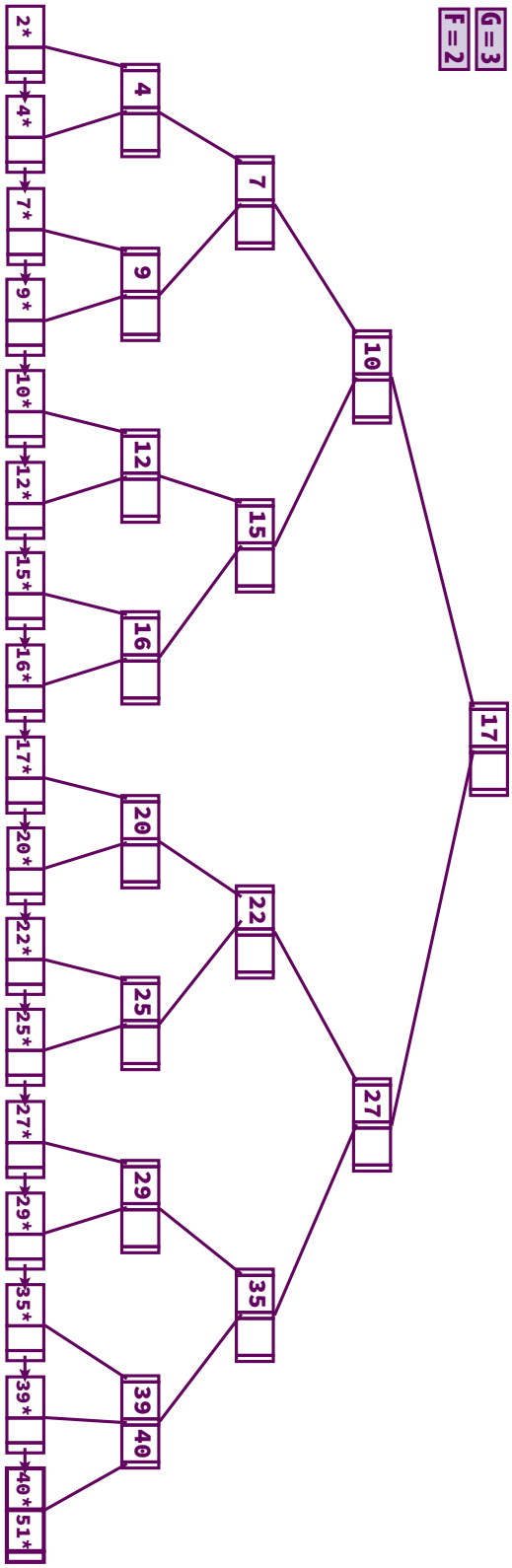

Figura E–97: Questão 49 — Capítulo 12