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?

```python
# Returns the sum of the integers in given array.
def alg1(arr):
  n = len(arr)
  total = 0
  for j in range(n):
    total += arr[j]
  return total
```

Exercício 2.6

(Solução 2.6)

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

```python
# Returns the sum of the integers with even index in given array.
def alg2(arr):
  n = len(arr)
  total = 0
  for j in range(0, n, 2):
    total += arr[j]
  return total
```

Exercício 2.7

(Solução 2.7)

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

```python
# Returns the sum of the prefix sums of given array.
def alg3(arr):
  n = len(arr)
  total = 0
  for j in range(n):
    for k in range(j + 1):
      total += arr[k]
  return total
```

Exercício 2.8

(Solução 2.8)

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

```python
# Returns the sum of the prefix sums of given array.
def alg4(arr):
  n = len(arr)
  prefix = 0
  total = 0
  for j in range(n):
    prefix += arr[j]
    total += prefix
  return total
```

Exercício 2.9

(Solução 2.9)

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

```python
# Returns the number of times second array stores sum of prefix sums from first.
def alg5(first, second):
  n = len(first)
  count = 0
  for i in range(n):
    total = 0
    for j in range(n):
      for k in range(j + 1):
        total += first[k]
    if second[i] == total:
      count += 1
  return count
```

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 de ordenação do python (`sort` ou `sorted`) 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 fundamentais e listas

Exercício 3.1 (**Solução 3.1**)
Forneça uma implementação para o método `size()` da classe `LinkedList`, 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**)
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 desse algoritmo?

Exercício 3.3 (**Solução 3.3**)
Descreva um algoritmo para concatenar duas listas duplamente encadeadas L e M com sentinelas, em uma lista única L'.

Exercício 3.4 (**Solução 3.4**)
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 3.5 (**Solução 3.5**)
Implemente uma lista simplesmente encadeada que forneça operações de inserção e remoção nas duas extremidades, além dos métodos `size` e `is_empty`.

Exercício 3.6 (**Solução 3.6**)
Uma forma de melhorar o desempenho de uma lista duplamente encadeada é buscar pelo nodo desejado (para inserção ou remoção) "de trás para frente", quando conveniente. Como essa abordagem pode ser implementada? Qual o impacto na complexidade do procedimento de busca?

Exercício 3.7 (**Solução 3.7**)
Forneça uma representação de uma lista `L`, inicialmente vazia, após realizar as seguintes operações: `insert(0,4)`, `insert(0,3)`, `insert(0,2)`, `insert(2,1)`, `insert(1,5)`, `insert(1,6)`, `insert(3,7)`, `insert(0,8)`.

Exercício 3.8 (**Solução 3.8**)
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 uma nova lista baseada em arranjo do tamanho exato ao necessário. Qual o tempo de processamento da adição de n elementos em uma coleção `C` inicialmente vazia?

Exercício 3.9 (**Solução 3.9**)
Considere a implementação de uma lista duplamente encadeada fornecida pela classe `LinkedList`. Implemente a função `toArray` que retorna uma lista baseada em arranjo com os elementos da lista encadeada.

Exercício 3.10 (**Solução 3.10**)
A lista baseada em arranjo fornece uma função `index(e)`, que retorna o índice onde o elemento e está armazenado, e dispara uma exceção, caso o elemento não é encontrado na estrutura. Implemente essa função na classe `LinkedList`.

Exercício 3.11 (**Solução 3.11**)
A lista baseada em arranjo fornece uma função `clear()`, que remove todos os elementos da coleção. Implemente essa função na classe `LinkedList`.

Exercício 3.12 (**Solução 3.12**)
Desenvolva um experimento para testar a eficiência de n chamadas sucessivas à função `insert` de uma lista baseada em arranjo para vários n diferentes, e analise os resultados empíricos sob os seguintes cenários:

a. Cada `insert` acontece no índice 0.

b. Cada `insert` acontece no índice `size()/2`.

c. Cada `insert` acontece no índice `size()`.

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 4 (Recursão) de Goodrich et al. (2013) – *Data Structures and Algorithms in Python*.

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?

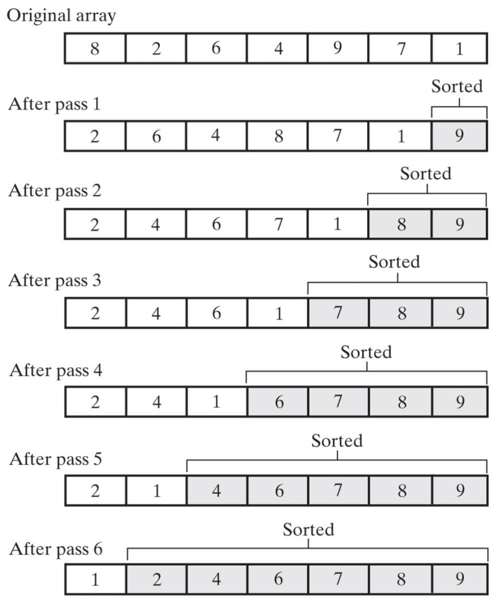
Exercício 4.4 (Solução 4.4)
Veja as demonstrações das buscas sequencial e binária disponíveis em https://www.cs.usfca.edu/~galles/visualization/Search.html.

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 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*.

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**)
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.insert(0,6)`, `L.insert(0,7)`, `L.insert(1,8)`, `P.push(9)`, `L.insert(1, P.top())`, `L.insert(2, F.dequeue())`, `L.insert(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 6.2 (**Solução 6.2**)

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 6.3 (**Solução 6.3**)

Sendo a pilha do exercício anterior implementada usando um arranjo seguindo as ideias estudadas em sala de aula, qual o valor final da variável `t`?

Exercício 6.4 (**Solução 6.4**)

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 6.5 (**Solução 6.5**)

Implemente uma função 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 6.6 (**Solução 6.6**)

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

Exercício 6.7 (**Solução 6.7**)

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 6.8 (**Solução 6.8**)

Sendo a fila do exercício anterior implementada seguindo as ideias estudadas em sala de aula e usando um arranjo com uma capacidade de 30 elementos nunca excedida, qual o valor final da variável `f`?

Exercício 6.9 (**Solução 6.9**)

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 6.10 (**Solução 6.10**)

Faça um adaptador simples (classe) que implemente uma pilha usando um deque para armazenamento. Use a implementação de deque fornecido pelo módulo `collections.deque`.

Exercício 6.11 (**Solução 6.11**)

Faça um adaptador simples (classe) que implemente uma fila usando uma instância de deque para armazenamento. Use a implementação de deque fornecido pelo módulo `collections.deque`.

Exercício 6.12 (**Solução 6.12**)
Quais são os valores retornados após as seguintes operações, se executadas em um deque inicialmente vazio? `add_first(3)`, `add_last(8)`, `add_last(9)`, `add_first(1)`, `last()`, `is_empty()`, `add_first(2)`, `remove_last()`, `add_last(7)`, `first()`, `last()`, `add_last(4)`, `size()`, `remove_first()`, `remove_first()`.

Exercício 6.13 (**Solução 6.13**)
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` incialmente 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 6.14 (**Solução 6.14**)
Repita o exercício anterior usando o deque `D` e uma pilha `S`, inicialmente vazia.

7 Filas de prioridade

Exercício 7.1 (**Solução 7.1**)
O que cada uma das chamadas `removeMin` retorna, dentre a seguinte sequências 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 7.2 (**Solução 7.2**)
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 7.3 (**Solução 7.3**)
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 7.4 (**Solução 7.4**)
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 7.5 (**Solução 7.5**)
Considere uma fila de prioridade que armazena nomes de pessoas, modelados pela classe `Name`, apresentada abaixo.

```java
public class Name implements Comparable<Name> {
  public String value;

  public Name(String value) {
    this.value = value;
  }
```

```

  @Override
  public int compareTo(Name o) {
    return this.value.length() - o.value.length();
  }
}
```

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 7.6 (Solução 7.6)

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

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

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

8 Mapas

Exercício 8.1 (Solução 8.1)

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

Exercício 8.2 (Solução 8.2)

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, 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 8.3 (Solução 8.3)

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

Exercício 8.4 (Solução 8.4)

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

Exercício 8.5 (Solução 8.5)

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:

```
if(M.get(k) == null)
  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`.

9 Tabelas hash

Exercício 9.1 (Solução 9.1)

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 9.2 (Solução 9.2)

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

```
public class Name {
  private String first;
  private String last;

  public void setName(String first, String last) {
    this.first = first;
    this.last = last;
  }

  public String getName() {
    return toString();
  }

  public String toString() {
    return first + " " + last;
  }

  // ...
}
```

Exercício 9.3 (Solução 9.3)

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 $u_0 g^{n-1} + u_1 g^{n-2} + \cdots + u_{n-2} g + u_{n-1}$, 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 9.4 (Solução 9.4)

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 9.5 (**Solução 9.5**)

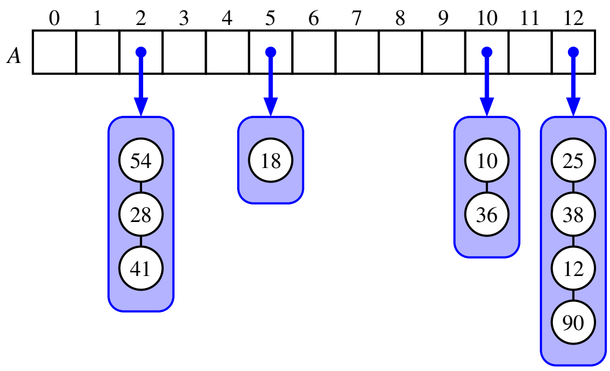
Você tem aproximadamente 1000 imagens de *thumbnail* e quer armazená-las em um mapa que usa *hashing*. Cada imagem possui 20 pixels de largura e 20 pixels de altura, e cada pixel é uma entre 256 cores. Sugira algumas funções *hash* que poderiam ser usadas.

Exercício 9.6 (**Solução 9.6**)

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 9.7 (**Solução 9.7**)

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 9.8 (**Solução 9.8**)

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

```
public class Pair<A,B> {
  A first;
  B second;

  public Pair(A a, B b) {
    first = a;
    second = b;
  }

  public A getFirst() { return first; }
  public B getSecond() { return second; }
}
```

Exercício 9.9 (**Solução 9.9**)

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 9.10 (**Solução 9.10**)

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.

10 Árvores

Exercício 10.1 (Solução 10.1)

Considere a árvore abaixo e informe:

a) Qual nodo é a raiz?

b) Quais são os nodos internos?

c) Quantos descendentes tem o nodo cs016/?

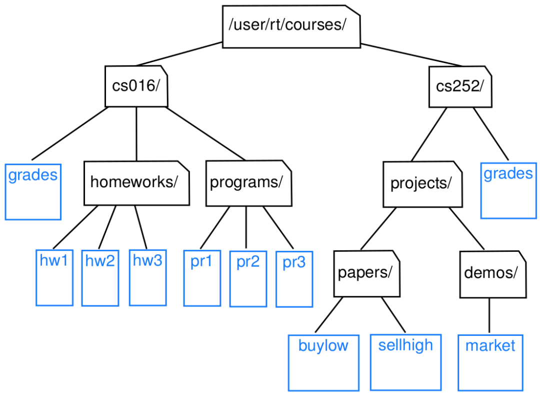
d) Quantos ancestrais tem o nodo cs016/?

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 10.2 (Solução 10.2)

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

Exercício 10.3 (Solução 10.3)

Considere uma árvore binária com três níveis.

a) Qual o número máximo de nodos da árvore?

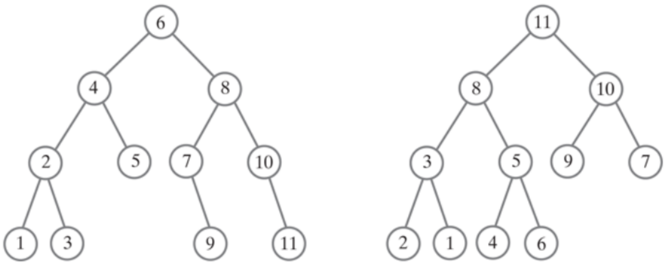
b) Qual o número máximo de folhas da árvore?

c) Responda às questões acima considerando uma árvore binária com dez níveis.

Exercício 10.4 (Solução 10.4)

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

Exercício 10.5 (**Solução 10.5**)
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 10.6 (**Solução 10.6**)
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 10.7 (**Solução 10.7**)
Uma árvore binária de busca pode ser também uma *max-heap* (i.e. ao mesmo tempo)? Explique.

Exercício 10.8 (**Solução 10.8**)
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 10.9 (**Solução 10.9**)
Represente (desenhe) uma *max-heap* que armazene as strings do exercício anterior. A *max-heap* é única?

Exercício 10.10 (**Solução 10.10**)
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 10.11 (**Solução 10.11**)
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:

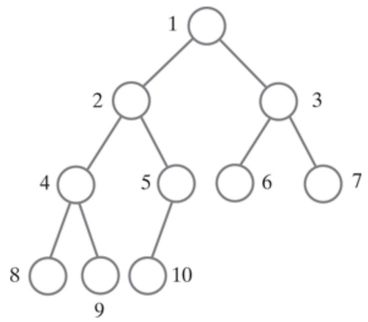
a) O irmão de i, se existente.

b) O filho esquerdo de i, se existente.

c) O filho direito de i, se existente.

d) O pai de i, se existente.

Exercício 10.12 (**Solução 10.12**)
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 10.13 (**Solução 10.13**)

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

Exercício 10.14 (**Solução 10.14**)

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 10.15 (**Solução 10.15**)

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 $\sum_{i=1}^{n} i = n(n+1)/2 = O(n^2)$.

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 fundamentais e listas

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 $\mathcal{O}(n)$.

```
def size(self):
  count = -1
  walk = self._header
  while walk != self._trailer:
    count += 1
    walk = walk._next
  return count
```

Solução 3.2 (**Exercício 3.2**)
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 $O(n)$.

```
def middle(self):
  if self.is_empty():
    raise Exception("List is empty")
  mid = self.header.next
  partner = self.trailer.prev
  while mid != partner and mid.next != partner:
    mid = mid.next
    partner = partner.prev
  return mid
```

Solução 3.3 (**Exercício 3.3**)
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 3.4 (**Exercício 3.4**)
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 $\mathcal{O}(n)$. 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 $\mathcal{O}(1)$ no pior caso.

Solução 3.5 (**Exercício 3.5**)
Exercício de implementação.

Solução 3.6 (**Exercício 3.6**)
Basta verificar se o índice buscado é menor ou maior que `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 3.7 (**Exercício 3.7**)
Desenhe a lista, mostrando os estados antes e depois de cada operação. A configuração final da lista deve ser $(8, 2, 6, 5, 7, 3, 1, 4)$.

Solução 3.8 (**Exercício 3.8**)
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 3.9 (**Exercício 3.9**)
A função deve criar uma lista alternativa, atribuir os elementos a ela e retornar a estrutura criada. A implementação pode devolver uma lista com as mesmas referências ou com cópias dos elementos.

Solução 3.10 (**Exercício 3.10**)
Você deve percorrer a estrutura em busca do elemento. A implementação abaixo retorna -1, caso não encontra o elemento.

```python
def index(self, e):
  if self.is_empty(): return -1
  count = -1
  walk = self.header.next
  while walk != self.trailer:
    count += 1
      if walk.element == e:
        return count
    walk = walk.next
  return -1
```

Solução 3.11 (**Exercício 3.11**)
Basta atualizar as referências `next` e `prev` dos nodos sentinelas para que um referencie o outro, e atualizar `size` para 0.

```
def clear(self):
  self.header.next = self.trailer
  self.trailer.prev = self.header
  self.size = 0
```

Solução 3.12 (**Exercício 3.12**)
Exercício de análise experimental.

4 Buscas em estruturas lineares

Solução 4.1 (**Exercício 4.1**)
Uma possível implementação é apresentada na Seção 4.1.3 (Busca Binária) de Goodrich et al. (2013) – *Data Structures and Algorithms in Python*.

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 aa lista ser totalmente percorrida. Log, a complexidade continua sendo $\mathcal{O}(n)$ no pior caso.

Busca sequencial modificada para um *array*:

```
def index(array, value):
  for i in range(len(array)):
    if array[i] == value:
      return i
    if array[i] < value:
      return -1
  return -1
```

Busca sequencial modificada para uma lista encadeada:

```
def index(self, e):
  if self.is_empty(): return -1
  count = -1
  walk = self.header.next
  while walk != self.trailer:
    count += 1
```

```
    if walk.element == e:
      return count
    if walk.element < e:
      return -1
    walk = walk.next
  return -1
```

Solução 4.4 (**Exercício 4.4**)
Analise as demostrações 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 `key < array[j]`, usamos `key > array[j]`. 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:

```
def bubble_sort(array):
  for last_index in range(len(array) - 1, 0, -1):
    for index in range(last_index):
      if array[index] > array[index + 1]:
        # Operação de swap/troca
        array[index], array[index + 1] = array[index + 1], array[index]
```

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**)

```
def is_sorted(array):
  for index in range(len(array) - 1):
    if array[index] > array[index + 1]:
      return False
  return True
```

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**)
Analise as demostrações para entender os diferentes algoritmos.

6 Pilhas, filas e deques

Exercício 6.1 (**Exercício 6.1**)
A configuração final das estruturas é:

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

P: $(9, 5)$.

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

Solução 6.2 (**Exercício 6.2**)
Se a pilha está vazia quando `pop` é chamado, seu tamanho não muda. Logo, o tamanho da pilha é $25 - 10 + 3 = 18$.

Solução 6.3 (**Exercício 6.3**)
É uma posição menor que o tamanho. Logo, $t = 17$.

Solução 6.4 (**Exercício 6.4**)
Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: $3, 8, 2, 1, 6, 7, 4, 9$.

Solução 6.5 (**Exercício 6.5**)
Você deve transferir um item de cada vez.

```
def transfer(S, T) {
  while not S.is_empty():
    T.push(S.pop())
```

Solução 6.6 (**Exercício 6.6**)
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 6.7 (**Exercício 6.7**)
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 6.8 (**Exercício 6.8**)
Cada operação `dequeue` de sucesso implica em mover o índice para a direita de maneira circular. Logo, $f = 10$.

Solução 6.9 (**Exercício 6.9**)
Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: $5, 3, 2, 8, 9, 1, 7, 6$.

Solução 6.10 (**Exercício 6.10**)
Dica: basta usar as operações apropriadas nas extremidades do deque.

Solução 6.11 (**Exercício 6.11**)
Dica: basta usar as operações apropriadas nas extremidades do deque.

Solução 6.12 (**Exercício 6.12**)
Desenhe a estrutura para simular as operações e mudanças realizadas. Resultado: 9, `false`, $9, 2, 7, 6, 2, 1$.

Solução 6.13 (**Exercício 6.13**)
A solução consiste em usar o resultado dos métodos de remoção como argumentos para os métodos de inserção. Solução:

```
D.add_last(D.remove_first())
D.add_last(D.remove_first())
D.add_last(D.remove_first())
Q.enqueue(D.remove_first())
Q.enqueue(D.remove_first())
D.add_first(Q.dequeue())
D.add_first(Q.dequeue())
D.add_first(D.remove_last())
D.add_first(D.remove_last())
D.add_first(D.remove_last())
```

Solução 6.14 (**Exercício 6.14**)
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.add_last(D.remove_first())
D.add_last(D.remove_first())
D.add_last(D.remove_first())
S.push(D.remove_first())
D.add_last(D.remove_first())
```

```
D.add_first(S.pop())
D.add_first(D.remove_last())
D.add_first(D.remove_last())
D.add_first(D.remove_last())
D.add_first(D.remove_last())
```

7 Filas de prioridade

Solução 7.1 (Exercício 7.1)
1, 3, 4, 5, 2, 6.

Solução 7.2 (Exercício 7.2)
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 7.3 (Exercício 7.3)
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 7.4 (Exercício 7.4)
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 7.5 (Exercício 7.5)
Retornos: `"Joey"`, `"Joey"`, `"Ross"`, `"Phoebe"`, `"Phoebe"`.
Configuração final: `("Monica", "Rachel")`

Solução 7.6 (Exercício 7.6)
Retornos: `"Joey"`, `"Joey"`, `"Phoebe"`, `"Rachel"`, `"Monica"`.
Configuração final: `("Ross", "Rachel")`

8 Mapas

Solução 8.1 (Exercício 8.1)
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 8.2 (Exercício 8.2)
Para isso, use o método `findIndex`.

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

Solução 8.3

(**Exercício 8.3**)

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 8.4

(**Exercício 8.4**)

Novamente, podemos utilizar o método `findIndex`.

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

Solução 8.5

(**Exercício 8.5**)

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

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

9 Tabelas hash

Solução 9.1

(**Exercício 9.1**)

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 9.2

(**Exercício 9.2**)

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

```java
public int hashCode() {
  return toString().hashCode();
}
```

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

```java
public int hashCode() {
  return first.hashCode() + last.hashCode();
}
```

Solução 9.3

(**Exercício 9.3**)

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 9.4 **(Exercício 9.4)**

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 9.5 **(Exercício 9.5)**

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 9.2 e a Solução 9.2]

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 píxels usado. Qualquer número abaixo de 10 causará uma má distribuição. Em geral, usar mais pixels é melhor.

Solução 9.6 **(Exercício 9.6)**

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

Solução 9.7 **(Exercício 9.7)**

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 9.8 **(Exercício 9.8)**

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

```
public int hashCode() {
  return a.hashCode() + b.hashCode();
}

public boolean equals(Object o) {
  if(o == null) return false;
  if(getClass() != o.getClass()) return false;
  Pair other = (Pair) o;
  return (a.equals(other.a) && b.equals(other.b));
}
```

Solução 9.9 (**Exercício 9.9**)
Exercício de implementação.

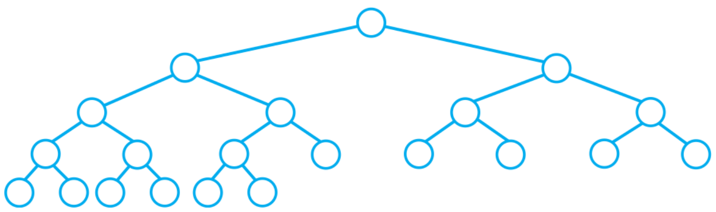
Solução 9.10 (**Exercício 9.10**)
Exercício de implementação.

10 Árvores

Solução 10.1 (**Exercício 10.1**)

a) /user/rt/courses/.
b) Aqueles representados na cor preta.
c) 9.
d) 1.
e) grades e programs/.
f) projects/, papers/, demos/, buylow, sellhigh, market.
g) 4.
h) 5.

Solução 10.2 (**Exercício 10.2**)
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 10.3 (**Exercício 10.3**)

a) $2^3 - 1 = 7$ nodos.
b) 4 folhas.

c) $2^{10} - 1 = 1023$ nodos; 512 folhas.

Solução 10.4 **(Exercício 10.4)**

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

Solução 10.5 **(Exercício 10.5)**

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 10.6 **(Exercício 10.6)**

a) Não. O 9 está na sub-árvore do 8, quando deveria ser o filho esquerdo do 10.

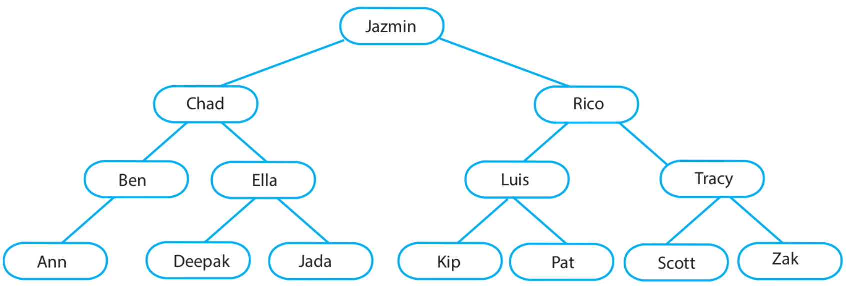
b) Não. O 5 e o 6 precisam ser trocados um com o outro para que a árvore seja um *max-heap*.

Solução 10.7 **(Exercício 10.7)**

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 10.8 **(Exercício 10.8)**

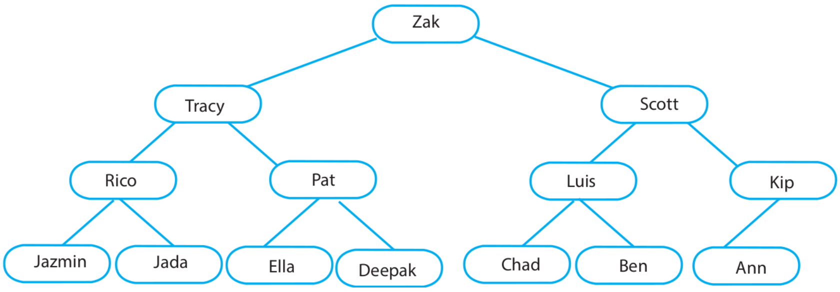
A árvore abaixo não é única. Uma segunda árvore binária de busca com a mesma altura poderia ser construída usando *Kip* como raiz.



Solução 10.9

(Exercício 10.9)

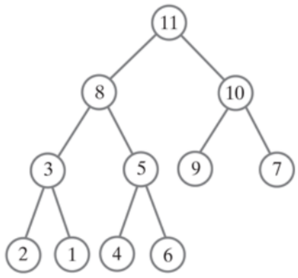
A *max-heap* abaixo não é única. Podemos trocar o conteúdo de quaisquer irmãos e manter o critério de ordenação da *max-heap*.



Solução 10.10

(Exercício 10.10)

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



Solução 10.11

(Exercício 10.11)

a) 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).

b) $2i$.

c) $2i + 1$.

d) $i/2$, para $i > 1$ (quando $i = 1$, trata-se da raiz da árvore, que não tem pai).

Solução 10.12

(Exercício 10.12)

```
count(root, leftChild):
  IF root == null THEN
    RETURN 0
  IF root.left == null AND roof.right == null THEN
    IF leftChild THEN
      RETURN 1
    ELSE
      RETURN 0
  ELSE
    RETURN 0 + count(root.left, true) + count(root.right, false)
```

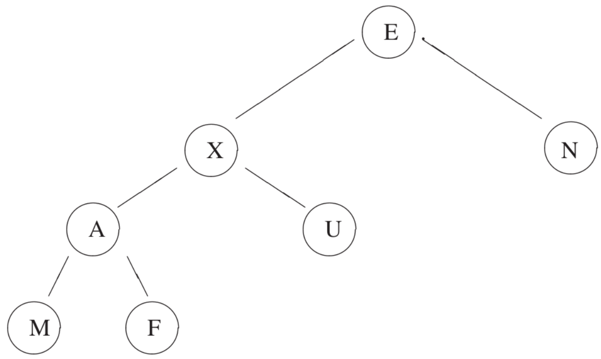
Solução 10.13

(Exercício 10.13)

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

Solução 10.14 (Exercício 10.14)

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



Solução 10.15 (Exercício 10.15)