UESC

Universidade Estadual de Santa Cruz

Departamento de Ciências Exatas e Tecnológicas

Bacharelado em Ciência da Computação

CET 077 Estruturas de Dados

Exercícios

e Soluções

Paulo Costa

Agosto de 2015

Sumário

Prefácio

Esta apostila contém mais de cem exercícios sobre os dez temas principais da disciplina CET 077 Estruturas de Dados do curso de bacharelado em Ciência da Computação da UESC. Seu objetivo é fornecer aos alunos da disciplina amplas oportunidades de prática dos conceitos e técnicas abordados em sala de aula e, assim, ajudar a consolidar seus conhecimentos.

Não foram incluídos alguns temas que integram o conteúdo programático da disciplina em outras instituições, como árvores AVL, árvores *splay*, árvores (2,4) e árvores vermelho-pretas, pois tais temas podem ser considerados desdobramentos de árvores binárias simples – estas, sim, incluídas na apostila.

A apostila está dividida em duas partes:

- **Parte 1 – Exercícios**
  Subdivida em dez seções, correspondendo aos dez temas abordados.
- **Parte 2 – Soluções**
  Também subdivida em dez seções, contendo soluções de exercícios selecionados.

A fim de facilitar a navegação na apostila, os exercícios e suas respectivas soluções foram conectados com *hiperlinks* clicáveis. Há também conexões clicáveis entre o sumário e as várias partes e seções.

Todos os exercícios possuem uma indicação do seu grau de dificuldade, desde os mais fáceis, marcados com uma estrela (✪), aos mais difíceis, marcados com quatro estrelas (✪✪✪✪). Em cada seção os exercícios são apresentados em ordem crescente de dificuldade.

A apostila é distribuída em um arquivo comprimido contendo recursos adicionais, dos quais os mais importantes são os projetos de programação. Um projeto de programação consiste na implementação da solução de determinado exercício da apostila. Cada projeto corresponde a um exercício, mas nem todo exercício possui um projeto correspondente. Os projetos foram implementados na linguagem C e no ambiente de desenvolvimento CodeBlocks. Há dois tipos de projetos:

- **Projetos completos**
  Trazem soluções completas, ou seja, o código está completo e pode ser prontamente recompilado.
- **Projetos a completar**
  São cópias dos projetos completos das quais foram removidas as partes principais do código. Assim, o interessado será capaz de solucionar o respectivo problema reimplementando somente a parte principal do programa, sem perder tempo construindo a infraestrutura necessária para compilar e testar o código. Nestes projetos, o código removido, e que deve ser reescrito, está marcado com comentários do tipo

```
/* +-------------------------+ */
/* | ESCREVA AQUI SEU CODIGO | */
/* +-------------------------+ */
```

Esta apostila e o material adicional que a acompanha foram concebidos como complementos, e não como substitutos, aos livros texto recomendados e às aulas presenciais. Sugestões, correções e críticas são bem-vindas, favor enviar para **psscosta@uesc.br**

Paulo Costa
Ilhéus, BA, 6 de Agosto de 2015

Parte 1 – Exercícios

1. Recursão

Exercício 1.1 ✪

1) Determine o que a seguinte função recursiva em C calcula:

```
int func (int n) {
  if (n == 0)
    return 0;
  return (n + func(n-1));
}
```

2) Escreva uma função iterativa que atinja o mesmo objetivo.

Exercício 1.2 ✪

Escreva as funções a seguir. Todas tomam como entrada dois inteiros $a, b \geq 0$ e retornam um inteiro.

a) `somaI`, uma função iterativa que retorna $a + b$.
b) `somaR`, uma função recursiva que retorna $a + b$.
c) `multI`, uma função iterativa que retorna $a \times b$. Faça uma chamada a `somaI` se desejar.
d) `multR`, uma função recursiva que retorna $a \times b$. Faça uma chamada a `somaR` se desejar.

Não use qualquer operador aritmético além de incremento (`++`) ou decremento (`--`).

Exercício 1.3 ✪

Escreva as funções recursivas listadas a seguir. Todas tomam como entrada um inteiro $n > 0$ e retornam um inteiro. Não use laços (`for`, `while`) nem declare variáveis locais.

a) `maiorDigito`, que retorna o maior de todos os dígitos de n.
b) `menorDigito`, que retorna o menor de todos os dígitos de n.

Exemplos:

```
Digite um inteiro positivo (ou 0 para sair): 1234
O menor digito de 1234 eh 1
O maior digito de 1234 eh 4

Digite um inteiro positivo (ou 0 para sair): 4321
O menor digito de 4321 eh 1
O maior digito de 4321 eh 4
```

Sugestão: $n\%10$ produz o dígito mais à direita de n, enquanto $n/10$ produz todos os dígitos de n exceto aquele mais à direita.

Exercício 1.4 ✪✪

Tente reescrever as funções do exercício 1.2 obedecendo todos os critérios abaixo:

a) Não utilize qualquer operador aritmético além de incremento (`++`) ou decremento (`--`).
b) Não passe qualquer parâmetro além dos dois operandos (a e b).
c) Não declare qualquer variável local, nem mesmo para o valor a ser retornado.

Se não for possível obedecer todos para alguma função, forneça uma solução que viole só um critério.

Exercício 1.5 ✪✪

Escreva as funções recursivas listadas a seguir. Todas tomam como entrada um inteiro $n > 0$ e retornam um inteiro. Não use laços (`for`, `while`) nem declare variáveis locais.

a) `contaDigitos`, que retorna a quantidade de dígitos de n.
b) `somaDigitos`, que retorna a soma dos dígitos de n.

Exemplos:

```
Digite um inteiro positivo (ou 0 para sair): 1234
1234 possui 4 digitos e sua soma eh 10

Digite um inteiro positivo (ou 0 para sair): 23456
23456 possui 5 digitos e sua soma eh 20
```

Exercício 1.6 ✪✪

Calcule o número de adições necessárias para computar $\text{fib}(n)$ para $0 \leq n \leq 10$ usando os métodos iterativo e recursivo implementados a seguir:

```
/* Números de Fibonacci, método iterativo */

int iFIB (int n) {
  int i, f1=0, f2=1, f3=0;
  for (i = 2; i <= n; i++) {
    f3 = f1 + f2; f1 = f2; f2 = f3;
  }
  return (f3);
}

/* Números de Fibonacci, método recursivo */

int rFIB (int n) {
  if (n<=1) return (n);
  return (rFib (n-1) + rFib (n-2));
}
```

n	$A_i(n)$	$A_r(n)$
0		
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		

Coloque seus resultados na tabela ao lado, onde
$A_i(n)$ é o número de adições no cômputo iterativo de $\text{fib}(n)$ e
$A_r(n)$ é o número de adições no cômputo recursivo de $\text{fib}(n)$.

Exercício 1.7 ✪✪✪

Escreva as funções recursivas listadas a seguir. Todas tomam como entrada um inteiro $n > 0$ e retornam um inteiro. Não use laços (`for`, `while`) nem declare variáveis locais.

a) `zeraPares`, que torna zero os dígitos pares de n e retorna o resultado.
b) `zeraImpares`, que torna zero os dígitos ímpares de n e retorna o resultado.
c) `removePares`, que remove os dígitos pares de n e retorna o resultado.
d) `removeImpares`, que remove os dígitos ímpares de n e retorna o resultado.

Exemplos:

```
Digite um inteiro positivo (ou 0 para sair): 1234
1234 com digitos pares   zerados   vira 1030
1234 com digitos impares zerados   vira 204
1234 com digitos pares   removidos vira 13
1234 com digitos impares removidos vira 24

Digite um inteiro positivo (ou 0 para sair): 111
111 com digitos pares   zerados   vira 111
111 com digitos impares zerados   vira 0
111 com digitos pares   removidos vira 111
111 com digitos impares removidos vira 0
```

Exercício 1.8 ✪✪✪

Considere um programa que lê números inteiros pelo teclado e inverte seus dígitos:

```
Digite um inteiro positivo (ou 0 para sair): 98765
O inverso de 98765 eh 56789
```

Vemos a seguir o código desse programa:

```
01 #include <stdio.h>
02 #include <stdlib.h>
03 int inverteRec (int numero, int inverso) {
04   if (numero == 0)
05     return (inverso);
06   return ( ... );      ← Complete este commando
07 }
08 int inverte (int numero){
09   return (inverteRec (numero, 0));
10 }
11 int main() {
12   int numero;
13   printf ("Digite um inteiro positivo (ou 0 para sair): ");
14   scanf ("%d", &numero);
15   while (numero > 0) {
16     printf ("O inverso de %d eh %d \n\n", numero, inverte(numero));
17     printf ("Digite um inteiro positivo (ou 0 para sair): ");
18     scanf ("%d", &numero);
19   }
20   return 0;
21 }
```

Uma parte da função recursiva `inverteRec` está incompleta. Forneça o código que deve ser colocado dentro do `return` na linha 06 para que o programa volte a funcionar corretamente.

Exercício 1.9 ✪✪✪

O máximo divisor comum de dois inteiros x e y pode ser assim definido:

$$\text{mdc}(x,y) = \begin{cases} y & \text{se } y \le x \text{ e } x\%y = 0, \\ \text{mdc}(y,x) & \text{se } x < y, \text{ ou} \\ \text{mdc}(y, x\%y) & \text{caso contrário.} \end{cases}$$

Escreva uma função recursiva e uma iterativa para calcular $\text{mdc}(x,y)$.

Exercício 1.10 ✪✪✪

Por que, no exercício 1.4, não é possível escrever todas as funções obedecendo os três critérios?

Exercício 1.11 ✪✪✪✪

Um problema clássico de *backtracking* é o seguinte: dado um conjunto de inteiros, é possível escolher um subconjunto tal que a soma de seus elementos seja igual a um resultado desejado? Essa estratégia recursiva pode ser usada para resolver outros problemas que envolvem busca em um espaço de opções.

Escreva a função recursiva

```
Bool SomaSubconjunto (int inicio, int *numeros, int totNumeros, int soma)
```

onde

`inicio`	é a posição do *array* a partir da qual se montam os subconjuntos de inteiros, para então testá-los; deve ser sempre 0 na primeira chamada em `main()`,
`numeros`	é o *array* que contém o conjunto de inteiros,
`totNumeros`	é a tamanho do conjunto de inteiros, e
`soma`	é o resultado desejado para a soma dos elementos do subconjunto.

A seguir, exemplos do resultado esperado de `SomaSubconjunto` para algumas entradas:

`SomaSubconjunto (0,{2,4,8},3,10)` deve retornar `TRUE`
`SomaSubconjunto (0,{2,4,8},3,14)` deve retornar `TRUE`
`SomaSubconjunto (0,{2,4,8},3, 9)` deve retornar `FALSE`

Sugestões:

- Nas chamadas recursivas, ao invés de considerar os números no *array* inteiro, considere apenas a parte do *array* começando em `inicio`. Na primeira chamada em `main()` o usuário especifica a busca no *array* inteiro passando o valor 0 para `inicio`. Não é necessário nenhum laço; a recursão trata porções cada vez menores do *array*.
- O caso base é quando `inicio >= totNumeros`. Nesse caso, retorne `TRUE` se `soma` é 0.
- Caso contrário, considere o número na posição `inicio` do *array*. Há apenas duas possibilidades: ou esse número é usado para formar `soma`, ou ele não é usado.
- Faça uma chamada recursiva para verificar se é possível obter uma solução usando esse número (lembre de ajustar o valor desejado de `soma` para refletir o fato que o número já foi usado).
- Se não for possível, faça outra chamada recursiva para verificar se é possível obter uma solução sem usar esse número.
- Se nenhuma das tentativas for bem sucedida, então o problema não tem solução.

2. Abstração de dados

Exercício 2.1 ✪

Defina o tipo de dados abstrato `Complexo`. A estrutura dos números complexos é:

- um número real (p.ex. *float*, *double* ou *racional*) representando a parte real
- um número real representando o coeficiente da parte imaginária.

Defina os métodos `cria`, `soma`, `mult` e `igual`. Lembre que a multiplicação de complexos é feita de forma distributiva, ou seja: $(a + b\mathrm{i}) \times (c + d\mathrm{i}) = ac + (b\mathrm{i} \times d\mathrm{i}) + ad\mathrm{i} + cb\mathrm{i}$

$= ac + bd(-1) + (ad + cb)\mathrm{i}$

$= (ac - bd) + (ad + cb)\mathrm{i}$

Exercício 2.2 ✪

Defina o tipo de dados abstrato `Vetor`. Um vetor v no espaço cartesiano bidimensional pode ser definido por um par ordenado (a, b) onde a e b são números reais, como na Figura 1 abaixo. Defina também os métodos abaixo com os seguintes parâmetros:

Método	Parâmetros	Resultado
`cria`	`float,float`	`vetor`
`modulo`	`vetor`	`float`
`produto`	`vetor,vetor`	`float`
`angulo`	`vetor,vetor`	`float`
`iguais`	`vetor,vetor`	`TRUE ou FALSE`

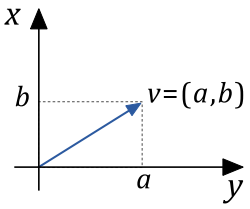


Figura 1

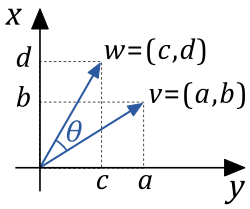


Figura 2

Dados dois vetores $v = (a, b)$ e $w = (c, d)$

- O módulo de v é dado por $|v| = \sqrt{a^2 + b^2}$
- O produto escalar de v e w é dado por $p = v \cdot w = ac + bd$
- O ângulo entre v e w é dado por $\cos^{-1}((v \cdot w)/(|v| \times |w|))$ onde $\cos^{-1}\theta$ é o arco-cosseno (Fig. 2).

Use a notação de vetor $v = \langle v[0], v[1] \rangle$ onde $v[0]$ e $v[1]$ são os coeficientes x e y de v. Note que um vetor não pode ser nulo, ou seja, seus coeficientes não podem ser ambos zero.

Exercício 2.3 ✪✪

Considere os seguintes tipos de dados em C e seus respectivos tamanhos em bytes:

Tipo de dado	Tamanho
`char`	1 byte
`short int`	2 bytes
`int`	4 bytes
`double`	8 bytes

Considere também a seguinte sequência de caracteres: `A B C D 1 2 3 A`. Escreva um programa em C que mostra na tela os valores numéricos obtidos quando se interpreta a sequência acima como `short int`, `int` e `double`. A saída do programa deverá ser algo como ilustrado a seguir.

```
Como char:      A B C D 1 2 3 A
Como short int: 16961 17475 12849 16691
Como int:       1145258561 1093874225
Como double:    1258033.266651
```

Naturalmente os números impressos deverão refletir os caracteres da sequência; se fosse usada outra sequência de caracteres, a saída do programa seria diferente.

3. Vetores

Exercício 3.1 ✪

Considere uma tabela de notas de uma turma de no máximo 30 alunos. Cada linha da tabela contém dois números: matrícula do aluno (do tipo inteiro) e sua nota (do tipo real). Escreva um programa que analise essa tabela e determine:

a) A média da turma.
b) Quantos alunos obtiveram nota acima da média.
c) Quantos alunos obtiveram nota igual à média.
d) Quantos alunos obtiveram nota abaixo da média.
e) A maior nota e quantos alunos a obtiveram.

Exercício 3.2 ✪

O imperador romano César usava um sistema simples para criptografar as mensagens que enviava a seus generais, substituindo cada letra por três letras à frente no alfabeto. Sua tarefa é semelhante, porém mais simples. Escreva um programa que converta cada letra, e somente as letras, de uma mensagem de até 100 caracteres para a letra imediatamente posterior. Note que `z` deve ser convertido para `a` e `Z` para `A`.

Exercício 3.3 ✪

Palíndromos são palavras que podem ser lidas igualmente nas duas direções, como `ARARA`, `RADAR` e `ANILINA`. Note que `AMORA` não é um palíndromo, ainda que `AROMA` seja uma palavra válida.

Escreva a função `ehPalindromo` que

- recebe como parâmetros um ponteiro para a primeira letra da palavra e o número total de letras,
- retorna um booleano (`TRUE` ou `FALSE`) indicando se a palavra é um palíndromo e
- não usa recursão; a função deve ser iterativa.

Sugestão: Primeiro leia os caracteres da palavra e armazene-os em um vetor, depois verifique se trata-se de um palíndromo.

Exercício 3.4 ✪✪

Considere duas sequências A e B, cada uma com até 8 números inteiros entre 0 e 9 inclusive. Escreva um programa que interprete A e B como números inteiros, some-os e produza uma nova sequência C contendo os dígitos da soma. Os dígitos das três sequências devem ser alinhados à esquerda. Posições não ocupadas das sequências devem conter -1. Exemplo:

A	1	2	3	-1	-1	-1	-1	-1
B	6	7	8	9	-1	-1	-1	-1
C	6	9	1	2	-1	-1	-1	-1

Exercício 3.5 ✪✪

Considere duas sequências ordenadas A e B com até max letras de `a` a `z`, por exemplo:

A = `g i i i m r t`
B = `d g g i i k o o t`

Escreva um programa que, dadas as sequências A e B, gere quatro sequências ordenadas com

a) a união de A e B, com repetições. Exemplo: `d g g g i i i i i k m o o r t t`
b) a união de A e B, sem repetições. Exemplo: `d g i k m o r t`
c) a interseção de A e B, com repetições. Exemplo: `g i i t`
d) a interseção de A e B, sem repetições. Exemplo: `g i t`

Exercício 3.6 ✪✪

Escreva um programa que, dado um conjunto de até max números inteiros positivos, calcule sua média, mediana e moda. O tamanho do conjunto, n, deve ser fornecido pelo usuário em tempo de execução. A mediana e a moda de um conjunto de n números são assim definidas:

- Se n for ímpar, a mediana é o número m tal que metade dos demais números é $\leq m$ e a outra metade é $\geq m$. Se n for par, a mediana é a média entre os números m_1 e m_2 tal que metade dos demais números é $\leq m_1$ e a outra metade é $\geq m_2$.
- A moda é o número que ocorre com maior frequência. Se mais de um número ocorre com a mesma frequência máxima, a moda não está definida. Seu programa deve indicar quando esse caso ocorrer.

Sugestão: Teste o programa com conjuntos de tamanho par e ímpar.

Exercício 3.7 ✪✪

Modifique o código do exercício 3.3 de modo que a função seja recursiva.

Exercício 3.8 ✪✪

Escreva a função `tipoDeMatriz` que, dada uma matriz quadrada de números inteiros, retorna o caractere

`V` se a matriz for vazia (contendo somente zeros em todas as posições),
`D` se a matriz for diagonal (contendo somente zeros acima e abaixo da diagonal principal),
`S` se a matriz for triangular superior (contendo somente zeros abaixo da diagonal principal),
`I` se a matriz for triangular inferior (contendo somente zeros acima da diagonal principal), ou
`X` em qualquer outro caso.

A função deve receber como parâmetros

- um ponteiro para o primeiro elemento da matriz e
- o tamanho da matriz, por exemplo, 3 para uma matriz quadrada 3 x 3.

Suponha que o tamanho da matriz é sempre maior que zero.

Exercício 3.9 ✪✪✪

Modifique o programa do exercício 3.5 para que aceite sequências desordenadas. Em nenhum momento ordene as sequências de entrada. Com esta modificação, o enunciado do exercício fica assim:

Considere duas sequências desordenadas A e B com até max letras de a a z, por exemplo:

A = `g z r i t m z i i`
B = `o o g t i g k i d`

Escreva um programa que, dadas as sequências A e B, gere quatro sequências desordenadas com

a) a união de A e B, com repetições. Exemplo: `g z r i t m z i i o o g t i g k i d`
b) a união de A e B, sem repetições. Exemplo: `g z r i t m o k d`
c) a interseção de A e B, com repetições. Exemplo: `g i t i`
d) a interseção de A e B, sem repetições. Exemplo: `g i t`

Exercício 3.10 ✪✪✪✪

Modifique o programa do exercício 3.9 para aceitar sequências desordenadas de qualquer coisa – inteiros, números decimais (`float` ou `double`), caracteres, estruturas, etc.

Sugestões:

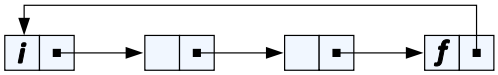
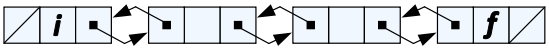
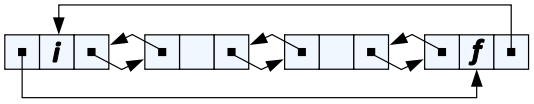
- Faça o cômputo da união e interseção em funções separadas.
- Use um `#define` para definir o tipo de dado contido nas sequências.
- Para determinar o tamanho desse tipo de dado, utilize a função `sizeof` da biblioteca de C.
- Para comparar valores de tipo e tamanho arbitrários, use a função `memcmp` da biblioteca de C.

4. Listas encadeadas

Exercício 4.1 ✪

Considere as quatro espécies de listas encadeadas descritas a seguir. Note que

- todas possuem um nó **inicial**, indicado com ***i***, e um nó **final**, indicado com ***f***
- os ponteiros de acesso às listas não estão representados nas figuras.

LS – Lista **linear simplesmente** encadeada • nós conectados por ponteiros numa única direção • encadeamento terminado com ponteiro NULL	
CS – Lista **circular simplesmente** encadeada • nós conectados por ponteiros numa única direção • com um ponteiro do nó final de volta ao inicial	
LD – Lista **linear duplamente** encadeada • nós conectados por ponteiros nas duas direções • encadeamentos terminados com ponteiros NULL	
CD – Lista **circular duplamente** encadeada • nós conectados por ponteiros nas duas direções • com um ponteiro do nó final de volta ao inicial • com um ponteiro do nó inicial de volta ao final	

Considere agora doze listas, três de cada espécie descrita acima, com essas características em comum:

- todas possuem o mesmo número n de nós
- cada nó armazena apenas um número inteiro, além do(s) ponteiro(s)
- em cada lista, nenhum número se repete
- toda lista está **ordenada**: ao percorrê-la do nó inicial ao final, os números estão em ordem crescente.

A diferença entre as listas está nos seus ponteiros de acesso:

- **LS*i*** é uma lista **linear simplesmente** encadeada com apenas um ponteiro para o nó ***inicial***
- **LS*f*** é uma lista **linear simplesmente** encadeada com apenas um ponteiro para o nó ***final***
- **LS*if*** é uma lista **linear simplesmente** encadeada com ponteiros para o nó ***inicial*** e para o nó ***final***

e assim sucessivamente, para as outras nove listas:

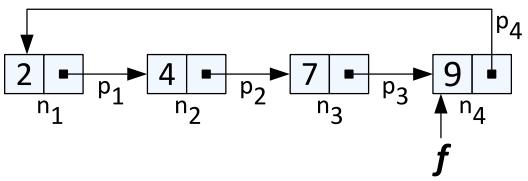
CS*i* , **CS*f*** , **CS*if*** , **LD*i*** , **LD*f*** , **LD*if*** , **CD*i*** , **CD*f*** , **CD*if***

Considere também as seguintes ações:

- **Visitar um nó:** obter o endereço do nó através de algum ponteiro de acesso.
- **Percorrer uma lista:** visitar cada nó ao menos uma vez, em qualquer direção.
- **Subir uma lista:** percorrer a lista em ordem crescente, desde o nó inicial até o final.
- **Descer uma lista:** percorrer a lista em ordem decrescente, desde o nó final o até o inicial.

Por exemplo, para **subir** a **CS*f*** a seguir é necessário visitar no mínimo **cinco** nós:

1. O nó $\mathbf{n_4}$ através do ponteiro ***f***
2. O nó $\mathbf{n_1}$ através do ponteiro $\mathbf{p_4}$
3. O nó $\mathbf{n_2}$ através do ponteiro $\mathbf{p_1}$
4. O nó $\mathbf{n_3}$ através do ponteiro $\mathbf{p_2}$
5. O nó $\mathbf{n_4}$ através do ponteiro $\mathbf{p_3}$



Note que não é possível **descer** a **CS*f*** acima.

Preencha a tabela a seguir. Todas as listas possuem n nós. Para cada lista, forneça o número mínimo de nós visitados ao percorrê-la na direção indicada. Se não for possível percorrê-la, escreva 0 (zero).

		LS*i*	LS*f*	LS*if*	CS*i*	CS*f*	CS*if*	LD*i*	LD*f*	LD*if*	CD*i*	CD*f*	CD*if*
Direção	Subir												
	Descer												

Exercício 4.2 ✪✪

Considere a função `TiraDoInicio` mostrada a seguir, que opera sobre uma lista linear simplesmente encadeada cuja estrutura é mostrada ao lado. O objetivo da função é remover o nodo inicial de uma lista não-vazia e retornar o dado nele contido. Suponha que `tDado` já foi definido.

```
tDado TiraDoInicio (struct nodo *lista) {
  struct nodo *inicio;
  tDado dadoNoInicio;
  inicio = lista;
  if (inicio == NULL)
    exit (-1);
  dadoNoInicio = inicio->dado;
  lista = inicio->prox;
  free (inicio);
  return (dadoNoInicio);
}
```

```
struct nodo {
    tDado dado;
    struct nodo *prox;
};
```

Essa função **não** realiza corretamente a operação esperada.

1) Por que?
2) Como corrigir o código para que a função realize corretamente a operação esperada?

Exercício 4.3 ✪✪

A função `removeDado` recebe dois parâmetros de entrada:

- um ponteiro de acesso a uma lista simplesmente encadeada e
- um dado que pode estar armazenado em um número arbitrário de nós da lista.

O objetivo da função é remover da lista encadeada todos os nós que possuem o dado especificado.

Considere a afirmação a seguir:

O ponteiro de acesso à lista deve ser (1) *porque a operação* (2).

Complete a afirmação indicando o que deve ser colocado no lugar de cada uma das posições marcadas (há apenas uma opção correta em cada posição).

(1)
A.( ) alocado dinamicamente
B.( ) acessado recursivamente
C.( ) um ponteiro duplo
D.( ) um ponteiro estático

(2)
A.( ) pode exigir que a lista seja percorrida mais de uma vez
B.( ) pode mudar o início da lista
C.( ) pode mudar o final da lista
D.( ) não pode ser realizada iterativamente

Exercício 4.4 ✪✪

Considere uma lista simplesmente encadeada com a mesma estrutura do exercício 4.2. Escreva a função `RemoveUltimo` que

- recebe um ponteiro para a lista L,
- remove o último nodo de L e
- não retorna nada.

A função deve funcionar corretamente em todos os casos, incluindo:

- lista vazia,
- lista com um único nodo,
- lista com dois nodos e
- lista com n nodos.

Exercício 4.5 ✪✪

Considere uma lista simplesmente encadeada com a mesma estrutura do exercício 4.2. Escreva a função `MoveParaInicio` que

- recebe um ponteiro para a lista L e o inteiro N,
- caso N se encontre em um ou mais nodos de L, move esses nodos para o início de L e
- retorna um inteiro indicando quantas vezes N ocorre em L.

Essa função deve manipular apenas ponteiros, ou seja, não deve criar nodos com `malloc` nem destruí-los com `free`. Além disso, a função deve funcionar corretamente em todos os casos, incluindo:

- lista vazia,
- lista com um nodo contendo N,
- lista com um nodo não contendo N,
- lista com dois nodos, nenhum contendo N,
- lista com dois nodos, apenas um contendo N,
- lista com dois nodos, ambos contendo N,
- lista com n nodos, nenhum contendo N,
- lista com n nodos, apenas um contendo N,
- lista com n nodos, alguns contendo N e outros não, e
- lista com n nodos, todos contendo N.

Exercício 4.6 ✪✪

Considere uma lista duplamente encadeada com a seguinte estrutura:

```
struct nodo {
  struct nodo* antes;
  char letra;
  struct nodo* depois;
};
```

Escreva a função `ehPalindromo` que

- recebe dois ponteiros para a lista L (um para o início e outro para o fim de L),
- determina se a palavra formada pelas letras em L é um palíndromo (veja o exercício 3.3) e
- retorna um booleano indicando se trata-se de um palíndromo; use o tipo `bool` e os valores `true` e `false` que estão definidos em `<stdbool>`.

A função deve ser iterativa (ou seja, não use recursão). Além disso, a função deve funcionar corretamente em todos os casos, incluindo:

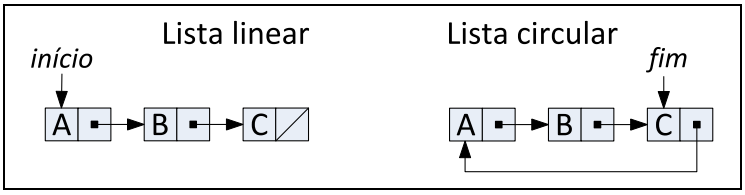
- lista vazia,
- lista com um nodo,
- lista com dois nodos contendo letras iguais,
- lista com dois nodos contendo letras diferentes,
- lista com três, quatro e n nodos contendo um palíndromo e
- lista com três, quatro e n nodos contendo um não-palíndromo.

Exercício 4.7 ✪✪

Implemente a versão recursiva da função `ehPalindromo` do exercício anterior.

Exercício 4.8 ✪✪✪

Uma lista circular simplesmente encadeada é uma lista linear onde o último nodo aponta de volta para o primeiro. A lista linear é acessada através de um ponteiro para o primeiro nodo; a circular, através de um ponteiro para o último nodo. Isso permite um acesso fácil a ambas as extremidades da lista circular.



Considere uma lista linear L e uma lista circular C, ambas simplesmente encadeadas e com a mesma estrutura do exercício 4.2.

1) Escreva a função `TornaCircular` que recebe um ponteiro para o primeiro nodo de uma lista linear L, transforma-a numa lista circular e retorna um ponteiro para seu último nodo.

2) Escreva a função `TornaLinear` que faz o oposto, ou seja, recebe um ponteiro para o último nodo de uma lista circular C, transforma-a numa lista linear e retorna um ponteiro para seu primeiro nodo.

3) Escreva a função `TamanhoCircular` que recebe um ponteiro para a lista circular C e retorna o número de nodos.

   As três funções acima devem funcionar corretamente em todos os casos, incluindo:
   - lista vazia,
   - lista com um nodo,
   - lista com dois nodos e
   - lista com n nodos.

4) Escreva a função `UneCirculares` que
   - recebe dois ponteiro para as listas circulares C e D,
   - une C e D em uma única lista circular,
   - coloca o resultado da união em C,
   - deixa D vazia e
   - não retorna nada.

A função acima deve executar em tempo constante, independente do tamanho de C e D. A função também deve funcionar corretamente em todos os casos, incluindo as 16 combinações ao lado.

		Nodos em D			
		0	1	2	n
Nodos em C	0				
	1				
	2				
	n				

Exercício 4.9 ✪✪✪

Considere uma lista simplesmente encadeada com a mesma estrutura do exercício 4.2, onde todos os inteiros são algarismos decimais, ou seja, estão entre 0 e 9 inclusive.

1) Escreva a função `ListaParaInt` que
   - recebe um ponteiro para a lista L,
   - interpreta os números em L como os dígitos de um número inteiro e
   - retorna o número resultante.

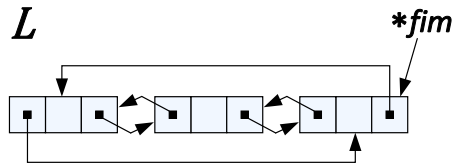
   Suponha que o dígito mais significativo esteja no início da lista e que o número final pode ser armazenado em um `int` sem causar *overflow*.

   A função deve funcionar corretamente em todos os casos, incluindo:
   - lista vazia (nesse caso a função deve retornar 0),
   - lista com um nodo,
   - lista com dois nodos e
   - lista com n nodos.

2) Escreva a função `IntParaLista` que faz o oposto da função acima, ou seja:
   - recebe um inteiro não negativo N do tipo `int`,
   - coloca cada um dos dígitos decimais de N em uma lista L e
   - retorna um ponteiro para o início de L.

Exercício 4.10 ✪✪✪

Considere uma lista circular L duplamente encadeada, como na figura abaixo.



Os nodos da lista possuem a seguinte estrutura:

```
struct nodo {
  struct nodo* antes;
  int dado;
  struct nodo* depois;
};
```

Considere também a função abaixo, onde `*fim` aponta para o último nodo de L. O que faz essa função?

```
void funcao (struct nodo **fim) {
  struct nodo *p;
  if (*fim == NULL) return;
  p = *fim->depois;
  p->antes = NULL;
  *fim->depois = NULL;
  *fim = p;
}
```

Exercício 4.11 ✪✪✪

A função ao lado opera sobre uma lista simplesmente encadeada com a mesma estrutura do exercício 4.2.

O que faz essa função?

```
void funcao (struct nodo** lista) {
  struct nodo* resultado = NULL;
  struct nodo* atual = *lista;
  struct nodo* prox;
  while (atual != NULL) {
    prox = atual->prox;
    atual->prox = resultado;
    resultado = atual;
    atual = prox;
  }
  *lista = resultado;
}
```

Exercício 4.12 ✪✪✪✪

A função recursiva ao lado opera sobre uma lista simplesmente encadeada com a mesma estrutura do exercício 4.2.

O que faz essa função?

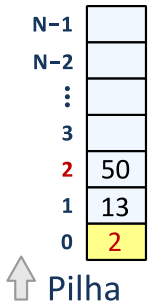
```
void funcao (struct nodo** lista) {
  struct nodo* primeiro;
  struct nodo* resto;
  if (*lista == NULL)
    return;
  primeiro = *lista;
  resto = primeiro->prox;
  if (resto == NULL)
    return;
  funcao (&resto);
  primeiro->prox->prox = primeiro;
  primeiro->prox = NULL;
  *lista = resto;
}
```

5. Pilhas, filas e deques

Exercício 5.1 ✪

Seja P uma pilha de inteiros não negativos ocupando um *array* A com N posições, como ilustra a figura ao lado, de modo que:

- números empilhados em P ocupam posições crescentes de A a partir de $A[1]$
- $A[0]$ contém o índice em A do número empilhado por último (topo da pilha)
- A pilha pode crescer enquanto houver posições livres em A



Implemente as funções de acesso às pilhas listadas abaixo.

```
void empilha    (int numero)
int  desempilha (void)
int  topo       (void)
int  tamanho    (void)
Bool vazia      (void)
Bool esvazia    (void)
Bool cheia      (void)
```

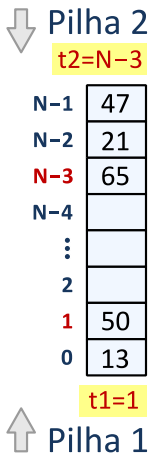
Note:

- A função `esvazia` pode apenas ajustar o topo da pilha para indicar que está vazia. Não é necessário "limpar" qualquer outra posição do *array*, mas se quiser fazê-lo, use um valor que não pode ser armazenado na pilha (p.ex. -1) para indicar que aquela posição do *array* está vazia.
- Onde necessário, use a função `void erro (char *)` que emite uma mensagem de erro e aborta o programa.

Exercício 5.2 ✪✪

Sejam P_1 e P_2 duas pilhas de inteiros não negativos ocupando um único *array* A com N posições, como ilustra a figura ao lado, de modo que:

- números empilhados em P_1 ocupam posições crescentes de A a partir de $A[0]$
- números empilhados em P_2 ocupam posições decrescentes de A a partir de $A[N-1]$
- `t1` contém o índice em A do número empilhado por último em P_1 (topo de P_1)
- `t2` contém o índice em A do número empilhado por último em P_2 (topo de P_2)
- as pilhas podem crescer enquanto houver posições livres em A



Implemente as funções de acesso às pilhas listadas abaixo.

```
void empilha    (int pilha, int numero)
int  desempilha (int pilha)
int  topo       (int pilha)
int  tamanho    (int pilha)
Bool vazia      (int pilha)
void esvazia    (void)
Bool cheias     (void)
```

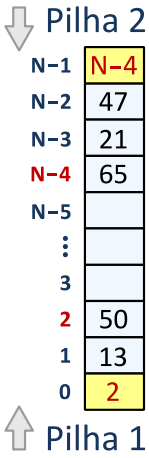
Note:

- Nas funções acima, `pilha` pode ser 1 ou 2.
- A função `esvazia` pode apenas ajustar o topo da pilha para indicar que está vazia. Não é necessário "limpar" qualquer outra posição do *array*, mas se quiser fazê-lo, use um valor que não pode ser armazenado na pilha (p.ex. -1) para indicar que aquela posição do *array* está vazia.
- Onde necessário, use a função `void erro (char *)` que emite uma mensagem de erro e aborta o programa.

Exercício 5.3 ✪✪

Repita o exercício anterior, com as seguintes modificações de projeto:

- números empilhados em P_1 ocupam posições crescentes de A a partir de $A[1]$
- números empilhados em P_2 ocupam posições decrescentes de A a partir de $A[N-2]$
- $A[0]$ contém o índice em A do número empilhado por último em P_1 (topo de P_1)
- $A[N-1]$ contém o índice em A do número empilhado por último em P_2 (topo de P_2)



Exercício 5.4 ✪✪

Escreva uma função `Bool ehPalindromo (void)` que determina se uma palavra é um palíndromo. As letras da palavra são obtidas com a função `char obtemLetra (void)`, que retorna uma letra de cada vez, enquanto houver letras sem serem lidas. Se não houver mais letras, uma chamada a `obtemLetra` causa erro. Para evitar esse erro, use a função `Bool acabaramAsLetras (void)`. Note que a palavra só pode ser lida uma vez; não é possível recomeçar do início da palavra.

Há três tipos de estruturas de dados disponíveis, mas apenas uma de cada tipo: uma ***pilha***, uma ***fila*** e um ***deque***. Todas armazenam caracteres. Você pode usar as estruturas em qualquer combinação: apenas uma das três, ou duas delas, ou todas as três. Para utilizá-las, chame as funções de acesso listadas abaixo. Implemente apenas a função `ehPalindromo` sem usar recursão. Suponha que todas as demais já existam. Suponha também que o tipo `Bool` já existe e que pode assumir os valores `TRUE` ou `FALSE`.

Pilha

```
void poePilha   (char c)
char tiraPilha  (void)
char topoPilha  (void)
int  tamPilha   (void)
Bool pilhaVazia (void)
```

Fila

```
void poeFila    (char c)
char tiraFila   (void)
char frenteFila (void)
int  tamFila    (void)
Bool filaVazia  (void)
```

Deque

```
void poeInicioDeque  (char c)
void poeFimDeque     (char c)
char tiraInicioDeque (void)
char tiraFimDeque    (void)
char inicioDeque     (void)
char fimDeque        (void)
int  tamDeque        (void)
Bool dequeVazio      (void)
```

Exercício 5.5 ✪✪

Um deque foi implementado num *array* circular A de tamanho T tal que objetos adicionados no final do deque ocupam posições crescentes no *array*, enquanto objetos adicionados no início ocupam posições decrescentes. Por exemplo:

	A	$inicio$	fim	**Deque**
Deque vazio	[• • • •]	-1	-1	Início → ← Fim
Após a operação `poeNoFim(A)`	[A • • •]	0	0	Início → A ← Fim
Após a operação `poeNoFim(B)`	[A B • •]	0	1	Início → A B ← Fim
Após a operação `poeNoInicio(C)`	[A B • C]	3	1	Início → C A B ← Fim
Após a operação `poeNoInicio(D)`	[A B D C]	2	1	Início → D C A B ← Fim
Após a operação `dado = tiraDoFim()`	[A • D C]	2	0	Início → D C A ← Fim
Após a operação `dado = tiraDoFim()`	[• • D C]	2	3	Início → D C ← Fim
Após a operação `dado = tiraDoFim()`	[• • D •]	2	2	Início → D ← Fim
Após a operação `dado = tiraDoFim()`	[• • • •]	-1	-1	Início → ← Fim

No exemplo acima:

- $T = 4$,
- o símbolo • indica posições vazias em A, e
- $inicio$ e fim são variáveis inteiras com as posições em A do início e fim do deque.

Indique se o método `tiraDoInicio()` pode ser escrito em pseudocódigo como:

se $inicio = -1$ então
  Erro deque vazio
fim_se
$dado \leftarrow A[inicio]$
incrementar $inicio$
se $inicio = T$ então
  $inicio \leftarrow 0$
fim_se
se $inicio = fim$ então
  $inicio \leftarrow fim \leftarrow -1$
fim_se
retornar dado
	se $fim = -1$ então
	  Erro deque vazio
	fim_se
	$dado \leftarrow A[inicio]$
	se $inicio = T - 1$ então
	  $inicio \leftarrow -1$
	fim_se
	incrementar $inicio$
	se $inicio = fim$ então
	  $inicio \leftarrow fim \leftarrow -1$
	fim_se
	retornar dado
		se $inicio = -1$ então
		  Erro deque vazio
		fim_se
		$dado \leftarrow A[inicio]$
		se $inicio = fim$ então
		  $inicio \leftarrow fim \leftarrow -1$
		senão
		  incrementar $inicio$
		  se $inicio = T$ então
		  $inicio \leftarrow 0$
		  fim-se
		fim_se
		retornar dado
			se $fim = -1$ então
			  Erro deque vazio
			fim_se
			$dado \leftarrow A[inicio]$
			se $inicio = fim$ então
			  $inicio \leftarrow fim \leftarrow -1$
			senão
			  se $inicio = T - 1$ então
			  $inicio \leftarrow -1$
			  fim_se
			  incrementar $inicio$
			fim_se
			retornar dado

( ) Sim ( ) Não	( ) Sim ( ) Não	( ) Sim ( ) Não	( ) Sim ( ) Não

Indique se o método `tiraDoFim()` pode ser escrito em pseudocódigo como:

se $fim = -1$ então
  Erro deque vazio
fim_se
$dado \leftarrow A[fim]$
se $inicio = fim$ então
  $inicio \leftarrow fim \leftarrow -1$
senão
  se $fim = 0$ então
  $fim \leftarrow T$
  fim_se
  decrementar fim
fim_se
retornar dado
	se $inicio = -1$ então
	  Erro deque vazio
	fim_se
	$dado \leftarrow A[fim]$
	se $inicio = fim$ então
	  $inicio \leftarrow fim \leftarrow -1$
	senão
	  decrementar fim
	  se $fim < 0$ então
	  $fim \leftarrow T - 1$
	  fim_se
	fim_se
	retornar dado
		se $inicio = -1$ então
		  Erro deque vazio
		fim_se
		$dado \leftarrow A[fim]$
		se $fim = 0$ então
		  $fim \leftarrow T$
		fim_se
		decrementar fim
		se $inicio = fim$ então
		  $inicio \leftarrow fim \leftarrow -1$
		fim_se
		retornar dado
			se $fim = -1$ então
			  Erro deque vazio
			fim_se
			$dado \leftarrow A[fim]$
			decrementar fim
			se $fim < 0$ então
			  $fim \leftarrow T - 1$
			fim_se
			se $inicio = fim$ então
			  $inicio \leftarrow fim \leftarrow -1$
			fim_se
			retornar dado

( ) Sim ( ) Não	( ) Sim ( ) Não	( ) Sim ( ) Não	( ) Sim ( ) Não

Exercício 5.6 ✪✪

Para cada afirmação a seguir, indique se é verdadeira (**V**) ou falsa (**F**). Em todos os casos, justifique.

( ) Ao implementar um deque através de uma lista encadeada com ponteiros para o início e fim da lista, consideramos duas opções: **a)** usar uma lista circular simplesmente encadeada, e **b)** usar uma lista linear duplamente encadeada. Dessas, a opção **b** é mais eficiente.

( ) Uma boa estrutura de dados para se implementar uma fila é um *array* circular com compactação após cada remoção.

( ) Se usarmos uma lista linear duplamente encadeada para implementar uma fila, tanto faz termos apenas um ponteiro para o início ou para o final da lista; em qualquer caso, inserções e remoções da fila levarão tempo $O(1)$.

Exercício 5.7 ✪✪✪

Implemente uma fila através de duas pilhas.

Pressupostos:

1) O tipo de dado a ser armazenado na fila, `tDado`, já está definido.
2) Todos os métodos de acesso às pilhas, listados na tabela abaixo, já estão implementados.
3) Esses métodos são capazes de manipular várias pilhas independentes ao mesmo tempo.
4) Esses métodos tomam o número da pilha como parâmetro.

Restrições:

1) Implemente apenas os métodos de acesso à fila listados na tabela abaixo.
2) Não declare nenhuma variável.
3) Use apenas os métodos de acesso às pilhas e os comandos de C que julgar convenientes.
4) Use apenas duas pilhas, numeradas 0 e 1.
5) Não use recursão.

Pilha
`void  poePilha   (int numPilha, tDado dado)`
`tDado tiraPilha  (int numPilha)`
`tDado topoPilha  (int numPilha)`
`int   tamPilha   (int numPilha)`
`Bool  pilhaVazia (int numPilha)`

Fila
`void  poeFila    (tDado dado)`
`tDado tiraFila   (void)`
`tDado frenteFila (void)`
`int   tamFila    (void)`
`Bool  filaVazia  (void)`

Exercício 5.8 ✪✪✪

Implemente um deque através de duas pilhas.

Pressupostos:

1) O tipo de dado a ser armazenado na fila, `tDado`, já está definido.
2) Todos os métodos de acesso às pilhas, listados na tabela abaixo, já estão implementados.
3) Esses métodos são capazes de manipular várias pilhas independentes ao mesmo tempo.
4) Esses métodos tomam o número da pilha como parâmetro.

Restrições:

1) Implemente apenas os métodos de acesso ao deque listados na tabela abaixo.
2) Não declare nenhuma variável.
3) Use apenas os métodos de acesso às pilhas e os comandos de C que julgar convenientes.
4) Use apenas duas pilhas, numeradas 0 e 1.
5) Não use recursão.

Pilha
`void  poePilha   (int numPilha, tDado dado)`
`tDado tiraPilha  (int numPilha)`
`tDado topoPilha  (int numPilha)`
`int   tamPilha   (int numPilha)`
`Bool  pilhaVazia (int numPilha)`

Deque
`void  poeInicioDeque  (tDado dado)`
`void  poeFimDeque     (tDado dado)`
`tDado tiraInicioDeque (void)`
`tDado tiraFimDeque    (void)`
`tDado inicioDeque     (void)`
`tDado fimDeque        (void)`
`int   tamDeque        (void)`
`Bool  dequeVazio      (void)`

Exercício 5.9 ✪✪✪

Implemente uma pilha através de uma fila.

Pressupostos:

1) O tipo de dado a ser armazenado na pilha, `tDado`, já está definido.
2) Todos os métodos de acesso à fila, listados na tabela abaixo, já estão implementados.

Restrições:

1) Implemente apenas os métodos de acesso à pilha listados na tabela abaixo.
2) O número total de variáveis usadas em todos os métodos não pode ser maior que três.
3) Use apenas os métodos de acesso à fila e os comandos de C que julgar convenientes.
4) Não use recursão.

Fila		
`void`	`poeFila`	`(tDado dado)`
`tDado`	`tiraFila`	`(void)`
`tDado`	`frenteFila`	`(void)`
`int`	`tamFila`	`(void)`
`Bool`	`filaVazia`	`(void)`

Pilha		
`void`	`poePilha`	`(tDado dado)`
`tDado`	`tiraPilha`	`(void)`
`tDado`	`topoPilha`	`(void)`
`int`	`tamPilha`	`(void)`
`Bool`	`pilhaVazia`	`(void)`

Exercício 5.10 ✪✪✪✪

Considere uma fila implementada em um vetor V não-circular de tamanho n, de modo que o primeiro item colocado na fila seja armazenado em $V[0]$, o próximo em $V[1]$ etc. Ao retirarmos um item da fila, sua frente passa a estar em $V[1]$; ao retirarmos outro item da fila, sua frente passa a estar em $V[2]$; e assim por diante. Isso cria espaços vazios no início do vetor. Pode-se evitar esse desperdício compactando o vetor – ou seja, deslocando a fila para baixo no vetor – de modo que a frente da fila volte a estar em $V[0]$. Essa compactação pode ser feita em momentos distintos:

1) após cada remoção da fila,
2) antes de cada inserção na fila, ou
3) antes de uma inserção quando o fim da fila já chegou ao final do vetor, ou seja, $V[n]$.

Para cada uma dessas estratégias:

1) Discuta suas vantagens e desvantagens.
2) Escreva os métodos de acesso à fila incorporando a respectiva estratégia de compactação.
3) Compare a complexidade dos métodos de acesso com uma implementação em *array* circular.

6. Complexidade

Exercício 6.1 ✪

Liste as limitações da análise experimental de complexidade de algoritmos.

Exercício 6.2 ✪

A análise teórica de complexidade de algoritmos utiliza os conceitos de melhor caso, caso médio e pior caso. Liste três motivos pelos quais a análise do caso médio pode ser problemática.

Exercício 6.3 ✪

Liste três limitações do modelo RAM de computação.

Exercício 6.4 ✪

Para cada uma das operações descritas abaixo, forneça o tempo de execução no pior caso. Use a notação O grande, por exemplo, $T(n) = \mathrm{O}(\log(n))$. Suponha que todas as operações básicas sejam realizadas em tempo constante.

1) Busca de um dado em *array* desordenado com n posições.
2) Busca binária de um dado em *array* ordenado com n posições.
3) Busca de um dado em uma matriz quadrada com n linhas e n colunas.
4) Cálculo de c^n usando apenas multiplicações.
5) Cálculo de c^n usando apenas adições.

Exercício 6.5 ✪✪

Coloque as funções abaixo em ordem crescente de complexidade assintótica. Atribua 1 para a função de menor complexidade e 8 para a função de maior complexidade.

A. ( ) $10n\,(\log n)^2$
B. ( ) n^3
C. ( ) $100\,(\log n)^3$
D. ( ) $n^2\,(\log n)^3$
E. ( ) $10^4 \log n$
F. ( ) $10^3 n^2$
G. ( ) $300n \log n$
H. ( ) 2^n

Exercício 6.6 ✪✪

Idem ao exercício anterior.

A. ( ) $10^4 n^2$
B. ( ) $10^{-2} \times 2^n$
C. ( ) $10^2 n\,(10^3 \log 10^3 n)^2$
D. ( ) $10^3 n \log 10^2 n$
E. ( ) $10n^3(10^2 \log 10^2 n)^3$
F. ( ) $10n^3$
G. ( ) $10^6(10^5 \log 10^4 n)^3$
H. ( ) $10^8 \log 10^9 n$

Exercício 6.7 ✪✪

Idem ao exercício anterior.

A. ( ) $100n^3$
B. ( ) $10^2 n \log 10^3 n$
C. ( ) $10^3 n^2$
D. ( ) $10^3 n \log(10^3 n^2)$
E. ( ) $10n^2(10^2 \log 10^2 n)^3$
F. ( ) 2^n
G. ( ) $10^7 \log 10^8 n$
H. ( ) $10n^3(10^2 \log 10^2 n)^2$
I. ( ) $10^6(10^5 \log 10^4 n)^3$
J. ( ) $10^2 n(10^3 \log 10^3 n)^2$

Exercício 6.8 ✪✪

Marque **V** ou **F** (verdadeiro ou falso), considerando que:

$f(n) = 5000n^2 + 800n \log n$
$g(n) = 90n^3 + 900n^2 + 9000n + 90000$
$h(n) = 4n^4 - 100n^2 - 3000$

01. ( ) $f(n) = O(g(n))$
02. ( ) $f(n) = \Omega(g(n))$
03. ( ) $f(n) = \theta(g(n))$
04. ( ) $g(n) = O(f(n))$
05. ( ) $g(n) = \Omega(f(n))$
06. ( ) $g(n) = \theta(f(n))$
07. ( ) $g(n) = O(h(n))$
08. ( ) $g(n) = \Omega(h(n))$
09. ( ) $g(n) = \theta(h(n))$
10. ( ) $h(n) = O(g(n))$
11. ( ) $h(n) = \Omega(g(n))$
12. ( ) $h(n) = \theta(g(n))$
13. ( ) $f(n) = O(h(n))$
14. ( ) $f(n) = \Omega(h(n))$
15. ( ) $f(n) = \theta(h(n))$
16. ( ) $h(n) = O(f(n))$
17. ( ) $h(n) = \Omega(f(n))$
18. ( ) $h(n) = \theta(f(n))$

Exercício 6.9 ✪✪

Marque **V** ou **F** (verdadeiro ou falso), considerando que:

$f(n) = n^3 \log n + 300n^2 \log n + 1000n \log n$
$g(n) = 10n^3 + 100n^3 \log n$
$h(n) = 500n^2 \log n - 700n^3 + 90n^3 \log n$

01. ( ) $f(n) = O(g(n))$
02. ( ) $f(n) = \Omega(g(n))$
03. ( ) $f(n) = \theta(g(n))$
04. ( ) $g(n) = O(f(n))$
05. ( ) $g(n) = \Omega(f(n))$
06. ( ) $g(n) = \theta(f(n))$
07. ( ) $g(n) = O(h(n))$
08. ( ) $g(n) = \Omega(h(n))$
09. ( ) $g(n) = \theta(h(n))$
10. ( ) $h(n) = O(g(n))$
11. ( ) $h(n) = \Omega(g(n))$
12. ( ) $h(n) = \theta(g(n))$
13. ( ) $f(n) = O(h(n))$
14. ( ) $f(n) = \Omega(h(n))$
15. ( ) $f(n) = \theta(h(n))$
16. ( ) $h(n) = O(f(n))$
17. ( ) $h(n) = \Omega(f(n))$
18. ( ) $h(n) = \theta(f(n))$

Exercício 6.10 ✪✪

Marque **V** ou **F** (verdadeiro ou falso):

01. ( ) Se $f(n) = \theta(g(n))$ então $f(n) = O(g(n))$
02. ( ) Se $f(n) = \theta(g(n))$ então $f(n) = \Omega(g(n))$
03. ( ) Se $f(n) = O(g(n))$ então $f(n) = \theta(g(n))$
04. ( ) Se $f(n) = \Omega(g(n))$ então $f(n) = \theta(g(n))$
05. ( ) Se $f(n) = O(g(n))$ então $g(n) = O(f(n))$
06. ( ) Se $f(n) = \Omega(g(n))$ então $g(n) = \Omega(f(n))$
07. ( ) Se $f(n) = O(g(n))$ e $g(n) = O(f(n))$ então $f(n) = \theta(g(n))$
08. ( ) Se $f(n) = O(g(n))$ e $g(n) = O(f(n))$ então $g(n) = \theta(f(n))$
09. ( ) Se $f(n) = O(g(n))$ e $g(n) = O(f(n))$ então $f(n) = \Omega(g(n))$
10. ( ) Se $f(n) = O(g(n))$ e $g(n) = O(f(n))$ então $g(n) = \Omega(f(n))$
11. ( ) Se $f(n) = O(h(n))$ e $h(n) = O(g(n))$ então $f(n) = O(g(n))$
12. ( ) Se $f(n) = O(h(n))$ e $g(n) = O(h(n))$ então $f(n) + g(n) = O(h(n))$
13. ( ) Se $f(n) = O(h(n))$ e $g(n) = O(h(n))$ então $f(n) \times g(n) = O(h(n))$
14. ( ) Se $f(n) = O(g(n))$ então $kf(n) = O(g(n))$ onde k é uma constante
15. ( ) Se $f(n) = O(g(n))$ no pior caso, então $f(n) = O(g(n))$ em qualquer caso
16. ( ) Se $f(n) = O(g(n))$ no melhor caso, então $f(n) = O(g(n))$ em qualquer caso
17. ( ) Se $f(n) = \theta(g(n))$ no pior caso, então $f(n) = \theta(g(n))$ em qualquer caso
18. ( ) Se $f(n) = \theta(g(n))$ no melhor caso, então $f(n) = \theta(g(n))$ em qualquer caso
19. ( ) Se $f(n) = \Omega(g(n))$ no pior caso, então $f(n) = \Omega(g(n))$ em qualquer caso
20. ( ) Se $f(n) = \Omega(g(n))$ no melhor caso, então $f(n) = \Omega(g(n))$ em qualquer caso

Exercício 6.11 ✪✪

Considere a definição de $O(g(n))$ a seguir:
$O(g(n)) = \{ f(n) : \exists$ constantes c e $n_0 > 0 \mid 0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0\}$
Considere também a função $f(n) = an + b$ onde a e b são constantes inteiras e $a > 0$.
Pergunta: $f(n)$ é $O(n^2)$? Prove que sua resposta está correta utilizando a definição de $O(g(n))$ acima.

Exercício 6.12 ✪✪

Para cada um dos trechos de código abaixo, forneça o tempo de execução no pior caso. Use a notação O grande, por exemplo, $T(n) = O(\log(n))$. Suponha que todas as variáveis são inteiras.

Trecho 1
```
for (i=0;i<n;i+=2)
  soma++;
```

Trecho 2
```
for (i=0;i<n;i++)
  for (j=0;j<n;j++)
    soma++;
```

Trecho 3
```
for (i=0;i<n;i++)
  soma++;
for (j=0;j<n;j++)
  soma++;
```

Trecho 4
```
for (i=0;i<n;i++)
  for (j=0;j<n*n;j++)
    soma++;
```

Trecho 5
```
for (i=0;i<n;i++)
  for (j=0;j<i;j++)
    soma++;
```

Trecho 6
```
for (i=0;i<n;i++)
  for (j=0;j<n*n;j++)
    for (k=0;k<j;k++)
      soma++;
```

Trecho 7
```
for (i=0;i<n/2;i++)
  for (j=0;j<i/2;j++)
    soma++;
```

Trecho 8
```
for (i=0;i<n;i++)
  for (j=0;j<LOG(i);j++)
    soma++;
```

Trecho 9
```
for (i=0;i<LOG(n);i++)
  soma++;
for (j=0;j<n;j++)
  soma++;
```

Trecho 10
```
for (i=0;i<LOG(n);i++)
  for (j=0;j<i;j+=2)
    soma++;
```

Trecho 11
```
for (i=0;i<n;i++)
  for (j=0;j<i*i;j++)
    for (k=0;k<LOG(j);k++)
      soma++;
```

Trecho 12
```
for (i=0;i<n;i++)
  for (j=0;j<LOG(i);j++)
    for (k=0;k<j;k++)
      soma++;
```

Exercício 6.13 ✪✪

Para cada estrutura de dados nas tabelas a seguir, forneça o tempo de execução no pior caso das operações de inserção (**Ins**) e remoção (**Rem**) de elementos. Use a notação O grande.

	Implementação	Variáveis disponíveis		Ins	Rem
Pilha	Array	Índice do topo da pilha	1		
	Lista linear simplesmente encadeada	Ponteiro p/ topo da pilha (último nodo)	2		
		Ponteiro p/ base da pilha (1º nodo)	3		
	Lista linear duplamente encadeada	Ponteiro p/ topo da pilha (último nodo)	4		
		Ponteiro p/ base da pilha (1º nodo)	5		
		Ponteiros p/ topo e base da pilha (1º e último nodos)	6		
	Lista circular simplesmente encadeada	Ponteiro p/ topo da pilha (último nodo)	7		
		Ponteiro p/ base da pilha (1º nodo)	8		
	Lista circular duplamente encadeada	Ponteiro p/ topo da pilha (último nodo)	9		
		Ponteiro p/ base da pilha (1º nodo)	10		
		Ponteiros p/ topo e base da pilha (1º e último nodos)	11		
Fila	*Array* sem compactação	Índices do início e fim da fila	12		
	Array c/ compactação após cada remoção	Índice do fim da fila	13		
	Array circular	Índices do início e fim da fila	14		
	Lista linear simplesmente encadeada	Ponteiro p/ início da fila (1º nodo)	15		
		Ponteiros p/ início e fim da fila (1º e último nodos)	16		
	Lista linear duplamente encadeada	Ponteiro p/ início da fila (1º nodo)	17		
		Ponteiro p/ fim da fila (último nodo)	18		
		Ponteiros p/ início e fim da fila (1º e último nodos)	19		
	Lista circular simplesmente encadeada	Ponteiro p/ início da fila (1º nodo)	20		
		Ponteiro p/ fim da fila (último nodo)	21		
		Ponteiros p/ início e fim da fila (1º e último nodos)	22		
	Lista circular duplamente encadeada	Ponteiro p/ início da fila (1º nodo)	23		
		Ponteiro p/ fim da fila (último nodo)	24		
		Ponteiros p/ início e fim da fila (1º e último nodos)	25		
	2 pilhas implementadas por *arrays*	Índice do topo de cada pilha	26	*Precedida por inserção*	
			27	*Precedida por remoção*	

Exercício 6.14 ✪✪✪

Considere o **Bubble Sort** mostrado abaixo. Forneça expressões, em função de n, para o número de operações primitivas em cada linha do pseudocódigo e calcule o total, para o melhor e o pior caso.

Algoritmo: Bubble Sort (A, n)
Entrada: *array* A com n elementos
Saída: *array* A em ordem crescente

```
1) para i de 0 até n-1 passo 1 faça
2)   para j de n-1 até i+1 passo -1 faça
3)     se A[j-1] > A[j] então
4)       temp = A[j-1]
5)       A[j-1] = A[j]
6)       A[j] = temp
```

NOTA: *A identação mostra o escopo dos comandos:*
- *o* `para` *na linha 2 está dentro do* `para` *na linha 1*
- *o* `se` *na linha 3 está dentro do* `para` *na linha 2*
- *as linhas 4 a 6 estão dentro do* `se` *na linha 3*

Linha	Melhor caso	Pior caso
1		
2		
3		
4		
5		
6		
Total		

Exercício 6.15 ✪✪✪

Idem para o **Selection Sort**.

Algoritmo: Selection Sort (A, n)
Entrada: *array* A com n elementos
Saída: *array* A em ordem crescente

```
1) para i de 0 até n-2 faça
2)   para j de i+1 até n-1 faça
3)     se A[j] < A[i] então
4)       temp = A[i]
5)       A[i] = A[j]
6)       A[t] = temp
```

NOTA: *A identação mostra o escopo dos comandos:*
- `para` *na linha 2: dentro do* `para` *na linha 1*
- `se` *na linha 3: dentro do* `para` *na linha 2*
- *linhas 4 e 5: dentro do* `se` *na linha 3*

Linha	Melhor caso	Pior caso
1		
2		
3		
4		
5		
6		
Total		

Exercício 6.16 ✪✪✪

Idem para o **Insertion Sort**.

Algoritmo: Insertion Sort (A, n)
Entrada: *array* A com n elementos
Saída: *array* A em ordem crescente

```
1) para i de 1 até n-1 faça
2)   atual = A[i]
3)   j = i-1
4)   enquanto j>=0 e A[j]>atual faça
5)     A[j+1] = A[j]
6)     j = j-1
7)   A[j+1] = atual
```

NOTA: *A identação mostra o escopo dos comandos:*
- *linhas 2 a 7: dentro do* `para` *na linha 1*
- *linhas 5 e 6: dentro do* `enquanto` *na linha 4*

Linha	Melhor caso	Pior caso
1		
2		
3		
4		
5		
6		
7		
Total		

Exercício 6.17 ✪✪✪

Idem para o **Stupid Sort**.

Algoritmo: Stupid Sort (A, n)
Entrada: *array* A com n inteiros não negativos
Saída: *array* A em ordem crescente

```
 1) para iB de 0 até n excl faça
 2)   para iA de 0 até n excl faça
 3)     se A[iA] != -1 então
 4)       menor = A[iA]
 5)       saia do para na linha 2
 6)   para iA de 0 até n excl faça
 7)     se A[iA]!=-1 e A[iA]<menor então
 8)       menor = A[iA]
 9)   B[iB] = menor
10)   para iA de 0 até n excl faça
11)     se A[iA]=menor então
12)       A[iA] = -1
13)       saia do para na linha 10
14) para iA de 0 até n excl faça
15)   A[iA] = B[iA]
```

NOTA 1: *A identação mostra o escopo dos comandos:*
- *o* `para` *na linha 2 está dentro do* `para` *na linha 1*
- *o* `se` *na linha 3 está dentro do* `para` *na linha 2*
- *as linhas 4 e 5 estão dentro do* `se` *na linha 3*

NOTA 2: *a semântica do* `para...excl` *é igual à de C:*
`para iB de 0 até n excl faça` *equivale a*
`for (iB=0; iB<n; iB++)`

Linha	Melhor caso	Pior caso
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		
Total		

Exercício 6.18 ✪✪✪

Idem para o **Less Stupid Sort**.

Algoritmo: Less Stupid Sort (A, n)
Entrada: *array* A com n inteiros não negativos
Saída: *array* A em ordem crescente

```
 1) para iB de 0 até n excl faça
 2)   para iA de 0 até n excl faça
 3)     se A[iA] != -1 então
 4)       menor = A[iA]
 5)       iMenor = iA
 6)       saia do para na linha 2
 7)   para iA de 0 até n excl faça
 8)     se A[iA]!=-1 e A[iA]<menor então
 9)       menor = A[iA]
10)       iMenor = iA
11)   B[iB] = menor
12)   A[iMenor] = -1
13) para iA de 0 até n excl faça
14)   A[iA] = B[iA]
```

NOTA 1: *A identação mostra o escopo dos comandos:*
- *o* `para` *na linha 2 está dentro do* `para` *na linha 1*
- *o* `se` *na linha 3 está dentro do* `para` *na linha 2*
- *as linhas 4 e 5 estão dentro do* `se` *na linha 3*

NOTA 2: *a semântica do* `para...excl` *é igual à de C:*
`para iB de 0 até n excl faça` *equivale a*
`for (iB=0; iB<n; iB++)`

Linha	Melhor caso	Pior caso
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
Total		

Exercício 6.19 ✪✪✪

Considere o algoritmo **Transpor** mostrado abaixo. Forneça expressões, em função de n, para o número de operações primitivas em cada linha do pseudocódigo e calcule o total, para o melhor e o pior caso. **Importante:** A indexação de um elemento da matriz (p.ex. `M[i,j]`) deve contar como uma única operação básica, ainda que seja feita através de dois índices.

Algoritmo: Transpor (M, n)
Entrada: Matriz M com n linhas e n colunas
Saída: Matriz M transposta

```
1) para i de 1 até n-1 faça
2)   para j de 0 até i-1 faça
3)     se M[i,j] != M[j,i]
4)       temp = M[i,j]
5)       M[i,j] = M[j,i]
6)       M[j,i] = temp
```

NOTA: *A identação mostra o escopo dos comandos:*
- `para` *na linha 2: dentro do* `para` *na linha 1*
- `se` *na linha 3: dentro do* `para` *na linha 2*
- *linhas 4 a 6: dentro do* `se` *na linha 3*

Linha	Melhor caso	Pior caso
1		
2		
3		
4		
5		
6		
Total		

Exemplo: $M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$. Após `Transpor`, $M = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$.

Exercício 6.20 ✪✪✪

O algoritmo **Girar** a seguir gira uma matriz quadrada $\boldsymbol{n \times n}$ 90° no sentido horário. Forneça expressões em função de n para o número de operações primitivas em cada linha do pseudocódigo e calcule o total, tanto para o melhor quanto para o pior caso. Suponha que n é sempre par.
Importante: A indexação de um elemento da matriz (p.ex. `M[i,j]`) deve contar como uma única operação básica, ainda que seja feita através de dois índices.

Algoritmo: Girar (M, n)
Entrada: Matriz M com n linhas e n colunas (n é par)
Saída: Matriz M girada 90° no sentido horário

```
1) para i de 0 até (n-1)/2 faça
2)   para j de i até n-2-i faça
3)     temp            = M[n-1-j,i]
4)     M[n-1-j,i]      = M[n-1-i,n-1-j]
5)     M[n-1-i,n-1-j]  = M[j,n-1-i]
6)     M[j,n-1-i]      = M[i,j]
7)     M[i,j]          = temp
```

NOTA: *A identação mostra o escopo dos comandos:*
- `para` *na linha 2: dentro do* `para` *na linha 1*
- *linhas 3 a 7: dentro do* `para` *na linha 2*

Linha	Melhor caso	Pior caso
1		
2		
3		
4		
5		
6		
7		
Total		

Exemplo: $M = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$. Após `Girar`, $M = \begin{bmatrix} 7 & 4 & 1 \\ 8 & 5 & 2 \\ 9 & 6 & 3 \end{bmatrix}$.

Exercício 6.21 ✪✪✪

Existem dois algoritmos para resolver certo problema, cujos tempos de execução são dados por:

Algoritmo 1	Algoritmo 2
$T_1(n) = n^2 - 100n + 4000$	$T_2(n) = 20n + 2000$

onde n é o tamanho da entrada. Responda e justifique: é possível afirmar que um dos algoritmos executa sempre mais rápido que o outro para entradas...

1) de qualquer tamanho?
2) de tamanho entre 10 e 30? Se for possível, qual?
3) de tamanho igual a 50? Se for possível, qual?
4) de tamanho entre 90 e 200? Se for possível, qual?
5) de tamanho acima de 500? Se for possível, qual?

Exercício 6.22 ✪✪✪

Existem dois algoritmos para resolver certo problema, cujos tempos de execução são dados por:

Algoritmo 1	Algoritmo 2
$T_1(n) = n^2 - 600n + 150000$	$T_2(n) = 2n^2 - 1200n + 200000$

onde n é o tamanho da entrada. Responda e justifique: para que tamanhos da entrada...

1) o algoritmo 1 executa mais rápido que o algoritmo 2?
2) o algoritmo 2 executa mais rápido que o algoritmo 1?
3) os dois algoritmos executam em tempos iguais?

Sugestão: Solucione as equações e, a partir daí, esboce os gráficos de $T_1(n)$ e $T_2(n)$.

Exercício 6.23 ✪✪✪

Existem dois algoritmos para resolver certo problema, cujos tempos de execução são dados por:

Algoritmo 1	Algoritmo 2
$T_1(n) = n^3 - 59n^2 + 1060n - 4700$	$T_2(n) = n^2 - 40n + 1300$

onde n é o tamanho da entrada. Responda e justifique: para que tamanhos da entrada...

1) o algoritmo 1 executa mais rápido que o algoritmo 2?
2) o algoritmo 2 executa mais rápido que o algoritmo 1?
3) os dois algoritmos executam em tempos iguais?

Dica: Uma equação cúbica com três raízes reais a, b e c pode ser escrita na forma

$$n^3 - (a+b+c)n^2 + (ab+bc+ac)n - (abc) = 0$$

Exercício 6.24 ✪✪✪

Considere a definição de $\theta(g(n))$ seguir:

$\theta(g(n)) = \{ f(n) : \exists$ constantes c_1, c_2 e $n_0 > 0 \mid 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0\}$

Use essa definição para mostrar que $2n^2 - {}^{n}/_{2} = \theta(n^2)$.

Exercício 6.25 ✪✪✪✪

Considere a definição de $\theta(g(n))$ no exercício anterior e a função $f(n) = 10n^3 - 300n^2 + 2000n$. Use o método analítico discutido em aula para responder as seguintes perguntas:

1) Qual o menor valor inteiro de n_0 que permite provar que $f(n) = \theta(n^3)$?
2) Qual o maior valor inteiro de c_1 que permite provar que $f(n) = \theta(n^3)$?
3) Se usarmos $c_1 = 5$ e $c_2 = 20$, qual o menor valor inteiro de n_0 que permite provar que $f(n) = \theta(n^3)$?

Dados: $\sqrt{5} = 2{,}24$ e $\sqrt{17} = 4{,}12$.

Exercício 6.26 ✪✪✪✪

Considere a definição de $\theta(g(n))$ no exercício 6.18 e a função $f(n) = 8n^3 - 300n^2 + 40n$. Use as informações nos gráficos 1 e 2 a seguir para responder as perguntas abaixo. Forneça suas respostas na forma de intervalos, por exemplo $(0, 10]$. Justifique as respostas de 1 a 6.

1) Qual a faixa de valores de n_0 que satisfazem a definição de θ para quaisquer c_1 e $c_2 > 0$?
2) Qual a faixa de valores de c_1 que satisfazem a definição de θ?
3) Qual a faixa de valores de c_2 que satisfazem a definição de θ?

Considere agora que c_1, c_2, n e n_0 são **inteiros** positivos (maiores que zero).

4) Qual a faixa de valores de n_0 que satisfazem a definição de θ para quaisquer c_1 e $c_2 > 0$?
5) Qual a faixa de valores de c_1 que satisfazem a definição de θ?
6) Qual a faixa de valores de c_2 que satisfazem a definição de θ?
7) Use a definição de θ para mostrar que $f(n) = \theta(n^3)$.
8) Use os resultados anteriores para mostrar graficamente que $f(n) = \theta(n^3)$.

Gráfico 1

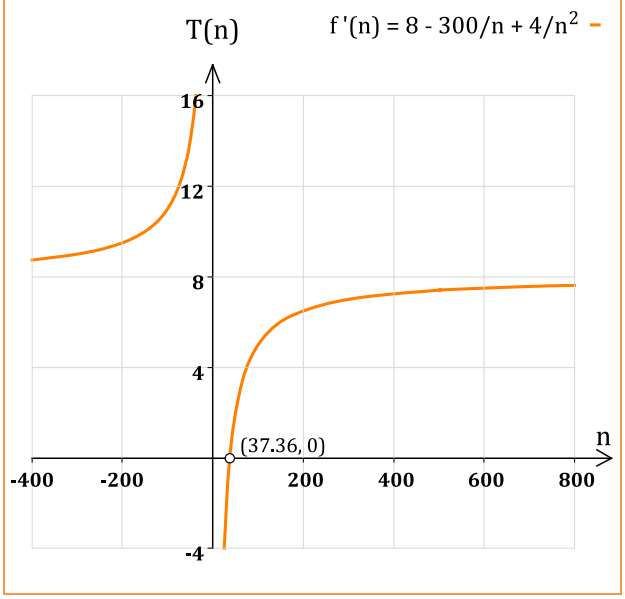
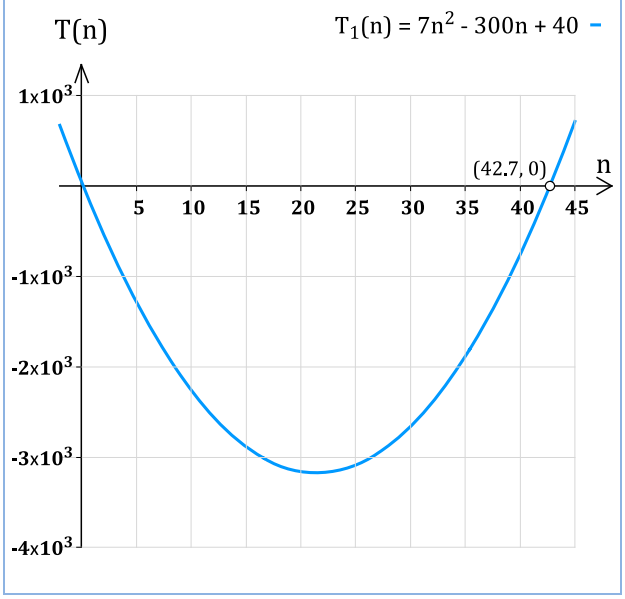


Gráfico 2



7. Ordenação

Exercício 7.1 ✪

O que é um algoritmo de ordenação estável? O que é um algoritmo de ordenação *in place*?

Exercício 7.2 ✪

Caracterize o melhor e o pior caso do Quick Sort.

Exercício 7.3 ✪✪

Diferencie o melhor e o pior caso do Merge Sort e justifique.

Exercício 7.4 ✪✪

1) O tempo de execução do Merge Sort é igual para o melhor caso e o pior caso? Justifique.
2) O tempo de execução do Quick Sort é igual para o melhor caso e o pior caso? Justifique.

Exercício 7.5 ✪✪

Descreva os principais métodos de escolha do pivô no Quick Sort. Para cada um, comente sobre seu impacto na eficiência da ordenação.

Exercício 7.6 ✪✪

Complete a tabela abaixo, fornecendo a complexidade assintótica dos vários algoritmos de ordenação.

Algoritmo	Tempo		
	Pior caso	Caso médio	Melhor caso
Bubble sort			
Bubble sort modificado			
Selection sort			
Insertion sort			
Heap sort			
Merge sort			
Quicksort			

Exercício 7.7 ✪✪✪

Verdadeiro ou falso:

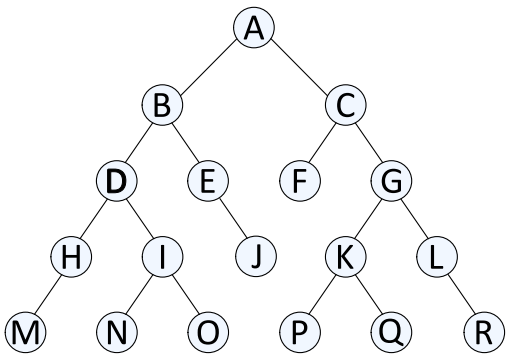
01. ( ) Todos os algoritmos de ordenação funcionam através de comparações entre chaves.
02. ( ) Os algoritmos de ordenação estáveis incluem o Merge Sort, o Quick Sort e o Bubble Sort.
03. ( ) Heap Sort pode ser feito *in place*, mas é mais simples e intuitivo se usa um *heap* separado.
04. ( ) O algoritmo de ordenação mais eficiente executa em tempo $O(n \log(n))$.
05. ( ) Selection Sort é uma variação do FP-Sort (FP = Fila de Prioridades) onde a fila de prioridades é implementada com uma sequência ordenada.
06. ( ) O Merge Sort pode ser implementado tanto de forma recursiva como de forma iterativa.
07. ( ) O Merge Sort, o Heap Sort e o Insertion Sort executam em tempo $O(n \log(n))$.
08. ( ) O tempo de execução do Merge Sort é igual para o melhor caso e o pior caso.
09. ( ) O tempo de execução do Quick Sort é igual para o melhor caso e o pior caso.

8. Árvores

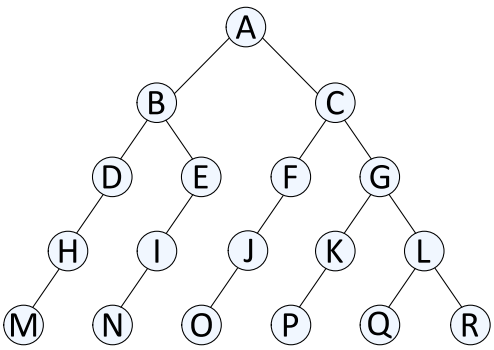
Exercício 8.1 ✪

Considere as duas árvores binárias a seguir.

Árvore 1



Árvore 2



1) Escreva a sequência em que os nós da árvore 1 são visitados em um caminhamento prefixado.
2) Escreva a sequência em que os nós da árvore 1 são visitados em um caminhamento infixado.
3) Escreva a sequência em que os nós da árvore 1 são visitados em um caminhamento posfixado.
4) Escreva a sequência em que os nós da árvore 2 são visitados em um caminhamento prefixado.
5) Escreva a sequência em que os nós da árvore 2 são visitados em um caminhamento infixado.
6) Escreva a sequência em que os nós da árvore 2 são visitados em um caminhamento posfixado.

Exercício 8.2 ✪✪

1) O que é uma árvore binária própria?
2) O que é uma árvore binária completa?
3) O que é uma árvore binária quase completa?
4) No caminhamento de Euler sobre uma árvore binária, quantas vezes os nós são visitados? Explique.
5) Qual o tempo de execução do caminhamento de Euler sobre uma árvore binária própria com n nós e altura h? Justifique.

Exercício 8.3 ✪✪

Desenhe a árvore binária correspondente à expressão prefixada abaixo. Todos os operadores são binários.

```
* + 5 ÷ 4 − 3 1 − * 8 ÷ 6 2 7
```

Exercício 8.4 ✪✪

Desenhe a árvore binária correspondente à expressão infixada abaixo. Todos os operadores são binários.

```
( 8 − ( ( 2 + 7 ) ÷ 4 ) ) + ( ( ( 5 ÷ 6 ) * 3 ) − 1 )
```

Exercício 8.5 ✪✪

Desenhe a árvore binária correspondente à expressão posfixada abaixo. Todos os operadores são binários.

```
5 7 ÷ 3 * 1 − 8 2 4 6 + ÷ * +
```

Exercício 8.6 ✪✪✪

O caminhamento prefixado de uma árvore binária visita seus nós na seguinte ordem:

```
A B D H M N E I O C F J K P G L Q R
```

O caminhamento infixado da mesma árvore binária visita seus nós na seguinte ordem:

```
M H N D B I O E A J F P K C G Q L R
```

O caminhamento posfixado da mesma árvore binária visita seus nós na seguinte ordem:

```
M N H D O I E B J P K F Q R L G C A
```

Desenhe a árvore.

Exercício 8.7 ✪✪✪

O caminhamento prefixado de uma árvore binária visita seus nós na seguinte ordem:

```
A B D H M N I O E J P C F K Q G L R
```

O caminhamento infixado da mesma árvore binária visita seus nós na seguinte ordem:

```
M H N D I O B E J P A F K Q C G L R
```

O caminhamento posfixado da mesma árvore binária visita seus nós na seguinte ordem:

```
M N H O I D P J E B Q K F R L G C A
```

Desenhe a árvore.

Exercício 8.8 ✪✪✪

Considere a expressão prefixada a seguir, onde todos os operadores são binários:

```
+ * − T ÷ L G K ÷ * X J + − C Q A
```

1) Escreva a expressão em notação infixada, com parênteses.
2) Escreva a expressão em notação posfixada, sem parênteses.

Exercício 8.9 ✪✪✪

Considere a expressão infixada a seguir, onde todos os operadores são binários:

```
( ( ( E − C ) ÷ H ) + ( A * ( ( G ÷ D ) − F ) ) ) * B
```

1) Escreva a expressão em notação prefixada, sem parênteses.
2) Escreva a expressão em notação posfixada, sem parênteses.

Exercício 8.10 ✪✪✪

Considere a expressão posfixada a seguir, onde todos os operadores são binários:

```
8 1 4 9 7 * − * + 3 8 − 6 + 5 ÷ 2 − ÷
```

1) Escreva a expressão em notação prefixada, sem parênteses.
2) Escreva a expressão em notação infixada, com parênteses.

Exercício 8.11 ✪✪✪

Verdadeiro ou falso:

01. ( ) Todo *heap* é uma árvore binária própria.
02. ( ) Toda árvore binária própria é um *heap*.
03. ( ) Toda árvore binária quase completa é um *heap*.
04. ( ) Todo *heap* é uma árvore binária quase completa.
05. ( ) Todo *heap* é uma árvore binária de busca.
06. ( ) Toda árvore binária de busca é um *heap*.
07. ( ) Nem toda árvore binária de busca é uma árvore binária própria.
08. ( ) Toda árvore binária de busca é uma árvore binária quase completa.
09. ( ) Nem toda árvore binária quase completa é uma árvore binária própria.
10. ( ) Toda árvore binária própria é uma árvore binária quase completa.

Exercício 8.12 ✪✪✪

Seja:

- G uma árvore binária genérica não vazia
- e_g o número de nodos externos de G
- i_g o número de nodos internos de G
- n_g o número total de nodos de G
- h_g a altura de G

- P uma árvore binária própria não vazia
- e_p o número de nodos externos de P
- i_p o número de nodos internos de P
- n_p o número total de nodos de P
- h_p a altura de P

Verdadeiro ou falso:

01. ( ) $2h_p + 1 \leq n_p$
02. ( ) $n_p \leq 2^{h_p+1} - 1$
03. ( ) $h_p + 1 \leq e_p$
04. ( ) $e_p \leq 2^{h_p}$
05. ( ) $h_p \leq i_p \leq 2^{h_p} - 1$
06. ( ) $n_g = 2e_g - 1$
07. ( ) $1 \leq e_g \leq 2^{h_g}$
08. ( ) $h_g \leq i_g \leq 2^{h_g} - 1$

Exercício 8.13 ✪✪✪

Considere uma árvore A com raiz r e um total de n nós. Define-se o **ancestral comum mais próximo** entre dois nós n e m ou $\mathrm{ACMP}(n, m)$ como o nó mais baixo de A (ou seja, o nó mais distante de r) que tem ambos n e m como descendentes. Um nó pode ser considerado descendente de si mesmo.

Escreva uma função em C que recebe ponteiros para dois nós como entrada e retorna um ponteiro para o ACMP entre eles. Suponha que a função `pai` já exista. Seu protótipo está a seguir, bem como o protótipo da função `ACMP` que você deve escrever.

```
/* ENTRADA: ponteiro para um nó    */
/* SAÍDA: ponteiro para seu pai,   */
/*        ou NULL se o nó é a raiz */

pNo *pai (tNo *pNo);
```

```
/* ENTRADA: ponteiros para 2 nós  */
/* SAÍDA: ponteiro p/ o ancestral */
/*        comum mais próximo      */

pNo *ACMP (tNo *pN1, tNo *pN2);
```

Exercício 8.14 ✪✪✪

A figura a seguir mostra vários vetores, cada um implementando uma árvore binária diferente.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1		6	7	9	15	10	18	11	13	17	21	19						
2		3	8	6	11	19	16	10	12	15	20	17	13	18			23	24
3		8	16	14	22	18	20	32	34	24	36	40	28					
4		30	14	46	10	21	38				16	26	35	41				
5		4	8	7	11	9	10	16	17	12	18	20	14	13			19	21
6		60	28	92		42	76				24	52	62	88				
7		4	8	7	11	9	10	16	17	12	18	20	14	13	19	21		
8		30	14	46		21	38				15	26	31	44				

Para cada vetor, indique, com **Sim** ou **Não**, se a árvore correspondente é

- Uma árvore binária própria.
- Uma árvore binária completa.
- Uma árvore binária quase completa.
- Uma árvore de busca binária.

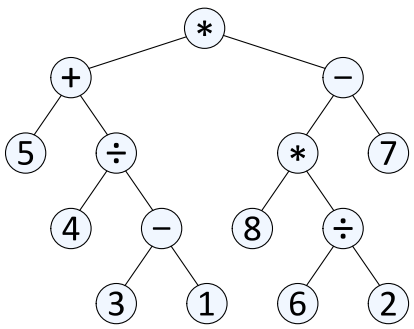
Vetor	Árvore binária...			
	própria?	completa?	quase completa?	de busca?
1				
2				
3				
4				
5				
6				
7				
8				

Exercício 8.15 ✪✪✪

Descreva um método usando o caminhamento de Euler para calcular o número de descendentes de cada nó em uma árvore binária. É possível armazenar informações nos nós, se necessário. Qual o tempo de execução desse algoritmo?

Exercício 8.16 ✪✪✪

Descreva um método usando o caminhamento de Euler para imprimir uma expressão aritmética infixada com parênteses a partir de sua árvore.

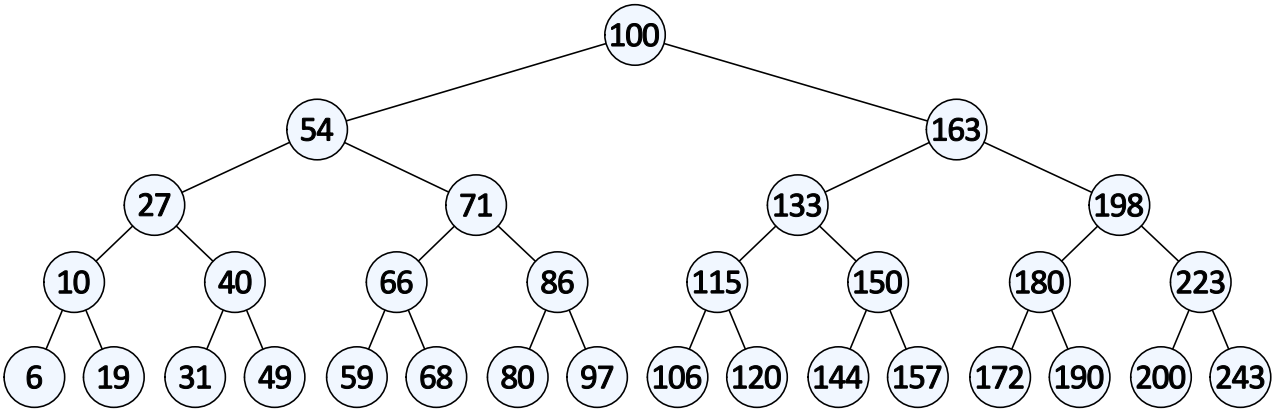


Por exemplo, se o método for aplicado à árvore ao lado, o resultado deve ser a impressão da expressão

```
( 5 + ( 4 ÷ ( 3 − 1 ) ) ) * ( ( 8 * ( 6 ÷ 2 ) ) − 7 )
```

Exercício 8.17 ✪✪✪

Considere a árvore de busca binária na figura a seguir.



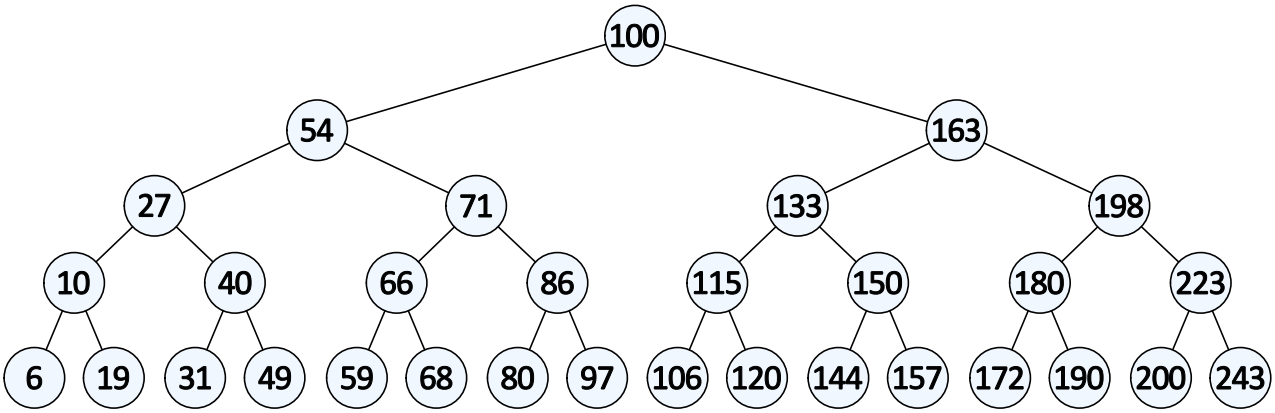
Mostre o conteúdo da árvore após a execução das seguintes operações:

```
remove (198); remove (190); remove (200); insere (198).
```

Certifique-se que em momento algum a altura da árvore seja superior a 4.

Exercício 8.18 ✪✪✪

Considere a árvore de busca binária na figura a seguir.



Mostre o conteúdo da árvore após a execução das seguintes operações:

```
remove (71); remove (68); remove (80); insere (71).
```

Certifique-se que em momento algum a altura da árvore seja superior a 4.

Exercício 8.19 ✪✪✪✪

Considere uma árvore binária ordenada A cujos nodos possuem a seguinte estrutura:

```
struct nodo {
  int dado;
  struct nodo* menor;
  struct nodo* maior;
};
```

Escreva a função recursiva `arvoreParaLista` que

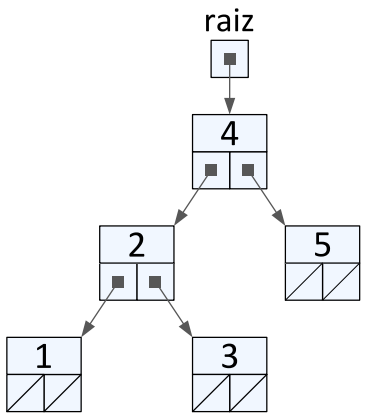
- recebe um ponteiro para a raiz de uma árvore A,
- reorganiza os ponteiros internos da árvore de modo a transformá-la numa lista circular duplamente encadeada, com os dados organizados em ordem crescente,
- retorna um ponteiro para o primeiro nodo da lista (o nodo com do dado de menor valor), e
- executa em tempo $O(n)$, operando uma única vez sobre cada nodo.

A função deve funcionar corretamente em todos os casos, incluindo:

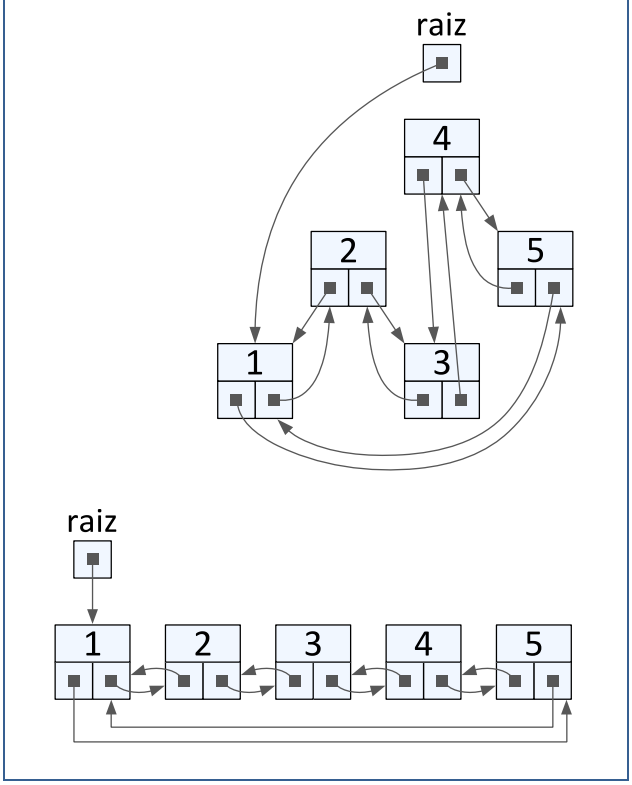
- árvore vazia,
- árvore com um único nodo,
- árvore com dois nodos, e
- árvore com n nodos.

As figuras a seguir mostram uma árvore antes e depois de ser transformada numa lista.

Antes



Depois



9. Heaps

Exercício 9.1 ✪

Um *heap* é uma árvore binária que armazena uma coleção de chaves em seus nodos e que satisfaz duas propriedades adicionais: uma relacional e outra estrutural. Explique essas propriedades.

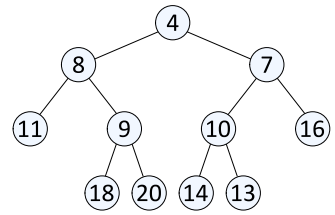
Exercício 9.2 ✪✪

Para cada uma das árvores abaixo, indique se ela é um *heap* ou, caso contrário, qual propriedade é violada. Use a seguinte marcação nas suas respostas:

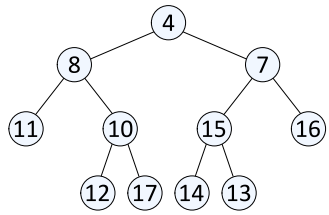
- **H *n*** a árvore não viola nenhuma propriedade, portanto é um *heap*; o último nó tem chave ***n***.
- **R** a árvore viola a propriedade relacional, portanto não é um *heap*.
- **E** a árvore viola a propriedade estrutural, portanto não é um *heap*.
- **RE** a árvore viola as duas propriedades, portanto não é um *heap*.

Por exemplo, se a árvore não é um *heap* porque viola apenas a propriedade estrutural, escreva **E**. Se for um *heap* e a chave no seu último nó for 40, escreva **H 40**.

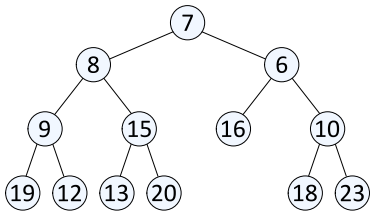
Árvore 1



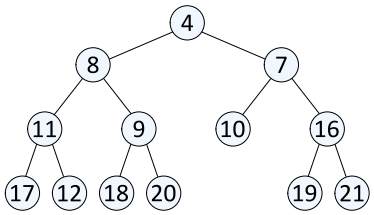
Árvore 2



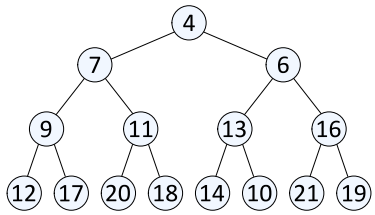
Árvore 3



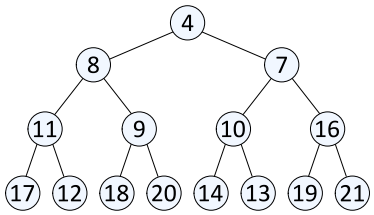
Árvore 4



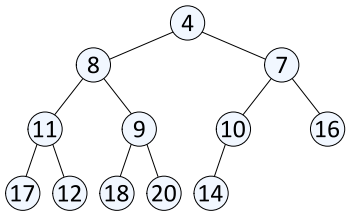
Árvore 5



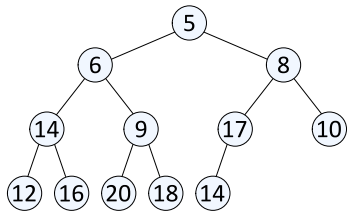
Árvore 6



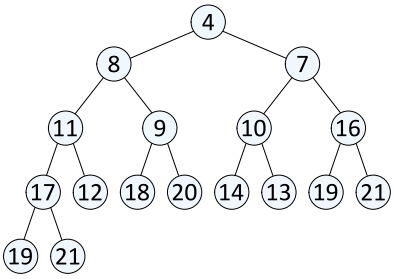
Árvore 7



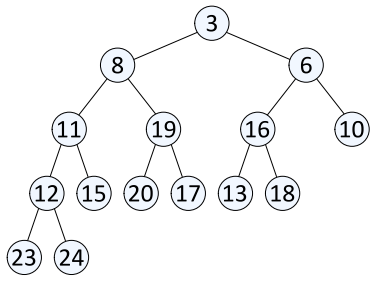
Árvore 8



Árvore 9

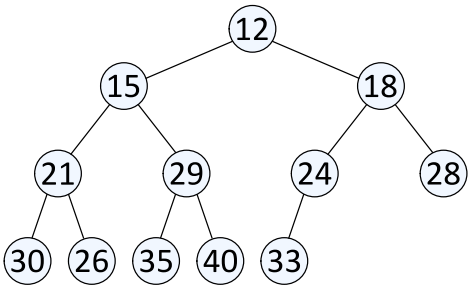


Árvore 10



Exercício 9.3 ✪✪

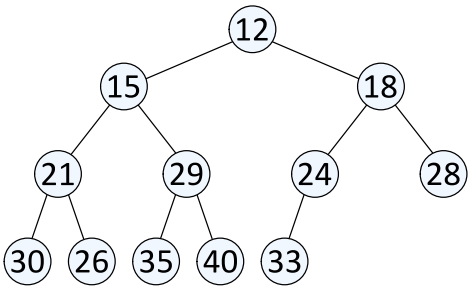
Considere o *heap* a seguir. Mostre a sequência de transformações feitas no *heap* ao inserirmos um nodo com chave 8. Para cada etapa, mostre também o conteúdo do vetor que armazena os nodos do *heap*.



0	1	2	3	4	5	6	7	8	9	10	11	12	13
	12	15	18	21	29	24	28	30	26	35	40	33	

Exercício 9.4 ✪✪

Considere o *heap* a seguir. Mostre a sequência de transformações feitas no *heap* ao removermos um nodo. Para cada etapa, mostre também o conteúdo do vetor que armazena os nodos do *heap*.



0	1	2	3	4	5	6	7	8	9	10	11	12	13
	12	15	18	21	29	24	28	30	26	35	40	33	

Exercício 9.5 ✪✪✪

Considere a figura do exercício 8.14, onde cada vetor implementa uma árvore binária diferente. Para cada um, determine se a árvore correspondente é um *heap*. Se for, indique o último nó; se não, indique qual a propriedade do *heap* é violada. Use a seguinte marcação nas suas respostas:

H *n* a árvore não viola nenhuma propriedade, portanto é um *heap*; o último nó tem chave ***n***.
R a árvore viola a propriedade relacional, portanto não é um *heap*.
E a árvore viola a propriedade estrutural, portanto não é um *heap*.
RE a árvore viola as duas propriedades, portanto não é um *heap*.

Exercício 9.6 ✪✪✪✪

Um *heap* é **crescente** quando a chave em um nó é menor ou igual às chaves nos seus filhos. Similarmente, um *heap* é **decrescente** quando a chave em um nó é maior ou igual às chaves nos seus filhos.

1) Em que posições de um *heap* crescente é possível encontrar a maior chave?
2) É possível armazenar 7 chaves distintas em um *heap* de modo que um percurso **prefixado** retorne as chaves em ordem crescente? Se não, justifique. Se sim, o *heap* deve ser crescente, decrescente, ou pode ser de ambos os tipos? Forneça como exemplo um *heap* contendo as chaves de 1 a 7.
3) É possível armazenar 7 chaves distintas em um *heap* de modo que um percurso **infixado** retorne as chaves em ordem crescente? Se não, justifique. Se sim, o *heap* deve ser crescente, decrescente, ou pode ser de ambos os tipos? Forneça como exemplo um *heap* contendo as chaves de 1 a 7.
4) É possível armazenar 7 chaves distintas em um *heap* de modo que um percurso **pósfixado** retorne as chaves em ordem crescente? Se não, justifique. Se sim, o *heap* deve ser crescente, decrescente, ou pode ser de ambos os tipos? Forneça como exemplo um *heap* contendo as chaves de 1 a 7.

10. Tabelas Hash

Exercício 10.1 ✪

Uma tabela *hash* consiste de um *array* A de tamanho N e de uma função *hash* $h(k)$. Quais os dois componentes da função *hash*? Forneça seus nomes, parâmetros de entrada e de saída, e explique seus objetivos.

Exercício 10.2 ✪

Liste quatro códigos *hash* mais utilizados. Para cada um, forneça seu nome, uma breve descrição de como funciona, e uma indicação dos tipos de chave às quais ele se adéqua.

Exercício 10.3 ✪

Liste duas funções de compressão mais utilizadas, seus nomes e suas expressões matemáticas.

Exercício 10.4 ✪✪

1) Algumas funções de compressão usam números primos e a função módulo (% em C). Por que?
2) Na função $(ay + b)\ \%\ N$, é necessário que $a\%N \neq 0$. Por que?

Exercício 10.5 ✪✪

1) O que é uma colisão em uma tabela *hash*?
2) Existem duas estratégias principais para o tratamento de colisões em tabelas *hash*. Para cada uma, forneça seu nome, uma breve descrição de como funciona, e vantagens e desvantagens.
3) Qual estratégia para o tratamento de colisões é mais apropriado quando se desconhece o número de itens a serem armazenados na tabela? Justifique.

Exercício 10.6 ✪✪

1) Qual o tempo de execução das operações de busca, inserção e remoção em uma tabela *hash*
   - no melhor caso?
   - no pior caso?
2) O que caracteriza o pior caso possível para as operações sobre uma tabela *hash*?
3) O principal fator que afeta o desempenho de uma tabela *hash* é o chamado fator de carga. Defina esse fator e explique seu efeito sobre o desempenho da tabela *hash*.

Exercício 10.7 ✪✪

Considere uma tabela *hash* de 11 posições usando encadeamento separado e a função *hash*

$$h(k) = (2k + 5)\ \%\ 11$$

Desenhe a tabela após a inserção das chaves a seguir, na ordem em que aparecem. Comece cada item com uma tabela *hash* vazia. Use os valores pré-calculados fornecidos no último exercício.

1) `7, 12, 19, 29, 35, 42, 56, 67, 76, 89, 81`
2) `89, 97, 76, 42, 19, 28, 50, 67, 35, 12, 7`
3) `7, 21, 92, 6, 53, 24, 65, 76, 80, 91, 19`
4) `9, 16, 32, 45, 69, 81, 94, 2, 26, 53, 19`
5) `12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5`

Exercício 10.8 ✪✪✪✪

Idem ao item 5 do exercício anterior, exceto que tabela usa endereçamento aberto e teste linear.

Exercício 10.9 ✪✪✪✪

Uma técnica de resolução de colisões em tabelas *hash* com endereçamento aberto é o *hashing* duplo. Essa técnica usa uma função de *hash* secundário $d(k)$ e resolve colisões colocando o item que colidiu na primeira célula disponível na série $\big(i + j \times d(k)\big)\ \%\ N$ para $j = 0, 1, \ldots, N-1$ onde N é o tamanho da tabela, k a chave do item que colidiu, e i a posição da tabela onde tentou-se inserir o item que colidiu.

Considere uma tabela *hash* com 11 posições que usa as seguintes funções de *hash*:

Hash primário: $h(k) = (2k + 5)\ \%\ 11$

Hash secundário: $d(k) = 7 - (k\ \%\ 7)$

Mostre a tabela *hash* após a inserção das chaves a seguir, na ordem em que aparecem. Use a tabela a seguir para facilitar os cálculos.

1) 87, 16, 94, 5, 12, 20, 13, 65, 43, 10, 27
2) 42, 11, 37, 15, 60, 20, 89, 29, 82, 72, 52
3) 15, 84, 32, 61, 75, 6, 1, 5, 18, 96, 28
4) 24, 46, 53, 28, 70, 91, 40, 60, 85, 81, 22
5) 31, 76, 15, 2, 44, 27, 86, 63, 9, 54, 93
6) 41, 11, 37, 15, 60, 20, 89, 29, 82, 72, 58
7) 18, 84, 32, 61, 75, 6, 1, 5, 18, 93, 31
8) 42, 64, 35, 82, 7, 19, 27, 66, 58, 75, 11

k	$(2k + 5)\ \mathbf{mod}\ 11$ $7 - (k\ \mathbf{mod}\ 7)$

0	5 7	1	7 6	2	9 5	3	0 4	4	2 3	5	4 2	6	6 1	7	8 7	8	10 6	9	1 5
10	3 4	11	5 3	12	7 2	13	9 1	14	0 7	15	2 6	16	4 5	17	6 4	18	8 3	19	10 2
20	1 1	21	3 7	22	5 6	23	7 5	24	9 4	25	0 3	26	2 2	27	4 1	28	6 7	29	8 6
30	10 5	31	1 4	32	3 3	33	5 2	34	7 1	35	9 7	36	0 6	37	2 5	38	4 4	39	6 3
40	8 2	41	10 1	42	1 7	43	3 6	44	5 5	45	7 4	46	9 3	47	0 2	48	2 1	49	4 7
50	6 6	51	8 5	52	10 4	53	1 3	54	3 2	55	5 1	56	7 7	57	9 6	58	0 5	59	2 4
60	4 3	61	6 2	62	8 1	63	10 7	64	1 6	65	3 5	66	5 4	67	7 3	68	9 2	69	0 1
70	2 7	71	4 6	72	6 5	73	8 4	74	10 3	75	1 2	76	3 1	77	5 7	78	7 6	79	9 5
80	0 4	81	2 3	82	4 2	83	6 1	84	8 7	85	10 6	86	1 5	87	3 4	88	5 3	89	7 2
90	9 1	91	0 7	92	2 6	93	4 5	94	6 4	95	8 3	96	10 2	97	1 1	98	3 7	99	5 6

Parte 2 – Soluções

1. Recursão

Exercício 1.1

A função calcula a soma $n + (n-1) + (n-2) + \cdots + 2 + 1$
Abaixo, uma versão iterativa dessa função:

```
int func (int n) {
  int i, soma;
  for (i=1, soma=0; i<=n; i++)
    soma+=i;
  return (soma);
}
```

Exercício 1.2

Soma iterativa

```
int somaI (int a, int b) {
  for ( ; b>0; b--)
    a++;
  return (a);
}
```

Soma recursiva

```
int somaR (int a, int b) {
  if (a==0) return (b);
  if (b==0) return (a);
  return (somaR (++a, --b));
}
```

Multiplicação iterativa

```
int multI (int a, int b) {
  int m;
  for (m=0; b>0; b--)
    m = somaI (m,a);
  return (m);
}
```

Multiplicação recursiva

```
/* Essa função apenas passa o terceiro */
/* parâmetro para a função recursiva   */

int multR (int a, int b) {
  return (multRaux (a, b, 0));
}

/* Cálculo recursivo propriamente dito */

int multRaux (int a, int b, int m) {
  if (a==0 || b==0) return (m);
  return (multRaux (a, --b, somaR(m,a)));
}
```

Exercício 1.3

```
int menorDigito (int n) {
  if (n<=9) return (n);
  return ((n%10) <= menorDigito (n/10) ? (n%10) : menorDigito (n/10));
}
int maiorDigito (int n) {
  if (n==0) return (0);
  return ((n%10) >= maiorDigito (n/10) ? (n%10) : maiorDigito (n/10));
}
```

Exercício 1.4

As funções de soma (`somaI` e `somaR`) na solução anterior já obedecem todos os critérios do exercício. As funções de multiplicação não podem obedecer todos os critérios ao mesmo tempo; ao menos um deve ser violado. Em `multI` o critério **c** foi violado; em `multR` o critério **b** foi violado.

Uma implementação alternativa de `multR`, que viola o critério **a**, é dada abaixo:

```
int multR (int a, int b) {
  if (a==0 || b==0) return (0);
  if (a==1) return (b);
  if (b==1) return (a);
  return (a + multR (a, --b));
}
```

Exercício 1.5

```
int contaDigitos (int n) {
  if (n<10) return (1);
  return (1 + contaDigitos (n/10));
}
int somaDigitos (int n) {
  if (n==0) return (0);
  return ((n%10) + somaDigitos (n/10));
}
```

Exercício 1.6

n	$A_i(n)$	$A_r(n)$
0	0	0
1	0	0
2	1	1
3	2	2
4	3	4
5	4	7
6	5	12
7	6	20
8	7	33
9	8	54
10	9	88

Exercício 1.7

```
int zeraPares (int numero) {
  if ((numero < 10) && (numero%2)==0)
    return (0);
  if ((numero < 10) && (numero%2)==1)
    return (numero);
  return (10*zeraPares(numero/10) + zeraPares(numero%10));
}
int zeraImpares (int numero) {
  if ((numero < 10) && (numero%2)==0)
    return (numero);
  if ((numero < 10) && (numero%2)==1)
    return (0);
  return (10*zeraImpares(numero/10) + zeraImpares(numero%10));
}
int removePares (int numero) {
  if ((numero < 10) && (numero%2)==0)
    return (0);
  if ((numero < 10) && (numero%2)==1)
    return (numero);
  if (numero%2 == 0)
    return (removePares(numero/10));
  else
    return (10*removePares(numero/10)+removePares(numero%10));
 /* return (10*removePares(numero/10)+(numero%10));  Isso também funciona */
}
int removeImpares (int numero) {
  if ((numero < 10) && (numero%2)==0)
    return (numero);
  if ((numero < 10) && (numero%2)==1)
    return (0);
  if (numero%2 == 0)
    return (10*removeImpares(numero/10)+(numero%10));
  return (removeImpares(numero/10));
}
```

Exercício 1.8

```
inverteRec (numero/10, 10*inverso+(numero%10))
```

Exercício 1.9

MDC recursivo

```
int mdcR (int x, int y) {
  if ((y<=x) && ((x%y)==0))
    return (y);
  if (x<y)
    return (mdcR (y, x));
  return (mdcR (y, x%y));
}
```

MDC iterativo

```
int mdcI (int x, int y) {
  int t;

  // Importante garantir que x >= y
  // Troque x e y se necessário

  if (x<y) {
    t=y; y=x; x=t;
  }
  while ((x%y)!=0) {
    t=x%y; x=y; y=t;
  }
  return (y);
}
```

Exercício 1.10

A soma de dois números pode ser feita apenas com incrementos e decrementos porque, à medida que os parâmetros são transformados a cada passo, o resultado da soma está contido nos valores dos dois parâmetros, sem necessidade de armazenar informação adicional em outro lugar ou através de outro mecanismo:

```
soma (3,6) =
soma (2,7) =
soma (1,8) =
soma (0,9) = 9
```

A multiplicação, por outro lado, não pode ser feita apenas com com incrementos e decrementos:

```
mult (3,6) ≠
mult (2,7) ≠
mult (1,8) ≠
mult (0,9)
```

A multiplicação requer ***informação ou mecanismo adicional para armazenar resultados parciais***. Uma maneira de fazer isso é através de uma variável local:

Multiplicação iterativa

```
int multI (int a, int b) {
  int m;
  for (m=0; b>0; b--)
    m = somaI (m,a);
  return (m);
}
```

Outra maneira é através de um parâmetro adicional:

Multiplicação recursiva, versão 1

```
/* Essa função apenas passa o terceiro */
/* parâmetro para a função recursiva   */

int multR (int a, int b) {
  return (multRaux (a, b, 0));
}

/* Cálculo recursivo propriamente dito */

int multRaux (int a, int b, int m) {
  if (a==0 || b==0) return (m);
  return (multRaux (a, --b, somaR(m,a)));
}
```

Outra maneira é através de uma operação adicional:

Multiplicação recursiva, versão 2

```
int multR (int a, int b) {
  if (a==0 || b==0) return (0);
  if (a==1) return (b);
  if (b==1) return (a);
  return (a + multR (a, --b));
}
```

Exercício 1.11

```
Bool SomaSubconjunto (int inicio, int *numeros, int totNumeros, int soma) {
  // Caso base:
  // se não há mais números, então só há uma solução se soma é 0

  if (inicio >= totNumeros)
    return (soma == 0);

  // Idéia principal: ou numeros[inicio] é usado, ou não é.
  // Trate numeros[inicio] explicitamente, e deixe a recursão
  // tartar o resto do array.

  // Chamada recursiva para tratar o caso de numeros[inicio] ser usado;
  // não esqueça de subtraí-lo de soma na chamada recursiva

  if (SomaSubconjunto (inicio+1, numeros, totNumeros, soma-numeros[inicio]))
    return (TRUE);

  // Chamada recursiva para tratar o caso de numeros[inicio] não ser usado

  if (SomaSubconjunto (inicio+1, numeros, totNumeros, soma))
    return (TRUE);

  // Se as tentativas acima falharam, então não é possível obter a soma

  return (FALSE);
}
```

2. Abstração de dados

Exercício 2.1

Definição do TDA Complexo

Define-se *complexo* como dois números *reais:* uma parte real e um coeficiente da parte imaginária:

$$a = a_{real} + i\ a_{imaginario}$$

Usando a notação de vetor:

$$a = \langle a[0], a[1] \rangle$$

onde

$a[0]$ é a_{real} (a parte real de a)

$a[1]$ é $a_{imaginario}$ (a parte imaginária de a)

Define-se a soma de dois números complexos a e b como

$$(a[0] + a[1]i) + (b[0] + b[1]i) = (a[0] + b[0]) + (a[1] + b[1]i)$$
$$= soma[0] + soma[1]i$$

Define-se a multiplicação de dois números complexos a e b como

$$(a[0] + a[1]i) \times (b[0] + b[1]i) = (a[0] \times b[0] - a[1] \times b[1]) + (a[0] \times b[1] + b[0] \times a[1])i$$
$$= mult[0] + mult[1]i$$

Define-se a igualdade dois números complexos a e b como

$$(a[0] + a[1]i) = (b[0] + b[1]i) \Leftrightarrow (a[0] = b[0]) \text{ AND } (a[1] = b[1])$$

Especificação do TDA Complexo

```
abstract typedef <real,real> COMPLEXO;

abstract COMPLEXO cria (real a, real b);
postcondition
  cria_racional[0] = a;
  cria_racional[1] = b;

abstract COMPLEXO soma (COMPLEXO a, COMPLEXO b);
postcondition
  soma[0] = a[0] + b[0];
  soma[1] = a[1] + b[1];

abstract COMPLEXO mult (COMPLEXO a, COMPLEXO b);
postcondition
  mult[0] = a[0]*b[0] - a[1]*b[1];
  mult[1] = a[0]*b[1] + b[0]*a[1];

abstract COMPLEXO igual (COMPLEXO a, COMPLEXO b);
postcondition
  igual = (a[0] == b[0]) AND (a[1] == b[1]);
```

Exercício 2.2

Especificação do TDA Vetor

```
abstract typedef <float,float> VETOR;

abstract VETOR cria (float a, float b);
precondition
  (a <> 0) OR (b <> 0);
postcondition
  cria_vetor[0] = a;
  cria_vetor[1] = b;

abstract float modulo (VETOR v);
postcondition
  modulo = SQRT (v[0]*v[0] + v[1]*v[1]);

abstract float produto (VETOR v, VETOR w);
postcondition
  produto = v[0]*w[0] + v[1]*w[1];

abstract float angulo (VETOR v, VETOR w);
postcondition
  angulo = ACOS (produto(v,w) / (modulo(v) * modulo(w)));

abstract boolean iguais (VETOR v, VETOR w);
postcondition
  iguais = (v[0] == w[0]) AND (v[1] == w[1]);
```

Exercício 2.3

```
int main (void) {
  union {
    char      c [8];
    short int si[4];
    int       i [2];
    double    d;
  }
    u;
  int k;

  u.c[0] = 'A';  u.c[1] = 'B';
  u.c[2] = 'C';  u.c[3] = 'D';
  u.c[4] = '1';  u.c[5] = '2';
  u.c[6] = '3';  u.c[7] = 'A';

  printf ("\n Como char: ");
  for (k=0; k<8; k++)
    printf ("%c ", u.c[k]);
  printf ("\n Como short int: ");
  for (k=0; k<4; k++)
    printf ("%d ", u.si[k]);
  printf ("\n Como int: ");
  for (k=0; k<2; k++)
    printf ("%d ", u.i[k]);
  printf ("\n Como double: ");
  printf ("%f", u.d);
  return 0;
}
```

3. Vetores

Exercício 3.1

```
#include <stdio.h>
#include <stdlib.h>

#define TAM_MAX 30

typedef struct sAluno {
  int mat;
  float nota;
}
  tAluno;

tAluno turma[TAM_MAX];

int main() {
  int
    tamTurma,
    iAluno,
    abaixoDaMedia = 0,
    naMedia       = 0,
    acimaDaMedia  = 0,
    naMaior       = 0;
  float
    soma  = 0,
    media = 0,
    maior = 0;

  printf ("Qual o tamanho da turma? ");
  scanf ("%d", &tamTurma);
  printf ("\n");
  for (iAluno = soma = 0; iAluno < tamTurma; iAluno++) {
    printf ("Aluno %d - matricula? ", iAluno+1);
    scanf ("%d", &turma[iAluno].mat);
    printf ("Aluno %d - nota? ", iAluno+1);
    scanf ("%f", &turma[iAluno].nota);
    soma += turma[iAluno].nota;
    if (turma[iAluno].nota > maior)
      maior = turma[iAluno].nota;
    printf ("\n");
  }
  media = soma / tamTurma;
  printf ("Nota media da turma: %.2f", media);
  printf ("\n");
  for (iAluno = 0; iAluno < tamTurma; iAluno++) {
    if (turma[iAluno].nota < media)
      abaixoDaMedia++;
    else if (turma[iAluno].nota == media)
      naMedia++;
    else
      acimaDaMedia++;
    if (turma[iAluno].nota == maior)
      naMaior++;
  }
  printf ("  %2d acima da media \n", acimaDaMedia);
  printf ("  %2d na media \n", naMedia);
  printf ("  %2d abaixo da media \n", abaixoDaMedia);
  printf ("\n");
  printf ("Maior nota da turma: %.2f \n", maior);
  printf ("  %2d com a maior nota \n", naMaior);
  printf ("\n");
  system ("PAUSE");
  return 0;
}
```

Exercício 3.2

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define TAM_MAX 100

bool ehLetra (char c) {
  return ((c>='a' && c<='z') || (c>='A' && c<='Z'));
}

char proximaLetra (char c) {
  c++;
  if (c=='z'+1) c = 'a';
  if (c=='Z'+1) c = 'A';
  return (c);
}

void criptografar (char *origem, char *destino) {
  int
    iPos, tamOrigem, trocar;
  char
    c;
  tamOrigem = strlen (origem);
  trocar = (tamOrigem < TAM_MAX ? tamOrigem : TAM_MAX);
  for (iPos=0; iPos<trocar; iPos++) {
    c = origem[iPos];
    destino[iPos]= (ehLetra(c) ? proximaLetra(c) : c);
  }
}

int main (void) {
  char
    mensagemOriginal      [TAM_MAX],
    mensagemCriptografada [TAM_MAX];

  printf ("Digite uma mensagem qualquer, ou 'fim' para terminar: ");
  scanf ("%s", mensagemOriginal);
  while (strcmp (mensagemOriginal, "fim")) {
    printf ("Mensagem original      = \"%s\"\n", mensagemOriginal);
    criptografar (mensagemOriginal, mensagemCriptografada);
    printf ("Mensagem criptografada = \"%s\"\n\n", mensagemCriptografada);
    printf ("Digite uma mensagem qualquer, ou 'fim' para terminar: ");
    scanf ("%s", mensagemOriginal);
  }
  return 0;
}
```

Exercício 3.3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define TAM_MAX 100

bool ehPalindromo (char *palavra, int tamanho) {
  int
    pos;

  for (pos = 0; pos < (int)((tamanho+1)/2); pos++)
    if (palavra[pos] != palavra[tamanho-pos-1])
      return (false);
  return (true);
}
```

```
int main(int argc, char *argv[]) {
  char
    palavra [TAM_MAX];

  printf ("Digite uma palavra qualquer, ou 'fim' para terminar: ");
  scanf ("%s", palavra);
  while (strcmp (palavra, "fim")) {
    printf ("[%s] ", palavra);
    if (! ehPalindromo(palavra, strlen(palavra)))
      printf ("nao ");
    printf ("eh um palindromo\n\n");
    printf ("Digite uma palavra qualquer, ou 'fim' para terminar: ");
    scanf ("%s", palavra);
  }
  return 0;
}
```

Exercício 3.4

```
#include <stdio.h>
#include <stdlib.h>

#define TAM_MAX    8
#define POS_VAZIA -1

int A [TAM_MAX],
    B [TAM_MAX],
    C [TAM_MAX];

void limpaSeq (int *pSeq) {
  int iPos;
  for (iPos=0; iPos < TAM_MAX; iPos++)
    *(pSeq++) = POS_VAZIA;
}

void mostraSeq (char *nomeSeq, int *pSeq) {
  int iPos;
  printf ("%s = [ ", nomeSeq);
  for (iPos=0; iPos < TAM_MAX; iPos++)
    printf ("%2d ", *(pSeq++));
  printf ("]\n");
}

void num2seq (int numero, int *pSeq) {
  int iPos, *pFimSeq, tamSeq;
  pFimSeq = pSeq + TAM_MAX - 1;
  while (numero > 0) {
    *(pFimSeq--) = numero%10;
    numero /= 10;
  }
  tamSeq = TAM_MAX - (pFimSeq - pSeq + 1);
  for (iPos=0; iPos < tamSeq; iPos++) {
    *(pSeq++) = *(++pFimSeq);
    *pFimSeq = POS_VAZIA;
  }
}
int seq2num (int *pSeq) {
  int numero = 0;
  while (*pSeq != POS_VAZIA) {
    numero *= 10;
    numero += *(pSeq++);
  }
  return (numero);
}
int main (void) {
  int a=0, b=0;
  limpaSeq (A);
  limpaSeq (B);
```

```
  limpaSeq (C);
  num2seq (123,  A);
  num2seq (6789, B);
  a = seq2num (A);
  b = seq2num (B);
  num2seq (a+b, C);
  mostraSeq ("A", A);
  mostraSeq ("B", B);
  mostraSeq ("C", C);
  printf ("\n");
  system("PAUSE");
  return 0;
}
```

Exercício 3.5

```
#define TAM_MAX   15
#define POS_VAZIA 0

char A [TAM_MAX],
     B [TAM_MAX],
     C [TAM_MAX];

void uniaoComRep (char *letraA, char *letraB, char *letraC) {
  while ((*letraA != POS_VAZIA) && (*letraB != POS_VAZIA))
    *letraC++ = (*letraA <= *letraB ? *letraA++ : *letraB++);
  while (*letraA != POS_VAZIA)
    *letraC++ = *letraA++;
  while (*letraB != POS_VAZIA)
    *letraC++ = *letraB++;
}
void uniaoSemRep (char *letraA, char *letraB, char *letraC) {
  while ((*letraA != POS_VAZIA) && (*letraB != POS_VAZIA))
    if (*letraA == *letraC)
      letraA++;
    else if (*letraB == *letraC)
      letraB++;
    else {
      if (*letraC != POS_VAZIA)
        letraC++;
      if (*letraA <= *letraB)
        *letraC = *letraA++;
      else
        *letraC = *letraB++;
    }
  while (*letraA != POS_VAZIA) {
    if (*letraA != *letraC) {
      if (*letraC != POS_VAZIA)
        letraC++;
      *letraC = *letraA;
    }
    letraA++;
  }
  while (*letraB != POS_VAZIA) {
    if (*letraB != *letraC) {
      if (*letraC != POS_VAZIA)
        letraC++;
      *letraC = *letraB;
    }
    letraB++;
  }
}
void intersecaoComRep (char *letraA, char *letraB, char *letraC) {
  while ((*letraA != POS_VAZIA) && (*letraB != POS_VAZIA))
    if (*letraA == *letraB) {
      *letraC = *letraA; letraA++; letraB++; letraC++;
    }
    else if (*letraA <= *letraB)
      letraA++;
```

```
    else
      letraB++;
}
void intersecaoSemRep (char *letraA, char *letraB, char *letraC) {
  while ((*letraA != POS_VAZIA) && (*letraB != POS_VAZIA))
    if (*letraA == *letraB) {
      if (*letraA != *letraC) {
        if (*letraC != POS_VAZIA)
          letraC++;
        *letraC = *letraA;
      }
      letraA++;
      letraB++;
    }
    else if (*letraA <= *letraB)
      letraA++;
    else
      letraB++;
}
```

Exercício 3.6

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define TAM_MAX 30

int
  totNum = 0,
  numeros [TAM_MAX];

bool ehImpar (int numero) {
  return ((numero % 2) == 1);
}

void obtemNumeros (void) {
  int
    iNum;
  printf ("Quantos numeros? ");
  scanf ("%d", &totNum);
  while (totNum > TAM_MAX) {
    printf ("Quantidade de numeros deve ser menor que %d\n", TAM_MAX);
    printf ("Quantos numeros? ");
    scanf ("%d", &totNum);
  }
  printf ("\n");
  for (iNum = 0; iNum < totNum; iNum++) {
    printf ("Numero %2d? ", iNum+1);
    scanf ("%d", &numeros[iNum]);
  }
  printf ("\n");
}

int comparaInteiros (const void * a, const void * b) {
  return ( *(int*)a - *(int*)b );
}

void ordenaNumeros (void) {
  qsort (numeros, totNum, sizeof (int), comparaInteiros);
}

float calculaMedia (void) {
  int
    iNum,
    soma;
  for (iNum = soma = 0; iNum < totNum; iNum++)
    soma += numeros[iNum];
  return ((float) soma / totNum);
```

```
}

int main (void) {
  int
    iNum,
    freqMax,
    posFreqMax,
    repetFreqMax,
    moda,
    totDistintos,
    distintos   [TAM_MAX],
    frequencias [TAM_MAX];
  float
    media   = 0.0,
    mediana = 0.0;
  bool
    bModaDefinida = false;

  /* Leitura dos numeros */

  obtemNumeros();

  /* Calculo da media */

  media = calculaMedia();
  printf ("Media: %.2f \n", media);

  /* Calculo da mediana */

  ordenaNumeros();
  if (ehImpar (totNum))
    mediana = (float) numeros [(totNum-1) / 2];
  else
    mediana = (float) (numeros [(totNum/2) - 1] + numeros [totNum/2]) / 2;
  printf ("Mediana: %.2f \n", mediana);

  /* Calculo da moda */

  distintos[0] = numeros[0];
  frequencias[0] = 1;
  totDistintos = 1;
  for (iNum = 1; iNum < TAM_MAX; iNum++)
    distintos[iNum] = frequencias[iNum] = -1;

  for (iNum = 1; iNum < totNum; iNum++) {
    if (distintos[totDistintos-1] == numeros[iNum])
      frequencias[totDistintos-1]++;
    else {
      totDistintos++;
      distintos[totDistintos-1] = numeros[iNum];
      frequencias[totDistintos-1] = 1;
    }
  }

  freqMax = posFreqMax = -1;
  for (iNum = 0; iNum < totDistintos; iNum++)
    if (frequencias[iNum] > freqMax) {
      freqMax = frequencias[iNum];
      posFreqMax = iNum;
    }

  for (iNum = repetFreqMax = 0; iNum < totDistintos; iNum++)
    if (frequencias[iNum] == freqMax)
      repetFreqMax++;

  bModaDefinida = (repetFreqMax == 1);

  if (bModaDefinida) {
    moda = distintos [posFreqMax];
    printf ("Moda: %d \n", moda);
  }
  else
```

```
    printf ("Moda indefinida \n");

  printf ("\n");
  system ("PAUSE");
  return 0;
}
```

Exercício 3.7

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define TAM_MAX 100

bool ehPalindromo (char *palavra, int tamanho, int pos) {
  bool
    letrasIguais;

  if (tamanho-2*pos <= 1)
    return (true);
  else
    letrasIguais = (palavra[pos] == palavra[tamanho-pos-1]);
    return (letrasIguais && ehPalindromo (palavra, tamanho, pos+1));
}

int main(int argc, char *argv[]) {
  char
    palavra [TAM_MAX];

  printf ("Digite uma palavra qualquer, ou 'fim' para terminar: ");
  scanf ("%s", palavra);
  while (strcmp (palavra, "fim")) {
    printf ("[%s] ", palavra);
    if (! ehPalindromo(palavra, strlen(palavra), 0))
      printf ("nao ");
    printf ("eh um palindromo\n\n");
    printf ("Digite uma palavra qualquer, ou 'fim' para terminar: ");
    scanf ("%s", palavra);
  }
  return 0;
}
```

Exercício 3.8

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#define TAM_MAX 10

/* Funcao que indica se a matriz contem zeros em todas as posicoes */

bool temApenasZeros (int *pMatriz, int tamanho) {
  int iLin, iCol;
  for (iLin = 0; iLin < tamanho; iLin++)
    for (iCol = 0; iCol < tamanho; iCol++)
      if (*(pMatriz + iLin*TAM_MAX + iCol) != 0)
        return (false);
  return (true);
}

/* Funcao que indica se a matriz contem zeros acima da diagonal principal */
/* As demais posicoes (diagonal principal e abaixo dela) sao ignoradas    */

bool temZerosAcima (int *pMatriz, int tamanho) {
```

```
  int iLin, iCol;
  for (iLin = 0; iLin < tamanho; iLin++)
    for (iCol = 0; iCol < tamanho; iCol++) {
      if (! (iLin < iCol))
        continue;
      if (*(pMatriz + iLin*TAM_MAX + iCol) != 0)
        return (false);
  }
  return (true);
}

/* Funcao que indica se a matriz contem zeros abaixo da diagonal principal */
/* As demais posicoes (diagonal principal e acima dela) sao ignoradas      */

bool temZerosAbaixo (int *pMatriz, int tamanho) {
  int iLin, iCol;
  for (iLin = 0; iLin < tamanho; iLin++)
    for (iCol = 0; iCol < tamanho; iCol++) {
      if (! (iLin > iCol))
        continue;
      if (*(pMatriz + iLin*TAM_MAX + iCol) != 0)
        return (false);
  }
  return (true);
}

char tipoDeMatriz (int *pMatriz, int tamanho) {
  if (temApenasZeros (pMatriz, tamanho))
   return ('V');
  if (temZerosAcima (pMatriz, tamanho) && temZerosAbaixo (pMatriz, tamanho))
   return ('D');
  if (temZerosAcima (pMatriz, tamanho))
   return ('S');
  if (temZerosAbaixo (pMatriz, tamanho))
   return ('I');
return ('X');
}

int main(int argc, char *argv[]) {
  int iLin, iCol, tamMatriz = 0, matriz [TAM_MAX][TAM_MAX];
  memset (matriz, 0, sizeof (matriz));
  printf ("Digite o tamanho da matriz quadrada (de 1 a %d)\n", TAM_MAX - 1);
  printf ("ou outro valor para terminar: ");
  scanf ("%d", &tamMatriz);
  while ((tamMatriz > 0) && (tamMatriz < TAM_MAX)) {
    for (iLin = 0; iLin < tamMatriz; iLin++)
      for (iCol = 0; iCol < tamMatriz; iCol++) {
        printf ("Digite M[%2d,%2d]: ", iLin, iCol);
        scanf ("%d", &matriz[iLin][iCol]);
      }
    printf ("A matriz eh do tipo %c \n\n", tipoDeMatriz ((int *)matriz, tamMatriz));
    memset (matriz, 0, sizeof (matriz));
    printf ("Digite o tamanho da matriz quadrada (de 1 a %d)\n", TAM_MAX - 1);
    printf ("ou outro valor para terminar: ");
    scanf ("%d", &tamMatriz);
  }
  return 0;
}
```

Exercício 3.9

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#define TAM_MAX    15
#define POS_VAZIA   0
```

```
#define MAX_A   TAM_MAX
#define MAX_B   TAM_MAX
#define MAX_C   (TAM_MAX * 2)

char
  A [MAX_A],
  B [MAX_B],
  C [MAX_C];

void limpaA (void) {
  int iPos;
  for (iPos=0; iPos < MAX_A; iPos++)
    A[iPos] = POS_VAZIA;
}

void limpaB (void) {
  int iPos;
  for (iPos=0; iPos < MAX_B; iPos++)
    B[iPos] = POS_VAZIA;
}

void limpaC (void) {
  int iPos;
  for (iPos=0; iPos < MAX_C; iPos++)
    C[iPos] = POS_VAZIA;
}

void imprimeConjunto (char conjunto, char *letra) {
  int iPos, max;
  switch (conjunto) {
    case ('A'): max = MAX_A; break;
    case ('B'): max = MAX_B; break;
    case ('C'): max = MAX_C; break;
    default: exit (-1);
  }
  printf ("%c = { ", conjunto);
  for (iPos=0; iPos < max; iPos++)
    if (*letra)
      printf ("%c ", *letra++);
  printf ("}\n");
}

bool jaExiste (char letra, char *sequencia) {
  while ((*sequencia != letra) && (*sequencia != POS_VAZIA))
    sequencia++;
  return (*sequencia == letra);
}

void uniaoComRep (char *seqOrigem1, char *seqOrigem2, char *seqDestino) {
  char
    *letraOrigem1 = seqOrigem1,
    *letraOrigem2 = seqOrigem2,
    *letraDestino = seqDestino;

  while (*letraOrigem1 != POS_VAZIA)
    *(letraDestino++) = *(letraOrigem1++);
  while (*letraOrigem2 != POS_VAZIA)
    *(letraDestino++) = *(letraOrigem2++);
}

void uniaoSemRep (char *seqOrigem1, char *seqOrigem2, char *seqDestino) {
  char
    *letraOrigem1 = seqOrigem1,
    *letraOrigem2 = seqOrigem2,
    *letraDestino = seqDestino;

  while (*letraOrigem1 != POS_VAZIA) {
    if (! jaExiste (*letraOrigem1, seqDestino))
      *(letraDestino++) = *letraOrigem1;
    letraOrigem1++;
  }
  while (*letraOrigem2 != POS_VAZIA) {
```

```
    if (! jaExiste (*letraOrigem2, seqDestino))
      *(letraDestino++) = *letraOrigem2;
    letraOrigem2++;
  }
}

void intersecaoComRep (char *seqOrigem1, char *seqOrigem2, char *seqDestino) {
  int
    posOrigem2;
  char
    *letraOrigem1,
    *letraOrigem2,
    *letraDestino;
  bool
    usadoOrigem2 [MAX_B];

  letraOrigem1 = seqOrigem1;
  letraDestino = seqDestino;
  for (posOrigem2 = 0; posOrigem2 < MAX_B; posOrigem2++)
    usadoOrigem2 [posOrigem2] = false;
  while (*letraOrigem1 != POS_VAZIA) {
    letraOrigem2 = seqOrigem2;
    posOrigem2 = 0;
    while (*letraOrigem2 != POS_VAZIA) {
      if (*letraOrigem2 == *letraOrigem1) {
        if (! usadoOrigem2 [posOrigem2]) {
          *(letraDestino++) = *letraOrigem1;
          usadoOrigem2 [posOrigem2] = true;
          break;
        }
      }
      letraOrigem2++;
      posOrigem2++;
    }
  letraOrigem1++;
  }
}

void intersecaoSemRep (char *seqOrigem1, char *seqOrigem2, char *seqDestino) {
  int
    posOrigem2;
  char
    *letraOrigem1,
    *letraOrigem2,
    *letraDestino;
  bool
    usadoOrigem2 [MAX_B];

  letraOrigem1 = seqOrigem1;
  letraDestino = seqDestino;
  for (posOrigem2 = 0; posOrigem2 < MAX_B; posOrigem2++)
    usadoOrigem2 [posOrigem2] = false;
  while (*letraOrigem1 != POS_VAZIA) {
    if (jaExiste (*letraOrigem1, seqDestino))
      break;
    letraOrigem2 = seqOrigem2;
    posOrigem2 = 0;
    while (*letraOrigem2 != POS_VAZIA) {
      if (*letraOrigem2 == *letraOrigem1) {
        if (! usadoOrigem2 [posOrigem2]) {
          *(letraDestino++) = *letraOrigem1;
          usadoOrigem2 [posOrigem2] = true;
          break;
        }
      }
      letraOrigem2++;
      posOrigem2++;
    }
  letraOrigem1++;
  }
}
```

```
int main (int argc, char *argv[])
{
  limpaA();
  limpaB();

  strcpy (A, "gzritmzii");
  strcpy (B, "oogtigkid");

  imprimeConjunto ('A', A);
  imprimeConjunto ('B', B);

  printf ("\nUNIAO COM REPETICAO:\n");
  limpaC();
  uniaoComRep (A, B, C);
  imprimeConjunto ('C', C);

  printf ("\nUNIAO SEM REPETICAO:\n");
  limpaC();
  uniaoSemRep (A, B, C);
  imprimeConjunto ('C', C);

  printf ("\nINTERSECAO COM REPETICAO:\n");
  limpaC();
  intersecaoComRep (A, B, C);
  imprimeConjunto ('C', C);

  printf ("\nINTERSECAO SEM REPETICAO:\n");
  limpaC();
  intersecaoSemRep (A, B, C);
  imprimeConjunto ('C', C);

  printf ("\n");
  system("PAUSE");
  return 0;
}
```

Exercício 3.10

```
/* Arquivo: tipoChar.h
 * Sequencias de caracteres
 */

typedef
  char TIPO_DADO;

#define PRINT_DADO(dado) (printf ("%c",(dado)))

TIPO_DADO DADO_NULO = 0;

#define SEQ_A_INICIAL_TAM  8
#define SEQ_A_INICIAL      {'a', 'a', 'b', 'c', 'c', 'c', 'd', 'd'}

#define SEQ_B_INICIAL_TAM  8
#define SEQ_B_INICIAL      {'b', 'b', 'c', 'c', 'd', 'd', 'd', 'e'}
```

```
/* Arquivo: tipoParInt.h
 * Sequencias de pares de inteiros
 */

typedef
  struct {int x; int y; } TIPO_DADO;

#define PRINT_DADO(dado) (printf ("%d%d",(dado).x,(dado).y))

TIPO_DADO DADO_NULO = {-1,-1};

#define SEQ_A_INICIAL_TAM  8
```

```
#define SEQ_A_INICIAL      {{1,1},{1,1},{1,2},{2,2},{2,2},{2,2},{3,3},{3,3}}

#define SEQ_B_INICIAL_TAM  8
#define SEQ_B_INICIAL      {{1,2},{1,2},{2,2},{2,2},{3,3},{3,3},{3,3},{4,4}}
```

```
/* Arquivo: tipoTrioBool.h
 * Sequencias de trios de booleanos
 */

typedef
  struct {bool b1; bool b2; bool b3; } TIPO_DADO;

#define BOOL2LETRA(b)    ((b)?'T':'F')
#define PRINT_DADO(dado) (printf ("%c%c%c", BOOL2LETRA((dado).b1),  \
                                            BOOL2LETRA((dado).b2),  \
                                            BOOL2LETRA((dado).b3)))

TIPO_DADO DADO_NULO = {false, false, false};

#define SEQ_A_INICIAL_TAM  6
#define SEQ_A_INICIAL      { {true,  true,  true},  \
                             {true,  true,  true},  \
                             {true,  true,  false}, \
                             {true,  false, false}, \
                             {true,  false, false}, \
                             {false, false, true}  }


#define SEQ_B_INICIAL_TAM  10
#define SEQ_B_INICIAL      { {true,  true,  true},  \
                             {false, true,  true},  \
                             {false, true,  true},  \
                             {false, false, true},  \
                             {true,  false, false}, \
                             {true,  false, false}, \
                             {true,  false, false}, \
                             {false, true,  false}, \
                             {false, true,  false}, \
                             {true,  false, true}   }
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

/* Foram implementados 3 tipos de dados para armazenar nas sequencias:
 *    tipoChar
 *    tipoParInt
 *    tipoTrioBool
 *  Escolha o tipo desejado no #include a seguir
 */

#include "tipoParInt.h"

#define TAM_DADO (sizeof (TIPO_DADO))

#define TAM_MAX_SEQ    15

#define TAM_MAX_SEQ_A   TAM_MAX_SEQ
#define TAM_MAX_SEQ_B   TAM_MAX_SEQ
#define TAM_MAX_SEQ_C   (TAM_MAX_SEQ * 2)

void copiaDado (TIPO_DADO *pDadoDestino, TIPO_DADO *pDadoOrigem) {
  memcpy ((void *) pDadoDestino, (const void *) pDadoOrigem, TAM_DADO);
}

bool saoDadosIguais (TIPO_DADO *pDado1, TIPO_DADO *pDado2) {
```

```
    return (memcmp ((const void *) pDado1, (const void *) pDado2, TAM_DADO) == 0);
  }

  bool ehDadoNulo (TIPO_DADO *pDado) {
    return (saoDadosIguais (pDado, &DADO_NULO));
  }

  TIPO_DADO
    A [TAM_MAX_SEQ_A] = SEQ_A_INICIAL,
    B [TAM_MAX_SEQ_B] = SEQ_B_INICIAL,
    C [TAM_MAX_SEQ_C];

  void limpaA (int totPosicoesInicializadas) {
    int iPos;
    for (iPos=totPosicoesInicializadas; iPos < TAM_MAX_SEQ_A; iPos++)
      copiaDado (&A[iPos], &DADO_NULO);
  }

  void limpaB (int totPosicoesInicializadas) {
    int iPos;
    for (iPos=totPosicoesInicializadas; iPos < TAM_MAX_SEQ_B; iPos++)
      copiaDado (&B[iPos], &DADO_NULO);
  }

  void limpaC (void) {
    int iPos;
    for (iPos=0; iPos < TAM_MAX_SEQ_C; iPos++)
      copiaDado (&C[iPos], &DADO_NULO);
  }

  void imprimeSequencia (char nomeSeq, TIPO_DADO *pSeq) {
    int
      iPos, max;
    switch (nomeSeq) {
      case ('A'): max = TAM_MAX_SEQ_A; break;
      case ('B'): max = TAM_MAX_SEQ_B; break;
      case ('C'): max = TAM_MAX_SEQ_C; break;
      default: exit (-1);
    }
    printf ("%c = { ", nomeSeq);
    for (iPos=0; iPos < max; iPos++) {
      if (! ehDadoNulo (pSeq)) {
        PRINT_DADO(*pSeq);
        pSeq++;
        if (! ehDadoNulo (pSeq))
          printf (", ");
      }
    }
    printf (" }\n");
  }

  bool jaExiste (TIPO_DADO *pDado, TIPO_DADO *pSeq) {
    while ( (! ehDadoNulo (pSeq)) &&
            (! saoDadosIguais (pSeq, pDado)) )
      pSeq++;
    return (saoDadosIguais (pSeq, pDado));
  }

  void uniaoComRep (TIPO_DADO *seqOrigem1, TIPO_DADO *seqOrigem2, TIPO_DADO
*seqDestino) {
    TIPO_DADO
      *dadoOrigem1 = seqOrigem1,
      *dadoOrigem2 = seqOrigem2,
      *dadoDestino = seqDestino;

    while (! ehDadoNulo (dadoOrigem1))
      copiaDado (dadoDestino++, dadoOrigem1++);
    while (! ehDadoNulo (dadoOrigem2))
      copiaDado (dadoDestino++, dadoOrigem2++);
  }
```

```
  void uniaoSemRep (TIPO_DADO *seqOrigem1, TIPO_DADO *seqOrigem2, TIPO_DADO
*seqDestino) {
    TIPO_DADO
      *dadoOrigem1 = seqOrigem1,
      *dadoOrigem2 = seqOrigem2,
      *dadoDestino = seqDestino;

    while (! ehDadoNulo (dadoOrigem1)) {
      if (! jaExiste (dadoOrigem1, seqDestino))
        copiaDado (dadoDestino++, dadoOrigem1);
      dadoOrigem1++;
    }
    while (! ehDadoNulo (dadoOrigem2)) {
      if (! jaExiste (dadoOrigem2, seqDestino))
        copiaDado (dadoDestino++, dadoOrigem2);
      dadoOrigem2++;
    }
  }

  void intersecaoComRep (TIPO_DADO *seqOrigem1, TIPO_DADO *seqOrigem2, TIPO_DADO
*seqDestino) {
    int
      posOrigem2;
    TIPO_DADO
      *dadoOrigem1,
      *dadoOrigem2,
      *dadoDestino;
    bool
      usadoOrigem2 [TAM_MAX_SEQ_B];

    dadoOrigem1 = seqOrigem1;
    dadoDestino = seqDestino;
    for (posOrigem2 = 0; posOrigem2 < TAM_MAX_SEQ_B; posOrigem2++)
      usadoOrigem2 [posOrigem2] = false;
    while (! ehDadoNulo (dadoOrigem1)) {
      dadoOrigem2 = seqOrigem2;
      posOrigem2 = 0;
      while (! ehDadoNulo (dadoOrigem2)) {
        if (saoDadosIguais (dadoOrigem2, dadoOrigem1)) {
          if (! usadoOrigem2 [posOrigem2]) {
            copiaDado (dadoDestino++, dadoOrigem1);
            usadoOrigem2 [posOrigem2] = true;
            break;
          }
        }
        dadoOrigem2++;
        posOrigem2++;
      }
    dadoOrigem1++;
    }
  }

  void intersecaoSemRep (TIPO_DADO *seqOrigem1, TIPO_DADO *seqOrigem2, TIPO_DADO
*seqDestino) {
    int
      posOrigem2;
    TIPO_DADO
      *dadoOrigem1,
      *dadoOrigem2,
      *dadoDestino;
    bool
      usadoOrigem2 [TAM_MAX_SEQ_B];

    dadoOrigem1 = seqOrigem1;
    dadoDestino = seqDestino;
    for (posOrigem2 = 0; posOrigem2 < TAM_MAX_SEQ_B; posOrigem2++)
      usadoOrigem2 [posOrigem2] = false;
    while (! ehDadoNulo (dadoOrigem1)) {
      if (jaExiste (dadoOrigem1, seqDestino)) {
        dadoOrigem1++;
        continue;
      }
```

```
      dadoOrigem2 = seqOrigem2;
      posOrigem2 = 0;
      while (! ehDadoNulo (dadoOrigem2)) {
        if (saoDadosIguais (dadoOrigem2, dadoOrigem1)) {
          if (! usadoOrigem2 [posOrigem2]) {
            copiaDado (dadoDestino++, dadoOrigem1);
            usadoOrigem2 [posOrigem2] = true;
            break;
          }
        }
        dadoOrigem2++;
        posOrigem2++;
      }
      dadoOrigem1++;
    }
  }

  int main (int argc, char *argv[])
  {
    limpaA (SEQ_A_INICIAL_TAM);
    limpaB (SEQ_B_INICIAL_TAM);

    imprimeSequencia ('A', A);
    imprimeSequencia ('B', B);

    printf ("\nUNIAO COM REPETICAO:\n");
    limpaC();
    uniaoComRep (A, B, C);
    imprimeSequencia ('C', C);

    printf ("\nUNIAO SEM REPETICAO:\n");
    limpaC();
    uniaoSemRep (A, B, C);
    imprimeSequencia ('C', C);

    printf ("\nINTERSECAO COM REPETICAO:\n");
    limpaC();
    intersecaoComRep (A, B, C);
    imprimeSequencia ('C', C);

    printf ("\nINTERSECAO SEM REPETICAO:\n");
    limpaC();
    intersecaoSemRep (A, B, C);
    imprimeSequencia ('C', C);

    printf ("\n");
    system("PAUSE");
    return 0;
  }
```

4. Listas encadeadas

Exercício 4.1

		LS*i*	LS*f*	LS*if*	CS*i*	CS*f*	CS*if*	LD*i*	LD*f*	LD*if*	CD*i*	CD*f*	CD*if*
Direção	Subir	n	0	n	n	$n+1$	n	n	$2n-1$	n	n	$n+1$	n
	Descer	0	0	0	0	0	0	$2n-1$	n	n	$n+1$	n	n

Exercício 4.2

1) O problema está na definição do parâmetro, que é um ponteiro para o início da lista. Esse parâmetro é passado por valor, mas deveria ser passado por referência. Como está, a função realiza a operação corretamente, mas não é capaz de atualizar o ponteiro que foi passado para a função; a modificação efetuada na lista é perdida quando a função retorna. Ao final da função, a lista original permanece como estava antes da chamada da função.
2) O código corrigido está a seguir, com as modificações em destaque.

```
tDado TiraDoInicio (struct nodo **lista) {
  struct nodo* inicio;
  tDado dadoNoInicio;
  inicio = *lista;
  if (inicio == NULL)
    exit (-1);
  dadoNoInicio = inicio->dado;
  *lista = inicio->prox;
  free (inicio);
  return (dadoNoInicio);
}
```

Exercício 4.3

1 C 2 B

Exercício 4.4

```
void RemoveUltimo (struct nodo **lista) {
  struct nodo *penultimo, *ultimo;

  if (*lista == NULL)              /* Lista vazia:  */
    return;                        /* não faça nada */

  if (*lista->prox == NULL) {      /* Lista */
    free (*lista);                 /* com   */
    *lista = NULL;                 /* um    */
    return;                        /* único */
  }                                /* nodo  */

  penultimo = *lista;              /* Lista */
  ultimo = penultimo->prox;        /* com   */
  while (ultimo->prox != NULL) {   /* dois  */
    penultimo = ultimo;            /* ou    */
    ultimo = ultimo->prox;         /* mais  */
  }                                /* nodos */
  free (ultimo);                   /*       */
  penultimo->prox = NULL;          /*       */
}
```

Exercício 4.5

```
int MoveParaInicio (struct nodo** inicio, tDado procurado) {
  int ocorrencias = 0;
  struct nodo *anterior = NULL, *atual = NULL, *proximo = NULL;

  atual = *inicio;

  /* Lista vazia: nada a fazer */

  if (atual == NULL)
    return (0);

  /* Lista com um único nodo */

  if (atual->prox == NULL)
    return (atual->dado == procurado ? 1 : 0);

  /* Lista com dois ou mais nodos */

  ocorrencias = (atual->dado==procurado ? 1 : 0); /* Se o primeiro nodo contem   */
                                                  /* o dado, nao precisa move-lo */
  anterior = atual;
  atual = atual->prox;
  while (atual != NULL) {
    proximo = atual->prox;
    if (atual->dado == procurado) {
      atual->prox = *inicio;
      *inicio = atual;
      anterior->prox = proximo;
      ocorrencias++;
    }
    else
      anterior = atual;
    atual = proximo;
  }
  return (ocorrencias);
}
```

Exercício 4.6

```
bool ehPalindromo (struct nodo *inicio, struct nodo *fim) {
  while ((inicio != fim) && (inicio->depois != fim)) {
    if (inicio->letra != fim->letra)
      return (false);
    inicio = inicio->depois;
    fim = fim->antes;
  }
  if (inicio == fim)
    return (true);
  return (inicio->letra == fim->letra);
}
```

Exercício 4.7

```
bool ehPalindromo (struct nodo *inicio, struct nodo *fim) {
  if (inicio->letra != fim->letra)
    return (false);
  if (inicio == fim)
    return (true);
  if (inicio->depois == fim)
    return (inicio->letra == fim->letra);
  return (ehPalindromo (inicio->depois, fim->antes));
}
```

Exercício 4.8

```
struct nodo *TornaLinear (struct nodo *fim) {
  struct nodo *inicio;
  if (fim == NULL)
    return (NULL);
  inicio = fim->prox;
  fim->prox = NULL;
  return (inicio);
}

struct nodo *TornaCircular (struct nodo *inicio) {
  struct nodo *fim;
  if (inicio == NULL)
    return (NULL);
  fim = inicio;
  while (fim->prox != NULL)
    fim = fim->prox;
  fim->prox = inicio;
  return (fim);
}

void UneCirculares (struct nodo **fim1, struct nodo **fim2) {
  struct nodo *inicio;
  if (*fim2 == NULL)
    return;
  if (*fim1 != NULL) {
    inicio = *fim1->prox;
    *fim1->prox = *fim2->prox;
    *fim2->prox = inicio;
  }
  *fim1 = *fim2;
  *fim2 = NULL;
}
```

Exercício 4.9

```
int listaParaInt (struct nodo *lista) {
  int numero = 0;
  while (lista != NULL) {
    numero = numero*10 + lista->dado;
    lista = lista->prox;
  }
  return (numero);
}
```

Exercício 4.10

A função transforma a lista circular em uma lista linear.

Exercício 4.11

A função inverte a lista de entrada.

Exercício 4.12

A função inverte a lista de entrada.

5. Pilhas, filas e deques

Exercício 5.1

```
#define N 100
int A[N];
void esvazia (void) {
  A[0] = 0;
}
bool vazia (void) {
  return (A[0] == 0);
}
bool cheia (void) {
  return (A[0] == N-1);
}
int tamanho (void) {
  return (A[0]);
}
```

```
int topo (void) {
  if (vazia())
    erro ("Pilha vazia");
  else
    return (A[A[0]]);
}
void empilha (int n) {
  if (cheia())
    erro ("Pilha cheia");
  else
    A[++A[0]] = n;
}
int desempilha (void) {
  if (vazia())
    erro ("Pilha vazia");
  else
    return (A[A[0]--]);
}
```

Exercício 5.2

```
#define N 100
int A[N], t1, t2;
void esvazia (void) {
  t1=-1; t2=N;
}
Bool vazia (int pilha) {
  switch (pilha) {
    case (1): return (t1==-1);
    case (2): return (t2==N);
    default : erro ("Pilha invalida");
  }
}
int tamanho (int pilha) {
  switch (pilha) {
    case (1): return (t1+1);
    case (2): return (N-t2);
    default : erro ("Pilha invalida");
  }
}
Bool cheias (void) {
  return ((t2-t1)==1);
}
int topo (int pilha) {
  switch (pilha) {
    case (1): {
      if (vazia(1))
        erro ("Pilha 1 vazia");
      else
        return A[t1];
    }
    case (2): {
      if (vazia(2))
        erro ("Pilha 2 vazia");
      else
        return A[t2];
    }
    default:
      erro ("Pilha invalida");
  }
}
```

```
void empilha (int pilha, int n) {
  switch (pilha) {
    case (1): {
      if (cheias())
        erro ("Pilhas cheias");
      else
        A[++t1]=n;
      return;
    }
    case (2): {
      if (cheias())
        erro ("Pilhas cheias");
      else
        A[--t2]=n;
      return;
    }
    default:
      erro ("Pilha invalida");
  }
}
int desempilha (int pilha) {
  switch (pilha) {
    case (1): {
      if (vazia(1))
        erro ("Pilha 1 vazia");
      else
        return A[t1--];
    }
    case (2): {
      if (vazia(2))
        erro ("Pilha 2 vazia");
      else
        return A[t2++];
    }
    default:
      erro ("Pilha invalida");
  }
}
```

Exercício 5.3

```
#define N 100
int A[N];
void esvazia (void) {
  A[0]=0; A[N-1]=N-1;
}
Bool vazia (int pilha) {
  switch (pilha) {
    case (1): return (A[0]==0);
    case (2): return (A[N-1]==N-1);
    default : erro ("Pilha invalida");
  }
}
int tamanho (int pilha) {
  switch (pilha) {
    case (1): return (A[0]);
    case (2): return (N-1-A[N-1]);
    default : erro ("Pilha invalida");
  }
}


Bool cheias (void) {
  return ((A[N-1]-A[0])==1);
}
int topo (int pilha) {
  switch (pilha) {
    case (1): {
      if (vazia(1))
        erro ("Pilha 1 vazia");
      else
        return A[A[0]];
    }
    case (2): {
      if (vazia(2))
        erro ("Pilha 2 vazia");
      else
        return A[A[N-1]];
    }
    default:
      erro ("Pilha invalida");
  }
}
```

```
void empilha (int pilha, int n) {
  switch (pilha) {
    case (1): {
      if (cheias())
        erro ("Pilhas cheias");
      else
        A[++A[0]]=n;
      return;
    }
    case (2): {
      if (cheias())
        erro ("Pilhas cheias");
      else
        A[--A[N-1]]=n;
      return;
    }
    default:
      erro ("Pilha invalida");
  }
}

int desempilha (int pilha) {
  switch (pilha) {
    case (1): {
      if (vazia(1))
        erro ("Pilha 1 vazia");
      else
        return A[A[0]--];
    }
    case (2): {
      if (vazia(2))
        erro ("Pilha 2 vazia");
      else
        return A[A[N-1]++];
    }
    default:
      erro ("Pilha invalida");
  }
}
```

Exercício 5.4

Usando um deque

```
Bool ehPalindromo (void) {
  while (! acabaramAsLetras())
    poeInicioDeque (obtemLetra());
  while (tamDeque() > 1)
    if (tiraInicioDeque() !=
        tiraFimDeque())
      return (FALSE);
  return (TRUE);
}
```

Usando uma pilha e uma fila

```
Bool ehPalindromo (void) {
  char letra;
  while (! acabaramAsLetras()) {
    letra = obtemLetra();
    poePilha (letra);
    poeFila  (letra);
  }
  while (! pilhaVazia())
    if (tiraPilha() != tiraFila())
      return (FALSE);
  return (TRUE);
}
```

Exercício 5.5

`tiraDoInicio()`: Não. Não. Sim. Sim.
`tiraDoFim()`: Sim. Sim. Não. Não.

Exercício 5.6

(V) **a**: remoção do início, inserção no início e inserção no final executam em tempo $O(1)$; remoção do final executa em tempo $O(n)$. **b**: as quatro operações executam em tempo $O(1)$.

(F) Se o *array* é circular, não faz sentido falar em compactação. Mesmo que fizesse, nunca haverá desperdício de espaço após remoções da fila; as compactações não serviriam para nada, consumindo tempo desnecessariamente.

(F) Se a lista é linear, o fato de ser simplesmente ou duplamente encadeada é irrelevante. Apenas um ponteiro para o início da lista (frente da fila) resulta em remoção em tempo $O(1)$ e inserção em tempo $O(n)$; apenas um ponteiro para o final da lista (fim da fila) resulta em inserção em tempo $O(1)$ e remoção em tempo $O(n)$.

Exercício 5.7

```
/* Insere dado no fim da fila */

void poeFila (tDado dado) {
  if (pilhaVazia (0))
    while (! pilhaVazia (1))
      poePilha (0, tiraPilha (1));
  poePilha (0, dado);
}

/* Retorna o dado na frente */
/* da fila sem removê-lo    */

tDado frenteFila (void) {
  if (pilhaVazia (1))
    while (! pilhaVazia (0))
      poePilha (1, tiraPilha (0));
  return (topoPilha (1));
}
```

```
/* Retorna o tamanho da fila */

int tamFila (void) {
  return (tamPilha(0) + tamPilha(1));
}

/* Remove dado do início da fila */

tDado tiraFila (void) {
  if (pilhaVazia (1))
    while (! pilhaVazia (0))
      poePilha (1, tiraPilha (0));
  return (tiraPilha (1));
}

/* Diz se a fila está vazia */

Bool filaVazia (void) {
  return (pilhaVazia(0) && pilhaVazia(1));
}
```

Exercício 5.8

Solução 1

```
/* Poe dado no inicio do deque */
void poeInicioDeque (tDado dado) {
  while (! pilhaVazia (0))
    poePilha (1, tiraPilha (0));
  poePilha (1, dado); }

/* Tira dado do inicio do deque */
tDado tiraInicioDeque (void) {
  while (! pilhaVazia (0))
    poePilha (1, tiraPilha (0));
  return (tiraPilha (1)); }

/* Retorna o dado no inicio */
/* do deque sem removê-lo   */
tDado inicioDeque (void) {
  while (! pilhaVazia (0))
    poePilha (1, tiraPilha (0));
  return (topoPilha (1)); }

/* Retorna o tamanho do deque */
int tamDeque (void) {
  return (tamPilha(0)+tamPilha(1));
}
```

```
/* Poe dado no fim do deque */
void poeFimDeque (tDado dado) {
  while (! pilhaVazia (1))
    poePilha (0, tiraPilha (1));
  poePilha (0, dado); }

/* Tira dado do fim do deque */
tDado tiraFimDeque (void) {
  while (! pilhaVazia (1))
    poePilha (0, tiraPilha (1));
  return (tiraPilha (0)); }

/* Retorna o dado no fim  */
/* do deque sem removê-lo */
tDado fimDeque (void) {
  while (! pilhaVazia (1))
    poePilha (0, tiraPilha (1));
  return (topoPilha (0)); }

/* Diz se o deque está vazio */
Bool dequeVazio (void) {
  return (pilhaVazia(0) && pilhaVazia(1));
}
```

Solução 2

```
/* Poe dado no inicio do deque */

void poeInicioDeque (tDado dado) {
  while (! pilhaVazia (1))
    poePilha (0, tiraPilha (1));
  poePilha (0, dado); }

/* Tira dado do inicio do deque */

tDado tiraInicioDeque (void) {
  while (! pilhaVazia (1))
    poePilha (0, tiraPilha (1));
  return (tiraPilha (0)); }

/* Retorna o dado no inicio */
/* do deque sem removê-lo   */

tDado inicioDeque (void) {
  while (! pilhaVazia (1))
    poePilha (0, tiraPilha (1));
  return (topoPilha (0)); }

/* Retorna o tamanho do deque */

int tamDeque (void) {
  return (tamPilha(1)+tamPilha(0));
}
```

```
/* Poe dado no fim do deque */

void poeFimDeque (tDado dado) {
  while (! pilhaVazia (0))
    poePilha (1, tiraPilha (0));
  poePilha (1, dado); }

/* Tira dado do fim do deque */

tDado tiraFimDeque (void) {
  while (! pilhaVazia (0))
    poePilha (1, tiraPilha (0));
  return (tiraPilha (1)); }

/* Retorna o dado no fim  */
/* do deque sem removê-lo */

tDado fimDeque (void) {
  while (! pilhaVazia (0))
    poePilha (1, tiraPilha (0));
  return (topoPilha (1)); }

/* Diz se o deque está vazio */

Bool dequeVazio (void) {
  return (pilhaVazia(1) &&
          pilhaVazia(0)); }
```

Exercício 5.9

```
/* Insere dado na pilha */

void poePilha (tDado dado) {
  poeFila (dado); }

/* Remove dado da pilha */

tDado tiraPilha (void) {
  int t = tamFila();
  while (t-- > 1)
    poeFila (tiraFila());
  return (tiraFila()); }

/* Retorna o tamanho da pilha */

int tamPilha (void) {
  return (tamFila()); }
```

```
/* Retorna o dado no topo */
/* da pilha sem removê-lo */

tDado topoPilha (void) {
  tDado dado;
  int t = tamFila();
  while (t-- > 1)
    poeFila (tiraFila());
  dado = frenteFila();
  poeFila (tiraFila());
  return (dado); }

/* Diz se a pilha está vazia */

Bool pilhaVazia (void) {
  return (filaVazia()); }
```

6. Complexidade

Exercício 6.1

- É necessário implementar o algoritmo, o que pode ser difícil ou oneroso.
- Se as entradas analisadas forem tendenciosas, os resultados serão enganadores.
- Para comparar dois algoritmos, devem-se replicar as condições experimentais (mesmo hardware e mesmo ambiente de execução).

Exercício 6.2

- Pode ser difícil caracterizar o caso médio.
- Exige prévio conhecimento da distribuição de probabilidades das diferentes entradas, normalmente desconhecida.
- O cálculo da expressão matemática resultante frequentemente é de tratamento complexo.
- Muitas vezes o caso médio é quase tão ruim quanto o pior caso.

Exercício 6.3

- Equivalência imperfeita com processadores reais.
- Simplificação às vezes irrealista do algoritmo.
- Um algoritmo simples pode ser difícil de analisar.
- A análise pode exigir ferramentas matemáticas.

Exercício 6.4

1) $\mathrm{T}(n) = \mathrm{O}(n)$
2) $\mathrm{T}(n) = \mathrm{O}(\log(n))$
3) $\mathrm{T}(n) = \mathrm{O}(n^2)$
4) $\mathrm{T}(n) = \mathrm{O}(n)$
5) $\mathrm{T}(n) = \mathrm{O}(n^2)$

Exercício 6.5

A	B	C	D	E	F	G	H
4	7	2	6	1	5	3	8

Exercício 6.6

A	B	C	D	E	F	G	H
5	8	4	3	7	6	2	1

Exercício 6.7

A	B	C	D	E	F	G	H	I	J
8	3	6	4	7	10	1	9	2	5

Exercício 6.8

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	F	F	F	V	F	V	F	F	F	V	F	V	F	F	F	V	F

Exercício 6.9

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V	V

Exercício 6.10

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
V	V	F	F	F	F	V	V	V	V	V	V	F	V	V	F	F	F	F	V

Exercício 6.11

O problema consiste em definir c e $n_0 > 0 \mid 0 \leq an + b \leq cn^2$ para todo $n \geq n_0$.
Dividindo a inequação acima por n^2 obtemos $0 \leq {}^{a}/_{n} + {}^{b}/_{n^2} \leq c$.
Fazendo $n_0 = 1$ obtemos $0 \leq a + b \leq c$ que é satisfeita por $c = a + |b|$.
A existência de valores para c e n_0 que satisfazem a definição de $O(g(n))$ prova que $an + b$ é $O(n^2)$.

Exercício 6.12

Trecho 1	Trecho 2	Trecho 3
$T(n) = O(n)$	$T(n) = O(n^2)$	$T(n) = O(n)$
Trecho 4	**Trecho 5**	**Trecho 6**
$T(n) = O(n^3)$	$T(n) = O(n^2)$	$T(n) = O(n^4)$
Trecho 7	**Trecho 8**	**Trecho 9**
$T(n) = O(n^2)$	$T(n) = O(n \log n)$	$T(n) = O(n)$
Trecho 10	**Trecho 11**	**Trecho 12**
$T(n) = O(\log^2 n)$	$T(n) = O(n^3 \log n)$	$T(n) = O(n \log^2 n)$

Exercício 6.13

	Inserção	Remoção
1	$O(1)$	$O(1)$
2	$O(1)$	$O(n)$
3	$O(n)$	$O(n)$
4	$O(1)$	$O(1)$
5	$O(n)$	$O(n)$
6	$O(1)$	$O(1)$
7	$O(1)$	$O(n)$

	Inserção	Remoção
8	$O(n)$	$O(n)$
9	$O(1)$	$O(1)$
10	$O(1)$	$O(1)$
11	$O(1)$	$O(1)$
12	$O(1)$	$O(1)$
13	$O(1)$	$O(n)$
14	$O(1)$	$O(1)$

	Inserção	Remoção
15	$O(n)$	$O(1)$
16	$O(1)$	$O(1)$
17	$O(n)$	$O(1)$
18	$O(1)$	$O(n)$
19	$O(1)$	$O(1)$
20	$O(n)$	$O(n)$
21	$O(1)$	$O(1)$

	Inserção	Remoção
22	$O(1)$	$O(1)$
23	$O(1)$	$O(1)$
24	$O(1)$	$O(1)$
25	$O(1)$	$O(1)$
26	$O(1)$	$O(n)$
27	$O(n)$	$O(1)$

Exercício 6.14

Algoritmo: Bubble Sort		
Linha	**Melhor caso**	**Pior caso**
1	$2n + 3$	$2n + 3$
2	$n^2 + 3n$	$n^2 + 3n$
3	$2n^2 - 2n$	$2n^2 - 2n$
4	0	$(3n^2 - 3n)/2$
5	0	$2n^2 - 2n$
6	0	$n^2 - n$
Total	$3n^2 + 3n + 3$	$(15n^2 - 3n + 6)/2$

A solução detalhada deste exercício é apresentada no tutorial ***Complexidade de Algoritmos e Modelo RAM de Computação***.

Exercício 6.15

Algoritmo: Selection Sort		
Linha	**Melhor caso**	**Pior caso**
1	$2n + 1$	$2n + 1$
2	$n^2 + 3n - 4$	$n^2 + 3n - 4$
3	$(3n^2 - 3n)/2$	$(3n^2 - 3n)/2$
4	0	$n^2 - n$
5	0	$(3n^2 - 3n)/2$
6	0	$n^2 - n$
Total	$(5n^2 + 7n - 6)/2$	$6n^2 - 3n$

A solução detalhada deste exercício é apresentada no tutorial ***Complexidade de Algoritmos e Modelo RAM de Computação***.

Exercício 6.16

Algoritmo: Insertion Sort		
Linha	**Melhor caso**	**Pior caso**
1	$2n + 1$	$2n + 1$
2	$2n - 2$	$2n - 2$
3	$2n - 2$	$2n - 2$
4	$4n - 4$	$2n^2 + 2n - 4$
5	0	$2n^2 - 2n$
6	0	$n^2 - n$
7	$3n - 3$	$3n - 3$
Total	$13n - 10$	$5n^2 + 8n - 10$

A solução detalhada deste exercício é apresentada no tutorial ***Complexidade de Algoritmos e Modelo RAM de Computação***.

Exercício 6.17

Algoritmo: Stupid Sort		
Linha	**Melhor caso**	**Pior caso**
1	$2n+2$	$2n+2$
2	$2n$	n^2+n
3	$2n$	n^2+n
4	$2n$	$2n$
5	n	n
6	$2n^2+2n$	$2n^2+2n$
7	$5n^2$	$5n^2$
8	n^2-n	0
9	$2n$	$2n$
10	n^2+n	n^2+n
11	n^2+n	n^2+n
12	$2n$	$2n$
13	n	n
14	$2n+2$	$2n+2$
15	$3n$	$3n$
Total	$10n^2+22n+4$	$11n^2+21n+4$

A solução detalhada deste exercício é apresentada no tutorial ***Complexidade de Algoritmos e Modelo RAM de Computação***.

Exercício 6.19

Algoritmo: Transpor		
Linha	**Melhor caso**	**Pior caso**
1	$2n+1$	$2n+1$
2	n^2+2n-3	n^2+2n-3
3	$3(n^2-n)/2$	$3(n^2-n)/2$
4	0	n^2-n
5	0	$3(n^2-n)/2$
6	0	n^2-n
Total	$(5n^2+5n-4)/2$	$6n^2-n-2$

A solução detalhada deste exercício é apresentada no tutorial ***Complexidade de Algoritmos e Modelo RAM de Computação***.

Exercício 6.20

Algoritmo: Girar		
Linha	**Melhor caso**	**Pior caso**
1	$n+4$	$n+4$
2	$(n^2+4n)/2$	$(n^2+4n)/2$
3	$4n^2/4$	$4n^2/4$
4	$9n^2/4$	$9n^2/4$
5	$9n^2/4$	$9n^2/4$
6	$5n^2/4$	$5n^2/4$
7	$2n^2/4$	$2n^2/4$
Total	$(31n^2+12n+16)/4$	$(31n^2+12n+16)/4$

A solução detalhada deste exercício é apresentada no tutorial ***Complexidade de Algoritmos e Modelo RAM de Computação***.

Exercício 6.21

Devemos comparar os tempos de execução dos algoritmos a fim de determinar se existe pelo menos um tamanho de entrada n para o qual ambos executam no mesmo tempo. Para isso, igualamos $T_1(n)$ e $T_2(n)$ e calculamos a raiz do polinômio resultante:

$$T_1(n) = T_2(n) \Rightarrow n^2 - 100n + 4000 = 20n + 2000 \Rightarrow n^2 - 120n + 2000 = 0$$
$$\Rightarrow n_1 = 20 \text{ e } n_2 = 100$$

O fato das duas raízes serem reais significa que a reta T_2 corta a parábola T_1 em dois pontos. O coeficiente do termo de mais alta ordem de T_1 é positivo, de forma que seu gráfico é uma parábola com a concavidade voltada para cima. Isso significa que, entre n_1 e n_2, T_1 é menor do que T_2. Ou seja:

$T_2 < T_1$ para $n < 20$

$T_1 < T_2$ para $20 < n < 100$

$T_2 < T_1$ para $n > 100$

Como n representa o tamanho da entrada, temos que n não pode ser negativo. Concluímos que:

- o algoritmo 2 é mais rápido que o algoritmo 1 para entradas de tamanho menor que 20,
- os dois algoritmos executam no mesmo tempo para entradas de tamanho igual a 20,
- o algoritmo 1 é mais rápido que o algoritmo 2 para entradas de tamanho entre 20 e 100,
- os dois algoritmos executam no mesmo tempo para entradas de tamanho igual a 100, e
- o algoritmo 2 é mais rápido que o algoritmo 1 para entradas de tamanho maior que 100.

Portanto, as respostas às perguntas são:

1) Não. 2) Não. 3) Sim, o algoritmo 1. 4) Não. 5) Sim, o algoritmo 2.

Exercício 6.22

Devemos comparar os tempos de execução dos algoritmos a fim de determinar se existe pelo menos um tamanho de entrada n para o qual ambos executam no mesmo tempo. Para isso, igualamos $T_1(n)$ e $T_2(n)$ e calculamos a raiz do polinômio resultante:

$$T_1(n) = T_2(n) \Rightarrow n^2 - 600n + 150000 = 2n^2 - 1200n + 200000 \Rightarrow n^2 - 600n + 50000 = 0$$
$$\Rightarrow n_1 = 100 \text{ e } n_2 = 500$$

As duas raízes reais significam que as parábolas T_1 e T_2 se cruzam em dois pontos. Os coeficientes dos termos de mais alta ordem de T_1 e de T_2 são positivos, de forma que seus gráficos são parábolas com concavidades voltadas para cima. Para determinarmos se T_1 é maior ou menor que T_2 entre n_1 e n_2, calculamos $T_1(n)$ e $T_2(n)$ para um n qualquer nesse intervalo, por exemplo $n = 200$:

$$T_1(200) = 200^2 - 600 \times 200 + 150000 = 70000$$

$$T_2(200) = 2 \times 200^2 - 1200 \times 200 + 200000 = 40000$$

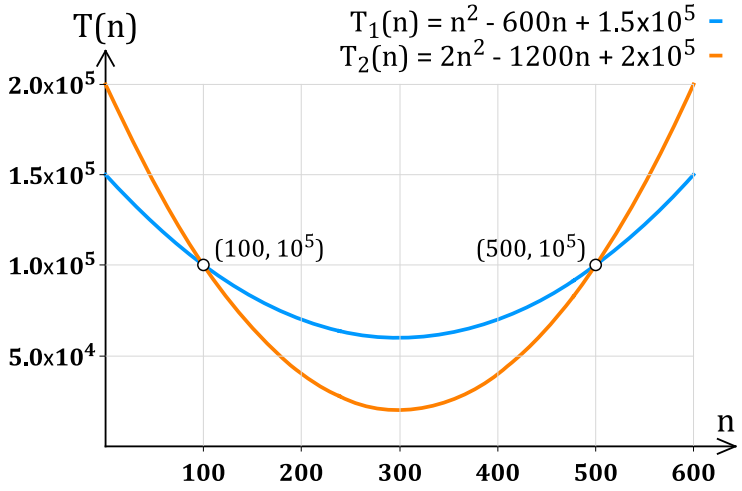
Isso significa que, entre n_1 e n_2, T_1 é maior do que T_2. Ou seja:

$T_1 < T_2$ para $n < 100$

$T_2 < T_1$ para $100 < n < 500$

$T_1 < T_2$ para $n > 500$

As conclusões acima podem ser obtidas a partir dos gráficos de $T_1(n)$ e $T_2(n)$ mostrados a seguir.



Como n representa o tamanho da entrada, temos que n não pode ser negativo. Concluímos que:

- o algoritmo 1 é mais rápido que o algoritmo 2 para entradas de tamanho menor que 100,
- os dois algoritmos executam no mesmo tempo para entradas de tamanho igual a 100,
- o algoritmo 2 é mais rápido que o algoritmo 1 para entradas de tamanho entre 100 e 500,
- os dois algoritmos executam no mesmo tempo para entradas de tamanho igual a 500, e
- o algoritmo 1 é mais rápido que o algoritmo 2 para entradas de tamanho maior que 500.

Assim, as respostas às perguntas são:

1) $n < 100$ e $n > 500$
2) $100 < n < 500$
3) $n = 100$ e $n = 500$

Exercício 6.23

Devemos comparar os tempos de execução dos algoritmos a fim de determinar se existe pelo menos um tamanho de entrada n para o qual ambos executam no mesmo tempo. Para isso, igualamos $T_1(n)$ e $T_2(n)$ e calculamos a raiz do polinômio resultante:

$$T_1(n) = T_2(n) \Rightarrow n^3 - 59n^2 + 1060n - 4700 = n^2 - 40n + 1300$$
$$\Rightarrow n^3 - 60n^2 + 1100n - 6000 = 0$$

Usando a dica no enunciado do problema, podemos escrever:

$$a + b + c = 60$$
$$ab + bc + ac = 1100$$
$$abc = 6000$$

A solução algébrica desse sistema de equações é simples, porém trabalhosa. Uma alternativa é observar que as três raízes a, b e c são múltiplos de 10 e menores que 100, já que:

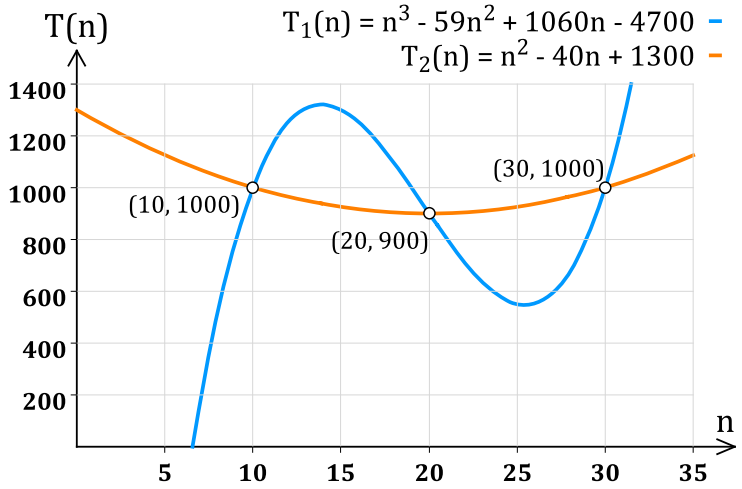
- a soma das três, 60, tem um zero,
- a soma dos produtos duas a duas, 1100, tem dois zeros, e
- o produto, 6000, tem três zeros.

Usando três variáveis auxiliares x, y e z tais que $x = a/10$, $y = b/10$ e $z = c/10$, obtemos

$$x + y + z = 6$$
$$xy + yz + xz = 11$$
$$xyz = 6$$

Daí pode-se facilmente deduzir que $x = 1$, $y = 2$ e $z = 3$, logo $a = 10$, $b = 20$ e $c = 30$.

As três raízes serem reais significam que as curvas de T_1 e de T_2 se cruzam em três pontos. O coeficiente do termo de mais alta ordem de T_2 é positivo, logo seu gráfico é uma parábola com a concavidade voltada para cima. O coeficiente do termo de mais alta ordem de T_1 também é positivo, logo seu gráfico pode ser representado esquematicamente como se vê a seguir:



Isso significa que, entre $n = 10$ e $n = 20$, T_1 é menor do que T_2. Ou seja:

$T_1 < T_2$ para $n < 10$
$T_2 < T_1$ para $10 < n < 20$
$T_1 < T_2$ para $20 < n < 30$
$T_1 < T_2$ para $n > 30$

Como n representa o tamanho da entrada, temos que n não pode ser negativo. Concluímos que:

- o algoritmo 1 é mais rápido que o algoritmo 2 para entradas de tamanho menor que 10,
- os dois algoritmos executam no mesmo tempo para entradas de tamanho igual a 10,
- o algoritmo 2 é mais rápido que o algoritmo 1 para entradas de tamanho entre 10 e 20,
- os dois algoritmos executam no mesmo tempo para entradas de tamanho igual a 20,
- o algoritmo 1 é mais rápido que o algoritmo 2 para entradas de tamanho entre 20 e 30,
- os dois algoritmos executam no mesmo tempo para entradas de tamanho igual a 30, e
- o algoritmo 2 é mais rápido que o algoritmo 1 para entradas de tamanho maior que 30.

Assim, as respostas às perguntas são:

1) $n < 10$ e $20 < n < 30$
2) $10 < n < 20$ e $n > 30$
3) $n = 10, n = 20$ e $n = 30$

Exercício 6.24

Definir c_1, c_2 e $n_0 > 0 \mid 0 \leq c_1 n^2 \leq 2n^2 - {}^{n}/_{2} \leq c_2 n^2$ para todo $n \geq n_0$

$0 \leq c_1 n^2 \leq 2n^2 - {}^{n}/_{2} \leq c_2 n^2 \Rightarrow 0 \leq c_1 \leq f'(n) \leq c_2$ onde $f'(n) = 2 - {}^{1}/_{2n}$

	$0 \leq c_1 \leq f'(n)$	$c_2 \geq f'(n)$
$n \geq 1/3$	$0 \leq c_1 \leq 1/2$	$c_2 \geq 1/2$
$n \geq 1/2$	$0 \leq c_1 \leq 1$	$c_2 \geq 1$
$n \geq 1$	$0 \leq c_1 \leq 3/2$	$c_2 \geq 3/2$
$n \geq 2$	$0 \leq c_1 \leq 7/4$	$c_2 \geq 7/4$
etc		

O gráfico ao lado mostra que

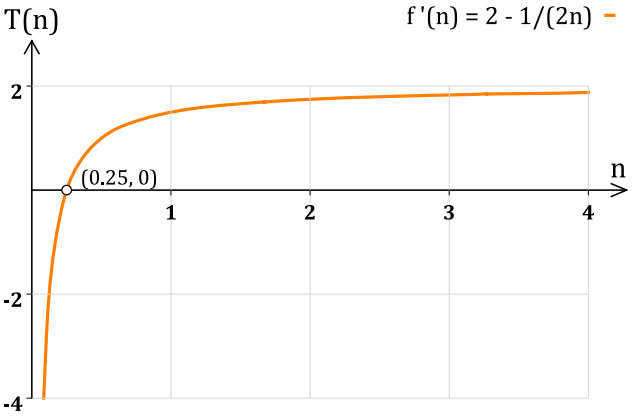
$$\lim_{n\to\infty} 2 - \frac{1}{2n} = 8$$

Segundo a definição de θ,

$$c_1 \le f'(n) \le c_2$$

portanto

- a faixa de valores de c_1 que satisfazem a definição de θ é $(0, 2)$ e
- a faixa de valores de c_2 que satisfazem a definição de θ é $[2, +\infty)$.



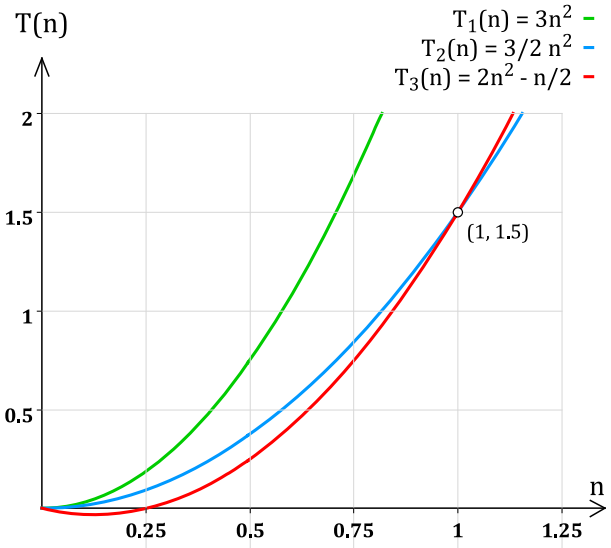
A inequação é válida para vários conjuntos de constantes; qualquer um prova que $2n^2 - {}^{n}/_{2} = \theta(n^2)$.

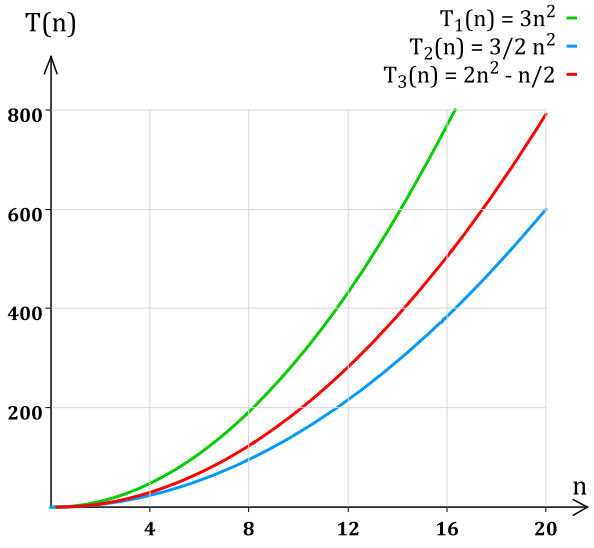
Exemplos:

$n_0 = 1/3,\ c_1 = 1/2$ e $c_2 = 2$
$n_0 = 1/3,\ c_1 = 1/4$ e $c_2 = 2$
$n_0 = 1/2,\ c_1 = 1/2$ e $c_2 = 2$
$n_0 = 1,\ \ c_1 = 3/2$ e $c_2 = 3$
$n_0 = 2,\ \ c_1 = 1\ \ $ e $c_2 = 4\ \ $ etc.

A seguir, a representação gráfica da solução para $n_0 = 1$, $c_1 = 3/2$ e $c_2 = 3$

$$0 \le \frac{3}{2}n^2 \le 2n^2 - \frac{n}{2} \le 3n^2$$





Exercício 6.25

Para determinar as faixas de valores aceitáveis para c_1, c_2 e n_0 utilizando o método analítico, devemos analisar as raízes de algumas equações e seus sinais em vários intervalos. Começamos com a expressão obtida a partir da definição de θ:

$$c_1 n^3 \le 10n^3 - 300n^2 + 2000n \le c_2 n^3$$

Dividindo por n^3 obtemos $c_1 \le f'(n) \le c_2$ onde $f'(n) = 10 - 300/n + 2000/n^2$

As raízes de $f'(n)$ são as mesmas de $f(n)$. Podemos deduzi-las facilmente fatorando $f(n)$:

$$f(n) = 10n^3 - 300n^2 + 2000n = 10n(n^2 - 30n + 200) = 10n(n - 10)(n - 20)$$

cujas raízes são 0, 10 e 20. Logo, as raízes de $f'(n)$ são 10 e 20; excluímos o 0 devido aos denominadores na expressão de $f'(n)$. Observamos que

$$\lim_{n\to\infty} f'(n) = 10$$

Concluímos que $c_1 < 10$, $c_2 \geq 10$ e $n_0 > 20$.

Note que n_0 não pode ser ≥ 20 pois $n_0 = 20$, por ser uma raiz de $f'(n)$, resultaria em $c_1 \leq 0 \leq c_2$, o que não é permitido pois a definição de θ afirma que c_1, c_2 e $n_0 > 0$.

Podemos agora responder as duas primeiras perguntas:

a) O menor valor inteiro de n_0 que nos permite provar que $f(n) = \theta(n^3)$ é 9.
b) O maior valor inteiro de c_1 que nos permite provar que $f(n) = \theta(n^3)$ é 9.

Terceira pergunta: Se usarmos $c_1 = 5$ e $c_2 = 20$ na definição de θ, obtemos

$$5n^3 \leq 10n^3 - 300n^2 + 2000n \leq 20n^3$$

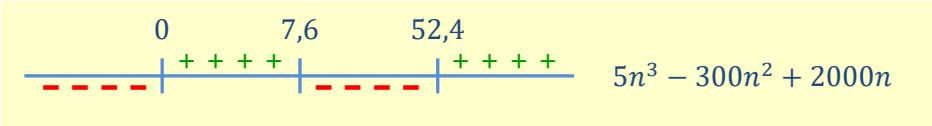
Resolvendo a primeira inequação:

$$5n^3 \leq 10n^3 - 300n^2 + 2000n \Rightarrow 5n^3 - 300n^2 + 2000n \geq 0$$

Dividindo por $5n$ obtemos $n^2 - 60n + 400n \geq 0$ cujas raízes são $30 \pm 10\sqrt{5}$, portanto

$$n_1 = 0, n_2 = 7{,}6 \text{ e } n_3 = 52{,}4.$$

O coeficiente de $n^3 = 5 > 0$, de modo que o sinal do polinômio de 3º grau pode ser assim resumido:



A solução dessa inequação é $0 \leq n \leq 7{,}6$ e $n \geq 52{,}4$

Contudo, a definição de θ afirma que $0 < n_0 < n$, portanto a solução da primeira inequação é

$$0 < n \leq 7{,}6 \text{ e } n \geq 52{,}4$$

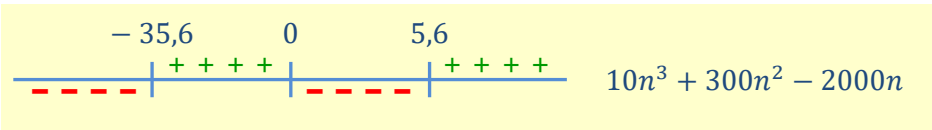
Resolvendo a segunda inequação:

$$10n^3 - 300n^2 + 2000n \leq 20n^3 \Rightarrow 10n^3 + 300n^2 - 2000n \geq 0$$

Dividindo por $10n$ obtemos $n^2 + 30n - 200n \geq 0$ cujas raízes são $-15 \pm 5\sqrt{17}$, portanto

$$n_1 = 0, n_2 = 5{,}6 \text{ e } n_3 = -35{,}6.$$

O coeficiente de $n^3 = 10 > 0$, de modo que o sinal do polinômio de 3º grau pode ser assim resumido:



A solução dessa inequação é $-35{,}6 \leq n \leq 0$ e $n \geq 5{,}6$

Contudo, a definição de θ afirma que $0 < n_0 < n$, logo a solução da segunda inequação é apenas $n \geq 5{,}6$.

Juntando os resultados parciais:

A interseção entre as soluções das duas inequações é

$$5{,}6 \leq n \leq 7{,}6 \text{ e } n \geq 52{,}4$$

Assim, o menor valor inteiro de n_0 que permite provar que $f(n) = \theta(n^3)$ é 53.

Exercício 6.26

Problema: definir c_1, c_2 e $n_0 > 0 \mid 0 \leq c_1 n^3 \leq 8n^3 - 300n^2 + 40n \leq c_2 n^3$ para todo $n \geq n_0$.

Dividindo por n^3 obtemos $c_1 \leq f'(n) \leq c_2$ onde $f'(n) = 8 - 300/n + 40/n^2$.

1) O gráfico 1 mostra que a maior raiz da função $f'(n) = 8 - 300/n + 40/n^2$ é $37{,}366$. Segundo a definição de θ, $0 \leq c_1 \leq f'(n)$, portanto $n \geq 37{,}366$ para que $f'(n) \geq 0$ para todo $n \geq n_0$. Logo, a faixa de valores de n_0 que satisfazem a definição de θ é $[37.366, +\infty)$.

2) O gráfico 2 mostra que $\lim_{n\to\infty} f'(n) = 8$. Segundo a definição de θ, $c_1 \leq f'(n)$, portanto a faixa de valores de c_1 que satisfazem a definição de θ é $(0, 8)$.

3) O gráfico 2 mostra que $\lim_{n\to\infty} f'(n) = 8$. Segundo a definição de θ, $c_2 \geq f'(n)$, portanto a faixa de valores de c_2 que satisfazem a definição de θ é $[8, +\infty)$.

4) Agora c_1, c_2 e $n_0 > 1$ (inteiros maiores que zero):

   $c_1 \leq f'(n) \Rightarrow 1 \leq 8 - 300/n + 40/n^2 \Rightarrow n^2 \leq 8n^2 - 300n + 40 \Rightarrow 7n^2 - 300n + 40 \geq 0$

   O gráfico 3 mostra que a maior raiz desse polinômio é $42{,}72$. Como n_0 é um número inteiro, o menor valor de n_0 que satisfaz a definição de θ para quaisquer inteiros c_1 e $c_2 > 0$ é 43. Logo, a faixa de valores de n_0 que satisfazem a definição de θ é $[43, +\infty)$.

5) O gráfico 2 mostra que $\lim_{n\to\infty} f'(n) = 8$. Segundo a definição de θ, $c_1 \leq f'(n)$, portanto a faixa de valores de c_1 que satisfazem a definição de θ é $[1, 7]$.

6) O gráfico 2 mostra que $\lim_{n\to\infty} f'(n) = 8$. Segundo a definição de θ, $c_2 \geq f'(n)$, portanto a faixa de valores de c_2 que satisfazem a definição de θ é $[8, +\infty)$.

7) Definir c_1, c_2 e $n_0 > 0 \mid 0 \leq c_1 n^3 \leq 8n^3 - 300n^2 + 40n \leq c_2 n^3$ para todo $n \geq n_0$

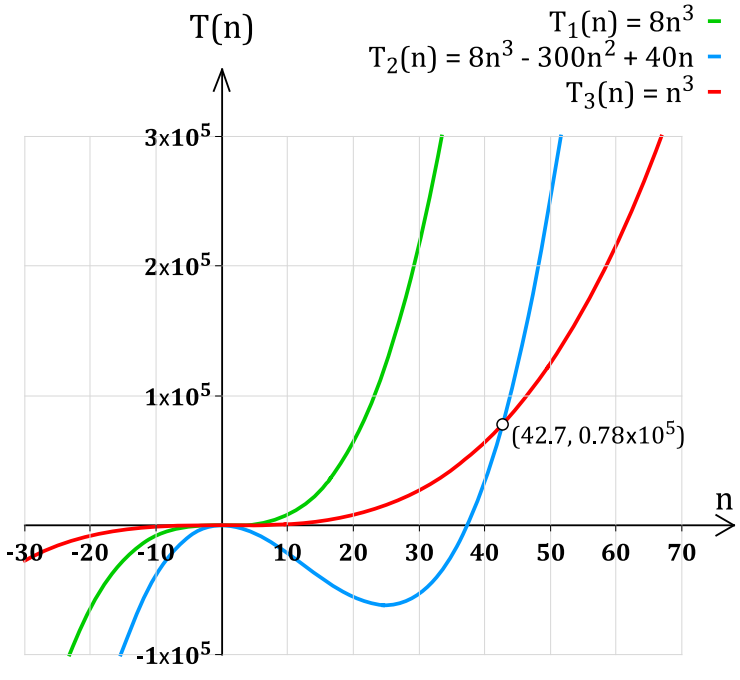
   $c_1 n^3 \leq 8n^3 - 300n^2 + 40n \leq c_2 n^3 \Rightarrow c_1 \leq 8 - 300/n + 40/n^2 \leq c_2$

Primeira parte: $c_1 \leq 8 - 300/n + 40/n^2$	**Segunda parte:** $c_2 \geq 8 - 300/n + 40/n^2$	
$n = 37 \Rightarrow c_1 \leq -0{,}079$	$n \leq 37 \Rightarrow c_2 \geq -0{,}079$	n_0 não pode ser ≤ 37 pois c_1 deve ser ≥ 1
$n = 40 \Rightarrow c_1 \leq 0{,}525$	$n \leq 40 \Rightarrow c_2 \geq 0{,}525$	n_0 não pode ser ≤ 40 pois c_1 deve ser ≥ 1
$n = 42 \Rightarrow c_1 \leq 0{,}880$	$n \leq 42 \Rightarrow c_2 \geq 0{,}880$	n_0 não pode ser ≤ 42 pois c_1 deve ser ≥ 1
$n = 43 \Rightarrow c_1 \leq 1{,}045$	$n \leq 43 \Rightarrow c_2 \geq 1{,}045$	$n_0 = 43, \quad c_1 = 1, \ c_2 = 8$
$n = 50 \Rightarrow c_1 \leq 2{,}016$	$n \leq 50 \Rightarrow c_2 \geq 2{,}016$	$n_0 = 50, \quad c_1 = 2, \ c_2 = 8$
$n = 100 \Rightarrow c_1 \leq 5{,}0004$	$n \leq 50 \Rightarrow c_2 \geq 5{,}0004$	$n_0 = 100, \quad c_1 = 5, \ c_2 = 9$
$n = 300 \Rightarrow c_1 \leq 7{,}00\cdots$	$n \leq 300 \Rightarrow c_2 \geq 7{,}00\cdots$	$n_0 = 300, \quad c_1 = 6, \ c_2 = 10$
$n = 3000 \Rightarrow c_1 \leq 7{,}90\cdots$	$n \leq 3000 \Rightarrow c_2 \geq 7{,}90\cdots$	$n_0 = 3000, \ c_1 = 7, \ c_2 = 11$

etc.

Todos os conjuntos de constantes c_1, c_2 e n_0 acima tornam a inequação válida, provando que $8n^3 - 300n^2 + 40n = \theta(n^3)$

8) O gráfico a seguir mostra que $8n^3 - 300n^2 + 40n = \theta(n^3)$. Foram usadas as constantes $n_0 = 43$, $c_1 = 1$ e $c_2 = 8$


T(n)
$T_1(n) = 8n^3$
$T_2(n) = 8n^3 - 300n^2 + 40n$
$T_3(n) = n^3$
3x10^5
2x10^5
1x10^5
-1x10^5
(42.7, 0.78x10^5)
n
-30
-20
-10
10
20
30
40
50
60
70

7. Ordenação

Exercício 7.1

Um algoritmo de ordenação **estável** é aquele que mantém a ordem relativa dos dados com chaves iguais. Um algoritmo de ordenação ***in place*** que ordena os dados na própria estrutura onde se encontram, não necessitando de memória extra além dos dados sendo ordenados. Tal algoritmo usa no máximo memória da ordem de $O(1)$ ou $O(\log n)$ além dos dados sendo ordenados.

Exercício 7.2

O pior caso ocorre quando, em cada chamada recursiva... *(respostas equivalentes)*

- O pivô é o maior ou o menor elemento da sequência e não possui duplicatas.
- O pivô é escolhido de forma que o particionamento resulte um uma subsequência vazia.

O melhor caso ocorre quando, em cada chamada recursiva... *(respostas equivalentes)*

- O pivô é o elemento intermediário da sequência.
- O pivô é a mediana dos elementos da sequência.
- O pivô é o mais próximo da média aritmética dos elementos da sequência.
- O pivô é escolhido tal que o particionamento resulte em subsequências de tamanhos iguais.

Exercício 7.3

Distinção: *(respostas equivalentes)*

- Não há distinção entre o melhor e pior caso.
- O tempo de execução do Merge Sort é igual para o melhor caso e o pior caso.

Justificativa: *(respostas equivalentes)*

- O número de chamadas recursivas e comparações entre dados independe da qualidade da entrada.
- Todo o funcionamento do merge sort independe do estado de ordenação prévia dos dados.

Exercício 7.4

1) Sim. O número de chamadas recursivas e de comparações entre dados é independente da qualidade da entrada. O funcionamento do Merge Sort independe do estado de ordenação prévia dos dados.
2) Não. O número de chamadas recursivas depende dos tamanhos dos dois sub-vetores obtidos após a partição; esses tamanhos, por sua vez, dependem da qualidade do pivô escolhido a cada passo da recursão. Um bom pivô é aquele que resulta em sub-vetores de tamanhos iguais ou aproximados, mas não há garantia que se escolherá sempre um bom pivô a cada passo da recursão.

Exercício 7.5

Método 1: escolha como pivô o primeiro (ou último) elemento da sequência ou *array*.
Observações sobre esse método: *(equivalentes entre si)*

- Tem a maior probabilidade de produzir particionamento desbalanceado da sequência de entrada.
- É uma boa escolha apenas se a entrada estiver razoavelmente desordenada.
- Resulta no pior caso se a entrada já estiver ordenada.

Método 2: escolha como pivô um elemento qualquer (aleatório) da sequência ou *array*.
Observações sobre esse método: *(equivalentes entre si)*

- Raramente resulta no pior caso.
- Frequentemente resulta em um caso médio próximo ao melhor caso.

Método 3: escolha como pivô a moda de três elementos aleatórios da sequência ou *array*.
Observações sobre esse método: *(equivalentes entre si)*

- Tem a maior probabilidade de produzir particionamento balanceado da sequência de entrada.
- Tem a maior probabilidade de se aproximar do melhor caso.

Exercício 7.6

Algoritmo	Tempo		
	Pior caso	Caso médio	Melhor caso
Bubble sort	n^2	n^2	n^2
Bubble sort modificado	n^2	n^2	n
Selection sort	n^2	n^2	n^2
Insertion sort	n^2	n^2	n
Heap sort	$n \log n$	$n \log n$	$n \log n$
Merge sort	$n \log n$	$n \log n$	$n \log n$
Quicksort	n^2	$n \log n$	$n \log n$

Exercício 7.7

1	2	3	4	5	6	7	8	9
F	V	V	F	F	V	F	V	F

8. Árvores

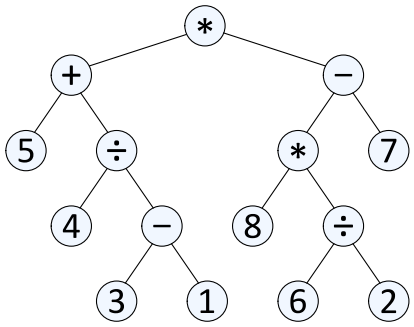
Exercício 8.1

1) A B D H M I N O E J C F G K P Q L R
2) M H D N I O B E J A F C P K Q G L R
3) M H N O I D J E B F P Q K R L G C A
4) A B D H M E I N C F J O G K P L Q R
5) M H D B N I E A O J F C P K G Q L R
6) M H D N I E B O J F P K Q R L G C A

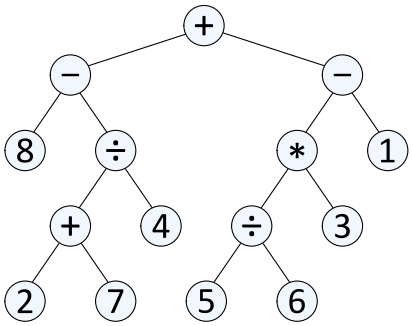
Exercício 8.2

1) É uma árvore binária onde cada nó tem zero ou dois filhos.
2) É uma árvore binária onde todos os níveis têm o número máximo de filhos.
3) É uma árvore binária onde todos os níveis até o penúltimo têm o número máximo de filhos e, no penúltimo nível, todos os nós internos estão à esquerda dos nós externos.
4) Cada nó é visitado três vezes:
   - a primeira vez antes de visitar a sub-árvore da esquerda,
   - a segunda vez após visitar a sub-árvore da esquerda e antes de visitar a sub-árvore da direita, e
   - a terceira vez após visitar a sub-árvore da direita.
5) $O(n)$. Todos os nós são visitados três vezes, logo o tempo de execução é proporcional a n.

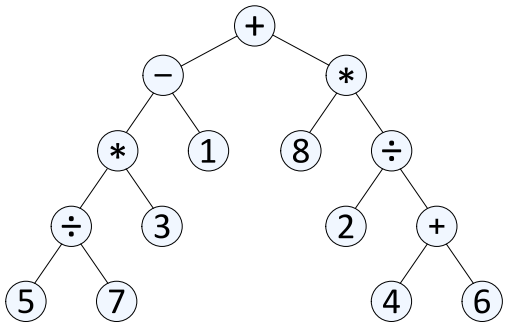
Exercício 8.3



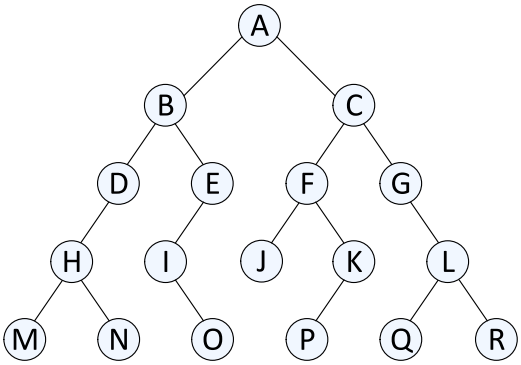
Exercício 8.4



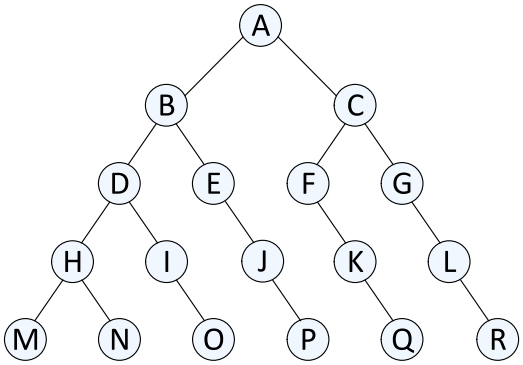
Exercício 8.5



Exercício 8.6



Exercício 8.7



Exercício 8.8

1) ((T − (L ÷ G)) * K) + ((X * J) ÷ ((C − Q) + A))

2) T L G ÷ − K * X J * C Q − A + ÷ +

Exercício 8.9

1) * + ÷ − E C H * A − ÷ G D F B

2) E C − H ÷ A G D ÷ F − * + B *

Exercício 8.10

1) ÷ + 8 * 1 − 4 * 9 7 − ÷ + − 3 8 6 5 2

2) (8 + (1 * (4 − (9 * 7)))) ÷ ((((3 − 8) + 6) ÷ 5) − 2)

Exercício 8.11

1	2	3	4	5	6	7	8	9	10
F	F	F	V	F	F	V	F	V	F

Exercício 8.12

1	2	3	4	5	6	7	8
V	V	V	V	V	V	V	V

Exercício 8.13

```
pNo *ACMP (tNo *pN1, tNo *pN2) {
  int profN1 = 0,
      profN2 = 0;
  tNo *aux1 = pN1,
      *aux2 = pN2;
  while (pai (aux1) != NULL) {
    aux1 = pai (aux1);
    profN1++; }
  while (pai (aux2) != NULL) {
    aux2 = pai (aux2);
    profN2++; }
  while (profN1 > profN2) {
    pN1 = pai (pN1);
    profN1--; }
  while (profN2 > profN1) {
    pN2 = pai (pN2);
    profN2--; }
  while (pN1 != pN2) {
    pN1 = pai (pN1);
    pN2 = pai (pN2); }
  return (pN1);
}
```

Exercício 8.14

Vetor	Árvore binária...			
	própria?	completa?	quase completa?	de busca?
1	Sim	Não	Sim	Não
2	Sim	Não	Não	Não
3	Não	Não	Sim	Não
4	Não	Não	Não	Sim
5	Sim	Não	Não	Não
6	Não	Não	Não	Não
7	Sim	Sim	Não	Não
8	Não	Não	Não	Sim

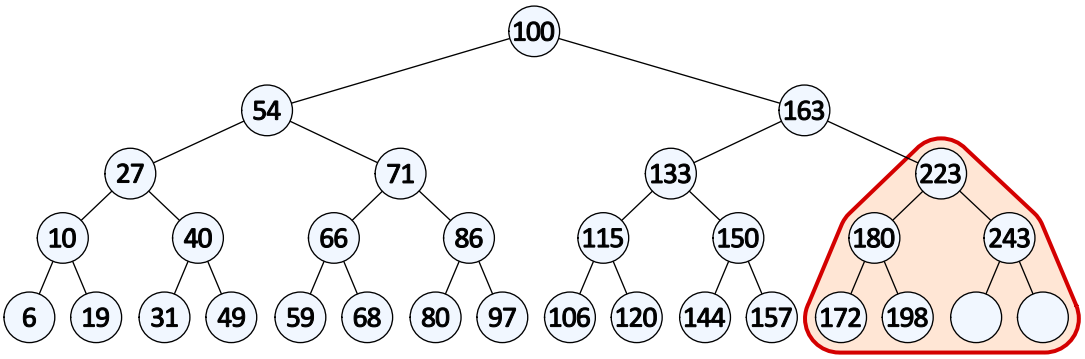
Exercício 8.15

- Inicialize um contador global com 0.
- Inicie o caminhamento de Euler da árvore a partir da raiz.
- Ao visitar um nó v pela esquerda, incremente o contador e guarde esse valor no nó ($v.c_{esq}$).
- Ao visitar um nó v pela direita, guarde o valor do contador no nó ($v.c_{dir}$).
- Ao final do caminhamento, o número de descendentes de um nó v é dado por $v.c_{esq} - v.c_{dir} + 1$.
- Tempo de execução: $O(n)$.

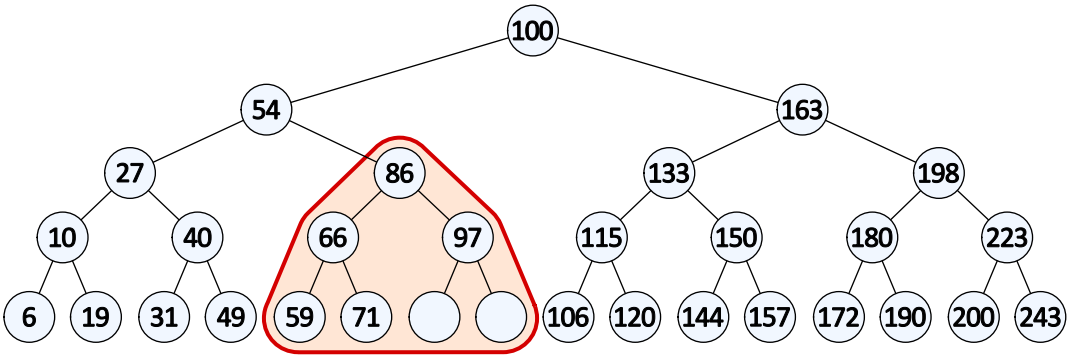
Exercício 8.16

- Inicie o caminhamento de Euler da árvore a partir da raiz.
- Ao visitar um nó v pela esquerda: se o nó é interno, imprima (.
- Ao visitar um nó por baixo: imprima o valor ou o operador armazenado no nó.
- Ao visitar um nó v pela direita: se o nó é interno, imprima).

Exercício 8.17



Exercício 8.18



Exercício 8.19

```
/* Nodos usados para construir a árvore e a lista encadeada */

typedef struct nodo* tNodo;

/* Une dois nodos tal que o 2o. segue após o 1o. Equivale a ajustar */
/* o ponteiro "proximo" do 1o. nodo e o ponteiro "anterior" do 2o.  */

void uneNodos (tNodo nodo1, tNodo nodo2) {
  nodo1->maior = nodo2; nodo2->menor = nodo1;
}

/* Une duas listas circulares duplamente encadeadas e     */
/* retorna um ponteiro para o início da lista resultante. */

tNodo uneListas (tNodo lista1, tNodo lista2) {
  tNodo ultimoLista1, ultimoLista2;

  if (lista1 == NULL) return (lista2);
  if (lista2 == NULL) return (lista1);
  ultimoLista1 = lista1->menor;
  ultimoLista2 = lista2->menor;
  uneNodos (ultimoLista1, lista2);
  uneNodos (ultimoLista2, lista1);
  return (lista1);
}

/* Transforma recursivamente uma árvore binária ordenada numa lista  */
/* circular duplamente encadeada e retorna um ponteiro p/ seu início */

tNodo arvoreParaLista (tNodo raiz) {
  tNodo lista1, lista2;

  if (raiz == NULL) return (NULL);

  /* Chamadas recursivas para transformar as duas sub-árvores */

  lista1 = arvoreParaLista (raiz->menor);
  lista2 = arvoreParaLista (raiz->maior);

  /* Crie com a raiz da árvore uma lista de um único nodo */

  raiz->menor = raiz; raiz->maior = raiz;

  /* Junte as três listas parciais em ordem crescente */

  lista1 = uneListas (lista1, raiz);
  lista1 = uneListas (lista1, lista2);
  return (lista1);
}
```

9. Heaps

Exercício 9.1

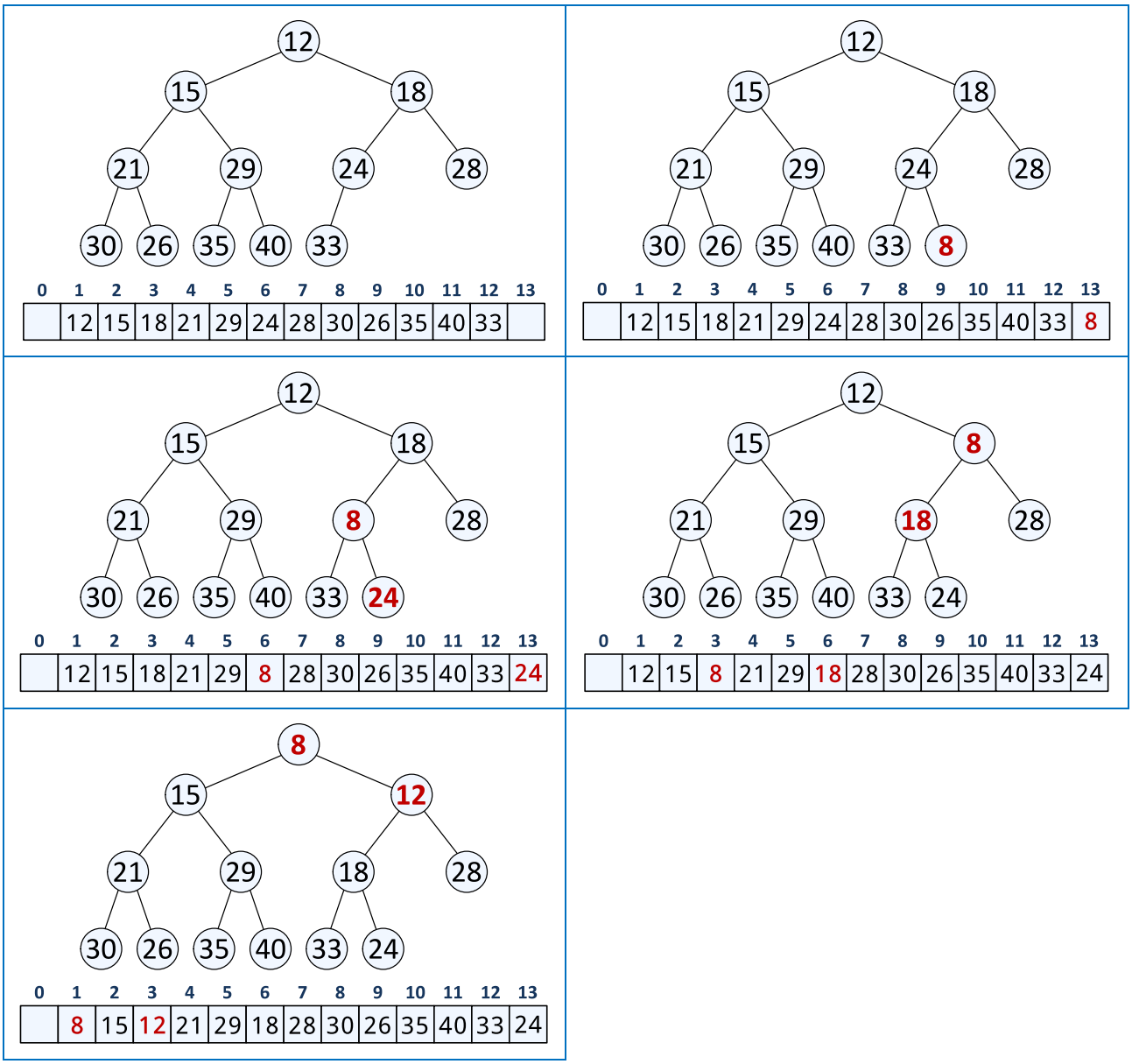
Propriedade relacional: para cada nodo v diferente da raiz, a chave em $\text{chave}(v) \geq \text{chave}(\text{pai}(v))$.

Propriedade estrutural: a árvore binária é completa. Se a árvore tem altura h, então:
1) todos os níveis entre 0 e $h - 1$ possuem o maior número de nodos possível, e
2) todos os nodos no nível h estão à esquerda dos nodos ausentes.

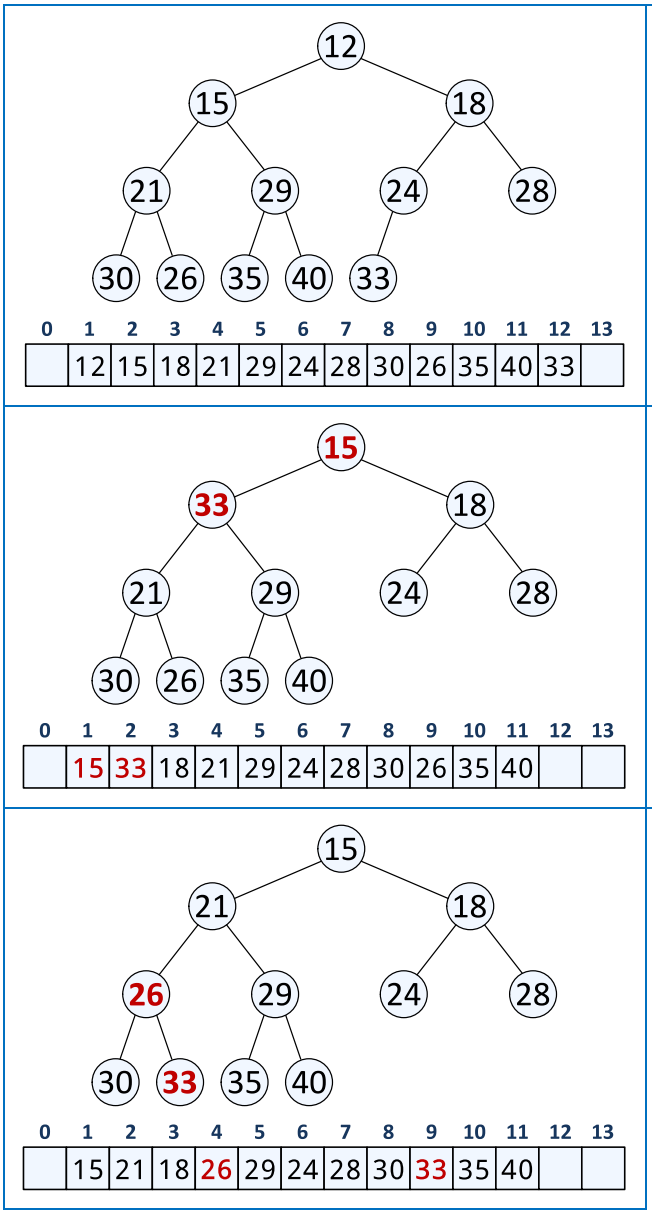
Exercício 9.2

Árvore									
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
E	RE	RE	E	R	H 21	H 14	R	H 21	RE

Exercício 9.3



Exercício 9.4



Exercício 9.5

Árvore							
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
R	RE	H 28	RE	E	RE	H 21	RE

Exercício 9.6

1) Em qualquer um dos nós externos, ou folhas da árvore.
2) Sim. O *heap* deve ser crescente. Exemplo: o *heap* representado pelo vetor (1, 2, 5, 3, 4, 6, 7).
3) Não. Para visitar as chaves em ordem crescente em um percurso infixado, a chave em qualquer nó teria que ser maior que a chave no filho da esquerda e menor que a chave no filho da direita. Isso não é possível em um *heap* com chaves distintas, onde a chave em um nó é sempre menor (ou maior) que as chaves de todos os seus descendentes, tanto à esquerda quanto à direita.
4) Sim. O *heap* deve ser decrescente. Exemplo: o *heap* representado pelo vetor (7, 3, 6, 1, 2, 4, 5).

10. Tabelas Hash

Exercício 10.1

	Código *hash*	**Função de compressão**
Entrada	Uma chave	O inteiro produzido pelo código *hash*
Saída	Um inteiro	Um inteiro no intervalo $[0, N-1]$
Objetivo	Transformar a chave em um inteiro que possa ser convertido pela função de compressão em um índice no *array* A	Transformar o inteiro produzido pelo código hash em um índice no *array* A, dispersando as chaves de maneira aparentemente aleatória

Exercício 10.2

1) Endereços de memória
   - Reinterpretamos os endereços de memória da chave como um inteiro.
   - Geralmente bom para qualquer tipo de chave, exceto chaves numéricas e cadeias (*strings*).
2) Cast para inteiro, ou conversão para inteiro
   - Reinterpretamos os bits da chave como um inteiro.
   - Apropriado para chaves de tamanho ≤ nro de bits do tipo inteiro. Ex: `byte`, `short`, `int` e `float`.
3) Soma de componentes
   - Particionamos os bits da chave em componentes de tamanho fixo (p.ex. 16 ou 32 bits) e somamos os componentes, ignorando *overflows*.
   - Apropriado para chaves de tamanho ≥ número de bits do tipo inteiro. Ex: `long` e `double`.
4) Acumulação polinomial, ou código *hash* polinomial
   - Particionamos os bits da chave em uma sequência de componentes de tamanho fixo (p.ex., 8, 16, ou 32 bits), usamos esses componentes como coeficientes de um polinômio $p(z)$ e calculamos $p(z)$ para um valor fixo z, ignorando *overflows*.
   - Apropriado para cadeias (*strings*).
5) Código *hash* com *shift*
   - Variação do código *hash* polinomial; substitui as multiplicações por *shifts* do resultado parcial.
   - Apropriado para cadeias (*strings*).

Exercício 10.3

1) Divisão: $h_2(y) = y \bmod N$
2) Multiplicação, Adição e Divisão: $h_2(y) = (ay + b) \bmod N$
   onde a e b são inteiros ≥ 0 tal que $a \bmod N \neq 0$

Exercício 10.4

1) Na função de compressão $h_2(y) = y \bmod N$ o tamanho da tabela *hash* (N) é um número primo. O objetivo é espalhar as chaves adequadamente, diminuindo a probabilidade de padrões na distribuição das chaves e, assim, evitar colisões.
2) Se $a \bmod N = 0$ então todo y mapeará para o mesmo valor b.

Exercício 10.5

1) Uma colisão ocorre quando a função *hash* produz o mesmo resultado para duas chaves distintas, ou seja, $h(k_1) = h(k_2)$ onde k_1 e k_2 são chaves e h é a função *hash*.
2) Encadeamento separado e endereçamento aberto.

a) Encadeamento separado: Cada posição $A[i]$ do array armazena um ponteiro para uma sequência, geralmente implementada por uma lista encadeada, contendo itens (k, v) tais que $h(k) = i$.
Vantagens: implementação simples; execução rápida.
Desvantagem: consumo maior de memória devido às estruturas de dados auxiliares.

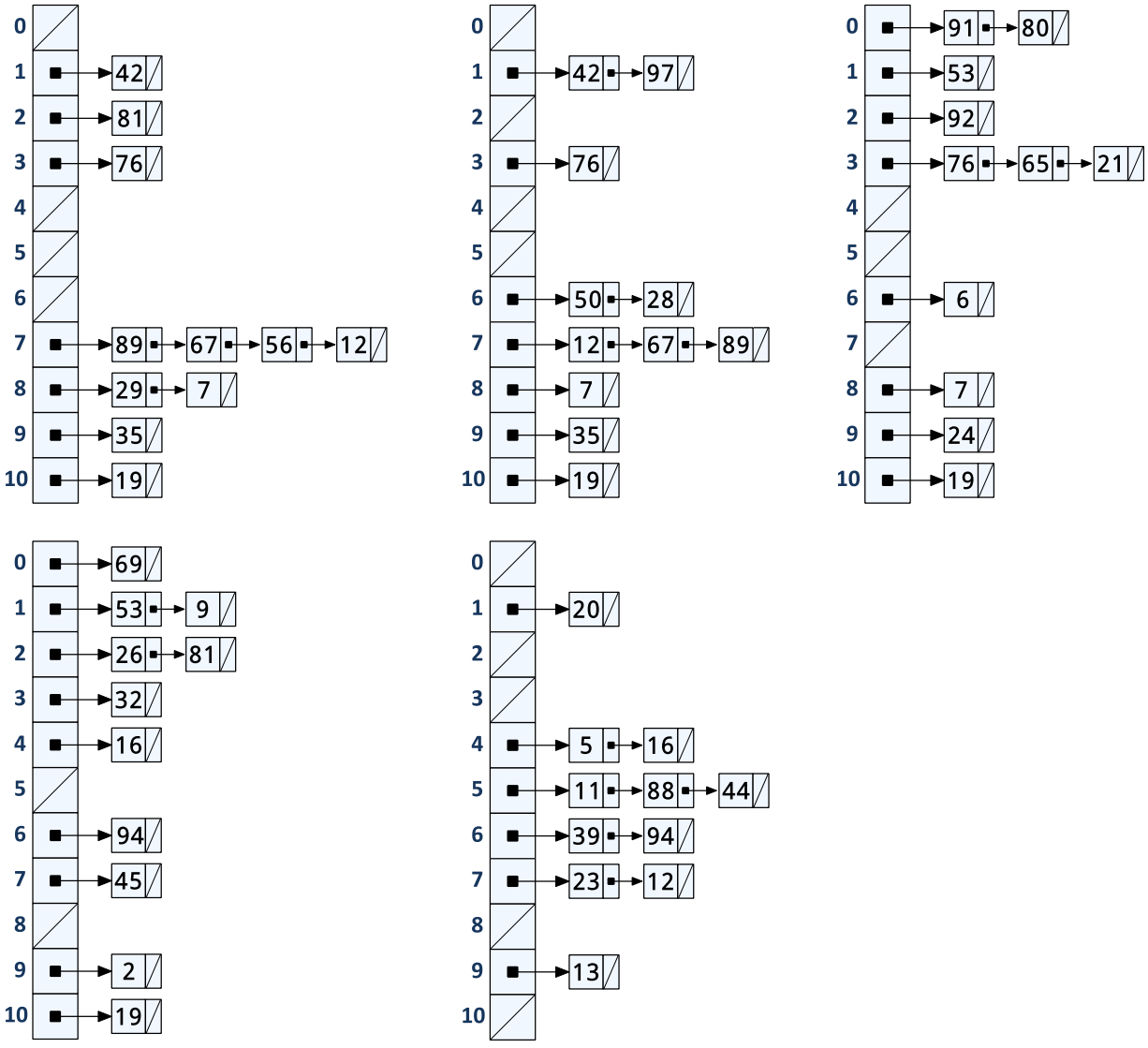
b) Endereçamento aberto: Todos os itens (k, v) são armazenados na própria tabela. Ao tentar inserir um item (k, v) em uma posição $A[i]$ que já está ocupada, tenta-se novamente em outras posições
Vantagem: uso econômico de memória.
Desvantagens: implementação mais complexa; eficiência prejudicada quando ocorre agrupamento.

3) Quando se desconhece o número de itens a serem armazenados na tabela, a melhor estratégia é o encadeamento separado, pois isso permite que a tabela cresça sem grande perda de desempenho. Com endereçamento aberto seria necessário o *rehashing* de toda a tabela, que pode ser lento.

Exercício 10.6

1) No melhor caso, $\mathrm{O}(1)$. No pior caso, $\mathrm{O}(n)$.
2) Quando todas as chaves colidem.
3) O fator de carga de uma tabela *hash* é dado por $\alpha = n/N$ onde n é o número de itens armazenados na tabela e N o tamanho do *array*. Quanto menor o fator de carga, menor a probabilidade de ocorrer colisões e maior a probabilidade das operações sobre a tabela serem efetuadas em tempo $\mathrm{O}(1)$.

Exercício 10.7



Exercício 10.8

0	1	2	3	4	5	6	7	8	9	10
11	39	20	5	16	44	88	12	23	13	94

Exercício 10.9

	0	1	2	3	4	5	6	7	8	9	10
1)	10	20	65	87	16	27	94	12	5	13	43
2)	72	42	37	20	60	11	82	89	15	29	52
3)	18	75	15	32	5	28	61	6	84	1	96
4)	91	46	70	22	53	81	28	60	40	24	85
5)	9	31	15	76	27	44	86	54	93	2	63
6)	72	20	37	29	60	11	82	89	15	58	41
7)	18	75	1	32	84	31	61	6	18	93	5
8)	58	42	66	75	82	27	11	64	7	35	19