



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º I

“Algoritmos e Estruturas de Dados”

“Algoritmos e Estruturas de Dados”

- Logística
 - Tecnologia
 - Aulas
 - Avaliação
- Programa da cadeira
- O ambiente de programação



Tecnologia

- Linguagem: Java.
- Componente gráfica: Processing.
- Ambiente: Eclipse, linha de comando.
- Sistema operativo: Windows, MacOS, Linux.
- Avaliador automático: Mooshak.
- Tutoria: Moodle.

Aulas

- Teóricas: Pedro Guerreiro.
- Práticas: Tiago Candeias, Hamid Shahbazkia.

Bibliografia principal

Algorithms, quarta edição (2011), Robert Sedgewick e Kevin Wayne.

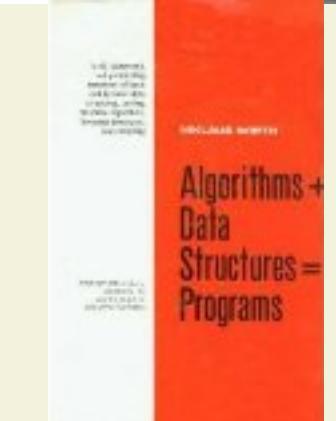
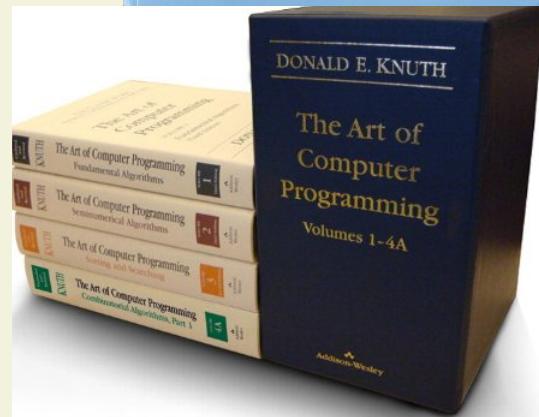
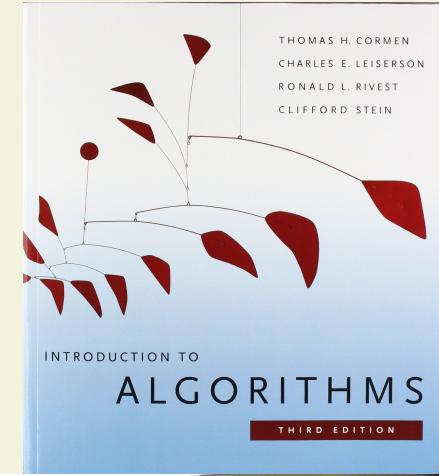
Book site: <http://algs4.cs.princeton.edu/>



<http://www.amazon.co.uk/Algorithms-Robert-Sedgewick/dp/032157351X/>

Bibliografia de interesse

- Introduction to Algorithms, 3.^a edição, Thomas Cormen, Charles Leiserson, Ronald Rivest, Clifford Stein.
<http://www.amazon.co.uk/Introduction-Algorithms-T-Cormen/>
- The Art of Computer Programming, Donald Knuth.
<http://www.amazon.co.uk/Art-Computer-Programming-Volumes-1-4a/>
- Algorithms + Data Structures = Programs, Niklaus Wirth (1975).
<http://www.amazon.co.uk/Algorithms-Structures-Prentice-Hall-automatic-computation/>



Algorithms, Part I



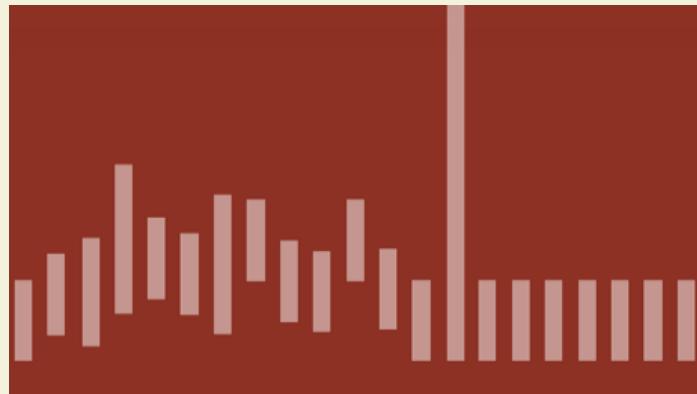
- Instructors: Robert Sedgewick, Kevin Wayne.
- “This course covers the essential information that every serious programmer needs to know about algorithms and data structures, with emphasis on applications and scientific performance analysis of Java implementations. Part I covers (...) union-find algorithms; basic iterable data types (stack, queues, and bags); sorting algorithms (quicksort, mergesort, heapsort) and applications; priority queues; binary search trees; red-black trees; hash tables; and symbol-table applications.”

<https://www.coursera.org/course/alg4partI>

Mais Coursera

Algorithms, Part II

- “Part II covers graph-processing algorithms, including minimum spanning tree and shortest paths algorithms, and string processing algorithms, including string sorts, tries, substring search, regular expressions, and data compression, and concludes with an overview placing the contents of the course in a larger context.”



<https://www.coursera.org/course/alg4partII>

Programa de AED

- Conceitos fundamentais
 - Modelo de programação
 - Sacos, pilhas e filas
 - Arrays
 - “Union-Find”
 - Análise de algoritmos
- Ordenação
 - Algoritmos elementares
 - Mergesort, quicksort
 - Filas com prioridade
- Busca
 - Árvores binárias de busca
 - Árvores equilibradas
 - Tabelas de dispersão
- Grafos
 - Busca em profundidade
 - Busca em largura
 - Árvores de cobertura
 - Caminho mais curto
 - Problema do caixeiro viajante
- Cadeias de carateres
 - Busca de subcadeias
 - Compressão de dados
- Estratégias programativas
 - Divisão-conquista
 - Algoritmos gananciosos
 - Programação dinâmica
- Intratabilidade

Algoritmos de programação

- Busca linear
- Busca dicotómica
- Insertionsort
- Mergesort
- Quicksort
- Conversão numérica
- Reversão numérica
- Máximo, mínimo de um array
- Remoção de duplicados
- Comparação lexicográfica de arrays

Algoritmos no ensino secundário

- Fatorização
- Máximo divisor comum
- Simplificação de frações
- Regra de Ruffini
- Método de Gauss

Algoritmos da escola primária

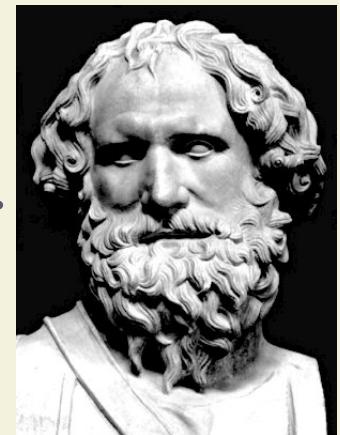
- Adição
- Subtração
- Multiplicação
- Divisão
- (Sucessor)

Algoritmos clássicos

- Algoritmo de Euclides.



- Método para o cálculo de π , de Arquimedes.
- Crivo de Eratóstenes.



Algoritmo de Euclides

- Calcula o máximo divisor comum de dois números inteiros positivos.
- É o mais antigo algoritmo inventado pelo espírito humano.

```
public static int euclidAlgorithm(int x, int y)
{
    while (x != y)
        if (x < y)
            y -= x;
        else
            x -= y;
    return x;
}
```

Isto será um método estático de alguma classe, em Java.

Algoritmo de Euclides, variantes recursivas

- Formulação recursiva

```
public static int euclid(int x, int y)
{
    return x < y ? euclid(x, y-x) : x > y ? euclid(x-y, y) : x;
```

- Formulação recursiva, curto-circuito:

```
public static int greatestCommonDivisor(int x, int y)
{
    int result = x;
    if (y != 0)
        result = greatestCommonDivisor(y, x % y);
    return result;
}
```

Esta versão substitui uma sequência de subtrações $x -= y$, por $x \% y$, até x ser menor que y .

Algoritmo de Euclides, versão de combate

- Normalmente, usamos a seguinte variante iterativa:

```
public static int gcd(int x, int y)
{
    while (y != 0)
    {
        int r = x % y;
        x = y;
        y = r;
    }
    return x;
}
```

Tenha esta sempre à mão!

Comparando as versões iterativas

- Eis uma função de teste para comparar na consolas duas versões iterativas:

```
public static void testIterativeVersions()
{
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        int y = StdIn.readInt();
        int z1 = euclidAlgorithm(x, y);
        StdOut.println(z1);
        int z2 = gcd(x, y);
        StdOut.println(z2);
    }
}
```

Lemos e escrevemos usando a biblioteca stdlib.jar, que teremos juntado ao projeto no Eclipse, importando-a e acrescentando-a ao *build path*.

Classes StdIn e StdOut

- Usaremos a classe StdIn para ler e a classe StdOut para escrever na consola.

StdIn

- isEmpty() : boolean
- hasNextLine() : boolean
- hasNextChar() : boolean
- readLine() : String
- readChar() : char
- readAll() : String
- readString() : String
- readInt() : int
- readDouble() : double
- readFloat() : float
- readLong() : long
- readShort() : short
- readByte() : byte
- readBoolean() : boolean
- readAllStrings() : String[]
- readAllInts() : int[]
- readAllDoubles() : double[]

StdOut

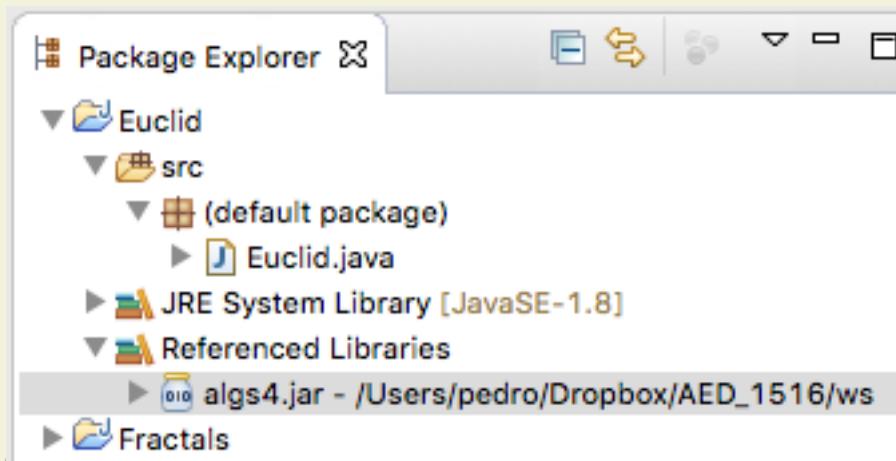
- close() : void
- println() : void
- println(Object) : void
- println(boolean) : void
- println(char) : void
- println(double) : void
- println(float) : void
- println(int) : void
- println(long) : void
- println(short) : void
- println(byte) : void
- print() : void
- print(Object) : void
- print(boolean) : void
- print(char) : void
- print(double) : void
- print(float) : void
- print(int) : void
- print(long) : void
- print(short) : void
- print(byte) : void
- printf(String, Object...) : void

Importando StdIn e StdOut

- Para usar StdIn e StdOut, temos de importar:

```
import edu.princeton.cs.algs4.StdIn;  
import edu.princeton.cs.algs4.StdOut;  
  
public final class Euclid  
{  
    ...  
}
```

- Além disso, temos de acrescentar a biblioteca algs4.jar ao “Build Path”:



No Eclipse, clica-se no projeto, botão direito, Build Path->Add External Archives..., navega-se para a diretoria do workspace e seleciona-se algs4.jar.

A função main

- Em cada classe, a função main apenas chamará as funções de teste.
- Por enquanto, só temos uma função de teste:

```
public static void main(String[] args)
{
    testIterativeVersions();
}
```

- A função main e todas as outras formam a classe Euclid, que teremos desenvolvido no Eclipse.

Classe Euclid

```
import edu.princeton.cs.algs4.StdIn;
import edu.princeton.cs.algs4.StdOut;

public final class Euclid
{
    public static int euclid(int x, int y)
    {
        return x < y ? euclid(x, y - x) : x > y ? euclid(x - y, y) : x;
    }

    // ...

    public static void testIterativeVersions()
    {
        while (!StdIn.isEmpty())
        {
            int x = StdIn.readInt();
            int y = StdIn.readInt();
            int z1 = euclidAlgorithm(x, y);
            StdOut.println(z1);
            int z2 = gcd(x, y);
            StdOut.println(z2);
        }
    }

    public static void main(String[] args)
    {
        testIterativeVersions();
    }
}
```

No Eclipse

Correndo no Eclipse, a interação dá-se na consola do Eclipse, terminando com **ctrl-z** (Windows) ou **ctrl-d** (Mac ou Linux).

The screenshot shows the Eclipse IDE interface. The top part displays the Java code for Euclid.java, which contains methods for testing iterative versions of the Euclidean algorithm and a main method that calls this test function. The bottom part shows the Eclipse Console view where the application's output is displayed. The console output shows several pairs of integers being processed by the algorithm, with the final result being 1.

```
84
85 public static void testIterativeVersions()
86 {
87     while (!StdIn.isEmpty())
88     {
89         int x = StdIn.readInt();
90         int y = StdIn.readInt();
91         int z1 = euclidAlgorithm(x, y);
92         StdOut.println(z1);
93         int z2 = gcd(x, y);
94         StdOut.println(z2);
95     }
96 }
97
98 public static void main(String[] args)
99 {
100     testIterativeVersions();
101 }
102
103 }
```

Euclid [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_65.jdk/Cor
5630 2220
10
10
99990 54540
9090
9090
3774 9897
3
3
7768523 8734658
1
1

Na linha de comando

- Para correr na linha de comando, colocamo-nos na diretoria bin, dentro do projeto, no workspace.
- Aí damos o comando

```
java -cp ../../algs4.jar:. Euclid
```

- Observe:

```
$ pwd  
/Users/pedro/Dropbox/AED_1516/ws  
Pedros-iMac-2:ws pedro$ ls -l  
total 3600  
drwxr-xr-x@ 7 pedro staff 238 Feb 1 17:59 Euclid  
-rw-r--r--@ 1 pedro staff 1841174 Feb 1 16:52 algs4.jar  
Pedros-iMac-2:ws pedro$ cd Euclid/bin  
Pedros-iMac-2:bin pedro$ ls  
Euclid.class  
Pedros-iMac-2:bin pedro$ java -cp ../../algs4.jar:. Euclid  
6770 9100  
10  
10  
70000 65850  
50  
50  
67 45  
1  
1  
$
```

Note que estamos “dentro” da diretoria bin e que a biblioteca algs4.jar está “ao lado” da diretoria Euclid, a qual contém a diretoria bin (e também a diretoria src). Note também que, neste caso, a classe Euclid tem o mesmo nome que o projeto, Euclid, mas nem sempre será assim.

O caminho das classes

- O caminho das classes, em inglês *class path*, é a sequência de diretórias onde a máquina virtual do Java procurará as classes necessárias para correr o programa (através da função `main` da classe invocada).
- Analogamente, o compilador recorre às classes do *class path* para compilar as classes que tiver de compilar.
- Na linha de comando, indica-se a *class path* por meio da opção `-cp`.
- No Eclipse, a *class path* fica guardada automaticamente, quando a definimos usando o comando Build Path e pode ser consultada nas propriedades do projeto: comando Project -> Properties->Java Build Path, separador Libraries.

Testando as funções todas

- Eis uma segunda função de teste, para comparar os resultados das quatro funções:

```
public static void testAllVersions()
{
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        int y = StdIn.readInt();
        int z1 = euclid(x, y);
        int z2 = euclidAlgorithm(x, y);
        int z3 = greatestCommonDivisor(x, y);
        int z4 = gcd(x, y);
        StdOut.printf("%d %d %d %d\n", z1, z2, z3, z4);
    }
}
```

Argumentos na linha de comando

- Selecionamos a função de teste por meio de um argumento na linha de comando:

```
public static void main(String[] args)
{
    char choice = 'A';
    if (args.length > 0)
        choice = args[0].charAt(0);
    if (choice == 'A')
        testIterativeVersions();
    else if (choice == 'B')
        testAllVersions();
    else
        StdOut.println("Illegal option: " + choice);
}
```

```
$ java -cp ../../algs4.jar:. Euclid A
```

```
333 99
```

```
9
```

```
9
```

```
$ java -cp ../../algs4.jar:. Euclid B
```

```
78 5
```

```
1 1 1 1
```

```
400 600
```

```
200 200 200 200
```

```
$ java -cp ../../algs4.jar:. Euclid
```

```
1000 825
```

```
25
```

```
25
```

```
$
```



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 2

Animação algorítmica

Animação algorítmica

- Visualização dos algoritmos
- Utilização do Processing
 - Recapitação
 - Integração com o Java
- Animação do algoritmo de Euclides



Animação

- Os algoritmos são pensamento puro.
- Devemos compreendê-los abstratamente.
- Mas é divertido vê-los a mexer.
- Por exemplo, eis um site com animações muito interessantes de algoritmos de ordenação:
<http://www.sorting-algorithms.com/>.
- De que precisamos para programarmos nós próprios animações destas?
- Precisamos de uma biblioteca gráfica, neste caso uma biblioteca gráfica para o Java.

Processing

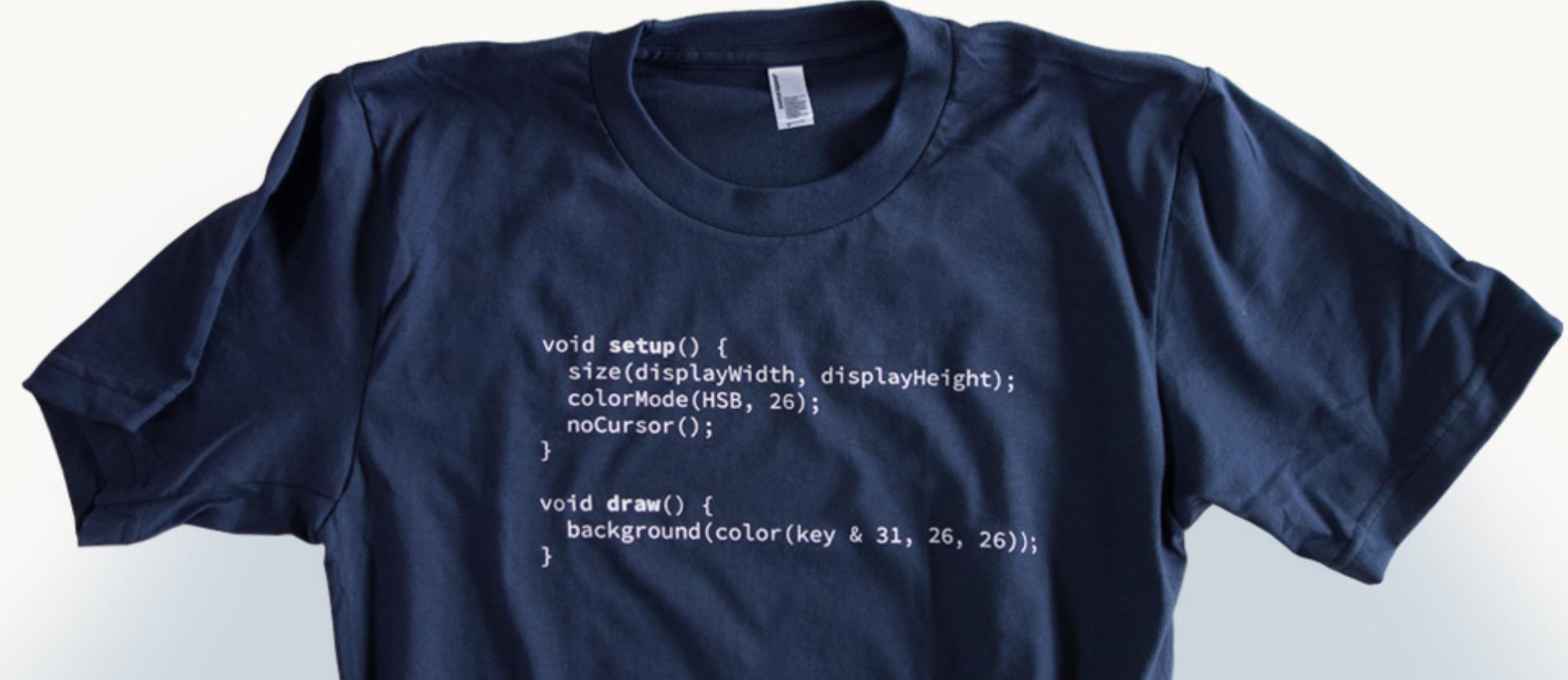
- O Processing é uma linguagem de programação que vem com o seu próprio ambiente de desenvolvimento.
- De certa forma, o Processing é uma “camada” sobre o Java, que pretende ocultar certas particularidades mais “difíceis”.
- Na verdade, o que nos interessa agora é que o Processing traz uma grande biblioteca gráfica, bem documentada, que podemos usar nos nossos programas Java.

Recapitulando...

- Em Processing, um programa é constituído por algumas declarações globais e pelas funções **setup** e **draw**.
- Ambas são chamadas automaticamente.
- A função **setup** é chamada uma vez, no início do programa.
- A função **draw** é chamada repetidamente, 60 vezes por segundo.

O número de vezes que a função draw é chamada por segundo pode ser mudado, com a função frameRate().

setup e draw



```
void setup() {
    size(displayWidth, displayHeight);
    colorMode(HSB, 26);
    noCursor();
}

void draw() {
    background(color(key & 31, 26, 26));
}
```

Java com Processing

```
import processing.core.PApplet;

public class TShirtSketch extends PApplet
{
    private final int displaywidth = 400;
    private final int displayHeight = 300;

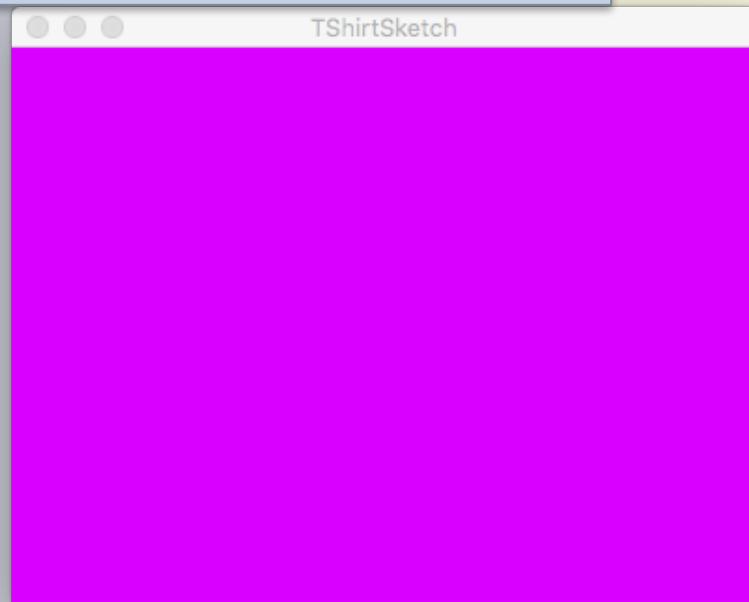
    public void settings()
    {
        size(displaywidth, displayHeight);
    }

    public void setup()
    {
        colorMode(HSB, 26);
        noCursor();
    }

    public void draw()
    {
        background(color(key & 31, 26, 26));
    }

    public static void main(String[] args)
    {
        PApplet.main(new String[] { TShirtSketch.class.getName() });
    }
}
```

Note bem: ao usar o Processing assim, a função size deve ser chamada na função settings e não na função setup.



Teremos acrescentado ao projeto a biblioteca core.jar do Processing, colocando-a na diretoria do workspace.

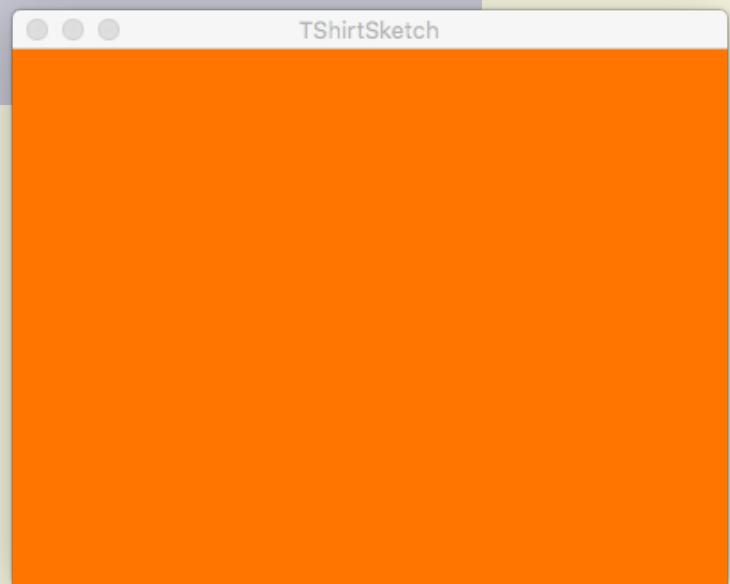
Usando da linha de comando

- Podemos correr o sketch de dentro do Eclipse ou a partir da linha de comando:

```
public class TShirtSketch extends PApplet
{
    // ...

    public static void main(String[] args)
    {
        PApplet.main(new String[] {TShirtSketch.class.getName()});
    }
}

$ pwd
/Users/pedro/Dropbox/AED_1516/ws2
$ cd TShirt/bin
$ ls
TShirtSketch.class
$ java -cp ../../core.jar:. TShirtSketch
```



Animando o algoritmo de Euclides

- Queremos representar cada variável, **x** e **y**, por uma barra vertical, com altura proporcional ao valor da variável.

```
public static int euclidAlgorithm(int x, int y)
{
    while (x != y)
        if (x < y)
            y -= x;
        else
            x -= y;
    return x;
}
```

- A cada passo do ciclo **while** atualizamos o a figura, por exemplo de segundo a segundo (ou de acordo com uma frequência pré-estabelecida).

Animação em direto

- Podemos considerar que no algoritmo de Euclides, em cada passo, calcula-se um novo par de valores $\langle x, y \rangle$ a partir do par de valores $\langle x, y \rangle$ corrente.
- Podemos programar essa transformação assim, na classe Euclid:

```
public static int[] next(int a[])
{
    int[] result = a;
    if (a[0] > a[1])
        result = new int[]{a[0] - a[1], a[1]};
    if (a[0] < a[1])
        result = new int[]{a[0], a[1] - a[0]};
    return result;
}
```

Testando a função next

- Eis uma função de teste que invoca repetidamente a função **next**, a partir de valores lidos, até ambos os valores serem iguais, escrevendo cada par de valores, ao longo do cálculo:

```
public static void testNext()
{
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        int y = StdIn.readInt();
        int[] a = new int[]{x, y};
        Stdout.println(a[0] + " " + a[1]);
        while (a[0] != a[1])
        {
            a = next(a);
            Stdout.println(a[0] + " " + a[1]);
        }
    }
}
```

```
$ java ... Euclid D
78 35
78 35
43 35
8 35
8 27
8 19
8 11
8 3
5 3
2 3
2 1
1 1
```

Esquete de animação

- No esquete de animação, cada novo par será calculado quando for preciso, na função **update**, e as barras que representam as variáveis mudam em conformidade.
- A função update é chamada pela função **draw**, para recalcular a figura, que depois a função **draw** desenhará.

Parametrização geral

- Parametrizamos a largura das barras, a margem esquerda, que é igual à margem direita e o intervalo entre as barras.
- A largura da janela é calculada a partir desses valores e altura da janela é fixa.
- Parametrizamos também a frequência da animação:

```
public class EuclidSketch extends PApplet
{
    private static final int BAR_WIDTH = 40;
    private static final int MARGIN = 40;
    private static final int GAP = 20;

    private static final int WIDTH = 2 * MARGIN + 2 * BAR_WIDTH + GAP;
    private static final int HEIGHT = 720;

    private static final double HERTZ = 1;

    // ...
}
```

Assim evitamos “números mágicos” no nosso programa.

Variáveis da animação

- Registamos o tempo em que a animação começa, o número de passos na animação e o valor do par de inteiros:

```
public class EuclidSketch extends PApplet
{
    // ...
    private double startTime; // time when the animation started
    private int steps;        // number of steps in the animation
    private int[] current;    // current values being shown
    // ...
}
```

- Usamos ainda duas variáveis para representar as cores usadas:

```
private int colorX = color(0, 0, 255); // BLUE;
private int colorY = color(255, 0, 0); // RED;
```

A função `setup`

- A função `setup` lê os dois números da consola e inicializa as variáveis:

```
public void setup()
{
    StdOut.print("Two numbers, please: ");
    int x = StdIn.readInt();
    int y = StdIn.readInt();
    startTime = millis();
    steps = 0;
    current = new int[]{x, y};
}
```

- A função `settings` ocupa-se do tamanho da janela:

```
public void settings()
{
    size(WIDTH, HEIGHT);
}
```

A função `draw`

- A função `draw` é chamada automaticamente, com a frequência determinada pela “frame rate”.
- De cada vez, desenha tudo: primeiro o `background`, diretamente, e depois as barras, invocando o método `display`.
- Entre as duas, invoca a função `update`, que calcula a figura a desenhar:

```
public void draw()
{
    background(100);
    update();
    display();
}
```

A função update

- Em geral, usamos a função **update** para fazer os cálculos que determinam o desenho que a função **draw** deve depois fazer.
- Neste caso, calcula o número de “unidades de tempo” que passaram desde o início; se for maior que o número de passos, está na hora de animação mexer:

```
private void update()
{
    double seconds = (millis() - startTime) / 1000.0;
    int timeUnits = (int)(seconds * HERTZ);
    if (timeUnits > steps)
    {
        current = Euclid.next(current);
        steps++;
    }
}
```

A constante **HERTZ** representa a frequência com que a imagem muda.

No fim, o sketch fica parado na última imagem, mas os cálculos continuam indefinidamente.

Desenhando as barras

- Temos uma função para cada barra e a função **display** para desenhar as duas barras, usando aquelas:

```
private void displayX(int h)
{
    stroke(colorX);
    fill(colorX);
    rect(MARGIN, height-h, BAR_WIDTH, h);
}

private void displayY(int h)
{
    stroke(colorY);
    fill(colorY);
    rect(MARGIN + BAR_WIDTH + GAP, height-h, BAR_WIDTH, h);
}

public void display()
{
    displayX(current[0]);
    displayY(current[1]);
}
```

Cada barra tem a altura correspondente ao valor da variável respetiva.

Correndo na linha de comandos

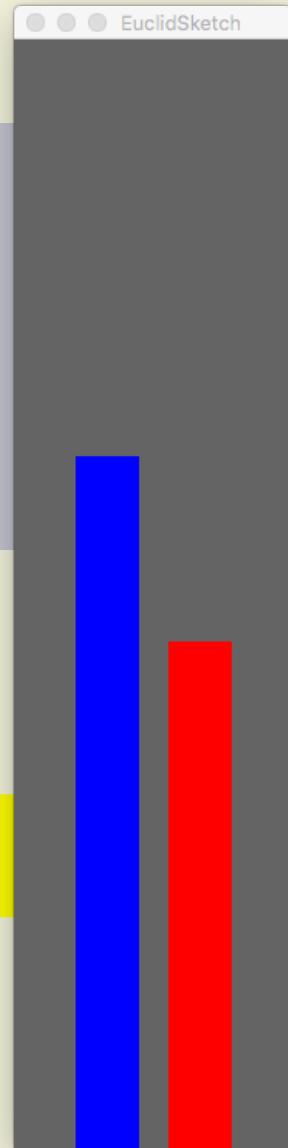
- A função **main** é como de costume:

```
public class EuclidSketch extends PApplet
{
    // ...

    public static void main(String[] args)
    {
        PApplet.main(new String[] {EuclidSketch.class.getName()});
    }
}
```

- Neste caso, ao invocar na linha de comando, temos de indicar as duas bibliotecas:

```
$ java -cp ../../algs4.jar:../../core.jar:. EuclidSketch
Two numbers, please: 450 330
```



Variantes

- Fazer uma cópia da barra mais baixa deslocar-se suavemente para sobre a barra mais alta, antes de esta baixar.
- Desafios:
 - Animar o algoritmo de Arquimedes.
 - Animar o algoritmo de Newton para a raiz quadrada.
 - Animar o algoritmo de Euclides para calcular o máximo divisor comum de um conjunto de números (e não de apenas dois números).
 - Animar o crivo de Eratóstenes.
 - Animar as torres de Hanói.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 3

Coleções: sacos, pilhas e filas

Coleções: sacos, pilhas e filas



- Especificação das coleções
- Classes iteráveis
- Sacos (em inglês *bags*)
 - Implementação com arrays
- Pilhas (em inglês *stacks*)
 - Implementação com listas
- Filas (em inglês *queues*)
 - Implementação como exercício.

Estudaremos hoje os sacos. As pilhas e as filhas ficam para uma das próximas lições.

Coleções

- Coleções são... coleções!
- Às coleções podemos acrescentar objetos, removê-los e “iterá-los”.
- Podemos ainda “perguntar-lhes” se estão vazias e quantos objetos têm.
- Em Java teremos coleções genéricas, parametrizadas pelo tipo dos objetos que contêm. Por exemplo:

```
public class Bag<T>
{
    // ...
}
```

As coleções apenas guardam os objetos. Por isso não há restrições sobre o tipo de objetos que podem conter.

Classes iteráveis

- Queremos que as nossas coleções sejam iteráveis, isto é, que permitam aceder sequencialmente a cada um dos objetos da coleção.
- Programaremos isso fazendo cada coleção dispor de um método **iterator** que devolve um iterador.
- Paralelamente, declaramos a classe como implementando a interface **Iterable<T>**:

```
public class Bag<T> implements Iterable<T>
{
    // ...
}
```

APIs

```
public class Bag<T> implements Iterable<T>
{
    public Bag()
    public void add(T x)
    public boolean isEmpty()
    public int size()
    public Iterator<T> iterator()
}
```

```
public class Stack<T> implements Iterable<T>
{
    public Stack()
    public void push(T x)
    public boolean isEmpty()
    public int size()
    public T pop()
    public Iterator<T> iterator()
}
```

```
public class Queue<T> implements Iterable<T>
{
    public Queue()
    public void enqueue(T x)
    public boolean isEmpty()
    public int size()
    public T dequeue()
    public Iterator<T> iterator()
}
```

Capacidade?

- Como indicamos a capacidade das coleções?
- Não indicamos! Há apenas um construtor, sem argumentos.
- Logo, as coleções têm de ser capazes de aumentar a capacidade, se necessário (e, porventura, de diminuir a capacidade, se desejável).
- Para isso, implementamo-las à base de listas ligadas ou à base de arrays redimensionáveis.

Um array redimensionável é um array cuja capacidade pode mudar durante a execução do programa. Aparentemente a expressão “array redimensionável” é um oximoro, pois habituámo-nos a que a capacidade de um array é fixada na sua criação e não muda mais.

Sacos, implementação

- Vamos implementar a classe **Bag<T>** usando arrays redimensionáveis.
- Haverá um campo **items**, que é o array, e um campo **size**, que representa o número de elementos presentes no array:

```
public class Bag<T> implements Iterable<T>
{
    private T[] items;
    private int size;

    // ...
}
```

Recorde que a capacidade do array está registada no membro `length` do array.

Sacos, construtor

- Inicialmente, o array terá capacidade 1 e tamanho 0:

```
public class Bag<T> implements Iterable<T>
{
    private T[] items;
    private int size;

    public Bag()
    {
        items = (T[]) new Object[1];
        size = 0;
    }

    // ...
}
```

Note bem: criamos um array de objetos, e forçamos a conversão para array de T.

Questão técnica: arrays genéricos

- Nós gostaríamos de ter programado a criação do array genérico items assim:

```
items = new T[1];
```
- No entanto, por razões técnicas, aceitáveis mas complicadas, o Java não permite a criação de arrays genéricos.
- Remedia-se criando um array de objetos **Object** e forçando a conversão para o tipo de arrays pretendido.
- Em geral, forçar conversões de tipo é mau estilo.

Supressão de avisos

- Como, forçar conversões é uma operação arriscada, o Java emite um warning:

```
9  public Bag()
10 {
11     items = (T[]) new Object[1];
12 }
13
14 Type safety: Unchecked cast from Object[] to
15 T[]
```

- Evitamos o warning com uma anotação:

```
@SuppressWarnings("unchecked")
public Bag()
{
    items = (T[]) new Object[1];
    size = 0;
}
```

Só usaremos o `SuppressWarnings` neste caso da construção de arrays genéricos!

Redimensionamento

- Na verdade, não redimensionamos o array: criamos outro array com a capacidade pretendida e fazemos a com que a variável que apontava para o array original passe a apontar para o array novo:

```
private void resize(int capacity)
{
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new Object[capacity];
    for (int i = 0; i < size; i++)
        temp[i] = items[i];
    items = temp;
}
```

Atenção: isto presume que `size <= capacity`.

A memória do array original fica inacessível e será libertada automaticamente pelo *garbage collector* dentro em pouco.

Método add

- Se, ao acrescentar um elemento ao saco, notarmos que já não há espaço no array, redimensionamo-lo para o dobro:

```
public void add(T x)
{
    if (size == items.length)
        resize(2 * items.length);
    items[size++] = x;
}
```

Métodos size e isEmpty

- Estes são muito simples:

```
public int size()  
{  
    return size;  
}
```

```
public boolean isEmpty()  
{  
    return size == 0;  
}
```

O iterador

- Optamos por iterar pela ordem de entrada.
- Usamos uma classe interna:

```
public class Bag<T> implements Iterable<T>
{
    // ...
    private class BagIterator implements Iterator<T>
    {
        private int i = 0;

        public boolean hasNext()
        {
            return i < size;
        }

        public T next()
        {
            return items[i++];
        }

        public void remove()
        {
            throw new UnsupportedOperationException();
        }
    }
}
```

O método remove
não é suportado.

O método iterator

- O método **iterator** apenas cria e retorna um objeto de tipo BagIterator:

```
public Iterator<T> iterator()
{
    return new BagIterator();
}
```

- Mais tarde, para, por exemplo, mostrar o conteúdo de saco de inteiros, b, faremos:

```
for (int x: b)
    stdout.println(x);
```

Exemplo: problema dos estádios

- A federação tem a lista dos estádios, com nome, comprimento do relvado e largura do relvado. Pretende saber quais são os estádios com relvado de área máxima, de entre aqueles cujo relvado tem as dimensões regulamentares.
- Para a FIFA, o comprimento do relvado deve estar entre 90 e 120 metros e a largura entre 45 e 90 metros.
- Cada linha do ficheiro tem o nome (um cadeia de caracteres sem espaços) o comprimento e a largura (ambos números inteiros).

```
drag 110 85
alg_arv 124 90
light 120 80
saintlouis 118 82
alv_xxi 115 80
```

Variante do [problema](#) usado em
Laboratório de Programação 13/14.

Classe Stadium

- Uma classe imutável, para representar estádios:

```
public class Stadium
{
    private final String name;
    private final int length;
    private final int width;

    public static final int MIN_LENGTH = 90;
    public static final int MAX_LENGTH = 120;
    public static final int MIN_WIDTH = 45;
    public static final int MAX_WIDTH = 90;

    Stadium(String name, int length, int width)
    {
        this.name = name;
        this.length = length;
        this.width = width;
    }

    // ...
}
```

Note bem: por opção de desenho, os valores destes membros não podem mudar após a construção.

Classe Stadium, continuação

```
public class Stadium
{
    // ...

    public int area()
    {
        return length * width;
    }

    public boolean isLegal()
    {
        return MIN_LENGTH <= length && length <= MAX_LENGTH
            && MIN_WIDTH <= width && width <= MAX_WIDTH;
    }

    public static Stadium read()
    {
        String s = StdIn.readString();
        int u = StdIn.readInt();
        int w = StdIn.readInt();
        return new Stadium(s, u, w);
    }

    public string toString()
    {
        return name + " " + length + " " + width;
    }
}
```

Note bem: método estático.

Note bem: método redefinido.

Classe do problema

- Usamos um saco de estádios.
- O saco é criado vazio (como todos os sacos) e depois é preenchido por leitura:

```
public class StadiumProblem
{
    private Bag<Stadium> stadiums = new Bag<Stadium>();

    public void read()
    {
        while (!StdIn.isEmpty())
            stadiums.add(Stadium.read());
    }

    // ...
}
```

Estádios selecionados

- Os estádios selecionados também formam um saco:

```
public class StadiumProblem
{
    // ...

    public Bag<Stadium> selected()
    {
        Bag<Stadium> result = new Bag<Stadium>();
        int maxArea = -1;
        for (Stadium x: stadiums)
            if (x.isLegal())
            {
                if (maxArea < x.area())
                {
                    maxArea = x.area();
                    result = new Bag<Stadium>();
                }
                if (x.area() == maxArea)
                    result.add(x);
            }
        return result;
    }
    // ...
}
```

De cada vez que surge uma nova área máxima, recomeça-se com um novo saco.

Método solve

- Resolvemos o problema no método **solve**:

```
public void solve()
{
    Bag<Stadium> solution = selected();
    if (solution.isEmpty())
        StdOut.println("(empty)");
    else
        for (Stadium x : selected())
            StdOut.println(x);
}
```

- A função de teste cria o objeto, lê e resolve:

```
public static void testSolve()
{
    StadiumProblem sp = new StadiumProblem();
    sp.read();
    sp.solve();
}
```

- A função **main** chama a função de teste:

```
public static void main(String [] args)
{
    testSolve();
}
```

Diretoria work

- Tipicamente, guardaremos os nossos ficheiros de dados numa diretoria **work**, dentro do projeto, ao lado das diretórias **bin** e **src** (estas duas criadas pelo Eclipse).
- Nesse caso, podemos correr o programa a partir da diretoria **work**, redirigindo o input, assim, por exemplo:

```
$ pwd  
/Users/pedro/Dropbox/AED_1516/ws2/Stadium  
$ ls  
bin src work  
$ cd work  
$ ls  
lz_in_01.txt lz_in_03.txt lz_in_05.txt lz_in_07.txt  
lz_in_02.txt lz_in_04.txt lz_in_06.txt lz_in_08.txt  
$ java -cp ../../algs4.jar:../../bin StadiumProblem < lz_in_01.txt  
saintlouis 118 82  
$
```

Também podemos redirigir o output, de maneira análoga, claro.

Correndo na diretoria bin

- Alternativamente, podemos correr o programa na diretoria bin, como antes, tendo o cuidado de ir buscar o ficheiro à diretoria work:

```
$ pwd  
/Users/pedro/Dropbox/AED_1516/ws2/Stadium  
$ ls  
bin src work  
$ cd bin  
$ java -cp ../../algs4.jar:. StadiumProblem < ../work/lz_in_07.txt  
(empty)  
$ java -cp ../../algs4.jar:. StadiumProblem < ../work/lz_in_08.txt  
arena 120 90  
stade_france 120 90  
millenium 120 90  
camp_nou 120 90  
bernabeu 120 90  
$
```

Custo do redimensionamento?

- Redimensionar o array que suporta o saco parece ser um grande trabalho suplementar. Será que é?
- Quando redimensionamos de N para $2N$, quantos acessos se fazem aos arrays items e temp?
- O array temp é criado com $2N$ elementos e cada um deles tem de ser inicializado com null. Logo $2N$ acessos.
- A afectação $\text{temp}[i] = \text{items}[i]$ faz-se N vezes. Logo, ao todo, mais $2N$ acessos.
- Portanto, ao redimensionar de 1 para 2, foram 4 acessos; de 2 para 4 foram 8 acessos; ...; de $N/2$ para N foram $2N$ acessos.
- Portanto, se o array tiver N elementos e N for uma potência de 2, teremos usado $4 + 8 + 16 + \dots + 2N$ acessos só para o redimensionamento.
- Ora $4 + 8 + 16 + \dots + 2N = 4N - 4$.
- A estes há que somar os N acessos normais, para acrescentar no novo elemento ao saco.
- Logo, ao todo teremos tido $5N - 4$ acessos, para N elementos no array.
- Note que se tivéssemos alocado logo N elementos na criação, teria havido $2N$ acessos ao array, ao todo, mas em geral não sabemos o número de elementos exato, por isso, ao dimensionar por excesso logo de início pode acontecer trabalharmos mais do que ir redimensionando aos poucos.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 4

Interlúdio: arrays à base de sacos

Interlúdio: arrays à base de sacos

- Arrays em Java e arrays em C
- Classe ArrayBag<T>.
- Interfaces funcionais.
- Expressões lambda.



Arrays em Java e arrays em C

- Em C, um array é um pedaço de memória bruto.
- Em Java, um array é um pedaço de memória não completamente bruto: por exemplo, se o índice estiver fora dos limites, o array reclama, lançando uma exceção.
- Em C, mesmo quando processamos um array usando índices, estamos de facto a usar aritmética de apontadores.
- Em Java não há aritmética de apontadores.
- A aritmética de apontadores em C permite-nos considerar pedaços do array com sendo subarrays, o que é muito prático.
- Em Java, a classe **Arrays** contém uma série de métodos estáticos com operações habituais sobre arrays.

Em C, podemos fazer contas com apontadores, e, por exemplo, criar um apontador a partir de outro, por via de uma operação “aritmética”. Em Java, a única operação que podemos fazer com apontadores é copiá-los.

ArrayList<T>

- **ArrayList<T>** é uma classe genérica que implementa arrays automaticamente redimensionáveis.
- Tem algumas operações habituais para arrays, como o acesso por índice, para consulta e modificação, e busca sequencial, mas não tem métodos de ordenação.
- É uma classe iterável.
- Se precisamos de ordenar arrays, então usamos arrays mesmo e os métodos estáticos da classe **Arrays**.

Os sacos não são arrays

- A característica fundamental dos arrays é permitirem aceder aos elementos, por índice, em tempo constante.
- Na classe **Bag<T>** prescindimos voluntariamente de acrescentar métodos de acesso por índice, porque isso não faz parte da natureza dos sacos.
- Mas podemos criar os nossos próprios arrays redimensionáveis, usando a técnicas que vimos com a classe **Bag<T>**.
- E podemos ir acrescentando os métodos que entendermos serem interessantes.

Classe ArrayBag<T>

- Para transformar um saco num array, basta acrescentar o acesso por índice:
- Na verdade, acrescentaremos dois acessos por índice: o acesso *absoluto*, **get**, que falha se o índice estiver fora dos limites, e o acesso modular, **at**, que usa o resto da divisão do índice indicado pelo tamanho do array:

```
public class ArrayBag<T> implements Iterable<T>
{
    public ArrayBag()
    public void add(T x)
    public boolean isEmpty()
    public int size()
    public Iterator<T> iterator()
    public T get(int x)
    public T at(int x)
}
```

Classe ArrayBag<T>, implementação

- Os membros de dados são como na classe Bag<T>, e as operações comuns são análogas:

```
public class ArrayBag<T> implements Iterable<T>
{
    private T[] items;
    private int size;

    // ...
}
```

- Mesmo assim, acrescentamos um segundo construtor, que especifica a capacidade inicial:

```
public ArrayBag()
{
    this(1);
}
```

```
@SuppressWarnings("unchecked")
public ArrayBag(int n)
{
    items = (T[]) new Object[n];
    size = 0;
}
```

Outros construtores

- O construtor de cópia:
- Frequentemente, teremos um array, com os elementos do qual queremos criar um **ArrayBag**:
- Mais geralmente, teremos um iterador:
- ... ou então um objeto de uma classe iterável:

```
public ArrayBag(ArrayBag<T> other)
{
    items = other.items.clone();
    size = other.size;
}
```

```
public ArrayBag(T[] a)
{
    items = a.clone();
    size = a.length;
}
```

```
public ArrayBag(Iterator<T> it)
{
    this();
    while (it.hasNext())
        add(it.next());
}
```

```
public ArrayBag(Iterable<T> x)
{
    this(x.iterator());
}
```

Métodos get e at

- O método **get** falha se o argumento representar um índice inválido:

```
public T get(int x)
{
    assert 0 <= x;
    assert x < size;
    return items[x];
}
```

- O método **at** é mais geral: se o argumento não representar um índice válido, é corrigido modularmente:

```
public T at(int x)
{
    // Note: in Java (-5)%3 is -2;
    assert size > 0;
    x = x % size;
    if (x < 0)
        x += size;
    return items[x];
}
```

Assim, a.at(-1) representa o último elemento do array a.

Contando

- Dado um array, frequentemente queremos contar o número de ocorrências de um dado valor.
- Com mais generalidade, queremos contar o número de elementos que verificam uma dada propriedade.
- A propriedade é representado por um predicado:

```
public int count(Predicate<T> p)
{
    int result = 0;
    for (int i = 0; i < size; i++)
        if (p.test(items[i]))
            result++;
    return result;
}
```

Tecnicamente, **Predicate<T>** é uma interface funcional que representa as funções booleanas com um argumento de tipo T.

Testando com expressões lambda

- Eis uma função de teste, que consulta o array para saber quantas palavras existem com o comprimento obtido interativamente:

```
public static void testCount()
{
    String[] cities = new String[] {
        "lisboa", "porto", "faro", "coimbra", "tavira",
        "aveiro", "braga", "viseu", "beja", "leiria",
        "guarda", "chaves", "portalegre", "funchal", "horta"};
    ArrayBag<String> a = new ArrayBag<>(cities);
    while (!StdIn.isEmpty())
    {
        int n = StdIn.readInt();
        int z = a.count(x -> x.length() == n);
        StdOut.println(z);
    }
}
```

A expressão a vermelho é
uma expressão *lambda*.

Filtrando

- Mais interessante do que saber quantos são os que verificam o predicado é saber quais são.
- Chama-se a isto filtrar o array (segundo o predicado).
- O resultado é outro array, bem entendido:

```
public ArrayBag<T> filter(Predicate<T> p)
{
    ArrayBag<T> result = new ArrayBag<>();
    for (int i = 0; i < size; i++)
        if (p.test(items[i]))
            result.add(items[i]);
    return result;
}
```

Testando a filtração

- Eis uma função de teste, que filtra as palavras que terminam pela cadeia lida na consola, e depois mostra o conteúdo do array resultado:

```
public static void testFilter()
{
    String[] cities = new String[] {
        "lisboa", "porto", "faro", "coimbra", "tavira",
        "aveiro", "braga", "viseu", "beja", "leiria",
        "guarda", "chaves", "portalegre", "funchal", "horta"};
    ArrayBag<String> a = new ArrayBag<>(cities);
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readLine();
        ArrayBag<String> z = a.filter(x -> x.endsWith(s));
        for (String i : z)
            StdOut.println(i);
    }
}
```

Visitação

- Visitar uma coleção é usar cada elemento da coleção com argumento de uma dada operação.
- Com arrays, exprime-se assim:

```
public void visit(Consumer<T> a)
{
    for (int i = 0; i < size; i++)
        a.accept(items[i]);
}
```

Consumer<T> é a interface funcional que representa as funções com um argumento de tipo T e resultado void.

- Com isto, a parte final da função de teste da página anterior fica mais elegante:

```
public static void testFilter()
{
    ...
    while (!StdIn.isEmpty())
    {
        String s = StdIn.readLine();
        ArrayBag<String> z = a.filter(x -> x.endsWith(s));
        z.visit(x -> StdOut.println(x));
    }
}
```

Máximo

- O máximo de um array calcula-se sempre com relação a uma função de comparação.
- Quer dizer, o máximo de um array é aquele elemento que nunca dá negativo quando qualquer um dos outros compara com ele.
- Calcula-se assim:

```
public T max(Comparator<T> cmp)
{
    assert size > 0;
    T result = items[0];
    for (int i = 1; i < size; i++)
        if (cmp.compare(result, items[i]) < 0)
            result = items[i];
    return result;
}
```

Comparator <T> é a interface funcional que representa as funções de comparação do tipo T.

Testando o máximo

- Observe, notando que para achar o mínimo basta trocar o sinal do resultado da comparação:

```
public static void testMax()
{
    String[] cities = new String[] {
        "lisboa", "porto", "faro", "coimbra", "tavira",
        "aveiro", "braga", "viseu", "beja", "leiria",
        "guarda", "chaves", "portalegre", "funchal", "horta"};
    ArrayBag<String> a = new ArrayBag<>(cities);
    // the last (max) in alphabetical order
    String s1 = a.max(x, y) -> x.compareTo(y));
    // the first (min) in alphabetical order
    String s2 = a.max(x, y) -> -x.compareTo(y));
    // the first longest
    String s3 = a.max(x, y) -> x.length() - y.length());
    // the first shortest
    String s4 = a.max(x, y) -> -(x.length() - y.length()));
    StdOut.printf("%s %s %s %s\n", s1, s2, s3, s4);
}
```

ArrayBags no problema dos estádios

- Em vez de `Bag<Stadium>` usamos `ArrayBag<Stadium>`:

```
public class StadiumProblem2016
{
    private ArrayBag<Stadium> stadiums = new ArrayBag<>();
    // ...
}
```

- A função `selected` fica assim:

```
public ArrayBag<Stadium> selected()
{
    ArrayBag<Stadium> result = stadiums.filter(x -> x.isLegal());
    if (!result.isEmpty())
    {
        Stadium max = result.max((x, y) -> x.area() - y.area());
        result = result.filter(x -> x.area() == max.area());
    }
    return result;
}
```

Palavras para quê?

E ainda, mapeamento

- Mapear um array é construir um array em que cada elemento resulta de uma dada transformação do elemento correspondente do array objeto:

```
public <R> ArrayBag<R> map(Function<T, R> f)
{
    ArrayBag<R> result = new ArrayBag<>();
    for (int i = 0; i < size; i++)
        result.add(f.apply(items[i]));
    return result;
}
```

Function<T, R> é a interface funcional que representa as funções com um argumento de tipo T e resultado de tipo R.

Também existe **UnaryOperator<T>**, que é a interface funcional que representa as funções com um argumento de tipo T e resultado de tipo T, e pode ser usada como argumento de map

E, para terminar, enrolamento

- Enrolar um array é aplicar sucessivamente uma dada função binária ao resultado anterior e ao elemento seguinte do array, sendo dado também o resultado inicial:

```
public <R> R fold(BiFunction<R, T, R> f, R zero)
{
    R result = zero;
    for (int i = 0; i < size; i++)
        result = f.apply(result, items[i]);
    return result;
}
```

BiFunction<T, U, V> é a interface funcional que representa as funções binárias com um primeiro argumento de tipo T, segundo argumento de tipo U e resultado de tipo R.

Testando o enrolamento

- Observe com atenção:

```
public static void testFold()
{
    BinaryOperator<Integer> plus = (x, y) -> x + y;
    BinaryOperator<String> concat = (x, y) -> x + y;
    BiFunction<Integer, Integer, Integer> plusf = (x, y) -> x + y;
    BiFunction<String, String, String> concatf = (x, y) -> x + y;
    ArrayBag<Integer> b = new ArrayBag<>(new Integer[]{4, 8, 2, 7});
    ArrayBag<String> s = new ArrayBag<>(new String[]{"aaa", "bbb", "ccc", "ddd"});
    int z1 = b.fold(plus, 0);
    StdOut.println(z1);
    String z2 = s.fold(concat, "");
    StdOut.println(z2);
    int z3 = b.fold(plusf, 0);
    StdOut.println(z3);
    String z4 = s.fold(concatf, "");
    StdOut.println(z4);
    String r1 = b.mkString("(((", "--", ")))");
    StdOut.println(r1);
    String r2 = s.mkString("<", "", ">");
    StdOut.println(r2);
}
```

```
$ java -cp ../../algs4.jar:. ArrayBag D
Test fold
21
aaabbccddd
21
aaabbccddd
(((4--2--7)))
<aaa,bbb,ccc,ddd>
$
```

Neste exemplo, as expressões lambda vêm representadas por variáveis, para ilustrar a técnica, mas isso não é essencial. Notamos que operadores binários podem ser tratados como funções binárias.

“Make string”

- Eis uma função que transforma um array bag numa string, indicando o separador:

```
public String mkString(String separator)
{
    String result = "";
    if (size > 0)
    {
        result = items[0].toString();
        for (int i = 1; i < size; i++)
            result += separator + items[i];
    }
    return result;
}
```

- E uma variante que indica também o prefixo e o sufixo

```
public String mkString(String start, String separator, String end)
{
    return start + mkString(separator) + end;
}
```

Exemplo, a área média

- Eis uma função para calcular a média da áreas dos estádios com dimensão legal, assumindo que há pelo menos um estádio desses:

```
double averageLegalArea()
{
    ArrayBag<Stadium> legal = stadiums.filter(x -> x.isLegal());
    assert !legal.isEmpty();
    ArrayBag<Integer> areas = legal.map(x -> x.area());
    int total = areas.fold((x, y) -> x + y, 0);
    return (double) total / legal.size();
}
```

Moral da história: nos processamentos sequenciais, faz-se quase tudo com **map**, **filter** e **fold**!



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 5

Pilhas

Pilhas



- Implementação com arrays redimensionáveis.
- Implementação com listas ligadas.
- Aplicação: avaliação de expressões aritméticas usando o algoritmo da pilha dupla, de Dijkstra.

Pilhas

- As pilhas chamam-se pilhas porque funcionam como pilhas: pilhas de livros, de pratos, de papéis; não pilhas no sentido de baterias elétricas!
- A ideia é que o elemento que sai da pilha é o último que entrou.
- Dizemos que as pilhas são coleções LIFO: *last in first out.*



API da pilha

- Já sabemos:

```
public class Stack<T> implements Iterable<T>
{
    public Stack()
    public void push(T x)
    public boolean isEmpty()
    public int size()
    public T pop()
    public Iterator<T> iterator()
}
```

- Comparando com **Bag<T>**, temos **push** em vez de **add**, temos **pop**, que antes não havia, e o iterador emitirá os elementos por ordem inversa de entrada na pilha (isto é, os que entraram mais recentemente surgirão primeiro, no iterador).

Implementação com arrays redimensionáveis

- É análoga à da classe **Bag<T>**:

```
public class Stack<T> implements Iterable<T>
{
    private T[] items;
    private int size;

    public Stack()
    {
        // ...
    }

    private void resize(int capacity)
    {
        // ...
    }

    public void push(T x)
    {
        if (size == items.length)
            resize(2 * items.length);
        items[size++] = x;
    }

    // ...
}
```

Se o array cresce quando está cheio, será que deve encolher quando fica relativamente vazio?

Encolhendo no pop

- A estratégia será encolher para metade quando o tamanho estiver a 25% da capacidade:

```
public class Stack<T> implements Iterable<T>
{
    // ...

    public T pop()
    {
        T result = items[--size];
        items[size] = null;
        if (size > 0 && size == items.length / 4)
            resize(items.length / 2);
        return result;
    }

    // ...
}
```

Anula-se o apontador para que não permaneça no array uma referência para o elemento removido, o que atrasaria a *garbage collection*. Se não anulássemos, teríamos “vadiagem” (em inglês, *loitering*): referências no array para valores inúteis.

Note bem: não seria sensato encolher para metade quando o array estivesse meio cheio, pois nesse caso ele ficava cheio logo após ter encolhido, e se houvesse um push a seguir, teria de crescer de novo.

Iterador reverso

- Nas pilhas, queremos iterar os elementos de maneira a ver primeiro os que entraram na pilha mais recentemente:

```
// ...

private class StackIterator implements Iterator<T>
{
    private int i = size;

    public boolean hasNext()
    {
        return i > 0;
    }

    public T next()
    {
        return items[--i];
    }

    // ...
}

public Iterator<T> iterator()
{
    return new StackIterator();
}

// ...
```

Função de teste

- Eis uma função de teste, que empilha os números lidos e desempilha com ‘-’. No fim, indica o tamanho, indica se é vazia, enumera os elementos, de várias maneiras e indica a capacidade.

```
private static void testStackInteger()
{
    Stack<Integer> s = new Stack<Integer>();
    while (!StdIn.isEmpty())
    {
        String t = StdIn.readString();
        if ("-".equals(t))
            s.push(Integer.parseInt(t));
        else if (!s.isEmpty())
        {
            int x = s.pop();
            StdOut.println(x);
        }
    }
    StdOut.println("size of stack = " + s.size());
    StdOut.println("is stack empty? " + s.isEmpty());
    StdOut.print("items:");
    for (int x : s)
        StdOut.print(" " + x);
    StdOut.println();
    s.visit(x -> StdOut.print(x));
    StdOut.println();
    StdOut.println(s.mkString(","));
    StdOut.println("capacity = " + s.capacity());
}
```

```
$ java ... Stack
Stack of Integers
7 3 9 2 15 6
-
6
-
15
-
2
size of stack = 3
is stack empty? false
items: 9 3 7
937
9,3,7
capacity = 8
$
```

Ctrl-D aqui!

Implementação com listas ligadas

- A lista ligada é programada em termos da classe interna **Node**, a qual detém um campo para o valor e uma referência para o próximo nó.
- Para distinguir, colocamos a nova classe no pacote **alt**:

```
package alt;

public class Stack<T> implements Iterable<T>
{
    private class Node
    {
        private T value;
        private Node next;

        Node(T x, Node z)
        {
            value = x;
            next = z;
        }
    }

    // ...
}
```

Ver [transparentes de POO](#),
página 13-26 e seguintes.

O primeiro nó

- Na pilha, um membro representa o primeiro nó da lista; outro membro representa o tamanho:

```
package alt;

public class Stack<T> implements Iterable<T>
{
    // ...

    private Node first;
    private int size;

    // ...
}
```

- Omitimos o construtor por defeito, uma vez que os valores iniciais automáticos (**null** para **first** e 0 para **size**) são os que nos interessam.

Empilhar, desempilhar

- Para empilar, criamos um novo nó, que passa a ser o primeiro, ligando-o ao que era o primeiro antes:

```
public void push(T x)
{
    first = new Node(x, first);
    size++;
}
```

- Para desempilar, devolvemos o valor do primeiro nó, e avançamos a referência para o seguinte:

```
public T pop()
{
    T result = first.value;
    first = first.next;
    size--;
    return result;
}
```

isEmpty, size

- Os métodos **isEmpty** e **size** são muito simples:

```
public boolean isEmpty()
{
    return first == null;
}

public int size()
{
    return size;
}
```

Poderíamos ter omitido o campo **size** e ter programado a função **size** contando iterativamente os nós, mas, por decisão de desenho, preferimos que função **size** seja calculada em tempo constante.

Iterador de listas

- O cursor é uma referência para o nó corrente:

```
private class StackIterator implements Iterator<T>
{
    private Node cursor = first;

    public boolean hasNext()
    {
        return cursor != null;
    }

    public T next()
    {
        T result = cursor.value;
        cursor = cursor.next;
        return result;
    }

    // ...
}

public Iterator<T> iterator()
{
    return new StackIterator();
```

Repare que o nó **first** contém o valor que mais recentemente foi empilhado, de entre todos os que estão na pilha.

Tal como na outra implementação, os valores são enumerados pela ordem inversa de entrada na pilha.

Função de teste

- Eis uma função de teste, análoga à das outras pilhas:

```
private static void testStackInteger()
{
    Stack<Integer> s = new Stack<>();
    while (!StdIn.isEmpty())
    {
        String t = StdIn.readString();
        if (!"-".equals(t))
            s.push(Integer.parseInt(t));
        else if (!s.isEmpty())
        {
            int x = s.pop();
            StdOut.println(x);
        }
    }
    StdOut.println("size of stack = " + s.size());
    StdOut.println("is stack empty? " + s.isEmpty());
    StdOut.print("items:");
    for (int x : s)
        StdOut.print(" " + x);
    StdOut.println();
}
```

```
$ ls
ArrayBag$ArrayBagIterator.class
Stack.class
ArrayBag.class
Stadium.class
Bag$BagIterator.class
StadiumProblem.class
Bag.class
StadiumProblem2017.class
Stack$StackIterator.class
alt
$ ls alt
Stack$Node.class
Stack.class
Stack$StackIterator.class
$ java ... alt.Stack
Stack of Integers
7 4 12 9 5 23 8 3
-
3
-
8
size of stack = 6
is stack empty? false
items: 23 5 9 12 4 7
$
```

Avaliação de expressões aritméticas

- Como exemplo de aplicação das pilhas, estudemos a avaliação de expressões aritméticas usando o algoritmo da pilha dupla de [Dijkstra](#).
- As expressões aritmética que nos interessam são completamente parentetizadas, para evitar as questões da precedência dos operadores.
- Admitiremos os quatro operadores básicos (“ $+$ ”, “ $-$ ”, “ $*$ ” e “ $/$ ”) e a raiz quadrada (“ sqrt ”).
- Os operandos são números decimais, com ou sem parte decimal.
- Exemplos:
 $((7*2)+5)$
 $((\text{sqrt}(9+16))/3)$
 $((6.5*3.0)+(7.1-3.3))/2.5$

Pilha dupla de Dijkstra

- Usamos duas pilhas: uma pilha de cadeias, para os operadores, e uma pilha de números, para os operandos e para os resultados parciais.
- Varremos a expressão da esquerda para a direita, recolhendo os operadores, os operandos e os parêntesis:
 - Cada operador é empilhado na pilha dos operandos.
 - Cada operando é empilhado na pilha dos números.
 - Os parêntesis a abrir são ignorados.
 - Ao encontrar um parêntesis a fechar, desempilha-se um operador, depois desempilham-se os operandos, em número apropriado ao operador, aplica-se a operação a esses operandos e empilha-se o resultado na pilha dos números.
 - Depois de o último parêntesis ter sido processado, a pilha dos números terá um único elemento, cujo valor é o valor da expressão.

Classe DijkstraTwoStack

- Primeiro as operações de suporte:

```
public class DijkstraTwoStackBasic
{
    private DijkstraTwoStackBasic() { }

    private static String[] binaryOperators = {"+", "-", "*", "/"};
    private static String[] unaryOperators = {"sqrt"};
    private static String[] knownOperators =
        Utils.concatenate(binaryOperators, unaryOperators);

    private static boolean isOperator(String s)
    {
        return Utils.has(knownOperators, s);
    }

    private static boolean isBinaryOperator(String s)
    {
        return Utils.has(binaryOperators, s);
    }

    private static boolean isUnaryOperator(String s)
    {
        return Utils.has(unaryOperators, s);
    }

    private static boolean isNumeric(String s)
    {
        return Character.isDigit(s.charAt(0));
    }

    // ...
}
```

Algumas funções utilitárias

- Vamos reunindo da classe Utils as funções utilitárias que formos “descobrindo”:

```
public final class Utils
{
    private Utils() { }

    // generic array utilities

    public static <T> int find(T[] a, T x)
    {
        for (int i = 0; i < a.length; i++)
            if (x.equals(a[i]))
                return i;
        return -1;
    }

    public static <T> boolean has(T[] a, T x)
    {
        return find(a, x) != -1;
    }

    public static <T> T[] concatenate(T[] a, T[] b)
    {
        T[] result = newArrayLike(a, a.length + b.length);
        for (int i = 0; i < a.length; i++)
            result[i] = a[i];
        for (int i = 0; i < b.length; i++)
            result[a.length + i] = b[i];
        return result;
    }
}
```

Criação de arrays genéricos

- Aproveitamos para definir três funções para criar arrays genéricos:

```
public final class utils
{
    // ...

    // creation of generic arrays

    @SuppressWarnings("unchecked")
    public static <T> T[] newArrayLike(T[] a, int n)
    {
        T[] result = (T[]) java.lang.reflect.Array.newInstance(a.getClass().getComponentType(), n);
        return result;
    }

    @SuppressWarnings("unchecked")
    public static <T> T[] newArrayWith(T a, int n)
    {
        T[] result = (T[]) java.lang.reflect.Array.newInstance(a.getClass(), n);
        return result;
    }

    @SuppressWarnings("unchecked")
    public static <T> T[] newArrayOf(class<T> c, int n)
    {
        T[] result = (T[]) java.lang.reflect.Array.newInstance(c, n);
        return result;
    }
}
```

Avaliação da expressão aritmética

- A seguinte função meramente exprime as regras do algoritmo:

```
public static double evaluationSimple(String[] expression)
{
    Stack<String> operators = new Stack<String>();
    Stack<Double> values = new Stack<Double>();
    for (String token : expression)
        if (".".equals(token)) { } // nothing to do
        else if (isOperator(token)) operators.push(token);
        else if (isNumeric(token)) values.push(Double.parseDouble(token));
        else if (")".equals(token))
    {
        double x = 0;
        double y = 0;
        // ...
        double z;
        // ...
        values.push(z);
    }
    else throw new UnsupportedOperationException();
    return values.pop();
}
```

A expressão é representada pelo array dos tóquenes: operandos, operadores e parêntesis.

Aplicação dos operadores

```
// ...
{
    double x = 0;
    double y = 0;
    String op = operators.pop();
    if (isUnaryOperator(op))
        x = values.pop();
    else if (isBinaryOperator(op))
    {
        y = values.pop();
        x = values.pop();
    }
    else
        throw new UnsupportedOperationException(); // will not happen
    double z;
    if ("+".equals(op))
        z = x + y;
    else if ("-".equals(op))
        z = x - y;
    else if ("*".equals(op))
        z = x * y;
    else if ("/".equals(op))
        z = x / y;
    else if ("sqrt".equals(op))
        z = Math.sqrt(x);
    else throw new UnsupportedOperationException(); // will not happen;
    values.push(z);
}
// ...
```

Comprido e
desinteressante.

Função de teste

- Para abreviar, determinamos que os tóquenes na expressão vêm separados por um espaço.
- Assim, podemos obter o array de tóquenes diretamente, usando a função **split**, da classe String:

```
public static void testEvaluationSimple()
{
    while (StdIn.hasNextLine())
    {
        String line = StdIn.readLine();
        double z = evaluationSimple(line.split(" "));
        StdOut.println(z);
    }
}
```

split

public **String[]** split(String regex)

Splits this string around matches of the given regular expression.

O método **split**, da classe **String**, devolve um array de **String**.

Experiência

- Na linha de comando:

```
$ java -cp ../../algs4.jar:. DijkstraTwoStackBasic
( 8 + 1 )
9.0
( sqrt 2 )
1.4142135623730951
( ( 6 + 9 ) / 3 )
5.0
( ( 5 + 1 ) / ( 7 + 17 ) )
0.25
( sqrt ( 1 + 1 ) ) ) ) ) ) ) )
1.6118477541252516
```

- Próximas tarefas:
 - Evitar ter de separar os tóquenes por espaços.
 - Evitar ter de distinguir os operadores com if-else em cascata

mkString para arrays

- Há de dar jeito:

```
public final class Utils
{
    // ...

    public static <T> String mkString(T[] a)
    {
        return mkString(a, " ");
    }

    public static <T> String mkString(T[] a, String separator)
    {
        String result = "";
        if (a.length > 0)
        {
            result += a[0];
            for (int i = 1; i < a.length; i++)
                result += separator + a[i];
        }
        return result;
    }

    public static <T> String mkString(T[] a, String before, String separator, String after)
    {
        return before + mkString(a, separator) + after;
    }
}
```

Também haverá uma variante para arrays de int, pois o tipo int não é uma classe.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 6

Avaliação de expressões aritméticas

Avaliação de expressões aritméticas

- Utilização de tabelas de dispersão.
- Toquenização de cadeias de caracteres.
- Expressões regulares.

Esta lição antecipa matérias que veremos mais adiante com mais profundidade.



Tabela de operações

- Na avaliação de expressões aritméticas, faz-nos falta uma tabela de operadores, que associe a cada sinal de operação o respetivo operador.
- Cada sinal de operação é uma cadeia de caracteres.
- Cada operador é um **DoubleBinaryOperator** ou um **DoubleUnaryOperator**.
- Portanto, precisamos de duas tabelas: uma para as operações binárias e outra para as operações unárias.

Interface Table<K,V>

- Uma tabela é uma classe que implementa a seguinte interface:

```
public interface Table<K, V>
{
    public V get(K key);
    public void put(K key, V value);
    public void delete(K key);
    public boolean has(K key);
    public boolean isEmpty();
    public int size();
    public Iterable<K> keys();
}
```

- A tabela guarda pares <K,V>. Os K são as chaves; os V são os valores.
- O acesso aos valores faz-se através da chave.

Tabela de dispersão: HashTable<K,V>

- Dispomos de uma classe para tabelas, que usa a técnica da *dispersão de chaves*:

```
public class HashTable<K, V> implements Table<K, V>, Iterable<K>
{
    private int size;
    private int capacity;
    private K[] keys;
    private V[] values;

    public HashTable()
    {
        // ...
    }

    // ...
}
```

Internamente, as chaves e os valores estão guardados em arrays, mas isso não nos diz respeito.

- Mais adiante, estudaremos isto; agora, só queremos usar.

Preenchimento das tabelas

- As tabelas são preenchidas no construtor:

```
public class DijkstraTwoStack
{
    private HashTable<String, DoubleBinaryOperator> ops2;
    private HashTable<String, DoubleUnaryOperator> ops1;

    public DijkstraTwoStack()
    {
        ops2 = new HashTable<>();
        ops2.put("+", (x, y) -> x + y);
        ops2.put("-", (x, y) -> x - y);
        ops2.put("*", (x, y) -> x * y);
        ops2.put("/", (x, y) -> x / y);
        ops2.put("%", (x, y) -> x % y);
        ops2.put("max", (x, y) -> x >= y ? x : y);
        ops2.put("min", (x, y) -> x <= y ? x : y);
        ops2.put("^", (x, y) -> Math.pow(x, y));

        ops1 = new HashTable<>();
        ops1.put("inv", x -> 1/x);
        ops1.put("sq", x -> x*x);
        ops1.put("sqrt", x -> Math.sqrt(x));
        // ...
    }
}
```

Temos aqui mais operações do que no exercício da lição anterior.

Avaliação

```
private double evaluationSimple(String[] expression)
{
    Stack<String> operators = new Stack<String>();
    Stack<Double> values = new Stack<Double>();
    for (String token : expression)
        if (".".equals(token)) { } // nothing to do
        else if (isOperator(token)) operators.push(token);
        else if (isNumeric(token)) values.push(Double.parseDouble(token));
        else if (")".equals(token))
        {
            String op = operators.pop();
            DoubleBinaryOperator dbo = ops2.get(op);
            if (dbo != null)
            {
                // ...
            }
            else
            {
                DoubleUnaryOperator duo = ops1.get(op);
                if (duo != null)
                {
                    // ...
                }
                else
                    assert false; // cannot happen.
            }
        }
        else throw new UnsupportedOperationException();
    return values.pop();
}
```

Avaliação dos operadores

```
private double evaluationSimple(String[] ss)
{
    // ...
    else if ("").equals(s))
    {
        String op = operators.pop();
        DoubleBinaryOperator dbo = ops2.get(op);
        if (dbo != null)
        {
            double y = values.pop();
            double x = values.pop();
            double z = dbo.applyAsDouble(x, y);
            values.push(z);
        }
        else
        {
            DoubleUnaryOperator duo = ops1.get(op);
            if (duo != null)
            {
                double x = values.pop();
                double z = duo.applyAsDouble(x);
                values.push(z);
            }
            else
                // ...
        }
    }
}
```

Toquenização

- Precisamos de transformar a cadeia lida na sequência de tóquenes para avaliação.
- Por exemplo, em “sqrt(3.0*3.0+4.0*4.0)*2”, os tóquenes são “sqrt”, “(”, “3.0”, “*”, “3.0”, “+”, “4.0”, “*”, “4.0”, “)”, “*”, “2”.
- Quer dizer, um tóquene é ou um identificador ou um número ou um sinal de operação ou um parêntesis a abrir ou um parêntesis a fechar.
- Os espaços são apenas separadores, sem outro valor.
- Há números inteiros e números com parte decimal.
- No nosso caso, um sinal de operação é formado por um único caractere, que não é nem uma letra nem um algarismo, mas, em geral, pode ser uma cadeia arbitrária, desde que não comece por parêntesis ou por algarismo.

Classe Utils

- Esta classe, apresentada na lição anterior, reúne funções geralmente úteis, algumas sobre cadeias de caracteres.
- Neste problema, precisamos de analisar sequências de caracteres dentro da cadeia.
- Usaremos as seguintes funções:

```
public class Utils
{
    public static int countwhile(String s, Predicate<Character> p)
    {
        int n = s.length();
        int result = 0;
        while (result < n && p.test(s.charAt(result)))
            result++;
        return result;
    }
    // ...
}
```

Quantos caracteres seguidos no início da cadeia satisfazem a propriedade representada pelo predicado?

Utils: take e drop; takeWhile e dropWhile

- Observe:

```
public static String take(String s, int n)
{
    assert n > 0;
    if (n > s.length())
        n = s.length();
    return s.substring(0, n);
}
```

Uma cadeia formada pelos **n** primeiros caracteres da cadeia dada.

```
public static String drop(String s, int n)
{
    assert n > 0;
    if (n > s.length())
        n = s.length();
    return s.substring(n, s.length());
}
```

Uma cadeia formada pelos caracteres da cadeia dada exceto os **n** primeiros.

```
public static String takeWhile(String s, Predicate<Character> p)
{
    return take(s, countWhile(s, p));
}
```

Uma cadeia formada pelos caracteres da cadeia dada até ao primeiro que não satisfaz o predicado.

```
public static String dropWhile(String s, Predicate<Character> p)
{
    return drop(s, countWhile(s, p));
}
```

Uma cadeia formada pelos caracteres da cadeia dada a partir do primeiro que não satisfaz o predicado.

Toquenizando

- Veja com atenção:

```
public static ArrayBag<String> tokens(String s)
{
    ArrayBag<String> result = new ArrayBag<>();
    while (!s.isEmpty())
    {
        char c = s.charAt(0);
        if (c == SPACE)
            s = StringUtils.drop(s, 1);
        else if (c == LPAREN || c == RPAREN)
        {
            result.add(StringUtils.take(s, 1));
            s = StringUtils.drop(s, 1);
        }
        else if (Character.isLetterOrDigit(c))
        {
            result.add(StringUtils.takeWhile(s, x -> Character.isLetterOrDigit(x) || x == PERIOD));
            s = StringUtils.drop(s, result.at(-1).length());
        }
        else
        {
            result.add(StringUtils.take(s, 1));
            s = StringUtils.drop(s, 1);
        }
    }
    return result;
}
```

```
private static final char SPACE = ' ';
private static final char PERIOD = '.';
private static final char RPAREN = ')';
private static final char LPAREN = '(';
```

Nova função de teste

- Lemos, uma linha, toquenizamos, etc.:

```
public static void testEvaluationTokenized()
{
    DijkstraTwoStack dts = new DijkstraTwoStack();
    while (StdIn.hasNextLine())
    {
        String line = StdIn.readLine();
        ArrayBag<String> t = tokens(line);
        t.forEach(x -> StdOut.print(" "+x));
        StdOut.println();
        double z = dts.evaluationSimple(t.toArray());
        StdOut.println(z);
    }
}
```

O método **forEach** é um método “default” da classe interface, análogo ao método **visit** da classe **ArrayBag**.

- Atenção ao método **toArray**, da classe **ArrayBag**:

```
public T[] array()
{
    @SuppressWarnings("unchecked")
    T[] result = (T[])java.lang.reflect.Array.newInstance(items[0].getClass(), size);
    for (int i = 0; i < size; i++)
        result[i] = items[i];
    return result;
}
```

Expressões regulares

- Modernamente, a análise de cadeias para efeitos de tokenização faz-se com expressões regulares.
- Uma expressão regular é uma cadeia de caracteres que representa de forma compacta um conjunto de cadeias de caracteres.
- A função **find** da classe de biblioteca **Matcher**, procura a “próxima” subcadeia que pertence ao conjunto de cadeias especificado pela expressão regular, se houver uma tal cadeia.
- Essa cadeia é depois obtida pela função **group**.
- A expressão regular é compilada por meio da função **compile** da classe de biblioteca **Pattern**.

Expressões regulares, exemplos

- “**a**”: representa o conjunto cujo único elemento é a cadeia “a”.
- “**abc**”: representa o conjunto cujo único elemento é a cadeia “abc”.
- “**a⁺**”: representa o conjunto das cadeias não vazias, formadas exclusivamente pela letra ‘a’.
- “**(abc)⁺**”: representa o conjunto das cadeias não vazias, formadas exclusivamente por uma ou mais repetições da cadeia “abc”.
- “**abc⁺**”: representa o conjunto das cadeias cujo primeiro caractere é ‘a’, cujo segundo caractere é ‘b’, cujos restantes caracteres são “c”, havendo pelo menos um ‘c’.
- “**(a|e|i|o|u)**”: representa o conjunto formado pelas cadeias “a”, “e”, “i”, “o” e “u”.
- “**(a|e|i|o|u)⁺**”: representa o conjunto das cadeias não vazias formadas exclusivamente pelas letras ‘a’, ‘e’, ‘i’, ‘o’, e ‘u’.
- “**[a-zA-Z]**”: representa o conjunto das cadeias formadas por uma única letra, minúscula ou maiúscula.
- “**[a-zA-Z]⁺**”: representa o conjunto dos nomes, isto é, das cadeias não vazias formadas exclusivamente por letras, minúsculas ou maiúsculas.

Expressões regulares, mais exemplos

- “[0-9]”: representa o conjunto das cadeias formadas por um único algarismo.
- “[0-9]+”: representa o conjunto dos numerais decimais, isto é, das cadeias não vazias formadas exclusivamente por algarismos decimais.
- “[a-zA-Z][a-zA-Z_0-9]*”: o conjunto dos identificadores, isto é, cadeias que começam por um letra seguida por zero ou mais letras, algarismos ou caractere de sublinhado.
- “[a-zA-Z]+|[0-9]+”: a reunião do nomes e dos numerais decimais.
- “\s”: o conjunto das cadeias formadas por um único caractere que um espaço em branco (em inglês, whitespace), por exemplo, espaço, tab, newline.
- “\S”: o conjunto das cadeias formadas por um único caractere que não é um espaço em branco (em inglês, whitespace).
- “\d”: o mesmo que “[0..9]”.
- “\w”: o mesmo que “[a-zA-Z_0-9]”.
- “\d+”: o mesmo que “[0..9]+”.
- “\w+”: o mesmo que “[a-zA-Z_0-9]+”.
- “\d+\.\d+”: os numerais com parte inteira e parte decimal, ambas não vazias, separadas por um ponto.

Para mais informação sobre expressões regulares em Java, veja
<http://docs.oracle.com/javase/tutorial/essential/regex/index.html>.

Aprendendo as expressões regulares

```
public static void learnRegularExpressions()
{
    StdOut.print("Enter your regular expression: ");
    String regex = StdIn.readLine();
    Pattern pattern = Pattern.compile(regex);
    StdOut.print("Enter a string to search: ");
    while (StdIn.hasNextLine())
    {
        String line = StdIn.readLine();
        Matcher matcher = pattern.matcher(line);
        int n = 0; // number of matches
        while (matcher.find())
        {
            String group = matcher.group();
            int start = matcher.start();
            int end = matcher.end();
            int size = end - start;
            StdOut.printf("Group: \"%s\". Start: %d. Length: %d\n", group, start,
                         size);
            n++;
        }
        StdOut.printf("Number of matches: %d\n", n);
        StdOut.print("Enter another string to search: ");
    }
}
```

Adaptado de http://docs.oracle.com/javase/tutorial/essential/regex/test_harness.html.

Esta função pertence à classe **RegularExpressions**. Aceita uma expressão regular, depois uma sequência de cadeias (até ao fim dos dados), “aplicando” a expressão regular a cada uma das cadeias.

Exemplos

```
$ java -cp ../../algs4.jar:. RegularExpressions
Enter your regular expression: (a|e|i|o|u)+
Enter a string to search: auuejjiiiappahhaip
Group: "auue". Start: 0. Length: 4
Group: "iiia". Start: 6. Length: 4
Group: "a". Start: 13. Length: 1
Group: "ai". Start: 17. Length: 2
Number of matches: 4
```

```
$ java -cp ../../algs4.jar:. RegularExpressions
Enter your regular expression: [0-9]+
Enter a string to search: hhhh76lshf3yyy66666666a
Group: "76". Start: 4. Length: 2
Group: "3". Start: 10. Length: 1
Group: "66666666". Start: 14. Length: 8
Number of matches: 3
```

```
$ java -cp ../../algs4.jar:. RegularExpressions
Enter your regular expression: [0-9]+|[a-z]+|\$S
Enter a string to search: aaa+15***34/(57)
Group: "aaa". Start: 0. Length: 3
Group: "+". Start: 3. Length: 1
Group: "15". Start: 4. Length: 2
Group: "*". Start: 6. Length: 1
Group: "*". Start: 7. Length: 1
Group: "*". Start: 8. Length: 1
Group: "34". Start: 9. Length: 2
Group: "/". Start: 11. Length: 1
Group: "(" . Start: 12. Length: 1
Group: "57". Start: 13. Length: 2
Group: ")". Start: 15. Length: 1
Number of matches: 11
```

Grupos

- Podemos obter os grupos que quisermos, parametrizando convenientemente a expressão regular:

```
public final class RegularExpressions
{
    // ...
    public static ArrayBag<String> groups(String s, String regex)
    {
        ArrayBag<String> result = new ArrayBag<>();
        Pattern pattern = Pattern.compile(regex);
        Matcher matcher = pattern.matcher(s);
        while (matcher.find())
        {
            String z = matcher.group();
            result.add(z);
        }
        return result;
    }
    // ...
}
```

Tóquenes

- Para obter os tóquenes de uma expressão aritmética, “basta” indicar a expressão regular apropriada:

```
public class DijkstraTwoStack2016
{
    // ...

    public static final String REGEX_TOKEN =
        "\\\d+\\.\\d+|\\d+[a-zA-Z]\\w+|\\s";

    public static ArrayBag<String> tokensRegEx(String s)
    {
        return RegularExpressions.groups(s, REGEX_TOKEN);
    }

    // ...
}
```

Atenção: ao escrever expressões regulares literais num programa, temos de escapar o carácter '\', o qual tem um significado especial quando inserido numa cadeia literal.

Análise da expressão regular

- A expressão regular que usámos tem quatro alternativas:

```
public static final String REGEX_INTEGER = "\d+";
public static final String REGEX_DECIMAL =
    "\d+\.\d+";
public static final String REGEX_IDENTIFIER =
    "[a-zA-Z]\w+";
public static final String REGEX_SINGLE_NON_WHITESPACE =
    "\s";
public static final String REGEX_TOKEN2 =
    REGEX_DECIMAL + "|" + REGEX_INTEGER
    + "|" + REGEX_IDENTIFIER + "|"
    + REGEX_SINGLE_NON_WHITESPACE;
```

- A expressão dos decimais tem de vir antes da dos inteiros. Caso contrário, a parte inteira seria considerada um grupo, e perdia-se a parte decimal.
- De facto, as alternativas são processadas da esquerda para a direita, até uma acertar.

Toque final

- Podemos agora substituir a tokenização que usa a classe **Utils** pela que é feita à base de expressões regulares:

```
public static void testEvaluationRegEx()
{
    DijkstraTwoStack dts = new DijkstraTwoStack();
    while (StdIn.hasNextLine())
    {
        String line = StdIn.readLine();
        ArrayBag<String> t = tokensRegEx(line);
        double z = dts.evaluationSimple(t.toArray());
        StdOut.println(z);
    }
}
```

Expressões regulares na classe String

- Alguns métodos da classe **String** usam expressões regulares:

```
public boolean matches(String regex)
```

```
public String replaceFirst(String regex,  
                           String replacement)
```

```
public String replaceAll(String regex,  
                           String replacement)
```

```
public String[] split(String regex)
```

Os nomes são autoexplicativos, mas consulte a documentação oficial antes de usar!

Método split e função groups

- Não confunda:
- O método **split** “parte” a cadeia em cada separador encontrado pela expressão regular e o resultado é um array de String, **String[]**.
- É um método oficial do Java.
- A função **groups** calcula as subcadeias da cadeia dada que são encontradas pela expressão regular e o resultado é um **ArrayBag<String>**.
- É uma função da “nossa” classe **Utils**.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 7

Filas

Filas



- Filas simples.
- Implementação das filas simples com listas ligadas.
- Filas com prioridade.
- Implementação das filas com prioridade com heaps.

Filas

- As filas são a contrapartida informática das filas de espera que encontramos na vida corrente.
- A ideia é que o próximo elemento a ser atendido é o que está na fila há mais tempo.
- Dizemos que as filas são coleções FIFO: *first in first out*.
- Nas filas com prioridade, elemento que entrarem mais tarde poderão passar à frente de outros que já estavam na fila, por terem maior **prioridade**.



API da fila simples

- Já a conhecemos:

```
public class Queue<T> implements Iterable<T>
{
    public Queue()
    public void enqueue(T x)
    public boolean isEmpty()
    public int size()
    public T dequeue()
    public T front()
    public Iterator<T> iterator()
}
```

Nota: o método `front` não apareceu na lição 3.

- Comparando com `Stack<T>`, temos `enqueue` em vez de `push`, `dequeue` em vez de `pop`, e `front` em vez de `top`. O iterador emitirá os elementos por ordem de entrada na fila. (Na pilha, era ao contrário).

Nota: o método `top` da classe `Stack<T>` não apareceu ainda nesta coleção de transparentes. No entanto, está na classe fornecida.

Implementação com listas

- É semelhante a `StackLists<T>`, com uma diferença essencial: os elementos são acrescentados no fim da lista e retirados do início da lista.
- Para isso, temos de manter uma referência para o último nó:

```
public class Queue<T> implements Iterable<T>
{
    private class Node
    {
        // ...
    }

    private Node first;
    private Node last;
    private int size;

    // ...
}
```

```
public Queue()
{
    first = null;
    last = null;
    size = 0;
}
```

Nas pilhas os elementos eram acrescentados no início da lista e retirados também do início da lista. Por isso bastava a referência `first`.

Sair da fila

- Para fazer um elemento sair da fila, basta avançar a referência **first** para o nó seguinte. Se não houver nó seguinte, **first** fica a valer **null**, e, neste caso, **last** também deve ficar **null**, pois a fila terá esvaziado.

```
public T dequeue()
{
    T result = first.value;
    first = first.next;
    if (first == null)
        last = null;
    size--;
    return result;
}
```

- A função **front** apenas retorna o primeiro elemento, sem o remover

```
public T front()
{
    return first.value;
}
```

Entrar na fila

- Para fazer um elemento entrar na fila, primeiro criamos um novo nó para esse elemento.
- Depois, das duas uma:
 - A fila estava vazia: então o novo nó passa a ser o primeiro e último nó da fila.
 - A fila não estava vazia: então o novo nó passa a ser o último.

```
public void enqueue(T x)
{
    Node z = new Node(x, null);
    if (first == null)
    {
        last = z;
        first = z;
    }
    else
    {
        last.next = z;
        last = z;
    }
    size++;
}
```

Aqui o novo nó é ligado ao anterior último nó.

isEmpty, size, iterator

- Os métodos **isEmpty**, **size** e **iterator** são iguais aos das pilhas (quando implementadas com listas):

```
public boolean isEmpty()
{
    return first == null;
}

public int size()
{
    return size;
}
```

```
private class QueueIterator implements Iterator<T>
{
    private Node cursor = first;
    // ...
}

public Iterator<T> iterator()
{
    return new QueueIterator();
}
```

Note que nas pilhas (com listas) cada novo elemento é acrescentado à cabeça, enquanto nas filas cada novo elemento é acrescentado à cauda.

Função de teste

```
private static void testQueueInteger()
{
    Queue<Integer> q = new Queue<Integer>();
    while (!StdIn.isEmpty())
    {
        String t = StdIn.readString();
        if (!"-".equals(t))
            q.enqueue(Integer.parseInt(t));
        else if (!q.isEmpty())
        {
            int x = q.dequeue();
            StdOut.println(x);
        }
    }
    StdOut.println("size of queue = " + q.size());
    StdOut.println("is queue empty? " + q.isEmpty());
    if (!q.isEmpty())
        StdOut.println("front of queue = " + q.front());
    StdOut.print("items:");
    for (int x : q)
        StdOut.print(" " + x);
    StdOut.println();
}
```

```
$ java ... Queue A
Queue of Integers
7 9 3 12 71 5
-
7
-
9
4
20
-
3
-
12
size of queue = 4
is queue empty? false
front of queue = 71
items: 71 5 4 20
```

Processamento por ordem de chegada

- Quando precisamos de processar elementos por ordem de chegada enquanto eles vão chegando, usamos filas.
- O pior é se alguns dos elementos são mais “importantes” que outros, e têm prioridade no atendimento.
- Nestes casos, o próximo elemento a ser atendido é o mais prioritário de todos os presentes na fila, e não o que chegou há mais tempo.
- Se a fila estiver implementada à base de uma lista ligada ou de um array, teremos de fazer uma busca linear para descobrir o elemento mais prioritário no momento de selecionar o próximo elemento a ser atendido.
- Ou então temos de reorganizar a fila de cada vez que chega um novo elemento, para a manter ordenada por prioridade decrescente.

Filas com prioridade

- Nas filas com prioridade, os elementos estão guardados num array, e organizados de tal maneira que aceder ao elemento com maior prioridade é uma operação instantânea e de tal maneira que, quando se retira um elemento ou se insere um novo elemento, a reorganização do array é muito eficiente.
- Por “muito eficiente”, queremos dizer “logarítmica”.
- De cada vez que removemos ou inserimos temos de reorganizar, para garantir que elemento com maior prioridade (que pode ter mudado) continua a ser acedido instantaneamente.

Classe PriorityQueue<T>

- No construtor indicamos o operador de comparação usado, por meio de um **comparador**:

```
public class PriorityQueue<T>
{
    PriorityQueue(Comparator<T> c) {...}

    public void insert(T x) {...}

    public T remove() {...}

    public T first() {...}

    public boolean isEmpty() {...}

    public int size() {...}
}
```

Remover significa remover o elemento mais prioritário

Funções de teste para filas com prioridade

- São semelhantes às usadas com pilhas e filas, anteriormente:

```
private static void testPriorityQueueInteger(Comparator<Integer> cmp)
{
    PriorityQueue<Integer> pq = new PriorityQueue<Integer>(cmp);
    while (!StdIn.isEmpty())
    {
        String t = StdIn.readString();
        if (!"-".equals(t))
            pq.insert(Integer.parseInt(t));
        else if (!pq.isEmpty())
        {
            int x = pq.first();
            Stdout.println(x);
            pq.remove();
        }
    }
    StdOut.println("size of queue = " + pq.size());
    StdOut.println("is queue empty? " + pq.isEmpty());
}
```

- A função de teste para filas de **String** é análoga.

Função main

```
public static void main(String[] args)
{
    char choice = 'A';
    if (args.length >= 1)
        choice = args[0].charAt(0);
    if (choice == 'A')
    {
        StdOut.println("Priority queue of Integers, max");
        testPriorityQueueInteger((x, y) -> x-y);
    }
    else if (choice == 'B')
    {
        StdOut.println("Priority queue of Integers, min");
        testPriorityQueueInteger((x, y) -> -(x-y));
    }
    else if (choice == 'C')
    {
        StdOut.println("Priority queue of Strings, max");
        testPriorityQueueString(String::compareTo);
    }
    else if (choice == 'D')
    {
        StdOut.println("Priority queue of Strings, min");
        testPriorityQueueString(Collections.reverseOrder(String::compareTo));
    }
    else
        StdOut.printf("Illegal choice: %s\n", args[0]);
}
```

Testando (por antecipação...)

- Com inteiros, max

```
$ java ... PriorityQueue A
Priority queue of Integers, max
12 56 28 43 91 64 16 27
-
91
-
64
89 5
-
89
-
56
-
43
-
28
-
27
-
16
-
12
-
5
-
-
```

- Com inteiros, min

```
$ java ... PriorityQueue B
Priority queue of Integers, min
74 12 98 51 73 17 33 29
-
12
-
17
21 50
-
21
-
29
-
33
-
50
-
51
-
73
-
74
-
98
-
-
```

Implementação das filas com prioridade

- Certamente não queremos implementações ingênuas, que ordenem o array de cada vez que entra um novo elemento ou que busquem o elemento mais prioritário usando um algoritmo linear.
- Essas implementações teriam um custo proibitivo.
- Usaremos **montes** (em inglês *heaps*).
- Os montes são arrays com uma arrumação interna muito bem imaginada!

Montes

- Os montes são arrays com a seguinte propriedade:

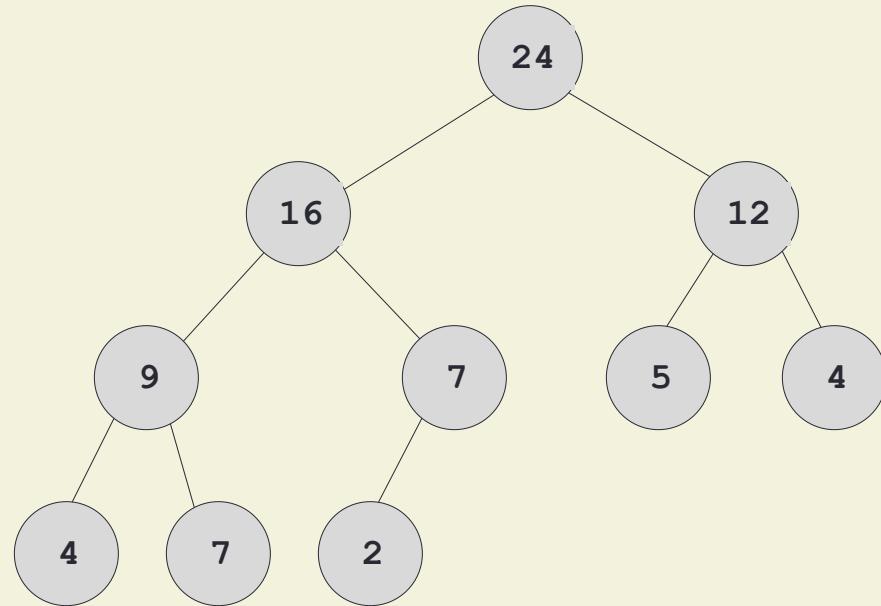
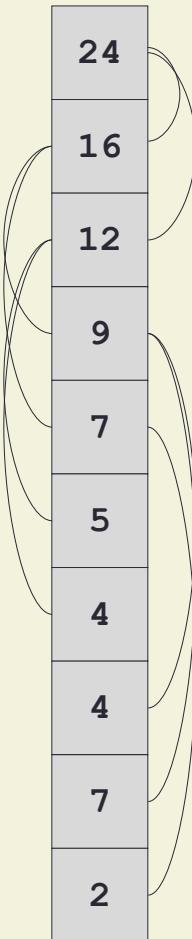
$$\begin{aligned} a[i] &\geq a[2*(i+1)-1] \\ a[i] &\geq a[2*(i+1)] \end{aligned}$$

Note bem: em geral, em vez do \geq teremos a correspondente operação expressa em termos do comparador:
`c.compare(a[i], a[2*(i+1)-1]) >= 0.`

- Estas fórmulas só se aplicam se os índices forem válidos, isto é, se o seu valor for menor que o número de elementos do array.
- Elas implicam que $a[0]$ é o máximo do array.
- As filas com prioridade “esforçar-se-ão” por manter a propriedade dos montes eficientemente, quando entra um novo elemento ou quando sai o primeiro, $a[0]$.

Representação visual dos montes

- No array
- Em árvore



A árvore mostra cada “filho” é menor ou igual ao seu “pai”.

A árvore mostra também que num monte com N elementos, há no máximo 2^X sequências decrescentes, começando na raiz, onde X = $\text{floor}(\log_2 N)$.

Implementação das filas com prioridade

- Membros de dados, construtor, comparação seletores básicos:

```
public class PriorityQueue<T>
{
    private final Comparator<T> cmp;
    private T[] items;
    private int size;

    @SuppressWarnings("unchecked")
    PriorityQueue(Comparator<T> cmp)
    {
        this.cmp = cmp;
        items = (T[]) new Object[1];
        size = 0;
    }

    private boolean less(T x, T y)
    {
        return c.compare(x, y) < 0;
    }

    ...
}
```

Usamos arrays redimensionáveis,
como na classe **Stack<T>**.

```
...
public boolean isEmpty()
{
    return size == 0;
}

public T first()
{
    return items[0];
}

public int size()
{
    return size;
}

...
```

Subindo o monte

- Para inserir um elemento no monte (mantendo a propriedade dos montes), faremos assim:
- Acrescentamos o novo elemento ao array, na primeira posição livre.
- Esse elemento será o último da sequência que vem da raiz até ele, na árvore.
- Provavelmente essa sequência, que estava ordenada antes da inserção, terá deixado de estar.
- Nesse caso, inserimos o último elemento na sequência ordenada, por trocas sucessivas, com o elemento **precedente**, até repor a ordenação, tal como no insertionsort.
- A diferença é que aqui o elemento precedente é o elemento precedente na **sequência** que vem da raiz e não o elemento precedente no array.

Pais e filhos

- Atenção à aritmética dos pais e dos filhos.
- Os filhos do elemento $a[i]$ são $a[2*(i+1)-1]$ e $a[2*(i+1)]$.
- O pai do elemento $a[i]$ é $a[(i-1)/2]$.
- Podemos expressir isto com operadores *bitwise*.
- Nesse caso:
- Os filhos do elemento $a[i]$ são $a[(i+1) \gg 1] - 1$ e $a[i+1 \gg 1]$.
- O pai do elemento $a[i]$ é $a[i-1 \ll 1]$.

Atenção à precedência dos operadores! Os operadores bitwise \gg e \ll têm precedência mais baixa que a dos operadores aritméticos.

Função increase

- A função privada **increase** faz o monte aumentar, inserindo o último elemento do array (que terá acabado de entrar) na sua posição ordenada na sequência decrescente respetiva:

```
private void increase(int k)
{
    while (k > 0 && less(items[(k-1)/2], items[k]))
    {
        exchange(items, k, (k-1)/2);
        k = (k-1)/2;
    }
}
```



```
public void exchange(T[] a, int x, int y)
{
    T m = a[x];
    a[x] = a[y];
    a[y] = m;
}
```

Função insert

- A função pública **insert** primeiro acrescenta o novo elemento ao array, redimensionando-o se necessário, e depois fá-lo subir, com **increase**, até a sua justa posição:

```
public void insert(T x)
{
    if (size == items.length)
        resize(2 * items.length);
    items[size++] = x;
    increase(size - 1);
}
```

Descendo o monte

- Nas nossas filas com prioridade, o único elemento que sai é o primeiro.
- Quando o primeiro sai, o array tem ser reorganizado.
- Faremos assim:
- Primeiro copiamos o último elemento ($a[\text{size}-1]$) para o primeiro ($a[0]$), assim removendo este, mas invalidando a propriedade dos montes.
- Enquanto o elemento que colocámos na raiz for menor que um dos filhos vamos fazê-lo descer no monte, trocando-o com o **maior** dos filhos.
- Desta forma, ele ~~parará~~ na sua posição ordenada numa das sequências decrescentes, e ao mesmo tempo garantimos que todas as outras se mantêm decrescentes.

Note bem: se trocassem com o menor dos filhos (sendo ambos maiores que o pai, bem entendido) estragaria a sequência do outro lado, que deixaria de ser decrescente.

O maior dos filhos

- Trata-se de um caso particular de calcular o índice do maior elemento de um subarray, especificado pelo array de base, o deslocamento e o número de elementos:

```
private int argMax(T[] a, int d, int n)
{
    int result = -1;
    if (n > 0)
    {
        T max = a[d];
        result = 0;
        for (int i = 1; i < n; i++)
            if (less(max, a[d+i]))
            {
                max = a[d+i];
                result = i;
            }
    }
    return result;
}
```

Note bem: a função **argMax** devolve o índice relativo ao subarray ou -1 se o subarray for vazio. Para obter o elemento usando no array, há que adicionar **d** ao resultado da função (se não for -1).

Função decrease

- A função privada **decrease** faz o monte diminuir, começando na posição indicada no argumento, trocando-o com o maior dos seus filhos se tiver filhos e for maior que todos eles ou retorna logo se o elemento na posição indicada não tiver filhos ou, tendo, for maior do que todos eles. Após uma troca, a função continua, a partir da posição do filho trocado:

```
private void decrease(int r)
{
    int left = 2*r+1;
    int z = argMax(items, left, Math.min(2, size - left));
    if (z != -1 && less(items[r], items[left+z]))
    {
        exchange(items, r, left+z);
        decrease(left+z);
    }
}
```

Note que o número de elementos no subarray será 0, 1 ou 2.

Questão técnica: esta função é recursiva, com recursividade terminal. Isto é, a última coisa que a função faz é chamar-se a si própria. Não seria complicado reprogramar iterativamente, mas em geral não vale a pena, pois o compilador “tem obrigação” de fazer isso sozinho, automaticamente.

Função remove

- A função pública **remove** copia o último elemento no monte para a primeira posição e depois fá-lo descer, com decrease, até a sua justa posição. No final, redimensiona o array, se for caso disso:

```
public T remove()
{
    T result = items[0];
    items[0] = items[--size];
    items[size] = null;
    if (size > 0)
        decrease(0);
    if (size > 0 && size == items.length / 4)
        resize(items.length / 2);
    return result;
}
```

Compare com a função
pop, da classe Stack<T>.

Extra: teoria de comparadores

```
public static void testComparatorsExamples(String[] args)
{
    Comparator<Integer> byValue0 = ((x, y) -> x-y);

    Comparator<Integer> byValue = Integer::compare;
    Comparator<Integer> byWeight = Comparator.comparing(PriorityQueue::weight);
    Comparator<Integer> byWeightThenValue = byWeight.thenComparing(byValue);

    Comparator<String> byString = String::compareTo;
    Comparator<String> byLength = Comparator.comparing(String::length);
    Comparator<String> byLengthThenString = byLength.thenComparing(byString);

    Comparator<String> byString1 = (x, y) -> x.compareTo(y);
    Comparator<String> byLength1 = (x, y) -> x.length() - y.length();
    Comparator<String> byLengthThenString1 = byLength1.thenComparing(byString1);

    testPriorityQueueInteger(byValue0);
    testPriorityQueueInteger((x, y) -> x-y);
    testPriorityQueueInteger(byWeight);
    testPriorityQueueInteger(byWeightThenValue);
    testPriorityQueueInteger(Integer::compare);
    testPriorityQueueInteger(Comparator.comparing(PriorityQueue::weight));
    testPriorityQueueInteger(byWeight.thenComparing(byValue));
    testPriorityQueueInteger(Collections.reverseOrder(byWeightThenValue));
    testPriorityQueueInteger(Integer::compare);

    testPriorityQueueString(String::compareTo);
    testPriorityQueueString(byString);
    testPriorityQueueString(Collections.reverseOrder(byString));
    testPriorityQueueString(byLengthThenString1);
    testPriorityQueueString(Collections.reverseOrder(byLengthThenString));
}
```

```
private static int weight(int x)
{
    int result = 0;
    while (x != 0)
    {
        result += x % 10;
        x /= 10;
    }
    return result;
}
```

O peso de um número é a soma dos seus algarismos.

Esta função de demonstração ilustra a utilização de comparadores.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 8

Análise de Algoritmos

Análise de Algoritmos

- Questões centrais: tempo e memória.
- Exemplo: soma tripla.
- Determinação da frequência absoluta das instruções.
- Aproximação til.
- Medição do tempo de cálculo.



Questões centrais

- Quanto tempo levará o meu programa?
- Quanta memória gastará o meu programa?

“Gastar” é apenas uma forma coloquial de falar, já que a memória não se gasta...

Exemplo: problema da soma tripla

- Dada uma lista de números, sem repetições, quantos triplos de números dessa lista somam X, para um dado X?
- Admitindo que os números residem num array, isso pode programar-se assim:

```
public final class ThreeSum
{
    public static int count0(int[] a, int x)
    {
        int result = 0;
        final int n = a.length;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                for (int k = 0; k < n; k++)
                    if (a[i] + a[j] + a[k] == x)
                        result++;
        return result;
    }
    // ...
}
```

Bom, mas não é bem isto, porque para efeitos do problema, queremos contar cada triplo só uma vez e, por exemplo $\langle 4, 6, 8 \rangle$ é o mesmo triplo que $\langle 6, 8, 4 \rangle$ ou $\langle 8, 6, 4 \rangle$, por exemplo.

Soma tripla: primeiro algoritmo

- Só contamos um triplo se for o primeiro. Isto é, se encontrarmos um triplo nas posições de índices i_0, j_0, k_0 , não queremos contá-lo de novo, quando passarmos pelos índices j_0, i_0, k_0 , por exemplo:

```
public static int countBrute(int[] a, int x)
{
    int result = 0;
    final int n = a.length;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                if (a[i] + a[j] + a[k] == x)
                    if (i < j && j < k)
                        result++;
    return result;
}
```

Claro que podemos fazer melhor...

Soma tripla: segundo algoritmo

- Podemos evitar a repetição da contagens, ajustando a variação dos índices diretamente, de maneira que o mesmo triplo não surja mais que uma vez:

```
public static int countBasic(int[] a, int x)
{
    int result = 0;
    final int n = a.length;
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            for (int k = j + 1; k < n; k++)
                if (a[i] + a[j] + a[k] == x)
                    result++;
    return result;
}
```

Esta é melhor que a anterior, mas quão melhor?

Testando cada um dos algoritmos

- Já que as duas funções de teste serão idênticas, a menos da função chamada, **countBrute** ou **countBasic**, vamos usar uma função de teste única, parametrizada pela função de contagem.
- Para fazer isso, precisamos da interface funcional da função de contagem:

```
@FunctionalInterface  
interface ThreeSumCount {  
    public int count(int [] a, int x);  
}
```

Função de teste, geral

- A função de teste geral é parametrizada pela função de contagem, pela soma alvo; o array será lido da consola:

```
public static void testCount(ThreeSumCount op, int x)
{
    int[] a = StdIn.readAllInts();
    int z = op.count(a, x);
    Stdout.println(z);
}
```

Função main

- A função **main** terá dois argumentos na linha de comando: uma letra, que servirá para selecionar o teste pretendido, e um número, que representará a soma alvo:

```
public static void main(String[] args)
{
    assert args.length >= 2;
    char choice = args[0].charAt(0);
    int target = Integer.parseInt(args[1]);
    if (choice == 'A')
        testCount((a, x) -> ThreeSum.countBrute(a, x), target);
    else if (choice == 'B')
        testCount((a, x) -> ThreeSum.countBasic(a, x), target);
    else
        StdOut.println("Illegal option: " + choice);
}
```

```
$ java -cp ../../algs4.jar:. ThreeSum A 20
12 7 5 1 8 3
3
$ java -cp ../../algs4.jar:. ThreeSum B 20
12 7 5 1 8 3
3
```

Comparando o tempo de execução

- A função **countBasic** faz nitidamente menos operações do que a função **countBrute**.
- Logo, chegará ao resultado mais depressa.
- Mas, quão mais depressa?
- Por outras palavras, para o mesmo input, qual é a razão entre os tempos de execução de uma e outra?
- Ou ainda, informalmente: quantas vezes é o **countBasic** mais rápido (a calcular...) do que o **countBrute**?

Quantas vezes mais rápido?

- “Quantas vezes mais rápido” é uma forma de falar, pois o “número de vezes” pode depender da dimensão dos dados.
- Por exemplo, um certo algoritmo pode ser duas vezes mais rápido que do outro, se houver 1000 números, e quatro vezes mais rápido, se houver 2000 números.
- Para responder à pergunta podemos fazer medições e depois, perante os resultados, propor uma explicação para os valores observados.
- Ou então, analisar primeiro os programas, colocar a seguir uma hipótese sobre quantas vezes um é mais rápido que o outro e só então fazer experiências para verificar a hipótese.

Se, mesmo tentando afincadamente, não conseguirmos refutar a hipótese, aceitá-la-emos como válida.

Tempo de execução

- O tempo de execução de um programa é determinado por dois fatores:
 - O tempo de execução de cada instrução.
 - O número de vezes que cada instrução é executada.

Esta observação, aparentemente trivial, foi feita originalmente por Knuth. Para Knuth, o desafio, era, bem entendido, determinar o número de vezes que cada instrução é executada.

O tempo de execução de cada instrução depende do computador, da linguagem e do sistema operativo.

O número de vezes que cada instrução é executada depende do programa e do input.

O ciclo interno

- Na prática, em muitos casos apenas as instruções que são executadas com maior frequência têm peso significativo no tempo de execução total.
- Essas instruções constituem o ciclo interno.
- No caso da função **countBrute**, o ciclo interno é constituído pela seguinte instrução:

```
if (a[i] + a[j] + a[k] == x)
    if (i < j && j < k)
        result++;
```

- É fácil concluir, do contexto, que esta instrução é executada N^3 vezes (representando por N o valor da variável n , que contém o número de elementos do array).

O ciclo interno, em countBasic

- Na função **countBasic**, o ciclo interno contém a instrução.

```
if (a[i] + a[j] + a[k] == x)  
    result++;
```

- Quantas vezes é executada?
- Para cada valor de i e j , a instrução é executada para k valendo $j+1, j+2, \dots, n-1$. Logo, é executada $(n-1)-(j+1)+1 = (n-1)+j$ vezes.
- Para cada valor de i , a instrução é executada para j valendo $i+1, i+2, \dots, n-1$. Logo, é executada, $((n-1)+(i+1))+((n-1)+(i+2)) + \dots + ((n-1)+(n-1))$ vezes.
- Ora i vale sucessivamente $0, 1, \dots, n-1$. Portanto, “basta” somar a expressão anterior n vezes, substituindo i por $0, 1, \dots, n-1$, o que dará uma expressão só envolvendo n .

“É só fazer as contas!”

O ciclo interno, ao contrário

- A seguinte variante é equivalente à anterior, mas faz as variáveis variar descendenteamente e torna a análise mais fácil:

```
public static int countBack(int[] a, int x)
{
    int result = 0;
    final int n = a.length;
    for (int i = n - 1; i >= 0; i--)
        for (int j = i - 1; j >= 0; j--)
            for (int k = j - 1; k >= 0; k--)
                if (a[i] + a[j] + a[k] == x)
                    result++;
    return result;
}
```

O ciclo interno, ao contrário

- O ciclo interno é o mesmo:

```
if (a[i] + a[j] + a[k] == x)  
    result++;
```

- Quantas vezes é executada esta instrução?
- Para cada valor de i e j , a instrução é executada para j vezes.
- Para cada valor de i , a instrução é executada para j valendo $i-1, i-2, \dots, 0$. Logo, é executada, $(i-1)+(i-2)+\dots+0$ vezes.
- Ora esta expressão vale $i^*(i-1)/2$ ou $i^2/2-i/2$ (usando a fórmula da soma da progressão aritmética).
- Ora i vale sucessivamente $0, 1, \dots, n-1$. Portanto, “basta” somar a expressão anterior n vezes, substituindo i por $0, 1, \dots, n-1$.
- A segunda parcela, $i/2$, dá $n^2/4-n/4$, claro (usando a mesma fórmula de há pouco).
- Para a fórmula da soma dos quadrados, podemos consultar o [Wolfram Alpha](#), por exemplo.
- Feitas as contas, no fim dá $n^3/6-n^2/2+n/3$.

Aproximação til

- A expressão $N^3/6-N^2/2+N/3$ é complicada...
- É mais prático usar uma aproximação, notando que para valores grandes de N (os que nos interessam) o valor da primeira parcela é muito maior do que os valores das outras.
- Em vez de dizer rigorosamente, mas complicativamente que a instrução é executada $N^3/6-N^2/2+N/3$ vezes, diremos que é executada $\sim N^3/6$ vezes.

De certa forma, a expressão $\sim f(N)$ representa o conjunto das funções $g(N)$ tais que $g(N)/f(N)$ tende para 1, à medida que N aumenta. Para significar que $g(N)$ é uma dessas funções, escrevemos $g(N) \sim f(N)$.

Comparação countBrute e countBasic

- O número de vezes que a instrução if é executada em **countBrute** é $\sim N^3$.
- Em **countBasic** é $\sim N^3/6$.
- Pela observação de Knuth, o segundo algoritmo é 6 vezes mais rápido do que o primeiro.
- Façamos medições, para verificar essa hipótese.

Medição do tempo

- Queremos comparar várias funções de contagem.
- Sendo assim, usaremos de novo a interface funcional, para parametrizar a função de contagem na função geral de medição de tempo.
- Essa função que mede o tempo tem como argumentos a função de contagem, o array e o valor da soma alvo:

```
public static double timing(ThreeSumCount op, int[] a, int x)
{
    Stopwatch timer = new Stopwatch();
    op.count(a, x); // result of this computation is not used.
    double result = timer.elapsedTime();
    return result;
}
```

A classe **Stopwatch** vem na biblioteca algs4.jar.

Resultados da observação

- Corremos com ficheiros com 1000, 2000, 3000, 4000 e 5000 números. Em cada ficheiro há números aleatórios entre 0 (inclusive) e 10 vezes o número de números (exclusive), sem repetição:

```
$ java -cp ../../algs4.jar:../../bin ThreeSum C 10000 < t_1000.txt  
0.428 0.097  
4.41  
$ java -cp ../../algs4.jar:../../bin ThreeSum C 20000 < t_2000.txt  
3.306 0.670  
4.93  
$ java -cp ../../algs4.jar:../../bin ThreeSum C 30000 < t_3000.txt  
11.008 2.217  
4.97  
$ java -cp ../../algs4.jar:../../bin ThreeSum C 40000 < t_4000.txt  
25.883 5.209  
4.97  
$ java -cp ../../algs4.jar:../../bin ThreeSum C 50000 < t_5000.txt  
50.443 10.332  
4.88
```

- Constatamos que a razão dos tempos está mais próxima do 5 do que de 6. Enfim, não é mau...

Este testes correm na diretoria work, que está ao lado da diretoria bin.

Função de teste

- A função de teste tem como argumento a soma alvo e lê o array da consola.
- Depois invoca a função **timing** para cada um dos algoritmos e mostra os tempos medidos e a razão entre eles:

```
public static void testTiming(int target)
{
    int[] numbers = StdIn.readAllInts();
    double t1 = timing((a, x) -> ThreeSum.countBrute(a, x), numbers, target);
    double t2 = timing((a, x) -> ThreeSum.countBasic(a, x), numbers, target);
    StdOut.printf("%.3f %.3f\n", t1, t2);
    StdOut.printf("%.2f\n", t1 / t2);
}
```

- A função **main** terá mais um caso, para este teste:

```
public static void main(String[] args)
{
    ...
    else if (choice == 'C')
        testTiming(Integer.parseInt(args[1]));
    ...
}
```



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 9

Ordem de crescimento do tempo de execução

Ordem de crescimento do tempo de execução



- Classificação por ordem de crescimento do tempo de execução.
- Geração de arrays aleatórios.
- Ensaios de razão dobrada.

Ordem de crescimento

- Em análise de algoritmos, as aproximações til que nos interessam são geralmente da forma $g(N) \sim a^*f(N)$, onde $f(N) = N^b(\log N)^c$, sendo a , b e c constantes.
- Dizemos que $f(N)$ é a **ordem de crescimento** de $g(N)$.

Nota técnica: quando usamos logaritmos neste contexto, a base é irrelevante, pois uma mudança de base será sempre absorvida pela constante a , por aplicação da fórmula da mudança de base:

$$\log_b x = \log_b a * \log_a x.$$

- Portanto, em **countBrute** e em **countBasic** a ordem de crescimento da frequência da instrução if é N^3 .

Ordens de crescimento habituais

Descrição	Função	
Constante	1	Troca do valor de duas variáveis
Logarítmica	$\log N$	Busca dicotómica
Linear	N	Máximo de um array
Linearítmica	$N \log N$	Quicksort, Mergesort
Quadrática	N^2	Selectionsort, Insertionsort
Cúbica	N^3	
Exponencial	2^N	Soma tripla... Busca exaustiva: por exemplo, encontrar todos os subconjuntos cuja soma é X.

Ordem de crescimento do tempo

- As medições que fizemos (na aula anterior) parecem indicar que o tempo de execução cresce aproximadamente 8 vezes quando o tamanho do array duplica, para ambos os algoritmos.

CountBrute		
N	T	$T(i+1)/T(i)$
1000	0.428	
2000	3.306	7.72
4000	25.883	7.83

CountBasic		
N	T	$T(i+1)/T(i)$
1000	0.097	
2000	0.670	6.91
4000	5.209	7.77

- Isso é um reflexo da ordem de crescimento cúbica da frequência da instrução if.
- Analisemos isso sistematicamente.

Ensaios de razão dobrada

- Primeiro, construímos um gerador de inputs apropriado ao problema.
- Depois experimentamos o programa repetidamente, de cada vez dobrando o tamanho dos dados.
- Medido o tempo numa experiência, calculamos a razão para o tempo da experiência anterior.
- Em muitos casos, observaremos que a razão tende para 2^b , para uma certa constante b.

Array de números aleatórios

- Para o problema da soma tripla precisamos de arrays de números arbitrários, sem duplicados.
- Comecemos por gerar arrays de **size** números aleatórios, em geral, no intervalo [0..**max**[.
- Em cada caso, indicamos os valores de **size** e de **max**:

```
public class RandomArrays {  
    // creates array with n uniformly generated  
    // random numbers in [0..max[  
    public static int[] uniform(int n, int max)  
    {  
        int[] result = new int[n];  
        for (int i = 0; i < size; i++)  
            result[i] = StdRandom.uniform(max);  
        return result;  
    }
```

Note bem: neste caso pode haver duplicados.

Cada chamada da função **StdRandom.uniform()** devolve um número aleatório no intervalo [0..**max**[. A classe **StdRandom** vem na biblioteca algs4.jar.

Números aleatórios sem duplicados

- Criar um array de números aleatórios é mais subtil.
- Observe:

```
// creates array with n uniformly generated
// unique random numbers in [0..max[
public static int[] uniformUnique(int n, int max)
{
    assert n <= max;
    int[] result = new int[n];
    int[] candidates = new int[max];
    for (int i = 0; i < max; i++)
        candidates[i] = i;
    int maxRandom = max;
    for (int i = 0; i < n; i++)
    {
        int r = StdRandom.uniform(maxRandom);
        result[i] = candidates[r];
        candidates[r] = candidates[maxRandom - 1];
        maxRandom--;
    }
    return result;
}
```

Explicação: inicialmente, todos os números em [0..max[são candidatos. Em cada passo, recolhe-se um candidato aleatoriamente, colocando na sua posição o último candidato e decrementando o número de candidatos.

Testando a criação de números aleatórios

- Montemos uma função para testar, observando, a criação de arrays de números aleatórios.
- Primeiro, uma interface funcional:

```
@FunctionalInterface  
interface SupplierRandomArray {  
    public int [] supply(int size, int max);  
}
```

- Agora, uma função de teste geral:

```
public static void testRandomArrays(SupplierRandomArray r, int n, int max)  
{  
    int[] a = r.supply(n, max);  
    show(a);  
}
```

- A função **show** mostra o array:

```
private static void show(int a[])  
{  
    for (int x : a)  
        StdOut.print(" " + x);  
    StdOut.println();  
}
```

Invocação da função de teste

- Eis a parte da função **main** que se encarrega de testar as duas funções de criação de arrays de números aleatórios:

```
public static void main(String[] args)
{
    char choice = 'A';
    int x = 10;
    int y = 20;
    if (args.length > 0)
        choice = args[0].charAt(0);
    if (args.length > 1)
        x = Integer.parseInt(args[1]);
    if (args.length > 2)
        y = Integer.parseInt(args[2]);
    if (choice == 'A')
        testRandomArrays((s, m) -> uniform(s, m), x, y);
    else if (choice == 'B')
        testRandomArrays((s, m) -> uniformUnique(s, m), x, y);
    else
        Stdout.println("Illegal option: " + choice);
```

Experimentando os números aleatórios

- Indicamos na linha de comando qual a função a testar e os argumentos para a função: tamanho do array e valor máximo; se não, usam-se os valores por defeito:

```
$ java -cp ../../algs4.jar:. RandomArrays  
7 0 3 18 8 3 8 12 6 6  
$ java -cp ../../algs4.jar:. RandomArrays A 20 100  
43 75 17 74 96 37 22 30 86 20 51 0 96 91 90 23 30 2 75 45  
$ java -cp ../../algs4.jar:. RandomArrays A 20 10  
3 6 3 6 9 2 0 9 5 6 4 7 6 7 8 9 0 2 7 7  
$ java -cp ../../algs4.jar:. RandomArrays B  
2 0 8 17 9 5 3 18 10 19  
$ java -cp ../../algs4.jar:. RandomArrays B 20 50  
34 42 15 24 17 38 40 16 37 18 19 6 10 1 31 3 22 48 14 0  
$ java -cp ../../algs4.jar:. RandomArrays B 25 25  
8 20 24 1 0 3 12 11 6 13 14 5 10 4 23 18 7 17 15 22 21 9 19 2 16  
$ java -cp ../../algs4.jar:. RandomArrays B 50 1000  
630 681 873 488 717 534 744 281 722 590 448 89 656 620 224 99 191  
875 834 308 12 275 454 379 97 278 878 260 968 144 228 785 311 790  
212 751 504 50 505 736 843 879 940 335 557 276 662 992 766 279
```

Os **azuis**, criados na opção A, podem ter duplicados; os **vermelhos**, criados na opção B, não têm duplicados, garantidamente.

Medição do tempo, com arrays aleatórios

- Queremos medir os tempos, agora com arrays aleatórios, e não com arrays lidos da consola.
- Usamos a seguinte função de teste, que é análoga à que usamos na lição anterior, mas agora usando arrays aleatórios:

```
public static void testTimingRandom2(int target)
{
    int[] numbers = RandomArrays.uniformUnique(target / 10, target);
    double t1 = timing((a, x) -> ThreeSum.countBrute(a, x), numbers, target);
    double t2 = timing((a, x) -> ThreeSum.countBasic(a, x), numbers, target);
    StdOut.printf("%.3f %.3f\n", t1, t2);
    StdOut.printf("%.2f\n", t1 / t2);
}
```

Comparando os dois algoritmos

- Haverá na função de teste um caso para este teste:

```
...
    int target = Integer.parseInt(args[1]);
...
else if (choice == 'D')
    testTimingRandom2(target);
else ...
```

```
$ java -cp ../../algs4.jar:. ThreeSum D 10000
0.435 0.098
4.44
$ java -cp ../../algs4.jar:. ThreeSum D 20000
3.330 0.680
4.90
$ java -cp ../../algs4.jar:. ThreeSum D 30000
11.141 2.194
5.08
$ java -cp ../../algs4.jar:. ThreeSum D 40000
25.851 5.233
4.94
$ java -cp ../../algs4.jar:. ThreeSum D 50000
50.414 10.046
5.02
```

Os resultados são muito parecidos com os da lição anterior, pois os argumentos foram escolhidos de forma a que os arrays fossem análogos aos usados então.

Razão dobrada

- Para os testes de razão dobrada, queremos usar um algoritmo e sucessivos arrays, cada vez com o dobro do tamanho.
- No entanto, para maior precisão, se os testes demorarem menos que um dado limiar, vamos repeti-los com arrays do mesmo tamanho, até o tempo acumulado ultrapassar esse limiar, e no fim tiramos a média.
- O limiar é dado por uma variável estática.

```
private static double tMin = 0.5;
```

Função de razão dobrada

- Parametrizamos o algoritmo e, antecipando futuras necessidades, a função criação do array, e ainda, o tamanho, o máximo, a soma-alvo e o número de testes:

```
public static double[] doubleRatio(
    ThreeSumCount op, SupplierRandomArray r,
    int size, int max, int x, int times)
{
    double[] result = new double[times];
    for (int i = 0; i < times; i++, size *= 2, max *= 2, x *= 2)
    {
        double t = 0.0;
        int n = 0;
        while (t < tMin)
        {
            t += timing(op, r.supply(size, max), x);
            n++;
        }
        result[i] = t / n;
    }
    return result;
}
```

O resultado é um array de double que contém os tempos medidos os testes com cada tamanho. Para tempos inferiores ao limiar, o valor será uma média.

Função printTable

- Mostra os números no array dos tempos, linha a linha, precedidos pelo tamanho do array usado no ensaio, e seguidos pela razão em relação ao ensaio anterior (se não for zero), com um formato minimalista:

```
private static void printTable(int size, double[] t)
{
    double t0 = 0.0;
    for (int i = 0; i < t.length; i++, size *= 2)
    {
        stdout.printf("%d %.4f", size, t[i]);
        if (t0 > 0) // for i == 0 and also in case t[i] == 0
            stdout.printf(" %.2f", t[i] / t0);
        stdout.println();
        t0 = t[i];
    }
}
```

Testando a razão dobrada

- Eis a função de teste e a parte da função **main** que interessa:

```
public static void testDoubleRatio(
    ThreeSumCount op, SupplierRandomArray r,
    int size, int max, int x, int times)
{
    double z[] = doubleRatio(op, r, size, max, x, times);
    printTable(size, z);
}
```

```
public static void main(String[] args)
{
    ...
    else if (choice == 'G')
        testDoubleRatio((a, x) -> ThreeSum.countBrute(a, x),
                        (a, x) -> RandomArrays.uniformUnique(a, x),
                        100, 1000, 1500, times);
    else if (choice == 'H')
        testDoubleRatio((a, x) -> ThreeSum.countBasic(a, x),
                        (a, x) -> RandomArrays.uniformUnique(a, x),
                        100, 1000, 1500, times);    else
        StdOut.println("Illegal option: " + choice);
    else
        ...
}
```

Resultados da razão dobrada

```
$ java -cp ../../algs4.jar:. ThreeSum G 7
```

```
100 0.0004
```

```
200 0.0038 8.43
```

```
400 0.0282 7.46
```

```
800 0.2177 7.73
```

```
1600 1.7100 7.86
```

```
3200 14.1000 8.25
```

```
6400 107.8180 7.65
```

```
$ java -cp ../../algs4.jar:. ThreeSum H 7
```

```
100 0.0001
```

```
200 0.0007 7.15
```

```
400 0.0050 7.60
```

```
800 0.0381 7.58
```

```
1600 0.2935 7.69
```

```
3200 2.3390 7.97
```

```
6400 18.0590 7.72
```

```
$
```

Este é o **countBrute**.

Este é o **countBasic**. É cúbico, como o outro, mas mais rápido, como sabemos.

Um algoritmo mais rápido

- Vimos que o **countBasic** é 6 vezes mais rápido do que o **countBrute**; no entanto, ambos são algoritmos cúbicos, isto é, em ambos o tempo de execução é proporcional ao cubo do tamanho do array.
- Será que conseguimos encontrar um algoritmo mais rápido, isto é, com ordem de crescimento subcúbica?
- Ora bem: dois valores do array $a[i]$ e $a[j]$, com $i < j$, farão parte de um triplo que soma x , se existir no array, à direita de j , um elemento de valor $x-(a[i]+a[j])$.
- Se o array estiver ordenado, podemos procurar esse valor usando **busca dicotómica**.
- Na verdade, de certa forma, os anteriores algoritmos procuram usando busca linear.
- Para decidir se um dado valor existe num array ordenado com N elementos bastam no máximo $\text{floor}(\log_2 N) + 1$ iterações, usando busca dicotómica.
- Portanto, descontando a ordenação, o algoritmo deve ser $\sim aN^2 \log_2 N$, para um certo a .
- A ordem de crescimento será $N^2 \log_2 N$.

Custo da ordenação

- Os algoritmos de ordenação elementares—insertionsort, selectionsort, bubblesort—são quadráticos.
- Portanto, ordenando com um desses, o tempo de execução do novo algoritmo seria $\sim(aN^2 \log_2 N + bN^2)$.
- Ora isto é o mesmo que $\sim aN^2 \log_2 N$.
- Repare, quando N for muito grande, a segunda parcela é desprezável face à primeira.
- Ainda por cima, os algoritmos de ordenação usados pela biblioteca do Java—uma variante do mergesort, para objetos e o quicksort para tipos primitivos—são $N \log N$, que é melhor que N^2 .
- Portanto, esta estratégia promete um algoritmo mais rápido.
- Experimentemos.

Busca dicotómica

- A busca dicotómica é o algoritmo que procura uma ocorrência de um valor dado num **array ordenado**, dividindo ao meio o intervalo de busca em cada passo, por comparação do valor procurado com o valor do elemento central no intervalo de busca.
- Se o valor procurado for menor que o valor do elemento central, a busca prossegue, agora no meio-intervalo da esquerda; se for maior, a busca prossegue no meio-intervalo da direita; se for igual, a busca termina, com êxito.
- Se o intervalo de busca ficar vazio, a busca termina, sem êxito.
- A busca dicotómica é um algoritmo logarítmico: o tempo de execução é proporcional ao logaritmo do tamanho do array.

Renque

- O renque de um valor relativamente a um array é o número de elementos do array cujo valor é menor que esse valor.
- Se o array estiver ordenado, o renque pode ser calculado dicotomicamente.
- Sobre este assunto, reveja os transparentes de Programação Imperativa, em

http://w3.ualg.pt/~pjguerreiro/sites/24_pi_1516/lessons/pi_1516_tudo.pdf.

Renque em Java

- Esta é a transcrição para Java da função usada em Programação Imperativa, colocada numa nova classe, **BinarySearch**:

```
public class BinarySearch
{
    // ...
}
```

```
public static int rank(int[] a, int x)
// a is sorted
{
    int result = 0;
    int d = 0;
    int n = a.length;
    while (n > 0)
    {
        int m = n / 2;
        if (x <= a[d+m])
            n = m;
        else
        {
            result += m+1;
            d += m+1;
            n -= m+1;
        }
    }
    return result;
}
```

Função da busca dicotómica

- A busca dicotómica programa-se nas calmas recorrendo ao renque.
- Em caso de busca falhada, a função retorna `-1`, como de costume; em caso de busca bem sucedida, a função retorna o índice do **primeiro** elemento do array cujo valor é igual ao valor procurado.

```
public static int bfind(int[] a, int x)
{
    int r = rank(a, x);
    return r < a.length && a[r] == x ? r : -1;
}
```

Esta função pertence à classe `BinarySearch`.

Note bem: esta função devolve o índice do primeiro elemento com o valor dado, mesmo que haja vários elementos iguais a esse. Outras variantes da busca dicotómica devolvem o índice de um dos elementos com o valor dado, não necessariamente o primeiro.

Curiosidade: busca dicotómica tradicional

- Esta é a versão tradicional, que não se baseia no renque.

```
public static int bsearch(int x, int[] a)
{
    int result = -1;
    int i = 0;
    int j = a.length - 1;
    while (result == -1 && i <= j)
    {
        int m = i + (j - i) / 2;
        if (x < a[m])
            j = m - 1;
        else if (x > a[m])
            i = m + 1;
        else
            result = m;
    }
    return result;
}
```

Porquê escrever $i+(j-i)/2$ e não, mais simplesmente, $(i+j)/2$?

Resposta em

<http://googleresearch.blogspot.pt/2006/06/extra-extra-read-all-about-it-nearly.html>.

Se houver mais do que um elemento com o valor procurado, esta acerta num deles, não necessariamente no primeiro.

Função countFaster

- Eis o novo algoritmo para o problema da soma tripla:

```
public static int countFaster(int[] a, int x)
{
    // a is sorted
    int result = 0;
    final int n = a.length;
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
    {
        int r = BinarySearch.bfind(a, x - (a[i] + a[j]));
        if (r > j)
            result++;
    }
    return result;
}
```

Com dois ciclos for imbricados e uma busca dicotómica lá dentro, o tempo de crescimento é de ordem $N^2\log N$, certamente.

Razão dobrada para o novo algoritmo

- Comprovemos experimentalmente que é $N^2 \log N$.
- Mas atenção: precisamos agora de um array ordenado:

```
// creates sorted array with n uniformly generated
// unique random numbers in [0..max[
public static int[] uniformUniqueSorted(int n, int max)
{
    assert n <= max;
    int[] result = uniformUnique(n, max);
    Arrays.sort(result);
    return result;
}
```

```
...
else if (choice == 'I')
    testDoubleRatio((a, x) -> ThreeSum.countFaster(a, x),
                    (a, x) -> RandomArrays.uniformUnique(a, x),
                    100, 1000, 1500, times);
...
...
```

Resultados do ensaio

- Eis os resultados do ensaio de razão dobrada com a função **countFaster**:

```
$ java -cp ../../algs4.jar:. ThreSum I 7  
100 0.0001  
200 0.0007 4.68  
400 0.0031 4.63  
800 0.0138 4.40  
1600 0.0598 4.32  
3200 0.2520 4.22  
6400 1.0510 4.17  
$
```

A razão é ligeiramente superior a 4 (e muito inferior a 8). Isto confirma a constatação de que esta função é $\sim N^2 \log N$.

- Conclusão: a utilização esclarecida de algoritmos que fazem parte do arsenal de qualquer programador evoluído permite conseguir impressionantes ganhos em eficiência.

Soma dupla

- Um problema análogo ao do soma tripla é o da soma dupla: contar os pares de números num array sem duplicados que somam um dado valor.
- Imaginamos facilmente a solução bruta, com dois ciclos for cada um varrendo o array todo, e a solução básica, em que o ciclo interno só percorre o “resto” do array.
- Ambas são quadráticas, mas quão melhor será a básica em relação à bruta?
- Por outro lado, se o array estiver ordenado, podemos substituir o ciclo interno por uma busca dicotómica, melhorando a complexidade para $N\log N$.
- Será que conseguiremos baixar ainda mais, para aquém de $N\log N$?

Algoritmo linear para a soma dupla

- Se o array estiver ordenado (e não tiver duplicados) e a soma do primeiro elemento (que é o menor) e do último (que é o maior) for maior que a soma alvo, concluímos que o último não fará parte de nenhum par cuja soma seja a soma alvo; se for menor, concluímos que o primeiro não fará parte de nenhum desses pares; se for igual, então os dois números formam um par cuja soma é a soma alvo, contabilizamos esse par e concluímos que esses dois números não farão parte de mais nenhum par.
- Vamos “encolhendo” o array e repetindo os cálculos, até o array se esgotar.

Classe TwoSum

- Eis a classe para a soma dupla, só com a função **countSuper**, que implementa o algoritmo descrito.

```
public class TwoSum
{
    // ...
}
```

```
public static int countSuper(int[] a, int x)
{
    int result = 0;
    int i = 0;
    int j = a.length - 1;
    while (i < j)
    {
        int s = a[i] + a[j];
        if (s < x)
            i++;
        else if (s > x)
            j--;
        else
        {
            i++;
            j--;
            result++;
        }
    }
    return result;
}
```

Claramente só há um ciclo e cada elemento do array só é visitado uma vez. Isto implica que o algoritmo é linear.

Ensaios de razão dobrada para a soma dupla

- Observemos o comportamento linear da soma dupla:

```
public static void main(String[] args)
{
    if (args.length < 2)
        return;
    char choice = args[0].charAt(0);
    int times = Integer.parseInt(args[1]);
    ...
    else if (choice == 'B')
        ThreeSum.testDoubleRatio((a, x) -> countSuper(a, x),
                                  (a, x) -> RandomArrays.uniformUniqueSorted(a, x),
                                  1000000, 10000000, 15000000, times);
    ...
}
```

```
$ java -cp ../../algs4.jar:. TwoSum B 6
1000000 0.0040
2000000 0.0081 2.00
4000000 0.0164 2.02
8000000 0.0332 2.03
16000000 0.0650 1.96
32000000 0.1335 2.05
$
```

Note que vamos buscar a função de teste à classe ThreeSum.

A razão é 2, demonstrando comportamento linear.

Soma tripla quadrática

- Na soma tripla, queremos contar os triplos $(a[i], a[j], a[k])$, como $i < j < k$, tais que $a[i] + a[j] + a[k] = x$, para um x dado.
- Seja $n = a.length$.
- Então contemos os pares $(a[i], a[j])$ para $i < j < n-1$, que somam $x - a[n-1]$, usando o algoritmo linear.
- Para cada um dos pares contados, o triplo $(a[i], a[j], a[n-1])$ soma x , e não há mais nenhum triplo que some x cujo terceiro elemento seja $a[n-1]$.
- Repetimos, para $n = a.length-1, a.length-2, \dots, 3$, acumulando os pares contados em cada passo.

Retocando a classe da soma dupla

- Observamos que precisamos de calcular a soma dupla, não para um array todo, mas apenas para a parte inicial de um array.
- Por isso, reorganizamos a classe **TwoSum** assim:

```
public class TwoSum
{
    public static int countSuper(int[] a, int x)
    {
        return countSuper(a, a.length, x);
    }

    public static int countSuper(int[] a, int n, int x)
    {
        int result = 0;
        int i = 0;
        int j = n - 1;
        while (i < j)
        {
            ...
        }
        return result;
    }
}
```

Algoritmo quadrático para a soma tripla

- Até fica simples:

```
public class ThreeSum
{
    // ...

    public static int countSuper(int[] a, int x)
    // a is sorted
    {
        int result = 0;
        for (int n = a.length; n >= 3; n--)
            result += TwoSum.countSuper(a, n-1, x - a[n-1]);
        return result;
    }
    // ...
}
```

Um algoritmo linear em N repetido
N vezes dá um algoritmo quadrático.

Razão dobrada para o algoritmo quadrático

- Observe:

```
...
else if (choice == 'J')
    testDoubleRatio((a, x) -> ThreeSum.countSuper(a, x),
                    (a, x) -> RandomArrays.uniformUnique(a, x),
                    100, 1000, 1500, times);
...
...
```

- Obtemos os resultados já apresentados na lição anterior:

```
$ java -cp ../../algs4.jar:. ThreeSum J 7
100 0.0000
200 0.0001 3.86
400 0.0003 4.01
800 0.0014 3.93
1600 0.0055 4.04
3200 0.0219 3.96
6400 0.0885 4.04
$
```

A razão ser 4 é um sinal claro de que se trata de um algoritmo quadrático.

Moral da história

- Não basta ter um programa que calcula o que queremos.
- A questão não é otimizar o código; é sim procurar algoritmos melhores.
- Os algoritmos brutos ou básicos são interessantes como padrão de comparação e permitem validar os algoritmos melhores.
- Temos de ser capazes de analisar os tempos de execução dos nossos algoritmos, matematicamente ou experimentalmente.

Extra: teste de todos

- Eis um teste suplementar, só para confirmar que, depois destas experiências todas, os algoritmos ainda estão a funcionar...

```
public static void testAll(int target)
{
    int[] numbers = RandomArrays.uniformUnique(target / 10, target);
    int n1 = countBrute(numbers, target);
    int n2 = countBasic(numbers, target);
    Arrays.sort(numbers);
    int n3 = countFaster(numbers, target);
    int n4 = countSuper(numbers, target);

    StdOut.printf("%d %d %d %d\n", n1, n2, n3, n4);
    assert n1 == n2;
    assert n2 == n3;
    assert n3 == n4;
}
```

```
$ java -cp ../../algs4.jar:. ThreeSum L 1000
83 83 83 83
$ java -cp ../../algs4.jar:. ThreeSum L 10000
8429 8429 8429 8429
$ java -cp ../../algs4.jar:. ThreeSum L 10000
7811 7811 7811 7811
$ java -cp ../../algs4.jar:. ThreeSum L 20000
33743 33743 33743 33743
```



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 10
Union-Find

Union-Find

- Problema da conectividade dinâmica.
- Union-Find, em geral.
- Quick-Find.
- Quick-Union.
- Quick-Union-Weighted.



Conectividade

- Temos um conjunto de nós: x_1, x_2, \dots, x_n .
- Alguns desses nós estão **ligados**, dois a dois.
- Um nó x_i estará **conectado** a um nó x_j se existir uma sequência de nós $\langle y_1, y_2, \dots, y_m \rangle$ tal que x_i está ligado a y_1 , y_1 está ligado a y_2 , etc., e y_m está ligado a x_j .
- Queremos um programa que, perante um par de nós, x_i e x_j , nos diga se esses nós estão conectados.
- Mas não só...

Conectividade dinâmica

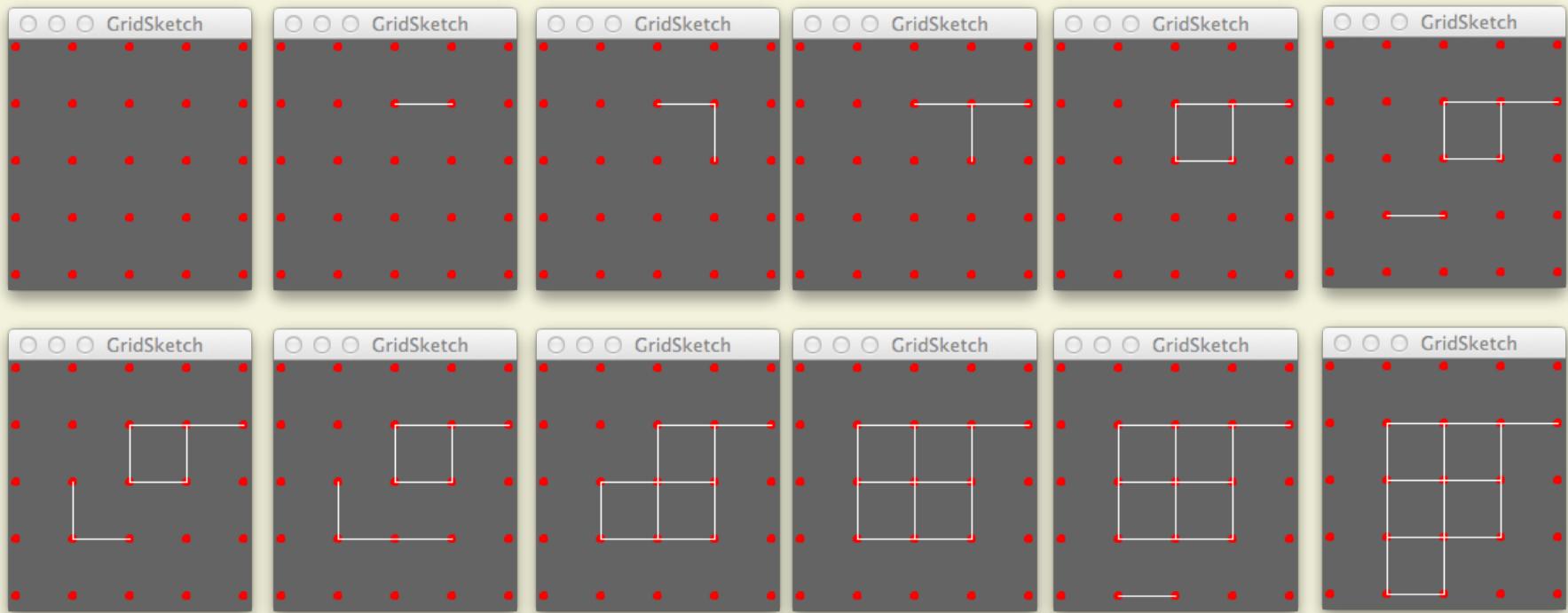
- A relação de “estar conectado” é uma relação de equivalência:
 - Reflexiva
 - Simétrica
 - Transitiva
- Portanto, o nosso programa, perante um par de nós, x_i e x_j , dir-nos-á se os dois nós pertencem à mesma classe de equivalência.
- Mas queremos também que o programa recalcule as classes de equivalência quando estabelecemos uma nova ligação entre dois nós que não estavam conectados.

Recorde: uma relação de equivalência partitiona o universo em classes de equivalência disjuntas.

Na nossa gíria, uma classe de equivalência de nós é uma “componente”.

Exemplo: reticulado dinâmico

- Visualizar um reticulado em que cada vez que ligamos dois nós vizinhos, as conexões aos vizinhos que pertencem à mesma componente são recalculadas e mostradas.



Note bem: o programa desenha uma conexão entre dois nós vizinhos se eles estiverem na mesma componente. Outra coisa seria desenhar uma conexão entre dois nós da mesma componente se eles fossem vizinhos, ainda que o efeito visual fosse o mesmo.

Classe abstrata UnionFind

- Precisamos de uma operação para ligar dois nós, **union**, e outra para ver se dois nós pertencem à mesma componente, **connected**.
- Já agora, também uma operação para dar o número de componentes, **count**, e outra para calcular o “identificador” da componente a que um dado nó pertence, **find**.
- Reunimos isto numa classe abstrata:

```
public abstract class UnionFind
{
    public abstract void union(int x, int y);
    public abstract boolean connected(int x, int y);
    public abstract int count();
    public abstract int find(int x);
}
```

O nome consagrado para a classe (e para o algoritmo em análise) vem do nome das operações típicas: **union** e **find**.

Programando connected

- Podemos oferecer já uma implementação da operação **connected**, uma vez que, por hipótese, dois nós estarão conectados se pertencerem à mesma componente:

```
public abstract class UnionFind
{
    public abstract void union(int x, int y);
    public abstract int find(int x);
    public abstract int count();

    public boolean connected(int x, int y)
    {
        return find(x) == find(y);
    }
}
```

Algumas classes derivadas poderão querer redefinir esta função.

Classes derivadas

- As classes derivadas terão de definir a estrutura de dados mediante a qual representamos as conexões.
- E, em função disso, terão de definir as funções **union**, **find** e **count**; e deverão redefinir **connected**, se valer a pena.
- Em todos os casos que nos interessam, os nós são representados pelos números inteiros do intervalo $[0..N[$, sendo N o número de nós.
- As conexões serão representadas implicitamente por meio de um array de nós, indexado pelos nós.
- A ideia é registar apenas uma de entre todas as conexões de um nó, sem perder as outras.

QuickFind

- Na estratégia **QuickFind**, todos os nós de uma componente registam a sua conexão ao “líder” da componente.
- Se um nó estiver isolado, ele é o seu próprio líder.
- Quando ligamos um nó **x** a um nó **y**, todos os nós da componente do nó **x** passam a ter como líder o líder da componente a que pertence o nó **y**.

Podia ser ao contrário, claro.
- Claro que a ligação é inócuia se **x** e **y** já estiverem conectados.

Classe QuickFind

- Os membros serão o array dos líderes e o número de componentes:

```
public class QuickFind extends UnionFind
{
    private int[] id;
    private int count;
    ...
}
```

- No construtor, indicamos o número de nós e assinalamos que inicialmente cada nó é líder de si próprio:

```
public QuickFind(int n)
{
    count = n;
    id = new int[n];
    for (int i = 0; i < n; i++)
        id[i] = i;
}
```

Note bem: inicialmente, todos os nós estão isolados; por isso, inicialmente o número de componentes é igual ao número de nós.

Funções count, find e connected

- São as três muito simples e eficientes:

```
public int count()  
{  
    return count;  
}
```

```
public int find(int x)  
{  
    return id[x];  
}
```

```
public boolean connected(int x, int y)  
{  
    return id[x] == id[y];  
}
```

As três operações são de tempo constante.

Redefinimos a função connected por uma questão de zelo algorítmico, visto que neste caso tão simples não se justifica o *overhead* da dupla chamada da função find.

Função union

- Esta é a única que exibe alguma subtileza.
- Cada nó cujo líder é o líder de x, passa a ter como líder o líder de y:

```
public void union(int x, int y)
{
    int ix = id[x];
    int iy = id[y];
    if (ix != iy)
    {
        for (int i = 0; i < id.length; i++)
            if (id[i] == ix)
                id[i] = iy;
        count--;
    }
}
```

O líder de x.

O líder de y.

Se tiverem o mesmo líder, não há nada a fazer.

Esta é uma operação linear, pois o ciclo for percorre todo o array, inexoravelmente.

Função de teste

- Eis uma função de teste que aceita pares de nós sucessivamente e mostra o estado do sistema a cada passo:

```
public static void testQuickFind(String[] args)
{
    int size = 10;
    if (args.length != 0)
        size = Integer.parseInt(args[0]);
    QuickFind qf = new QuickFind(size);
    qf.show(-1, -1); // just to show the initial state
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        int y = StdIn.readInt();
        qf.union(x, y);
        qf.show(x, y);
    }
}

public static void main(String[] args)
{
    testQuickFind(args);
}
```

```
private void show(int x, int y)
{
    StdOut.printf("%3d%3d  ", x, y);
    for (int z : id)
        StdOut.printf("%3d", z);
    StdOut.println(" / " + count);
}
```

Experimentando

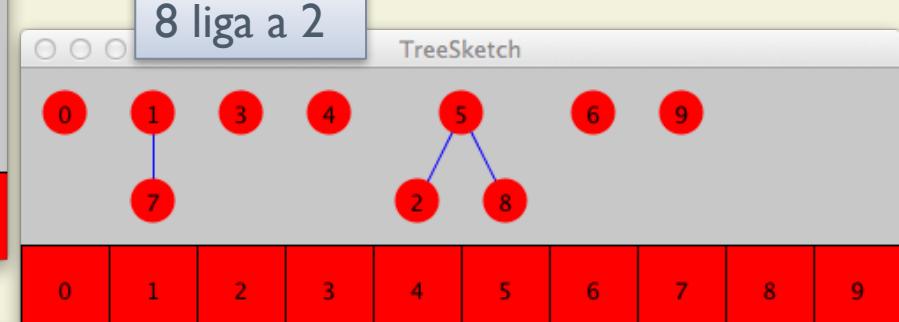
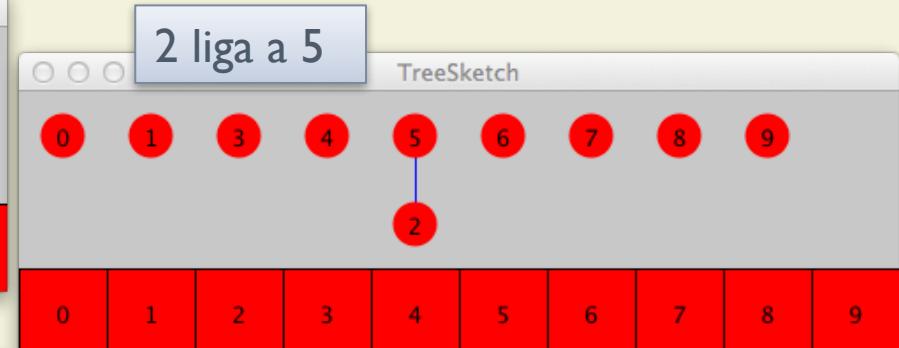
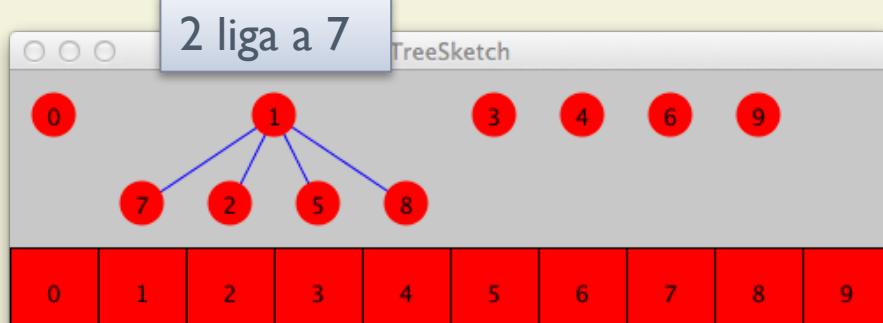
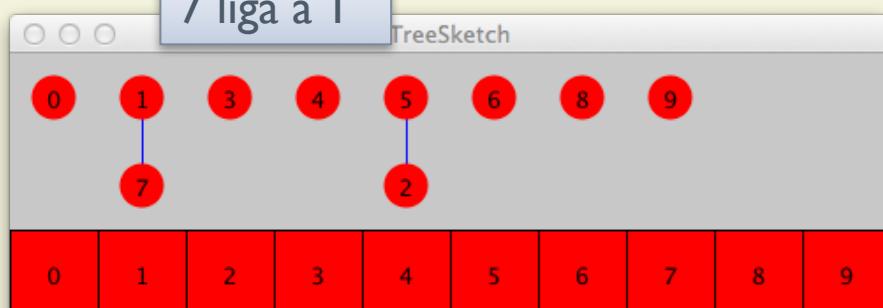
- Na consola:

```
$ java -cp ../../stdlib.jar:. QuickFind
-1 -1  0  1  2  3  4  5  6  7  8  9 / 10
5 7
5 7  0  1  2  3  4  7  6  7  8  9 / 9
3 2
3 2  0  1  2  2  4  7  6  7  8  9 / 8
9 2
9 2  0  1  2  2  4  7  6  7  8  2 / 7
9 5
9 5  0  1  7  7  4  7  6  7  8  7 / 6
3 5
3 5  0  1  7  7  4  7  6  7  8  7 / 6
1 0
1 0  0  0  7  7  4  7  6  7  8  7 / 5
1 4
1 4  4  4  7  7  4  7  6  7  8  7 / 4
0 2
0 2  7  7  7  7  7  7  6  7  8  7 / 3
8 6
8 6  7  7  7  7  7  7  6  7  6  7 / 2
2 8
2 8  6  6  6  6  6  6  6  6  6  6 / 1
```

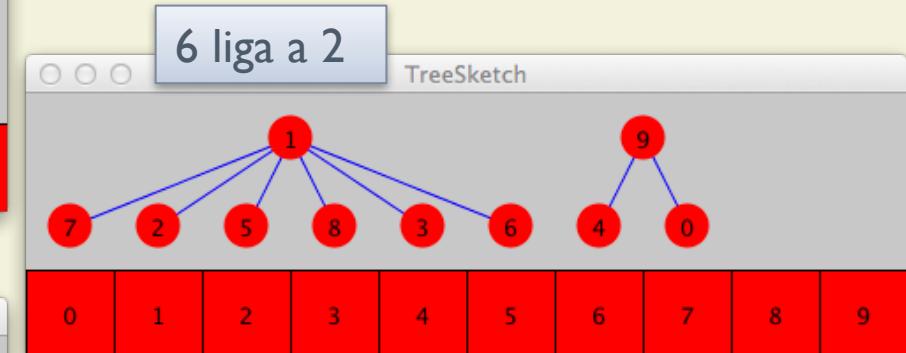
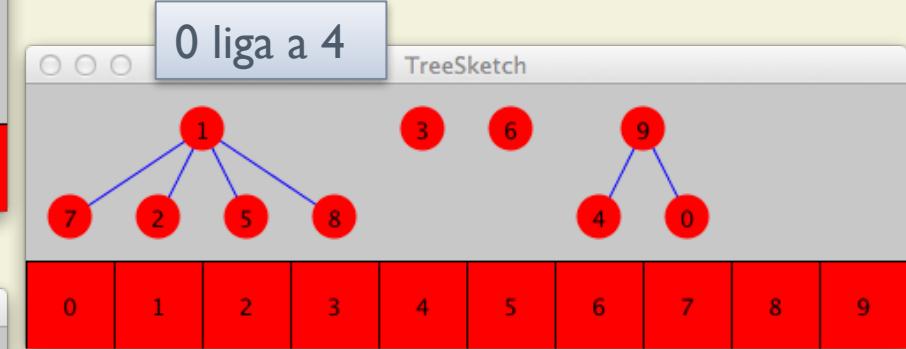
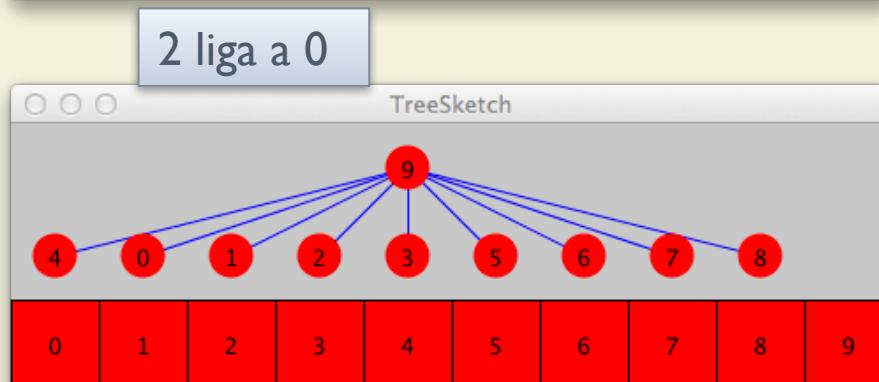
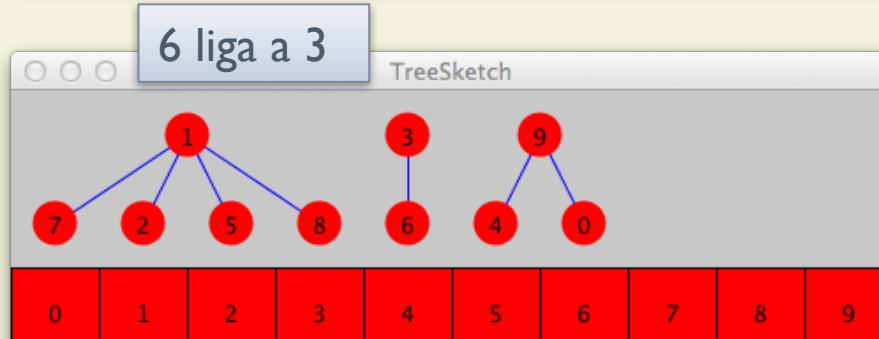
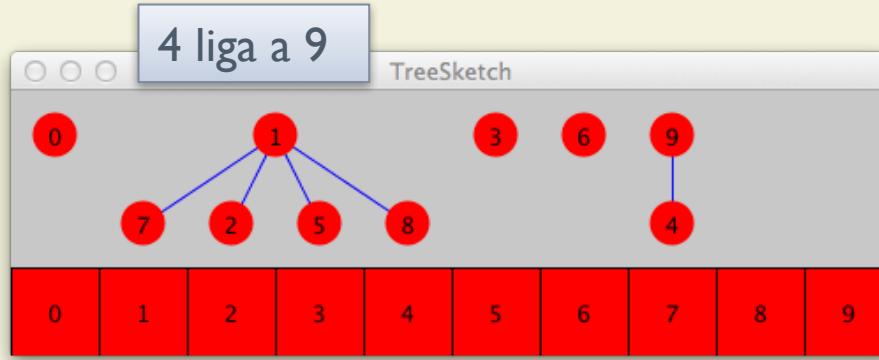
Confira, com atenção.

Outra visualização

- Talvez ajude, observar as componentes na forma de árvores, em que a raiz de cada árvore é o líder da componente:

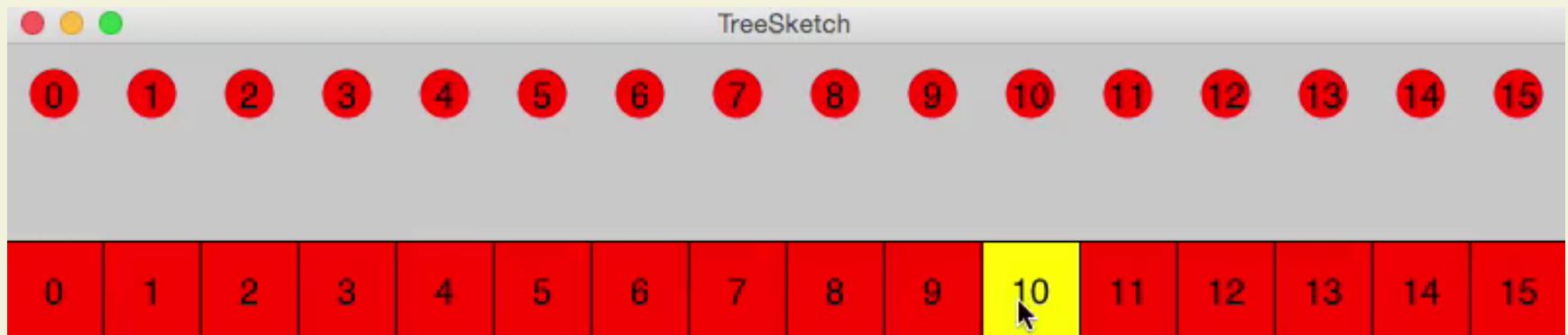


(Continuação)



Filme de uma experiência

- Observe:



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/23_aed_1415/movies/quick_find.m4v.

Análise

- Cada chamada de **union** faz no mínimo $N+3$ acessos ao array **id** e no máximo $2*N+1$ acessos, de cada vez que efetivamente combina duas componentes.
- Em cada chamada de **union**, o número de componentes diminui de uma unidade.
- Logo, se o processo continuar até haver só uma componente, teremos quando muito $(N+3)*(N-1) \sim N^2$ acessos ao array.
- Concluímos que o QuickFind é um algoritmo quadrático.

Repare:
$$(2*N+1)+2*(N-1)+1+\dots+2*2+1 =$$
$$((2*N+1)+5)/2)*(N-1) =$$
$$(N+3)*(N-1).$$

Considere um computador que realiza mil milhões de operações por segundo (1 gigahertz). Se um acesso ao array contar como uma operação e tivermos um milhão de nós, podemos estimar que o algoritmo demorará 1000 segundos. Por outro lado, se usássemos um algoritmo linearítmico, com um milhão de nós haveria cerca de 30 milhões de operações, o que quer dizer que o programa demoraria 0.03 segundos.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 11

QuickUnion

Union-Find

- QuickUnion.
- QuickUnion Weighted.



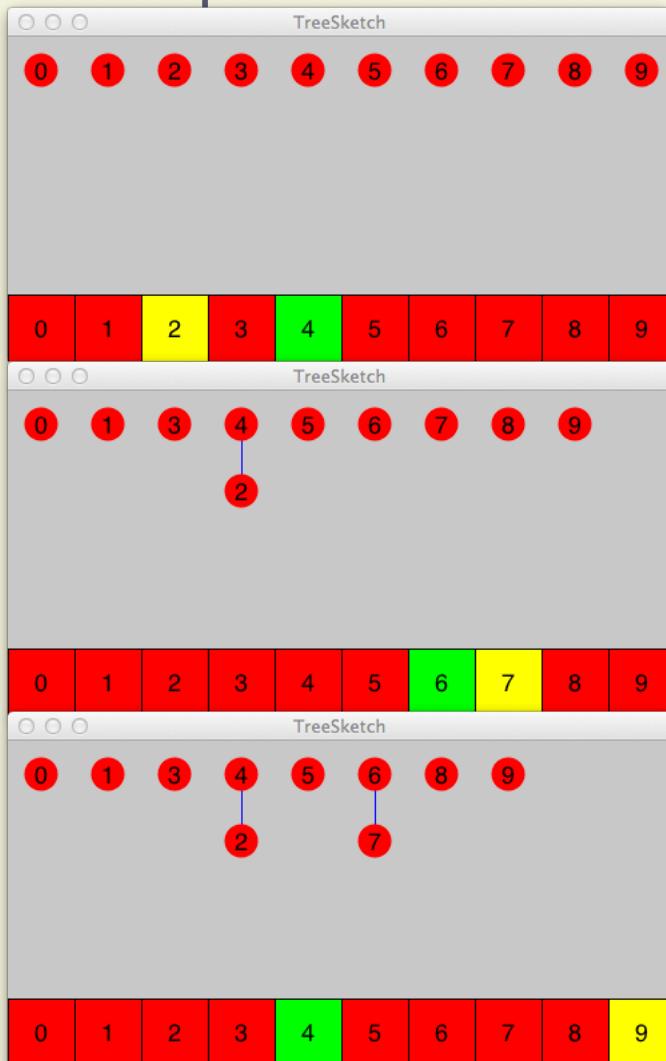
QuickUnion

- Na estratégia QuickUnion, nem todos os não líderes terão uma ligação direta ao líder da componente.
- Agora, quando ligamos um nó **x** a um nó **y**, apenas o líder da componente a que o nó **x** pertence regista a sua conexão ao líder da componente a que pertence o nó **y**.

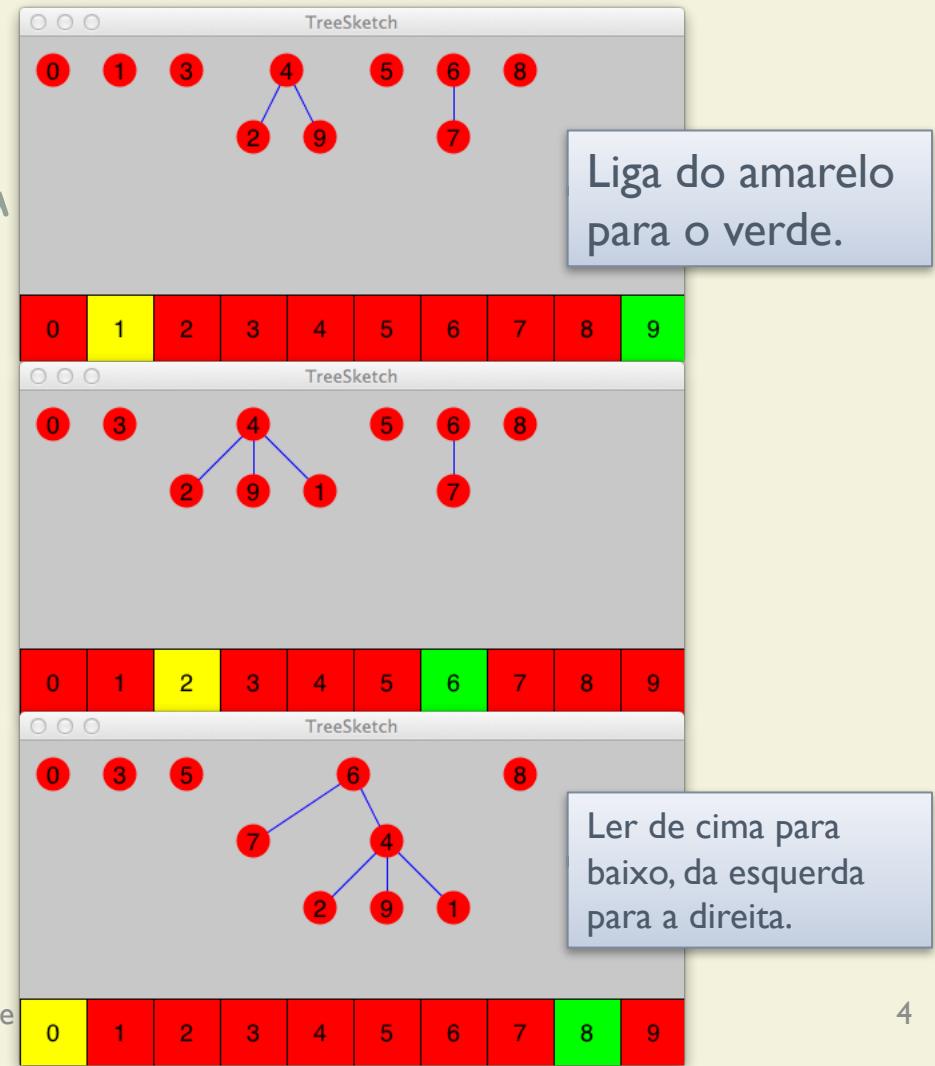
Podia ser ao contrário, claro.
- Note bem: só muda um nó: o líder da componente a que o nó **x** pertence.
- (No QuickFind, mudavam todos os nós da componente a que **x** pertence.)
- A ligação é inócua se **x** e **y** já estiverem conectados.

Visualizando

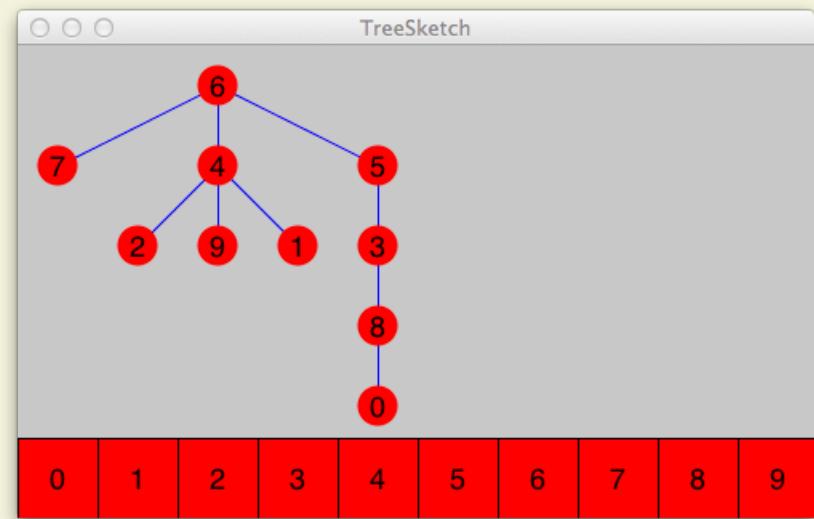
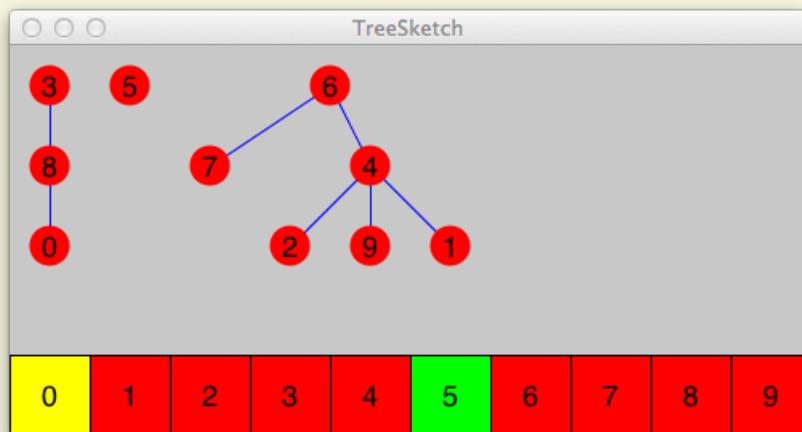
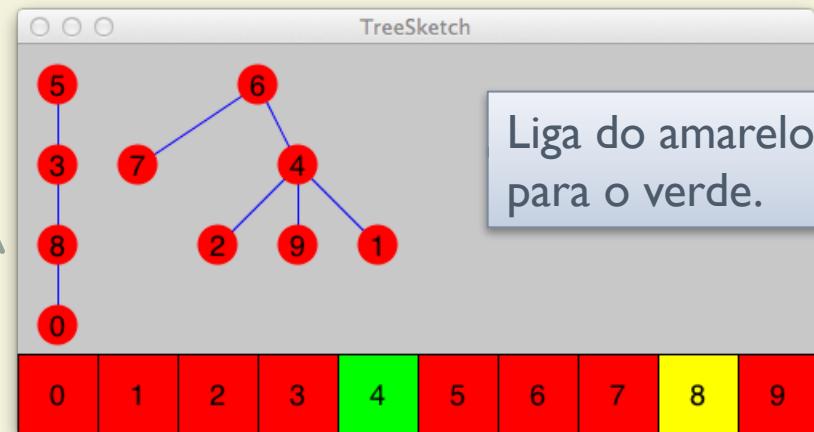
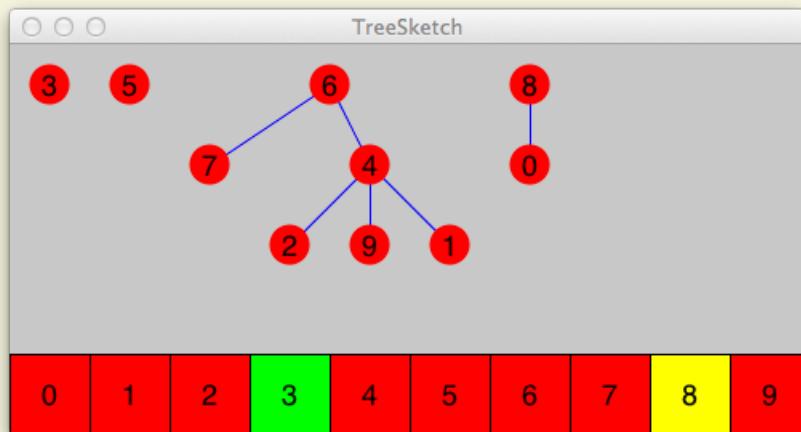
- O funcionamento do QuickUnion é o que os seguintes esquemas ilustram:



Algoritmos e



(Continuação)



Isto fica muito diferente do QuickFind. Qual será melhor?

Classe QuickUnion

- Os membros serão o array dos líderes e o número de componentes, como no QuickFind:

```
public class QuickUnion extends UnionFind
{
    private int[] id;
    private int count;
    ...
}
```

- No construtor, indicamos o número de nós e assinalamos que inicialmente cada nó é líder de si próprio, como no QuickFind:

```
public QuickUnion(int n)
{
    count = n;
    id = new int[n];
    for (int i = 0; i < n; i++)
        id[i] = i;
}
```

Funções count e union

- O método **count** é muito simples:

```
public int count()
{
    return count;
}
```

- O método **union** também:

```
public void union(int x, int y)
{
    int ix = find(x);
    int iy = find(y);
    if (ix != iy)
    {
        id[ix] = iy;
        count--;
    }
}
```

Parece simples, mas falta ver find.

Nem é preciso programar, o método **connected** porque fica acessível por herança.

Função find

- É preciso subir sucessivamente até chegar ao líder:

```
public int find(int x)
{
    while (x != id[x])
        x = id[x];
    return x;
}
```

- Até parece simples, mas atenção ao ciclo while!

Função de teste

- A função de teste é análoga à do QuickFind:

```
public static void testQuickUnion(String[] args)
{
    int size = 10;
    if (args.length != 0)
        size = Integer.parseInt(args[0]);
    QuickUnion qu = new QuickUnion(size);
    qu.show(-1, -1);
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        int y = StdIn.readInt();
        if (!qu.connected(x, y))
            qu.union(x, y);
        qu.show(x, y);
    }
}
```

Experimentando

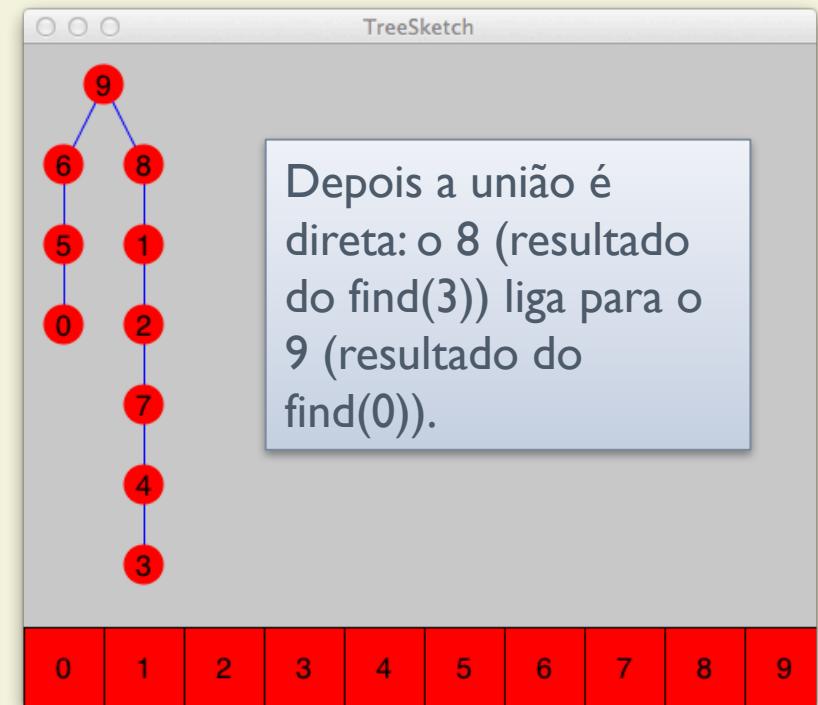
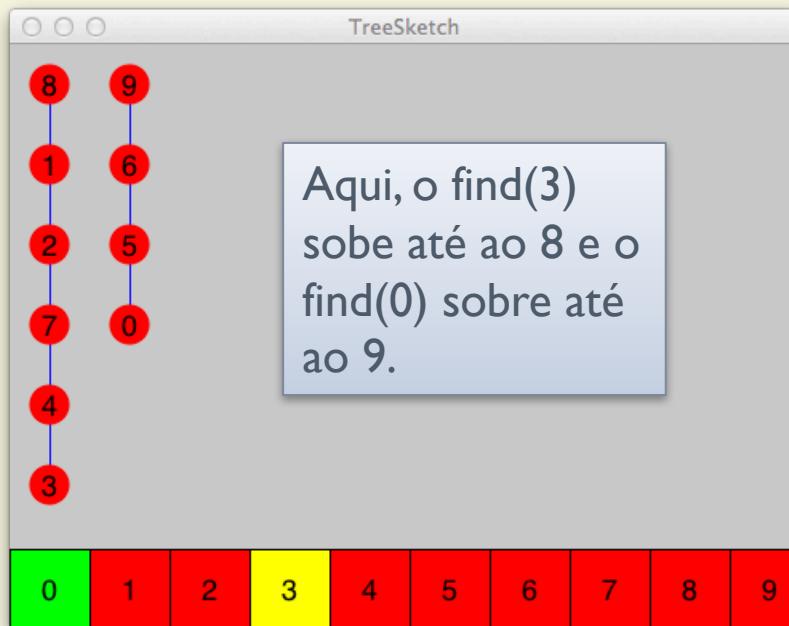
- Na consola:

```
pedros-imac-6:bin pedro$ java -cp ../stdlib.jar:../core.jar:..  
QuickUnion  
-1 -1 0 1 2 3 4 5 6 7 8 9 / 10  
2 4  
2 4 0 1 4 3 4 5 6 7 8 9 / 9  
7 6  
7 6 0 1 4 3 4 5 6 6 8 9 / 8  
9 4  
9 4 0 1 4 3 4 5 6 6 8 4 / 7  
1 9  
1 9 0 4 4 3 4 5 6 6 8 4 / 6  
2 6  
2 6 0 4 4 3 6 5 6 6 8 4 / 5  
0 8  
0 8 8 4 4 3 6 5 6 6 8 4 / 4  
8 3  
8 3 8 4 4 3 6 5 6 6 3 4 / 3  
0 5  
0 5 8 4 4 5 6 5 6 6 3 4 / 2  
8 4  
8 4 8 4 4 5 6 6 6 3 4 / 1
```

Confira, com atenção. É a mesma sequência de ligações do exemplo com as árvores.

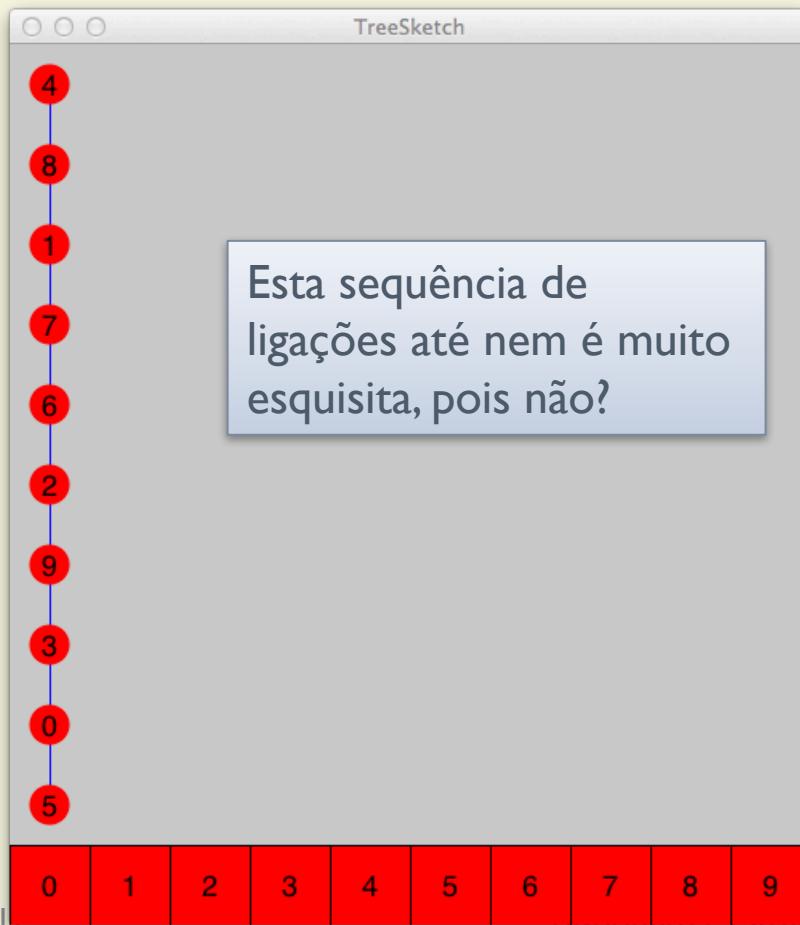
Análise

- Deve ser mais rápido do que o QuickFind, pois não precisa de percorrer o array todo no método **union**.
- Só num caso é que o **union** visita todos os nós (por intermédio do **find**): quando há só duas componentes com todos os nós em filinha e mandamos unir os nós de baixo:



Análise do pior caso

- O pior caso seria um em que só há uma filinha, ao longo dos cálculos, obtida, por exemplo, ligando um dos nós a cada um dos outros, por exemplo (5,0), (5,3), (5,9), (5,2), etc.



Neste caso o comportamento é quadrático.

Repare: na primeira vez, o `find(5)` visitou apenas o nó 5; da segunda o nó 5 e o nó 0; da terceira o 5, o 0 e o 3; etc.

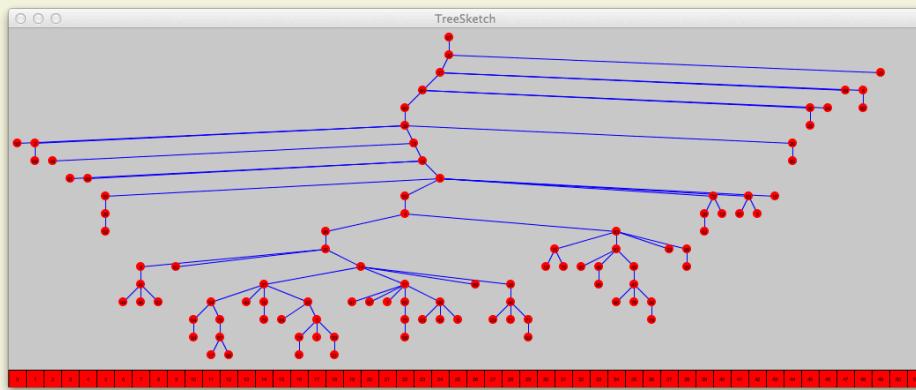
De cada vez, há ainda o `find` do outro nó, que só visita esse nó.

Portanto, na i -ésima ligação são visitados $i+1$ nós. Ora, havendo N nós, i varia de 1 a $N-1$.

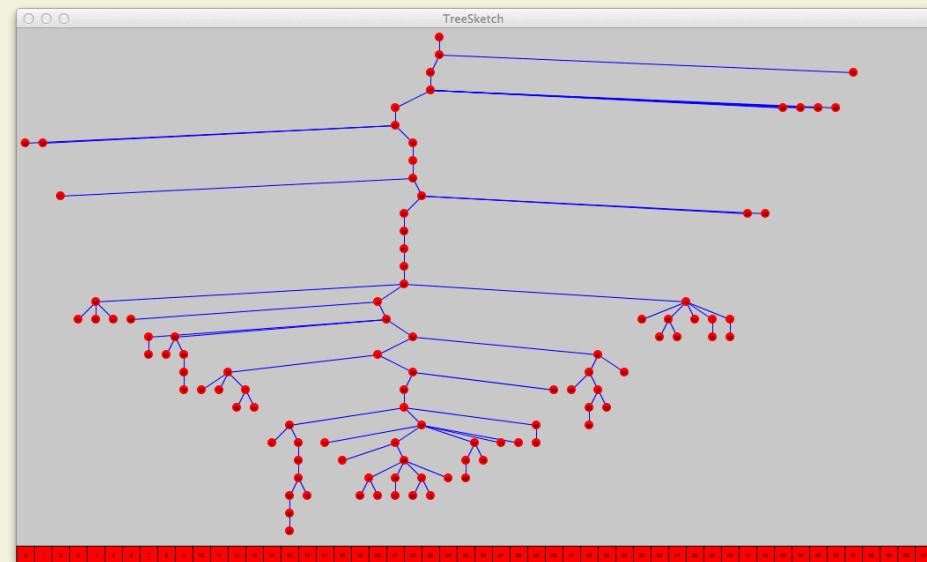
Ao todo o número de nós visitados é $2+3+4+\dots+N \sim N^2/2$.

Altura da árvore

- O caso anterior é fabricado, mas plausível.
- Mostra que não convém que as árvores fiquem muito altas.
- Podemos esperar que se as ligações forem feitas de forma aleatória, as alturas se distribuam melhor
- Eis dois exemplos com 100 nós, ligados aleatoriamente:

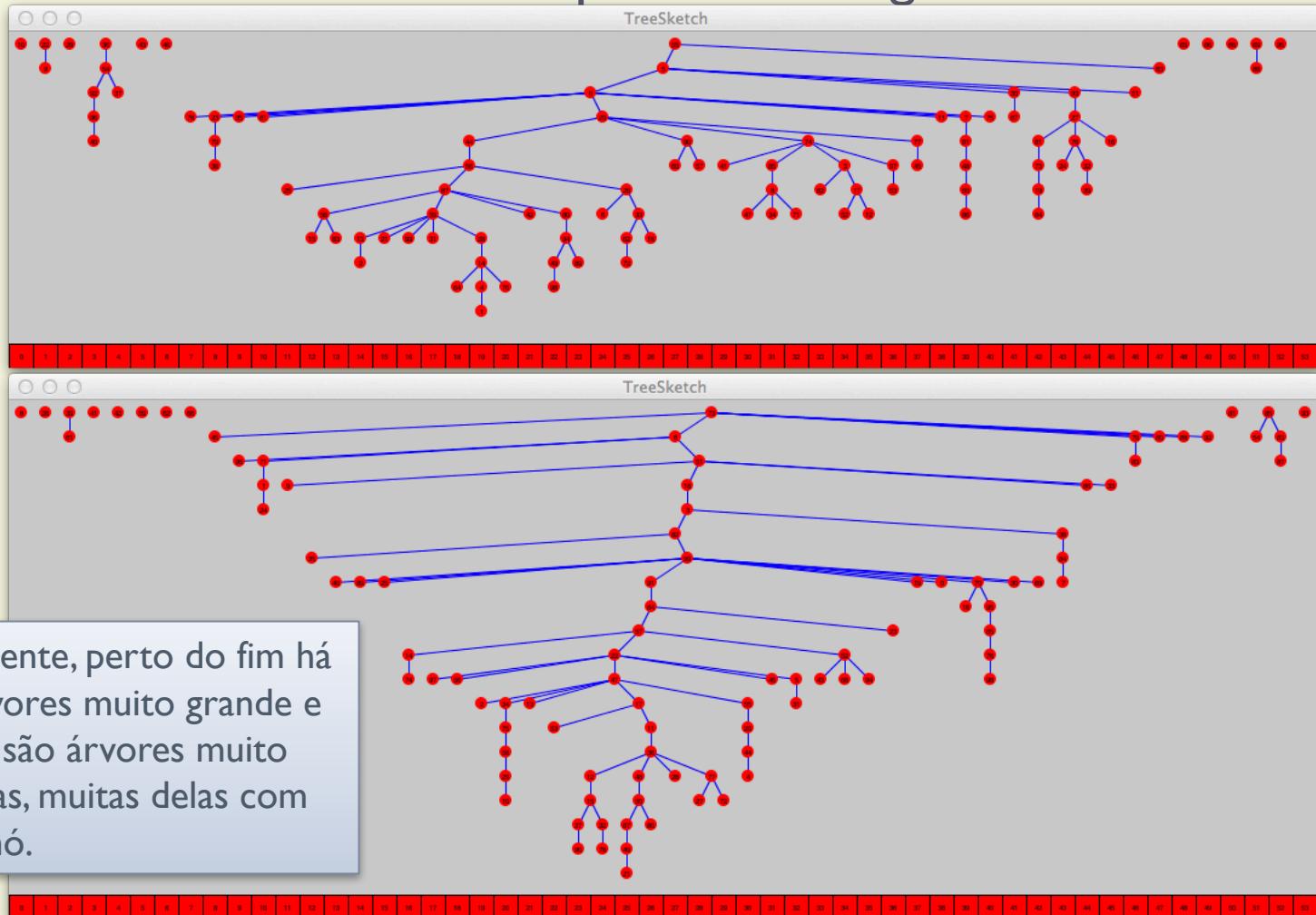


Mesmo assim, a altura final é mais de 20 e tal, nos dois casos. Veremos a seguir como fazer muito melhor.



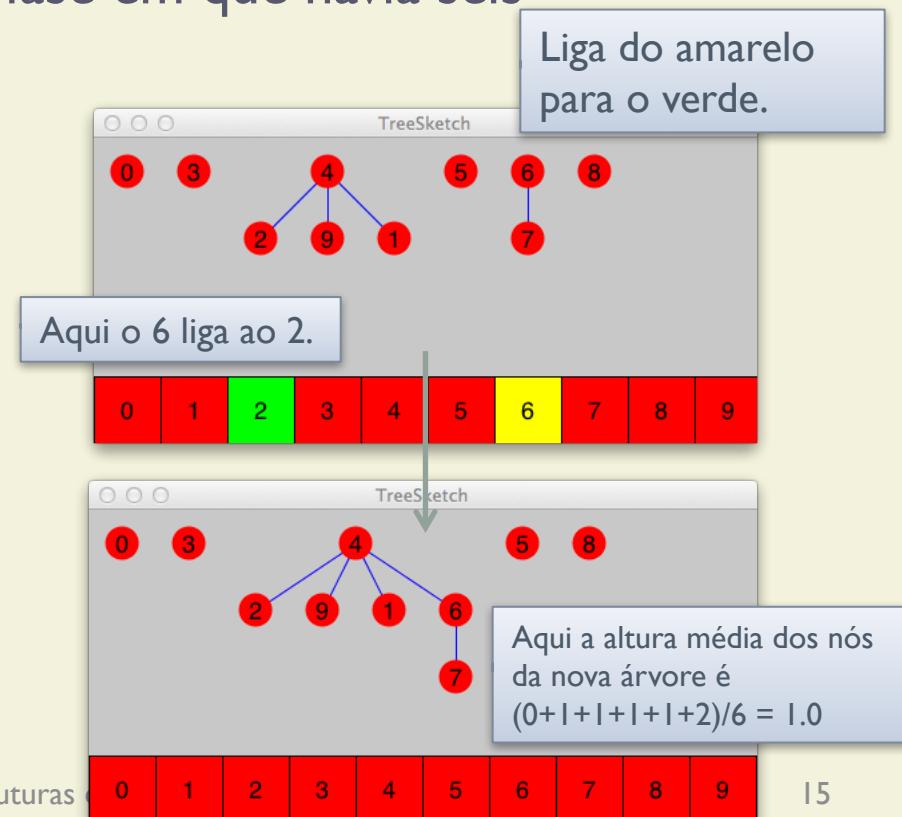
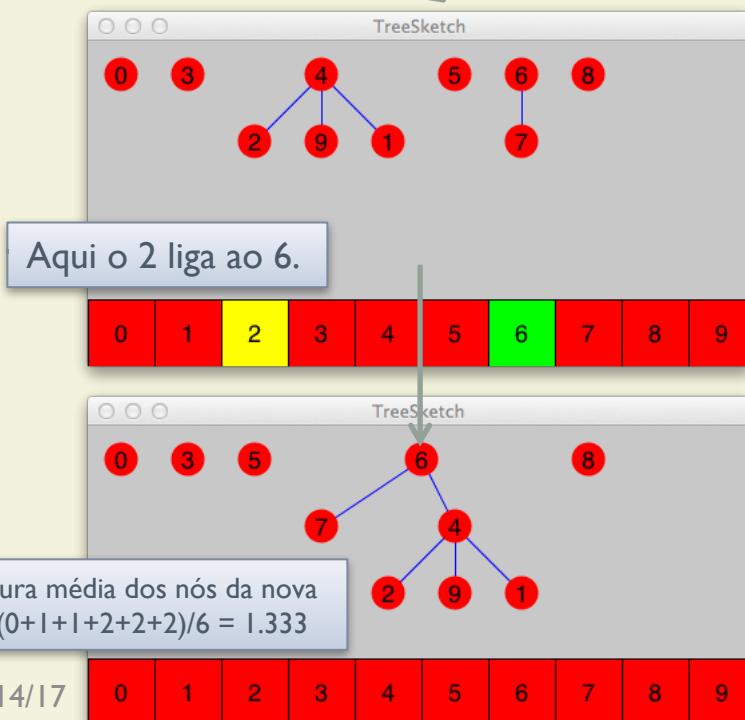
Altura da árvore, antes do fim

- Para referência eis dois exemplos da situação de 100 nós quando o número de componentes chegou a 12:



Ligando ao contrário

- Em rigor, é indiferente ligar **x** com **y** ou **y** com **x**.
- Portanto, o melhor é escolher a ligação que faça a altura da árvore crescer menos.
- Por exemplo, no exemplo da página 4, teria sido preferível ligar o 6 ao 2, em vez de ligar o 2 ao 6, na fase em que havia seis componentes:



Weighted QuickUnion

- A estratégia Weighted QuickUnion é como a QuickUnion, mas em cada ligação, o líder da componente menor regista a sua ligação ao líder da componente maior.
- Maior ou menor dirá respeito ao número de elementos, e não à altura.
- Portanto, convém controlar o tamanho de cada componente, ao longo do processo.

Classe QuickUnionWeighted

- Além do array dos líderes e do número de componentes, existe o array dos tamanhos:

```
public class QuickUnionweighted extends UnionFind
{
    private int[] id;
    private int count;
    private int[] size;
    ...
}
```

- O construtor tem de inicializar também o array dos tamanhos:

```
public QuickUnionweighted(int n)
{
    count = n;
    id = new int[n];
    for (int i = 0; i < n; i++)
        id[i] = i;
    size = new int[n];
    for (int i = 0; i < n; i++)
        size[i] = 1;
}
```

Função union, na classe “pesada”

- É mais comprida, mas é simples:

```
public void union(int x, int y)
{
    int ix = find(x);
    int iy = find(y);
    if (ix != iy)
    {
        if (size[ix] < size[iy])
        {
            id[ix] = iy;
            size[iy] += size[ix];
        }
        else
        {
            id[iy] = ix;
            size[ix] += size[iy];
        }
        count--;
    }
}
```

Aqui liga do x para o y.

Aqui liga do y para o x.

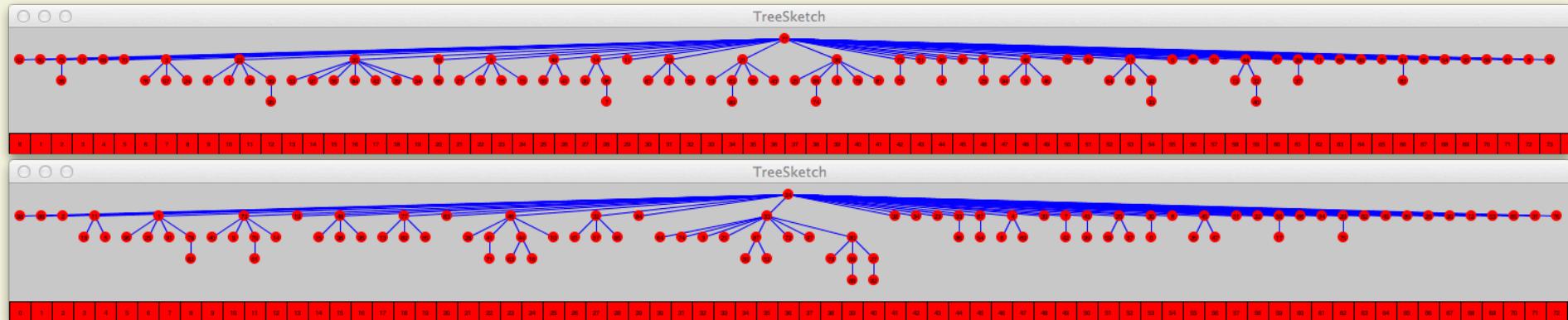
As outras funções

- As funções count, find e connected são como em QuickUnion.
- Acrescentamos uma função para o tamanho:

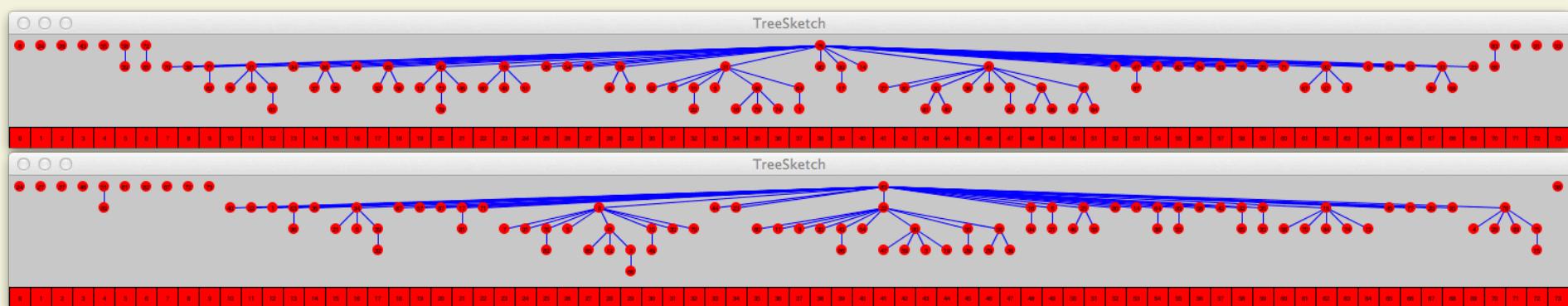
```
// the size of the component to which x belongs.  
public int size(int x)  
{  
    return size[find(x)];  
}
```

Árvores do QuickUnion Weighted

- Eis dois exemplos aleatórios, no final:



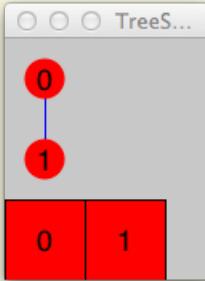
- E mais dois, com 12 componentes:



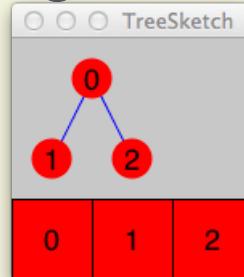
- Estas árvores são nitidamente melhores que as do QuickUnion.

Análise do caso mais desfavorável

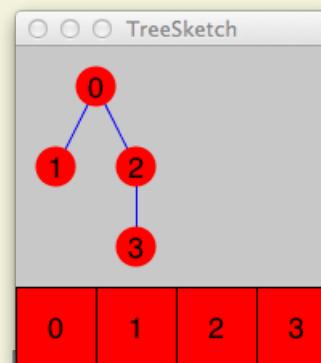
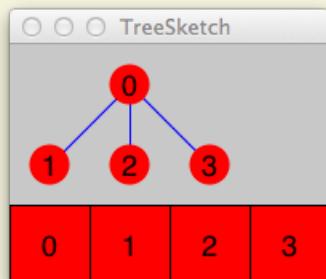
- Se houver 2 nós, a configuração final só pode ser uma:



- Se houver 3 nós, a configuração final só pode ser a seguinte:



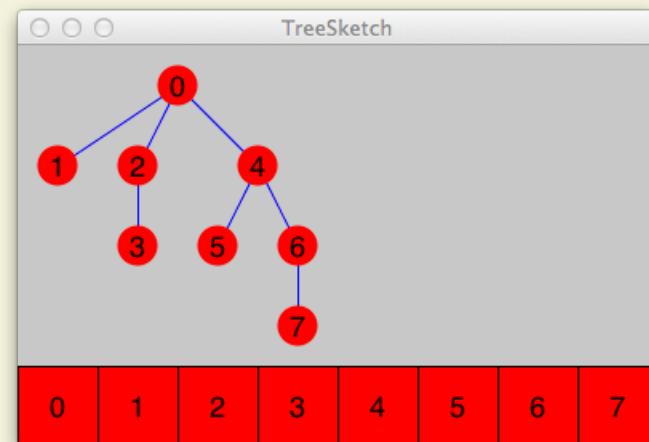
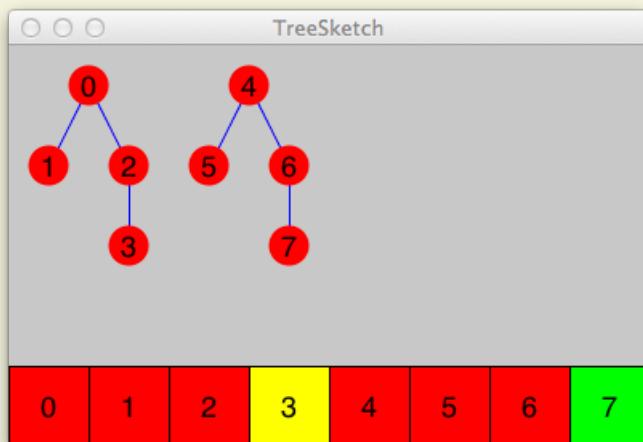
- Se houver 4 nós, só há duas hipóteses:



4 + 4

- Se juntarmos uma árvore de 4 nós com uma árvore de 1 nó, dará uma árvore com a mesma altura que a de quatro nós.
- Se juntarmos uma árvore de 4 nós com uma de 2 nós ou com uma de 3 nós, a altura não ficará maior que 2, certamente.
- Só se juntarmos duas árvores de 4 nós das “desequilibradas” é que a altura aumenta, para 3:

Note que todas as árvores com 2 nós têm altura 1 e que todas as árvores com 3 nós no QuickUnionWeighted terão também altura 1.



Comportamento logarítmico do `find`

- Generalizando a observação anterior, concluímos que a altura de uma árvore com 2^N nós será sempre menor ou igual a N .
- Ou, inversamente, que a altura de uma árvore com N nós é menor ou igual a $\text{floor}(\log_2 N)$.
- Por conseguinte, o `find` no QuickUnion Weighted é logarítmico.
- Havendo N nós, o número de operações necessárias conectar os nós todos é proporcional a $N \log N$ (excluindo o tempo perdido com ligações de nós já ligados...)

Moral da história

- É notável que uma ideia simples mais engenhosa permita melhorar tão drasticamente o desempenho do algoritmo.
- Na prática, usaremos o QuickUnion Weighted e não se fala mais nisso. Os outros têm interesse “teórico” apenas.
- Existe uma variante chamada QuickUnion pesado com compressão de caminho, a qual ainda é mais eficiente.
- Consiste em acrescentar um segundo ciclo no find, ligando cada nó diretamente ao líder.
- Assim teremos árvores planas (como no QuickUnion), ou quase planas, quase sempre.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 12

Ordenação de arrays

Ordenação de arrays



- Preliminares.
- Classes para os algoritmos de ordenação.
- Animação algorítmica.
- Algoritmos quadráticos:
 - Bubblesort.
 - Selectionsort.
 - Insertionsort.

Ordenação de arrays

- Ordenar um array é reorganizá-lo de maneira a que os elementos sejam colocados numa sequência que nos convenha.
- Convém-nos ordenar para depois encontrar mais depressa.
- Convém-nos ordenar para que elementos “parecidos” fiquem próximos uns dos outros.
- Convém-nos ordenar para os “melhores” aparecerem primeiro.
- Na prática, para ordenar, usamos funções de biblioteca.
- Mesmo assim, conhecer os principais algoritmos de ordenação faz parte da bagagem técnica dos programadores.

Provérbio de programação: *se não sabes o que fazer, ordena.*

Regras

- Ordenaremos arrays com elementos de um tipo **T**, qualquer, desde que **T** seja “comparável”, isto é, desde que implemente a interface **Comparable<T>**.
- Nos algoritmos, usaremos uma função **less** para comparar dois elementos do array. Esta função é implementada em termos do método **compareTo**, da interface **Comparable<T>**.
- Em muitos algoritmos, a ordenação consegue-se por uma sucessão de trocas de pares de elementos no array. Essa troca é operada por meio de uma função **exchange**.

Classe abstrata Sort

- As classes que implementam os diversos algoritmos que vamos estudar, estendem a classe **Sort**, a qual reúne os recursos comuns e deixa abstrato o método **sort** , o qual será depois implementado em cada uma das classes derivadas:

```
public abstract class Sort<T extends Comparable<T>>
{
    public abstract void sort(T[] a);

    public boolean less(T x, T y)
    {
        return x.compareTo(y) < 0;
    }

    public void exchange(T[] a, int x, int y)
    {
        T m = a[x];
        a[x] = a[y];
        a[y] = m;
    }

    ...
}
```

Funções isSorted

- Acrescentamos uma função para verificar se um array se está ordenado:

```
public boolean isSorted(T[] a)
{
    for (int i = 1; i < a.length; i++)
        if (less(a[i], a[i-1]))
            return false;
    return true;
}
```

Tipicamente usa-se *a posteriori*, para asseverar que a ordenação funcionou.

Funções de teste

- Equipamos a classe **Sort** com duas funções de teste, para arrays de **Integer** e para arrays de **String**:

```
public static void testIntegers(Sort<Integer> s)
{
    Integer[] a = Utils.integersFromInts(StdIn.readAllInts());
    s.sort(a);
    StdOut.println(utils.mkString(a, "\n"));
}
```

```
public static void testStrings(Sort<String> s)
{
    String[] a = StdIn.readAllStrings();
    s.sort(a);
    StdOut.println(utils.mkString(a, "\n"));
}
```

A função **integersFromInts** vem na página seguinte.

Note bem: a nossa construção não serve para ordenar arrays de **int**, pois o tipo **int** não é uma classe.

Funções de apoio

- A função **integersFromInts** transforma um array de int em um array de Integer:

```
public static Integer[] integersFromInts(int[] a)
{
    Integer[] result = new Integer[a.length];
    for (int i = 0; i < a.length; i++)
        result[i] = a[i];
    return result;
}
```

- A função **intsFromIntegers** faz o contrário:

```
public static int[] intsFromIntegers(Integer[] a)
{
    int[] result = new int[a.length];
    for (int i = 0; i < a.length; i++)
        result[i] = a[i];
    return result;
}
```

Ambas as funções residem na classe Utils.

Classe Bubblesort

- Experimentemos, com uma classe para o bubblesort:

```
public class Bubblesort<T extends Comparable<T>>
    extends Sort<T>
{
    public void sort(T[] a)
    {
        int n = a.length;
        for (int i = 1; i < n; i++)
            for (int j = n-1; j >= i; j--)
                if (less(a[j], a[j-1]))
                    exchange(a, j-1, j);
    }
    ...
}
```

Haverá uma classe destas para cada algoritmo.

Testando o bubblesort

- Podemos testar usando as funções de teste herdadas:

```
public static void main(String[] args)
{
    char choice = 'A';
    if (args.length >= 1)
        choice = args[0].charAt(0);
    if (choice == 'A')
        testIntegers(new Bubblesort<Integer>());
    else if (choice == 'B')
        testStrings(new Bubblesort<String>());
    else
        StdOut.printf("Illegal choice: %s\n", args[0]);
}
```

```
$ java -ea -cp ../../*:. Bubblesort A
64 12 89 5 52 81 62 10 99 18 12 55
5 10 12 12 18 52 55 62 64 81 89 99
$ java -ea -cp ../../*:. Bubblesort B
uuu rrr yyy qqq aaa yyy rrr iii ooo qqq aaa qqq ooo
aaa aaa iii ooo ooo qqq qqq qqq rrr rrr uuu yyy yyy
$
```

Bubblesort

- “Percorrer sucessivamente o array da direita para a esquerda trocando os elementos adjacentes que estejam fora de ordem”.
- É um algoritmo bom para programar.
- Não é difícil de entender.
- É muito instrutivo.
- De todos os algoritmos elementares de ordenação é o menos eficiente.

Complexidade do Bubblesort

- Já sabemos:

```
public class Bubblesort<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        int n = a.length;
        for (int i = 1; i < n; i++)
            for (int j = n-1; j >= i; j--)
                if (less(a[j], a[j-1]))
                    exchange(a, j-1, j);
    }
    ...
}
```

- Nitidamente, o número de comparações é $(n-1) + (n-2) + \dots + 1 \sim n^2 / 2$.
- O número de trocas depende dos dados. Se o array estiver ordenado, não haverá trocas. Se estiver por ordem inversa, haverá $\sim n^2/2$ trocas. Por um argumento de simetria, em média haverá $\sim n^2/4$ trocas.

Ensaios de razão dobrada

- Desenhemos uma classe **SortDoubleRatio** para ensaios de razão dobrada com algoritmos de ordenação:
- Seguimos o modelo usado no problema da soma tripla, com adaptações:

```
public class SortDoubleRatio
{
    private static double tMin = 0.5;
    // minimum duration of a test in the double ratio experiment

    public static double timing(Sort<Integer> s, Integer[] a)
    {
        Stopwatch timer = new Stopwatch();
        s.sort(a);
        double result = timer.elapsedTime();
        assert s.isSorted(a); // just in case something went wrong...
        return result;
    }
}
```

Classe SortDoubleRatio, continuação

```
public static double[] doubleRatio(Sort<Integer> s, int size, int max, int times)
{
    double[] result = new double[times];
    for (int i = 0; i < times; i++, size *= 2, max *= 2)
    {
        double t = 0.0;
        int n = 0;
        while (t < tMin)
        {
            Integer[] a = Utils.IntegersFromInts(RandomArrays.uniform(size, max));
            t += timing(s, a);
            n++;
        }
        result[i] = t / n;
    }
    return result;
}

private static void printTable(int size, double[] t)
{
    double t0 = 0.0;
    for (int i = 0; i < t.length; i++, size *= 2)
    {
        StdOut.printf("%d %.4f", size, t[i]);
        if (t0 > 0.0) // for i == 0 and also in case t[i] == 0
            StdOut.printf(" %.2f", t[i] / t0);
        StdOut.println();
        t0 = t[i];
    }
}

public static void test(Sort<Integer> s, int size, int max, int times)
{
    double z[] = doubleRatio(s, size, max, times);
    printTable(size, z);
}
```

Razão dobrada, Bubblesort

- Acrescentamos um teste na função main:

```
public static void main(String[] args)
{
    ...
    else if (choice == 'C')
        testDoubleRatio(new Bubblesort<Integer>(), 1000, 10000, 7)
    ...
}
```

```
$ java -cp ../../*:. Bubblesort C
1000 0.0024
2000 0.0086 3.64
4000 0.0358 4.16
8000 0.1697 4.74
16000 0.8360 4.93
32000 3.9060 4.67
64000 16.6370 4.26
$
```

Observamos comportamento quadrático, tal como esperávamos.

Regra prática: com o bubblesort ordenamos cerca de 20000 Integers num segundo.

Comparação com arrays de ints

- Qual será a penalização em tempo por usarmos arrays de Integer em vez de arrays de int?
- Experimentemos, com uma classe *ad hoc*:

```
public final class BubblesortInts
{
    public static void sort(int[] a)
    {
        int n = a.length;
        for (int i = 1; i < n; i++)
            for (int j = n-1; j >= i; j--)
                if (a[j] < a[j-1])
                {
                    int m = a[j-1];
                    a[j-1] = a[j];
                    a[j] = m;
                }
    }
    ...
}
```

Note bem: a classe **SortDoubleRatio** não é aplicável aqui, porque a classe **BubblesortInts** não deriva de **Sort<T>**. É preciso repetir tudo!

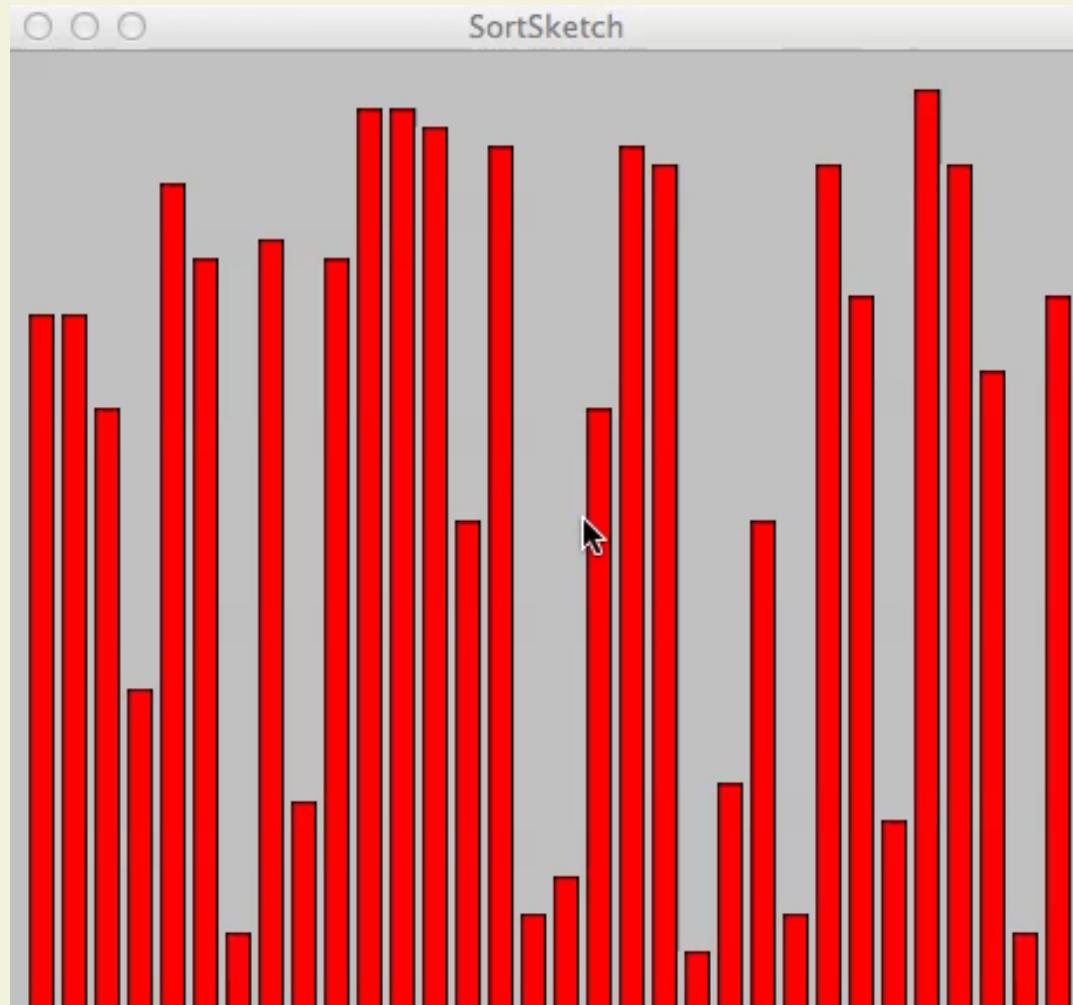
```
$ java ... BubblesortInts C
1000 0.0014
2000 0.0053 3.87
4000 0.0220 4.16
8000 0.0928 4.22
16000 0.3855 4.15
32000 1.6080 4.17
64000 6.5060 4.05
```

Concluímos que os tempos com este bubblesort com **ints** estão consistentemente abaixo de metade dos tempos do anterior bubblesort com **Integers**.

Overhead de memória

- Observámos que ordenar arrays de **Integer** leva mais tempo que ordenar arrays de **int**.
- E também gasta mais memória.
- De facto, em Java, um **int** ocupa 4 bytes; um **Integer** ocupa 24 bytes.
- Em geral, num objeto, há a memória dos membros (*instance variables*), mais 16 bytes de *overhead* (referência para a classe, informação para a *garbage collection* e informação para sincronização); além disso, a memória é “arredondada” para cima, para um múltiplo de 8 bytes.
- Num **Integer**, há um membro, de tipo **int**, ocupando 4 bytes; com o *overhead* chegamos a 20 bytes e com o arredondamento chegamos a 24.

Filme do bubblesort



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/bubblesort.m4v

Selection sort

- “Percorrer o array da esquerda para a direita trocando cada elemento com o menor elemento existente daí para a direita”.
- É talvez o algoritmo mais intuitivo.
- Baseia-se no algoritmo do mínimo de um array.

Complexidade do Selectionsort

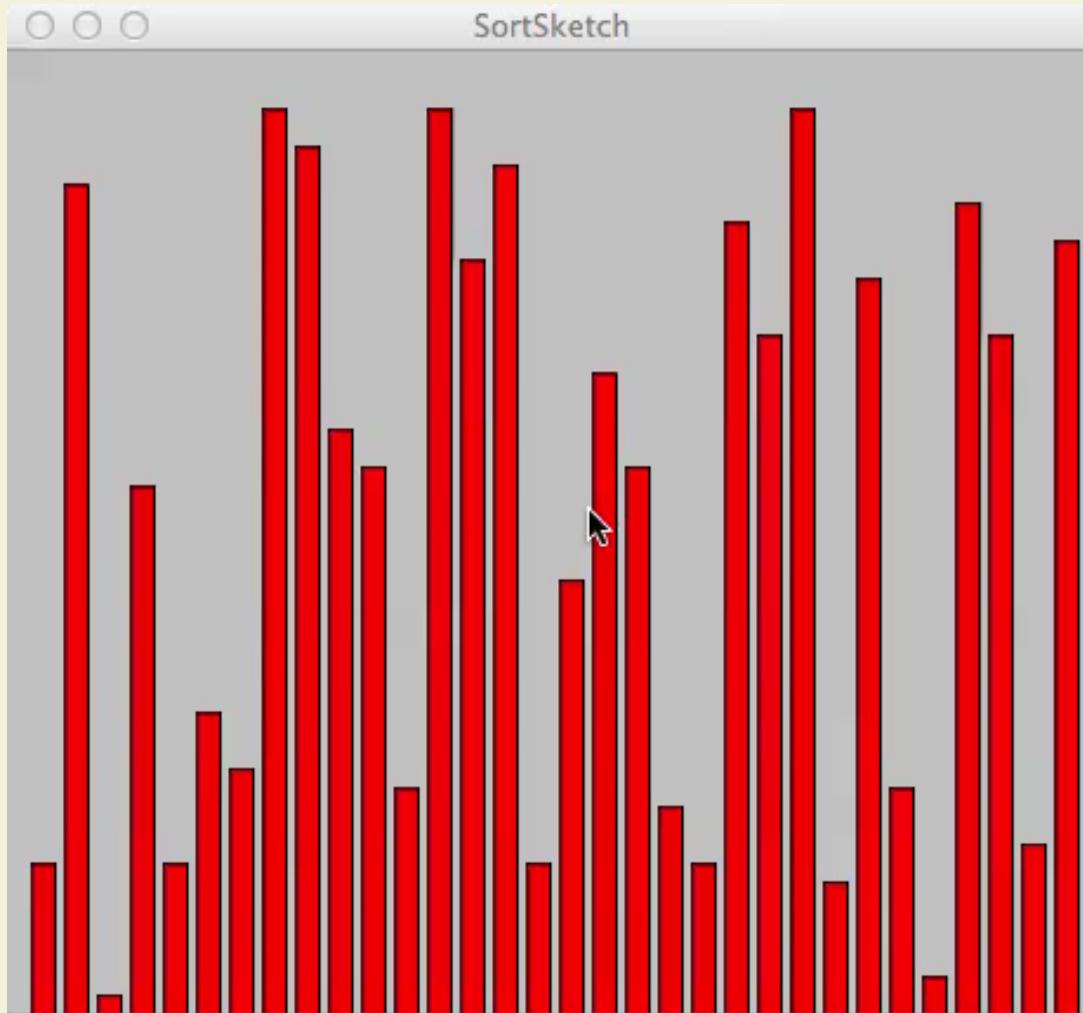
- A função **sort** exprime o algoritmo:

```
public class Selectionsort<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        int n = a.length;
        for (int i = 0; i < n; i++)
        {
            int m = i;
            for (int j = i + 1; j < n; j++)
                if (less(a[j], a[m]))
                    m = j;
            exchange(a, i, m);
        }
    }
    ...
}
```

No final do ciclo for interno, a variável **m** contém o índice do elemento mínimo no subarray correspondente ao intervalo de índices $[i..n[$.

- Nitidamente, o número de comparações é $(n-1) + (n-2) + \dots + 1 \sim n^2 / 2$.
- Haverá **n** trocas, mesmo se o array estiver ordenado.
- Logo, o tempo de execução não depende dos valores no array.

Filme do selectionsort



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/selectionsort.m4v

Razão dobrada, Selectionsort

- No meu computador:

```
$ java -cp ../../*:. Selectionsort C  
1000 0.0011  
2000 0.0042 3.75  
4000 0.0168 3.96  
8000 0.0663 3.95  
16000 0.2625 3.96  
32000 1.1050 4.21  
64000 4.6990 4.25  
$
```

Comportamento quadrático.

Observamos que é mais de três vezes mais rápido do que o bubblesort, com arrays aleatórios.

```
$ java -cp ../../*:. Bubblesort C  
1000 0.0024  
2000 0.0086 3.64  
4000 0.0358 4.16  
8000 0.1697 4.74  
16000 0.8360 4.93  
32000 3.9060 4.67  
64000 16.6370 4.26  
$
```

Insertion sort

- “Percorrer o array da esquerda para a direita inserindo ordenadamente cada elemento no subarray à sua esquerda, o qual está ordenado”.
- Dá jeito para ordenar uma mão de cartas de jogar.
- No nosso caso, a inserção ordenada faz-se por trocas sucessivas de dois elementos adjacentes, da direita para a esquerda, enquanto os dois elementos adjacentes estiverem fora de ordem.
- Quer dizer, as trocas terminam ou quando os dois elementos adjacentes estiverem por ordem ou quando o percurso tiver chegado ao início do array.

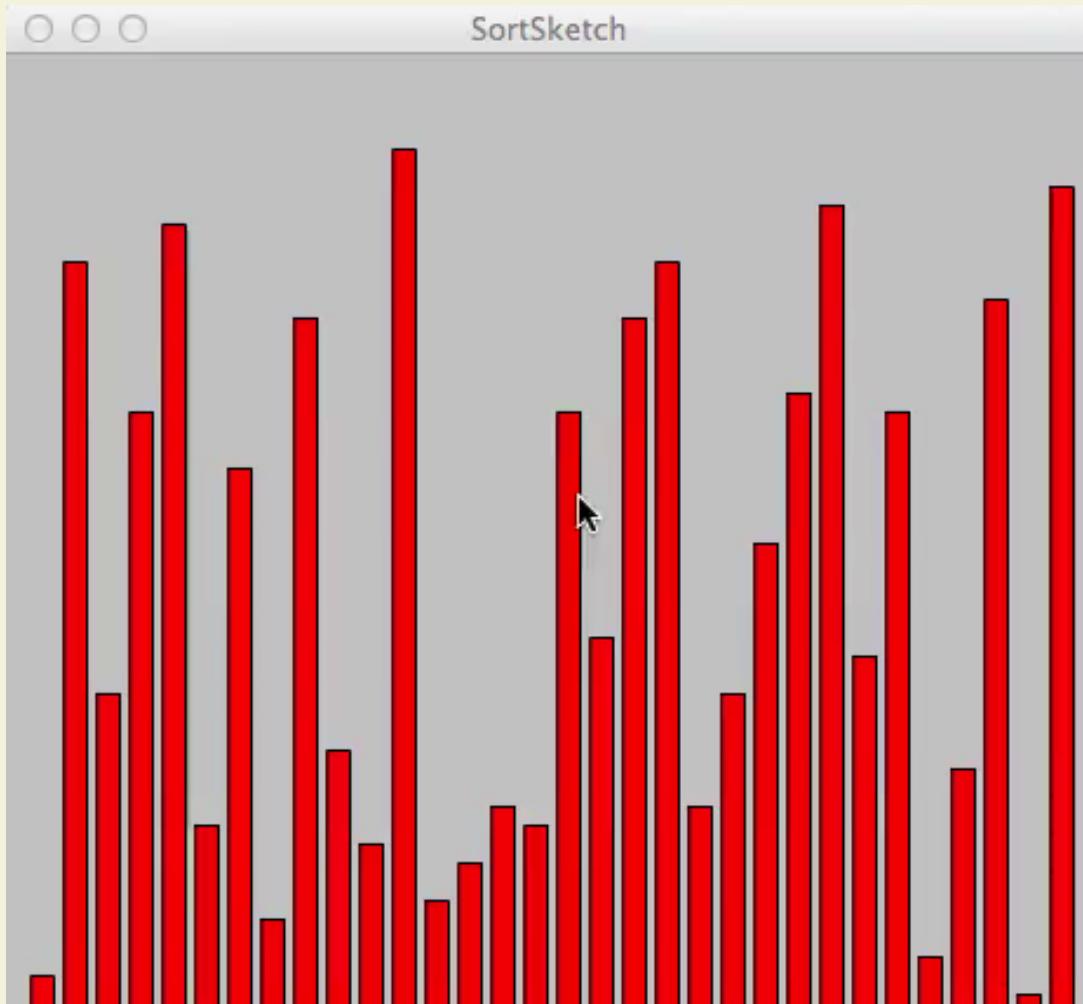
Classe Insertionsort

- A função **sort** exprime o algoritmo:

```
public class Insertionsort<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        int n = a.length;
        for (int i = 1; i < n; i++)
            for (int j = i; j >= 1 && less(a[j], a[j - 1]); j--)
                exchange(a, j, j - 1);
    }
    ...
}
```

- No pior caso (com o array por ordem inversa) o número de comparações é $1 + 2 + \dots + (n-1) \sim n^2 / 2$; no melhor caso (com o array ordenado) há **n-1** comparações.
- Com arrays aleatórios, em média a condição **less** no ciclo for anterior interrompe o ciclo a meio, entre i e 0.
- Logo, com arrays aleatórios, em média o número de comparações é $\sim n^2/4$.

Filme do insertionsort



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/insertionsort.m4v

Ensaio de razão dobrada, insertionsort

- No meu computador:

```
$ java -cp ../../*:. Insertionsort C  
1000 0.0014  
2000 0.0053 3.88  
4000 0.0215 4.03  
8000 0.0878 4.09  
16000 0.3530 4.02  
32000 1.5850 4.49  
64000 6.8030 4.29  
$
```

Comportamento quadrático,
tal como o bubblesort e o
selectionsort.

Observamos os tempos são
comparáveis aos do Selectionsort,
ainda que ligeiramente piores.

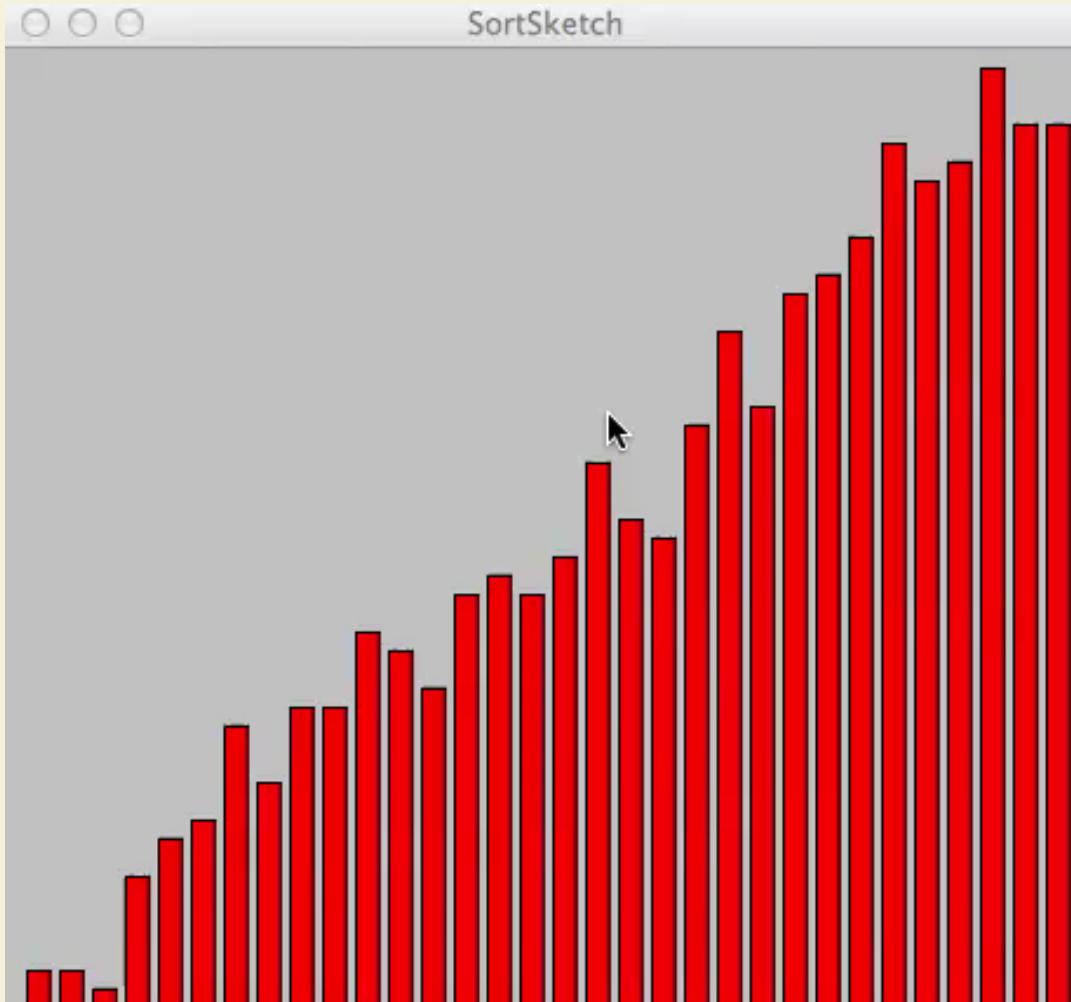
```
$ java -cp ../../*:. Selectionsort C  
1000 0.0011  
2000 0.0042 3.75  
4000 0.0168 3.96  
8000 0.0663 3.95  
16000 0.2625 3.96  
32000 1.1050 4.21  
64000 4.6990 4.25  
$
```

Arrays parcialmente ordenados

- Os arrays “parcialmente ordenados” dão pouco trabalho a ordenar com o insertionsort.
- Exemplos de arrays “parcialmente ordenados”:
 - Um array em que cada elemento está perto da sua posição final.
 - Um array com apenas poucos elementos fora de ordem.
 - Um array formado por um pequeno array concatenado no fim a um array grande que já está ordenado.
- Estas situações são relativamente frequentes, na prática.

Filme do insertionsort, especial-I

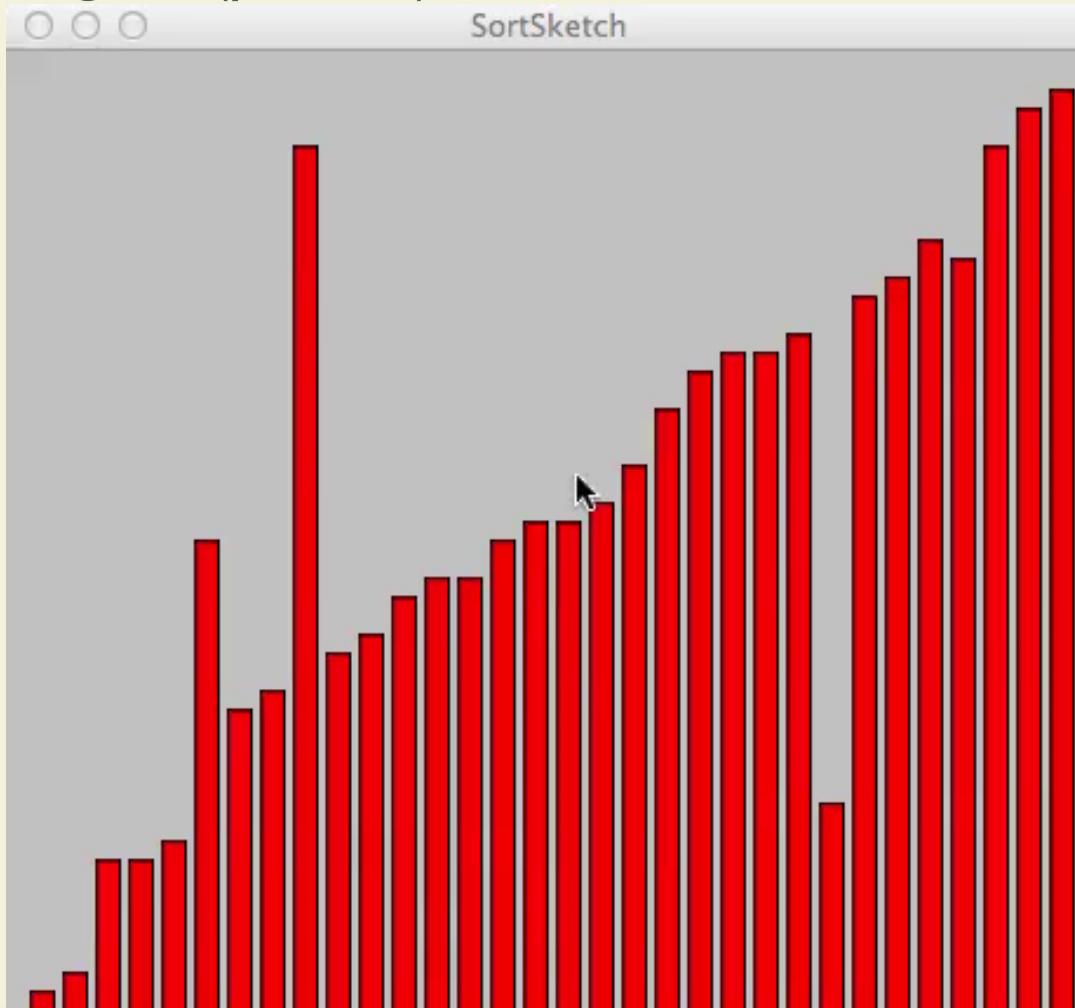
- Neste caso, cada elemento do array está perto da sua posição final:



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/insertionsort_specialI.m4v

Filme do insertionsort, especial–2

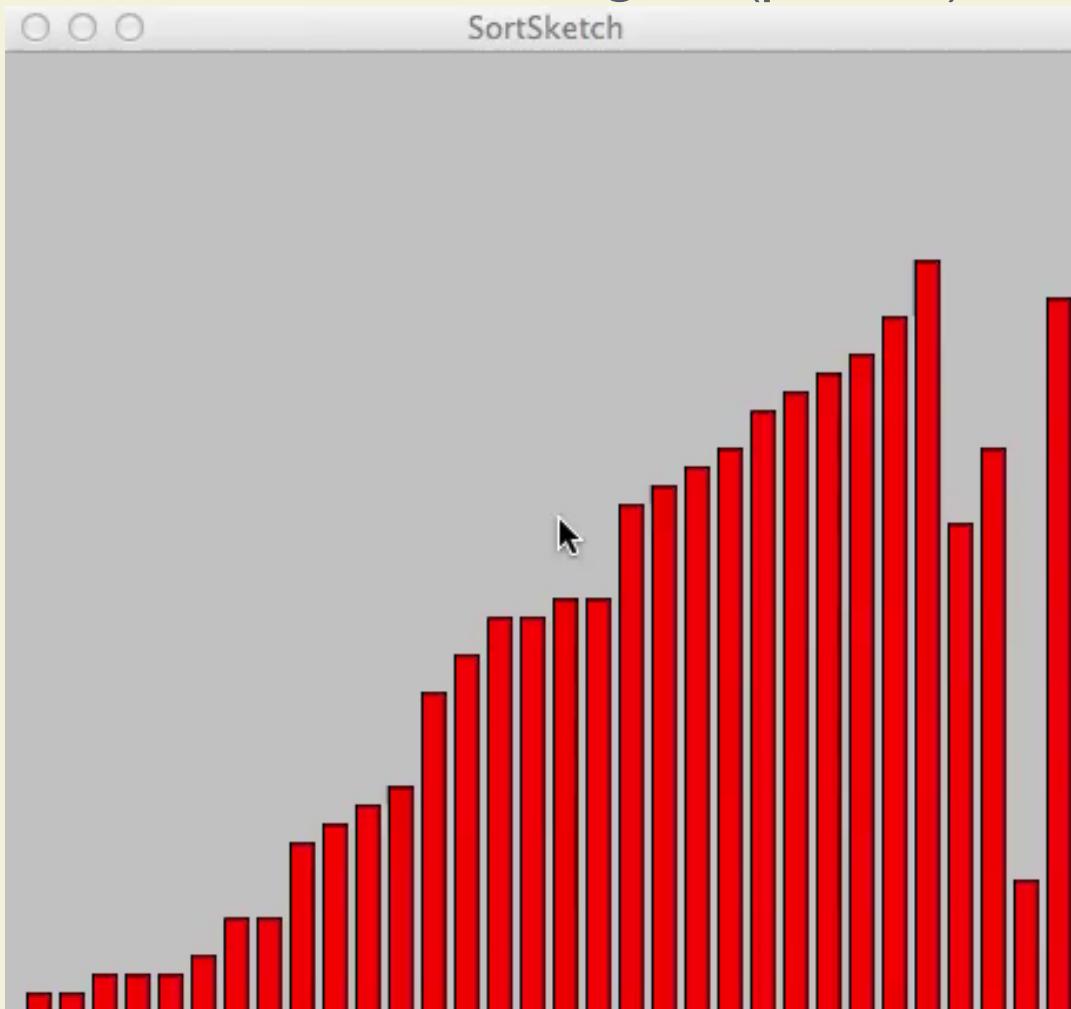
- Neste caso, alguns (poucos) elementos estão fora de ordem:



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/insertionsort_special2.m4v

Filme do insertionsort, especial–3

- Neste caso, foram acrescentados alguns (poucos) elementos a um array ordenado:



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/insertionsort_special3.m4v



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 13

Ordenações subquadráticas

Ordenações subquadráticas

- Shellsort.
- Mergesort.
- Quicksort.



Shellsort

- Inventado por Donald Shell, em 1959.
- Ideia: no insertionsort, cada elemento viaja para a sua posição final de uma em uma posição.
- Seria melhor que viajasse dando saltos maiores...
- Pois bem: modifica-se o insertionsort para no ciclo interior comparar o elemento na posição j com o elemento na posição $j - h$, para um número “milagroso” h .
- Se for caso disso, troca-se o elemento na posição j com o elemento na posição $j - h$.
- Desta forma, os dois elementos trocados dão saltos “grandes”.
- Começa-se com h grandes e vai-se diminuindo até 1.
- Quando h for 1 temos um insertionsort normal, que ordena o que faltar.
- Mas agora, depois das anteriores passagens com valores de h maiores, estamos no caso especial em que os elementos estão perto da sua posição final.
- A arte está em escolher bem a sequência dos h .

Classe Shellsort

- Este é o Shellsort na sua forma mais habitual:

```
public class Shellsort<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        int n = a.length;
        int h = 1;
        while (h < n/3)
            h = 3*h+1;
        while (h >= 1)
        {
            for (int i = h; i < n; i++)
                for (int j = i; j >= h && less(a[j], a[j-h]); j-=h)
                    exchange(a, j, j-h);
            h /= 3;
        }
    }
    ...
}
```

Este ciclo constrói implicitamente a sequência dos **h**. Por exemplo, se **n** for 1000, o valor inicial de **h** é 364. Depois, no segundo ciclo, **h** toma os valores 121, 40, 13, 4, 1 (e 0, que faz parar o ciclo).

Repare: ciclo triplo. Muito difícil!

Shellsort explicado

- Eis uma versão mais entendível, sem o ciclo triplo:

```
public class ShellsortExplained<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        int n = a.length;
        int h = 1;
        while (h < n/3)
            h = 3*h+1;
        while (h >= 1)
        {
            shell(a, n, h);
            h /= 3;
        }
    }

    public void extend(T[] a, int n, T x, int h)
    {
        for (int j = n; j >= h && less(a[j], a[j-h]); j-=h)
            exchange(a, j, j-h);
    }

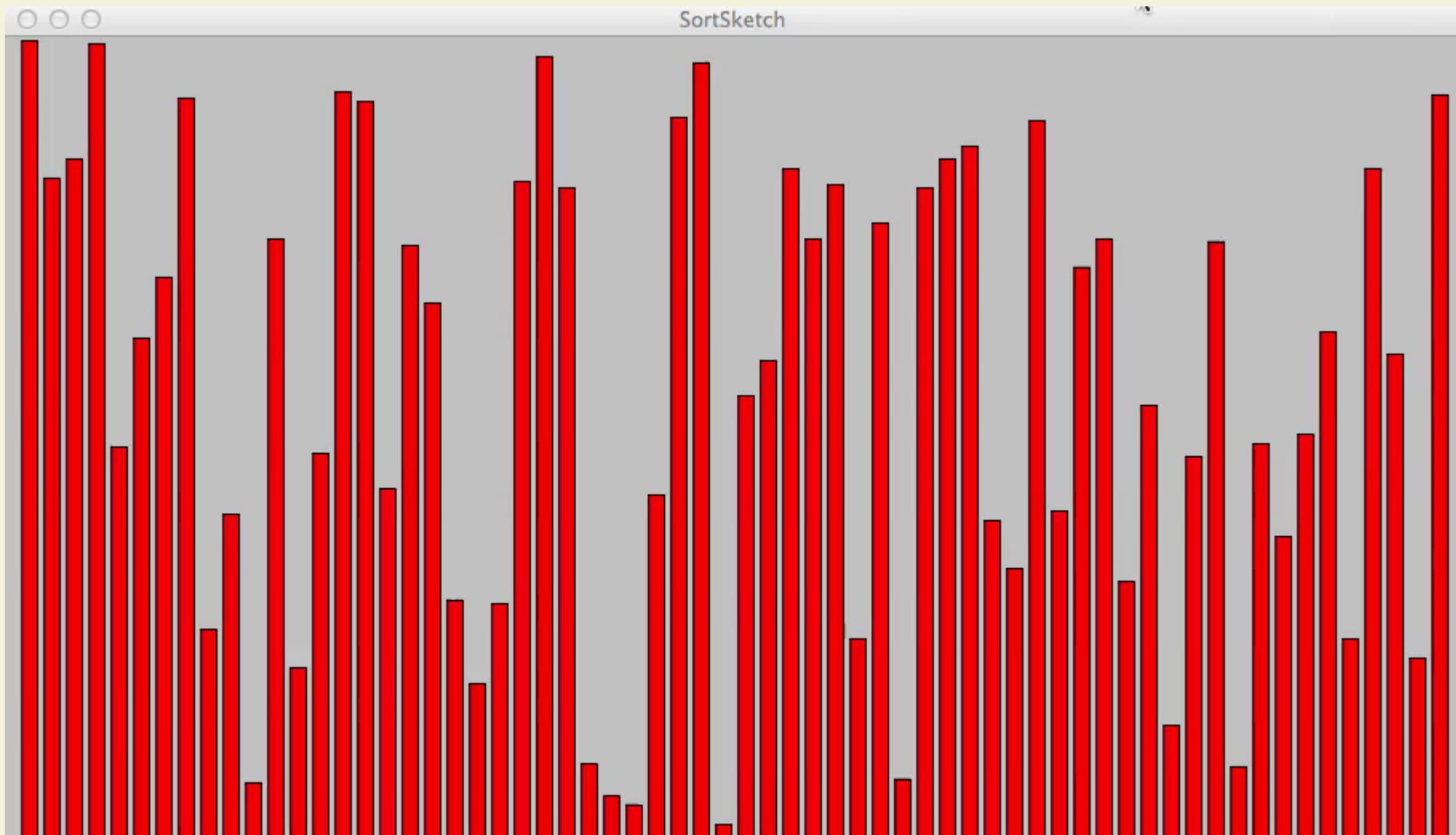
    public void shell(T[] a, int n, int h)
    {
        for (int i = h; i < n; i++)
            extend(a, i, a[i], h);
    }
    ...
}
```

Substituímos os dois ciclos for por uma chamada da função shell.

Esta insere o valor **x** ordenadamente no subarray inicial de **a** com **n** elementos, dando saltos de **h** elementos de cada vez.

Esta repete a anterior para todos os elementos a partir da posição **h**.

Filme do shellsort



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/shellsort.m4v

Ensaio de razão dobrada, Shellsort

- No meu computador:

```
$ java -cp ../../*:. Shellsort C  
10000 0.0025  
20000 0.0057 2.30  
40000 0.0136 2.39  
80000 0.0324 2.38  
160000 0.0813 2.51  
320000 0.2575 3.17  
640000 0.7050 2.74  
1280000 1.6690 2.37  
2560000 4.4150 2.65  
5120000 10.3830 2.35  
$
```

O ensaio de razão dobrada indica que o algoritmo é subquadrático. Knuth mostrou que com a sequência 1, 4, 13, 40, 121, 364, ..., a ordem de crescimento do tempo de execução no pior caso é $N^{3/2}$, o que é compatível com as nossas observações, pois $2^{3/2} = 2.82842712474619$.

A melhoria em relação ao insertionsort é abismal!

```
$ java -cp ../../*:. Insertionsort C  
1000 0.0014  
2000 0.0053 3.88  
4000 0.0215 4.03  
8000 0.0878 4.09  
16000 0.3530 4.02  
32000 1.5850 4.49  
64000 6.8030 4.29  
$
```

Mergesort

- Baseia-se na operação de fusão de arrays ordenados: dados dois arrays ordenados, reunir todos os elementos num único array, também ordenado.
- É um exercício de programação clássico.
- Faz-se numa só passagem por cada um dos arrays.
- Logo, a fusão de arrays ordenados é um algoritmo linear.
- O mergesort aplica a estratégia algorítmica “dividir para reinar” (em inglês, “divide and conquer”; em latim, “divide et impera”).
- Explica-se facilmente, recursivamente:
 - Se o array tiver menos de dois elementos, já está ordenado.
 - Se não, faz-se assim:
 - Ordena-se a primeira metade do array.
 - Ordena-se a segunda metade do array.
 - Fundem-se as duas metades, já ordenadas, assim obtendo um array ordenado.

Aqui se “divide” o problema em subproblemas mais “pequenos”.

Aqui se “reina”, combinando as soluções dos subproblemas pequenos para construir a solução do problema grande.

Mergesort funcional

- Adotemos uma atitude funcional, não modificando os arrays fornecidos, antes criando novos arrays para o resultado das operações.
- A função **merge** toma dois arrays ordenados como argumentos e funde-os no resultado:

```
public class MergesortFunctional<T extends Comparable<T>> extends Sort<T>
{
    public T[] merge(T[] a, T[] b)
    {
        T[] result = Utils.newArrayListLike(a, a.length + b.length);
        int i = 0;
        int j = 0;
        for (int k = 0; k < result.length; k++)
            if (i == a.length)
                result[k] = b[j++];
            else if (j == b.length)
                result[k] = a[i++];
            else
                result[k] = less(a[i], b[j]) ? a[i++] : b[j++];
        return result;
    }
    ...
}
```

Se o primeiro array já chegou ao fim, copia-se do segundo; se o segundo já chegou ao fim, copia-se do primeiro; se nenhum dos dois chegou ao fim, copia-se de aquele cujo elemento corrente for menor.

Note bem: a função cria um novo array para conter o resultado da fusão.

Nota técnica

- A função **merge** precisa de criar um array com elementos do tipo genérico **T**.

- Já sabemos que isso não se pode fazer de maneira limpa em Java.
- Temos ultrapassado a questão usando uma conversão:

```
T[] temp = (T[]) new Object[capacity];
```

- Neste caso, essa técnica não dá, porque, para complicar, **T** é um tipo genérico restrinrido: **T extends Comparable<T>**.
- Temos de usar uma artilharia mais pesada, recorrendo aos mecanismos de introspeção do Java, que permitem consultar os tipos dos objetos em tempo de execução.
- Dessa forma, conseguimos dinamicamente criar um array do mesmo tipo de outro array pré-existente.

```
T[] result = Utils.newArrayListLike(a, a.length + b.length);
```

- A função **newArrayList** faz o trabalho sujo.
- No exemplo, o array **result** fica com elementos do mesmo tipo que o array **a** e com tamanho **a.length + b.length**.

Note bem: esse tipo só é fixado em tempo de execução.

Função newArrayLike

- A função `newInstance`, do classe `Array` (esta do pacote `java.lang.reflect`), cria um array com elementos do tipo dos elementos do array indicado no primeiro argumento e com o tamanho indicado no segundo:

```
public final class utils
{
    ...
    @SuppressWarnings("unchecked")
    public static <T> T[] newArrayLike(T[] a, int n)
    {
        T[] result = (T[]) java.lang.reflect.Array.newInstance(
                        a.getClass().getComponentType(), n);
        return result;
    }
    ...
}
```

- O método `getClass`, da classe `Object`, retorna o tipo do objeto, em tempo de execução (neste caso, array do que for).
- O método `getComponentType`, da classe `Class`, retorna o tipo dos elementos do array.

Mergesort funcional, ordenação

- A ordenação funcional por fusão é estupidamente simples:

```
public class MergesortFunctional<T extends Comparable<T>> extends Sort<T>
{
    public T[] sortf(T[] a)
    {
        return (a.length <= 1) ? a : merge(
            sortf(utils.slice(a, 0, a.length / 2)),
            sortf(utils.slice(a, a.length / 2, a.length - a.length / 2)));
    }
    ...
}
```

Palavras para quê?

- A função utilitária **slice(a, start, n)**, cria um novo array com os **n** elementos a partir de **a[start]: a[start], a[start+1], ..., a[start+n-1]**:

```
public static <T> T[] slice(T[] a, int start, int n)
{
    T[] result = newArrayLike(a, n);
    for (int i = 0; i < n; i++)
        result[i] = a[start+i];
    return result;
}
```

Esta fica na classe **Utils**.

Função sort do mergesort funcional

- Para compatibilidade com as outras classes de ordenação, temos de fornecer também a função **sort** que modifica o array que lhe é passado:

```
public class MergesortFunctional<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        utils.copyFrom(a, sortf(a));
    }
    ...
}
```

- A função utilitária **copyFrom** copia do segundo argumento para o primeiro:

```
public static <T> void copyFrom(T[] a, T[] b)
{
    assert a.length == b.length;
    for (int i = 0; i < a.length; i++)
        a[i] = b[i];
}
```

Note que não estaria bem fazer apenas **a = sortf(a)** pois nesse caso estariámos a mudar a cópia local da referência para o array. O array passado em argumento ficaria na mesma!

Ensaio de razão dobrada, mergesort funcional

- No meu computador:

```
$ java ... MergesortFunctional C  
10000 0.0027  
20000 0.0042 1.60  
40000 0.0088 2.06  
80000 0.0193 2.21  
160000 0.0447 2.31  
320000 0.0905 2.03  
640000 0.2183 2.41  
1280000 0.5040 2.31  
2560000 1.9450 3.86  
5120000 3.8270 1.97  
$
```

Notamos que os quocientes são acima de 2 (quando os tempos deixam de ser minutos) mas abaixo dos do shellsort. Estes valores são um sintoma de complexidade $N \log N$.

O mergesort funcional é mais de duas vezes mais rápido do que Shellsort, para arrays grandes.

```
$ java ... Shellsort C  
10000 0.0025  
20000 0.0057 2.30  
40000 0.0136 2.39  
80000 0.0324 2.38  
160000 0.0813 2.51  
320000 0.2575 3.17  
640000 0.7050 2.74  
1280000 1.6690 2.37  
2560000 4.4150 2.65  
5120000 10.3830 2.35  
$
```

Mergesort tradicional

- Normalmente, queremos fazer o mergesort usando só dois arrays: o array a ordenar, o qual vai receber o resultado de cada uma das fusões, e um array auxiliar, para onde copiamos, justapostas, as duas metades do array, já ordenadas, em cada chamada recursiva, para depois serem fundidas de volta para o array dado.
- Normalmente esse array auxiliar seria declarado na função **merge** e copiariámos para ele o array cujas metades ordenadas queremos fundir, assim, em esquema:

```
public void merge(T[] a, ...)  
{  
    T[] aux = (T[]) Array.newInstance(a.getClass().getComponentType(), ...);  
    ...  
}
```

- No entanto, esta solução envolveria a criação de um novo array em cada chamada recursiva, o que causa um *overhead* que será melhor evitar.

Recorde que, em Java, a criação de um array é uma operação linear, uma vez que todas as referências presentes no array têm de ser anuladas na criação.

Classe Mergesort

- Declaramos o array auxiliar no método **sort** e passamo-lo por argumento às outras funções:

```
public class Mergesort<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        T[] aux = (T[]) Utils.newArrayListLike(a, a.length);
        sort(a, 0, a.length, aux);
    }

    private void sort(T[] a, int start, int n, T[] aux)
    {
        if (n <= 1)
            return;
        sort(a, start, n/2, aux);
        sort(a, start+n/2, n-n/2, aux);
        merge(a, start, n/2, n-n/2, aux);
    }

    ...
}
```

Esta função corresponde à ordenação da fatia de **a** que começa na posição **start** e tem **n** elementos, usando o array **aux**.

Ver a função **merge** na página seguinte.

Função merge

- Esta é a única que trabalha realmente:

```
public void merge(T[] a, int start, int n, int m, T[] aux)
{
    for (int i = 0; i < n+m; i++)
        aux[i] = a[start+i];
    int p = 0;
    int q = n;
    for (int i = 0; i < n+m; i++)
        if (p == n)
            a[start+i] = aux[q++];
        else if (q == n+m)
            a[start+i] = aux[p++];
        else
            a[start+i] = less(aux[p], aux[q]) ? aux[p++] : aux[q++];
}
```

...

Funde o “subarray” de **a** que começa na posição **start** e tem **n** elementos (e que já está ordenado) com o “subarray” que começa na posição **start+n** e tem **m** elementos (e que também já está ordenado). De início copia os dois “subarrays” para o início do array **aux** e depois faz a fusão do array **aux** para o array **a**.

Ensaio de razão dobrada, mergesort

- No meu computador:

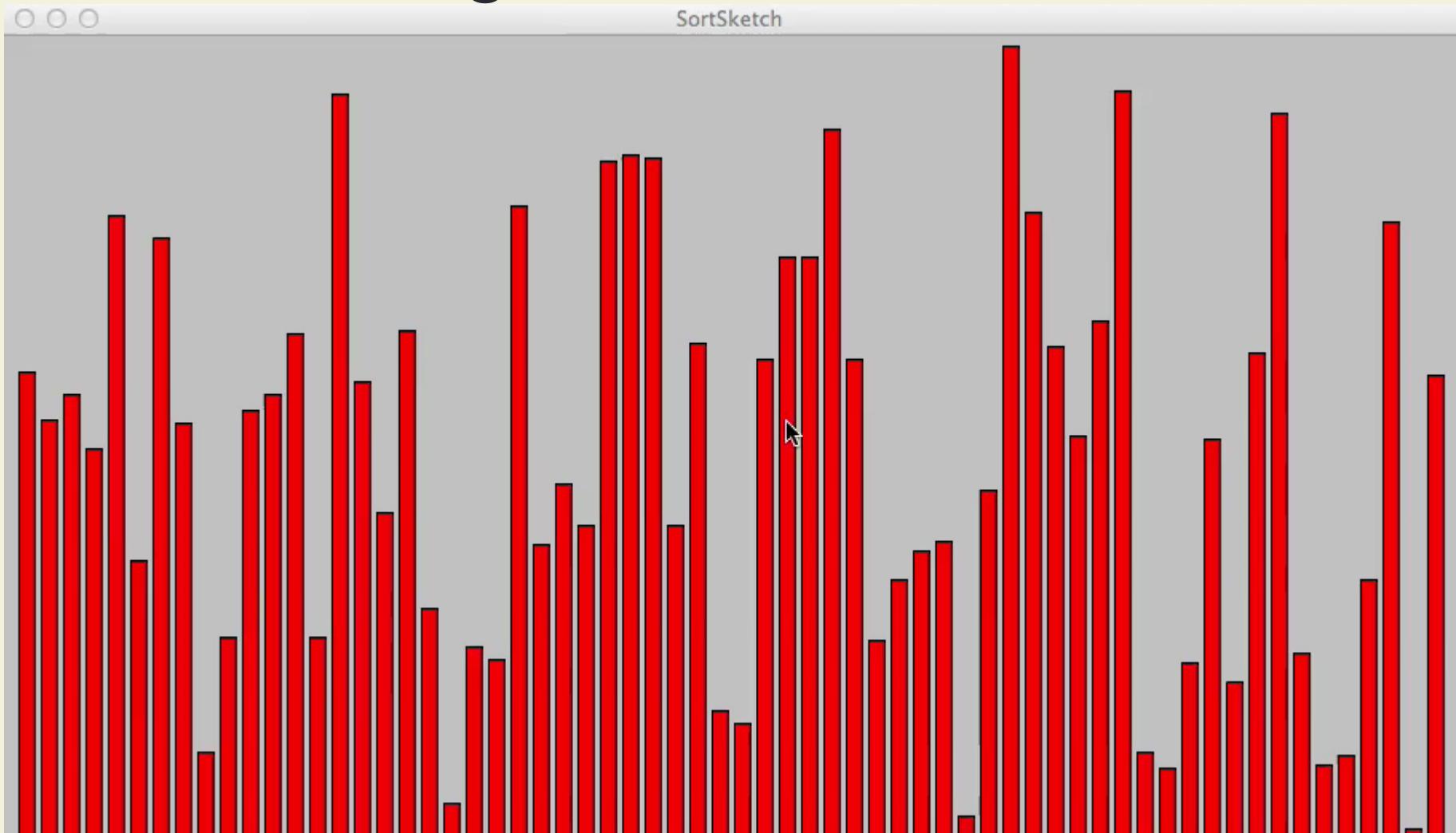
```
$ java ... Mergesort C  
10000 0.0023  
20000 0.0038 1.68  
40000 0.0083 2.16  
80000 0.0183 2.20  
160000 0.0395 2.16  
320000 0.0883 2.24  
640000 0.2023 2.29  
1280000 0.4630 2.29  
2560000 1.0580 2.29  
5120000 2.3960 2.26  
$
```

Estes quocientes parecem mais regulares do que os do mergesort funcional. Talvez seja porque aqui ocorrerá menos a intervenção inesperada da *garbage collection*.

O mergesort “normal” é marginalmente mais rápido do que o mergesort funcional.

```
$ java ... MergesortFunctional C  
10000 0.0027  
20000 0.0042 1.60  
40000 0.0088 2.06  
80000 0.0193 2.21  
160000 0.0447 2.31  
320000 0.0905 2.03  
640000 0.2183 2.41  
1280000 0.5040 2.31  
2560000 1.9450 3.86  
5120000 3.8270 1.97  
$
```

Filme do mergesort



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/mergesort.m4v



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 14

Complexidade da ordenação

Complexidade da ordenação

- Limites da complexidade da ordenação.
- Mergesort bottom-up.



Perguntas metafísicas

- Quantas comparações precisamos de fazer para ordenar um array?
- Ou, alternativamente: perante um array **a**, desconhecido, com **N** elementos, quantas perguntas do estilo “é **a[i]** menor que **a[j]**? ” tenho de fazer para garantidamente descobrir por que ordem estão os elementos de **a**?

```
32, 5, 91, 95, 21, 70, 82, 14
```

Perguntas metafísicas, ainda

- Ou ainda, perante um array **a** que contém uma permutação dos números de 0 a N-1, quantas perguntas como a anterior tenho de fazer para descobrir a permutação presente em **a**?
- Esta pergunta é equivalente à anterior, considerando que o array **a** é o array dos rankings dos elementos do array que queremos ordenar.
- Com efeito, descobrir o array dos rankings dos elementos de um array equivale a ordenar o array.

Array:

32, 5, 91, 95, 21, 70, 82, 14

Array dos rankings:

3, 0, 6, 7, 2, 4, 5, 1

Note que a resposta à pergunta “é $a[i]$ menor que $a[j]$? ” é a mesma para ambos os arrays.

PermutationMind

- Imaginemos um jogo estilo MasterMind, onde em vez de peças coloridas temos N peças, cada uma com um número, de 0 a $N-1$.
- Como no MasterMind, há dois jogadores, A e B.
- O jogador A fica com as peças e dispõe-nas sequencialmente, escondendo-as do jogador B.
- O objetivo do jogador B é descobrir a sequência das peças escondidas.
- Para descobrir, B faz perguntas “está a peça x à esquerda da peça y ?”, às quais A responderá “sim” ou “não”.
- Quantas perguntas tem B de fazer para descobrir?
- Ou, por outras palavras, que sequência de perguntas deve B fazer para minimizar o número de perguntas?



Estratégia força bruta, oito peças

- Uma estratégia vitoriosa consiste em fazer a seguinte sequência de perguntas:
 - Está a peça 0 antes da peça 1?
 - Está a peça 0 antes da peça 2?
 - ...
 - Está a peça 0 antes da peça 7?
 - Está a peça 1 antes da peça 2?
 - Está a peça 1 antes da peça 3?
 - ...
 - Está a peça 1 antes da peça 7?
 - ...
 - Está a peça 6 antes da peça 7?
- Com as respostas, facilmente se estabelece a ordenação, não é?

Por exemplo, o peça x tal que a resposta às perguntas “está a peça x antes da peça y” seja “sim” e a resposta às perguntas “está a peça y antes da peça x?” seja “não”, essa peça é a primeira.

A força bruta é quadrática

- A estratégia da força bruta, com 8 peças usa 28 perguntas e garante a vitória. (Note bem: $28 = 7 + 6 + \dots + 1 + 0 = (8^2 - 8) / 2$).
- Esta é, essencialmente a estratégia do selection sort (e também dos outros algoritmos elementares, cujo número de comparações, é $(N^2-N)/2$, no pior caso).
- Será que existe uma estratégia que garanta a vitória com 27 perguntas? E com 26? E com 25? ... E com 0?
- Com 0 não há certamente. Com 28 já vimos que sim.
- Qual será então o menor número **X** para o qual existe uma estratégia que resolve o problema com **X** perguntas?

Usamos aqui 8 peças, para concretizar a discussão, mas o problema é o mesmo com qualquer número de peças, claro.

PermutationMind computacional

- Juguemos o PermutationMind, mas com ajuda computacional, para o jogador B.
- Antes de formular cada pergunta, o jogador B analisa, com a ajuda do programa, o progresso que fará com cada uma das perguntas válidas, quando a resposta seja sim, e quando a resposta for não.
- O progresso é medido pelo número de configurações ainda possíveis, considerando as respostas dadas às perguntas até à altura.
- Logo, o jogador B pode optar pela estratégia de em cada passo escolher a pergunta que garante a máxima redução no número de configurações possíveis.

Exemplo: PermutationMind com 4 peças

- A configuração secreta é 3, 0, 1, 2, mas o jogador B não sabe.

```
$ java ... Permutations 4
Initial permutations: 24
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 0 2
1 3 2 0
2 0 1 3
2 0 3 1
2 1 0 3
2 1 3 0
2 3 0 1
2 3 1 0
3 0 1 2
3 0 2 1
3 1 0 2
3 1 2 0
3 2 0 1
3 2 1 0
```

```
Questions
0 1 12 12
0 2 12 12
0 3 12 12
1 2 12 12
1 3 12 12
2 3 12 12
-----
->1
0 1
s
Remaining: 12
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 1 2
0 3 2 1
2 0 1 3
2 0 3 1
2 3 0 1
3 0 1 2
3 0 2 1
3 2 0 1
```

Aqui todas as perguntas asseguram o mesmo progresso: 12.

```
Questions
0 1 12 0
0 2 8 4
0 3 8 4
1 2 4 8
1 3 4 8
2 3 6 6
-----
->2
2 3
n
Remaining: 6
0 1 3 2
0 3 1 2
0 3 2 1
3 0 1 2
3 0 2 1
3 2 0 1
```

Aqui, se B escolher 2 3 garante que a seguir restarão 6 permutações.

Note bem: se B tivesse escolhido 0 2, por exemplo, com sorte ficaria com 4 permutações, mas com azar ficaria com 8!

(Continuação)

Questions

0 1 6 0
0 2 5 1
0 3 3 3
1 2 3 3
1 3 1 5
2 3 0 6

Aqui, se B escolher 0 3 ou 1 2 garante que a seguir restarão 3 permutações.

->3

0 3

n

Remaining: 3

3 0 1 2

3 0 2 1

3 2 0 1

Questions

0 1 3 0
0 2 2 1
0 3 0 3
1 2 1 2
1 3 0 3
2 3 0 3

->4

0 2

s

Remaining: 2

3 0 1 2

3 0 2 1

Questions

0 1 2 0
0 2 2 0
0 3 0 2
1 2 1 1
1 3 0 2
2 3 0 2

->5

1 2

s

Remaining: 1

3 0 1 2

Aqui, só faz sentido escolher 1 2, que garante que só restará uma permutação

Está encontrada a configuração inicial!

Note bem: ao escolher 0 2 ou 1 2, o jogador B garante que não restarão mais do que 2 permutações mas, com sorte, restará só uma.

Análise

- Começámos com 24 permutações possíveis.
- Após cada pergunta, o número de permutações restantes diminui.
- Mas, qualquer que fosse a pergunta usada, B nunca conseguiria fazer reduzir o número de permutações para menos de metade.
- Logo, se, por hipótese, B só tivesse direito a 4 perguntas, não haveria uma estratégia ganhadora, pois após 4 perguntas, B só poderíamos garantir duas permutações: $24 \rightarrow 12 \rightarrow 6 \rightarrow 3 \rightarrow 2$.
- Isto não quer dizer que, com sorte, B não pudesse reduzir o número de permutações a 1, com 4 perguntas.
- Mas não era garantido.

Generalização

- Se tivermos N números, haverá $N!$ (fatorial de N) permutações e $N*(N-1)/2$ perguntas diferentes.
- Cada pergunta não pode garantir um redução do número de permutações restantes para menos de metade.
- Portanto, é impossível garantir que a sequência do número de permutações restantes seja melhor (isto é, mais favorável para B) do que $N!, N!/2, N!/4, \dots, 1$.
- Em rigor, aqui o operador \lceil representa o arredondamento para cima do quociente exato.
- Em geral, uma sequência $X, X/2, X/4, \dots, 1$ tem $\lceil \log_2(X) \rceil + 1$ elementos. Por exemplo.: 24, 12, 6, 3, 2, 1 tem 6 elementos, $\log_2 24 = 4.58$, $\lceil 4.58 \rceil = 5$.
- Logo, a sequência $N!, N!/2, N!/4, \dots, 1$ tem $\lceil \log_2(N!) \rceil + 1$ elementos.
- Ou seja, nenhuma estratégia ganhadora no PermutationMind pode ter menos do que $\lceil \log_2(N!) \rceil$ perguntas.

Quanto vale $\log_2(N!)$?

- $\log_2(N!) = \log_2(e) * \ln(N!) = \log_2(e) * (\ln(1) + \ln(2) + \ln(3) + \dots + \ln(N))$
- Aproximamos este somatório com a aproximação de Stirling:

Input interpretation:

Stirling's approximation

Isto é tirado do Wolfram Alpha.

Definition:

More details

Stirling's approximation gives an approximate value for the factorial function $n!$ or the gamma function $\Gamma(n)$ for $n \gg 1$. The approximation can most simply be derived for n an integer by approximating the sum over the terms of the factorial with an integral, so that

$$\begin{aligned}\ln n! &= \ln 1 + \ln 2 + \dots + \ln n \\&= \sum_{k=1}^n \ln k \\&\approx \int_1^n \ln x \, dx \\&= [x \ln x - x]_1^n \\&= n \ln n - n + 1 \\&\approx n \ln n - n.\end{aligned}$$

Concluímos que a soma dos logaritmos é $\approx \log_2(e) * (N * \ln N - N) \approx N * \log_2 N - \log_2(e) * N \approx N * \log_2 N$.

Moral da história

- Não existe nenhuma estratégia ganhadora para o PermutationMind com menos de $\sim N * \log_2 N$ perguntas.
- E também, já que jogar o PermutationMind é essencialmente equivalente a ordenar o array inicial:

Não existe nenhum algoritmo de ordenação baseado exclusivamente em comparações que use menos de $\sim N * \log_2 N$ comparações, para ordenar qualquer array com N elementos.

Por conseguinte, não existe nenhum algoritmo “melhor” que o mergesort!

Dizemos que o mergesort é “ótimo”.

Filme falado do mergesort

- No mergesort “normal” de um array a com 8 elementos, por exemplo, há duas chamadas de sort para subarrays com 4 elementos: $a[0..3]$ e $a[4..7]$; a primeira destas chamadas dá origem a duas chamadas de sort para subarrays com 2 elementos: $a[0..1]$ e $a[2..3]$; a primeira destas chamadas dá origem a duas chamadas para subarrays com 1 elemento: $a[0..0]$ e $a[1..1]$, que retornam logo.
- Segue-se a fusão desses dois arrays com 1 elemento, que fica guardada no subarray $a[0..1]$.
- Depois ocorre a segunda chamada de sort para subarrays com dois elementos, agora para o subarray $a[2..3]$, a qual dá origem a duas chamadas para subarrays de um elemento $a[2..2]$ e $a[3..3]$, as quais retornam logo.
- Segue-se a fusão desses dois arrays com 1 elemento, que fica guardada no subarray $a[2..3]$.
- Segue-se a fusão dos subarrays (já ordenados) $a[0..1]$ e $a[2..3]$, que fica guardada no subarray $a[0..3]$.

Filme falado do mergesort (segunda parte)

- Depois ocorre a segunda chamada de sort para subarrays com 4 elementos, neste caso respeitante ao subarray $a[4..7]$.
- Por um processo análogo ao do subarray $a[0..3]$, essa chamada terminará com um subarray ordenado guardado em $a[4..7]$.
- Segue-se a fusão dos dois subarrays ordenados $a[0..3]$ e $a[4..7]$, ficando o resultado a ocupar o espaço do array inicial e o algoritmo termina.
- Há portanto 4 fusões 1+1, 2 fusões 2+2 e uma fusão 4+4.
- Constatamos que em cada fusão $i+i$ há no mínimo i comparações e no máximo $2*i - 1$ comparações.
- Generalizando, concluímos que começando com um array com n elementos, para cada tipo de fusão o número de comparações é $\sim n$, no pior caso.
- O número de tipos de fusão (1 a 1 , 2 a 2 , etc.) é $\log_2 n$.
- Logo, o número de comparações no pior caso é $\sim n \log_2 n$.

O número mínimo de comparações na fusão ocorre quando o último elemento de um dos subarrays ordenados é menor que o primeiro elemento do outro subarray. O número máximo ocorre quando o último elemento de cada um dos arrays é maior que o penúltimo do outro. Este é o “pior” caso.

Mergesort bottom up

- Retomando o exemplo das páginas anteriores, constatamos que as chamadas recursivas ocorrem pela seguinte ordem: S8, S4, S2, S1, S1, F1+1, S2, S1, S1, F1+1, F2+2, S4, S2, S1, S1, F1+1, S2, S1, S1, F1+1, F2+2, F4+4.
- Ora, em vez de recursivamente ordenar e depois fundir arrays cada vez mais pequenos (com metade do tamanho), começando pelo array original, “de cima para baixo”, podemos proceder “de baixo para cima”, fundindo primeiro, numa só passagem, todos os pares de subarrays com 1 elemento; depois, fundindo noutra passagem todos os pares de subarrays com 2 elementos no array resultantes da fusão 1 a 1, os quais estarão ordenados; depois fundindo numa terceira passagem todos pares de subarrays com 4 elementos no array resultantes da fusão 2 a 2, os quais estarão ordenados; etc.
- Desta maneira “ascendente” conseguimos o mesmo efeito, sem a maçada das chamadas recursivas.

Classe MergesortBottomUp

- É simples mas denso:

```
public class MergesortBottomUp<T extends Comparable<T>>
    extends Mergesort<T>
{
    public void sort(T[] a)
    {
        T[] aux = (T[]) Utilities.newArrayLike(a, a.length);
        int n = a.length;
        for (int z = 1; z < n; z += z)
            for (int i = 0; i < n - z; i += 2*z)
                merge(a, i, z, Math.min(z, n - (i+z)), aux);
    }
    ...
}
```

Note que não há chamadas recursivas.

As fusões são as mesmas que no mergesort top-down, mas realizam-se por outra ordem.

- Fazemos esta classe derivar de Mergesort<T> para poder usar a função **merge** por herança.

Ensaio de razão dobrada, mergesort bottom-up

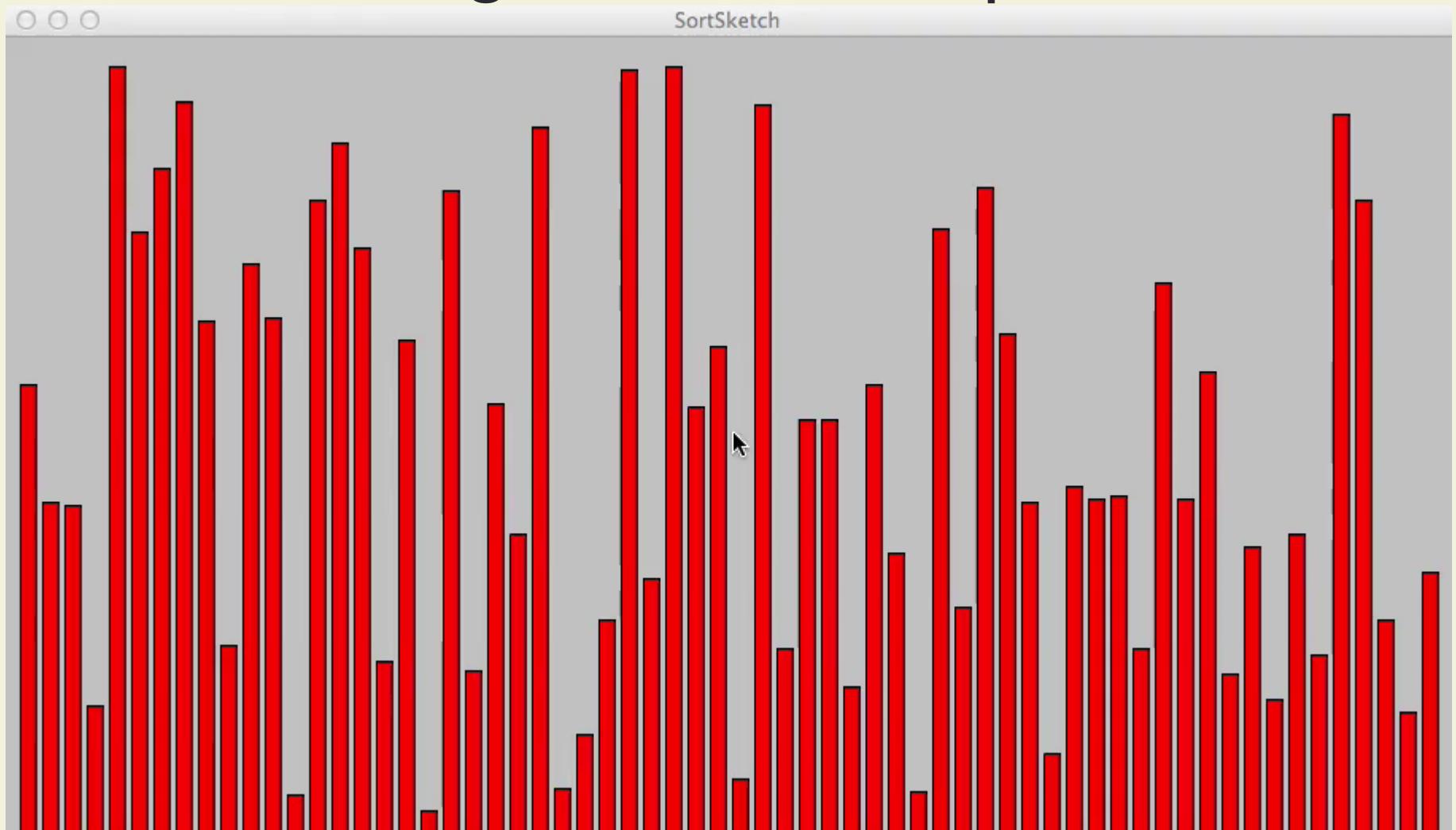
- No meu computador:

```
$ java ... MergesortBottomUp C  
10000 0.0023  
20000 0.0041 1.80  
40000 0.0088 2.18  
80000 0.0191 2.16  
160000 0.0416 2.18  
320000 0.0993 2.39  
640000 0.2333 2.35  
1280000 0.5320 2.28  
2560000 1.2060 2.27  
5120000 2.6940 2.23  
$
```

Os valores são semelhantes aos do mergesort top-down, ainda que, nesta experiência, ligeiramente menos bons.

```
$ java ... Mergesort C  
10000 0.0023  
20000 0.0038 1.68  
40000 0.0083 2.16  
80000 0.0183 2.20  
160000 0.0395 2.16  
320000 0.0883 2.24  
640000 0.2023 2.29  
1280000 0.4630 2.29  
2560000 1.0580 2.29  
5120000 2.3960 2.26  
$
```

Filme do mergesort bottom-up



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/mergesort_bottomup.m4v



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 15
Quicksort

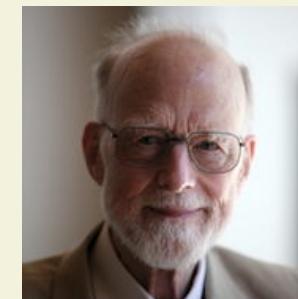
Quicksort

- Quicksort clássico.
- Complexidade do quicksort.
- Vulnerabilidade do quicksort.



Quicksort

- Inventado por [Hoare](#), em 1962.
- Ideia: reorganizar o array, “partindo-o” em dois subarrays justapostos, de tal maneira que cada um dos elementos do primeiro subarray é menor (ou igual) que cada um dos elementos do segundo subarray.
- Depois, reorganizar da mesma maneira, recursivamente, cada um dos subarrays, isto é, “partir” cada um deles em dois subsubarrays justapostos, etc.
- No final, o array estará ordenado.
- A arte está na partição!



Função partition

- Teremos na classe Quicksort<T> uma função “milagrosa”, **partition**, para realizar a partição:

```
public class Quicksort<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        ...
    }

    ...

    public int partition(T[] a, int start, int n)
    {
        ...
    }
}
```

Nesta função, **a** é o array, **start** é o deslocamento do subarray que nos interessa e **n** é o número de elementos desse subarray. Em conjunto, os três argumentos representam o subarray **a[start..start+n-1]**. Sendo **p** o valor retornado pela função, então, após a função, “milagrosamente”, cada um dos elementos do subarray **a[start..start+p-1]** é menor ou igual a cada um dos elementos do subarray **a[start+p..start+n-1]** e nenhum destes subarrays é vazio.

Função quicksort

- Já podemos programar o quicksort, dado o array, a posição inicial do subarray e o número de elementos:

```
public class Quicksort ...  
{  
    ...  
    public void quicksort(T[] a, int start, int n)  
    {  
        int p = partition(a, start, n);  
        if (p > 1)  
            quicksort(a, start, p);  
        if (n - p > 1)  
            quicksort(a, start+p, n-p);  
    }  
    ...
```

Palavras para quê?

Função sort

- Implementamos a função **sort** em termos da função **quicksort**:

```
public class Quicksort ...  
{  
    public void sort(T[] a)  
    {  
        quicksort(a, 0, a.length);  
    }  
    ...  
}
```

A primeira chamada diz respeito ao array todo, claro.

Como se faz a partição?

Há várias técnicas. A seguinte é a partição clássica, usada por Hoare:

1. Selecionamos o elemento de índice médio, chamado pivô.
2. Percorremos o array da esquerda para a direita até encontrar um elemento maior ou igual ao pivô.
3. Percorremos o array da direita para a esquerda até encontrar um elemento menor ou igual ao pivô.
4. Trocamos os elementos selecionados nos passos 2 e 3, se o primeiro ainda estiver à esquerda do segundo, no array.
5. Repetimos a partir de 2, recomeçando os percursos nas posições imediatamente a seguir às dos elementos que foram trocados (no sentido respetivo), até os percursos se cruzarem.

Agora é só programar isto!

Partição de Hoare

```
public class Quicksort ...  
{  
    ...  
    public int partition(T[] a, int start, int n)  
    {  
        int i = start;  
        int j = start + n - 1;  
        T p = a[start+n/2];  
        do  
        {  
            while (less(a[i], p))  
                i++;  
            while (less(p, a[j]))  
                j--;  
            if (i <= j)  
                exchange(a, i++, j--);  
        }  
        while (i <= j);  
        return j+1 - start;  
    }  
    ...
```

Durante a reorganização do subarray $a[\text{start.. start+n-1}]$, invariantemente, todas as posições à esquerda de i têm elementos menores ou iguais ao pivô e todas as posições à direita de j têm elementos maiores ou iguais ao pivô. Logo, no fim do ciclo, quando $j < i$, não existe à direita de j nenhum elemento menor que o pivô, nem existe à esquerda de i nenhum elemento maior que o pivô. Portanto, cada elemento de $a[\text{start.. j}]$ é menor ou igual a cada elemento de $a[j+1.. \text{start+n-1}]$. Logo, o número de elementos do primeiro subarray resultante da partição é $j+1-\text{start}$.

Exemplo

- Observe:

17	4	16	6	2	18	12	0	13	14	8	29	15
----	---	----	---	---	----	----	---	----	----	---	----	----

17	4	16	6	2	18	12	0	13	14	8	29	15
----	---	----	---	---	----	----	---	----	----	---	----	----

8	4	16	6	2	18	12	0	13	14	17	29	15
---	---	----	---	---	----	----	---	----	----	----	----	----

8	4	0	6	2	18	12	16	13	14	17	29	15
---	---	---	---	---	----	----	----	----	----	----	----	----

8	4	0	6	2	12	18	16	13	14	17	29	15
---	---	---	---	---	----	----	----	----	----	----	----	----

A vermelho, o pivô; a azul, os elementos que vão ser trocados.

- Nesta altura, j vale 5 e i vale 6.

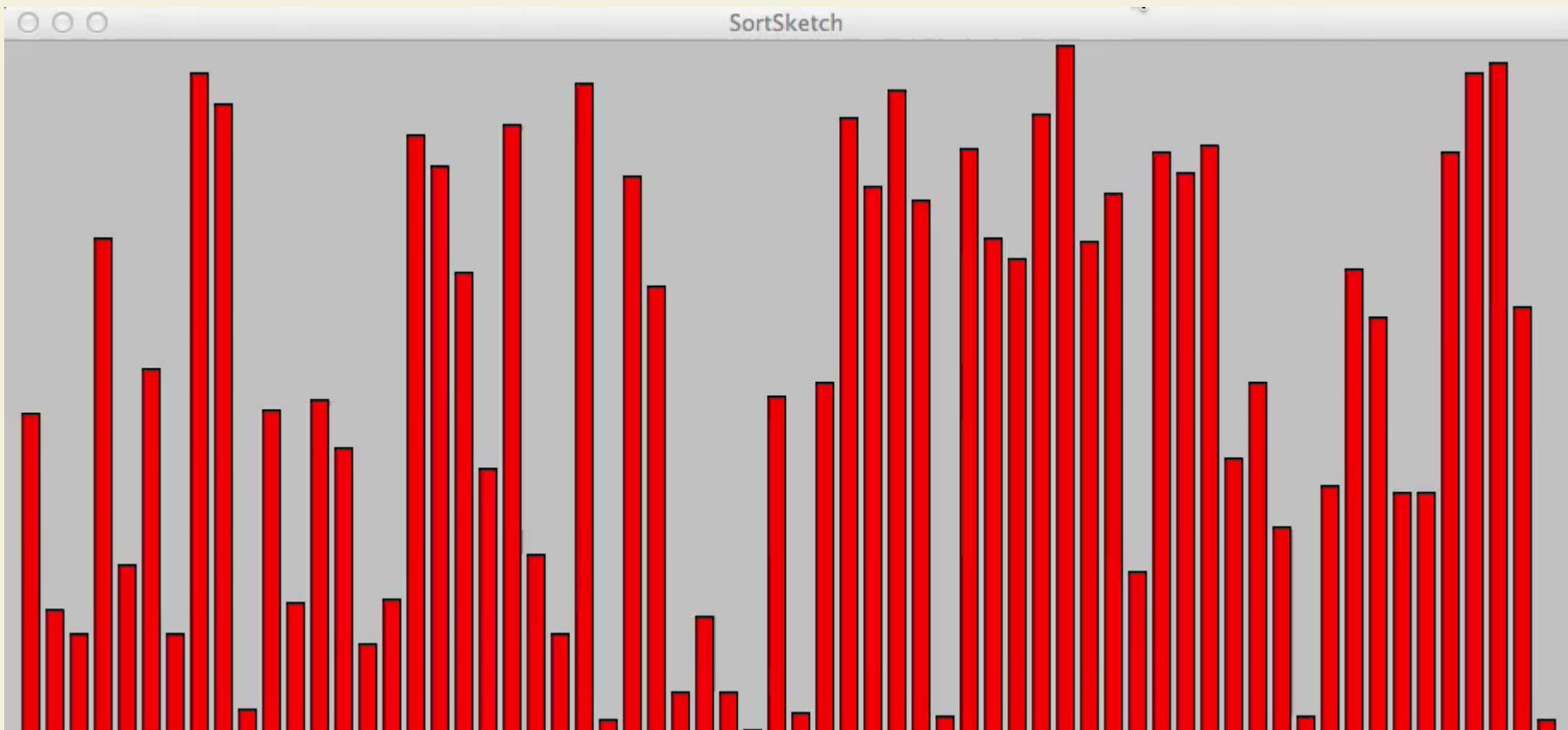
- O primeiro subarray é:

8	4	0	6	2	12
---	---	---	---	---	----

- E o segundo subarray é:

18	16	13	14	17	29	15
----	----	----	----	----	----	----

Filme do quicksort



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/quicksort.m4v

Ensaio de razão dobrada, quicksort

- No meu computador:

```
$ java ... Quicksort C  
10000 0.0018  
20000 0.0035 1.96  
40000 0.0077 2.22  
80000 0.0165 2.15  
160000 0.0364 2.21  
320000 0.0900 2.47  
640000 0.2313 2.57  
1280000 0.5490 2.37  
2560000 1.2410 2.26  
5120000 2.8320 2.28
```

Constatamos que o mergesort é um pouco mais rápido que o quicksort, para arrays de Integer aleatórios.

```
$ java ... Mergesort C  
10000 0.0023  
20000 0.0038 1.68  
40000 0.0083 2.16  
80000 0.0183 2.20  
160000 0.0395 2.16  
320000 0.0883 2.24  
640000 0.2023 2.29  
1280000 0.4630 2.29  
2560000 1.0580 2.29  
5120000 2.3960 2.26  
$
```

Complexidade do quicksort

- O desempenho do quicksort depende de as partições serem equilibradas.
- Seja um array com N elementos:
- Na primeira partição são visitados os N elementos.
- Seguem-se duas chamadas recursivas de primeiro nível, as quais em conjunto visitam todos os N elementos.
- Cada uma delas dá origem a duas chamadas recursivas de segundo nível.
- Há, portanto, quatro chamadas recursivas de segundo nível, as quais em conjunto visitam os N elementos.
- E assim por diante.
- Em cada nível são visitados os N elementos, até que, por os intervalos se tornarem unitários, algumas das chamadas recursivas deixem de se fazer.
- Nessa altura, em cada nível visitam-se menos de N elementos.

Quicksort com sorte

- Num quicksort com sorte, todas as partições seriam perfeitamente equilibradas: em cada chamada recursiva da função quicksort, o intervalo seria dividido ao meio.
- Nesse caso, todas as chamadas recursivas terminam ao mesmo nível (mais ou menos 1).
- Se N fosse uma potência de 2 e partições fossem equilibradas, o número máximo de chamadas recursivas da função quicksort empilhadas na pilha de execução seria $\log_2 N$.
- Logo, com partições equilibradas, o procedimento exige $N \log_2 N$ acessos aos elementos do array.
- Esta é a melhor situação possível: o tempo de execução é proporcional a $N \log N$.

Quicksort com azar

- Num quicksort com azar, a partição é completamente desequilibrada: em cada nível um dos subarrays fica com 1 elemento e o outro com os restantes.
- Neste caso haverá N níveis de recursividade, ainda que em cada nível haja apenas uma chamada recursiva, para a parte não unitária.
- No nível k são visitados $N-k$ elementos.
- Ao todo, o procedimento desequilibrado exige $N + (N-1) + \dots + 1 \sim N^2/2$ acessos ao elementos do array.
- Aqui o desempenho é quadrático!
- Este é o caso mais desfavorável.
- Para arrays grandes, haverá “stack overflow”.

Pivô central

- Escolher para pivô o elemento central tem como vantagem que um array ordenado dá origem a uma partição perfeitamente equilibrada.
- Podemos admitir que acontece por vezes ordenarmos arrays que afinal já estavam ordenados.
- Se tivéssemos escolhido para pivô o primeiro elemento, um array ordenado provocaria uma partição completamente desequilibrada.
- Nestas condições, o pivô central é preferível.
- Mas mesmo com pivô central, há casos patológicos que provocam partições completamente desequilibradas.

Atacando o quicksort

- O quicksort é vulnerável a ataques.
- Isto é, podemos fabricar um array que faça o quicksort “agachar-se”, descambando em comportamento quadrático.
- Considere o seguinte array, com 10 elementos:

```
1 3 5 7 9 10 6 4 8 2
```

- Apesar do seu aspecto inocente, este array é uma “bomba” para o quicksort, pois só dá partições desequilibradas, já que de cada vez o pivô é o maior dos elementos restantes.

Filme do ataque

- Observe:

1 3 5 7 9 **10** 6 4 8 2

1 3 5 7 **9** 2 6 4 8 **10**

1 3 5 7 **8** 2 6 4 **9** 10

1 3 5 **7** 4 2 6 **8** 9 10

1 3 5 **6** 4 2 **7** 8 9 10

1 3 **5** 2 4 6 7 8 9 10

1 3 **4** 2 **5** 6 7 8 9 10

1 **3** 2 4 5 6 7 8 9 10

1 **2** 3 4 5 6 7 8 9 10

De cada vez o **i** avança até ao pivô e o **j** fica com o valor inicial. Segue-se a troca, que atira o pivô para a última posição. No passo seguinte do ciclo exterior da partição, o **i** avança de novo até ao pivô (que agora está à na última posição) e não há mais trocas. O **j** terá ficado na posição à esquerda da final.

Como evitar o ataque?

Evitando o ataque

- Evita-se o ataque escolhendo um pivô que não seja nem o maior elemento do array nem o menor.
- Há várias técnicas para fazer isso.
- A mais prática é baralhar aleatoriamente o array antes de ordenar.
- Assim, fica impossível fabricar bombas deliberadamente.
- A probabilidade de a baralhaçāo aleatória de um array grande produzir uma bomba é ínfima.

Protegendo o quicksort

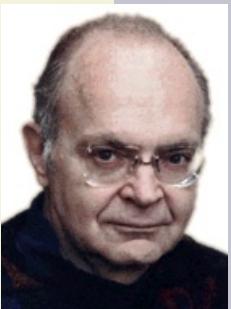
- Recorremos à função **StdRandom.shuffle()** da biblioteca algs4.jar.

```
public class Quicksort ...  
{  
    public void sort(T[] a)  
    {  
        StdRandom.shuffle(a);  
        quicksort(a, 0, a.length);  
    }  
    ...  
}
```

Quer dizer, antes de ordenar, desordena-se!

Baralhação de Knuth

- A função **StdRandom.shuffle()** usa o algoritmo da baralhação de [Knuth](#), que consiste em trocar sucessivamente cada elemento do array com outro escolhido numa posição aleatória daí para a frente.
- Para arrays de ints, está programada assim:



```
public static void shuffle(int[] a)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
    {
        int r = i + uniform(N - i); // between i and N-1
        int temp = a[i];
        a[i] = a[r];
        a[r] = temp;
    }
}
```

Note bem: a baralhação tem complexidade linear.

Este algoritmo baralha e não é enviesado. Quer dizer, gera todas as permutações do array com igual probabilidade.

Curiosidade: quicksort funcional

- Usando a artilharia da nossa classe **ArrayBag**, rapidamente construímos um quicksort funcional, em que a partição se faz à base de filtração, usando como pivô o primeiro elemento do *arraybag* a ser ordenado:

```
public class QuicksortFunctional<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        if (a.length > 1)
            Utils.copyFrom(a, quicksort(new ArrayBag<T>(a)).toArray());
    }

    public ArrayBag<T> quicksort(ArrayBag<T> a)
    {
        ArrayBag<T> result = new ArrayBag<T>();
        if (!a.isEmpty())
        {
            T p = a.get(a.size() / 2);
            ArrayBag<T> aLess = quicksort(a.filter(x -> less(x, p)));
            ArrayBag<T> aEqual = a.filter(x -> equal(x, p));
            ArrayBag<T> aGreater = quicksort(a.filter(x -> greater(x, p)));
            result.add(aLess);
            result.add(aEqual);
            result.add(aGreater);
        }
        return result;
    }
    ...
}
```

Ensaio de razão dobrada, quicksort funcional

- No meu computador:

```
$ java ... QuicksortFunctional C  
10000 0.0125  
20000 0.0169 1.35  
40000 0.0368 2.18  
80000 0.0688 1.87  
160000 0.1478 2.15  
320000 0.3765 2.55  
640000 0.7620 2.02  
1280000 1.7950 2.36  
2560000 4.8460 2.70  
5120000 10.3400 2.13
```

Constatamos que o nosso quicksort funcional é quase quatro vezes mais lento do que o quicksort “clássico”.

```
$ java ... Quicksort C  
10000 0.0018  
20000 0.0035 1.96  
40000 0.0077 2.22  
80000 0.0165 2.15  
160000 0.0364 2.21  
320000 0.0900 2.47  
640000 0.2313 2.57  
1280000 0.5490 2.37  
2560000 1.2410 2.26  
5120000 2.8320 2.28  
$
```



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 16

Quicksorts especializados

Quicksorts especializados

- Quicksort com cutoff.
- Quicksort tripartido.
- Quicksort com duplo pivô.



Quicksort com cutoff

- Como no quicksort há muitas chamadas recursivas para arrays pequenos, poupar-se-á algum trabalho, parando a recursividade antes dessas chamadas.
- Isto é, em vez de:

```
if (p > 1)
    quicksort(a, f, p);
```

teremos:

```
if (p > cutoff)
    quicksort(a, f, p);
```

- Claro que assim o array não ficará ordenado.
- Mas ficará “quase” ordenado.
- Então faz-se uma passagem final com o insertionsort, que é muito eficiente para esse tipo de arrays.

Classe QuicksortCutoff

```
public class QuicksortCutoff<T extends Comparable<T>> extends Quicksort<T>
{
    public static final int CUT_OFF = 5;
    private int cutoff = CUT_OFF;
    private Insertionsort<T> isort = new Insertionsort<T>();

    public QuicksortCutoff(int x)
    {
        cutoff = x;
    }

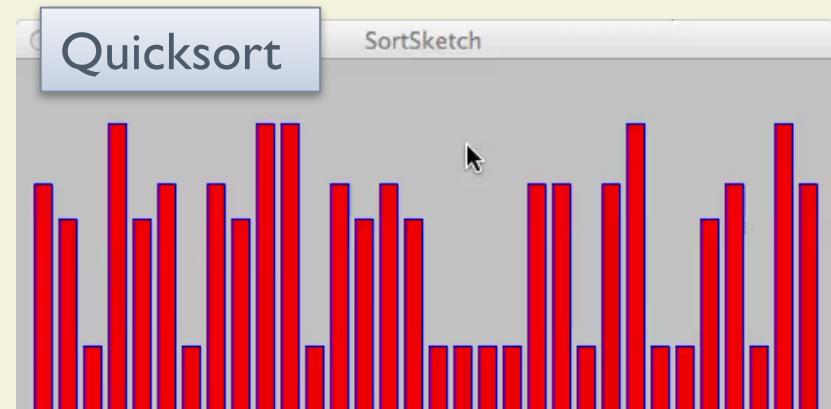
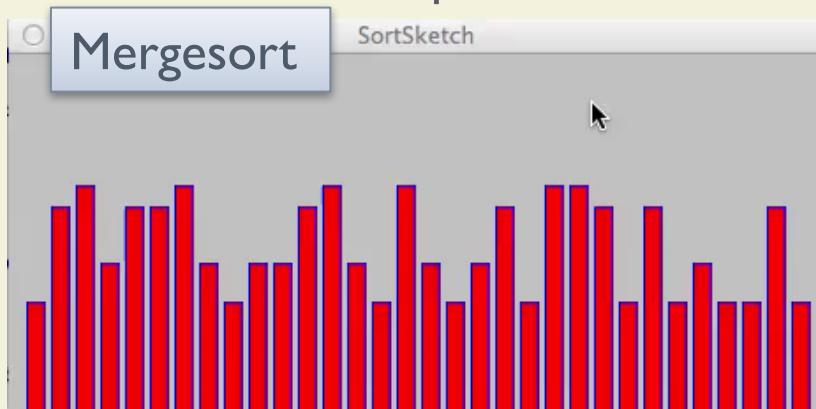
    public void sort(T[] a)
    {
        StdRandom.shuffle(a);
        quicksort(a, 0, a.length);
        isort.sort(a);
    }

    public void quicksort(T[] a, int x, int n)
    {
        int p = partition(a, x, n);
        if (p > cutoff)
            quicksort(a, x, p);
        if (n - p > cutoff)
            quicksort(a, x + p, n - p);
    }
}
```

O valor ideal para o cutoff tem de ser afinado em cada “sistema”.

Caso dos arrays com muitos duplicados

- Na prática, muitas vezes os arrays que queremos ordenar têm muitos duplicados, isto é, elementos iguais, ou, mais geralmente, elementos com a mesma chave de ordenação.
- O mergesort não tira partido dessa circunstância, porque faz sempre a mesma sequência de comparações, independentemente dos valores no array.
- O quicksort também não, porque os elementos iguais ao pivô também são trocados e, portanto, no caso em que há muitos duplicados teremos muitas trocas de elementos iguais, inutilmente.
- Eis dois filmes que ilustram esta situação:



Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/mergesort_duplicates.m4v
e http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/quicksort_duplicates.m4v

Caso dos arrays com dois valores

- De facto, para ordenar um array com dois valores, basta fazer uma partição.
- Em vez da partição de Hoare, podemos fazer um só varrimento, da esquerda para a direita e trocar cada elemento maior que o pivô com o último elemento do array, dos que ainda não foram objeto de troca.
- Veja a demonstração, em que os valores são **verde** e **vermelho** e queremos colocar todos os verdes antes de todos os vermelhos:

Este é o problema da bandeira portuguesa... (Cf. página seguinte.)

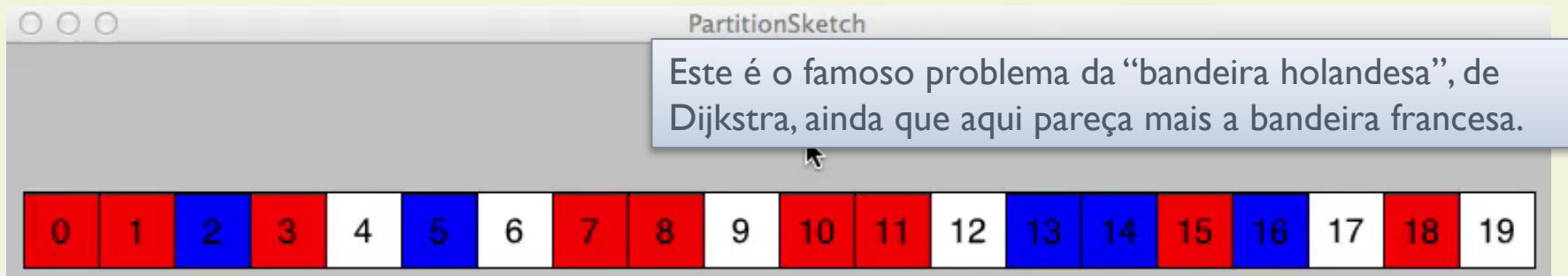
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/partition_2.m4v

Neste caso simples, com dois valores conhecidos (verde e vermelho), é como se fixássemos logo que o pivô era o valor mínimo (neste caso, o verde).

Caso dos arrays com três valores

- Se o array tiver apenas três valores diferentes também conseguimos ordenar numa só passagem.
- Por exemplo, os valores são **azul**, **branco** e **vermelho**, e queremos todos azuis antes de todos os brancos antes de todos os vermelhos.
- Em cada momento, teremos um grupo de azuis, depois um grupo de brancos, depois um grupo de desconhecidos, depois um grupo de vermelhos.
- Varremos o array da esquerda para a direita, observando o primeiro dos desconhecidos:
 - Se for azul, trocamos com o primeiro dos brancos (se houver).
 - Se for branco, deixamos onde está.
 - Se for vermelho, trocamos com o último dos desconhecidos (e não avançamos).



4/6 Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/partition_3.m4v

Quicksort tripartido

- Faz uma partição ternária, análoga à da bandeira holandesa.
- Os azuis são os elementos menores que o pivô; os brancos são os iguais ao pivô; os vermelhos são os maiores que o pivô:

```
public void partitionTri(T[] a, int d, int n)
{
    int i = 1; // number of elements less than pivot, so far, plus 1 for the pivot
    int j = 0; // number of elements equal to pivot, so far;
    int k = 0; // number of elements greater than pivot, so far;
    T p = a[d];
    while (i+j+k < n)
    {
        T x = a[d+i+j];
        if (less(x, p))
        {
            exchange(a, d+i, d+i+j);
            i++;
        }
        else if (less(p, x))
        {
            exchange(a, d+n-1-k, d+i+j);
            k++;
        }
        else
            j++;
    }
    exchange(a, d, d+i-1); // move pivot to its sorted position
    i--;
}
```

Note bem: o pivô é o primeiro elemento, que fica de fora das comparações; no fim será trocado para a posição certa.

Azuis

Vermelhos

Brancos

OK, mas os valores finais de i, j e k têm de ser devolvidos pela função. Como fazer?

Longos, para pares

- A função `partitionTri` tem de devolver os valores finais de `i`, `j` e `k`, para serem usados nas chamadas recursivas subsequentes.
- Na verdade, basta devolver dois deles, por exemplo, `i` e `k`, pois então $j = n - (i+k)$.
- Mas as funções só podem devolver um valor!
- Agrupemos então dois `int` num `long`, e devolvamos um `long`.
- Eis a infraestrutura:

```
public static long pair(int x, int y)
{
    long x1 = x;
    long y1 = y;
    x1 = x1 << 32;
    y1 = y1 & 0xFFFFFFFFL;
    return x1 | y1;
}
```

```
public static int first(long x)
{
    long result = x >> 32;
    return (int) result;
}

public static int second(long x)
{
    return (int) x;
}
```

Partição ternária

- Fica assim, usando a técnica dos longos:

```
public long partitionTri(T[] a, int d, int n)
{
    int i = 1; // ...
    int j = 0; // ...
    int k = 0; // ...
    T p = a[d];
    while (i+j+k < n)
    {
        ...
    }
    exchange(a, d, d+i-1); // ...
    i--;
    return pair(i, k);
}
```

Classe QuicksortTripartite

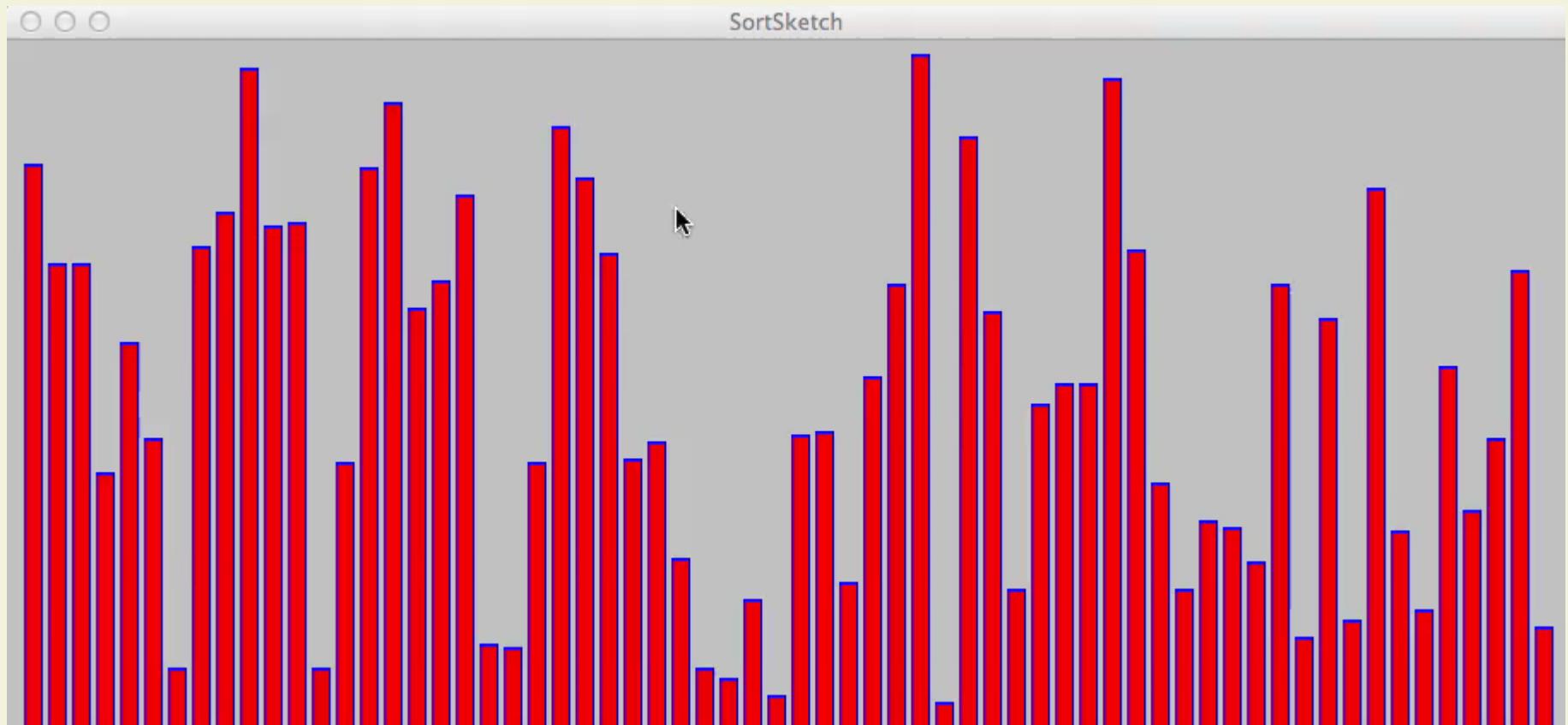
```
public class QuicksortTripartite<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        StdRandom.shuffle(a);
        quicksort(a, 0, a.length);
    }

    public long partitionTri(T[] a, int d, int n, T p)
    {
        ...
    }

    public void quicksort(T[] a, int x, int n)
    {
        long p = partitionTri(a, x, n);
        int n1 = first(p);
        int n2 = second(p);
        if (n1 > 1)
            quicksort(a, x, n1);
        if (n2 > 1)
            quicksort(a, x+n-n2, n2);
    }
    ...
}
```

Baralhação, para evitar ataques!

Filme do quicksort tripartido

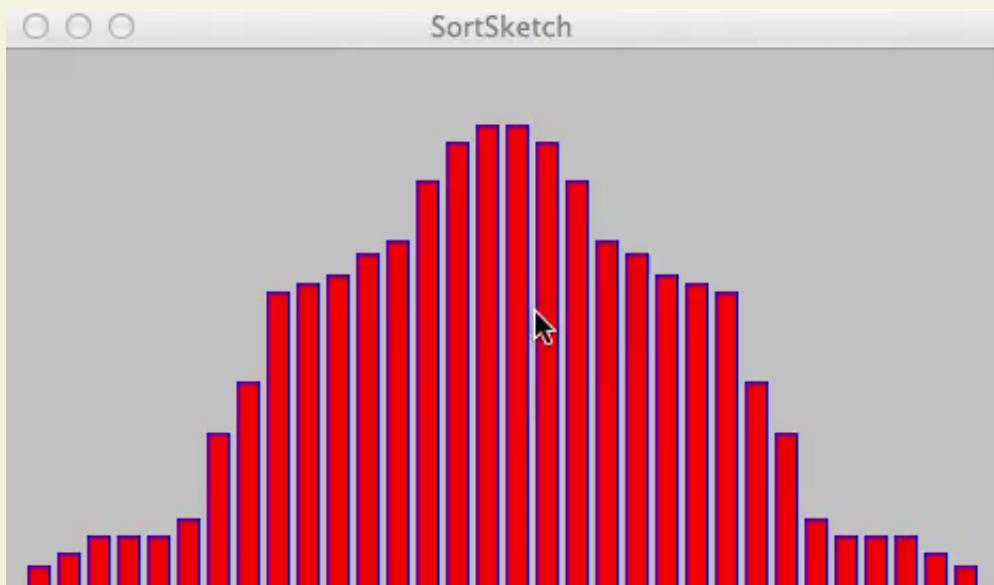


Veja online em http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/quicksort_tripartite.m4v

Atacando o quicksort tripartido

- Se não fosse a baralhação, o quicksort tripartido com pivô central agachar-se-ia com um ficheiro “em órgão de igreja”:
- Observe:

Um ficheiro “em órgão de igreja” é um ficheiro em que a primeira metade é crescente e a segunda metade é “simétrica” da primeira, por exemplo, [1 2 3 4 5 4 3 2 1].



Neste caso, nunca há elementos vermelhos (maiores do que o pivô) e o número de azuis (menores do que o pivô) decresce linearmente, de 2 em 2. Isto configura comportamento quadrático!

Na verdade, no exemplo, há um caso de 6 elementos iguais, que, quando apanhado, faz o intervalo de ordenação decrescer 6 unidades de uma vez.

Veja online em

http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/quicksort_tripartite_organ.m4v

Caso de haver muitos duplicados

- O quicksort tripartido porta-se bem no caso de haver muitos duplicados, pois todos os elementos iguais ao pivô ficam logo arrumados, de uma vez.
- Com efeito, havendo muitos duplicados, haverá muitos elementos iguais ao pivô.
- Observe:



Veja online em

http://w3.ualg.pt/~pjguerreiro/sites/20_aed_1314/movies/quicksort_tripartite_lots_duplicates.m4v

Quicksort com pivô duplo

- É uma variante do tripartido, usando dois pivôs. Assim, os azuis são os menores que o primeiro pivô, os vermelhos os maiores que segundo pivô e os outros são brancos.
- O pivô da esquerda é menor ou igual ao pivô da direita:

```
private long partitionDual(T[] a, int d, int n)
{
    T p = a[d];          // left pivot
    T q = a[d+n-1];     // right pivot
    int i = 1;           // number of elements < p, so far, plus 1 for a[0];
    int j = 0;           // number of elements >= p and <= q, so far;
    int k = 1;           // number of elements > q so far, plus 1 for a[n-1];
    while (i+j+k < n)
    {
        T x = a[d+i+j];
        if (less(x, p))
            ...
        else if (less(q, x))
            ...
        else
            ...
    }
    return pair(i, k);
}
```

The diagram illustrates the partitioning process. Three parallel arrows originate from specific code segments and point to three rectangular boxes. The top box is labeled 'Azuis' (Blues), the middle box is labeled 'Vermelhos' (Reds), and the bottom box is labeled 'Brancos' (Whites). The arrows indicate that elements less than the left pivot (p) are categorized as 'Azuis', elements greater than the right pivot (q) are categorized as 'Vermelhos', and elements between the two pivots or equal to them are categorized as 'Brancos'.

“The devil is in the details”

```
private long partitionDual(T[] a, int d, int n)
{
    if (less(a[d+n-1], a[d]))
        exchange(a, d, d+n-1); Trocamos os candidatos a pivôs, se for preciso.
    T p = a[d]; // left pivot
    T q = a[d+n-1]; // right pivot
    int i = 1; // number of elements < p, so far, plus 1 for a[0];
    int j = 0; // number of elements >= p and <= q, so far;
    int k = 1; // number of elements > q so far, plus 1 for a[n-1];
    while (i+j+k < n) De início, não mexemos nos pivôs. Fica para o fim.
    {
        ...
    }
    // exchange left pivot with last element less than left pivot.
    exchange(a, d, d+i-1);
    i--;
    // j++; OK, but not necessary
    // exchange right pivot with first element greater than right pivot.
    exchange(a, d+n-1, d+n-k);
    k--;
    // j++; OK, but not necessary
    return pair(i, k);
}
```

No fim, trocamos os pivôs, que estavam nas pontas, para as suas posições ordenadas. Note que estes elementos atingiram as suas posições finais e não entrarão em mais operações.

Classe QuicksortDualPivot

```
public class QuicksortDualPivot<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        StdRandom.shuffle(a);
        quicksort(a, 0, a.length);
    }

    public void quicksort(T[] a, int x, int n)
    {
        long k = partitionDual(a, x, n);
        int n1 = first(k);
        int n2 = second(k);
        if (n1 > 1)
            quicksort(a, x, n1);
        // the 2 pivots have reached their final positions
        // a[x+n1] is the left pivot, a[x+n1-n2] is the right pivot
        if ((n-2) - (n1+n2) > 1)
            if (less(a[x+n1], a[x+n1-n2])) quicksort(a, x+n1+1, n - (n1+n2) - 2);
        if (n2 > 1)
            quicksort(a, x+n1-n2, n2);
    }
    ...
}
```

Em geral haverá três chamadas recursivas.

Comparando os quatro quicksorts, pelo tempo

- No meu computador:

```
$ java ... Quicksort C  
10000 0.0017  
20000 0.0034 1.98  
40000 0.0074 2.16  
80000 0.0162 2.18  
160000 0.0357 2.20  
320000 0.0917 2.57  
640000 0.2333 2.55  
1280000 0.5450 2.34  
2560000 1.2370 2.27  
5120000 2.9260 2.37  
$
```

```
$ java ... QuicksortCutoff C  
10000 0.0015  
20000 0.0033 2.17  
40000 0.0074 2.23  
80000 0.0161 2.18  
160000 0.0355 2.21  
320000 0.0902 2.54  
640000 0.2383 2.64  
1280000 0.5700 2.39  
2560000 1.3130 2.30  
5120000 3.0880 2.35  
$
```

```
$ java ... QuicksortTripartite C  
10000 0.0022  
20000 0.0044 2.02  
40000 0.0101 2.29  
80000 0.0222 2.20  
160000 0.0492 2.22  
320000 0.1202 2.44  
640000 0.3310 2.75  
1280000 0.7630 2.31  
2560000 1.7940 2.35  
5120000 4.1690 2.32  
$
```

```
$ java ... QuicksortDualPivot C  
10000 0.0015  
20000 0.0030 2.06  
40000 0.0067 2.20  
80000 0.0145 2.17  
160000 0.0329 2.26  
320000 0.0823 2.50  
640000 0.2183 2.65  
1280000 0.5120 2.35  
2560000 1.2010 2.35  
5120000 2.7070 2.25  
$
```

Note bem, estes ensaios são com arrays aleatórios.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 17

Outros algoritmos de ordenação

Outros algoritmos de ordenação

- Timsort.
- Ordenações de arrays de números.
 - Bucketsort.
 - Countsort.
 - Bitsort.



Timsort

Diz-se que um algoritmo de ordenação é adaptativo se tira partido da existência de troços ordenados.

Diz-se que um algoritmo de ordenação é estável se não troca elemento iguais.

- O Timsort é um mergesort natural, **adaptativo, estável**.
- Para arrays parcialmente ordenados tem comportamento **supernatural**, isto é, faz menos que $\log_2 N!$ comparações.
- Usa uma memória auxiliar de $N/2$, no máximo.
- Foi inventado por Tim Peters, recentemente:
<http://svn.python.org/projects/python/trunk/Objects/listsort.txt>.
- Programado para Java por Joshua Bloch:
<http://cr.openjdk.java.net/~martin/webrevs/openjdk7/timsort/src/share/classes/java/util/TimSort.java.html>.
- É usado desde o Java 7 para ordenar arrays de objetos.
- Ainda não percebi completamente, mas é muito interessante.

Sorts do Java 6

- `public static void sort(int[] a)`
 - The sorting algorithm is a tuned quicksort, adapted from Jon L. Bentley and M. Douglas McIlroy's "Engineering a Sort Function", Software-Practice and Experience, Vol. 23(11) P. 1249-1265 (November 1993). This algorithm offers $n \log(n)$ performance on many data sets that cause other quicksorts to degrade to quadratic performance.
- `public static void sort(Object[] a)`
 - The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed $n \log(n)$ performance.

Sorts do Java 7 e também do Java 8

- `public static void sort(int[] a)`
 - The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers $O(n \log(n))$ performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations.
- `public static void sort(Object[] a)`
 - The implementation was adapted from Tim Peters's list sort for Python (TimSort). It uses techniques from Peter McIlroy's "Optimistic Sorting and Information Theoretic Complexity", in Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, pp 467-474, January 1993.

Introdução ao Timsort

- O Timsort é um mergesort natural.
- Quer dizer que identifica os troços ordenados no array.
- Na verdade, identifica os troços ascendentes e também os troços estritamente descendentes.
- Estes são logo invertidos localmente, tornando-se troços ascendentes.
- A fusão dá-se entre o antepenúltimo e o penúltimo ou entre o penúltimo e últimos troços identificados ou previamente fundidos, de acordo com um critério muito bem apanhado.
- Para fazer a fusão, apenas o mais curto dos dois troços a fundir é copiado para a memória auxiliar.

Copia-se o mais curto porque dá menos trabalho. Depois faz-se a fusão entre o troço que ficou no array e o que foi para a memória auxiliar, deixando o resultado no array. Mas atenção: se o troço mais curto for o primeiro, a fusão faz-se da direita para a esquerda.

Estratégia de fusão

- Sejam A, B e C os três últimos troços no array, “neste momento” e também os respetivos comprimentos.
- Se $A \leq B+C$ então funde-se A e B.
- Se não, se $B \leq C$, então funde-se B e C.
- Se não, vai-se buscar o próximo troço.
- Nos casos em que tenha havido fusão, repete-se a tentativa de fusão.
- Este esquema garante que temos sempre $A > B+C$ e $B > C$, em cada passo do algoritmo.
- Isto implica que a sequência dos tamanhos dos troços, do fim para o princípio cresce mais depressa do que os números de Fibonacci.
- Logo, nunca haverá muitos troços “pendentes”.

Note bem: os comprimentos dos troços são guardados numa pilha.

Por exemplo, $\text{Fibonacci}(47) = 2971215073$, que já é mais do que $2^{31}-1$

Classe MyTimsort, preliminar...

```
public class MyTimsort<T extends Comparable<T>> extends Sort<T>
{
    public void sort(T[] a)
    {
        collapse(a);
    }

    public void collapse(T[] a) // preliminary
    {
        int[] stack = new int[64];
        int r = 0;
        int n = a.length;
        int z = 0;
        while (z < n)          Enquanto houver mais troços...
        {
            int x = runTim(a, z);
            stack[r++] = x;
            z += x;
            if (r >= 3)
                r = mergeCollapse(a, z, stack, r);
        }
        while (r >= 2)          No fim, fundir todos os troços pendentes, do fim para o princípio.
        {
            int b = stack[r - 2];
            int c = stack[r - 1];
            mergeAdjacentRuns(a, n - (b + c), b, c);
            stack[r - 2] += c;
            r--;
        }
    }
    ...
}
```

Função mergeCollapse

- Esta é o cerne da questão:

```
public int mergeCollapse(T[] t, int z, int[] stack, int r)
{
    int result = r;
    int a = stack[r - 3];
    int b = stack[r - 2];
    int c = stack[r - 1];
    if (a <= b + c)
    {
        mergeAdjacentRuns(t, z - (a + b + c), a, b);
        stack[r - 3] += b;
        stack[r - 2] = c;
        result--;
        if (result >= 3)
            result = mergeCollapse(t, z, stack, result);
    }
    else if (b <= c)
    {
        mergeAdjacentRuns(t, z - (b + c), b, c);
        stack[r - 2] += c;
        result--;
        if (result >= 3)
            result = mergeCollapse(t, z, stack, result);
    }
    return result;
}
```

Aqui **t** é o array, **z** é o número de elementos já processados no array, **stack** é a pilha dos comprimentos dos troços pendentes e **r** é o comprimento da pilha.
A função retorna o comprimento da pilha depois das fusões que tiverem ocorrido.

Outras funções

```
// compute length of ascending run starting at x  
// after having reversed the run in case it was a  
// strictly descending run.  
public int runTim(T[] a, int x)  
{  
    ...  
}
```

```
// merge runs a[x..x+n[ and a[x+n..x+n+m[ semi-locally.  
public void mergeAdjacentRuns(T[] a, int x, int n, int m)  
{  
    ...  
}
```

Outras funções surgirão na implementação destas.

Ensaio de razão dobrada, MyTimsort

- No meu computador:

```
$ java ... MyTimsort C  
10000 0.0021  
20000 0.0037 1.75  
40000 0.0079 2.15  
80000 0.0172 2.17  
160000 0.0373 2.17  
320000 0.0816 2.19  
640000 0.1873 2.30  
1280000 0.4325 2.31  
2560000 1.0080 2.33  
5120000 2.5070 2.49
```

Note bem, estes ensaios são com arrays aleatórios.

```
$ java ... Mergesort C  
10000 0.0021  
20000 0.0040 1.88  
40000 0.0087 2.15  
80000 0.0187 2.17  
160000 0.0405 2.16  
320000 0.0910 2.25  
640000 0.2050 2.25  
1280000 0.4685 2.29  
2560000 1.0710 2.29  
5120000 2.6320 2.46
```

Observando a pilha

R: 2
 R: 2 2
 R: 2 2 2
 C: 4 2
 R: 4 2 2
 C: 6 2
 R: 6 2 4
 C: 8 4
 R: 8 4 3
 R: 8 4 3 2
 C: 8 7 2
 C: 15 2
 R: 15 2 2
 C: 15 4
 R: 15 4 3
 R: 15 4 3 3
 C: 15 7 3
 R: 15 7 3 2
 R: 15 7 3 2 3
 C: 15 7 5 3
 C: 15 12 3
 C: 27 3
 R: 27 3 2
 R: 27 3 2 2
 C: 27 5 2
 R: 27 5 2 2
 C: 27 5 4
 R: 27 5 4 3
 C: 27 9 3
 R: 27 9 3 2
 R: 27 9 3 2 2
 C: 27 9 5 2
 R: 27 9 5 2 2
 C: 27 9 5 4
 C: 27 14 4
 R: 27 14 4 3

R: 27 14 4 3 2
 C: 27 14 7 2
 R: 27 14 7 2 2
 C: 27 14 7 4
 R: 27 14 7 4 2
 R: 27 14 7 4 2 2
 C: 27 14 7 6 2
 C: 27 14 13 2
 C: 27 27 2
 C: 54 2
 R: 54 2 2
 C: 54 4
 R: 54 4 2
 R: 54 4 2 2
 C: 54 6 2
 R: 54 6 2 3
 C: 54 6 5
 R: 54 6 5 4
 C: 54 11 4
 R: 54 11 4 2
 R: 54 11 4 2 2
 C: 54 11 6 2
 R: 54 11 6 2 3
 C: 54 11 6 5
 C: 54 17 5
 R: 54 17 5 2
 R: 54 17 5 2 2
 C: 54 17 5 4
 R: 54 17 5 4 2
 C: 54 17 5 4 2
 C: 54 17 9 2
 C: 54 17 9 2 5
 C: 54 17 9 7
 R: 54 17 9 7 4
 C: 54 17 16 4
 C: 54 33 4
 R: 54 33 4 2

R: 54 33 4 2 2
 C: 54 33 6 2
 R: 54 33 6 2 3
 C: 54 33 6 5
 R: 54 33 6 5 4
 C: 54 33 11 4
 R: 54 33 11 4 2
 R: 54 33 11 4 2 2
 C: 54 33 11 6 2
 R: 54 33 11 6 2 2
 C: 54 33 11 6 4
 R: 54 33 11 6 4 3
 C: 54 33 11 10 3
 C: 54 33 21 3
 R: 54 33 21 3 3
 C: 54 33 21 6
 R: 54 33 21 6 4
 R: 54 33 21 6 4 3
 C: 54 33 21 10 3
 R: 54 33 21 10 3 2
 R: 54 33 21 10 3 2 2
 C: 54 33 21 10 5 2
 R: 54 33 21 10 5 2 2
 C: 54 33 21 10 5 4
 R: 54 33 21 10 5 4 2
 C: 54 33 21 10 9 2
 C: 54 33 21 19 2
 C: 54 33 40 2
 C: 54 73 2
 C: 127 2
 R: 127 2 2
 C: 127 4
 R: 127 4 3
 R: 127 4 3 3
 C: 127 7 3
 R: 127 7 3 5

C: 127 10 5
 R: 127 10 5 4
 R: 127 10 5 4 4
 C: 127 10 9 4
 C: 127 19 4
 R: 127 19 4 3
 R: 127 19 4 3 2
 C: 127 19 7 2
 R: 127 19 7 2 2
 C: 127 19 7 4
 R: 127 19 7 4 3
 C: 127 19 11 3
 R: 127 19 11 3 2
 R: 127 19 11 3 2 3
 C: 127 19 11 5 3
 R: 127 19 11 5 3 2
 C: 127 19 11 8 2
 R: 127 19 11 8 2 3
 C: 127 19 11 8 5
 C: 127 19 19 5
 C: 127 38 5
 R: 127 38 5 3
 C: 127 38 8 2
 R: 127 38 8 2 2
 C: 127 38 8 4
 C: 127 38 8 4 2
 R: 127 38 8 4 2 3
 C: 127 38 8 6 3
 C: 127 38 14 3
 R: 127 38 14 3 2
 R: 127 38 14 3 2 3
 C: 127 38 14 5 3
 R: 127 38 14 5 3 3
 C: 127 38 14 8 3
 R: 127 38 14 8 3 2

R: 127 38 14 8 3 2 2
 C: 127 38 14 8 5 2
 R: 127 38 14 8 5 2 2
 C: 127 38 14 8 5 4
 C: 127 38 14 13 4
 C: 127 38 27 4
 R: 127 38 27 4 2
 R: 127 38 27 4 2 2
 C: 127 38 27 6 2
 F: 127 38 27 8
 F: 127 38 35
 F: 127 73
 F: 200

Trata-se de um array aleatório com tamanho 200. O resultado tem 157 linhas.

Ensaio triplo de Timsorts

MyTimsort

10000	0.0021	
20000	0.0037	1.74
40000	0.0079	2.14
80000	0.0171	2.17
160000	0.0373	2.18
320000	0.0833	2.23
640000	0.1880	2.26
1280000	0.4305	2.29
2560000	0.9780	2.27
5120000	2.2210	2.27

Timsort

10000	0.0047	
20000	0.0032	0.69
40000	0.0069	2.15
80000	0.0150	2.19
160000	0.0331	2.20
320000	0.0739	2.23
640000	0.1693	2.29
1280000	0.3785	2.24
2560000	0.8610	2.27
5120000	1.9430	2.26

Arrays.sort

10000	0.0026	
20000	0.0031	1.21
40000	0.0068	2.18
80000	0.0145	2.15
160000	0.0325	2.23
320000	0.0719	2.21
640000	0.1643	2.29
1280000	0.3740	2.28
2560000	0.8480	2.27
5120000	1.9860	2.34

Quicksort duplo pivô,
para referência.

10000	0.0016	
20000	0.0030	1.94
40000	0.0068	2.24
80000	0.0147	2.16
160000	0.0339	2.31
320000	0.0837	2.47
640000	0.2140	2.56
1280000	0.5010	2.34
2560000	1.1680	2.33
5120000	2.7170	2.33

Note bem, estes ensaios são
todos com arrays aleatórios.

TimSort, programado
por Joshua Bloch.

Conclusão

- Os números do TimSort e do Arrays.sort são muito parecidos, o que está de acordo com o facto de o Arrays.sort ser um TimSort.
- O Quicksort com duplo pivô, que é o melhor dos quicksorts, é menos rápido que o Timsort.
- O Timsort é adaptativo, e é mais vantajoso ainda com arrays parcialmente ordenados.
- A diferença observada entre o Timsort e o Quicksort com duplo pivô com arrays aleatórios não é muito significativa e pode ser devida a causas fortuitas (carga da máquina) ou a fatores não algorítmicos (gestão da memória).

Ensaios em Java e em C

Java

size	shell	quick	cutoff	tripart	dual	merge	mergebu	my_tim	bloch	arrays	natural
1000	0.001	0.002	0.002	0.002	0.002	0.003	0.003	0.002	0.002	0.002	0.002
2000	0.000	0.002	0.001	0.001	0.001	0.004	0.002	0.001	0.002	0.004	0.003
4000	0.001	0.001	0.001	0.001	0.001	0.002	0.000	0.003	0.004	0.007	0.006
8000	0.001	0.002	0.001	0.002	0.002	0.003	0.001	0.006	0.007	0.010	0.004
16000	0.003	0.004	0.002	0.003	0.002	0.003	0.002	0.006	0.009	0.002	0.004
32000	0.008	0.006	0.006	0.006	0.006	0.006	0.005	0.007	0.005	0.005	0.009
64000	0.018	0.012	0.012	0.014	0.011	0.012	0.010	0.014	0.011	0.010	0.019
128000	0.043	0.027	0.027	0.030	0.025	0.027	0.022	0.031	0.023	0.023	0.042
256000	0.117	0.065	0.067	0.072	0.059	0.062	0.055	0.068	0.053	0.053	0.105
512000	0.351	0.167	0.174	0.191	0.153	0.147	0.136	0.160	0.123	0.126	0.290
1024000	0.968	0.411	0.435	0.487	0.380	0.348	0.307	0.375	0.291	0.282	0.948
2048000	2.432	0.930	1.007	1.123	0.862	0.777	0.706	0.864	0.650	0.634	1.448

C

size	shell	quick	cutoff	tripart	dual	merge	mergebu	my_tim	qsort
1000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
2000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4000	0.001	0.001	0.000	0.001	0.001	0.001	0.001	0.001	0.000
8000	0.002	0.001	0.001	0.002	0.001	0.001	0.001	0.001	0.001
16000	0.005	0.002	0.002	0.004	0.002	0.003	0.003	0.002	0.002
32000	0.011	0.005	0.005	0.008	0.005	0.007	0.006	0.005	0.005
64000	0.024	0.010	0.010	0.016	0.011	0.014	0.013	0.011	0.010
128000	0.054	0.021	0.020	0.035	0.022	0.030	0.027	0.023	0.021
256000	0.124	0.046	0.042	0.074	0.048	0.063	0.058	0.049	0.044
512000	0.283	0.094	0.088	0.155	0.100	0.133	0.122	0.103	0.092
1024000	0.646	0.196	0.190	0.327	0.211	0.279	0.256	0.216	0.196
2048000	1.460	0.417	0.398	0.705	0.454	0.583	0.541	0.451	0.408
4096000	3.388	0.906	0.862	1.498	0.978	1.216	1.143	0.945	0.843

Note bem: em Java com arrays de Integer; em C com arrays de int.

Epílogo: o bug do Timsort

- Um grupo de cientistas descobriu em 2015 que o Timsort está errado.
- No entanto, o erro só se manifestaria em arrays muito maiores do que os que existem atualmente.
- O erro está na ideia base da fusão:
- Se $A \leq B+C$ então funde-se A e B.
 - Se não, se $B \leq C$, então funde-se B e C.
 - Se não, vai-se buscar o próximo troço.
 - Nos casos em que tenha havido fusão, repete-se a tentativa de fusão.
 - Este esquema garante que temos sempre $A > B+C$ e $B > C$, em cada passo do algoritmo.
- Ora, se os troços forem 120, 80, 25, 20, 30, temos $120 > 80+25$ e $80 > 25$; também $80 > 25+20$ e $25 > 20$; mas $25 \leq 20+30$. Logo, funde-se 25 e 20. Obtém-se 120, 80, 45, 30.
- Isto significa que os comprimentos dos troços podem crescer mais devagar do que se pensava e causar overflow no array dos comprimentos dos troços.
- Ver <http://envisage-project.eu/proving-android-java-and-python-sorting-algorithm-is-broken-and-how-to-fix-it/>.

Ainda outros algoritmos

- Heapsort.
 - Relacionado com as filas com prioridade.
 - É $N \log N$, sempre.
- Introsort.
 - É um quicksort introspetivo: quando nota que a recursividade está a ficar muito profunda, muda para Heapsort.
 - Assim, garante $N \log N$, sempre.

Bucketsort

- Serve para ordenar arrays de números inteiros não negativos, uniformemente distribuídos num intervalo da ordem de grandeza do tamanho do array.
- Nesse caso, tem complexidade linear.
- Usa memória adicional, proporcional ao tamanho do array.
- Não faz comparações, exceto residualmente, na fase final.

Explicação do Bucketsort

- Para ordenar um array de tamanho N , usa-se um array de N bags.
- Seja M o máximo do array, mais 1.
- Percorre-se o array, colocando cada elemento x no saco de índice $x * N / M$.
- Se os elementos estiverem bem distribuídos, cada saco fica com poucos elementos.
- Depois, copiam-se os elementos dos sucessivos sacos de novo para o array.
- O array estará “quase ordenado”.
- O insertionsort completa a operação.

Classe Bucketsort

```
public class Bucketsort extends Sort<Integer>
{
    public void sort(Integer[] a)
    {
        Integer top = max(a)+1;
        int n = a.length;
        Bag<Integer>[] buckets = utilities.newArray(new Bag<Integer>(), n);
        for (int i = 0; i < n; i++)
            buckets[i] = new Bag<Integer>();
        for (int i = 0; i < n; i++)
        {
            long p = (long) a[i] * n;
            int z = (int)(p / top);
            buckets[z].add(a[i]);
        }
        int k = 0;
        for (int i = 0; i < n; i++)
        {
            for (Integer x: buckets[i])
                a[k++] = x;
        }
        new Insertionsort<Integer>().sort(a);
    }
    ...
}
```

Usamos long para evitar overflow com tamanhos grandes.

```
private static int max(Integer[] a)
{
    int result = Integer.MIN_VALUE;
    for (int i = 0; i < a.length; i++)
    {
        if (result < a[i])
            result = a[i];
    }
    return result;
}
```

Countsort

- Serve para ordenar arrays de números inteiros não negativos.
- Tem complexidade linear.
- Não faz comparações.
- Usa um array auxiliar cujo tamanho é igual ao máximo do array a ordenar mais um.
- Este array auxiliar contém, de forma acumulada, o número de ocorrências de cada valor no array a ordenar.
- Internamente, ordena copiando valores do array a ordenar para um segundo array, guiando-se pelas contagens presentes no array auxiliar.
- No fim, copia o array auxiliar para o array original.

Explicação do Countsort

- O countsort baseia-se na seguinte observação:
- Se no array a , desordenado, houver n elementos com valor menor ou igual a $a[k]$, então o elemento $a[k]$ deve ocupar a posição de índice $n-1$ no array auxiliar, isto admitindo que não há no array a outros elementos com o mesmo valor que $a[k]$.
- Se houver um segundo elemento com o valor igual a $a[k]$, então uma cópia do elemento $a[k]$ deve ocupar a posição de índice $n-2$ no array auxiliar.
- E analogamente, se houver mais do que 2.

Classe Countsort

```
public class Countsort extends Sort<Integer>
{
    public void sort(Integer[] a)
    {
        Utilities.copyFrom(a, sorted(a));
    }

    public Integer[] sorted(Integer[] a)
    {
        int n = a.length;
        Integer[] result = new Integer[n];
        int m = max(a) + 1;
        int[] c = new int[m];
        for (int i = 0; i < n; i++)
            c[a[i]]++;
        for (int j = 1; j < m; j++)
            c[j] += c[j-1];
        for (int i = n-1; i >= 0; i--)
            result[c[a[i]]-- -1] = a[i];
        return result;
    }
    ...
}
```

Primeiro, usamos o array c para contar as ocorrências de cada valor do array a. Depois, acumulamos as contagens, usando o mesmo array.

O último ciclo for é deveras subtil. Veja com atenção.

Bitsort

- O bitsort é uma simplificação do countsort, aplicável a arrays sem duplicados.
- Usa um array auxiliar, para registar a ocorrência de cada valor presente no array a ordenar.
- Depois, com uma passagem no array das ocorrências, recolhemos para o array, ordenadamente, os valores dos elementos que ocorrem.

Classe Bitsort

```
public class Bitsort extends Sort<Integer>
{
    public void sort(Integer[] a)
    {
        int n = a.length;
        int m = max(a) + 1;
        int[] c = new int[m];
        for (int i = 0; i < n; i++)
            c[a[i]]++;
        int k = 0;
        for (int j = 0; j < m; j++)
        {
            assert c[j] <= 1;
            if (c[j] == 1)
                a[k++] = j;
        }
    }
    ...
}
```

Registo das ocorrências.

Recolha dos valores.

Observações sobre os algoritmos lineares

- Os valores obtidos em testes de razão dupla não são significativos, pois dependem muitos das características dos arrays e são penalizados por estarmos a usar arrays de objetos.
- Com arrays grandes, a vantagem computacional que se esperaria esbate-se devido ao consumo de memória.
- O consumo de memória é ainda mais violento numa linguagem como Java (e com arrays de objetos).
- E, com arrays pequenos, tanto faz...



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 18

Tabelas

Tabelas

- Tabelas naturais.
- Tabelas sequenciais.
- Tabelas de dispersão.
- Tabelas de encadeamentos separados.
- Tabelas de endereçamento aberto.



Interface Table<K,V>

- Eis uma interface **Table** típica, para tabelas sem chaves duplicadas:

```
public interface Table<K, V> extends Iterable<K>
{
    public V get(K key);
    public void put(K key, V value);
    public void delete(K key);
    public boolean has(K key);
    public boolean isEmpty();
    public int size();
}
```

K é o tipo das chaves;
V é o tipo dos valores.

A função **get** é a função de acesso, por chave; a função **put** acrescenta um par à tabela; a função **delete** remove da tabela o par com a chave dada; a função **has** indica se existe na tabela um par com a chave dada; a função **isEmpty** indica se a tabela está vazia, isto é, se não tem elementos; a função **size** calcula o número de pares na tabela. As classes que implementarem esta interface devem fornecer um método **iterator**, que será o iterador das chaves.

Tabelas naturais, classe NaturalTable<T>

- Nas tabelas naturais, as chaves são números inteiros não negativos.
- Os valores são guardados num array.
- O array será redimensionado quando for preciso.

```
public class NaturalTable<T>
    implements Table<Integer, T>
{
    private static final int DEFAULT_CAPACITY = 16;
    private int size;
    private T[] values;

    ...
}
```

Construtores

- Oferecemos dois construtores: um que permite indicar a capacidade inicial, outro que usa a capacidade fixada por defeito:

```
@SuppressWarnings("unchecked")
public NaturalTable(int capacity)
{
    this.size = 0;
    values = (T[]) new Object[capacity];
}

public NaturalTable()
{
    this(DEFAULT_CAPACITY);
}
```

Função get

- A função **get** usa a chave passada em argumento como índice no array, se der; se não der, retorna **null**, significando que não existe valor para aquela chave:

```
public T get(Integer key)
{
    assert key >= 0;
    T result = null;
    if (key < values.length)
        result = values[key];
    return result;
}
```

Note bem: qualquer argumento não negativo é válido. Pelo contrário, um argumento negativo é inválido sempre e causará uma exceção. Cabe à função que chama garantir que o argumento não é negativo.

Função put

- A função **put** guarda o valor na posição do array especificada pela chave, fazendo o array crescer, se necessário.
- Também atualiza a contagem:

```
public void put(Integer key, T value)
{
    assert key >= 0;
    if (key >= values.length)
        resize(Math.max(2 * values.length, key+1));
    if (values[key] == null)
        size++;
    values[key] = value;
}
```

Se o valor da chave for demasiado grande haverá uma exceção, por falta de memória.

Função delete

- A função **delete** coloca **null** na posição indicada pela chave e atualiza a contagem, se necessário:

```
public void delete(Integer key)
{
    assert key >= 0;
    if (key < values.length && values[key] != null)
    {
        size--;
        values[key] = null;
    }
}
```

Note que a função **delete** não faz o array encolher.

Função has, size e isEmpty

- Estas três são muito simples:

```
public boolean has(Integer key)
{
    return get(key) != null;
}

public int size()
{
    return size;
}

public boolean isEmpty()
{
    return size == 0;
}
```

O iterador das chaves

Usamos a técnica do costume, mas observe a utilização da função **advance**:

```
public Iterator<Integer> iterator()
{
    return new Keys();
}
```

```
private class Keys implements Iterator<Integer>
{
    private int i;

    public Keys()
    {
        i = advance(0);
    }

    private int advance(int x)
    {
        int result = x;
        while (result < values.length && values[result] == null)
            result++;
        return result;
    }

    public boolean hasNext()
    {
        return i < values.length;
    }

    public Integer next()
    {
        int result = i++;
        i = advance(i);
        return result;
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}
```

Função `toString`

- Definimos a função `toString`, para depois podermos observar a tabela, nos testes:

```
public String toString()
{
    String result = "";
    for (int i = 0; i < values.length; i++)
        if (values[i] != null)
            result += i + ": " + values[i] + newLine;
    result += "capacity: " + values.length;
    return result;
}
```

A função mostra as chaves e os valores presentes e no fim indica a capacidade do array.

Função de teste

```
$ java ... NaturalTable 10
p 4 uuu
p 7 fff
s
4: uuu
7: fff
capacity: 10
size: 2
isEmpty? false
p 100 aaa
s
4: uuu
7: fff
100: aaa
capacity: 101
size: 3
isEmpty? false
d 100
p 40 hhh
i
 4 7 40
g 30
null
g 1000
null
g 40
hhh
s
4: uuu
7: fff
40: hhh
capacity: 101
size: 3
isEmpty? false
```

```
public static void testNaturalTable(int d)
{
    NaturalTable<String> table = new NaturalTable<String>(d);
    while (!StdIn.isEmpty())
    {
        String cmd = StdIn.readString();
        if ("p".equals(cmd))
        {
            int x = StdIn.readInt();
            String s = StdIn.readString();
            table.put(x, s);
        }
        else if ("g".equals(cmd))
        {
            int x = StdIn.readInt();
            String s = table.get(x);
            StdOut.println(s);
        }
        else if ("d".equals(cmd))
        {
            int x = StdIn.readInt();
            table.delete(x);
        }
        else if ("i".equals(cmd))
        {
            for (int x : table)
                StdOut.print(" " + x);
            StdOut.println();
        }
        else if ("s".equals(cmd))
        {
            StdOut.println(table);
            StdOut.println("size: " + table.size());
            StdOut.println("isEmpty? " + table.isEmpty());
        }
    }
}
```

Tabelas sequenciais

- Nas tabelas sequenciais, os pares chave-valor são guardados numa lista.
- Na nossa implementação, cada novo par é colocado à cabeça da lista.
- A memória é usada dinamicamente, à justa.
- Inserções e buscas têm complexidade linear.
- Logo, não é recomendável que estas tabelas sejam muito grandes.
- Usaremos listas **imutáveis**.

Classe List<T>

- List<T> é a nossa classe de listas imutáveis, inspirada nas listas da programação funcional.
- List<T> é uma classe abstrata, da qual derivam as classes Empty<T> e Cons<T>.
- A classe Empty<T> representa listas vazias.
- A classe Cons<T> representa listas não vazias.

```
public abstract class List<T>
{
    ...
}

class Empty<T> extends List<T>
{
    ...
}

class Cons<T> extends List<T>
{
    ...
}
```

Para não atravancar a nossa diretoria **src**, colocamos as três classes no mesmo ficheiro .java.

Funções abstratas

- Na classe **List<T>** haverá funções abstratas, que serão implementadas nas classes “de baixo” e funções não abstratas, programadas em termos das funções abstratas:

```
public abstract class List<T>
{
    public abstract List<T> cons(T x);
    public abstract T head();
    public abstract List<T> tail();
    public abstract boolean isEmpty();
    public abstract int size();

    public abstract String mkString(String separator);

    public String toString()
    {
        return "[" + mkString(",") + "]";
    }

    ...
}
```

A função **cons** constrói uma nova lista, em que o primeiro elemento vale **x** e o resto é a lista objeto; a função **head** devolve o objeto que está na primeira posição; a função **tail** devolve a lista formada por todos os elementos da lista objeto, menos o primeiro.

A função **mkString** constrói uma cadeia forma pelas representações textuais dos elementos da lista separadas pelo separador indicado. A função **toString** não é abstrata e fica logo programada

Classe Empty<T>

- Observe:

Claro: o número de elementos de uma lista vazia é zero; não tem cabeça nem cauda; acrescentando um elemento, cria-se uma lista não vazia que tem só esse elemento; e é verdade que é vazia!

```
class Empty<T> extends List<T>
{
    public int size()
    {
        return 0;
    }

    public T head()
    {
        throw new NoSuchElementException();
    }

    public List<T> tail()
    {
        throw new NoSuchElementException();
    }

    public List<T> cons(T x)
    {
        return new Cons<T>(x, this);
    }

    public boolean isEmpty()
    {
        return true;
    }

    ...
}
```

Classe Cons<T>

- Esta é que trabalha verdadeiramente:

```
class Cons<T> extends List<T>
{
    private final T value;
    private final List<T> next;

    public Cons(T x, List<T> w)
    {
        value = x;
        next = w;
    }

    ...
}
```

O membro **value**, de tipo **T**, representa o valor do primeiro elemento da lista, e o membro **next**, de tipo **List<T>**, representa o resto da lista.

Funções size, head, tail, isEmpty, cons

- Também são simples:

```
public int size()
{
    return 1 + next.size();
}

public T head()
{
    return value;
}

public List<T> tail()
{
    return next;
}

public boolean isEmpty()
{
    return false;
}

public List<T> cons(T x)
{
    return new Cons<x>(x, this);
}
```

Repare nesta função recursiva: quando **tail()** for a lista vazia, a função **size()** chamada será a da classe **Empty<T>**, retornando 0 e terminando a recursividade.

Uma vez que a programação da função **cons** é a mesma das duas classes “de baixo”, podíamos tê-la colocado na classe de base.

Função mkString

- Na classe Empty<T>:

```
public String mkString(String separator)
{
    return "";
}
```

- Na classe Cons<T>:

```
public String mkString(String separator)
{
    String result = value.toString();
    if (!next.isEmpty())
        result += separator + next. mkString(separator);
    return result;
}
```

Funções has e first

- Na classe List<T>:

```
public abstract boolean has(T x);  
public abstract List<T> first(T x);
```

- Na classe Empty<T>:

```
public boolean has(T x)  
{  
    return false;  
}  
  
public List<T> first(T x)  
{  
    return this;  
}
```

A função **has** dá true se existir na lista um elemento com o valor indicado ou, melhor, um elemento que seja igual ao argumento (no sentido da função **equals**).

A função **has** dá a sublista que começa na primeira ocorrência de um elemento que seja igual ao argumento, ou uma lista vazia, se nenhum dos elementos for igual ao argumento.

- Na classe Cons<T>:

```
public boolean has(T x)  
{  
    return value.equals(x) || next.has(x);  
}  
  
public List<T> first(T x)  
{  
    return value.equals(x) ? this : next.first(x);  
}
```

Funções delete

- Esta é mais subtil.
- Na classe List<T>:

```
public abstract List<T> delete(T x);
```

- Na classe Empty<T>:

```
public List<T> delete(T x)
{
    return this;
}
```

- Na classe Cons<T>:

```
public List<T> delete(T x)
{
    return value.equals(x) ? next : next.delete(x).cons(value);
}
```

A função **delete** apaga a primeira ocorrência de uma elemento que é igual ao argumento ou, melhor, cria uma nova lista com todos os elementos da lista original menos esse que foi apagado.

Repare que, no limite, ao apagar um elemento que não existe, a função **delete** constrói uma lista nova, igual à lista objeto.

Exercício: função append

- Na classe List<T>:

```
public abstract List<T> append(List<T> other);
```

- Na classe Empty<T>:

```
public List<T> append(List<T> other)
{
    return other;
}
```

- Na classe Cons<T>:

```
public List<T> append(List<T> other)
{
    return next.append(other).cons(value);
}
```

Interface Iterable<T>

- Convém que a lista tenha um iterador:

```
public abstract class List<T> implements Iterable<T>
{
    ...
}
```

- As classes “de baixo” devem fornecer a função **iterator**, a qual devolve um objeto de uma classe privada:

```
...
private class Items implements Iterator<T>
{
    ...
}

public Iterator<T> iterator()
{
    return new Items();
}
...
```

Classes privadas, para o iterador

- Na classe Empty<T>:

```
private class Items implements Iterator<T>
{
    public boolean hasNext()
    {
        return false;
    }

    public T next()
    {
        throw new UnsupportedOperationException();
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}
```

- Na classe Cons<T>:

```
private Cons<T> self()
{
    return this;
}

private class Items implements Iterator<T>
{
    private List<T> current = self();

    public boolean hasNext()
    {
        return !current.isEmpty();
    }

    public T next()
    {
        T result = current.head();
        current = current.tail();
        return result;
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}
```

Repare na técnica usada para nos referirmos ao objeto da classe exterior na classe interna, por meio da função privada **self**.

Funções de teste

```
public static void testListInteger()
{
    List<Integer> w = new Empty<Integer>();
    while (!StdIn.isEmpty())
    {
        String cmd = StdIn.readString();
        if ("p".equals(cmd))
        {
            int x = StdIn.readInt();
            w = w.cons(x);
        }
        else if ("d".equals(cmd))
        {
            int x = StdIn.readInt();
            w = w.delete(x);
        }
        else if ("h".equals(cmd))
        {
            int x = StdIn.readInt();
            boolean b = w.has(x);
            StdOut.println(b);
        }
        else if ("i".equals(cmd))
        {
            for (int x : w)
                StdOut.print(" " + x);
            StdOut.println();
        }
        else if ("s".equals(cmd))
        {
            StdOut.println(w);
            StdOut.println("size: " + w.size());
            StdOut.println("isEmpty? " + w.isEmpty());
        }
    }
}
```

} 4/18/17

```
public static void testListString()
{
    List<String> w = new Empty<String>();
    while (!StdIn.isEmpty())
    {
        String cmd = StdIn.readString();
        if ("p".equals(cmd))
        {
            String x = StdIn.readString();
            w = w.cons(x);
        }
        else if ("d".equals(cmd))
        {
            String x = StdIn.readString();
            w = w.delete(x);
        }
        else if ("h".equals(cmd))
        {
            String x = StdIn.readString();
            boolean b = w.has(x);
            StdOut.println(b);
        }
        else if ("i".equals(cmd))
        {
            for (String x : w)
                StdOut.print(" " + x);
            StdOut.println();
        }
        else if ("s".equals(cmd))
        {
            StdOut.println(w);
            StdOut.println("size: " + w.size());
            StdOut.println("isEmpty? " + w.isEmpty());
        }
    }
}
```

Algoritmos e Estruturas de Dados

25

Pares

- Para a lista de pares das tabelas sequenciais, precisamos de pares:

```
public final class Pair<T, U>
{
    public final T _1;
    public final U _2;

    public Pair(final T x, final U y)
    {
        _1 = x;
        _2 = y;
    }

    public static <A, B> Pair<A, B> of(final A x, final B y)
    {
        return new Pair<A, B>(x, y);
    }

    public String toString()
    {
        return "<" + _1 + "," + _2 + ">";
    }
}
```

Classe SequentialTable<K,V>

- Tem um membro de dados para a lista de pares e um para o tamanho:

```
public class SequentialTable<K, V> implements Table<K, V>
{
    private List<Pair<K, V>> items = new Empty<>();
    private int size = 0;

    ...
}
```

Funções com argumentos funcionais

- Convém enriquecer a classe das listas com funções com argumentos funcionais, para mais generalidade no acesso os elementos da lista:

```
public abstract class List<T> implements Iterable<T>
{
    // ...
    public abstract boolean has(Predicate<T> p);
    public abstract List<T> first(Predicate<T> p);
    public abstract List<T> delete(Predicate<T> p);
    // ...
}
```

- A função **has** dá true se existir um elemento na lista que satisfaça o predicado; a função **first** dá a lista que começa na primeira ocorrência de um elemento que satisfaça o predicado; a função **delete** dá a lista que resulta de remover a primeira ocorrência de um elemento que satisfaça o predicado.

Funções get, has

- Observe, com atenção:

```
public V get(K key)
{
    V result = null;
    List<Pair<K, V>> z = items.first(x -> key.equals(x._1));
    if (!z.isEmpty())
        result = z.head()._2;
    return result;
}
```

```
public boolean has(K key)
{
    return items.has(x -> key.equals(x._1));
}
```

Funções put, delete

- Observe:

```
public void put(K key, V value)
{
    this.delete(key);
    items = items.cons(Pair.of(key, value));
    size++;
}
```

Para evitar duplicados, antes de acrescentar, a função **put** apaga o par que tem a chave dada, se houver.

```
public void delete(K key)
{
    if (items.has(x -> x._1 == key))
    {
        items = items.delete(x -> key.equals(x._1));
        size--;
    }
}
```

Na função **delete**, se a chave não existir, fica tudo na mesma.

Funções size, isEmpty e toString

- Estas são triviais:

```
public int size()
{
    return size;
}

public boolean isEmpty()
{
    return size == 0;
}

public String toString()
{
    return items.toString();
}
```

O iterator das chaves

- Temos de “adaptar” o iterador da lista, para obter apenas as chaves:

```
private class Keys implements Iterator<K>
{
    private Iterator<Pair<K, V>> it = items.iterator();

    public boolean hasNext()
    {
        return it.hasNext();
    }

    public K next()
    {
        return it.next()._1;
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}

public Iterator<K> iterator()
{
    return new Keys();
}
```

Função de teste

- É análoga às anteriores:

```
public static void testSequentialTable()
{
    SequentialTable<Integer, String> t =
        new SequentialTable<>();
    while (!StdIn.isEmpty())
    {
        String cmd = StdIn.readString();
        if ("p".equals(cmd))
        {
            int x = StdIn.readInt();
            String s = StdIn.readString();
            t.put(x, s);
        }
        else if ("g".equals(cmd))
        {
            int x = StdIn.readInt();
            String s = t.get(x);
            StdOut.println(s);
        }
        else if ("d".equals(cmd))
        {
            int x = StdIn.readInt();
            t.delete(x);
        }
    ...
}
```

```
...
else if ("h".equals(cmd))
{
    int x = StdIn.readInt();
    boolean b = t.has(x);
    StdOut.println(b);
}
else if ("i".equals(cmd))
{
    for (int x : t)
        StdOut.print(" " + x);
    StdOut.println();
}
else if ("s".equals(cmd))
{
    StdOut.println(t);
    StdOut.println("size: " + t.size());
    StdOut.println("isEmpty? " + t.isEmpty());
}
}
```

Classe SequentialTable<T> (alternativa)

- Mais convencionalmente, usariámos uma lista ligada, mutável, para guardar os pares chave-valor:

```
public class SequentialTable<K, V>
    extends Table<K, V>
    implements Iterable<K>
{
    private Node head = null;
    private int size = 0;

    private class Node
    {
        private final K key;
        private V value;
        private Node next;

        public Node(K key, V value, Node next)
        {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }
    ...
}
```

A lista é representada por uma referência para o primeiro nó.

A lista mantém a contagem do número de elementos.

A classe interna **Node** representa os nós da lista ligada.

Note bem: o construtor constrói uma nova lista a partir de um par chave-valor e de um nó, o qual normalmente será a cabeça de uma lista pré-existente.

Função get (alt.)

- A função **get** realiza uma busca linear na lista.
- Eis a formulação iterativa padrão:

```
public V get(K key)
{
    V result = null;
    Node x = head;
    while (x != null && !key.equals(x.key))
        x = x.next;
    if (x != null)
        result = x.value;
    return result;
}
```

Função delete (alt.)

- Remover um nó de uma lista ligada é um clássico da programação:

```
public void delete(K key)
{
    if (head != null)
    {
        if (key.equals(head.key))
        {
            head = head.next;
            size--;
        }
        else
        {
            Node x = head;
            while (x.next != null && !key.equals(x.next.key))
                x = x.next;
            if (x.next != null)
            {
                x.next = x.next.next;
                size--;
            }
        }
    }
}
```

Se a lista for vazia, nada a fazer.

Se a chave estiver à cabeça...

Senão procura-se um nó cujo seguinte tenha a chave dada.

!key.equals(x.next.key))

Se tal nó existir, liga-se ao seguinte do seguinte, assim efetivamente eliminando o seguinte, que era o tal que tinha a chave dada.

Se não existir, a lista fica na mesma.

Função put (alt.)

- Como não queremos chaves duplicadas, temos de procurar primeiro a chave, não vá ela já estar presente.
- É mais prático apagar o elemento com a chave dada, se houver, e depois acrescentar o novo par chave-valor num nó à cabeça:

```
public void put(K key, V value)
{
    delete(key);
    head = new Node(key, value, head);
    size++;
}
```

Repare como se faz para
acrescentar um nó à cabeça.

Função has, size e isEmpty (alt.)

- Estas três não têm novidade:

```
public boolean has(K key)
{
    return get(key) != null;
}

public int size()
{
    return size;
}

public boolean isEmpty()
{
    return head == null;
}
```

O iterador das chaves (alt.)

- O iterador guarda uma referência para o nó corrente:

```
private class KeysIterator implements Iterator<K>
{
    private Node i = head;

    public boolean hasNext()
    {
        return i != null;
    }

    public K next()
    {
        K result = i.key;
        i = i.next;
        return result;
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}

public Iterator<Integer> iterator()
{
    return new KeysIterator();
}

public Iterable<Integer> keys()
{
    return this;
}
```

Função `toString` (alt.)

- A lista aparece entre parêntesis retos e cada par aparece entre parêntesis angulosos.

```
public String toString()
{
    String result = "[";
    Node x = head;
    while (x != null)
    {
        result += "<" + x.key + " " + x.value + ">";
        x = x.next;
    }
    result += "]";
    return result;
}
```

Neste caso, omitimos a vírgula entre cada dois pares chave-valor consecutivos, o que simplifica a programação...



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 19

Tabelas de dispersão, encadeamentos separados

Tabelas de dispersão

- Tabelas naturais.
- Tabelas sequenciais.
- Tabelas de dispersão
- Tabelas de encadeamentos separados.
- Tabelas de endereçamento aberto.



Tabelas de fantasia

- Na verdade, tudo no computador são números.
- Portanto, nos casos em que as chaves não são do tipo **int** ou **Integer** (ou de um outro tipo para números inteiros), poderíamos facilmente transformá-las em números inteiros, de maneira injetiva, por meio de uma função **h**, que, por exemplo, calculasse o valor numérico da representação binária da chave.
- Essa função **h** é injetiva, pois duas chaves diferentes terão representação binária diferente e, portanto, imagem diferente pela função **h**.
- Só que, em geral, rapidamente obteríamos valores numéricos para as chaves que ultrapassam a capacidade de memória dos computadores atuais.

Tabelas de dispersão

- As tabelas de dispersão são uma aproximação às “tabelas de fantasia”, onde cada valor seria guardado numa posição calculada por uma função injetiva h do conjunto das chaves sobre o conjuntos dos números naturais.
- A aproximação consiste em renunciar à injetividade da função h , resolvendo *ad hoc* os casos em que surjam chaves com a mesma imagem pela função h .
- Aliás, na prática, para os tipos fornecidos pelo Java, não temos de programar a tal função h fictícia, que transformaria a representação binária da chave num número inteiro; em vez disso usamos logo a função **hashCode**, disponível em todas as classes (a qual não será injetiva, em geral).
- A partir da função **hashCode** na classe da chave, programaremos em cada caso a função **hash**, que fará as vezes da fantasiosa função h .

Método hashCode

- Todas as classes do Java vêm equipadas com o método **hashCode**.
- Nas classes **Integer** e **String**, por exemplo, o método é utilizável diretamente.
- Em classes definidas no programa, convém redefinir o método **hashCode**, porque a função herdada da classe **Object** apenas devolve o endereço do objeto, o que normalmente não serve para as tabelas.
- A função **hashCode** redefinida deve ser programada de maneira a dispersar bem as chaves.
- Isso é uma arte.

Ver <http://stackoverflow.com/questions/113511/best-implementation-for-hashcode-method>, que descreve o método proposto por Joshua Bloch, no livro *Effective Java*.

Função de dispersão

- Cada tabela de dispersão terá uma função privada **hash**, programada em termos da função **hashCode** da classe das chaves.
- Ao acrescentar um par chave-valor, aplicamos a função **hash** à chave, para calcular a posição em que o par chave-valor será guardado, num array interno da tabela.
- Para procurar um valor, dada a chave, aplicamos a função **hash** à chave e vamos buscar o valor ao array, na posição calculada.
- Mas pode acontecer que ao acrescentar um par chave-valor, a posição calculada já esteja ocupada, porque anteriormente outra chave deu o mesmo resultado com a função **hash**.

Colisões

- Uma colisão é uma situação em que a posição calculada pela função **hash** para um novo par chave-valor já está ocupada.
- É uma situação inevitável, em geral.
- Há várias técnicas para resolver as colisões.
- Cada uma delas dá origem a um tipo de tabelas de dispersão.
- Vamos estudar duas dessas técnicas:
 - Tabelas de encadeamentos separados.
 - Tabelas de endereçamento aberto.

Tabelas de encadeamentos separados

- Nas tabelas de encadeamentos separados, os pares chave-valor são guardados num array de tabelas sequenciais.
- Cada par é guardado na tabela cujo índice é calculado pela função **hash**.
- Idealmente, cada tabela só teria um par (ou zero pares).
- Mas, na presença de colisões, terá vários pares.
- Havendo N tabelas sequenciais, a função **hash** meramente calcula o resto da divisão do resultado do método **hashCode** por N .

Classe SeparateChaining<K, V>

- Usaremos uma tabela de dimensão fixa, estabelecida no construtor.
- A dimensão é o comprimento do array de tabelas sequenciais.

```
public class SeparateChaining<K, V>
    implements Table<K, V>
{
    private static final int DEFAULT_DIMENSION = 997;
    private int size;
    private final int dimension;
    private SequentialTable<K, V>[] st;
    ...
}
```

Construtores

- Programamos um construtor por defeito, que usa a dimensão dada por defeito, e um outro, que permite fixar a dimensão:

```
@SuppressWarnings("unchecked")
public SeparateChaining(int dimension)
{
    this.dimension = dimension;
    this.size = 0;
    st = (SequentialTable<K, V>[]) new SequentialTable[dimension];
    for (int i = 0; i < dimension; i++)
        st[i] = new SequentialTable<K, V>();
}

public SeparateChaining()
{
    this(DEFAULT_DIMENSION);
}
```

Atenção!

Função hash

- Observe:

```
private int hash(K k)
{
    return (k.hashCode() & 0x7fffffff) % dimension;
}
```

- Note que, em geral, a expressão **x & 0x7fffffff** resulta num valor cuja representação binária é igual à de **x**, exceto que o primeiro bit é zero.
- Com efeito, o valor **0x7fffffff** tem como representação binária um zero seguido de 31 uns.
- Isto é importante para garantir que o primeiro operando do **%** não é negativo!

Função get

- É simples mas sofisticada:

```
public V get(K key)
{
    return st[hash(key)].get(key);
}
```

Função put

- Também não é complicada, mas trabalha um pouco mais para acertar a contagem:

```
public void put(K key, V value)
{
    int h = hash(key);
    int s = st[h].size();
    st[h].put(key, value);
    size += st[h].size() - s;
}
```

Função delete

- É parecida:

```
public void delete(K key)
{
    int h = hash(key);
    int s = st[h].size();
    st[h].delete(key);
    size += st[h].size() - s;
}
```

Funções has, size e isEmpty

- Estas três são de veras elementares:

```
public boolean has(K key)
{
    return get(key) != null;
}

public int size()
{
    return size;
}

public boolean isEmpty()
{
    return size == 0;
}
```

O iterador das chaves

- Criamos uma bag com as chaves todas e devolvemos o iterador da bag:

```
public Iterator<K> iterator()
{
    Bag<K> b = new Bag<>();
    for (int i = 0; i < dimension; i++)
        for (K k : st[i])
            b.add(k);
    return b.iterator();
}
```

Exercício: programa o iterador recorrendo diretamente aos iteradores das tabelas sequenciais.

Função `toString`

- O resultado é uma sequência de linhas, uma para cada tabela sequencial:

```
public String toString()
{
    String result = "";
    for (int i = 0; i < dimension; i++)
        result += i + ":" + st[i] + newLine;
    return result;
}
```

- Recorde:

```
private static String newLine = System.getProperty("line.separator");
```

Função de teste

- É semelhante às anteriores:

```
public static void testSeparateChaining(int d)
{
    SeparateChaining<Integer, String> ht = new SeparateChaining<>(d);
    while (!StdIn.isEmpty())
    {
        String cmd = StdIn.readString();
        if ("p".equals(cmd))          // put
        {
            ...
        }
        else if ("g".equals(cmd))   // get
        {
            ...
        }
        else if ("d".equals(cmd))   // delete
        {
            ...
        }
        else if ("i".equals(cmd))   // iterator
        {
            ...
        }
        else if ("s".equals(cmd))   // toString
        {
            ...
        }
    }
}
```

```
public static void main(String[] args)
{
    int d = 7;
    if (args.length > 0)
        d = Integer.parseInt(args[0]);
    testSeparateChaining(d);
}
```

Testando

```
$ java SeparateChaining
s
0: []
1: []
2: []
3: []
4: []
5: []
6: []
size: 0
isEmpty? true
p 5 uuu
p 22 ooo
p 77 www
s
0: [<77 www>]
1: [<22 ooo>]
2: []
3: []
4: []
5: [<5 uuu>]
6: []
size: 3
isEmpty? false
i
77 22 5
```

```
p 701 yyy
p 8 ttt
p 40 mmm
s
0: [<77 www>]
1: [<8 ttt>,<701 yyy>,<22 ooo>]
2: []
3: []
4: []
5: [<40 mmm>,<5 uuu>]
6: []
size: 6
isEmpty? false
g 22
ooo
g 5
uuu
d 800
s
0: [<77 www>]
1: [<8 ttt>,<701 yyy>,<22 ooo>]
2: []
3: []
4: []
5: [<40 mmm>,<5 uuu>]
6: []
size: 6
isEmpty? false
d 77
g 77
null
d 22
g 22
null
s
0: []
1: [<8 ttt>,<701 yyy>]
2: []
3: []
4: []
5: [<40 mmm>,<5 uuu>]
6: []
size: 4
isEmpty? false
p 70000 kkk
p 171 sss
p 2103 qqq
s
0: [<70000 kkk>]
1: [<8 ttt>,<701 yyy>]
2: []
3: [<2103 qqq>,<171 sss>]
4: []
5: [<40 mmm>,<5 uuu>]
6: []
size: 7
isEmpty? false
```

Observações

- Uma tabela de dispersão de encadeamentos separados com dimensão N é N vezes mais rápida do que a tabela sequencial correspondente.
- Em situações típicas, cada tabela sequencial na tabela de dispersão tem **size / dimension** elementos, mais ou menos.
- Se tivermos uma boa estimativa do tamanho que a tabela atingirá, podemos dimensionar o array de tabelas sequenciais em conformidade, de maneira a garantir inserções, buscas e eliminações muito rápidas.

Valor da dimensão

- Em geral, usamos números primos para a dimensão do array.
- Isto porque a dimensão é usada como divisor no cálculo da função de hash.
- Contraexemplo: a chave representa a hora do dia, no formato *hhmmss*, por exemplo 142034. Se tivéssemos uma tabela com, por exemplo, dimensão 1000, as tabelas sequenciais 60, 61, ..., 99, 160, ..., 199, 260, ..., 299 e muitas outras estavam condenadas a ficar vazias.
- Nesse caso seria preferível usar 997, que é um número primo, em vez de 1000, pois evita esse fenômeno.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 20

Tabelas de dispersão, endereçamento aberto

Tabelas de dispersão, de novo

- Tabelas naturais.
- Tabelas sequenciais.
- Tabelas de encadeamentos separados.
- Tabelas de endereçamento aberto.



Tabelas de endereçamento aberto

- Nas tabelas de endereçamento aberto, os pares chave-valor são guardados em dois arrays paralelos, um para as chaves e outro para os valores.
- Cada chave é guardada no array das chaves na posição cujo índice é o resultado da função **hash**.
- Cada valor é guardado no array dos valores, na posição paralela à da chave (isto é, na posição com o mesmo índice).
- Sendo N o comprimento de cada um dos arrays, a função **hash** meramente calcula o resto da divisão do resultado do método **hashCode** por N .
- A capacidade das tabelas de endereçamento aberto está limitada ao comprimento dos arrays.
- Na prática, não convém que o tamanho ultrapasse cerca de 75% do comprimento dos arrays.

Classe Hashtable<T>

- Usaremos uma tabela de redimensionável, em que em cada momento a capacidade será um número primo:

```
public class HashTable<K, V> implements Table<K, V>
{
    private int size;
    private int capacity;
    private K[] keys;
    private V[] values;

    private static int[] primes = {
        17, 37, 79, 163, 331,
        673, 1361, 2729, 5471, 10949,
        21911, 43853, 87719, 175447, 350899,
        701819, 1403641, 2807303, 5614657,
        11229331, 22458671, 44917381, 89834777, 179669557
    };
    ...
}
```

Construtores

- O construtor público não tem argumentos e cria uma tabela com capacidade 17; o construtor privado é usado internamente:

```
@SuppressWarnings("unchecked")
// Always use a prime number for the capacity
private HashTable(int x) // yes, this is private
{
```

```
    this.capacity = primes[x];
    this.size = 0;
    keys = (K[]) new Object[capacity];
    values = (V[]) new Object[capacity];
    resizeCount = x;
}
```

O argumento **x** representa o número de vezes que a tabela terá crescido. De cada vez, a nova capacidade é próximo número primo na tabela.

```
public HashTable()
{
    this(0); // default capacity is primes[0], i.e., 17.
}
```

Função get

- Procuramos a chave circularmente a partir da posição de dispersão (isto é, a partir do índice calculado pela função **hash**).
- A busca quando chegamos a uma posição livre ou quando encontramos a chave:

```
public V get(K key)
{
    int i = hash(key);
    while (keys[i] != null && !key.equals(keys[i]))
        i = (i+1) % capacity;
    return values[i];
}
```

Note bem: se houver bastantes posições livres e se estas estiverem bem distribuídas, as buscas serão curtas. É por isso (isto é, é para garantir que as buscas são curtas) que não convém deixar os arrays encher muito.

Função privada putBasic

- Se não for preciso redimensionar, começa como a função **get**, procurando a chave.
- Se encontrar, reafeta o valor; se não, acrescenta a chave na posição livre e o valor na posição correspondente:

```
private void putBasic(K key, V value)
{
    int i = hash(key);
    while (keys[i] != null && !key.equals(keys[i]))
        i = (i+1) % capacity;
    if (keys[i] == null)
    {
        size++;
        keys[i] = key;
    }
    values[i] = value;
```

Função put

- Será preciso redimensionar quando o fator de carga é maior do que o máximo fator de carga permitido:

```
public void put(K key, V value)
{
    if (loadFactor() >= maxLoadFactor)
    {
        assert resizeCount + 1 < primes.length;
        // if it fails, it fails...
        resize(resizeCount + 1);
    }
    putBasic(key, value);
}
```

Fator de carga

- O fator de carga é o quociente do tamanho pela capacidade:

```
public double loadFactor()
{
    return (double) size / capacity;
}
```

- É claro que quanto maior for o fator de carga, mais trabalho terá a tabela, na função **get** e na função **put**.

Fator de carga máximo

- Cada tabela conhece o seu fator de carga máximo:

```
public class HashTable<K, V> implements Table<K, V>
{
    ...
    public final double DEFAULT_MAX_LOAD_FACTOR = 0.5;
    private double maxLoadFactor = DEFAULT_MAX_LOAD_FACTOR;
    ...
}
```

- Por defeito é 50%, mas pode ser mudado e consultado:

```
public double maxLoadFactor()
{
    return maxLoadFactor;
}

public void setMaxLoadFactor(double x)
{
    assert 0.1 <= x && x <= 0.9;
    maxLoadFactor = x;
}
```

Não deixaremos que o fator de carga seja muito grande ou muito pequeno.

Função delete

- A nossa classe não permite eliminar, nesta fase.
- Eliminar é possível, mas mais trabalhoso.
- Note que não basta colocar a **null** a posição onde está a chave a eliminar, pois isso poderia quebrar uma sequência de chaves colididas.

```
public void delete(K key)
{
    throw new UnsupportedOperationException();
}
```

Funções has, size, isEmpty, hash

- São idênticas às da classe **SeparateChaining**:

```
public boolean has(K key)
{
    return get(key) != null;
}

public int size()
{
    return size;
}

public boolean isEmpty()
{
    return size == 0;
}

private int hash(K k)
{
    return (k.hashCode() & 0x7fffffff) % capacity;
}
```

O iterador das chaves

- Usamos a técnica das tabelas naturais:

```
private class Keys implements Iterator<K>
{
    private int i;

    public Keys()
    {
        i = advance(0);
    }

    private int advance(int x)
    {
        int result = x;
        while (result < keys.length && keys[result] == null)
            result++;
        return result;
    }

    public boolean hasNext()
    {
        return i < keys.length;
    }

    public K next()
    {
        K result = keys[i++];
        i = advance(i);
        return result;
    }

    ...
}
```

```
public Iterator<K> iterator()
{
    return new Keys();
}
```

O iterador dos valores

- Por simetria, acrescentamos o iterador dos valores:

```
private class Values implements Iterator<V>
{
    private int i;

    public Values()
    {
        i = advance(0);
    }

    private int advance(int x)
    {
        int result = x;
        while (result < keys.length && keys[result] == null)
            result++;
        return result;
    }

    public boolean hasNext()
    {
        return i < keys.length;
    }

    public V next()
    {
        V result = values[i++];
        i = advance(i);
        return result;
    }

    ...
}
```

```
public Iterator<V> values()
{
    return new Values();
}
```

Para normalizar a nomenclatura, juntamos uma função **keys** que dá o iterador da chaves (que também é dado pela função **iterator**):

```
public Iterator<K> keys()
{
    return new Keys();
}
```

Função `toString`

- O resultado é uma sequência de linhas, uma para cada posição nos dois arrays:

```
public String toString()
{
    String result = "";
    for (int i = 0; i < capacity; i++)
    {
        result += i + ":";
        if (keys[i] != null)
            result += " " + keys[i] + " " + values[i];
        result += newLine;
    }
    return result;
}
```

- Recorde:

```
private static String newLine = System.getProperty("line.separator");
```

Função de teste

- É semelhante às anteriores:

```
public static void testHashTable()
{
    HashTable<Integer, String> ht = new HashTable<>();
    while (!StdIn.isEmpty())
    {
        String cmd = StdIn.readString();
        if ("p".equals(cmd))
            ...
        else if ("g".equals(cmd))
            ...
        // else if ("d".equals(cmd))
        // ...
        else if ("i".equals(cmd))
        {
            ...
        }
        else if ("s".equals(cmd))
        {
            StdOut.print(ht);
            StdOut.println("size: " + ht.size());
            StdOut.println("isEmpty? " + ht.isEmpty());
            StdOut.printf("load factor: %.3f", ht.loadFactor());
            StdOut.println();
        }
    }
}
```

Não há comando ‘d’ porque a tabela não implementa o método **delete**.

Aqui mostra o iterador das chaves e o iterador dos valores.

Testando

```
$ java ... HashTable A
s
0:
1:
2: 2 aaa
3: 172 sss
4: 20 vvv
5:
6:
7:
8:
9:
10:
11:
12:
13:
14:
15:
16:
size: 0
isEmpty? true
load factor: 0.000
p 2 aaa
p 10 bbb
p 172 sss
p 20 vvv
```

```
s
0:
1:
2: 2 aaa
3: 172 sss
4: 20 vvv
5:
6:
7:
8:
9:
10: 10 bbb
11:
12:
13:
14:
15:
16:
size: 4
isEmpty? false
load factor: 0.235
g 2
aaa
g 172
sss
g 20
vvv
g 10
bbb
g 8
null
```

```
i
2 172 20 10
aaa sss vvv bbb
p 180 uuu
p 197 vvv
s
0:
1:
2: 2 aaa
3: 172 sss
4: 20 vvv
5:
6:
7:
8:
9:
10: 10 bbb
11: 180 uuu
12: 197 vvv
13:
14:
15:
16:
size: 6
isEmpty? false
load factor: 0.353
i
2 172 20 10 180 197
aaa sss vvv bbb uuu vvv
```

Observamos várias colisões mas ainda
não houve redimensionamento.

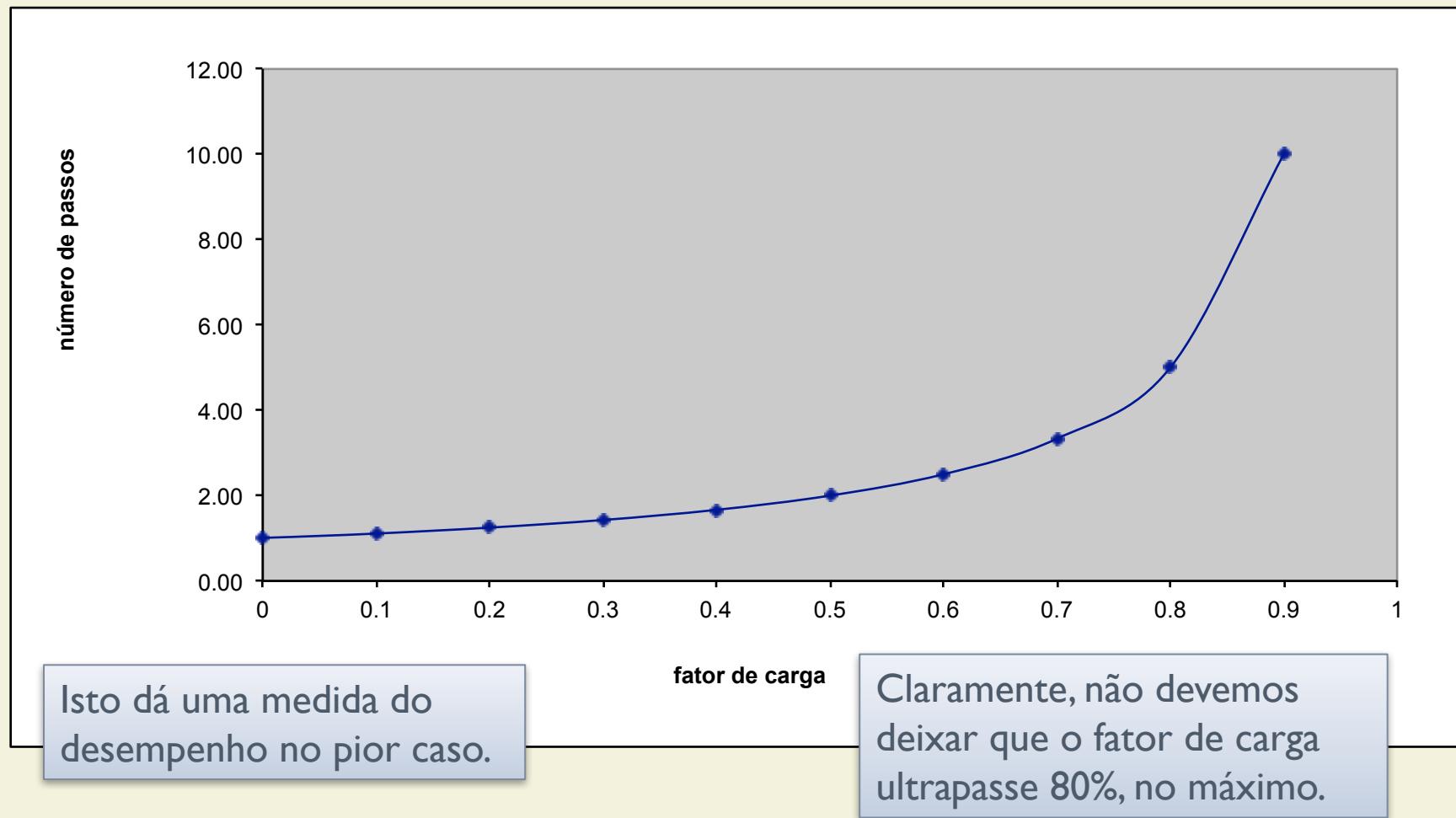
Propriedade fundamental do endereçamento aberto

- Numa tabela de dispersão com endereçamento aberto, em que as chaves estejam bem dispersas, o número médio de passos na função **get** e na função **put** depende apenas do fator de carga.

Isto significa que o desempenho de uma tabela de dispersão de endereçamento aberto não depende do tamanho da tabela, o que é uma circunstância notável. Depende apenas do quociente do tamanho pela capacidade.

Desempenho das tabelas de dispersão

- Se o fator de carga for f , o número médio de passos na função **get** quando a chave não existe é $1/(1-f)$.



Redimensionamento

- A função `put` redimensiona automaticamente:

```
public void put(K key, V value)
{
    if (loadFactor() >= maxLoadFactor)
    {
        assert resizeCount + 1 < primes.length;
        // if it fails, it fails...
        resize(resizeCount + 1);
    }
    putBasic(key, value);
}
```

- Eis a função `resize`:

```
private void resize(int m)
{
    HashTable<K, V> t = new HashTable<>(m);
    for (int i = 0; i < capacity; i++)
        if (keys[i] != null)
            t.put(keys[i], values[i]);
    capacity = t.capacity;
    keys = t.keys;
    values = t.values;
    resizeCount = m;
}
```

Primeiro cria uma segunda tabela, depois insere lá tudo; finalmente “rouba-lhe” os arrays. (A capacidade é igual a `primes[m]`).

Testando o redimensionamento

Prosseguimos a experiência anterior, numa altura em que o tamanho era 6:

```
p 23 ttt
p 340 zzz
s
0: 340 zzz
1:
2: 2 aaa
3: 172 sss
4: 20 vvv
5:
6: 23 ttt
7:
8:
9:
10: 10 bbb
11: 180 uuu
12: 197 vvv
13:
14:
15:
16:
size: 8
isEmpty? false
load factor: 0.471
```

```
p 1700 mmm
s
0: 340 zzz
1: 1700 mmm
2: 2 aaa
3: 172 sss
4: 20 vvv
5:
6: 23 ttt
7:
8:
9:
10: 10 bbb
11: 180 uuu
12: 197 vvv
13:
14:
15:
16:
size: 9
isEmpty? false
load factor: 0.529
```

Aqui o fator de carga ultrapassou o limite. No próximo **put** haverá redimensionamento.

```
p 51 qqq
s
0:
1:
2: 2 aaa
3:
4:
5:
6:
7: 340 zzz
8:
9:
10: 10 bbb
11:
12: 197 vvv
13:
14: 51 qqq
15:
16:
17:
18:
19:
20: 20 vvv
21:
22:
23: 23 ttt
24: 172 sss
25:
26:
27:
28:
29:
30:
31:
32: 180 uuu
33:
34:
35: 1700 mmm
36:
size: 10
isEmpty? false
load factor: 0.270
```

Aglomeração primária

- Considere, como exemplo, uma tabela com capacidade 7, inicialmente vazia. Ao chegar a primeira chave, a probabilidade de ela vir a preencher a posição 3, por exemplo, é igual à de ela preencher qualquer outra posição, ou seja é $1/7$. Suponhamos que ela preenche a posição 3.
- Agora chega a segunda chave. A probabilidade de esta vir a preencher a posição 3 é zero, porque a posição 3 já está ocupada. Mas a probabilidade de vir a preencher a posição 4 é $2/7$, a soma da probabilidade de a posição de dispersão calculada ser 3 com a probabilidade de ser 4.
- Quer dizer, a probabilidade de uma nova chave ficar a seguir a uma posição ocupada é maior do que a probabilidade média. Logo, há uma certa tendência indesejável para as chaves se aglomerarem no vector das chaves, em vez de terem uma distribuição uniforme.

Dupla dispersão

- Evita-se a aglomeração primária usando a dupla dispersão: em vez de tentar apanhar uma posição livre linearmente na tabela incrementa-se de um valor obtido por uma segunda função de dispersão.
- Agora, na função **get**, por exemplo, avança-se não de 1 em 1 mas de **h2** em **h2**, representando por **h2** o resultado da segunda função de dispersão:

```
public V get(K key)
{
    int i = hash(key);
    while (keys[i] != null && !key.equals(keys[i]))
        i = (i+h2) % capacity;
    return values[i];
}
```

O valor de **h2** deve estar entre 2 e **capacity**-2 (inclusive). De facto, não convém que seja 1 ou **capacity**-1, pois então estariámos no caso “normal”; e não pode ser 0 ou **capacity**, porque assim a tabela não funcionaria. Qualquer outro valor serve, admitindo que, como sempre, **capacity** é um número primo. Se **capacity** não fosse um número primo, nem todos os valores do intervalo [0..**capacity**] seriam apanhados, no ciclo da função **get**, e a tabela não funcionaria.

Paradoxo do dia de aniversário

- Qual é a probabilidade de haver nesta sala duas pessoas ou mais que fazem anos no mesmo dia?
- Generalizando: qual é a probabilidade de, escolhendo sucessivamente K números de um conjunto com N números, haver nos números escolhidos dois ou mais números iguais.
- Bom, é 1 menos a probabilidade não haver dois números iguais, entre os escolhidos.
- Há quantas sequências com K elementos escolhidos entre os N? N^K .
- Há quantas sequências com K elementos não repetidos escolhidos entre os N? $N!/(N-K)!$
- Todas as sequências poder ser escolhidas com igual probabilidade.
- Logo, se houver K pessoas na sala, a probabilidade de haver duas ou mais com o mesmo dia de aniversário é:

Repare que para $K = 0$ ou para $K = 1$ a expressão vale 0. Realmente se houver zero pessoas ou uma pessoa na sala, a probabilidade de haver duas que fazem anos no mesmo dia é zero!

$$1 - \frac{365!}{(365-K)!} \frac{1}{365^K}$$

É só fazer as contas!

Tabela paradoxal

- Feitas as contas, com **BigInteger** e **BigDecimal**, obtemos esta tabela: na coluna da esquerda, o número de pessoas na sala; na coluna da direita a probabilidade de haver duas ou mais pessoas com a mesma data de aniversário.
- Observamos, que a partir de 23, a probabilidade é maior que 50%.
- Moral da história: há mais colisões do que parece...

0	0.000000
1	0.000000
2	0.002740
3	0.008204
4	0.016356
5	0.027136
6	0.040462
7	0.056236
8	0.074335
9	0.094624
10	0.116948
11	0.141141
12	0.167025
13	0.194410
14	0.223103
15	0.252901
16	0.283604
17	0.315008
18	0.346911
19	0.379119
20	0.411438
21	0.443688
22	0.475695
23	0.507297
24	0.538344
25	0.568700
26	0.598241
27	0.626859
28	0.654461
29	0.680969
30	0.706316



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 21

Árvores

Árvores



- Árvores.
- Árvores simples.
- Árvores binárias.
- Árvores mutáveis e árvores imutáveis.

Árvores

Dizemos que estas árvores são **binárias**, porque quando não são vazias têm “lá dentro” duas árvores.

- Em programação, uma árvore é ou uma estrutura vazia ou uma estrutura que tem um valor, uma árvore dita árvore esquerda e outra árvore dita árvore direita.
- Podemos representar isto em Java por meio de uma classe abstrata, para o tipo árvore, e duas classes não abstratas, uma para árvores vazias e outra para árvores não vazias:

```
public abstract class Tree<T>
{
    ...
}
```

```
class Empty<T> extends Tree<T>
{
    ...
}
```

*I think that I shall never see
A poem lovely as a tree.*

Joyce Kilmer (1913), citado em Knuth,
The Art of Computer Programming, vol. I,
Fundamental Algorithms.

```
class Cons<T> extends Tree<T>
{
    private T value;
    private Tree<T> left;
    private Tree<T> right;
    ...
}
```

Funções para as árvores simples

- Equipemos as nossas árvores simples com funções para contar o número de elementos, para ver se existe um dado valor, e para representar uma árvore por meio de uma cadeia de caracteres.
- Estas funções ficam abstratas na classe **Tree<T>**:

```
public abstract class Tree<T>
{
    public abstract int size();
    public abstract boolean has(T x);
    ...
}
```

Falta aqui a função **toString**, porque essa vem herdada da classe **Object**.

- Agora temos de programar as três funções em cada uma das outras classes.

Funções nas árvores vazias

- Ficam muito simples:

```
class Empty<T> extends Tree<T>
{
    public int size()
    {
        return 0;
    }

    public boolean has(T x)
    {
        return false;
    }

    public String toString()
    {
        return "()";
    }
}
```

Note que esta classe
não tem membros de
dados.

Funções nas árvores não vazias

```
class Cons<T> extends Tree<T>
{
    private T value;
    private Tree<T> left;
    private Tree<T> right;

    public Cons(T value, Tree<T> left, Tree<T> right)
    {
        this.value = value;
        this.left = left;
        this.right = right;
    }

    public int size()
    {
        return 1 + left.size() + right.size();
    }

    public boolean has(T x)
    {
        return value.equals(x) || left.has(x) || right.has(x);
    }

    public String toString()
    {
        return "(" + value + left.toString() + right.toString() + ")";
    }
}
```

Dizemos que esta árvore é “simples” porque tem apenas um membro de dados além das referências para as subárvore.

Neste caso, precisamos também do construtor.

Funções de teste para as árvores simples

- Programamos na classe abstrata:

```
public abstract class Tree<T>
{
    public abstract int size();
    public abstract boolean has(T x);

    public static void testTree()
    {
        Tree<Integer> empty = new Empty<>();
        Tree<Integer> t1 = new Cons<>(6, empty, empty);
        StdOut.println(t1);
        StdOut.printf("%d %b %b\n", t1.size(), t1.has(6), t1.has(25));
        Tree<Integer> t2 = new Cons<>(4, empty, empty);
        StdOut.println(t2);
        StdOut.printf("%d %b %b\n", t2.size(), t2.has(4), t2.has(6));
        Tree<Integer> t3 = new Cons<>(15, t1, t2);
        StdOut.println(t3);
        StdOut.printf("%d %b %b\n", t3.size(), t3.has(4), t3.has(15));
    }

    public static void main(String[] args)
    {
        testTree();
    }
}
```

```
$ java ... Tree
(6()())
1 true false
(4()())
1 true false
(15(6()())(4()())))
3 true true
```

Árvores de nós

- Convencionalmente, em Java, as árvores binárias são representadas por um conjunto de nós, cada um com valor e com duas referências, uma para o nó que representa a subárvore esquerda e a outra para a referência que representa a subárvore direita.
- Um dos nós é especial, pois representa a raiz da árvore.
- Cada nó da árvore é referenciado por outro nó, exceto a raiz, que é referenciada por um membro de dados.
- Quando a árvore é vazia, esse membro de dados vale **null**.

Classe Tree<T>, alternativa

- É um clássico:

```
public class Tree<T>
{
    private Node root;

    private class Node
    {
        private T value;
        private Node left;
        private Node right;

        public Node(T value, Node left, Node right)
        {
            this.value = value;
            this.left = left;
            this.right = right;
        }
    }

    ...
}
```

Não confunda. Esta classe tem o mesmo nome que a outra classe abstrata, mas não é a mesma. Existe num “universo” diferente. Mostramo-la aqui só para comparar.

Funções da classe Tree<T> alternativa

- Dois construtores:

```
public Tree()  
{  
    root = null;  
}
```

Este constrói uma árvore vazia.

```
public Tree(T value, Tree<T> left, Tree<T> right)  
{  
    root = new Node(value, left.root, right.root);  
}
```

Este constrói uma árvore não vazia.

- A função **size**:

```
public int size()  
{  
    return size(root);  
}  
  
private int size(Node p)  
{  
    return p == null ? 0 : 1 + size(p.left) + size(p.right);  
}
```

Repare: o método público invoca uma função privada, recursiva, que trabalha sobre os nós. Ainda que tenham o mesmo nome (por conveniência), são funções diferentes.

Mais funções da classe Tree<T> alternativa

- Função **has**:

```
public boolean has(T x)
{
    return has(root, x);
}

private boolean has(Node p, T x)
{
    return p != null && (x.equals(p.value) || has(p.left, x) || has(p.right, x));
}
```

- Função **toString**:

```
public String toString()
{
    return stringof(root);
}

private String stringof(Node p)
{
    return "(" + (p == null ? "" : p.value.toString() + stringof(p.left) +
               stringof(p.right)) + ")";
}
```

É sempre o mesmo esquema: o método público e a função recursiva privada.

Funções de teste para a classe alternativa

- Programamos na classe abstrata:

```
public abstract class Tree<T>
{
    ...

    public static void testTree()
    {
        Tree<Integer> empty = new Tree<>();
        Tree<Integer> t1 = new Tree<>(6, empty, empty);
        System.out.println(t1);
        StdOut.printf("%d %b %b\n", t1.size(), t1.has(6), t1.has(25));
        Tree<Integer> t2 = new Tree<Integer>(4, empty, empty);
        System.out.println(t2);
        StdOut.printf("%d %b %b\n", t2.size(), t2.has(4), t2.has(6));
        Tree<Integer> t3 = new Tree<Integer>(15, t1, t2);
        System.out.println(t3);
        StdOut.printf("%d %b %b\n", t3.size(), t3.has(4), t3.has(15));
    }

    public static void main(String[] args)
    {
        testTree();
    }
}
```

```
$ java ... Tree
(6()())
1 true false
(4()())
1 true false
(15(6()())(4()())))
3 true true
```

Dá o mesmo que a outra, claro.

Exercício: juntar um valor à árvore

- Queremos uma função para acrescentar um valor à árvore, do lado mais pequeno, **recursivamente**.
- O lado mais pequeno é, por definição, o lado cujo tamanho (calculado pela função **size**) é menor.
- Em caso de empate, acrescenta-se do lado esquerdo.
- O “recursivamente” significa que, se a árvore não for vazia, acrescenta na subárvore mais pequena, usando a mesma regra, isto é na subárvore mais pequena da subárvore mais pequena, e assim sucessivamente.
- Se a árvore for vazia, o resultado é uma árvore só com o elemento acrescentado.

Juntar, na classe abstrata

- Primeiro programamos na classe **Tree** original (aquela que tem a classe abstrata com duas classes derivadas, não abstratas).
- Atenção: neste caso, “acrescentar um valor” quer dizer construir uma nova árvore igual à anterior mas com mais um elemento inserido:

```
public abstract class Tree<T>
{
    ...
    public abstract Tree<T> add(T x);
    ...
}
```

Repare: a função devolve uma árvore.

Juntar, nas classes Empty e Cons

- Na classe **Empty**, é simples.

```
public Tree<T> add(T x)
{
    return new Cons<>(x, this, this);
}
```

- Na classe **Cons**, também:

```
public Tree<T> add(T x)
{
    return left.size() <= right.size() ?
        new Cons<>(value, left.add(x), right) :
        new Cons<>(value, left, right.add(x));
}
```

Palavras para quê?

Testando a função add

```
public static void testAdd()
{
    Tree<Integer> t = new Empty<>();
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        t = t.add(x);
        StdOut.println(t);
    }
}
```

Juntámos 7, 1000, 32, 90, 55, 2 e 256. Ficámos com uma árvore com 7 elementos, perfeitamente equilibrada: no primeiro nível está a raiz, 7; no segundo nível estão 1000 e 32; no terceiro nível estão 90, 55, 42 e 256.

```
$ ls
Cons.class
Empty.class
Tree.class
$ java -ea -cp .:.../* Tree B
7
(7())
1000
(7(1000()))
32
(7(1000())(32()))
90
(7(1000(90())))
55
(7(1000(90()))(32(55())))
42
(7(1000(90()))(42()))
256
(7(1000(90()))(42()))(32(55()))
(256())()
```

Agora na classe Tree<T> alternativa

- Observe, é mais complicado:

```
public class Tree<T>
{
    ...
    public void add(T x)
    {
        root = add(x, root);
    }
    private Node add(T x, Node p)
    {
        Node result = p;
        if (p == null)
            result = new Node(x, null, null);
        else
            if (size(p.left) <= size(p.right))
                p.left = add(x, p.left);
            else
                p.right = add(x, p.right);
        return result;
    }
}
```

De facto, as funções das árvores com nós são mais complicadas do que as das outras árvores.

Repare: a função modifica a árvore, em vez de criar uma árvore nova!

Repare: exceto no caso em que o **p** vale **null**, o resultado é **p**. Entretanto, não sendo **p** **null**, um dos “descendentes” de **p** deixará de ser **null**, ficando a apontar para um novo nó que contém **x**.

Testando a função add, classe alternativa

- É parecida com a outra, mas repare nas diferenças.

```
public static void testAdd()
{
    Tree<Integer> t = new Tree<>();
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        t.add(x);
        StdOut.println(t);
    }
}
```

Repare: a função **add** é um modificador:
modifica o objeto sobre o qual se aplica.

```
$ ls
Tree$Node.class  Tree.class
$ java -ea -cp .:.../* Tree B
78
(78())
2
(78(2()))
999
(78(2())(999()))
23
(78(2(23())())(999()))
```

Árvores mutáveis e árvores imutáveis

- As árvores com nós são mutáveis: ao acrescentarmos um elemento, modificamos o objeto.
- As árvores “abstratas” são imutáveis: ao acrescentarmos um nó criamos uma árvore nova, com os mesmos valores da outra, mais um.
- Note que os valores da árvore original não são duplicados para irem para a árvore nova, com poucas exceções.
- Na verdade, a memória é reaproveitada: algumas posições de memória são partilhadas pela árvore original e pela árvore nova.
- Usaremos preferencialmente as árvores “abstratas”.

Usaremos as expressões “árvore abstrata” e “árvore com nós” informalmente, para distinguir as duas classes, na linguagem corrente.

Distinguindo, usando pacotes

- Colocaremos as árvores imutáveis no pacote **trees.immutable**:

```
package trees.immutable;  
  
// ...  
  
public abstract class Tree<T>  
// ...
```

- E colocaremos as árvores mutáveis no pacote **trees.mutable**:

```
package trees.mutable;  
  
// ...  
  
public class Tree<T>  
// ...
```

Mais adiante, colocaremos isto no nosso **jar** AED.jar, dentro de um pacote **aed**. Aliás, todas as classes presentes no **jar** irão para esse pacote também. Nessa altura, para serem usadas sem qualificação, terão de ser importadas explicitamente.

Joshua Bloch: *Classes should be immutable unless there's a very good reason to make them mutable (in Effective Java)*.

Problema clássico: somar a árvore

- Somar a árvore é abuso de linguagem; queremos dizer somar todos os valores presentes na árvore.
- Não podemos fazer isso diretamente na classe, pois os elementos são de tipo **T**, e não sabemos nada sobre o tipo **T**.
- Mas podemos funcionalmente aplicar uma operação binária todos os elementos da árvore, sucessivamente.
- E como veremos, até ganhamos com isso.

As únicas operações que podemos fazer com um objeto de tipo **T** na classe **Tree<T>** é compará-lo com outro para ver se são iguais, com o método **equals**, e “transformá-lo” numa cadeia de carateres, com o método **toString**. Não podemos somar um valor de tipo **T** com outro!

Dobrando a árvore

- A função que aplica sucessivamente uma operação binária a um elemento de uma estrutura e ao resultado anterior, começando por um dado valor, chama-se **fold**:

Na classe abstrata.

```
public abstract T fold(BinaryOperator<T> f, T zero);
```

```
public T fold(BinaryOperator<T> f, T zero)
{
    return zero;
}
```

Na classe Empty<T>.

```
public T fold(BinaryOperator<T> f, T zero)
{
    return f.apply(f.apply
        (value, left.fold(f, zero)), right.fold(f, zero));
}
```

Na classe Cons<T>.

Testando a função fold

- Como exemplo, somamos e achamos o mínimo da árvore, usando **fold**:

```
public static void testFold()
{
    Tree<Integer> t = new Empty<>();
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        t = t.add(x);
        StdOut.println(t);
        int sum = t.fold((a, b) -> a+b, 0);
        StdOut.println(sum);
        int min = t.fold((a, b) -> Math.min(a, b), Integer.MAX_VALUE);
        StdOut.println(min);
    }
}
```

```
$ java -ea -cp .:.../* Tree D
85
(85()())
85
85
32
(85(32()())())
117
32
100
(85(32()())(100()()))
217
32
20
(85(32(20()())())(100()()))
237
20
```



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 22

Exercícios com árvores imutáveis

Exercícios com árvores imutáveis

- Escrever as árvores.
- Ler as árvores.
- Desenhar as árvores.



Escrevendo a árvore

- Para escrever árvores, preferimos a escrita parentética, através da função **toString()**:

```
public String toString()
{
    return "()";    Na classe Empty<T>.
}
```

```
public String toString()      Na classe Cons<T>.
{
    return "(" + value + left.toString() + right.toString() + ")";
}
```

- Experimentamos assim:

```
public static void testAdd()
{
    Tree<Integer> t = new Empty<Integer>();
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        t = t.add(x);
        StdOut.println(t);
    }
}
```

```
$ java -cp .:.../*: Tree B
12
(12()())
88
(12(88()())())
97
(12(88()())(97()()))
5
(12(88(5()())())(97()()))
2016
(12(88(5()())())(97(2016()())()))
-45
(12(88(5()())(-45()())))(97(2016()())()))
```

Lendo as árvores

- Com árvores de inteiros, a escrita parentética identifica univocamente a árvore. Logo deve ser invertível.
- Quer dizer, dada uma cadeia parentética que representa uma árvore de inteiros, devemos conseguir construir a árvore.
- Faremos isso usando um “método de fábrica”, isto é, uma função estática que devolve um objeto do tipo da classe:

```
public static Tree<Integer> fromString(String s)
{
    ...
}
```

Expressões regulares

- A primeira tarefa é obter os tóquenes que constituem a cadeia.
- Por exemplo, na cadeia `(15(6()())(4()5()())()` os tóquenes são os seguintes (separados por vírgula):
`(,,15,,(,,6,,(,,),,,(,,4,,(,,),,,(,,5,,(,,),,,),,,))`
- Usamos expressões regulares para obter os tóquenes: um tóquene é, ou um número inteiro, ou um parêntesis a abrir, ou um parêntesis a fechar; um número inteiro é, ou zero, ou um sinal menos seguido de um ou mais algarismos:

```
public static final String REGEX_TOKEN = "\\\-?\\\d+|\\\\(|\\\\)|";
```

`"\\-?\\d+|\\\\(|\\\\)"`

Toquenizando a cadeia

- Usamos a função **groups** da classe **RegularExpressions**:

```
public static ArrayBag<String> groups(String s, String regex)
{
    ArrayBag<String> result = new ArrayBag<>();
    Pattern pattern = Pattern.compile(regex);
    Matcher matcher = pattern.matcher(s);
    while (matcher.find())
    {
        String z = matcher.group();
        result.add(z);
    }
    return result;
}
```

- A função **groups** devolve um **ArrayBag**, mas o que nos convém é uma **Queue**:

```
public static Tree<Integer> fromString(String s)
{
    ArrayBag<String> tokens = RegularExpressions.groups(s, REGEX_TOKEN);
    Queue<String> queue = new Queue<>();
    tokens.visit(x -> queue.enqueue(x));
    ...
}
```

Árvore a partir de Fila

- Transformar uma fila de tóquenes numa árvore é um exercício simples, mas sofisticado:

```
public static Tree<Integer> fromQueue(Queue<String> q)
{
    Tree<Integer> result = new Empty<Integer>();
    assert ".equals(q.front())";
    q.dequeue();
    if (!)".equals(q.front()))
    {
        int x = Integer.parseInt(q.dequeue());
        Tree<Integer> left = fromQueue(q);
        Tree<Integer> right = fromQueue(q);
        result = new Cons<Integer>(x, left, right);
    }
    assert ")".equals(q.front());
    q.dequeue();
    return result;
}
```

Escrevendo indentadamente

- Se escrevermos a árvore de forma indentada, percebe-se melhor a estrutura.
- A ideia é escrever a raiz e depois a subárvore esquerda e depois a subárvore direita, ambas um pouco desviadas para direita na linha, isto é, com uma margem maior.
- Cada nova subárvore vem numa nova linha.
- As subárvores vazias são representadas por um hífen, convenientemente indentado.
- A função que escreve tem como argumentos a margem corrente e a tabulação.
- Na verdade, a função não escreve; cria sim uma cadeia de caracteres que há de ser escrita.
- A tabulação indica quanto avançamos na linha, ao passar para a subárvore.

Função indent

- É declarada na classe abstrata:

```
public abstract String indent(String margin, String tab);
```

- ... e implementada assim na classe **Empty<T>**:

```
public String indent(String margin, String tab)
{
    return margin + "-" + newLine;
}
```

- ... e assim na classe **Cons<T>**:

```
public String indent(String margin, String tab)
{
    String result = "";
    result += margin + root + newLine;
    result += left.indent(margin + tab, tab);
    result += right.indent(margin + tab, tab);
    return result;
}
```

A variável **newLine** representa o fim de linha de maneira portável, e vem declarada na classe abstrata:

```
protected static String newLine = System.getProperty("line.separator");
```

Experimentando `fromString` e `indent`

```
public static void testReadIndent()
{
    while (StdIn.hasNextLine())
    {
        String line = StdIn.readLine();
        Tree<Integer> t = Tree.fromString(line);
        StdOut.println(t.indent("", " "));
    }
}
```

```
$ java -cp .:../*: Tree E
()
```

```
-
```

```
(89())()
89
```

```
-
```

```
(15(6())(4()(5())))
15
6
```

```
-
```

```
4
5
```

```
-
```

```
-
```

```
(5(-7(10(666()())())(-100()())(6(17()())(88()()))))
```

```
5
```

```
-7
```

```
10
```

```
666
```

```
-
```

```
-
```

```
-
```

```
-100
```

```
-
```

```
-
```

```
6
```

```
17
```

```
-
```

```
-
```

```
88
```

```
-
```

```
-
```

Árvore diagramática

- Se na indentação escrevermos primeiro a subárvore direita, depois a raiz e depois a subárvore esquerda e rodarmos 90° para a direita, teremos um diagrama arborescente!
- Eis as funções:

Na classe `Tree<T>`

```
public abstract String diagram(String margin, String tab);
```

```
public String diagram(String margin, String tab)
{
    return margin + "-" + newLine;
```

Na classe `Empty<T>`

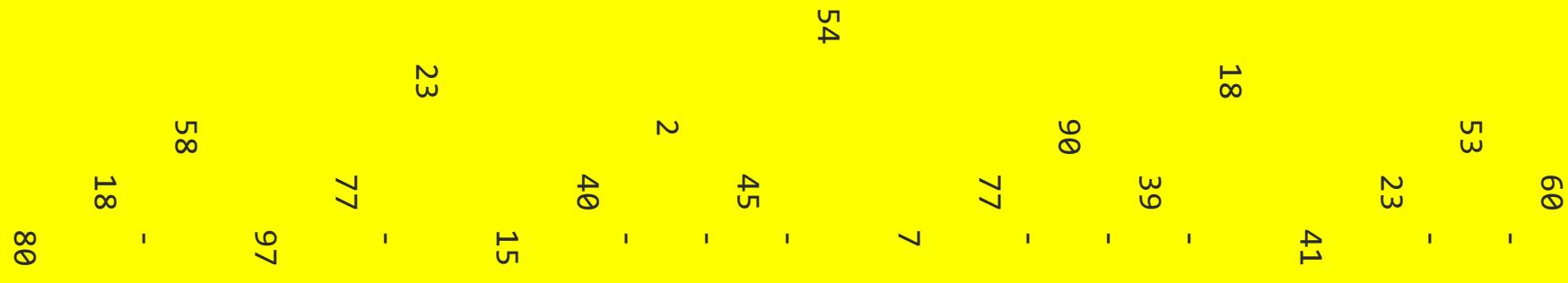
```
public String diagram(String margin, String tab)
{
    String result = "";
    result += right.diagram(margin + tab, tab);
    result += margin + root + newLine;
    result += left.diagram(margin + tab, tab);
    return result;
}
```

Na classe `Cons<T>`

Experimentando a função diagram

```
public static void testDiagram()
{
    Tree<Integer> t = new Empty<Integer>();
    while (!StdIn.isEmpty())
    {
        int x = StdIn.readInt();
        t = t.add(x);
    }
    StdOut.println(t.diagram("", "    "));
}
```

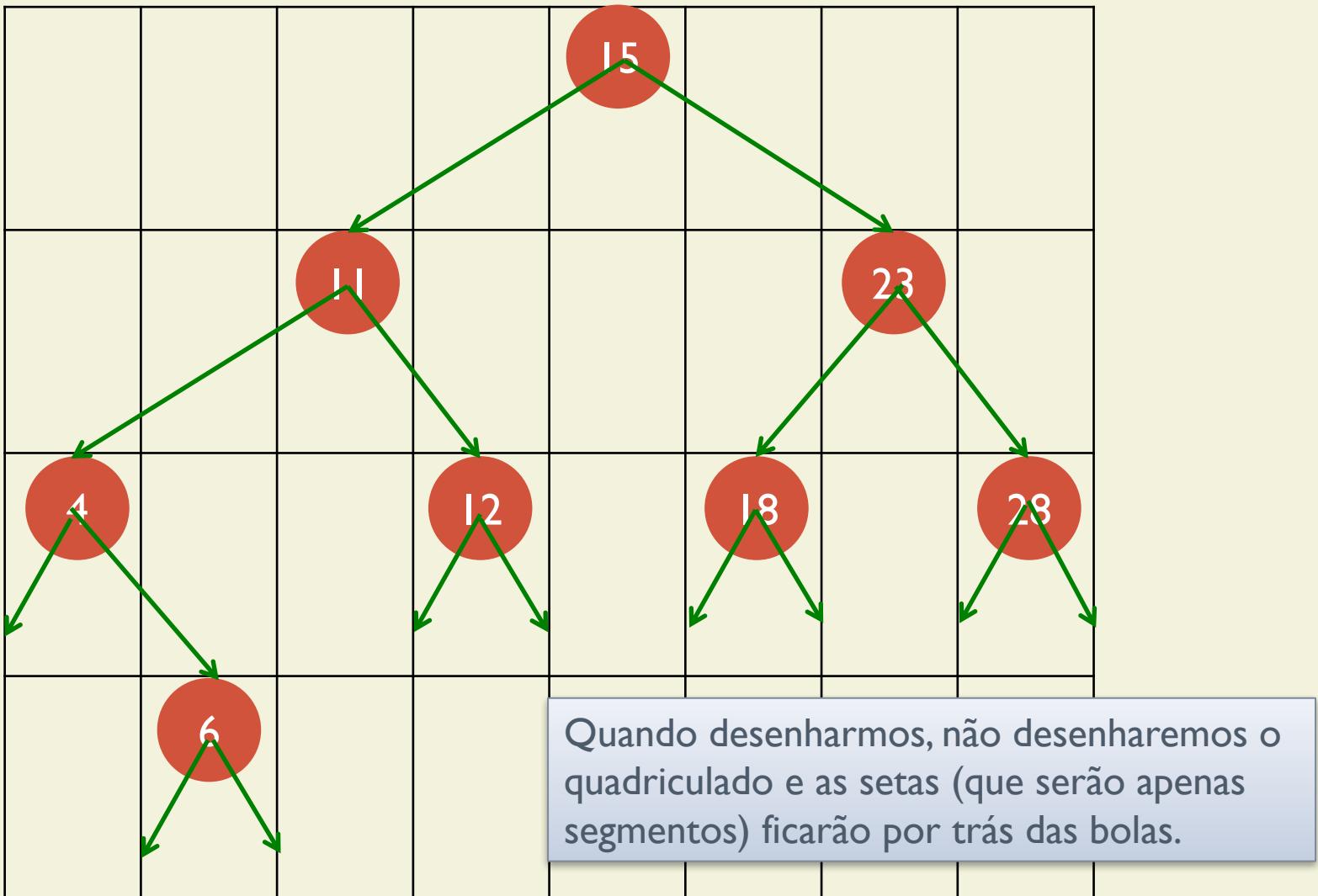
```
$ java ... Tree
54 23 18 58 90 2 53 18 77 40 23 77 39 45 60 80 7 15 41 97
```



Desenhando com o Processing

- Ganhando inspiração na função **diagram**, abalancemo-nos a desenhar árvores no Processing.
- Cada árvore será desenhada num retângulo cujo canto superior esquerdo é o ponto $\langle x_0, y_0 \rangle$.
- Cada nó ocupa um retângulo de largura **w** e altura **h**, num quadriculado de retângulos destes.
- O nó é um círculo de raio **r**, centrado horizontalmente no retângulo do nó e tangente por baixo ao lado superior do retângulo.
- Os arcos entre nós partem do centro do círculo e chegam ao ponto médio do lado superior do retângulo do outro nó.
- Os arcos para árvores vazias têm altura igual a **h/2** e vão até ao lado correspondente do retângulo.

Exemplos



Desenhando as bolas

- A bola correspondente à raiz da árvore é desenhada na quadricula da primeiro linha que corresponde ao tamanho da subárvore esquerda, considerando o quadriculado que começa no ponto $\langle x0, y0 \rangle$.
- O seu centro é o ponto $\langle xc, yc \rangle$, com $xc = x0 + sz * w + w/2$ e $yc = y0 + r$, onde sz é o tamanho da subárvore esquerda.
- A subárvore esquerda é desenhada depois no quadriculado que começa no ponto $\langle x0, y0 + h \rangle$.
- A subárvore direita é desenhada depois no quadriculado que começa no ponto $\langle x0 + (sz + l) * w \rangle$.

Desenhando os arcos

- Do centro de cada bola saem dois arcos, uma para cada subárvore.
- Se a subárvore não for vazia, o outro extremo do arco é o ponto médio do lado superior quadrícula da subárvore.
- Tratando-se da subárvore esquerda, esse extremo será o ponto $\langle xc - (szr+1)*w, y0+h \rangle$, onde szr é o tamanho da subárvore direita da subárvore esquerda (recordando que esta não é vazia).
- Tratando-se da subárvore direita, as contas são análogas.
- Se a subárvore for vazia, então, tratando-se da subárvore esquerda, o outro extremo do arco estará no ponto $\langle xc - w/2, yc+h/2 \rangle$ e tratando-se da subárvore direita, no ponto $\langle xc + w/2, yc+h/2 \rangle$.

Função draw

- É comprida, mas sistemática:

```
public void draw(PApplet p, double x0, double y0, double w,
                 double h, double r)
{
    double x = x0 + left.size() * w;
    double y = y0;
    double xc = x + w/2;
    double yc = y + r;

    // draw the arcs
    ...
    // draw the nodes
    ...

    // continue recursively
    left.draw(p, x0, y0 + h, w, h, r);
    right.draw(p, x+w, y0 + h, w, h, r);
}
```

Isto é a definição na classe **Cons<T>**. Na classe **Empty<T>**, a função não faz nada.

Função **draw**, desenhando os arcos

- Esquerda, direita, vazio, não vazio:

```
public void draw(PApplet p, double x0, double y0, double w,
                 double h, double r)
{
    ...
    // draw the arcs
    p.stroke(COLOR_LINES);
    p.fill(COLOR_LINES);
    if (left.isEmpty())
        p.line((float)xc, (float)yc,
               (float)x, (float)(yc+h/2));
    else
        p.line((float)xc, (float)yc,
               (float)(xc-w*(1+left.right().size())), (float)(y+h));

    if (right.isEmpty())
        p.line((float)xc, (float)yc,
               (float)(x+w), (float)(yc+h/2));
    else
        p.line((float)xc, (float)yc,
               (float)(xc+w*(1+right.left().size())), (float)(y+h));
    ...
}
```

NB: as funções da classe **PApplet** esperam **floats**, não **doubles**.
Por isso, convertemos no momento da chamada.

public final int COLOR_LINES = Colors.DARKRED;

É “só” acertar a aritmética...

Função **draw**, desenhando as bolas

- Esquerda, direita, vazio, não vazio:

```
public void draw(PApplet p, double x0, double y0, double w,  
    double h, double r)  
{  
    ...  
    // draw the nodes  
    p.fill(COLOR_VALUES);  
    p.stroke(COLOR_VALUES);  
    p.ellipseMode(PApplet.CENTER);  
    p.ellipse((float)xc, (float)yc, 2*(float)r, 2*(float)r);  
    p.textAlign(PApplet.CENTER, PApplet.CENTER);  
    p.fill(COLOR_TEXT);  
    p.text(value.toString(), (float)xc, (float)yc);    ...  
}
```

```
public final int COLOR_VALUES = Colors.BLACK;  
public final int COLOR_TEXT = Colors.YELLOW;
```

O sketch

- Em esquema:

```
public class TreeSketch extends PApplet
{
    public static final int FONT_SIZE = 18;
    public static final String FONT_NAME = "Helvetica";
    public static final int WIDTH = 1000;
    public static final int HEIGHT = 600;
    public static final int SIZE = 32; // size of tree

    public static final int LEFT_MARGIN = 10;
    public static final int TOP_MARGIN = 10;
    public static final int GRID_WIDTH = 12;
    public static final int GRID_HEIGHT = 48;
    public static final int NODE_RADIUS = 10;

    private int fontSize = FONT_SIZE;
    private int mywidth = WIDTH;
    private int myHeight = HEIGHT;
    private static int size = SIZE;

    private static int gridwidth = GRID_WIDTH;
    private static int gridHeight = GRID_HEIGHT;
    private static int nodeRadius = NODE_RADIUS;
```

```
    public static void getArgs(String[] args)
    {
        if (args.length > 0)
            size = Integer.parseInt(args[0]);
        if (args.length > 1)
            gridwidth = Integer.parseInt(args[2]);
        if (args.length > 2)
            gridHeight = Integer.parseInt(args[3]);
        if (args.length > 3)
            nodeRadius = Integer.parseInt(args[4]);
    }
```

```
    public static void main(String[] args)
    {
        getArgs(args);
        PApplet.main(new String[] { TreeSketch.class.getName() });
    }
```

O sketch, settings, setup, draw e update

```
public void settings()
{
    mywidth = 2 * LEFT_MARGIN + size * gridwidth;
    size(mywidth, myHeight);
}

public void setup()
{
    tree = new Empty<>();
    fontSize = 18*nodeRadius/10;
    textFont(createFont(FONT_NAME, fontSize));
    background(Colors.ALICEBLUE);
    tree.draw(this, LEFT_MARGIN, TOP_MARGIN, gridwidth, gridHeight, nodeRadius);
}

public void draw()
{
    background(Colors.ALICEBLUE);
    update();
    tree.draw(this, LEFT_MARGIN, TOP_MARGIN, gridwidth, gridHeight, nodeRadius);
}

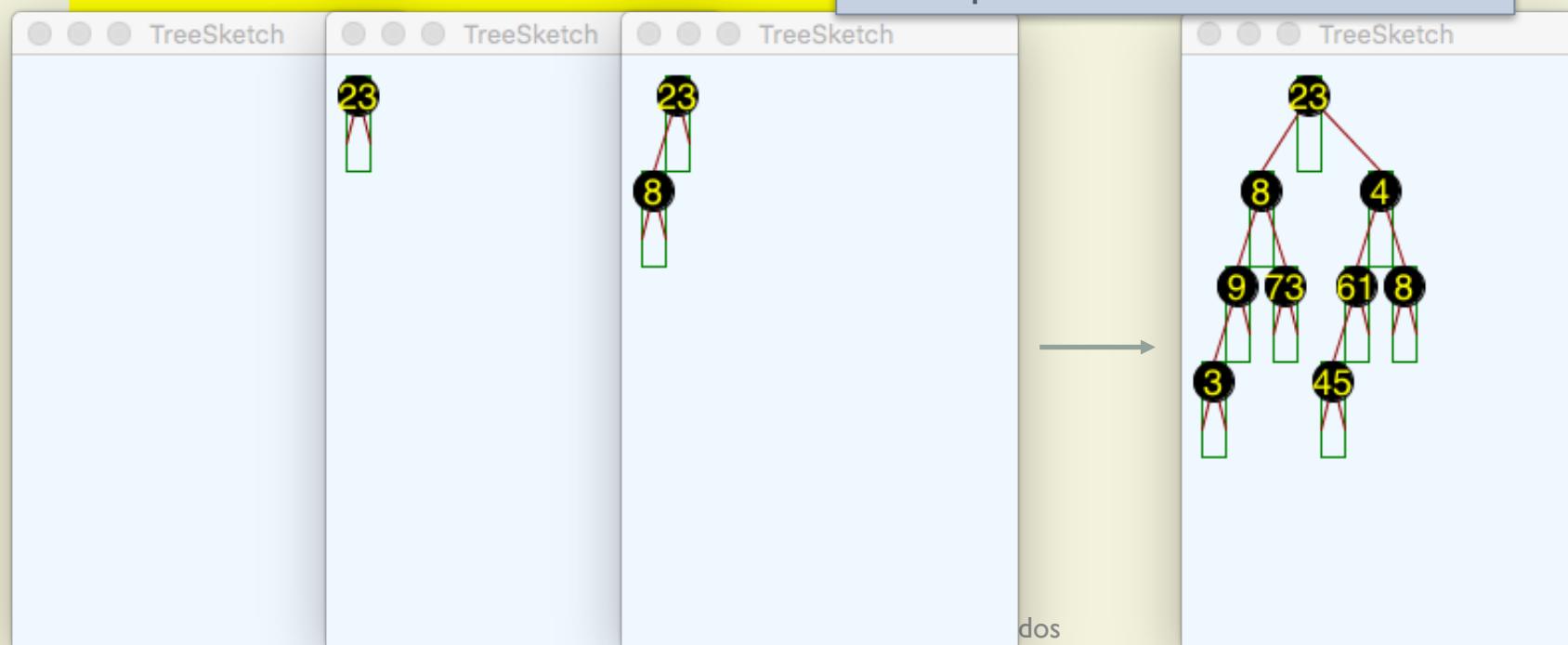
private void update()
{
    int x = StdIn.readInt();
    tree = tree.add(x);
}
```

O fundo é azul muito clarinho.

Testando

```
$ java -cp .:/.../*: TreeSketch  
23  
8  
4  
9  
61  
73  
8  
3  
45
```

Nota: este exemplo usa uma janela mais pequena do que a indicada nas páginas anteriores. Além disso, mostra a retângulo que corresponde a cada nó da árvore.



Funções left e right

- A função **draw** na classe **Cons<T>** usa funções **left()** e **right()**, e não apenas os membros de dados **left** e **right**, para aceder à subárvores das suas subárvores:

```
public void draw(PApplet p, double x0, double y0, double w,
                 double h, double r)
{
    ...
    if (left.isEmpty())
        ...
    else
        p.line((float)xc, (float)yc,
               (float)(xc-w*(1+left.right().size())), (float)(y+h));

    if (right.isEmpty())
        ...
    else
        p.line((float)xc, (float)yc,
               (float)(xc+w*(1+right.left().size())), (float)(y+h));
    ...
}
```

Se escrevêssemos apenas `1+left.right.size()` estaria mal, porque `left` é de tipo `Tree<T>`, e não `Cons<T>`.
(Logo, não tem membro de dados `right`.)

Idem para `1+right.left.size()`.

Programando left e right

- Na classe Tree<T>:

```
public abstract Tree<T> left();  
public abstract Tree<T> right();
```

- Na classe Empty<T>:

```
public Tree<T> left()  
{  
    throw new NoSuchElementException();  
}  
  
public Tree<T> right()  
{  
    throw new NoSuchElementException();  
}
```

- Na classe Cons<T>:

```
public Tree<T> left()  
{  
    return left;  
}  
  
public Tree<T> right()  
{  
    return right;  
}
```

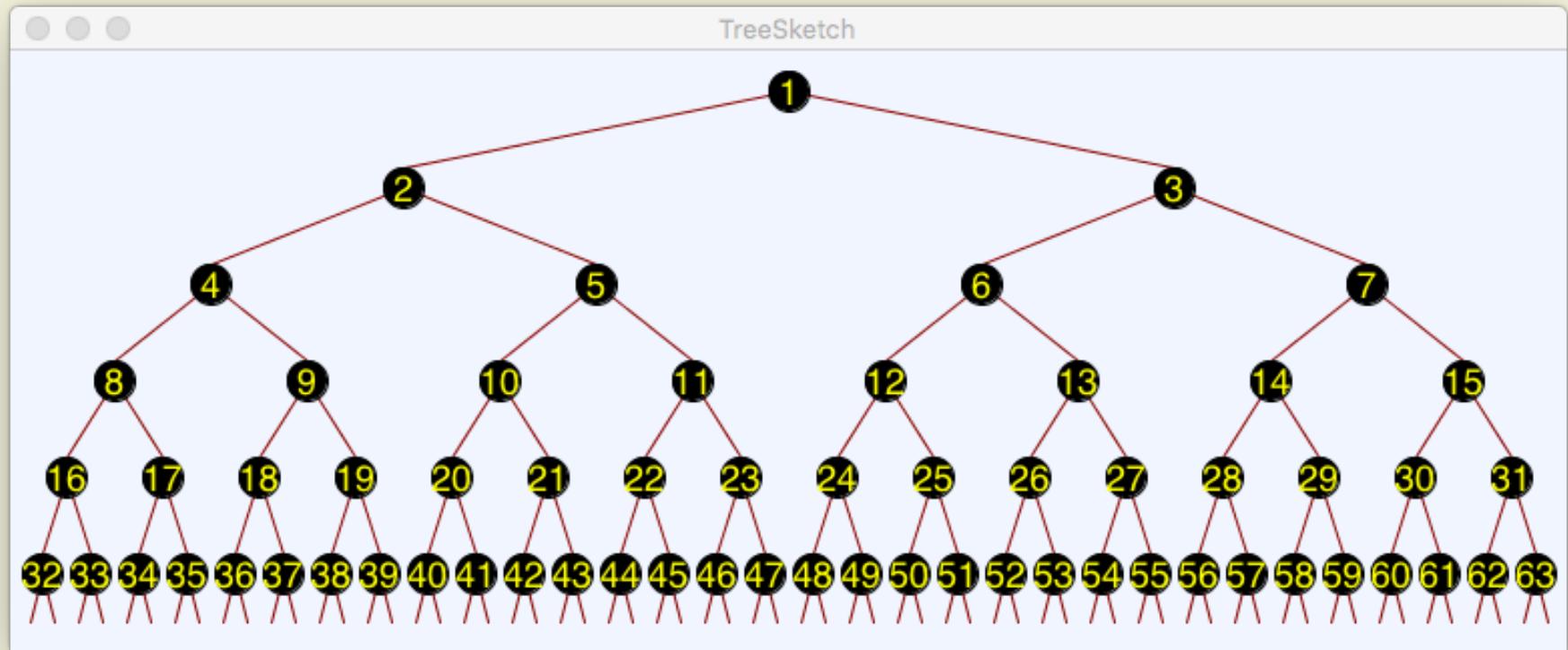
Apesar de serem triviais,
fazem falta, como vimos.

Exercícios

- Programar uma função booleana **isLeaf**, que dá true se a árvore for uma folha. Uma folha é uma árvore não vazia que cujas subárvore são ambas vazia.
- Programar uma função **countLeaves**, que conta as folhas.
- Programar uma função **deleteLeaves**, que elimina todas as folhas.
- Programar uma função **cut**, com uma argumento inteiro **x**, tal que **cut(x)** elimina todos os valores da árvore a partir do **x**-ésimo nível. O nível da raiz é zero.

Mais exercícios

- Programar uma função estática para criar uma árvore análoga à da figura, com um número dado de níveis:



- Programar uma função que calcula a “primeira” subárvore de uma árvore não vazia. Por definição, a “primeira” subárvore de uma árvore que não tem filho esquerdo é ela própria; tendo filho esquerdo, é a primeira do filho esquerdo.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 23

Árvores de busca

Árvores de busca

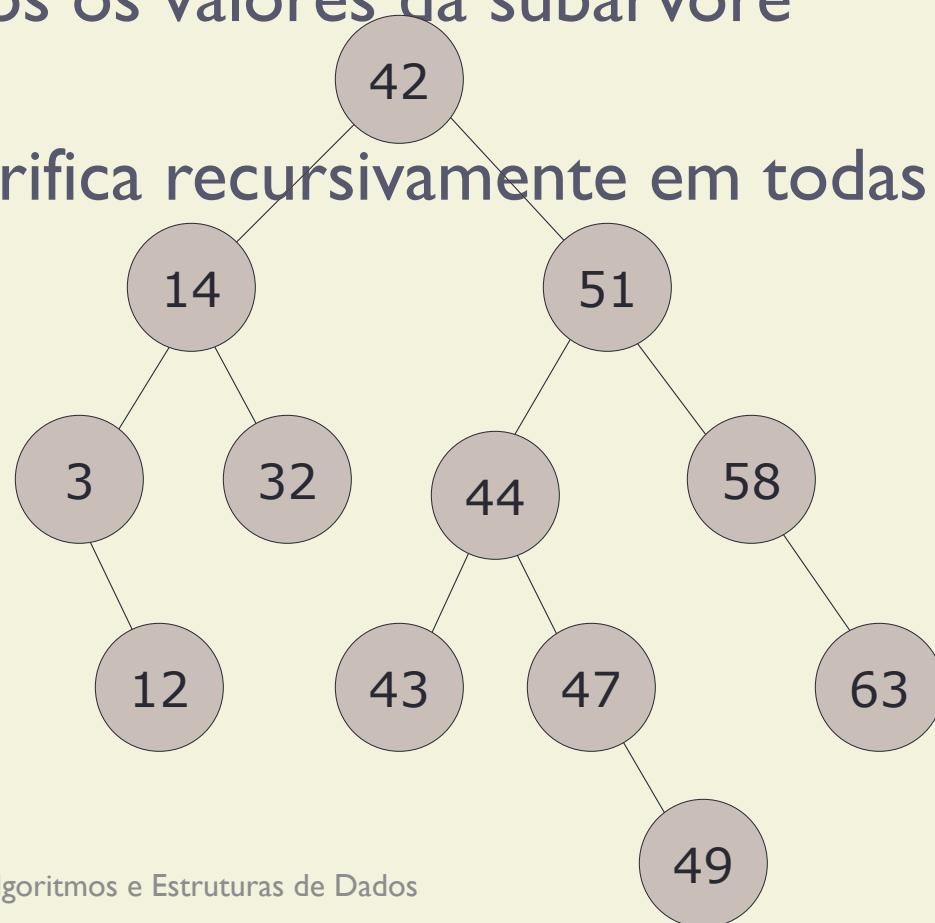
- Árvores imutáveis de busca.
- Implementação.
- Esquetes.



Árvores de busca

- Uma árvore de busca, em inglês **binary search tree**, é uma árvore binária tal que o valor da árvore é maior do que todos os valores da subárvore esquerda e menor do que todos os valores da subárvore direita...
- ... e o mesmo se verifica recursivamente em todas as subárvore.

Estruturalmente, as árvores de busca são árvores binárias como as outras. Apenas os valores estão distribuídos verificando aquela propriedade.



Árvores de busca, imutáveis

- A classe para as árvores de busca é muito parecida com a das árvores que já vimos.
- Todas as operações anteriores estarão disponíveis na classe das árvores de busca, algumas tal e qual, outras com implementações melhoradas.
- Além disso, há novas operações, que são próprias das árvores de busca.
- Teremos as três classes do costume: `Tree<T>`, `Empty<T>` e `Cons<T>`.

Colocaremos estas classes no pacote `bst.immutable`.

Classe abstrata Tree<T>

- O tipo paramétrico deve ser comparável e as árvores são iteráveis:

```
public abstract class Tree <T extends Comparable<T>>
    implements Iterable<T>
{
    public abstract T value();           // requires !isEmpty();
    public abstract int size();
    public abstract boolean isEmpty();
    public abstract boolean has(T x);

    public abstract Tree<T> left();     // requires !isEmpty();
    public abstract Tree<T> right();    // requires !isEmpty();

    public abstract Tree<T> put(T x);
    public abstract Tree<T> delete(T x);
    public abstract Tree<T> remove();   // requires !isEmpty();
    ...
}
```

A função **put** acrescenta um valor à árvore de busca; a função **delete** elimina o elemento com o valor dado da árvore de busca, se houver; a função **remove** elimina a raiz da árvore de busca.

Classe Empty<T>

- As seis primeiras funções são como antes.
- As funções **put**, **delete** e **remove** são novas:

```
public Tree<T> put(T x)
{
    return new Cons<>(x, this, this);
```

Ao acrescentar a uma árvore vazia um elemento com um valor dado, criamos uma árvore não vazia, com esse valor na raiz e mais nada.

```
public Tree<T> delete(T x)
{
    return this;
```

Ao retirar de uma árvore vazia um elemento com um valor dado, o resultado é uma árvore vazia.

```
public Tree<T> remove()
{
    throw new NoSuchElementException();
}
```

Remover a raiz de uma árvore vazia não faz sentido.

Classe Cons<T>

- Acrescentamos um membro de dados para guardar o tamanho da árvore:

```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    private final T value;
    private final Tree<T> left;
    private final Tree<T> right;
private final int size;

    public Cons(T value, Tree<T> left, Tree<T> right)
    {
        assert (left.isEmpty() || value.compareTo(left.value()) > 0) &&
               (right.isEmpty() || value.compareTo(right.value()) < 0);
        this.value = value;
        this.left = left;
        this.right = right;
        this.size = 1 + left.size() + right.size();
    }
}
```

Algumas operações, mais à frente, precisam de conhecer o tamanho da árvore. Seria pouco prático ter de recalcular de cada vez.

Buscabilidade

- Todas as árvores de busca verificam a seguinte propriedade, que constitui o invariante das árvores de busca:

```
public abstract boolean invariant();
```

Na classe abstrata.

```
public boolean invariant()
{
    return (left.isEmpty() || value.compareTo(left.value()) > 0) &&
           (right.isEmpty() || value.compareTo(right.value()) < 0) &&
           size == 1 + left.size() + right.size() &&
           left.invariant() &&
           right.invariant();
}
```

Na classe das árvores não vazias.

```
public boolean invariant()
{
    return true;
}
```

Na classe das árvores vazias.

Pós-condição do construtor

- À saída do construtor, assertamos que a árvore é uma árvore de busca:

```
public Cons(T value, Tree<T> left, Tree<T> right)
{
    assert (left.isEmpty() || value.compareTo(left.value()) > 0) &&
           (right.isEmpty() || value.compareTo(right.value()) < 0);
    this.value = value;
    this.left = left;
    this.right = right;
    this.size = 1 + left.size() + right.size();
    assert invariant();
}
```

Classe Cons<T>, seletores simples

- São todos como antes, exceto **size**, que agora se limita a consultar o membro de dados:

```
public T value()
{
    return value;
}

public int size()
{
    return size;
}

public boolean isEmpty()
{
    return false;
}
```

```
public Tree<T> left()
{
    return left;
}

public Tree<T> right()
{
    return right;
}
```

Classe Cons<T>, has

- Expectemos que a árvore tenha **x**, mas, se **x** for menor que a raiz, terá se o filho esquerdo tiver, e se **x** for maior que a raiz, terá se o filho direito tiver:

```
public boolean has(T x)
{
    boolean result = true;
    int cmp = x.compareTo(value);
    if (cmp < 0)
        result = left.has(x);
    else if (cmp > 0)
        result = right.has(x);
    return result;
}
```

Numa árvore de busca razoavelmente equilibrada, esta função tem comportamento logarítmico, claramente!

Classe Cons<T>, acrescentar um elemento

- Atenção à forma de falar: não é bem acrescentar à árvore um elemento com um valor dado mas sim construir uma nova árvore igual à outra, com mais um elemento, o qual terá esse valor.
- Mas só haverá um novo elemento se não existir já na árvore um elemento com o valor dado.
- Expectemos que a nova árvore seja igual à outra (porque a raiz é igual ao valor dado) mas se o valor dado for menor que a raiz, então criamos uma nova árvore igual à subárvore esquerda da outra acrescentada de um novo elemento com o valor dado, fazendo dessa nova árvore a subárvore esquerda do resultado; a raiz do resultado é a raiz da outra e a subárvore direita do resultado é a subárvore direita da outra.
- E se o valor dado for maior que a raiz é a mesma lenga-lenga, trocando “esquerda” por “direita” e “direita” por “esquerda”.
- Se o valor dado não é nem menor nem maior que a raiz então é igual à raiz e a nossa expectativa inicial ter-se-á confirmado, sendo a própria árvore o resultado da função.

Classe Cons<T>, put

- É semelhante ao has:

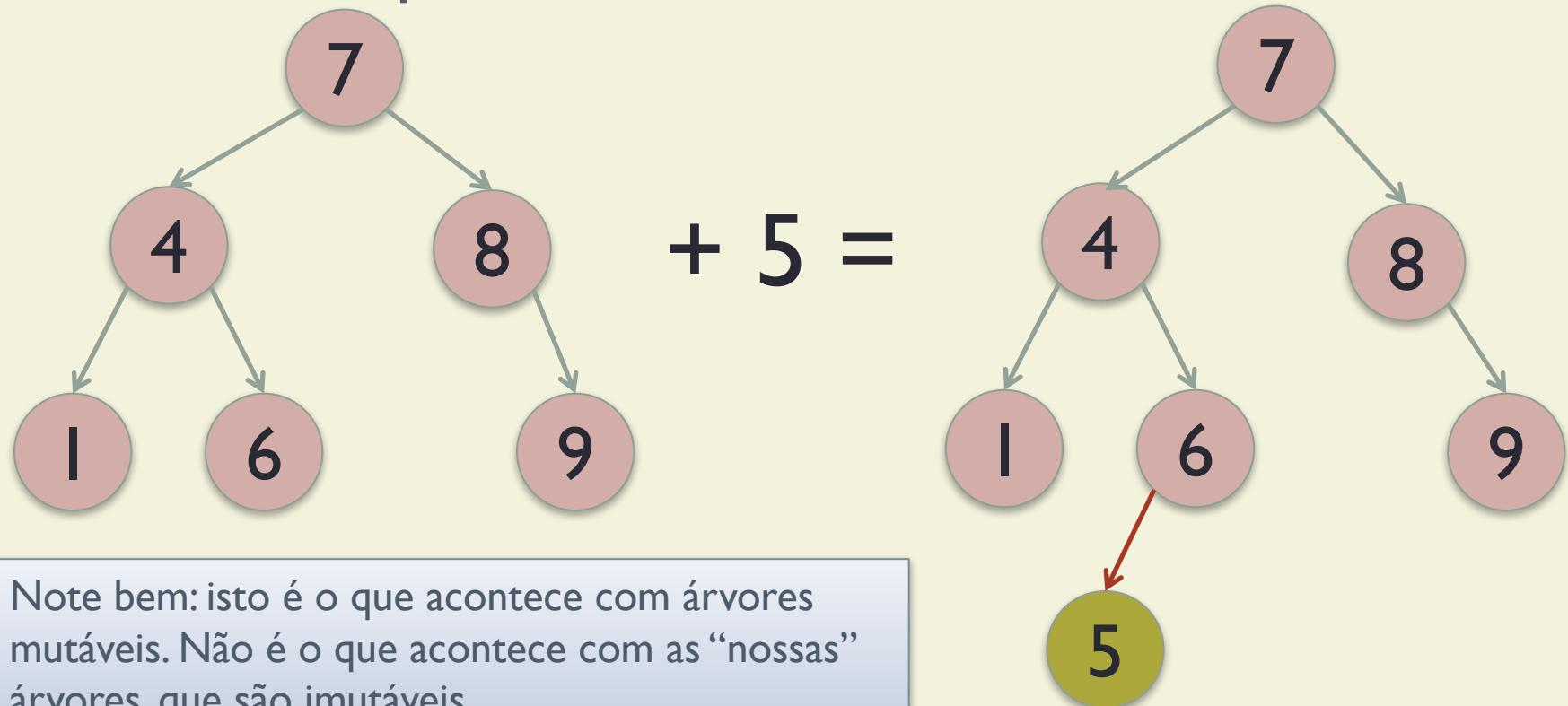
```
public Tree<T> put(T x)
{
    Tree<T> result = this;
    int cmp = x.compareTo(value);
    if (cmp < 0)
        result = new Cons<T>(value, left.put(x), right);
    else if (cmp > 0)
        result = new Cons<T>(value, left, right.put(x));
    return result;
}
```

Esta função é exemplar. Assegure-se de que a percebe completamente!

Tal como a função **has**, a função **put** tem comportamento logarítmico, desde que a árvore esteja razoavelmente equilibrada.

Acrescentar modificando

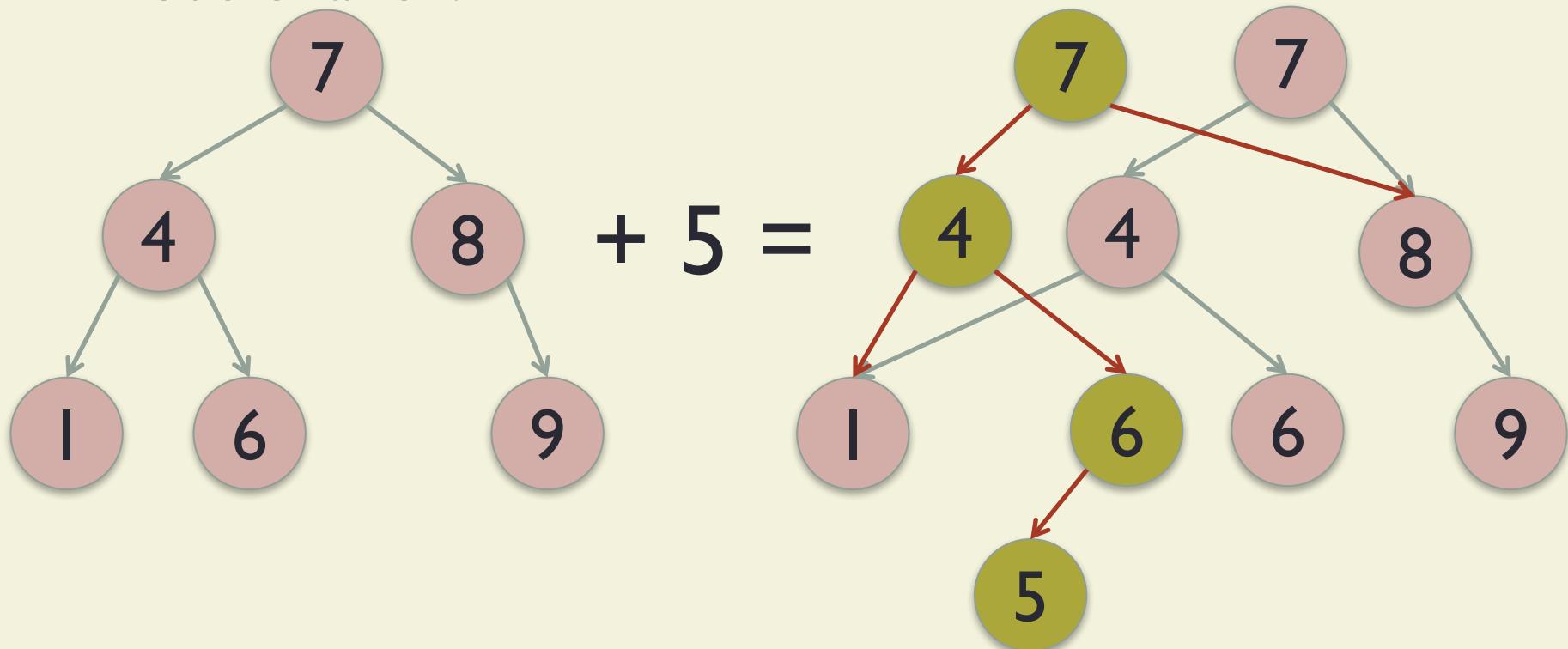
- Acrescentar modificando envolve percorrer a árvore e ligar o novo nó no local certo.
- Por exemplo:



Note bem: isto é o que acontece com árvores mutáveis. Não é o que acontece com as “nossas” árvores, que são imutáveis.

Acrescentar imutavelmente

- Acrescentar imutavelmente envolve criar uma nova árvore a cada nível.
- A cada nível, a nova árvore partilha a árvore do “outro lado”:



Classe Cons<T>, apagar um elemento

- Esta função é a mais trabalhosa de todas, na classe Cons<T>.
- Já vimos que na classe Empty<T> é trivial:

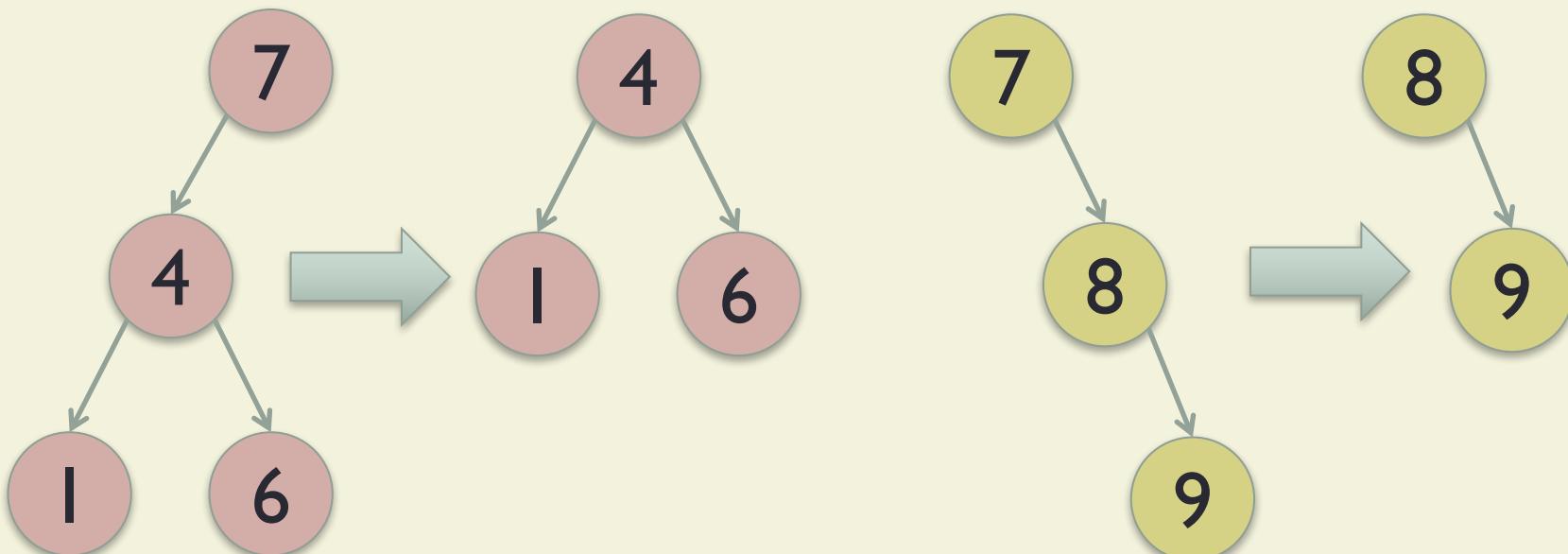
```
class Empty<T extends Comparable<T>> extends Tree<T>
{
    public Tree<T> delete(T x)
    {
        return this;
    }
}
```

- Em árvores não vazias, apagar é complicado, porque se o valor a apagar estiver no “meio” da árvore, temos que decidir com cuidado como ocupar o lugar que ficou vazio, por assim dizer.

Estar no “meio” da árvore significa, em rigor, que ambas as subárvore sãos não vazias.

Remover a raiz

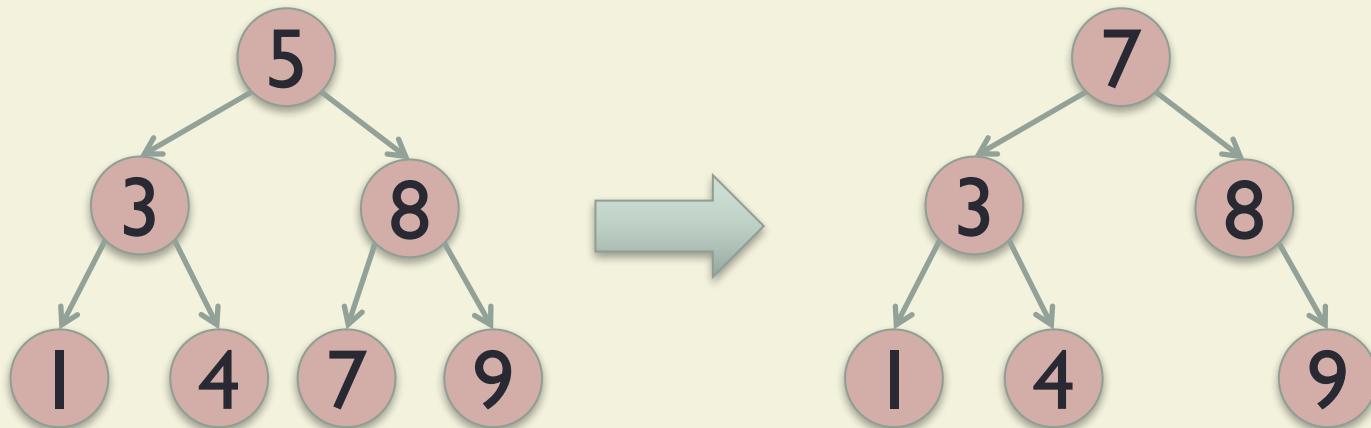
- Vejamos primeiro o problema de remover a raiz.
- Não é complicado, a não ser que um dos filhos (ou os dois) seja a árvore vazia.



Quer dizer: quando exatamente um dos filhos é a árvore vazia, o resultado é o outro filho. Se ambos os filhos forem vazios, o resultado é a árvore vazia (mas este caso pode ser apanhado pelo caso anterior).

Remover a raiz, caso complicado.

- Se ambos os filhos não são vazios, a nova raiz será o elemento com valor imediatamente superior ao valor da raiz eliminada, que é precisamente a raiz da primeira subárvore da subárvore direita:

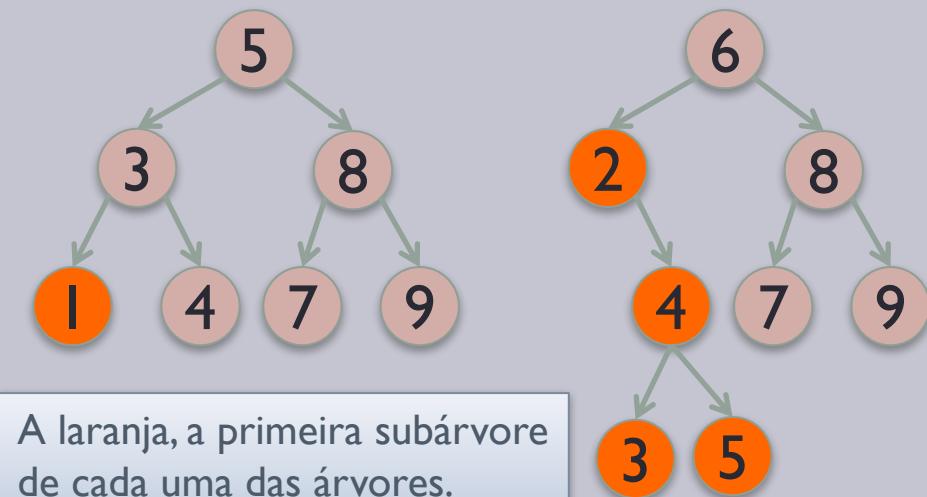


- A primeira subárvore do filho direito terá sempre um filho vazio (o filho esquerdo), pelo menos, e portanto a sua raiz pode ser removida usando a técnica da página anterior.

A primeira subárvore

- A primeira subárvore é a subárvore cuja raiz fica mais à esquerda, quando desenhamos a árvore (usando a função **draw**):

```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    ...
    public Tree<T> first()
    {
        Tree<T> result = this;
        if (!left.isEmpty())
            result = left.first();
        return result;
    }
    ...
}
```

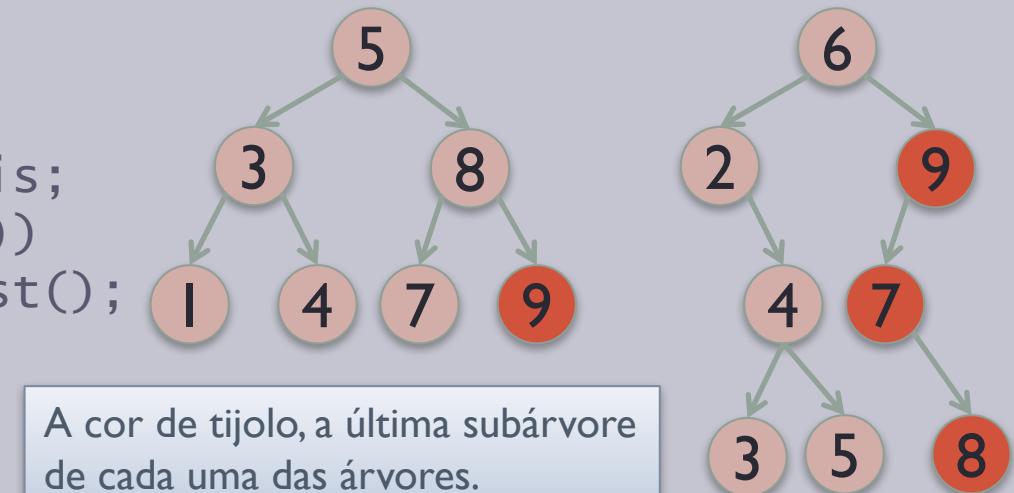


- Na classe **Empty<T>**, a função **first** lança uma exceção.
- Também haverá a função **last**, por simetria.

A última subárvore

- Por simetria, a última subárvore é a subárvore cuja raiz fica mais à direita, quando desenharmos a árvore:

```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    ...
    public Tree<T> last()
    {
        Tree<T> result = this;
        if (!right.isEmpty())
            result = right.last();
        return result;
    }
    ...
}
```



Remover a raiz da primeira subárvore

- Observe com atenção:

```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    ...
    protected Tree<T> removeFirst()
    {
        Tree<T> result = right; // if left is empty, result is right
        if (!left.isEmpty())
            result = new Cons<T>(value, left.removeFirst(), right);
        return result;
    }
    ...
}
```

Repare: o método é **protected**, e não **private**, porque é herdado da classe abstrata e não queremos que seja usado fora das classes derivadas. Na classe **Empty**, lança uma exceção.

- Quer dizer: se o filho esquerdo é vazio, então o resultado é o filho direito;
- Se não, o resultado é uma nova árvore, com a mesma raiz, com o mesmo filho direito, e com o filho esquerdo resultante de se remover a raiz da primeira subárvore do filho esquerdo (o tal que não era vazio).

Remover a raiz

- Já sabemos: se um dos filhos é vazio, o resultado é o outro filho; se ambos não são vazios, o resultado é uma árvore em que a raiz é a raiz da primeira da subárvore direita, em que o filho esquerdo é o mesmo e em que o filho direito é o resultado de remover a raiz da primeira subárvore do filho direito:

```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    ...
    public Tree<T> remove()
    {
        Tree<T> result;
        if (right.isEmpty())
            result = left;
        else if (left.isEmpty())
            result = right;
        else
            result = new Cons<T>(right.first().value(), left, right.removeFirst());
        return result;
    } ...
}
```

Método delete

- Podemos finalmente programar o método **delete**.
- Afinal, nem é assim tão complicado:

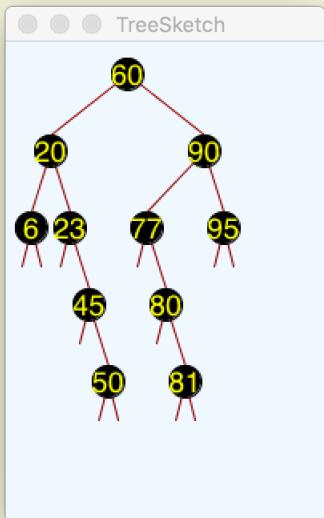
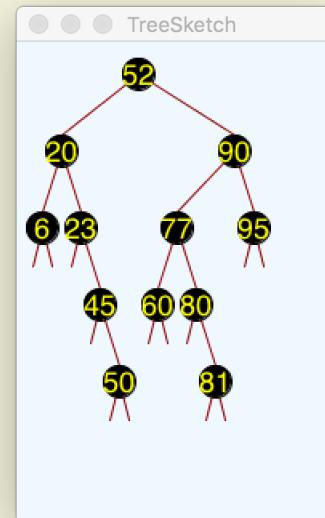
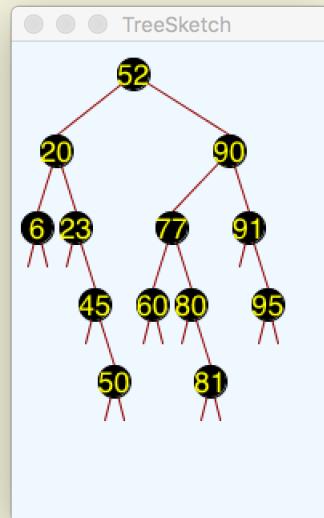
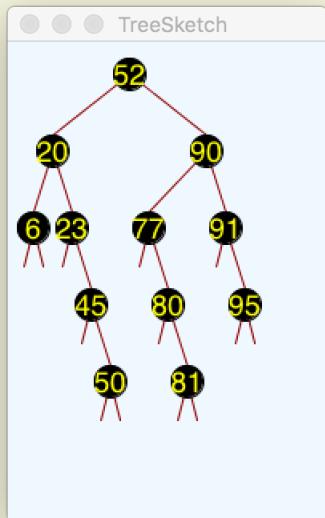
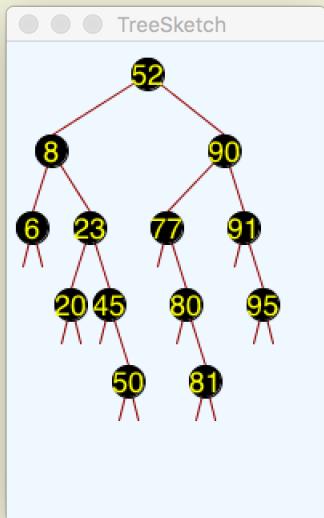
```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    ...
    public Tree<T> delete(T x)
    {
        Tree<T> result;
        int cmp = x.compareTo(value);
        if (cmp < 0)
            result = new Cons<T>(value, left.delete(x), right);
        else if (cmp > 0)
            result = new Cons<T>(value, left, right.delete(x));
        else
            result = remove();
        return result;
    }
    ...
}
```

Se o valor a remover é menor que a raiz, remove-se à esquerda; se é maior, remove-se à direita; se não, remove-se a raiz. Isto se a árvore for não vazia. Se for vazia, não se faz nada, na classe **Empty<T>**.

Esquete de teste

- O esquete desenha a árvore e a cada passo acrescenta ou elimina:

```
private void update()
{
    int x = StdIn.readInt();
    if (x > 0)
        tree = tree.put(x);
    else
        tree = tree.delete(-x);
}
```



`t = t.delete(20)`

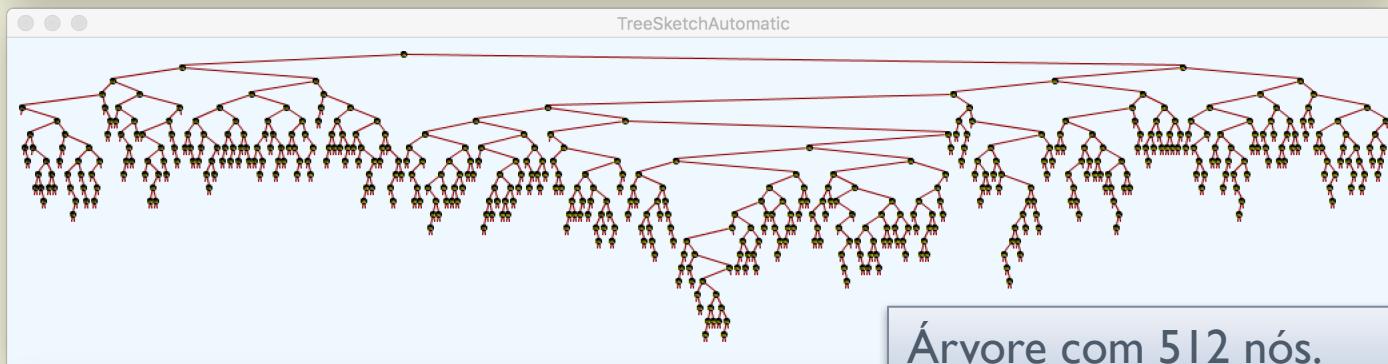
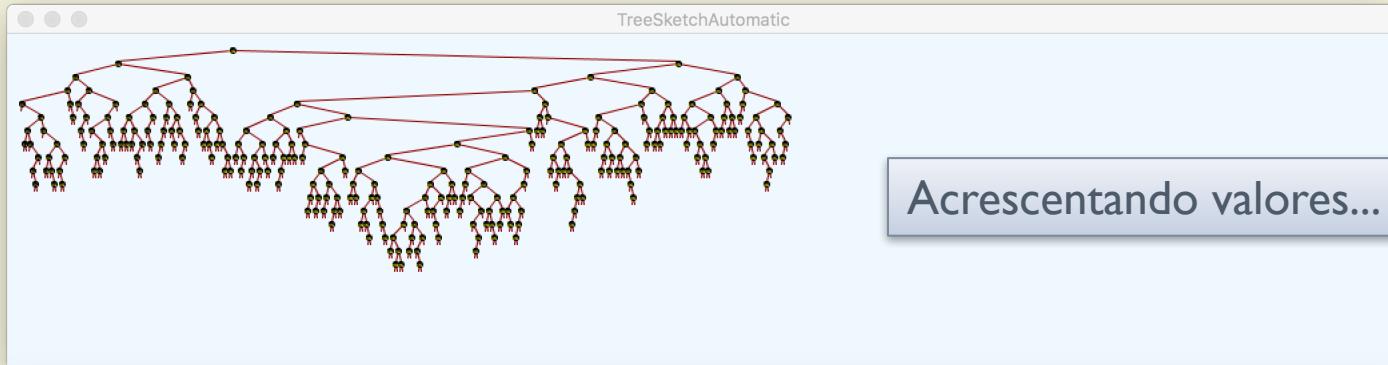
`t = t.put(60)`

`t = t.delete(91)`

`t = t.delete(52)`

Esquete de estresse

- Acrescentar muitos nós e depois eliminá-los:





UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 24

Árvores de busca, complementos

Árvores de busca, complementos



- Funções de ordem.
- Iteração.
- Atravessamento.
- Iteradores internos.
- Árvores de busca chave-valor.
- Tabelas ordenadas.

Funções de ordem

- Nas nossas árvores, os elementos estão “por ordem”. Por isso, é simples calcular o elemento mínimo, o elemento máximo, o elemento que está na posição de ordem k e o renque de um elemento:

```
public abstract class Tree<T extends Comparable<T>>
{
    ...
    public abstract T minimum();           // requires !isEmpty();
    public abstract T maximum();          // requires !isEmpty();
    public abstract T select(int k);     // requires 0 <= k < size();
    public abstract int rank(T x);      ...
}
```

Na classe **Empty<T>** as três primeiras lançam uma exceção e a última retorna sempre zero. Recorde que o renque de x é o número de elementos menores ou iguais a x .

Funções de ordem, Empty<T>

```
class Empty<T extends Comparable<T>> extends Tree<T>
{
    ...
    public T minimum()
    {
        throw new UnsupportedOperationException();
    }

    public T maximum()
    {
        throw new UnsupportedOperationException();
    }

    public T select(int k)
    {
        throw new UnsupportedOperationException();
    }

    public int rank(T x)
    {
        return 0;
    }
    ...
}
```

Funções de ordem, Cons<T>

- O mínimo é o valor da raiz da primeira subárvore e o máximo é o valor da raiz da última subárvore:

```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    ...
    public T minimum()
    {
        return first().value();
    }

    public T maximum()
    {
        return last().value();
    }
    ...
}
```

Select

- O nome **select** é consagrado para representar a função que devolve o elemento dada a sua “ordem”:

```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    ...
    public T select(int k)
    {
        T result = value;
        int t = left.size();
        if (k < t)
            result = left.select(k);
        else if (k > t)
            result = right.select(k-t-1);
        return result;
    }
    ...
}
```

Se a ordem **k** é menor que o tamanho da subárvore esquerda, basta considerar a subárvore esquerda; se é maior, então selecionamos na subárvore direita, na posição **k-t-1**, onde **t** é o tamanho da subárvore esquerda; se é igual, o resultado é o valor da raiz.

Rank

- A função **rank** é a inversa de **select**, para os elementos do contradomínio. Em geral, o renque de um valor **x**, presente ou não na árvore, é o número de elementos na árvore que são menores que **x**:

```
class Cons<T extends Comparable<T>> extends Tree<T>
{
    ...
    public int rank(T x)
    {
        int result;
        int cmp = x.compareTo(value);
        if (cmp < 0)
            result = left.rank(x);
        else if (cmp > 0)
            result = 1 + left.size() + right.rank(x);
        else
            result = left.size();
        return result;
    }
    ...
}
```

A explicação é análoga à da função **select**.

Chão, Teto

- O chão de x é o maior elemento presente que é menor ou igual a x ; o teto de x é o menor elemento presente que é maior ou igual a x .
- Em ambos os casos, não havendo, o resultado é **null**.
- Para o chão, se x é menor que a raiz, basta considerar a subárvore esquerda; se é maior, das duas uma: ou o chão de x não existe na subárvore direita (porque todos os elementos da subárvore direita são maiores que x) e o resultado é a raiz; ou existe e então é esse o resultado.
- Para o teto é a mesma coisa, trocando “esquerda” e “direita” e “menor” e “maior”.

Funções floor e ceiling

- Na classe abstrata:

```
public abstract class Tree<T extends Comparable<T>>
{
    ...
    public abstract T floor(T x);
    public abstract T ceiling(T x);
    ...
}
```

- Na classe **Empty<T>**:

```
class Empty<T extends Comparable<T>> extends Tree<T>
{
    ...
    public T floor(T x)
    {
        return null;
    }

    public T ceiling(T x)
    {
        return null;
    }
    ...
}
```

Funções floor e ceiling, na classe Cons<T>

- Ambas implementam diretamente a explicação:

```
public T floor(T x)
{
    T result = value;
    int cmp = x.compareTo(value);
    if (cmp < 0)
        result = left.floor(x);
    else if (cmp > 0)
    {
        T t = right.floor(x);
        if (t != null)
            result = t;
    }
    return result;
}
```

```
public T ceiling(T x)
{
    T result = value;
    int cmp = x.compareTo(value);
    if (cmp > 0)
        result = right.ceiling(x);
    else if (cmp < 0)
    {
        T t = left.ceiling(x);
        if (t != null)
            result = t;
    }
    return result;
}
```

Iterador de árvore

- Declarámos que a classe **Tree<T>** é iterável; portanto tem de fornecer um método **iterator**, que devolve um objecto **Iterator<T>**:

```
public abstract class Tree<T extends Comparable<T>>
    implements Iterable<T>
{
    ...
    public Iterator<T> iterator() { ... }
}
```

- Mais tarde, sendo **t** uma árvore, poderemos escrever, por exemplo:

```
for (T x : t)
    stdOut.print(x);
```

Pilha de árvores

- Usaremos uma pilha para guardar as subárvore ainda não processadas:

```
public abstract class Tree<T extends Comparable<T>>
    implements Iterable<T>
{
    ...
    private class TreeIterator implements Iterator<T>
    {
        private Stack<Tree<T>> i;

        public TreeIterator() { ... }

        public boolean hasNext() { ... }

        public T next() { ... }
    }
}
```

Classe Treeliterator

- No construtor, empilhamos a árvore, se a árvore não for vazia:

```
public TreeIterator()  
{  
    i = new Stack<>();  
    if (!Tree.this.isEmpty())  
        i.push(Tree.this);  
}
```

Note bem: acedemos à árvore com **Tree.this**. Na classe Treeliterator, a expressão **this** refere o próprio iterador, não a árvore.

- O iterador terminará quando a pilha ficar vazia:

```
public boolean hasNext()  
{  
    return !i.isEmpty();  
}
```

Note bem: todas as árvores na pilha serão não vazias.

Treelterminator.next

- O próximo elemento no iterador é a raiz da árvore que está no topo da pilha, se esta não tiver subárvore esquerda; se tiver, o próximo elemento é, recursivamente, a raiz de subárvore esquerda;

```
public T next()
{
    Tree<T> t = i.pop();
    T result = t.value();
    if (!t.left().isEmpty())
    {
        i.push(new Cons<T>(result, new Empty<T>(), t.right()));
        i.push(t.left());
        result = next();
    }
    else if (!t.right().isEmpty())
        i.push(t.right());
    return result;
}
```

Se a subárvore esquerda não for vazia, empilha-se a subárvore direita com a raiz (para serem processadas mais tarde) e depois empilha-se a subárvore esquerda, e avança-se recursivamente (anulando a primeira escolha de **result**).

Função de teste

```
public static void testOrderMethods()  
{
```

```
    Tree<Integer> t = new Empty<Integer>();  
    int[] numbers = StdIn.readAllInts();  
    for (int x : numbers)  
        t = t.put(x);  
    StdOut.println(t);  
    for (int x : t)  
        StdOut.print(" " + x);  
    StdOut.println();  
    int min = t.minimum();  
    int max = t.maximum();  
    int median = t.select(t.size() / 2);  
    StdOut.printf("Min, max, median: %d %d %d\n", min, max, median);  
    StdOut.printf("(min + max) / 2 = %d\n", (min + max) / 2);  
    int r1 = t.rank(min - 1);  
    int r2 = t.rank(max + 1);  
    int r3 = t.rank((min + max) / 2);  
    StdOut.printf("Ranks: %d %d %d\n", r1, r2, r3);  
    Integer f1 = t.floor(min - 1);  
    Integer f2 = t.floor(max + 1);  
    Integer f3 = t.floor((min + max) / 2);  
    StdOut.printf("Floors: %d %d %d\n", f1, f2, f3);  
    Integer c1 = t.ceiling(min - 1);  
    Integer c2 = t.ceiling(max + 1);  
    Integer c3 = t.ceiling((min + max) / 2);  
    StdOut.printf("Ceilings: %d %d %d\n", c1, c2, c3);  
}
```

Esta função de teste põe na árvore os números lidos da consola e depois exerce as diversas funções de ordem e o iterador.

```
$ java ... Tree  
65 12 8 19 71 43 95 12 6 51 82 84 20 28  
→(65(12(8(6())())())(19()(43(20()(28())())(5  
1())())))(71()(95(82()(84())())())  
→ 6 8 12 19 20 28 43 51 65 71 82 84 95  
Min, max, median: 6 95 43  
(min + max) / 2 = 50  
Ranks: 0 13 7  
Floors: null 95 43  
Ceilings: 6 null 51
```

Atravessamento em profundidade

- O iterador *visita* a árvore, pela ordem dita **entreordem**: primeiro *visita* subárvore esquerda, depois *observa* a raiz, depois *visita* a subárvore direita.
- Retocando a função **next**, poderíamos atravessar em **pré-ordem**: primeiro *observar* a raiz, depois *visitar* subárvore esquerda, depois *visitar* a subárvore direita.
- Ou então em **pós-ordem**: primeiro *visitar* a subárvore esquerda, depois *visitar* a subárvore direita, depois *observar* a raiz.
- Estes são os *atravessamentos em profundidade*.

Atravessamento em largura

- Atravessar uma árvore em largura é observar a raiz, depois as raízes das subárvore penduradas na raiz, depois as raízes das subárvore penduradas na raiz, e assim sucessivamente.
- Programemos um iterador para isso:

```
private class TreeIteratorwide implements Iterator<T>
{
    ...
}
```

Fila de árvores

- No atravessamento em largura, usamos uma fila de árvores no iterador:

```
public abstract class Tree<T extends Comparable<T>>
    implements Iterable<T>
{
    ...
    private class TreeIteratorwide implements Iterator<T>
    {
        private Queue<Tree<T>> i;

        TreeIteratorwide()
        {
            i = new Queue<>();
            if (!Tree.this.isEmpty())
                i.enqueue(Tree.this);
        }

        public boolean hasNext()
        {
            return !i.isEmpty();
        }
        ...
    }
}
```

TreeIteratorWide.next

- O próximo elemento no iterador é o valor da raiz da árvore que está na frente da fila.
- De cada vez que o iterador avança, sai a árvore que estava na frente da fila e entram na fila as subárvore, desde que não sejam vazias:

```
public T next()
{
    Tree<T> t = i.dequeue();
    T result = t.value();
    if (!t.left().isEmpty())
        i.enqueue(t.left());
    if (!t.right().isEmpty())
        i.enqueue(t.right());
    return result;
}
```

Testando os iteradores

- Eis uma função de teste que testa os dois iteradores:

```
public static void testIterators()
{
    Tree<Integer> t = new Empty<Integer>();
    int[] numbers = StdIn.readAllInts();
    for (int x : numbers)
        t = t.put(x);
    StdOut.println(t);

    StdOut.println(t.diagram("", " "));

    Iterator<Integer> itDeep = t.iterator();
    while (itDeep.hasNext())
        StdOut.print(" " + itDeep.next());
    StdOut.println();

    Iterator<Integer> itwide = t.iteratorwide();
    while (itwide.hasNext())
        StdOut.print(" " + itwide.next());
    StdOut.println();
}
```

```
$ java ... Tree
65 12 8 19 71 43 95 12 6 51 82 84 20 28
(65(12(8(6())())())(19()(43(20()(28())())
(51())())))(71()(95(82()(84())())())()))
      -
      95
      -
      84
      -
      82
      -
      71
      -
      65
      -
      51
      -
      43
      -
      28
      -
      20
      -
      19
      -
      12
      -
      8
      -
      6
      -
      6 8 12 19 20 28 43 51 65 71 82 84 95
      65 12 71 8 19 95 6 43 82 20 51 84 28
$
```

Iteradores internos

- Os objetos de tipo **Iterator<T>** são chamados iteradores *externos*: enumeram os elementos de uma coleção, colocando-os à disposição de algum troço de código.
- Inversamente, chamamos iteradores *internos* às funções das coleções que aplicam a cada elemento da coleção uma função passada em argumento.
- Observámos isso na classe **Queue<T>**, por exemplo, com a função **visit**:

```
public class Queue<T> implements Iterable<T>
{
    ...
    public void visit(Consumer<T> action)
    {
        for (Node p = first; p != null; p = p.next)
            action.accept(p.value);
    }
    ...
}
```

A função **visit** é um iterador interno.

Iteradores internos para árvores

- Implementamos três iteradores internos, que visitam a árvore em profundidade:
 - O iterador **em pré-ordem**, que primeiro atua sobre o valor da raiz, depois visita a subárvore esquerda e depois visita a subárvore direita.
 - O iterador **em entreordem**, que primeiro visita a subárvore esquerda, depois atua sobre o valor da raiz e depois visita a subárvore direita.
 - O iterador **em pós-ordem**, que primeiro visita a subárvore esquerda, depois visita a subárvore direita e depois atua sobre o valor da raiz:

```
public abstract void preorder(Consumer<T> action);  
public abstract void inorder(Consumer<T> action);  
public abstract void postorder(Consumer<T> action);
```

Estas são as declarações na classe abstrata.

Iteradores internos, na classe Empty<T>

- Na classe **Empty<T>**, os iteradores não fazem nada:

```
public void preorder(Consumer<T> action)
{
    // Nothing to do
}

public void inorder(Consumer<T> action)
{
    // Nothing to do
}

public void postorder(Consumer<T> action)
{
    // Nothing to do
}
```

Iteradores internos, na classe Cons<T>

- Na classe **Cons<T>**, os iteradores fazem o que definição diz:

```
public void preorder(Consumer<T> action)
{
    action.accept(value);
    left.preorder(action);
    right.preorder(action);
}

public void inorder(Consumer<T> action)
{
    left.inorder(action);
    action.accept(value);
    right.inorder(action);
}

public void postorder(Consumer<T> action)
{
    left.postorder(action);
    right.postorder(action);
    action.accept(value);
}
```

Testando os iteradores internos

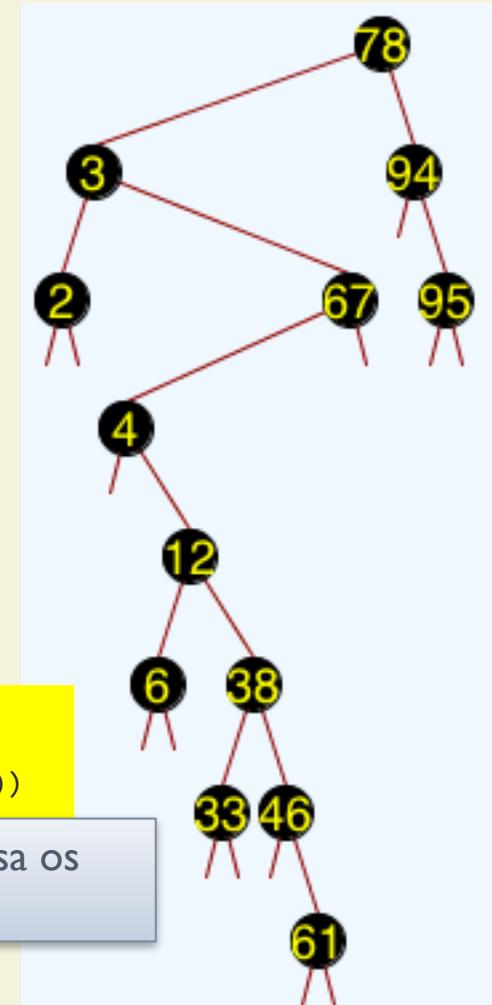
- Eis uma função de teste:

```
public static void testIteratorsInternal()
{
    Tree<Integer> t = new Empty<Integer>();
    int[] numbers = StdIn.readAllInts();
    for (int x : numbers)
        t = t.put(x);
    StdOut.println(t);

    t.preorder(x -> StdOut.print(" " + x));
    StdOut.println();
    t.inorder(x -> StdOut.print(" " + x));
    StdOut.println();
    t.postorder(x -> StdOut.print(" " + x));
    StdOut.println();
}
```

```
$ java -ea -cp .:../..//* Tree
78 3 67 4 12 94 2 38 6 46 33 95 61
(78(3(2())())(67(4())(12(6())())(38(33())())(46()(61()())))))(())(94()(95()()))
78 3 2 67 4 12 6 38 33 46 61 94 95
2 3 4 6 12 33 38 46 61 67 78 94 95
2 6 33 61 46 38 12 4 67 3 95 94 78
```

A função **inorder** é a que processa os valores por ordem crescente.



Tabelas ordenadas

- Talvez a principal utilização das árvores de busca seja a implementação de tabelas ordenadas.
- Uma tabela ordenada é uma tabela em que o iterador da chaves despeja as chaves **por ordem**.
- Aqui, **por ordem** significa pela ordem representada pela função **compareTo** na classe das chaves.

Classe TableSorted<K,V>

- Eis o invólucro:

```
public class SortedTable<K extends Comparable<K>, V>
    implements Table<K, V>
{
    ...
}
```

- Recordemos a interface **Table<K,V>**:

```
public interface Table<K, V> extends Iterable<K>
{
    public V get(K key);
    public void put(K key, V value);
    public void delete(K key);
    public boolean has(K key);
    public boolean isEmpty();
    public int size();
}
```

Classe KeyValue<K,V>

- Convém-nos uma classe para representar pares chave-valor:

```
class KeyValue<K extends Comparable<K>, V>
    implements Comparable<KeyValue<K, V>>
{
    public final K key;
    public final V value;

    public KeyValue(K key, V value)
    {
        this.key = key;
        this.value = value;
    }

    public KeyValue(K key)
    {
        this.key = key;
        this.value = null;
    }

    @Override public String toString()
    {
        return "[" + key.toString() + "," + value.toString() + "]";
    }
}
```

Na verdade, é uma variante da classe **Pair<K,V>**, estudada anteriormente. As diferenças residem nas duas funções adicionais da página seguinte e também no segundo construtor, em que só é dada a chave. (Veremos a utilização disto daqui a pouco.)

Comparação e igualdade de chaves-valor

- Comparar um par chave-valor significa comparar as chaves:

```
public int compareTo(KeyValue<K, V> other)
{
    return key.compareTo(other.key);
}
```

- As chaves são únicas, em cada tabela, e portanto, a igualdade dos pares chave-valor é a igualdade das chaves:

```
@SuppressWarnings("unchecked")
@Override
public boolean equals(Object other)
{
    return other != null &&
           other instanceof KeyValue &&
           (this == other || key.equals(((KeyValue<K, V>)other).key));
}
```

Implementação da tabela ordenada

- Uma tabela ordenada tem “lá dentro” uma árvore de busca de pares chave-valor:

```
public class SortedTable<K extends Comparable<K>, V>
    implements Table<K, V>
{
    private Tree<KeyValue<K, V>> tree = new Empty<>();
    ...
}
```

- Mais adiante, quando estudarmos outros tipos de árvores de busca mais eficientes, bastará mudar o tipo do membro de dados **tree**.

Funções get e put, na tabela

- Ei-las:

```
public V get(K key)
{
    V result = null;
    KeyValue<K, V> k = new KeyValue<>(key);
    Tree<KeyValue<K, V>> z = tree.get(k);
    if (!z.isEmpty())
        result = z.value().value;
    return result;
}

// Notes for put:
// we delete first, in order to allow for updating the value of the given key,
// if the key exists already.
// We check if the key exists before deleting, because we know that deleting
// involves reconstructing the tree, even if the key does not exist.

public void put(K key, V value)
{
    assert(value != null);
    if (has(key))
        delete(key);
    tree = tree.put(new KeyValue<>(key, value));
}
```

As outras funções

- As outras funções meramente encaminham a chamada para a árvore:

```
public void delete(K key)
{
    tree = tree.delete(new KeyValue<>(key));
}
```

```
public boolean has(K key)
{
    return tree.has(new KeyValue<>(key));
}
```

```
public boolean isEmpty()
{
    return tree.isEmpty();
}
```

```
public int size()
{
    return tree.size();
}
```

Recorde que as tabelas de dispersão de encadeamento aberto não têm delete.
Estas aqui têm!

O iterador das chaves

- Como de costume, usa uma classe interna:

```
public Iterator<K> iterator()
{
    return new Keys();
}
```

Esta função **iterator** é obrigatória, porque a classe **SortedTable<K,V>** declara que implementa a interface **Table<K,V>** e esta declara que estende a interface **Iterable<K>**.

```
public class Keys implements Iterator<K>
{
    private Iterator<KeyValue<K, V>> it = tree.iterator();
```

```
public boolean hasNext()
{
    return it.hasNext();
}
```

Este iterador meramente adapta o iterador da árvore, que percorre a árvore “da esquerda para a direita”.

```
public K next()
{
    return it.next().key;
}
```

```
public void remove()
{
    throw new UnsupportedOperationException();
}
```

Outros iteradores

- Em geral, nas tabelas, convém ter o iterador das chaves, o iterador dos valores e o iterador dos pares:

```
public Iterator<K> keys()
{
    return new Keys();
}

public Iterator<V> values()
{
    return new Values();
}

public Iterator<KeyValue<K, V>> pairs()
{
    return tree.iterator();
}

public class Values implements Iterator<V>
{
    private Iterator<KeyValue<K, V>> it = tree.iterator();

    public boolean hasNext()
    {
        return it.hasNext();
    }

    public V next()
    {
        return it.next().value;
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}
```

O iterador **keys** é igual ao iterador **iterator**; o iterador **values** é análogo ao iterador **iterator**, mas dá o valor, em vez da chave; o iterador **pairs** é o iterador da árvore.

Função de teste

- A função de teste é análoga à das outras tabelas:

```
public static void testSortedTable()
{
    SortedTable<Integer, String> st = new SortedTable<>();
    while (!StdIn.isEmpty())
    {
        String cmd = StdIn.readString();
        if ("p".equals(cmd))
        {
            int x = StdIn.readInt();
            String s = StdIn.readString();
            st.put(x, s);
        }
        else if ("g".equals(cmd))
        {
            int x = StdIn.readInt();
            String s = st.get(x);
            StdOut.println(s);
        }
        else if ("d".equals(cmd))
        {
            int x = StdIn.readInt();
            st.delete(x);
        }
        else if ("i".equals(cmd))
        {
            for (int x : st)
                StdOut.print(" " + x);
            StdOut.println();
            for (Iterator<String> it = st.values(); it.hasNext())
                StdOut.print(" " + it.next());
            StdOut.println();
            for (Iterator<KeyValue<Integer, String>> it = st.pairs(); it.hasNext())
                StdOut.print(" " + it.next());
            StdOut.println();
        }
        else if ("s".equals(cmd))
        {
            StdOut.println(st);
            StdOut.println("size: " + st.size());
            StdOut.println("isEmpty? " + st.isEmpty());
        }
    }
}
```

```
$ java ... SortedTable
p 66 gggg
p 45 aaaa
p 90 ssss
s
([66,gggg]([45,aaaa]())()([90,ssss]())())
size: 3
isEmpty? false
i
45 66 90
aaaa gggg ssss
[45,aaaa] [66,gggg] [90,ssss]
d 45
i
66 90
gggg ssss
[66,gggg] [90,ssss]
g 67
null
g 66
gggg
```



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 25

Outras árvores

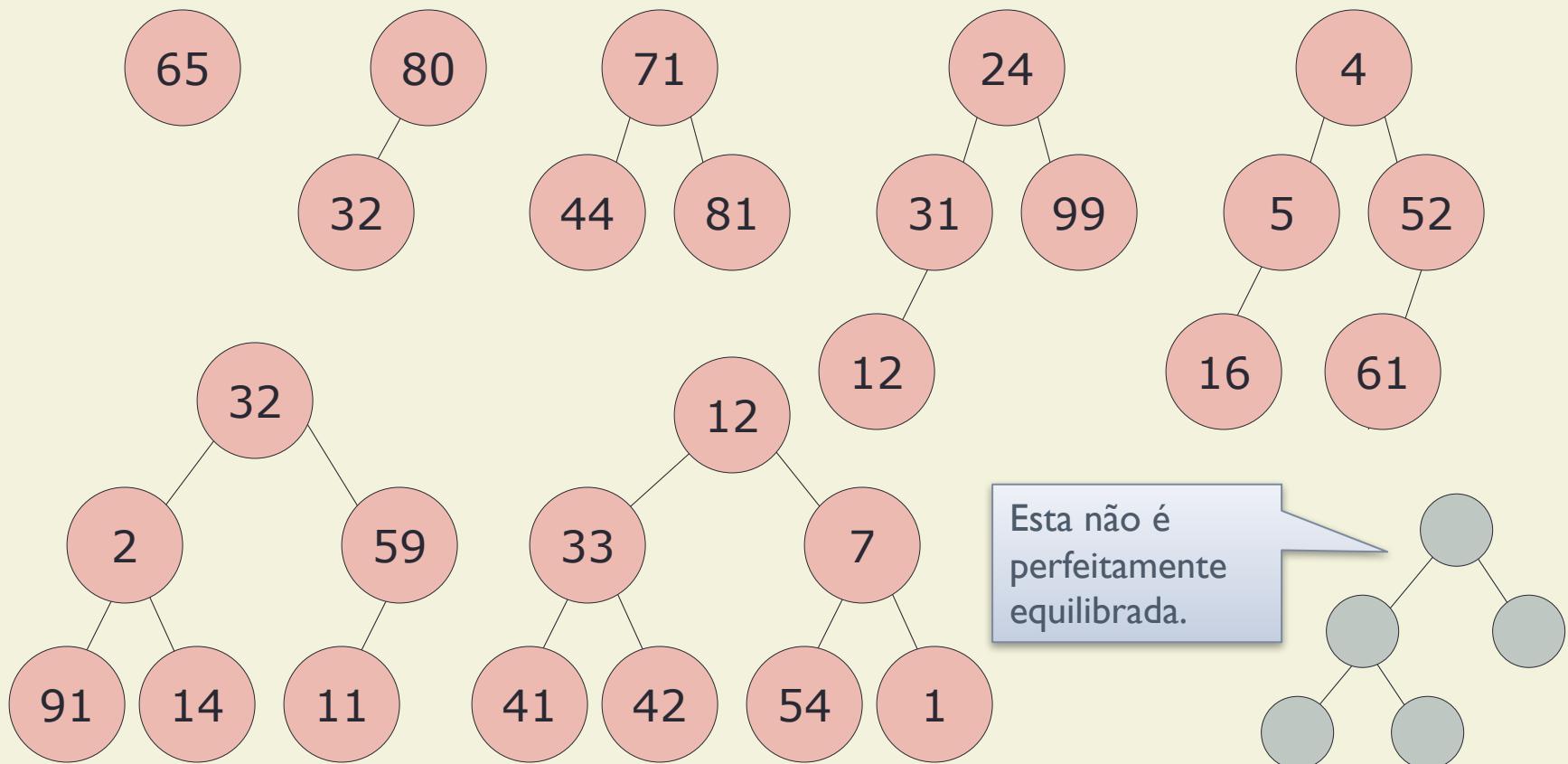
Outras árvores

- Árvores equilibradas.
- Árvores AVL, árvores rubinegras e árvores chanfradas.
- Árvores dois-três.
- Árvores rubinegras imutáveis.



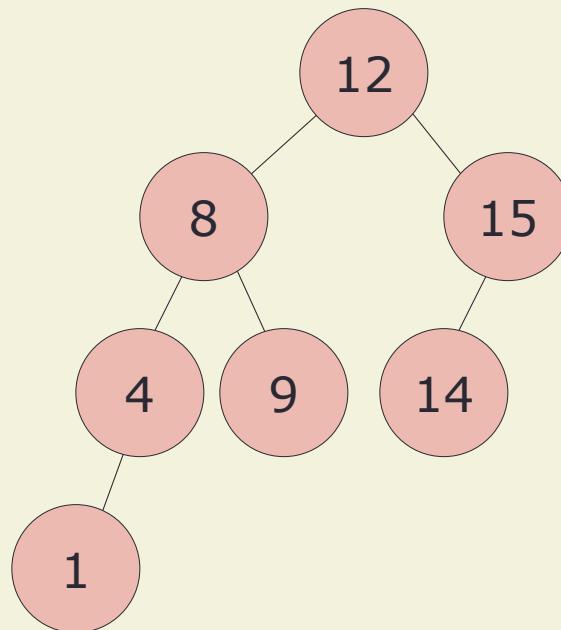
Árvores perfeitamente equilibradas

- Uma árvore é perfeitamente equilibrada se para cada nó o tamanho das suas subárvore diferir de 1 no máximo. Exemplos:



Árvores equilibradas

- Uma árvore é equilibrada se para cada nó a altura da suas subárvore diferir de 1 no máximo.
- Manter uma árvore equilibrada é menos complicado do que mantê-la perfeitamente equilibrada, ao pôr e ao eliminar.



Esta é equilibrada mas
não perfeitamente
equilibrada.

Árvores AVL

- Uma árvore AVL é uma árvore de busca equilibrada.
- “AVL” são as iniciais dos inventores, os matemáticos russos Adelson-Velskii e Landis (1962).
- Em certas implementações, os nós das árvores AVL têm um membro que guarda a altura da subárvore cuja raiz é esse nó, ou então a diferença da altura com a subárvore “irmã”.
- Após inserir um elemento, a árvore AVL reequilibra-se automaticamente, se tiver ficado desequilibrada.
- Idem, ao eliminar um elemento.
- As árvores AVL garantem comportamento logarítmico em inserções, buscas e remoções.

Árvores rubinegras (red-black)

- Uma árvore rubinegra (*red-black tree*) é uma árvore binária de busca, em que cada nó tem uma de duas cores: vermelho ou preto.
- Restringindo as maneiras de colorir os nós ao longo dos caminhos da raiz até às folhas, garante-se que nenhum caminho da raiz até uma folha tem mais do dobro dos nós do que qualquer outro.
- Assim, a árvore fica “aproximadamente” equilibrada.
- Quando se insere ou remove um elemento, a árvore reorganiza-se automaticamente, de maneira a repor a propriedade das árvores rubinegras, isto é, de maneira a que nenhum caminho seja mais do dobro dos nós que qualquer outro.
- Isto garante comportamento logarítmico em inserções, buscas e remoções, tal como nas árvores AVL.
- As árvores AVL são mais “rígidas” do que as rubinegras e, por isso, trabalham mais para inserir ou remover mas menos para procurar.



Árvores chanfradas

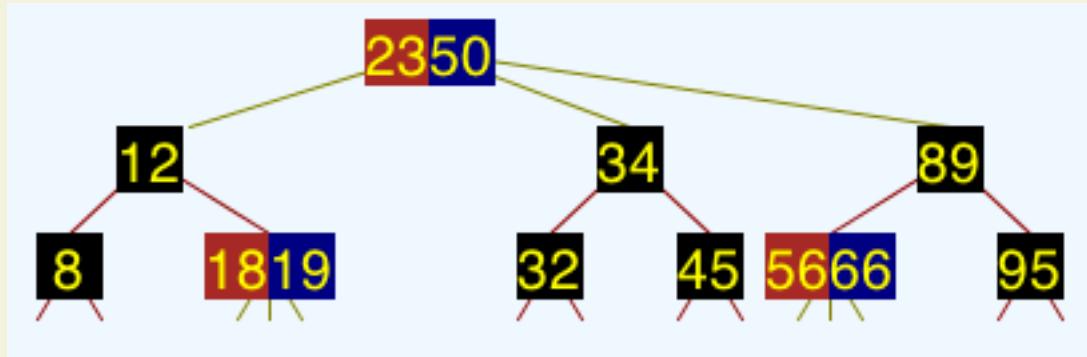
- As árvores chanfradas (*splay trees*) são árvores binárias de busca autoajustáveis.
- Depois de uma operação de acesso—busca, inserção ou remoção—o elemento acedido é promovido até à raiz.
- A ideia é acelerar os acessos subsequentes aos nós mais recentemente acedidos.
- A promoção é feita de dois em dois níveis, excepto quando o nó está no primeiro nível.
- Foram inventadas por Sleator e Tarjan em 1985. Daniel Sleator é professor na Universidade Carnegie-Mellon e Robert Tarjan é professor na Universidade de Princeton.
- As árvores podem estar bastante desequilibradas episodicamente, e não garantem comportamento logarítmico em todas as operações, mas têm comportamento logarítmico “em média”.

Árvores dois-três

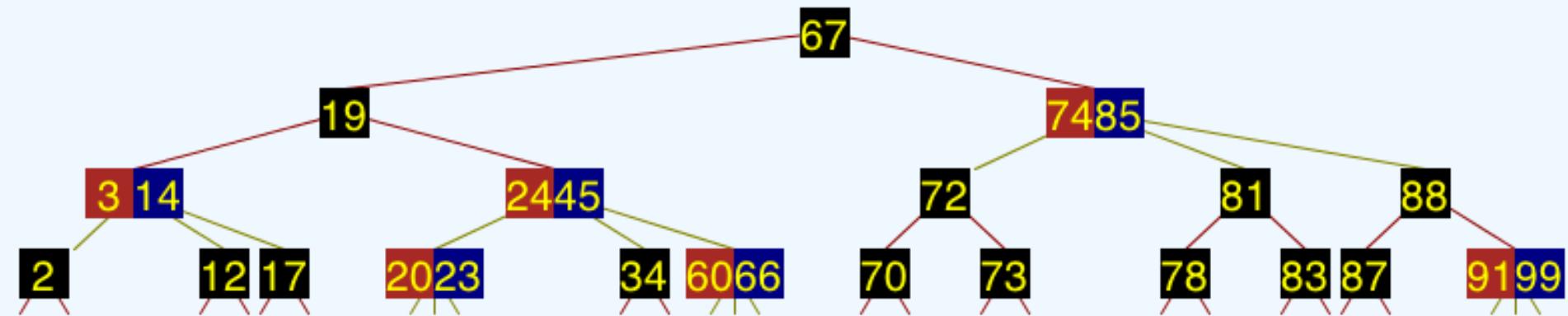
- As árvores **dois-três** têm dois tipos de nós: nós com um valor e **duas** subárvores, e nós com dois valores e **três** subárvores.
- Portanto, as árvores dois-três não são árvores binárias.
- No caso dos nós com duas subárvores, as coisas passam-se como nas árvores de busca: todos os valores presentes na subárvore da esquerda são menores que o valor da raiz e todos os valores presentes na subárvore da direita são maiores que o valor da raiz.
- No caso dos nós com três subárvores, as coisas são ligeiramente mais complicadas: todos os valores presentes na subárvore da esquerda são menores que o *menor* dos dois valores da raiz, todos os valores presentes na subárvore da direita são maiores que o *maior* dos dois valores da raiz, e todos os valores da subárvore do meio são maiores que o *menor* dos dois valores da raiz e menores que o *maior* dos dois valores da raiz.

Árvores dois-três, exemplos

- Uma árvore dois-três com três níveis:



- E uma outra com quatro níveis:



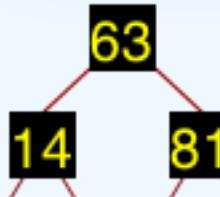
Notamos que, em qualquer nó, ou todas as subárvore são vazias ou todas são não vazias. Isso não acontece por acaso: as árvores dois-três têm essa propriedade (porque estão programadas para tal, bem entendido).

Árvores dois-três, evolução

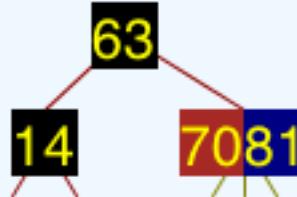
- Inserir 63, na árvore vazia.
- Inserir 14.



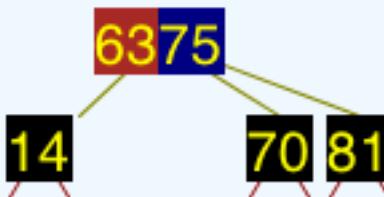
- Inserir 81.



- Inserir 70.

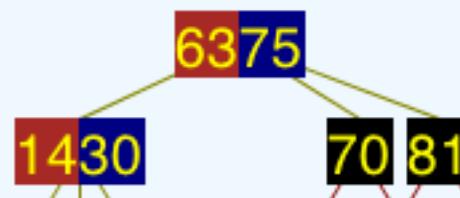


- Inserir 75.



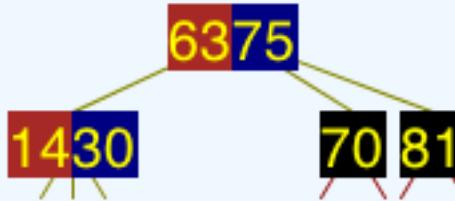
O nó 70+81 partiu-se.

- Inserir 30.

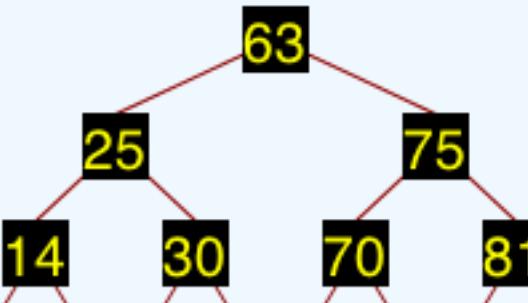


Árvores dois-três, evolução (2)

- (Página anterior.)

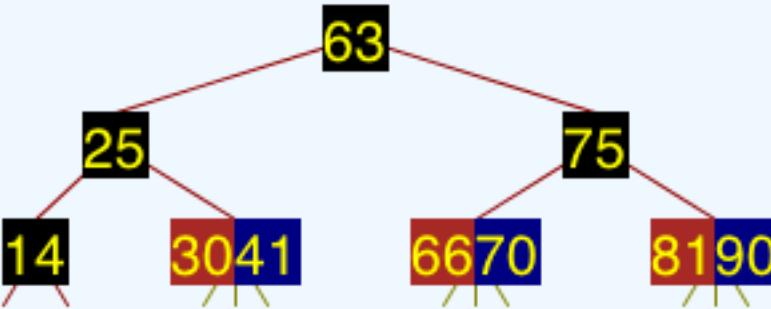


- Inserir 25.

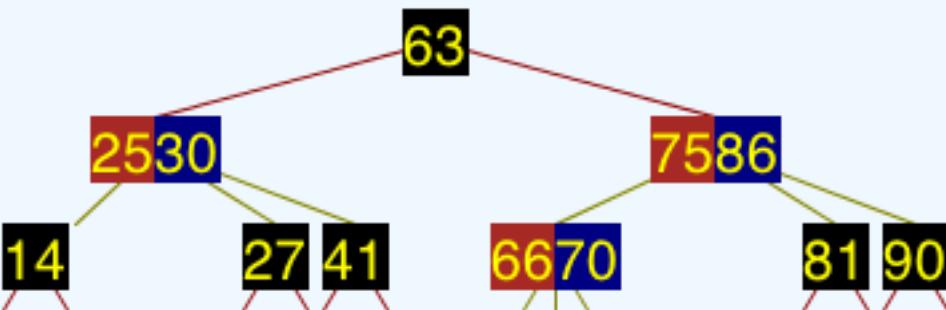


O número de níveis aumentou!

- Inserir 41, 66, 90.

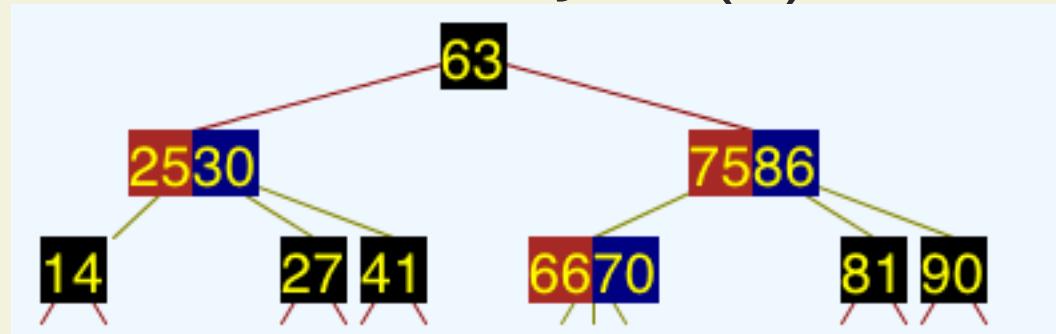


- Inserir 86, 27.

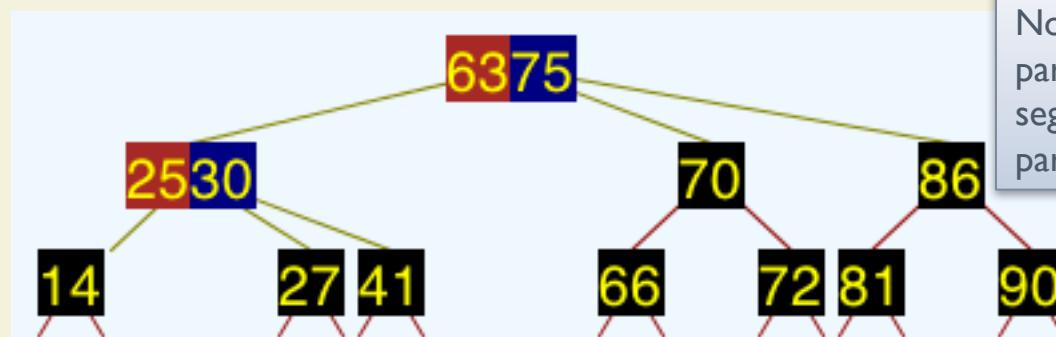


Árvores dois-três, evolução (3)

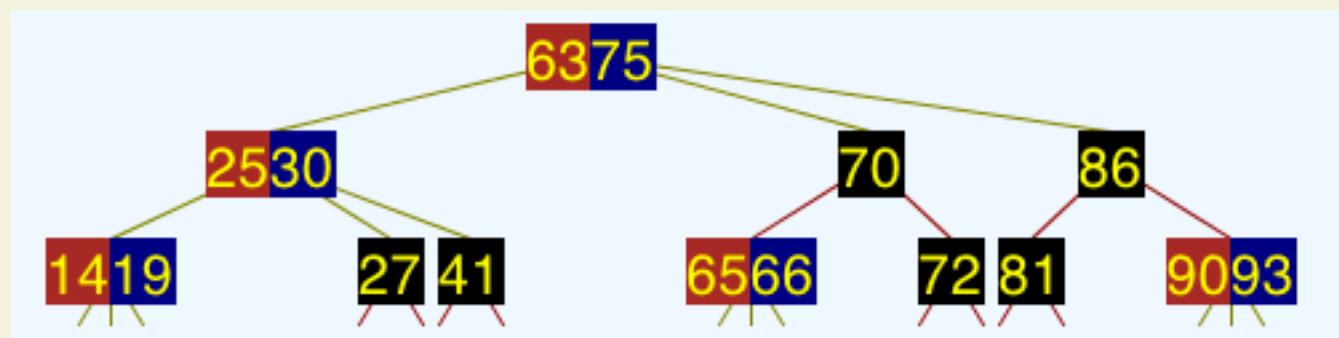
- (Página anterior.)



- Inserir 72

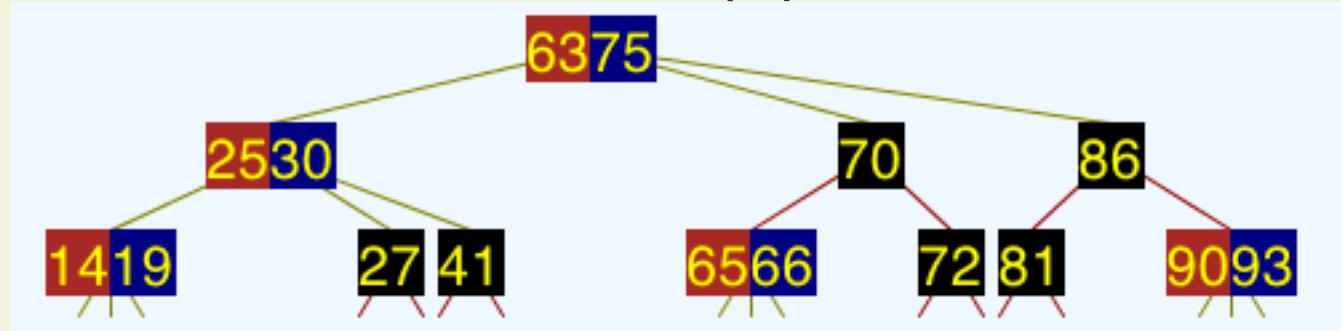


- Inserir 19, 65, 93.

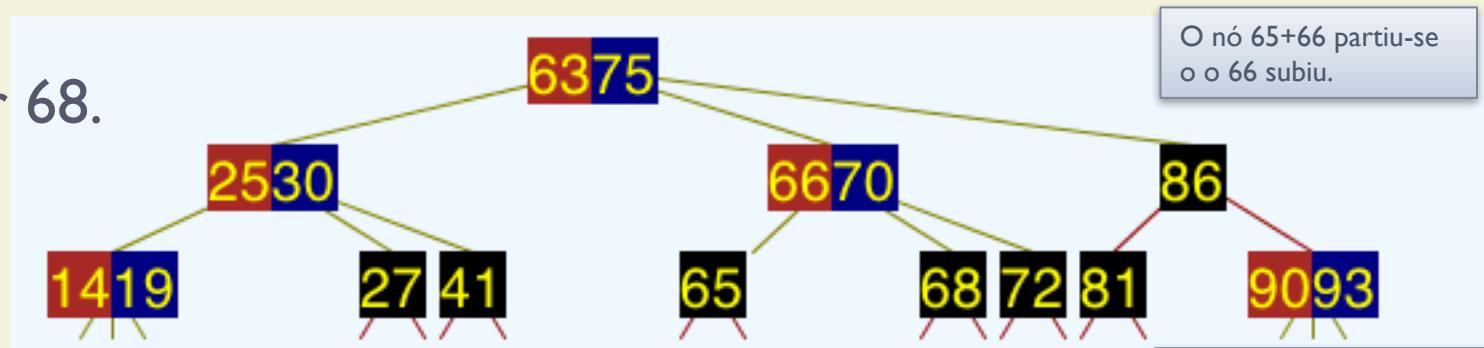


Árvores dois-três, evolução (4)

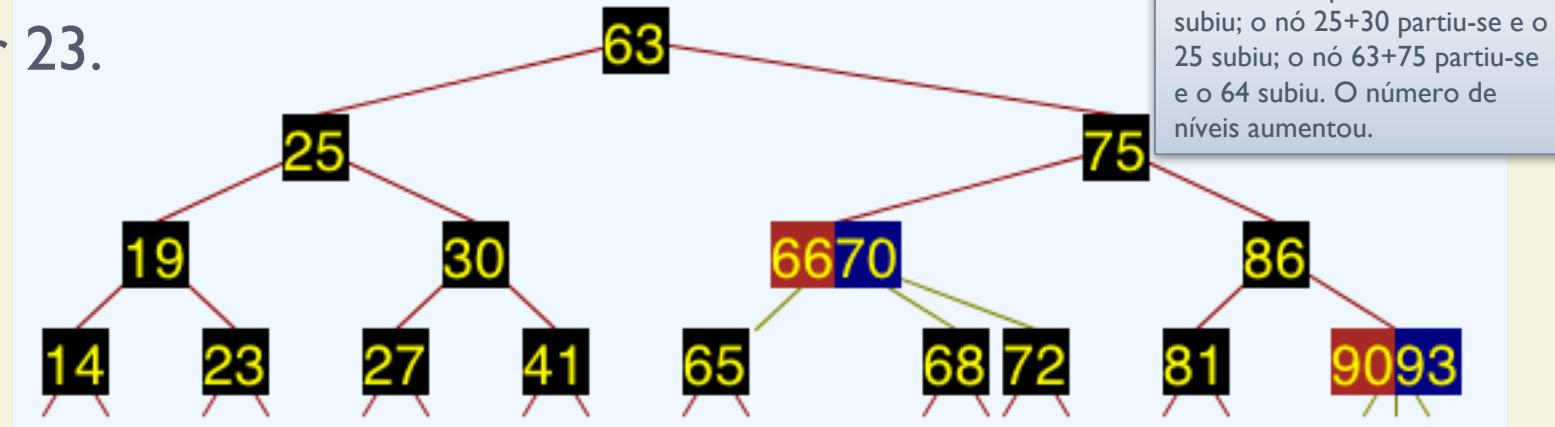
- (Página anterior.)



- Inserir 68.

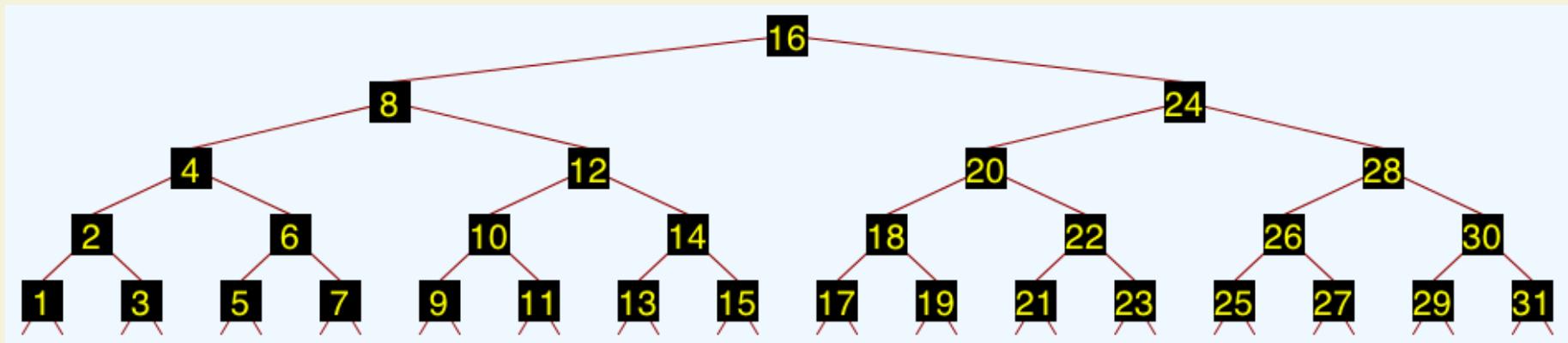


- Inserir 23.



Propriedade fundamental das árvores 2-3

- Uma árvore dois-três com menos de 2^N valores terá no máximo N níveis.
- De facto, na pior da hipóteses, todos os nós de uma árvore com $2^N - 1$ valores terão um só valor; ora, nesse caso, a árvore terá a forma de uma árvore binária completa com N níveis.



Programando as árvores dois-três

- Usaremos árvores imutáveis.
- Começamos com uma classe abstrata:
 - **Tree<T>**. Colocamos isto num pacote novo, por exemplo TwoThreeTrees.immutable.
- Acresentamos três classes concretas:
 - **Empty<T>**, para árvores vazias.
 - **Two<T>**, para árvores com duas subárvore.
 - **Three<T>**, para árvores com três subárvore.
- Usaremos ainda uma classe *transiente*:
 - **Four<T>**, para árvores efémeras com quatro subárvore, que só existem momentaneamente.

Classe abstrata Tree<T>

- Apenas indicamos a funções essenciais:

```
public abstract class Tree<T extends Comparable<T>>
{
    public abstract T value();                                // requires isTwo();
    public abstract int size();
    public abstract boolean isEmpty();
    public abstract boolean has(T x);

    public abstract Tree<T> left();                         // requires isTwo() || isThree();
    public abstract Tree<T> right();                        // requires isTwo() || isThree();

    public abstract T valueLeft();                           // requires isTwo() || isThree();
    public abstract T valueRight();                          // requires isTwo() || isThree();
    public abstract Tree<T> middle();                       // requires isThree();

    public abstract boolean isTwo();
    public abstract boolean isThree();
    protected abstract boolean isTransient();

    public abstract boolean invariant();

    protected abstract Tree<T> split(); // requires isTransient();

    public abstract Tree<T> put(T x);
    protected abstract Tree<T> putr(T x);
```

Repare nos comentários **//requires** que restringem a utilização das funções às árvores que verificam a pré-condição.

Classe Empty<T>

```
class Empty<T extends Comparable<T>> extends TwoThreeTree<T>
{
    public T value()
    {
        throw new UnsupportedOperationException();
    }

    public T valueLeft()
    {
        throw new UnsupportedOperationException();
    }

    public T valueRight()
    {
        throw new UnsupportedOperationException();
    }

    public int size()
    {
        return 0;
    }

    public boolean isEmpty()
    {
        return true;
    }

    public boolean has(T x)
    {
        return false;
    }

    public Tree<T> put(T x)
    {
        return new Two<x>(x, this, this);
    }

    public Tree<T> putr(T x)
    {
        throw new UnsupportedOperationException();
    }
}
```

```
protected Tree<T> split()
{
    throw new UnsupportedOperationException();
}

public boolean isTwo()
{
    return false;
}

public boolean isThree()
{
    return false;
}

protected boolean isTransient()
{
    return false;
}

public boolean invariant()
{
    return true;
}

public Tree<T> left()
{
    throw new UnsupportedOperationException();
}

public Tree<T> right()
{
    throw new UnsupportedOperationException();
}

public Tree<T> middle()
{
    throw new UnsupportedOperationException();
}
```

Nada de muito especial aqui. Mas veja alguns casos interessantes, na página seguinte.

Funções selecionadas, Empty<T>

- A árvore vazia não tem valores:

```
public boolean has(T x)
{
    return false;
}
```

- A árvore vazia respeita o invariante das árvores dois-três:

```
public boolean invariant()
{
    return true;
}
```

- Ao inserir um valor na árvore vazia, cria-se uma árvore “binária”:

```
public Tree<T> put(T x)
{
    return new Two<>(x, this, this);
}
```

Classe Two<T>

- Esta é parecida com a classe **Cons** das árvores binárias de busca:

```
class Two<T extends Comparable<T>> extends Tree<T>
{
    private final T value;
    private final Tree<T> left;
    private final Tree<T> right;
    private final int size;

    public Two(T value, Tree<T> left, Tree<T> right)
    {
        assert left.invariant() && right.invariant() &&
        (left.isEmpty() == right.isEmpty()) &&
        (left.isEmpty() || value.compareTo(left.valueRight()) > 0) &&
        (right.isEmpty() || value.compareTo(right.valueLeft()) < 0);

        this.value = value;
        this.left = left;
        this.right = right;
        this.size = 1 + left.size() + right.size();

        assert invariant();
    }
}
```

Invariante das dois-três binárias

- É análogo à pré-condição do construtor:

```
public boolean invariant()
{
    return left.invariant() && right.invariant() &&
           (left.isEmpty() == right.isEmpty()) &&
           (left.isEmpty() || value.compareTo(left.valueRight()) > 0) &&
           (right.isEmpty() || value.compareTo(right.valueLeft()) < 0);
}
```

- Na árvores **Two**, as funções **valueLeft** e **valueRight** equivalem à função **value**:

```
public T value()
{
    return value;
}
```

```
public T valueLeft()
{
    return value;
}
```

```
public T valueRight()
{
    return value;
}
```

Função has

- A função **has** é análoga à das árvores binárias de busca:

```
public boolean has(T x)
{
    boolean result = true;
    int cmp = x.compareTo(value);
    if (cmp < 0)
        result = left.has(x);
    else if (cmp > 0)
        result = right.has(x);
    return result;
}
```

- Mas note que as chamadas recursivas podem invocar a função **has** de subárvores ternárias.

Função put

- A função **put** é a nossa *pièce de resistance*:
- Por analogia com o que virá a seguir na classe **Three**, convém deixar o trabalho para uma função recursiva, protegida, **putr**:

```
public Tree<T> put(T x)
{
    Tree<T> result = putr(x);
    assert result.invariant();
    return result;
}
```

```
protected Tree<T> putr(T x)
{
    ...
}
```

Função putr

- Caso especial: se o valor a inserir é igual ao valor da raiz, não há nada a fazer:

```
protected Tree<T> putr(T x)
{
    Tree<T> result = this;
    int cmp = x.compareTo(value);
    if (cmp == 0)
        return result;
    ...
}
```

Nota: usamos aqui, excepcionalmente, uma saída antecipada, no caso de **cmp** ser zero. Se for diferente de zero, a função segue em frente, por assim dizer.

Função putr, folhas

- Caso simples: se a árvore for uma folha, transforma-se numa árvore ternária:

```
protected Tree<T> putr(T x)
{
    Tree<T> result = this;
    int cmp = x.compareTo(value);
    if (cmp == 0)
        return result;
    assert left.isEmpty() && right.isEmpty() ||
           !left.isEmpty() && !right.isEmpty();
    if (!left.isEmpty() && right.isEmpty())
        // note: right.isEmpty() is redundant;
    {
        if (cmp < 0)
            result = new Three<>(x, value, left, left, left);
        else
            result = new Three<>(value, x, left, left, left);
    }
    else
        ...
}
```

Note que neste caso, a árvore **left** é a árvore vazia.

Função putr, caso mais estimulante

- Se a árvore não for uma folha, então, tal como nas árvores de busca, insere-se na subárvore do lado apropriado, consoante o valor for menor ou maior que o valor da raiz.
- A árvore retornada pela chamada recursiva substitui a subárvore respetiva, a não ser que seja uma árvore quaternária, transiente, a qual então é preciso partir.
- No caso em que há uma árvore quaternária, o valor que “subiu” junta-se ao valor da raiz da árvore, para construir uma árvore ternária.

Função putr, para árvores “internas”

- Observe:

```
protected Tree<T> putr(T x)
{
    Two<T> result = this;
    int cmp = x.compareTo(value);
    if (...)
    ...
    else
    {
        if (cmp < 0)
        {
            Tree<T> t = left.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = new Three<>(t.value(), value, t.left(), t.right(), right);
            }
            else
                result = new Two<>(value, t, right);
        }
        ...
    }
}
```

Este parte corresponde a inserir do lado esquerdo. Para o lado direito é análogo.

Função putr, inteira

```
protected Tree<T> putr(T x)
{
    Two<T> result = this;
    int cmp = x.compareTo(value);
    if (cmp == 0)
        return result;
    assert left.isEmpty() && right.isEmpty() || !left.isEmpty() && !right.isEmpty();
    if (left.isEmpty() && right.isEmpty()) // note: right.isEmpty() is redundant;
    {
        if (cmp < 0)
            result = new Three<>(x, value, left, left, left);
        else
            result = new Three<>(value, x, left, left, left);
    }
    else
    {
        if (cmp < 0)
        {
            Tree<T> t = left.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = new Three<>(t.value(), value, t.left(), t.right(), right);
            }
            else
                result = new Two<>(value, t, right);
        }
        else
        {
            Tree<T> t = right.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = new Three<>(value, t.value(), left, t.left(), t.right());
            }
            else
                result = new Two<>(value, left, t);
        }
    }
    return result;
}
```

Bela função!

Classe Three<T>

- De certa forma, as árvores ternárias são uma generalização das árvores binárias:

```
class Three<T extends Comparable<T>> extends Tree<T>
{
    private final T valueLeft;
    private final T valueRight;
    private final Tree<T> left;
    private final Tree<T> middle;
    private final Tree<T> right;
    private final int size;

    public Three(T valueLeft, T valueRight,
                Tree<T> left, Tree<T> middle, Tree<T> right)
    {
        assert left.invariant() && middle.invariant() && right.invariant()
            && (left.isEmpty() == middle.isEmpty() && (middle.isEmpty() == right.isEmpty()))
            && (valueLeft.compareTo(valueRight) < 0)
            && (left.isEmpty() || valueLeft.compareTo(left.valueRight()) > 0)
            && (right.isEmpty() || valueRight.compareTo(right.valueLeft()) < 0);

        this.valueLeft = valueLeft;
        this.valueRight = valueRight;
        this.left = left;
        this.middle = middle;
        this.right = right;
        this.size = 2 + left.size() + middle.size() + right.size();

        assert invariant();
    }
}
```

Invariante das dois-três ternárias

- É, de novo, análogo à pré-condição do construtor:

```
public boolean invariant()
{
    return left.invariant() && middle.invariant() && right.invariant() &&
        (left.isEmpty() == middle.isEmpty()) &&
        (middle.isEmpty() == right.isEmpty()) &&
        (valueLeft.compareTo(valueRight) < 0) &&
        (left.isEmpty() || valueLeft.compareTo(left.valueRight()) > 0) &&
        (right.isEmpty() || valueRight.compareTo(right.valueLeft()) < 0);
}
```

- Neste caso, as **valueLeft** e **valueRight** devolvem os valores respetivos, e a função **value** não está disponível:

```
public T valueLeft()
{
    return value;
}

public T valueRight()
{
    return value;
}
```

```
public T value()
{
    throw new
        UnsupportedOperationException();
}
```

Função has para árvores ternárias de busca

- É uma generalização do caso já conhecido:

```
public boolean has(T x)
{
    boolean result = true;
    int cmpLeft = x.compareTo(valueLeft);
    int cmpRight = x.compareTo(valueRight);
    if (cmpLeft < 0)
        result = left.has(x);
    else if (cmpRight > 0)
        result = right.has(x);
    else if (cmpLeft > 0 && cmpRight < 0)
        result = middle.has(x);
    return result;
}
```

- Note que as chamadas recursivas podem invocar polimorficamente a função **has** de subárvores vazias, binárias ou ternárias.

Função put, árvores ternárias

- Se nas árvores binárias deu luta, aqui mais luta dará.
- No entanto, a ideia é a mesma:
- Há a função **put** e uma função recursiva, **putr**:

```
public Tree<T> put(T x)
{
    Tree<T> result = putr(x);
    if (result.isTransient())
        result = result.split();
    return result;
}
```

Atenção: se o resultado da função **putr** for uma árvore quaternária, então o resultado do **put** é a árvore que resulta de partir a árvore quaternária.

```
protected Tree<T> putr(T x)
{
    ...
}
```

Função putr, árvores ternárias

- Caso especial: se o valor a inserir é igual a um dos valores da raiz, não há nada a fazer:

```
public Tree<T> putr(T x)
{
    Tree<T> result = this;
    int cmpLeft = x.compareTo(valueLeft);
    int cmpRight = x.compareTo(valueRight);
    if (cmpLeft == 0 || cmpRight == 0)
        return result;
    ...
}
```

De novo a técnica da saída antecipada,
no caso em que não há nada a fazer.

Função putr, folhas

- Caso simples: se a árvore for uma folha, transforma-se numa árvore quaternária:

```
public Tree<T> putr(T x)
{
    Tree<T> result = this;
    ...
    assert left.isEmpty() && middle.isEmpty() && right.isEmpty()
        || !left.isEmpty() && !middle.isEmpty() && !right.isEmpty();
    if (left.isEmpty() && middle.isEmpty() && right.isEmpty())
    {
        if (cmpLeft < 0)
            result = new Four<>(x, valueLeft, valueRight, left, left, left, left);
        else if (cmpRight > 0)
            result = new Four<>(valueLeft, valueRight, x, left, left, left, left);
        else
            result = new Four<>(valueLeft, x, valueRight, left, left, left, left);
    }
    else
    ...
}
```

A árvore quaternária é transiente e não durará muito tempo. Ela será partida logo a seguir, pela função que chamou esta!

Função putr, caso mais estimulante

- Na verdade, é perfeitamente análogo ao caso das árvores binárias.
- Se a árvore ternária não for uma folha, então insere-se na subárvore apropriada, consoante o valor for menor que o menor dos valores da raiz, maior que o maior dos valores da raiz ou estiver entre os dois.
- Tal como antes, a árvore retornada pela chamada recursiva substitui a subárvore respetiva, a não ser que seja uma árvore quaternária, transiente, a qual então é preciso partir.
- No caso em que há uma árvore quaternária, o valor que “subiu” junta-se aos valores da raiz da árvore, para construir uma árvore, desta vez quaternária, que há de ser partida pela função que chamou esta.

Função putr, para árvores ternárias internas

- Confira:

```
public Tree<T> putr(T x)
{
    Tree<T> result = this;
    int cmpLeft = x.compareTo(valueLeft);
    int cmpRight = x.compareTo(valueRight);
    if (...)
    {
        ...
    }
    else
    {
        assert !left.isEmpty() && !middle.isEmpty() && !right.isEmpty();
        if (cmpLeft < 0)
        {
            Tree<T> t = left.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = new Four<>
                    (t.value(), valueLeft, valueRight, t.left(), t.right(), middle, right);
            }
            else
                result = new Three<>(valueLeft, valueRight, t, middle, right);
        }
        else
            ...
    }
}
```

Este parte corresponde a inserir do lado esquerdo. Para inserir ao meio ou do lado direito é análogo.

Função putr, para árvores ternárias

```
public Tree<T> putr(T x)
{
    Tree<T> result = this;
    int cmpLeft = x.compareTo(valueLeft);
    int cmpRight = x.compareTo(valueRight);
    if (cmpLeft == 0 || cmpRight == 0)
        return result;
    assert left.isEmpty() && middle.isEmpty() && right.isEmpty()
        || !left.isEmpty() && !middle.isEmpty() && !right.isEmpty();
    if (left.isEmpty() && middle.isEmpty() && right.isEmpty())
    {
        if (cmpLeft < 0)
            result = new Four<>(x, valueLeft, valueRight, left, left, left, left);
        else if (cmpRight > 0)
            result = new Four<>(valueLeft, valueRight, x, left, left, left, left);
        else
            result = new Four<>(valueLeft, x, valueRight, left, left, left, left);
    }
    else
    {
        assert !left.isEmpty() && !middle.isEmpty() && !right.isEmpty();
        if (cmpLeft < 0)
        {
            Tree<T> t = left.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = new Four<>(t.value(), valueLeft, valueRight, t.left(), t.right(), middle, right);
            }
            else
                result = new Three<>(valueLeft, valueRight, t, middle, right);
        }
        else if (cmpRight > 0)
        {
            Tree<T> t = right.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = new Four<>(valueLeft, valueRight, t.value(), left, middle, t.left(), t.right());
            }
            else
                result = new Three<>(valueLeft, valueRight, left, middle, t);
        }
        else
        {
            Tree<T> t = middle.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = new Four<>(valueLeft, t.value(), valueRight, left, t.left(), t.right(), right);
            }
            else
                result = new Three<>(valueLeft, valueRight, left, t, right);
        }
    }
    return result;
}
```

Árvores quaternárias

- As árvores quaternárias têm três valores e quatro subárvores.
- No contexto das árvores dois-três, as árvores quaternárias são transientes: são criadas por uma função, e devolvidos como resultado, mas são imediatamente partidas, pela função que recebe o resultado, para formar uma árvore binária.
- Sendo assim, a maior parte das funções herdadas da classe abstrata são dispensáveis.
- Basta o construtor, a função **split** e a função **isTransient**.

Classe Four<T>

- Seria uma generalização, mas não vale a pena, pois bastam algumas poucas funções:

```
class Four<T extends Comparable<T>> extends TwoThreeTree<T>
{
    private final T valueLeft;
    private final T valueMiddle;
    private final T valueRight;
    private final Tree<T> left;
    private final Tree<T> middleLeft;
    private final Tree<T> middleRight;
    private final Tree<T> right;

    public FourCons(T valueLeft, T valueMiddle, T valueRight,
                    Tree<T> left, Tree<T> middleLeft,
                    Tree<T> middleRight, Tree<T> right)
    {
        assert valueLeft.compareTo(valueMiddle) < 0;
        assert valueMiddle.compareTo(valueRight) < 0;
        this.valueLeft = valueLeft;
        this.valueMiddle = valueMiddle;
        this.valueRight = valueRight;
        this.left = left;
        this.middleLeft = middleLeft;
        this.middleRight = middleRight;
        this.right = right;
    }

    ...
}
```

Classe Four<T>, split e isTransient

- A função **split** é simples: o resultado é uma árvore binária de árvores binárias:

```
public Two<T> split()
{
    Two<T> newLeft = new Two<>(valueLeft, left, middleLeft);
    Two<T> newRight = new Two<>(valueRight, middleRight, right);
    return new Two<>(valueMiddle, newLeft, newRight);
}
```

- A função **isTransient** retorna **true**:

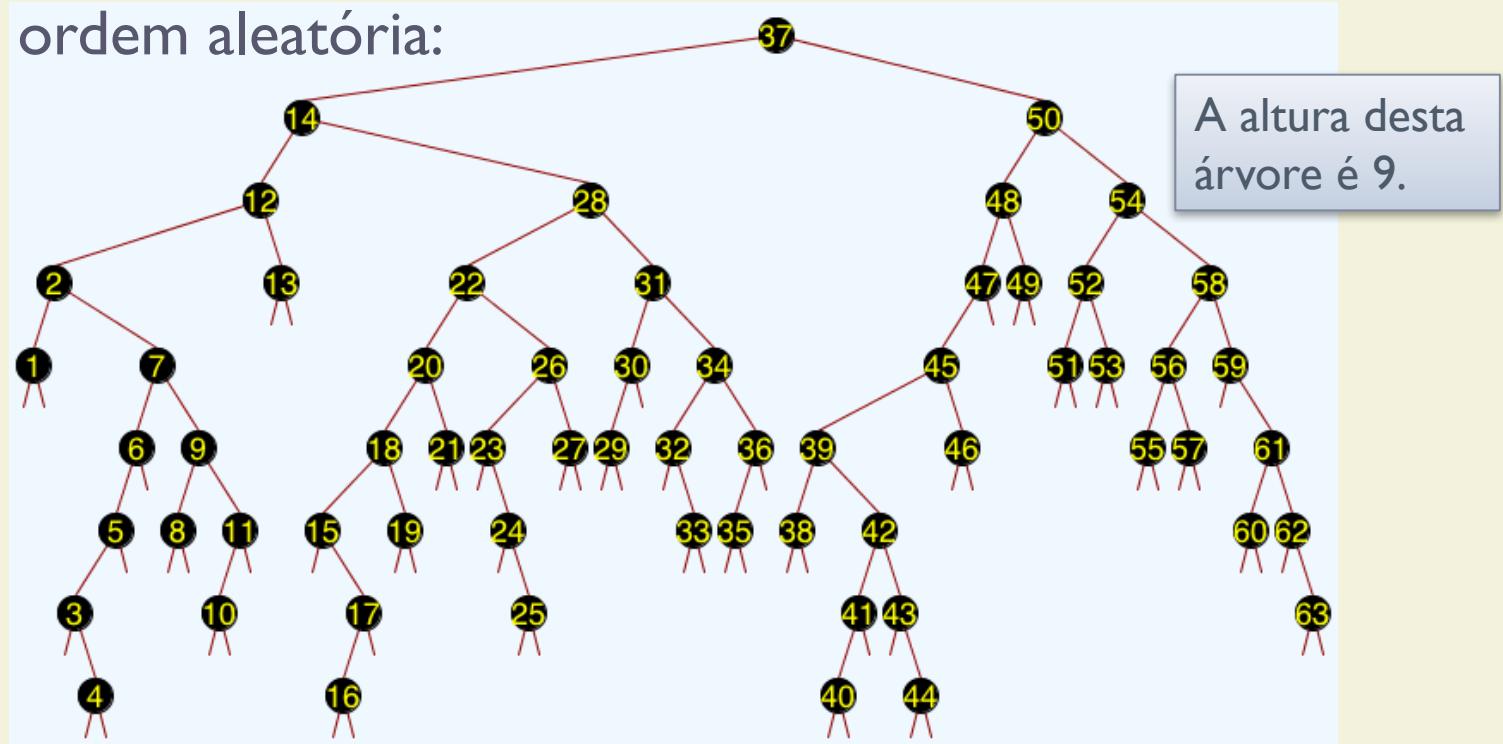
```
protected boolean isTransient()
{
    return true;
}
```

Nas outras classes,
retorna **false**, claro.

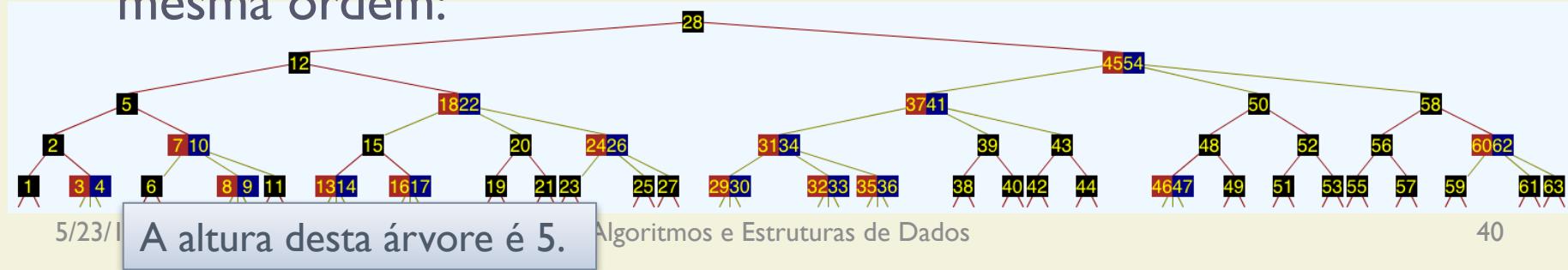
E assim concluímos a programação da árvores dois-três. As funções que não considerámos aqui explicitamente são triviais.

Experiência

- Árvore binária de busca, com 63 valores, de 1 a 63, inseridos por ordem aleatória:

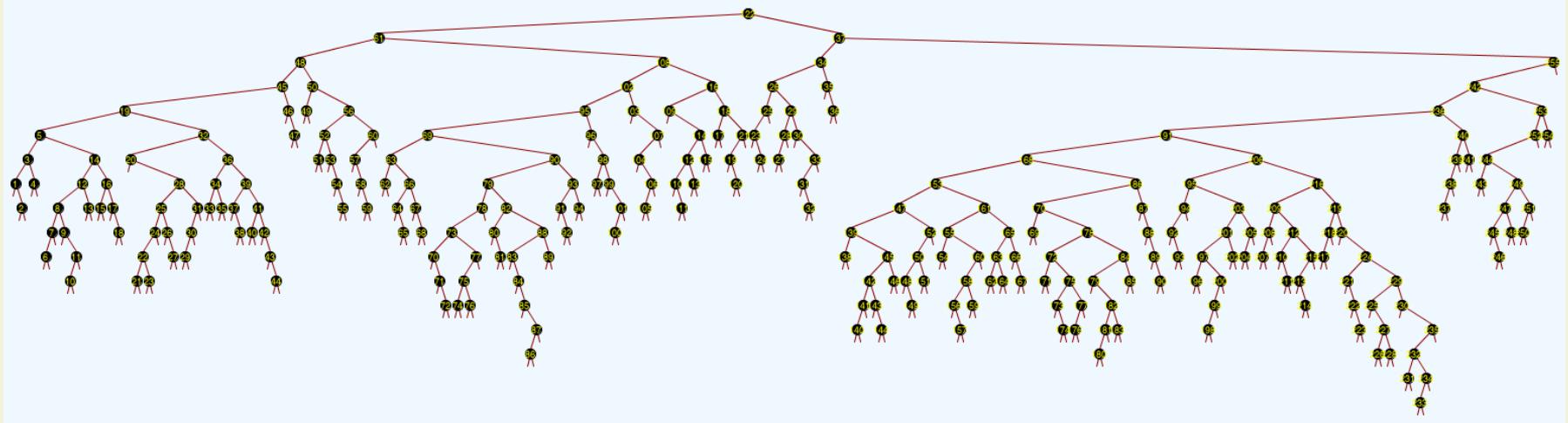


- Árvore dois-três, com os mesmos 63 valores, inseridos pela mesma ordem:



Outra experiência, maior

- Árvore binária de busca, com 255 valores, de 1 a 255, inseridos por ordem aleatória:



- Árvore dois-três, com os mesmos 255 valores,



Moral da história

- As árvores dois-três têm comportamento logarítmico garantido, em buscas e inserções.
- Isso é uma propriedade muito interessante.
- No entanto, a sobrecarga de ter de gerir vários tipos de nós é considerável, e o volume de código é grande, quando comparado com o das árvores binárias de busca.
- E note que ainda faltaria programar uma série de coisas: a função **delete**, as funções de ordem, os iteradores, etc.
- O ideal seria ter este tipo de garantia logarítmica, mas no contexto de árvores binárias.
- É o que faremos a seguir, com as árvores **red-black**.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 26

Árvores **red-black**

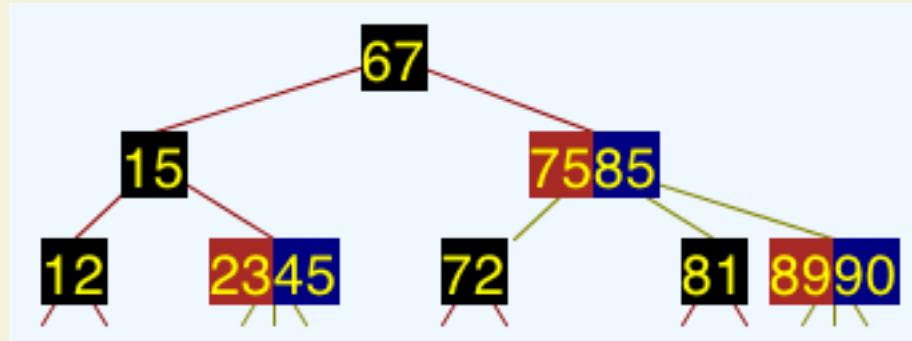
Árvores **red-black**

- Árvores dois-três.
- Árvores **red-black** imutáveis.

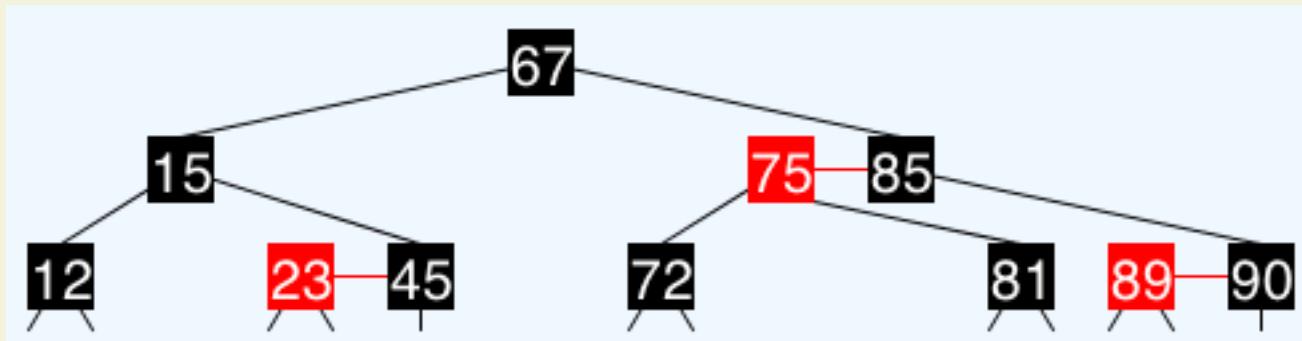


Redesenhando as árvores dois-três

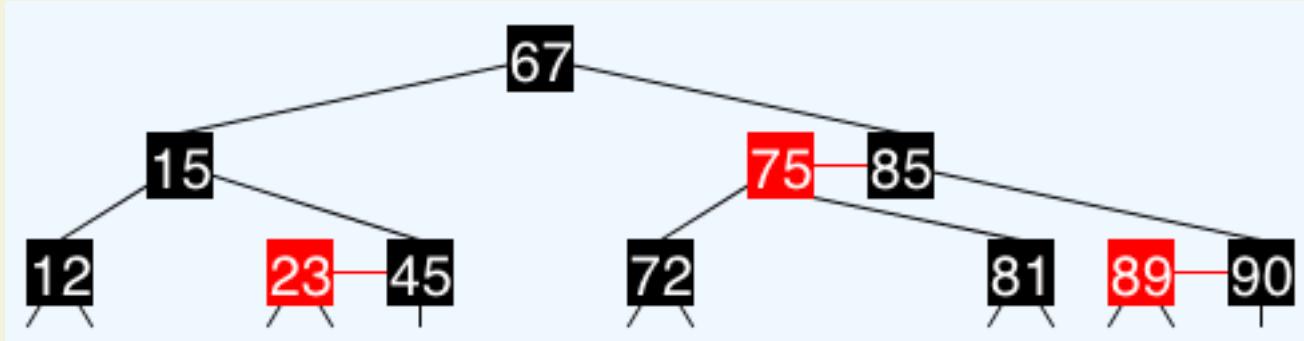
- Eis um exemplo:



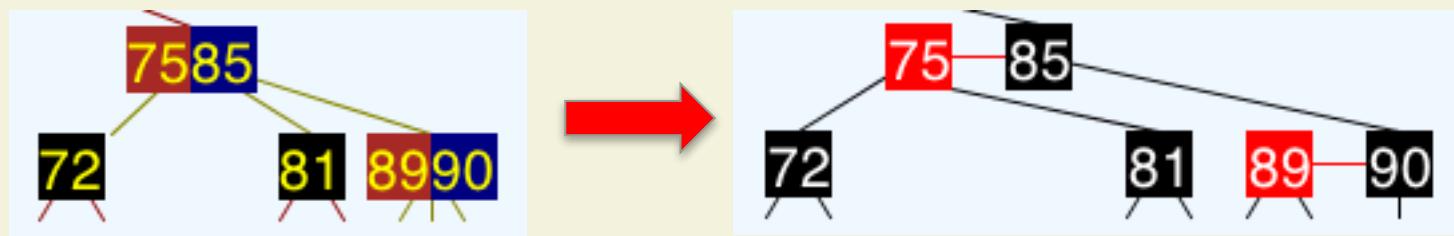
- Eis a mesma árvore desenhada com os dois valores dos nós ternários separados, ligados por um segmento horizontal:



Árvores red-black



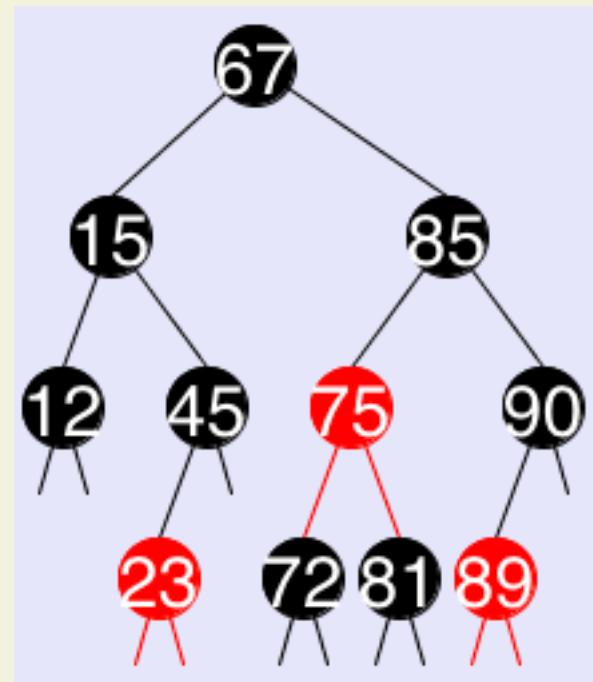
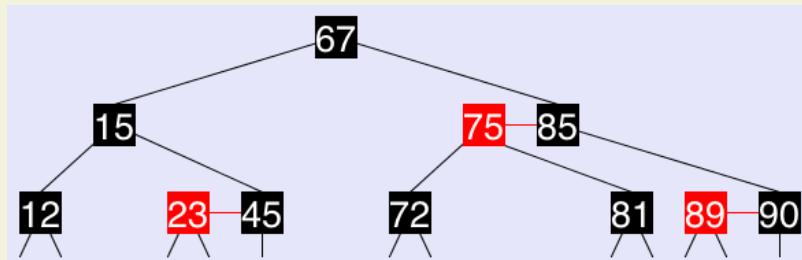
- As árvores **red-black** são árvores dois-três em que os nós ternários são representados por um par de nós binários:



Tipicamente, ao desenhar os nós ternários separados, o da esquerda fica **vermelho** e o da direita fica preto. Daí o nome **red-black**.

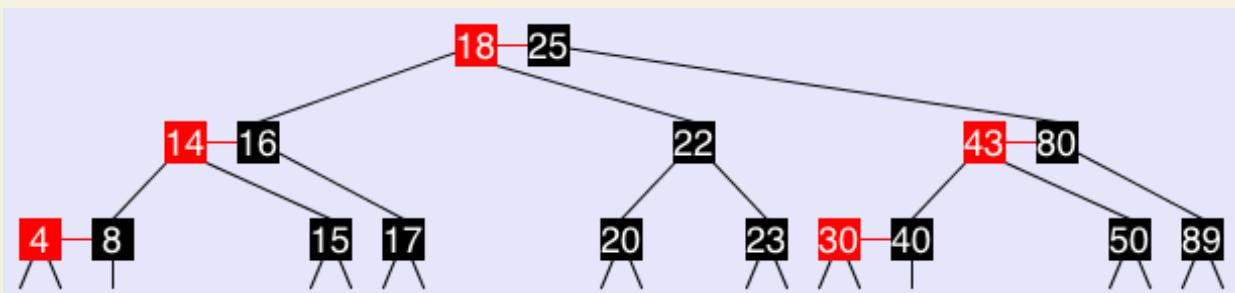
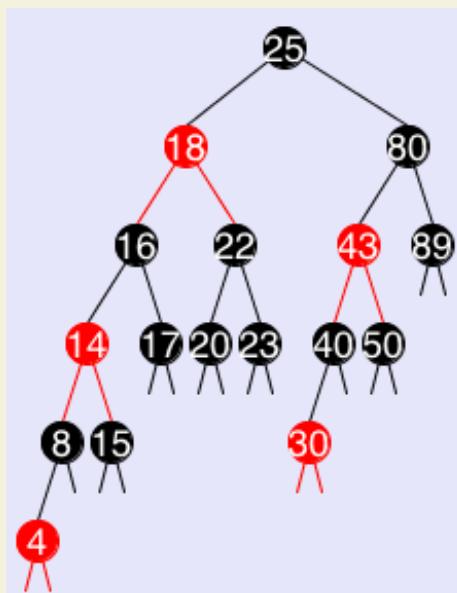
As RBs são BSTs

- Se baixarmos as ligações horizontais, percebemos claramente que as árvores red-black são árvores binárias de busca.



Propriedade fundamental das árvores RB

- Uma árvore red-black com menos de 2^N valores terá no máximo $2N$ níveis.
- De facto, já sabemos que a árvore dois-três correspondente tem no máximo N níveis. Ora, a árvore red-black acrescenta no máximo um nível por cada nível da árvore dois-três correspondente.



Este exemplo mostra uma árvore RB de seis níveis que corresponde a uma árvore 2-3 de três níveis.

Outras propriedades

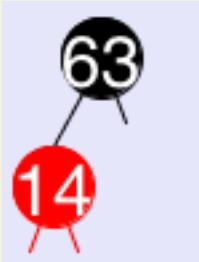
- A raiz é preta.
- A subárvore direita de uma árvore preta é preta (se não for vazia); a subárvore esquerda é preta ou **vermelha** (se não for vazia).
- Ambas as subárvores de uma árvore **vermelha** são pretas (se não forem vazias).
- As subárvores de uma árvore **vermelha** são ambas vazias ou são ambas não vazias.
- Se a subárvore direita de uma árvore **preta** for vazia, a subárvore esquerda é a árvore vazia ou é uma árvore **vermelha** cujas subárvores são ambas vazias.

Árvores red-black, evolução

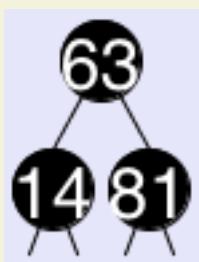
- Inserir 63, na árvore vazia.



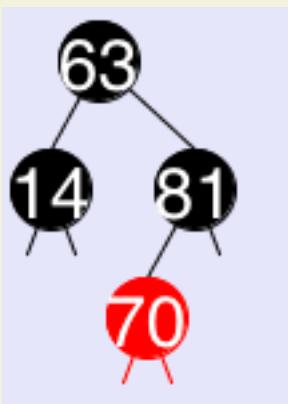
- Inserir 14.



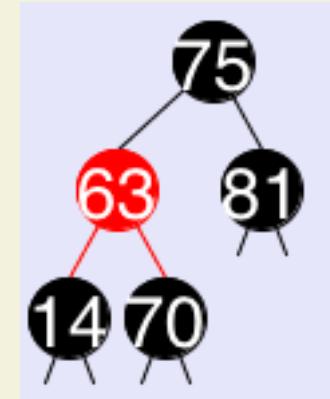
- Inserir 81.



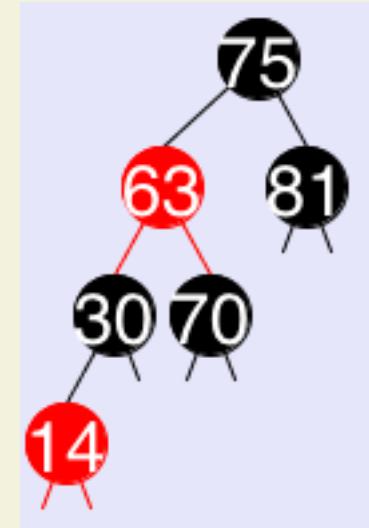
- Inserir 70.



- Inserir 75.

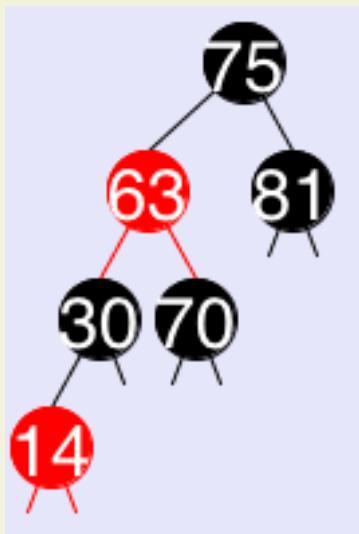


- Inserir 30.

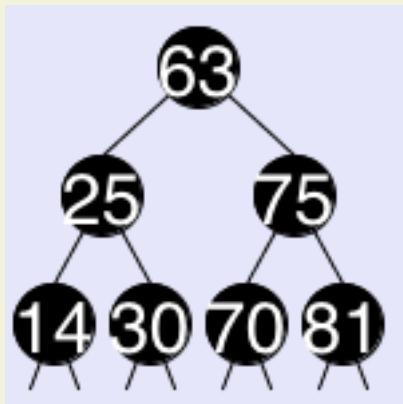


Árvores red-black, evolução (2)

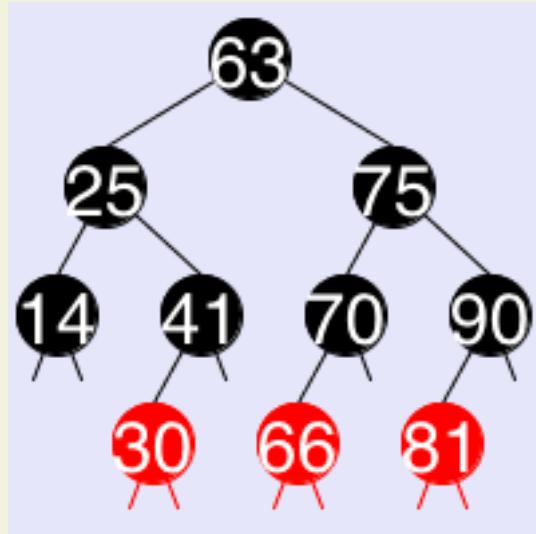
- (Página anterior.)



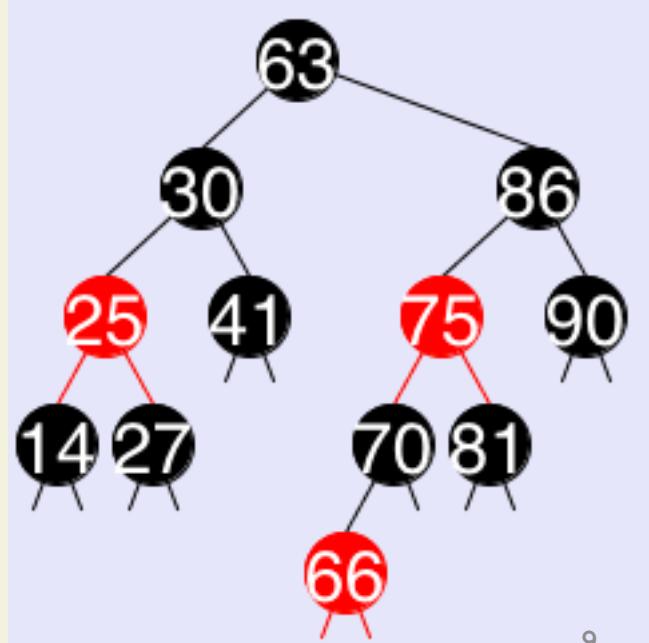
- Inserir 25.



- Inserir 41, 66, 90.

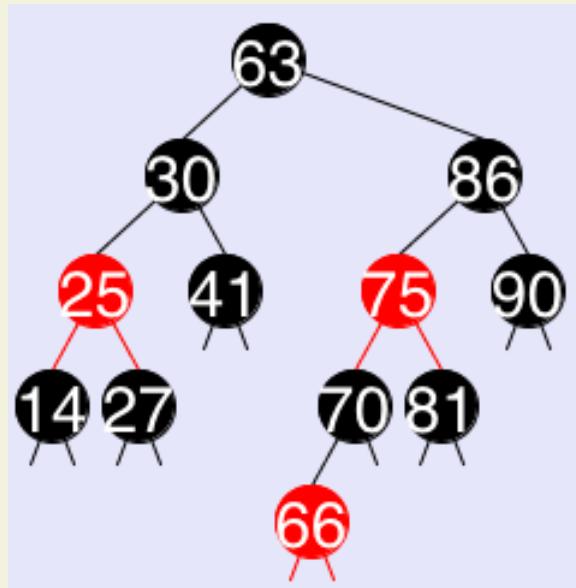


- Inserir 86, 27.

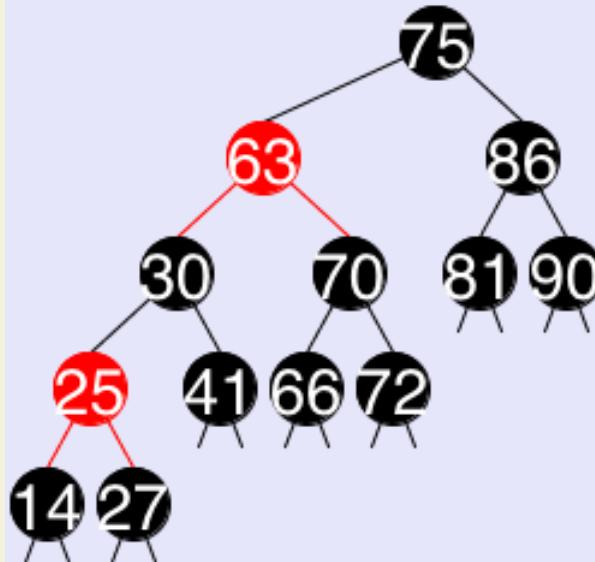


Árvores red-black, evolução (3)

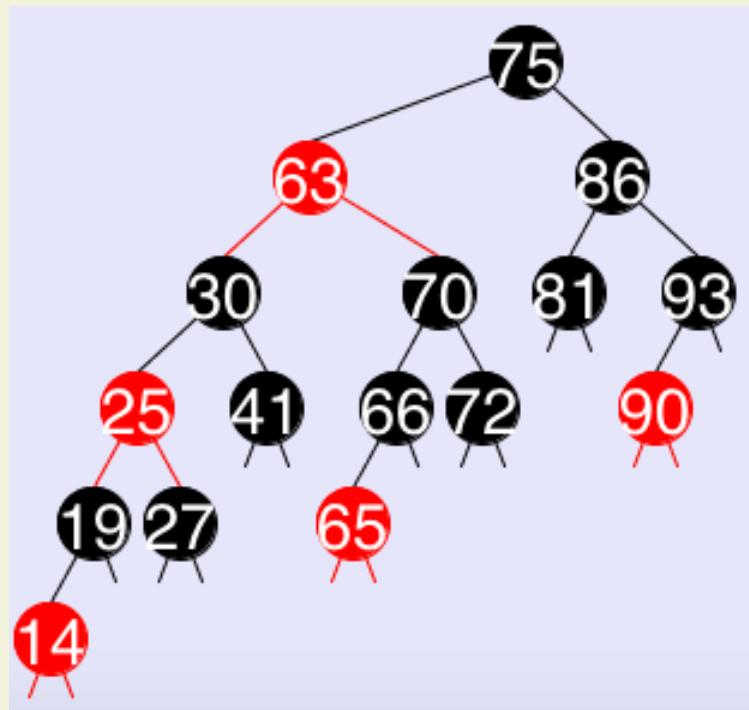
- (Página anterior.)



- Inserir 72

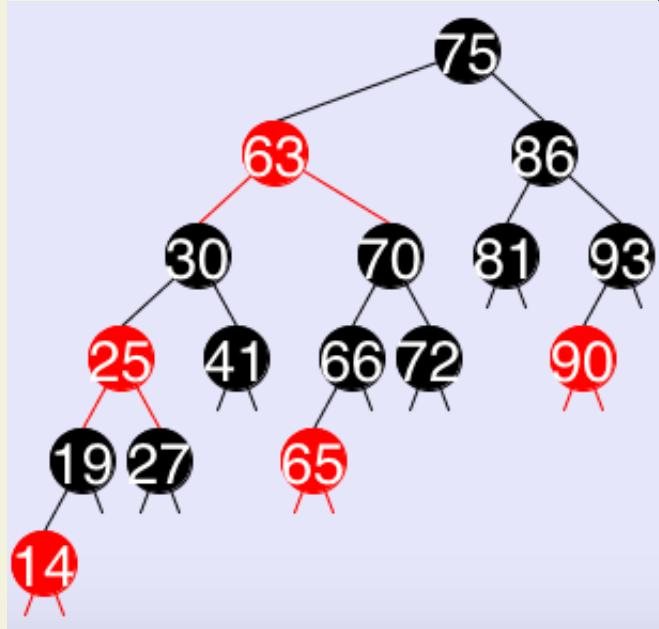


- Inserir 19, 65, 93.

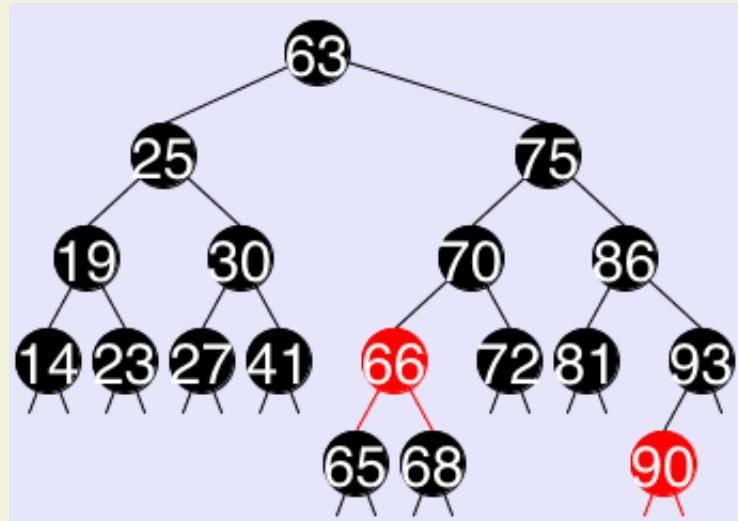


Árvores red-black, evolução (4)

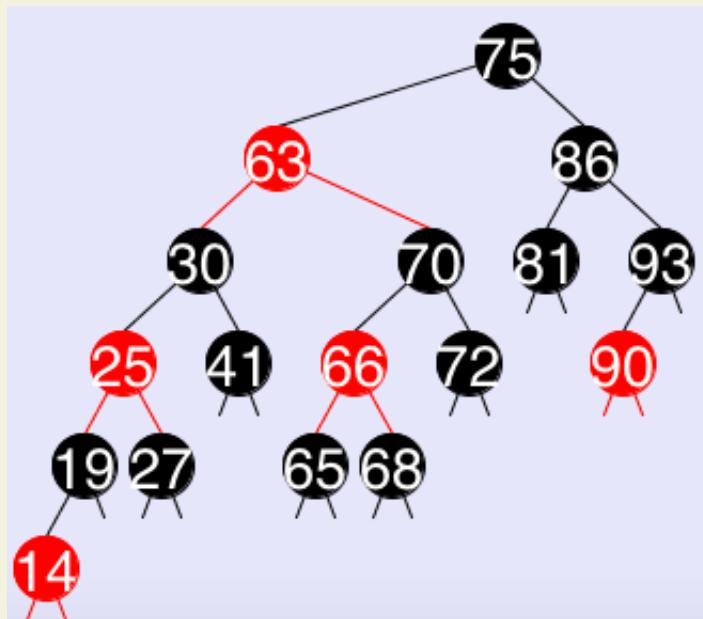
- (Página anterior.)



- Inserir 23.



- Inserir 68.



Ao inserir 23, a árvore reorganizou-se dramaticamente e o número de níveis baixou de 6 para 5.

Programando as árvores red-black

- Usaremos árvores imutáveis.
- Começamos com uma classe abstrata:
 - **Tree<T>**. Esta e as outras existirão no pacote `redblacktrees.immutable`.
- Acresentamos três classes concretas:
 - **Empty<T>**, para árvores vazias.
 - **Black<T>**, para árvores pretas.
 - **Red<T>**, para árvores **vermelhas**.
- Usaremos ainda uma classe *transiente*:
 - **Brown<T>**, para árvores efémeras que correspondem às árvores dois-três **Four**.

Classe abstrata Tree<T>

```
public abstract class Tree<T extends Comparable<T>>
{
    public abstract T value();
    public abstract int size();
    public abstract boolean isEmpty();
    public abstract boolean has(T x);

    public abstract boolean isRed();
    public abstract boolean isBlack();

    public abstract boolean invariant();
    public abstract boolean isSearchTree();

    public abstract boolean isTwo();
    public abstract boolean isThree();

    public abstract Tree<T> left();    // requires !isEmpty();
    public abstract Tree<T> right();   // requires !isEmpty();

    protected abstract boolean isTransient();
    protected abstract Black<T> split(); // requires isTransient();

    public abstract Tree<T> put(T x);
    protected abstract Tree<T> putr(T x);
    ...
}
```

Segue o padrão habitual.

O método **isTwo** (resp. **isThree**) dá **true** se a árvore corresponder a um nó binário (resp. ternário) de uma árvore 2-3.

Classe Empty<T>

```
class Empty<T extends Comparable<T>>
    extends Tree<T>
{
    public T value()
    {
        throw new UnsupportedOperationException();
    }

    public int size()
    {
        return 0;
    }

    public boolean isEmpty()
    {
        return true;
    }

    public boolean has(T x)
    {
        return false;
    }

    public boolean isRed()
    {
        return false;
    }

    public boolean isBlack()
    {
        return false;
    }

    public boolean invariant()
    {
        return true;
    }
}
```

```
public boolean isTwo()
{
    throw new UnsupportedOperationException();
}

public boolean isThree()
{
    throw new UnsupportedOperationException();
}

public Tree<T> left()
{
    throw new UnsupportedOperationException();
}

public Tree<T> right()
{
    throw new UnsupportedOperationException();
}

protected boolean isTransient()
{
    return false;
}

protected Black<T> split()
{
    throw new UnsupportedOperationException();
}

public Tree<T> put(T x)
{
    return new Black<T>(x, this, this);
}

public Tree<T> putr(T x)
{
    throw new UnsupportedOperationException();
}
}
```

Classe Black<T>

- Esta é a classe que realmente trabalha.

```
class Black<T extends Comparable<T>> extends Tree<T>
{
    private final T value;
    private final Tree<T> left;
    private final Tree<T> right;
    private final int size;

    public Tree(T value, Tree<T> left, Tree<T> right)
    {
        assert left.invariant() && right.invariant() &&
        (left.isEmpty() || value.compareTo(left.value()) > 0) &&
        (right.isEmpty() || value.compareTo(right.value()) < 0) &&
        (right.isEmpty() || right.isBlack()) &&
        (!left.isEmpty() || right.isEmpty());

        this.value = value;
        this.left = left;
        this.right = right;
        this.size = 1 + left.size() + right.size();

        assert invariant();
    }
    ...
}
```

Invariante das árvores pretas

- Como de costume, é análogo à pré-condição do construtor:

```
public boolean invariant()
{
    return left.invariant() && right.invariant()
        && (!left.isEmpty() || value.compareTo(left.value()) > 0)
        && (!right.isEmpty() || value.compareTo(right.value()) < 0)
        && (right.isEmpty() || right.isBlack())
        && (!left.isEmpty() || right.isEmpty());
}
```

Repare: a penúltima linha significa que a subárvore direita tem de ser preta, se não for vazia; a última linha significa que se a subárvore esquerda for vazia, então a subárvore direita também é. A segunda linha e a terceira exprimem (recursivamente) que a árvore é uma árvore binária de busca.

Função has

- A função **has** é copiada da das árvores binárias de busca:

```
public boolean has(T x)
{
    boolean result = true;
    int cmp = x.compareTo(value);
    if (cmp < 0)
        result = left.has(x);
    else if (cmp > 0)
        result = right.has(x);
    return result;
}
```

- Note que as chamadas recursivas invocarão a função **has** de subárvores pretas, **vermelhas** ou vazias.

Função put

- A função **put** é o busílis.
- Por analogia com as árvores dois-três, convém deixar o trabalho para uma função recursiva, protegida, **putr**:

```
public Tree<T> put(T x)
{
    Tree<T> result = putr(x);
    if (result.isTransient())
        result = result.split();
    assert result.invariant();
    return result;
}
```

Compare com a função **put** da classe **Three**, das árvores 2-3.

```
protected Tree<T> putr(T x)
{
    ...
}
```

Função putr

- Caso especial: se o valor a inserir é igual ao valor da raiz da árvore preta, não há nada a fazer:

```
public Tree<T> putr(T x)
{
    Tree<T> result = this;
    int cmp = x.compareTo(value);
    if (cmp == 0)
        return result;
    ...
}
```

Análogo à função homónima da classe TwoCons, das árvores 2-3.

Função putr, folhas

- Caso simples: se a árvore preta for uma folha, transforma-se numa árvore ternária, ligando-se à esquerda a uma nova árvore **vermelha**:

```
...
assert (!left.isEmpty() || right.isEmpty());

if (left.isEmpty()) // this is a black node that is a leaf
{
    assert right.isEmpty();
    // just turn this into a ternary node
    if (cmp < 0)
    {
        result = ternary(x, value, left, left, left);
    }
    else
    {
        result = ternary(value, x, left, left, left
    }
}
else
{
    ...
}
```

Nota: a função **ternary** cria um nó duplo, formado por uma árvore preta ligada à esquerda a uma árvore **vermelha**. Esse nó duplo representa um nó ternário nas árvores 2-3.

Função putr, duas subárvore pretas

- Se houver duas subárvore pretas, então primeiro insere-se do lado certo.
- Depois, se o resultado for uma árvore transiente, parte-se a árvore e cria-se um nó duplo.
- Se não, o resultado, isto é, a subárvore com a inserção feita, substitui a subárvore original.
- Até aqui, é análogo ao caso da classe **Two**, das árvores 2-3, com as devidas adaptações.

Função putr, árvores pretas “internas”

- Observe:

```
else if (isTwo()) // This is a black node with two black subtrees.  
{  
    assert (!left.isEmpty());  
    assert (!right.isEmpty());  
    assert (left.isBlack());  
    assert (right.isBlack());  
  
    if (cmp < 0)  
    {  
        Tree<T> t = left.putr(x);  
        if (t.isTransient())  
        {  
            t = t.split();  
            result = ternary(t.value(), value, t.left(), t.right(), right);  
        }  
        else  
            result = new Tree<>(value, t, right);  
    }  
    else  
        ...  
}
```

Este parte corresponde a inserir do lado esquerdo. Para o lado direito é análogo.

Função putr, nós duplos

- Se a árvore preta tiver uma subárvore esquerda **vermelha**, então corresponde a um nó ternário da árvore dois-três.
- Programamos por analogia com o caso da classe **Three**, das árvores dois-três.
- Se o valor a inserir for igual ao valor da raiz da subárvore **vermelha**, não há nada a fazer:

```
else
{
    // this is a black node whose left subtree is red
    assert (isThree());
    assert (!left.isEmpty() || right.isEmpty());
    assert (left.isRed());
    assert (right.isEmpty() || right.isBlack());
    int cmpRed = x.compareTo(left.value());
    if (cmpRed == 0)
        return result;
    ...
}
```

Função putr, nós duplos, caso geral

- É análogo ao caso das árvores Three.
- Se o nó duplo corresponde a uma árvore ternária que é uma folha, então o resultado é uma árvore transiente, castanha.
- Se o nó duplo corresponde a uma árvore ternária que não é uma folha, então insere-se na subárvore apropriada, consoante o valor for menor que valor da raiz da subárvore vermelha, maior que o valor da raiz da árvore preta ou estiver entre os dois.
- Tal como antes, a árvore retornada pela chamada recursiva substitui a subárvore respetiva, a não ser que corresponda a uma árvore quaternária, transiente, a qual então é preciso partir.
- No caso em que há uma árvore quaternária, o valor que “subiu” junta-se ao valor da raiz da subárvore vermelha e ao valor da raiz da árvore preta para construir uma árvore, desta vez quaternária, que há de ser partida pela função que chamou esta.

Função putr, nós duplos folhas

- Observe:

```
assert (left.left().isEmpty() && left.right().isEmpty() && right.isEmpty())
|| (!left.left().isEmpty() && !left.right().isEmpty() && !right.isEmpty());

if (right.isEmpty()) // this is a double node that is a leaf
{
    assert left.left().isEmpty()
        && left.right().isEmpty()
        && right.isEmpty();
    if (cmpRed < 0)
        result = new Brown<>(x, left.value(), value,
                               right, right, right, right);
    else if (cmp > 0)
        result = new Brown<>(left.value(), value, x,
                               right, right, right, right);
    else
        result = new Brown<>(left.value(), x, value,
                               right, right, right, right);
}
...

```

Nota: aqui, **right** é a árvore vazia.

Função putr, nós duplos internos, à esquerda

- Inserir à esquerda, isto é, na subárvore esquerda da subárvore **vermelha**:

```
else // This is double node that is NOT a leaf.  
{  
    assert !left.left().isEmpty()  
        && !left.right().isEmpty()  
        && !right.isEmpty();  
    if (cmpRed < 0) // x is inserted to the left.  
    {  
        Tree<T> t = left.left().putr(x);  
        if (t.isTransient())  
        {  
            t = t.split();  
            result = new Brown<>(t.value(), left.value(), value,  
                t.left(), t.right(), left.right(), right);  
        }  
        else  
            result = ternary(left.value(), value, t, left.right(), right);  
    }  
    else  
        ...  
}
```

Função putr, nós duplos internos, à direita

- Inserir à direita, isto é, na subárvore direita (que é preta) da árvore preta:

```
else // This is double node that is NOT a leaf.  
{  
    ...  
    if (cmpRed < 0) // x is inserted to the left.  
    {  
        ...  
    }  
    else if (cmp > 0) // x is inserted to the right  
    {  
        Tree<T> t = right().putr(x);  
        if (t.isTransient())  
        {  
            t = t.split();  
            result = new Brown<>(left.value(), value, t.value(),  
                left.left(), left.right(), t.left(), t.right());  
        }  
        else  
            result = new Black<>(value, left, t); }  
    else  
    {  
        ...  
    }  
}
```

Este caso é mais simples que o anterior, porque a árvore **t** pode ser usada diretamente. No caso anterior, a árvore **t** tinha de ser integrada num nó duplo.

Função putr, nós duplos internos, ao meio

- Inserir ao meio, isto é, na subárvore direita da subárvore **vermelha**:

```
else // This is like a 3-node that is NOT a leaf.  
{  
    if (cmpRed < 0) // x is inserted to the left.  
    {  
        ...  
    }  
    else if (cmp > 0) // x is inserted to the right  
    {  
        ...  
    }  
    else  
    {  
        Tree<T> t = left.right().putr(x);  
        if (t.isTransient())  
        {  
            t = t.split();  
            result = new Brown<>(left.value(), t.value(), value,  
                                  left.left(), t.left(), t.right(), right);  
        }  
        else  
            result = ternary(), value, left.left(), t, right);  
    }  
}
```

Este caso é análogo ao primeiro, mas agora trata-se da subárvore direita da subárvore **vermelha**.

Função putr, inteira

Esta é certamente a maior função da nossa cadeira: tem 121 linhas, incluindo comentários e asserções.

```
public Tree<T> putr(T x)
{
    Tree<T> result = this;
    // If x is equal to the value, nothing to do.
    int cmp = x.compareTo(value);
    if (cmp == 0)
        return result;
    // Remember: this is a black node: if the left subtree is empty, the right subtree must be empty.
    assert (!left.isEmpty() || right.isEmpty());
    // Is this a binary node that is a leaf?
    if (left.isEmpty()) // This is a black node that is a leaf.
    {
        assert right.isEmpty();
        // Just turn this into a ternary node
        if (cmp < 0)
        {
            result = ternary(x, value, left, left, left); // left is empty
        }
        else
        {
            result = ternary(value, x, left, left, left); // left is empty
        }
    }
    else if (isTwo()) // This is a black node whose left subtree is black.
    {
        assert (!left.isEmpty());
        assert (!left.isBlack());
        assert (left.isBlack());
        assert (right.isBlack());
        if (cmp < 0)
        {
            Tree<T> t = left.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = ternary(t.value(), value, t.left(), t.right(), right);
            }
            else
                result = new Black<>(value, t, right);
        }
        else
        {
            Tree<T> t = right.putr(x);
            if (t.isTransient())
            {
                t = t.split();
                result = ternary(value, t.value(), left, t.left(), t.right());
            }
            else
                result = new Black<>(value, left, t);
        }
    }
    else
    {
        // This is a black node whose left subtree is red. It corresponds to a Three node, in 2-3 trees.
        assert (!isThree());
        assert (!left.isEmpty() || right.isEmpty());
        assert (left.isRed());
        assert (right.isEmpty() || right.isBlack());
        int cmpRed = x.compareTo(left.value());
        if (cmpRed == 0)
            result = result;
        assert (left.left().isEmpty() && left.right().isEmpty() && right.isEmpty());
        || (left.left().isEmpty() && !left.right().isEmpty() && !right.isEmpty());
        if (right.isEmpty()) // This is a 3-node that is a leaf
        {
            assert left.left().isEmpty() && left.right().isEmpty() && right.isEmpty();
            if (cmpRed < 0)
                result = new Brown<>(x, left.value(), value, right, right, right, right); // right is empty.
            else if (cmp > 0)
                result = new Brown<>(left.value(), value, x, right, right, right, right);
            else
                result = new Brown<>(left.value(), x, value, right, right, right, right);
        }
        else // This is like a 3-node that is NOT a leaf.
        {
            assert !left.left().isEmpty() && !left.right().isEmpty() && !right.isEmpty();
            if (cmpRed < 0) // x is inserted to the left.
            {
                Tree<T> t = left.left().putr(x); // Insert on the left of the red node.
                if (t.isTransient())
                {
                    t = t.split();
                    result = new Brown<>(t.value(), left.value(), value, t.left(), t.right(), left.right(), right);
                }
                else
                    // Here we have to rebuild the red-black pair, with the new left subtree of the red node.
                    result = ternary(left.value(), value, t, left.right(), right);
            }
            else if (cmp > 0) // x is inserted to the right (directly: this is a black node)
            {
                Tree<T> t = right().putr(x);
                if (t.isTransient())
                {
                    t = t.split();
                    result = new Brown<>(left.value(), value, t.value(), left.left(), left.right(), t.left(), t.right());
                }
                else
                    result = new Black<>(value, left, t); // the new tree replaces the right subtree, so to speak.
                    // This case is simpler than the other two, because the right subtree is black.
                    // On the previous case, the red-pair node had to be rebuilt.
            }
            else
            {
                // x is inserted in the middle, i.e. to the right of the red node
                Tree<T> t = left.right().putr(x);
                if (t.isTransient())
                {
                    t = t.split();
                    result = new Brown<>(left.value(), t.value(), value, left.right(), t.left(), t.right(), right); // last bug found! :-)
                }
                else
                    // Here we have to rebuild the red-black pair, with the new right subtree of the red node.
                    // This is similar to the first case, but now to the right.
                    result = ternary(left.value(), value, left.left(), t.left(), t.right(), right);
            }
        }
    }
    return result;
}
```

Funções auxiliares

- A função **ternary**, constrói um nó duplo, vermelho-preto:

```
protected Black<T> ternary(T x, T y,
                           Tree<T> left, Tree<T> middle, Tree<T> right)
{
    assert x.compareTo(y) < 0;
    Red<T> r = new Red<T>(x, left, middle);
    return new Black<>(y, r, right);
}
```

- A função **isTwo** verifica se a árvore representa um nó binário:

```
// A black node represents a binary 2-3 tree if it is a leaf or both subtrees are black
public boolean isTwo()
{
    return left.isEmpty() || left.isBlack();
    // Note: by the invariant, the right subtree is black, if it is not empty.
}
```

- A função **isThree** verifica se a árvore corresponde a um nó ternário:

```
// A black node represents a ternary 2-3 tree if the left subtree is red.
public boolean isThree()
{
    return !left.isEmpty() && left.isRed();
}
```

Na verdade, esta função não é usada.

Classe Red<T>

- As árvores **vermelhas** têm pouca funcionalidade.
- Todas as funções são triviais ou idênticas às das árvores pretas.
- A função **put** é proibida: não faz sentido acrescentar diretamente às árvores **vermelhas**, pois elas são parte de nós duplos e entram sempre como subárvore esquerdas de árvores pretas.

```
class Red<T extends Comparable<T>> extends Tree<T>
{
    ...
    public Tree<T> put(T x)
    {
        throw new UnsupportedOperationException();
    }

    protected Tree<T> putr(T x)
    {
        throw new UnsupportedOperationException();
    }
    ...
}
```

Árvores vermelhas

- Apenas o invariante é ligeiramente diferente:

```
class Red<T extends Comparable<T>> extends Tree<T>
{
    private final T value;
    private final Tree<T> left;
    private final Tree<T> right;
    private final int size;

    public Red(T value, Tree<T> left, Tree<T> right)
    {
        assert left.invariant() && right.invariant()
        && (left.isEmpty() == right.isEmpty())
        && (left.isEmpty() || value.compareTo(left.value()) > 0)
        && (right.isEmpty() || value.compareTo(right.value()) < 0)
        && (left.isEmpty() || left.isBlack())
        && (right.isEmpty() || right.isBlack());

        this.value = value;
        this.left = left;
        this.right = right;
        this.size = 1 + left.size() + right.size();

        assert invariant();
    }

    public boolean invariant()
    {
        return left.invariant() && right.invariant()
            && (left.isEmpty() == right.isEmpty())
            && (left.isEmpty() || value.compareTo(left.value()) > 0)
            && (right.isEmpty() || value.compareTo(right.value()) < 0)
            && (left.isEmpty() || left.isBlack())
            && (right.isEmpty() || right.isBlack());
    }
}
```

Classe Brown<T>

- Usamos árvores castanhas para representar as árvores quaternárias efémeras:

```
class Brown<T extends Comparable<T>> extends Tree<T>
{
    private final T valueLeft;
    private final T valueMiddle;
    private final T valueRight;
    private final Tree<T> left;
    private final Tree<T> middleLeft;
    private final Tree<T> middleRight;
    private final Tree<T> right;

    public Brown(T valueLeft, T valueMiddle, T valueRight,
                Tree<T> left, Tree<T> middleLeft,
                Tree<T> middleRight, Tree<T> right)
    {
        assert valueLeft.compareTo(valueMiddle) < 0;
        assert valueMiddle.compareTo(valueRight) < 0;
        this.valueLeft = valueLeft;
        this.valueMiddle = valueMiddle;
        this.valueRight = valueRight;
        this.left = left;
        this.middleLeft = middleLeft;
        this.middleRight = middleRight;
        this.right = right;
    }
}
```

Árvores castanhas, split e isTransient

- Tal como na classe **FourCons**, as únicas operações interessantes são **split** e **isTransient**.
- Ambas são análogas às da classe **FourCons**
- A função **split** constrói árvore uma preta com subárvores pretas:

```
protected Black<T> split()
{
    Black<T> newLeft = new Black<>(valueLeft, left, middleLeft);
    Black<T> newRight = new Black<>(valueRight, middleRight, right);
    return new Black<>(valueMiddle, newLeft, newRight);
}
```

- A função **isTransient** retorna **true**:

```
protected boolean isTransient()
{
    return true;
}
```

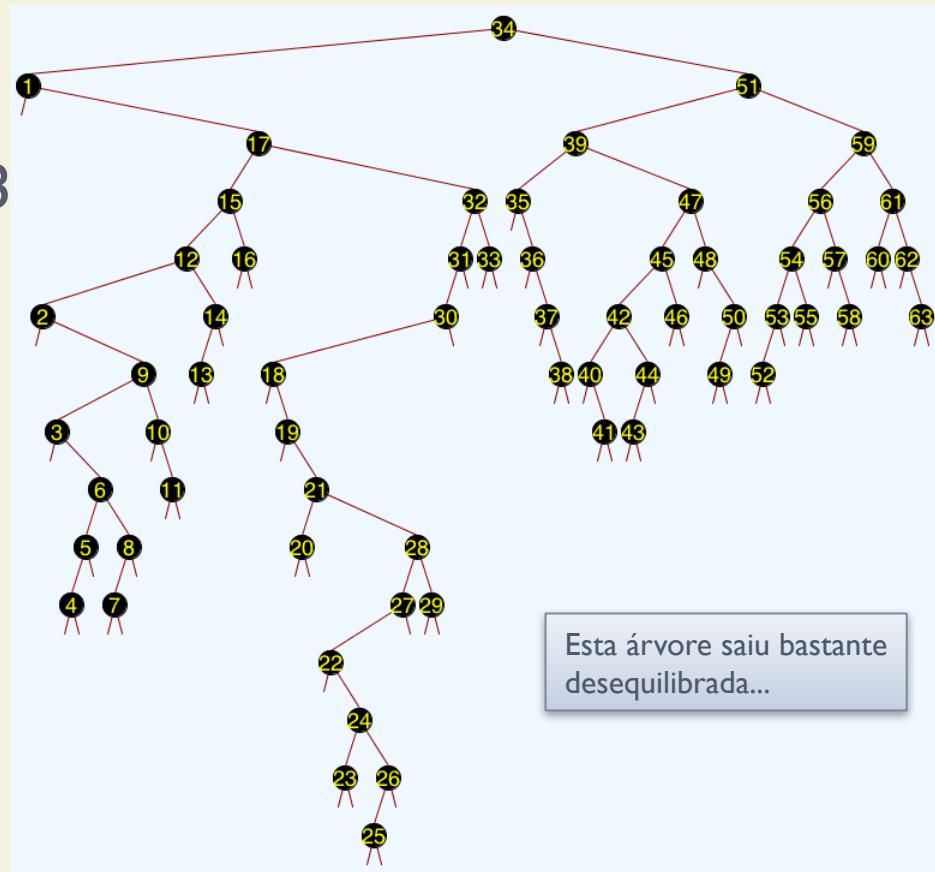
Nas outras classes,
retorna **false**.

E assim concluímos a programação da árvores **red-black**. As funções que não considerámos aqui explicitamente são triviais.

Experiência

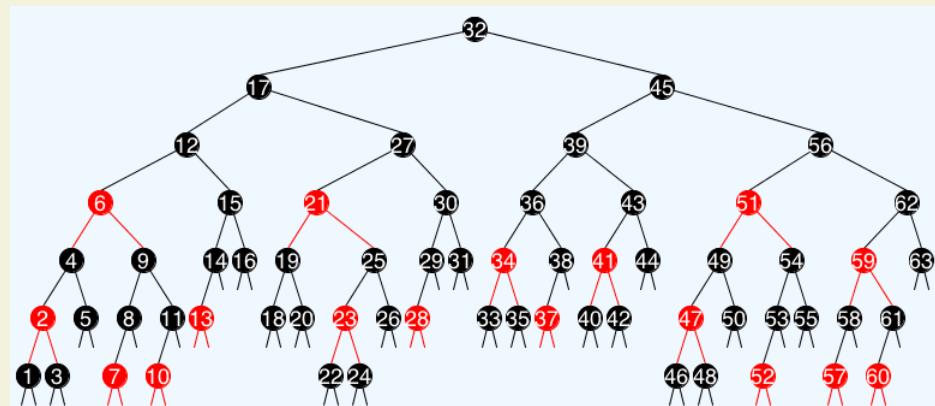
- Árvore binária de busca, com 63 valores, de 1 a 63, inseridos por ordem aleatória:

A altura desta árvore é 15.



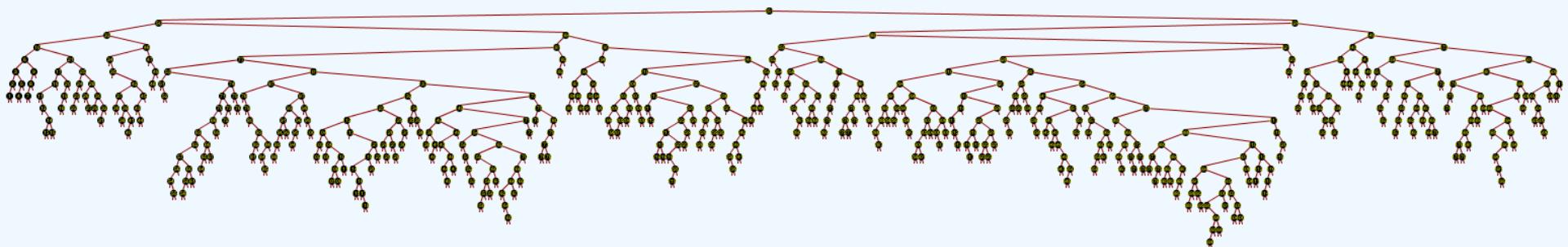
- Árvore **red-black**, com os mesmos valores, inseridos pela mesma ordem.

A altura desta árvore é 7.



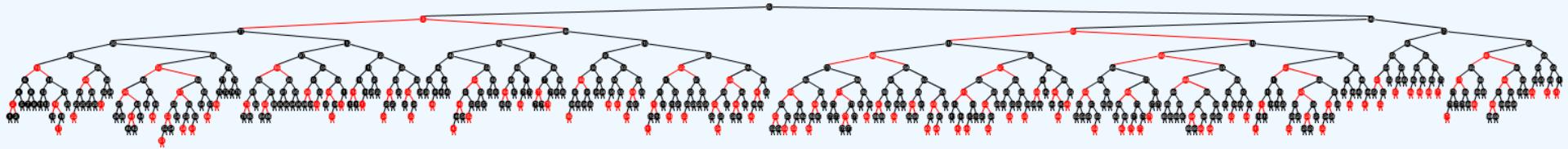
Outra experiência, maior

- Árvore binária de busca, com 511 valores, de 1 a 511, inseridos por ordem aleatória:



A altura desta árvore é 20.

- Árvore **red-black**, com os mesmos 511 valores, inseridos pela mesma ordem:



A altura desta árvore é 12.

Moral da história

- As árvores **red-black** são fantásticas!
- Sendo árvores binárias de busca, podemos acrescentar-lhe as funções que programámos na nossa classe original para árvores binárias de busca, sem complicações: funções de ordem, iteradores, **draw**, etc.
- Por outro lado, falta programar a função **delete**.
- E faltaria experimentar implementar isto tudo com árvores de nós, mutáveis.

Apontamento final

- As árvores **red-black** foram publicadas no artigo “[A dichromatic framework for balanced trees](#)”, de Leonidas J. Guibas e Robert Sedgewick, in 1978.
- Segundo Sedgewick, o nome “**red-black**” tem a seguinte [explicação](#):

A lot of people ask why did we use the name red–black. Well, we invented this data structure, this way of looking at balanced trees, at Xerox PARC which was the home of the personal computer and many other innovations that we live with today including graphic user interfaces, Ethernet and object-oriented programming and many other things. But one of the things that was invented there was laser printing and we were very excited to have nearby a color laser printer that could print things out in color and out of the colors the red looked the best. So, that's why we picked the color red to distinguish red links, the types of links, in tree nodes. So, that's an answer to the question for people that have been asking.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 27

Grafos e buscas em grafos

Grafos

- Grafos em programação.
- Grafos dirigidos e não dirigidos.
- Grafos pesados.
- Busca em largura.
- Busca em profundidade.

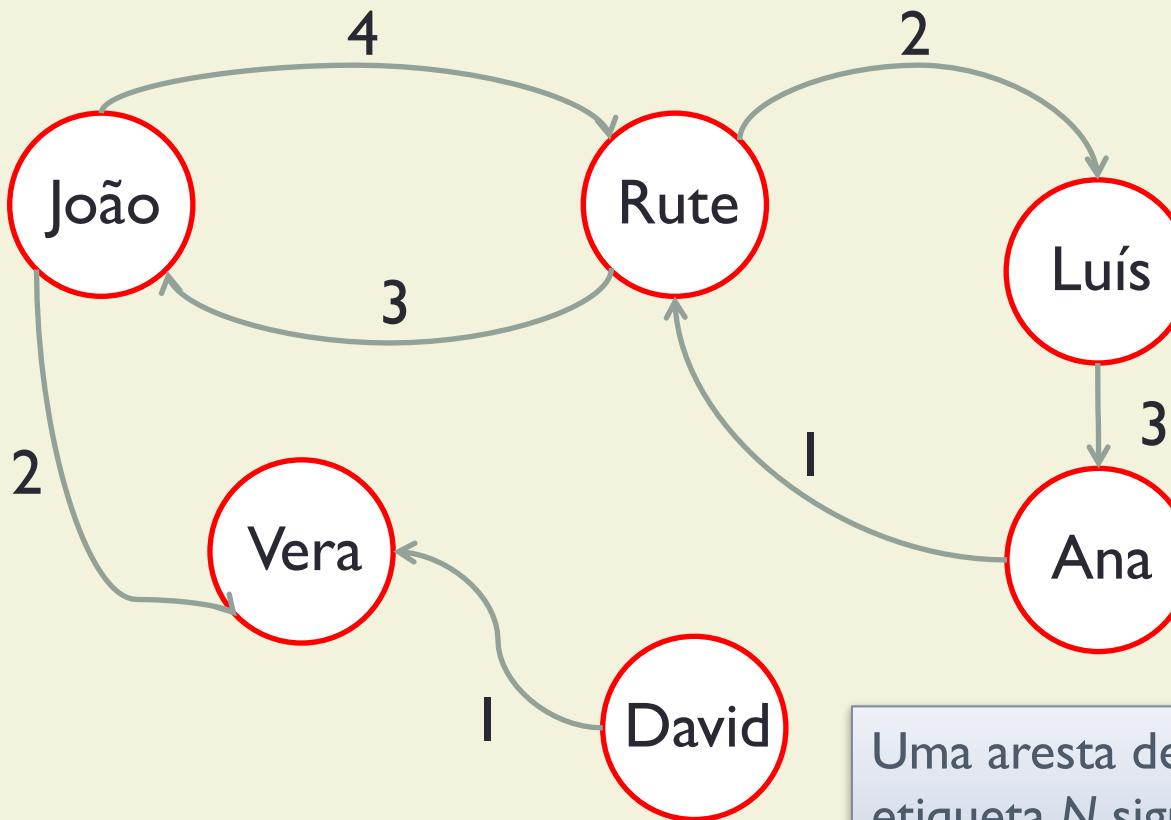


Grafos

- Um grafo é um conjunto de **vértices** mais uma coleção de **arestas**.
- Os vértices são o que nós quisermos: números, pontos, cadeias de caracteres, pessoas, cidades, etc.
- As arestas são pares ordenados de vértices, associados a um número, dito o **peso** da aresta.
- Frequentemente, visualizamos os grafos representando os vértices por meio de pontos e as arestas por meio de setas ou segmentos.

Exemplo: livros emprestados

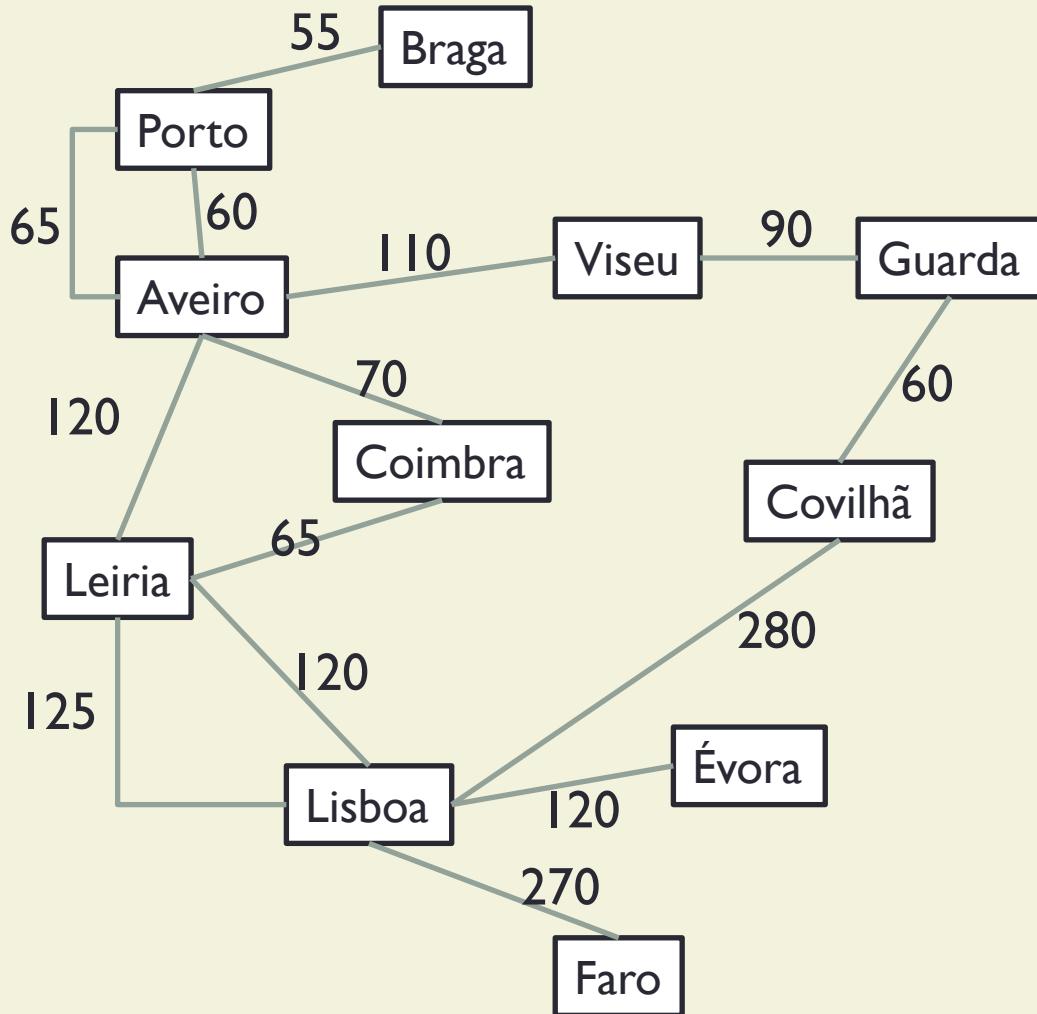
- Um grafo que representa o número de livros emprestados entre um conjunto de amigos:



Uma aresta de X para Y com etiqueta N significa que X emprestou N livros a Y .

Exemplo: autoestradas

- Grafo das autoestradas em Portugal



Neste caso, cada ligação representa duas arestas, uma para cada lado, ambas com o mesmo peso.

Note que há duas autoestradas entre Lisboa e Leiria e também entre Aveiro e Porto.

Exemplo: quadrilha

Quadrilha, de Carlos Drummond de Andrade:

*João amava Teresa que amava Raimundo
que amava Maria que amava Joaquim que amava Lili,
que não amava ninguém.
João foi para os Estados Unidos, Teresa para o convento,
Raimundo morreu de desastre, Maria ficou para tia,
Joaquim suicidou-se e Lili casou com J. Pinto Fernandes
que não tinha entrado na história.*



Uma seta de X para Y significa “X amava Y”. Neste caso, o peso não é importante. É como se valesse 1 em todos os casos.

Grafos em programação

- Representamos os vértices por números inteiros, no intervalo de 0 até ao número de vértices menos um.
- Representamos cada aresta por um triplo: vértice de partida, vértice de chegada e peso.
- A cada vértice, associamos o saco das arestas que dele partem.
- Juntamos o saco de todas as arestas:

```
public class Graph
{
    public final int nVertices;      // number of vertices
    private Bag<Edge>[] adj;        // one bag for each vertex
    private Bag<Edge> edges;        // bag of all edges
    ...
}
```

Mais convencionalmente, usariámos listas, em vez de sacos.

Classe Edge

```
public class Edge
{
    public final int from;
    public final int to;
    public final double weight;

    public Edge(int from, int to, double weight)
    {
        assert from != to;
        this.from = from;
        this.to = to;
        this.weight = weight;
    }

    public Edge(int from, int to)
    {
        this(from, to, 1.0);
    }

    public Edge reverse()
    {
        return new Edge(to, from, weight);
    }

    public static int compareByWeight(Edge e1, Edge e2)
    {
        double z = e1.weight - e2.weight;
        return z < 0 ? -1 : z > 0 ? 1 : 0;
    }
}
```

Esta função de comparação
será usada mais adiante.

Classe Graph, construtor

```
public class Graph
{
    public final int nVertices;      // number of vertices
    private Bag<Edge>[] adj;       // one bag for each vertex
    private Bag<Edge> edges;        // bag of all edges

    @SuppressWarnings("unchecked")
    public Graph(int nVertices)
    {
        this.nVertices = nVertices;
        edges = new Bag<>();
        adj = (Bag<Edge>[]) new Bag[nVertices];
        for (int i = 0; i < nVertices; i++)
            adj[i] = new Bag<Edge>();
    }
    ...
}
```

De início, fica tudo vazio.

Classe Graph, acrescentar arestas

- A operação básica de acrescentar uma aresta é a seguinte:

```
public void addEdgeDirected(Edge x)
{
    edges.add(x);
    adj[x.from].add(x);
}
```

Note que a mesma aresta é acrescentada a dois sacos: o saco geral e o saco do vértice de partida.

- Frequentemente, queremos acrescentar ao mesmo tempo uma aresta e a sua inversa:

```
public void addEdge(Edge x)
{
    addEdgeDirected(x);
    addEdgeDirected(x.reverse());
}
```

Note bem: não controlamos isso no nosso programa, mas nesta fase é proibido acrescentar mais do que uma aresta com o mesmo vértice de partida e o mesmo vértice de chegada ou uma aresta em que o vértice de partida seja igual ao vértice de chegada.

Iterando vértices, arestas

- Iteramos os vértices enumerando (num ciclo for...) os números inteiros de 0 a nVertices-1.
- Iteramos as arestas por meio das seguintes funções:

```
public Iterable<Edge> edges()
{
    return edges;
}

public Iterable<Edge> adj(int v)
{
    return adj[v];
```

Fora da classe ninguém sabe que na classe as arestas estão guardadas em sacos.

Grafos não dirigidos

- Um grafo **não dirigido** é um grafo onde quando existe uma aresta também existe a sua inversa.
- Se essa propriedade não se verificar, o grafo é um grafo **dirigido**.
- Certas questões dizem respeito apenas a grafos dirigidos; outras não distinguem, mas a interpretação do resultado pode variar, consoante o grafo for dirigido ou não.
- Nos exemplos introdutórios, usaremos apenas grafos não dirigidos.

Problema da busca

- O problema da busca num grafo consiste em calcular os vértices que são **alcançáveis** a partir de um vértice dado.
- Um vértice é alcançável a partir de outro se existir no grafo um **caminho** que vai do outro até ele.
- Um caminho é uma sequência de arestas e_1, e_2, \dots, e_N , tal que $e_i.to == e_{i+1}.from$.
- Acessoriamente, calculamos, para cada vértice alcançável, um caminho do vértice de partida até esse vértice.

Classe abstrata Search

- Representamos o problema da busca num grafo por meio da classe abstrata **Search**:

```
public abstract class Search
{
    public final Graph g;
    public final int start;

    public abstract boolean isMarked(int v);
    public abstract int count();
    public abstract Iterable<Integer> result();
    public abstract Edge edgeTo(int v);
    public abstract Iterable<Edge> searchTree();
    public Iterable<Edge> pathTo(int v) {...}
}
```

Todas as funções são abstratas, exceto **pathTo**.

Note bem: no problema da busca, o peso das arestas não interessa.

Os vértices *marcados* são os vértices alcançáveis; a função **count** diz quantos são; a função **result** enumera-os; **edgeTo(v)** indica a aresta com a qual chegámos ao vértice **v**; a função **searchTree** enumera as arestas visitadas durante a busca; **pathTo(v)** enumera as arestas que compõem um caminho de **start** até **v**.

Busca em profundidade

- A busca em profundidade é uma estratégia de busca num grafo.
- Consideremos a analogia com uma rede de túneis, por exemplo do metro.
- Para passear nos túneis, visitando todas as estações, podemos adotar a seguinte disciplina:
 - Inicialmente, por hipótese, toda a rede está às escuras.
 - A chegar a uma estação que está às escuras, acendemos a luz dessa estação e marcamos o túnel que nos trouxe até à estação; depois, espreitamos por cada um dos túneis que partem dessa estação: se houver uma luz ao fundo do túnel, esse túnel não interessa, porque a estação na outra ponta já foi visitada anteriormente; inversamente se algum dos túneis não tiver luz ao fundo, esse interessa: seguimos por ele e, chegados à estação que está ao fim do túnel, procedemos de acordo com a mesma disciplina (acendendo a luz, marcando o túnel de chegada, etc.)
 - Se, ao chegar a uma estação, nenhum dos túneis interessar, retrocedemos, usando o túnel pelo qual tínhamos chegado a essa estação da primeira vez (o qual nós tínhamos deixado marcado).
 - Quando, nestas voltas, passarmos de novo pela estação inicial e já não houver nenhuns túneis que interessem, todas as estações alcançáveis estarão iluminadas.

Classe DepthFirstSearch

- Implementa a classe abstrata Search:

```
class DepthFirstSearch extends Search
{
    private Bag<Integer> result;
    private boolean[] marked;
    private Edge[] edgeTo;
    private Bag<Edge> searchTree;

    public DepthFirstSearch(Graph g, int s)
    {
        super(g, s);
        result = new Bag<Integer>();
        marked = new boolean[g.nVertices];
        edgeTo = new Edge[g.nVertices];
        searchTree = new Bag<>();
        dfs(s);
    }

    private void dfs(int v)
    {
        ...
    }
}
```

Em **result**, guardamos as estações visitadas; **marked[x]** vale **true** se a luz da estação **x** estiver acesa; **edgeTo[x]** assinala o túnel pelo qual chegámos à estação **x** pela primeira vez; **searchTree** coleciona os túneis que percorremos.

Os cálculos são feitos no construtor, pela função **dfs**.

Função dfs

- Observe. É um clássico:

```
private void dfs(int v)
{
    marked[v] = true;
    result.add(v);
    for (Edge x : g.adj(v))
        if (!marked[x.to])
    {
        edgeTo[x.to] = x;
        searchTree.add(x);
        dfs(x.to);
    }
}
```

As outras funções

- São simples:

```
public boolean isMarked(int v)
{
    return marked[v];
}

public int count()
{
    return result.size();
}

public Iterable<Integer> result()
{
    return result;
}

public Edge edgeTo(int x)
{
    return edgeTo[x];
}

public Iterable<Edge> searchTree()
{
    return searchTree;
}
```

Função pathTo

- Também é muito interessante:

```
public Iterable<Edge> pathTo(int v)
{
    Stack<Edge> result = null;
    if (isMarked(v))
    {
        result = new Stack<>();
        for (int x = v; x != start; x = edgeTo(x).from)
            result.push(edgeTo(x));
    }
    return result;
}
```

Recorde que colocámos esta função na classe abstrata.

Note que se o vértice **v** não estiver marcado, o resultado é **null**, assinalando que não existe caminho de **start** até **v**; se **v** for igual a **start**, existe caminho, mas é vazio.

Busca em largura

- A busca em largura é outra estratégia de busca num grafo.
- Consideremos a mesma analogia com uma rede de túneis, que vai neste caso ser explorada por um robô que tem a capacidade de se clonar.
- Adotamos a seguinte disciplina:
- Inicialmente, tal como antes, toda a rede está às escuras.
- Quando o robô chega a uma estação que esteja às escuras, acende a luz, marca o túnel que o trouxe até ali, espreita cada um dos túneis que saem dessa estação e, se não houver luz ao fundo desse túnel, clona-se e faz o clone seguir por esse túnel, para cada um desses túneis, ao mesmo tempo. Os clones terão o mesmo comportamento: quando chegarem à estação ao fundo do túnel, acenderão a luz (se a luz ainda estiver apagada), etc.
- Se não houver túneis em que a estação ao fundo esteja apagada, o robô desfaz-se em fumo.
- Pode acontecer que, em algum momento, dois ou mais robôs se dirijam para a mesma estação, através de túneis diferentes. Nesse caso, ganha o que chegar primeiro: é esse que acende a luz, e que depois se clona. Os que chegam depois desfazem-se em fumo.
- Quando não houver mais robôs, todas as estações alcançáveis estarão iluminadas.

Classe BreadthFirstSearch

- Implementa também a classe abstrata Search:

```
class BreadthFirstSearch extends Search
{
    private Bag<Integer> result;
    private boolean[] marked;
    private Edge[] edgeTo;
    private Bag<Edge> searchTree;

    public BreadthFirstSearch(Graph g, int s)
    {
        super(g, s);
        result = new Bag<Integer>();
        marked = new boolean[g.nVertices];
        edgeTo = new Edge[g.nVertices];
        searchTree = new Bag<>();
        bfs(s);
    }

    private void bfs(int v)
    {
        ...
    }
    ...
}
```

É igual à outra, exceto que na função que faz os cálculos, agora chamada **bfs**.

Função bfs

- Outro clássico:

```
private void bfs(int v)
{
    marked[v] = true;
    result.add(v);
    Queue<Integer> q = new Queue<>();
    q.enqueue(v);
    while (!q.isEmpty())
    {
        int w = q.dequeue();
        for (Edge x : g.adj(w))
            if (!marked[x.to])
            {
                marked[x.to] = true;
                result.add(x.to);
                edgeTo[x.to] = x;
                searchTree.add(x);
                q.enqueue(x.to);
            }
    }
}
```

As outras funções são iguais às da classe DepthFirstSearch.

Caminho mais curto

- Na classe **BreadthFirstSearch**, o resultado de **pathTo(v)** representa o caminho mais curto de **start** até **v**.
- Mais precisamente, se houver vários caminhos com comprimento mínimo, isto é, com comprimento igual ao comprimento do caminho mais curto, representa um deles.
- Havendo vários caminhos mais curtos, aquele dentre eles que será calculado pela função depende da ordem por que as arestas foram acrescentadas ao grafo.

Relação com o atravessamento de árvores

- De certa forma, os algoritmos de atravessamento de árvores (lição 24) são casos particulares das buscas em grafos.
- Com efeito, uma árvore é um grafo dirigido, com um vértice distinto—a raiz—e tal que da raiz até qualquer outro vértice há exatamente um caminho.
- Numa árvore binária, de cada vértice saem quando muito duas arestas.
- A busca em profundidade corresponde ao atravessamento em profundidade e a busca em largura corresponde ao atravessamento por nível.



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

Algoritmos e Estruturas de Dados

Lição n.º 28

Grafos: Prim e Dijkstra

Grafos: Prim e Dijkstra

- Conectividade.
- Árvore de cobertura mínima.
- Caminho mais curto.



Conectividade

- Problema da conectividade num grafo (não dirigido): dados dois nós, há um caminho de um até ao outro?
- Num grafo, uma **componente conexa** é um subconjunto maximal de vértices que estão todos conectados uns com os outros.
- Portanto, haverá um caminho de um vértice para outro se e só se pertencerem à mesma componente conexa.
- Problema: calcular as componentes conexas de um grafo.

Classe ConnectedComponents

```
public class ConnectedComponents
{
    private int[] id;
    private int count;

    public ConnectedComponents(Graph g)
    {
        ...
    }

    public boolean connected(int v, int w)
    {
        return id[v] == id[w];
    }

    public int id(int v)
    {
        return id[v];
    }

    ...
}
```

Os cálculos fazem-se no construtor.

Isto é parecido com o Union-Find, mas recorde que nesse caso, tínhamos conectividade dinâmica, isto é, as ligações iam chegando. Aqui, as arestas já estão todas no grafo quando calculamos as componentes conexas.

Calculando as componentes

- Calculamos as componentes usando DFS, sucessivamente:

```
public ConnectedComponents(Graph g)
{
    id = new int[g.nVertices];
    for (int i = 0; i < id.length; i++)
        id[i] = -1;
    int n = 0;
    for (int i = 0; i < id.length; i++)
        if (id[i] == -1)
    {
        DepthFirstSearch dfs = new DepthFirstSearch(g, i);
        for (int x : dfs.result())
            id[x] = n;
        n++;
    }
    count = n;
}
```

Isto é o construtor da classe ConnectedComponents.

Enumerando os vértices de uma componente

- Usamos sacos internamente e devolvemos um objeto iterável:

```
public Iterable<Integer> component(int x)
{
    Bag<Integer> result = new Bag<>();
    for (int i = 0; i < id.length; i++)
        if (id[i] == x)
            result.add(x);
    return result;
}
```

Encerramento de autoestradas

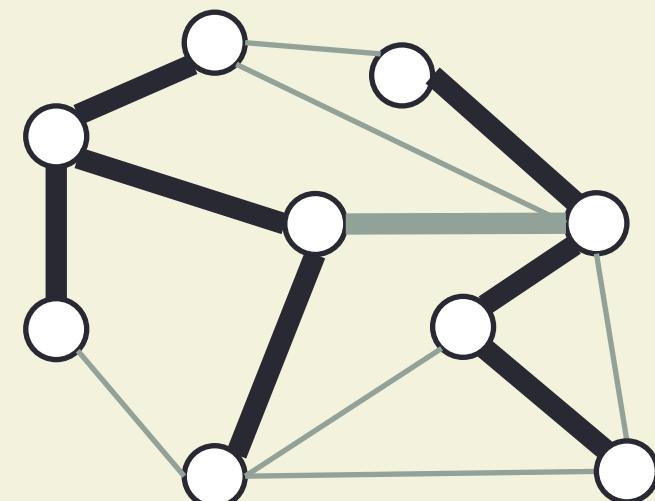
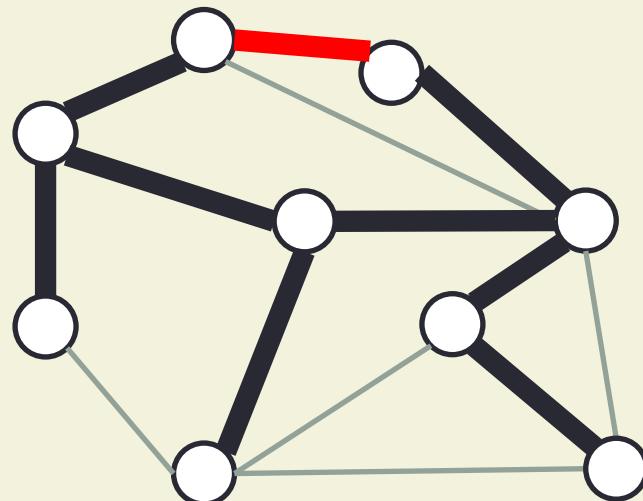
- Considere a seguinte situação ficcional:
- Um certo país dispõe de uma rede de autoestradas sem portagem, que liga todas as cidades. Por causa da crise, o governo vai ter de encerrar algumas dessas autoestradas, para poupar nos custos de manutenção. No entanto, para evitar problemas sociais, tem de garantir que todas as cidades continuam servidas por pelo menos uma autoestrada e que de cada cidade se pode ir, sempre por autoestrada, até qualquer outra cidade (ainda que dando uma volta maior que antes...) tal como antes da crise.
- Que autoestradas deve o governo encerrar, de maneira a gastar o mínimo com a manutenção, admitindo que o custo da manutenção é proporcional ao comprimento da autoestrada?
- Ou, inversamente, que autoestradas deve manter, de maneira a gastar o menos possível com a manutenção?
- Supomos que a rede é um grafo, onde as cidades são os vértices e as autoestradas são as arestas.

Árvore de cobertura mínima

- Uma **árvore de cobertura** de um grafo é um subgrafo conexo desse grafo, subgrafo esse sem ciclos.
- Um **ciclo** é um caminho onde o último vértice é igual ao primeiro.
- Em teoria de grafos, uma **árvore** é, precisamente, um grafo conexo sem ciclos.
- Num grafo pesado, uma **árvore de cobertura mínima** é uma árvore de cobertura cujo peso é menor ou igual ao peso de qualquer outra árvore de cobertura.
- O peso de uma árvore (ou de um grafo, em geral) é a soma dos pesos das arestas.
- Problema da árvore de cobertura mínima: dado um grafo não dirigido, conectado, pesado, calcular uma **árvore de cobertura mínima**.

Propriedades das árvores de cobertura

- Acrescentar uma aresta a uma árvore de cobertura cria um ciclo.
- Remover uma aresta de uma árvore de cobertura parte-a em duas componentes (que são árvores).



Propriedade de corte

- Num grafo, um **corte** é uma partição do conjunto de vértices em dois conjuntos não vazios.
- Uma **aresta de corte** é uma aresta que liga um vértice de um desses conjuntos a um vértice do outro.
- Propriedade de corte: qualquer que seja o corte, a aresta de corte de peso mínimo pertence à árvore de cobertura mínima.

Algoritmo de Prim

- Constrói a árvore de cobertura acrescentando uma aresta em cada passo.
- Em cada passo, a parte da árvore de cobertura já construída define um corte: de um lado os nós que estão na árvore, do outro os que não estão.
- Em cada passo, acrescenta-se à árvore de cobertura em construção a aresta de corte de peso mínimo (porque essa tem de pertencer à árvore de cobertura mínima).
- Começa-se com um corte em que dum lado está um vértice qualquer e do outro os outros vértices todos; nessa altura a árvore está vazia.
- Havendo N vértices, o algoritmo termina quando a árvore tiver $N-1$ arestas.

Fila com prioridade das arestas de corte

- Quando acrescentamos uma aresta de corte à árvore em construção, acrescentamos um vértice ao conjunto de vértices que pertencem à árvore já construída.
- Porventura partirão desse vértice novas arestas de corte.
- Essas arestas de corte são acrescentadas a uma fila com prioridade, comparando por peso.
- Assim, da próxima vez, escolhemos como aresta de corte a acrescentar a aresta mais prioritária da fila (isto é, a menos pesada).
- Mas atenção: algumas das arestas na fila terão deixado de ser arestas de corte, porque entretanto o corte mudou!

Classe abstrata MinimumSpanningTree

- A classe abstrata especifica a função **mst**, que devolve as arestas da árvore de cobertura mínima:

```
public abstract class MinimumSpanningTree
{
    public final Graph g;

    public MinimumSpanningTree(Graph g)
    {
        assert new ConnectedComponents(g).count == 1;
        this.g = g;
    }

    public abstract Iterable<Edge> mst();
}
```

Fazemos assim, porque haverá vários algoritmos para calcular a árvore de cobertura mínima, ainda que aqui só estudemos um.

Classe Prim

- A classe **Prim** implementa o algoritmo de Prim:

```
class Prim extends MinimumSpanningTree
{
    private Bag<Edge> mst = null;
    private boolean[] marked;
    private PriorityQueue<Edge> queue;

    private final Comparator<Edge> byWeight = Edge::compareByWeight;

    public Prim (Graph g)
    {
        super(g);
        prim();
    }

    private void prim()
    {
        ...
    }
}
```

A árvore é construída no construtor, por meio da função **prim**.

Função prim

- Observe:

```
private void prim()
{
    mst = new Bag<>();
    queue = new PriorityQueue<>(byweight.reversed());
    marked = new boolean[g.nvertices];
    visit(g, 0) ;
    while (!queue.isEmpty())
    {
        Edge e = queue.remove();
        if (marked[e.to])
            continue;
        mst.add(e);
        visit(g, e.to);
    }
}

public Iterable<Edge> mst()
{
    return mst;
}
```

Os vértices marcados são os que já estão na árvore; por isso, as arestas respetivas não são “elegíveis”, isto é, já não são arestas de corte.

```
private void visit(Graph g, int v)
{
    marked[v] = true;
    for (Edge e : g.adj(v))
        if (!marked[e.to])
            queue.insert(e);
}
```

Problema do caminho mais curto

- O problema do caminho mais curto num grafo pesado consiste em descobrir se há um caminho de um vértice dado a outro e, havendo pelo menos um, calcular de entre todos esses o de menor peso.
- Acessoriamente, calculamos o menor caminho do vértice dado a todos os outros vértices do grafo que sejam alcançáveis e o peso de cada um desses caminhos.
- Admitimos nesta fase que os pesos são positivos.

Classe abstrata ShortestPaths

- A classe abstrata especifica funções para realizar os cálculos de que falámos:

```
public abstract class ShortestPaths
{
    public final Graph g;
    public final int start;

    public ShortestPaths(Graph g, int start)
    {
        this.g = g;
        this.start = start;
    }

    public abstract double distanceTo(int v);
    public abstract boolean hasPathTo(int v);
    public abstract Iterable<Edge> pathTo(int v);
    public abstract Iterable<Edge> shortestPathsTree();
}
```

A função **shortestPathsTree** devolve as arestas da árvore que representa os caminhos mais curtos; a função **pathTo** devolve a sequência de arestas do caminho mais curto desde **start** até ao argumento.

Algoritmo de Dijkstra

- Constrói a árvore dos caminhos mais curtos, passo a passo, começando pelo vértice **start**.
- Em cada passo, o algoritmo terá calculado os caminhos mais curtos de **start** até cada um dos vértices presentes na árvore do caminhos mais curtos em construção.
- Terá calculado também, a distância de **start** até cada um desses vértices.
- Em cada passo, o algoritmo acrescenta à árvore uma aresta, “cuidadosamente” selecionada, de entre as arestas de corte.
- A aresta selecionada é a que leva ao vértice mais próximo de **start**, de entre todos os vértices de destino das arestas de corte.
- Desta forma se garante que os vértices são acrescentados à árvore pela ordem da sua distância a **start**; logo, o caminho registado na árvore é o caminho mais curto.

Classe Dijkstra

- A classe Dijkstra implementa o algoritmo de Dijkstra:

```
class Dijkstra extends ShortestPaths
{
    private Edge[] edgeTo;
    private double[] distanceTo;
    private boolean[] marked;
    private Bag<Edge> spt;

    public Dijkstra(Graph g, int start)
    {
        super(g, start);
        marked = new boolean[g.nVertices];
        edgeTo = new Edge[g.nVertices];
        spt = new Bag<>();
        dijkstra(start);
    }

    private void dijkstra(int v)
    {
        ...
    }
}
```

Os cálculos propriamente ditos são feitos na função **dijkstra**.

Função dijkstra

- Observe:

```
private void dijkstra(int v)
{
    marked[v] = true;
    edgeTo = new Edge[g.nVertices];
    distanceTo = new double[g.nVertices];
    Arrays.fill(distanceTo, Double.POSITIVE_INFINITY);
    distanceTo[v] = 0;
    while (true)
    {
        Edge e = dijkstraEdge();
        if (e == null)
            break;
        spt.add(e);
        distanceTo[e.to] = distanceTo[e.from] + e.weight;
        marked[e.to] = true;
        edgeTo[e.to] = e;
    }
}
```

Os vértices marcados são os que já estão na árvore dos caminhos mais curtos, em construção.

A função **dijkstraEdge** seleciona a próxima aresta a acrescentar à árvore; se não houver, o algoritmo termina.

Função dijkstraEdge

- Nesta implementação, não muito inteligente, o algoritmo enumera todas as arestas para descobrir quais são arestas de corte.
- De entre essas, escolhe a que dá a distância mínima:

```
private Edge dijkstraEdge()
{
    Edge result = null;
    double min = Double.POSITIVE_INFINITY;
    for (Edge e : g.edges())
        if (marked[e.from] && !marked[e.to])
            if (min > distanceTo[e.from] + e.weight)
            {
                min = distanceTo[e.from] + e.weight;
                result = e;
            }
    return result;
}
```

Classe Dijkstra, restantes funções

- São análogas a outras que já vimos noutras classes:

```
public double distanceTo(int v)
{
    return distanceTo[v];
}

public boolean hasPathTo(int v)
{
    return marked[v];
}

public Iterable<Edge> pathTo(int v)
{
    Stack<Edge> result = null;
    if (marked[v])
    {
        result = new Stack<>();
        for (int x = v; x != start; x = edgeTo[x].from)
            result.push(edgeTo[x]);
    }
    return result;
}

public Iterable<Edge> shortestPathsTree()
{
    return spt;
}
```

Análise da complexidade

- O tempo de execução da função **prim** é dominado pela fila com prioridade.
- No pior dos casos todas as arestas entram na fila e todas saem.
- Logo, havendo E arestas, a complexidade do algoritmo de Prim é proporcional a $E \log E$.
- O ciclo while da função **dijkstra** dá um passo por cada aresta.
- Em cada passo, enumera todas as arestas, para calcular a distância mínima.
- Logo, havendo E arestas, a complexidade do algoritmo de Dijkstra, na versão que estudámos é E^2 .

Algoritmo de Dijkstra E log V

- No algoritmo de Dijkstra, não pudemos usar filas com prioridade para as arestas de corte, porque a comparação é feita com as distâncias ao ponto de destino, as quais variam de passo para passo, e não com os pesos, que são fixos.
- Por isso, em cada passo, tivemos de inspecionar todas as arestas, ainda que descartando as que não eram arestas de corte.
- No entanto, usando uma **fila com prioridades indexada** para as arestas de corte consegue-se calcular a próxima aresta com complexidade $\text{Log } V$, sendo V o número de vértices.
- Assim, a complexidade do algoritmo de Dijkstra nessa variante é $E \text{ Log } V$.
- As filas com prioridade indexadas são uma variante das filas com prioridade onde é possível fazer variar a prioridade de um elemento da fila, acessível através de uma chave.