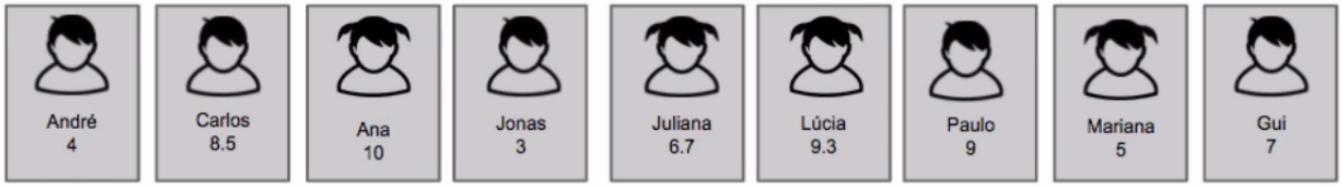
01

Quicksort: Implementação de ordenação baseada em pivos

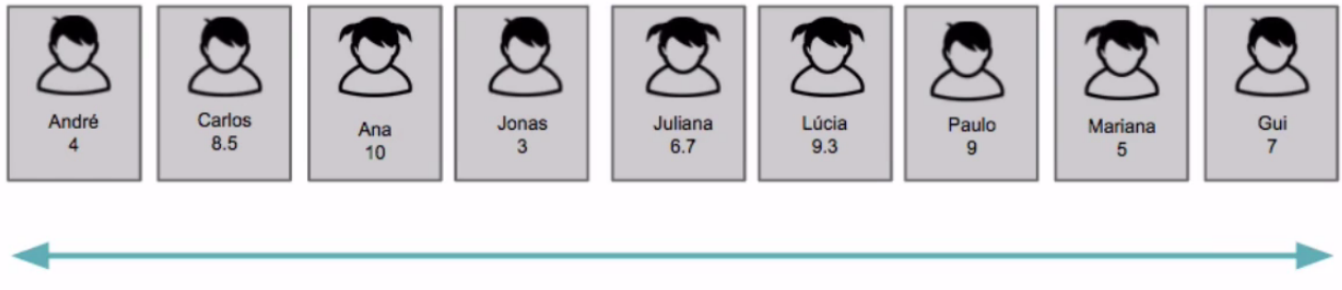
Transcrição

Nós já sabemos fazer diversas coisas com o nosso *array*: encontrar quem é maior ou menor, quantos menores temos, qual posição o elemento deveria ficar, colocar os itens menores à esquerda e os maiores, à direita. Também particionamos o *array* em duas partes.

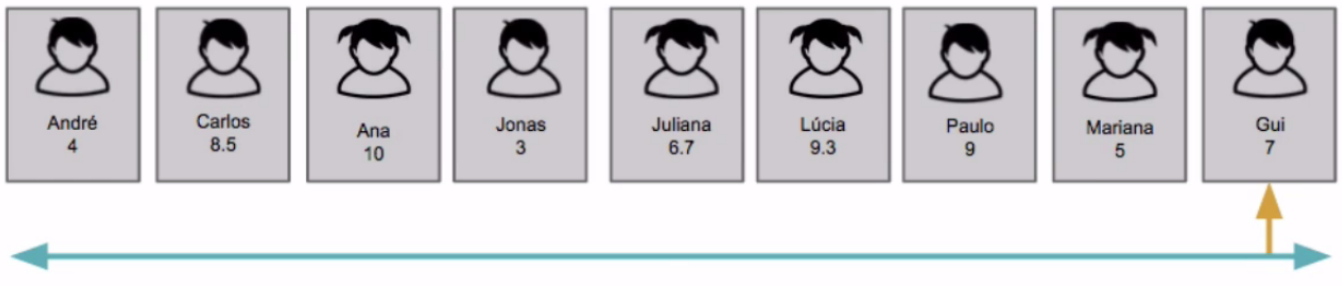
É possível responder muitos problemas do mundo real. Como por exemplo, se queremos ajudar os alunos que estão com notas abaixo de 4, que foi a nota que tirei. Então, eu posso ajudar aqueles que se sairam pior. Separo estas pessoas e ficam separadas também as pessoas que tiraram notas maiores do que a minha. Em uma única varrida dos elementos foi possível descubrir tudo isto. Podemos fazer muitas coisas com os algoritmos que aprendemos até agora.



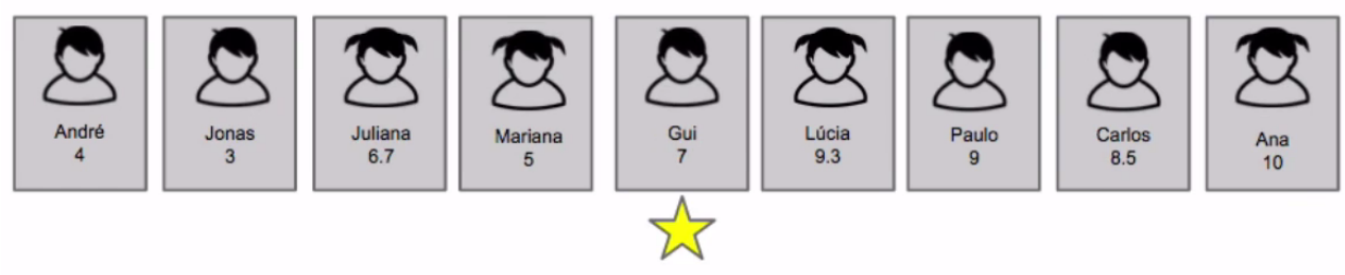
Quando rodamos o particiona e temos um *array* com as notas de todos os alunos, sempre iremos indicar o `início` e o `termino`, quais elementos serão analisados.



Então, colocaremos o último elemento (o pivô)...



no lugar adequado.



Ao trocarmos os elementos de lugar, os demais também serão reposicionados. Desta forma, todos que estarão à esquerda serão menores e os que estarão à direita, serão maiores. Assim rodamos o `pivota`, o algoritmo que particiona o *array* em duas partes com o pivô. Este foi o resultado.

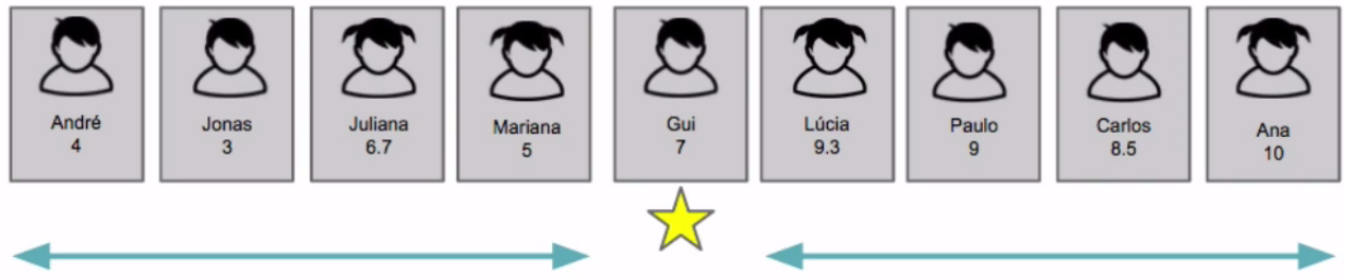
Porém, o que é possível fazer com o *array* agora que sabemos a posição correta do Guilherme?

Particiona particiona particiona

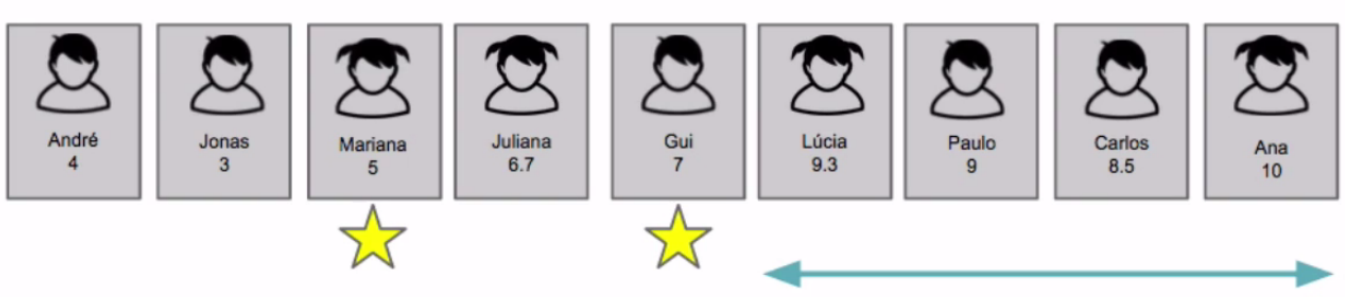
O nosso *array* foi pivotado uma vez. Se ele foi particionado, o elemento que anteriormente ocupava a última posição já foi colocado na posição correta, que será a definitiva.

O Guilherme está na posição que merece. Os demais ainda não. Tanto os elementos que estão posicionados à esquerda, como os que estão à direita do pivô não estão adequados. Talvez, por sorte, algum esteja ocupando a posição correta - como a Ana, por exemplo. Mas não sabemos com segurança... O único que temos certeza é o que rodamos o **particiona**.

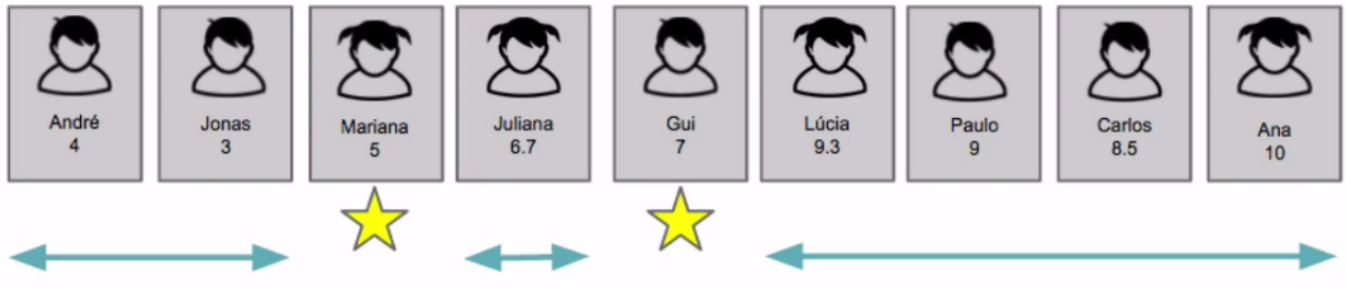
Se o Guilherme está na posição correta e os itens da esquerda não estão, podemos posicionar a Mariana adequadamente? Sabemos como fazer isto. Também sei como reposicionar corretamente a Ana. Se mandarmos particionar o quatro primeiros elementos, aonde a Mariana irá terminar? Na posição certa dela. O mesmo irá acontecer se mandarmos particionar os quatro últimos elementos do *array*: a Ana ficará na posição certa. Logo, depois que particionamos o *array*, iremos particionar as duas parte menores.



Vamos começar a particionar a parte da esquerda? O pivô será a Mariana. Ela irá ficar na casinha 2, a posição correta.

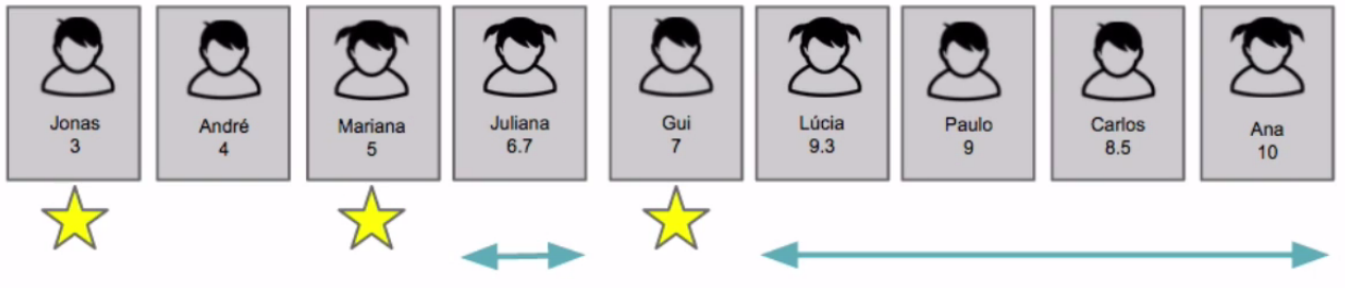


Porém, os elementos que ficaram a sua esquerda, estão em posições adequadas? E quem ficou à direita? Não sabemos. Precisaremos rodar o particiona também para quem está posicionado tanto à direita como à esquerda.



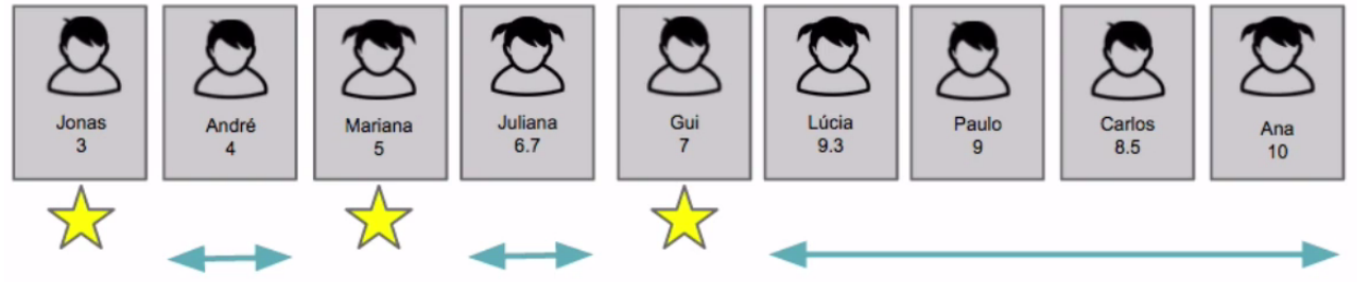
Se rodarmos o particiona para estes elementos também, o que acontecerá? Veremos. Iremos particionar o André e o Jonas. Quem será o pivô? O Jonas.

Logo, colocaremos o Jonas no lugar correto.

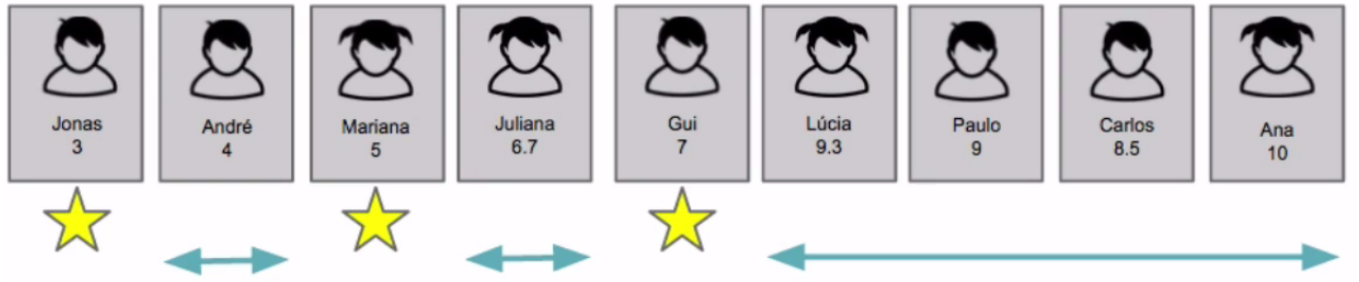


Agora precisamos pivotear os elementos que estão à esquerda e à direita do Jonas.

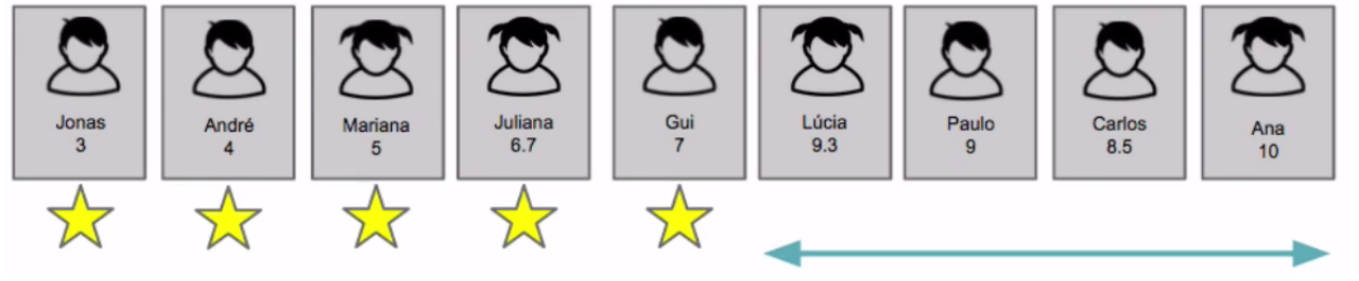
Porém, no lado esquerdo do Jonas, não temos elementos. Se o número de elementos é igual a 0, não fará sentido particionar.



No lado direito, temos apenas um elemento. Se temos que ordenar um elemento, também não será preciso reposicioná-lo.



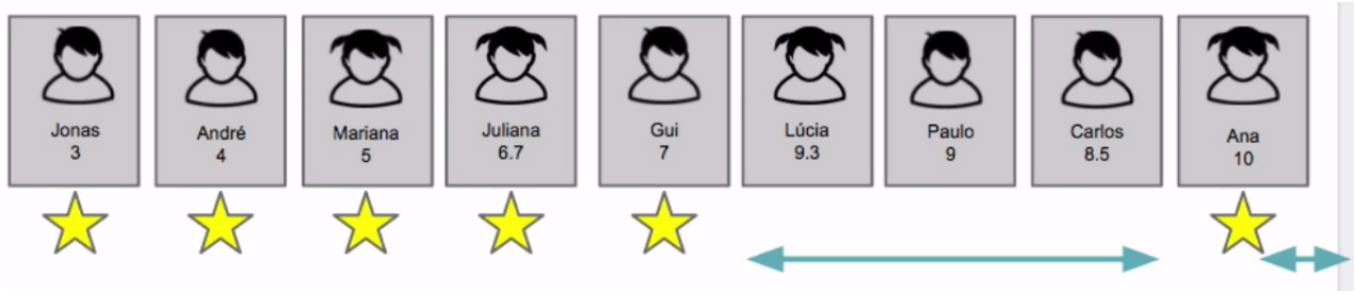
Será o mesmo com a Juliana. Também teremos apenas um elemento e por isso, não iremos reposicioná-la.



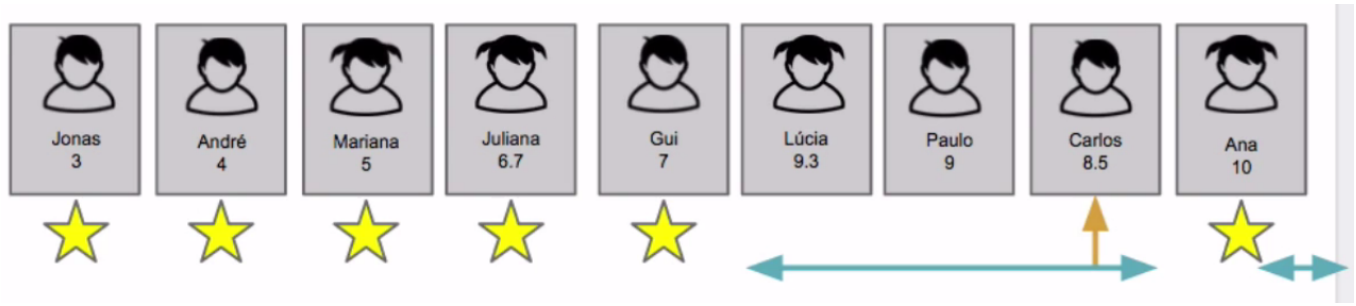
Agora iremos particionar os quatro elementos posicionas à direita. Quem será o pivô? A Ana.

Vamos colocá-la no seu lugar! Ela já ocupa a posição adequada.

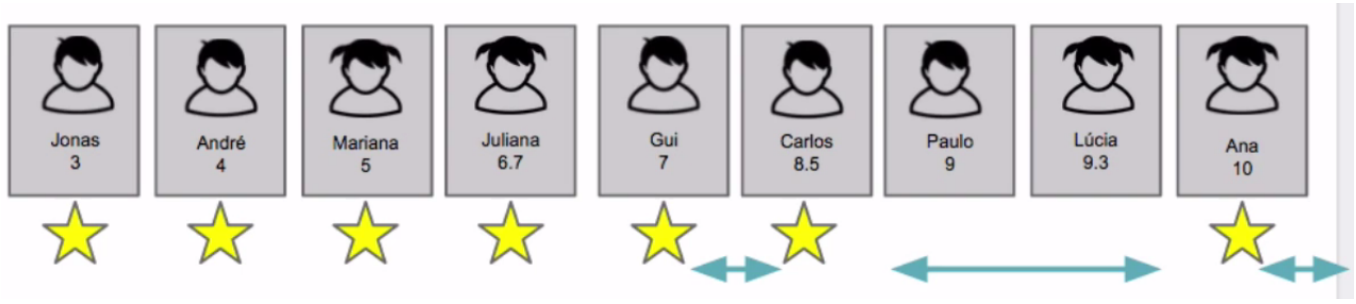
A Ana está no lugar certo. Em seguida, teremos que particionar a esquerda e a direita do pivô.



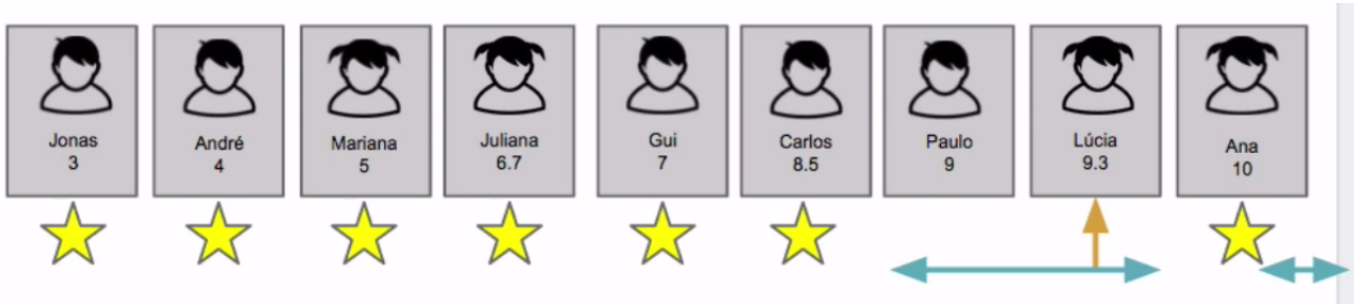
Começaremos pelo lado esquerdo. Quem será o novo pivô? O Carlos.



Ao pivotearmos o Carlos, teremos que repetir a ação com os elementos posicionados a sua esquerda e sua direita.



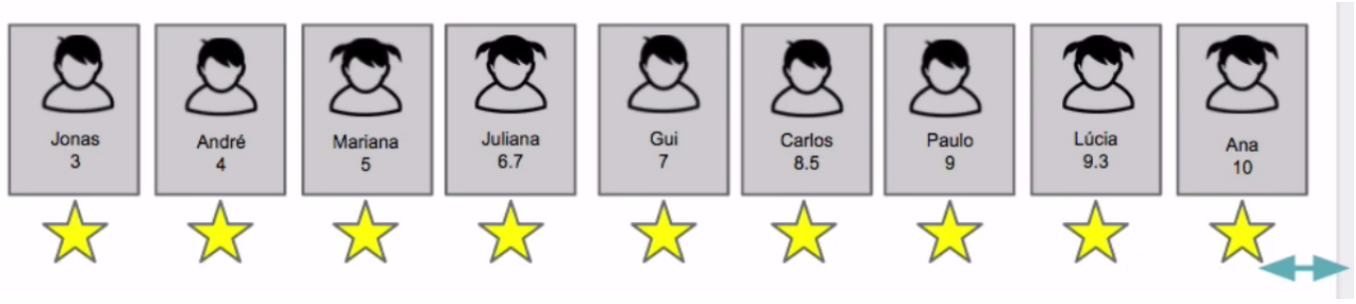
No entanto, à esquerda do Carlos está vazia e só nos resta elementos à direita. Quem será o pivô dos itens restantes? A Lúcia.



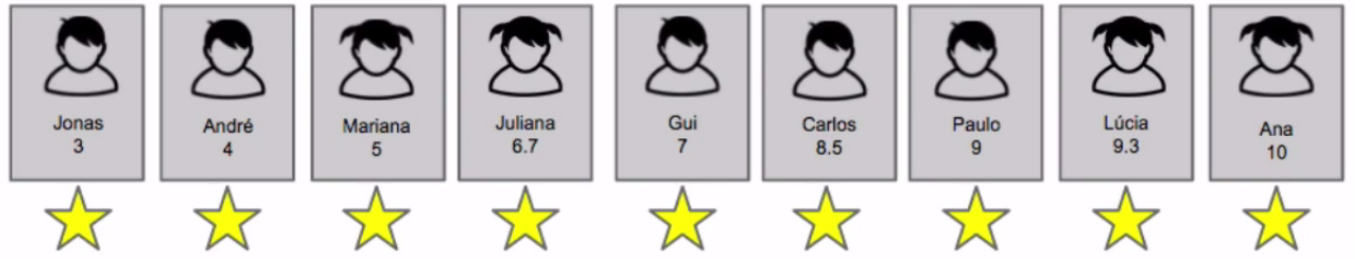
A Lúcia está na posição correta. Temos agora que pivotear à sua esquerda e direita.

Vamos pivotear o lado esquerdo. Teremos apenas um elemento, logo ele permanecerá no mesmo lugar.

À direita, não haverá elementos, por isso, não precisaremos fazer nada. Faltou um último pivoteamento, novamente de tamanho 0.

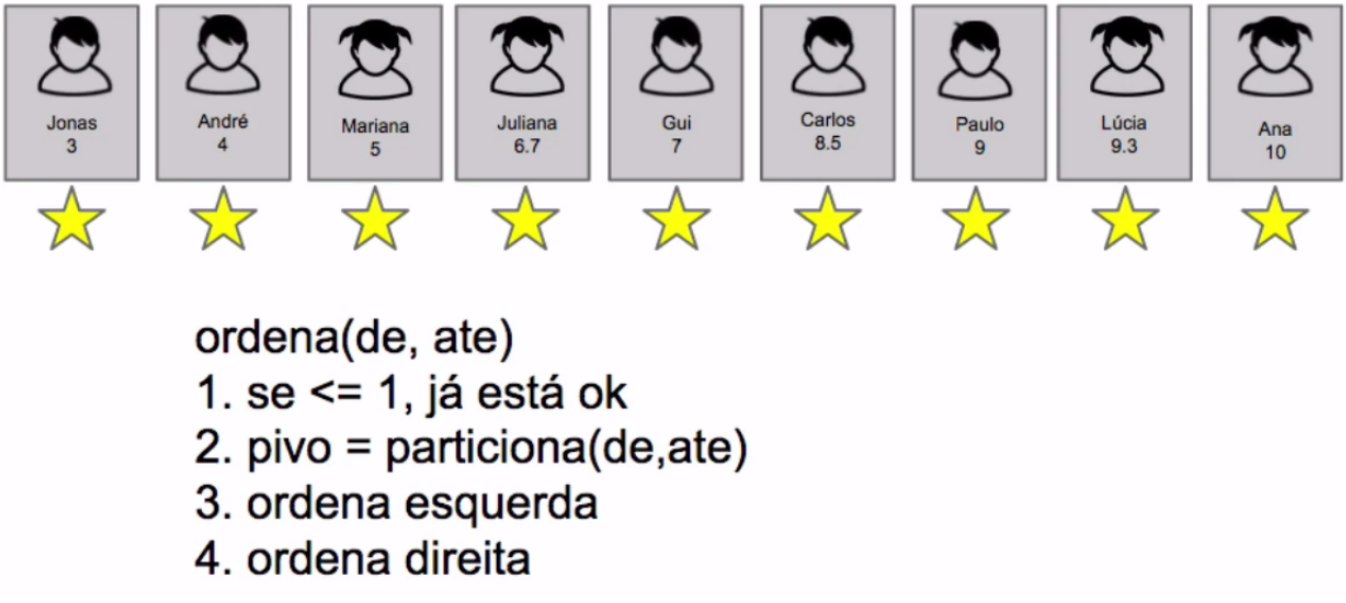


O que faremos? Nada, novamente.



O *array* está completamente ordenado. Nós usamos a sacada de, ao pivotear um elemento, colocávamos em seguida na posição certa. Quando mandávamos ordenar os elementos posicionados à direita e à esquerda do novo pivô, pisoteávamos e ordenávamos os pedaços menores que se formavam. Repetimos o processo até que precisávamos pivotear 0 ou 1, e não era preciso fazer nada. Seguimos por todos elementos, até que *array* estava ordenado e cada um ocupava a sua posição certa.

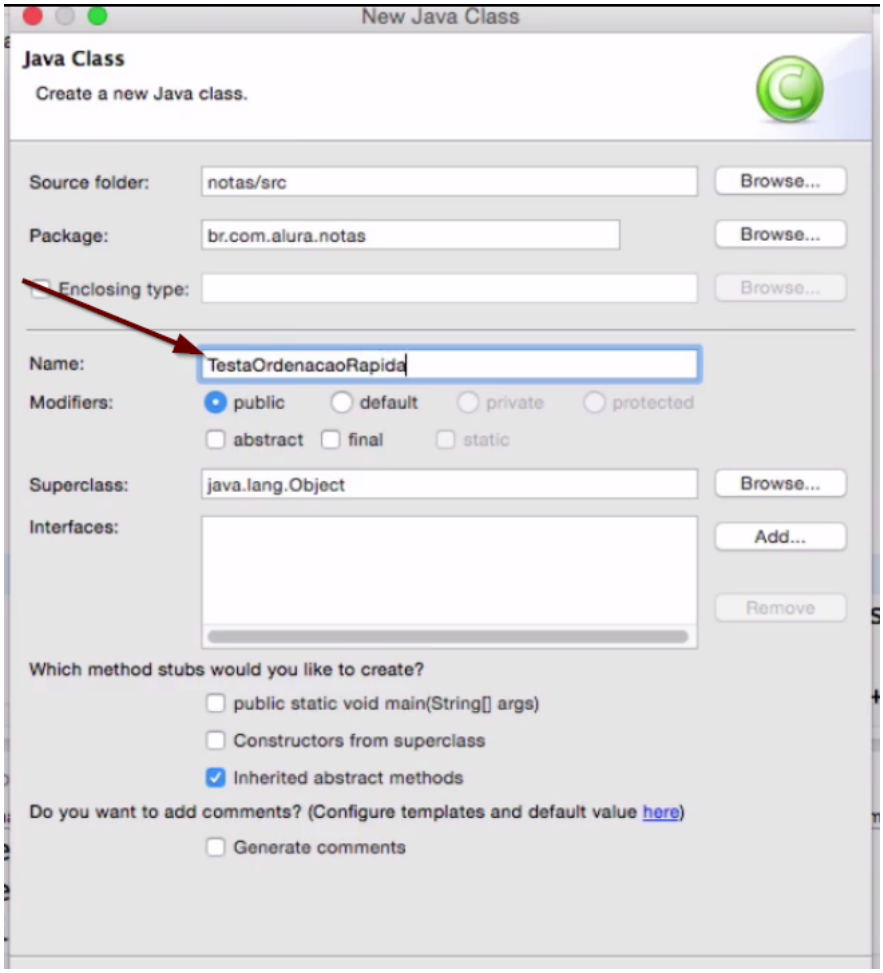
Vamos revisar o que fizemos:



Para ordenar **de** uma posição **até** outra, precisamos considerar o número de elementos. Se ele for de 0 até 1, não é preciso fazer nada. Caso ele seja maior, mandamos particionar o pedaço e colocar o pivô na posição adequada. Ordenamos os elementos posicionados às esquerda e à direita do pivô. Em seguida, o algoritmo irá ordenando os trechos menores que irão se formando. Ele seguirá pivoteando, até que todos os elementos estejam todos ordenados. Vamos tentar implementar isto?

Ordenando particionando o quicksort

Vamos usar a função de pivotar o `quebraNoPivo`, que na verdade pode ser renomeada como `particiona`. Ela particionará o *array* utilizando como pivô o último elemento, com o objetivo de ordenar a lista completa. Então iremos criar uma classe nova de teste que se chamará `TestaOrdenacaoRapida`.



Dentro do `TestaOrdenacaoRapida` criaremos o método `main()`:

```
package br.com.alura.notas;

public class TestaOrdenacaoRapida {
    public static void main(String[] args) {

    }
}
```

Iremos copiar as notas do método `main()` do `TestaPivota`. Nós utilizaremos as mesmas.

```
Nota guilherme = new Nota("guilherme", 7);
Nota[] notas = {
        new Nota("andre", 4),
        new Nota("carlos", 8.5),
        new Nota("ana", 10),
        new Nota("jonas", 3)
        new Nota("juliana", 6.7),
        new Nota("lucia", 9.3),
        new Nota("paulo", 9),
        new Nota("mariana", 5),
        guilherme

     };
```

No fim, queremos imprimir também o resultado da ordenação. Copiaremos da outra classe também o `for`:

```
for(int atual = 0; atual < notas.length; atual++) {
    Nota nota = notas[atual];
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

Nosso código ficará assim:

```
Nota guilherme = new Nota("guilherme", 7);
Nota[] notas = {
        new Nota("andre", 4),
        new Nota("carlos", 8.5),
        new Nota("ana", 10),
        new Nota("jonas", 3)
        new Nota("juliana", 6.7),
        new Nota("lucia", 9.3),
        new Nota("paulo", 9),
        new Nota("mariana", 5),
        guilherme

     };

    for(int atual = 0; atual < notas.length; atual++) {
    Nota nota = notas[atual];
    System.out.println(nota.getAluno() + " " + nota.getValor());
}
```

Nós precisaremos ordenar os elementos, por isso, teremos que pivotar. Vamos ordenar do 0 até o `notas.length`. Colocares a seguinte linha acima do `for`:

```
ordenar(notas, 0, notas.length);
```

Mais abaixo, iremos criar a função de ordenação e teremos o `de` e o `ate`:

```
private static void ordena(Nota[] notas, int de, int ate) {
}
```

Nós iremos particionar as notas do `de` ao `ate`.

```
private static void ordena(Nota[] notas, int de, int ate) {
}
```

Isto irá nos devolver a posição do pivô (`posicaoDoPivo`). O algoritmo nos informará a posição adequada para o elemento.

```
private static void ordena(Nota[] notas, int de, int ate)
    int posicaoDoPivo = particiona(notas, de, ate);
    {
}
```

Se o pivô estiver na posição correta e todos os elementos posicionados à esquerda forem menores, assim como os localizados à direita forem maiores, basta ordenarmos estas duas partes. Então, mandaremos ordenar as notas da posição `de` até a `posiçãoDoPivo`.

```
ordena(notas, de, posicaoDoPivo);
```

Depois, mandaremos ordenar os elementos da direita. Ordenaremos as notas da `posicaoDoPivo + 1` até o fim.

```
ordena(notas, posicaoDoPivo + 1, ate);
```

A parte da direita será ordenada também. Mas esta será a maneira de ordenar para sempre? Não, se temos 0 elementos não será preciso fazer nada. Logo, o número de elementos será: `até` menos o `de`.

```
int elementos = ate - de;
```

Se (`if`) o número de elementos for maior do que 1, ou seja, superior a dois elemento, teremos que ordená-los. Quando temos um número de elementos igual ou menor que 1, não teremos que nos preocupar, porque já estará ordenado. Vamos ver como ficou o código com as novas linhas:

```
private static void ordena(Nota[] notas, int de, int ate) {
    int elementos = ate - de;
    if(elementos > 1) {
        int posicaoDoPivo = particiona(notas, de, ate);
        ordena(notas, de, posicaoDoPivo);
        ordena(notas, posicaoDoPivo + 1, ate);
    }
}
```

Em seguida, iremos copiar da classe `TestaPivota`, as funções `particiona()` e `troca()`. Ambas serão usadas após o `ordena`.

```
private static void ordena(Nota[] notas, int de, int ate) {
    int elementos = ate - de;
    if(elementos > 1) {
        int posicaoDoPivo = particiona(notas, de, ate);
        ordena(notas, de, posicaoDoPivo);
        ordena(notas, posicaoDoPivo + 1, ate);
    }
}

private static int particiona(Nota[] notas, int inicial, int termino) {
    int menoresEncontrados = 0;

    Nota pivo = notas[termino - 1];
    for(int analisando = 0; analisando < termino -1; analisando ++) {
    Nota atual = notas[analisando];
    if(atual.getValor() <= pivo.getValor()) {
        troca(notas, analisando, menoresEncontrados);
        menoresEncontrados++;
```