Respostas e Sugestões para os Exercícios de Revisão

Capítulo 1 — Elementos Básicos da Linguagem C

1. (a) É a especificação oficial da linguagem. (b) Principalmente, por razões de portabilidade. (c) É o padrão da linguagem C definido por um comitê ISO. (d) C11.
2. Significa que ela não é especificado por nenhum padrão aceito de C e, portanto, não é portável.
3. Consulte a **Seção 1.2**.
4. **int**, **double** e **void** são palavras-chave; `$a` e `a$` usam $, que não é um caractere permitido em identificadores de C.
5. Consulte a **Seção 1.2**.
6. Notação convencional e notação científica.
7. Consulte a **Seção 1.2**.
8. `'\n'` provoca quebra de linha e `'\t'` causa tabulação horizontal.
9. (a) Um string constante consiste em um ou mais caracteres constantes entre aspas. (b) Um string constante pode conter mais de um caractere; os delimitadores de strings (aspas) e caracteres constantes (apóstrofos) também são diferentes.
10. (a) Dividindo o string em substrings separados por espaços em branco. (b) Melhora a legibilidade.
11. (a) Sim. É a soma do valor inteiro associado a `'A'` com o valor inteiro associado a `'B'`. (b) Não, pois ela depende de implementação. (c) Depende do código de caracteres usado.

12. O resultado de `'Z' - 'A'` depende de implementação (i.e., ele depende do código de caracteres usado, o que não é especificado por nenhum padrão de C).

13. (a) `9` (b) `d`.

14. (a) Um operador representa uma operação elementar de uma linguagem de programação. (b) É um valor sobre o qual atua um operador. (c) É uma combinação legal de operadores e operandos.

15. Consulte o **Apêndice A**.

16. (a) É a alteração de valor produzida pelo operador em um de seus operandos. (b) Atribuição, incremento e decremento.

17. Conjunção, disjunção, condicional e vírgula.

18. Não há como alterar essa propriedade.

19. Porque a expressão à direita da primeira atribuição pode produzir dois valores válidos, dependendo da ordem com que os operandos do operador de multiplicação são avaliados.

20. (a) É um operador utilizado para efetuar comparações de valores numéricos. (b) Consulte a **Seção 1.3**. (c) 0 ou 1 (d) Não (v. **Apêndice A**).

21. (a) Negação (**!**), conjunção (**&&**) e disjunção (**||**). (b) `0` ou `1`.

22. (a) É o fato de um operador nem sempre avaliar todos os seus operandos. (b) Conjunção (**&&**) e disjunção (**||**).

23. Não. Na expressão (2) o valor de `y` é sempre incrementado, o que não ocorre sempre no caso da expressão (1) devido ao curto-circuito do operador de disjunção.

24. Se o valor de `x` for `0`, a avaliação da expressão (1) causa o aborto devido a tentativa de cálculo de resto de divisão por `0`. Na expressão (2), o curto-circuito do operador de conjunção impede que a expressão `y%x` seja avaliada quando `x` é `0`.

25. Zero.

26. (a) Sim. (b) Zero, porque o resultado de `x = 0` é sempre `0`.

27. (a) Uma definição de variável especifica o nome e o tipo de uma variável. (b) Uma definição de variável informa o compilador como ele deve interpretar o conteúdo da variável, a quantidade de espaço que deve ser alocada e o identificador associado à variável.

28. Para facilitar rápida identificação visual das categorias às quais os identificadores pertencem.

29. `1`.

30. Consulte a **Tabela 1–2**.

31. (a) `i` recebe `2`; `d` recebe `2.0`. (a) `i` recebe `2`; `d` recebe `2.5`.

32. (a) É uma conversão de tipos efetuada automaticamente pelo compilador. (b) Para permitir mistura de tipos numa expressão. (c) Em expressões envolvendo tipos diferentes (incluindo atribuição), em passagem de parâmetros, em retorno de função e quando o tipo de uma variável ou parâmetro é **char**. (d) Consulte a **Seção 1.6**.

33. (a) Não. (b) Sim. (c) Real.

34. (a) Para converter `2` em **double**. (b) Não, pois essa conversão ocorreria implicitamente. (c) A conversão explícita melhora a legibilidade da instrução.

35. (a) É uma operação que acrescenta `1` a uma variável. (b) É uma operação que subtrai `1` de uma variável. (c) São os operadores representados por `++` e `--`.

36. Se x é uma variável, o resultado de `++x` é `x + 1` e o resultado de `x++` é `x`.

37. Incremento não pode ter uma expressão como operando.

38. Se a variável `j` for avaliada primeiro, a expressão `j * j++` será igual a `4 * 4`, que resulta em `16`. Se a expressão j++ for avaliada primeiro, a expressão `j * j++` será igual a `5 * 4`, que resulta em `20`.

39. Não. Se pois o operador de soma não possui ordem de avaliação de operandos definida, de modo que há dois resultados possíveis e legítimos: `5` e `3`.

40. (a) Por meio de delimitadores de comentários. (b) Como se fossem espaços em branco. (c) Para facilitar seu entendimento assim como prover outras informações úteis sobre ele. (d) Um programa sem comentários pode requerer que o leitor faça inferências para tentar entender o que (e como) o programa faz.

41. Comentários encontrados em livros de ensino de programação são didáticos.

42. Porque comentários redundantes desviam a atenção de quem lê o programa.

43. (a) Uma biblioteca é uma coleção de componentes prontos para uso em programas. (b) Usando os componentes providos por uma biblioteca, o programador não precisa programá-los. (c) Estritamente falando, sim. Mas, uma linguagem de programação que não possui biblioteca é severamente limitada do ponto de vista pragmático. (c)

44. (a) É a biblioteca que deve acompanhar qualquer compilador que segue o padrão da linguagem C. (b) Nenhum componente de biblioteca têm significado especial para um compilador de C. Portanto não fazem parte dessa linguagem. (c) Porque, apesar de acompanhar obrigatoriamente qualquer compilador padrão, seus componentes não fazem parte da linguagem em si.

45. É um arquivo que contém informações úteis para o compilador sobre os componentes de uma biblioteca ou módulo de biblioteca.

46. (a) Para incluir arquivos de cabeçalhos em programas. (b) O conteúdo do arquivo cujo nome acompanha a diretiva é inserido no arquivo que a contém a partir da linha na qual ela se encontra.

47. Normalmente, os símbolos `<` e `>` são usados para cabeçalhos da biblioteca padrão, enquanto aspas são usadas para cabeçalhos de outras bibliotecas.

48. Consulte a **Seção 1.10**.

49. É uma sequência de caracteres que começa com **%** e especifica como um valor deve ser escrito.

50. (a) É um string que contém especificadores de formato. (b) Quaisquer caracteres.

51. (a) Usando o especificador de formato **%c**. (b) Usando o especificador de formato **%d**.

52. (a) Incluindo `\n` num string de formatação de **printf()**. (b) Incluindo `\t` num string de formatação de **printf**().

53. Seis.

54. (a) `2.456900`. (b) Um valor que não faz sentido. (c) `2.46`. (d) `2.46`.

55. Consulte a **Seção 1.11**.

56. Consulte a **Seção 1.11**.

57. (a) Consulte a **Seção 1.11**. (b) Por duas razões. A principal delas é que a função **getchar()** nem sempre retorna um valor associado a um caractere; i.e., ela também pode retornar **EOF**, que, tipicamente não cabe numa variável do tipo **char**. A segunda razão é mais sutil: devido a conversões implícitas de alargamento, não se costumam definir varáveis, parâmetros ou tipos de retorno com o tipo **char**. Do ponto de vista prático, essa segunda justificativa não faz diferença num programa, mas pode demonstrar falta de conhecimento do programador. (c) Quando encontra caracteres remanescentes no buffer associado a **stdin**, **getchar()** não interrompe a execução de um programa.

58. Porque o programador tem controle total sobre o que é escrito na tela, mas não tem nenhum controle sobre o que um usuário de seu programa pode digitar.

59. Por intermédio de uma função que lê caracteres e os converte, quando possível, em números, como faz a função **scanf()**, por exemplo.

60. Tipicamente, apenas especificadores de formato e espaços em branco.

61. Significam parâmetros que não têm quantidades ou tipos especificados.

62. Consulte a **Seção 1.11**.

63. (a) O número de variáveis que tiveram seus valores modificados pela função **scanf()**. (b) Para testar se algum valor que deveria ter sido lido foi realmente lido.

64. (a) (i) Faltou preceder a variável `x` com **&**. (ii) o especificador de formato deveria ser **%d**. (b) O compilador GCC é capaz de apresentar mensagens de advertência nos dois casos. Mas, nem todo compilador o faz nem tem obrigação de fazê-lo.

65. (a) A função **scanf()** não leu o valor que deveria ler. (b) A função **scanf()** leu o valor que deveria ler e o atribuiu à variável `x`. (c) Ocorreu erro de leitura.

66. (a) Sim. (b) Não.

67. O fato de ambos usarem especificadores de formato.

68. (a) Um. (b) Dois.

69. (a) É um identificador que representa um valor constante. (b) Usando **#define**, o nome da constante e seu valor. (c) Legibilidade, manutenibilidade e portabilidade são favorecidas com o uso de constantes simbólicas.

70. Porque uma definição de constante simbólica é uma diretiva e diretivas terminam com quebra de linha, e não com ponto e vírgula.

71. Consulte a **Seção 1.13**.

72. Consulte a **Seção 1.13**.

73. Não.

74. Melhora de legibilidade apenas.

75. Porque compiladores de C não respeitam o conceito de enumeração.

76. `C1` recebe `-1` (óbvio), `C2` recebe `0` (`C1 + 1`), `C3` recebe `0` (óbvio), `C4` recebe `1` (`C3 + 1`).

77. Consulte a **Seção 1.14**.

78. Consulte a **Seção 1.14**.

79. Porque uma expressão sem efeito colateral não altera nenhum dado processado pelo programa.

80. (a) Ponto e vírgula. (b) Não (v. **Seção 1.14**).

81. (a) É a sequência e a frequência com que as instruções do programa são executadas.(b) Fluxo natural de execução de um programa é a execução sequencial da primeira à última instrução do programa, sendo cada uma delas executada uma única vez e na ordem em que se encontra no programa. (c) Por meio de estruturas de controle.

82. Laços de repetição, desvios condicionais e desvios incondicionais.

83. (a) São instruções que alteram a ordem de execução de outras instruções de um programa de acordo com o resultado da avaliação de uma expressão condicional. (b) **if-else** e **switch-case**.

84. (a) São instruções que permitem o desvio do fluxo de execução de um programa independentemente da avaliação de qualquer condição. (b) **break**, **continue** e **goto**.

85. (a) É uma instrução que altera a frequência com que outras instruções são executadas. (b) **while**, **do-while** e **for**.

86. Consulte a **Seção 1.15.1**.

87. (a) Cinco vezes. (b) `5` e `5`. (c) Sim, pois os resultados das expressões contendo ++ e -- não são usados.

88. (a) No caso específico em que `x` e `y` assumem os valores iniciais que aparecem nas questões, não há diferença. Mas, se inicialmente o valor de `x` for maior do que ou igual a `y`, haverá diferença. (b) Cinco vezes. (c) `5` e `5`.

89. (a) Cinco vezes. (b) `6` e `4`, respectivamente . (c) Não, pois os resultados das expressões contendo `++` e `--` são usados.

90. (a) Essa questão é uma pegadinha, porque aparentemente o valor de `i++` será sempre maior do que zero e, assim, o laço **while** nunca terminará. Acontece, entretanto, que o incremento contínuo da variável i terminará acarretando em overflow e essa variável se tornará negativa, encerrando o laço. (b) Valor final de `i`: `-2147483647`. Esse valor corresponde ao maior valor do tipo **int** com sinal invertido na implementação na qual o programa foi compilado.

91.

```
int i = 0;

while (i < 10) {
   printf("i = %d\n", i);
   ++i;
}
```

92. (a) É uma expressão que, quando resulta num valor diferente de zero, provoca o encerramento do laço. (b) `i <= 0`, que é a negação da expressão condicional do laço.

93. (a) Consulte a **Seção 1.15.1**. (b) **for**.

94. A expressão `0 < x < 10` resulta sempre em `1`, pois `0 < x` resulta em `0` ou `1` e, em qualquer caso, esse valor é menor do que `10`.

95. Instruções **break** previnem o efeito cascata de instruções **switch-case**.

96. Para que a parte **default** não seja esquecida quando ela for necessária.

97. Consulte a **Seção 1.15.3**.

98. (a) Consulte a **Seção 1.15.3**. (b) Porque o uso desmesurado de **goto** pode gerar código espaguete.

99. Não. O desvio só é permitido para instruções que façam parte da mesma função na qual se encontra a instrução **goto**.

100. Ela pode deixar de ser válida quando o corpo da instrução for contém uma instrução **continue**.

101. Consulte a **Seção 1.16.1**.

102. Consulte a **Seção 1.16.1**.

103. `2`.

104. Se você teve dificuldade para responder a questão anterior sabe a resposta desta questão.

105. `printf("x %se' positivo", x > 0 ? "" : "nao ");`

106. É a menor precedência da linguagem C.

107. Consulte a **Seção 1.16.2**.

108. Consulte a **Seção 1.16.3**.

109. (a) **size_t**. (b) Derivado.

110. O operador **sizeof** pode ter um tipo de dado como operando.

111. Para saber o tipo do valor resultante de uma expressão não é necessário conhecer o valor da expressão.

112. Deve-se ter cuidado para não misturar inteiros com sinal e sem sinal numa mesma expressão que não seja de atribuição.

113. Consulte a **Seção 1.16.3**.

114. Porque o resultado da aplicação do operador **sizeof** independe da avaliação de seu operando; i.e., só é necessário saber o tamanho do operando.

115. Não. Apenas programas executados sob supervisão de um hospedeiro devem incluir uma função **main**().

116. (a) É um sistema que supervisiona e dá suporte à execução de um programa. (b) É um sistema que não oferece nenhum suporte à execução de um programa. (c) É um programa cuja interface é baseada apenas em texto.

117. Consulte a **Seção 1.17**.

118. (a) É o nome da primeira função a ser executada num programa com hospedeiro. (b) Porque o hospedeiro executa essa função logo após o programa ter sido carregado em memória. (c) Sim, dependendo do sistema usado como hospedeiro. Por exemplo, em sistemas da família Microsoft Windows, o nome da primeira função a ser executada é `WinMain()`.

119. Quando uma função **main()** retorna zero, o programa que a contém foi executado sem anormalidades.

120. Por meio do operador de endereço.

121. Uma variável do tipo **int**, por exemplo, sempre armazenará um valor desse tipo, quer ela tenha sido iniciada ou não. Por outro lado, o conteúdo de um ponteiro não iniciado pode não corresponder a um endereço válido.

122. Consulte a **Seção 1.18**.

123. (a) Sim. (b) Não.

124. Consulte a **Seção 1.18**.

125. A variável `p2` não é ponteiro e, portanto, não deveria ser iniciada com um endereço.

126. (a) `&x`. (b) `x`.

Capítulo 2 — Funções e Programas Multiarquivo

1. Consulte a **Seção 2.1**.

2. (a) Consulte a **Seção 2.1**. (b) Por meio de variáveis globais ou com escopo de arquivo.

3. Não.

4. (a) O uso de **void** indica que a função não retorna nenhum valor. (b) Nesse caso, o uso de **void** indica que a função não possui nenhum parâmetro.

5. O tipo do parâmetro `y` não foi declarado.

6. Consulte a **Seção 2.1**.

7. (a) Sim. (b) Não.

8. Não. Mas esses tipos devem ser compatíveis.

9. Sim. Consulte a **Seção 2.1**.

10. O ponteiro `aux` não foi iniciado.

11. Essa função retorna `x` quando `x` é positivo ou zero e retorna `-x` quando `x` é positivo. Portanto ela calcula o valor absoluto de um número inteiro.

12. Quando o valor do parâmetro é menor do que ou igual a zero, essa função não retorna nenhum valor.

13. Se você escrever um programa contendo uma chamada da função `Abs()`, tal como: `Abs(INT_MAX)`, verá que o resultado retornado por ela é 1, o que é um absurdo. Isso acontece devido à ocorrência de overflow na chamada da função **sqrt()**.

14. (a) É um parâmetro que aparece numa definição de função. (b) É um parâmetro que aparece numa chamada de função. (c) Eles são ligados durante uma chamada de função.

15. Quando ele é um parâmetro saída, parâmetro de entrada e saída ou representa um array.

16. Quando ele é um parâmetro de entrada, mas ocupa muito espaço em memória.

17. O modo de um parâmetro informa se ele é de entrada, saída ou entrada e saída.

18. Consulte a **Seção 2.1**.

19. Nunca.

20. Consulte a **Seção 2.1**.

21. Se os parâmetros forem compatíveis, haverá conversão do tipo do parâmetro real para o tipo do parâmetro formal. Se os parâmetros não forem compatíveis, poderá ocorrer erro de sintaxe.

22. Consulte a **Seção 2.1**.

23. Porque esse é o único tipo de passagem de parâmetros especificado pelo padrão de C.

24. Não.

25. Sim. Nesse caso, o valor retornado é desprezado.

26. Significa que a função retorna um valor, mas ele está sendo desprezado. Mas, o uso de conversão explícita não é estritamente necessário (apenas melhora a legibilidade).

27. Sim. A função `F2()` deve ser definida antes de `F1()`, a não ser que haja uma alusão à função `F2()` antes da definição de `F1()`.

28. Consulte a **Seção 2.1**.

29. Incluindo alusões antes das definições de funções, o programador não precisa se preocupar com a ordem com a qual elas são definidas.

30. Consulte a **Seção 2.1**.

31. O uso de parênteses vazios numa alusão é interpretado como uma alusão sem protótipo.

32. (a) Não. (b) Não. (c) Por uma razão prática: o programador pode criar uma alusão de função copiando e colando o cabeçalho da função.

33. (a) A função **scanf()** lê os caracteres `'1'`, `'2'` e `'3'`. (b) Os caracteres `'z'` e `'\n'` permanecerão armazenados no buffer e poderá causar problemas na próxima operação de leitura via teclado.

34. (a) Nada será lido. (b) O conteúdo do buffer será: `'z'`, `'1'`, `'2'`, `'3'` e `'\n'`.

35. `'\n'`.

36. (a) Quando a função **scanf()** é usada com o especificador **%c**. (b) Quando caracteres em branco devem ser saltados no início de uma operação de leitura. (c) Quando esse caractere encerra uma operação de leitura.

37. (a) Espaços em branco (incluindo `'\n'`). (b) Quando **scanf()** lê caracteres.

38. Consulte a **Seção 2.2**.

39. Para remover caracteres remanescentes após uma operação de leitura.

40. Consulte a **Seção 2.2**.

41. É uma variável que é alocada quando o bloco no qual ela é definida é executado e liberada quando encerra a execução desse bloco. (b) No interior de um bloco sem uso de **static**. (c) Escopo de bloco. (d) Seu conteúdo é indefinido.

42. **auto** serve para qualificar variáveis de duração automática. (b) Não. Ela é redundante. (c) Não, porque programas antigos que a usam deixariam de ser compilados.

43. (a) Significa que o espaço alocado para a variável ou parâmetro torna-se livre para uso posterior. (b) Quando encerra a execução do bloco no qual ela é definida. (c) Quando o programa encerra.

44. (a) O conteúdo da variável é indefinido. (b) A variável é implicitamente iniciada com zero.

45. Consulte a **Seção 2.3**.

46. (a) Porque uma variável de duração fixa deve ser iniciada antes da execução de qualquer função e uma variável de duração automática só é alocada quando a função que contém sua definição é chamada. (b) Porque isso imporia uma ordem de alocação de variáveis de duração fixa. Por exemplo, suponha que `x` e `y` sejam variáveis de duração fixa e que fosse permitido iniciar a variável `x` com o valor de `y`. Então, o compilador

teria que gerar código para alocar `y` antes de `x`. Mas, o padrão de C especifica apenas que essas variáveis devem ser alocadas ao início da execução do programa que as contém.

47. Sim. Apesar de a variável `y` ter duração fixa, sua iniciação não usa o valor da variável `x`. Em outras palavras, a expressão `sizeof(x)` pode ser resolvida em tempo de compilação.

48. (a) Não. (b) Sim.

49. (a) Consulte a **Seção 2.4**. (b) Variáveis e parâmetros. (c) Rótulos de instruções.

50. (a) O escopo de `i` é global; o escopo de `j` começa na linha em que ela é definida e termina ao final do arquivo que contém essa definição; o escopo do parâmetro `k` é todo o bloco que constitui o corpo da função `F()`; o escopo de m é o bloco que constitui o corpo da função `F()` a partir da linha que contém a definição dessa variável; o escopo de `n` é o bloco que constitui o corpo da função `F()` a partir da linha que contém a definição dessa variável. (b) O escopo de `F()` é global; o escopo de `G()` é de arquivo. (c) As variáveis `i`, `j` e `n` têm duração fixa; a duração de `m` é automática.

51. (a) Sim. (b) Não.

52. Consulte a **Seção 2.4**.

53. (a) Sim, desde que tenham escopos diferentes. (b) Sim, desde que elas sejam definidas em blocos diferentes. (c) Não.

54. Sim. O escopo deve ser de bloco.

55. Nada. O escopo pode ser de bloco, de arquivo ou de programa.

56. Porque, em princípio, qualquer instrução de um programa pode modificar uma variável global, dificultando a depuração e a manutenção do programa.

57. Consulte a **Seção 2.5.1**.

58. Consulte a **Seção 2.5.1**.

59. Consulte a **Seção 2.5.1**.

60. Consulte a **Seção 2.5.1**.

61. Consulte a **Seção 2.5.1**.

62. Consulte a **Seção 2.5.1**.

63. Consulte a **Seção 2.5.1**.

64. Consulte a **Seção 2.5.2**.

65. Consulte a **Seção 2.5.2**.

66. Consulte a **Seção 2.5.2**.

67. Consulte a **Seção 2.5.3**.

68. Consulte a **Seção 2.5.3**.

69. Ocorrerá overflow se o argumento `x` for demasiadamente grande.

70. Consulte a **Seção 2.6.1**.

71. Consulte a **Seção 2.6.1**.

72. Consulte a **Seção 2.6.3**.

73. Consulte a **Seção 2.6.3**.

74. Consulte a **Seção 2.6.3**.

75. Consulte a **Seção 2.6.4**.

76. Rigorosamente falando, compilar um programa significa exatamente compilar os arquivos-fonte que o constituem. Construir um programa executável significa, além de compilar os arquivos do programa, efetuar as devidas ligações dos arquivos-objeto resultantes da compilação.

77. Normalmente, *Make* atualiza o arquivo executável apenas quando algum arquivo-fonte tiver sido alterado, enquanto *Build* atualiza o arquivo executável em qualquer circunstância.

78. Consulte a **Seção 2.6.4**.

79.

(a)

```
gcc Entrada.c Saida.c Process.c main.c -o prog1.exe
```

(b) Utilizando as opções `-lLib1 -L/home/bibs` (Linux).

(c)

```
COMP = gcc

LINKER = gcc

OPCOES_COMP = -c -std=c99 -Wall -g

# Diretório das bibliotecas
BIBS = /home/bibs

OPCOES_LINK = -lLib1 -L$(BIBS)

OBJETOS = Entrada.o Saida.o Process.o main.o

EXEC = prog1.exe

$(EXEC): $(OBJETOS)
$(LINKER) $(OPCOES_LINK) $(OBJETOS) -o $(EXEC)

Entrada.o: Entrada.c Entrada.h
$(COMP) $(OPCOES_COMP) Entrada.c -o Entrada.o

Saida.o: Saida.c Saida.h
$(COMP) $(OPCOES_COMP) Saida.c -o Saida.o

Process.o: Process.c Process.h
$(COMP) $(OPCOES_COMP) Process.c -o Process.o

main.o: main.c Entrada.h Saida.h Process.h
$(COMP) $(OPCOES_COMP) main.c -o main.o
```

80. (a) (i) `Entrada.c` e `main.c` (ii) `Saida.c` (iii) `Process2.c`.

(b)

```
COMP = gcc

LINKER = gcc

OPCOES_COMP = -c -std=c99 -Wall -g

OPCOES_LINK = -lm

OBJETOS = Entrada.o Saida.o Process1.o Process2.o main.o

EXEC = processos

$(EXEC): $(OBJETOS)
      $(LINKER) $(OPCOES_LINK) $(OBJETOS) -o $(EXEC)
Entrada.o: Entrada.c Entrada.h
$(COMP) $(OPCOES_COMP) Entrada.c -o Entrada.o

Saida.o: Saida.c Saida.h
$(COMP) $(OPCOES_COMP) Saida.c -o Saida.o
```

CONTINUA

(b)

```
Process1.o: Process1.c Process1.h
$(COMP) $(OPCOES_COMP) Process1.c -o Process1.o

Process2.o: Process2.c Process2.h
$(COMP) $(OPCOES_COMP) Process2.c -o Process2.o

main.o: main.c Entrada.h Saida.h Process1.h Process2.h
$(COMP) $(OPCOES_COMP) main.c -o main.o
```

CONTINUAÇÃO

81. Consulte a **Seção 2.7.1**.

82. Consulte a **Seção 2.7.2**.

83. Consulte a **Seção 2.7.2**.

Capítulo 3 — Tipos de Dados Estruturados

1. Consulte a **Seção 3.1**.

2. Consulte a **Seção 3.1**.

3. Facilidade de manutenção e melhor legibilidade.

4. (a) Consulte a **Seção 3.1**. (b) Valores entre *0* e *n – 1,* sendo *n* o número de elementos do array. (c) Corrupção de memória.

5. (a) Esse programa corrompe memória, pois o índice 5 é inválido para o array `ar[]`. (b) O problema é gravíssimo, pois ocorre corrupção de memória.

6. (a) Esse programa não é portável porque a expressão `ar2[i] = ar1[i++]` não é portável. Como o operador de atribuição não possui ordem de avaliação definida, qualquer um dos seus operandos (`ar2[i]` ou `ar1[i++]`) pode ser avaliado primeiro. Em cada caso, o resultado da atribuição é diferente. (b) A melhor maneira de corrigir esse problema é substituir `i++` na referida expressão por `i` e acrescentar uma instrução na linha seguinte contendo apenas `i++`.

7. (a) Consulte a **Seção 3.1**. (b) Não. (c) Iniciando apenas o primeiro elemento com zero. (d) Não.

8. Consulte a **Seção 3.1**.

9. Não é ilegal, mas os valores excedentes não serão usados.

10. (a) Por meio da expressão `sizeof(ar)`. (b) Por meio da expressão `sizeof(ar[0])`. (c) Por meio da expressão `sizeof(ar)/sizeof(ar[0])`.

11. Porque o parâmetro é interpretado como um ponteiro.

12. Consulte a **Seção 3.2**.

13. Consulte a **Seção 3.2**.

14. Pode-se concluir que o tipo do ponteiro `p` é **char** ou um tipo derivado de mesmo tamanho.

15. Não, mas pode-se inferir que o tamanho desse tipo é 4 bytes.

16. (a) Quando `p1` e `p2` são de tipos diferentes. (b) Quando `p1` e `p2` não apontam para elementos de um array.

17. (a) Válida. (b) Válida, desde que, nesse instante, `p1` e `p2` estejam apontando para elementos de um array; caso contrário, é inválida. (c) Idem. (d) Válida. (e) Inválida. (f) Inválida.

18. (a) A expressão `*p` resulta no conteúdo da variável do tipo **int** para a qual `p` aponta. (b) A expressão `++p` faz `p` apontar para a próxima variável do **int**. (c) A expressão `++*p` incrementa o conteúdo da variável correntemente apontada por `p`. (d) A expressão `*++p` representa o conteúdo da variável do tipo **int** que segue aquela para a qual `p` aponta correntemente. (e) A expressão `*p++` representa o conteúdo da variável do tipo **int** para a qual `p` aponta; `p` passa a apontar para a variável do tipo **int** seguinte. (f) A expressão `p++*` é ilegal. (g) A expressão `p*++` é ilegal.

19. (a) `&ar[i]` e `ar + i`. (b) `ar[i]` e `*(ar + i)`.

20. (a) É o endereço do primeiro elemento. (b) É o endereço do terceiro elemento. (c) `10`. (d) `12`. (e) `30`.

21. (a) `25`. (b) `15`. (c) `-2`. (d) `3`. (e) `3`.

22.

(a)

```
for (int i = 0; i < sizeof(ar)/sizeof(ar[0]); ++i)
   printf("%d\t", ar[i]);
```

(b)

```
int *p1, *p2 = ar + sizeof(ar)/sizeof(ar[0]);

for (p1 = ar; p2 - p1 > 0; ++p1)
   printf("%d\t", *p1);
```

23. É mais eficiente começar indexação com zero (v. **Seção 3.3**).

24. (a) Consulte a **Seção 3.4**. (b) Sim, indiretamente por intermédio de um ponteiro. (c) Na qualificação de parâmetros que apontam para conteúdos que não devem ser modificados.

25. Constantes simbólicas definidas com **#define** não ocupam espaço em memória, como ocorre com variáveis qualificadas com **const**.

26. (a) `p` é um ponteiro para conteúdo constante do tipo **int**. (b) Idem. (c) `p` é ponteiro constante para um conteúdo do tipo **int**. (d) `p` é ponteiro constante para conteúdo constante do tipo **int**. (e) Idem.

27. (a) O parâmetro `x` só pode ser de entrada. (b) O parâmetro `x` pode ter qualquer modo. (c) Entrada.

28. Porque passagem de parâmetro em C se dá apenas por valor.

29. Consulte a **Seção 3.5**.

30. (a) Sim. (b) Qualificando o parâmetro com **const**.

31. Rigorosamente falando, não. Mas o retorno de uma função pode ser um endereço de array.

32. (a) Porque o espaço ocupado por ele é liberado quando a função retorna. (b) Porque uma variável zumbi pode ter seu valor alterado pelo simples fato de uma função ser chamada, o que torna difícil a percepção do erro por programadores sem muita experiência. (c) Porque o espaço ocupado por uma variável de duração fixa só é liberado quando o programa encerra.

33. (a) i. (b) ii. [Se fosse legal, a função do item (ii) retornaria um zumbi.]

34. (a) O ponteiro `ptr` não é iniciado e a função `F()` não trata o conteúdo apontado por seu parâmetro como constante. (b) Porque a probabilidade de ocorrer violação de memória aumenta, pois haverá um número maior de bytes que poderão ser acessados indevidamente.

35. Não há nenhum problema. Nesse caso, se está retornando um endereço armazenado num parâmetro, e não o endereço desse parâmetro.

36. Não. Ele pode representar um ponteiro para uma única variável.

37. Sim.

38. Consulte a **Seção 3.6**.

39. Para não causar danos à legibilidade do programa que contém tal construção.

40. Consulte a **Seção 3.6**.

41. (a) Consulte a **Seção 3.6**. (b) Não.

42. Facilita a legibilidade.

43. Consulte a **Seção 3.6**.

44. Os elementos `arBi[0][0]`, `arBi[0][1]`, `arBi[0][2]`, `arBi[1][0]`, `arBi[1][1]`, `arBi[1][2]` e `arBi[2][0]` são iniciados, respectivamente, com `1`, `2`, `3`, `4`, `5`, `6` e `7`. Os demais elementos são iniciados com `0`.

45. (a) Consulte a **Seção 3.7**. (b) Não. (c) Sim.

46. (a) É o caractere que termina qualquer string em C. (b) Zero. (c) Para terminar strings em C.

47. Para facilitar a vida do programador. (Ou você prefere a primeira notação?)

48. Sim, mas o array `ar[]` não armazenará um string.

49. Consulte a **Seção 3.7**.

50. (a) Sim. (b) `ar[0]` armazena `'b'`; `ar[1]` armazena `'o'`; `ar[2]` armazena `'l'`; `ar[3]` armazena `'a'`.

51. (a) `ar[]` é um array de ponteiros do tipo **char** *. (b) `33`. (**NB**: A questão não solicita o número de bytes alocados apenas para o array. Ela refere-se ao número total de bytes alocados. Assim deve-se levar em conta o número de bytes alocados para armazenar os strings constantes.)

52. Consulte a **Seção 3.7**.

53. (a) Porque, caso haja tentativa de alteração do string constante, o compilador indicará o erro; caso contrário, o erro se manifestará em tempo de execução e será mais difícil corrigi-lo. (b) `const char *p = "Bola"`.

54. (a) `Valor de '0': n` [sendo `n` o valor associado ao caractere `'0'` no código de caracteres usado]. (b) `Valor de ` [a sequencia de escape `\0` termina o string de formatação prematuramente]. (c) `Valor de '0': 0`. (d) `Valor de '\0':` [o caractere `'\0'` não possui representação gráfica].

55. **stdin**.

56. O ponteiro `p` aponta para uma área de memória que não deve ser alterada.

57. Primeiro erro: o ponteiro `str` deveria ser incrementado apenas uma vez no laço **while**. Correção: usar uma instrução vazia no corpo desse laço. Segundo erro: na saída do laço o ponteiro `str` não está apontando para o caractere terminal do string apontado por esse ponteiro. Correção: inserir a instrução `--str` entre os dois laços **while**. Terceiro erro: a função não retorna um ponteiro para o início do array resultante da concatenação. Correção: definir uma variável local do tipo **char** * e iniciá-la com `str`; então, a função deve retornar essa variável local (em vez de `str`).

58. O valor retornado por **strlen()** é do tipo **size_t**, que é um tipo inteiro sem sinal e, portanto, não deve ser misturado com um valor de um tipo inteiro com sinal.

59. Para permitir que o resultado da cópia possa ser usada em outra operação sobre strings sem que seja necessária uma linha de instrução adicional.

60. (a) Mais ou menos. Se os strings comparados não envolverem caracteres especiais, a deficiência dessa função pode ser contornada. (b) Para verificar se dois strings são iguais ou diferentes.

61. Consulte a **Tabela 3–2**.

62. Consulte a **Tabela 3–2**.

63. O modo mais simples é assim: `p = strchr(str, '\0')`. Mas também se pode obter idêntico resultado com **strrchr()** ou **strlen()**.

64. Assim: `strchr(str, '\0') - str`.

65. O primeiro parâmetro deve apontar para um array com tamanho suficiente para conter o resultado da concatenação.

66. (a) **isalnum()**. (b) **isalpha()**. (c) **isdigit()**. (d) **isupper()**.

67. **tolower()**.

68. (a) Definitivamente, não! (b) Porque muitas implementações de C adotaram essa bizarrice.

69. (a) Consulte a **Seção 3.8**. (b) Pesquise na internet. (c) Não. O programador pode denominá-los `joao` (sem acento) e `maria`, se preferir.

70. Usando `argv[0]` [ou substitua *argv* por qualquer outro nome com o qual você tenha batizado o segundo parâmetro de **main**()].

71. Ele deve testar se `argc` é maior do que 2 [ou substitua *argc* por qualquer outro nome que você tenha atribuído ao primeiro parâmetro de **main**()].

72. Consulte a **Seção 3.9**.

73. (a) Consulte a **Seção 3.10**. (b) Arrays são variáveis estruturadas homogêneas, enquanto estruturas são variáveis estruturadas heterogêneas.

74. Sim.

75. (a) Consulte a **Seção 3.10**. (b) Para definir tipos incompletos e estruturas com autorreferência.

76. (a) Consulte a **Seção 3.10**. (b) Permitem a implementação de estruturas de dados definidas recursivamente, como listas encadeadas e árvores.

77. (a) No local onde o campo `proximo` é declarado, o tipo `tNo` ainda não é conhecido. (b) Assim:

```
typedef struct rotNo {
           int           conteudo;
           struct rotNo *proximo;
        } tNo, *tLista;
```

78. a) Consulte a **Seção 3.10**.(b) Não. Em nenhuma definição de tipo é permitida iniciação.

79. Consulte a **Seção 3.10**.

80. Para permitir acesso aos campos de uma estrutura.

81. Consulte a **Seção 3.10**.

82. Consulte a **Seção 3.10**.

83. Quando o parâmetro é de saída ou de entrada e saída ou quando a estrutura é um parâmetro de entrada, mas ocupa relativamente muito espaço em memória.

84. Porque, tipicamente, na prática, um ponteiro ocupa menos espaço em memória do que uma estrutura.

85. Quando a estrutura é um parâmetro de entrada e o parâmetro é representado por um ponteiro.

86. Sim. Consulte a **Seção 3.10**.

87. (a) Porque, tipicamente, um ponteiro ocupa menos espaço em memória do que uma estrutura. (b) O programador deve tomar cuidado para não retornar um zumbi.

88. Consulte a **Seção 3.10**.

89. Consulte a **Seção 3.10**.

90. Consulte a **Seção 3.10**.

91. Consulte a **Seção 3.10**.

92. Consulte a **Seção 3.11**.

93. (a) O operador de indireção. (b) É a precedência mais elevada da linguagem C. (c) À esquerda.

94. (a) Porque o operador ponto tem maior precedência do que o operador de indireção. Portanto o operando esquerdo do operador ponto é `p`, que não é uma estrutura. (b) Primeira correção: `(*p).b = 2.5`; segunda correção (melhor): `p->b = 2.5`.

95. (a) O operador ponto tem maior precedência do que o operador de indireção e como, nesse caso, o resultado da aplicação do operador ponto é um endereço, pode-se aplicar o operador de indireção sobre ele. (b) Com 100% de certeza, o programa será abortado, pois, como o segundo campo da estrutura não foi

explicitamente iniciado, o valor que lhe é atribuído (implicitamente) é 0, que, nesse caso, significa endereço nulo. Assim, quando for aplicado o operador de indireção sobre esse ponteiro, o programa será abortado.

96. É uma construção da linguagem C que permite definir tipos derivados. (b) `*` (ponteiro), `[]` (array), `()` (função) **struct** (estrutura) e **union** (união).

97. Consulte a **Seção 3.12.2**.

Capítulo 4 — Recursão e Retrocesso

1. (a) Repetição. (b) Em programação, recursão refere-se a algoritmos ou subprogramas que invocam a si próprios ou a estruturas de dados definidas em termos de si mesmas.

2. Não. Existem inúmeros algoritmos que não usam nem repetição nem recursão.

3. (a) É uma função que usa iteração em vez de recursão. (b) Consulte a **Seção 4.1**.

4. Consulte a **Seção 4.1**.

5. Consulte a **Seção 4.1**.

6. Apresenta invertido na tela aquilo que for introduzido via teclado.

7. (a) `1 2 3 4 5`. (b) `5 9 13 17 21`.

8. (a) Essa função possui dois casos base, que correspondem às duas primeiras instruções **return**. (b) A terceira instrução **return**. (c) `-1`. (d) `1`.

9. Essa função é semelhante à função `SomaAteN()` apresentada na **Seção 4.1**.

10. `525`.

11. (a) Quando o valor do parâmetro é positivo, a condição de parada não é atingida. (b) A função não retorna valor quando é feita essa chamada. (c) O programa é abortado devido à ocorrência de *stack overflow*.

12. `8`. Para valores de `n` maiores do que `4`.

13. `5`.

14. É fácil mostrar por indução (v. **Apêndice B**) que o número de chamadas é `n`.

15. `14`.

16. `2`.

17. (a) `1213121`. (b) 14 vezes.

18. Consulte a **Seção 4.2**.

19. Consulte a **Seção 4.2**.

20. `15`.

21. Consulte a **Seção 4.3**.

22. Consulte a **Seção 4.3**.

23. Consulte a **Seção 4.3**.

24. O registro de ativação da função **main()**.

25. Por causa do tempo e espaço gastos com os registros de ativação associados às chamadas recursivas.

26. Esgotamento de espaço na pilha de execução.

27. (a) Significa que o espaço reservado para a pilha de execução foi exaurido. (b) A causa mais provável de *stack overflow* é um número excessivo de chamadas recursivas. (c) Sim, mas, na prática, isso é muito difícil de ocorrer. Tal função teria que ter variáveis locais de duração automática e parâmetros que consumissem todo o espaço da pilha.

28. (a) A fase de acréscimo ocorre quando chamadas recursivas de uma função acrescentam registros de ativação na pilha de execução. (b) *Stack overflow*.

29. (a) A fase de decréscimo ocorre quando os registros de ativação de chamadas recursivas de uma função são removidos da pilha de execução. (b) É impossível.

30. Consulte a **Seção 4.4**.

31. Consulte a **Seção 4.4**.

32. Sim, pois a última instrução a ser executada é uma chamada recursiva.

33. Não, porque a última instrução a ser executada não é uma chamada recursiva.

34. Consulte a **Seção 4.5.2**.

35. Consulte a **Seção 4.5**.

36. Retrocesso é implementado por meio de recursão.

37. Consulte a **Seção 4.5.2**.

38. Consulte a **Seção 4.6**.

39. Consulte a **Seção 4.6**.

40. Devido aos parâmetros adicionais necessários para a implementação da recursão. Exemplo: uma função que processa um array precisa conhecer apenas o seu tamanho, em vez dos seus limites inferior e superior.

41. É uma função que serve como interface de uma função recursiva que usa parâmetros adicionais, tornando a chamada dessa função mais natural.

42. Existem duas situações gerais: (1) problemas com soluções inerentemente recursivas (p.ex., Torres de Hanói) e (2) algoritmos que processam estruturas de dados definidas recursivamente.

43. A melhor implementação para fatorial é iterativa. As duas versões são igualmente fáceis de entender e implementar. Então vence a versão mais eficiente que é a iterativa.

44. Por razões didáticas, obviamente.

45. É mais provável que a função seja encerrada devido à ocorrência de overflow de inteiro. Numa implementação de C na qual o tipo **int** tem largura de 4 bytes, o maior número de Fibonacci que pode ser calculado é o de ordem 46. Portanto a tentativa de calcular um número de Fibonacci de ordem maior do que essa causará overflow de inteiro e o valor se torna negativo, encerrando a função antes que ocorra *stack overflow*.

46. 88 operações.

47. (a) A função `ExibeArquivoNaTelaRec()` não é justificável, pois uma função iterativa seria mais eficiente e bem mais fácil de entender. (b) A função `ExibeArquivoInvNaTelaRec()` também não se justifica. A melhor opção nesse caso é alocar espaço dinamicamente para conter todos os caracteres lidos no arquivo e depois escrevê-los na tela em ordem inversa. (Em ambos os casos, o programa pode ser abortado devido a esgotamento de pilha se o arquivo for muito grande.)

48. Consulte a **Seção 4.8.1**.

49. Consulte a **Seção 4.8.5**.

50. A função `ExponenciacaoAlt()` pode efetuar muitas operações redundantes. Por exemplo, se essa função receber como parâmetros `5` (x) e `8` (`y`), ela efetuará duas chamadas `ExponenciacaoAlt(5, 4)`. Cada uma delas efetuará duas chamadas `ExponenciacaoAlt(5, 2)`, de modo que essa última chamada será efetuada quatro vezes.

51. Os exemplos nos quais o uso de recursão é justificável na prática são:

- Problema das N rainhas (**Seção 4.5.1**)
- Problema das torres de Hanói (**Seção 4.8.1**)

- Exponenciação por quadratura (**Seção 4.8.5**)
- Invertendo entradas (**Seção 4.8.6**)
- Resolvendo Sudoku (**Seção 4.8.8**)

Capítulo 5 — Conceitos Básicos de Estruturas de Dados

1. Você comprou este livro sem saber disso? Consulte a **Seção 5.1**.
2. As mesmas operações sobre os mesmos dados podem ser mais ou menos eficientes dependendo do modo como eles são organizados. Uma vez que os dados de um programa tenham sido convenientemente organizados, a escrita dos algoritmos que irão processá-los pode ser facilitada.
3. Consulte a **Seção 5.1**.
4. Uma definição de variável informa o compilador qual é o tamanho do espaço que a variável ocupa e como os bits armazenados nesse espaço devem ser interpretados.
5. Consulte a **Seção 5.1**.
6. Consulte a **Seção 5.1**.
7. Consulte a **Seção 5.1**.
8. O nome e o tipo da variável.
9. É um programa que usa o tipo.
10. Consulte a **Seção 5.1**.
11. Consulte a **Seção 5.1**.
12. Consulte a **Seção 5.1**.
13. Consulte a **Seção 5.2**.
14. É o mesmo que simplesmente abstração. (b) É o mesmo que implementação.
15. Consulte a **Seção 5.2**.
16. Consulte a **Seção 5.2**.
17. Não.
18. Consulte a **Seção 5.2**.
19. (a) Não. (b) Sim.
20. (a) Quando a representação de dados não é exposta, essa representação e as operações que a manipulam podem ser alteradas sem afetar os clientes. (b) Para impedir que clientes tenham acesso à representação do tipo.
21. (a) Não. (b) Sim. (c) Sim. (d) Opaco.
22. Consulte a **Seção 5.3**.
23. Programa ou subprograma é um algoritmo escrito usando uma linguagem de programação.
24. (a) Ambos possuem entrada, saída e sequência de instruções. (b) Algoritmos devem ser precisos, enquanto receitas culinárias raramente são.
25. Consulte a **Seção 5.3**.
26. Consulte a introdução do **Capítulo 5**.
27. São algoritmos que resolvem um mesmo problema adequadamente.
28. Entendimento de linguagem natural e aprendizagem.
29. Consulte a **Seção 5.3**.
30. Consulte a **Seção 5.3**.

31. Linguagem natural pura raramente é usada na escrita de algoritmos, pois ela apresenta problemas inerentes, tais como prolixidade, imprecisão, ambiguidade e dependência de contexto. A escrita de algoritmos numa linguagem de programação impõe sérias dificuldades para aqueles que ainda não adquiriram prática na construção de algoritmos nem conhecem bem a linguagem de programação utilizada. Uma ideia que facilita a vida do programador consiste em usar, na construção de algoritmos, uma linguagem algorítmica próxima à linguagem natural do programador que incorpore construções semelhantes àquelas encontradas comumente em linguagens de programação.
32. (a) É um algoritmo ou trecho de um algoritmo escrito em pseudolinguagem. (b) Não.
33. Consulte a **Seção 5.3.3**.
34. Consulte a **Seção 5.3.3**.
35. Consulte a **Seção 5.3.3**.
36. (a) São casos de entrada que produzem resultados que têm significados distintos. (b) Eles testam todas as saídas possíveis de um programa.
37. Exemplos de execução podem ser usados como casos de entrada durante a fase de testes do algoritmo e do programa resultante.
38. Porque podem ser introduzidos erros durante a codificação do programa ou mesmo durante sua tradução pelo compilador.
39. Consulte a **Seção 5.3.4**.
40. Consulte a **Seção 5.3.4**.
41. Consulte a **Seção 5.4**.
42. Arquivo de interface (ou cabeçalho) e arquivo de implementação (ou de programa).
43. É um ponteiro que aponta para uma estrutura parcialmente definida.
44. Consulte a **Seção 5.4**.
45. (a) Escoamento de memória.(b) Porque não há garantia que o cliente irá usá-la devidamente.
46. Contagem de referência consiste em checar, em intervalos regulares de tempo, se um dado espaço alocado dinamicamente possui algum ponteiro apontando para ele e, se esse não for o caso, liberar esse espaço.
47. Se a representação do tipo `tComplexo` for alterada, essa instrução poderá deixar de funcionar.
48. Por duas razões: (1) o tipo `tComplexo2` é um tipo de ponteiro para estrutura (e não um tipo de estrutura) e (2) o campo `pReal` não é exposto no arquivo de interface e, portanto, o compilador o desconhece no local do programa em que a instrução se encontra.

Capítulo 6 — Análise de Algoritmos

1. Consulte a introdução do **Capítulo 6**.
2. A eficiência (ou desempenho) de um programa é medida em termos do espaço de armazenamento e pelo tempo que ele utiliza para realizar uma tarefa.
3. Não.
4. As funções `Fatorial()` e `Fatorial2()` discutidas na **Seção 4.7** são funcionalmente equivalentes e ambas têm custo temporal $\theta(n)$.
5. Consulte a **Seção 6.1**.
6. É o mesmo que complexidade de algoritmo.
7. (a) É a complexidade de um algoritmo medida em termos de tempo de processamento. (b) É a complexidade medida em termos de espaço adicional ocupado pelo dados que um algoritmo processa.

8. Esse tipo de análise é baseada no número de operações elementares que o algoritmo executa.

9. Porque a maioria dos algoritmos que manipulam as estruturas de dados mais comuns não usam espaço adicional.

10. Espera-se que a função `F()` seja executada 7.500 vezes e que a função `G()` seja executada 625 vezes.

11. Consulte a **Seção 6.2**.

12. Ela permite avaliar a eficiência de um algoritmo de modo independente de hardware e software e sem que o algoritmo sequer precise ser implementado.

13. Deseja-se determinar um valor de n que satisfaça a seguinte desigualdade:

$10\, n \log_2 n < 2n^2$

Note que, neste caso, a base do logaritmo é importante e, assim, assume-se que seja 2.

Considerando-se $n > 0$, essa desigualdade pode ser simplificando e rearranjada de modo a resultar em:

$n > 5 \log_2 n$

Essa desigualdade não possui solução analítica. Mas, para que serve computador e programador? Construa um programa e verifique que essa desigualdade é válida para $n \geq 23$.

14. $n^{3/2}$ é o mesmo que $n.n^{1/2}$. Logo comparar $n \log n$ com $n^{3/2}$ é o mesmo que comparar $\log n$ com $n^{1/2}$. Agora, com um pouco de conhecimento sobre logaritmos (v. **Apêndice B**), é fácil provar que $\log n$ é $O(n^{1/2})$, como se mostra abaixo:

Para todo $n > 0$, tem-se:

$$\log n < n \Rightarrow 2 \log \sqrt{n} < 2\sqrt{n} \Rightarrow \log n < 2\sqrt{n}$$

Portanto, para mostrar que $\log n$ é $O(n^{1/2})$, basta considerar $c_0 = 2$ e $n_0 = 1$.

A conclusão que se tira desse arrazoado é que os dois algoritmos crescem assintoticamente à mesma taxa, de modo nenhum deles usa um número menor de operações do que o outro quando n se torna suficientemente grande.

15. Consulte a **Seção 6.3**.

16. Resumindo, a notação ó define uma função que é limite superior de outra, mas nada informa com respeito à proximidade entre as duas funções. A notação ômega refere-se ao limite inferior de uma função e também não é muito informativa. A notação teta refere-se ao limite estrito de uma função e, portanto, não sofre com a frouxidão que aflige as notações ó e ômega.

17. V. resposta da questão anterior.

18. A afirmação não é correta. Primeiro, o termo *melhor* é vago em análise de algoritmos utilizando análise assintótica. Em segundo lugar, a notação ó grande é imprecisa. Quer dizer, pode ser que o algoritmo B seja $\Omega(1)$.

19. Depende. Análise assintótica só garante que A é melhor do que B para valores de n suficientemente grandes. É possível que B seja melhor do que A para valores pequenos de n.

20. (a) $f(n)$ cresce com uma taxa inferior a n. (b) $f(n)$ cresce com uma taxa superior a n. (c) $f(n)$ cresce com a mesma taxa que n.

21. Considere as seguintes funções:

$$f(n) = \begin{cases} n & \text{se } n \text{ for par} \\ 1 & \text{se } n \text{ for ímpar} \end{cases} \qquad g(n) = \begin{cases} n & \text{se } n \text{ for ímpar} \\ 1 & \text{se } n \text{ for par} \end{cases}$$

Verifique que nem $f(n)$ é $O(g(n))$ nem $g(n)$ é $O(f(n))$.

22. $\theta(n)$.

23. Como $T(n)$ é $O(f(n))$, então existem constantes $c_0 > 0$ e $n_0 \geq 0$ tais que $T(n) \leq c_0 f(n)$ para todo $n \geq n_0$. Mas, como $c > 0$, tem-se que $cT(n) \leq c.c_0 f(n)$ para todo $n \geq n_0$. Logo $cT(n)$ é $O(f(n))$.

24. Por hipótese, tem-se que $T_1(n) \leq c_1 f(n)$, para todo $n \geq n_1$, e $T_2(n) \leq c_2 g(n)$, para todo $n \geq n_2$.

Sejam $c_0 = max(c_1, c_2)$ e $n_0 = max(n_1, n_2)$. Então, tem-se:

$T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n) \leq c_0 f(n) + c_0 g(n) = c_0(f(n) + g(n))$, para todo $n \geq n_0$. Logo $T_1(n) + T_2(n)$ é $O(f(n) + g(n))$.

(b) Não contradiz o **Teorema 6.4**, mas esse teorema é mais utilizado por ser mais útil.

25. Sejam $T_1(n) = n$, $T_2(n) = 1$, $f(n) = n$ e $g(n) = n$. Então, $T_1(n)$ é $O(f(n))$ e $T_2(n)$ é $O(g(n))$. Agora, $T_1(n) - T_2(n) = n - 1$, que não é $O(f(n) - g(n)) = O(1)$.

26. Para todo $n > 1$, $1 \leq \log n \Rightarrow n \leq n \log n$. Portanto, considerando-se $c_0 = 1$ e $n_0 = 1$, tem-se que n é $O(n \log n)$.

27. Para todo $n \geq 1$, tem-se que:

$n^2 \leq n^2 + n = (n + 1)n \Rightarrow (n^2 + 1)/(n + 1) \leq n = 1n$. Logo, considerando-se, $c_0 = 1$ e $n_0 = 1$, a proposição está provada.

28. É fácil mostrar que n^2 é $O(n^3)$. Para aprender a mostrar que n^3 não é $O(n^2)$, estude o **Exemplo 6.3**.

29. Por hipótese, para todo $n \geq n_0 \geq 0$, tem-se que:

$T(n) \leq c_0 n \leq c_0 n^2$. Logo $T(n)$ é $O(n^2)$.

30. $\theta(n)$.

31. No **Apêndice B**, mostra-se que:

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

Rearranjando-se o lado direito dessa expressão, obtém-se um polinômio de grau 2. Portanto, de acordo com o **Teorema 6.2**, esse somatório é $\theta(n^2)$.

32. (a) O custo temporal de T_1 é $\theta(n)$ e o custo temporal de T_2 é $\theta(n^2)$. (b) O algoritmo com tempo T_1 tem melhor custo temporal. (c) Quando $n < 250$, é melhor usar o algoritmo com tempo T_2.

33. Essa é uma decorrência direta do **Teorema 6.3**, com $c_1 = c$ e $c_1 = 0$. A prova desse teorema encontra-se no **Apêndice B**.

34. É trivial. Se você ainda não sabe demonstrar isso, releia todo o **Capítulo 6**.

35. (a) Para todo $n > 0$, tem-se:

$$2^n = \overbrace{2.2. \ldots .2}^{n \text{ fatores}} \leq \overbrace{1.2. \ldots .(n-1).n}^{n \text{ fatores}} = n!$$

Do mesmo modo que só se aprende a programar programando, só se aprende a provar provando! Portanto conclua a prova. (b) Estude o **Exemplo 6.3**.

36. Para todo n e para todo c, tem-se: $2^{n+c} = 2^c 2^n$. Portanto, de acordo com a afirmação da questão **33**, 2^{n+c} é $\theta(2^n)$.

37. Para todo $n > 0$, tem-se:

$n! = 1.2. \ldots . n \leq n.n. \ldots . n = n^n \Rightarrow log(n!) \leq log\, n^n = n \log n$. Portanto ... (complete os três pontinhos).

38. ($\Rightarrow$) Suponha que $f(n)$ é $O(g(n))$, então... (complete os três pontinhos).

($\Leftarrow$) Suponha que $g(n)$ é $\Omega(f(n))$, então... (complete os três pontinhos).

39. Para todo $n > 1$, $\log n \geq 1 \Rightarrow n \log n \geq n$. Portanto, considerando-se $c_0 = 1$ e $n_0 = 1$, tem-se que $n \log n$ é $\Omega(n)$.

40. Para todo $n > 1$, $n^2 = n.n > n \log n$. Logo, considerando $c_0 = 1$ e $n_0 = 1$, tem-se que n^2 é $\Omega(n \log n)$.

41. Use a **Definição 6.1**.

42. Consequência direta do **Teorema 6.2**.

43. Consulte o **Exemplo 6.3**.

44. $2 + 4 + 6 + \ldots + 2n$ constitui uma série aritmética cujo resultado é .... Portanto ... (complete os três pontinhos).

45. $1^2 + 2^2 + \ldots + n^2$ constitui uma série geométrica cujo resultado é .... Portanto ... (complete os três pontinhos).

46. Consulte a **Seção 6.4**.

47. O pior caso de entrada para um algoritmo corresponde ao maior número de operações necessárias para resolver o problema em questão e é normalmente fácil de ser identificado. Além disso, se um algoritmo possui boa avaliação no pior caso, certamente será bem avaliado nos casos melhor e mediano. Mais ainda, de acordo com a Lei de Murphy, sempre se deve esperar pelo pior.

48. Não. O pior e o melhor caso podem ser equivalentes.

49. Sim.

50. (a) $\theta(n)$. (b) $\theta(n)$. (c) $\theta(n)$.

51. Significa que $T(n)$ independe do tamanho da entrada n.

52. Consulte a **Tabela 6–2**.

53. (a) Acesso a um elemento de um array. (b) Busca binária. (c) Busca sequencial. (d) Consulte a **Seção 6.11.5**.(e) Soma de matrizes. (f) Consulte a **Seção 6.11.6**.

54. (a) $\sqrt{n} \leq n \leq n \log n \leq n \log n^2 \leq n \log^2 n \leq n^{1.5} \leq n^2 \leq n^3 \leq 2^n$. (b) As funções $n \log n$ e $n \log n^2$ possuem a mesma taxa de crescimento (se você não entendeu, consulte as propriedades de logaritmos no **Apêndice B**).

55. Um computador pessoal ainda é incapaz de realizar esses cálculos. Portanto deve-se usar conhecimentos sobre propriedades de exponenciação para reduzir a complexidade dos cálculos.

56. (a) $2{,}5\,\mu s$. (b) $12{,}5\,\mu s$. (c) $62{,}5\,\mu s$.

57. (a) 10^9. (b) $10^{300.000.000}$. (c) $39.620.077$. (d) 31.622. (e) . (f) 29.

58. (a) Dobra. (b) É acrescida de um valor constante. (c) Quadruplica.

59. Porque à medida que o tamanho da entrada cresce elas tornam-se irrelevantes.

60. (a) A regra da soma é importante na avaliação de blocos de instruções. (b) A regra do produto é importante na avaliação de laços de repetição.

61. (a) Em análise assintótica, busca-se encontrar uma função mais simples do que $2n^3 + n - 1$ como limite de crescimento, que obviamente existe. (b) De acordo com a notação ó, o correto é afirmar que esse algoritmo tem custo temporal $O(n^3)$.

62. Porque à medida que o tamanho da entrada cresce o valor de c se torna irrelevante.

63. Estude a **Tabela 6–4** e faça você mesmo.

64. Consulte a **Seção 6.7.2**.

65. Consulte a **Seção 6.7.2**.

66. Consulte a **Seção 6.7.2**.

67. Consulte a **Seção 6.7.3**.

68. (i) $\theta(n)$. (ii) $\theta(n^2)$. (iii) $\theta(n^3)$. (iv) $\theta(n^2)$. (v) A variável j pode crescer até o valor de i^2, que pode ser igual a n^2. Por outro lado, k pode crescer tanto quanto j, que é igual a n^2. Portanto o tempo de execução é proporcional a $n.n^2.n^2$, que é $\theta(n^5)$. (vi) $\theta(n^4)$. (vii) O tamanho do problema é reduzido à metade a cada passagem pelo corpo do laço. Portanto esse é um caso clássico de custo temporal $\theta(\log n)$. (viii) Embora possa não parecer,

esse caso é semelhante ao anterior e seu custo é $\theta(\log n)$. (ix) Esse laço encerra quando $i^2 > n$. Portanto seu custo temporal é $\theta(n^{1/2})$. (x) O laço interno tem custo temporal $\theta(\log n)$ e o laço externo considerado isoladamente tem avalia $\theta(n)$. Portanto, em conjunto, o custo temporal é $\theta(n \log n)$.

69. $\theta(1)$.

70. $\theta(n)$.

71. Se n é o tamanho do string `s1` e m é o tamanho do string `s2`, então o custo temporal do primeiro laço **while** é $\theta(n)$ e o custo temporal do segundo **while** é $\theta(m)$. Portanto, de acordo com a regra da soma, o custo temporal dessa função é $\theta(max(n, m))$.

72. $\theta(n^2)$.

73. $\theta(n^3)$.

74. $\theta(n^2)$.

75. Em ambos os casos, o custo temporal é $\theta(n^2)$.

76. $\theta(n^2)$.

77. Cada laço **while** considerado isoladamente possui custo temporal $\theta(\log n)$. Portanto o custo temporal da função é $\theta(\log^2 n)$.

78. $\theta(n)$.

79. (a) O primeiro algoritmo tem custo $\theta(n^3)$ e o segundo tem custo $\theta(n)$. (b) O segundo algoritmo é mais eficiente em termos de notação teta. (c) Quando $n \leq 9$. A resposta pode ser obtida por meio da execução do seguinte programa:

```
#include <stdio.h>

int main()
{
   int n = 1;

   while (n*n*n < 2*n + 500)
      ++n;

   printf("\nResposta: n <= %d\n", n);

   return 0;
}
```

80. $\theta(1)$.

81. Consulte a **Seção 6.8**.

82. A análise é baseada no número de chamadas recursivas que ele executa.

83. V. **Exemplo 6.26**.

84. Quando a função `Fatorial()` é chamada com um valor `n > 1`, ela mantém `n` registros de ativação alocados ao mesmo tempo na pilha de ativação. Portanto o custo espacial dessa função é $\theta(n)$.

85. Consulte o **Apêndice B**.

86. Tipicamente, um algoritmo recursivo está associado a uma relação de recorrência.

87. 4.

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

89. Consulte a **Seção 6.10**.

90. Consulte a **Seção 6.10**.

91. A árvore de recursão solicitada é aquela da **Figura D–1**. Essa questão é similar ao primeiro exemplo apresentado na **Seção 6.10** e o custo é o mesmo [i.e., $T(n)$ é $\theta(n \log n)$].

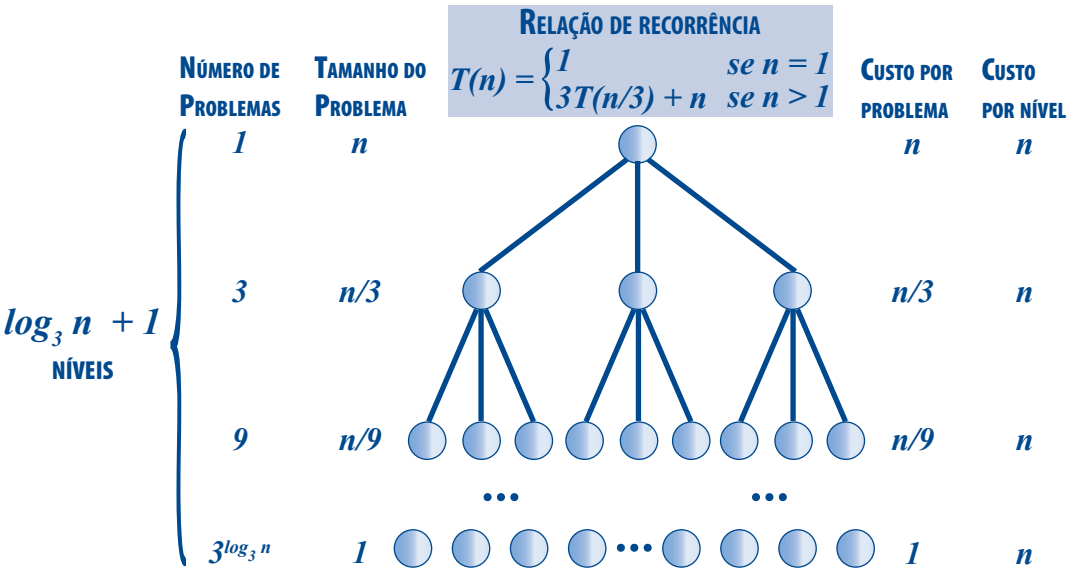

FIGURA D–1: QUESTÃO 91 — CAPÍTULO 6

92. (a) Para evitar o uso das funções piso e teto (v. **Apêndice B**), o que complicaria a análise. Ou seja, se não for feita essa suposição, o tamanho de cada problema num dado nível poderia não ser o mesmo. Por exemplo, se $n = 5$ no nível inicial, no próximo nível, um problema teria tamanho igual a 2 (i.e., $\lfloor 5/2 \rfloor$) e o outro teria tamanho igual a 3 (i.e., $\lceil 5/2 \rceil$). (b) O raciocínio é o mesmo do item (a).

93. (a) V. **Figura D–2**. (b) V. **Figura D–3**.

94. (a) $\theta(n^{\log 3})$. (b) $\theta(n \log n)$.

95. Consiste em atribuir valores à variável n da relação de recorrência para obter alguma conjectura sobre o resultado e depois tentar provar essa conjectura.

96. O argumento é que nem todos os registros de ativação necessários para processar uma chamada são alocados simultaneamente na pilha de execução.

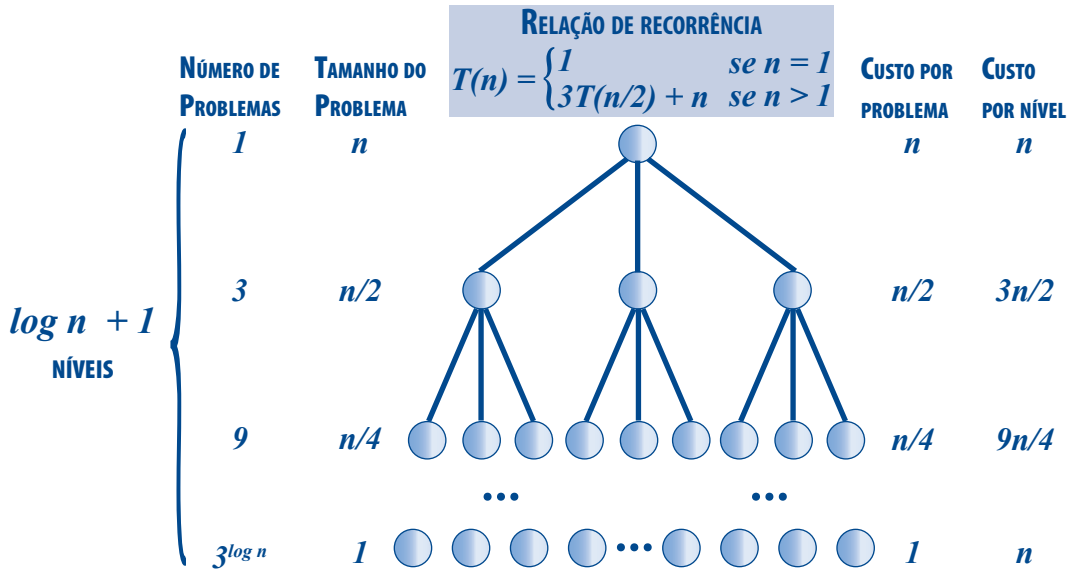

FIGURA D–2: QUESTÃO 93 (A) — CAPÍTULO 6

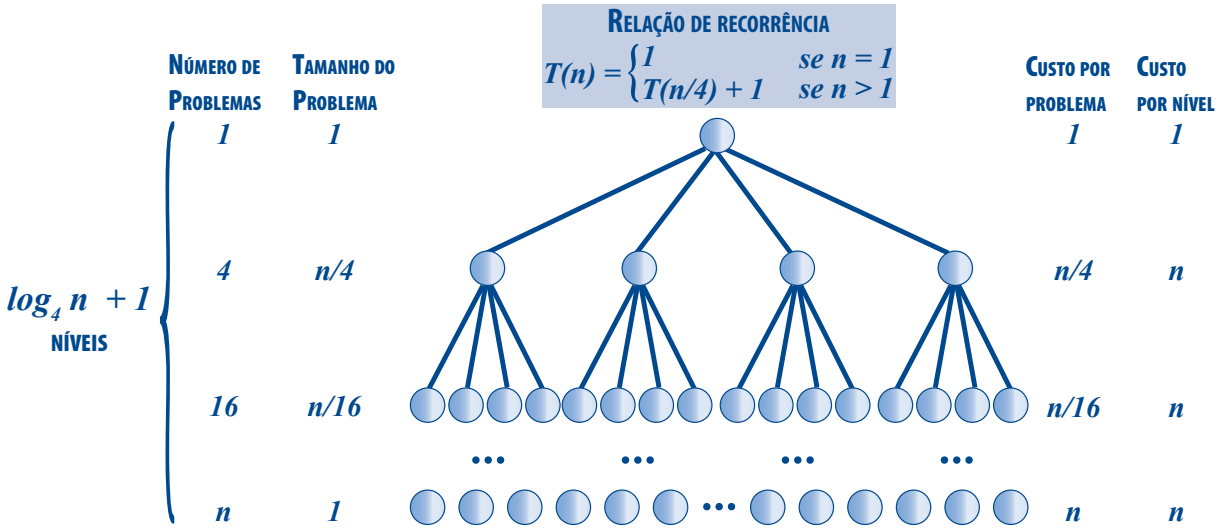

Figura D–3: Questão 93 (b) — Capítulo 6

Capítulo 7 — Listas Indexadas

1. É um array cujo tamanho é conhecido em tempo de programação.
2. (a) Consulta e alteração. (b) $\theta(1)$.
3. Consulte a **Tabela 7–1**.
4. Índices de arrays são estáticos e índices dos elementos de uma lista variam de acordo com as operações de inserção e remoção efetuadas na lista.
5. É a obtenção do valor de um elemento por meio de seu índice.
6. Consulte a **Seção 7.1.1**.
7. Consulte a **Seção 7.1.1**.
8. Consulte a **Seção 7.1.1** .
9. (a) Consulte a **Seção 7.1.1**. (b) Idem. (c) Exibição de cada elemento na tela.
10. Porque elas podem ser implementadas por meio de operações essenciais.
11. (a) Calculando o comprimento da lista e, então, acessando seus elementos sequencialmente até o final da lista ou até que o elemento procurado seja encontrado. (b) Calculando o comprimento da lista e, então, inserindo o elemento na posição igual a esse comprimento. (c) Calculando o comprimento da lista e verificando se ele é igual a zero. (d) Removendo o elemento e acrescentando-o modificado na mesma posição em que se encontrava antes da remoção.
12. Essa operação pode ser dividida em duas partes: (1) a identificação dos elementos negativos e (2) a remoção propriamente dita. A primeira parte dessa operação tem custo temporal $\theta(n)$ em todos os casos, visto que todos os elementos da lista devem ser analisados. A segunda parte dessa operação tem custo temporal $\theta(1)$ no melhor caso e $\theta(n)$ no pior caso. Combinadas, as duas partes da operação têm custo temporal $\theta(n)$ em todos os casos. (a) O pior caso ocorre quando os primeiros elementos da lista são negativos e o custo temporal é, como foi visto, $\theta(n)$. (b) O melhor caso ocorre quando o único elemento negativo é o último elemento da lista e o custo temporal é, novamente, $\theta(n)$.
13. Para ocorrer inserção numa determinada posição, é necessário que essa posição esteja desocupado, pois, em caso contrário, o elemento que se encontra naquela posição será removido da lista.

14. (a) Caso não ocorresse a movimentação descrita na **Seção 7.1.1**, o elemento que era antecessor do elemento removido deixaria de ter sucessor e o elemento que era sucessor do elemento removido deixaria de ter antecessor. (b) Quando se remove o último elemento da lista.

15. Porque acréscimo de elemento é efetuado ao final da lista.

16. Porque essa operação pode detonar a ordenação da lista.

17. Porque ela pode retornar um valor indicando quando é mal sucedida.

18. Porque não lhe resta nenhum valor que possa ser retornado para indicar que ela foi mal sucedida.

19. Em vez de retorna o valor do elemento, essa nova versão da função `ObtemElementoListaIdx()` deve retorna um valor que indica quando é bem sucedida ou não. Então, quando encontrado, o valor do elemento é armazenado num novo parâmetro de saída, como mostrado a seguir.

```
int ObtemElementoListaIdx(const tLista *lista, int indice, tElemento *elemento)
{
   if (indice < 0 || indice >= lista->nElementos)
      return 1;

   *elemento = lista->elementos[indice];

   return 0;
}
```

20. Os puristas acham que ela não deve fazer parte da interface de um tipo que represente lista indexada, visto que essa função não faz parte da abstração. Os pragmáticos, por sua vez, acham que é essencial tê-la como parte dessa interface, pois ela pode ser necessária. (O autor prefere a visão pragmática.)

21. Consulte a **Seção 7.1.3**.

22. Consulte a **Seção 7.1.3**.

23. Quando dois elementos possuem a mesma chave, um deles pode nunca ser encontrado numa busca simples.

24. Busca sequencial pode ser sempre utilizada quer a lista esteja ordenada ou não, mas, se a lista estiver ordenada, pode-se usar busca binária, que é mais eficiente.

25. Ao final, porque não requer movimentação de elementos.

26. Numa lista não ordenada o elemento pode ser inserido em qualquer posição. Portanto ele é acrescentado ao final da lista. A inserção numa lista ordenada requer que, antes que ela ocorra, a posição de inserção seja encontrada.

27. (a) Quando o elemento procurado não se encontra na lista, uma busca sequencial pode ser encerrada quando se encontra uma chave maior do que a chave de busca. (b) Não, o custo temporal continua sendo $\theta(n)$.

28. Remoção de elemento não destrói ordenação.

29. Consulte a **Seção 7.2.1**.

30. Acréscimo de um elemento ao final da lista e alteração de valor de um elemento.

31. Quando o elemento procurado encontra-se próximo ao início da lista.

32. Consulte a **Seção 7.2.2** e leia os comentários no corpo dessa função.

33. Seguindo o seguinte procedimento: (1) Encontre o elemento a ser alterado; (2) remova-o da lista; (3) altere seu conteúdo e (4) insira-o mantendo a ordenação da lista.

34. (a) Consulte a **Seção 7.1.3**. (b) Consulte a **Seção 7.2.3**. (c) Sim. (d) Não.

35. Consulte a **Seção 7.2.3**.

36. (a) O primeiro elemento. (b) Depende da posição de cada elemento na lista e não da posição relativa entre eles. Por exemplo, se um desses elementos se encontra no meio da lista, ele será encontra primeiro, não importando se ele vem antes ou depois do outro elemento com a mesma chave.

37. Consulte as referidas figuras e mãos à obra.

38. Nesse caso, uma busca sequencial é mais eficiente, pois usando-a, no máximo, três elementos serão visitados. Por outro lado, uma busca binária poderá visitar até seis elementos.

39. Consulte a **Seção 6.11.5** (**Capítulo 6**).

40. Porque essas operações não utilizam espaço adicional que dependa do tamanho da lista processada.

41. (a) O melhor caso de busca sequencial é quando o elemento procurado é o primeiro da lista e seu custo temporal é $\theta(1)$. (b) O pior caso de busca sequencial ocorre quando o elemento procurado é o último da lista ou quando não se encontra na lista e seu custo temporal é $\theta(n)$.

42. (a) O melhor caso de busca binária ocorre quando o elemento se encontra no meio da lista e seu custo temporal é $\theta(1)$. (b) O pior caso de busca binária é quando o elemento procurado se encontra em uma das metades da lista ou não se encontra na lista e seu custo temporal é $\theta(\log n)$.

43. Consulte a **Seção 7.4**.

44. Consulte a **Seção 7.4** e, se não entender, consulte também a **Seção 2.5.1**.

45. É uma condição que deve ser satisfeita para que a função seja bem sucedida em sua missão.

46. Consulte a **Seção 7.4**.

47. Na linha i, têm-se i elementos não nulos. Portanto o número total de elementos não nulos é dado por:

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

■

48. Consulte a **Seção 7.5.1**.

49. Consulte a **Seção 7.5.2**.

50. Consulte a **Seção 7.5.2**.

51. Para economizar memória.

52. Consulte o **Preâmbulo** da **Seção 7.6.1**.

53. A função `ApresentaMenu()` apresenta um menu sensível ao contexto de acordo com o estado corrente da lista. Quer dizer, é essa lista que determina o contexto. (b) Sabendo que a lista está cheia a função não permite que estejam disponíveis para o usuário a opção de inserção ou acréscimo de elemento.

54. Consulte a **Seção 7.6.2**.

Capítulo 8 — Pilhas e Filas

1. (a) Dicionário é uma estrutura de dados que permite acesso a seus componentes de acordo com seus conteúdos. (b) Contêiner é uma estrutura de dados que permite armazenamento e recuperação de dados independentemente de conteúdo.

2. Consulte a **Seção 8.1.1**.

3. Consulte a **Seção 8.1.1**.

4. Porque uma pilha permite acesso a seus elementos independentemente de seus conteúdos.

5. Consulte a **Seção 8.1.1**.

6. Pilhas não permitem operação de busca nem inserção em posição arbitrária.

7. Criação e checagem de pilha vazia (lista vazia).

8. (a) Não existe operação de busca em pilhas e não existe inserção de elemento numa determinada posição numa pilha. (b) Empilhamento e desempilhamento.

9. Consulte a **Seção 4.3** (**Capítulo 4**).

10. A única incorreta é (c). (Verifique que as demais estão corretas.)

11. Consulte a **Seção 8.1.3**.

12. Para exibir todo o conteúdo de uma pilha mantendo-a intacta, é necessário utilizar uma pilha auxiliar. O seguinte trecho de código resolve o problema (lembre-se que se supõe que a pilha `p` já existe):

```
tPilhaCar pilhaAux;

while (!PilhaVazia(p)) {
   putchar(ElementoTopo(p));
   Empilha(Desempilha(&p), &pAux);
}

while (!PilhaVazia(pAux))
   Empilha(Desempilha(&pAux), &p);
```

13.

(a)

```
(void) DesempilhaIdx(&p);
item = DesempilhaIdx(&p);
```

(b) É necessário remover o primeiro elemento e guardá-lo para em seguida restaurar a pilha ao seu conteúdo original:

```
tItemPilha aux;

aux = DesempilhaIdx(&p);
item = ElementoTopoIdx(p);
EmpilhaIdx(aux, &p);
```

(c) Primeiro, usa-se um laço de repetição **for** para desempilhar os `n - 1` elementos de cima do elemento desejado. Então, desempilha-se esse elemento atribuindo-o a `item`:

```
for (int i = 1; i <= n - 1; i++)
   (void) DesempilhaIdx(&p);

item = DesempilhaIdx(&p);
```

(d) Este problema é semelhante ao do item anterior, mas agora é necessária uma pilha auxiliar para guardar os elementos desempilhados e, após realizar a atribuição desejada, restaurar a pilha ao seu estado original.

```
tPilhaIdx pAux;
int       i;

CriaPilhaIdx(&pAux);

for (i = 1; i <= n - 1; i++)
   EmpilhaIdx(DesempilhaIdx(&p), &pAux);

item = ElementoTopoIdx(p);

for (i = 1; i <= n - 1; i++)
   EmpilhaIdx(DesempilhaIdx(&pAux), &p);
```

(e) Este problema é semelhante ao item (c), mas tem uma sutileza, pois não se sabe quantos elementos existem na pilha. Portanto a estratégia é desempilhar todos os elementos da pilha, mas sempre guardando o último elemento desempilhado na variável `item`, pois não se sabe de antemão se o elemento desempilhado é o desejado (i.e., o último).

```
while (!PilhaIdxVazia(p))
   item = DesempilhaIdx(&p);
```

(f) Utiliza-se uma pilha auxiliar para guardar todos os elementos desempilhados da pilha original. Ao final dessa operação, o último elemento da pilha original estará no topo da pilha auxiliar. Então atribui-se esse elemento à variável `item` e restaura-se a pilha original.

```
tPilhaIdx pAux;

CriaPilhaIdx(&pAux);

while (!PilhaVazia(p))
   EmpilhaIdx(DesempilhaIdx(&p), &pAux);

item = ElementoTopoIdx(pAux);

while (!PilhaIdxVazia(pAux))
   EmpilhaIdx(DesempilhaIdx(&pAux), &p);
```

14. 8.

15. 9, 1 e 7.

16. (a), (c), (d), (f) e (h)

17. Consulte a **Seção 8.2.1**.

18. Não existe operação de busca em filas e não existe inserção de elemento numa determinada posição numa fila.

19. Consulte a **Seção 8.2.1**.

20. Criação e checagem de fila vazia (lista vazia).

21. Consulte a **Seção 8.2.1**.

22. Consulte a **Seção 8.2.1**.

23. Porque uma fila permite acesso a seus elementos independentemente de seus conteúdos.

24. Consulte a **Seção 8.2.3**.

25. 5, 6, 2 e 3.

26. (a) 2 (usando a convenção da **Seção 8.2.2**). (b) 11.

27. (b), (e), (g) e (i).

28. (a) Consulte a **Seção 8.3**. (b) Consulte a **Seção 8.3**. (c) Filas circulares são mais eficientes do que filas lineares.

29. Implementação. Os conceitos de fila linear e fila circular são os mesmos.

30. Para levar em consideração que um dos elementos do array no qual a fila é implementada não é utilizado.

31. (a) 3. (b) 8.

32. As operações de enfileiramento, desenfileiramento e verificação de fila cheia.

33. Consulte a **Seção 8.3**.

34. Consulte a **Seção 8.2.3** e a **Seção 8.3**.

35. Consulte a **Seção 8.3**.

36. Sim. Consulte a **Seção 8.4**.

37. Um meta-algoritmo é um algoritmo que guia a construção de outros algoritmos.

38. Consulte a **Seção 8.4**.

39. Pilha. Consulte a **Seção 8.4**.

40. Em linguagem cotidiana, um palíndromo não leva em consideração acentuação, pontuação, hifens ou espaços entre palavras. Em programação, um palíndromo é um string que representa a mesma sequência de caracteres quando examinado tanto do início para o final quanto do final para o início.

41. As avaliações de complexidade temporal e espacial das duas funções é a mesma. Em termos de legibilidade, a versão recursiva é a favorita porque não requer conhecimento de pilha para ser implementada.

42. Consulte a **Seção 8.5.4**.

43. Consulte a **Seção 8.5.4**.

44. Consulte a **Seção 8.5.4**.

45. Porque quando se executa uma operação, não há o que desfazer. É assim que funciona a maioria dos aplicativos.

46. (a) O conteúdo do texto que foi recortado e a posição em que ele se encontrava. (b) O texto que foi colado, o conteúdo do texto que foi substituído (se for o caso) e a posição no texto em que a operação ocorreu. (c) A palavra removida e a posição em que ela se encontrava. (d) O texto substituído, aquele que o substituiu e a posição no texto em que ocorreu a substituição.

47. (a) Siga o seguinte algoritmo:

> **ENTRADA:** Uma fila (*f*) do tipo `tFilaIdx`
> **RETORNO:** O número de elementos da fila
> **1.** Inicie o contador de elementos *cont* com *0*
> **1.** Crie uma fila auxiliar *aux* do tipo `tFilaIdx`
> **2.** Enquanto *f* não estiver vazia, faça
> **2.1** Retire um elemento de *f*
> **2.2** Acrescente o elemento retirado de *f* em *aux*
> **2.3** Incremente *cont*
> **3.** Enquanto *aux* não estiver vazia, faça
> **3.1** Retire um elemento de *aux*
> **3.2** Acrescente o elemento retirado de *aux* em *f*
> **4.** Retorne *cont*

(b) O algoritmo a ser seguido é o mesmo.

Capítulo 9 — Alocação Dinâmica de Memória

1. (1) Leitura de linhas de um arquivo de texto cujas linhas tenham tamanhos variados; (2) alocação de espaço para armazenamento de um arquivo cujo tamanho não é conhecido em tempo de compilação; (3) cópia de string de tamanho arbitrário.

2. Consulte a **Seção 9.1**.

3. Consulte a **Seção 9.1**.

4. (a) Consulte a **Seção 9.1**. (b) Ele pode encerrar prematuramente por falta de memória disponível para sua execução.

5. (a) Consulte a **Seção 9.1**. (b) Desperdício de memória.

6. (a) Todas as variáveis simbólicas são estáticas. (b) A única variável dinâmica é a variável anônima alocada com **malloc()**.

7. (a) Sim. (b) Também.

8. A indireção de um ponteiro iniciado com **NULL** causa aborto de programa, que é um erro de execução. A indireção de um ponteiro não iniciado pode causar um erro lógico. Erros lógicos são bem mais difíceis de corrigir do que erro de execução.

9. Consulte a **Seção 9.2**.

10. (a) Sim. (b) Não.

11. Consulte a **Seção 9.2**.

12. Consulte a **Seção 9.2**.

13. Porque se **realloc()** retornar **NULL**, o bloco para o qual o ponteiro apontava estará irremediavelmente perdido.

14. (a) Literalmente, é uma variável sem nome. (b) Porque, em programação de alto nível, apenas variáveis alocadas dinamicamente são anônimas.

15. Resposta simples: porque elas não são alocadas automaticamente. Resposta complexa: mesmo assim, elas poderiam ser liberadas automaticamente, mas seria necessário implementar um esquema como o de contagem de referência que não é trivial.

16. Uso múltiplo de **free()** e passagem de um ponteiro inválido para **free()** (i.e., um ponteiro que não aponta para início de um bloco alocado dinamicamente)

17. Quando essa função for chamada, o ponteiro passado como primeiro parâmetro não estará apontando para o array criado pela função.

18. Se **realloc()** retornar **NULL**, o bloco para o qual o ponteiro `p` apontava não poderá mais ser acessado.

19. Consulte a **Seção 9.3**.

20. Consulte a **Seção 9.3**.

21. Consulte a **Seção 9.3**.

22. Consulte a **Seção 9.2**. O **Capítulo 11** apresenta outros exemplos.

23. Não é permitida a aplicação do operador de indireção sobre ponteiros do tipo **void** *.

24. Consulte a **Seção 9.4**.

25. Consulte a **Seção 9.4**.

26. Consulte a **Seção 9.4**.

27. (a) Zumbi de heap é uma variável anônima que foi liberada com **free()** e continua sendo utilizada. (b) Zumbi de pilha é uma variável local de duração automática que continua sendo usada após ter sido automaticamente liberada (porque encerrou a execução do bloco no qual ela é definida). Zumbis de pilha são discutidos no **Capítulo 3**.

28. (a) Se uma função, logo antes de retornar, tem um ponteiro local apontando para um bloco alocado dinamicamente cujo endereço não é retornado, esse espaço jamais poderá ser acessado novamente. É isso que se chama escoamento de memória. (b) Se essa função for chamada muitas vezes, a capacidade do heap poderá ser esgotada bem antes do previsto devido ao escoamento de memória causado pela função e, consequentemente, o programa será abortado.

29. Consulte a **Seção 9.5**.

30. Consulte a **Seção 9.5**.

31. Não há garantia de que o espaço para o qual `ar` aponta tenha sido alocado.

32. (a) É um array cujo tamanho é conhecido em tempo de programação. (b) É um array cujo tamanho só é conhecido em tempo de execução e pode variar durante a execução de um programa.

33. Além de não precisar ser previamente estimado, o tamanho de um array dinâmico pode ser ajustado de acordo com a demanda do programa. O tamanho de um array estático precisa ser previamente estimado e não pode ser alterado durante a execução do programa.

34. **realloc()** permite o redimensionamento de arrays dinâmicos de acordo com a demanda.

35. Consulte a **Seção 9.6**.

36. (a) Para liberar o espaço ocupado por listas desse tipo. (b) Sim porque, mesmo quando os elementos da lista são alocados estaticamente, a lista em si é alocada dinamicamente.

37. Não.

38. Porque se **realloc()** for bem sucedida, o bloco para o qual esse ponteiro apontava pode ter sido realocado noutro endereço em memória.

39. Um programa-cliente não precisa ser alterado quando usa qualquer das duas implementações.

40. A chamada de **malloc()** aloca espaço para a estrutura que armazena a pilha enquanto a chamada de **calloc()** aloca espaço para o array que armazena os elementos da pilha.

41. Uma chamada é para liberar o espaço ocupado pela variável que representa a pilha e a outra é para para liberar o espaço ocupado pelo array que armazena os elementos da pilha.

42. Consulte a **Seção 9.7.2**.

43. (a) Porque, durante o processo de redimensionamento, o array original precisa ser copiado para o início do array novo e essa operação tem custo temporal $\theta(n)$, em que n é o tamanho do array. (b) Porque, durante esse processo, dois arrays são mantidos em memória.

44. Porque a função `AcrescentaListaIdxD()` da **Seção 9.6** usa **realloc()**, que, no pior caso, tem custo temporal $\theta(n)$. A função `AcrescentaListaIdx()` da **Seção 7.1** não faz o mesmo.

45. $\theta(2n)$ é o mesmo que $\theta(n)$, como se mostra no **Capítulo 6**.

46. $\theta(1)$.

47. Consulte a **Seção 9.9.1**.

48. Consulte a **Tabela 9–2**.

49. Consulte a **Seção 9.9.2**.

Capítulo 10 — Listas Encadeadas

1. (a) Em situações que requerem pouca ou nenhuma redimensionamento. (b) Consulte a **Seção 10.1**.

2. Porque a contiguidade em memória deixa de ser garantida.

3. (a) Consulte a **Seção 10.2.1**. (b) Consulte a **Seção 9.1** (**Capítulo 9**).

4. (1) Listas encadeadas não permitem acesso direto e (2) usam espaço adicional com custo $\theta(n)$.

5. Listas encadeadas podem aumentar e diminuir de tamanho de acordo com a demanda de um programa.

6. Listas encadeadas não requerem o uso de **realloc()** para serem redimensionadas.

7. Representação de polinômios.

8. Porque é o único modo garantido de localização desse elemento, já que os elementos de uma lista encadeada podem não ser contíguos em memória.

9. Visitar um elemento de uma lista (encadeada ou não) significa acessá-lo com o objetivo de realizar alguma operação sobre seu conteúdo.

10. Insere `valor[j]` depois de `valor[i]` na lista.

11. (a) No início da lista. (b) Idem.

12. Porque o último nó da lista precisa ser acessado sequencialmente com custo $\theta(n)$.Inserção no início, por outro lado, tem custo $\theta(1)$.

13. (a) O ponteiro `p` apontará para o nó a ser removido e o ponteiro `q` apontará para o nó que antecede o nó a ser removido. (b) O papel de `p` é encontrar o nó a ser removido e o papel de `q` é fazer as devidas alterações no nó anterior ao nó a ser removido. Usando apenas o ponteiro `q` é possível efetuar todas as alterações necessárias para a remoção do nó, mas é impossível fazer o ponteiro `q` apontar para o nó anterior sem a ajuda do nó `p`.

14. O parâmetro do tipo `tListaSE *` da função `RemoveListaSE()` é um parâmetro de entrada e saída, enquanto o parâmetro `tListaSE` da função `BuscaListaSE()` é um parâmetro de entrada (apenas). Se não entendeu, releia a **Seção 2.1** (**Capítulo 2**).

15. Porque se a função `RemoveListaSE()` assim procedesse causaria a perda do endereço inicial da lista.

16. Deve-se ter cuidado para não liberar um nó antes de acessar seu sucessor.

17. Para guardar o endereço do próximo nó a ser liberado antes que o nó corrente seja destruído.

18. Iniciação, comprimento, checagem de lista vazia e destruição.

19. Porque não há garantia de que elementos de uma lista encadeada sejam adjacentes em memória.

20. (1) Alocar espaço dinamicamente para uma variável do tipo `tConteudo`, armazenar o resultado da busca nessa variável e retornar seu endereço. (2) Armazenar o resultado da busca numa variável local de duração fixa do tipo `tConteudo` e retornar o endereço dessa variável. (3) Usar um parâmetro adicional de saída do tipo `tConteudo *` e armazenar o resultado da busca nesse parâmetro. (A última opção é a melhor.)

21. (a) Para permitir aos clientes acessar sequencialmente os conteúdos dos elementos da lista. (b) (1) Exibição de conteúdos de todos os elementos, (2) escrita desses conteúdos em arquivo, (3) execução de uma operação qualquer sobre o conteúdo de cada elemento da lista.

22. Apenas inversão [item (g)].

23. (a) Um novo nó contendo `chave` substituirá o primeiro elemento da lista. (b) Apenas a afirmação (iii) é verdadeira.

24. (a) No início da lista. (b) Idem.

25. `p-> proximo = q-> proximo; q-> proximo = p;` (b) $\theta(1)$.

26. (a) O ponteiro `p` apontará para a posição de inserção e o ponteiro `q` apontará para o nó que antecede o nó a ser removido. (b) O papel de `p` é encontrar o nó a ser removido e o papel de `q` é fazer as devidas alterações no nó anterior ao nó a ser removido. Usando apenas o ponteiro `q` é possível efetuar todas as alterações necessárias para a remoção do nó, mas é impossível fazer o ponteiro `q` apontar para o nó anterior sem a ajuda do nó `p`.

27. Porque se `p` assumir o valor **NULL**, a expressão `p->conteudo` causará o aborto do programa. Se o ponteiro `p` for avaliado antes e for **NULL**, o curto circuito do operador de conjunção (**&&**) impedirá que essa expressão seja avaliada.

28. (a) Válida. (b) Inválida porque `ptr2` aponta para uma estrutura (e não para um inteiro). (c) Inválida porque o operador ponto tem precedência maior do que o operador de indireção e `ptr1` é um ponteiro. (d) Válida.

29. (a) `21`. (b) `1`. (c) É imprevisível. (d) `0`. (e) `1`. (f) `35`. (g) É imprevisível.

30.

```
ptr1 = ptr1->proximo;
lista->proximo->proximo = ptr1->proximo;
free(ptr1);
```

31.

```
pNovo->proximo = ptr1->proximo->proximo->proximo;
ptr1->proximo->proximo->proximo = pNovo;
```

32. Esta questão é facilitada com o auxílio da **Figura D–4**.

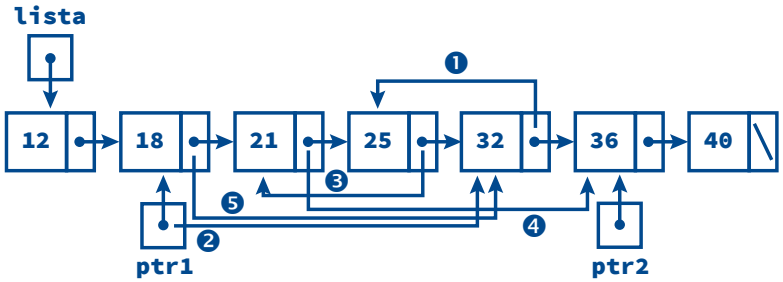

FIGURA D–4: QUESTÃO 50 — CAPÍTULO 10

```
1 ptr1->proximo->proximo->proximo->proximo = ptr1->proximo->proximo;
2 ptr1 = ptr1->proximo->proximo->proximo;
3 lista->proximo->proximo->proximo->proximo = lista->proximo->proximo;
4 lista->proximo->proximo->proximo = pt2;
5 lista->proximo->proximo = ptr1;
```

33.

(a) `ptr2 = lista;`

(b) `ptr1 = ptr2->proximo;`

(c) `ptr2->proximo->proximo = lista;`

(d) `lista = ptr1->proximo->proximo;`

(e) `ptr2 = NULL;`

34. Remoção do último elemento da lista requer que o penúltimo elemento da lista seja alterado e para tal quase todos os nós da lista precisam ser acessados.

35. Porque busca binária requer acesso direto aos elementos da lista e lista encadeada não permite esse tipo de acesso.

36. Consulte a **Seção 10.4**.

37. Quando se requer com frequência acesso ao antecessor e ao sucessor de um nó da lista.

38. Consulte a **page 418**.

39. Consulte a **Seção 10.4.4**.

40. Consulte a **Seção 10.4.2**.

41. (a) A partir de um nó é possível acessar qualquer outro nó da lista. (b) Usam mais espaço em memória do que listas simplesmente encadeadas.

42. (a) Consulte a **Seção 10.4.4**. (b) Deve-se tomar cuidado para não confundir a cabeça com um nó ordinário da lista.

43. (a) No início da lista. (b) Idem.

44. (a) No início da lista. (b) Idem.

45. (a) Não há. (b) Idem.

46. Todas, exceto (a).

47. (a) 40. (b) `1`. (c) `1`. (d) Essa operação não faz sentido. Então, o resultado pode ser `0` ou `1`. (e) `17`.

48.

```
free(lista->anterior);
lista->anterior = ptr2;
ptr2->proximo = lista;
```

49.

```
pNovo->proximo = lista;
pNovo->anterior = lista->anterior;
pNovo->anterior->proximo = pNovo;
lista = pNovo;
ptr1->anterior->anterior = pNovo;
```

50. Esta questão é muito difícil de resolver sem o auxílio gráfico da **Figura D–5**.

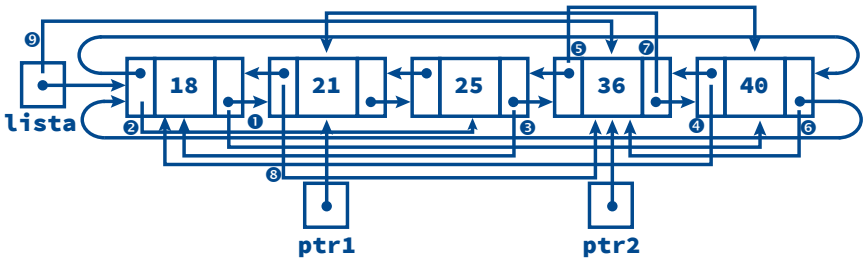

FIGURA D–5: QUESTÃO 50 — CAPÍTULO 10

```
1 lista->proximo = lista->anterior;
2 lista->anterior = ptr1->proximo;
3 ptr1->proximo->proximo = lista;
4 ptr2->proximo->anterior = lista;
5 ptr2->anterior = ptr2-> proximo;
6 ptr2->proximo->proximo = ptr2;
7 ptr2->proximo = ptr1;
8 ptr1->anterior = ptr2
9 lista = ptr2;
```

51. (a) Essa operação permite aos clientes acessar os conteúdos da lista sequencialmente do último para o primeiro elementos. (b) Listas duplamente encadeadas.

52. (a) Lista simplesmente encadeada circular. (b) Lista simplesmente encadeada ordenada. (c) Lista duplamente encadeada circular.

53. (a) `p3->proximo`. (b) `p1->anterior`. (c) `p1->proximo` ou `p2->anterior`. (d) `p3->anterior->anterior`.

54. Testando se o ponteiro `proximo` da cabeça aponta para a própria cabeça.

55. Consulte a **Seção 10.5.1**, o **Capítulo 8** e o **Capítulo 9**.

56. (a) Não. (b) Não. (c) Não.

57. Todas elas possuem custo temporal $\theta(1)$.

58. Consulte a **Seção 10.5.2**, o **Capítulo 8** e o **Capítulo 9**.

59. (a) Não. (b) Não. (c) Sim.

60. Todas elas possuem custo temporal $\theta(1)$.

61. Não. São maneiras diferentes de implementação do mesmo conceito.

62. Se a lista não for ordenada, nenhuma delas leva vantagem sobre a outra, pois, em ambos os casos, a inserção pode ser efetuada com custo temporal $\theta(1)$. Se a lista for ordenada, lista encadeada leva vantagem com relação a lista indexada devido ao fato de não requerer deslocamento de elementos durante a operação. No entanto, em termos de análise assintótica, ambas as operações têm custo $\theta(n)$.

63. (a) (2), pois lista encadeada não requer o deslocamento de elementos durante a inserção. (b) (1), porque lista indexada ordenada permite busca binária. (c) (1), pois lista encadeada não permite acesso direto.

64. (a) Encerrando a busca quando for encontrado um valor maior do que aquele procurado. (b) Não.

65. Consulte a **Seção 10.7.1**.

66. Consulte a **Seção 10.7.3**.

67. $\theta(n)$, em que n é o grau do polinômio que possui o maior grau.

68. Se a lista fosse linear, seria necessária uma instrução adicional para checar quando o final da lista fosse atingido e, quando esse fosse o caso, reiniciar o processamento a partir do início da lista. Em resumo, esse problema tem natureza inerentemente circular e o uso de lista circular é mais eficiente e natural.

Capítulo 11 — Estruturas de Dados e Algoritmos Genéricos

1. O nome de uma função corresponde ao seu endereço em C. Assim um nome de função pode ser atribuído a um ponteiro para função (desde que haja compatibilidade, é claro).

2. (a) A variável `pf` é um ponteiro para funções que recebem um parâmetro do tipo **int** e retornam um valor do tipo **int**. (b) Essa declaração será interpretada como uma alusão de função.

3. Ponteiros para funções são usados com mais frequência como parâmetros de funções (v. **Seção 11.1.5**).

4. (a) `int (*pf)(int, int)`. (b) `int *(*pf)(int *, int *)`.
(c) `int *F(int (*pf)(int, int))`.

5. (b) `F2` não é compatível com o segundo parâmetro de `UmaFuncao()`. (c) `F3(1.5)` é um valor do tipo **int** e não é compatível com ponteiro para função. (d) `F4` não é compatível com o segundo parâmetro de `UmaFuncao()`. (e) `F4(2.1)` é um valor constante e não se pode aplicar o operador `&` a ele. (f) Similar ao item (d).

6. Todas são legais.

7. Consulte a **Seção 11.2.1**.

8. Consulte a **Seção 11.2.1**.

9. (a) *(a)*. (b) *((b, (c)))*. (c) *(b, (c))*. (d) *()*. (e) Indefinido. (f) *a*.

10. (a) *(d)*. (b) Indefinido.

11. (ii).

12. Para indicar se ele é um átomo ou um ponteiro para uma sublista.

13. Consulte a **Seção 11.2.3**.

14. Porque listas generalizadas são definidas recursivamente.

15. Consulte a **Seção 11.3**.

16. Consulte a **Seção 11.3**.

17. É uma pilha que serve para armazenar elementos de quaisquer tipos de dados.

18. Consulte a **Seção 11.4.5**.

19. Consulte a **Seção 11.4.5**.

20. (a) O ponteiro `p->elementos` é do tipo **void** *. (b) Erro de compilação.

21. Consulte a **Seção 11.5**.

22. Trocando o primeiro pelo segundo parâmetro dessa função no corpo dela e vice-versa.

23. Para permitir comparação de elementos de quaisquer tipos.

24. Consulte a **Seção 11.5**.

25. Consulte a **Seção 11.5**.

26. Consulte a **Seção 11.5**.

27. Consulte a **Seção 11.6**.

28. (a) Historicamente, notação polonesa era o mesmo que notação prefixa. Posteriormente, essa denominação passou a identificar tanto a notação prefixa quanto a sufixa. (b) Nenhuma dessas notações requer o uso de parênteses.

29.

(a) Prefixa: `-+ABC`. Sufixa: `AB+C-`.

(b) Prefixa: `*+AB-CD`. Sufixa: `AB+CD-*`.

(c) Prefixa: `**+AB%-CDEF`. Sufixa: `AB+CD-E%*F*`.

(d) Prefixa: `-*+AB+%C-DEFG`. Sufixa: `AB+CDE-%F+*G-`.

(e) Prefixa: `/*+AB%C-DEF`. Sufixa: `AB+CDE-%*F/`.

(f) Prefixa: `++*+ABD/E+F*ADC`. Sufixa: `AB+D*EFAD*+/+C+`.

30. (a) `C + B - A`. (b) `(E - F + D)%(C + A - B)`. (c) `E*F - ((D - E + C)%B)*A`. (d) `D%E*C/B - A`.

31. `A/(B%C) + D*E - A*C`

32. Consulte a **Seção 11.6**.

33. É sua vez.

34. Idem.

35. (a) Consulte a **Seção 11.7.1**. (b) Porque tudo o que se sabe a respeitos de cada elemento do array a ser ordenado é o seu tamanho (i.e., seu número de bytes).

Capítulo 12 — Árvores

1. Consulte a introdução do **Capítulo 12**.

2. Consulte a **Seção 12.1**.

3. Tomada de decisões, jogos, busca e codificação.

4. Porque um elemento de uma árvore pode ser considerado como superior, inferior ou do mesmo nível que outro elemento.

5. Consulte a **Seção 12.1**.

6. (a) 3. (b) `A`. (c) `E`, `F`, `G`, `H`, `I` e `J`. (d) 3. (e) 2. (f) 2.

7. (a) `A` não possui pai; pai de `B`, `C` e `D`: `A`; pai de `E`, `F` e `G`: `B`; pai de `H` e `I`: `C`; pai de `J`: `D`. (b) Filhos de `A`: `B`, `C` e `D`; filhos de `B`: `E`, `F` e `G`; filhos de `C`: `H` e `I`; filho de `D`: `J`; os demais nós não possuem filhos. (c) `A` não possui ancestrais; ancestral de `B`, `C` e `D`: `A`; ancestrais de `E`, `F` e `G`: `B` e `A`; ancestrais de `H` e `I`: `C` e `A`; ancestrais de `J`: `D` e `A`. (d) Nível de `A`: 1; nível de `B`, `C` e `D`: 2; nível de `E`, `F`, `G`, `H`, `I` e `J`: 3.

8. A menor árvore com profundidade p possui um nó em cada nível. Portanto o número mínimo de nós numa árvore com profundidade p é dado por:

$$\sum_{i=1}^{p} 1 = p$$

9. A prova será feita por indução sobre o nível p de um nó.

Base da indução. Para $p = 2$, a afirmação é válida pois um nó com essa profundidade é filho da raiz e, portanto, a raiz é ancestral de qualquer nó no nível *2*.

Hipótese indutiva. Suponha que a raiz seja ancestral de qualquer nó no nível $p = k$.

Passo indutivo. Resta mostrar que a raiz é ancestral de qualquer nó no nível $k + 1$.

O pai de um nó que se encontra no nível $k + 1$ situa-se no nível k e, de acordo com a hipótese indutiva, a raiz é ancestral de qualquer nó no nível k. Portanto a raiz é ancestral de qualquer nó que se encontre no nível $k + 1$. ■

10. Consulte a **Seção 12.2**.

11. Não. Consulte a **Seção 12.2**.

12. Consulte a **Seção 12.2**.

13. (a) `DEHIJL`. (b) `GCA`. (c) 5. (d) 4. (e) 16.

14. $n - 1$. Provar essa afirmação por indução é relativamente fácil (v. **Apêndice B**).

15. Depende. Elas serão consideradas equivalentes se os filhos forem ambos direitos ou esquerdos.

16. V. **Figura D–6**.

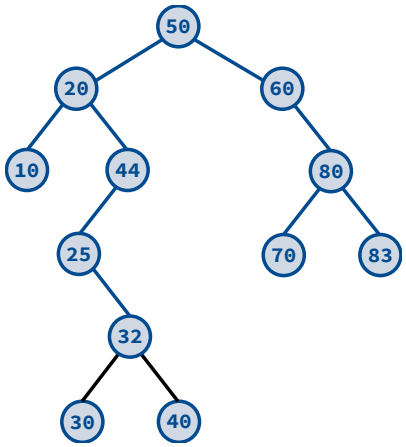

Figura D–6: Questão 16 — Capítulo 12

17. Pode-se mostrar que o número de árvores binárias estruturalmente diferentes possíveis com n nós é dado por:

$$\frac{(2n)!}{(n+1)!\,n!}$$

Portanto, para $n = 3$, tem-se que o número de árvores binárias estruturalmente diferentes possíveis com três nós é *5*. (b) Essas árvores são apresentadas na **Figura D–7**.

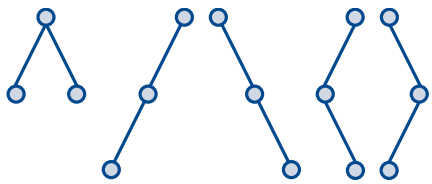

Figura D–7: Questão 17 — Capítulo 12

18. Consulte a **Seção 12.2.3**.

19. Consulte a **Seção 12.2.4**.

20. A prova é feita por redução ao absurdo. Suponha que um determinado nó N possui dois pais $P1$ e $P2$. Então, esse nó faz parte da subárvore que tem $P1$ como raiz e também faz parte da subárvore que tem $P2$ como raiz. Portanto essas duas subárvores não são disjuntas, o que contradiz a hipótese de que elas fazem parte de uma árvore binária.

21. Consulte a **Seção 12.2**.

22. De acordo com o **Corolário 12.1**, o número de nós dessa árvore é $2n_0 - 1$.

23. Consulte o **Teorema 12.3**.

24. De acordo com o **Teorema 12.1**, o número máximo de nós no nível k de uma árvore binária é 2^{k-1}.

25. (a) 2^i. (b) $2^{n+1} - 1$. A prova desses resultados é semelhante àquela do **Teorema 12.1**, que se encontra no **Apêndice B**.

26. Se a árvore possui pelo menos n níveis, ela é repleta até o nível $n - 1$ e o número de nós até esse nível é, de acordo como o **Teorema 12.1**, $2^{n-1} - 1$. Como a numeração começa com 0, o último nó no nível $n - 1$ é numerado com $2^{n-1} - 2$. Assim o primeiro nó no próximo nível será numerado como $2^{n-1} - 1$. ■

27. Seja v_n o número de ramificações de uma árvore com n nós. Então, deseja-se provar que $v_n = n + 1$. A prova será feita por indução (v. **Apêndice B**).

Para $n = 1$, a árvore possui apenas a raiz com duas ramificações nulas, de modo que $v_1 = 2 = 1 + 1$. Agora, suponha que a proposição é válida para $n = k$; i.e., $v_k = k + 1$. Então, deve-se mostrar que $v_{k+1} = k + 2$. Mas, acrescentando-se um nó à árvore com k nós que tinha $k + 1$ ramificações nulas, a árvore perde uma de suas ramificações nulas e ganha duas novas ramificações nulas. Assim o número de ramificações nulas na árvore com $k + 1$ nós é $k + 1 - 1 + 2 = k + 2$. ■

28. (a) Consulte a **Seção 12.2.4**. (b) Quando a árvore é patológica. (c) Quando a árvore é perfeita.

29. (a) Para um dado número de nós, a árvore mais profunda que se pode ter é uma árvore patológica que possui apenas um nó em cada nível. Portanto a resposta é 100. (b) Para um dado número de nós, a árvore menos profunda que se pode ter é uma árvore que tenha o número máximo de nós até o nível $p - 1$, que é uma árvore completa. De acordo com o **Teorema 12.6**, a profundidade dessa árvore deve ser:

$$p = \lfloor \log_2 n + 1 \rfloor = \lfloor \log_2 100 + 1 \rfloor = 7$$

30. De acordo com o **Teorema 12.7**, o número de nós internos é 100.

31. O número de ancestrais de um nó corresponde ao seu nível menos um. Numa árvore binária perfeita todos as folhas estão no último da árvore. Portanto o número de ancestrais de uma folha de uma árvore binária perfeita é igual a profundidade da árvore menos um.

De acordo com o **Teorema 12.1**, o número de nós de uma árvore binária perfeita é dado por:

$n = 2^p - 1 \Rightarrow p = \lfloor log_2(n + 1) \rfloor$

O uso da função piso (v. **Apêndice B**) na expressão acima é necessário porque não se deseja que o valor de p seja maior do que o valor resultante do logaritmo. Portanto o número de ancestrais de uma folha de uma árvore binária perfeita é dado por $\lfloor log_2(n + 1) \rfloor - 1$.

32. V. **Figura D–8**.

33. (a) Se o valor de cada nó for maior do que o seu antecessor, só haverá inserção de nós em subárvores direitas e a árvore será inclinada à direita. (b) (a) Se o valor de cada nó for menor do que o seu antecessor, só haverá inserção de nós em subárvores esquerdas e a árvore será inclinada à esquerda.

34. V. **Figura D–9**.

35. Essa é uma paráfrase do **Teorema 12.2**, que é demonstrado no **Apêndice B**.

36. Consulte o **Teorema 12.7** na **Seção 12.2.4**.

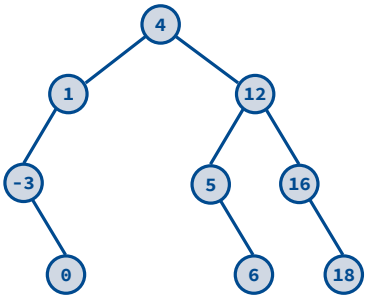

Figura D–8: Questão 32 — Capítulo 12

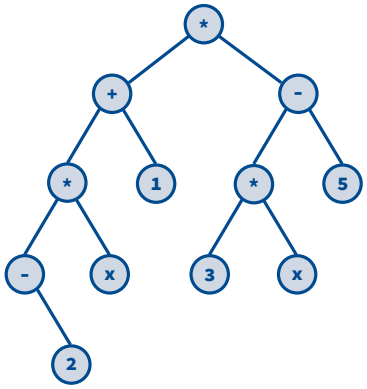

Figura D–9: Questão 34 — Capítulo 12

37. (a) Árvores binárias completas: (i), (ii), (v) e (vi). (b) Árvores binárias perfeitas: (i) e (v). (c) Não são completas nem perfeitas: (iii) e (iv). (d) Árvores estritamente binárias: (i), (v) e (vi).

38. V. **Figura D–10**.

Figura D–10: Questão 38 — Capítulo 12

39. (a) A **Figura D–11** mostra uma árvore binária de Fibonacci de ordem 4. Essa figura mostra claramente que essa árvore não é estritamente binária. (b) O número de folhas na árvore de Fibonacci de ordem *n* corresponde exatamente ao número de Fibonacci de ordem *n*.

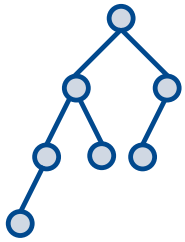

Figura D–11: Questão 39 — Capítulo 12

40. Consulte a **Seção 12.3**.

41. Consulte a **Seção 12.3**.

42. Consulte a **Seção 12.3**.

43. Em cada tipo de caminhamento, primeiro efetua-se um caminhamento recursivo na subárvore esquerda e depois faz-se o mesmo na subárvore direita. Além disso, em cada um desses caminhamentos recursivos, o último nó visitado é uma folha.

44. (a) Ordem infixa: `DHBEAIFCGJ`. (b) Ordem prefixa: `ABDHECFIGJ`. (c) Ordem sufixa: `HDEBIFJGCA`.

45. (a) (i) `BCDAFEHG` (ii) `CDBAFEHG` (iii) `BFDAEGC`. (b) (i) `ABCDEFGH` (ii) `ABCDEFGH` (iii) `ABDFCEG`. (c) (i) `DCBFHGEA` (ii) `DCBFHGEA` (iii) `FDBGECA`.

46. V. **Figura D–12**.

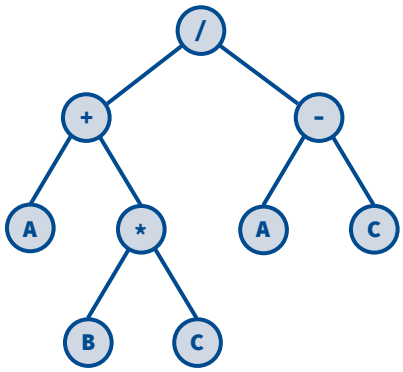

Figura D–12: Questão 46 — Capítulo 12

47. Não. Efetue um caminhamento em ordem infixa e verá que o resultado não será uma expressão aritmética bem formada.

48. A **Figura D–13** mostra onde o nó contendo a letra *N* deve ser inserido.

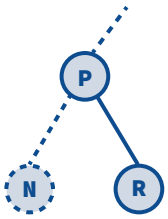

Figura D–13: Questão 48 — Capítulo 12

49. `2*(a + b - c/d)`.

50. `33`.

51. O caminhamento em ordem sufixa na **Árvore 1** coincide com o caminhamento em ordem infixa na **Árvore 2** (confira).

52. (a) V. abaixo. (b) `ZURPLKIGE`. (c) `LUZPRGKIE`.

Caminhamento Árabe Sufixo

1. Caminhe na subárvore direita
2. Caminhe na subárvore esquerda
3. Visite a raiz

53. Essa função exibe o conteúdo de cada nó da árvore por nível, da esquerda para a direita.

54. A chamada de `Misteriosa(arvore)` apresenta na tela: `ABCDEF`.

55. Porque o uso de ponteiros para funções permite alterar o tipo de operação a ser efetuada em cada nó sem ter que alterar a função que implementa um caminhamento.

56. Árvores binárias são estruturas de dados definidas recursivamente. Portanto implementações recursivas de operações sobre árvores binárias são naturais.

57. Consulte a **Seção 12.4.5**.

58. Não. Se o caminhamento fosse efetuado em ordem prefixa, uma raiz seria destruída antes de seus possíveis filhos, o que tornaria os ponteiros para eles inválidos. Se o caminhamento fosse efetuado em ordem infixa, uma raiz seria destruída antes de seu possível filho direito, o que tornaria o ponteiro para esse filho inválido.

59. V. **Figura D–14**.

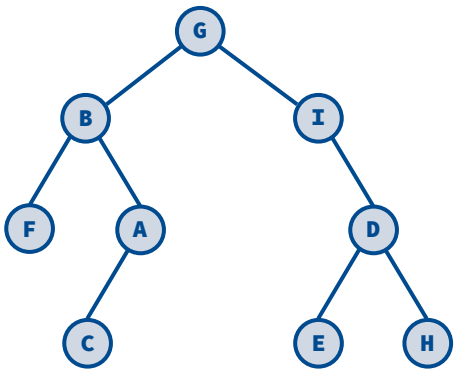

Figura D–14: Questão 59 — Capítulo 12

60. (a) Não. (b) Não.

61. (a) Sim. (b) Sim. (c) Não.

62. Quando há nós com mesmo conteúdo.

63. Não.

64. (a) V. **Figura D–15**. (b) Impossível.

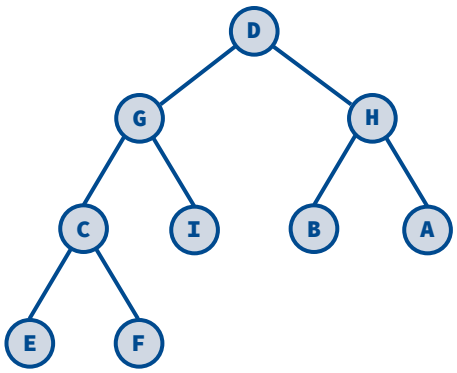

Figura D–15: Questão 64 — Capítulo 12

65. Sim.

66. Consulte a **Seção 12.6**.

67. Consulte a **Seção 12.6**.

68. Consulte a **Seção 12.6**.

69. Não.

70. Não confundir ramificação da árvore com costura.

71. Por uma questão de legibilidade.

72. Provavelmente, iteração infinita.

73. Consulte a **Seção 12.7**.

74. V. **Figura D–16**.

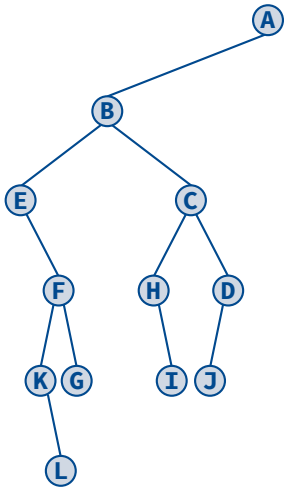

FIGURA D–16: QUESTÃO 74 — CAPÍTULO 12

75. Consulte a **Seção 12.7**.

76. V. **Figura D–17**.

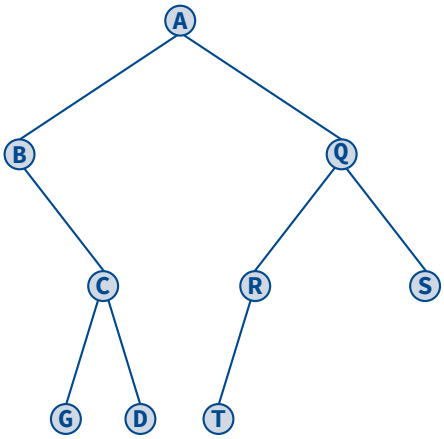

FIGURA D–17: QUESTÃO 76 — CAPÍTULO 12

77. Consulte a **Seção 12.8.1**.

78. Consulte a **Seção 12.8.2**.

79. Consulte as **Seções 12.8.2** e **12.2.2**.

80. 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. Em consonância com o **Teorema 12.2**, 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 $n - 1$. ■

81. Consulte a **Seção 12.8.2**.

82. (a) `'a'`, `'b'` e `'n'`. (b) `00`, `01` e `1`, respectivamente. (c) Significam que os conteúdos dos respectivos nós não são relevantes.

83. Seria decodificada como `banana`.