

Algoritmos com **vetores**

Geração, aleatorização, buscas e deslocamentos

Prof. Manuel F. Paradela Ledón



Conceitos gerais sobre vetores

Vetores (arrays, arranjos unidimensionais)

Vetor de n elementos

a_1 a_2 a_3 ... a_n

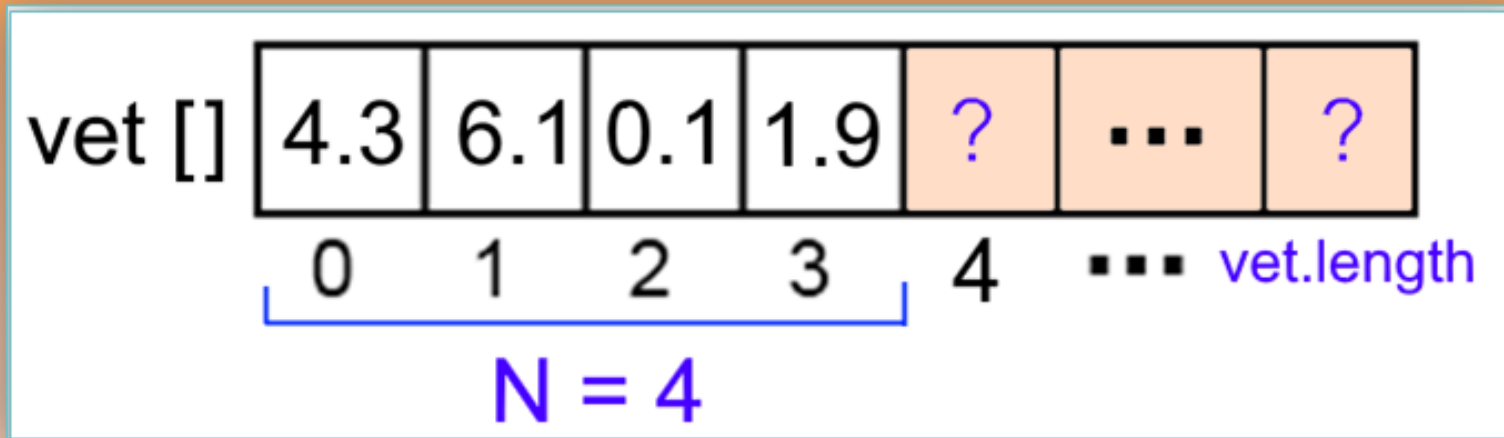
ou

a_0 a_1 a_2 ... a_{n-1}

Exemplo: um vetor de $n = 4$ elementos

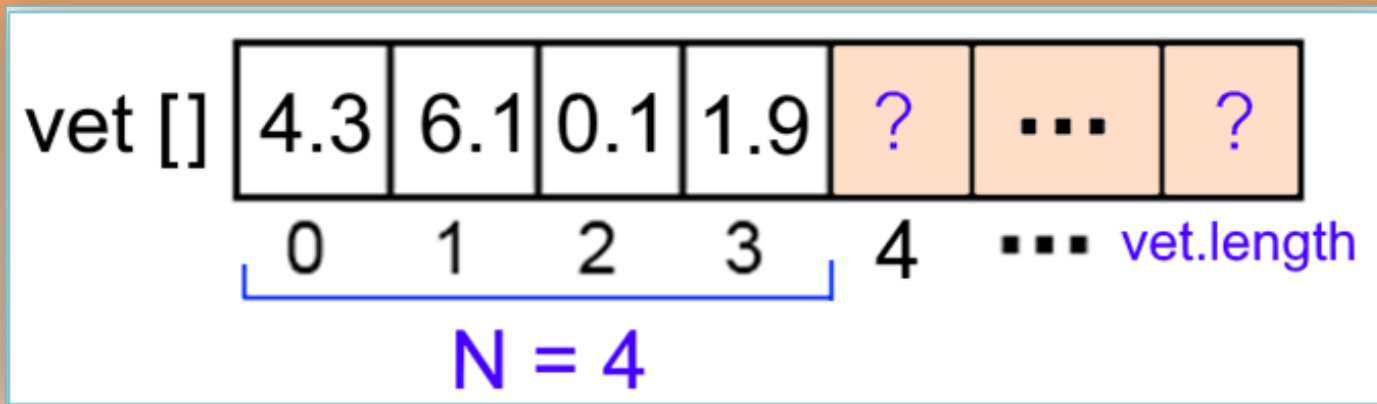
itens:	$a[0]$	$a[1]$	$a[2]$	$a[3]$
	12.3	4.1	-12.7	6.5
posições:	0	1	2	3

Vetores (arrays, arranjos unidimensionais)



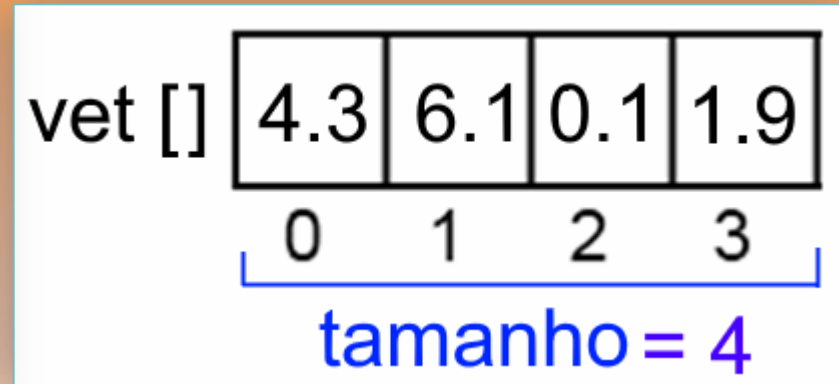
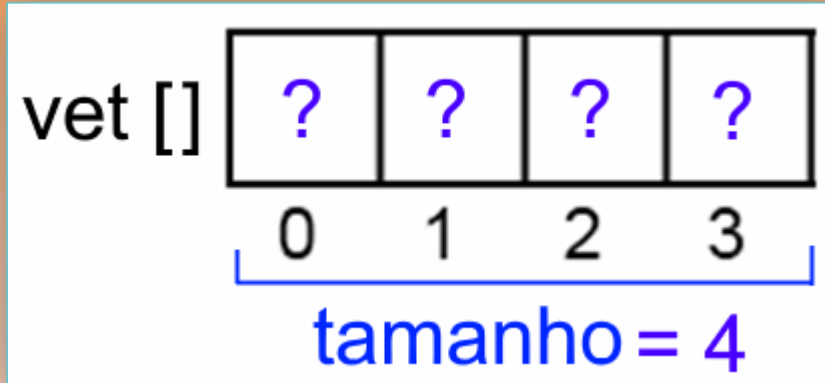
- Podemos alocar memória para o vetor e utilizar somente uma parte do espaço alocado.
- Observe na figura acima que N (4 elementos) é menor que o tamanho real/máximo do vetor. As posições com ? não são utilizadas.
- Algumas linguagens de programação poderiam permitir redimensionar um vetor, mas, na maioria, para efetuar o redimensionamento será necessário criar um novo vetor e efetuar a cópia dos itens/elementos do vetor original para o novo vetor.

Vetores (arrays, arranjos unidimensionais)



- Para acessar um elemento utilizamos a notação `vet[i]`, sendo `vet` o nome do vetor e `i` a posição que queremos referenciar. Por exemplo, com o comando `double v = vet[3];` guardaremos o valor 1.9 na variável `v`.
- Também, para alterar o valor de um elemento do vetor, por exemplo, para guardar o valor 5.2 na posição 2, podemos executar o comando `vet[2] = 5.2;`

Vetores (arrays, arranjos unidimensionais)



- Ao alocar memória para um vetor, os valores dos itens/elementos estão inicialmente indefinidos (1ª figura).
- Posteriormente, utilizaremos algum algoritmo para armazenar valores específicos nas diferentes posições do vetor (2ª figura).
- Nos próximos slides apresentamos algoritmos (programado em Java) para criar e guardar valores aleatórios nas diferentes posições de um vetor.

Colocar, atribuir, valores
aleatórios (quaisquer) a um
vetor

Exemplo: colocando valores aleatórios em um vetor

```
public Ex_random () {
    //Geração aleatória dos itens do vetor:
    float vet [] = new float[10];
    for (int i=0; i < vet.length; i++) {
        vet[i] = geraFloat();
    }
    System.out.println("\nVetor de 10 elementos gerados aleatoriamente");
    System.out.println("com valores entre 0 e 9,999: \n");
    visualizaVetor(vet);
}


public float geraFloat() {
    //Oracle, nextFloat(): "Retorna o próximo valor real pseudo-aleatório uniformemente
    //distribuído entre 0,0 e 1,0 a partir da sequência deste gerador de números aleatórios."
    Random rnd = new Random();
    float numero = rnd.nextFloat();
    return (numero * 10);
}

public void visualizaVetor(float vetor[]) {
    for (int i=0; i < vetor.length; i++) {
        System.out.print(vetor[i] + "   ");
    }
    System.out.println();
}
```

Exemplo em Ex_random.java

Reposicionar (embaralhar) os valores já armazenados em um vetor (aleatorização)

Algoritmo para 'organizar' (embaralhar) um vetor aleatoriamente. Algoritmo de Fisher-Yates.

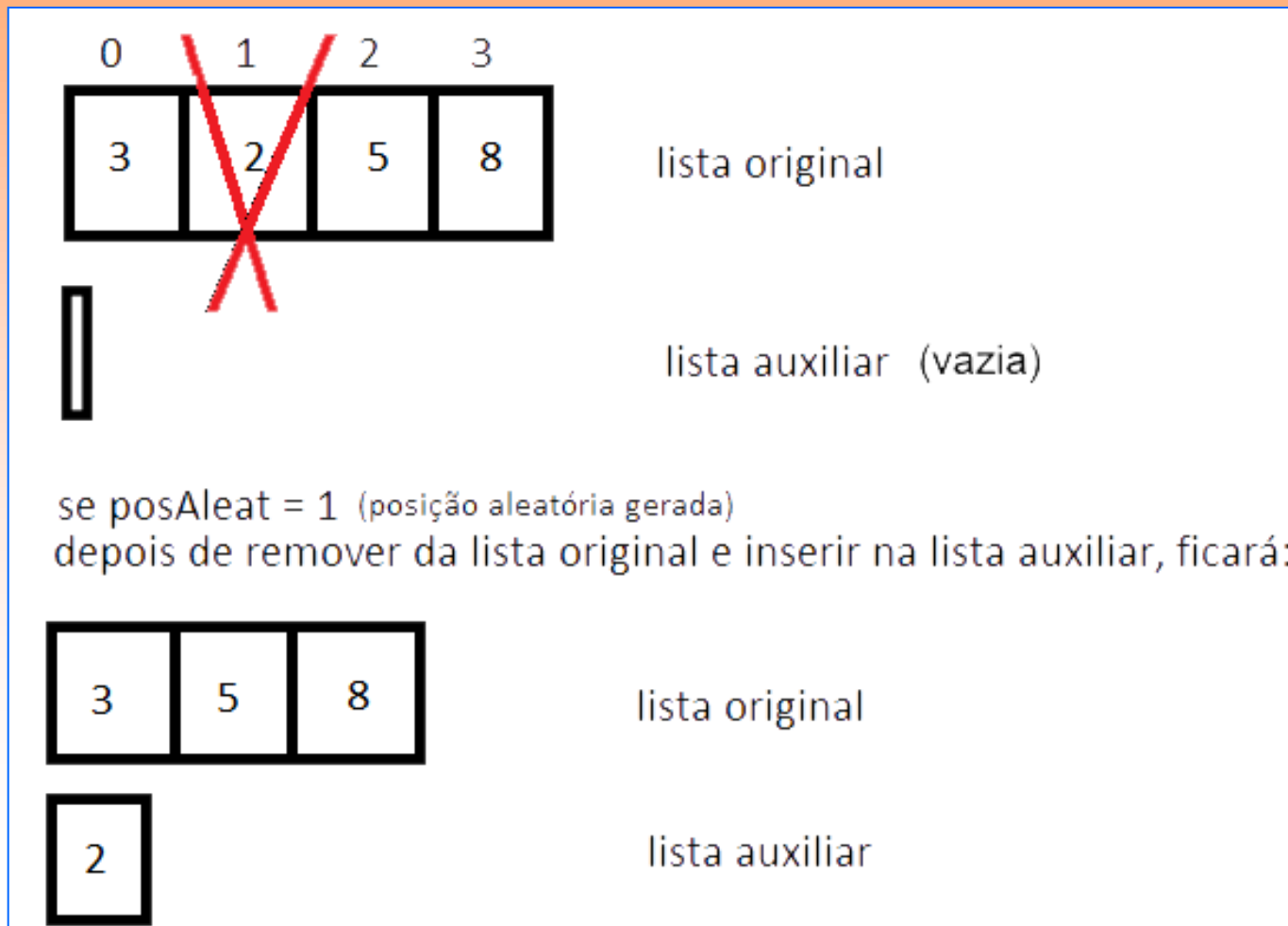
```
ArrayList lista = new ArrayList();
lista.add(1.2f); lista.add(4.3f); lista.add(6.1f); lista.add(-7.7f);
lista.add(0.4f); lista.add(-8.8f); lista.add(9.0f); lista.add(3.3f);
System.out.println("\nLista original:");
visualizaArrayList(lista);
aleatorizar(lista); 
System.out.println("\nLista anterior aleatorizada com o algoritmo de Fisher-Yates:");
visualizaArrayList(lista);
```

```
public void visualizaArrayList(ArrayList lista) {
    for (int i=0; i < lista.size(); i++) {
        System.out.print(lista.get(i) + "  ||  ");
    }
    System.out.println();
}
```

Algoritmo de Fisher-Yates (aleatorização de um vetor)

O algoritmo de Fisher-Yates utiliza uma **lista auxiliar**, onde serão inseridos os elementos que foram selecionados aleatoriamente da **lista original**. O elemento sorteado na lista original será eliminado.

A figura a seguir demonstra a lógica que será executada se for sorteada a posição 1 da lista original, em um exemplo com uma lista inicialmente com quatro elementos 3, 2, 5, 8.



Algoritmo de Fisher-Yates (aleatorização de um vetor)

```
public void aleatorizar(ArrayList lista) { //parâmetro por referência
    // Algoritmo de Fisher-Yates, implementado em Java com ArrayList
    ArrayList listaTemp = new ArrayList(); // criamos uma lista auxiliar adicional
    Random rnd = new Random();
    while ( lista.size() > 1 ) {
        // selecionamos aleatoriamente uma posição da lista inicial:
        int posAleat = rnd.nextInt(lista.size());
        // adicionamos o elemento sorteado no final da lista auxiliar:
        listaTemp.add(lista.get(posAleat));
        // eliminamos o elemento da lista original:
        lista.remove(posAleat);
    }
    // adicionamos o único restante no final da lista adicional:
    listaTemp.add(lista.get(0));
    lista.clear(); //limpamos a lista original
    // por último, passamos todos os elementos da lista adicional para a lista original
    for (int i=0; i < listaTemp.size(); i++) {
        lista.add(listaTemp.get(i));
    }
}
```

[nextInt](#)(int bound)

Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.

Exemplo em Ex_random.java

Busca sequencial em um vetor

Busca sequencial (linear) em um vetor

vet []	4.3	6.1	0.1	1.9	2.4	7.2	5.5
	0	1	2	3	4	5	6

Buscar: 2.4

- O valor a buscar poderá ser encontrado ou não.
- Como o vetor poderá estar desordenado, a busca começará sempre na posição 0 (zero) e terminará se o valor foi encontrado ou se ultrapassamos a última posição do vetor.
- No pior dos casos (quando o valor se encontra na última posição ou não se encontra no vetor) serão necessárias **n** comparações, considerando que **n** é o tamanho do vetor => a complexidade deste algoritmo é de **O(n)**. No melhor dos casos (se o elemento buscado for o primeiro) temos **O(1)**. Na média temos $O(n/2)$, considerado **O(n)**.

Busca sequencial (linear) em um vetor

```
public class Buscas {  
  
    public static void main (String args[]) {  
        new Buscas();  
    }  
  
    public Buscas() {  
        double a [] = {4.3, 6.1, 0.1, 1.9, 2.4, 7.2, 5.5};  
        int pos = buscaSequencial (a, 2.4); // busca o valor 2.4 no vetor a  
        if (pos != -1) System.out.println("O valor foi encontrado na posição " + pos);  
        else System.out.println("O valor não foi encontrado.");  
    }  
  
    int buscaSequencial (double vet[], double buscado) { //busca sequencial num vetor (reais)  
        for (int i = 0; i < vet.length; i++) {  
            if (vet[i] == buscado) return i; // encontramos o valor buscado, retornamos i  
        }  
        return -1; // o item não se encontra no vetor  
    }  
}
```

Exemplo no projeto NetBeans Buscas

Busca sequencial (linear) em um trecho de um vetor

```
public Buscas() {  
    //...  
    double a [] = {4.3, 6.1, 0.1, 1.9, 2.4, 7.2, 5.5};  
    int pos = buscaSequencial (a, 2.4, 0, 3); // busca o valor 2.4 no vetor a, entre 0 e 3  
    if (pos != -1) System.out.println("O valor foi encontrado na posição " + pos);  
        else System.out.println("O valor não foi encontrado nesse trecho.");  
}  
  
int buscaSequencial (double vet[], double buscado, int de, int ate) { //busca em vetor real  
    for (int i = de; i <= ate; i++) {  
        if (vet[i] == buscado) return i; // encontramos o valor buscado  
    }  
    return -1; // o item não se encontra nesse trecho do vetor  
}
```

Exemplo no projeto NetBeans Buscas

Busca binária em um vetor ordenado

Busca binária em um vetor ordenado

vet []	4.3	6.1	7.1	8.9	9.0	9.7	9.8
	0	1	2	3	4	5	6

vet []	4.3	6.1	7.1	8.9	9.0	9.7	9.8
	0	1	2	3	4	5	6
	inf			meio			sup

vet []	4.3	6.1	7.1	8.9	9.0	9.7	9.8
	0	1	2	3	4	5	6
	inf	meio	sup				

Buscando o valor 6.1 dentro do vetor vet

A partir de um **vetor ordenado** (crescente na figura), selecionamos a posição central (meio) e verificamos se esse é o elemento buscado.

Se não for o item buscado, buscaremos no trecho inferior ou no trecho superior, dependendo da comparação do valor buscado com esse elemento central. O processo se repetirá para cada novo trecho.

A busca terminará se encontramos o elemento ou se os ponteiros inf e sup já se cruzaram.

A complexidade deste método é $O(\log_2 n)$ no pior caso e $O(1)$ no melhor.

Busca binária em um vetor ordenado (algoritmo iterativo)

```
double b [] = {4.3, 6.1, 7.1, 8.9, 9.4, 9.6, 10.5}; // vetor ordenado
pos = buscaBinaria (b, 9.4);
if (pos != -1) System.out.println("O valor foi encontrado na posição " + pos);
    else System.out.println("O valor não foi encontrado no vetor.");
```

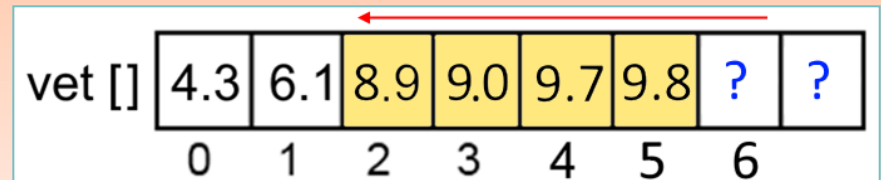
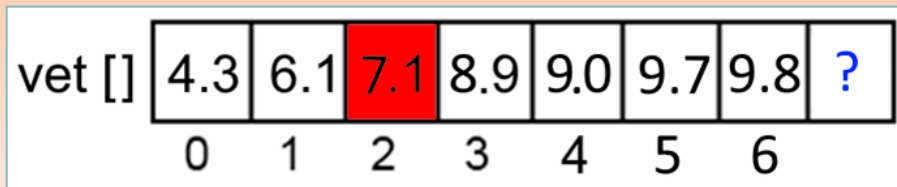
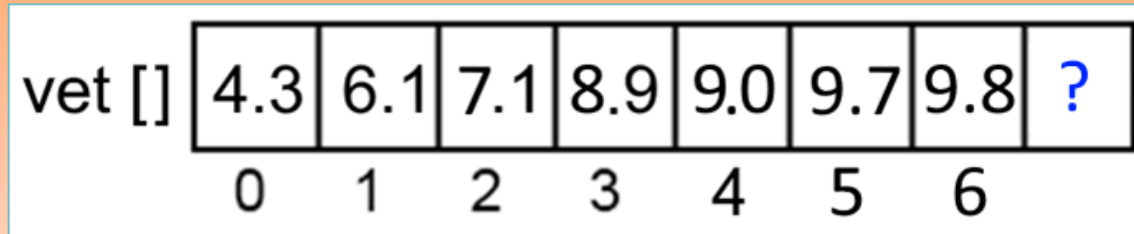
```
int buscaBinaria (double vet[], double buscado) {
    int inf = 0; // limite inferior
    int sup = vet.length - 1; // limite superior
    int meio;
    while (inf <= sup) {
        meio = (inf + sup) / 2;
        if ( buscado == vet[meio] ) return meio; // o valor buscado foi encontrado
        if ( buscado < vet[meio] ) sup = meio - 1; else inf = meio + 1;
    }
    return -1; // o elemento não foi encontrado
    // poderíamos retornar -(inf + 1) para especificar que: não foi encontrado
    // e que a posição de inserção para esse valor buscado seria -inf - 1
}
```

Exemplo no projeto
NetBeans Buscas

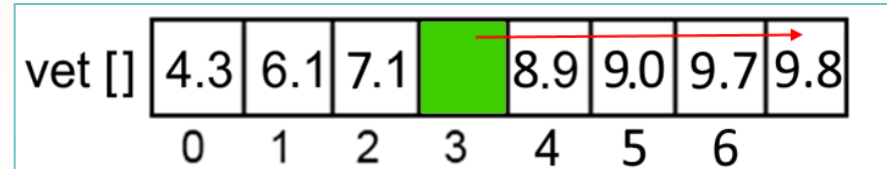
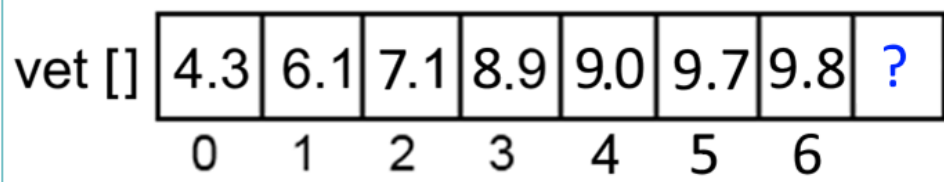
Deslocamentos em vetores

Deslocamento de vetores

- Situações onde um trecho do vetor será deslocado para direita ou para esquerda. Por exemplo, ao **eliminar** um item, deslocaremos os restantes **para a esquerda**:



- Para **inserir** um novo item em uma posição, por exemplo, deslocaremos os itens restantes **para a direita** (usando a memória livre do vetor):

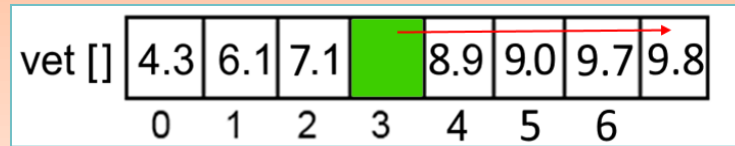


Deslocamento de vetores (uma versão simplificada)

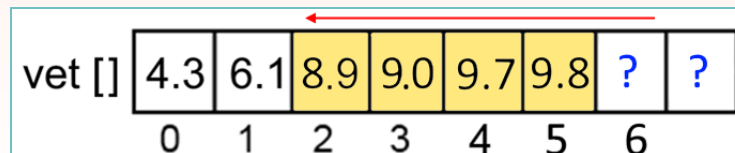
// Obs: na chamada destes algoritmos talvez melhor não utilizar *vet.length*.

// Utilize uma variável com a quantidade verdadeira de itens (para vetores sem ocupação total).

```
void deslocaParaDireita (double vet[], int de, int ate) {  
    // abre um espaço e ocupa uma posição no final  
    for (int i = ate; i >= de; i--) vet[i+1] = vet[i];  
    vet[de] = 0; // só para marcar o item que ficou "vazio"  
}
```



```
void deslocaParaEsquerda (double vet[], int de, int ate) {  
    // elimina o item na posição (de-1) e desloca os restantes para a esq.  
    for (int i = (de - 1); i < ate; i++) vet[i] = vet[i+1];  
    vet[ate] = 0; // só para marcar o item final  
}
```



Deslocamento de vetores (uma versão mais completa)

```
void deslocaDireita (double vet[], int de, int ate) {  
    //alguns testes de pré-requisitos permitem evitar erros:  
    if(de>ate)return;  
    if(de<0)de=0;  
    if(ate > vet.length-2)ate=vet.length-2;  
    // abre um espaço na posição 'de' e ocupamos mais uma posição no final  
    for (int i = ate; i >= de; i--) vet[i+1] = vet[i]; //obs: ciclo começando no final  
    vet[de] = 0; // só para marcar o item que ficou "vazio"  
}
```

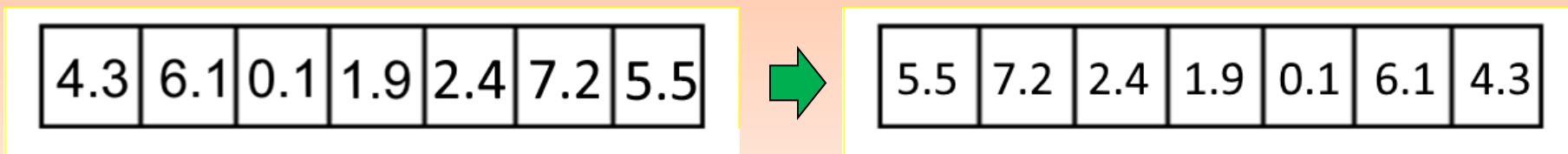
```
void deslocaEsquerda (double vet[], int de, int ate) {  
    //alguns testes de pré-requisitos para evitar erros:  
    if(de>ate)return;  
    if(de<=0)de=1; //corrigimos a posição, mas poderíamos abortar  
    if(ate > vet.length-1)ate=vet.length-1; //corrigimos a posição, mas...  
    // elimina o item na posição (de-1) e desloca os restantes para a esquerda  
    for (int i = (de - 1); i < ate; i++) vet[i] = vet[i+1];  
    vet[ate] = 0; // só para marcar o item final  
}
```

Exemplo no projeto NetBeans Buscas

Exercício 1 (ver figura no próximo slide)

Elabore um método (e os testes necessários) para inverter completamente os elementos guardados em um vetor. Considere uma variável N que armazena a quantidade de elementos reais do vetor e não seu tamanho alocado. Visualize o vetor depois de invertido.

```
public void inverter(double vet[], int N)
```



Exercício 2

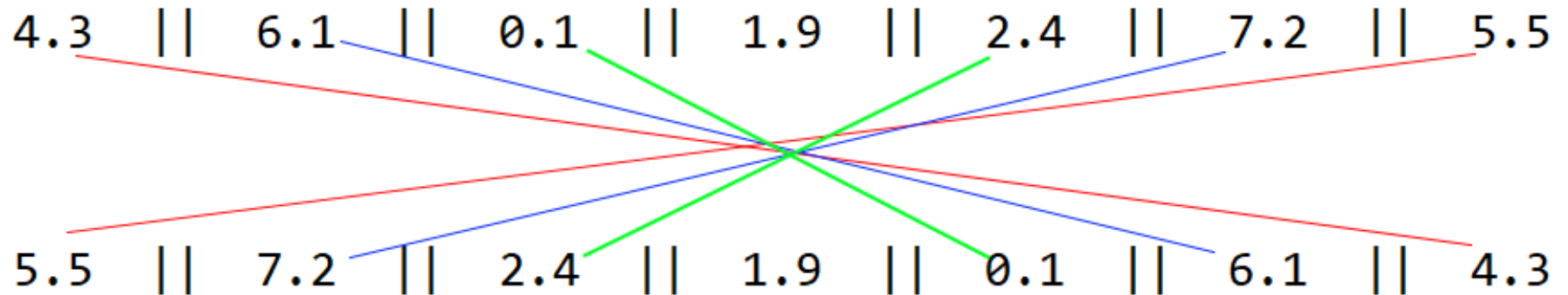
Elabore um método (e os testes necessários) para inverter completamente os elementos guardados em um ArrayList. Visualize o ArrayList depois de invertido.

```
public void inverter(ArrayList arr)
```


Exercícios

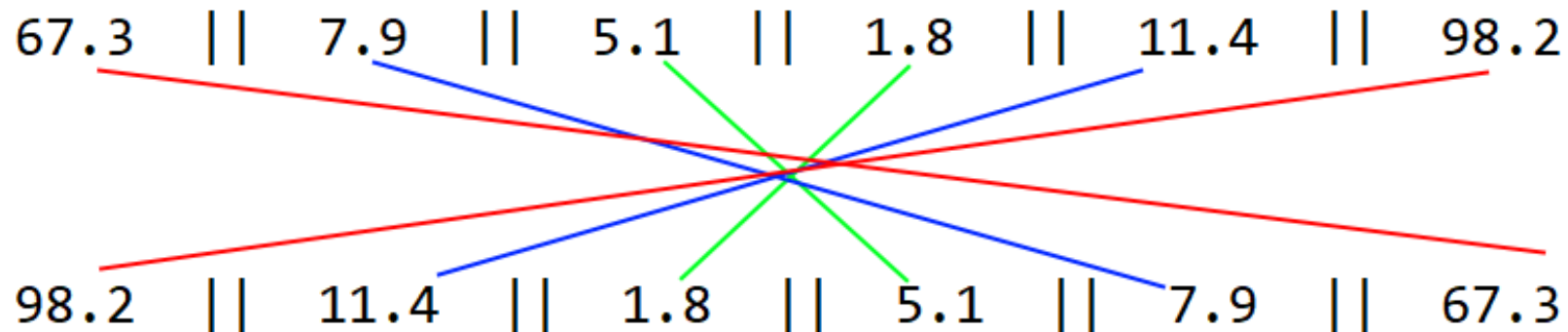
----- Vetor com qtde. elementos ímpar -----

4.3 || 6.1 || 0.1 || 1.9 || 2.4 || 7.2 || 5.5
5.5 || 7.2 || 2.4 || 1.9 || 0.1 || 6.1 || 4.3



----- Vetor com qtde. elementos par -----

67.3 || 7.9 || 5.1 || 1.8 || 11.4 || 98.2
98.2 || 11.4 || 1.8 || 5.1 || 7.9 || 67.3



Bibliografia (oficial) da disciplina

BIBLIOGRAFIA BÁSICA

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, Campus, 2002.

GOODRICH, Michael T.; TAMASSIA, Roberto. Estruturas de dados e algoritmos em Java. 2. ed. Porto Alegre, São Paulo: Bookman, 2002.

SZWARCFITER, Jayme Luiz. Estruturas de dados e seus algoritmos. 3. Rio de Janeiro: LTC, 2010, recurso online, ISBN 978-85-216-2995-5.

BIBLIOGRAFIA COMPLEMENTAR

ASCENCIO, A. F. G.; ARAÚJO, G. S. Estruturas de Dados. São Paulo: Pearson, 2011. [eBook]

EDELWEISS, N.; GALANTE, T. Estruturas de Dados. Porto Alegre: Bookman, 2009. [eBook]

MORIN, P. Open Data Structures (in Java) Creative Commons, 2011. Disponível em <http://opendatastructures.org/ods-java.pdf> [eBook]

PUGA, S.; RISSETTI, G. Estruturas de Dados com aplicações em Java, 2a ed. São Paulo: Pearson, 2008. [eBook]

SHAFFER, C. A.; Data Structures and Algorithm Analysis. Virginia Tech, 2012. Disponível em