

1

**Estruturas de Dados**  
tipo de dados / estrutura de dados

**Tipo de dados**

- Especifica como é que uma dada sequência de bits deve ser interpretada
- Determina o espaco (em memória) que deve ser reservado para armazenar o(s) objecto(s) declarado(s)

**Exemplo:**

Type nome = array[1..60] of char;

Ocupa um espaço de 60 bytes

A informação armazenada deve ser interpretada como uma cadeia de caracteres

© DEI Carlos Bento      ALGORITMOS E ESTRUTURAS DE DADOS      03 -

2

# Estruturas de Dados

tipo de dados / estrutura de dados

## Tipo de dados

Hardware  
(assembly)

- vírgula flutuante
- inteiros
- cadeias de bits
- bits

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

3

# Estruturas de Dados

tipo de dados / estrutura de dados

## Tipo de dados

Linguagens de  
alto nível  
(Pascal, C,...)

- records
- arrays
- strings
- booleanos

Hardware  
(assembly)

- vírgula flutuante
- inteiros
- cadeias de bits
- bits

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

4

2

# Estruturas de Dados

tipo de dados / estrutura de dados

## Tipo de dados

Estruturas de dados mais complexas (definidas a partir dos dados e das operações existentes nas linguagens de alto nível)
Linguagens de alto nível (Pascal, C,...)
Hardware (assembly)

- árvores
- filas de espera
- pilhas
- listas
  
- records
- arrays
- strings
- booleanos
  
- vírgula flutuante
- inteiros
- cadeias de bits
- bits

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

5

# Estruturas de Dados

tipo de dados / estrutura de dados

## Tipo de dados

Modelos de dados de mais alto nível (SGBD, ...)
Estruturas de dados mais complexas (definidas a partir dos dados e das operações existentes nas linguagens de alto nível)
Linguagens de alto nível (Pascal, C,...)
Hardware (assembly)

- modelo relacional
- UML
  
- árvores
- filas de espera
- pilhas
- listas
- grafos
  
- records
- arrays
- strings
- booleanos
  
- vírgula flutuante
- inteiros
- cadeias de bits
- bits

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

6

# [Estruturas de Dados]

tipo de dados / estrutura de dados

## Estrutura de dados

Uma estrutura de dados compreende:

- uma **representação** dos dados (definição do tipo)
- um conjunto de **operações** sobre esses dados

# [Estruturas de Dados]

tipo de dados / estrutura de dados

## Porque as estruturas de dados são importantes:

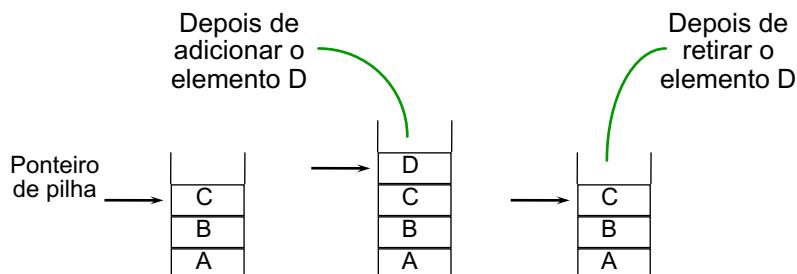
- uma **estrutura de dados adequada** permite otimizar os acessos e manipulações dos dados
- as operações necessárias sobre uma estrutura de dados **dependem** **fortemente das tarefas** da minha aplicação
- em geral uma estrutura de dados **mais especializada** (focada nas operações que necessito) **será mais eficiente**
- em muitos casos justifica-se **desenvolver estruturas de dados especificamente desenhadas** para as tarefas que tenho em mãos

# Pilhas

conceitos

Colecção ordenada de elementos em que estes são adicionados ou retirados a partir de uma das suas extremidades designada por **topo da pilha**.

Limita o acesso ao elemento mais recentemente inserido na pilha.



© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

9

# Pilhas

conceitos

## Operações sobre pilhas

`void push( x )` ➔ inserir x

`Object pop( )` ➔ remover e devolver o elemento mais recentemente inserido

`void makeEmpty( )` ➔ remover todos os elementos

`boolean isEmpty( )` ➔ devolver true se vazia; senão false

`boolean isFull()` ➔ devolve true se a cheia; senão false

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

10

# Pilhas

## conceitos

**INTERFACE**

```

package DataStructures;
import Exceptions.*;
// Stack interface
// ****ERRORS*****
// pop on empty stack
/**
 * Protocol for stacks.
 * @author Mark Weiss
 */
public interface Stack
{
    Object pop() throws Underflow;
    void push( Object x );
    void makeEmpty();
    boolean isEmpty();
    boolean isFull();
}

```

**APLICAÇÃO**

```

import DataStructures.*;
import Exceptions.*;
// Simple test program for stacks
public final class TestStack
{
    public static void main( String [] args )
    {
        Stack s = new StackAr( );
        for( int i = 0; i < 5; i++ )
            s.push( new Integer( i ) );
        System.out.print( "Contents:" );
        try
        {
            for( ; ; )
                System.out.print( " " + s.pop() );
        }
        catch( Underflow e ) {}
        System.out.println( );
    }
}

```

© DEI Carlos Bento      ALGORITMOS E ESTRUTURAS DE DADOS      03 -

11

# Pilhas

## aplicações

Apl – Exemplo #1  
Análise sintactica de programas - verificação do balanceamento de símbolos  
ex.: () {} [] /\* \*/ //

```

delimiterMatching(file)
    read character ch from file;
    while not end of file
        if ch is '(', '[', or '{'
            push(ch);
        else if ch is '/'
            read the next character;
            if this character is '*'
                push(ch);
            else ch = the character read in;
            continue; // go to the beginning of the loop;
        else if ch is ')', ']', or '}'
            if ch and popped off delimiter do not match
                failure;
            else if ch is '*'
                read the next character;
                if this character is '/' and popped off delimiter is not '/'
                    failure;
                else ch = the character read in;
                push back the popped off delimiter;
                continue;
            // else ignore other characters;
            read next character ch from file;
            if stack is empty
                success;
            else failure;

```

© DEI Carlos Bento      ALGORITMOS E ESTRUTURAS DE DADOS      03 -

12

# Pilhas

[
]

**aplicações**

Apl – Exemplo #1

Análise sintactica de programas - verificação do balanceamento de símbolos  
ex.: () {} []

Stack	Nonblank Character Read	Input Left
empty		$s = t[5] + u / (v * (w + y));$
empty	s	$= t[5] + u / (v * (w + y));$
empty	=	$t[5] + u / (v * (w + y));$
empty	t	$[5] + u / (v * (w + y));$
[]	[	$5] + u / (v * (w + y));$
[]	5	$] + u / (v * (w + y));$
empty	]	$+ u / (v * (w + y));$
empty	+	$u / (v * (w + y));$
empty	u	$/ (v * (w + y));$
empty	/	$(v * (w + y));$
[]	(	$v * (w + y);$
[]	v	$* (w + y);$
[]	*	$(w + y);$
[]	(	$w + y);$
[]	w	$+y);$
[]	+	$y);$
[]	y	$);$
[]	)	$;$
empty	)	
empty	;	

© DEI Carlos Bento
ALGORITMOS E ESTRUTURAS DE DADOS
03 -

13

# Pilhas

[
]

**aplicações**

Apl – Exemplo #2 Adição de números inteiros muito longos

Data Structures and Algorithms in JAVA, Adam Drozdak

© DEI Carlos Bento
ALGORITMOS E ESTRUTURAS DE DADOS
03 -

14

# Pilhas

## aplicações

Apl – Exemplo #2

Adição de números inteiros muito longos

```

AddingLargeNumbers()
read the numerals of the first number and store the numbers corresponding to
them on one stack;
read the numerals of the second number and store the numbers corresponding
to them on another stack;
result = 0;
while at least one stack is not empty
    pop a number from each nonempty stack and add them to result;
    push the unit part on the result stack;
    store carry in result;
    push carry on the result stack if it is not zero;
    pop numbers from the result stack and display them;

```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

15

# Pilhas

## aplicações

Apl – Exemplo #4

Implementação dos mecanismos de chamada de métodos em muitas linguagens de programação.

Apl – Exemplo #5

Avaliação de expressões em linguagens de programação - o valor de expressões intermédias guardado numa pilha.

© DEI Carlos Bento

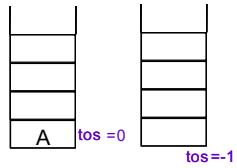
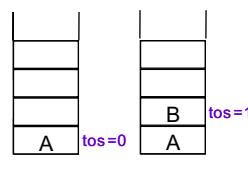
ALGORITMOS E ESTRUTURAS DE DADOS

03 -

16

# Pilhas

## implementação

`pop()``push(B)`

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

17

# Pilhas

## implementação

```

public boolean isEmpty( )
{
    return topOfStack == -1;
}

public void makeEmpty( )
{
    topOfStack = -1;
}

public Object pop( ) throws Underflow
{
    if( isEmpty( ) )
        throw new Underflow( "Stack pop" );
    return theArray[ topOfStack-- ];
}

public void push( Object x )
{
    if( topOfStack + 1 == theArray.length )
        doubleArray();
    theArray[ ++topOfStack ] = x;
}

private void doubleArray()
{
    Object [ ] newArray;
    newArray = new Object[ theArray.length * 2 ];
    for( int i = 0; i < theArray.length; i++ )
        newArray[ i ] = theArray[ i ];
    theArray = newArray;
}

private Object [ ] theArray;
private int topOfStack;
static final int DEFAULT_CAPACITY = 10;
}

```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

18

# Filas de Espera

conceitos

Sequência ordenada de elementos, podendo ser adicionados novos elementos numa das extremidades (parte de trás ou **cauda**) e retirados na outra extremidade (parte da frente ou **cabeça**).

- Só os elementos que estão à frente é que podem ser retirados
- Só se pode adicionar novos elementos na parte de trás
- A fila de espera comporta-se como um FIFO ("First-IN, First-Out")

© DEI Carlos Bento

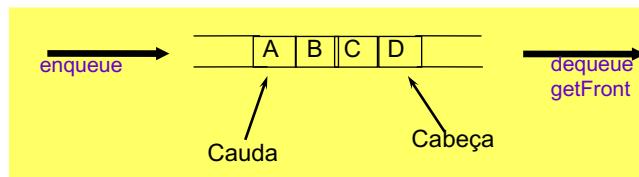
ALGORITMOS E ESTRUTURAS DE DADOS

03 -

19

# Filas de Espera

conceitos



```

void enqueue( Object x )           --> insere x
Object getFront( )                --> devolve o primeiro elemento inserido
Object dequeue( )                 --> devolve e remove o primeiro elemento inserido
boolean isEmpty( )                --> devolver true se vazia; senão false
void makeEmpty( )                 --> remove todos os elementos
  
```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

20

## Filas de Espera

tipo abstracto de dados

**INTERFACE**

```

package DataStructures;
import Exceptions.*;

// ****ERRORS*****
// getFront or dequeue on empty queue

// @author Mark Allen Weiss

public interface Queue
{
    boolean isEmpty();
    Object getFront( ) throws Underflow;
    Object dequeue( ) throws Underflow;
    void enqueue( Object X );
    void makeEmpty();
}

```

**APLICAÇÃO**

```

import DataStructures.*;
import Exceptions.*;

// Simple test program for queues

public final class TestQueue
{
    public static void main( String [ ] args )
    {
        Queue q = new QueueAr( );

        for( int i = 0; i < 5; i++ )
            q.enqueue( new Integer( i ) );

        System.out.print( "Contents:" );
        try
        {
            for( ; ; )
                System.out.print( " " + q.dequeue( ) );
        }
        catch( Underflow e ) { }

        System.out.println( );
    }
}

```

© DEI Carlos Bento      ALGORITMOS E ESTRUTURAS DE DADOS      03 -

21

## Filas de Espera

aplicações

As filas de espera utilizam-se vulgarmente em três tipos de situações:

- Quando é necessário efectuar um **tratamento sequencial** de eventos ou de dados (o primeiro a chegar é o primeiro a ser tratado: **FIFO**);
- Como **buffer** (amortecedor) para comunicação entre processos assíncronos (permite atenuar as diferenças na cadências de produção/consumo da informação):
  - p. ex. entre um computador e uma impressora
- Na **simulação** de mecanismos de espera em processos envolvendo o acesso de conjuntos de consumidores a recursos limitados:
  - p. ex. filas esperas nas caixas de um hipermercado

© DEI Carlos Bento      ALGORITMOS E ESTRUTURAS DE DADOS      03 -

22

# Filas de Espera

## implementação

**Deslocamento da fila ao longo da memória -** Caso de uma fila de espera construída sobre um array de cinco posições

1) Início

```
4
3
2
1
0
    front =-1
    rear =-1
```

2) Inseriu A

```
4
3
2
1
0
    A
    front = rear = 0
```

3) Inseriu B e C

```
4
3
2
1
0
    C   rear = 2
    B   front = 0
```

4) Removeu A

```
4
3
2
1
0
    C   rear = 2
    B   front = 1
```

5) Removeu B

```
4
3
2
1
0
    C   front = rear = 2
    1
```

6) Inseriu D e E

```
4
3
2
1
0
    E   rear = 4
    D   front = 2
    C   front = 1
    1
```

Fila de espera cheia !!

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

23

# Filas de Espera

## implementação

**Filas de espera sobre um array circular - Solução**

1) Após várias operações

```
4   E   rear = 4
3   D
2   C   front = 2
1
0
```

2) Inseriu F

```
4   E
3   D
2   C   front = 2
1
0   F   rear = 0
```

3) Removeu C e D

```
4   E   front = 4
3
2
1
0   F   rear = 0
```

4) Inseriu G

```
4   E   front = 4
3
2
1   G   rear = 1
0   F
```

5) Removeu E

```
4
3
2
1   G   rear = 1
0   F   front = 0
```

6) Removeu F

```
4
3
2
1   G   rear = front = 1
0
```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

24

# Filas de Espera

## implementação

Implementação (como apresentada no livro do Adam Drozdek)

```

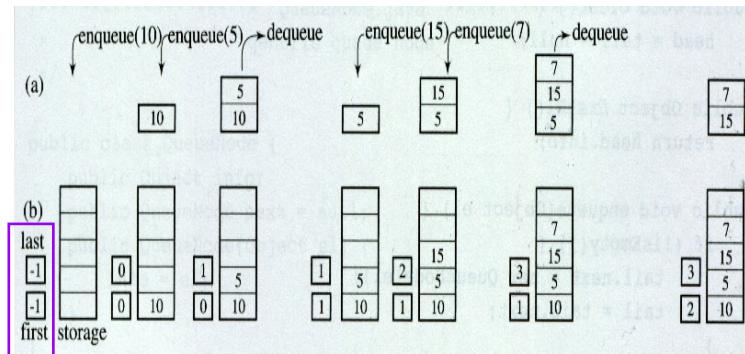
void clear( ) ➔ remove todos os elementos
void enqueue( Object x ) ➔ insere x
Object dequeue( ) ➔ devolve e remove o último elemento inserido
Object firstEl( ) ➔ devolve o último elemento inserido
boolean isEmpty( ) ➔ devolver true se vazia; senão false
boolean isFull( ) ➔ devolver true se cheia; senão false

```

# Filas de Espera

## implementação

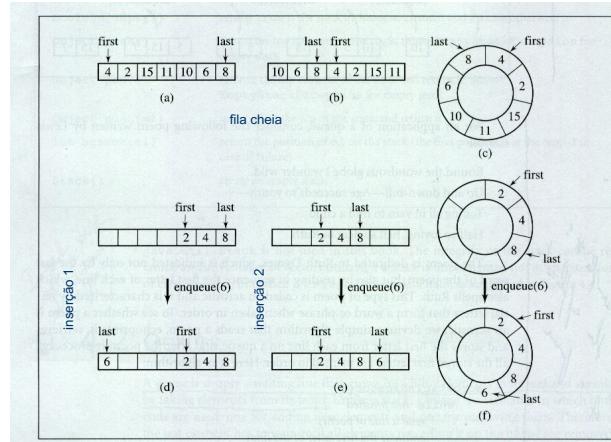
Implementação (como apresentada no livro do Adam Drozdek)



# Filas de Espera

## implementação

Condições de fila de espera cheia



© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

27

# Filas de Espera

## implementação

Implementação (como apresentada no livro do Adam Drozdek)

```
// Adam Drozdek - queue implemented as an array
public class ArrayQueue {
    private int first, last, size;
    private Object[] storage;
    public ArrayQueue() {
        this(100);
    }
    public ArrayQueue(int n) {
        size = n;
        storage = new Object[size];
        first = last = -1;
    }
    public boolean isFull() {
        return first == 0 && last == size-1 || first == last + 1;
    }
    public boolean isEmpty() {
        return first == -1;
    }
}
```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

28

14

# Filas de Espera

## implementação

Implementação (como apresentada no livro do Adam Drozdek)

```
public void enqueue(Object el) {
    if (last == size-1 || last == -1) {
        storage[0] = el;
        last = 0;
        if (first == -1)
            first = 0;
    } else storage[++last] = el;
}
public Object dequeue() {
    Object tmp = storage[first];
    if (first == last)
        last = first = -1;
    else if (first == size-1)
        first = 0;
    else first++;
    return tmp;
}
public void printAll() {
    for (int i = 0; i < size; i++)
        System.out.print(storage[i] + " ");
}
```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

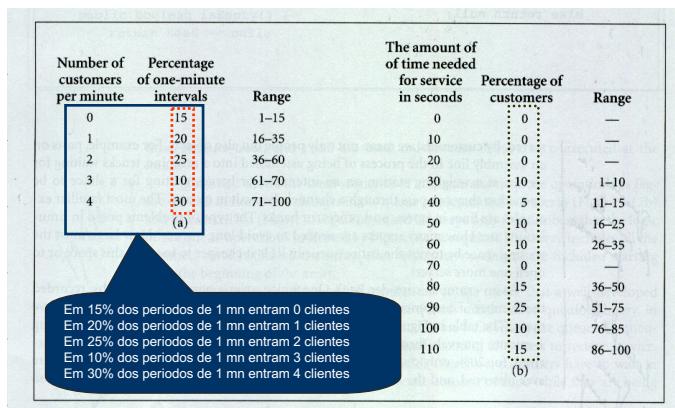
03 -

29

# Filas de Espera

## aplicações

Aplicação # 1 – Simulação do balcão de atendimento de um banco.  
Tendo dados sobre a cadência de chegada de clientes ao banco e os tempos de atendimento determinar quantos caixas são necessários para ter um atendimento com tempos de espera aceitáveis (do livro do Adam Drozdek)



© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

30

# Filas de Espera

## aplicações

Ap1 – Exemplo # 1 – Simulação do balcão de atendimento de um banco.

```
import java.util.*;

class BankSimulation {
    static Random rd = new Random();

    static int Option(int percents[]) {
        int i = 0, perc, choice = Math.abs(rd.nextInt()) % 100 + 1;
        for (perc = percents[0]; perc < choice; perc += percents[i+1], i++);
        // System.out.println(" option2 "+choice+" "+ i +" "+perc+" "+percents[i]+" ");
        return i;
    }

    public static void main(String args[]){
        int[] arrivals = {15,20,25,10,30};
        int[] service = {10,0,0,10,5,10,10,0,15,25,10,15};
        // NUMERO DE CAIXAS
        int[] clerks = {0,0,0,0};
        int clerksSize = clerks.length;
        int customers, t, i, numOfMinutes = 100, x;
        double thereisLine = 0.0;
        Queue simulQ = new Queue();
    }
}
```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

31

# Filas de Espera

## aplicações

Ap1 – Exemplo # 1 – Simulação do balcão de atendimento de um banco.

```
for (t = 1; t <= numOfMinutes; t++) {
    System.out.print(" t = " + t);
    for (i = 0; i < clerksSize; i++) // after each minute subtract
        if (clerks[i] < 60)           // at most 60 seconds from time
            clerks[i] = 0;           // after the 60 sec the clerk does not have work to do
        else clerks[i] -= 60;        // after the 60 sec the clerk has x-60 sec work to do

    customers = Option(arrivals);      // number of costumers arriving this minute
    for (i = 0; i < customers; i++) {   // enqueue all new customers
        x = Option(service)*10;        // time need for the costumer attendance (0-110 sec)
        simulQ.enqueue(new Integer(x)); // they require
    }
}
```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

32

# Filas de Espera

aplicações

Ap1 – Exemplo # 1 – Simulação do balcão de atendimento de um banco.

```
// dequeue customers when clerks are available:  
for (i = 0; i < clerksSize && !simulQ.isEmpty(); )  
    if (clerks[i] < 60) {  
        x = ((Integer) simulQ.dequeue()).intValue();  
        clerks[i] += x; // to a clerk if service time is still below 60 sec  
    }  
    else i++; // only increment clerk number if the current one is above 60 sec loaded  
    if (!simulQ.isEmpty()) {  
        therelsLine++;  
    }  
    else System.out.println(" wait = 0");  
}  
System.out.println("\nFor " + clerksSize + " clerks, there was a line "  
+ therelsLine/numOfMinutes*100.0 + "% of the time;\n");  
}
```

© DEI Carlos Bento

ALGORITMOS E ESTRUTURAS DE DADOS

03 -

33

# Análise de Complexidade

(leituras)

Sedgewick, Cap.4, pp127-153

Sedgewick, Cap.3, pp69-126

- Compreender o conceito de Tipo Abstracto de Dados (TAD)
- Conhecer e compreender as seguintes estruturas de dados:
- matrizes;
- pilhas; filas
- Saber como desenhar um programa simples de simulação de filas de espera

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

34

# Análise de Complexidade

... bom trabalho, FIM!



© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

35

# Algoritmos e Estruturas de Dados

listas ligadas

■ 2020-2021

■ Carlos Lisboa Bento

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

36

# Listas Ligadas

## roadmap

- listas simplesmente ligadas
  - conceitos
  - vantagens e desvantagens
  - implementação
- listas duplamente ligadas
  - conceitos
  - vantagens e desvantagens
  - implementação
- listas circulares
  - conceitos
  - vantagens e desvantagens
  - implementação
- listas auto-organizadas
  - conceitos
  - exemplo
  - vantagens e desvantagens
- listas ligadas e tabelas esparsas
  - exemplo de uma aplicação das listas ligadas
  - vantagens e desvantagens
- listas de saltos
  - conceitos
  - vantagens e desvantagens
  - implementação

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

37

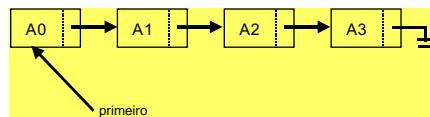
# Listas Ligadas

## conceitos

Estrutura em que cada elemento contém o endereço do elemento seguinte (a sequência é estabelecida explicitamente).

Cada elemento de uma lista contém dois campos:

- Campo da informação;
- Campo do endereço do elemento seguinte.



A lista é acedida através de um ponteiro externo que aponta para o primeiro nó.

O fim da lista é indicado pelo ponteiro para elemento seguinte a NULL

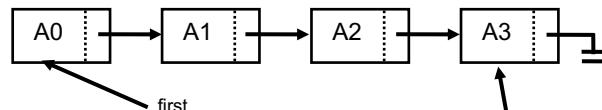
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

38

19

# Listas Ligadas

## conceitos



```

Class ListNode
{
    // construtores

    ListNode()
    { this( null, null ); }

    ListNode( Object theElement )
    { this( theElement, null ); }

    ListNode( Object theElement, ListNode n )
    { element = theElement; next = n; }

    Object element;
    ListNode next;
}

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

```

Inserção a seguir  
ao último elemento  
!!

39

# Listas Ligadas

## conceitos

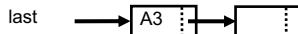
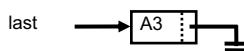
/\* inserção de um elemento x a seguir  
a ultimo \*/

last.next = new ListNode(); // 1

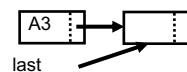
last = last.next; // 2

last.data = x; // 3

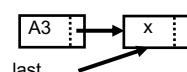
last.next = null; // 4



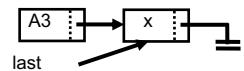
1



2



3



4

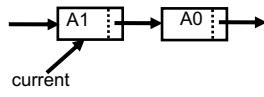
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

40

# Listas Ligadas

## implementação

Duas operações básicas numa lista - inserção a partir de current



```
tmp = new ListNode();
tmp.element = x;
tmp.next = current.next;
current.next = tmp;

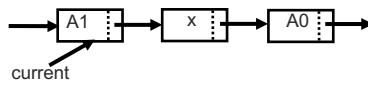
// MELHOR

tmp = new ListNode( x, current.next );
current.next = tmp;

// MELHOR AINDA

current.next = new ListNode( x, current.next );
```

eliminação a partir de current



current.next = current.next.next;

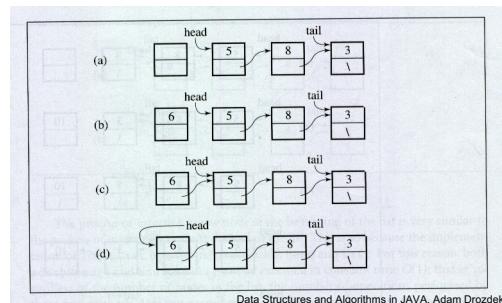
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

41

# Listas Ligadas

## vantagens e desvantagens

Inserção na cabeça de uma lista ligada ☺



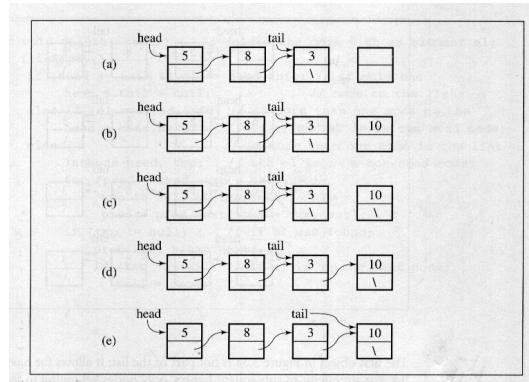
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

42

# Listas Ligadas

## vantagens e desvantagens

Inserção na cauda de uma lista ligada ☺



Data Structures and Algorithms in JAVA, Adam Drozdek

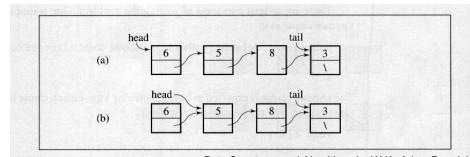
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

43

# Listas Ligadas

## vantagens e desvantagens

Eliminação na cabeça de uma lista ligada ☺



Data Structures and Algorithms in JAVA, Adam Drozdek

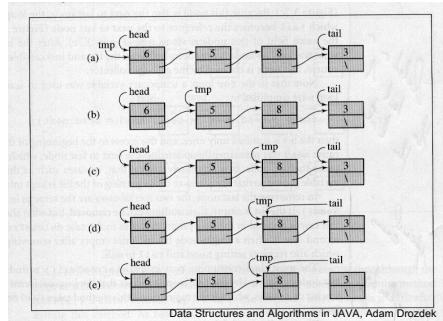
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

44

# Listas Ligadas

## vantagens e desvantagens

Eliminação na cauda de uma lista ligada ☹



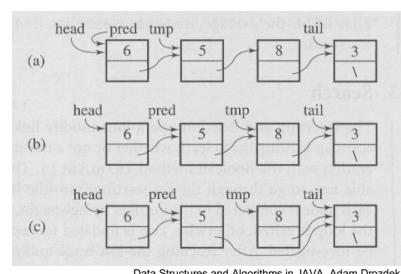
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

45

# Listas Ligadas

## vantagens e desvantagens

Eliminação num ponto intermédio de uma lista ligada ☹



© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

46

# Listas Ligadas

## vantagens e desvantagens

### Vantagens

- A inserção de um elemento no meio de uma lista ligada não implica mover todos os elementos que se seguem.
- É ocupada apenas a memória necessária em cada momento, não havendo limitações prévias às dimensões destas estruturas (excepto a capacidade de memória do computador utilizado). Esta vantagem não é substancial em JAVA, embora o seja noutras linguagens.

### Desvantagens

- Acesso sequencial.
- Acesso ou eliminação de um nó intermédio.
- Eliminação de um nó no fim da lista.
- Na prática a movimentação de elementos numa lista ligada é dispendiosa.

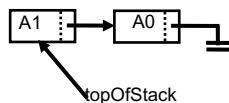
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

47

# Listas Ligadas

## implementação

### Implementação de uma PILHA sobre uma lista ligada



```
package DataStructures;
import Exceptions.*;

// *****PUBLIC OPERATIONS*****
// void push( x )      --> Insert x
// Object pop( )        --> Return and
//                           Remove most recently
//                           inserted item
// boolean isEmpty( )   --> Return true if empty;
//                           else false
// void makeEmpty()     --> Remove all items
// *****ERRORS*****  
// pop on empty stack
```

```
/*
 * List-based implementation of the stack.
 * @author Mark Allen Weiss
 * @changes Francisco Pereira
 */
public class StackLi implements Stack
{
    public StackLi()
    { topOfStack = null; }

    public boolean isEmpty( )
    { return topOfStack == null; }

    public void makeEmpty( )
    { topOfStack = null; }
```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

48

# Listas Ligadas

## implementação

Implementação de uma PILHA sobre uma lista ligada (cont.)

```

public Object pop( ) throws Underflow
{
    if( isEmpty( ) )
        throw new Underflow( "Stack topAndPop" );

    Object topItem = topOfStack.element();
    topOfStack = topOfStack.next();
    return topItem;
}

public void push( Object x )
{
    topOfStack = new ListNode( x, topOfStack );
}

private ListNode topOfStack;
}

```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

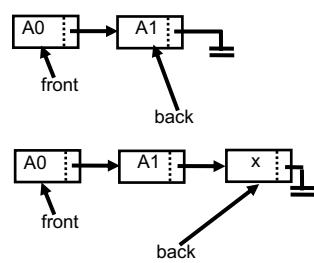
49

# Listas Ligadas

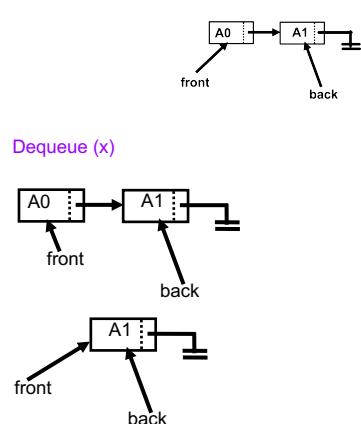
## implementação

Implementação de uma FILA DE ESPERA sobre uma lista ligada

Enqueue (x)



Dequeue (x)



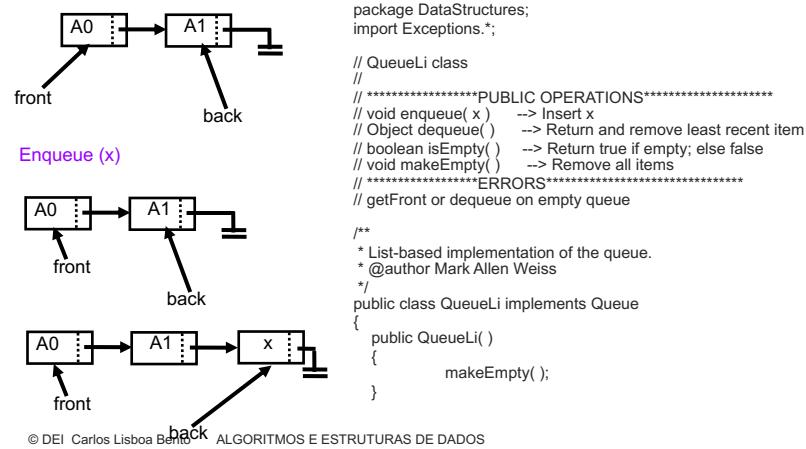
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

50

# Listas Ligadas

## implementação

Implementação de uma FILA DE ESPERA sobre uma lista ligada



51

# Listas Ligadas

## implementação

Implementação de uma FILA DE ESPERA sobre uma lista ligada (cont.)

```

public boolean isEmpty()
{
    return front == null;
}

public void makeEmpty()
{
    front = null;
    back = null;
}

public Object dequeue() throws Underflow
{
    if( isEmpty() )
        throw new Underflow( "QueueLi dequeue" );

    Object returnValue = front.element;
    front = front.next;
    return returnValue;
}

public void enqueue( Object x )
{
    if( isEmpty() ) // Make queue of one element
        back = front = new ListNode( x );
    else // Regular case
        back = back.next = new ListNode( x );
}

private ListNode front;
private ListNode back;
}

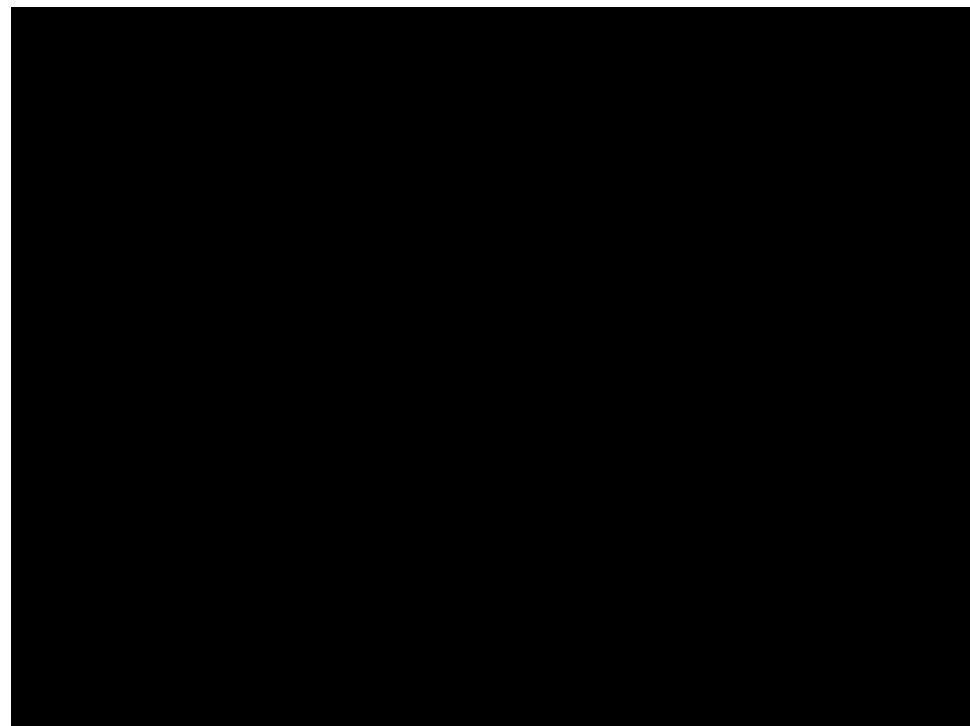
```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

52

## [ Array or ArrayList ?? ]

Array vs ArrayList (in JAVA)



# Listas Ligadas

## implementação

```
*****
 * IntSLList.java
 * singly-linked list class to store integers
 ****

public class IntSLList {
    private IntNode head, tail;
    public IntSLList() {
        head = tail = null;
    }
    public boolean isEmpty() {
        return head == null;
    }
    public void addToHead(int el) {
        head = new IntNode(el,head);
        if (tail == null)
            tail = head;
    }
    public void addToTail(int el) {
        if (!isEmpty()) {
            tail.next = new IntNode(el);
            tail = tail.next;
        }
        else head = tail = new IntNode(el);
    }
}
```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

57

# Listas Ligadas

## implementação

```
public int deleteFromHead() { // delete the head and return its info;
    int el = head.info;
    if (head == tail) // if only one node on the list;
        head = tail = null;
    else head = head.next;
    return el;
}

public int deleteFromTail() { // delete the tail and return its info;
    int el = tail.info;
    if (head == tail) // if only one node in the list;
        head = tail = null;
    else { // if more than one node in the list,
        IntNode tmp; // find the predecessor of tail;
        for (tmp = head; tmp.next != tail; tmp = tmp.next);
        tail = tmp; // the predecessor of tail becomes tail;
        tail.next = null;
    }
    return el;
}
```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

?

58

# Listas Ligadas

## implementação

?

```

public void printAll() {
    for (IntNode tmp = head; tmp != null; tmp = tmp.next)
        System.out.print(tmp.info + " ");
}

public boolean isInList(int el) {
    IntNode tmp;
    for (tmp = head; tmp != null && tmp.info != el; tmp = tmp.next);
    return tmp != null;
}

```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

59

# Listas Ligadas

## implementação

?

```

public void delete(int el) { // delete the node with an element el;
    if (isEmpty())
        if (head == tail && el == head.info) // if only one
            head = tail = null; // node on the list;
        else if (el == head.info) // if more than one node on the list;
            head = head.next; // and el is in the head node;
        else { // if more than one node in the list
            IntNode pred, tmp; // and el is in a non-head node;
            for (pred = head, tmp = head.next;
                 tmp != null && tmp.info != el;
                 pred = pred.next, tmp = tmp.next);
            if (tmp != null) { // if el was found;
                pred.next = tmp.next;
                if (tmp == tail) // if el is in the last node;
                    tail = pred;
            }
        }
    }
}

```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

60

# Listas Ligadas

## complexidade

Inserção

CABEÇA

O(1)

CAUDA

O(1)

PONTO INTERMÉDIO <sup>(1)</sup>

O(n)

Eliminação

CABEÇA

O(1)

CAUDA

O(n)

PONTO INTERMÉDIO <sup>(1)</sup>

O(n)

<sup>(1)</sup>

$$\frac{0+1+\dots+(n-1)}{n} = \frac{\frac{(n-1)n}{2}}{n} = \frac{n-1}{2}$$

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

61

# Listas Duplamente Ligadas

## conceitos

### Problemas das listas lineares

- Dado um ponteiro para um nó, não se pode ter acesso aos nós que o precedem;
- Quando se atravessa a lista é necessário preservar sempre o ponteiro para o início da lista  
Uma possível solução (que também tem inconvenientes) é ter uma lista circular

As listas duplamente ligadas permitem percursos em ambos os sentidos

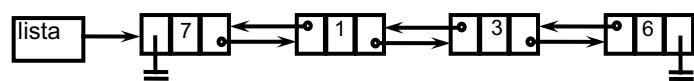
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

62

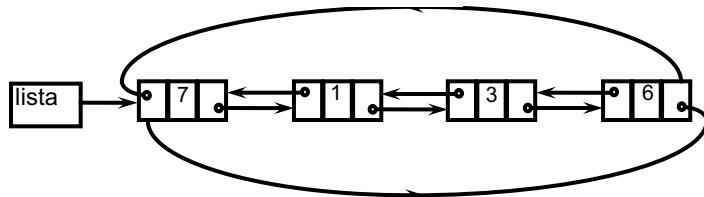
# Listas Duplamente Ligadas

conceitos

Simples



Circular



Cada nó destas listas tem, pelo menos, três campos:

- campo (ou campos) para a **informação**;
- **ponteiro** para o elemento da **esquerda**;
- **ponteiro para elemento da direita**.

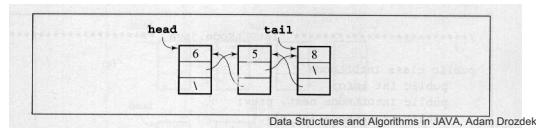
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

63

# Listas Duplamente Ligadas

vantagens e desvantagens

Uma lista duplamente ligada de inteiros



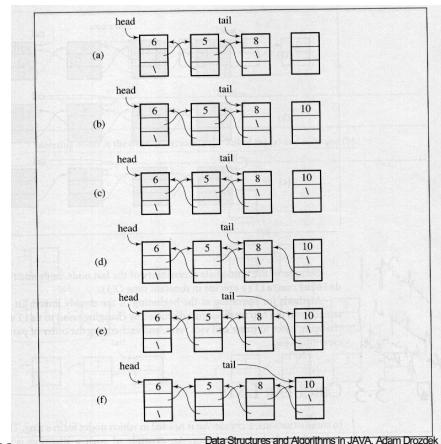
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

64

# Listas Duplamente Ligadas

vantagens e desvantagens

Inserção na cauda de uma lista duplamente ligada



© DEI Carlos Lisboa Bento ALGORITMOS

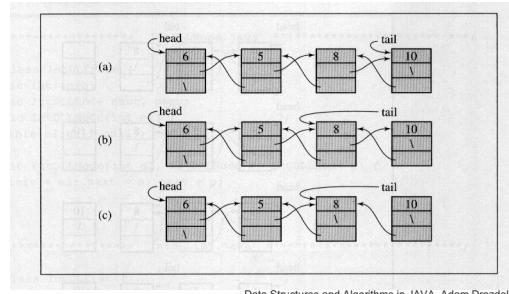
Data Structures and Algorithms in JAVA, Adam Drozdek

65

# Listas Duplamente Ligadas

vantagens e desvantagens

Eliminação na cauda de uma lista duplamente ligada



© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

66

# >Listas Duplamente Ligadas

implementação LDLs ordenadas

```
***** IntDLLNode.java *****
```

```
class IntDLLNode {
    int info;
    IntDLLNode next = null, prev = null;

    IntDLLNode() {}

    IntDLLNode(int el) {
        info = el;
    }

    IntDLLNode(int el, IntDLLNode n, IntDLLNode p) {
        info = el; next = n; prev = p;
    }
}
```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

67

# Listas Duplamente Ligadas

implementação LDLs ordenadas

```
***** IntDLLList.java *****
```

```
public class IntDLLList {
    private IntDLLNode head, tail;
    public IntDLLList() {
        head = tail = null;
    }
    public boolean isEmpty() {
        return head == null;
    }
    public void setToNull() {
        head = tail = null;
    }
    public int firstEl() {
        if (isEmpty())
            return head.info;
        else return 0;
    }
}
```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

68

# Listas Duplamente Ligadas

implementação LDLs ordenadas

```

public void addToDLLListHead(int el) {
    if (!isEmpty()) {
        head = new IntDLLNode(el,head,null);
        head.next.prev = head;
    }
    else head = tail = new IntDLLNode(el);
}
public void addToDLLListTail(int el) {
    if (!isEmpty()) {
        tail = new IntDLLNode(el,null,tail);
        tail.prev.next = tail;
    }
    else head = tail = new IntDLLNode(el);
}

```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

69

# Listas Duplamente Ligadas

implementação LDLs ordenadas

```

public int deleteFromDLLListHead() {
    if (!isEmpty()) { // if at least one node in the list;
        int el = head.info;
        if (head == tail) // if only one node in the list;
            head = tail = null;
        else { // if more than one node in the list;
            head = head.next;
            head.prev = null;
        }
        return el;
    }
    else return 0;
}
public int deleteFromDLLListTail() {
    if (!isEmpty()) {
        int el = tail.info;
        if (head == tail) // if only one node on the list;
            head = tail = null;
        else { // if more than one node in the list;
            tail = tail.prev;
            tail.next = null;
        }
        return el;
    }
    else return 0;
}

```

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

70

# Listas Duplamente Ligadas

implementação LDLs ordenadas

```
public void printAll() { //OutputStream Out {
    for (IntDLLNode tmp = head; tmp != null; tmp = tmp.next)
        System.out.print(tmp.info + " ");
}

public int find(int el) {
    IntDLLNode tmp;
    for (tmp = head; tmp != null && tmp.info != el; tmp = tmp.next);
    if (tmp == null)
        return 0;
    else return tmp.info;
}
```

Inserção ou eliminação no ponto intermédio de uma lista duplamente ligada???

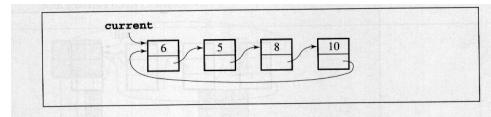
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

71

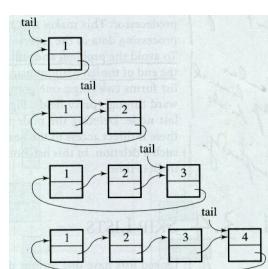
# Listas Circulares

conceitos

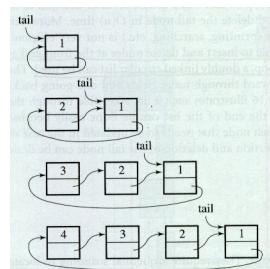
Uma lista circular ligada de inteiros



Inserção na cauda de uma lista circular



Inserção na cabeça de uma lista circular



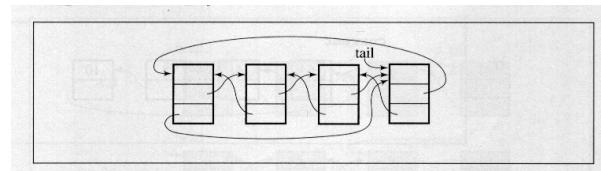
© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

72

# Listas Circulares

## conceitos

Uma lista circular duplamente ligada



Data Structures and Algorithms in JAVA, Adam Drozdek

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

73

# Listas Circulares

## vantagens e desvantagens

??

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

74

36

## [Listas Duplamente Ligadas ]

LinkedList vs ArrayList (in JAVA)

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

75

## [Listas Duplamente Ligadas ]

76

37

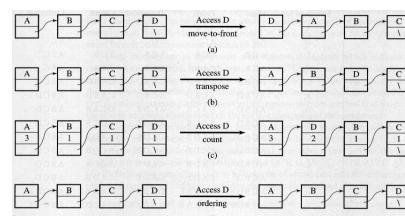
# Listas Auto-Organizadas

## conceitos

Quatro formas de organizar os elementos que se procuram numa lista:

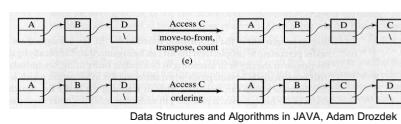
- À CABEÇA – quando o elemento é encontrado coloca-lo à cabeça.
- TRANSPOSIÇÃO - quando o elemento é encontrado troca-o com o seu predecessor a menos que já esteja à cabeça.
- CONTAGEM – ordena a lista pelo número de vezes que cada elemento é acedido.
- POR CRITÉRIO – recorre a um critério adequado à informação sob escrutínio.

### CONSULTA



© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

### INSCRIÇÃO



Data Structures and Algorithms in JAVA, Adam Drozdek

77

# Listas Auto-Organizadas

## exemplo

Element Searched for	Plain	Move-to-Front	Transpose	Count	Ordering
A:	A	A	A	A	A
C:	AC	AC	AC	AC	AC
B:	ACB	ACB	ACB	ACB	ABC
C:	ACB	CAB	CAB	CAB	ABC
D:	ACBD	CABD	CABD	CABD	ABCD
A:	ACBD	ACBD	ACBD	CABD	ABCD
D:	ACBD	DACB	ACDB	DCAB	ABCD
A:	ACBD	ADCB	ACDB	ADCB	ABCD
C:	ACBD	CADB	CADB	CADB	ABCD
A:	ACBD	ACDB	ACDB	ACDB	ABCD
C:	ACBD	CADB	CADB	ACDB	ABCD
E:	ACBDE	CADBE	CADBE	CADBE	ABCDE
E:	ACBDE	ECADB	CADEB	CAEDB	ABCDE

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

78

# Listas Auto-Organizadas

vantagens e desvantagens

Quatro formas de organizar os elementos que chegam a uma lista:

- À CABEÇA – quando o elemento é encontrado coloca-lo à cabeça.
- TRANSPOSIÇÃO - quando o elemento é encontrado troca-o com o seu predecessor a menos que já esteja à cabeça.
- CONTAGEM – ordena a lista pelo número de vezes que cada elemento é acedido.

??

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

79

# Análise de Complexidade

(leituras)

Sedgewick, Cap.4, pp. 127-153

Sedgewick, Cap.3, pp. 69-126

- Compreender o conceito de Tipo Abstracto de Dados (TAD)
- Conhecer e compreender as seguintes estruturas de dados:
  - matrizes;
  - pilhas; filas
  - listas ligadas
  - cadeias de caracteres
  - estruturas de dados compostas
  - Saber como desenhar um programa simples de simulação de filas de espera

© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

105

# Análise de Complexidade

... bom trabalho, FIM!



© DEI Carlos Lisboa Bento ALGORITMOS E ESTRUTURAS DE DADOS

106

# Algoritmos e Estruturas de Dados

listas de saltos

■ 2020-2021

■ Carlos Lisboa Bento

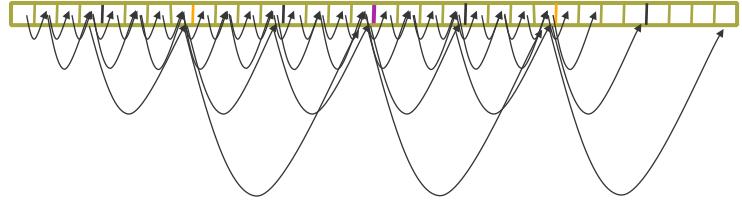
107

1  
0  
9

## Listas de Saltos

conceitos

Ainda por outras palavras ... ...  
 o primeiro nó aponta para o seguinte  
 o segundo para o nó duas posições à frente  
 o quarto para o nó quatro posições à frente  
 o oitavo para o nó oito posições à frente



109

1  
1  
0

## Listas de Saltos

conceitos

*Data Structures and Algorithms in JAVA, Adam Drozdak*

Consideremos uma situação em que temos 32 nós

*Lista com n nós*  
*Para k e i tal que 1 ≤ k ≤ ⌈lg n⌉ e 1 ≤ i ≤ ⌊n / 2^{k-1}⌋ - 1*  
*Um nó na posição  $2^{k-1} \cdot i$  aponta para um nó na posição  $2^{k-1} \cdot (i + 1)$*

Número de referências nos nós:

1/2 dos nós	1 referência
1/4 dos nós	2 referências
1/8 dos nós	3 referências
1/16 dos nós	4 referências

O nº de referências de um nó indica o nível do nó.  
**NUM DE NIVEIS** é maxLevel =  $\log n + 1$

110

1  
1  
1

# Listas de Saltos

exemplo

Procura numa lista de saltos

Procura do elemento 19

19 > 10

19 > 17

19 = 19

19 < 22

19 < 22

Data Structures and Algorithms in JAVA, Adam Drozdek

111

1  
1  
2

# Listas de Saltos

conceitos

INSERÇÃO ?

ELIMINAÇÃO ?

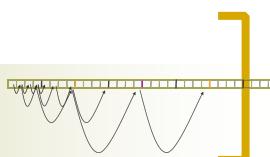
112

1  
1  
3

## Listas de Saltos

conceitos

**Solução:**



- Relaxar o posicionamento dos nós de diferentes níveis.
- Manter (estatisticamente) o número de nós por nível.
- Na geração de novos nós criar nós de diferentes níveis segundo uma probabilidade que segue o número pretendido de nós por nível.

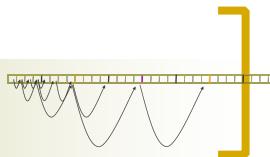
113

1  
1  
4

## Listas de Saltos

conceitos

**Exemplo:**



- Para uma lista de saltos em que maxLevel = 5 que poderá conter 31 elementos
- Na inserção de cada nó é gerado um valor aleatório entre 0 e 30 que conforme esteja num dos seguintes intervalos vai resultar na geração de um nó do seguinte nível:

r	nível do nó a ser criado
[0]	5
[1..2]	4
[3..6]	3
[7..14]	2
[15..30]	1

114

1  
1  
5

## Listas de Saltos

demos

<https://people.ok.ubc.ca/y lucet/DS/SkipList.html>

115

1  
1  
6

## Listas de Saltos

vantagens e desvantagens

PIOR CASO ?

MELHOR CASO ?

CASO MÉDIO ?

116

1  
1  
7

# Listas de Saltos

vantagens e desvantagens

## Vantagens

- implementação mais simples do que para uma árvore equilibrada.
- menos memória ocupada do que numa árvore equilibrada.
- inserção e eliminação não envolve re-equilíbrio
- tempo de procura, inserção e apagamento em situações não degeneradas  $O(\ln N)$

## Desvantagens

- possibilidade de ter situações degeneradas e em geral tempos de procura e de inserção mais altos do que nas árvores
- depende de um gerador de números aleatórios o que dificulta a depuração do programa.

David M. Howard

117

1  
1  
8

# Listas de Saltos

implementação

```
public class IntSkipListNode {
    public int key;
    public IntSkipListNode[] next;
    IntSkipListNode(int i, int n) {
        key = i;
        next = new IntSkipListNode[n];
        for (int j = 0; j < n; j++)
            next[j] = null;
    }
}
```

118

1  
1  
9

## Listas de Saltos

### implementação

```

import java.util.Random;

public class IntSkipList {
    private int maxLevel;
    private IntSkipListNode[] root;
    private int[] powers;
    private Random rd = new Random();
    IntSkipList() {
        this(4);
    }
    IntSkipList (int i) {
        maxLevel = i;
        root = new IntSkipListNode[maxLevel];
        powers = new int[maxLevel];
        for (int j = 0; j < maxLevel; j++)
            root[j] = null;
        choosePowers();
    }
    public boolean isEmpty() {
        return root[0] == null;
    }
}

```

119

1  
2  
0

## Listas de Saltos

### implementação

```

public void choosePowers() {
    powers[maxLevel-1] = (2 << (maxLevel-1)) - 1; // 2^(maxLevel - 1)-1
    for (int i = maxLevel - 2, j = 0; i >= 0; i--, j++)
        powers[i] = powers[i+1] - (2 << j); // powers[i+1]-2^j
}

public int chooseLevel() {
    int i, r = Math.abs(rd.nextInt()) % powers[maxLevel-1] + 1;
    for (i = 1; i < maxLevel; i++)
        if (r < powers[i])
            return i-1; // return a level < the highest level;
    return i-1; // return the highest level;
}

```

120

1  
2  
1

# Listas de Saltos

## implementação

```
// make sure (with isEmpty()) that skipListSearch() is called for a nonempty list;
public int skipListSearch (int key) {
    int lvl;
    IntSkipListNode prev, curr; // find the highest non-null
    for (lvl = maxLevel-1; lvl >= 0 && root[lvl] == null; lvl--); // level;
    prev = curr = root[lvl];
    while (true) {
        if (key == curr.key) // success if equal;
            return curr.key;
        else if (key < curr.key) { // if smaller, go down,
            if (lvl == 0) // if possible
                return 0;
            else if (curr == root[lvl]) // by one level
                curr = root[-lvl]; // starting from the
            else curr = prev.next[-lvl]; // predecessor which
                // can be the root;
        } else { // if greater,
            prev = curr; // go to the next
            if (curr.next[lvl] != null) // non-null node
                curr = curr.next[lvl]; // on the same level
            else { // or to a list on a lower level;
                for (lvl--; lvl >= 0 && curr.next[lvl] == null; lvl--);
                if (lvl >= 0)
                    curr = curr.next[lvl];
                else return 0;
            } } } }
```

121

1  
2  
2

# Listas de Saltos

## implementação

```
public void skipListInsert (int key) {
    IntSkipListNode curr = new IntSkipListNode[maxLevel];
    IntSkipListNode[] prev = new IntSkipListNode[maxLevel];
    IntSkipListNode newNode;
    int lvl, i;
    curr[maxLevel-1] = root[maxLevel-1];
    curr[maxLevel-1].next[maxLevel-1] = null;
    for (lvl = maxLevel - 1; lvl >= 0; lvl--) {
        while (curr[lvl] != null && curr[lvl].key < key) { // go to the next
            prev[lvl] = curr[lvl]; // if smaller;
            curr[lvl] = curr[lvl].next[lvl];
        }
        if (curr[lvl] != null && curr[lvl].key == key) // don't include
            return; // duplicates;
        if (lvl > 0) // go one level down
            if (prev[lvl] == null) { // if not the lowest
                curr[lvl-1] = root[lvl-1]; // level, using a link
                prev[lvl-1] = null; // either from the root
            } else { // or from the predecessor;
                curr[lvl-1] = prev[lvl].next[lvl-1];
                prev[lvl-1] = prev[lvl];
            }
    }
    lvl = chooseLevel(); // generate randomly level for newNode;
    newNode = new IntSkipListNode(key,lvl+1);
    for (i = 0; i <= lvl; i++) { // initialize next fields of
        newNode.next[i] = curr[i]; // newNode and reset to newNode
        if (prev[i] == null) // either fields of the root
            root[i] = newNode; // or next fields of newNode's
        else prev[i].next[i] = newNode; // predecessors;
    }
}
```

122

1  
2  
3

## Listas de Saltos

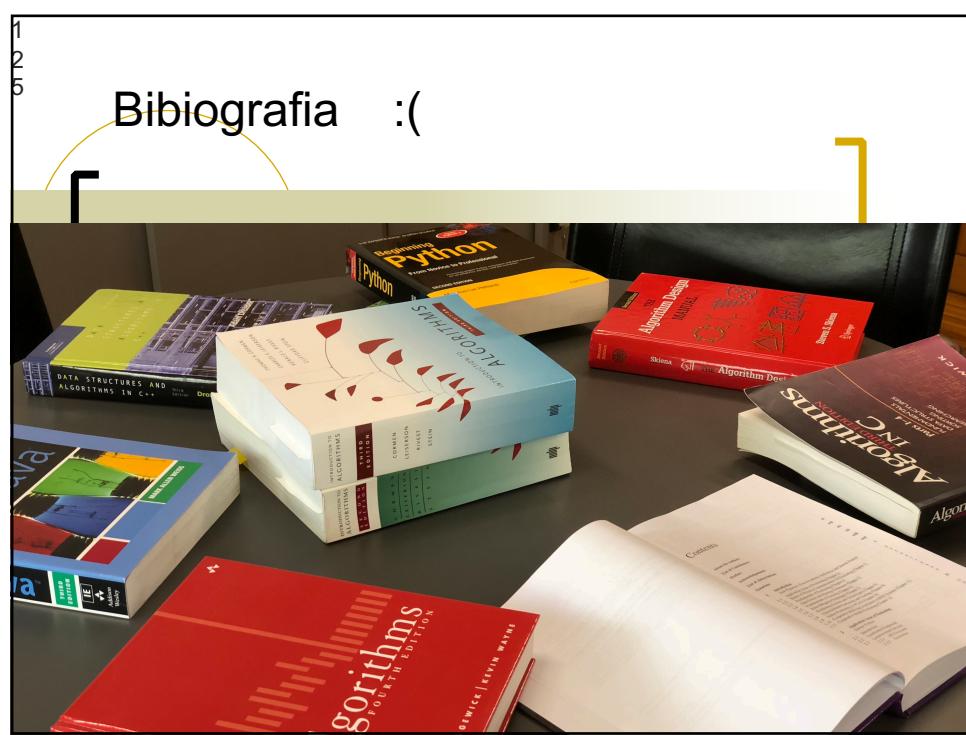
implementação

```

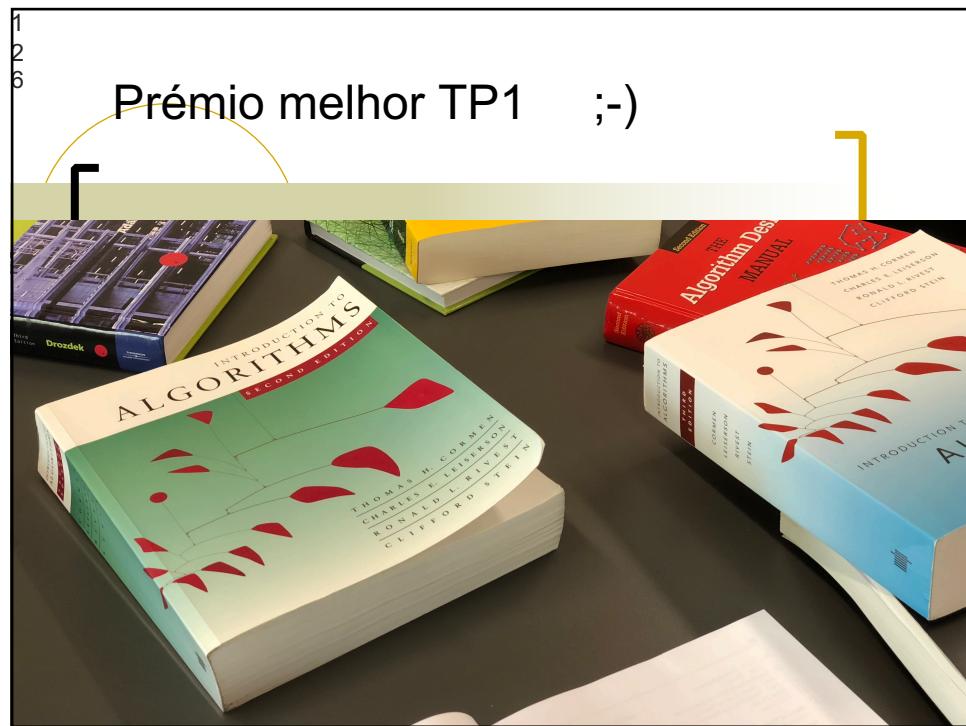
public void display() {
    System.out.println("Print list:");
    for (int i = maxLevel-1; i >= 0; i--) { // print list level by level;
        for (IntSkipListNode p = root[i]; p != null; p = p.next[i])
            System.out.print(p.key + " ");
        System.out.println("|");
    }
}

```

123



125



129

The slide features a light blue background with a large, semi-transparent yellow circle at the top left. Inside the circle, the text 'Algoritmos e Estruturas de Dados' is written in black, bold font. Below it, the word 'árvores' is written in a smaller, regular black font. To the right of the title is a large, stylized yellow bracket that spans most of the slide's width. At the bottom center, there is a small yellow square followed by the text '2020-2021'. Further down, another small yellow square is followed by the text 'Carlos Lisboa Bento'.

130

The slide has a light blue background. In the top left corner, there are three small numbers: '1', '3', and '1'. To the right of these numbers is the word 'Árvore' in large black font, with the suffix 's' partially cut off. Above 'Árvore' is the word 'conceitos' in a smaller black font. Below 'Árvore' is a large, semi-transparent yellow circle containing the text 'LISTAS LINEARES' in black, bold font. To the right of the circle is a large, stylized yellow bracket. At the bottom of the slide, there is a block of text in purple:

Acesso aos seus elementos é feito **sequencialmente**  
(num ou nos dois sentidos consoante se trate de listas simples ou duplas).  
INCONVENIENTE: por exemplo na **pesquisa de informação**.

Below this text is the heading 'ÁRVORES / ÁRVORES BINÁRIAS' in black, bold font. At the very bottom, there is a small line of text in black: 'Uma possível solução...'

131

1  
3  
2

## Árvores

conceitos

[ ]

**Árvore:**

Um nó **R** é a **RAIZ**.

Todos os nós excepto **R** têm exactamente uma ligação para eles a partir de um nó **p**. **p** é o **pai** de **c**, **c** é o **filho** de **p**.

Existe um caminho único da raiz **c** para cada nó.  
O número de ligações que têm de ser seguidas é o comprimento do caminho.

```

graph TD
    A((A)) --> B((B))
    A --> C((C))
    B --> D((D))
    B --> E((E))
    C --> F((F))
    E --> G((G))
    E --> H((H))
    E --> I((I))
  
```

132

1  
3  
3

## Árvores

conceitos

[ ]

Altura de um nó (**h**) - comprimento do caminho mais longo entre esse nó e as folhas da árvore (incluindo o próprio).

Profundidade de um nó (**d**) - comprimento do caminho da raiz até esse nó.

Tamanho de um nó (**s**) - número de descendentes desse nó mais ele próprio

```

graph TD
    A((A)) --> B((B))
    A --> C((C))
    B --> F((F))
    B --> G((G))
    C --> D((D))
    D --> H((H))
    E((E)) --> I((I))
    E --> J((J))
    J --> K((K))
  
```

Nó	Altura	Profund.	Tam.
A	4	0	11
B	2	1	3
C	1	1	1
D	2	1	2
E	3	1	4
F	1	2	1
G	1	2	1
H	1	2	1
I	1	2	1
J	2	2	2
K	1	3	1

133

1  
3  
4

## Árvores

aplicações

Apl #1  
Organização de dados, por exemplo que contêm vários níveis de informação (por exemplo, um edifício tem salas, uma sala tem objectos, os objectos podem ter outros objectos...)

Apl #2  
Árvores de expressões. O valor de um nó é o resultado de aplicar o operador representado nesse nó utilizando como operandos os filhos desse nó

Apl #3  
Organizações taxonómicas de dados (por exemplo taxonomia de Lineu, "família, género, espécie...")

```

graph TD
    x((x)) --> plus((+))
    x --> minus((-))
    plus --> a((a))
    plus --> b((b))
    minus --> c((c))
    minus --> d((d))
  
```

134

1  
3  
5

## Árvores Binárias

conceitos

Árvore binária:  
conjunto finito de elementos que ou está vazio ou contém um elemento chamado raiz da árvore ligado a dois conjuntos disjuntos em que cada um é, por sua vez uma árvore binária.

Árvore binária

```

graph TD
    A((A)) --> B((B))
    A --> C((C))
    B --> D((D))
    B --> E((E))
    C --> F((F))
    F --> G((G))
    F --> H((H))
  
```

Árvore não binária

```

graph TD
    A((A)) --> B((B))
    A --> C((C))
    B --> D((D))
    B --> E((E))
    C --> F((F))
    F --> G((G))
    F --> H((H))
    F --> I((I))
  
```

135

1  
3  
6

## Árvores Binárias

conceitos

Raiz

Subárvore esquerda

Subárvore direita

Folha

- Dois nós são irmãos se são os filhos direito e esquerdo do mesmo pai;
- Um nó que não tem filhos é uma folha;
- No exemplo acima, n4 e n5 são descendentes de n2 e este é ascendente de ambos.

136

1  
3  
7

## Árvores Binárias

conceitos

- Nível de uma árvore binária:
  - Nível da raiz = 0;
  - Nível de qualquer outro nó = nível do pai + 1.
- Árvore binária completa de nível  $n$  é uma árvore em que cada nó de nível  $n$  é uma folha e em que todos os nós de nível menor do que  $n$  têm subárvore direita e esquerda não vazias

Árvore binária completa de nível 2

137

1  
3  
8

# Árvores Binárias

conceitos

## Travessia de uma árvore

- Em muitos problemas é necessário percorrer uma árvore binária, atravessando todos os seus nós e enumerando-os;
- Numa árvore não há uma ordem “natural” de percurso, como, por exemplo, nas listas ligadas;
- Designa-se por *visitar um nó* a acção de atingir um dado nó de uma árvore e efectuar (ou não) uma qualquer operação com a informação nele armazenada;
- Vamos definir três métodos para atravessar uma árvore:
  - *pré-ordem*, *ordem*, *pos-ordem*
- Utiliza-se uma definição recursiva, pelo que percorrer uma árvore implicará percorrer a raiz e percorrer as suas subárvores esquerda e direita.

138

1  
3  
9

# Árvores Binárias

conceitos

## Travessia de uma árvore (cont.)

- Em *pré-ordem* (ou pré-fixado)
  - 1) Visitar a raiz;
  - 2) Atravessar a subárvore esquerda em pré-ordem;
  - 3) Atravessar a subárvore direita em pré-ordem.
- Em *ordem* (ou central)
  - 1) Atravessar a subárvore esquerda em ordem;
  - 2) Visitar a raiz;
  - 3) Atravessar a subárvore direita em ordem.
- Em *pos-ordem* (ou pos-fixado)
  - 1) Atravessar a subárvore esquerda em pos-ordem;
  - 2) Atravessar a subárvore direita em pos-ordem.
  - 3) Visitar a raiz;

139

1  
4  
0

# Árvores Binárias

conceitos

Travessia de uma árvore (exemplo)

```

graph TD
    A((A)) --- B((B))
    A --- C((C))
    B --- D((D))
    B --- G((G))
    C --- E((E))
    C --- F((F))
    D --- G
    E --- H((H))
    E --- I((I))
  
```

140

1  
4  
1

# Árvores Binárias

conceitos

Travessia de uma árvore (exemplo)

```

graph TD
    A((A)) --- B((B))
    A --- C((C))
    B --- D((D))
    B --- G((G))
    C --- E((E))
    C --- F((F))
    D --- G
    E --- H((H))
    E --- I((I))
  
```

Pré-ordem: A B D G C E H I F  
 Em ordem: D G B A H E I C F  
 Pos-ordem: G D B H I E F C A

141

1  
4  
2

## Árvores Binárias

conceitos

Travessia de uma árvore (outro exemplo...)

```

graph TD
    A((A)) --- B((B))
    A --- C((C))
    B --- D((D))
    B --- E((E))
    C --- F((F))
    C --- G((G))
    D --- H((H))
    D --- I((I))
    E --- J((J))
    E --- K((K))
    F --- L((L))
  
```

142

1  
4  
3

## Árvores Binárias

conceitos

Travessia de uma árvore (outro exemplo...)

Pré-ordem: A B C E I F J D G H K L
Em ordem: E I C F J B G D K H L A
Pos-ordem: I E J F C G K L H D B A

143

1  
4  
4

## Árvores Binárias

implementação

```

final class BinaryNode
{
    BinaryNode( )
    { this( null ); }

    BinaryNode( Object theElement )
    { this( theElement, null, null ); }

    BinaryNode( Object theElement,
                BinaryNode lt, BinaryNode rt )
    {
        element = theElement;
        left = lt;
        right = rt;
    }

    static int size( BinaryNode t )
    {
        if( t == null )
            return 0;
        else
            return 1 + size( t.left ) + size( t.right );
    }

    static int height( BinaryNode t )
    {
        if( t == null )
            return -1;
        else
            return 1 +
                Math.max( height( t.left ), height( t.right ) );
    }
}

void printPreOrder( )
{
    System.out.println( element ); // Node
    if( left != null )
        left.printPreOrder(); // Left
    if( right != null )
        right.printPreOrder(); // Right
}

void printPostOrder( )
{
    if( left != null )
        left.printPostOrder(); // Left
    if( right != null )
        right.printPostOrder(); // Right
    System.out.println( element ); // Node
}

void printlnOrder( )
{
    if( left != null )
        left.printlnOrder(); // Left
    System.out.println( element ); // Node
    if( right != null )
        right.printlnOrder(); // Right
}

```

144

1  
4  
5

## Árvores Binárias

implementação

```

BinaryNode duplicate( )
{
    BinaryNode root = new BinaryNode( element );

    if( left != null ) // If there's a left subtree
        root.left = left.duplicate(); // Duplicate; attach
    if( right != null ) // If there's a right subtree
        root.right = right.duplicate(); // Duplicate; attach
    return root; // Return resulting tree
}

// Friendly data; accessible by other package routines
Object element;
BinaryNode left;
BinaryNode right;
}

```

Aqui a referência para o nó não é passada como um parâmetro - é usado element, left e right que são declarados friendly na classe BinaryNode (comparar com o método size)

Recordar ainda que este método pertence à classe BinaryNode

145

1  
4  
6

## Árvores Binárias

implementação

... ainda sobre os métodos de travessia definidos nesta classe

```

public void printPreOrder()
{
    if( root != null )
        root.printPreOrder();
}

void printPreOrder()
{
    System.out.println( element );
    if( left != null )
        left.printPreOrder();
    if( right != null )
        right.printPreOrder();
}

```

```

public void printInOrder()
{
    if( root != null )
        root.printInOrder();
}

void printInOrder()
{
    if( left != null )
        left.printInOrder();
    System.out.println( element );
    if( right != null )
        right.printInOrder();
}

```

```

public void printPostOrder()
{
    if( root != null )
        root.printPostOrder();
}

void printPostOrder()
{
    if( left != null )
        left.printPostOrder();
    if( right != null )
        right.printPostOrder();
    System.out.println( element );
}

```

quantas chamadas dos métodos respectivos?  
número de chamadas igual ao número de nós  
O(N)

146

1  
4  
7

## Árvores Binárias de Pesquisa

conceitos

A definição de árvore apresentada anteriormente não apresenta qualquer restrição acerca do posicionamento dos nós dentro da árvore.

EM CONSEQUÊNCIA: a pesquisa de um qualquer elemento teria que passar por todos os nós da árvore (tal como no caso das listas ligadas).

- SOLUÇÃO: impôr que, para qualquer nó da árvore, todos os elementos da sua subárvore esquerda sejam menores que ele e que todos os elementos da sua subárvore direita sejam maiores que ele

Por exemplo, para pesquisar o elemento 17 na árvore da figura seria apenas necessário percorrer os elementos assinalados:

Operações find, insert e remove c/ compl. O(log N) SE FIZERMOS ALGUMAS ALTERAÇÕES, SENÃO temos O(N) !!!

147

1  
4  
8

## Árvores Binárias de Pesquisa

conceitos

```

package DataStructures;
import Supporting.*;
import Exceptions.*;
import Supporting.Comparable;

// *****PUBLIC OPERATIONS*****
// void insert( x ) --> Insert x
// void remove( x ) --> Remove x
// void removeMin() --> Remove smallest item
// Comparable find( x ) --> Return item that matches x
// Comparable findMin() --> Return smallest item
// Comparable findMax() --> Return largest item
// boolean isEmpty() --> Return true if empty; else false
// void makeEmpty() --> Remove all items
// void printTree() --> Print tree in sorted order
// *****ERRORS*****
// Most routines throw ItemNotFound on
// various degenerate conditions
// Insert throws DuplicateItem if item is already
// in the tree
// @author Mark Allen Weiss

```

```

public interface SearchTree
{
    void insert( Comparable x ) throws DuplicateItem;
    void remove( Comparable x ) throws ItemNotFound;
    void removeMin() throws ItemNotFound;
    Comparable findMin() throws ItemNotFound;
    Comparable findMax() throws ItemNotFound;
    Comparable find( Comparable x ) throws ItemNotFound;
    void makeEmpty();
    boolean isEmpty();
    void printTree();
}

```

148

1  
4  
9

## Árvores Binárias de Pesquisa

conceitos

find( x ) --> Return item that matches x

149

1  
5  
0

## Árvores Binárias de Pesquisa

conceitos

`findMin() --> Return smallest item`  
`findMax() --> Return largest item`

150

1  
5  
1

## Árvores Binárias de Pesquisa

conceitos

`void removeMin() --> Remove smallest item`

151

1  
5  
2

## Árvores Binárias de Pesquisa

conceitos

`void remove( x )` --> Remove x

remover uma folha

Pode ser simplesmente removida

152

1  
5  
3

## Árvores Binárias de Pesquisa

conceitos

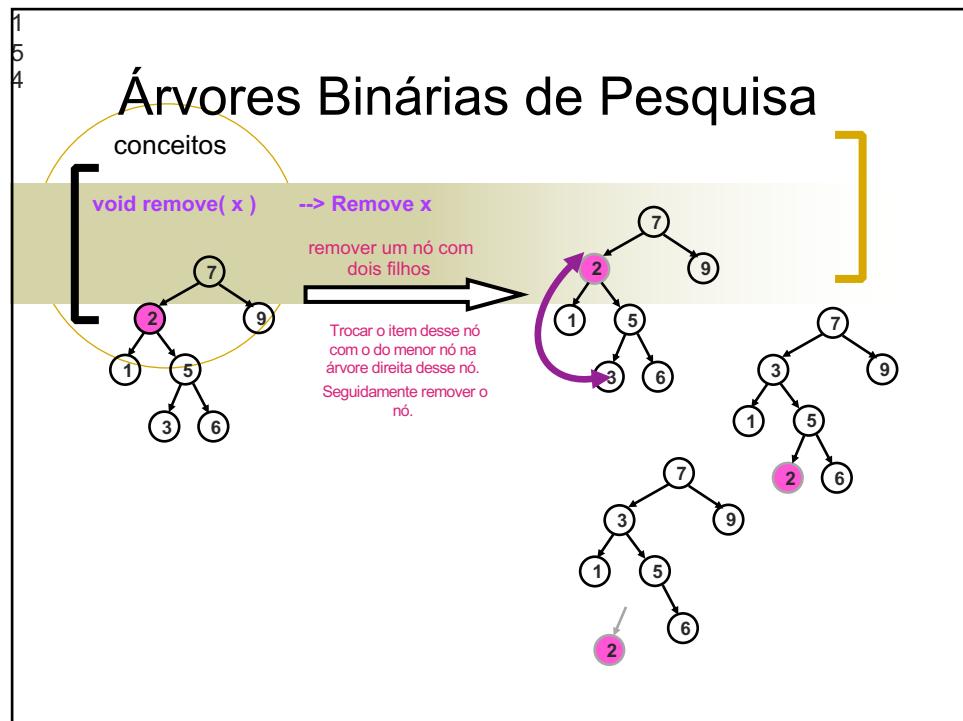
`void remove( x )` --> Remove x

remover um nó que só tem um filho

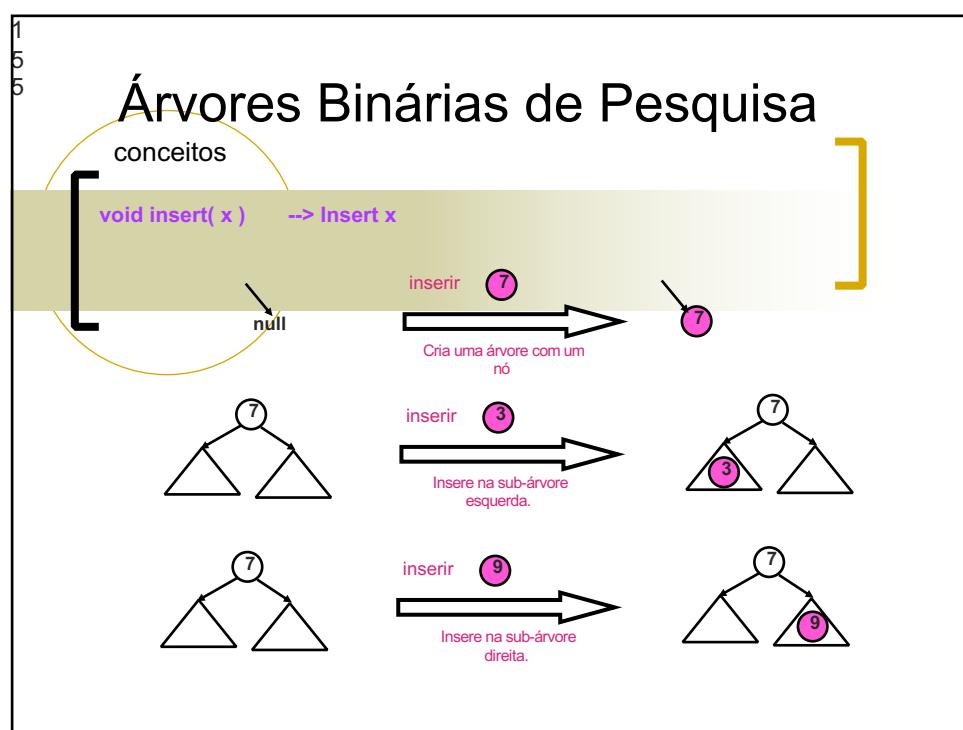
Pode ser removido, sendo a referência do pai para bypassed para o filho do nó a remover.

A raiz é um caso especial (não tem pai) no entanto este procedimento aplica-se também a este caso.

153



154



155

1  
5  
6

## Árvores Binárias de Pesquisa

implementação

```

package DataStructures;
import Supporting.*;
import Exceptions.*;
import Supporting.Comparable;
// @author Mark Allen Weiss
public class BinarySearchTree
    implements SearchTree
{
    public BinarySearchTree()
    {
        root = null;
    }

    public void insert( Comparable x )
        throws DuplicateItem
    {
        root = insert( x, root );
    }

    public void remove( Comparable x )
        throws ItemNotFound
    {
        root = remove( x, root );
    }

    public void removeMin()
        throws ItemNotFound
    {
        root = removeMin( root );
    }

    public Comparable findMin() throws ItemNotFound
    {
        return findMin( root ).element;
    }

    public Comparable findMax() throws ItemNotFound
    {
        return findMax( root ).element;
    }

    public Comparable find( Comparable x )
        throws ItemNotFound
    {
        return find( x, root ).element;
    }

    public void makeEmpty()
    {
        root = null;
    }

    public boolean isEmpty()
    {
        return root == null;
    }

    public void printTree()
    {
        if( root == null )
            System.out.println( "Empty tree" );
        else
            printTree( root );
    }
}

```

156

## Árvores Binárias de Pesquisa

implementação

```

protected BinaryNode insert( Comparable x, BinaryNode t ) throws DuplicateItem
{
    if( t == null )
        t = new BinaryNode( x, null, null );
    else if( x.compareTo( t.element ) < 0 )
        t.left = insert( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = insert( x, t.right );
    else
        throw new DuplicateItem( "SearchTree insert" );
    return t;
}

protected BinaryNode remove( Comparable x, BinaryNode t ) throws ItemNotFound
{
    if( t == null )
        throw new ItemNotFound( "SearchTree remove" );
    if( x.compareTo( t.element ) < 0 )
        t.left = remove( x, t.left );
    else if( x.compareTo( t.element ) > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null ) // Two children
    {
        t.element = findMin( t.right ).element;
        t.right = removeMin( t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return t;
}

```

157

1  
5  
8

## Árvores Binárias de Pesquisa

### implementação

```

protected BinaryNode removeMin( BinaryNode t ) throws ItemNotFound
{
    if( t == null )
        throw new ItemNotFound( "SearchTree removeMin" );
    if( t.left != null )
        t.left = removeMin( t.left );
    else
        t = t.right;
    return t;
}

protected BinaryNode findMin( BinaryNode t ) throws ItemNotFound
{
    if( t == null )
        throw new ItemNotFound( "SearchTree findMin" );
    while( t.left != null )
        t = t.left;
    return t;
}

protected BinaryNode findMax( BinaryNode t ) throws ItemNotFound
{
    if( t == null )
        throw new ItemNotFound( "SearchTree findMax" );
    while( t.right != null )
        t = t.right;
    return t;
}

```

158

1  
5  
9

## Árvores Binárias de Pesquisa

### implementação

```

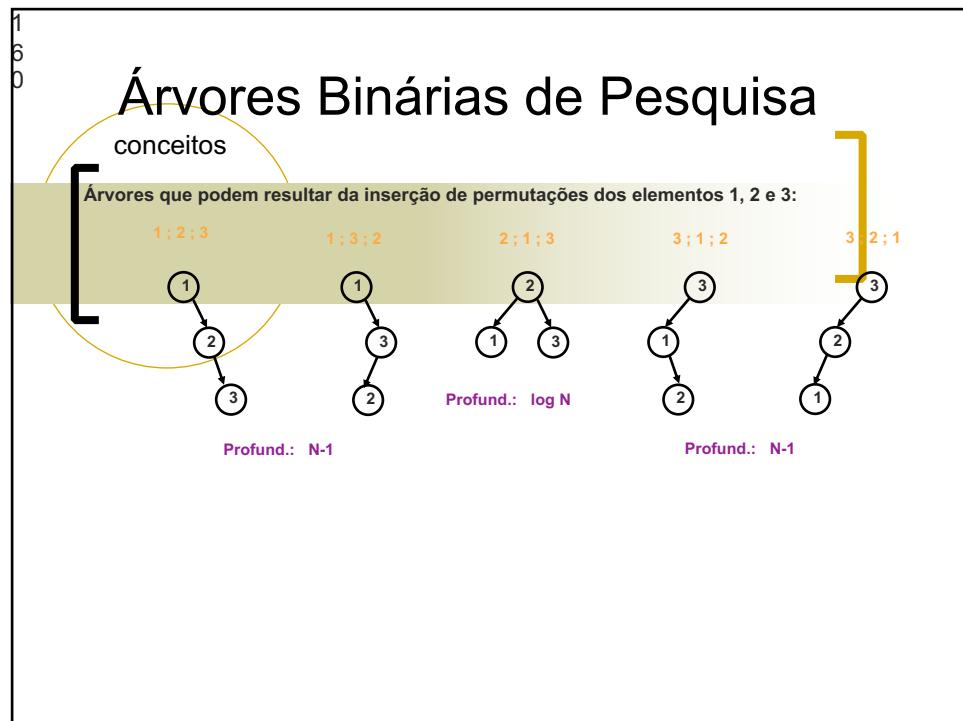
protected BinaryNode find( Comparable x, BinaryNode t ) throws ItemNotFound
{
    while( t != null )
        if( x.compareTo( t.element ) < 0 )
            t = t.left;
        else if( x.compareTo( t.element ) > 0 )
            t = t.right;
        else
            return t; // Match
    throw new ItemNotFound( "SearchTree find" );
}

protected void printTree( BinaryNode t )
{
    if( t != null )
    {
        printTree( t.left );
        System.out.println( t.element.toString() );
        printTree( t.right );
    }
}

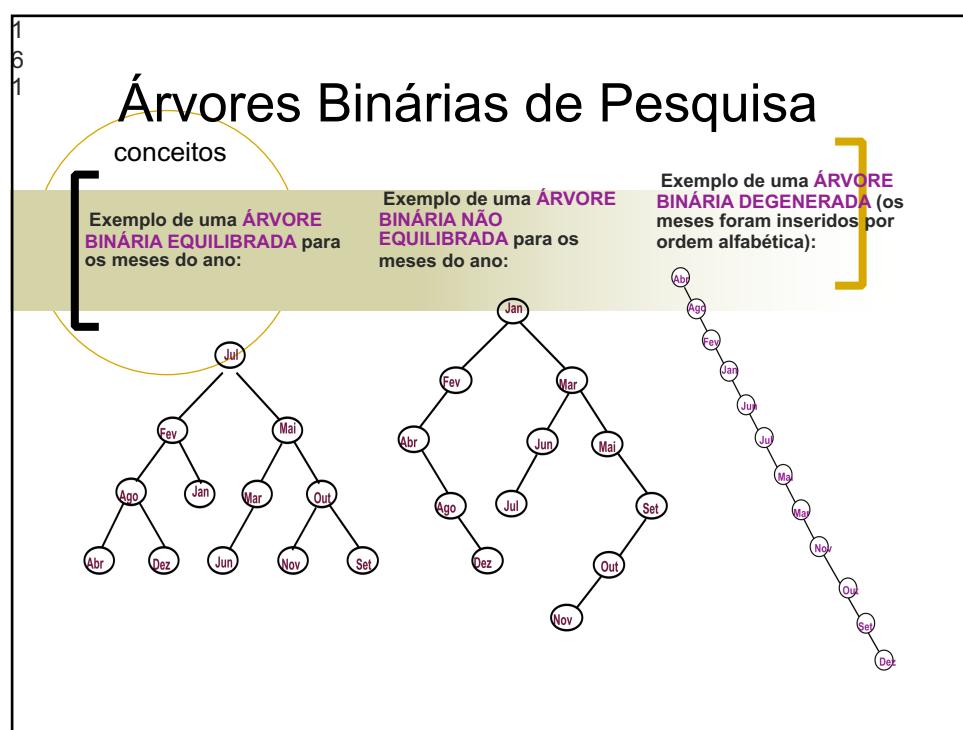
/** The tree root. */
protected BinaryNode root;

```

159



160



161

1  
6  
2

## Árvores Binárias de Pesquisa

conceitos

**OBSERVAÇÃO:** as árvores binárias equilibradas têm como propriedade estrutural o garantirem profundidade logarítmica no pior caso.

**DIFICULDADE:** Um método de inserção de elementos numa árvore em que, após cada inserção, a árvore fique perfeitamente equilibrada é muito complexo.

**SOLUÇÃO:** uma possível solução são as árvores AVL.

162

1  
6  
3

## Árvores AVL

conceitos

Definição de árvore equilibrada segundo Adelson-Velkii e Landis (árvore AVL):

Uma árvore binária está equilibrada se e só se

para cada elemento a altura das suas subárvores diferir no máximo de uma unidade.

163

1  
6  
4

## Árvores

conceitos

**Altura de um nó ( $h$ )** - comprimento do caminho mais longo entre esse nó e as folhas da árvore (incluindo o próprio).

**Profundidade de um nó ( $d$ )** - comprimento do caminho da raiz até esse nó.

**Tamanho de um nó ( $s$ )** - número de descendentes desse nó mais ele próprio

Nó	Altura	Profund.	Tam.
A	4	0	11
B	2	1	3
C	1	1	1
D	2	1	2
E	3	1	4
F	1	2	1
G	1	2	1
H	1	2	1
I	1	2	1
J	2	2	2
K	1	3	1

164

1  
6  
5

## Árvores AVL

conceitos

**Caso da inserção de um elemento na subárvore esquerda**

$h_e = h_d$  : E e D ficam com alturas diferentes mas o critério AVL ainda é válido

$h_e < h_d$  : E e D ficam com alturas iguais

$h_e > h_d$  : O critério AVL deixa de ser válido; é preciso reestruturar a árvore

**Factor de equilíbrio de um elemento  $t$  de uma árvore binária:**  $FE(t) = h_e - h_d$

**Numa árvore AVL:**  $FE(t) = -1, 0, +1$

**Para reequilibrar uma árvore são efectuadas rotações em torno do antecessor mais próximo do elemento inserido que tenha  $FE = \pm 2$**

165

1  
6  
6  
**Árvores AVL**  
conceitos

Nas árvores AVL são feitas 4 tipo de rotações que vão resolver 4 situações de inserção.

Situação 1:  
DD - O novo elemento Y é inserido na subárvore Direita da subárvore Direita de k1.

Inseriu Y

Faz rotação simples com filho k2 à direita

166

1  
6  
7  
**Árvores AVL**  
conceitos

Exemplo de inserção AVL

Meses a inserir na árvore:  
Mai, Mar, Nov, Ago, Abr, Jan, Dez, Jun, Fev

Elemento a inserir	Depois da inserção	Depois de equilibrada
Mai		—
Mar		—

167

1  
6  
8

## Árvores AVL

conceitos

**Exemplo de inserção AVL (cont.)**

Meses a inserir na árvore:  
Mai, Mar, Nov, Ago, Abr, Jan, Dez, Jun, Fev

Elemento a inserir      Depois da inserção      Depois de equilibrada

Nov

Mai<sup>-2</sup>  
D Mar<sup>-1</sup>  
D Ago<sup>0</sup>

Rotação DD

Mai<sup>0</sup>  
Mar<sup>0</sup>  
Nov<sup>0</sup>

168

1  
6  
9

## Árvores AVL

conceitos

**Exemplo de inserção AVL (cont.)**

Meses a inserir na árvore:  
Mai, Mar, Nov, Ago, Abr, Jan, Dez, Jun, Fev

Elemento a inserir      Depois da inserção      Depois de equilibrada

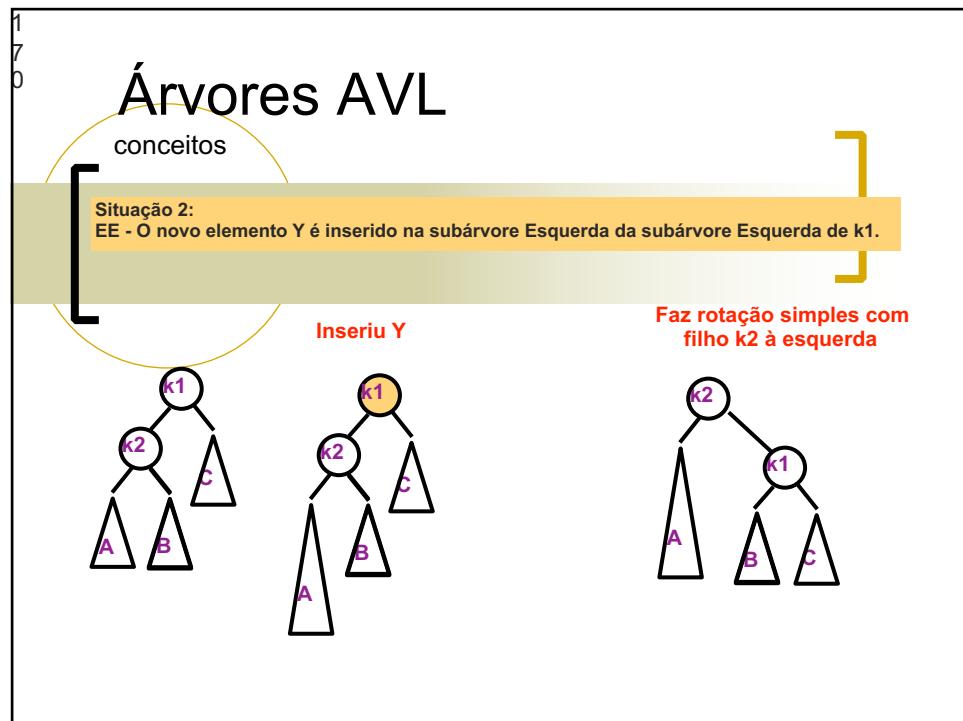
Ago

Mai<sup>+1</sup>  
Ago<sup>0</sup>

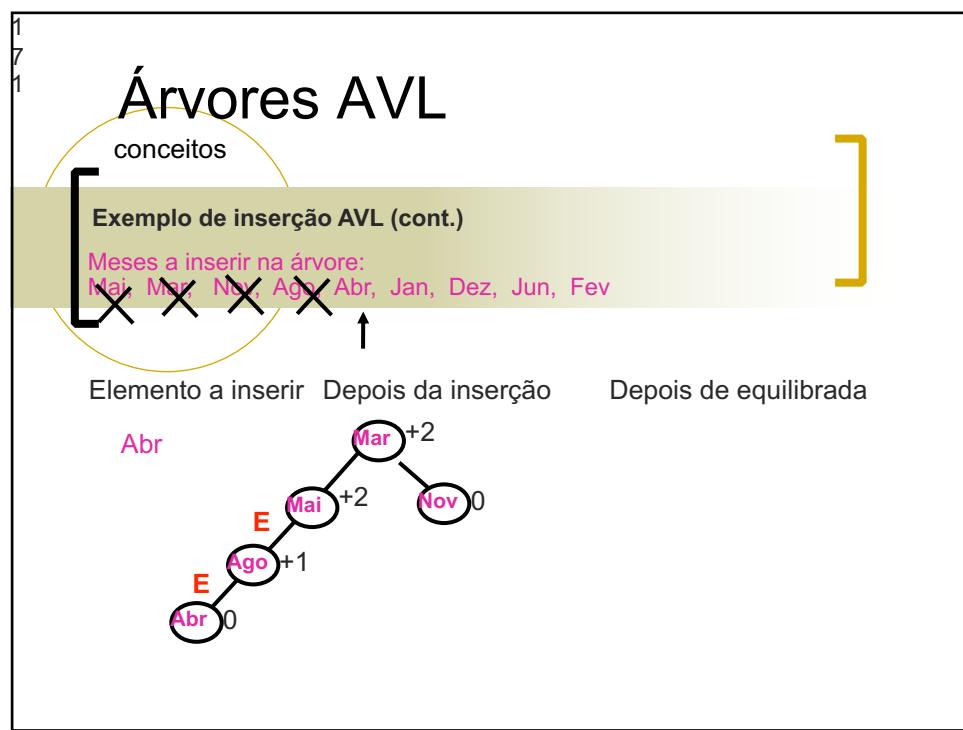
Mar<sup>+1</sup>  
Mai<sup>+1</sup>  
Nov<sup>0</sup>

—

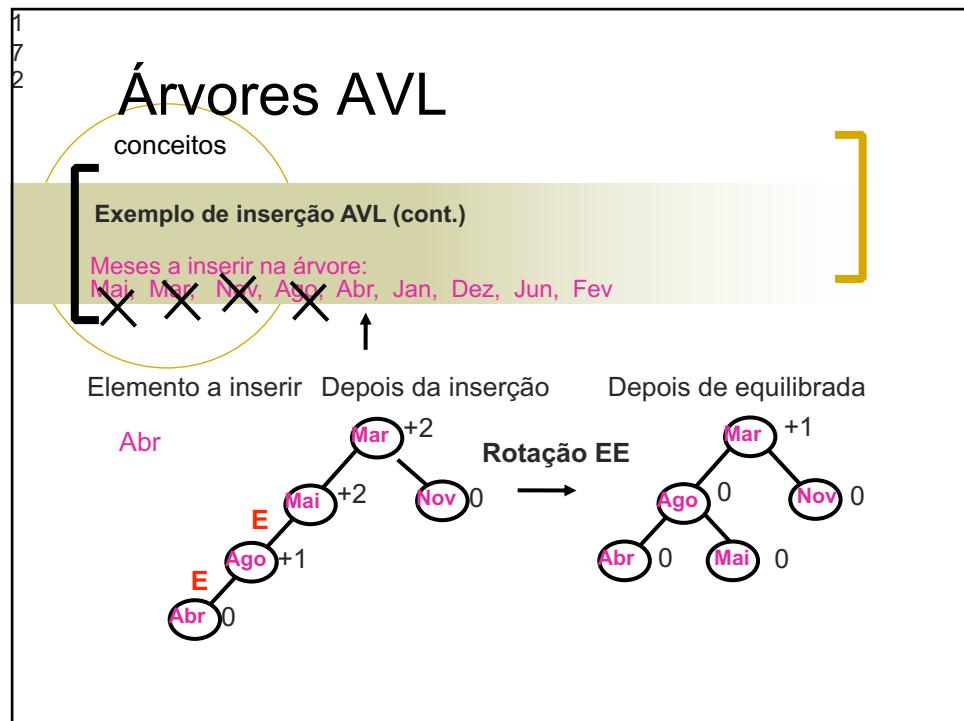
169



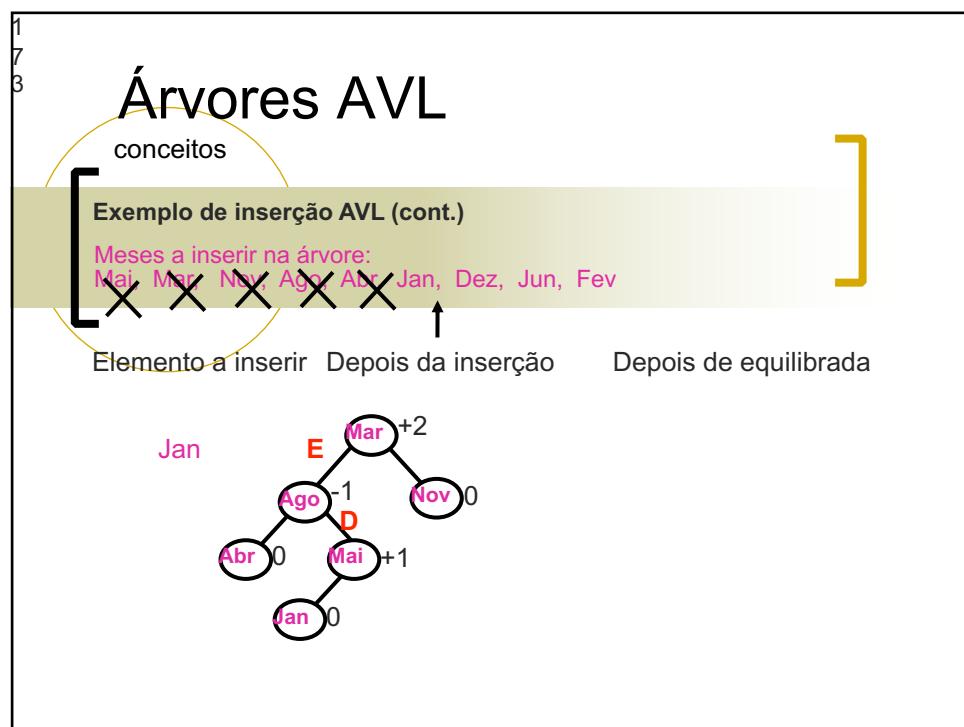
170



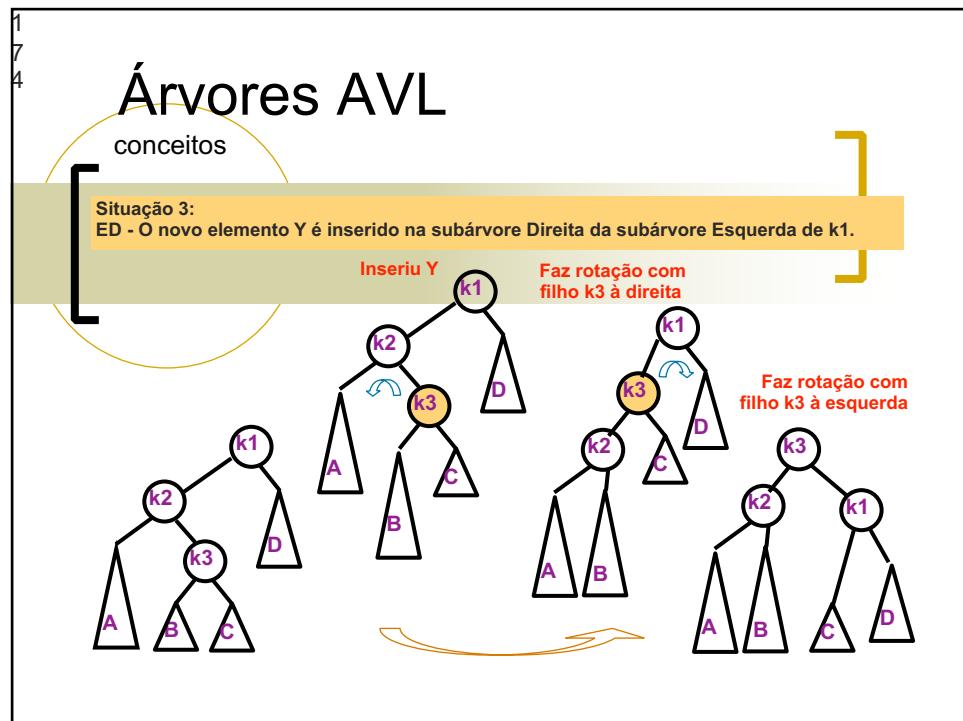
171



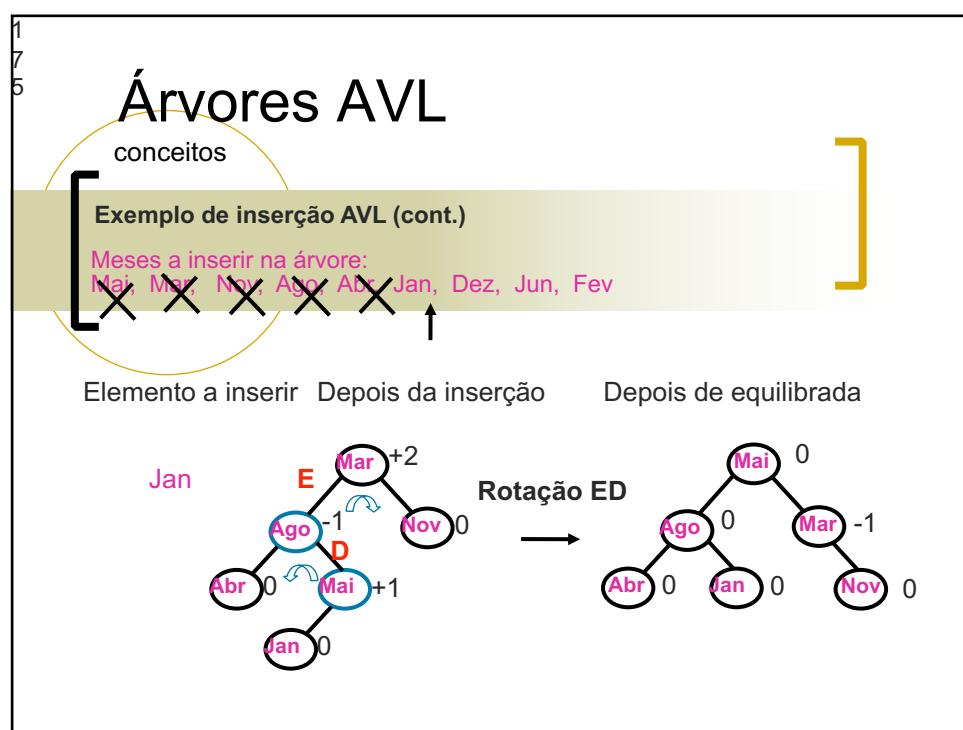
172



173



174



175

1  
7  
6

## Árvores AVL

conceitos

**Exemplo de inserção AVL (cont.)**

Meses a inserir na árvore:  
Mai, Mai, Nov, Ago, Abr, Jan, Dez, Jun, Jun, Fev

Elemento a inserir      Depois da inserção      Depois de equilibrada

```

graph TD
    Mai((Mai)) --> Ago((Ago))
    Mai --> Mar((Mar))
    Ago --> Abr((Abr))
    Ago --> Jan((Jan))
    Mar --> Nov((Nov))
    Jan --> Dez((Dez))
    Jan --> Jun((Jun))
    
```

Dez

Mai  
+1

Ago  
-1

Mar  
-1

Abr  
0

Jan  
+1

Nov  
0

Dez  
0

—

176

1  
7  
7

## Árvores AVL

conceitos

**Exemplo de inserção AVL (cont.)**

Meses a inserir na árvore:  
Mai, Mai, Nov, Ago, Abr, Jan, Dez, Jun, Jun, Fev

Elemento a inserir      Depois da inserção      Depois de equilibrada

```

graph TD
    Mai((Mai)) --> Ago((Ago))
    Mai --> Mar((Mar))
    Ago --> Abr((Abr))
    Ago --> Jan((Jan))
    Mar --> Nov((Nov))
    Jan --> Dez((Dez))
    Jan --> Jun((Jun))
    
```

Jun

Mai  
+1

Ago  
-1

Mar  
-1

Abr  
0

Jan  
0

Nov  
0

Dez  
0

Jun  
0

—

177

1  
7  
8 Árvores AVL  
conceitos

**Exemplo de inserção AVL (cont.)**

Meses a inserir na árvore:  
Mai, Mar, Nov, Ago, Abr, Jan, Dez, Jun, Fev

Elemento a inserir      Depois da inserção      Depois de equilibrada

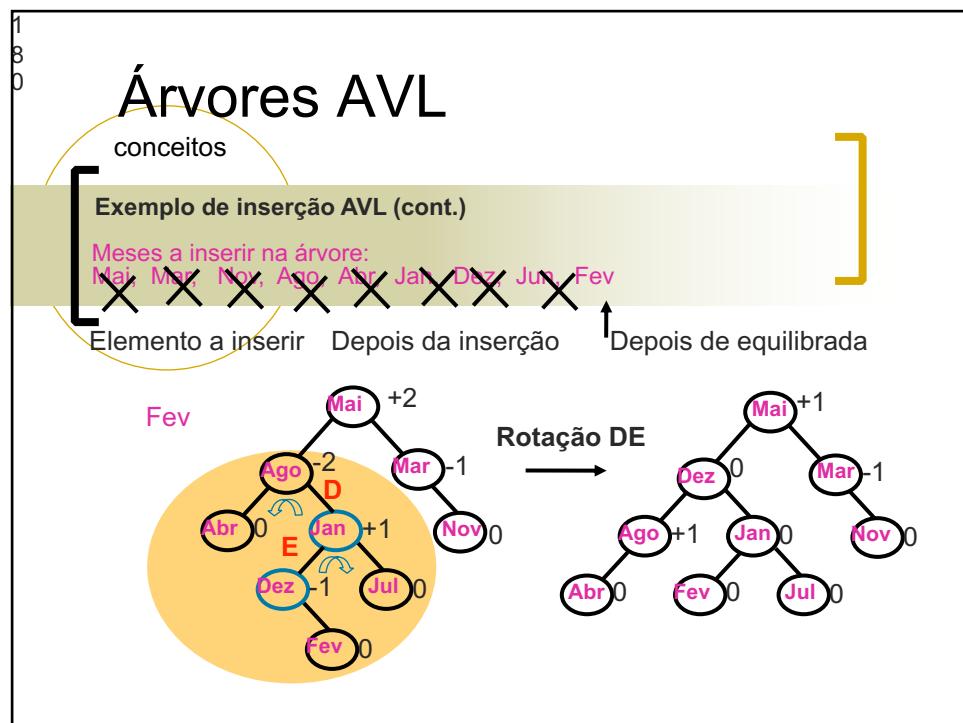
178

1  
7  
9 Árvores AVL  
conceitos

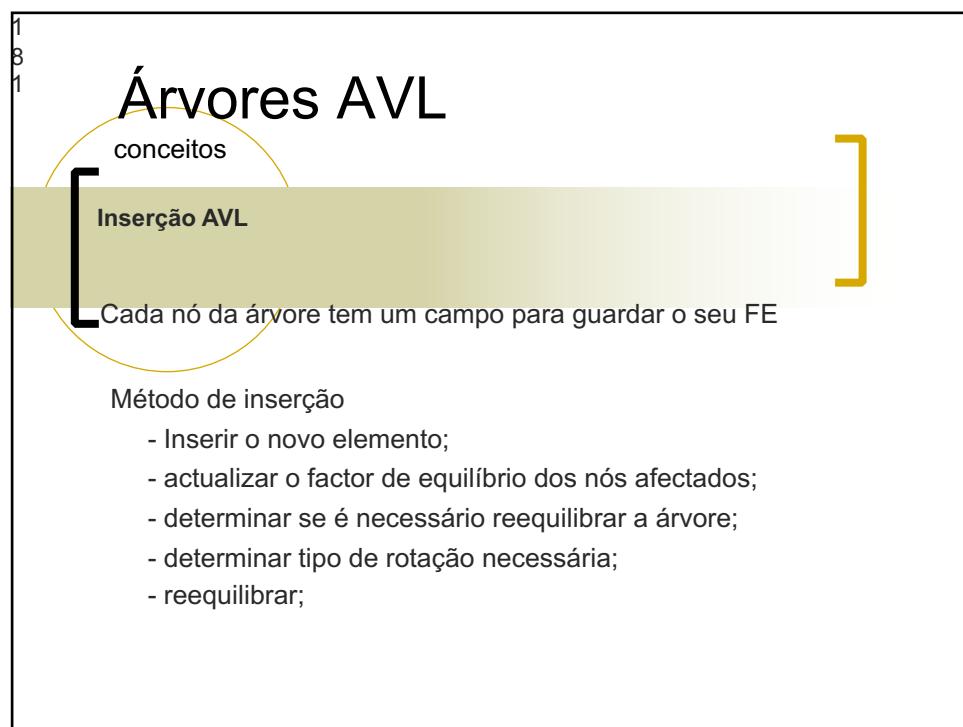
**Situação 3:**  
DE - O novo elemento Y é inserido na subárvore Esquerda da subárvore Direita de k1.

Inseriu Y      Faz rotação com filho k3 à esquerda      Faz rotação com filho k3 à direita

179



180



181

1  
8  
2

## Árvores AVL

implementação

```
package DataStructures;
final class Rotations
{
    static BinaryNode withLeftChild( BinaryNode k1 )
    {
        BinaryNode k2 = k1.left;
        k1.left = k2.right;
        k2.right = k1;
        return k2;
    }
}
```

Faz rotação simples com filho k2 à esquerda

182

1  
8  
3

## Árvores AVL

implementação

```
static BinaryNode withRightChild( BinaryNode k1 )
{
    BinaryNode k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;
    return k2;
}
```

Faz rotação simples com filho k2 à direita

183

1  
8  
4

## Árvores AVL

implementação

```
static BinaryNode doubleWithLeftChild( BinaryNode k1 )
{
    k1.left = withRightChild( k1.left );
    return withLeftChild( k1 );
}
```

Faz rotação com filho k3 à direita

Faz rotação com filho k3 à esquerda

184

1  
8  
5

## Árvores AVL

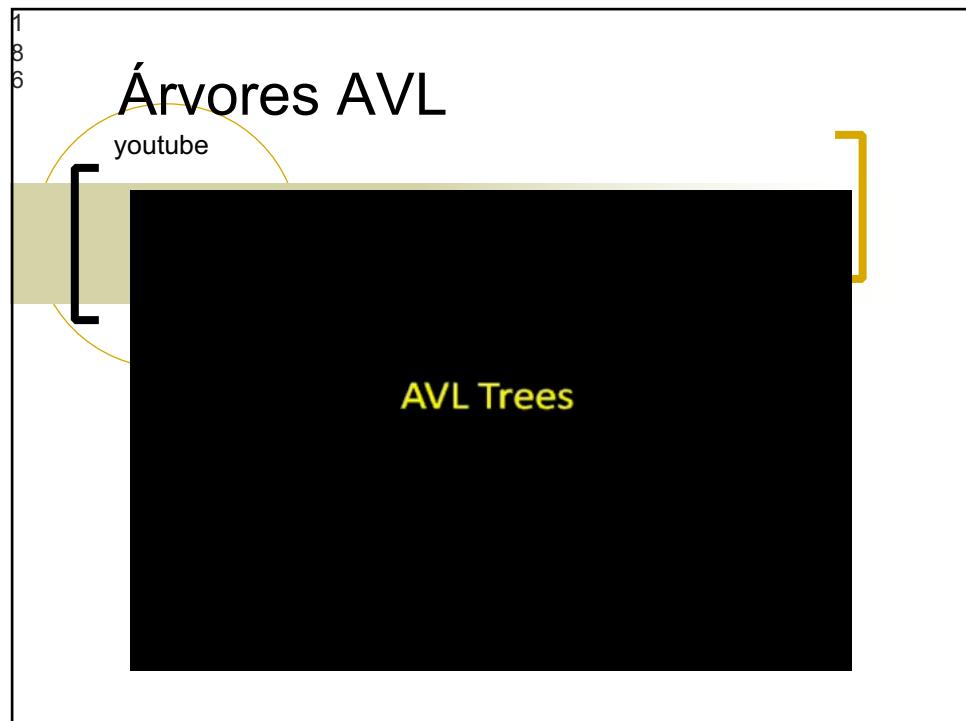
implementação

```
static BinaryNode doubleWithRightChild( BinaryNode k1 )
{
    k1.right = withLeftChild( k1.right );
    return withRightChild( k1 );
}
```

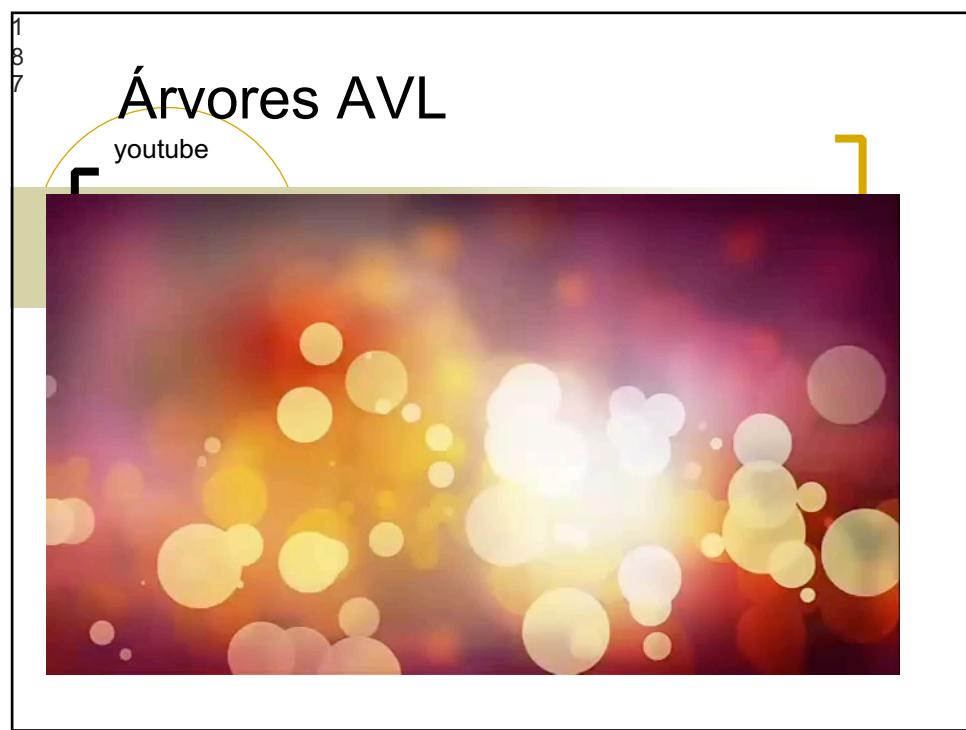
Faz rotação com filho k3 à esquerda

Faz rotação com filho k3 à direita

185



186



187

1  
8  
8

## Árvores AVL

demonstrações na Web

ex. de sequência de teste: 30 50 80 20 15 40 18

<http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html>

188

1  
9  
0

## Árvores Vermelhas e Pretas

conceitos

Uma árvore VP é uma árvore binária de pesquisa em que:

Data Structures & Problem Solving Using JAVA  
Mark Weiss

1. Cada nó é colorido de preto ou vermelho.
2. A raiz é colorida de preto.
3. Se um nó é vermelho os seus filhos são coloridos de preto.
4. Todos os caminhos de cada nó até às suas folhas têm o mesmo número de nós pretos.

190

1  
9  
1

## Árvores Vermelhas e Pretas

conceitos

Complexidade::

Data Structures & Problem Solving Using JAVA  
Mark Weiss

- o Se cada caminho da raiz até ás folhas contém B nós pretos então existirão pelo menos  $2^{B-1}$  nós pretos na árvore.
- o  $h \leq 2B$
- o  $n \geq 2^{B-1}$
- o  $h \leq 2\log(n+1)$
- o Como a raiz tem um nó preto e não podem haver nós vermelhos consecutivos, então a altura máxima de uma árvore VP é no máximo  $2 \log(n+1)$ .
- o Está garantido que a procura numa árvore VP tem complexidade logarítmica !!

191

1  
9  
2

## Árvores Vermelhas e Pretas

conceitos

Inserção::

Data Structures & Problem Solving Using JAVA  
Mark Weiss

- o Os nós são inseridos nas folhas da árvore → um novo nó tem de ter a cor vermelha caso contrário viola 4.
- o Se o nó antecessor for vermelho, a inserção do nó leva à violação de 3.
- o LOGO: vamos ter de fazer **rotações e trocas de cor** nos nós para resolver o conflito 3.

192

1  
9  
8

## Árvores Vermelhas e Pretas

conceitos

Inserção:: O ASCENDENTE É VERMELHO E O SEU IRMÃO PRETO

*inserção na sub árvore esquerda de uma sub árvore esquerda*

193

1  
9  
4

## Árvores Vermelhas e Pretas

conceitos

Inserção:: O ASCENDENTE É VERMELHO E O SEU IRMÃO PRETO

*inserção na sub árvore direita de uma sub árvore direita*

situação simétrica da anterior !!

194

1  
9  
5

## Árvores Vermelhas e Pretas

conceitos

Inserção:: O ASCENDENTE É VERMELHO E O SEU IRMÃO PRETO

*inserção na sub árvore direita de uma sub árvore esquerda*

195

1  
9  
6

## Árvores Vermelhas e Pretas

conceitos

Inserção:: O ASCENDENTE É VERMELHO E O SEU IRMÃO PRETO

*inserção na sub árvore esquerda de uma sub árvore direita*

situação simétrica da anterior !!

196

1  
9  
7

## Árvores Vermelhas e Pretas

conceitos

Inserção:: O ASCENDENTE É VERMELHO E O SEU IRMÃO VERMELHO

nesta situação nem a rotação simples nem a dupla resolvem o problema – em ambas as soluções ficamos com dois nós vermelhos consecutivos !!

VEJAMOS...

# Black = 0

# Black = 1

197

1  
9  
8

## Árvores Vermelhas e Pretas

conceitos

Inserção:: O ASCENDENTE É VERMELHO E O SEU IRMÃO VERMELHO

nesta situação nem a rotação simples nem a dupla resolvem o problema – em ambas as soluções ficamos com dois nós vermelhos consecutivos !!

VEJAMOS...

# Black = 0

# Black = 1

198

1  
9  
9

## Árvores Vermelhas e Pretas

conceitos

Inserção:: O ASCENDENTE É VERMELHO E O SEU IRMÃO VERMELHO

BOM PODÍAMOS FAZER ALGUMA REPINTURA PARA NÃO TER DOIS NÓS VERMELHOS CONSECUTIVOS !!

altera o balanceamento de # de nós pretos em todos os ramos a montante deste nó !!

```

graph TD
    R((R)) --- K((K))
    K --- P((P))
    K --- G((G))
    P --- A[A]
    P --- B[B]
    G --- S((S))
    S --- D[D]
    S --- E[E]
    style P fill:#ff0000,stroke:#000
    style K fill:#ff0000,stroke:#000
    style G fill:#000,stroke:#000
    style S fill:#ff0000,stroke:#000
    style R fill:#000,stroke:#000
    style A fill:#000,stroke:#000
    style B fill:#000,stroke:#000
    style D fill:#000,stroke:#000
    style E fill:#000,stroke:#000
    style G fill:#000,stroke:#000
  
```

199

2  
0  
0

## Árvores Vermelhas e Pretas

conceitos

SOLUÇÃO :: Abordagem TOP-DOWN

200

2  
0  
1

## Árvores Vermelhas e Pretas

conceitos

### SOLUÇÃO :: Abordagem TOP-DOWN

Na travessia da árvore para inserção de um nó garantir que nunca temos os 2 antecedentes directos do nó a inserir vermelhos !

Garantida esta condição:: podemos inserir o novo nó como nova folha da árvore e realizar as operações de rotação como especificadas anteriormente.

SOLUÇÃO:: quando no percurso descendente da árvore encontramos um nó X com os seus descendentes ambos vermelhos pintamos o nó X a vermelho e os seus descendentes directos a preto.

PROBLEMA:: o antecedente de X pode já ser vermelho!

SOLUÇÃO:: aplicamos sobre X e o seu ascendente, ambos a vermelho, as rotações estudadas anteriormente!

201

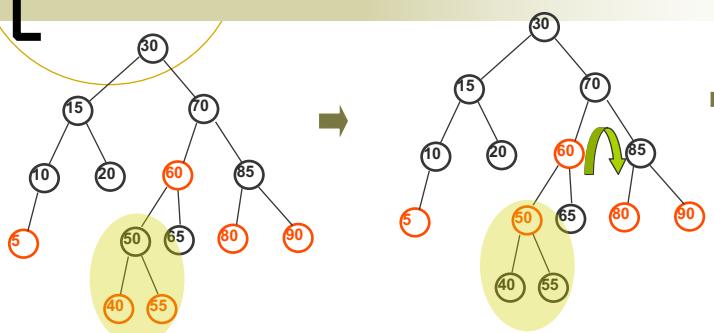
2  
0  
2

## Árvores Vermelhas e Pretas

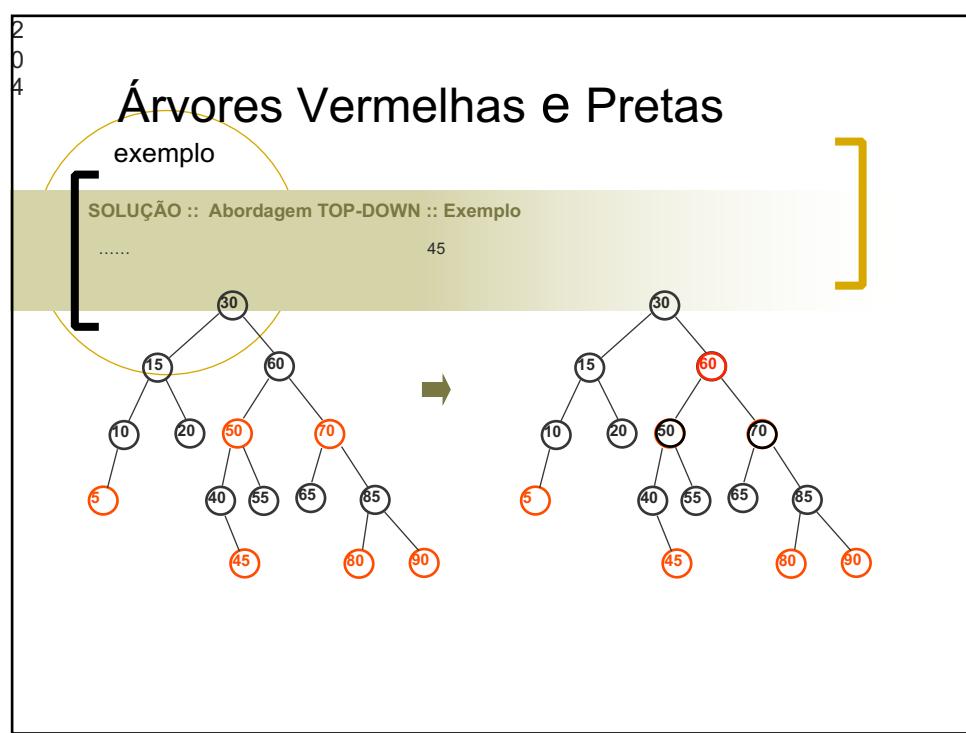
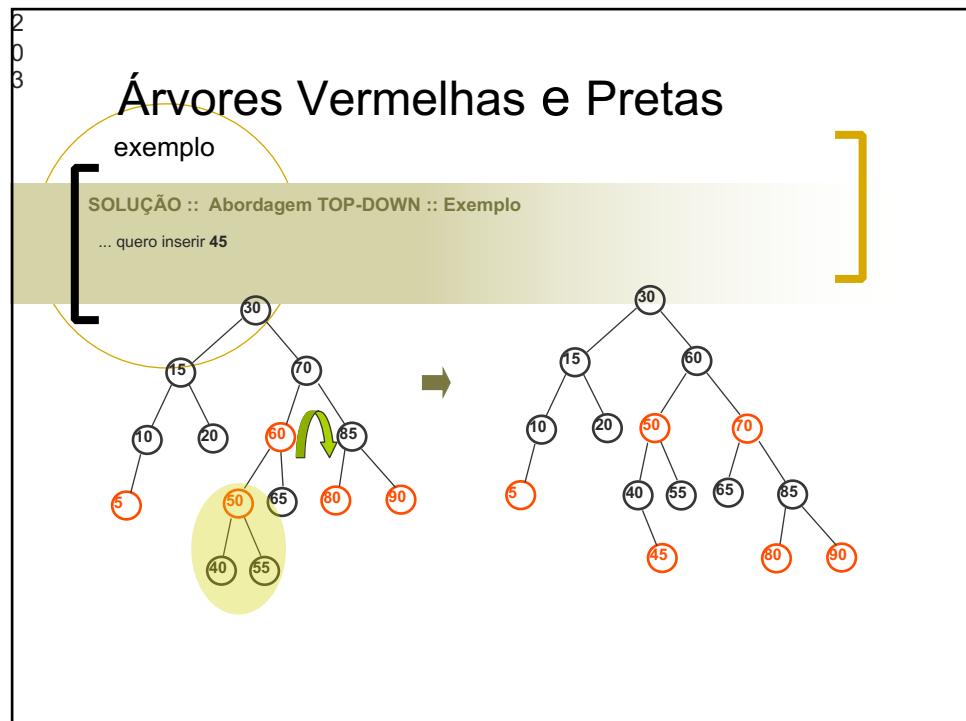
exemplo

### SOLUÇÃO :: Abordagem TOP-DOWN :: Exemplo

... quero inserir 45



202



2  
0  
5

## Árvores Vermelhas e Pretas

exemplo

SOLUÇÃO :: Abordagem TOP-DOWN :: Exemplo

Observações:

- o a árvore resultante é bastante equilibrada.
- o o número de nós atravessados em média durante uma pesquisa é muito semelhante ao que temos na travessia de uma árvore AVL.
- o ... isto embora o método de equilíbrio de uma árvore VP seja mais frágil do que o que temos numa AVL.
- o as vantagens das árvores VP estão num processo de inserção relativamente simples em que na prática poucas rotações são feitas
- o uma desvantagem está em que o processo de eliminação de nós numa árvore VP é pouco eficiente.

205

2  
0  
6

## Árvores Vermelhas e Pretas

conceitos

Eliminação::

Top-down

A remoção é feita sobre nós que são folhas ou só têm um descendente.

A remoção de nós com dois descendentes passa por movimentações de nós.

Se o nó a remover for vermelho – não há problema!

Se o nó a remover for preto (há a violação de 4.)

**SOLUÇÃO:**

assegurar que um nó a remover foi antecipadamente recolorido de vermelho.

Data Structures & Problem Solving Using JAVA  
Mark Weiss

206

2  
0  
7

## Árvores Vermelhas e Pretas

demonstração na Web

Case 1 Case 2 Case 3 Case 4 Case 5

Insert node

- Tree is a valid binary search tree
- All nodes are either red or black
- The root is black
- All leaves are black
- Every red node has two black children
- Every path from root to leaf contain the same number of black nodes

207

2  
0  
8

## Árvores Vermelhas e Pretas

demonstração na Web

A blurred, colorful background image showing a gradient of purple, yellow, and orange with numerous glowing, out-of-focus circular lights, likely representing a bokeh effect or a blurred digital interface.

208

2  
0  
9

## Árvores Vermelhas e Pretas

demonstração na Web

ex. de sequência de teste: 30 50 80 20 15 40 18

<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

<http://cs.armstrong.edu/liang/animation/web/RBTree.html>

209

2  
1  
0

## Árvores Vermelhas e Pretas

... end ;-)



210

[ Algoritmos e Estruturas de Dados ]  
heaps  
■ 2020-2021  
■ Carlos Lisboa Bento

211

[ Heaps ]  
conceitos  
**HEAPS**  
 1. Árvores binárias.  
 2. Nenhum nó tem valor inferior ao dos seus descendentes (max Heap).  
 3. A árvore é perfeitamente equilibrada e os nós no último nível ocupam as posições mais à esquerda.

Não cumpre 2.

Não cumprem 3.

Data Structures and Algorithms in JAVA, Adam Drozdek

212

2  
1  
3

**ÁRVORES PERF. EQUILIBRADAS e ARRAYS**

Data Structures and Algorithms in JAVA, Adam Drozdak

Uma árvore perfeitamente equilibrada pode ser representada por um array segundo a sequência:

- Nós da raiz para as folhas
- Em cada nível da esquerda para a direita
- Ex.: [ 2 8 6 1 10 15 3 12 11 ]

(... esta árvore é uma Heap?)

Temos assim num array HEAP(descendente) de comprimento n (max Heap):

$$\text{heap}[i] \geq \text{heap}[2 \cdot i + 1] \text{ para } 0 \leq i \leq \frac{n-1}{2}$$

e  $\text{heap}[i] \geq \text{heap}[2 \cdot i + 2] \text{ para } 0 \leq i \leq \frac{n-1}{2}$

213

2  
1  
4

**Heaps**  
conceitos

(a)                   (b)                   (c)

NÃO É GARANTIDO ordenamento na Horizontal

Ordenada na Vertical

214

2  
1  
5

# Heaps

conceitos

HEAPS como LISTAS DE PRIORIDADES

```
graph TD; 20[20] --> 15L[15]; 20 --> 15R[15]; 15L --> 8L[8]; 15L --> 10L[10]; 8L --> 2L[2]; 8L --> 5L[5]; 10L --> 6L[6]; 10L --> 7L[7]; 15R --> 13R[13]; 15R --> 14R[14]
```

215

2  
1  
6

# Heaps

conceitos

HEAPS como LISTAS DE PRIORIDADES

Insere 15

```
graph TD; 20[20] --> 10L[10]; 20 --> 15R[15]; 10L --> 8L[8]; 10L --> 7L[7]; 8L --> 2L[2]; 8L --> 5L[5]; 7L --> 6L[6]; 15R --> 13R[13]; 15R --> 14R[14]
```

(a)

Data Structures and Algorithms in JAVA, Adam Drozdek

216

2  
1  
7

## Heaps

conceitos

**HEAPS como LISTAS DE PRIORIDADES**

**Insere 15**

(a)

(b)

(c)

Data Structures and Algorithms in JAVA, Adam Drozdek

217

2  
1  
8

## Heaps

conceitos

**HEAPS como LISTAS DE PRIORIDADES**

**Insere 15**

(a)

(b)

(c)

Data Structures and Algorithms in JAVA, Adam Drozdek

218

2  
1  
9

## Heaps

conceitos

**HEAPS como LISTAS DE PRIORIDADES**

Insere 15

Data Structures and Algorithms in JAVA, Adam Drozdek

219

2  
2  
0

## Heaps

implementação

**HEAPS como LISTAS DE PRIORIDADES**

Inserção numa HEAP definida como LISTA DE PRIORIDADES

```

heapEnqueue(el)
    put el at the end of heap;
    while el is not in the root and el > parent(el)
        swap el with its parent;
  
```

220

2  
2  
1

## Heaps

conceitos

**HEAPS como LISTAS DE PRIORIDADES**

Eliminação numa HEAP definida como LISTA DE PRIORIDADES

(a)

221

2  
2  
2

## Heaps

conceitos

**HEAPS como LISTAS DE PRIORIDADES**

Eliminação numa HEAP definida como LISTA DE PRIORIDADES

(a)

(b)

222

2  
2  
3

## Heaps

conceitos

**HEAPS como LISTAS DE PRIORIDADES**

Eliminação numa HEAP definida como LISTA DE PRIORIDADES

The diagram shows a binary heap represented as a tree with root 20. The tree has nodes: 20 (root), 10, 15, 8, 7, 13, 14, 2, 5, 6. A red arrow labeled "dequeue" points from the root 20 to its left child 10. In (a), node 20 is highlighted. In (b), node 10 is highlighted. In (c), node 15 is highlighted. The final state (d) shows the heap after the deletion of node 20.

(a)

(b)

(c)

(d)

223

2  
2  
4

## Heaps

conceitos

**HEAPS como LISTAS DE PRIORIDADES**

Eliminação numa HEAP definida como LISTA DE PRIORIDADES

The diagram shows a binary heap represented as a tree with root 20. The tree has nodes: 20 (root), 10, 15, 8, 7, 13, 14, 2, 5, 6. A red arrow labeled "dequeue" points from the root 20 to its left child 10. In (a), node 20 is highlighted. In (b), node 10 is highlighted. In (c), node 15 is highlighted. In (d), node 15 has become the new root, with 6 as its left child and 14 as its right child. The final state (d) shows the heap after the deletion of node 20.

(a)

(b)

(c)

(d)

224

2  
2  
5

## Heaps

implementação

**HEAPS como LISTAS DE PRIORIDADES**

Eliminação numa HEAP definida como LISTA DE PRIORIDADES

```

heapDequeue()
extract the element from the root;
put the element from the last leaf in its place;
remove the last leaf;
// both subtrees of the root are heaps;
p = the root;
while p is not a leaf and p < any of its children
    swap p with the larger child;

```

225

2  
2  
6

## Heaps

conceitos

**ARRAYS organizados como HEAPS (abordagem top-down) "heapify"**

226

2  
2  
7

## Heaps

conceitos

ARRAYS organizados como HEAPS (abordagem top-down) “heapify”

$$\sum_{k=1}^n \lfloor \lg k \rfloor \leq \sum_{k=1}^n \lg k = \lg 1 + \cdots + \lg n = \lg(1 \cdot 2 \cdot \cdots \cdot n) = \lg(n!) = O(n \lg n)$$

227

2  
2  
8

## Heaps

aplicações

- o Filas de Prioridades
- o Gestor de eventos (ex. jogos)
- o Ordenamento Heap (Heap Sort)
- o Algoritmo de Dijkstra (caminho mais curto)  
 $O(n^*m) \Rightarrow O(m \lg n)$  n # vertices, m # arcos

228



229



230

**Algoritmos e Estruturas de Dados**

equilíbrio de árvores (... RE-VISITA)

■ 2020-2021

■ Carlos Lisboa Bento

231

2  
3  
2

## Equilíbrio de árvores

TRÊS técnicas para manter uma árvore binária equilibrada:

- **equilíbrio dinâmico** : depois de cada inserção reequilibrar a árvore de forma que inserção e procura decorram em tempo logarítmico  
 >>> AVL e VPS
- “randomização” : se os elementos fossem inseridos depois de baralhados (ordem de inserção “randomizada”) tenderíamos para uma árvore equilibrada!
- **Desafio: garantir este efeito mas sem obrigar a ter os elementos todos disponíveis para inserção à partida**  
 >>> TREAPS com prioridade aleatória (árvores aleatórias)
- **amortização** : em vez de re-equilibrar somente quando faço uma inserção, podemos também fazer re-equilíbrio como efeito lateral da pesquisa (amortizamos o custo do re-equilíbrio)      >>> splay trees

232

**Algoritmos e Estruturas de Dados**  
 equilíbrio de árvores  
 PRELIMINAR: inserção de nós pela raiz

■ 2019-2020

■ Carlos Lisboa Bento

233

2  
3  
4

**ABPs com inserção na raiz**

conceitos

Inserção na raiz... em vez de nas folhas como vimos antes

Temos

?

234

2  
3  
5

## ABPs com inserção na raiz

conceitos

Solução: começar por levar o nó até às folhas e trazê-lo para a raiz através de rotações sucessivas !!

235

2  
3  
6

## ABPs com inserção na raiz

conceitos

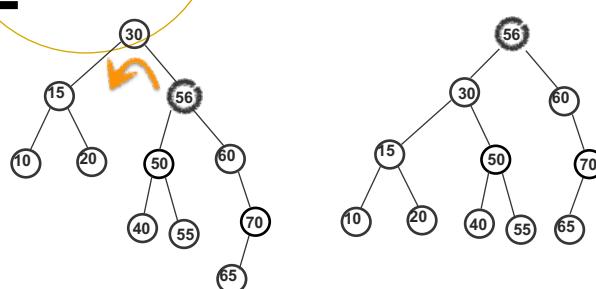
Solução: começar por levar o nó até às folhas e trazê-lo para a raiz através de rotações sucessivas !!

236

2  
3  
7

## ABPs com inserção na raiz

Solução: começar por levar o nó até às folhas e trazê-lo para a raiz através de rotações sucessivas !!

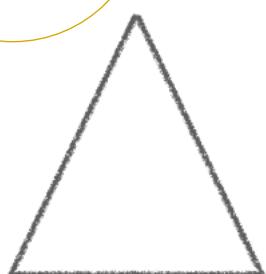


237

2  
3  
8

## ABPs com inserção na raiz

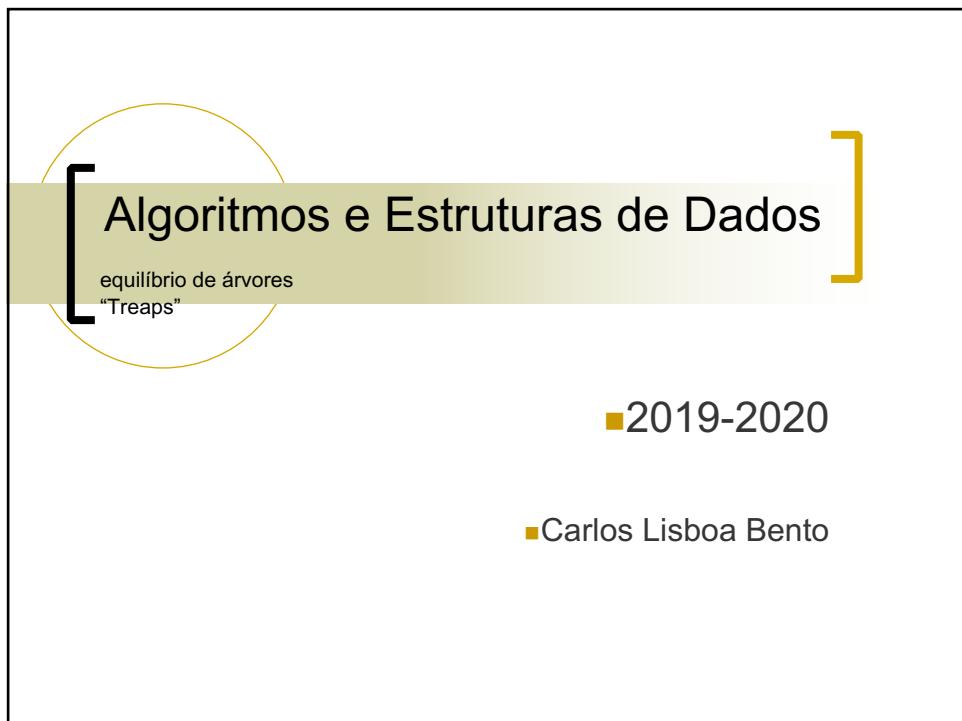
... mas afinal porque ter este trabalho em vez de fazer sempre a inserção por nós inferiores na árvore??



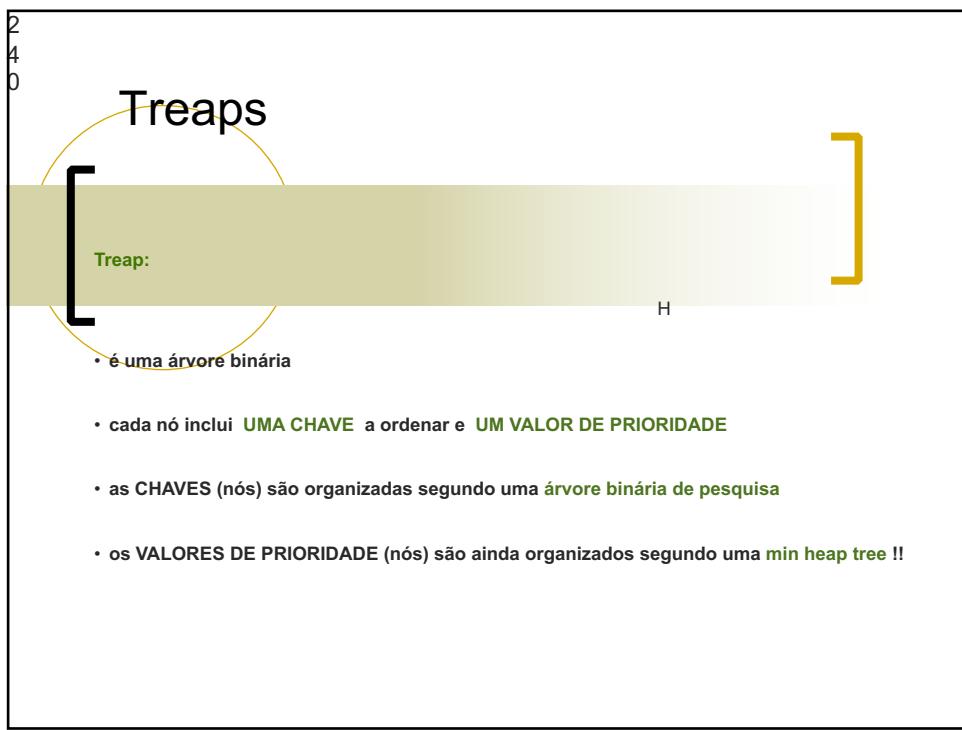
*nos mais recentemente inseridos junto à raiz*

... corresponde a necessidades comuns de pesquisa... as últimas ordens de serviço são aqueles que tendencialmente vão ser mais acedidas; os subscriptores mais recentes de um serviço tendencialmente são os que mais o vão usar; etc...

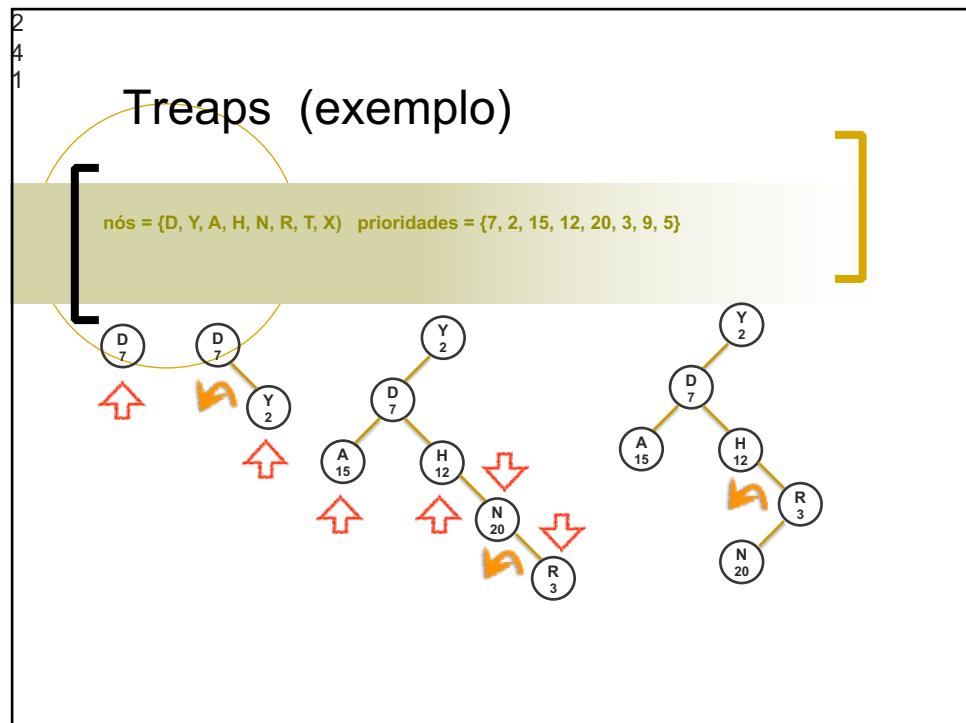
238



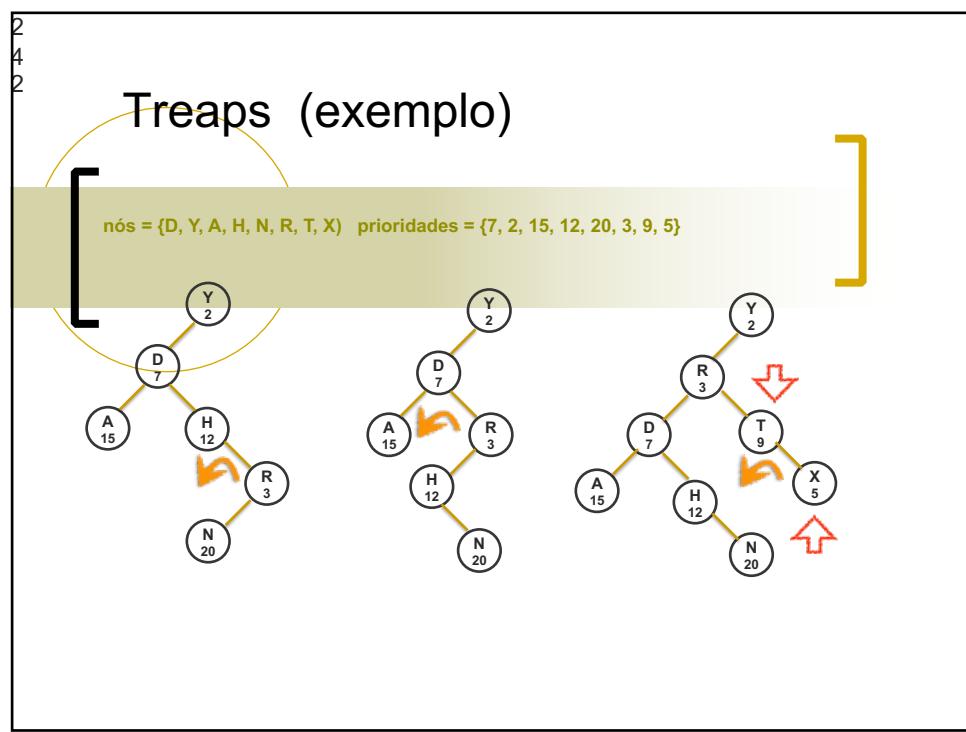
239



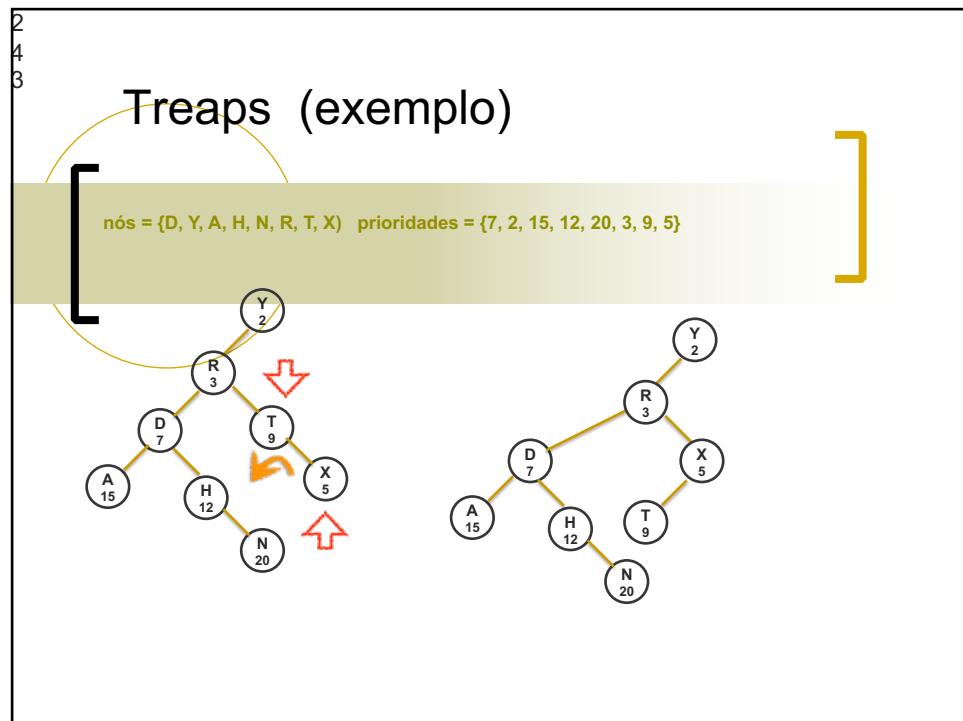
240



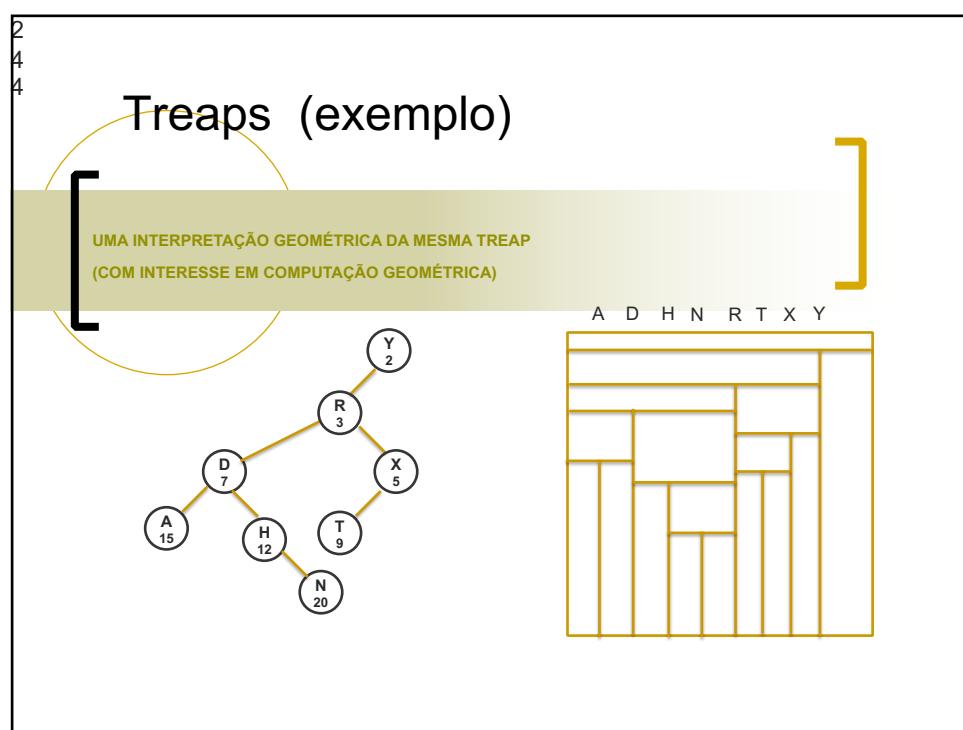
241



242



243



244

2  
4  
5

## Treaps (exemplo)

ESTAS TREAPS SÃO DESIGNADAS POR TREAPS DE PRIORIDADE FIXA:

QUESTÃO:

O QUE TENDENCIALMENTE ACONTECE SE À MEDIDA QUE OS ELEMENTOS SEJAM DISPONIBILIZADOS PARA INSERÇÃO NA TREAP A PRIORIDADE ASSOCIADA A CADA NÓ FOR GERADA ALEATORIAMENTE (PODE ASSUMIR QUE O GERADOR NÃO PRODUZ VALORES REPETIDOS) ?

245

2  
4  
6

## Treaps (exemplo)

246

**Algoritmos e Estruturas de Dados**

equilíbrio de árvores  
“Treaps” com prioridades aleatórias (árvores aleatórias)

■ 2019-2020

■ Carlos Lisboa Bento

247

2  
4  
8

TREAPs com prioridades aleatórias  
(árvores aleatórias)

SOLUÇÃO

- as prioridades associadas aos nós são variáveis continuas aleatórias uniformemente distribuídas compreendendo valores reais entre 0 e 1 (não são gerados valores iguais).

ALTERNATIVA

- para cada nó gerar uma probabilidade (com distribuição uniforme) valores reais entre 0 e 1
- para cada nó já inserido atribuir uma prioridade entre 0 e 1 igual a  $1/(n+1)$  em que  $n$  representa o tamanho de cada nó (número de descendentes desse nó mais ele próprio)

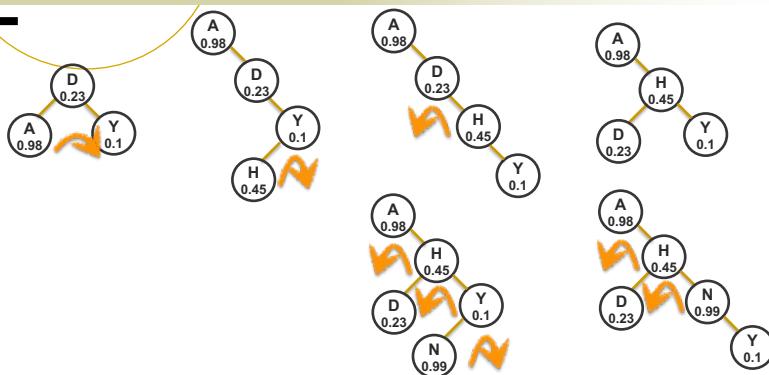
248

# ABP aleatórias

## exemplo

Exemplo: cada nó com a probabilidade de estar na raiz como um valor aleatório 0..1 uniformemente distribuído.

nós = {D, Y, A, H, N, R, T, X} probabilities = {0.23, 0.1, 0.98, 0.45, 0.99, 0.15, 0.3, 0.2}



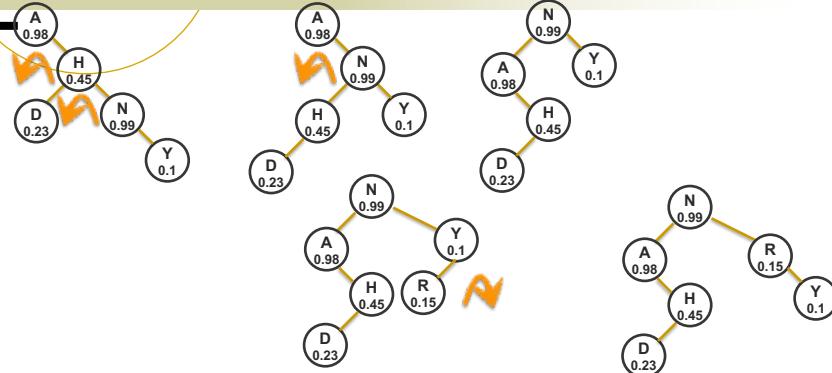
249

# ABP aleatórias

## exemplo

Exemplo: cada nó com a probabilidade de estar na raiz como um valor aleatório 0.1 uniformemente distribuído.

nós = {D, Y, A, H, N, R, T, X} probabilities = {0.23, 0.1, 0.98, 0.45, 0.99, 0.15, 0.3, 0.2}



250

2  
5  
1

## ABP aleatórias

exemplo

**Exemplo: cada nó com a probabilidade de estar na raiz como um valor aleatório 0..1 uniformemente distribuído.**

nós = {D, Y, A, H, N, R, T, X} probabilities = {0.23, 0.1, 0.98, 0.45, 0.99, 0.15, 0.3, 0.2}

251

## ABP aleatórias

DEMO

**Exemplo: cada nó com a probabilidade de estar na raiz como um valor aleatório 0..1 uniformemente distribuído.**

**Randomized Binary Search Trees**

A **map** is a binary search tree in which each node has both a key and a priority: nodes are ordered in an inorder fashion by their keys (as in a standard binary search tree) and are heap-ordered by their priorities (so that the each parent has a higher priority than its children). To implement randomized binary search trees, random priorities are assigned to each node when an insertion is performed. **Insertion** uses the standard algorithm for inserting the new node as a leaf and then restores heap ordering by "bubbling up" the node through a sequence of rotations. **Deletion** reverses this procedure by first "bubbling down" the node until it is a leaf, and then pruning off the tree. One can show that the expected depth of any node in this tree is about  $\ln n$ , and the expected number of rotations per insertion or deletion is about 2.

The following demo starts with a sequence of 12 insertions in a "worst case" (increasing) order. Priorities are entirely random.

**Tree + Heap = TREAP**

<http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/Treap-Example.html>

252

2  
5  
3

## TREAPs com prioridades aleatórias (árvores aleatórias)

**SOLUÇÃO**

- as prioridades associadas aos nós são variáveis contínuas aleatórias uniformemente distribuídas compreendendo valores reais entre 0 e 1 (não são gerados valores iguais).

**ALTERNATIVA**

- para cada nó gerar uma probabilidade (com distribuição uniforme) valores reais entre 0 e 1
- para cada nó já inserido atribuir uma prioridade entre 0 e 1 igual a  $1/(n+1)$  em que  $n$  representa o tamanho de cada nó (número de descendentes desse nó mais ele próprio)

253

2  
5  
6

## PPTs :)

Árvores remoção de

ABP aleatórias exemplo

Exemplo: cada nó com a probabilidade de nós = {D, Y, B, A, N, R, T, X} probabilities = {0.1, 0.2, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1}

D  $1/(1+1)$  Y  $1/(2+1)$   
 D  $1/(1+1)$  B  $1/(1+2)$   
 B  $1/(1+1)$

Heaps conceitos

ÁRVORES PERF. EQUILIBRADAS e ARRAYS

Uma árvore perfeitamente equilibrada pode ser representada por um array segundo a sequência:

- Nós da raiz para as folhas
- Em cada nível da esquerda para a direita

Ex: [2 8 6 6 10 15 3 12 11] (... esta árvore é uma Heap!)

Temos assim num array HEAP(i) os descendentes de comprimento n (max Heap):

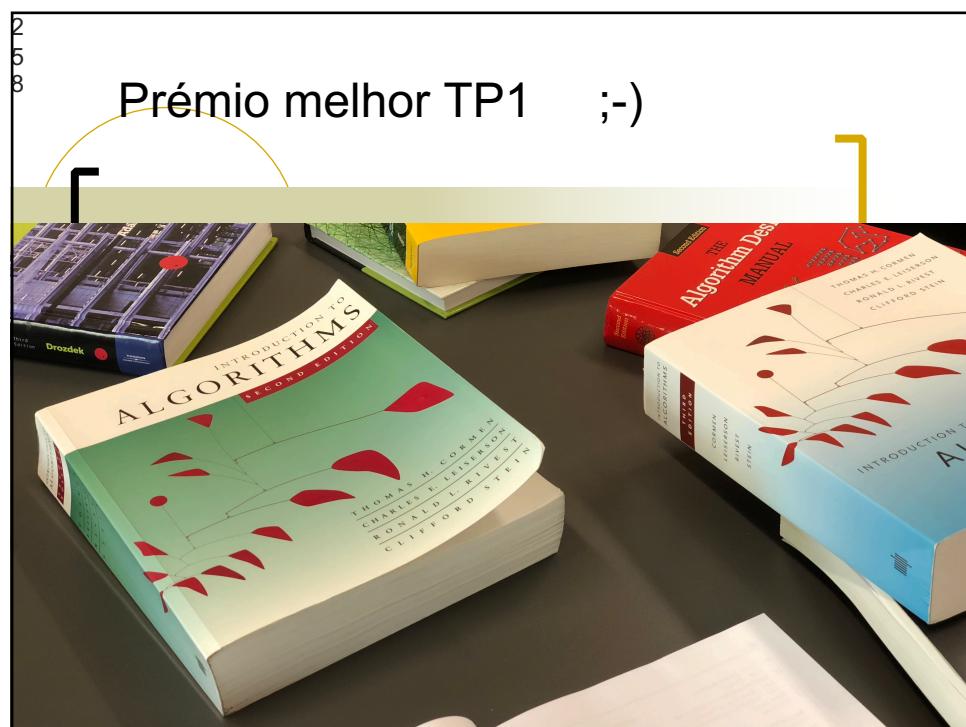
$$\text{heap}[i] \rightarrow \text{heap}[2 \cdot i + 1] \text{ para } 0 \leq i \leq \frac{n-1}{2}$$

$$\text{e } \text{heap}[i] \rightarrow \text{heap}[2 \cdot i + 2] \text{ para } 0 \leq i \leq \frac{n-1}{2}$$

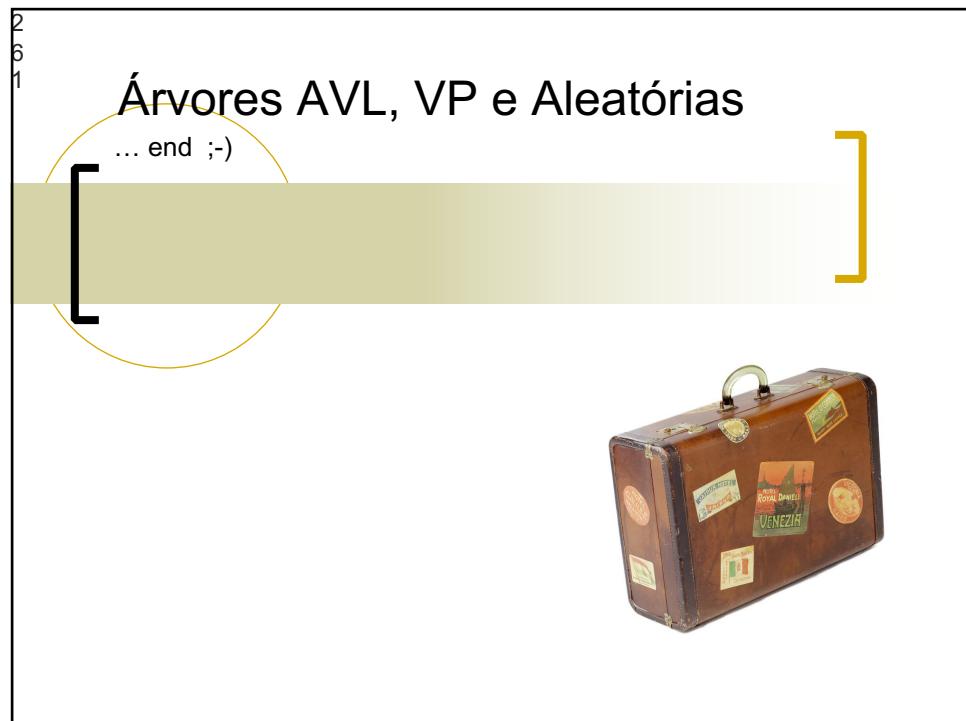
256



257



258



261



262

2  
6  
8

## Equilíbrio de árvores

**TRÊS** técnicas para manter uma árvore binária equilibrada:

- **equilíbrio dinâmico** : depois de cada inserção reequilibrar a árvore de forma que inserção e procura decorram em tempo logarítmico

>>> AVL e VPs

- “randomização” : se os elementos fossem inseridos depois de baralhados (ordem de inserção “randomizada”) tenderíamos para uma árvore equilibrada! Desafio: garantir este efeito mas sem obrigar a ter os elementos todos disponíveis para inserção à partida

>>> TREAPS com prioridade aleatória (árvores aleatórias)

- **amortização** : em vez de re-equilibrar somente quando faço uma inserção, podemos também fazer re-equilíbrio como efeito lateral da pesquisa (amortizamos o custo do re-equilíbrio)

>>> splay trees

263

## Algoritmos e Estruturas de Dados

Splay Trees

■ 2019-2020

■ Carlos Lisboa Bento

264

2  
6  
5

## Splay Tree

**Splay Tree:**

- é uma árvore binária auto-ajustada
- os elementos recentemente inseridos vão ocupar posições próximo da raiz (são mais rapidamente acedidos)
- as operações de inserção, consulta e remoção são realizadas em tempo amortizado  $O(\log N)$
- as splay trees incorporam uma operação de SPLAYING que re-estrutura a árvore e coloca o elemento em foco na raiz da árvore

265

2  
6  
6

## Splay Tree (INSERÇÃO)

**INSERÇÃO de um valor  $x$ :**

- insere na árvore como se fosse uma ABP normal
- aplica sobre o nó inserido a operação de SPLAYING

Alternativa (usando as operações de SPLIT e JOIN):

- faz um splitting na árvore em volta do valor de  $x$  criando uma sub-árvore S e uma T
- cria a nova árvore com  $x$  na raiz em que S é a sua sub-árvore esquerda e T a sua sub-árvore direita

266

2  
6  
7

## Splay Tree (CONSULTA)

**CONSULTA de um valor  $x$ :**

- percorre a árvore como qualquer árvore binária de pesquisa até encontrar o nó
- aplica sobre o nó consultado a operação de SPLAYING

267

2  
6  
8

## Splay Tree (SPLAYING)

considerar o nó a inserir  $x$  c/ pai  $p$  e c/ avó  $g$ :

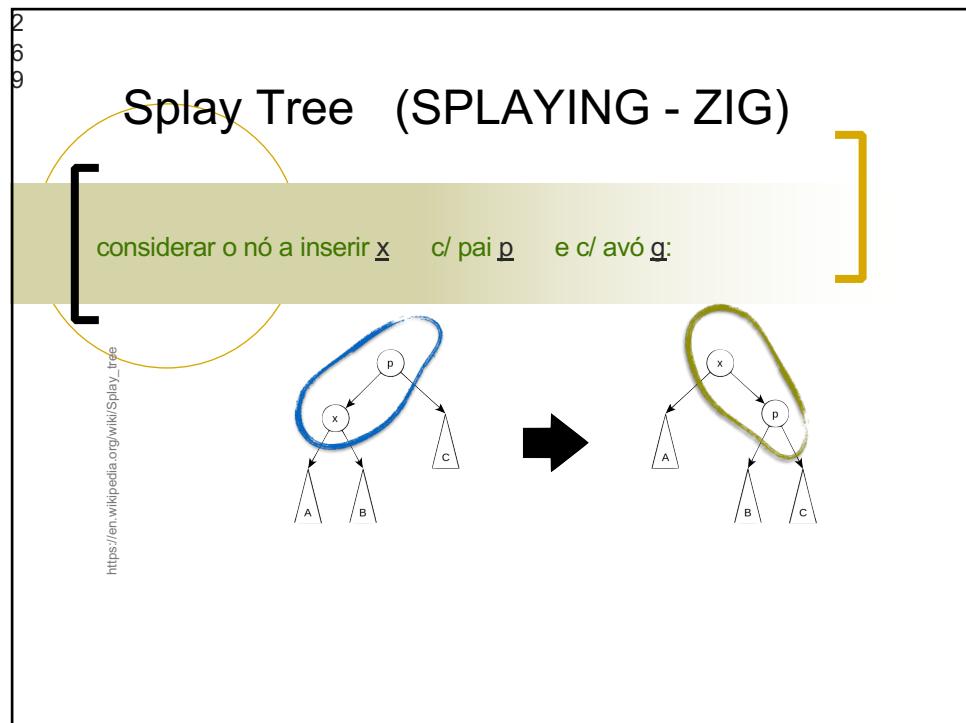
**Temos 3 tipos de passos de splaying**  
(cada um com respectivo simétrico, aqui é apresentado uma das versões)

**ZIG**

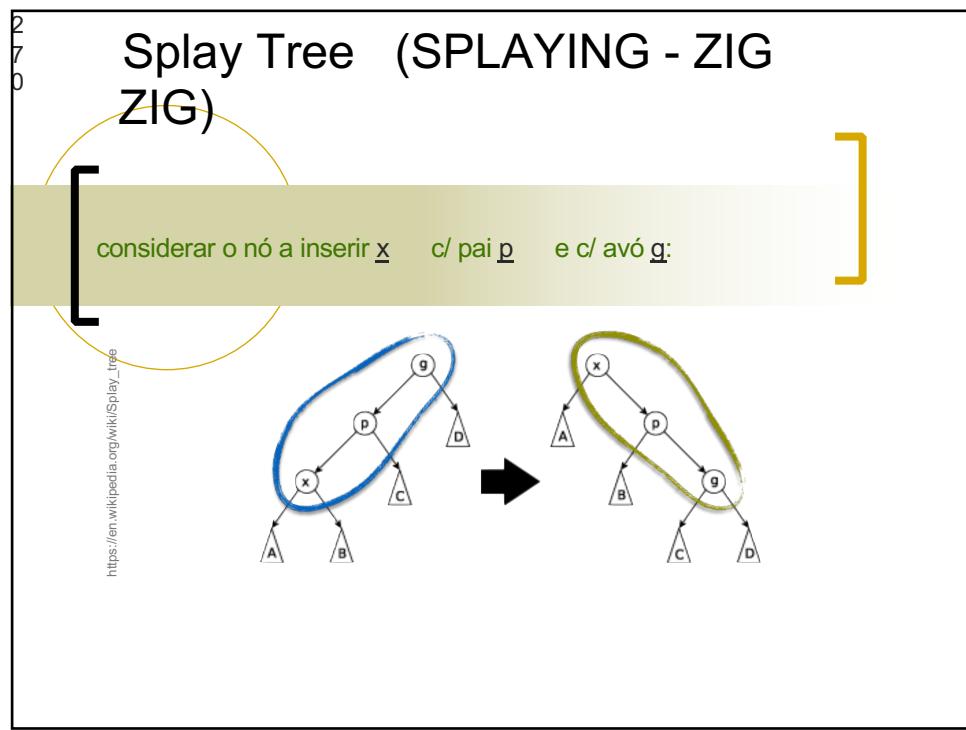
**ZIG ZIG**

**ZIP ZAG**

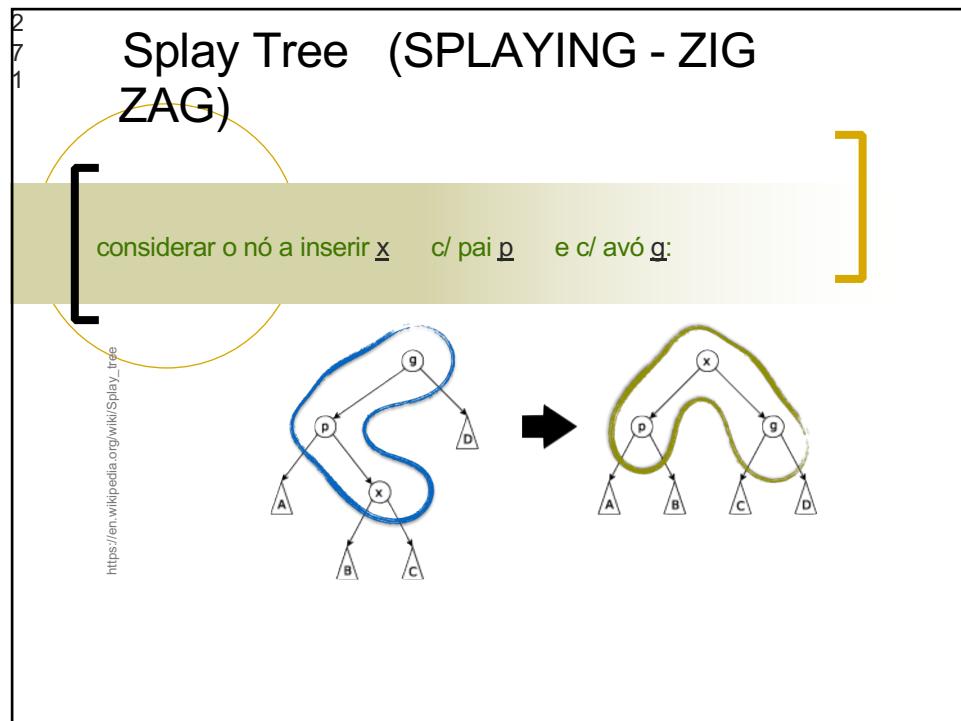
268



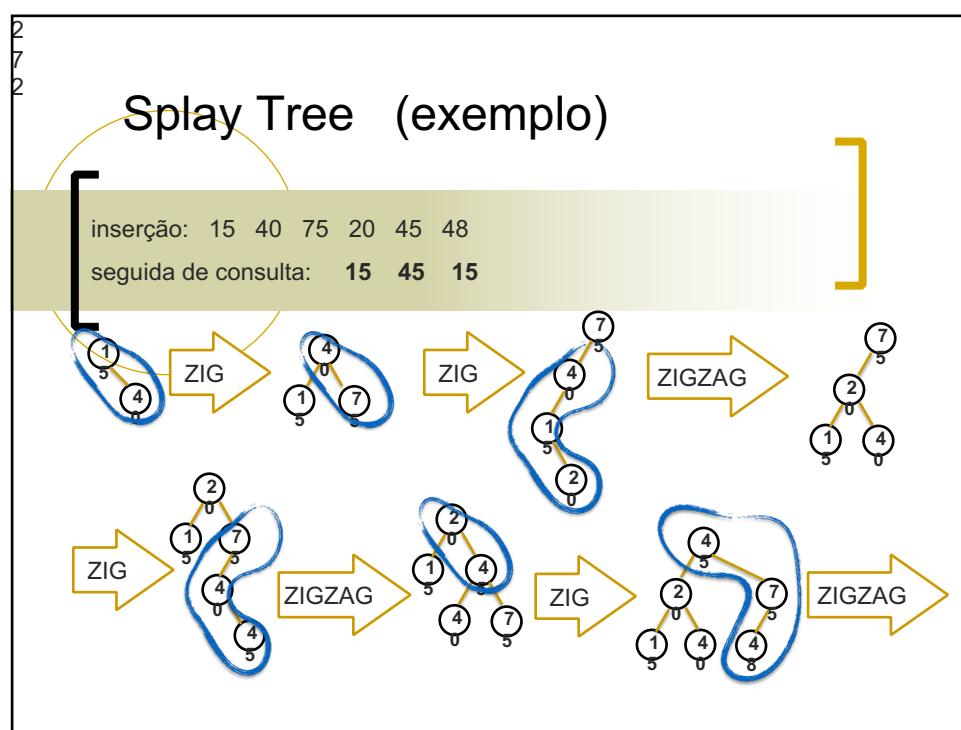
269



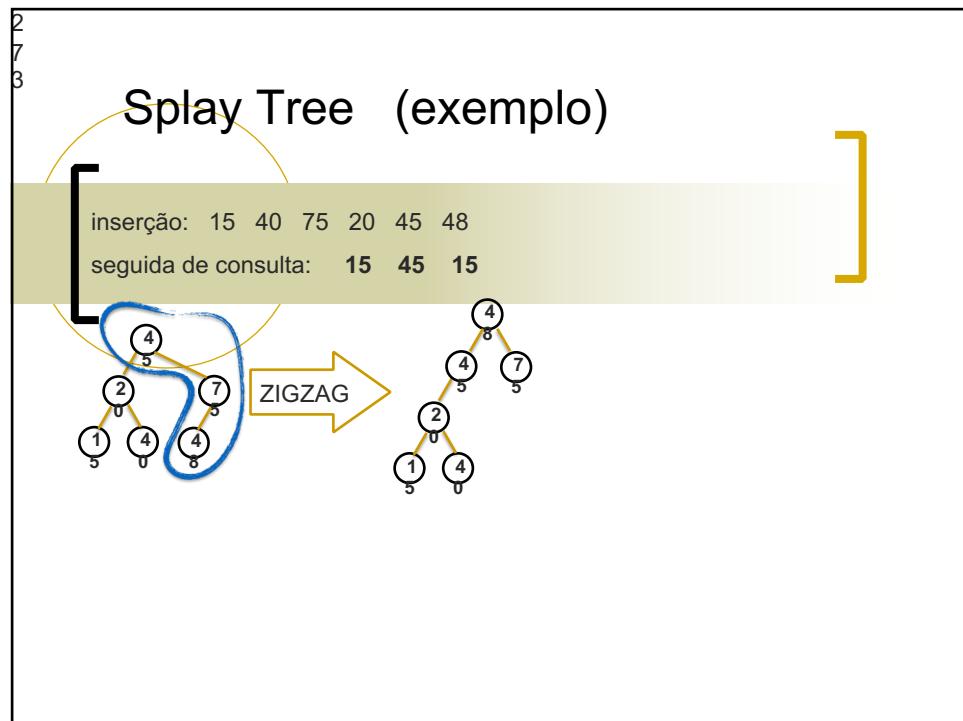
270



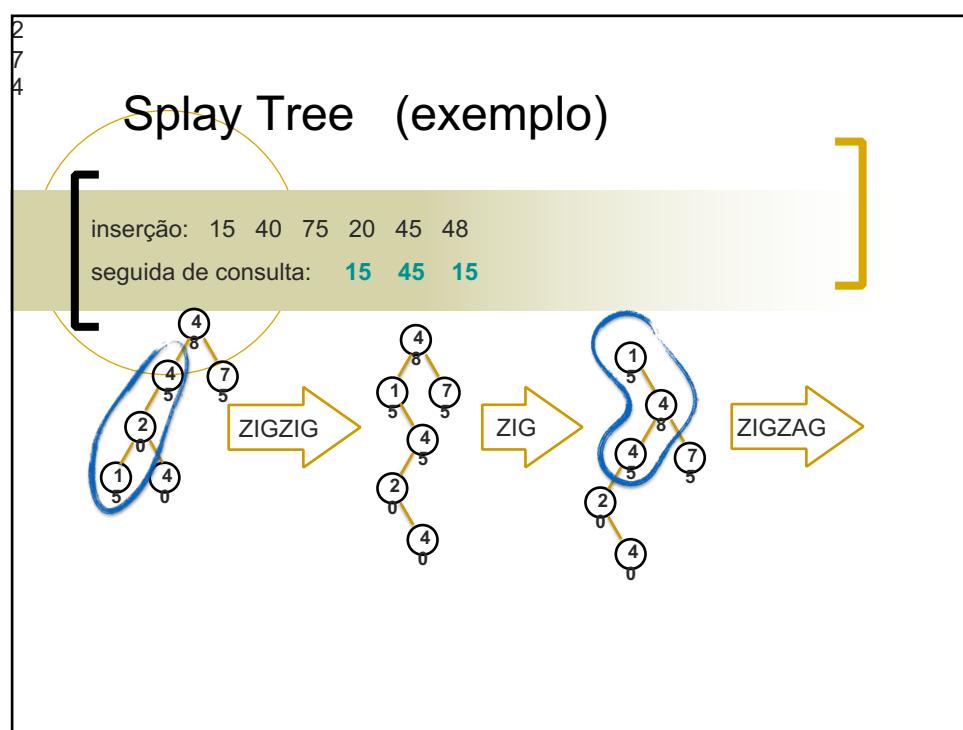
271



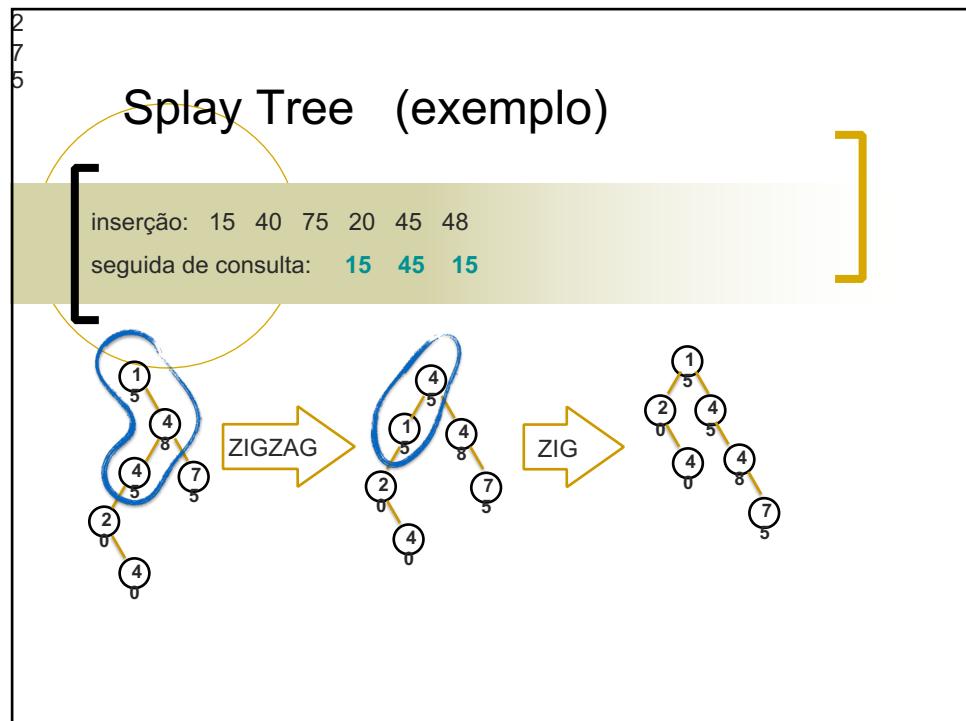
272



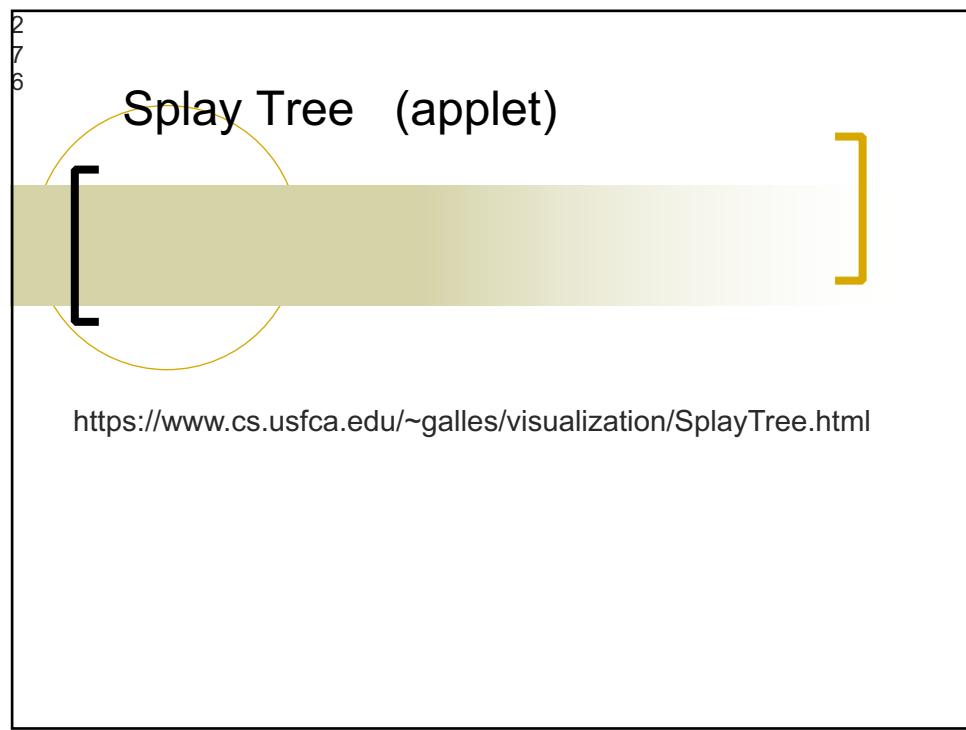
273



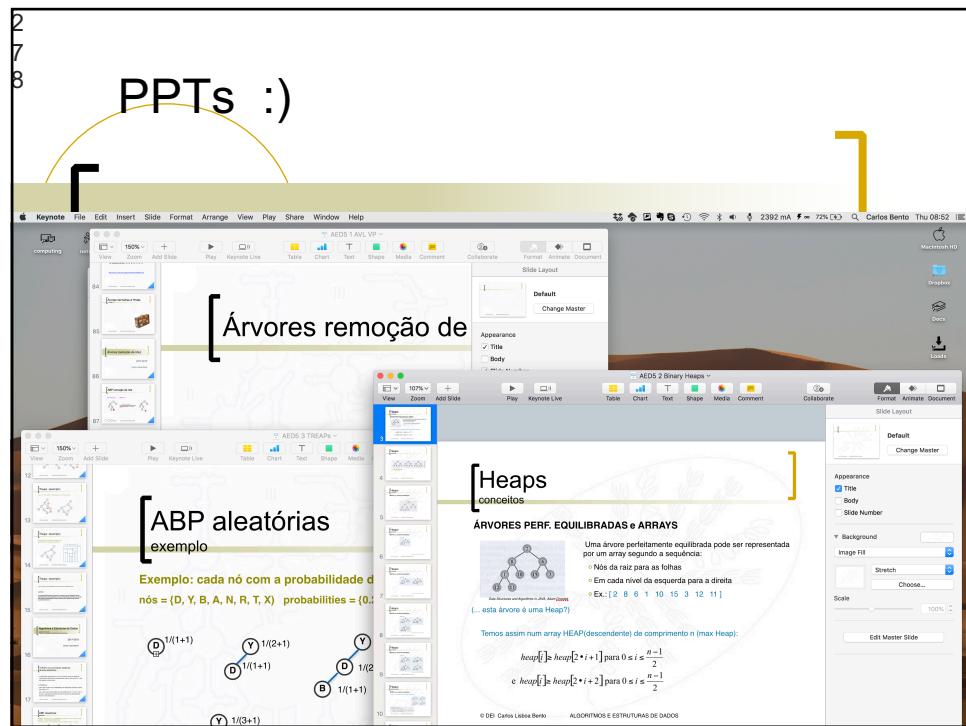
274



275



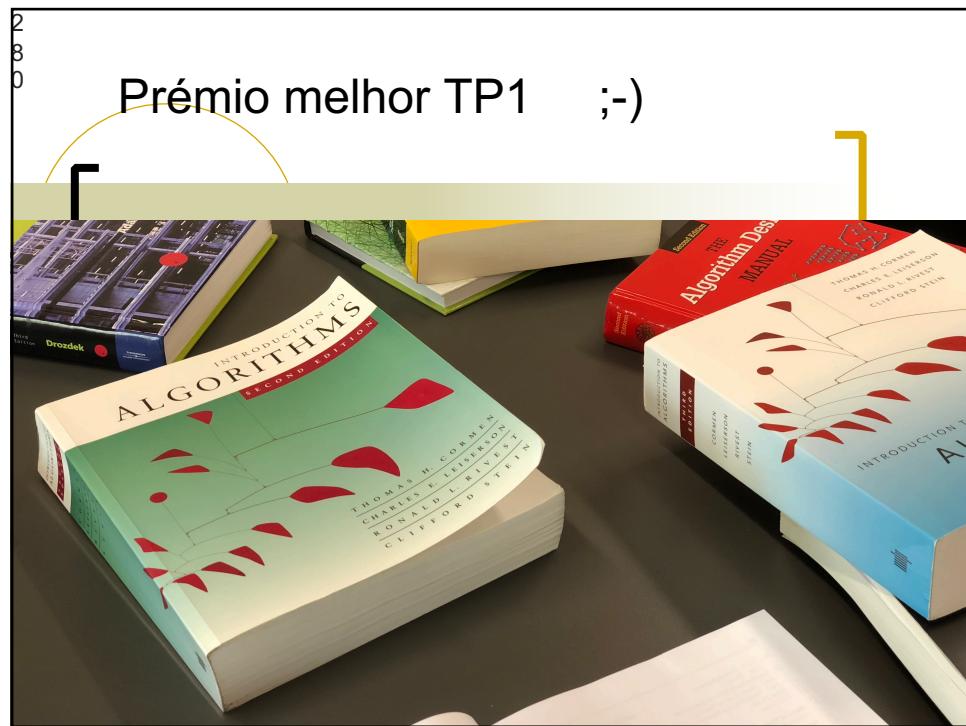
276



278



279



## [ Árvores (remoção de nós) ]

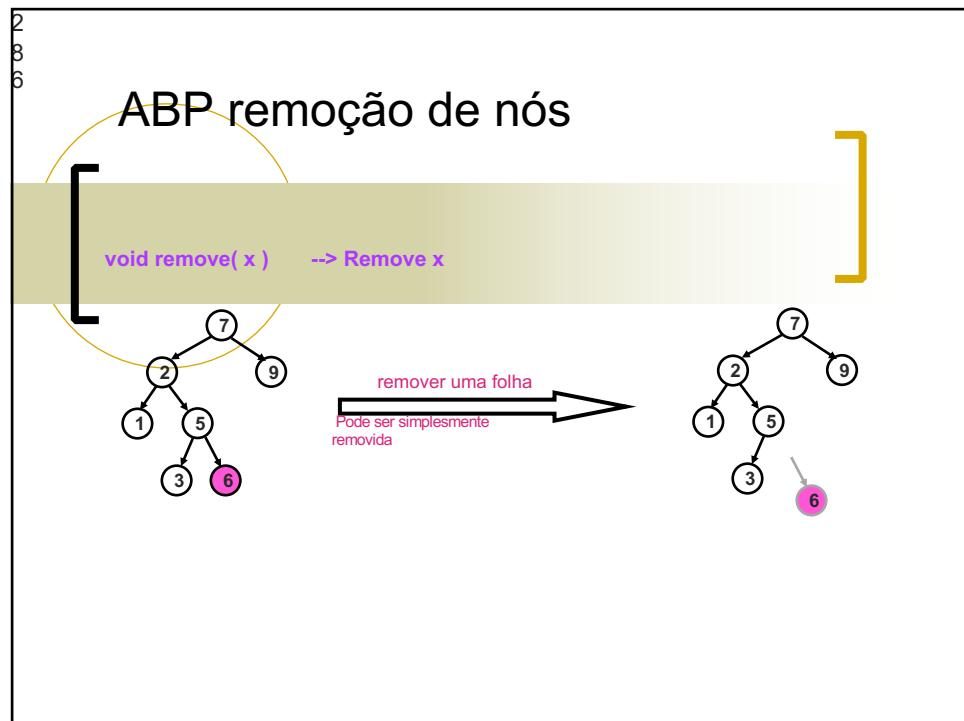
Carlos Lisboa Bento

284

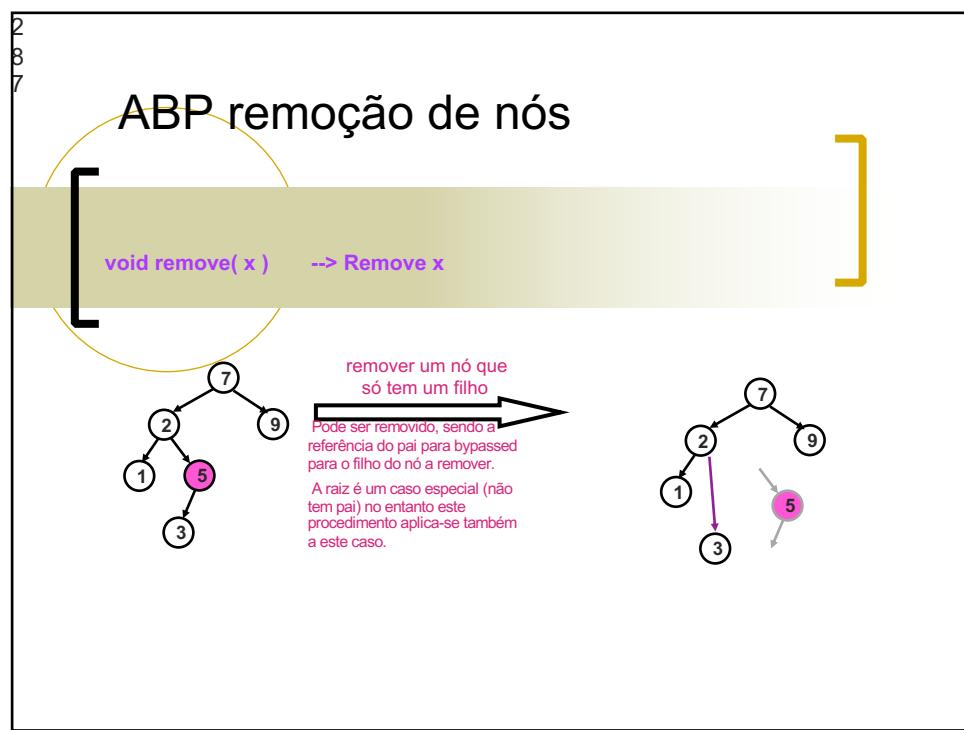
## [ REMOÇÃO (ABPs) ]

Carlos Lisboa Bento

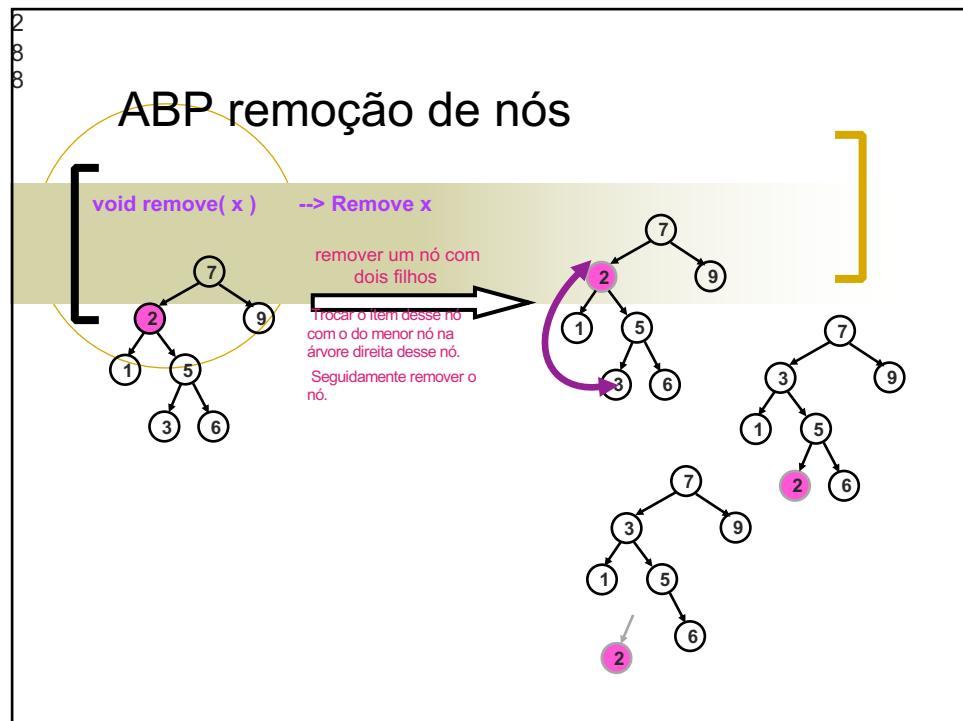
285



286



287



288



289

2  
9  
0

## Árvores AVL remoção de nós

Idêntico à inserção: realiza o apagamento<sup>(a)</sup> e reequilibra

- 1) rotações simples e duplas
- 2) desequilíbrio pode propagar-se até à raiz (o que não acontece na inserção!), necessidade eventual de reequilibrar até à raiz

<sup>(a)</sup> tal como é feito numa árvore binária de pesquisa

290

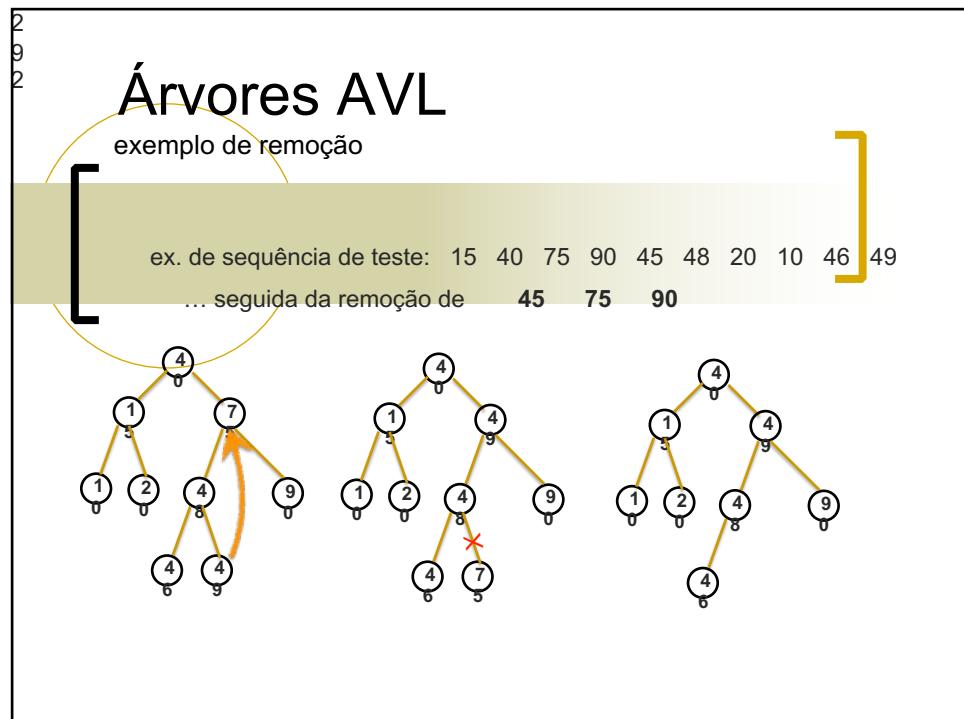
2  
9  
1

## Árvores AVL

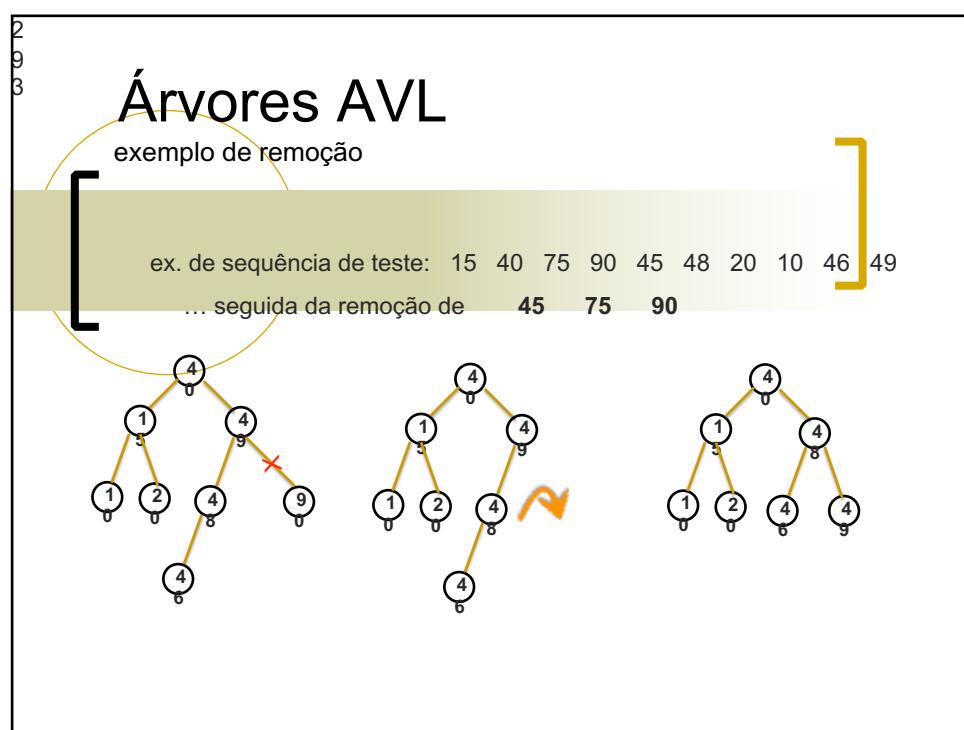
exemplo de remoção

ex. de sequência de teste: 15 40 75 90 45 48 20 10 46 49  
... seguida da remoção de 45 75 90

291



292



293

2  
9  
4

## Árvores AVL

demonstrações na Web

[ ex. de sequência de teste: 30 50 80 20 15 40 18  
... seguida de remoções ]

<http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html>

294

## REMOÇÃO (VPs)

Carlos Lisboa Bento

295

2  
9  
6

## Árvores VP remoção de nós

### Remoção (ideia geral):

- Começamos por remover o nó como fazemos em geral numa ABP
- ... a seguir o problema “resume-se” a remover um nó numa folha ou que só tem um descendente
- Se o nó a remover for vermelho – não há problema!
- Se o nó a remover for preto (há a violação de 4.)
- SOLUÇÃO:  
assegurar que um nó a remover foi antecipadamente recolorido de vermelho.



296

2  
9  
7

## Árvores VP remoção de nós

### Remoção (temos 3 casos):

- CASO 1: nó a remover é uma folha e está colorido de vermelho
- CASO 2: nó a remover está colorido de preto, tem um filho colorido de vermelho
- CASO 3: nó a remover é uma folha e está colorido de preto



297

2  
9  
8

## Árvores VP remoção de nós

CASO 1: nó a remover é uma folha e está colorido de vermelho

Neste caso a remoção geral para ABPs é suficiente

(a remoção não altera o número de nós pretos nos ramos da árvore)

298

2  
9  
9

## Árvores VP remoção de nós

CASO 2: nó a remover está colorido de preto, tem um filho colorido de vermelho

Neste caso depois remoção geral para ABPs, basta re-colorir de preto o nó filho

(assim asseguramos que se mantém o número de nós pretos nos ramos da árvore)

299

3  
0  
0

## Árvores VP remoção de nós

### CASO 3: nó a remover é uma folha e está colorido de preto

este caso recorre a um novo estado de coloração designado por duplo-preto que resulta em fundir o nó removido com o seu antecessor

e desenvolve-se em três sub-casos:

- CASO 3A: o irmão do nó colorido de duplo-preto está colorido de preto e tem um filho colorido de vermelho
- CASO 3B: o irmão do nó colorido de duplo-preto está colorido de preto e tem dois filhos coloridos de preto
- CASO 3C: o irmão do nó colorido de duplo-preto está colorido de vermelho

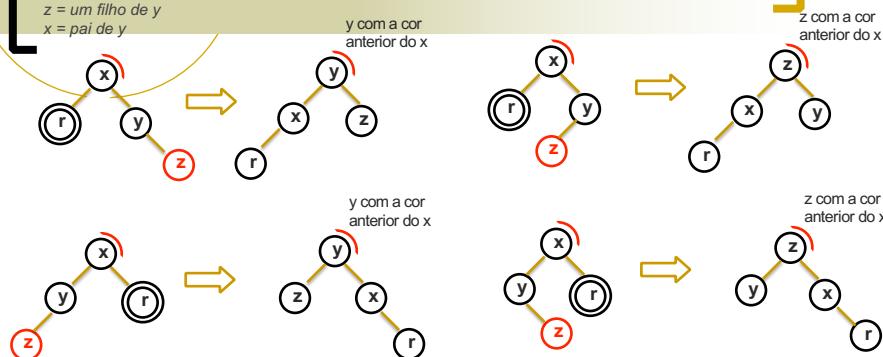
300

3  
0  
1

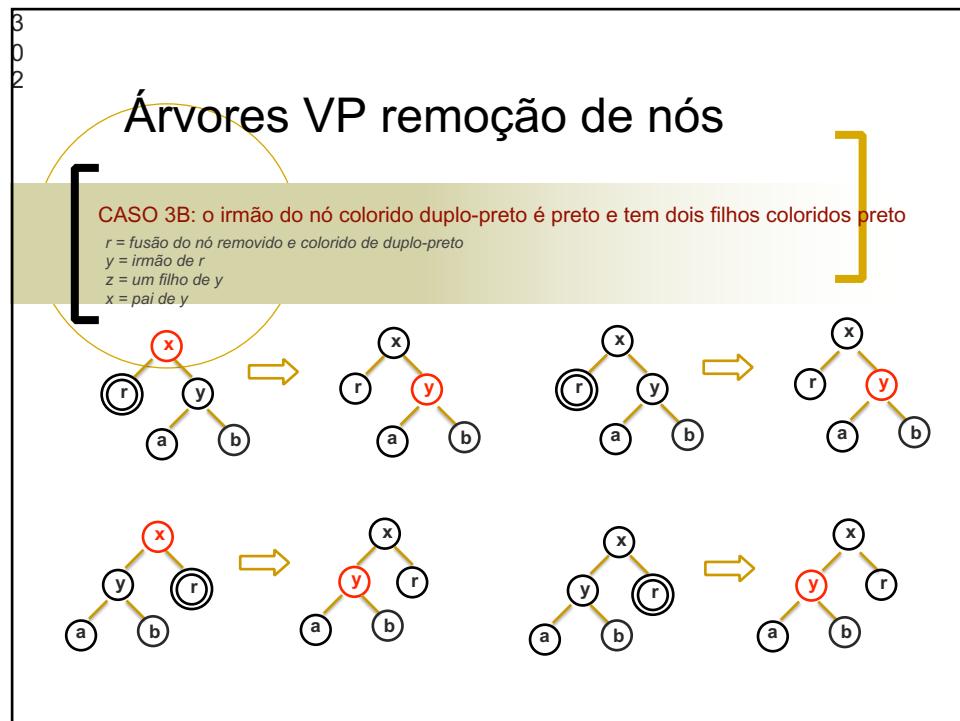
## Árvores VP remoção de nós

### CASO 3A: o irmão do nó colorido de duplo-preto é preto e tem um filho vermelho

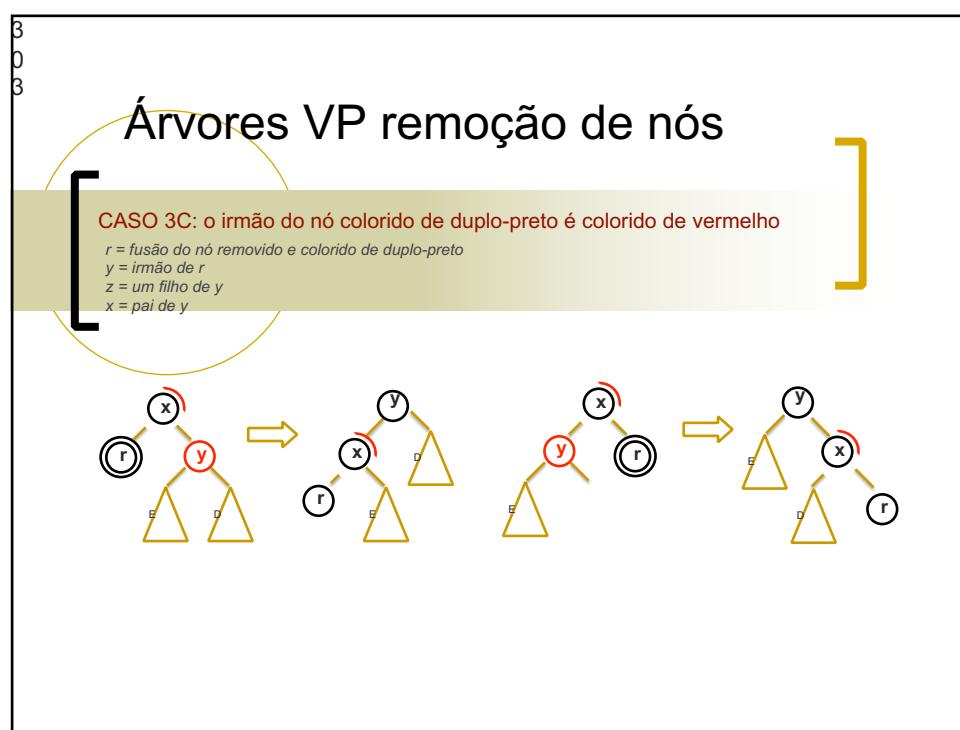
$r$  = fusão do nó removido e colorido de duplo-preto  
 $y$  = irmão de  $r$   
 $z$  = um filho de  $y$   
 $x$  = pai de  $y$



301



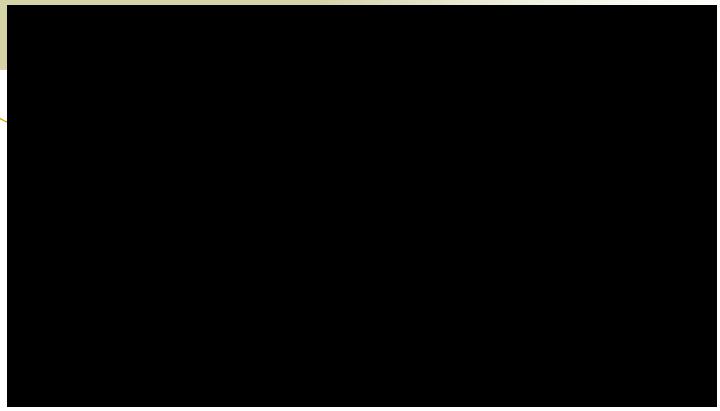
302



303

3  
0  
4

### Árvores VP remoção de nós



[https://www.youtube.com/watch?v=\\_ybZCHNSFOY](https://www.youtube.com/watch?v=_ybZCHNSFOY)

304

3  
0  
5

### Árvores VP remoção de nós

ex. de sequência de teste: 30 50 80 20 15 40 18  
... seguido de remoção

<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>

305

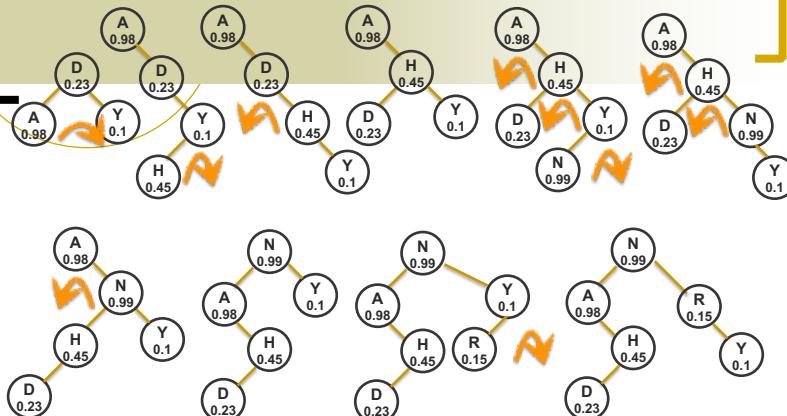
# REMOÇÃO (TREAPs)

Carlos Lisboa Bento

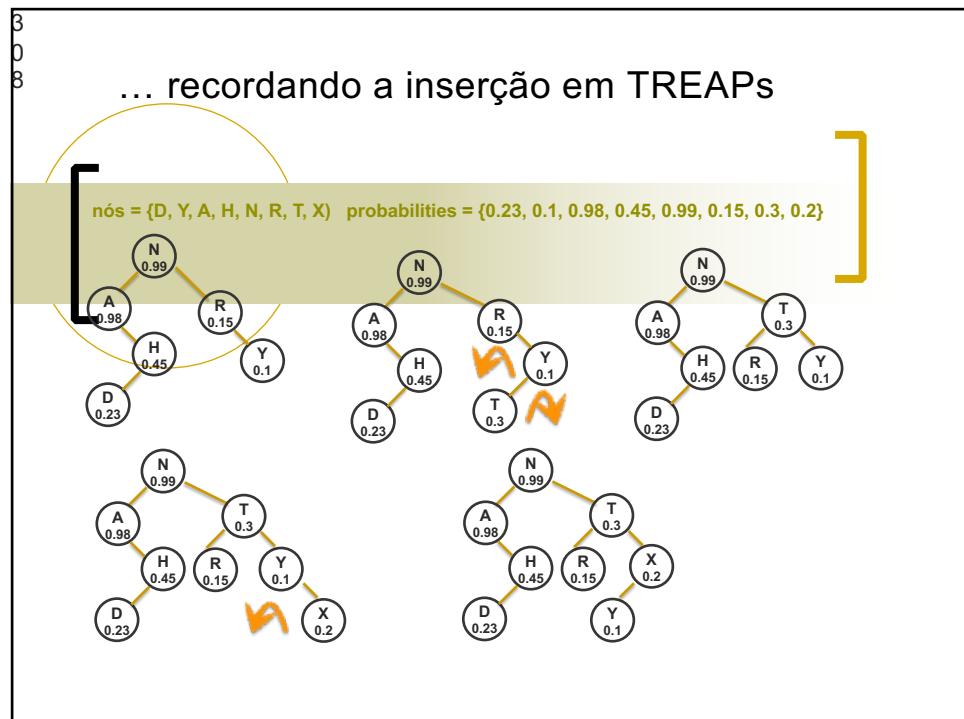
306

... recordando a inserção em TREAPs

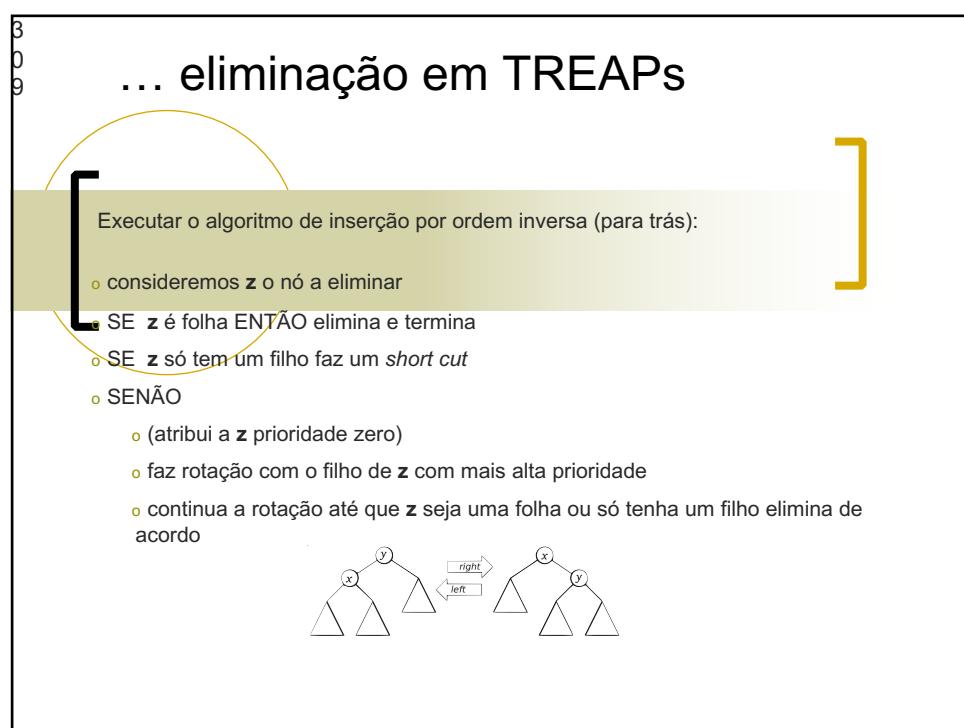
nós = {D, Y, A, H, N, R, T, X} probabilities = {0.23, 0.1, 0.98, 0.45, 0.99, 0.15, 0.3, 0.2}



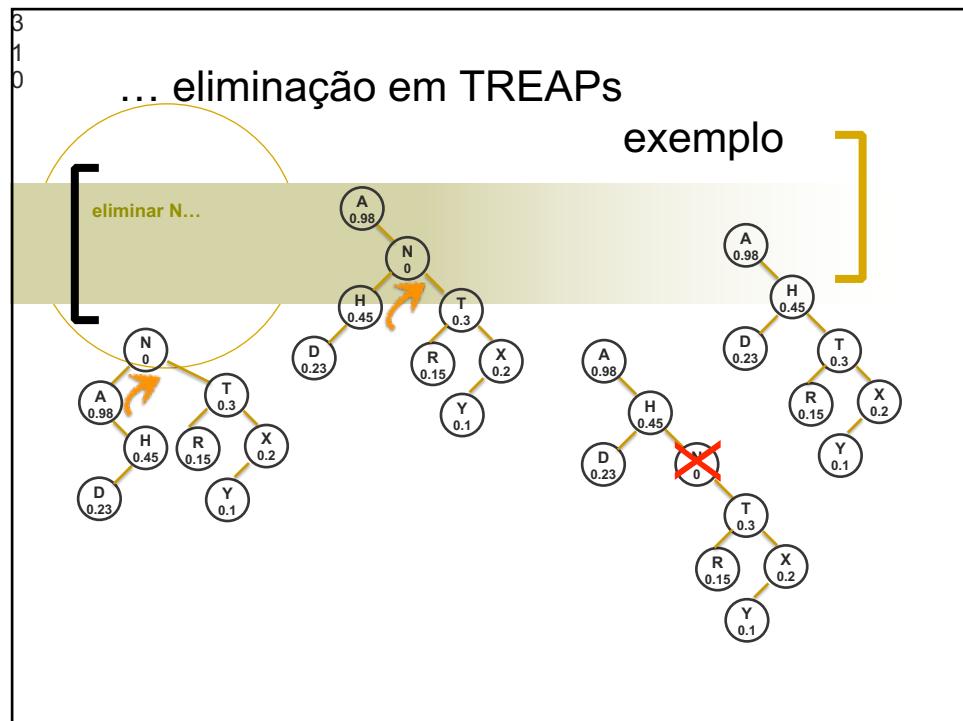
307



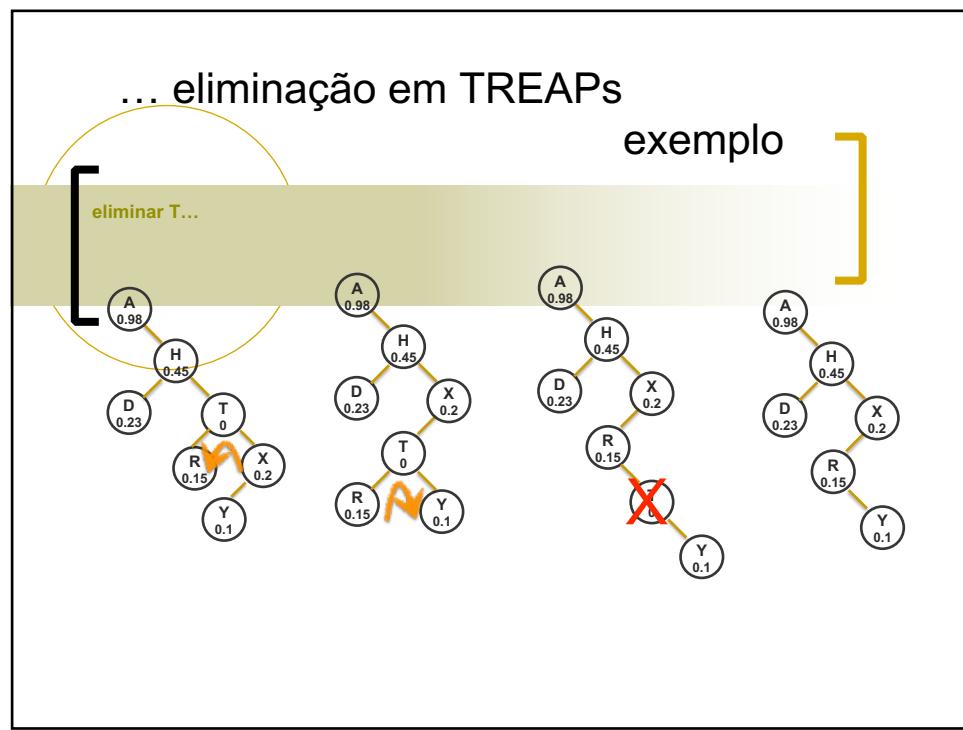
308



309



310



311

## REMOÇÃO (SPLAY TREES)

Carlos Lisboa Bento

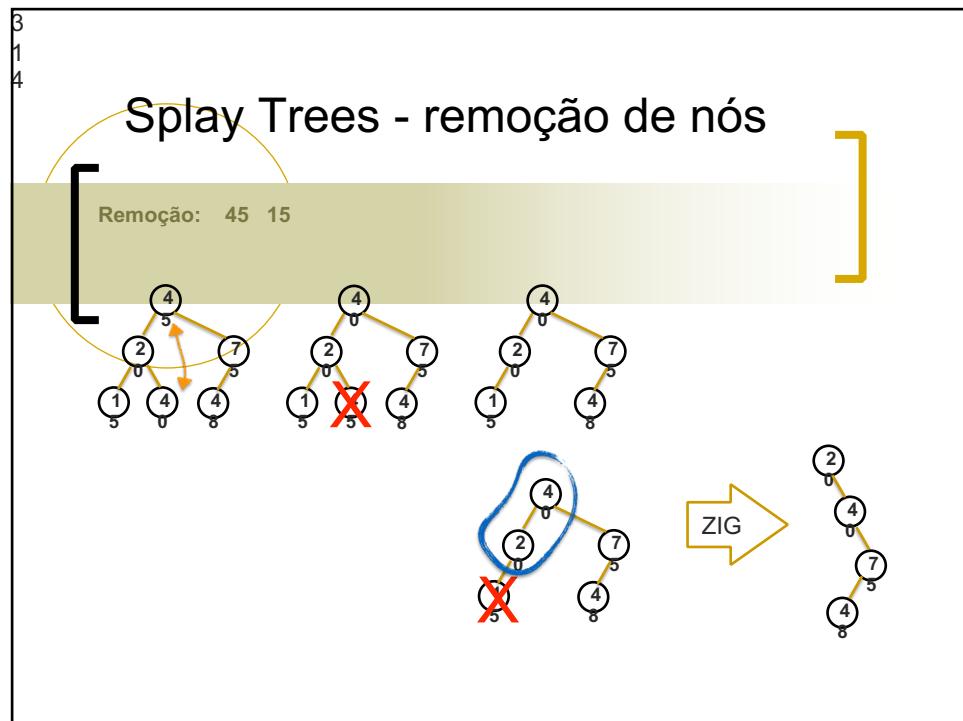
312

### Splay Trees - remoção de nós

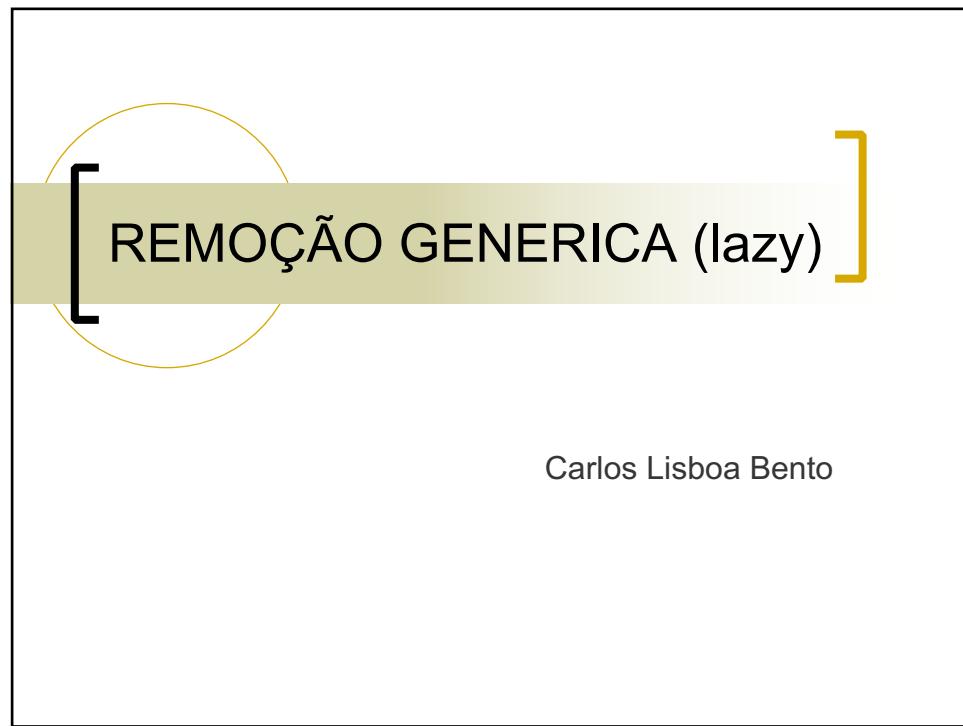
Remoção:

- Começar por fazer a remoção como em geral numa ABP
- Depois da remoção aplicar a operação de splaying sobre o pai do nó removido

313



314



315

3  
1  
6

## Árvores remoção de nós (lazy)

- o cria um contador de total de nós e de nós removidos
- o marca o nó removido como tal
- o quando a percentagem de nós removidos atinge um determinado limite reconstrói a árvore

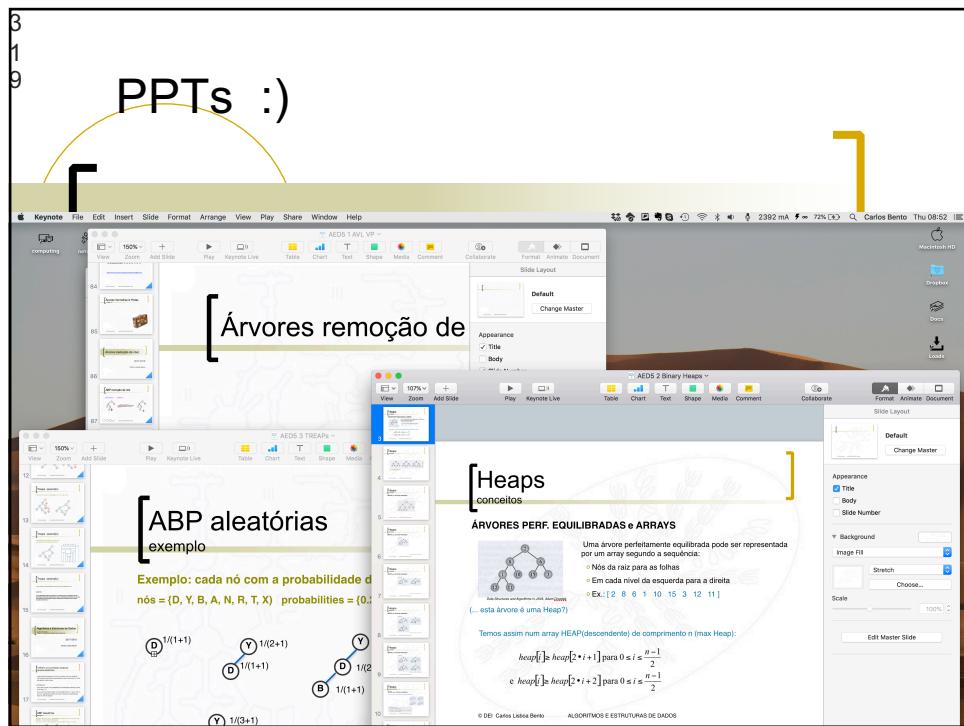
316

3  
1  
7

## Bibliography (update)



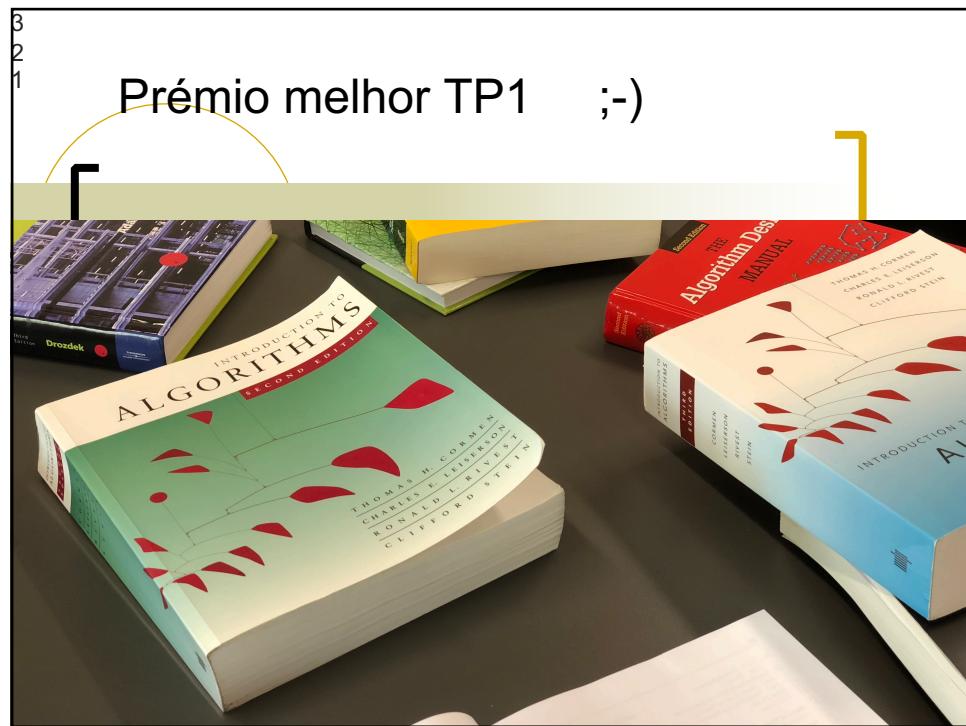
317



319



320



321



322

[ Algoritmos e Estruturas de Dados ]

árvore B

■ 2020-2021

■ Carlos Lisboa Bento

323

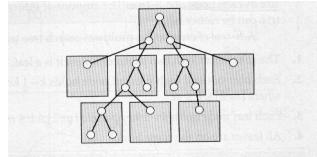
# Árvores B

conceitos

Múltiplos acessos a disco

Tempos típicos de acesso  
seek time: 10ms  
latency (7200rpm): 4ms  
data transfer rate: 300MB/s

As árvores binárias não são adequadas para manipulação de memória secundária



Data Structures and Algorithms in JAVA, Adam Drozdak

324

# Árvores B

conceitos

**ÁRVORES MULTICAMINHO**

1. Cada nó com m descendentes e m-1 chaves.
2. Chaves em cada nó ordenadas.
3. Chaves nos primeiros i descendentes menores que a i-ésima chave.
4. Chaves nos últimos m-i descendentes maiores que a i-ésima chave.

Data Structures and Algorithms in JAVA, Adam Drozdok

325

# Árvores B

conceitos

**ÁRVORES B**

Uma árvore-B de ordem m é uma árvore m-ária com as seguintes propriedades:

1. A raiz tem pelo menos 2 subárvores a menos que seja folha.
2. Os nós que não são raiz nem folha têm k-1 chaves e k referências para subárvores c/  $m/2 \leq k \leq m$ .
3. Os nós folha têm k-1 chaves c/  $m/2 \leq k \leq m$ .
4. Todas as folhas estão no mesmo nível.

Data Structures and Algorithms in JAVA, Adam Drozdok

326

# Árvores B

conceitos

```

class BTreenode {
    int m = 4;
    boolean leaf = true;
    int keyTally = 1;
    int keys[] = new int[m-1];
    BTreenode references[] = new BTreenode[m];
    BTreenode(int key) {
        keys[0] = key;
        for (int i = 0; i < m; i++)
            references[i] = null;
    }
}

```

327

# Árvores B

conceitos

Procura

Pior caso – todos os nós com o n. mínimo de chaves, 1 na raiz e q = m/2 nos restantes nós.

Para uma árvore de altura h temos:

$$\begin{aligned}
& 1 \text{ chave na raiz} + \\
& 2(q-1) \text{ chaves no nível 2} + \\
& 2q(q-1) \text{ chaves no nível 3} + \\
& 2q^2(q-1) \text{ chaves no nível 4} + \\
& \dots \\
& 2q^{h-2}(q-1) \text{ chaves no nível } h +
\end{aligned}$$

Temos então:

$$1 + \left( \sum_{i=0}^{h-2} 2q^i \right) (q-1) \text{ chaves numa árvore B}$$

$$1 + \left( \sum_{i=0}^{h-2} 2q^i \right) (q-1) = 1 + 2(q-1) \left( \sum_{i=0}^{h-2} q^i \right) = 1 + 2(q-1) \left( \frac{q^{h-1}-1}{q-1} \right) = -1 + 2q^{h-1}$$

328

# Árvores B

conceitos

**Procura**

Pior caso – todos os nós com o n. mínimo de chaves,  
1 na raiz e q = m/2 nos restantes nós.

Para uma árvore de altura h temos:

$$n \geq -1 + 2q^{h-1} \quad h \leq \log_q \frac{n+1}{2} + 1$$

Ex.:  
 $m=200 \rightarrow q=100$   
 $n=2\ 000\ 000$   
 $h \leq 4$

329

# Árvores B

conceitos / implementação

**Procura**

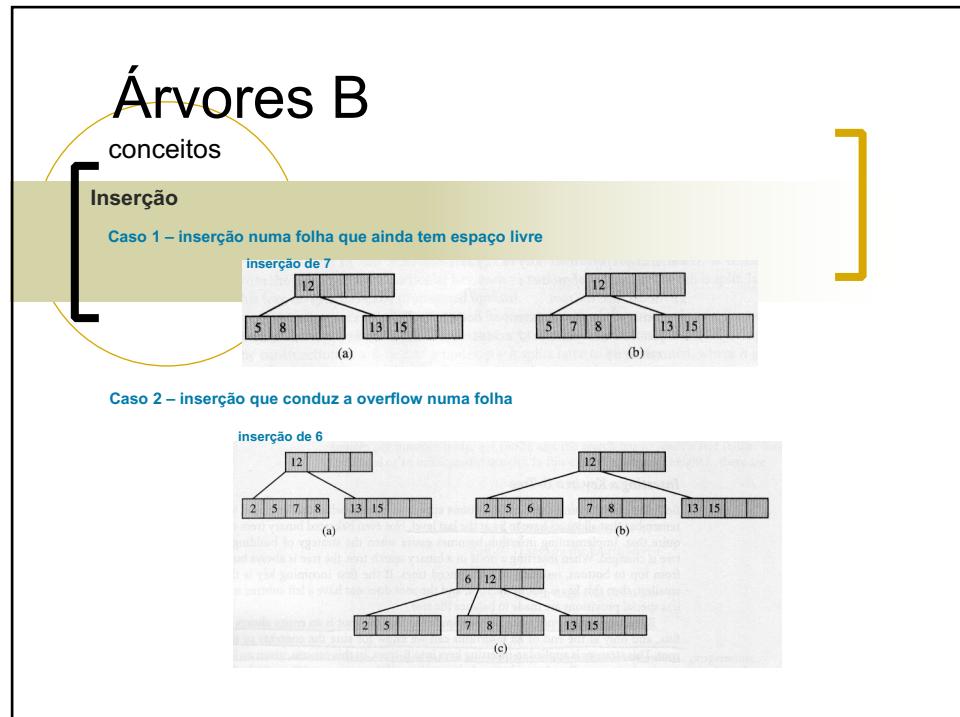
Procurar 76

```

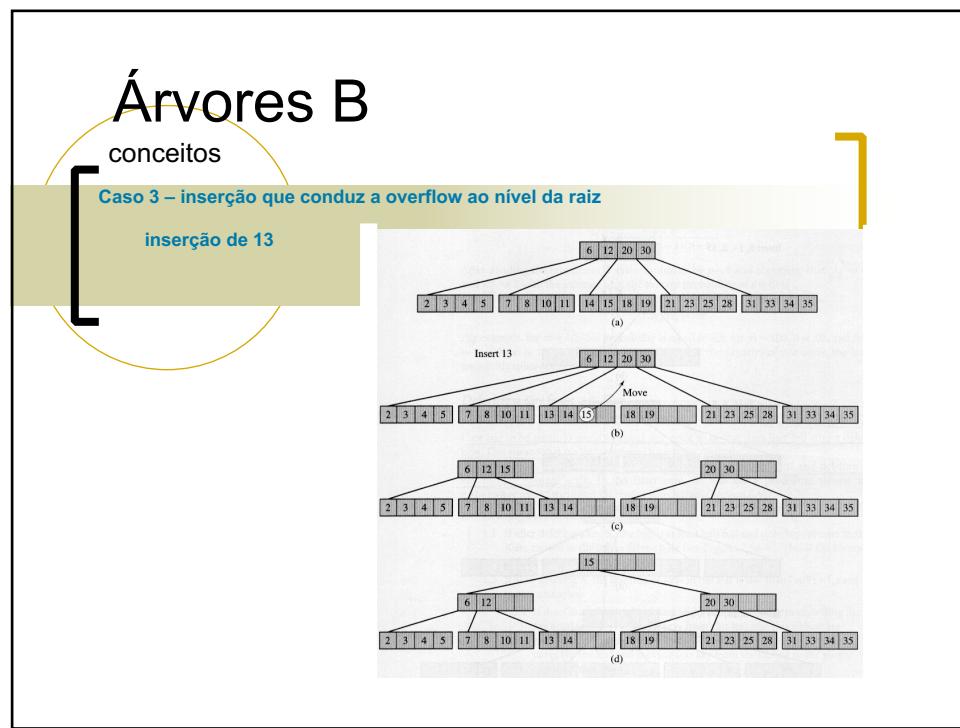
public BTreenode BTreesearch(int key) {
    return BTreesearch(key,root);
}
protected BTreenode BTreesearch(int key, BTreenode node) {
    if (node != null) {
        int i = 1;
        for ( ; i <= node.keyTally && node.keys[i-1] < key; i++);
        if (i > node.keyTally || node.keys[i-1] > key)
            return BTreesearch(key,node.references[i-1]);
        else return node;
    }
    else return null;
}

```

330



331



332

# Árvores B

implementação

## Inserção - Algoritmo

```

BTreeInsert (K)
    find a leaf node to insert K;
    while (true)
        find a proper position in array keys for K;
        if node is not full
            insert K and increment keyTally;
            return;
        else split node in node1 and node2; // node1 = node, node2 is new;
            distribute keys and references evenly between node1 and node2 and
            initialize properly their keyTally's;
            K = the last key of node1;
            if node was the root
                create a new root as parent of node1 and node2;
                put K and references to node1 and node2 in the root, and set its keyTally to 1;
                return;
            else node = its parent; // and now process the node's parent;
    
```

333

# Árvores B

conceitos

## Inserção

Exemplo árvore c/ m=5

8	14	2	15	3	1	16	6	5	27
37	18	25	7	13	20	22	23	24	

Insert 8, 14, 2, 15  
 (a)

334

# Árvores B

conceitos

**Inserção**  
Exemplo árvore c/ m=5

8 14 2 15 3 1 16 6 5 27  
37 18 25 7 13 20 22 23 24

Insert 8, 14, 2, 15      (a)

Insert 3      (b)

335

# Árvores B

conceitos

**Inserção**  
Exemplo árvore c/ m=5

8 14 2 15 3 1 16 6 5 27  
37 18 25 7 13 20 22 23 24

Insert 8, 14, 2, 15      (a)

Insert 3      (b)

Insert 1, 16, 6, 5      (c)

336

# Árvores B

conceitos

**Inserção**  
Exemplo árvore c/ m=5

8 14 2 15 3 1 16 6 5 27  
37 18 25 7 13 20 22 23 24

337

# Árvores B

conceitos

**Inserção**  
Exemplo árvore c/ m=5

8 14 2 15 3 1 16 6 5 27 37  
18 25 7 13 20 22 23 24

338

# Árvores B

conceitos

**Inserção**

Exemplo árvore c/ m=5

8 14 2 15 3 1 16 6 5 27 37  
18 25 7 13 20 22 23 24

Insert 27, 37

(d)

Insert 18, 25, 7, 13, 20

(e)

339

# Árvores B

conceitos

**Inserção**

Exemplo árvore c/ m=5

8 14 2 15 3 1 16 6 5 27 37  
18 25 7 13 20 22 23 24

Insert 27, 37

(d)

Insert 18, 25, 7, 13, 20

(e)

Insert 22, 23, 24

(f)

340

**Árvores B**

conceitos

**Caso 1A – Numa folha em que esta não fica com menos de  $m/2 - 1$  chaves  
eliminação de 6**

**Eliminação**  
Exemplo árvore c/  $m=5$

341

**Árvores B**

conceitos

**Caso 1A – Numa folha em que esta não fica com menos de  $m/2 - 1$  chaves  
eliminação de 6**

**Eliminação**  
Exemplo árvore c/  $m=5$

342

# Árvores B

conceitos

**Eliminação**  
eliminação de 7

Caso 1B – Numa folha em que esta fica com menos de  $m/2 - 1$  chaves, mas pelo menos um dos seus irmãos não está no limite inferior do número de chaves (redistribuição)

**Eliminação**  
Exemplo árvore c/ m=5

(b)

Delete 7

343

# Árvores B

conceitos

**Eliminação**  
eliminação de 7

Caso 1B – Numa folha em que esta fica com menos de  $m/2 - 1$  chaves, mas pelo menos um dos seus irmãos não está no limite inferior do número de chaves (redistribuição)

**Eliminação**  
Exemplo árvore c/ m=5

(b)

Delete 7

(c)

344

# Árvores B

conceitos

**Caso 1C – Numa folha em que esta fica com menos de  $m/2 - 1$  chaves e todos os seus irmãos estão no limite inferior do número de chaves**

**eliminação de 8**

**Eliminação**  
Exemplo árvore c/ m=5

(c)

345

# Árvores B

conceitos

**Caso 1C – Numa folha em que esta fica com menos de  $m/2 - 1$  chaves e todos os seus irmãos estão no limite inferior do número de chaves**

**eliminação de 8**

**Eliminação**  
Exemplo árvore c/ m=5

(c)

(d)

346

# Árvores B

conceitos

**Caso 1D – Numa folha ou não folha que resulte na fusão de nós em que o ascendente directo é uma raiz com uma única chave.**

**eliminação de 8**

**Eliminação**  
Exemplo árvore c/ m=5

(c)

(d)

347

# Árvores B

conceitos

**Caso 1C – Numa folha em que esta fica com menos de  $m/2 - 1$  chaves e todos os seus irmãos estão no limite inferior do número de chaves**

**eliminação de 8**

**Eliminação**  
Exemplo árvore c/ m=5

(c)

(d)

(e)

348

**Árvores B**

conceitos

**Eliminação**

Caso 2 – Num nó não folha. Vai ser reduzido ao problema de eliminar um nó de uma folha – caso contrário levaria a problemas de equilíbrio da árvore.

**eliminação de 16**

**Eliminação**  
Exemplo árvore c/ m=5

349

**Árvores B**

conceitos

**Eliminação**

Caso 2 – Num nó não folha. Vai ser reduzido ao problema de eliminar um nó de uma folha – caso contrário levaria a problemas de equilíbrio da árvore.

**eliminação de 16**

**Eliminação**  
Exemplo árvore c/ m=5

350

# Árvores B

conceitos  
Eliminação - Algoritmo

```

BTreeDelete (K)
    node = BTreeSearch(K,root);
    if (node != null)
        if node is not a leaf
            find a leaf with the closest successor S of K;
            copy S over K in node;
            node = the leaf containing S;
            delete S from node;
        else delete K from node;
        while (true)
            if node does not underflow
                return;
            CASO 1A else if there is a sibling of node with enough keys
                redistribute the keys between node and its sibling;
                return;
            CASO 1B else if node's parent is the root
                if the parent has only one key
                    merge node, its sibling, and the parent to form a new root;
                else merge node and its sibling;
                return;
            CASO 1C else merge node and its sibling;
            CASO 1D node = its parent;
    
```

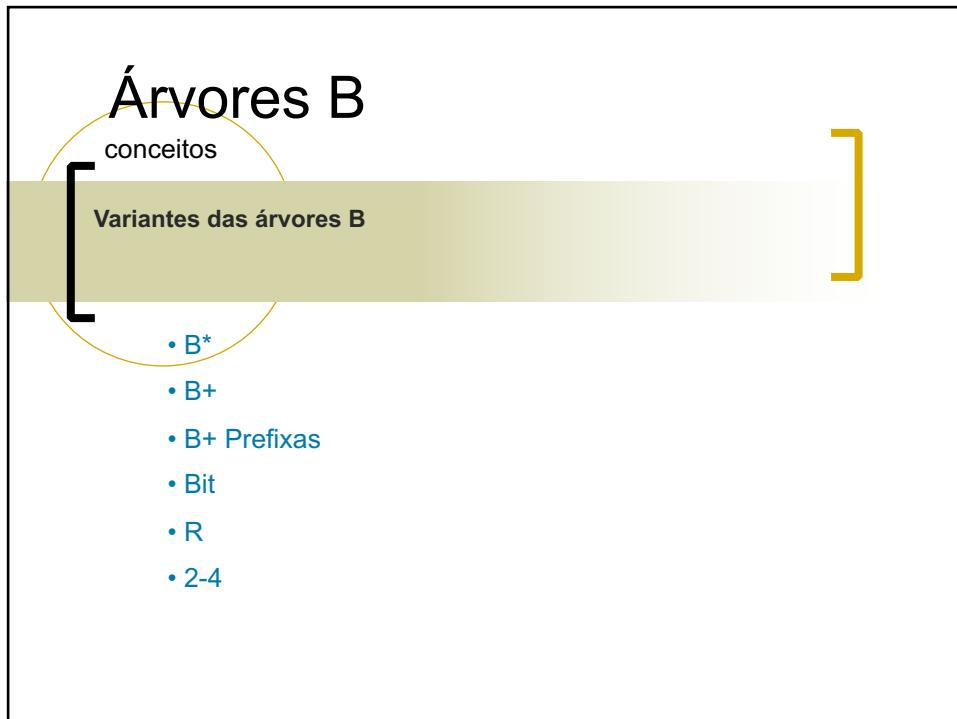
351

# Árvores B

Exemplo cálculo de M

- Cada nó vai ocupar num bloco em disco
- Bloco = 8192 bytes
- Chave = 32 bytes
- M-1 chaves
- Cada nó  $(M-1) * 32 + M \text{ Refs} = 32 * M - 32 + M \text{ Refs}$
- Ref = um número de referência para outro bloco em disco
- Referencia = 4 bytes
- Necessidades de memória para um nó não folha  $36 * M - 32$
- $(8192+32) / 36 = 228,444 \rightarrow M=228$

352



353

The diagram illustrates the etymology of B-trees. At the top, the title "Árvores B" is followed by the subtitle "... porquê B?". A yellow bracket groups the text "Etymology [edit]" and a block of text about the origin of the name. Below this, a yellow circle contains the text "... porquê B?".

**Etymology [edit]**

Rudolf Bayer and Ed McCreight invented the B-tree while working at Boeing Research Labs in 1971 (Bayer & McCreight 1972), but they did not explain what, if anything, the *B* stands for. Douglas Comer explains:

The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees. (Comer 1979, p. 123 footnote 1)

Donald Knuth speculates on the etymology of B-trees in his May, 1980 lecture on the topic "CS144C classroom lecture about disk storage and B-trees", suggesting the "B" may have originated from Boeing or from Bayer's name.<sup>[4]</sup>

Ed McCreight answered a question on B-tree's name in 2013:

Bayer and I were in a lunchtime where we get to think [of] a name. And ... B is, you know ... We were working for Boeing at the time, we couldn't use the name without talking to lawyers. So, there is a B. [The B-tree] has to do with balance, another B. Bayer was the senior author, who [was] several years older than I am and had many more publications than did. So there is another B. And so, at the lunch table we never did resolve whether there was one of those that made more sense than the rest. What really lives to say is: the more you think about what the B in B-trees means, the better you understand B-trees.<sup>[5]</sup>

[https://en.wikipedia.org/wiki/B-tree#Etymology\\_unknown](https://en.wikipedia.org/wiki/B-tree#Etymology_unknown)

355



356



357

# Grafos

conceitos

**Grafos: terminologia e representações**

- Um grafo  $G$  é constituído por dois conjuntos,  $N$  e  $A$ :

  - $N$  é um conjunto finito, não vazio, de elementos denominados **nós** (ou vértices);
  - $A$  é um conjunto de pares de nós ( $n_i, n_j$ ) denominados **arcos**. Há uma relação binária entre cada elemento do conjunto  $A$  (pares de nós).

$G = (N, A)$

- Exemplo
- $G = (N, A)$  em que:

  - $N = \{1, 2, 3, 4\}$
  - $A = \{(1,2), (2,1), (2,3), (2,4), (3,3), (4,1), (4,3)\}$

**Representação gráfica de  $G$**

9

358

# Grafos

conceitos

**Grafos dirigidos e grafos não dirigidos**

**G1 - grafo dirigido**  
Relação:  $A$  ajuda  $B$

**G2 - grafo não dirigido**  
Relação:  $A$  é parente de  $B$

**EXEMPLOS**

**N1 = {José, Ana, Júlio, Pedro}**  
**A1 = {(Pedro,José), (José,Pedro), (Pedro,Júlio), (Ana,Júlio)}**

**N2 = {José, Ana, Júlio, Pedro}**  
**A2 = {(Pedro,José), (Pedro,Júlio), (Ana,Júlio)}**

10

359

# Grafos

conceitos

Grafos: terminologia e representação (cont.)

**Grafos dirigidos**

- Um nó pode não ter nenhum arco associado (caso do nó 7).
- Nós ligados por arcos são ditos adjacentes; o nó 2 é adjacente de 1, 3 e 4.
- Um arco que liga nós adjacentes diz-se incidente a esses nós.
- Um arco  $\alpha$  é incidente ao nó  $n$  se chega a esse nó. O arco  $\alpha$  é incidente ao nó 6.
- Um arco  $\alpha$  é incidente do nó  $n$  se parte desse nó. O arco  $\alpha$  é incidente do nó 5.
- O arco  $\alpha$  tem a cauda no nó 5 e a cabeça no nó 6. Notar que  $\alpha$  representa-se por (5,6).

11

360

# Grafos

conceitos

**Grafos dirigidos**

- **Grau de um nó:** número de arestas incidentes nesse nó.  
Ex: o grau do nó "Júlio" é 2
- **Grau interno de um nó:** número de arcos que têm esse nó como cabeça.
- **Grau externo de um nó:** número de arcos que têm esse nó como cauda.
- Ex: Nô 1      | Grau interno = 2  
                  | Grau externo = 1

12

361

# Grafos

conceitos

Grafos: terminologia e representação (cont.)

**Caminho:**  
sequência de um ou mais arcos em que o segundo nó de cada arco coincide com o primeiro do arco seguinte:  
 $\{(a,n_1), (n_1,n_2), \dots (n_i, b)\}$  – caminho de a para b  
 se  $a = b$  temos um ciclo

- É possível a partir do nó 1 atingir o nó 3 percorrendo os arcos  $(1,2)$  e  $(2,3)$ . Estes arcos formam um **caminho** de 1 para 3.
- O **comprimento do caminho** é igual ao número de arcos existentes no caminho
- Se o nó de partida coincide com o de chegada temos um **ciclo**. Existe um ciclo unindo os nós 1, 2 e 4.
- Um ciclo de um único arco chama-se um **laço**. Caso do nó 3.
- Grafos cíclicos: os que contêm um ou mais ciclo. Caso contrário são **acíclicos**

13

362

# Grafos

conceitos

Grafos: terminologia e representação (cont.)

- No caso de grafos não dirigidos usa-se o termo **círcuito** em vez de ciclo
- Existe um circuito unindo os nós Pedro, José e Júlio

14

363

# Grafos

conceitos

Grafos: terminologia e representação (cont.)

- Subgrafo:** Subconjunto de nós de um dado grafo juntamente com todos os arcos cujas duas extremidades são nós desse subconjunto

$G$

$G'$  é um subgrafo de  $G$

$N' = \{1, 4, 3\}$   
 $A' = \{(4,1), (4,3), (3,3)\}$

15

364

# Grafos

conceitos

Grafos: terminologia e representação (cont.)

- Grafo parcial:** grafo constituído pelos mesmos nós do grafo original mas em que se considera apenas um subconjunto dos arcos

$G$

$G'$  é um grafo parcial de  $G$

16

365

## Grafos

conceitos

**Grafos: terminologia e representação (cont.)**

- Grafo conexo: se tem pelo menos um nó a partir do qual existem caminhos para todos os restantes

Ex: a partir de Pedro é possível atingir qualquer dos outros nós.

**Grafo fortemente conexo:** se de todos os nós é possível atingir todos os demais

17

366

## Grafos

conceitos

- Grafo completo: se tem o número máximo de arcos
- Multigrafo: se tem múltiplas ocorrências de um mesmo arco

■ Uma árvore é um caso particular de um grafo, mas nem todos os grafos são árvores

Grafo completo

Multigrafo

18

367

# Grafos

conceitos

Grafos ponderados

```

graph LR
    Pedro((Pedro)) -- 3 --> Jose((José))
    Jose -- 5 --> Pedro
    Pedro -- 1 --> Julio((Julio))
    Julio -- 1 --> Pedro
    Julio -- 10 --> Ana((Ana))
    Ana -- 10 --> Julio
  
```

Um grafo pode ter números associados a cada arco. Nesse caso o grafo chama-se **ponderado** e o número junto a cada arco designa-se por **peso do arco**

19

368

# Grafos

conceitos

Para que servem os grafos?

Sistemas de Informação Geográfica

Bibliografia

- Algorithms in C, ROBERT SEDGEWICK
- Data Structures and Problem Solving Using JAVA, MARK LEN WEISS
- Data Structures and Algorithms in JAVA, Adam Drozdek

2

369

**Grafos**  
conceitos

Para que servem os grafos?

Sistemas de Navegação

Bibliografia

- Algorithms in C*, ROBERT SEDGEWICK
- Data Structures and Problem Solving Using JAVA*, MARK ALLEN WEISS
- Data Structures and Algorithms in JAVA*, Adam Drozdek

2

370

**Grafos**  
conceitos

Para que servem os grafos?

Alguns exemplos: Teoria da computação (exp: Máquinas de Turing)

Turing Machine

(potentially) infinite tape

1	0	1	1	0	0	1	1	0	b	b	b
---	---	---	---	---	---	---	---	---	---	---	---

read/write head

finite state control

5

371

## Grafos

conceitos

Para que servem os grafos?

Alguns exemplos: Optimização de percursos (exp. Caminhos num mapa)

5

372

## Grafos

conceitos

Para que servem os grafos?

Alguns exemplos: Análise e planeamento de projectos (exp: diagrama de Gantt)

The critical path

5

373

# Grafos

conceitos

Para que servem os grafos?

Alguns exemplos: Identif. de componentes químicos (exp: interac̄es moléculas)

5

374

# Grafos

conceitos

Para que servem os grafos?

Alguns exemplos: Genética (exp: co-relações genéticas)

5

375

# Grafos

conceitos

Para que servem os grafos?

Alguns exemplos: Linguística (ex.: represent. gramáticas; redes semânticas)

5

376

# Grafos

conceitos

Para que servem os grafos?

Alguns exemplos: Ciências sociais (exp: interacções sociais)

**Nobel Prize NOBEL LAUREATES**  
By country of origin and number of prizes awarded,  
1901 - 2013  
TOTAL 876

Note: Data for U.K. includes Northern Ireland. Formerly known as Germany or as Soviet Union, East Germany includes former German Democratic Republic and former Russian Empire and C.I.S.R.

Source: Nobel Foundation © 2013/2014

5

377

# Grafos

conceitos

Para que servem os grafos?

Alguns exemplos: Robótica (exp: máquina de estados finitos)

5

378

# Grafos

representação

ข้อบคณ

20

379

## Grafos

representação

**Representação de grafos**

■ **Matriz de adjacência:** dado um grafo  $G = (N, A)$  de  $n$  nós a sua matriz de adjacência  $MA$  é uma matriz  $n \times n$  com a propriedade de  $M(i, j) = 1$  se existe um arco do nó  $n_i$  para o nó  $n_j$ . Caso contrário  $M(i, j) = 0$ .

	1	2	3	4
1	0	1	0	0
2	1	0	1	1
3	0	0	1	0
4	1	0	1	0

	José	Pedro	Júlio	Ana
José	0	1	1	0
Pedro	1	0	1	0
Júlio	1	1	0	1
Ana	0	0	1	0

Matriz simétrica

21

380

## Grafos

representação

**Vantagens da matriz de adjacência**

- Torna fácil verificar se dois nós são adjacentes (estão ligados por um arco)
- Torna fácil juntar ou retirar arcos (ou arestas) do grafo
- Torna fácil determinar o grau de um grafo

**Grafo**

**Matriz de adjacência**

	1	2	3	4	Grau externo
1	0	1	0	0	1
2	1	0	1	1	3
3	0	0	1	0	1
4	1	0	1	0	2

Grau interno

	1	2	3	4	Grau
José	0	1	1	0	2
Pedro	1	0	1	0	2
Júlio	1	1	0	1	3
Ana	0	0	1	0	1

No caso de não haver informação associada aos nós nem aos arcos o grafo pode ser completamente descrito pela sua matriz de adjacência

22

381

## Grafos

representação

**Caminhos e transitividade**

- Num grafo descrito pela sua matriz de adjacência  $\text{adj}$
- $\text{adj}[i, k] == 1 \&& \text{adj}[k, j] == 1$  Verdadeira se e só se existe um arco de  $i$  para  $k$  e um arco de  $k$  para  $j$
- $\text{adj}[i, k] == 1 \&& \text{adj}[k, j] == 1$  Existe um caminho de comprimento 2 do nó  $i$  para o nó  $j$  passando pelo nó  $k$
- Consideremos a expressão:  
 $(\text{adj}[i, 0] == 1 \&& \text{adj}[0, j] == 1) \mid\mid \dots \mid\mid (\text{adj}[i, \text{max}-1] == 1 \&& \text{adj}[\text{max}-1, j] == 1)$   
 Verdadeira se e só se há pelo menos um caminho de comprimento 2 entre os nós  $i$  e  $j$

Matriz de caminhos de comprimento 2  
 $\text{adj}_{[i, j]}^2$  — Matriz em que cada elemento tem o valor 1 se e só se existe um caminho de comprimento 2 entre  $i$  e  $j$

23

382

## Grafos

representação

**Matriz de caminhos de comprimento 2 —  $\text{adj}_2$**

$\text{adj}$  MATRIZ DE ADJACÊNCIA

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

$\text{adj}_2$  obtém-se através do produto lógico de  $\text{adj}$  por ela própria

$$\left[ \begin{array}{c} \\ \\ \end{array} \right] \wedge \left[ \begin{array}{c} \\ \\ \end{array} \right] = \left[ \begin{array}{c} \\ \\ \end{array} \right] \text{adj}_2$$

Nota: produto lógico de duas matrizes obtém-se multiplicando as duas matrizes mas substituindo a multiplicação por AND e a soma por OR

24

383

## Grafos

representação

**adj<sub>3</sub>** — Matriz de caminhos de comprimento 3: cada elemento  $\text{adj}_3[i, j]$  é verdadeiro se e só se houver pelo menos um caminho de comprimento 3 entre i e j

$\text{adj}_3 = \text{adj} \wedge \text{adj}_2$

**adj**

	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

**adj<sub>3</sub>**

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

**adj<sub>4</sub>**

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

Genericamente:  $\text{adj}_n = \text{adj} \wedge \text{adj}_{n-1}$

25

384

## Grafos

representação

Matriz dos caminhos de comprimento  $\leq n$

Existe um caminho de comprimento menor ou igual a 3 se a seguinte expressão for verdadeira:

$\text{adj} \vee \text{adj}_2 \vee \text{adj}_3$

**Matriz de caminhos de comprimento  $\leq 3$**

	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

26

385

## Grafos

representação

**QUESTÃO:**  
Dado um grafo, pretende-se saber se entre dois quaisquer nós existe um caminho, qualquer que seja o comprimento desse caminho

**SOLUÇÃO:**  
Para um grafo de  $n$  nós:

$\text{Caminho}[i, j] = \text{adj}[i, j] \vee \text{adj}_2[i, j] \vee \dots \vee \text{adj}_n[i, j]$

Esta matriz chama-se **fecho transitivo**

$\text{Caminho} = \text{adj} \parallel \text{adj}_2 \parallel \text{adj}_3 \parallel \text{adj}_4 \parallel \text{adj}_5$

	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

Se  $\text{Caminho}[i, j] = 1$  - existe um caminho entre o nó  $i$  e o nó  $j$

27

386

## Grafos

representação

Representação de grafos ponderados

**Matriz de adjacência:** dado um grafo  $G = (N, A)$  de  $n$  nós, a sua matriz de adjacência  $MA$  é uma matriz  $n \times n$  com a propriedade de  $M(i, j) = x$  se existe um arco do nó  $n_i$  para o nó  $n_j$ , sendo  $x$  o peso associado ao arco do nó  $n_i$  para  $n_j$ . Caso contrário  $M(i, j) = \infty$ .

	1	2	3	4
1	1	2	3	4
2	$\infty$	6	$\infty$	$\infty$
3	8	$\infty$	2	16
4	$\infty$	$\infty$	3	$\infty$

28

387

## Grafos

representação

Um problema da representação de grafos através da matriz de adjacência

**Grafo**

**Matriz de adjacência**

	Porto	Aveiro	Viseu	Coimbra	Lisboa	Évora	Setúbal	Beja	Faro
Porto	0	60	140						
Aveiro	60	0	75	82					
Viseu	140	75	0	85					
Coimbra		82	85	0	198				
Lisboa			198	0	44	186			
Évora				44	0	135	190		
Setúbal					186	135	0	165	
Beja						190	165	0	
Faro									0

- Responder a questões do tipo:
- • Quantas arestas existem neste grafo?
- • O grafo é conexo?

→ algoritmos de Ordem  $n^2$  i.e., é necessário examinar todos os elementos da matriz, incluindo os nulos

29

388

## Grafos

representação

Outro problema da representação de grafos através da matriz de adjacência

**Grafo**

- É necessário conhecer previamente o número de nós existentes no grafo.
- O que fazer no caso de ser necessário aumentar o número de nós?

**Novos nós:**

- Braga
- Tomar
- Sines
- Lagos
- Tavira

30

389

## Grafos

representação

Ainda outro probl. da represent. de grafos através da matriz de adjacência

	Porto	Aveiro	Viseu	Coimbra	Lisboa	Évora	Setúbal	Beja	Faro	Braga	Tomar	Sines	Lagos	Tavira
Porto	0	60	140											
Aveiro	60	0	75	82										
Viseu	140	75	0	85										
Coimbra		82	85	0	198									
Lisboa				198	0					44	186			
Évora										44	0	135	190	
Setúbal										186	135	0	165	
Beja														190
Faro														165
Braga														
Tomar														
Sines														
Lagos														
Tavira														

Há tendência para a matriz de adjacência ficar esparsa

31

390

## Grafos

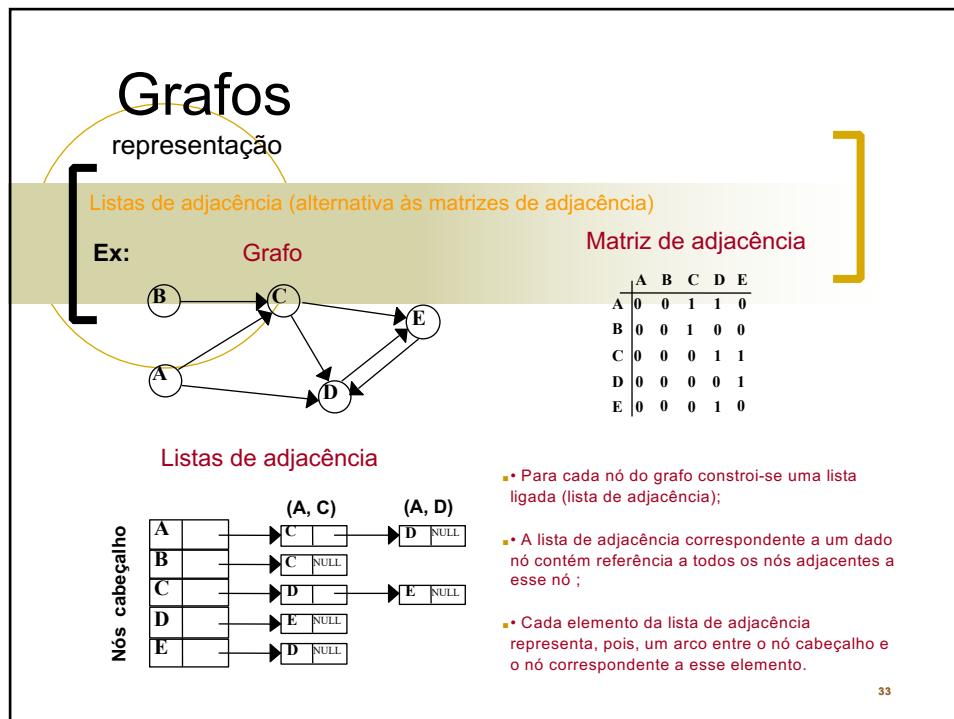
representação

Resumindo... nas matrizes de adjacências temos:

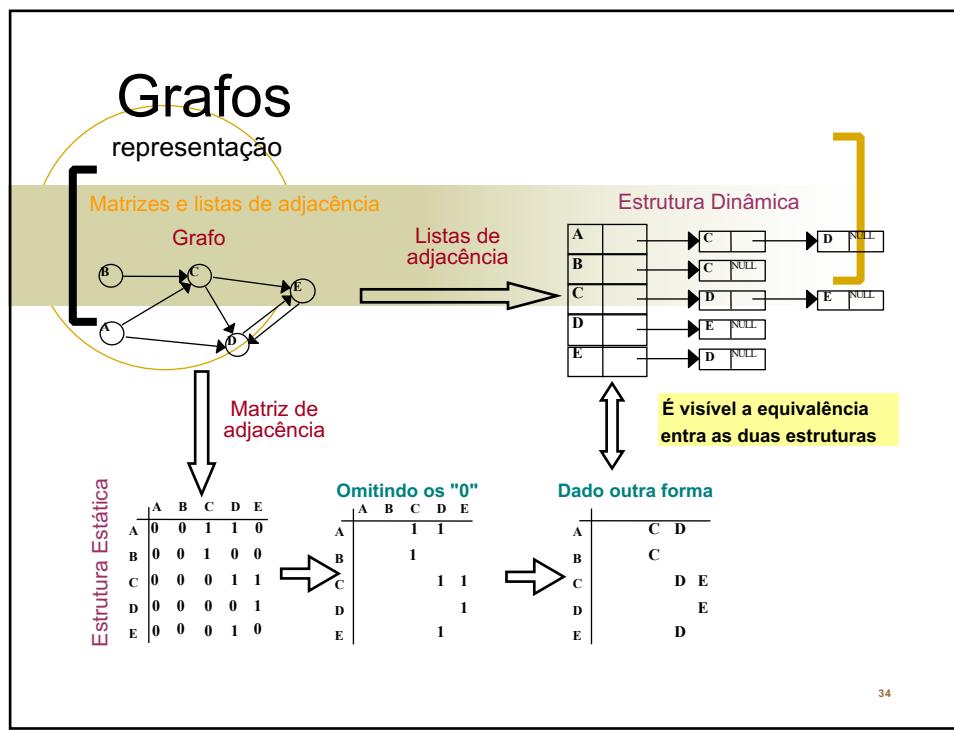
- Ineficiente na alteração dinâmica do número de nós - na prática o número de nós tem de ser conhecido previamente (ou, pelo menos, majorado);
- Os algoritmos de manipulação do grafo podem ter complexidade inadequada (Ordem  $N^2$ );
- A matriz de adjacência pode ser esparsa implicando elevados custos de armazenamento.

32

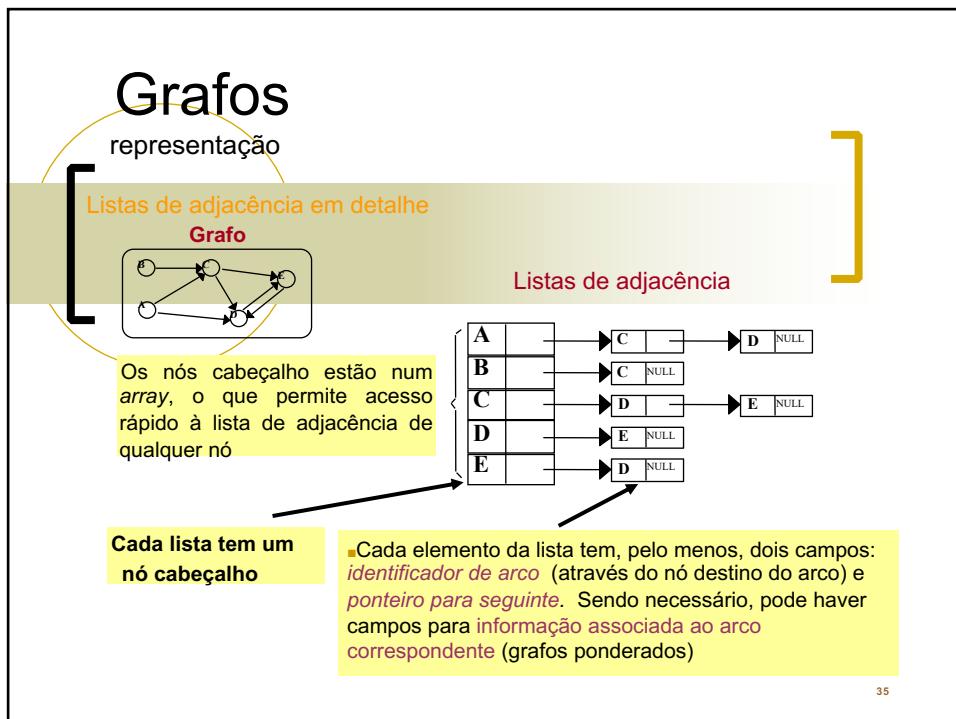
391



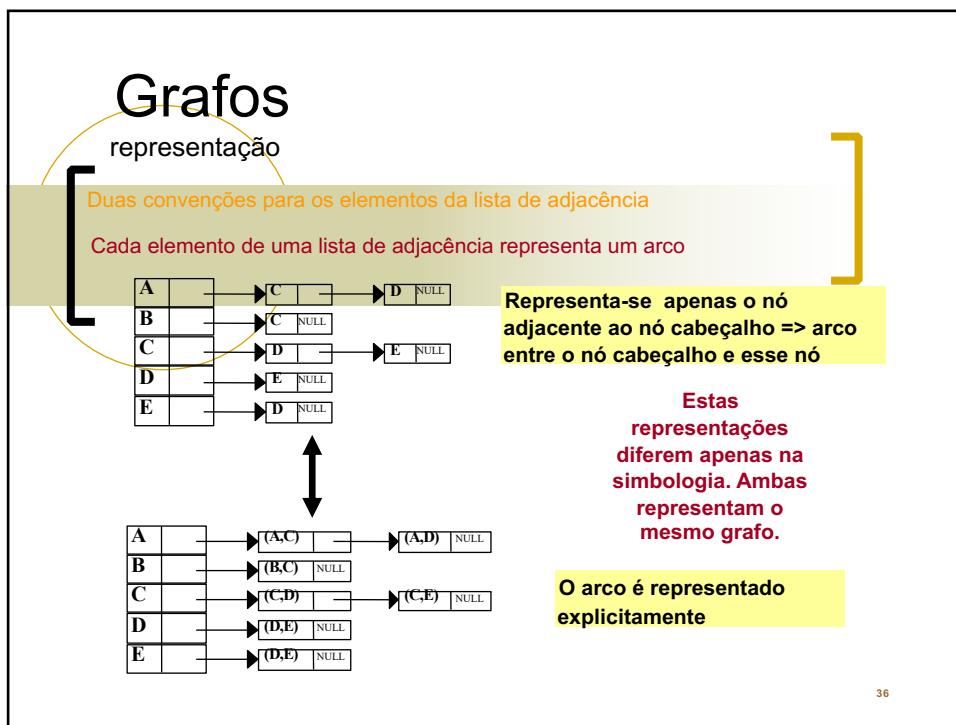
392



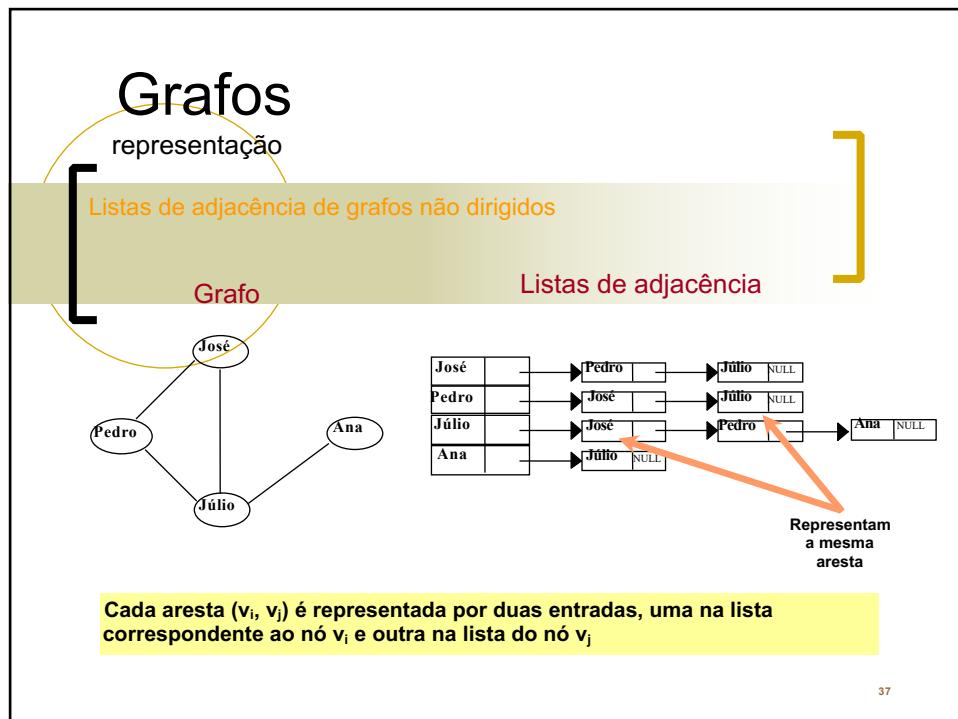
393



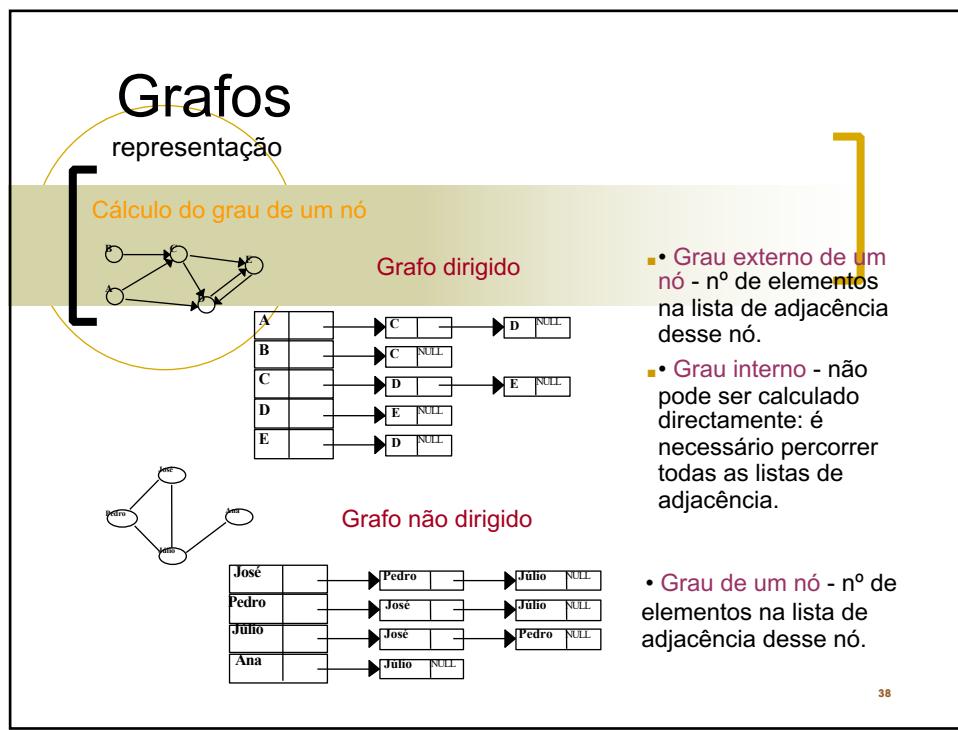
394



395



396



397

**Grafos**

representação

Vantagens das listas de adjacência

- Reduzem o tempo necessário para efectuar certas operações sobre os grafos.

Ex: Determinar quantos arcos tem um grafo de  $n$  nós e  $a$  arcos é uma operação de Ordem ( $n^2$ ) através da matriz de adjacência e de Ordem ( $n + a$ ) através das listas de adjacência

▪ Permitem economizar espaço para o caso de grafos cujas matrizes de adjacência são esparsas.

39

398

**Grafos**

representação

Desvantagens das listas de adjacência

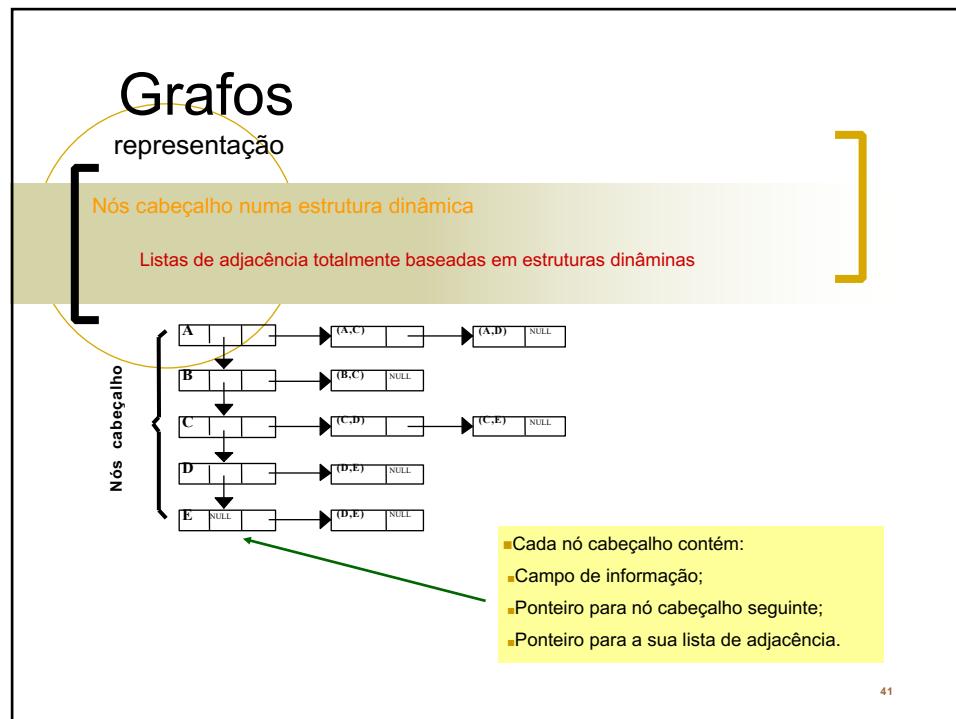
- Certas operações básicas são mais complexas do que no caso da matriz de adjacência.

Ex: ligar dois nós através de um arco ou remover o arco entre dois nós são operações em listas, enquanto que na matriz de adjacência consistem em alterar apenas um elemento da matriz

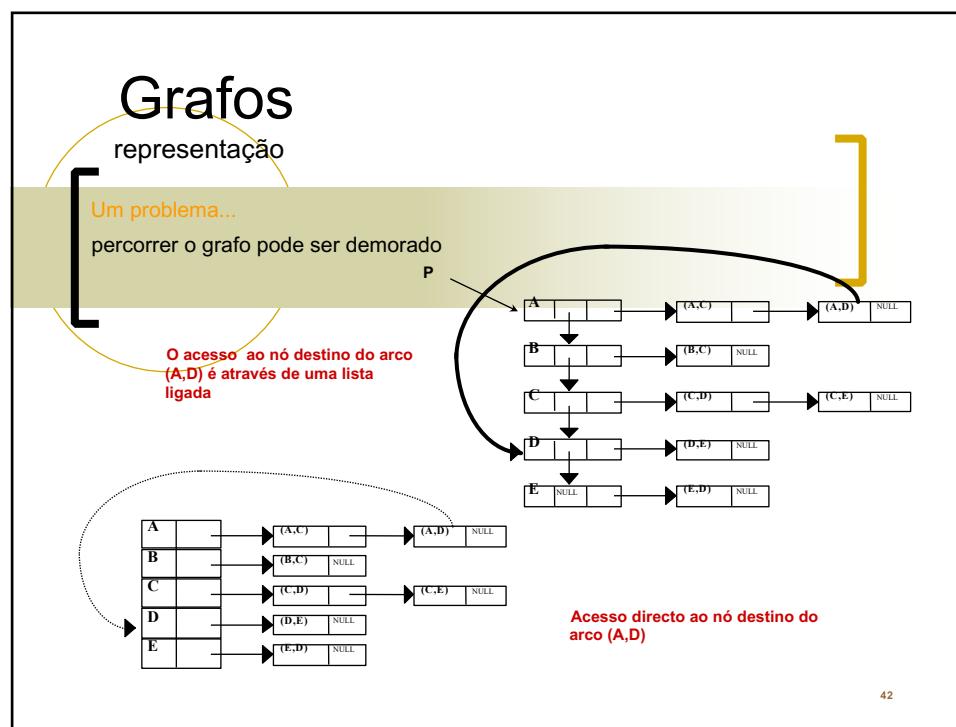
▪ Os nós cabeçalho estão num array (estático) o que causa dificuldades no aumento dinâmico do número de nós do grafo (pode não ser possível saber qual o tamanho adequado desse array no momento da sua criação).

40

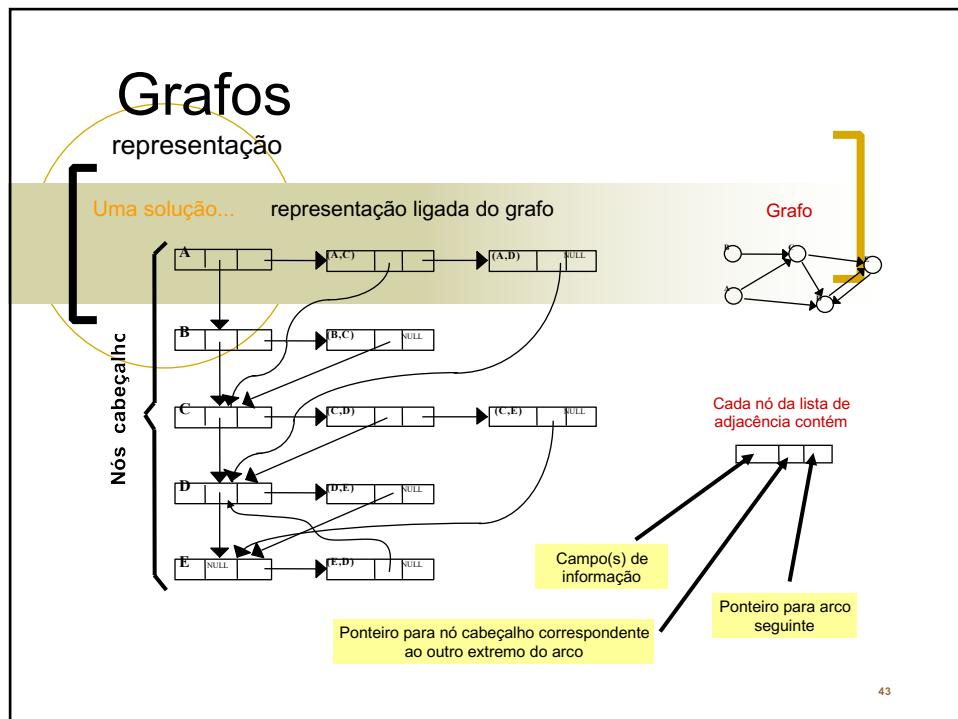
399



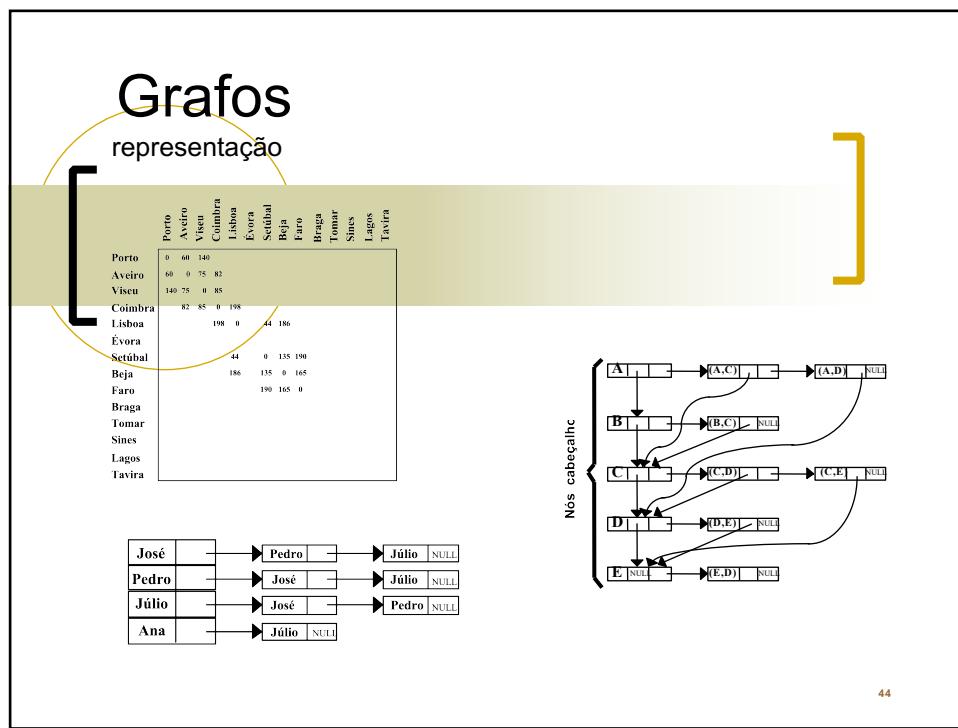
400



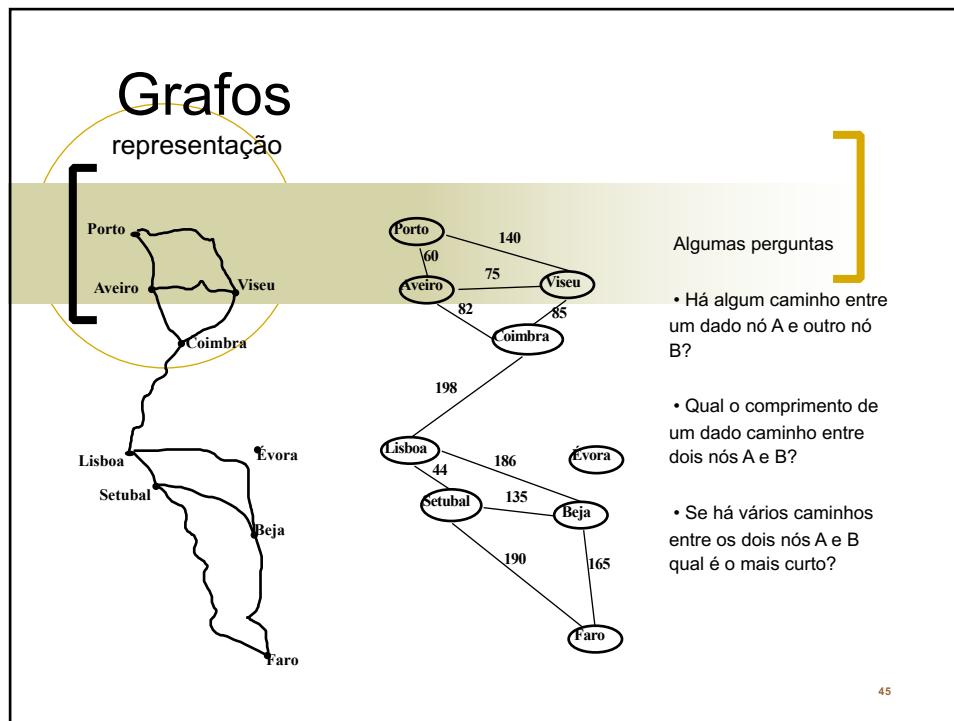
401



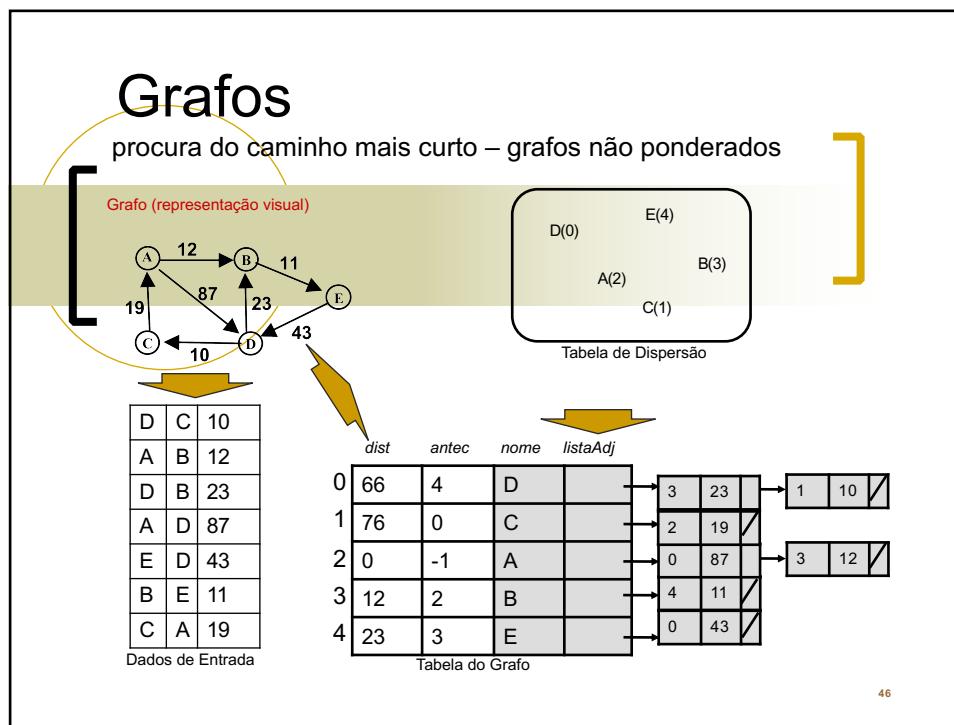
402



403



404



405

## Grafos

procura do caminho mais curto – grafos não ponderados

**Problema**

Caminho mais curto num grafo não ponderado

Determinar o caminho mais curto (definido pelo número de arcos) do nó S para todos os nós.

Procura primeiro em largura (por níveis)

Explora os nós por camadas em que:

$D_v$  →  $D_w = D_v + 1$  se  $D_w = \infty$

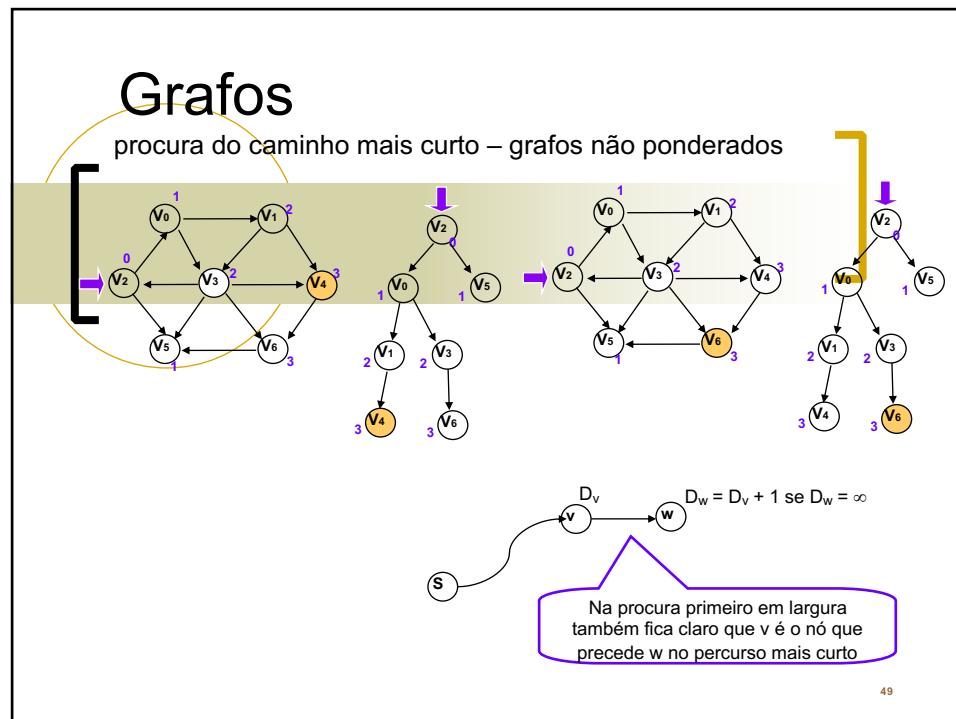
47

406

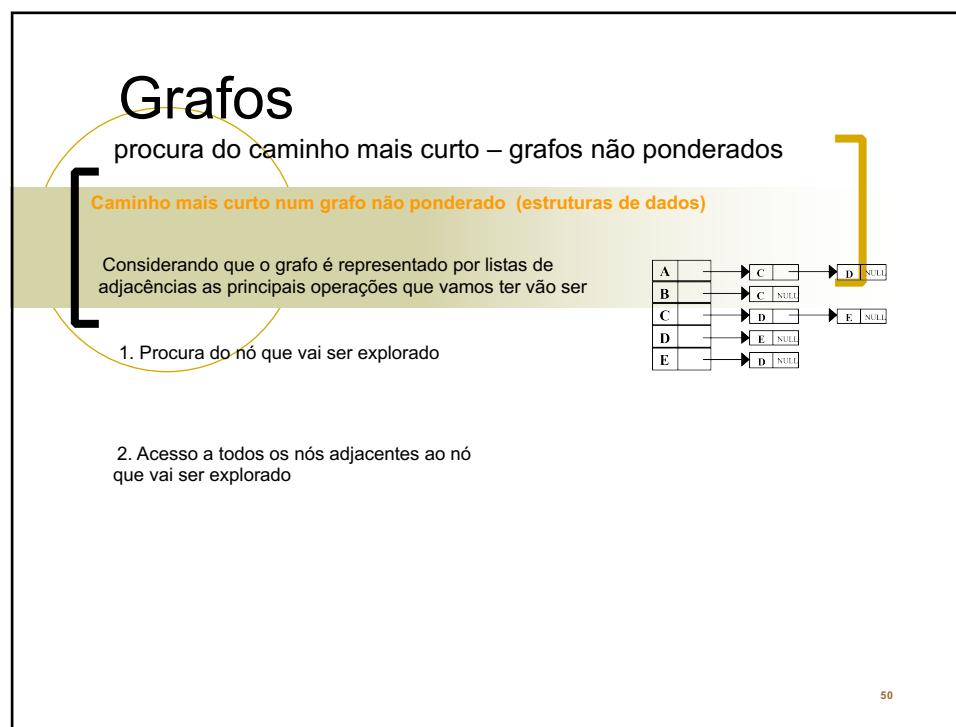
## Grafos

procura do caminho mais curto – grafos não ponderados

407



408



409

## Grafos

procura do caminho mais curto – grafos não ponderados

Caminho mais curto num grafo não ponderado (estruturas de dados)

Colocar cada nó cujo valor D foi actualizado numa fila de espera.

O próximo nó a ser explorado é o nó na cabeça da fila de espera.

No início a fila está vazia e o primeiro nó que vai receber é o nó S.

Como cada nó vai entrar e sair uma única vez na fila de espera temos que este processo tem complexidade  $O(N)$ .

O custo de procurar o caminho mais curto usando procura prima em largura vai ser dominado pela procura na lista de adjacências ou seja  $O(A)$ , isto para cada nó ainda não explorado.

...resultando numa complexidade  $O(NxA)$ .

51

410

## Grafos

procura do caminho mais curto – grafos ponderados

Dependendo do número de vezes que as etiquetas com o valor de distância são actualizadas temos:

- Métodos de **fixação de etiquetas**  
cada vértice uma vez etiquetado não volta a ser processado  
(só aplicável a grafos com pesos positivos)

- Métodos de **revisão de etiquetas**  
cada vértice pode ser re-etiquetado quando for necessário  
(aplicável a grafos com pesos negativos e com ciclos só com pesos positivos)

52

411

# Grafos

procura do caminho mais curto – grafos ponderados

Tanto os **métodos fixação** como de **correcção de etiquetas** podem ser genericamente descritos da seguinte forma (Gallo and Pallottino 86):

```

genericShortestPathAlgorithm( weighted simple digraph, vertex first)
for all vertices v
    currDist(v) = ∞;
    currDist(first) = 0;
    initialize toBeChecked;
    while toBeChecked is not empty
        v = a vertex in toBeChecked;
        remove v from toBeChecked;
        for all nodes u adjacent to v
            if currDist(u) > currDist(v)+weigth(edge(v u))
                currDist(u) = currDist(v)+weigth(edge(v u));
                predecessor(u) = v;
                add u to toBeChecked if it is not there;

```

**QUESTÕES EM ABERTO:: estrutura de toBeChecked**

**QUESTÕES EM ABERTO:: ordem de atribuição a v de valores de toBeChecked**

53

412

# Grafos

procura do caminho mais curto – grafos ponderados

Cada nó etiquetado pela distância corrente e pela indicação do seu antecessor:

$$\text{label}(v) = (\text{currDist}(v), \text{predecessor}(v))$$

Recordar....

Grafo (representação visual)

Dados de Entrada

	D	C	10
A	B	12	
D	B	23	
A	D	87	
E	D	43	
B	E	11	
C	A	19	

Tabela do Grafo

	dist	antec	nome	listaAdj
0	66	4	D	
1	76	0	C	
2	0	-1	A	
3	12	2	B	
4	23	3	E	

Dicionário

	D(0)	E(4)
D(0)	A(2)	B(3)
E(4)	C(1)	

54

413

# Grafos

## algoritmo de dijkstra

- o São explorados caminhos  $p_1, \dots, p_n$  com origem em  $v$ ;
- o a cada momento o caminho mais curto  $p_i$  entre  $p_1, \dots, p_n$  é escolhido;
- o é acrescentada uma aresta  $a$  a  $p_i$ ;
- o se  $p_i + a$  deixou de ser o caminho mais curto então é abandonado e o novo caminho mais curto é retomado;
- o cada vértice só é explorado uma vez;
- o após todos os vértices terem sido explorados o processo termina.

```

genericShortestPathAlgorithm( weighted simple digraph, vertex first)
for all vertices v
    currDist(v) = ∞;
    currDistFirst() = 0;
    initialize toBeChecked;

    while toBeChecked is not empty
        v = a vertex in toBeChecked;
        remove v from toBeChecked;

        for all nodes u adjacent to v
            if currDist(u) > currDist(v)+weight(edge(v u))
                currDist(u) = currDist(v)+weight(edge(v u));
                predecessor(u) = v;
                add u to toBeChecked if it is not there;

```

55

414

# Grafos

## algoritmo de dijkstra

```

DijkstraAlgorithm( weighted simple digraph,
vertex first)
for all vertices v
    currDist(v) = ∞;
    currDist(first) = 0;
    toBeChecked = all vertices;

    while toBeChecked is not empty
        v = a vertex in toBeChecked with minimal currDist(v);
        remove v from toBeChecked;

        for all nodes u adjacent to v and in toBeChecked
            if currDist(u) > currDist(v)+weight(edge(v u))
                currDist(u) = currDist(v)+weight(edge(v u));
                predecessor(u) = v;
                add u to toBeChecked if it is not there;

```

56

415

# Grafos

procura do caminho mais curto – **grafos ponderados**

Dependendo do número de vezes que as etiquetas com o valor de distância são actualizadas temos:

- Métodos de **fixação de etiquetas**  
cada vértice uma vez etiquetado não volta a ser processado  
(só aplicável a grafos com pesos positivos)
- Métodos de **revisão de etiquetas**  
cada vértice pode ser re-etiquetado quando for necessário  
(aplicável a grafos com pesos negativos e com ciclos só com pesos positivos)

57

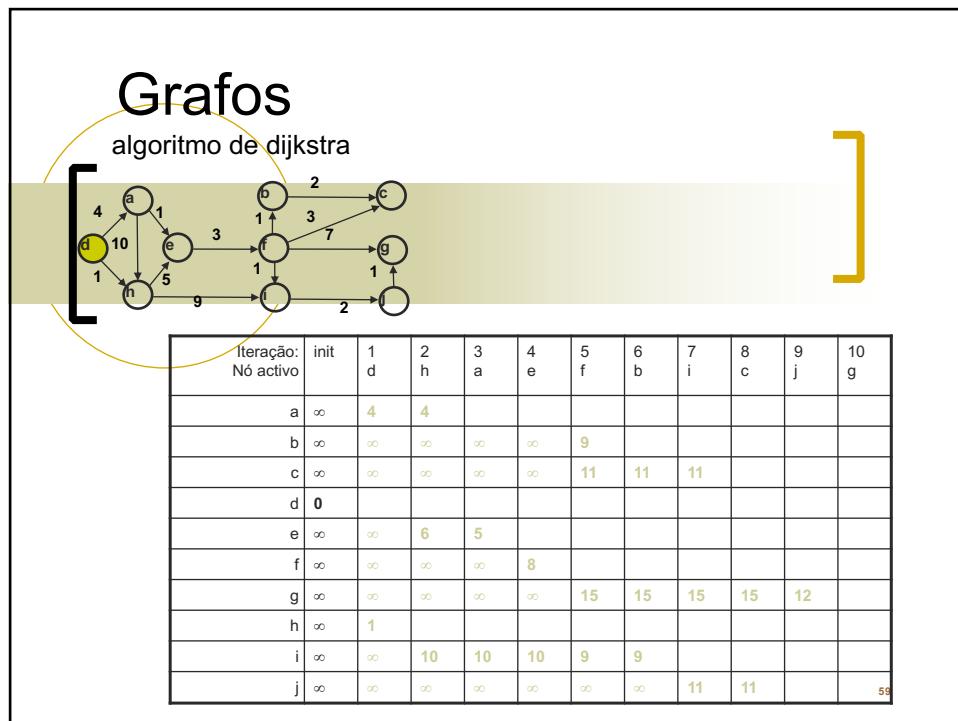
416

# Grafos

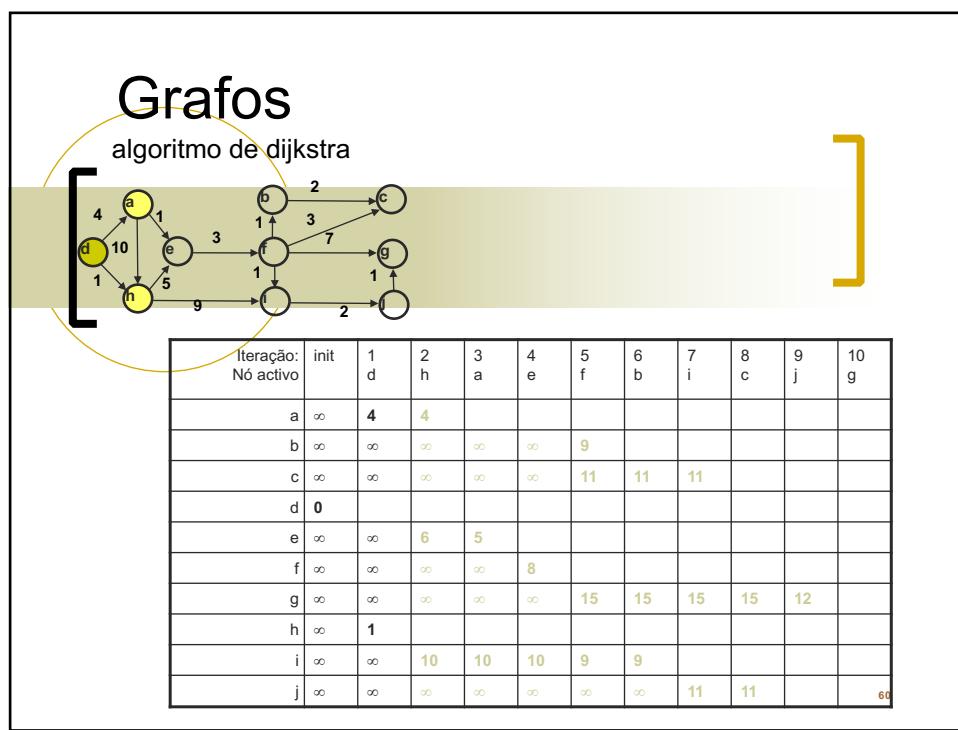
algoritmo de dijkstra

Iteração: Nó activo	init	1 d	2 h	3 a	4 e	5 f	6 b	7 i	8 c	9 j	10 g
a	$\infty$	4	4								
b	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	9					
c	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	11	11	11			
d	0										
e	$\infty$	$\infty$	6	5							
f	$\infty$	$\infty$	$\infty$	$\infty$	8						
g	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	15	15	15	15	12	
h	$\infty$	1									
i	$\infty$	$\infty$	10	10	10	9	9				
j	$\infty$	11	11		58						

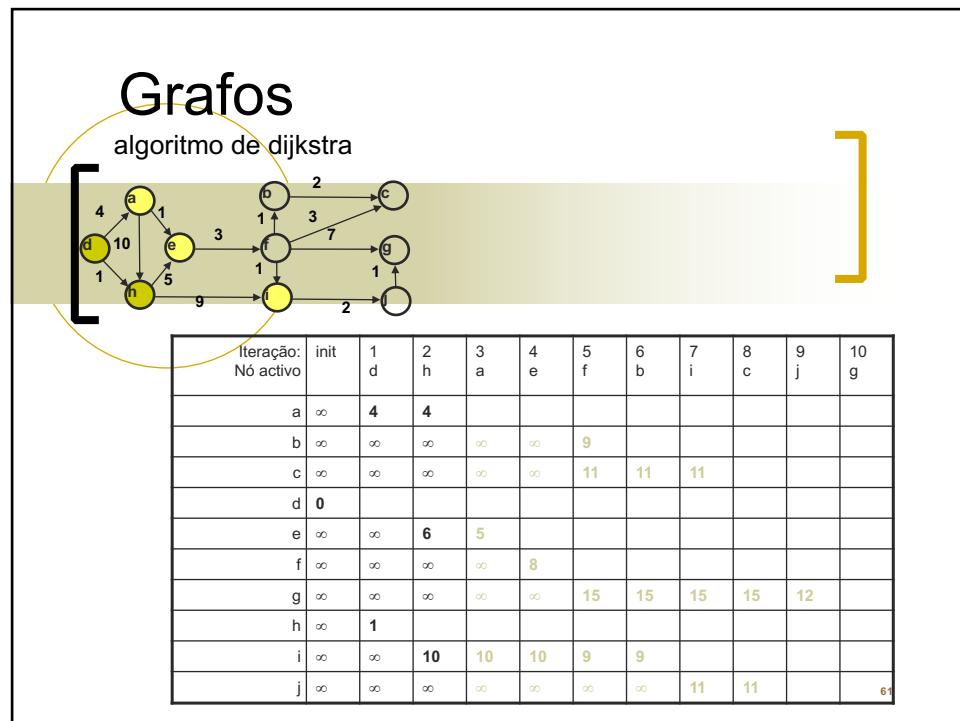
417



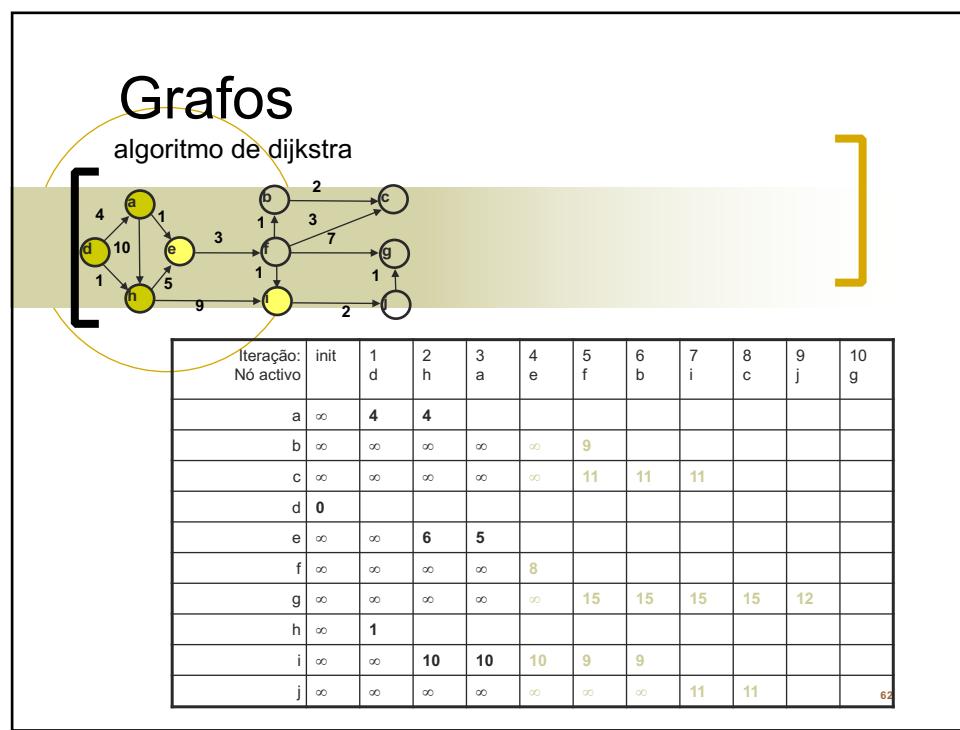
418



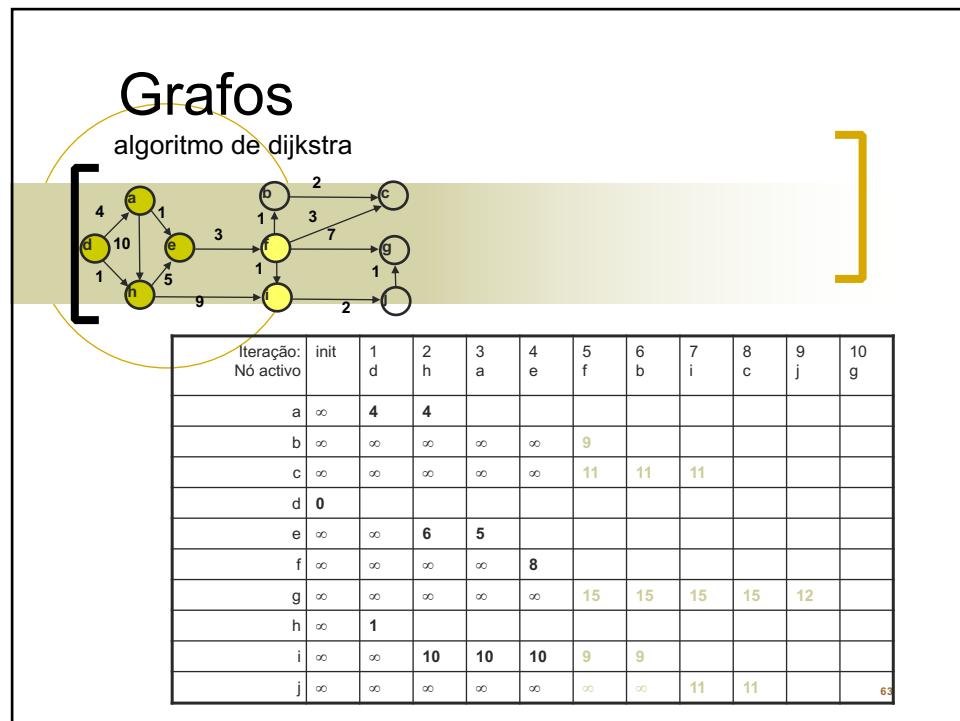
419



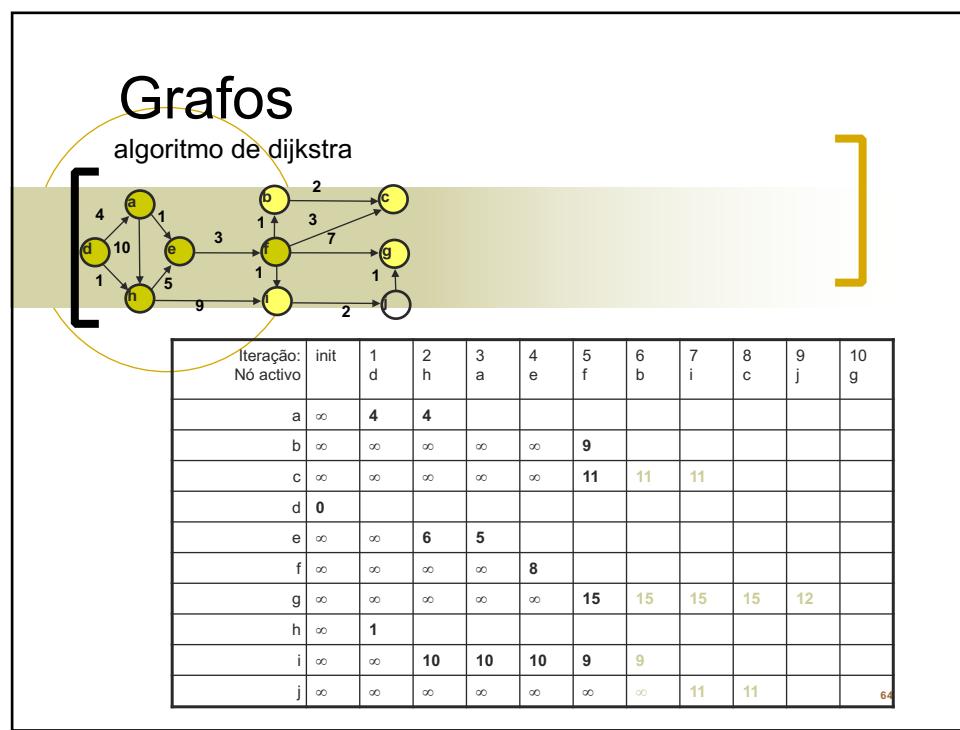
420



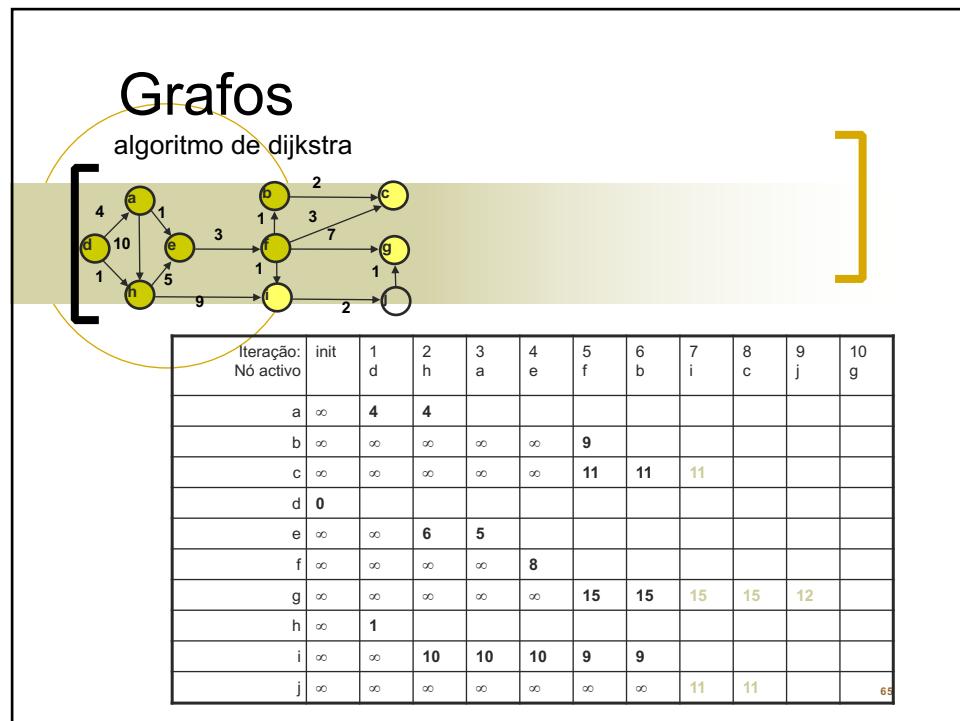
421



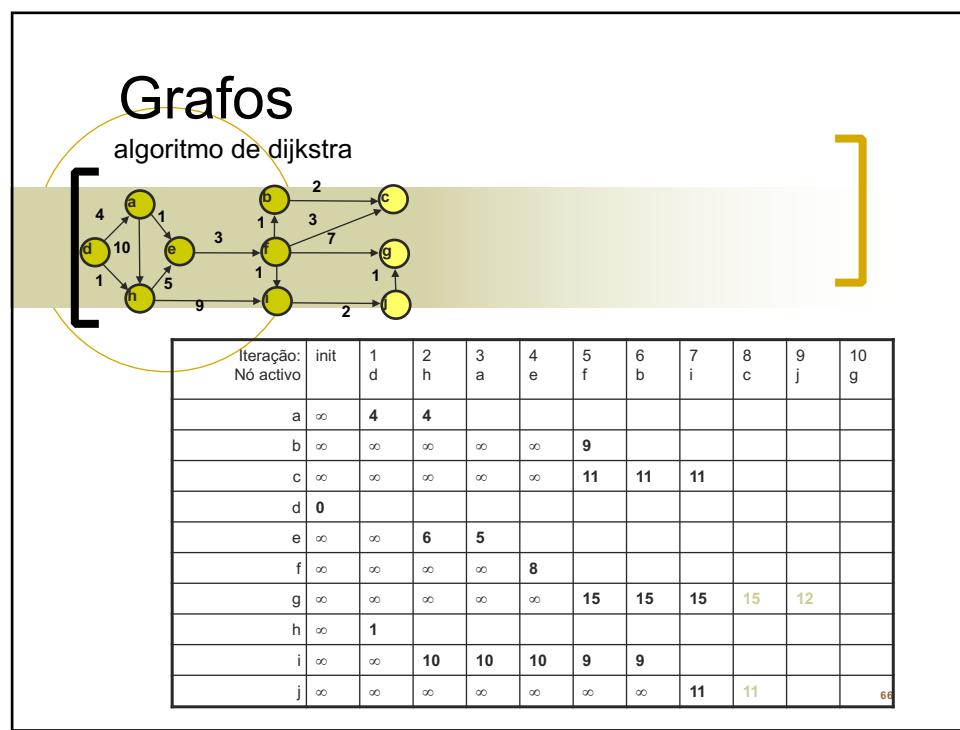
422



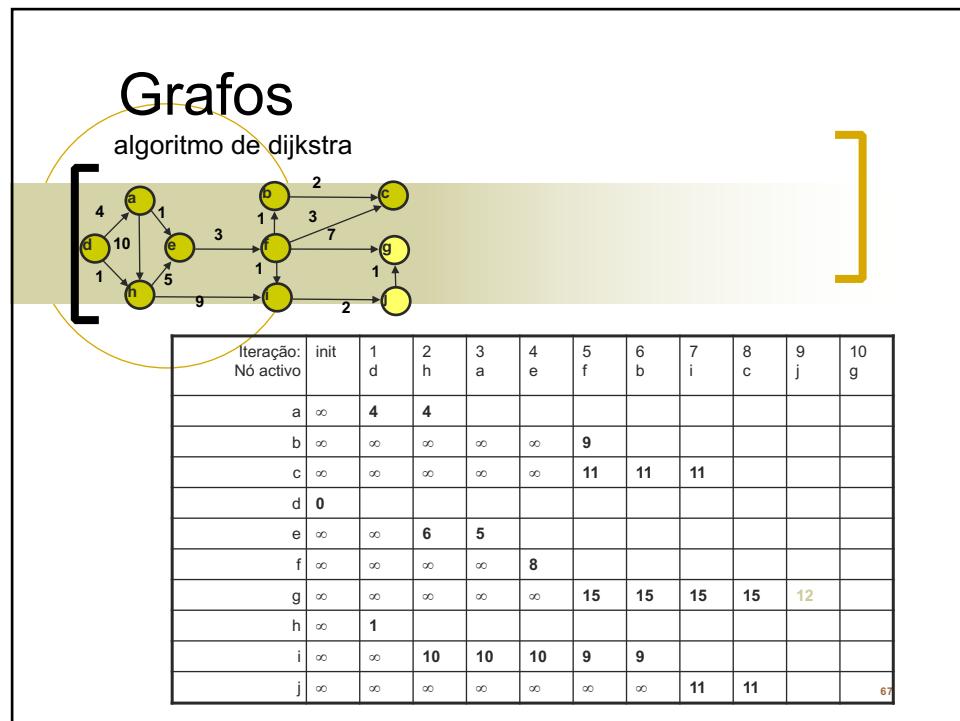
423



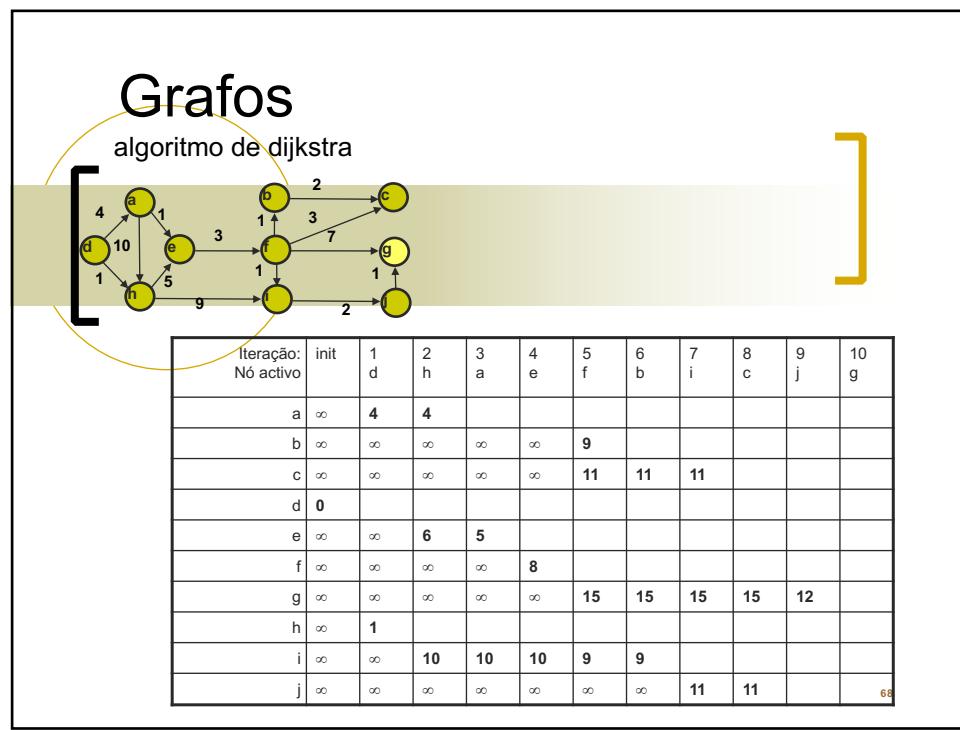
424



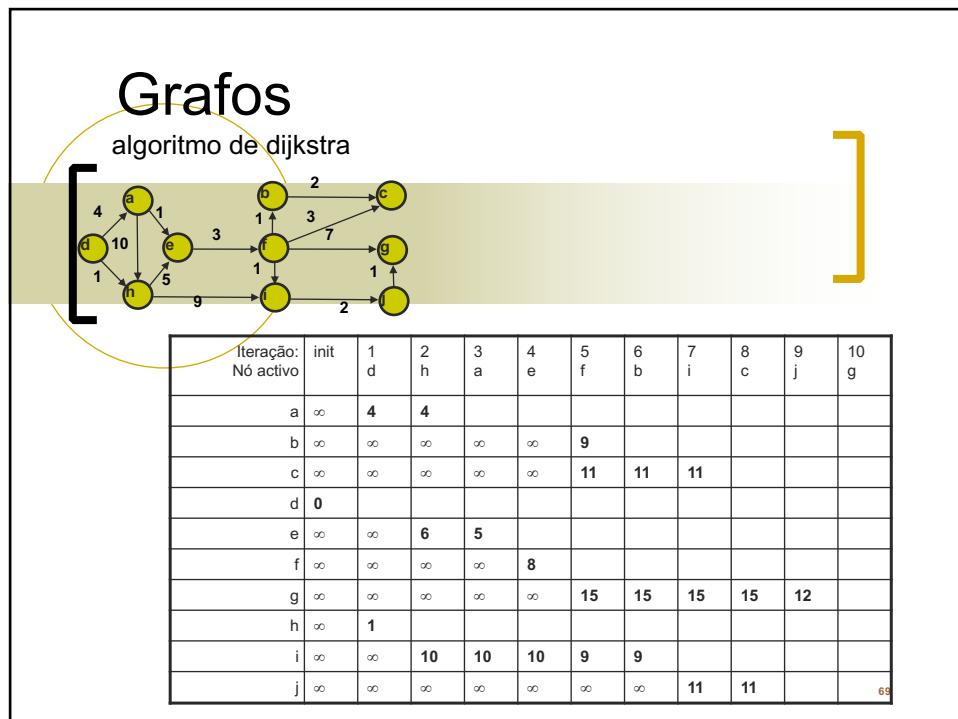
425



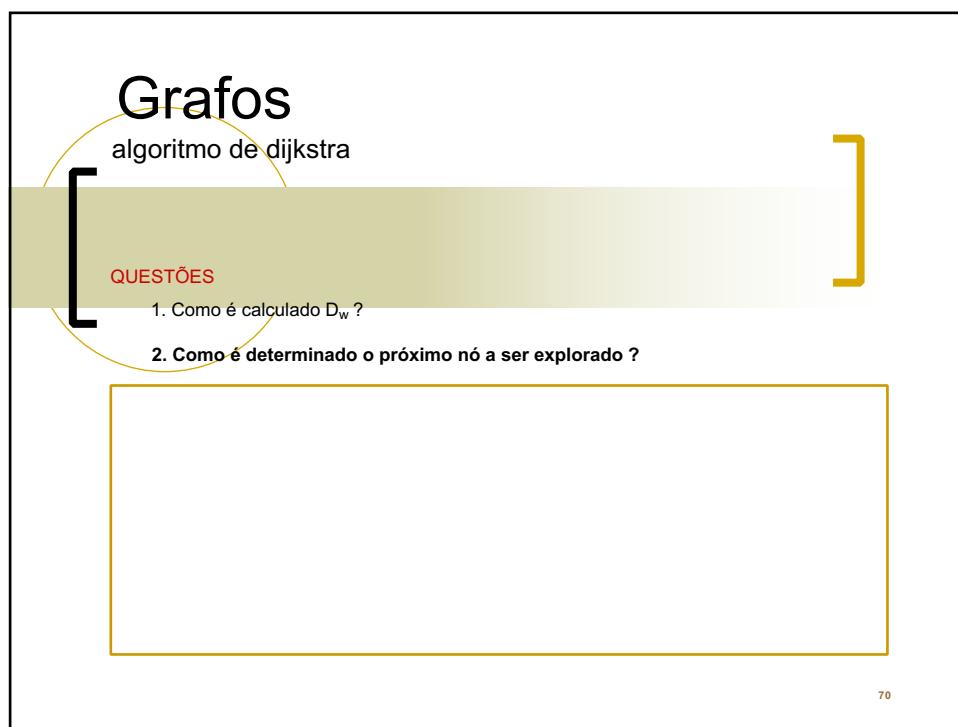
426



427



428



429

**Grafos**

algoritmo de dijkstra

[ QUESTÕES ]

1. Como é calculado  $D_w$  ?
2. **Como é determinado o próximo nó a ser explorado ?**
2. Criar uma fila de prioridades.  
Sempre que um nó tem o seu valor  $D_w$  reduzido coloca (nó, $D_w$ ) na fila de prioridades.

Para seleccionar um novo nó para explorar remove o item com menor valor  $D_w$  (distância à origem) na lista de prioridades. Analisa os vizinhos directos até que um nó ainda não explorado seja encontrado. Esse é o novo nó a explorar.

70

430

**Grafos**

algoritmo de dijkstra – Demos na Web

[ ]

<http://weierstrass.is.tokushima-u.ac.jp/ikeda/suuri/dijkstra/DijkstraApp.shtml?demo2>

71

431

## Grafos

algoritmo de dijkstra

Iteração:	init	1	2	3	4	5	6	7	8	9	10
Nó activo		d	h	a	e	f	b	i	c	j	g
a	$\infty$	4	4								
b	$\infty$		$\infty$				9				
c	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	11	11	11		
d	0										
e	$\infty$	$\infty$	6	5							
f	$\infty$	$\infty$	$\infty$	$\infty$	8						
g	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	15	15	15	15	12	
h	$\infty$	1									
i	$\infty$	$\infty$	10	10	10	9	9				
j	$\infty$	11	11								

DESV.: o algoritmo de dijkstra falha quando temos pesos negativos !!

432

## Grafos

procura do caminho mais curto – grafos ponderados c/ p. neg.

Problema  
Determinar o caminho mais curto (definido pelo custo total associado aos arcos que compõem o caminho) do nó S para todos os nós.  
Considerar que os arcos podem ter custos negativos.

Consideremos o seguinte grafo:

Duas situações críticas:

- Quando um vértice v é processado pode haver um outro vértice u (não processado) a partir do qual voltando a v tenhamos um caminho de custo mais baixo.
- Existência de ciclos de custo negativo

433

# Grafos

procura do caminho mais curto – grafos ponderados c/ p. neg.

```

labelCorrectingAlgorithm( weighted simple digraph,
vertex first)

for all vertices v
currDist(v) = ∞;
currDist(first) = 0;
toBeChecked = { first };

while toBeChecked is not empty
v = a vertex in toBeChecked;
remove v from toBeChecked;

for all nodes u adjacent to v
if currDist(u) > currDist(v)+weight(edge(v u))
currDist(u) = currDist(v)+weight(edge(v u));
predecessor(u) = v;
add u to toBeChecked if it is not there;

genericShortestPathAlgorithm( weighted simple digraph, vertex first)
for all vertices v
currDist(v) = ∞;
currDist(first) = 0;
initialize toBeChecked;

while toBeChecked is not empty
v = a vertex in toBeChecked;
remove v from toBeChecked;

for all nodes u adjacent to v
if currDist(u) > currDist(v)+weight(edge(v u))
currDist(u) = currDist(v)+weight(edge(v u));
predecessor(u) = v;
add u to toBeChecked if it is not there;

```

**aplicável a grafos com pesos negativos e com ciclos com pesos positivos**

74

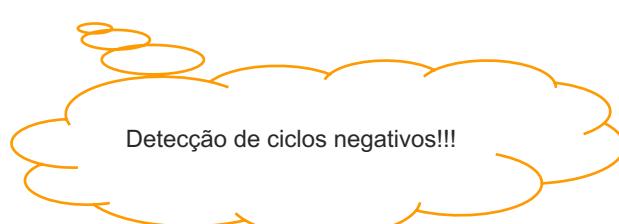
434

# Grafos

procura do caminho mais curto – grafos ponderados c/ p. neg.

**Observações:**

- Quando  $D_w$  tem o seu valor alterado este nó deve ser revisitado no futuro, logo recolocá-lo na fila de nós a explorar com  $D_w = D_v + c_{v,w}$
- Quando um nó  $v$  é explorado pela  $i$ -ésima vez, significa que existem pelo menos  $i$  caminhos diferentes entre o nó origem e o nó  $v$ . Assim não havendo ciclos negativos, o número máximo possível de visitas será igual ao número de arestas, pois caso fosse superior, teria que existir um ciclo.



75

435

# Grafos

procura do caminho mais curto – grafos ponderados c/ p. neg.

```
/*
 * Run shortest path algorithm;
 * Negative edge weights are allowed.
 * Return false if negative cycle is detected
 */
private boolean negative( int startNode )
{
    int v, w;
    Queue q = new QueueAr();
    int cvw;

    clearData();
    table[ startNode ].dist = 0;
    q.enqueue( new Integer( startNode ) );
    table[ startNode ].scratch++;

    try
    {
        while( !q.isEmpty() )
        {
            v = ( (Integer) q.dequeue() ).intValue();
            if( table[ v ].scratch++ > 2 * numVertices )
                return false;

            ListItr p = new LinkedListItr( table[ v ].adj );
            while( p.hasNext() )
            {
                w = ( (Edge) p.next() ).dest;
                cvw = ( (Edge) p.next() ).cost;
                if( table[ w ].dist > table[ v ].dist + cvw )
                {
                    table[ w ].dist = table[ v ].dist + cvw;
                    table[ w ].prev = v;

                    // Enqueue only if not already on queue
                    if( table[ w ].scratch++ % 2 == 0 )
                        q.enqueue( new Integer( w ) );
                    else
                        table[ w ].scratch++; // In effect, adds 2
                }
            }
        }
    } catch( Underflow e ) {} // Cannot happen

    return true;
}
```

76

436

# Grafos

procura do caminho mais curto – grafos ponderados c/ p. neg.

```
for( ; p.isInList( ); p.advance( ) )
{
    w = ( (Edge) p.retrieve( ) ).dest;
    cvw = ( (Edge) p.retrieve( ) ).cost;
    if( table[ w ].dist > table[ v ].dist + cvw )
    {
        table[ w ].dist = table[ v ].dist + cvw;
        table[ w ].prev = v;

        // Enqueue only if not already on queue
        if( table[ w ].scratch++ % 2 == 0 )
            q.enqueue( new Integer( w ) );
        else
            table[ w ].scratch++; // In effect, adds 2
    }
}
catch( Underflow e ) {} // Cannot happen

return true;
}
```

77

437

# Grafos

procura do caminho mais curto – grafos ponderados c/ p. neg.

## Observações:

1. Quando  $D_w$  tem o seu valor alterado este nó deve ser revisitado no futuro, logo recolocá-lo na fila de nós a explorar com  $D_w = D_v + c_{v,w}$
2. Quando um nó  $v$  é explorado pela  $i$ -ésima vez, significa que existem pelo menos  $i$  caminhos diferentes entre o nó origem e o nó  $v$ . Assim não havendo ciclos negativos, o número máximo possível de visitas será igual ao número de arestas, pois caso fosse superior, teria que existir um ciclo.

•Complexidade: cada nó vai ser retirado da fila de nós pelo menos uma vez. Existem  $N$  nós. Cada nó pode ser retirado no máximo  $A$  vezes, sendo  $A$  o número de arestas.

•O algoritmo vai assim ter complexidade  $O(A^N)$  com  $A$  igual ao número de arestas e  $N$  igual ao número de nós.

•Se um vértice é retirado da fila de vértices a explorar mais do que  $N$  vezes, significa que temos um ciclo de custo negativo.

78

438

# Grafos

procura do caminho mais curto – grafos acíclicos

## Problema

Determinar o caminho mais curto (definido pelo custo total associados aos arcos que compõem o caminho) do nó  $S$  para todos os nós.  
Considerar que o grafo não tem ciclos.

## Ordenamento Topológico

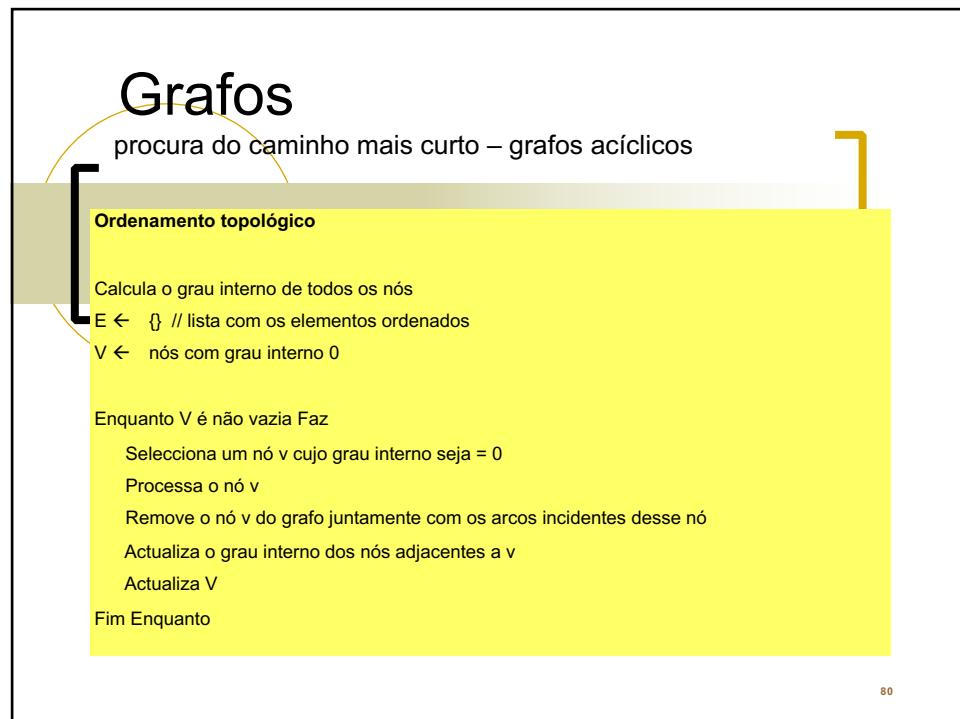
Num ordenamento topológico os nós de um grafo acíclico estão ordenados de tal forma que se existir um caminho de  $u$  para  $v$ ,  $u$  precede  $v$  no grafo.

Ex.: grafo usado para representar as precedências das disciplinas de um curso. Um arco  $(v, w)$  representa que a disciplina  $v$  deve estar concluída antes de frequentar a disciplina  $w$ .

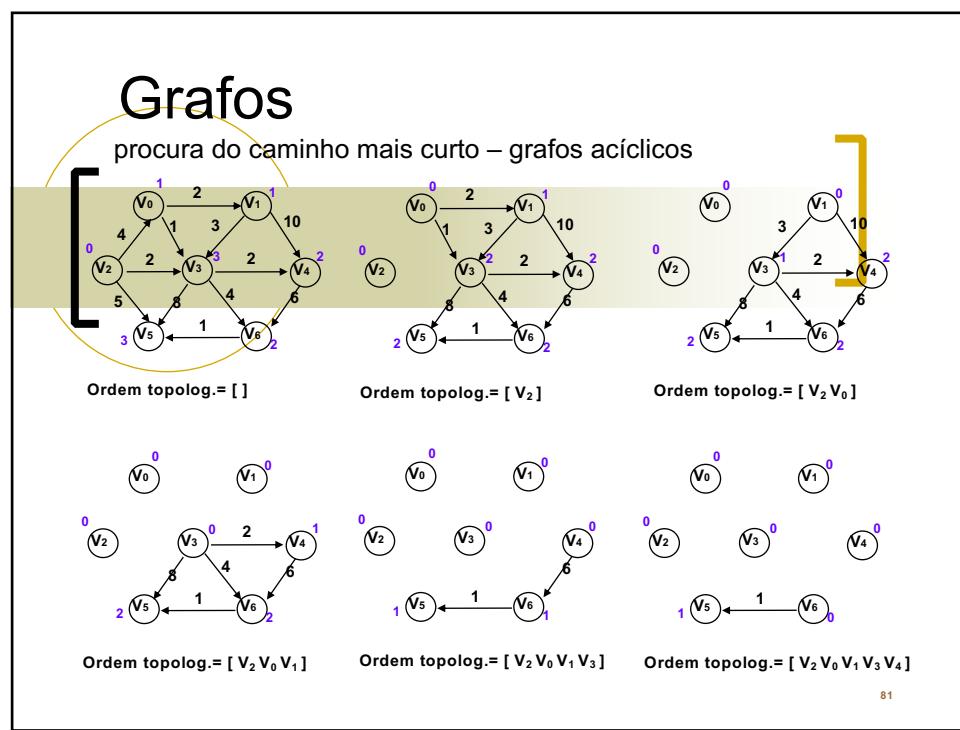
Um grafo pode ter vários ordenamentos topológicos

79

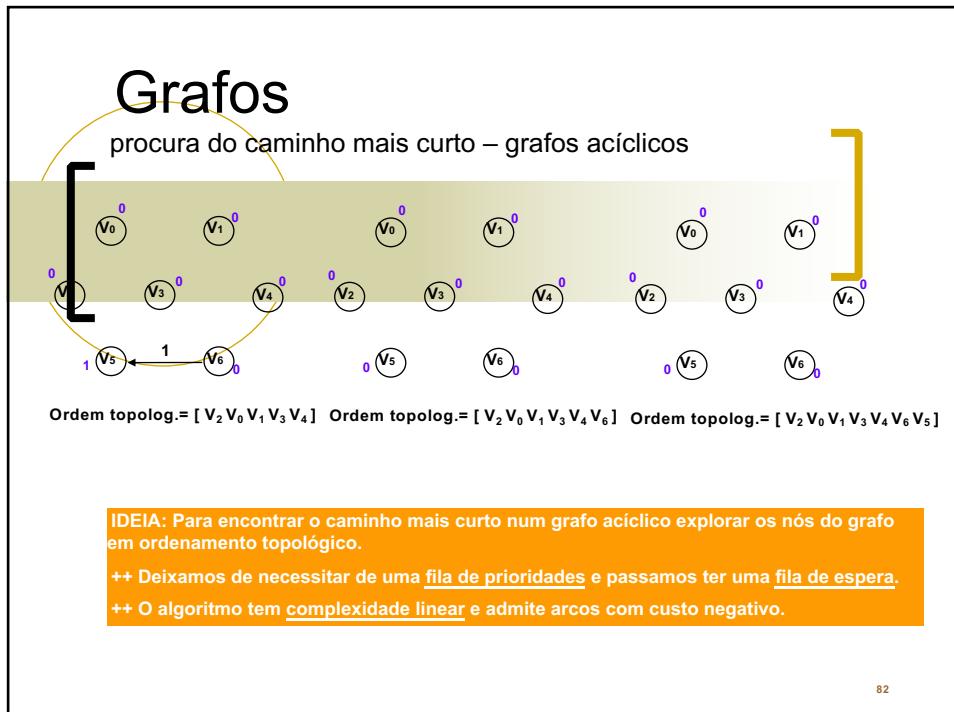
439



440

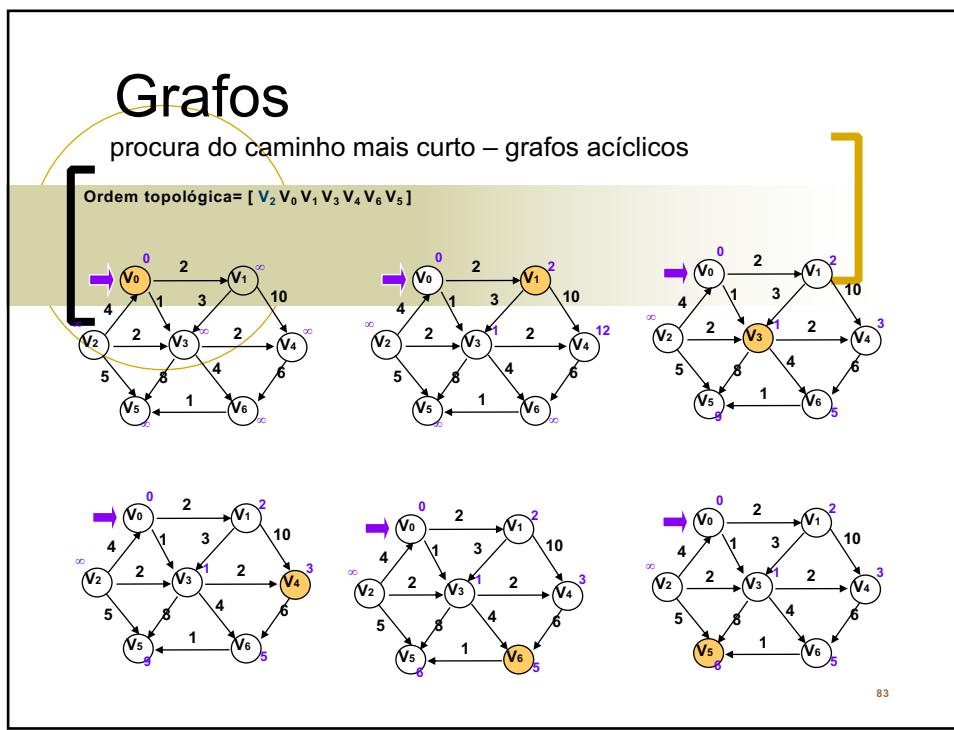


441



82

442



83

443

# Grafos

procura do caminho mais curto – grafos acíclicos

Ordem topológica

<https://www.cs.usfca.edu/~galles/visualization/TopoSortDFS.html>

83

444

# Grafos

- Os grafos são estruturas de dados versáteis adequadas para: análise de circuitos eléctricos; métodos de detecção de erros em computadores; optimização de percursos; análise e planeamento de projectos; identificação de componentes químicos; genética; linguística; ciências sociais; robótica.
- Um grafo pode ser representado por  $G=(N,A)$ , com N conj. de nós e A conj. de ligações (designados por arcos ou arestas).
- Os grafos podem ser dirigidos, não dirigidos, ponderados, não ponderados.
- Alguns conceitos associados aos grafos são: grau, grau interno, grau externo, caminho, ciclo, laço, circuito, subgrafo, grafo parcial, grafo conexo, fortemente conexo, completo, multigrafo.
- Um grafo pode ser representado por uma matriz de adjacências ou por listas de adjacências.
- Na representação por listas de adjacências os cabeçalhos podem estar organizados num array, numa lista ligada ou numa hash table.

84

445

# Grafos

- Os grafos são estruturas aplicadas a um grande número de problemas nomeadamente em circunstâncias em que se pretende encontrar o caminho mais curto entre dois nós.
- Para grafos não ponderados, o caminho mais curto pode ser encontrado fazendo presquisa primeiro em largura.
- Para grafos ponderados com valores positivos, o algoritmo de Dijkstra com recurso a uma fila de prioridades, com tempo de computação ligeiramente superior ao algoritmo para grafos não ponderados.
- Para grafos ponderados com valores negativos recorremos a algoritmos que comportam correcção de etiquetas (vs algoritmos de fixação de etiquetas, de que o Dijkstra é um exemplo) com tempo de computação significativamente superior –  $O(a^*n)$ .
- Para grafos acíclicos com pesos positivos e negativos a complexidade é linear se recorremos a ordenamento topográfico.

85

446

Grafos  
... end ;-)



08 38

456

203