

# Lista de exercícios

## 1 Introdução

### Exercício 1.1

(Solução 1.1)

Assista ao vídeo “*What’s an algorithm?*” (David J. Malan).

⇒ <https://youtu.be/6hfOvs8pY1k>

### Exercício 1.2

(Solução 1.2)

Assista ao vídeo “*Top 7 Data Structures for Interviews*”.

⇒ <https://youtu.be/cQWr9DFE1ww>

## 2 Complexidade de algoritmos

### Exercício 2.1

(Solução 2.1)

Desenhe o gráfico das funções  $8n$ ,  $4n \log n$ ,  $2n^2$ ,  $n^3$  e  $2^n$  usando uma escala logarítmica para os eixos  $x$  e  $y$ , isto é, se o valor da função  $f(x)$  é  $y$ , desenhe esse ponto com a coordenada  $x$  em  $\log x$  e a coordenada  $y$  em  $\log y$ .

### Exercício 2.2

(Solução 2.2)

O número de operações executadas por dois algoritmos A e B é  $40n^2$  e  $2n^3$ , respectivamente. Determine  $n_0$  tal que A seja melhor que B para  $n \geq n_0$ .

### Exercício 2.3

(Solução 2.3)

O número de operações executadas por dois algoritmos A e B é  $8n \log n$  e  $2n^2$ , respectivamente. Determine  $n_0$  tal que A seja melhor que B para  $n \geq n_0$ .

### Exercício 2.4

(Solução 2.4)

Ordene as funções a seguir por sua taxa assintótica de crescimento.

- $4n \log n + 2n$
- $2^{10}$
- $3n + 100 \log n$
- $4n$
- $2^n$
- $n^2 + 10n$
- $n^3$
- $n \log n$

### Exercício 2.5

(Solução 2.5)

Qual a complexidade assintótica no pior caso (em termos de  $\mathcal{O}$ ) do algoritmo abaixo?

```
1  /** Returns the sum of the integers in given array. */
2  public static int alg1(int[] arr) {
3      int n = arr.length, total = 0;
4      for (int j=0; j < n; j++)
5          total += arr[j];
6      return total;
7  }
```

### Exercício 2.6

(Solução 2.6)

Qual a complexidade assintótica no pior caso (em termos de  $\mathcal{O}$ ) do algoritmo abaixo?

```
1  /** Returns the sum of the integers with even index in given array. */
2  public static int alg2(int[] arr) {
3      int n = arr.length, total = 0;
4      for (int j=0; j < n; j += 2)
5          total += arr[j];
6      return total;
7  }
```

### Exercício 2.7

(Solução 2.7)

Qual a complexidade assintótica no pior caso (em termos de  $\mathcal{O}$ ) do algoritmo abaixo?

```
1  /** Returns the sum of the prefix sums of given array. */
2  public static int alg3(int[] arr) {
3      int n = arr.length, total = 0;
4      for (int j=0; j < n; j++)
5          for (int k=0; k <= j; k++)
6              total += arr[k];
7      return total;
8  }
```

### Exercício 2.8

(Solução 2.8)

Qual a complexidade assintótica no pior caso (em termos de  $\mathcal{O}$ ) do algoritmo abaixo?

```
1  /** Returns the sum of the prefix sums of given array. */
2  public static int alg4(int[] arr) {
3      int n = arr.length, prefix = 0, total = 0;
4      for (int j=0; j < n; j++) {
5          prefix += arr[j];
6          total += prefix;
7      }
8      return total;
9  }
```

### Exercício 2.9

(Solução 2.9)

Qual a complexidade assintótica no pior caso (em termos de  $\mathcal{O}$ ) do algoritmo abaixo?

```
1  /** Returns the number of times second array stores sum of prefix sums from first. */
2  public static int alg5(int[] first, int[] second) {
3      int n = first.length, count = 0;
4      for (int i=0; i < n; i++) {
5          int total = 0;
6          for (int j=0; j < n; j++)
7              for (int k=0; k <= j; k++)
8                  total += first[k];
9          if (second[i] == total) count++;
10     }
11     return count;
12 }
```

#### Exercício 2.10

(Solução 2.10)

O algoritmo A executa uma computação em tempo  $\mathcal{O}(\log n)$  para cada entrada de um arranjo de  $n$  elementos. Qual o pior caso em relação ao tempo de execução de A?

#### Exercício 2.11

(Solução 2.11)

Dado um arranjo X de  $n$  elementos, o algoritmo B escolhe  $\log n$  elementos de X, aleatoriamente, e executa um cálculo em tempo  $\mathcal{O}(n)$  para cada um. Qual o pior caso em relação ao tempo de execução de B?

#### Exercício 2.12

(Solução 2.12)

Dado um arranjo X de  $n$  elementos inteiros, o algoritmo C executa uma computação em tempo  $\mathcal{O}(n)$  para cada número par de X e uma computação em tempo  $\mathcal{O}(\log n)$  para cada elemento ímpar de X. Qual o melhor caso e o pior caso em relação ao tempo de execução de C?

#### Exercício 2.13

(Solução 2.13)

Dado um arranjo X de  $n$  elementos, o algoritmo D chama o algoritmo E para cada elemento X[i]. O algoritmo E executa em tempo  $\mathcal{O}(i)$  quando é chamado sobre um elemento X[i]. Qual o pior caso em relação ao tempo de execução do algoritmo D?

#### Exercício 2.14

(Solução 2.14)

Implemente os algoritmos `disjoint1` e `disjoint2` (apresentados nos materiais de aula), e execute uma análise experimental dos seus tempos de execução. Visualize seus tempos de execução como uma função do tamanho da entrada usando um gráfico *di-log*.

#### Exercício 2.15

(Solução 2.15)

Execute uma análise experimental para testar a hipótese de que o método da biblioteca Java, `java.util.Arrays.sort` executa em um tempo médio  $\mathcal{O}(n \log n)$ .

#### Exercício 2.16

(Solução 2.16)

Execute uma análise experimental para determinar o maior valor de  $n$  para os algoritmos `unique1` e `unique2` (apresentados nos materiais de aula), de modo que o algoritmo execute em um minuto ou menos.

## 3 Estruturas de dados fundamentais

#### Exercício 3.1

(Solução 3.1)

Forneça uma implementação para o método `size()` da classe `SinglyLinkedList`, considerando que a mesma não mantenha o tamanho armazenado em uma variável. Qual a implicação dessa modificação na complexidade assintótica do método?

#### Exercício 3.2

(Solução 3.2)

Proponha um algoritmo para encontrar o penúltimo nodo em uma lista simplesmente encadeada na qual o último nodo possui uma referência nula no campo `next`.

#### Exercício 3.3

(Solução 3.3)

Forneça a implementação do método `removeLast` para a classe `SinglyLinkedList`, para remover o último elemento da lista.

#### Exercício 3.4

(Solução 3.4)

Descreva um método para encontrar o nodo central de uma lista duplamente encadeada com nodos sentinelas, sem o uso de informações sobre o tamanho da lista. No caso de um número par de nodos, o método deve devolver o nodo à esquerda do ponto central. Qual a complexidade deste algoritmo?

#### Exercício 3.5

(Solução 3.5)

Forneça uma implementação para o método `size()` da classe `CircularlyLinkedList`, considerando que a mesma não mantenha o tamanho armazenado em uma variável. Qual a implicação dessa modificação na complexidade assintótica do método?

#### Exercício 3.6

(Solução 3.6)

Descreva um algoritmo para concatenar duas listas simplesmente encadeadas  $L$  e  $M$ , em uma lista única  $L'$  que contém todos os nodos de  $L$  seguido de todos os nodos de  $M$ .

#### Exercício 3.7

(Solução 3.7)

Descreva um algoritmo para concatenar duas listas duplamente encadeadas  $L$  e  $M$  com sentinelas, em uma lista única  $L'$ .

#### Exercício 3.8

(Solução 3.8)

Descreva em detalhes como trocar dois nodos  $x$  e  $y$  de posição (não apenas seu conteúdo) em uma lista simplesmente encadeada  $L$ , dadas as referências para  $x$  e  $y$  somente. Repita este exercício para o caso em que  $L$  é uma lista duplamente encadeada. Qual algoritmo possui maior complexidade?

#### Exercício 3.9

(Solução 3.9)

Descreva em detalhes um algoritmo para reverter uma lista simplesmente encadeada  $L$  usando somente uma quantidade constante de espaço adicional.

## 4 Buscas em estruturas lineares

#### Exercício 4.1

(Solução 4.1)

Implemente uma versão recursiva do algoritmo de busca binária, conforme as ideias do Capítulo 5 (Recursão) de Goodrich et al. (2014) – *Data Structures and Algorithms in Java*.

#### Exercício 4.2

(Solução 4.2)

Simule o algoritmo de busca binária para os seguintes casos:

- a)  $x = 15$ ,  $v = \{15, 27, 33, 46, 51, 63, 71, 82, 90\}$ .
- b)  $x = 33$ ,  $v = \{15, 27, 33, 46, 51, 63, 71, 82, 90\}$ .
- c)  $x = 63$ ,  $v = \{15, 27, 33, 46, 51, 63, 71, 82, 90\}$ .
- d)  $x = 81$ ,  $v = \{15, 27, 33, 46, 51, 63, 71, 82, 90\}$ .
- e)  $x = 22$ ,  $v = \{15, 27, 33, 46, 51, 63, 71, 82, 90\}$ .

Compare o número de avaliações realizadas pelas buscas binária e sequencial.

#### Exercício 4.3

(Solução 4.3)

Quando o vetor está ordenado, a busca sequencial não precisa percorrer toda a lista para saber que

o elemento buscado não existe. Ela pode parar quando o elemento analisado for maior que o buscado. Implemente as modificações necessárias para essa estratégia. Qual o impacto na complexidade assintótica do novo algoritmo?

#### Exercício 4.4

(Solução 4.4)

Implemente os algoritmos de busca sequencial e binária para a lista dinâmica implementada pela classe `ArrayList`. Os métodos devem retornar a posição onde o elemento foi encontrado, ou -1 caso ele não seja encontrado.

#### Exercício 4.5

(Solução 4.5)

Veja as demonstrações das buscas sequencial e binária disponíveis em <https://www.cs.usfca.edu/~galles/visualization/Search.html>.

## 5 Ordenação de estruturas lineares

#### Exercício 5.1

(Solução 5.1)

Mostre o conteúdo do *array* de inteiros (5, 7, 4, 9, 8, 5, 6, 3) para cada vez que o que o algoritmo insertion sort o modifica durante o processo de ordenação.

#### Exercício 5.2

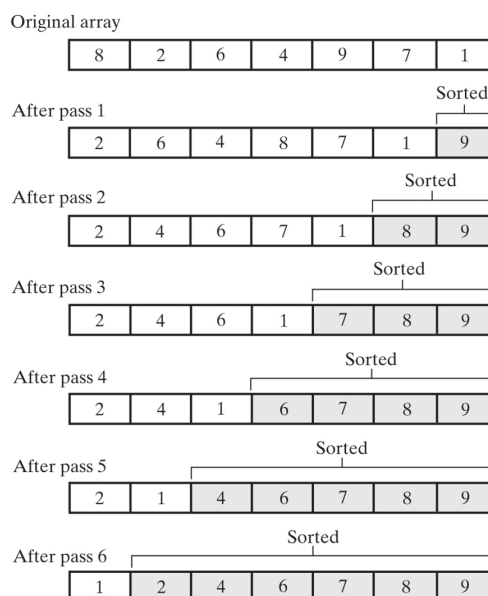
(Solução 5.2)

Qual modificação é necessária para que o algoritmo insertion sort ordene os elementos de forma decrescente?

#### Exercício 5.3

(Solução 5.3)

O algoritmo bubble sort ordena um *array* de  $n$  elementos em ordem crescente, executando  $n - 1$  passagens pelo *array*. Em cada passagem, ele compara elementos adjacentes e os troca se estiverem fora de ordem. Por exemplo, na primeira passagem ele compara o primeiro e o segundo elementos, depois o segundo e o terceiro elementos, e assim por diante. No final da primeira passagem, o maior elemento está em sua posição adequada no final do *array*. Cada passagem subsequente ignora os elementos no final do *array*, pois eles estão ordenados e são maiores que qualquer um dos elementos restantes. Assim, cada passagem faz uma comparação a menos que a passagem anterior. A figura abaixo ilustra seu funcionamento.



Implemente o bubble sort para ordenar um *array* genérico.

**Exercício 5.4**

([Solução 5.4](#))

Qual a complexidade assintótica de tempo do bubble sort?

**Exercício 5.5**

([Solução 5.5](#))

Implemente um algoritmo para verificar se um *array* está em ordem não-decrescente. Você pode usar esse método para verificar se um algoritmo de ordenação executou corretamente.

**Exercício 5.6**

([Solução 5.6](#))

No algoritmo insertion sort, podemos usar uma busca binária para encontrar a posição de inserção de cada elemento. Qual o impacto na complexidade assintótica do algoritmo?

**Exercício 5.7**

([Solução 5.7](#))

Estude os seguintes algoritmos de ordenação:

- Merge sort;
- Quick sort;
- Radix sort.

**Exercício 5.8**

([Solução 5.8](#))

Veja as demonstrações dos diferentes algoritmos de ordenação estudados em <https://www.cs.usfca.edu/~galles/visualization/Algorithms.html> e <https://visualgo.net/en/sorting>.

## 6 Pilhas, Filas e Deques

**Exercício 6.1**

([Solução 6.1](#))

Suponha que inicialmente uma pilha vazia *S* tenha realizado um total de 25 operações *push*, 12 operações *top* e 10 operações *pop*, 3 das quais retornaram *null*, indicando uma pilha vazia. Qual é o tamanho atual de *S*?

**Exercício 6.2**

([Solução 6.2](#))

Sendo a pilha do exercício anterior uma instância da classe `ArrayStack`, qual o valor final da variável *t*?

**Exercício 6.3**

([Solução 6.3](#))

Quais valores são retornados durante as seguintes operações, se executadas em uma pilha inicialmente vazia? `push(5)`, `push(3)`, `pop()`, `push(2)`, `push(8)`, `pop()`, `pop()`, `push(9)`, `push(1)`, `pop()`, `push(7)`, `push(6)`, `pop()`, `pop()`, `push(4)`, `pop()`, `pop()`.

**Exercício 6.4**

([Solução 6.4](#))

Implemente um método com a assinatura `transfer(S, T)` que transfere todos os elementos da pilha *S* para a pilha *T*, de modo que o elemento que iniciou no topo de *S* é o primeiro elemento a ser inserido em *T*, e o último elemento de *S* termina no topo de *T*.

### Exercício 6.5

(Solução 6.5)

Apresente um método recursivo que remove todos os elementos de uma pilha.

### Exercício 6.6

(Solução 6.6)

Suponha que uma fila vazia *Q* realizou um total de 32 operações de `enqueue`, 10 operações *first* e 15 operações `dequeue`, 5 das quais retornaram `null`, indicando uma fila vazia. Qual é o tamanho atual de *Q*?

### Exercício 6.7

(Solução 6.7)

Sendo a fila do exercício anterior uma instância da classe `ArrayQueue` com uma capacidade de 30 elementos nunca excedida, qual o valor final da variável *f*?

### Exercício 6.8

(Solução 6.8)

Quais são os valores retornados após as seguintes operações, se executadas em uma fila inicialmente vazia? `enqueue(5)`, `enqueue(3)`, `dequeue()`, `enqueue(2)`, `enqueue(8)`, `dequeue()`, `dequeue()`, `enqueue(9)`, `enqueue(1)`, `dequeue()`, `enqueue(7)`, `enqueue(6)`, `dequeue()`, `dequeue()`, `enqueue(4)`, `dequeue()`, `dequeue()`.

### Exercício 6.9

(Solução 6.9)

Faça um adaptador simples (classe) que implemente uma pilha usando uma instância de deque para armazenamento.

### Exercício 6.10

(Solução 6.10)

Faça um adaptador simples (classe) que implemente uma fila usando uma instância de deque para armazenamento.

### Exercício 6.11

(Solução 6.11)

Quais são os valores retornados após as seguintes operações, se executadas em um deque inicialmente vazio? `addFirst(3)`, `addLast(8)`, `addLast(9)`, `addFirst(1)`, `last()`, `isEmpty()`, `addFirst(2)`, `removeLast()`, `addLast(7)`, `first()`, `last()`, `addLast(4)`, `size()`, `removeFirst()`, `removeFirst()`.

### Exercício 6.12

(Solução 6.12)

Suponha que você tenha um deque *D* contendo os números (1, 2, 3, 4, 5, 6, 7, 8), nessa ordem. Supondo que além disso você tenha uma fila *Q* inicialmente vazia. Apresente um pseudocódigo que utilize somente *D* e *Q* (mais nenhuma variável) e resulte em *D* armazenando os elementos na ordem (1, 2, 3, 5, 4, 6, 7, 8).

### Exercício 6.13

(Solução 6.13)

Repita o exercício anterior usando o deque *D* e uma pilha *S*, inicialmente vazia.

## 7 Listas dinâmicas

### Exercício 7.1

(Solução 7.1)

Uma forma de melhorar o desempenho de uma `LinkedList` é fazer com que a busca implementada no procedimento `searchNode` seja feita “de trás para frente”, quando conveniente. Como essa abordagem pode ser implementada? Qual o impacto na complexidade do procedimento de busca?

### Exercício 7.2

(Solução 7.2)

Forneça uma representação de uma lista *L*, inicialmente vazia, após realizar as seguintes operações: `add(0, 4)`, `add(0, 3)`, `add(0, 2)`, `add(2, 1)`, `add(1, 5)`, `add(1, 6)`, `add(3, 7)`, `add(0, 8)`.

### Exercício 7.3

(Solução 7.3)

Considere uma lista dinâmica *L*, uma pilha *P* e uma fila *F*, inicialmente vazias. As seguintes operações são executadas, nesta ordem: `P.push(1)`, `P.push(2)`, `F.enqueue(3)`, `F.enqueue(4)`, `F.enqueue(5)`, `L.add(0, 6)`, `L.add(0, 7)`, `L.add(1, 8)`, `P.push(9)`, `L.add(1, P.top())`, `L.add(2, F.dequeue())`, `L.add(0, P.pop())`, `P.push(F.dequeue())`, `P.push(L.get(3))`, `F.enqueue(L.remove(2))`, `L.set(2, F.first())`. Apresente a configuração final das estruturas *L*, *P* e *F*, após a execução dessas operações.

### Exercício 7.4

(Solução 7.4)

Supondo que estamos mantendo uma coleção *C* de elementos de tal modo que, cada vez que adicionamos um novo elemento na coleção, copiamos o conteúdo de *C* em um novo `ArrayList` do tamanho exato ao necessário. Qual o tempo de processamento de adição de *n* elementos em uma coleção *C* inicialmente vazia?

### Exercício 7.5

(Solução 7.5)

Crie um método `toArray` na classe `LinkedList` que retorne um *array* com os elementos da lista encadeada. Implemente a operação inversa na classe `ArrayList`.

### Exercício 7.6

(Solução 7.6)

Implemente uma pilha usando um `ArrayList` para armazenamento dos dados.

### Exercício 7.7

(Solução 7.7)

O `java.util.ArrayList` possui um método `trimToSize` que substitui o *array* de dados por um com capacidade equivalente ao número de elementos atuais da lista. Implemente tal método na classe `ArrayList`.

### Exercício 7.8

(Solução 7.8)

A interface `java.util.Collection` inclui um método `contains(o)`, que retorna `true` se a coleção possui um objeto que é igual a `Object o`. Implemente tal método na classe `ArrayList`.

### Exercício 7.9

(Solução 7.9)

A interface `java.util.Collection` inclui um método `clear()`, que remove todos os elementos de uma coleção. Implemente tal método na classe `ArrayList`.

### Exercício 7.10

(Solução 7.10)

Desenvolva um experimento para testar a eficiência de *n* chamadas sucessivas ao método `add` de um `ArrayList` para vários *n* diferentes, e analise os resultados empíricos sob os seguintes cenários:

- Cada `add` acontece no índice 0.
- Cada `add` acontece no índice `size()/2`.
- Cada `add` acontece no índice `size()`.



## 8 Filas de prioridade

### Exercício 8.1

(Solução 8.1)

O que cada uma das chamadas `removeMin` retorna, dentre a seguinte sequência de operações em uma fila de prioridade: `insert(5)`, `insert(4)`, `insert(7)`, `insert(1)`, `removeMin()`, `insert(3)`, `insert(6)`, `removeMin()`, `removeMin()`, `insert(8)`, `removeMin()`, `insert(2)`, `removeMin()`, `removeMin()`?

### Exercício 8.2

(Solução 8.2)

Um aeroporto está desenvolvendo uma simulação computacional de controle de tráfego aéreo para lidar com eventos como aterrissagens e decolagens. Cada evento tem um *timestamp* que simboliza o tempo em que o evento ocorrerá. A simulação necessita realizar eficientemente duas operações fundamentais:

- Inserir um evento com um *timestamp* (isto é, adicionar um evento futuro).
- Retornar o evento com o *timestamp* mais próximo (i.e., determinar o próximo evento para processar).
- Qual estrutura de dados deverá ser usada para realizar essas operações? Por quê?

### Exercício 8.3

(Solução 8.3)

O método `min` de uma fila de prioridade não-ordenada executa em tempo  $\mathcal{O}(n)$ . Proponha uma alteração nessa estrutura para que o método `min` execute em tempo  $\mathcal{O}(1)$ . Explique qualquer modificação necessária em outros métodos da estrutura.

### Exercício 8.4

(Solução 8.4)

Você pode adaptar sua solução do exercício anterior para fazer o método `removeMin` de uma fila de prioridade não-ordenada executar em tempo  $\mathcal{O}(1)$ ? Justifique sua resposta.

### Exercício 8.5

(Solução 8.5)

Considere uma fila de prioridade que armazena nomes de pessoas, modelados pela classe `Name`, apresentada abaixo.

```
1 public class Name implements Comparable<Name> {
2     public String value;
3
4     public Name(String value) {
5         this.value = value;
6     }
7
8     @Override
9     public int compareTo(Name o) {
10         return this.value.length() - o.value.length();
11     }
12 }
```

Informe a sequência de elementos retornados e a configuração final de uma fila de prioridade ordenada ao executar esta sequência de comandos: `insert(new Name("Phoebe"))`, `insert(new Name("Joey"))`, `insert(new Name("Ross"))`, `min()`, `insert(new Name("Rachel"))`, `removeMin()`, `removeMin()`, `min()`, `insert(new Name("Monica"))`, `removeMin()`.

### Exercício 8.6

(Solução 8.6)

Considere a implementação alternativa abaixo do método `compareTo` para a classe `Name`, do exercício anterior.

```
1  @Override
2  public int compareTo(Name o) {
3      return this.value.compareTo(o.value);
4  }
```

Qual a nova sequência de elementos retornados e a nova configuração final da fila de prioridade ao executar a mesma sequência de comandos?

## 9 Mapas

### Exercício 9.1

([Solução 9.1](#))

Qual a complexidade de tempo no pior caso de inserir  $n$  pares chave-valor em um mapa inicialmente vazio, implementado pela classe `UnsortedArrayMap`?

### Exercício 9.2

([Solução 9.2](#))

O uso de valores `null` em um mapa é problemático, uma vez que não é possível diferenciar se um retorno `null` do método `get(k)` representa um valor legítimo de uma entrada  $(k, \text{null})$ , ou representa que a chave  $k$  não foi encontrada. A interface `java.util.Map` inclui um método booleano `containsKey(k)` que resolve essa ambiguidade. Implemente este método na classe `UnsortedArrayMap`.

### Exercício 9.3

([Solução 9.3](#))

Qual a complexidade de tempo no pior caso de realizar  $n$  remoções de uma instância de `SortedArrayMap` que contém inicialmente  $2n$  entradas?

### Exercício 9.4

([Solução 9.4](#))

Implemente o método `containsKey(k)` para a classe `SortedArrayMap`.

### Exercício 9.5

([Solução 9.5](#))

Considere o objetivo de adicionar uma entrada  $(k, v)$  em um mapa somente se não existir outra entrada com a mesma chave  $k$ . Para um mapa  $M$  sem valores `null`, isso pode ser feito da seguinte forma:

```
1  if (M.get(k) == null)
2      M.put(k, v);
```

Apesar de atingir o objetivo, essa estratégia é ineficiente, uma vez que gasta tempo para verificar que não existe entrada com a chave  $k$ , e novamente para buscar a posição de inserção da nova entrada. Para evitar isso, algumas implementações de mapas suportam um método `pullAbsent(k, v)`, que realiza a inserção assim que identifica a não existência de entrada com a chave  $k$ . Forneça a implementação deste método para a classe `UnsortedArrayMap`.

## 10 Tabelas hash

### Exercício 10.1

([Solução 10.1](#))

Qual a complexidade assintótica de tempo no pior caso de inserir  $n$  entradas em uma tabela *hash* inicialmente vazia, com as colisões resolvidas por encadeamento? E qual a complexidade no melhor caso?

### Exercício 10.2

(Solução 10.2)

Considere a classe `Name` definida abaixo, e proponha uma implementação para o método `hashCode`.

```
1 public class Name {
2     private String first;
3     private String last;
4
5     public void setName(String first, String last) {
6         this.first = first;
7         this.last = last;
8     }
9
10    public String getName() {
11        return toString();
12    }
13
14    public String toString() {
15        return first + " " + last;
16    }
17
18    // ...
19 }
```

### Exercício 10.3

(Solução 10.3)

Suponha que o tamanho da sua tabela *hash* seja 31, que você use o código *hash* padrão do Java para *strings* (descrito abaixo), e que você use encadeamento para resolver colisões. Liste cinco nomes distintos que serão armazenados na mesma posição da tabela, gerando colisões.

O Java implementa o método de Horner para gerar códigos *hash* para *strings*. Dado  $g = 31$  (valor padrão para essa variável), o código *hash* é dado por  $u_0g^{n-1} + u_1g^{n-2} + \dots + u_{n-2}g + u_{n-1}$ , onde  $u_i$  é o  $i$ -ésimo caractere da string. Abaixo é fornecida uma implementação da função *hash* usando o método de Horner para gerar o código *hash*. Note que ao somar um caractere a um inteiro (linha 5), o caractere é convertido ao seu código ASCII (inteiro).

```
1 public int hashFunction(String s, int size) {
2     int hash = 0;
3     int n = s.length();
4     for (int i = 0; i < n; i++)
5         hash = g * hash + s.charAt(i);
6     return hash % size;
7 }
```

### Exercício 10.4

(Solução 10.4)

Considere um tipo de dado cuja chave de busca consiste em três valores de ponto flutuante (latitude, longitude e altitude, por exemplo). Sugira pelo menos duas possíveis funções *hash* para esse tipo de dado.

### Exercício 10.5

(Solução 10.5)

Você tem aproximadamente 1000 imagens de *thumbnail* e quer armazená-las em um mapa que usa *hashing*. Cada imagem possui 20 pixels de largura e 20 pixels de altura, e cada pixel é uma entre 256 cores. Sugira algumas funções *hash* que poderiam ser usadas.

### Exercício 10.6

(Solução 10.6)

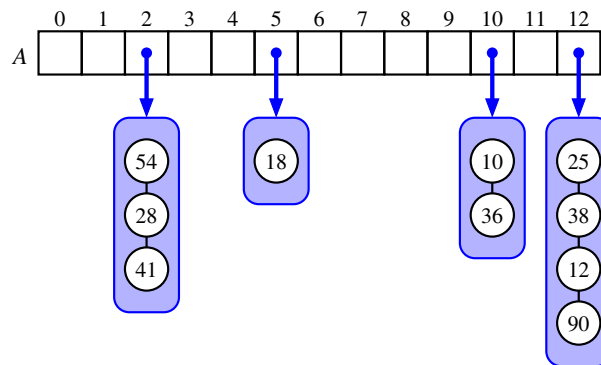
Desenhe a tabela *hash* de tamanho 11 resultante do uso da função *hash*  $h(i) = (3i + 5) \% 11$  ao armazenar as chaves 12, 44, 13, 88, 23, 94, 11, 39, 20, 16 e 5, assumindo que as colisões são tratadas com encadeamento.

### Exercício 10.7

(Solução 10.7)

A operação de *rehashing* consiste em aumentar a capacidade de uma tabela *hash* e realocar suas entradas.

Mostre o resultado do *rehashing* da tabela *hash* ilustrada abaixo para uma tabela de tamanho 19 usando a nova função *hash*  $h(k) = 3k \% 19$ .



### Exercício 10.8

(Solução 10.8)

Considere a classe `Pair` definida abaixo e proponha implementações para os métodos `equals` e `hashCode`.

```

1 public class Pair<A,B> {
2     A first;
3     B second;
4
5     public Pair(A a, B b) {
6         first = a;
7         second = b;
8     }
9
10    public A getFirst() { return first; }
11    public B getSecond() { return second; }
12 }

```

### Exercício 10.9

(Solução 10.9)

Considere um tipo de dado para registrar pacientes em uma centro de assistência médica. Cada registro contém um identificador inteiro para o paciente, e strings para a data, motivo da visita e tratamento prescrito. Projete e implemente a classe `PatientRecord`, sobrescrevendo o método `hashCode`. Escreva um programa para testar essa classe.

### Exercício 10.10

(Solução 10.10)

Projete a classe `PatientDataBase` que armazena instâncias de `PatientRecord`, conforme descrito no exercício anterior. Essa classe deve prover duas operações de consulta. Dada a identificação de um paciente e uma data, a primeira operação deve retornar o motivo da visita, e a segunda operação deve retornar o tratamento prescrito.

## 11 Árvores

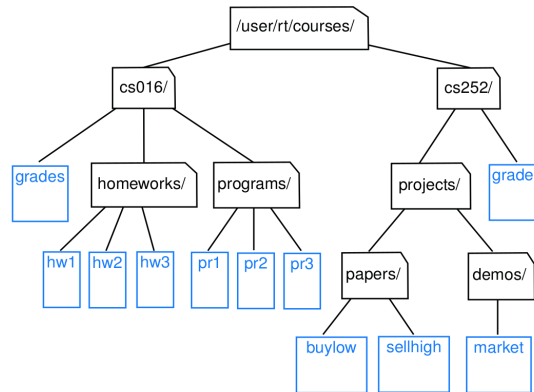
### Exercício 11.1

(Solução 11.1)

Considere a árvore abaixo e informe:

- Qual nodo é a raiz?
- Quais são os nodos internos?
- Quantos descendentes tem o nodo `cs016`/?

- d) Quantos ancestrais tem o nodo `cs016/`?
- e) Quais são os irmãos do nodo `homeworks/`?
- f) Quais nodos estão na sub-árvore cuja raiz é o nodo `projects/`?
- g) Qual o nível do nodo `papers/`?
- h) Qual a altura da árvore?



### Exercício 11.2

(Solução 11.2)

Qual a altura da menor (mais baixa) árvore binária que contém 21 nodos? Essa árvore é cheia? É balanceada?

### Exercício 11.3

(Solução 11.3)

Considere uma árvore binária com três níveis.

- a) Qual o número máximo de nodos da árvore?
- b) Qual o número máximo de folhas da árvore?
- c) Responda às questões acima considerando uma árvore binária com dez níveis.

### Exercício 11.4

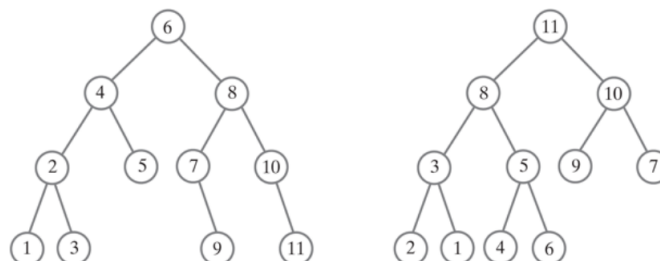
(Solução 11.4)

Escreva um algoritmo recursivo que conta os nodos de uma árvore binária.

### Exercício 11.5

(Solução 11.5)

Considere a travessia de uma árvore binária. Suponha que a visitação de um nodo consiste em apresentar em tela seu elemento. Qual o resultado das travessias em pré-ordem e level-ordem nas árvore abaixo?



### Exercício 11.6

(Solução 11.6)

Considere as árvores do exercício anterior, contendo os elementos inteiros identificados nos nodos.

- a) A primeira árvore é uma árvore binária de busca? Justifique.
- b) A segunda árvore é uma *max-heap*? Justifique.

**Exercício 11.7**

(Solução 11.7)

Uma árvore binária de busca pode ser também uma *max-heap* (i.e. ao mesmo tempo)? Explique.

**Exercício 11.8**

(Solução 11.8)

Represente (desenhe) a menor árvore binária de busca possível (menor altura) que armazene as seguintes strings: *Ann, Ben, Chad, Deepak, Ella, Jada, Jazmin, Kip, Luis, Pat, Rico, Scott, Tracy, Zak*.

**Exercício 11.9**

(Solução 11.9)

Represente (desenhe) uma *max-heap* que armazene as strings do exercício anterior. A *max-heap* é única?

**Exercício 11.10**

(Solução 11.10)

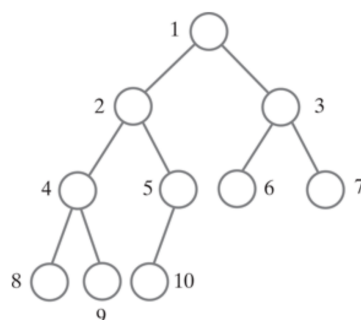
Considere que a ordem de uma visitação em largura de uma árvore binária completa seja 11, 8, 10, 3, 5, 9, 7, 2, 1, 4, 6. Qual a ordem de visitação em profundidade dessa mesma árvore? Qual a configuração da árvore?

**Exercício 11.11**

(Solução 11.11)

Suponha que os nodos de uma árvore binária completa sejam numerados conforme a ordem de visitação de uma travessia em largura. A raiz da árvore é o nodo 1, e assim por diante. Abaixo é mostrada uma árvore com essas características. Dado um nodo  $i$ , como podemos computar:

- a) O irmão de  $i$ , se existente.
- b) O filho esquerdo de  $i$ , se existente.
- c) O filho direito de  $i$ , se existente.
- d) O pai de  $i$ , se existente.



**Exercício 11.12**

(Solução 11.12)

Apresente o pseudocódigo de um algoritmo que conta o número de folhas em uma árvore binária que são filhas esquerdas dos seus respectivos pais.

**Exercício 11.13**

(Solução 11.13)

Quantas árvores binárias de busca diferentes podem armazenar os valores  $\{1, 2, 3\}$ ?

**Exercício 11.14**

(Solução 11.14)

Represente (desenhe) uma árvore binária  $T$  que satisfaça todas as seguintes condições:

- Cada nodo interno armazena um caractere.
- O percurso em pré-ordem de  $T$  gera a sequência EXAMFUN.
- O percurso em level-ordem de  $T$  gera a sequência EXNAUMF.

**Exercício 11.15**

([Solução 11.15](#))

Insira, em uma árvore binária de busca inicialmente vazia, os elementos 30, 40, 24, 58, 48, 26, 11, 13 (nessa ordem). Desenhe a árvore após cada inserção.

# Soluções dos exercícios

## 1 Introdução

### Solução 1.1

([Exercício 1.1](#))

Assista ao vídeo sugerido.

### Solução 1.2

([Exercício 1.2](#))

Assista ao vídeo sugerido.

## 2 Complexidade de algoritmos

### Solução 2.1

([Exercício 2.1](#))

Gráfico disponível [aqui](#).

### Solução 2.2

([Exercício 2.2](#))

Igualando as equações, temos que  $40n^2 = 2n^3$  para  $n' = 0$  e  $n'' = 20$ . Logo,  $40n^2 \leq 2n^3$  para  $n \geq 20$ .  
Veja a representação gráfica [aqui](#).

### Solução 2.3

([Exercício 2.3](#))

Por inspeção, assumindo  $n_0 = 10$ , temos que  $8n \log n \leq 2n^2$  para todo  $n \geq n_0$ . Veja a representação gráfica [aqui](#).

### Solução 2.4

([Exercício 2.4](#))

$2^{10} \ll 3n + 100 \log n = 4n \ll n \log n = 4n \log n + 2n \ll n^2 + 10n \ll n^3 \ll 2^n$ .

### Solução 2.5

([Exercício 2.5](#))

$\mathcal{O}(n)$ .

### Solução 2.6

([Exercício 2.6](#))

$\mathcal{O}(n)$ .

### Solução 2.7

([Exercício 2.7](#))

$\mathcal{O}(n^2)$ .

### Solução 2.8

([Exercício 2.8](#))

$\mathcal{O}(n)$ .

### Solução 2.9

([Exercício 2.9](#))

$\mathcal{O}(n^3)$ .

### Solução 2.10

([Exercício 2.10](#))

$\mathcal{O}(n \log n)$ .



### Solução 2.11

(Exercício 2.11)

$\mathcal{O}(n \log n)$ .

### Solução 2.12

(Exercício 2.12)

Melhor caso:  $\mathcal{O}(n \log n)$ , quando todos os elementos são ímpares. Pior caso:  $\mathcal{O}(n^2)$ , quando todos os elementos são pares.

### Solução 2.13

(Exercício 2.13)

O tempo de execução é proporcional a  $\sum_{i=1}^n i = n(n-1)/2 = \mathcal{O}(n^2)$ .

### Solução 2.14

(Exercício 2.14)

Exercício de análise experimental.

### Solução 2.15

(Exercício 2.15)

Exercício de análise experimental.

### Solução 2.16

(Exercício 2.16)

Exercício de análise experimental.

## 3 Estruturas de dados fundamentais

### Solução 3.1

(Exercício 3.1)

Uma possível solução é percorrer a estrutura, contando a quantidade de nodos. Isso implica em um aumento na complexidade assintótica de constante para linear, i.e.  $\mathcal{O}(1)$  para  $\mathcal{O}(n)$ .

```
1 public int size(){
2     int count = 0;
3     Node<E> walk = head;
4     while(walk != null){
5         count++;
6         walk = walk.getNext();
7     }
8     return count;
9 }
```

### Solução 3.2

(Exercício 3.2)

Uma possível solução consiste em um algoritmo linear simples.

```
1 private Node<E> penultimate(){
2     if(size < 2) throw new IllegalStateException("List must have 2 or more entries");
3     Node<E> walk = head;
4     while(walk.getNext().getNext() != null)
5         walk = walk.getNext();
6     return walk;
7 }
```

### Solução 3.3

(Exercício 3.3)

Dica: use o método `penultimate` do exercício anterior para atualizar a referência do atributo `tail`.

### Solução 3.4

(Exercício 3.4)

Considere uma busca combinada de ambas extremidades. Lembre-se que um *link hop* é uma atribuição do formato  $p = p.getNext()$ ; ou  $p = p.getPrev()$ . O método a seguir executa em tempo  $O(n)$ .

```
1 private Node<E> middle(){
2     if(size == 0) throw new IllegalStateException("List must be nonempty");
3     Node<E> middle = header.getNext();
4     Node<E> partner = trailer.getPrev();
5     while(middle != partner && middle.getNext() != partner){
6         middle = middle.getNext();
7         partner = partner.getPrev();
8     }
9     return middle;
10 }
```

### Solução 3.5

(Exercício 3.5)

Uma possível solução é percorrer a estrutura, contando a quantidade de nodos. Isso implica em um aumento na complexidade assintótica de constante para linear, i.e.  $O(1)$  para  $O(n)$ .

```
1 public int size2(){
2     if(tail == null)
3         return 0;
4     Node<E> walk = tail.getNext();
5     int count = 1;
6     while(walk != tail){
7         count++;
8         walk = walk.getNext();
9     }
10    return count;
11 }
```

### Solução 3.6

(Exercício 3.6)

A operação de concatenação não precisa procurar tudo em  $L$  e  $M$ . Use um nodo temporário para caminhar até o final da lista  $L$ . Então, faça o último elemento de  $L$  apontar para o primeiro elemento de  $M$  como seu próximo nodo. O número de passos é proporcional ao tamanho de  $L$ .

```
1 Concatenate(L,M):
2     Create a node v
3     v = L.getHeader()
4     while v.getNext() != null do
5         v = v.getNext()
6     v.setNext(M.getHeader())
7     return L
```

### Solução 3.7

(Exercício 3.7)

Junte o final de  $L$  no começo de  $M$ . Use dois nodos temporários, **temp1** e **temp2**. Inicialize **temp1** como o **trailer** de  $L$  e **temp2** como o **header** de  $M$ . Atribua **temp2** como o próximo elemento de **temp1** e **temp1** como o elemento anterior de **temp2**. Faça  $L' \leftarrow L$  e atribua o **trailer** de  $M$  como **trailer** de  $L'$ .

### Solução 3.8

(Exercício 3.8)

Realizar uma troca (*swap*) em uma lista simplesmente encadeada levará mais tempo do que em uma lista duplamente encadeada. Essa implementação requer muito cuidado, especialmente quando  $x$  e  $y$  são vizinhos um do outro. A dificuldade na eficiência ocorre porque para trocar  $x$  e  $y$  em uma lista simplesmente encadeada devemos localizar os nodos imediatamente anteriores a  $x$  e  $y$  percorrendo a estrutura, e não tem uma maneira rápida de fazer isso. A complexidade assintótica dessa operação no pior caso, quando todos os elementos devem ser percorridos, é  $O(n)$ . Em uma lista duplamente encadeada, não é necessário percurso na lista, pois cada nodo já possui apontamentos para seus antecessores e sucessores, reduzindo a complexidade assintótica para  $O(1)$  no pior caso.

### Solução 3.9

(Exercício 3.9)

Dica: considere mudar a orientação das ligações enquanto faz uma única passagem pela lista. Tal método da classe `SinglyLinkedList` poderá ser implementado da seguinte forma (observe que esta implementação funciona mesmo para listas triviais).

```
1 public void reverse(){
2     Node<E> prev = null;
3     Node<E> walk = head;
4     while(walk != null){
5         Node<E> adv = walk.getNext();
6         walk.setNext(prev);
7         prev = walk;
8         walk = adv;
9     }
10    head = prev;
11 }
```

## 4 Buscas em estruturas lineares

### Solução 4.1

(Exercício 4.1)

Uma possível implementação é apresentada na Seção 5.1.3 (Busca Binária) de Goodrich et al. (2014) – *Data Structures and Algorithms in Java*.

### Solução 4.2

(Exercício 4.2)

Com uma caneta, marque as referências para início e fim da sub-estrutura considerada em cada iteração da busca. Caso essas referências se cruzem, o elemento não foi encontrado. Caso a referência para o meio da lista aponte para o elemento buscado, seu índice é encontrado.

- a) Busca binária: 3 avaliações (51, 27, 15). Busca sequencial: 1 avaliação (15).
- b) Busca binária: 3 avaliações (51, 27, 33). Busca sequencial: 3 avaliações (15, 27, 33).
- c) Busca binária: 3 avaliações (51, 71, 63). Busca sequencial: 6 avaliações (15, 27, 33, 46, 51, 63).
- d) Busca binária: 3 avaliações (51, 71, 82). Busca sequencial: 9 avaliações (todos os elementos).
- e) Busca binária: 3 avaliações (51, 27, 15). Busca sequencial: 9 avaliações (todos os elementos).

### Solução 4.3

(Exercício 4.3)

Para implementar essa modificação, você deve parar a busca com sucesso quando o elemento buscado é igual ao elemento analisado. A busca continua se o elemento buscado for menor que o elemento analisado, e pára sem sucesso, caso contrário. Na prática, essa modificação torna o algoritmo mais eficiente nos casos em que ele pára antes de percorrer toda a lista. Porém, no pior caso o elemento buscado não está na lista e é maior que qualquer elemento dela, implicando na necessidade da lista ser totalmente percorrida. Logo, a complexidade continua sendo  $\mathcal{O}(n)$  no pior caso.

Busca sequencial modificada simples em um *array* de inteiros:

```
1 public int search(int[] array, int value) {
2     for(int i = 0; i < array.length; i++) {
3         if(array[i] == value)
4             return i;
5         if(array[i] < value)
6             return -1;
7     }
8 }
```

```
8     return -1;
9 }
```

Busca sequencial modificada genérica em uma lista:

```
1 public int indexOf_sorted(List<E> list, E value) {
2     for(int i = 0; i < list.size(); i++) {
3         if(comp.compare(list.get(i), value) == 0)
4             return i;
5         if(comp.compare(list.get(i), value) < 0)
6             return -1;
7     }
8
9     return -1;
10 }
```

#### Solução 4.4

(Exercício 4.4)

Dica: use o código das buscas sequencial e binária em *arrays* genéricos para buscar o elemento desejado no *array data*. Para isso, deverá ser criado um atributo para armazenar o comparador, o qual deve ser usado para ambas as buscas.

#### Solução 4.5

(Exercício 4.5)

Analise as demonstrações para entender os dois tipos de busca.

## 5 Ordenação de estruturas lineares

#### Solução 5.1

(Exercício 5.1)

*Array* inicial:

(5, 7, 4, 9, 8, 5, 6, 3)

*Array* após cada inserção:

(5, 7, 4, 9, 8, 5, 6, 3)  
(5, 7, 4, 9, 8, 5, 6, 3)  
(4, 5, 7, 9, 8, 5, 6, 3)  
(4, 5, 7, 9, 8, 5, 6, 3)  
(4, 5, 7, 8, 9, 5, 6, 3)  
(4, 5, 5, 7, 8, 9, 6, 3)  
(4, 5, 5, 6, 7, 8, 9, 3)  
(3, 4, 5, 5, 6, 7, 8, 9)

#### Solução 5.2

(Exercício 5.2)

Basta inverter o operador relacional na comparação dos elementos. Ou seja, em vez de usar `comp.compare(element, array[index]) < 0`, usamos `comp.compare(element, array[index]) > 0`. Com isso, é fácil implementar um método de ordenação que recebe a ordem desejada (“normal” ou “inversa”) como um parâmetro e executa a ordenação correspondente.

#### Solução 5.3

(Exercício 5.3)

Algoritmo bubble sort:

```
1 public void bubbleSort(E[] array) {
2     for (int lastIndex = array.length - 1; lastIndex > 0; lastIndex--) {
3         for (int index = 0; index < lastIndex; index++) {
4             if (comp.compare(array[index], array[index + 1]) > 0) {
5                 E temp = array[index];
6                 array[index] = array[index + 1];
7                 array[index + 1] = temp;
8             }
9         }
10    }
11 }
```

#### Solução 5.4

(Exercício 5.4)

O bubble sort tem complexidade assintótica de tempo  $\mathcal{O}(n^2)$  no pior, médio e melhor casos. É possível interromper o algoritmo quando uma passagem não faz nenhuma modificação, indicando que o *array* já está ordenado. Neste caso, a complexidade assintótica no melhor caso é reduzida para  $\mathcal{O}(n)$ .

#### Solução 5.5

(Exercício 5.5)

```
1 public boolean isSorted(E[] array) {
2     boolean sorted = true;
3     for (int index = 0; sorted && (index < array.length - 1); index++) {
4         if (comp.compare(array[index], array[index + 1]) > 0)
5             sorted = false;
6     }
7     return sorted;
8 }
```

#### Solução 5.6

(Exercício 5.6)

Ao usar uma busca binária, reduzimos a complexidade assintótica da busca pela posição de inserção do elemento de  $\mathcal{O}(n)$  para  $\mathcal{O}(\log n)$  no pior caso. Apesar dessa redução, o algoritmo ainda precisa deslocar elementos pela estrutura para efetivar a inserção, o que faz com que sua complexidade assintótica se mantenha em  $\mathcal{O}(n^2)$  no pior caso. Além disso, a busca binária ainda executaria em  $\mathcal{O}(\log n)$  no melhor caso (quando a estrutura já está ordenada), enquanto a busca sequencial executa em  $\mathcal{O}(1)$ . Portanto, a modificação proposta aumenta a complexidade do algoritmo de  $\mathcal{O}(n)$  para  $\mathcal{O}(n \log n)$  no melhor caso.

#### Solução 5.7

(Exercício 5.7)

As obras listadas na bibliografia da disciplina apresentam esses algoritmos.

#### Solução 5.8

(Exercício 5.8)

Análise as demonstrações para entender os diferentes algoritmos.

## 6 Pilhas, filas e deque

#### Solução 6.1

(Exercício 6.1)

Se a pilha está vazia quando *pop* é chamado, seu tamanho não muda. Logo, o tamanho da pilha é  $25 - 10 + 3 = 18$ .

#### Solução 6.2

(Exercício 6.2)

É uma posição menor que o tamanho. Logo,  $t = 17$ .

### Solução 6.3

(Exercício 6.3)

Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: 3, 8, 2, 1, 6, 7, 4, 9.

### Solução 6.4

(Exercício 6.4)

Você deve transferir um item de cada vez.

```
1 static <E> void transfer(Stack<E> S, Stack<E> T) {  
2     while (!S.isEmpty()) {  
3         T.push(S.pop());  
4     }  
5 }
```

### Solução 6.5

(Exercício 6.5)

Se a pilha está vazia, retorne “pilha vazia”. Caso contrário, remova o elemento do topo da pilha e chame a operação recursivamente com a pilha atualizada.

### Solução 6.6

(Exercício 6.6)

Se a pilha está vazia quando `dequeue` é chamado, seu tamanho não é modificado. Logo, o tamanho da fila é  $32 - 15 + 5 = 22$ .

### Solução 6.7

(Exercício 6.7)

Cada operação `dequeue` de sucesso implica em mover o índice para a direita de maneira circular. Logo,  $f = 10$ .

### Solução 6.8

(Exercício 6.8)

Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: 5, 3, 2, 8, 9, 1, 7, 6.

### Solução 6.9

(Exercício 6.9)

Dica: basta usar as operações apropriadas nas extremidades do deque.

### Solução 6.10

(Exercício 6.10)

Dica: basta usar as operações apropriadas nas extremidades do deque.

### Solução 6.11

(Exercício 6.11)

Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: 9, false, 9, 2, 7, 6, 2, 1.

### Solução 6.12

(Exercício 6.12)

A solução consiste em usar o resultado dos métodos de remoção como argumentos para os métodos de inserção. Solução:

```
D.addLast(D.removeFirst())  
D.addLast(D.removeFirst())  
D.addLast(D.removeFirst())  
Q.enqueue(D.removeFirst())  
Q.enqueue(D.removeFirst())  
D.addFirst(Q.dequeue())  
D.addFirst(Q.dequeue())  
D.addFirst(D.removeLast())
```

```
D.addFirst(D.removeLast())  
D.addFirst(D.removeLast())
```

### Solução 6.13

(Exercício 6.13)

A solução consiste em usar o resultado dos métodos de remoção como argumentos para os métodos de inserção. Adicionalmente, você precisará usar mais de uma pilha para armazenamento temporário. Solução:

```
D.addLast(D.removeFirst())  
D.addLast(D.removeFirst())  
D.addLast(D.removeFirst())  
S.push(D.removeFirst())  
D.addLast(D.removeFirst())  
D.addFirst(S.pop())  
D.addFirst(D.removeLast())  
D.addFirst(D.removeLast())  
D.addFirst(D.removeLast())  
D.addFirst(D.removeLast())
```

## 7 Listas dinâmicas

### Solução 7.1

(Exercício 7.1)

Basta verificar se o índice buscado é menor ou maior que  $\text{size}/2$ , para saber em qual metade da estrutura o índice se encontra. Caso se trate da primeira metade, a busca deve ser realizada a partir do primeiro elemento. Caso contrário, a busca inicia pelo último elemento. Com isso, apenas metade dos elementos precisará ser varrido no pior caso. Logo, a complexidade cai para  $n/2$ , o que é mais eficiente na prática, mas não altera a complexidade assintótica  $\mathcal{O}(n)$ .

### Solução 7.2

(Exercício 7.2)

Desenhe a lista, mostrando os estados antes e depois de cada operação. A configuração final da lista deve ser (8, 2, 6, 5, 7, 3, 1, 4).

### Exercício 7.3

(Exercício 7.3)

A configuração final das estruturas é:

L: (9, 7, 5, 8, 6).

P: (9, 5).

F: (1, 2, 4, 3).

### Solução 7.4

(Exercício 7.4)

O tempo de execução para inserir um novo elemento é  $\mathcal{O}(n)$ . Como  $n$  elementos são incluídos, o tempo de execução total é  $\mathcal{O}(n^2)$ .

### Solução 7.5

(Exercício 7.5)

Os métodos devem criar as listas alternativas, atribuir os elementos a elas e retornar a estrutura criada. Diferentes implementações podem devolver uma lista com as mesmas referências ou com cópias dos elementos.

### Solução 7.6

(Exercício 7.6)

Dica: use o método `size` para ajudar a manter o controle do topo da pilha.

### Solução 7.7

(Exercício 7.7)

O método `resize` pode ser usado para diminuir o vetor.

```
1 public void trimToSize() {  
2     if (data.length != size)  
3         resize(size);  
4 }
```

### Solução 7.8

(Exercício 7.8)

Lembre-se de usar o método `equals` para testar a igualdade. A classe do elemento deve implementar esse método.

```
1 public boolean contains(Object o) {  
2     for (int k = 0; k < size; k++)  
3         if (data[k].equals(o))  
4             return true;  
5     return false;  
6 }
```

### Solução 7.9

(Exercício 7.9)

Uma boa estratégia consiste em atualizar todas as referências para `null`.

```
1 public void clear() {  
2     for (int k = 0; k < size; k++)  
3         data[k] = null;  
4     size = 0;  
5 }
```

### Solução 7.10

(Exercício 7.10)

Exercício de análise experimental.

## 8 Filas de prioridade

### Solução 8.1

(Exercício 8.1)

1, 3, 4, 5, 2, 6.

### Solução 8.2

(Exercício 8.2)

A melhor estrutura de dados para uma simulação de controle de tráfego aéreo é uma fila de prioridade. Essa estrutura permite manipular os *timestamps* e manter os eventos em ordem, de tal forma que o evento com menor instante de tempo seja facilmente extraído.

### Solução 8.3

(Exercício 8.3)

Mantenha uma variável adicional que referencie a entrada mínima atual. Isso permite executar a operação `min` em tempo constante  $\mathcal{O}(1)$ . Para que isso funcione, o método `insert` deve ser alterado, atualizando a variável adicional sempre que o novo elemento sendo inserido seja menor que `min`, bem como ao inserir quando a estrutura está vazia. O método `removeMin` também deve ser alterado, pois ele será responsável por identificar o novo elemento mínimo e atualizar a referência da variável adicional, para então remover o `min`.



#### Solução 8.4

(Exercício 8.4)

Não. A operação `removeMin` continua necessitando tempo linear  $\mathcal{O}(n)$ . Apesar do `min` atual ser facilmente encontrado e removido, tal método precisa percorrer todos os elementos restantes para identificar o novo mínimo.

#### Solução 8.5

(Exercício 8.5)

Retornos: "Joey", "Joey", "Ross", "Phoebe", "Phoebe".

Configuração final: ("Monica", "Rachel")

#### Solução 8.6

(Exercício 8.6)

Retornos: "Joey", "Joey", "Phoebe", "Rachel", "Monica".

Configuração final: ("Ross", "Rachel")

## 9 Mapas

#### Solução 9.1

(Exercício 9.1)

A primeira inserção consome  $\mathcal{O}(1)$ , a segunda consome  $\mathcal{O}(2)$ , e assim por diante, com a última inserção consumindo  $\mathcal{O}(n)$ . A execução completa dessa operação consome  $\mathcal{O}(n^2)$ .

#### Solução 9.2

(Exercício 9.2)

Para isso, use o método `findIndex`.

```
1 public boolean containsKey(K key) {  
2     return (findIndex(key) != -1);  
3 }
```

#### Solução 9.3

(Exercício 9.3)

Dado que o mapa seguirá contendo  $n$  entradas no final do procedimento, você pode assumir que cada operação `remove` consome o mesmo tempo assintótico  $\mathcal{O}(n)$ . Logo, a complexidade total no pior caso é  $\mathcal{O}(n^2)$ .

#### Solução 9.4

(Exercício 9.4)

Novamente, podemos utilizar o método `findIndex`.

```
1 public boolean containsKey(K key) {  
2     int j = findIndex(key);  
3     return (j < data.size() && compare(key, data.get(j)) == 0);  
4 }
```

#### Solução 9.5

(Exercício 9.5)

A solução deve fazer uma única chamada ao método `findIndex`.

```
1 public V pullAbsent(K key, V value) {  
2     int j = findIndex(key);  
3     if (j == -1) {  
4         data.add(new MapEntry<>(key, value));  
5         return null;  
6     } else {  
7         return data.get(j).getValue();  
8     }
```

```
8     }  
9 }
```

## 10 Tabelas hash

### Solução 10.1

(Exercício 10.1)

Pior caso: todas as entradas ocupam a mesma posição,  $\mathcal{O}(n^2)$ .

Melhor caso: cada entrada ocupa uma posição diferente,  $\mathcal{O}(n)$ .

### Solução 10.2

(Exercício 10.2)

Uma possibilidade é usar a implementação padrão do Java para o nome completo.

```
1 public int hashCode() {  
2     return toString().hashCode();  
3 }
```

Uma alternativa é computar os códigos *hash* separadamente (para o primeiro e último nomes) e então agregá-los.

```
1 public int hashCode() {  
2     return first.hashCode() + last.hashCode();  
3 }
```

### Solução 10.3

(Exercício 10.3)

Neste exercício, o tamanho da tabela *hash* é igual ao valor de  $g$  (i.e., 31). Logo, a função *hash* vai considerar o último caractere da string para definir o índice do elemento na tabela (pela operação da linha 6). Uma possível resposta: **Jim, Tim, Tom, Sam e Kim**.

Exemplo para o nome *Jim*:

- Os valores dos caracteres  $J$ ,  $i$  e  $m$  são 74, 105 e 109, respectivamente.
- Logo, o código *hash* é  $74 \cdot 31^2 + 105 \cdot 31 + 109 = 74478$ .
- O índice é  $74478 \% 31 = 16$ .

### Solução 10.4

(Exercício 10.4)

Alternativas de código *hash*:

1. Converta cada um dos valores em string e os concatene, formando uma string única. Use então a função *hashCode* para strings.
2. Seja  $x$ ,  $y$  e  $z$  os três valores reais, compute o polinômio  $xd^2 + yd + z$ , para algum  $d > 0$ . Trunque então o resultado para um número inteiro.

Em ambos os casos, o código *hash* gerado deverá ser comprimido pela função *hash* usando a operação módulo (resto da divisão inteira) e o tamanho da tabela. Você também pode escalar e/ou arredondar os valores reais antes do processamento.

### Solução 10.5

(Exercício 10.5)

Alternativas de código *hash*:

1. Some os valores de todos os 400 pixels.

2. Usar a mesma técnica aplicada a strings (e a implementação pelo método de Horner), onde os valores de  $u$  são os pixels da imagem. [veja o [Exercício 10.2](#) e a [Solução 10.2](#)]

**Nota:** as melhores práticas recomendam que todas as partes da chave contribuam para a função *hash*. No entanto, para fins de eficiência qualquer das abordagens poderia usar um subconjunto dos pixels da imagem. Se a abordagem 1 for usada com uma tabela de tamanho 2000, a distribuição de índices será sensível ao número de pixels usado. Qualquer número abaixo de 10 causará uma má distribuição. Em geral, usar mais pixels é melhor.

#### Solução 10.6

([Exercício 10.6](#))

0	1	2	3	4	5	6	7	8	9	10
13	94				44			12	16	20
	39				88			23	56	
					11					

#### Solução 10.7

([Exercício 10.7](#))

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
38				90				28	41	54	10		36			18	12	25

#### Solução 10.8

([Exercício 10.8](#))

Dica: combinar os códigos *hash* dos dois componentes.

```

1 public int hashCode() {
2     return a.hashCode() + b.hashCode();
3 }
4
5 public boolean equals(Object o) {
6     if(o == null) return false;
7     if(getClass() != o.getClass()) return false;
8     Pair other = (Pair) o;
9     return (a.equals(other.a) && b.equals(other.b));
10 }

```

#### Solução 10.9

([Exercício 10.9](#))

Exercício de implementação.

#### Solução 10.10

([Exercício 10.10](#))

Exercício de implementação.

## 11 Árvores

#### Solução 11.1

([Exercício 11.1](#))

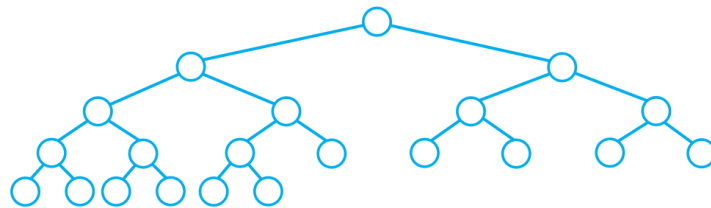
- /user/rt/courses/.
- Aqueles representados na cor preta.
- 9.
- 1.

- e) 2.
- f) projects/, papers/, demos/, buylow, sellhigh, market.
- g) 4.
- h) 5.

### Solução 11.2

(Exercício 11.2)

A altura é 5. Ela não é cheia, mas é balanceada. A árvore abaixo é completa, mas se movermos alguma folha (no nível 5) para outro pai, ela mantém sua altura, mas deixa de ser completa.



### Solução 11.3

(Exercício 11.3)

- a)  $2^3 - 1 = 7$  nodos.
- b) 4 folhas.
- c)  $2^{10} - 1 = 1023$  nodos; 512 folhas.

### Solução 11.4

(Exercício 11.4)

```
1 count(root):  
2   IF root != null THEN  
3     RETURN 1 + count(root.left) + count(root.right)  
4   ELSE  
5     RETURN 0
```

### Solução 11.5

(Exercício 11.5)

Travessias para a primeira árvore:

- **Pré-ordem** (profundidade): 6, 4, 2, 1, 3, 5, 8, 7, 9, 10, 11.
- **Level-ordem** (largura): 6, 4, 8, 2, 5, 7, 10, 1, 3, 9, 11.

Travessias para a segunda árvore:

- **Pré-ordem** (profundidade): 11, 8, 3, 2, 1, 5, 4, 6, 10, 9, 7.
- **Level-ordem** (largura): 11, 8, 10, 3, 5, 9, 7, 2, 1, 4, 6.

### Solução 11.6

(Exercício 11.6)

- Não. O 9 está na sub-árvore do 8, quando deveria ser o filho esquerdo do 10.
- Não. O 5 e o 6 precisam ser trocados um com o outro para que a árvore seja um *max-heap*.

### Solução 11.7

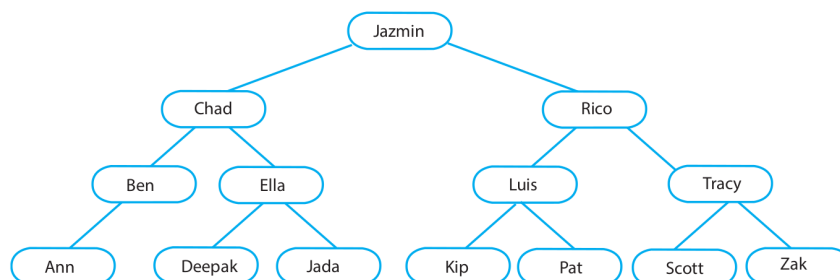
(Exercício 11.7)

Sim, uma árvore de busca binária pode ser uma *max-heap*. Considere uma árvore binária de busca com os valores 7 e 2, onde 7 é a raiz. A árvore é completa e satisfaz a propriedade da *heap*.

### Solução 11.8

(Exercício 11.8)

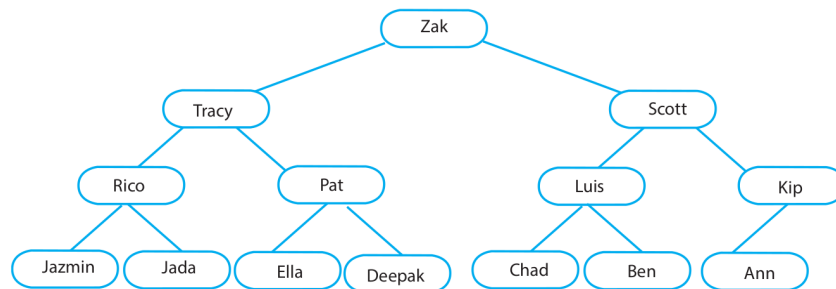
A árvore abaixo não é única. Uma segunda árvore binária de busca com a mesma altura poderia ser construída usando *Kip* como raiz.



### Solução 11.9

(Exercício 11.9)

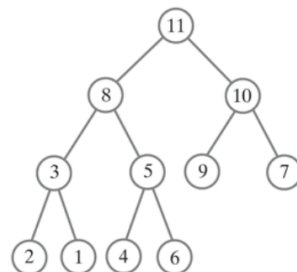
A *max-heap* abaixo não é única. Podemos trocar o conteúdo de quaisquer irmãos e manter o critério de ordenação da *max-heap*.



### Solução 11.10

(Exercício 11.10)

A ordem de visitação em profundidade será 11, 8, 3, 2, 1, 5, 4, 6, 10, 9, 7. A árvore é apresentada abaixo.



### Solução 11.11

(Exercício 11.11)

- Se  $i$  é par, o irmão de  $i$  é o nodo  $i + 1$ . Se  $i$  é ímpar e maior que 1, seu irmão é o nodo  $i - 1$  (quando  $i = 1$ , trata-se da raiz da árvore, que não tem irmão).
- $2i$ .
- $2i + 1$ .
- $i/2$ , para  $i > 1$  (quando  $i = 1$ , trata-se da raiz da árvore, que não tem pai).

### Solução 11.12

(Exercício 11.12)

```

1  count(root, leftChild):
2    IF root == null THEN
3      RETURN 0
4    IF root.left == null AND root.right == null THEN
5      IF leftChild THEN
6        RETURN 1
7      ELSE
8        RETURN 0
9    ELSE
10     RETURN 0 + count(root.left, true) + count(root.right, false)
  
```

### Solução 11.13

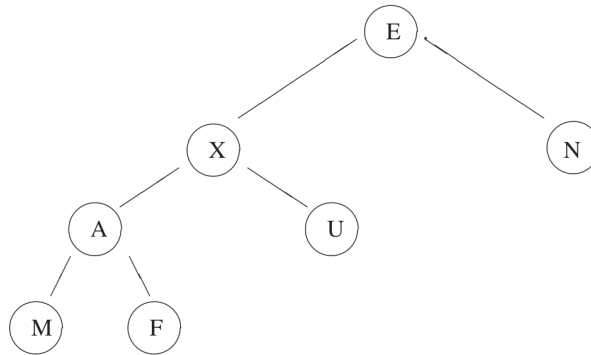
(Exercício 11.13)

Cinco (2 com raiz 1; 1 com raiz 2; 2 com raiz 3).

### Solução 11.14

(Exercício 11.14)

A árvore abaixo satisfaz as condições (mas não é a única).



### Solução 11.15

(Exercício 11.15)

